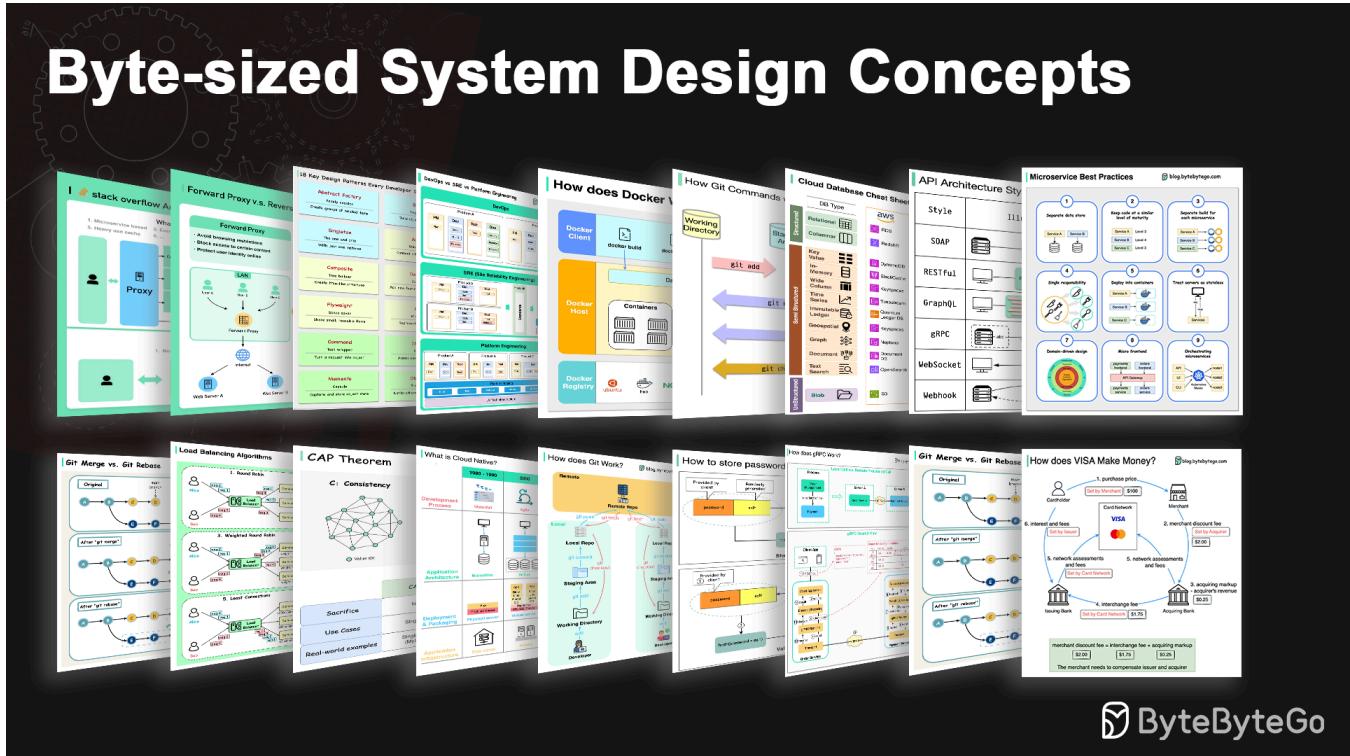




Byte-sized System Design Concepts



ByteByteGo

【 🧑‍💻 YouTube | 📰 Newsletter 】

系统设计 101 (System Design 101)

使用可视化和简单术语来解释复杂系统。

无论你是在准备系统设计面试，还是单纯想了解系统的底层工作原理，我们都希望这个仓库可以帮助你实现此目标。

目录

- 系统设计 101 (System Design 101)
- 目录
 - 通信协议
 - REST API vs. GraphQL
 - gRPC 是如何工作的？
 - 什么是 webhook？
 - 如何改善 API 性能？
 - HTTP 1.0 -> HTTP 1.1 -> HTTP 2.0 -> HTTP 3.0

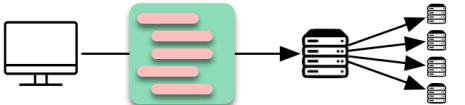
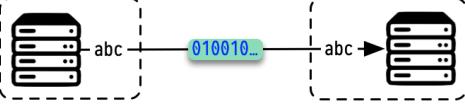
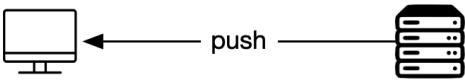
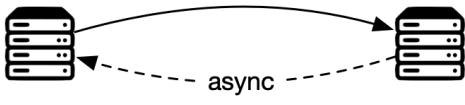
- (QUIC)
 - SOAP vs REST vs GraphQL vs RPC
 - 代码优先 (Code First) vs. API 优先 (API First)
 - HTTP 状态码
 - API 网关有什么作用?
 - 我们要如何设计高效安全的 API?
 - TCP/IP 封装
 - 为什么 Nginx 被称为“反向”代理?
 - 常见的负载均衡算法是什么?
 - URL, URI, URN - 你知道它们之间的差别吗?
- CI/CD
 - 简单解释 CI/CD 管道
 - Netflix 技术栈 (CI/CD 管道)
- 架构模式
 - MVC、MVP、MVVM、MVVM-C 和 VIPER
 - 每一位开发者都必须知道的 18 个关键设计模式
- 数据库
 - 关于云服务中不同类型的数据库的一个便携速查表
 - 8 种支撑数据库的数据结构
 - 一条 SQL 语句是如何在数据库中执行的?
 - CAP 理论
 - 内存和存储的类型
 - 可视化 SQL 查询
 - SQL 语言
- 缓存
 - 缓存无处不在
 - 为什么 Redis 那么快?
 - Redis 的使用场景
 - 最佳缓存策略
- 微服务架构
 - 典型的微服务架构长啥样?
 - 微服务最佳实践
 - 微服务通常使用哪些技术栈?

- 为什么 Kafka 快?
- 支付系统
 - 如何学习支付系统?
 - 信用卡为何被称为“银行最赚钱的产品”? VISA/万事达卡是如何赚钱的?
 - 当我们在商家处刷卡时, VISA 是如何运作的?
 - 世界各地的支付系统系列 (第一部分) : 印度的统一支付接口 (Unified Payments Interface, UPI)
- DevOps
 - DevOps vs. SRE vs. 平台工程 (Platform Engineering)。有何不同?
 - k8s (Kubernetes) 是什么?
 - Docker vs. Kubernetes。我们应该用哪一个?
 - Docker 的工作原理
- GIT
 - Git 命令是如何工作的?
 - Git 的工作原理
 - Git merge vs. Git rebase
- 云服务
 - 不同云服务的便捷速查表 (2023 年版本)
 - 什么是云原生 (cloud native) ?
- 开发者生产力工具
 - 可视化 JSON 文件
 - 自动将代码转换为架构图
- Linux
 - Linux 文件系统解析
 - 你应该知道的 18 个最常用的 Linux 命令
- 安全
 - HTTPS 是如何工作的?
 - 简明扼要解释下 Oauth 2.0。
 - 四种最常见的身份认证机制
 - 会话、Cookie、JWT、令牌、SSO 和 OAuth 2.0 - 它们分别是什么?
 - 如何将密码安全地存储到数据库中, 以及如何验证密码?

- 向一个十岁小孩解释 JSON Web Token (JWT)
- Google Authenticator (或者其它类型的两步认证器) 是如何工作的?
- 真实案例学习
 - Netflix 的技术栈
 - Twitter 架构 2022
 - 过去 15 年 Airbnb 微服务架构的演进之路
 - Monorepo vs. Microrepo.
 - 如果是你, 你要如何设计 Stack Overflow 网站?
 - 为什么 Amazon Prime Video 监控从无服务 (Serverless) 转向了单体架 (Monolithic) ? 它是怎样节省九成成本的呢?
 - Disney Hotstar 是如何在锦标赛期间捕获 50 亿个表情符号的?
 - Discord 是怎样存储数万亿条消息的
 - YouTube、TikTok Live 或 Twitch 上的视频直播是如何工作的呢?
- 许可

通信协议

架构风格定义了应用编程接口 (application programming interface, API) 的不同组件之间是如何相互交互的。因此, 通过提供设计和构建 API 的标准方法, 它们一起保证了效率、可靠性以及与其他系统集成的便捷性。以下是最常用的风格:

Style	Illustration	Use Cases
SOAP		XML-based for enterprise applications
RESTful		Resource-based for web servers
GraphQL		Query language reduce network load
gRPC		High performance for microservices
WebSocket		Bi-directional for low-latency data exchange
Webhook		Asynchronous for event-driven application

- SOAP:

成熟、全面、基于 XML

最适用于企业应用程序

- RESTful:

流行、易于实现的 HTTP 方法

网络服务的理想之选

- GraphQL:

查询语言、请求特定数据

减少网络开销，提高响应速度

- gRPC:

现代、高性能、协议缓冲

适用于微服务架构

- WebSocket:

实时、双向、持久连接

非常适合低延迟数据交换

- Webhook:

事件驱动、HTTP 回调、异步

事件发生时通知系统

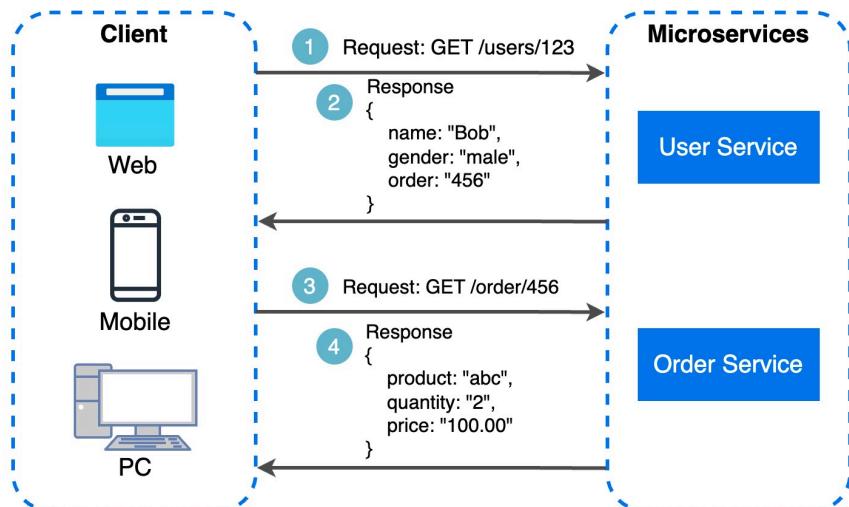
REST API vs. GraphQL

在API设计方面，REST 和 GraphQL 各有优劣。

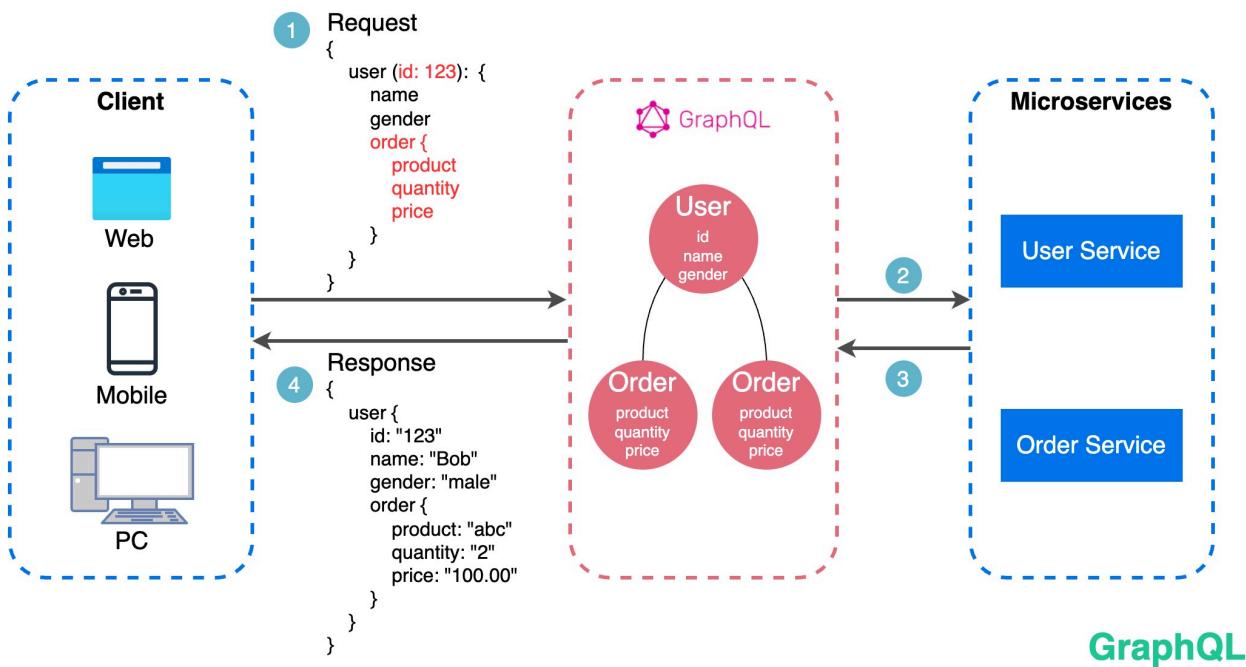
下图显示了 REST 和 GraphQL 之间的简单比较。

REST v.s. GraphQL

 blog.bytebybytego.com



REST



GraphQL

REST

- 使用标准的 HTTP 方法，例如 GET、POST、DELETE，来进行 CRUD 操作
- 当你需要在独立服务或应用程序之间实现简单、统一的接口时，这种架构风格表现良好
- 缓存策略的实现很直接
- 缺点是可能需要多次往返，才能从不同的端点组装相关数据。

GraphQL

- 为客户端提供单一端点，从而实现所需数据的精确查询。
- 客户端在嵌套查询中指定需要返回的字段，而服务端返回仅包含这些字段的已优化数据体。（译注：返回结果中字段是固定的，GraphQL 可以在查询时根据实际情况做一些优化，减少返回的数据量。）
- 支持用于修改数据的变更（Mutations）和用于实时通知的订阅（Subscriptions）。
- 非常适合来自多个来源的数据的聚合，并可以很好地满足快速变化的前端需求。
- 但是，它将复杂性转移到了前端，并且如果没有适当保护，则可能会允许滥用查询。
- 缓存策略可能比 REST 更复杂。

REST 和 GraphQL 之间哪一个是最佳选择取决于应用和开发团队之间的具体需求。GraphQL 非常适合复杂或频繁变化的前端需求，而 REST 则适合那些更倾向于简单和一致的应用程序。

这两种 API 方法都不是灵丹妙药。仔细评估需求以及权衡其优劣，对于选择正确的 API 方法非常重要。REST 和 GraphQL 都是用于暴露数据和支持现代应用的有效选择。

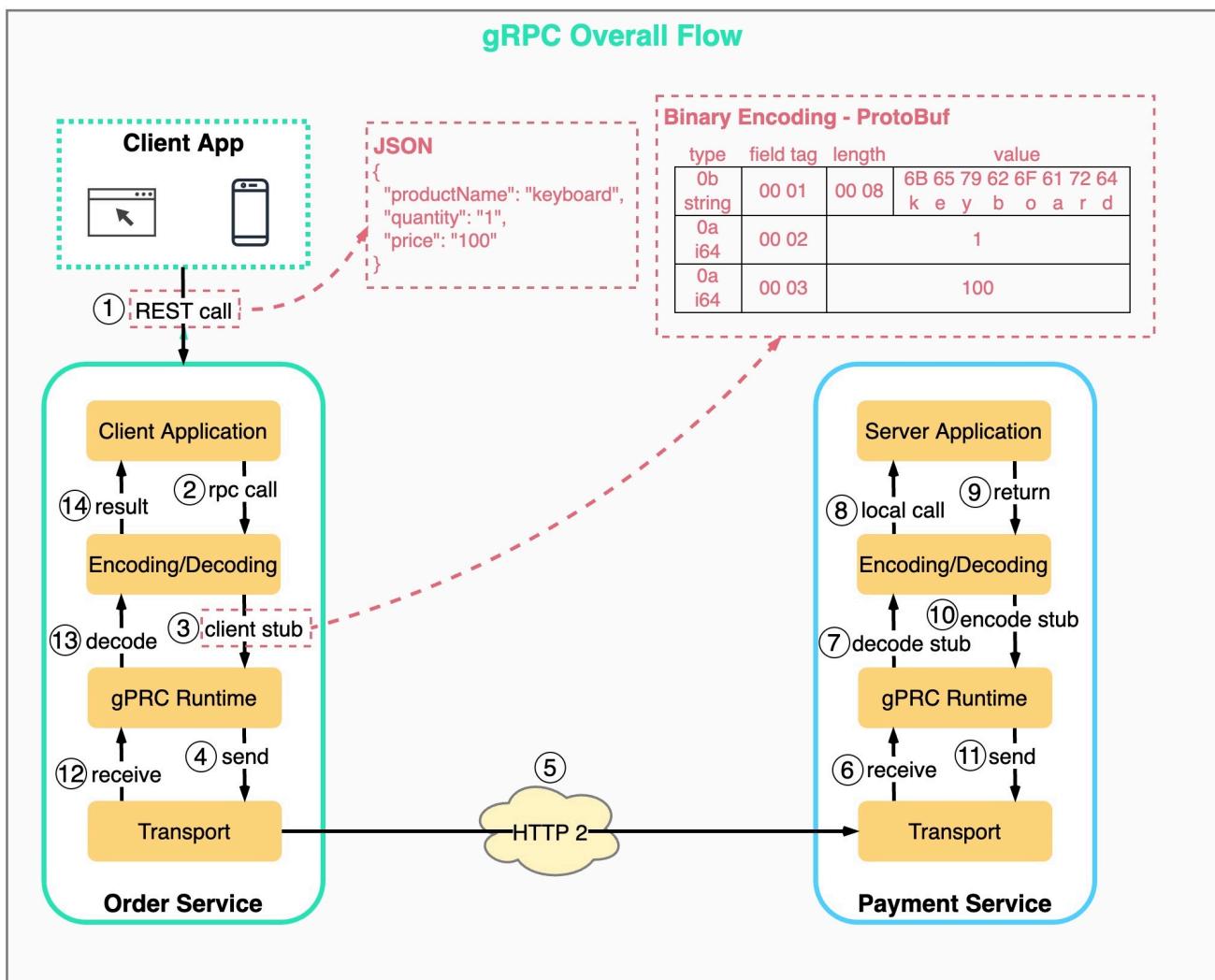
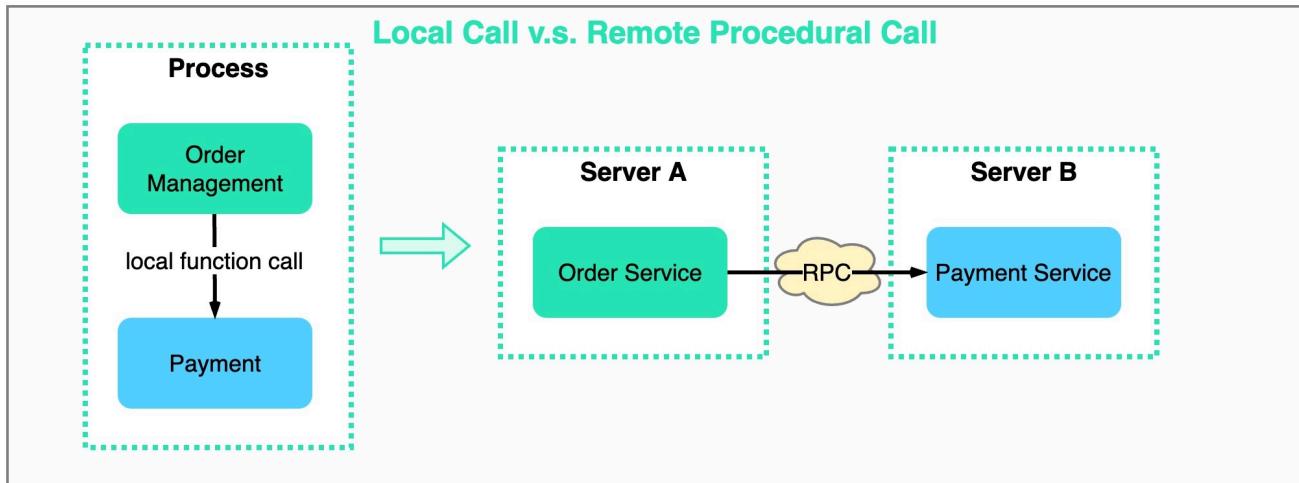
gRPC 是如何工作的？

RPC（远程过程调用，Remote Procedure Call）之所以被称为“远程”是因为在微服务架构下，服务被部署到不同服务器时，它允许远程服务之间进行通信。从用户的角度来说，使用它就像使用本地函数调用一样。

下图说明了 gRPC 的整个数据流。

How does gRPC Work?

 blog.bytebybytego.com



步骤 1：客户端发起一个 REST 调用。请求体通常是 JSON 格式。

步骤 2 - 4: 订单服务 (gRPC 客户端) 收到 REST 调用后, 对其进行转换, 然后向支付服务发起一个 RPC 请求。gRPC 将**客户端存根 (client stub) **编码为二进制格式, 然后将其发送到低层次的传输层。

步骤 5: gRPC 通过 HTTP2 在网络上发送数据包。由于二进制编码和网络优化, gRPC 据说比 JSON 快 5 倍。

步骤 6 - 8: 支付服务 (gRPC 服务器) 收到来自网络的数据包后, 对其进行解码, 然后调用服务器应用程序。

步骤 9 - 11: 服务器应用程序返回调用结果, 将其编码后发送到传输层。

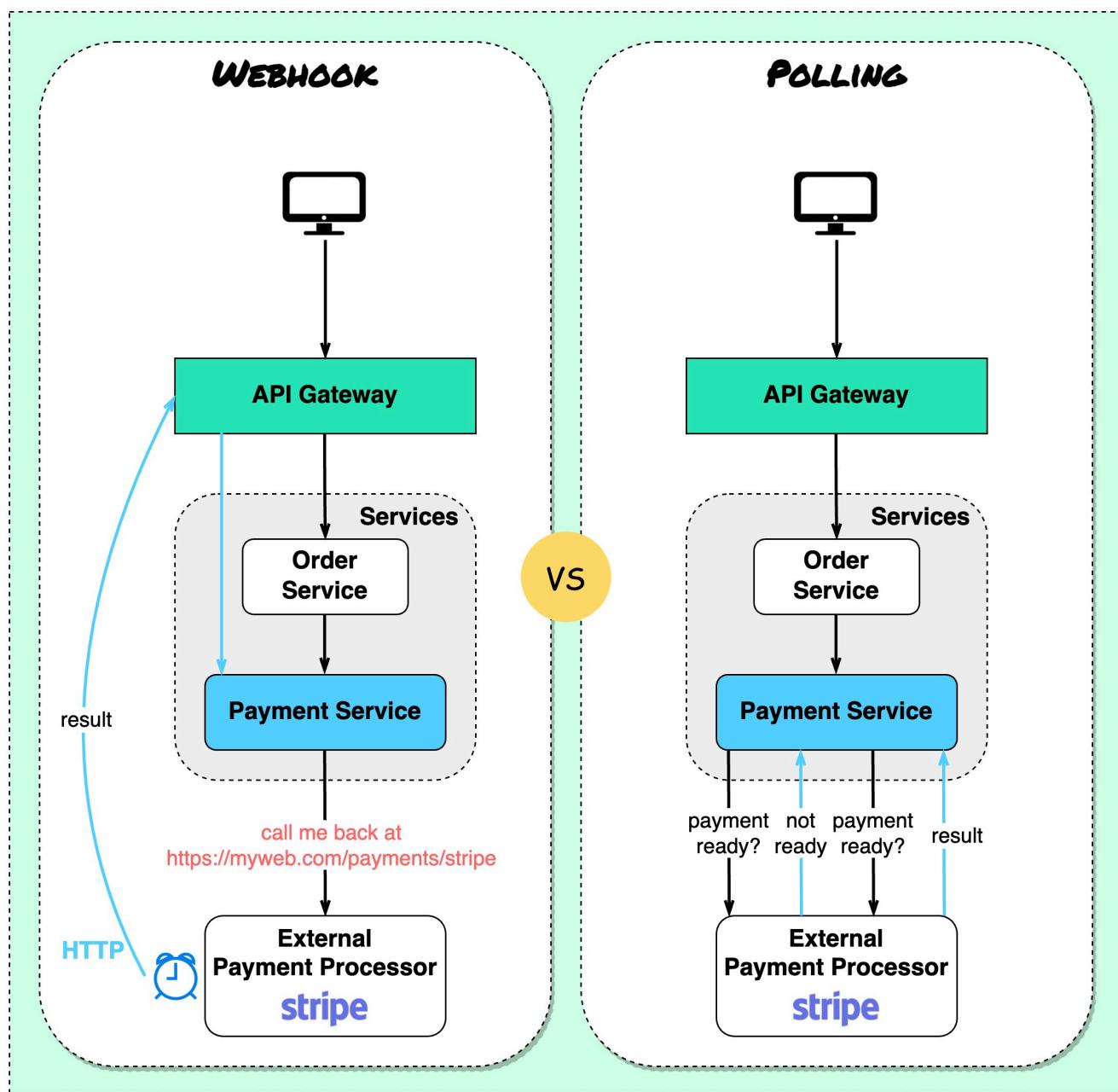
步骤 12 - 14: 订单服务接收数据包后解码, 然后将结果发送给客户端应用。

什么是 webhook?

下图显示了轮询 (polling) 和 Webhook 之间的比较。

What is a Webhook?

 blog.bytebytego.com



假设我们经营着一个电子商务网站。客户端通过 API 网关发送订单到订单服务，订单服务转到支付服务请求支付交易。支付服务接着与外部的支付服务提供商（payment service provider, PSP）通信以完成交易。

处理与外部支付服务提供商（PSP）的通信存在两种方法。

1. 短轮询

在发送支付请求到 PSP 后，支付服务不断向 PSP 询问支付状态。几轮后，PSP 终于返回状态。

短轮询有两个缺点：

- 持续轮询状态需要消耗支付服务的资源。
- 外部服务与支付服务直接通信，存在安全漏洞。

2. Webhook

我们可以向外部服务注册一个 webhook。这意味着：当你有请求更新的时候，通过某个 URL 对我进行回调。当 PSP 完成处理后，它会调用 HTTP 请求来更新支付状态。

这样就改变了编程范式，并且支付服务不再需要浪费资源来轮询支付状态。

那如果 PSP 永不回调怎么办？我们可以设置一个内部定时任务，每小时检查付款状态。

Webhook 通常被称为反向 API 或者推送 API，因为服务器向客户端发送 HTTP 请求。使用 webhook 时，我们需要注意三件事：

1. 我们需要为外部服务调用设计一个合适的 API。
2. 出于安全考虑，我们需要在 API 网关中设置适当的规则。
3. 我们需要在外部服务中注册正确的 URL。

如何改善 API 性能？

下图显示了提高 API 性能的五种常见技巧。

How to Improve API Performance?

PAGINATION	<pre> graph LR Client[Client] -- "request" --> Services[Services] Services --> Result[Result: 3 items] Result -.-> Page1[page 1] Result -.-> Page2[page 2] Result -.-> Page3[page 3] Page1 --> Client Page2 --> Client Page3 --> Client </pre>	<ul style="list-style-type: none"> an ordinal numbering of pages handles a large number of results
ASYNC LOGGING	<pre> graph LR Logs[Logs] --> Buffer[Buffer] Buffer -- "flush" --> Disk[Disk] </pre>	<ul style="list-style-type: none"> send logs to a lock-free ring buffer and return flush to the disk periodically higher throughput and lower latency
CACHING	<pre> graph LR Client[Client] -- "read from cache" --> Cache[Cache] Cache --> DB[DB] DB -- "read from db" --> Cache Cache -- "update cache" --> DB Cache --> Client </pre>	<ul style="list-style-type: none"> store frequently used data in the cache instead of database query the database when there is a cache miss
PAYLOAD COMPRESSION	<pre> graph LR Client[Client] -- "compress payload" --> Request[request] Request --> Services[Services] Services -- "compress payload" --> Response[response] Response --> Client </pre>	<ul style="list-style-type: none"> reduce the data size to speed up the download and upload
CONNECTION POOL	<pre> graph LR Client1[Client] --> Pool((connection pool)) Client2[Client] --> Pool Client3[Client] --> Pool Pool --> DB[DB] </pre>	<ul style="list-style-type: none"> opening and closing DB connections add significant overhead a connection pool maintains a number of open connections for applications to reuse

Reference: Rapid API

分页 (Pagination)

当结果很大的时候，常用此种优化方式。结果会以类似流的形式，分片返回客户端，以提高服务响应能力。

异步日志记录 (Asynchronous Logging)

同步日志记录每次调用时都会与磁盘进行交互，这会降低系统速度。异步日志记录首先将日志发送到无锁缓冲区并立即返回。日志将会被定期刷新到磁盘。这显著减少了 I/O 开销。

缓存 (Caching)

我们可以将经常访问的数据存储到缓冲区中。客户端可以先查询缓存，而不是直接访问数据库。如果缓存未命中，客户端可以查询数据库。像 Redis 这样的缓存将数据存储在内存中，因此和数据库相比，数据访问要快得多。

数据压缩 (Payload Compression)

可以使用 gzip 等工具压缩请求和响应，这样，传输的数据大小会小得多。这可以加速上传和下载速度。

连接池 (Connection Pool)

访问资源时，我们通常需要从数据库中加载数据。打开正关闭的数据库连接会增加大量开销。因此，我们应该通过一个包含打开的连接的连接池来连接数据库。连接池负责管理连接的生命周期。

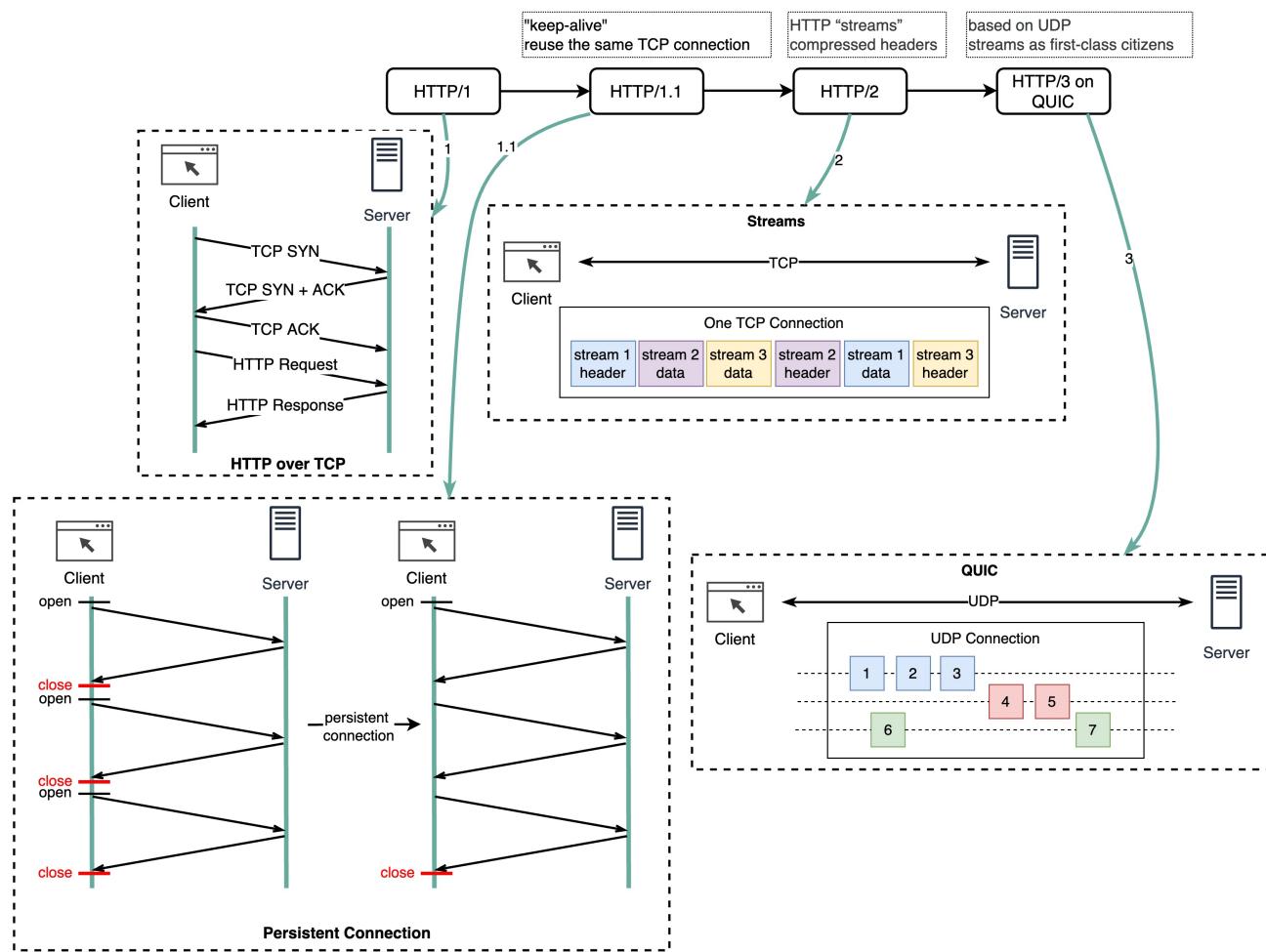
HTTP 1.0 -> HTTP 1.1 -> HTTP 2.0 -> HTTP 3.0 (QUIC)

每一代 HTTP 都解决了些什么问题呢？

下图说明了每一代 HTTP 的主要特性。

How did we get to HTTP/3?

ByteByteGo



- HTTP 1.0 在 1996 年才最终被确定下来并完整记录。对于同一服务器的每个请求都需要建立独立的 TCP 连接。

- HTTP 1.1 在 1997 年发布。TCP 连接可以保持打开状态以供重用（持久连接），但它并没有解决 HOL (head-of-line, 队头) 阻塞问题。

HOL 阻塞 —— 当浏览器允许的并发请求数用完时，后续请求需要等待前面的请求完成。

- HTTP 2.0 在 2015 年发布。通过请求复用解决了 HOL 问题，在应用层消除了 HOL 阻塞，但在传输层 (TCP) 仍然存在 HOL。

如图所示，HTTP 2.0 引入了 HTTP “流”的概念：这是一种抽象，允许在同一个 TCP 连接上多路复用不同的 HTTP 交换。每个流不需要按序发送。

- HTTP 3.0 初稿于 2020 年发布。它是 HTTP 2.0 的拟议继任者。它使用 QUIC 替代 TCP 作为底层传输协议，从而消除了传输层的 HOL 阻塞。

QUIC 基于 UDP。它将流作为传输层的一等公民引入。QUIC 流共享相同的 QUIC 连接，因此创建新流不需要额外的握手和慢启动，但 QUIC 流是独立传送的，因此在大多数情况下，一个流的数据包丢失不会影响其他流。

SOAP vs REST vs GraphQL vs RPC

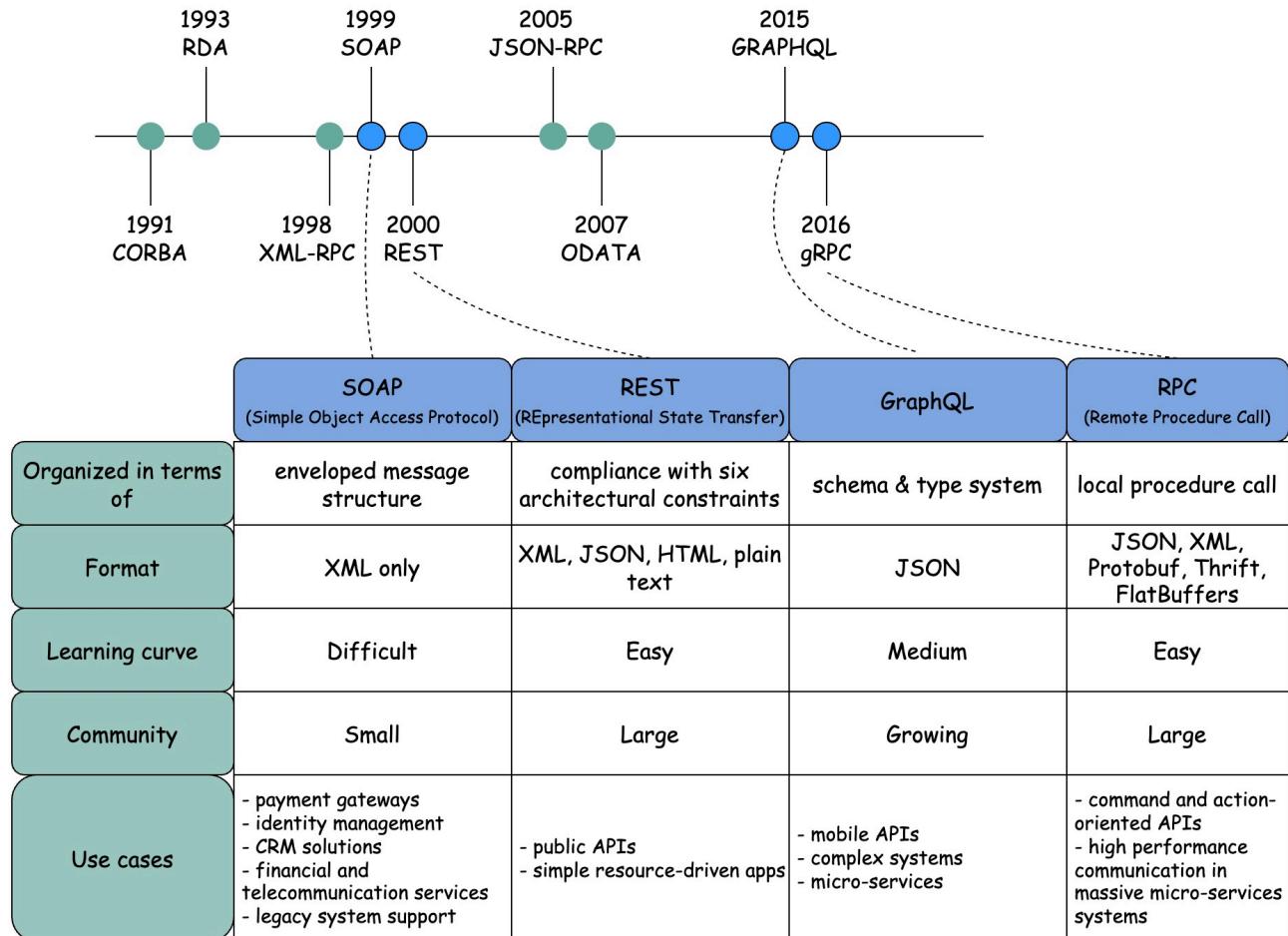
下图显示了 API 时间轴和 API 风格比较。

随着时间的推移，发布了不同的 API 架构风格。它们每一个都有自己的标准化数据交换模式。

你可以在图中查看每种风格的用例。

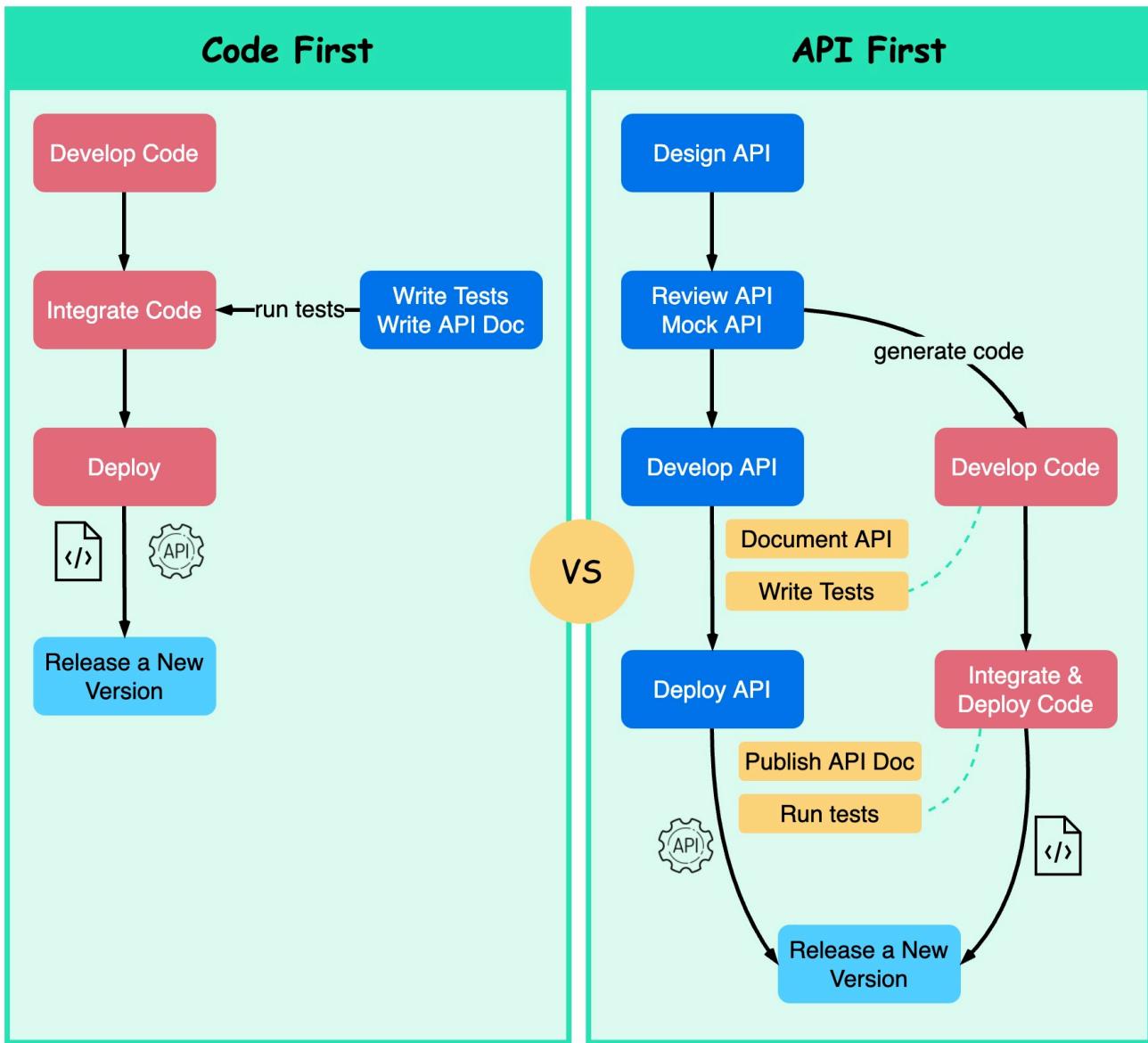
API Architectural Styles Comparison

Source: altexsoft



代码优先 (Code First) vs. API 优先 (API First)

下图显示了代码优先开发和 API 优先开发之间的区别。为什么我们要考虑 API 优先的设计呢？



- 微服务增加了系统复杂性，我们划分服务，以提供系统的不同功能。虽然这种架构有利于解耦和分离，但是，我们还需要处理服务之间的各种通信。

在编写代码并仔细定义服务边界之前，最好先考虑下系统的复杂性。

- 不同的功能团队需要对功能的理解等方面沟通一致，而专门的功能团队只负责自己的组件和服务。建议组织通过 API 设计确保沟通上的一致。

在写代码前，我们就可以模拟请求和响应来验证 API 设计。

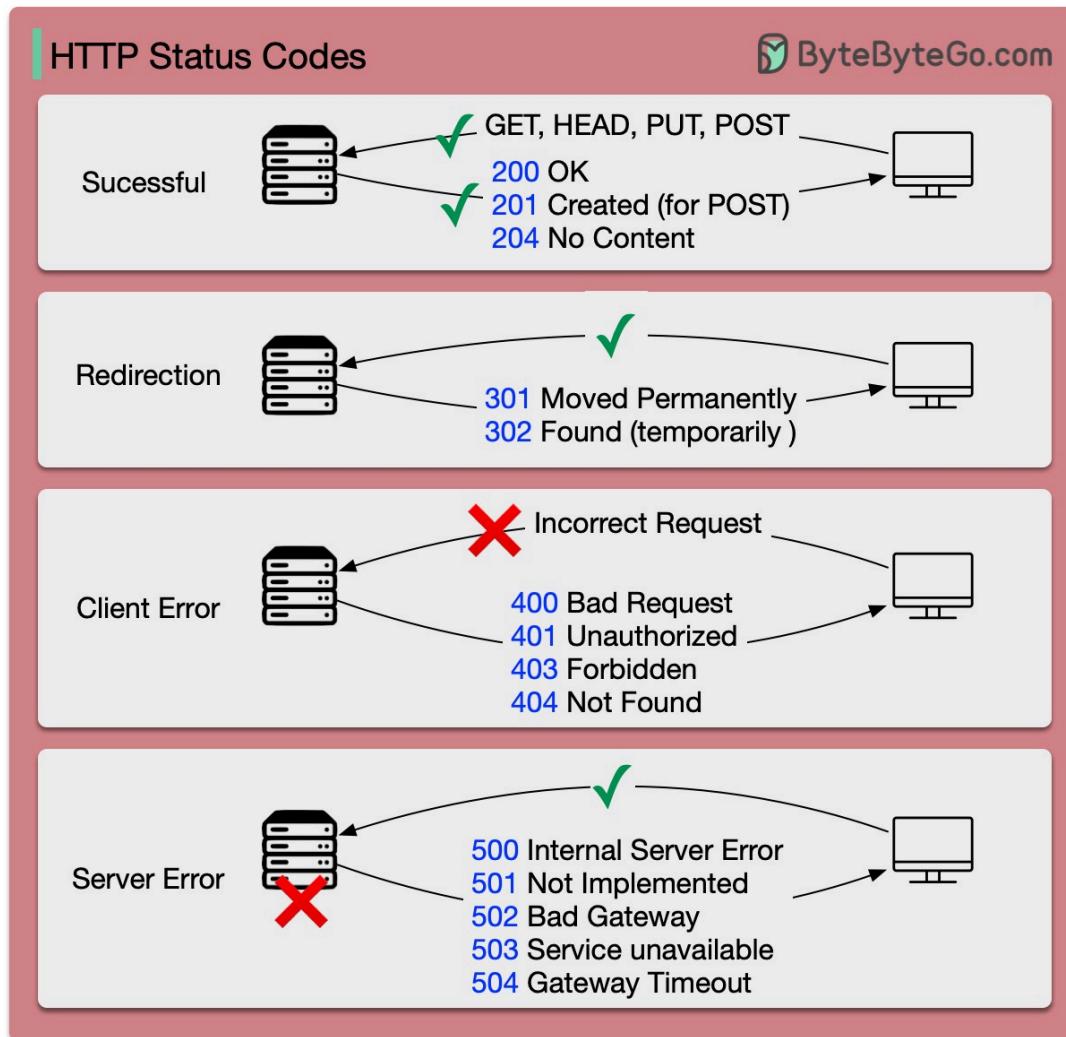
- 提高软件质量和开发人员生产力。由于我们在项目启动时已经解决了大部分不确定因素，因此，整体开发过程会更加顺利，软件质量也会得到了很大的提升。

开发人员也会对这个过程感到满意，因为他们可以专注于功能开发，而不用要应对突如其来的（功能）变化。

在项目生命周期结束时出现意外的可能性会降低。

因为我们先设计了API，所以可以在开发代码的同时设计测试。在某种程度上，使用API优先开发的同时也可以有 TDD（测试驱动设计）。

HTTP 状态码



HTTP 响应码分为五类：

- 信息性 (Informational) (100-199)

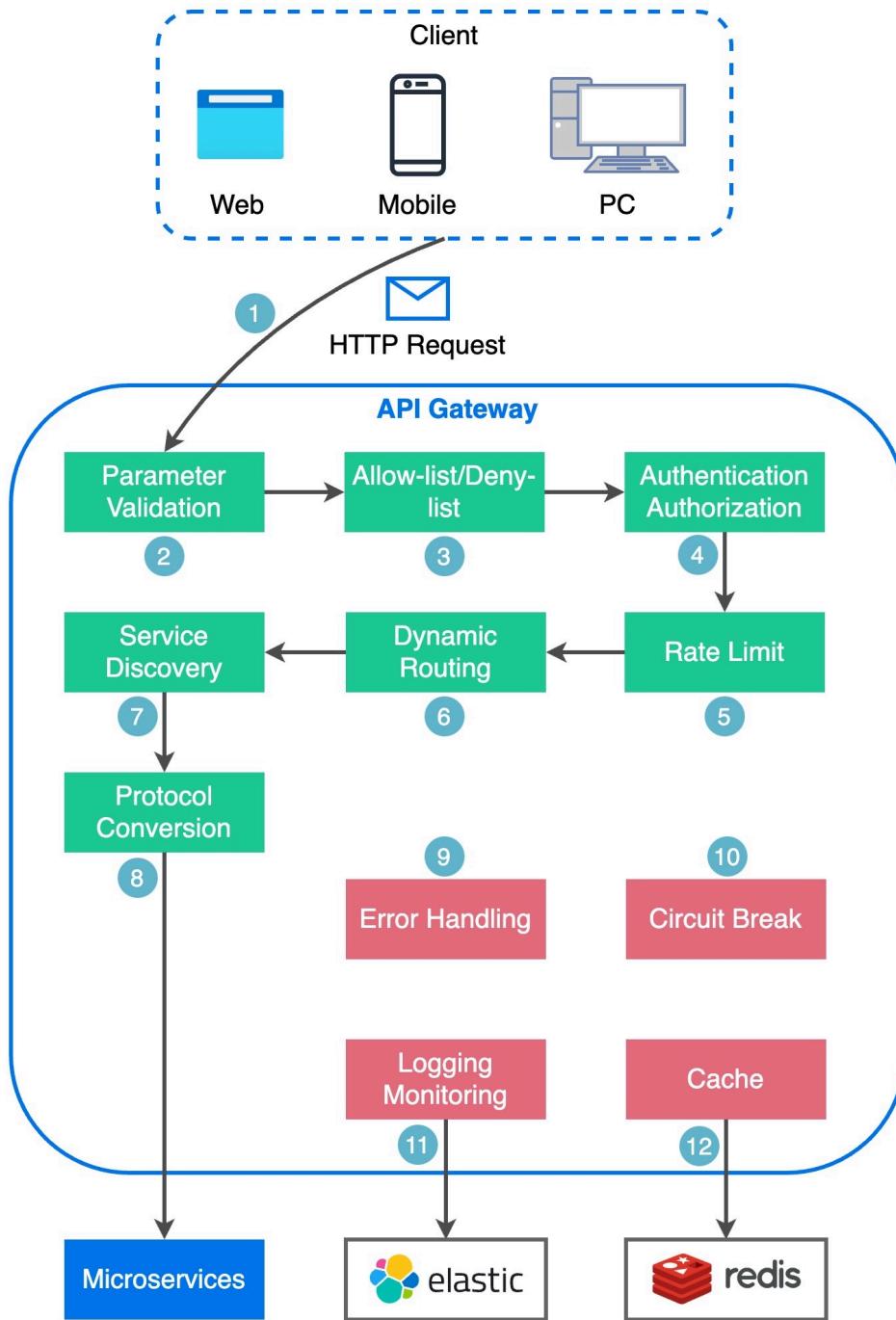
- 成功 (Success) (200-299)
- 重定向 (Redirection) (300-399)
- 客户端错误 (400-499)
- 服务端错误 (500-599)

API 网关有什么作用？

下图显示了其细节。

What does API Gateway do?

 blog.bytebytogo.com



步骤 1 - 客户端发送 HTTP 请求给 API 网关。

步骤 2 - API 网关解析和验证 HTTP 请求的属性。

步骤 3 - API 网关进行允许列表/拒绝列表检查。

步骤 4 - API 网关和身份提供商交互，进行身份验证和授权。

步骤 5 - 对请求应用速率规则，如果超过限制，请求将被拒绝。

步骤 6 和 7 - 现在，请求通过了基本检查，API 网关通过路径匹配将请求路由到相关服务。

步骤 8 - API 网关将请求转换为合适的协议，然后将其发送给后端微服务。

步骤 9-12: API 网关可以妥善处理错误，并且在错误需要较长时间才能恢复（熔断）的时候处理故障。它还可以利用 ELK (Elastic-Logstash-Kibana) 堆栈进行日志记录和监控。有时，我们还会在 API 网关中缓存数据。

我们要如何设计高效安全的 API?

下图用购物车的例子，显示了典型的 API 设计。

Design Effective & Safe APIs

 blog.bytebybytego.com



Design a Shopping Cart

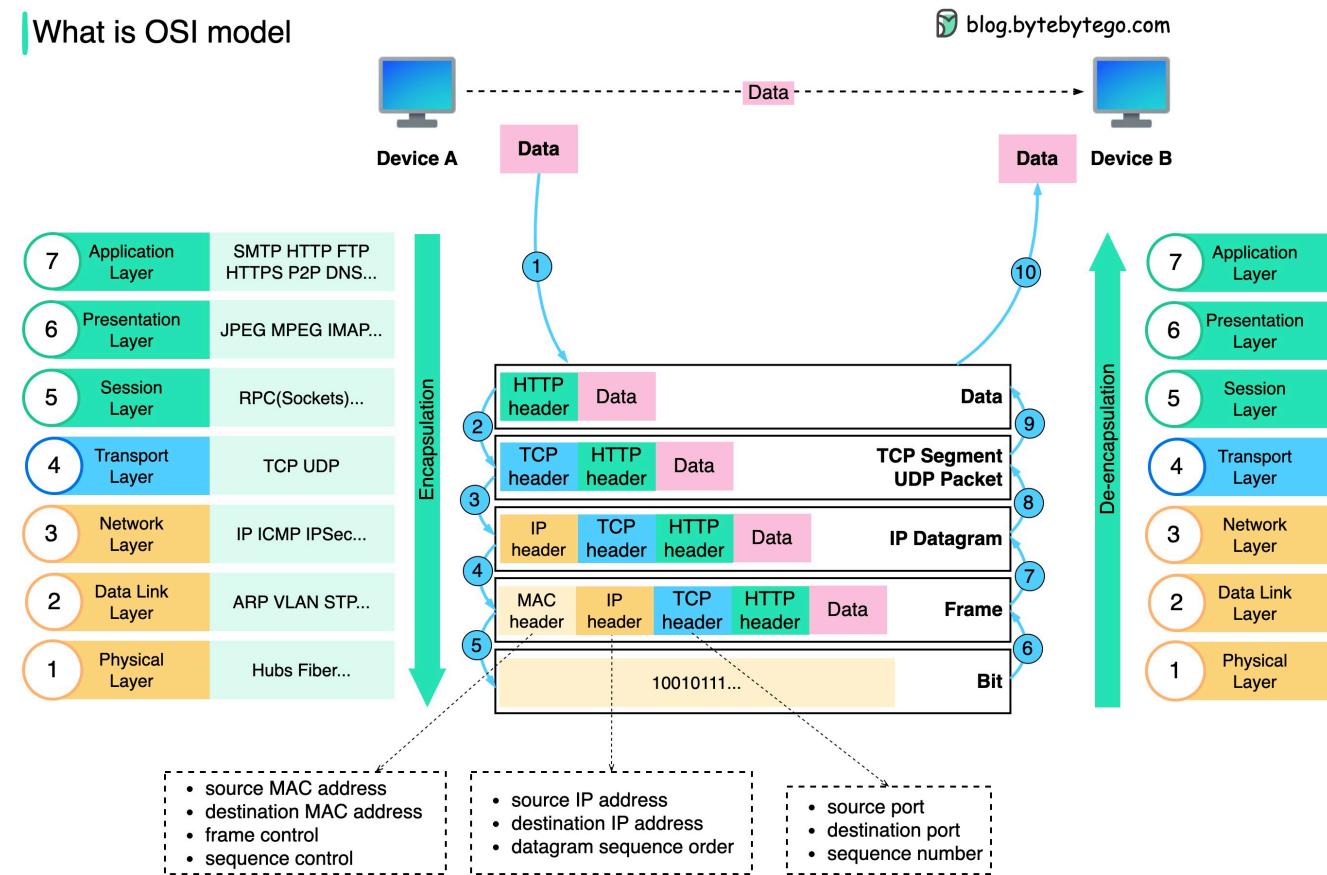
Use resource names (nouns)	GET /querycarts/123	GET /carts/123
Use plurals	GET /cart/123	GET /carts/123
Idempotency	POST /carts	POST /carts {requestId: 4321}
Use versioning	GET /carts/v1/123	GET /v1/carts/123
Query after soft deletion	GET /carts	GET /carts? includeDeleted=true
Pagination	GET /carts	GET /carts? pageSize=xx&pageToken=xx
Sorting	GET /items	GET /items? sort_by=time
Filtering	GET /items	GET /items? filter=color:red
Secure Access	X-API-KEY=xxx	X-API-KEY = xxx X-EXPIRY = xxx X-REQUEST-SIGNATURE = xxx 
Resource cross reference	GET /carts/123? item=321	GET /carts/123/items/321
Add an item to a cart	POST /carts/123? addItem=321	POST /carts/123/items:add { itemId: "items/321" }
Rate limit	No rate limit - DDoS	Design rate limiting rules based on IP, user, action group etc

注意，API 设计不仅仅是 URL 路径设计。大多数情况下，我们需要选择合适的资源名称、标识符和路径模式。设计合适的 HTTP 头部字段或者设计高效的 API 网关的限速规则同样重要。

TCP/IP 封装

数据是如何通过网络发送的？为什么在 OSI 模型中，我们需要那么多层？

下图显示了数据通过网络传输时如何被封装和解封装的。



步骤 1：当设备 A 通过 HTTP 协议经过网络向设备 B 发送数据时，它首先在应用层添加一个 HTTP 头。

步骤 2：然后一个 TCP 或者 UDP 标头被添加到数据中。在传输层将其封装成 TCP 报文段。标头包含源端口、目标端口和序列号。

步骤 3：接着在网络层增加 IP 标头，对这些报文段进行封装。IP 标头包含源/目标 IP 地址。

步骤 4：IP 在数据链路层，向数据报添加 MAC 标头，其中包含源/目标 MAC 地址。

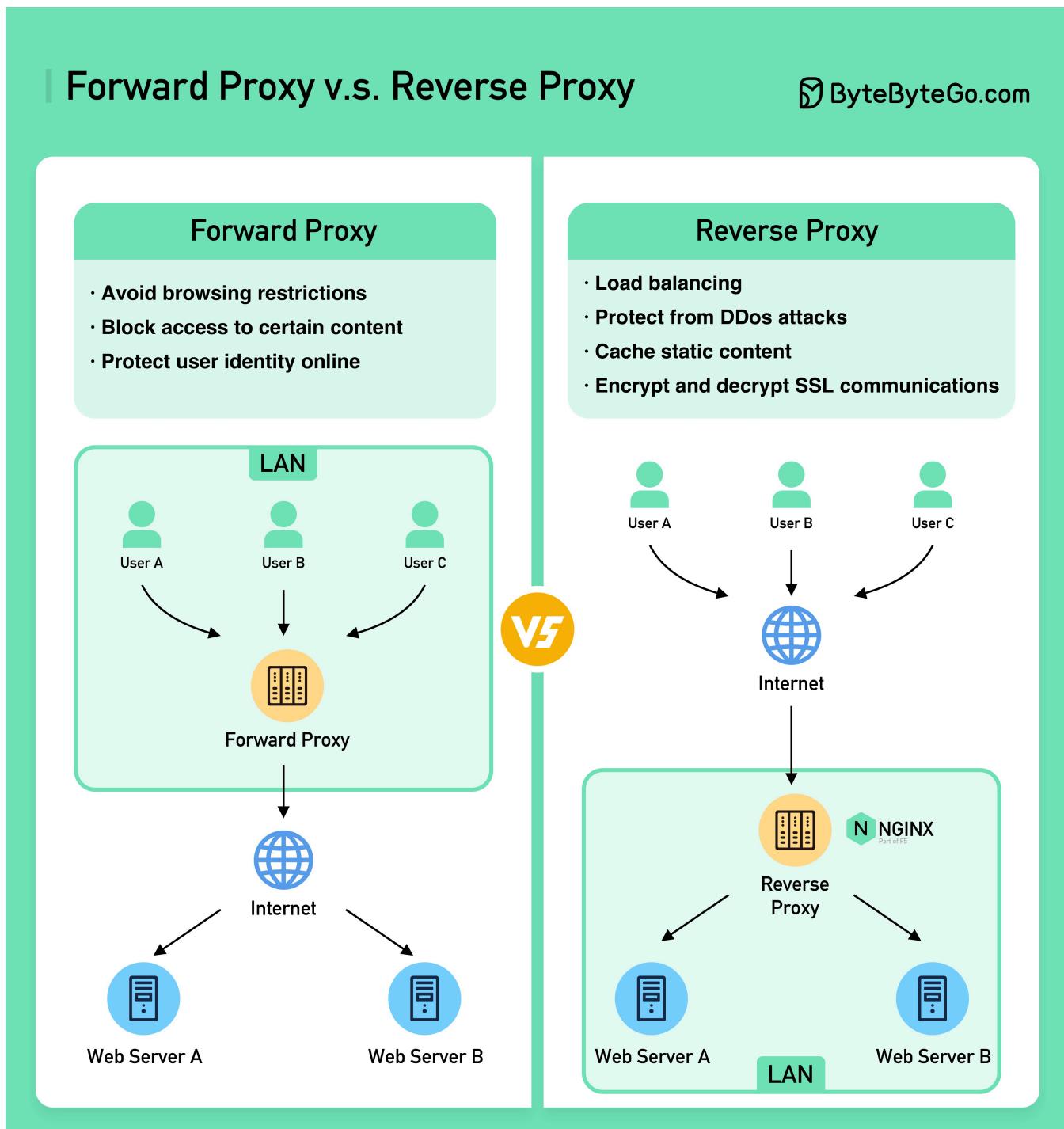
步骤 5：封装后的数据帧被发送到物理层，然后以二进制位的形式通过网络发送。

步骤 6-10：当设备 B 从网络上接收到这些二进制位数据时，执行解封装过程（封装过程的反向过程）。逐层去除数据头部，最后，设备 B 得以读取数据。

网络模型中之所以需要层的概念，是因为每一层都专注于自己的职责。每层都可以依赖标头来处理指令，而不需要知道最后一层数据的含义。

为什么 Nginx 被称为“反向”代理？

下图显示了正向代理（Forward Proxy）和反向代理（Reverse Proxy）之间的区别。



正向代理指的是位于用户设备和网络之间的服务器。

正向代理通常用于：

1. 保护客户端
2. 规避浏览限制
3. 阻止访问某些内容

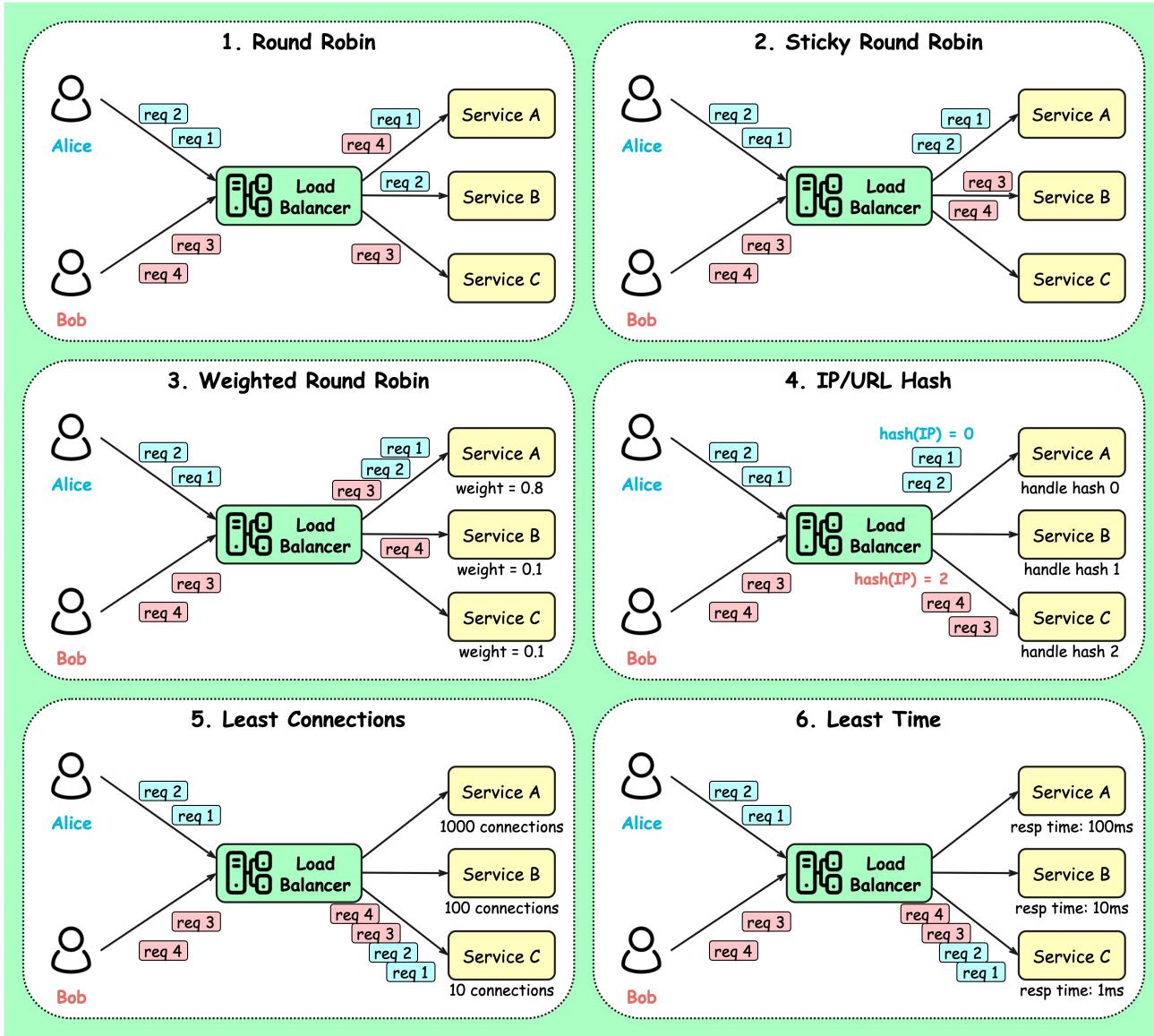
反向代理指的是一个这样的服务器，它接收来自客户端的请求，将其转发给网络服务器，然后将结果返回给客户端，就好像代理服务器已经处理了请求一样。

反向代理适用于：

1. 保护服务端
2. 负载均衡
3. 缓存静态内容
4. 加密和解密 SSL 通信

常见的负载均衡算法是什么？

下图显示了六种常见的算法。



- 静态算法

1. 轮询

客户端请求按顺序发送到不同服务实例。通常要求这些服务实例是无状态的。

2. 粘性轮询

轮询算法的改进版。如果 Alice 的第一个请求被发送给服务 A，那么她后续的请求也会被发送到服务 A。

3. 加权轮询

管理员可以指定每个服务的权重。权重较高的服务比权重较低的其他服务处理更多请求。

4. 哈希值

算法对传入请求的 IP 或者 URL 应用哈希函数。基于哈希函数的返回值将请求路由到相应实例。

- 动态算法

1. 最少连接数

新请求将被发送到并发连接数最少的服务实例。

2. 最短响应时间

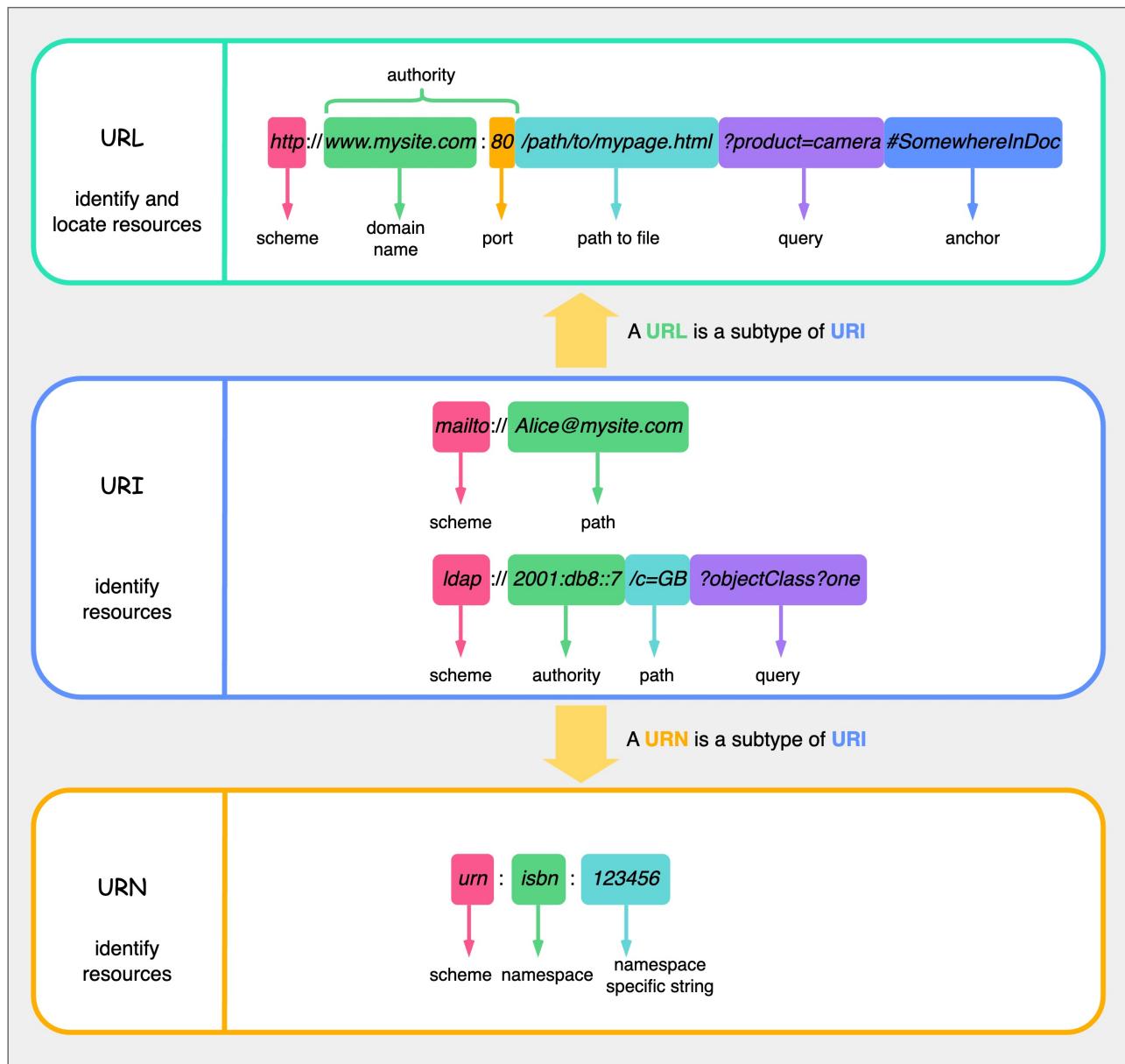
新请求将被发送到响应最快的服务实例。

URL, URI, URN - 你知道它们之间的差别吗?

下图显示了 URL、URI 和 URN 的比较。

URL vs URI vs URN

 blog.bytebytogo.com



- URI

URI 表示统一资源标识符（Uniform Resource Identifier）。它标识网络上的逻辑或物理资源。URL 和 URN 是 URI 的子类型。URL 定位资源，而 URN 命名资源。

URI 由以下部分组成： scheme:[//authority]path[?query][#fragment]

- URL

URL 表示统一资源定位符（Uniform Resource Locator），是 HTTP 的关键概念。它是网络上

唯一资源的地址。它可以与 FTP 和 JDBC 等其他协议一起使用。

- URN

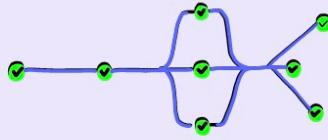
URN 表示统一资源名称 (Uniform Resource Name) 。它使用 urn 方案。不能用 URN 来定位资源。图中给出了一个简单示例，它由命名空间和特定于命名空间的字符串组成。

如果您想了解有关该主题的更多详细信息，我建议您查看 [W3C 的说明](#)。

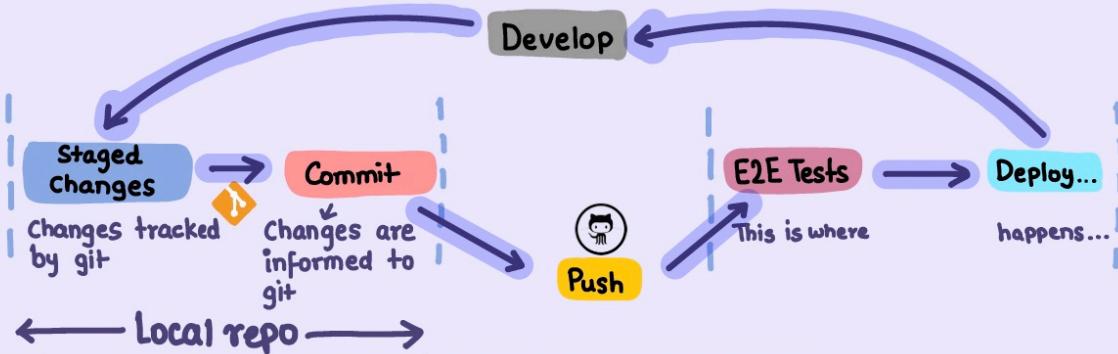
CI/CD

简单解释 CI/CD 管道

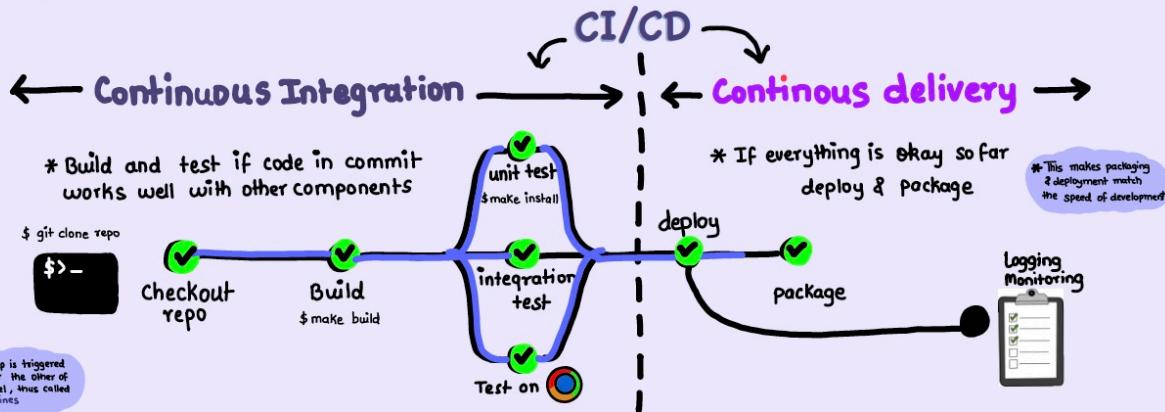
CI/CD Pipelines



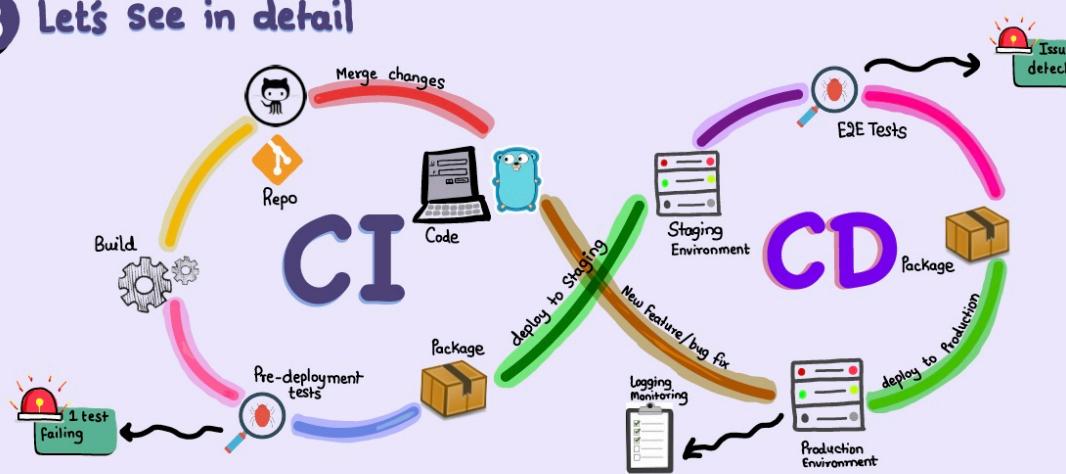
1 Software development Lifecycle



2



3 Let's see in detail



第 1 节 - SDLC 和 CI/CD

软件开发生命周期 (software development life cycle, SDLC) 由几个关键阶段组成：开发、测试、部署和维护。CI/CD 自动化并集成这些阶段，以实现更快、更可靠的发布。

当代码被推送到 git 仓库时，它会触发自动构建和测试过程。端到端 (e2e) 测试用例被运行以验证代码。如果测试通过，代码可以自动部署到预发布 (staging) / 生产 (production) 环境。如果发现问题，代码将被发送回开发以修复错误。这种自动化为开发人员提供了快速反馈，并降低了生产中出现错误的风险。

第 2 节 - CI 和 CD 的区别

持续集成 (Continuous Integration, CI) 自动执行构建、测试和合并过程。只要提交代码，它就会运行测试以尽早检测集成问题。它鼓励频繁的代码提交和快速反馈。

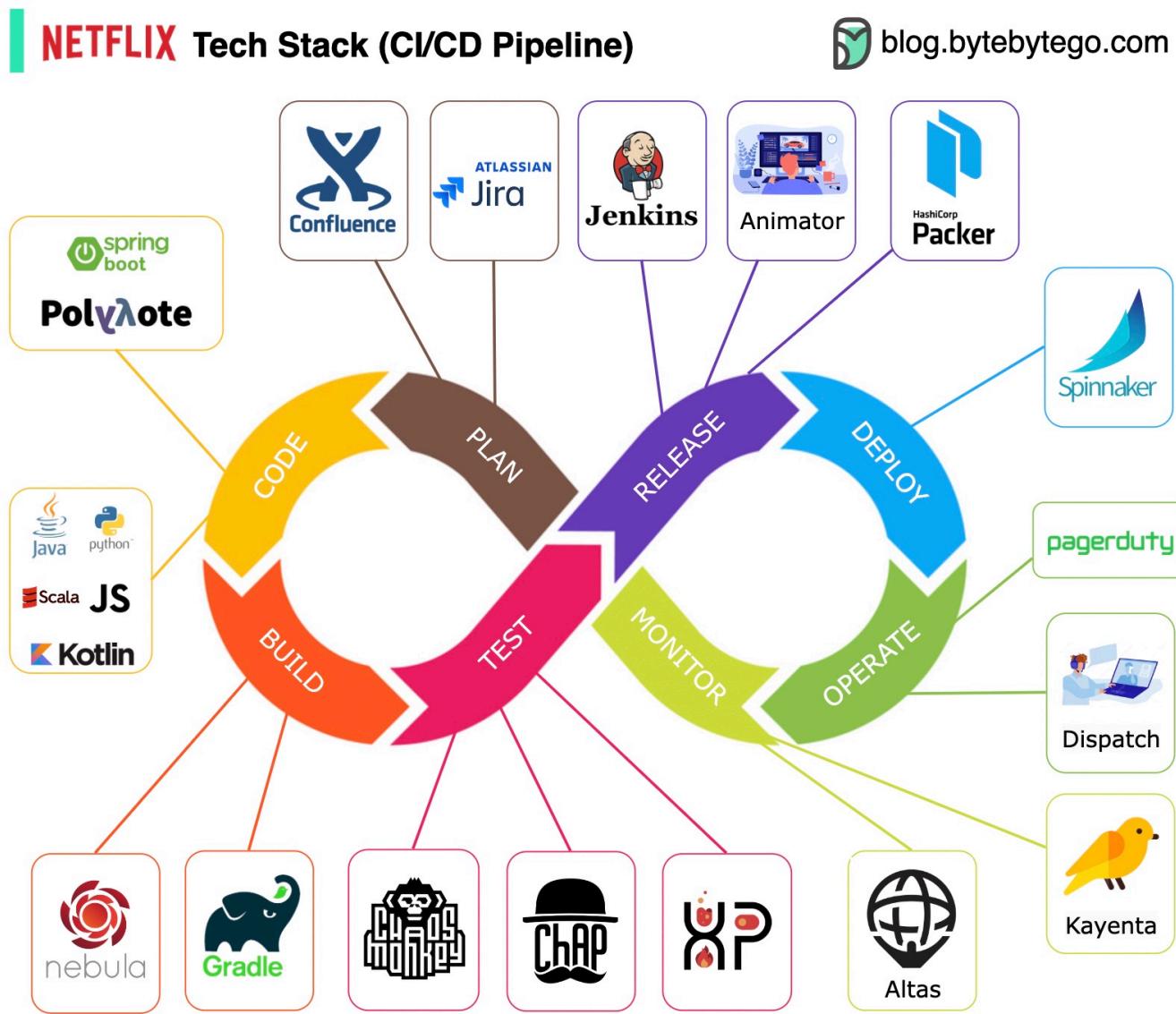
持续交付 (Continuous Delivery, CD) 可自动执行基础架构更改和部署等发布流程。它确保可以通过自动化工作流程随时可靠地发布软件。CD 还可以自动执行生产部署之前所需的手动测试和批准步骤。

第 3 节 - CI/CD 管道

典型的 CI/CD 管道有几个相连的阶段：

- 开发者提交代码更改到源代码管理
- CI 服务器检测变更，然后触发构建
- 对代码进行编译测试（单元测试，集成测试）
- 将测试结果报告给开发者
- 一旦成功，就会将其部署到预发布环境
- 发布前，可能会在预发布环境上进行进一步测试
- CD 系统将批准的变更发布到生产环境

Netflix 技术栈 (CI/CD 管道)



规划: Netflix 工程使用 JIRA 进行规划, 使用 Confluence 进行文档编制。

编码: Java 是后端服务的主要编程语言, 同时, 不同场景中也会使用其他语言。

构建: Gradle 主要用于构建, 构建不同的 Gradle 插件以支持不同的使用场景。

打包: 包和依赖项被打包到 Amazon 系统映像 (AMI) 中以供发布。

测试: 测试强调生产文化对构建混沌工具 (chaos tool) 的关注。 (译注: 混沌测试是一种手段, 随机在系统里触发一些故障, 看系统的反映情况。)

部署: Netflix 使用自建的 Spinnaker 进行金丝雀部署 (canary rollout deployment)

| 更多关于“金丝雀部署”，可以查看[这里](#)

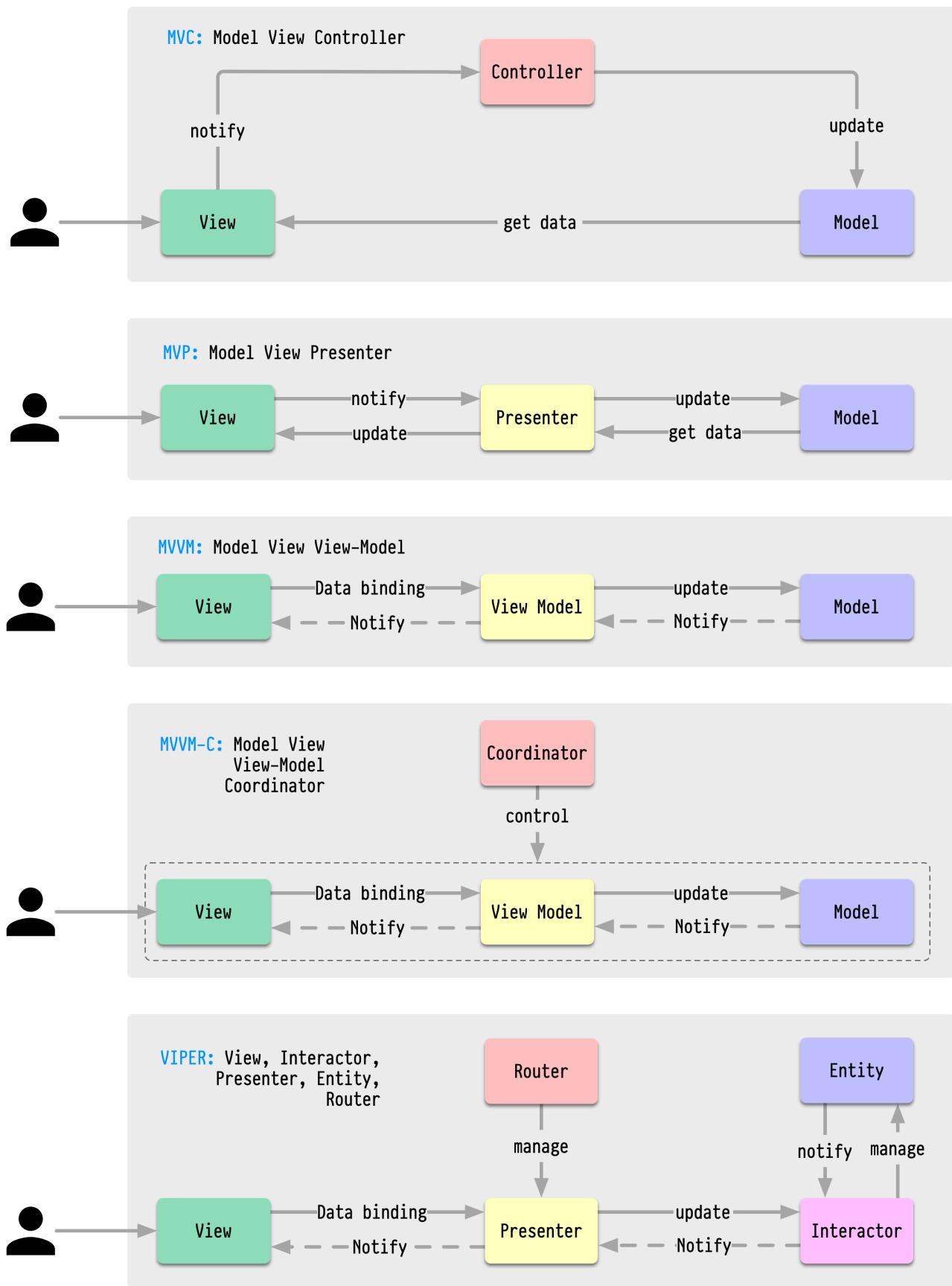
监控：监控指标集中在 Atlas 中，使用 Kayenta 来检测异常情况。

事件（incident）报告：事件按照优先级调度，使用 PagerDuty 进行事件处理。

架构模式

MVC、MVP、MVVM、MVVM-C 和 VIPER

这些架构模式是应用程序开发（无论是是在 iOS 还是 Android 平台上）中最常用的模式。开发人员引入它们是为了克服早期模式的局限性。那么，它们有何不同？



- MVC，最古老的模式，最早可追溯到近 50 年前
- 每一种模式都有一个“视图” (view, V) ，负责显示内容和接收用户输入
- 大部分的模式都包括一个“模型 (model, M) ”，用来管理业务数据
- “控制器 (Controller) ”、“主持人 (Presenter) ”和“视图模型 (View-Model) ”都是转换器，负责视图和模型 (VIPER 模式中的“实体”) 之间的通信。

每一位开发者都必须知道的 18 个关键设计模式

模式是针对常见设计问题的可重用解决方案，以实现更顺畅高效的开发过程。它们是构建更好的软件结构的蓝图。下面是一些最流行的模式：

Abstract Factory

Family creator

Create groups of related items

Builder

Lego master

Build object step by step

Prototype

Cloner

Create copies from examples

Singleton

The one and only

With just one instance

Adapter

Universal plug

Connect different interfaces

Bridge

Connector

Link what is to how it works

Composite

Tree builder

Create tree-like structure

Decorator

Customizer

Add new features to existing object

Facade

One-stop shop

Single interface to all functions

Flyweight

Space saver

Share small, reusable items

Proxy

Middle man

Represent another object

Chain of responsibility

Replayer

Relay requests until it is handles

Command

Task wrapper

Turn a request into object

Iterator

Explorer

Assess element one by one

Mediator

Hub

Simplify communication between classes

Memento

Capsule

Capture and store object state

Observer

Broadcaster

Notify others about the change

Visitor

Guests

Explore an object without changing it

- 抽象工厂 (Abstract Factory) 模式：族群创建者 - 创建一组或多组相关对象。
- 建造者 (Builder) 模式：乐高大师 —— 逐步构建对象，分离创建和外观。
- 原型 (Prototype) 模式：克隆制造者 —— 创建完备实例的副本。
- 单例 (Singleton) 模式：唯一实例 —— 一个只有一个实例的特殊类。
- 适配器 (Adapter) 模式：通用插头 —— 连接具有不同接口的东西。

- 桥接 (Bridge) 模式：功能连接器 —— 将对象的工作方式连接到其功能。
- 组合 (Composite) 模式：树形构建器 —— 形成简单和负责部分的树形结构。
- 装饰器 (Decorator) 模式：定制器 —— 给对象添加功能而无需改变其内核。
- 外观 (Facade) 模式：一站式商店 —— 通过单一简化接口来代表整个系统。
- 享元 (Flyweight) 模式：节约空间小能手 —— 高效共享小型可重用项。
- 代理 (Proxy) 模式：替身演员 —— 代表另一个对象，控制访问或者操作。
- 责任链 (Chain of Responsibility) 模式：请求中继 —— 将请求通过一系列对象传递，直至被处理。
- 命令 (Command) 模式：任务包装器 —— 将请求转化为一个准备就绪的对象。
- 迭代器 (Iterator) 模式：集合探索者 —— 逐个访问集合中的元素。
- 中介者 (Mediator) 模式：通信中心 —— 简化不同类之间的交互。
- 备忘录 (Memento) 模式：时间胶囊 —— 捕获并恢复对象状态。
- 观察者 (Observer) 模式：新闻广播员 —— 通知类其他对象的变化。
- 访问者 (Visitor) 模式：熟练的访客 —— 为类增加新操作而不对其进行更改。

数据库

关于云服务中不同类型的数据库的一个便携速查表

Cloud Database Cheat Sheet

 blog.bytebytogo.com

DB Type	
Structured	Relational 
	Columnar 
	Key Value 
	In-Memory 
	Wide Column 
	Time Series 
	Immutable Ledger 
	Geospatial 
	Graph 
	Document 
Semi Structured	Text Search 
	Blob 
UnStructured	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
	warn: This section is currently under construction.
aws	
RDS	
Redshift	
DynamoDB	
ElastiCache	
Keyspaces	
Timestream	
Quantum Ledger DB	
Keyspaces	
Neptune	
Document DB	
OpenSearch	
S3	
Azure	
SQL Database	
Synapse Analytics	
Cosmos DB	
Azure Cache for Redis	
Time Series Insights	
Confidential Ledger	
Cosmos DB	
BigTable	
BigTable	
BigQuery	
CloudSpanner	
BigTable	
CloudSpanner	
FireStore	
Cognitive Search	
Cloud Storage	
Google Cloud	
Cloud SQL	
BigQuery	
BigTable	
Memory Store	
BigTable	
BigTable	
BigQuery	
CloudSpanner	
Hyper Ledger Fabric	
PostGIS	
geomesa	
OrientDB	
Dgraph	
MongoDB	
Couchbase	
Elasticsearch	
Elassandra	
Ceph	
OpenIO	
Open Source / 3rd Party	
Oracle	
PostgreSQL	
MySQL	
SQL Server	
Snowflake	
Click House	
Redis	
Scylla	
Redis	
Memcached	
Cassandra	
Scylla	
Influx	
OpenTSDB	
Hyper Ledger Fabric	
PostGIS	
geomesa	
OrientDB	
Dgraph	
MongoDB	
Couchbase	
Elasticsearch	
Elassandra	
Ceph	
OpenIO	

为你的项目选择正确的数据库是一项复杂的任务。许多数据库选项都有各自的适用场景，互不重复，这很容易导致决策疲劳。

我们希望这份速查表能够提供高层次的指导，以帮你找到符合项目需求的正确服务，并避免潜在陷阱。

注意：Google 关于其数据库用例的文档有限。尽管我们尽力查看可用的内容并得出最佳选择，但某些条目可能需要更加准确。

8 种支撑数据库的数据结构

根据使用场景不同，答案也不同。数据可以在内存或磁盘上建立索引。同样，数据格式也各不相同，例如数字、字符串、地理坐标等。系统可能是写入密集型的，也可能是读取密集型的。所有这些因素都会影响您对数据库索引格式的选择。

8 Data Structures That Power Your Databases



Types	Illustration	Use Case	Note
Skiplist		In-memory	used in Redis
Hash index		In-memory	Most common in-memory index solution
SSTable		Disk-based	Immutable data structure. Seldom used alone
LSM tree		Memory + Disk	High write throughput. Disk compaction may impact performance
B-tree		Disk-based	Most popular database index implementation
Inverted index		Search document	Used in document search engine such as Lucene
Suffix tree		Search string	Used in string search, such as string suffix match
R-tree		Search multi-dimension shape	Such as the nearest neighbor

下面是一些用于索引数据的最流行的数据结构：

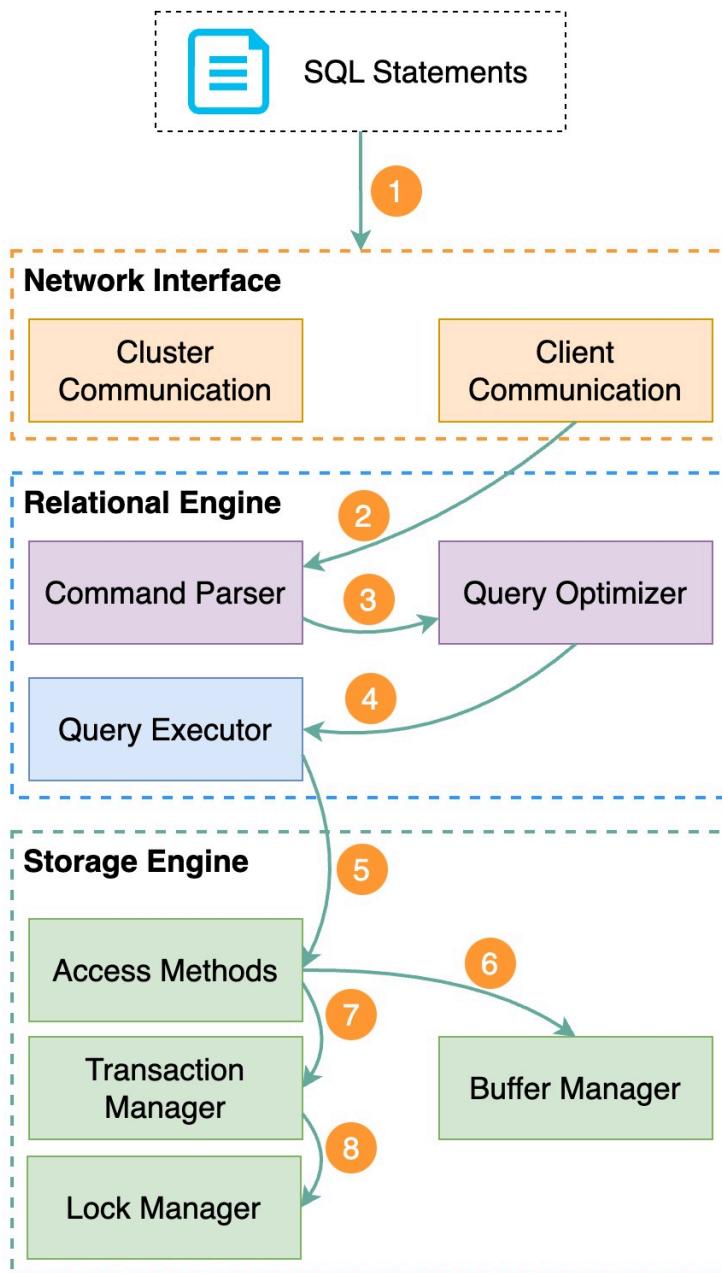
- 跳跃表 (SkipList) : 常见的内存索引类型。在 Redis 中使用
- 哈希索引 (Hash index) : “Map” 数据结构（或者“Collection”）的一个非常常见的实现
- SSTable: 不可变的磁盘“Map”实现
- LSM 树: SkipList + SSTable。高写入吞吐量
- B 树 (B-tree) : 基于磁盘的方案。一致读写性能
- 倒排索引 (Inverted index) : 用于文档索引。在 Lucene 中使用
- 后缀树 (Suffix tree) : 用于字符串模式搜索
- R 树 (R-tree) 多维搜索，例如寻找最近邻

一条 SQL 语句是如何在数据库中执行的？

下图显示了 SQL 语句的执行过程。注意，不同数据库的架构不一样，下图演示的是一些常见设计。

How is SQL Executed in DB?

 blog.bytebytego.com



步骤 1 - 通过传输层协议（例如，TCP）将 SQL 语句发送到数据库。

步骤 2 - 发送 SQL 语句到命令解析器，然后命令解析器对其进行语法和语义分析，生成查询树。

步骤 3 - 发送查询树到优化器。优化器创建执行计划。

步骤 4 - 发送执行计划到执行器。执行器根据执行检索数据。

步骤 5 - 访问方法提供执行所需的数据获取逻辑，从存储引擎检索数据。

步骤 6 - 访问方法决定 SQL 语句是否只读。如果是只读查询（SELECT 语句），就会将其传到缓冲区管理器以进行进一步处理。缓冲区管理器在缓存或者数据文件中查询数据。

步骤 7 - 如果是 UPDATE 或者 INSERT 语句，则将其传到事务管理器进行进一步处理。

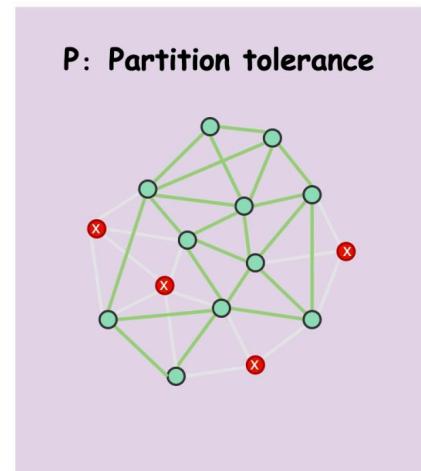
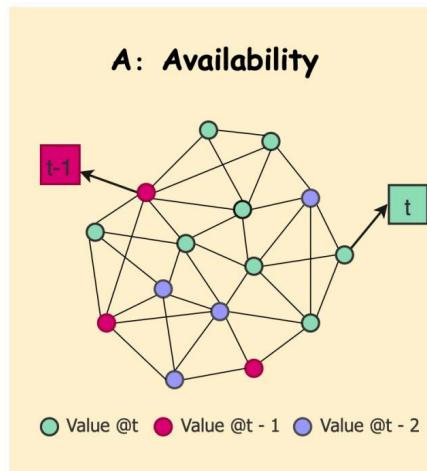
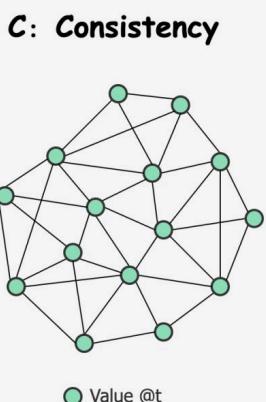
步骤 8 - 事务期间，数据处于锁定模式。这由锁管理器保证。它还确保事务的 ACID 属性。

CAP 理论

CAP 理论是计算机科学中最著名的术语之一，但我打赌，不同的开发者对其有不同的理解。让我们来看看它是什么，以及为什么它会令人困惑。

CAP Theorem

 blog.bytebytogo.com



	CA systems	CP systems	AP systems
Sacrifice	No sacrifice	Availability	Consistency
Use Cases	Single-node only	Strong consistency. Banks, financial systems	Low latency. Consistency requirement is not high
Real-world examples	Single node RBMS (MySQL, Oracle)	Zookeeper, BigTable	Cassandra, CouchDB

CAP 理论指出，分布式系统不能同时提供这三个保证中的两个以上的保证。

一致性 (Consistency) : 一致性意味着，不管客户端连接到哪个节点，它们都同时看到相同的数据。

可用性 (Availability) : 可用性意味着，即使某些节点发生了故障，任何客户端请求都能得到响应。

分区容错性 (Partition Tolerance) : 分区表示两个节点之间的通信发生了中断。分区容错性意味着，尽管存在网络分区，系统仍能继续运行。

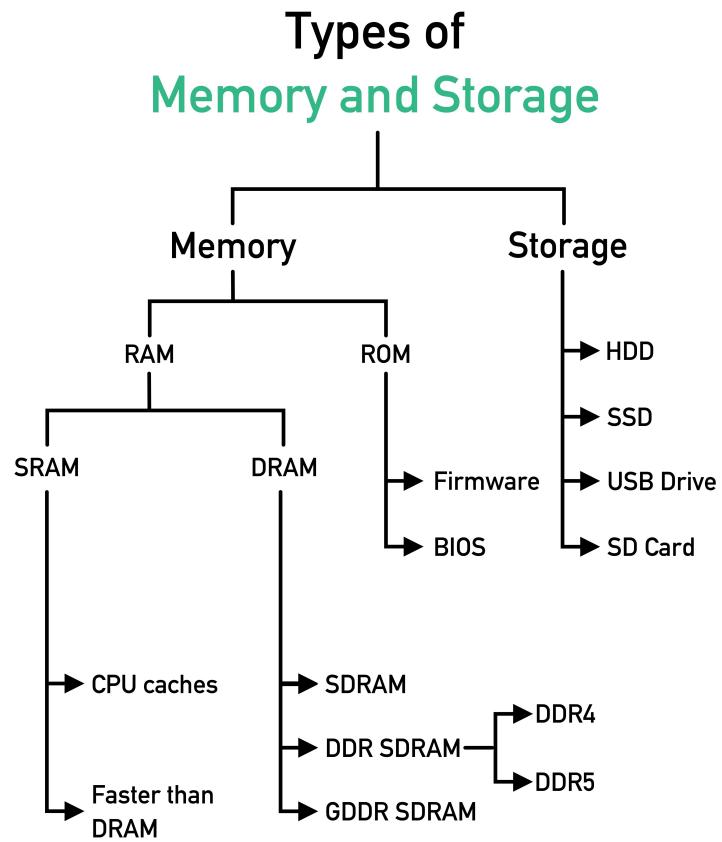
“三分之二”表述可能有用，但这种简化可能会产生误导。

1. 选择合适的数据库不容易。仅仅根据 CAP 定理来证明我们的选择并不够。例如，公司不会只因为 Cassandra 是一个 AP 系统就选择它作为聊天应用。Cassandra 具有一系列不错的特性，使其成为存储聊天消息的理想选择。我们需要更深入地挖掘。
2. “CAP 仅限制了一小部分的设计空间：在存在分区的情况下实现完美的可用性和一致性是很少见的”。引自论文：十二年后的 CAP：“规则”是如何改变的 (CAP Twelve Years Later: How the “Rules” Have Changed) 。
3. 该理论是关于百分百可用性和一致性的。更现实的讨论是在没有网络分区的情况下，延迟和一致性之间的权衡。详情请参阅 PACELC 定理。

CAP 理论真的有用吗？

我认为它依然有用，因为它让我们能够进行一系列权衡讨论，但这只是故事的一部分。在选择正确的数据库时，我们需要更深入挖掘。

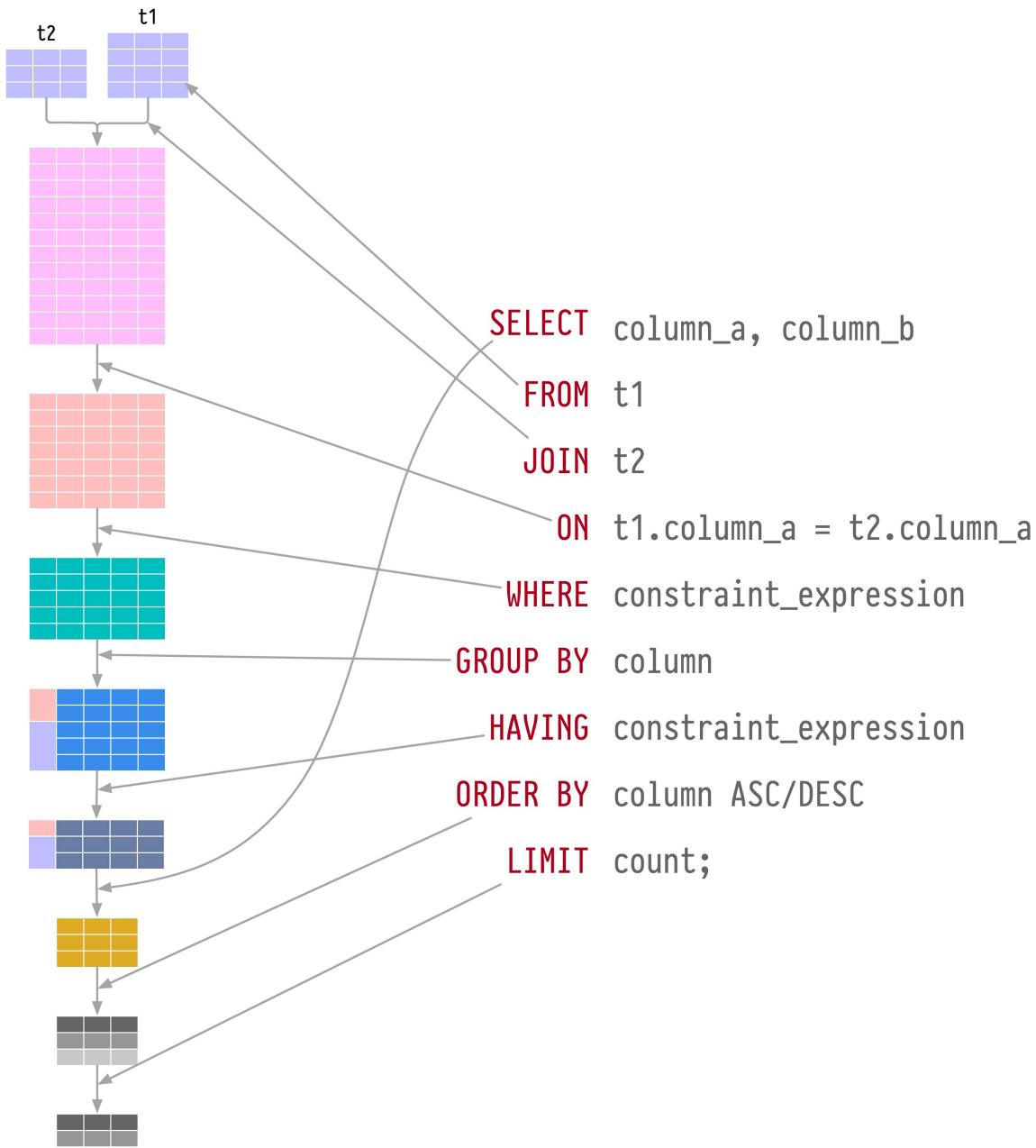
内存和存储的类型



可视化 SQL 查询

SQL Query Execution Order

 ByteByteGo.com



SQL 语句由数据系统执行，包含以下几个执行步骤：

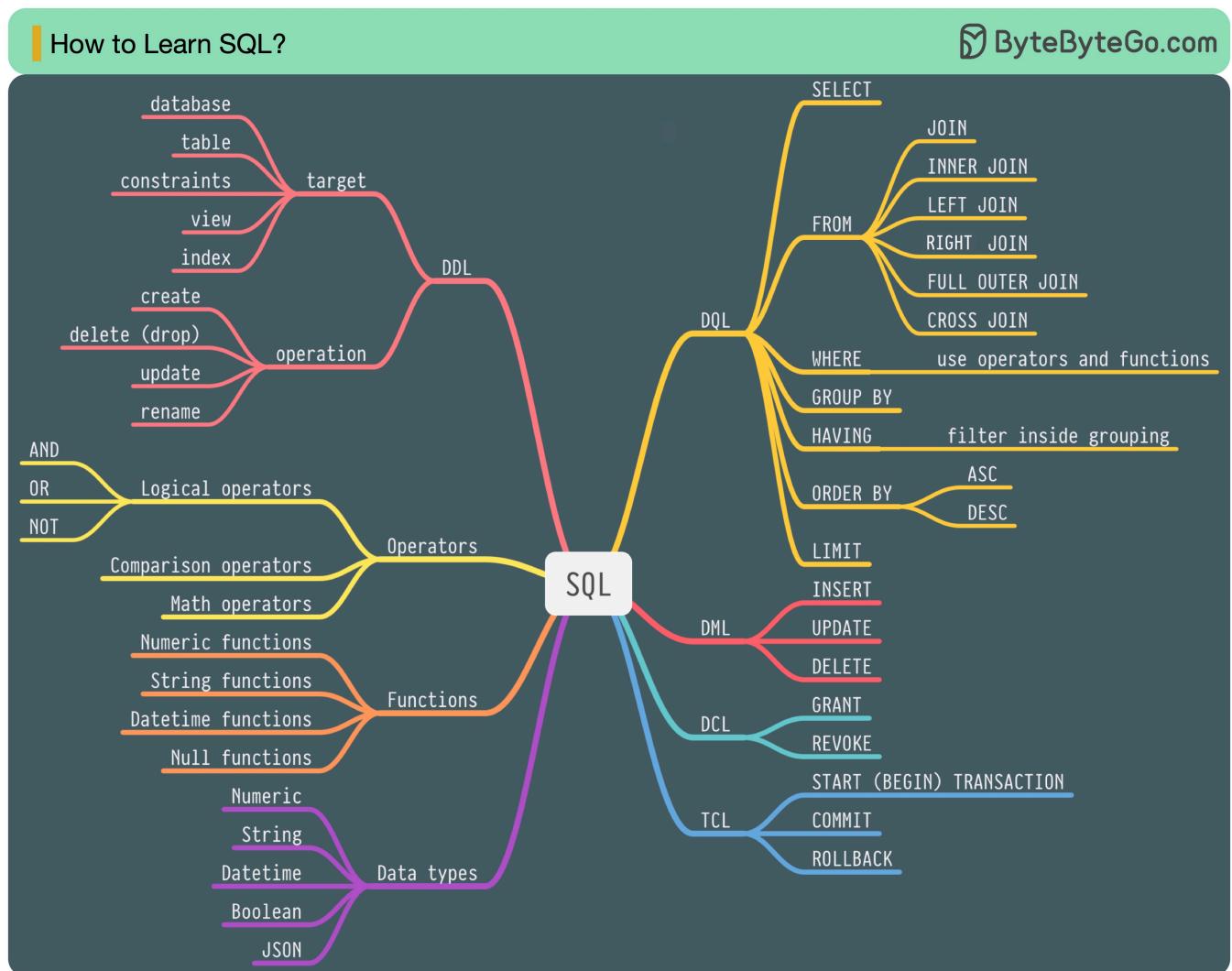
- 解析 SQL 语句并检查其有效性
- 将 SQL 语句转化为内部表达形式，例如，关系代数
- 优化内部表达形式，并创建利用索引信息的执行计划
- 执行计划并返回结果

SQL 的执行是非常复杂的，涉及很多考虑因素，例如：

- 索引和缓存的使用
- 表连接的顺序
- 并发控制
- 事务管理

SQL 语言

1986 年，SQL (Structured Query Language, 结构化查询语言) 成为标准。在接下来的 40 年间，它成为了关系数据库管理系统的主导语言。阅读最新标准 (ANSI SQL 2016) 可能会耗费你大量的时间。那么，我要怎样才能学习它呢？



SQL 语言有五个组件：

- DDL: 数据定义语言, 例如 CREATE、ALTER、DROP
- DQL: 数据查询语言, , 例如 SELECT
- DML: 数据操作语言, 例如 INSERT、UPDATE、DELETE
- DCL: 数据控制语言, 例如 GRANT、REVOKE
- TCL: 事务控制语言, 例如 COMMIT、ROLLBACK

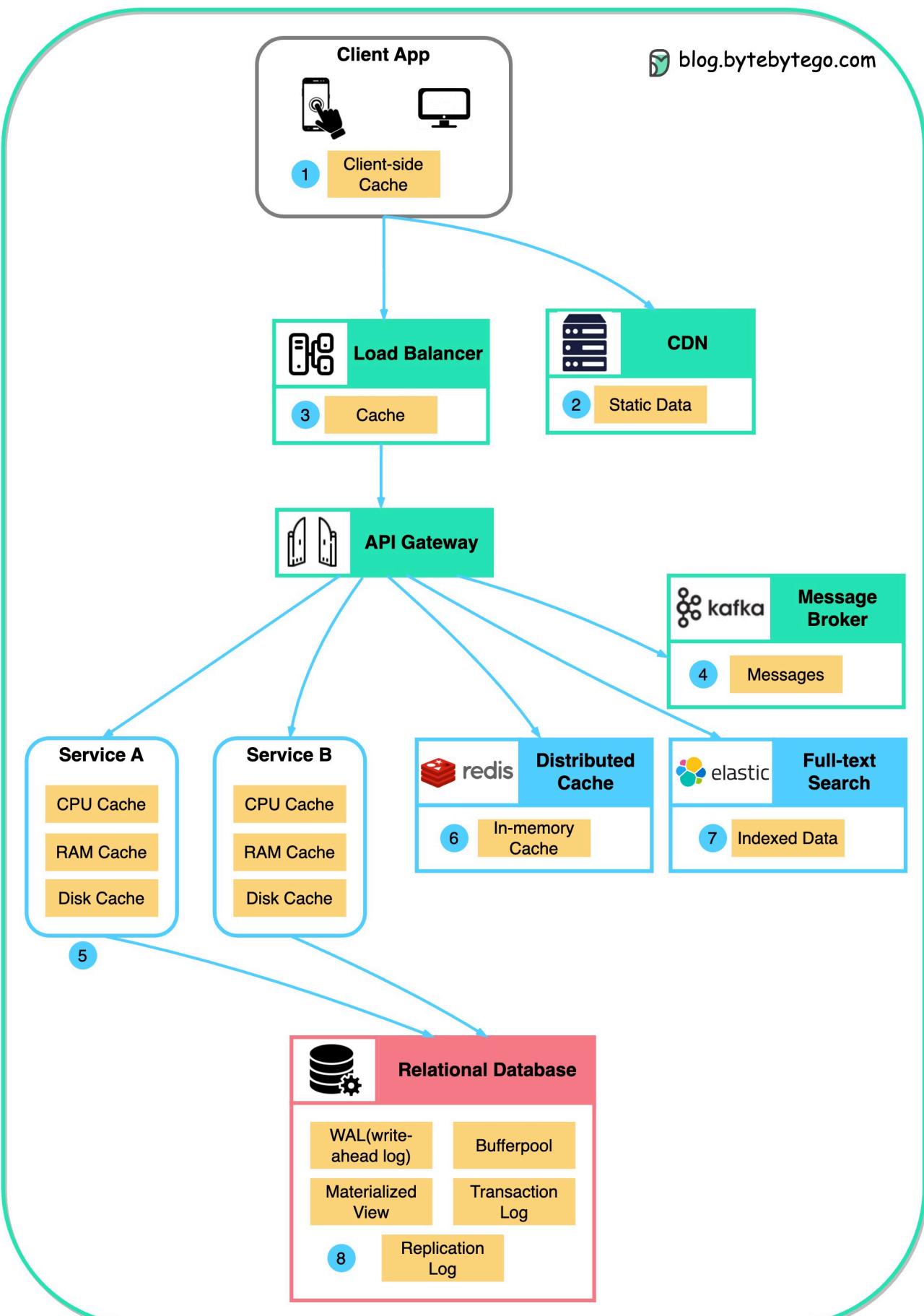
对于一个后端工程师来说, 你可能需要了解上面大多数组件。而作为一名数据分析师, 你可能需要很好地了解 DQL。根据需求, 选择与自己关系最大的主题吧。

缓存

缓存无处不在

下图说明了在典型架构中, 我们缓存数据的位置。

Cache Systems Every Developer Should Know



在这个架构中，有多个层。

1. 客户端应用：HTTP 响应可以由浏览器进行缓存。我们第一次通过 HTTP 请求数据，数据返回时在 HTTP 标头中包含过期策略；当我们再次请求数据时，客户端应用会先尝试从浏览器缓存中检索数据。
2. CDN：CDN 缓存静态网络资源。客户端可以就近从 CDN 节点检索数据。
3. 负载均衡器：负载均衡器也可以缓存资源。
4. 消息传递基础设施（Messaging infra）：消息代理首先将消息存储在磁盘上，然后消费者按需检索消息。根据保留策略（retention policy），数据会在 Kafka 集群中缓存一段时间。
5. 服务：服务存在多层缓存。如果数据没有缓存在 CPU 缓存中，那么服务会尝试从内存中检索数据。有时，服务会有二级缓存，来将数据存储到磁盘中。
6. 分布式缓存：像 Redis 这样的分布式缓存在内存中保存多个服务的键值对。和数据库相比，它能提供更好的读/写性能。
7. 全文搜索：有时，我们需要使用像 Elastic Search 这样的全文搜索来对文档或日志进行搜索。因此，搜索引擎中也会对数据副本进行索引。
8. 数据库：即使在数据库中，我们也有不同级别的缓存：
 - WAL(Write-ahead Log)：在构建 B 树索引前，先将数据写入 WAL
 - Bufferpool：一块内存区域，分配于缓存查询结果
 - 物化视图（Materialized View）：预计算查询结果并将其存储在数据库表中，以获得更好的查询性能
 - 事务日志：记录所有事务和数据库更新
 - 复制日志：用于记录数据库集群中的复制状态

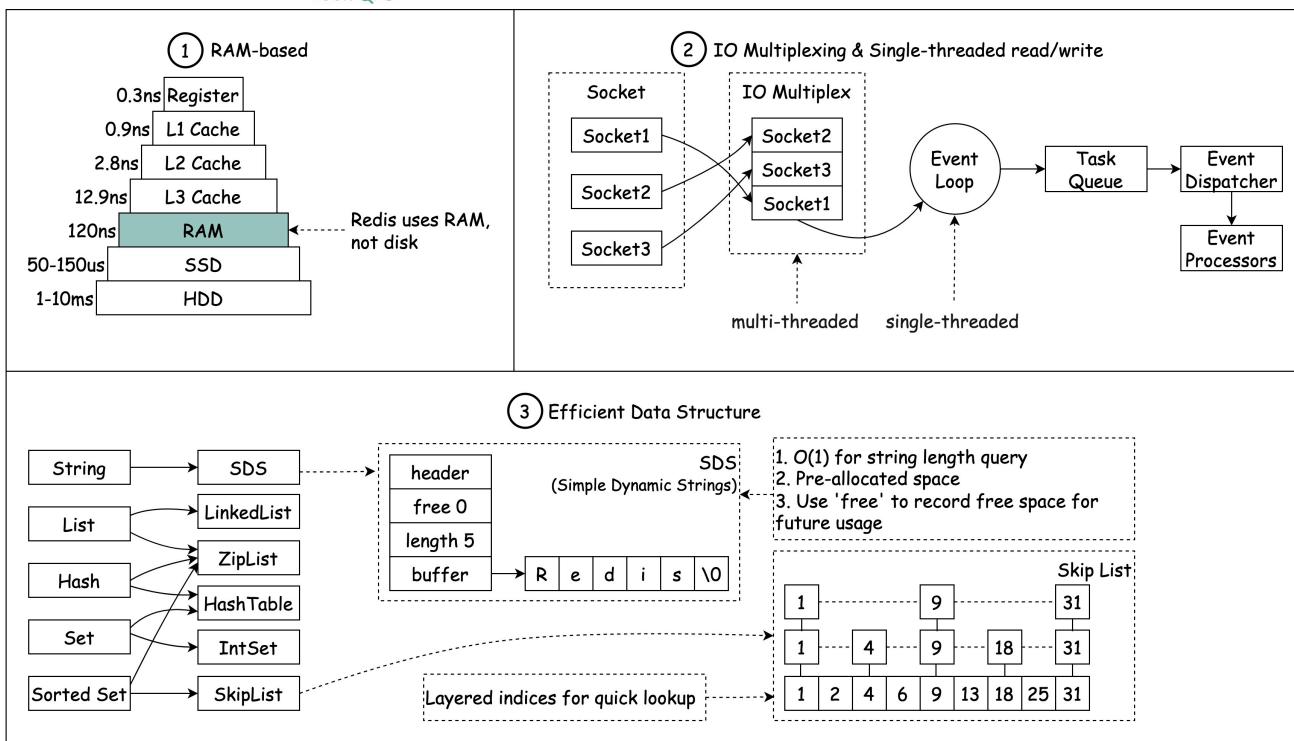
为什么 Redis 那么快？

主要有三个原因，如下图所示。

Why is Redis so fast?



100k QPS



1. Redis 是一个基于 RAM 的数据存储。RAM 访问至少比随机磁盘访问快 1000 倍。
2. Redis 利用 IO 多路复用和单线程执行循环来提高执行效率。
3. Redis 利用多种高效的底层数据结构。

问题：另一种流行的内存存储是 Memcached。你知道Redis和Memcached的区别吗？

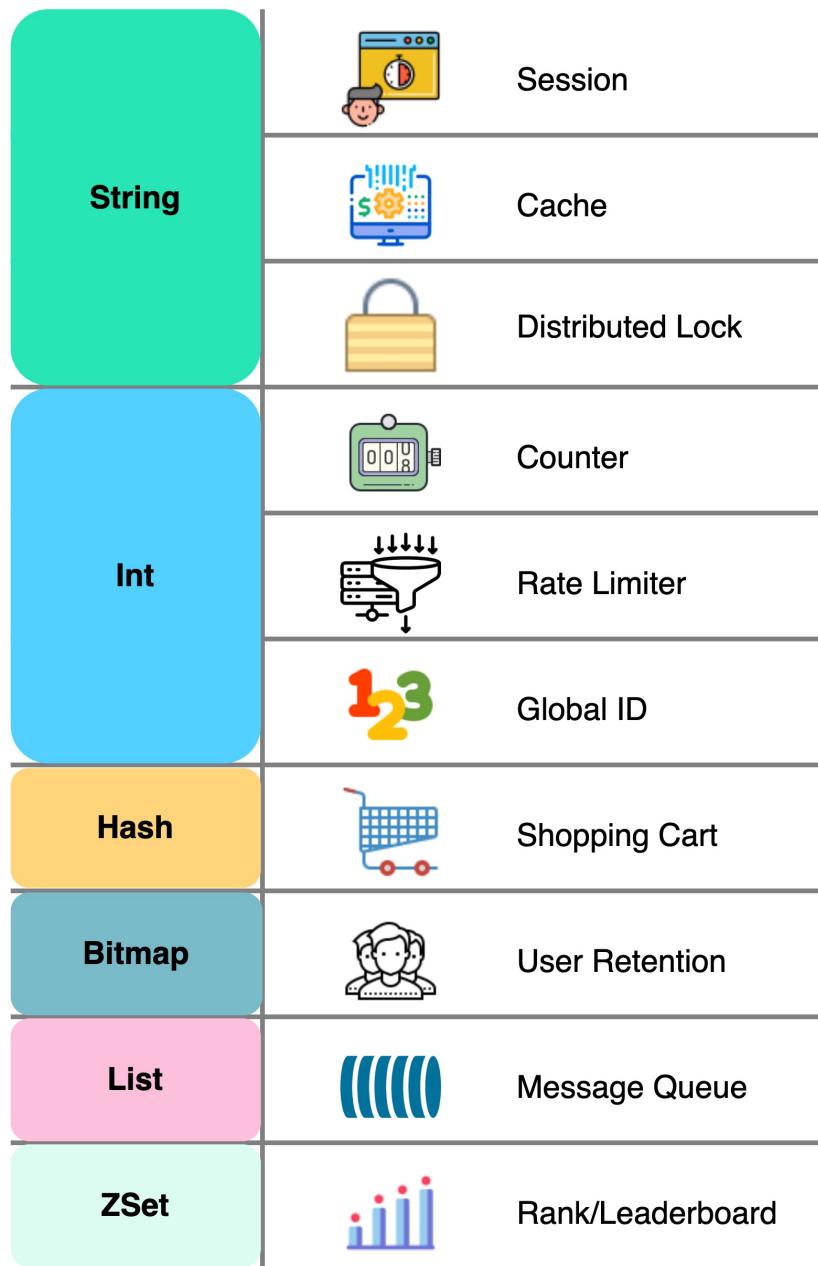
译者注：答案可以参考 [redis和memcached的区别和使用场景](#)

您可能已经注意到该图的风格与我之前的帖子不同。请告诉我您更喜欢哪一个。

Redis 的使用场景

Top Redis Use Cases

 blog.bytebytogo.com



Redis 不仅仅是缓存。

Redis 可用于多种场景，如图所示。

- 会话 (Session)

我们可以使用 Redis 在不同服务之间共享用户会话数据。

- 缓存 (Cache)

我们可以使用 Redis 来缓存对象或页面，尤其是热点数据。

- 分布式锁

我们可以使用 Redis 字符串来获取分布式服务之间的锁。

- 计数器 (Counter)

我们可以用它来统计文章的点赞数或阅读量。

- 速率限制器 (Rate limiter)

我们可以对某些用户 IP 应用速率限制器。

- 全局 ID 生成器 (Global ID generator)

我们可以使用 Redis Int 作为全局 ID。

- 购物车 (Shopping cart)

我们可以使用 Redis Hash 来表示购物车中的键值对。

- 计算用户留存率

我们可以使用 Bitmap 来表示用户每天的登录情况并计算用户留存情况。

- 消息队列

我们可以使用 List 作为消息队列。

- 排行 (Ranking)

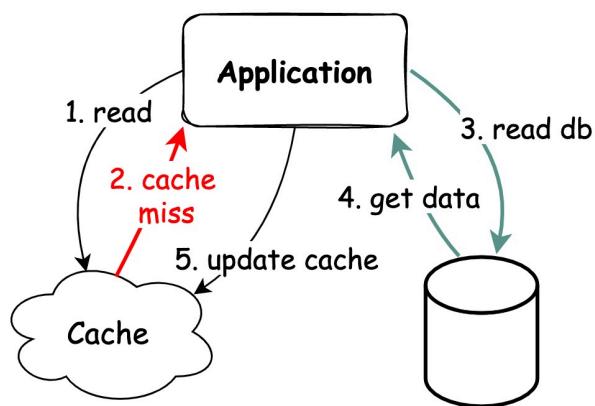
我们可以使用 ZSet 对文章进行排序。

最佳缓存策略

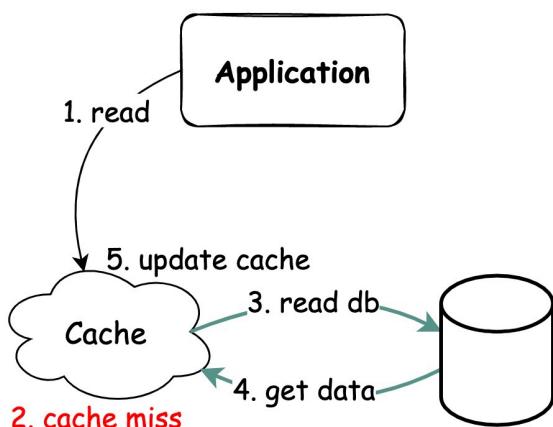
设计大型系统通常需要仔细地考虑缓存。下面是五种常用的缓存策略。

Top caching strategies

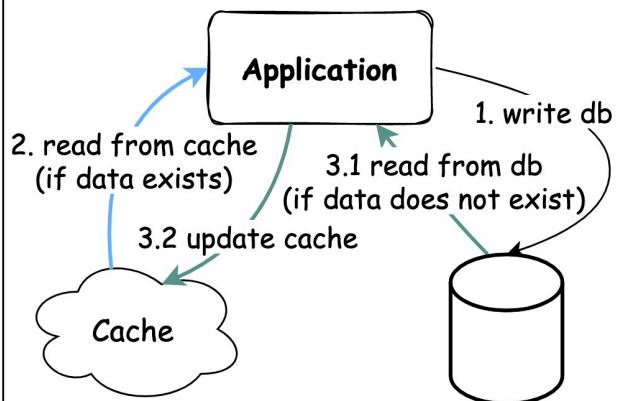
Read Strategy - Cache Aside



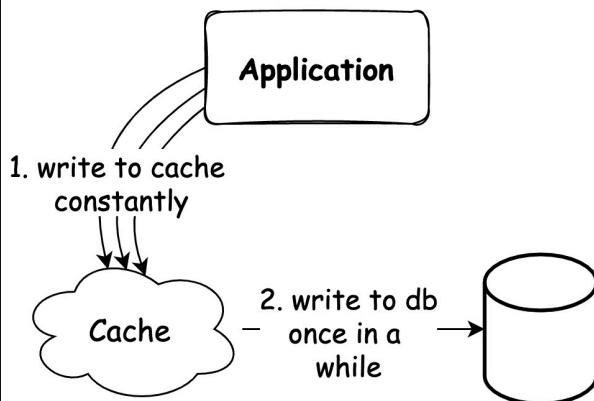
Read Strategy - Read Through



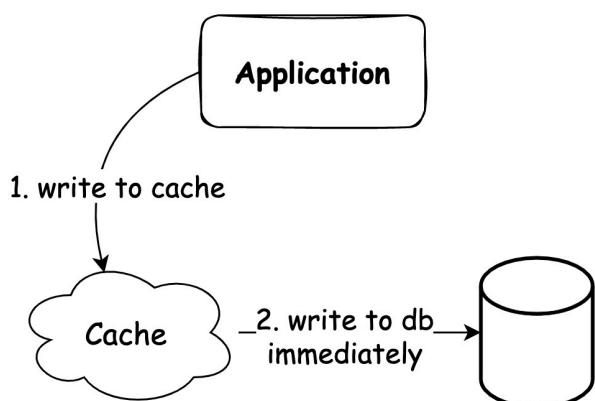
Write Strategy - Write Around



Write Strategy - Write Back

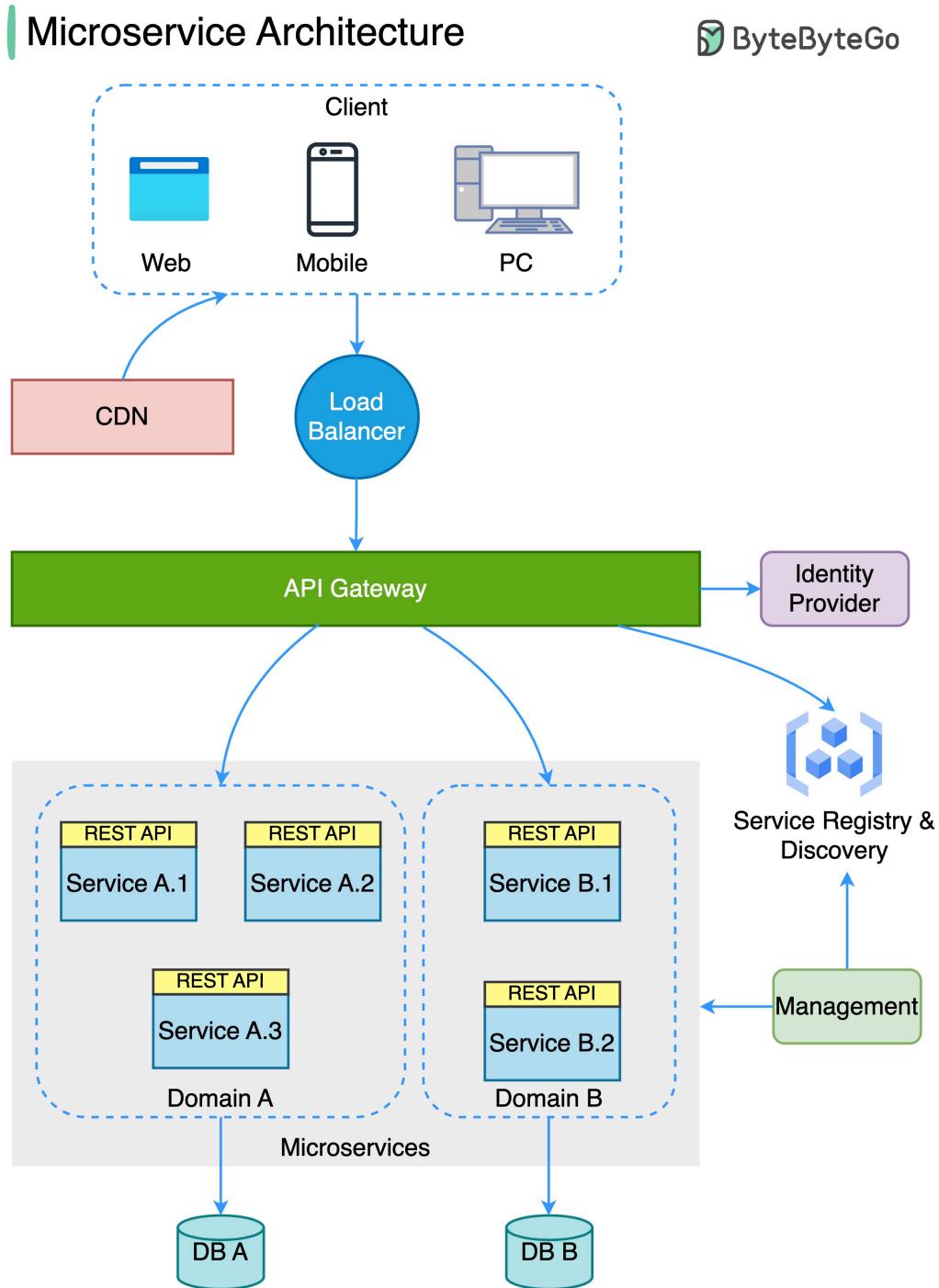


Write Strategy - Write Through



微服务架构

典型的微服务架构长啥样？



上图显示了一个典型的微服务架构。

- 负载均衡器 (Load Balancer) : 它将传入的流量分发到多个后端服务。
- CDN (Content Delivery Network, 内容分发网络): CDN 是一组地理分布的服务器，用于保存静态内容以实现更快的传输。客户端首先在 CDN 查找内容，然后再进行后端服务。
- API 网关 (API Gateway) : 它处理进来的请求并将其路由到相关服务。它与身份提供者和服务发现进行通信。
- 身份提供者 (Identity Provider) : 负责处理用户的身份验证和授权。
- 服务注册和发现 (Service Registry & Discovery) : 该组件负责微服务注册和发现，此外，API 网关从此组件查找相关服务进行通信。
- 管理 (Management) : 该组件负责监控服务。
- 微服务：根据域不同来设计和部署微服务。每个域都有其自身的数据库。API 网关通过 REST API 或者其他协议与微服务进行通信，同一个域中的微服务使用 RPC (远程过程调用) 相互通信。

微服务的优点：

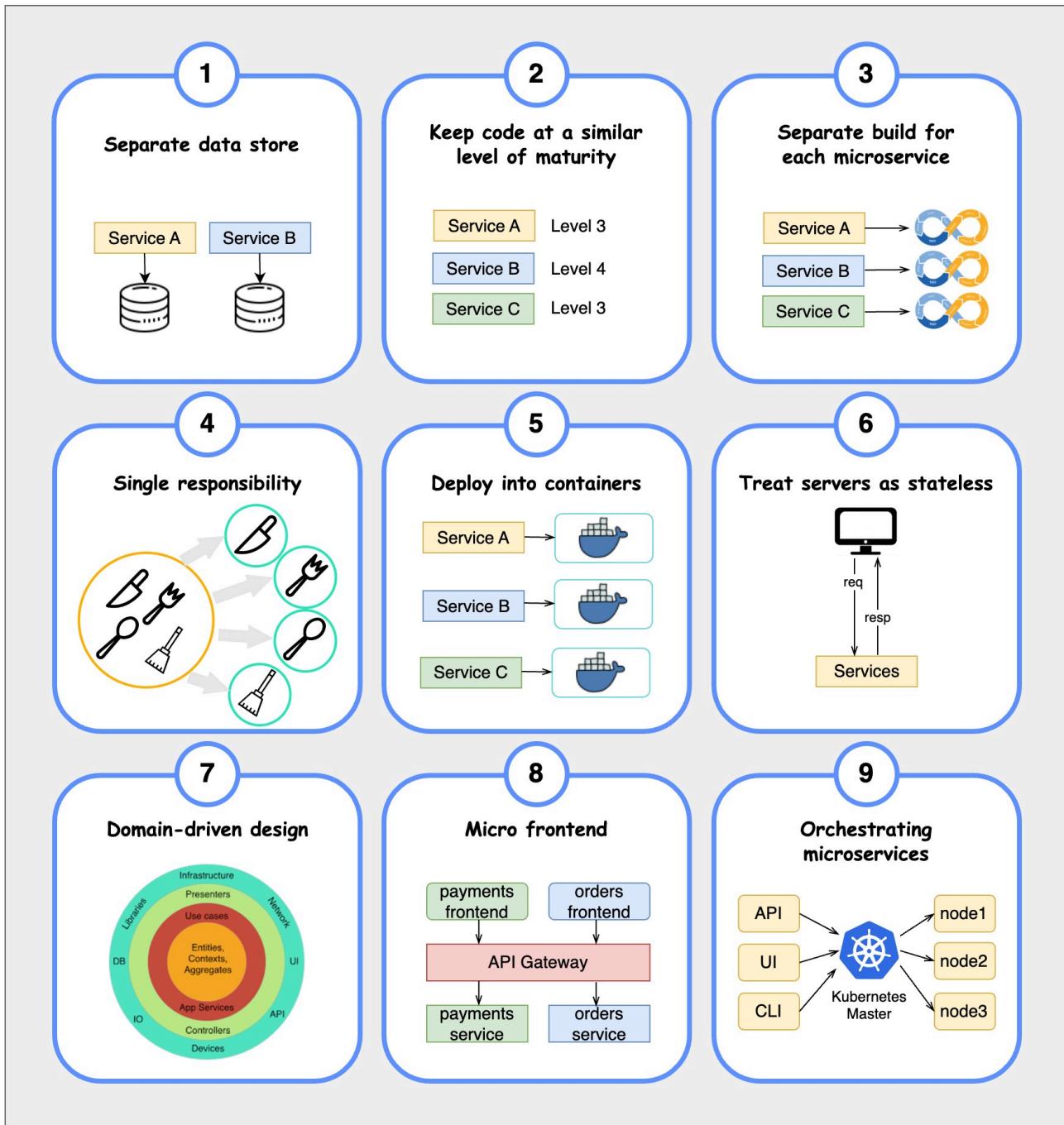
- 可以对其进行快速设计、部署和水平扩展。
- 每个域都可以由专门的团队独立维护。
- 可以在每个域对业务需求进行定制，从而获得更好的支持。

微服务最佳实践

一图胜千言：开发微服务的 9 个最佳实践。

Microservice Best Practices

 blog.bytebytego.com



开发微服务时，我们需要遵循以下最佳实践：

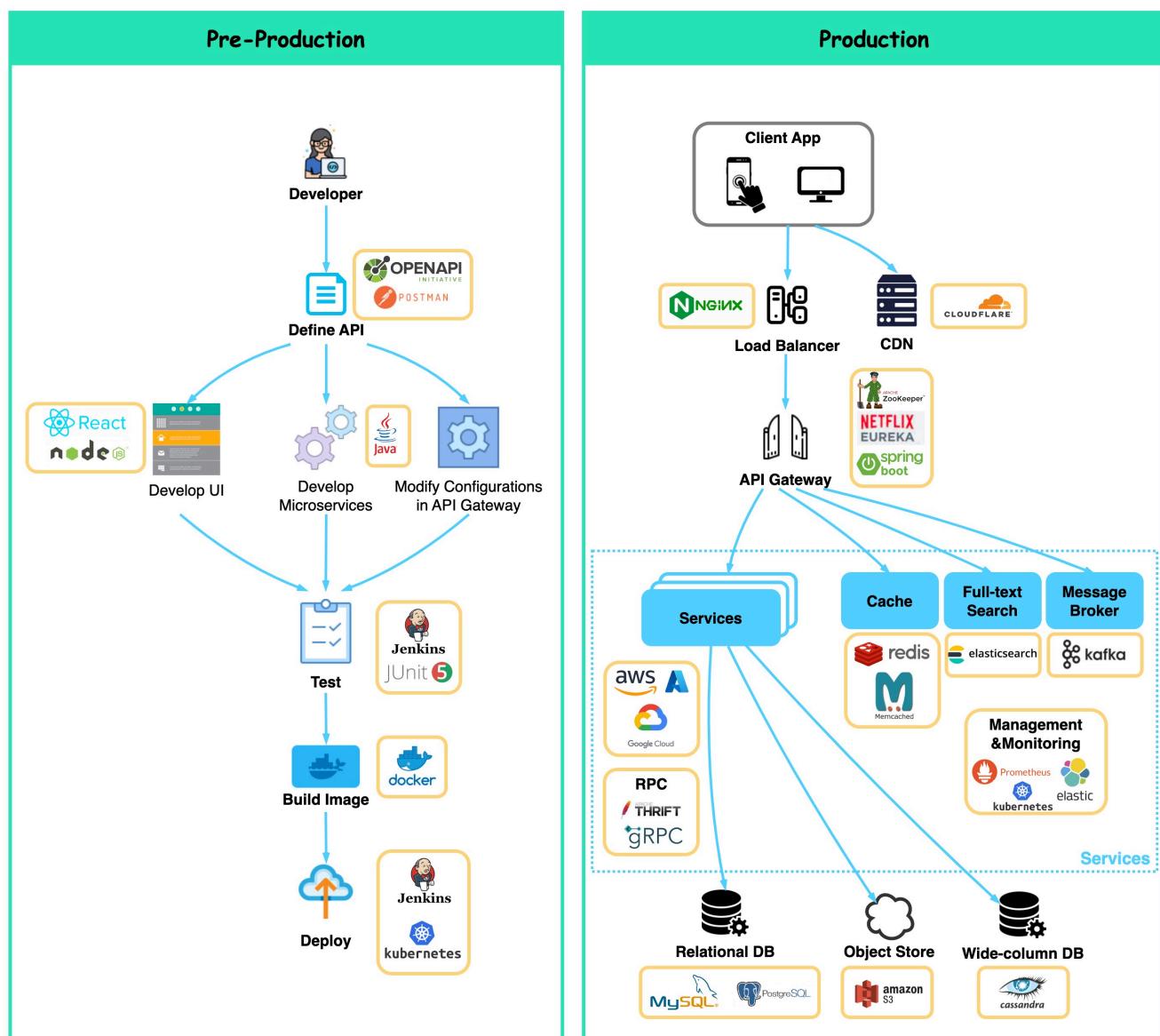
1. 为每个微服务使用单独的数据存储
2. 保持代码处于相似的成熟度
3. 单独构建每个微服务

4. 为每个微服务分配单一职责
5. 部署到容器中
6. 设计无状态服务
7. 采用领域驱动设计
8. 设计微前端
9. 编排微服务

微服务通常使用哪些技术栈？

下面，你将看到一张显示微服务技术栈的图标，同时涉及开发阶段和生产阶段。

Microservice Tech Stack blog.bytebytogo.com



► 预生产 (◆◆◆-◆◆◆◆◆◆◆◆)

- 定义 API - 这为前后端之间建立契约。为此，我们可以使用 Postman 或者 OpenAPI。
- 开发 - Node.js 或 react 在前端开发中很是流行，而 java/python/go 则是后端开发的流行之选。此外，我们需要根据 API 定义更改 API 网关的配置。
- 持续集成 - JUnit 和 Jenkins 用于自动化测试。打包代码到 Docker 镜像并部署为微服务。

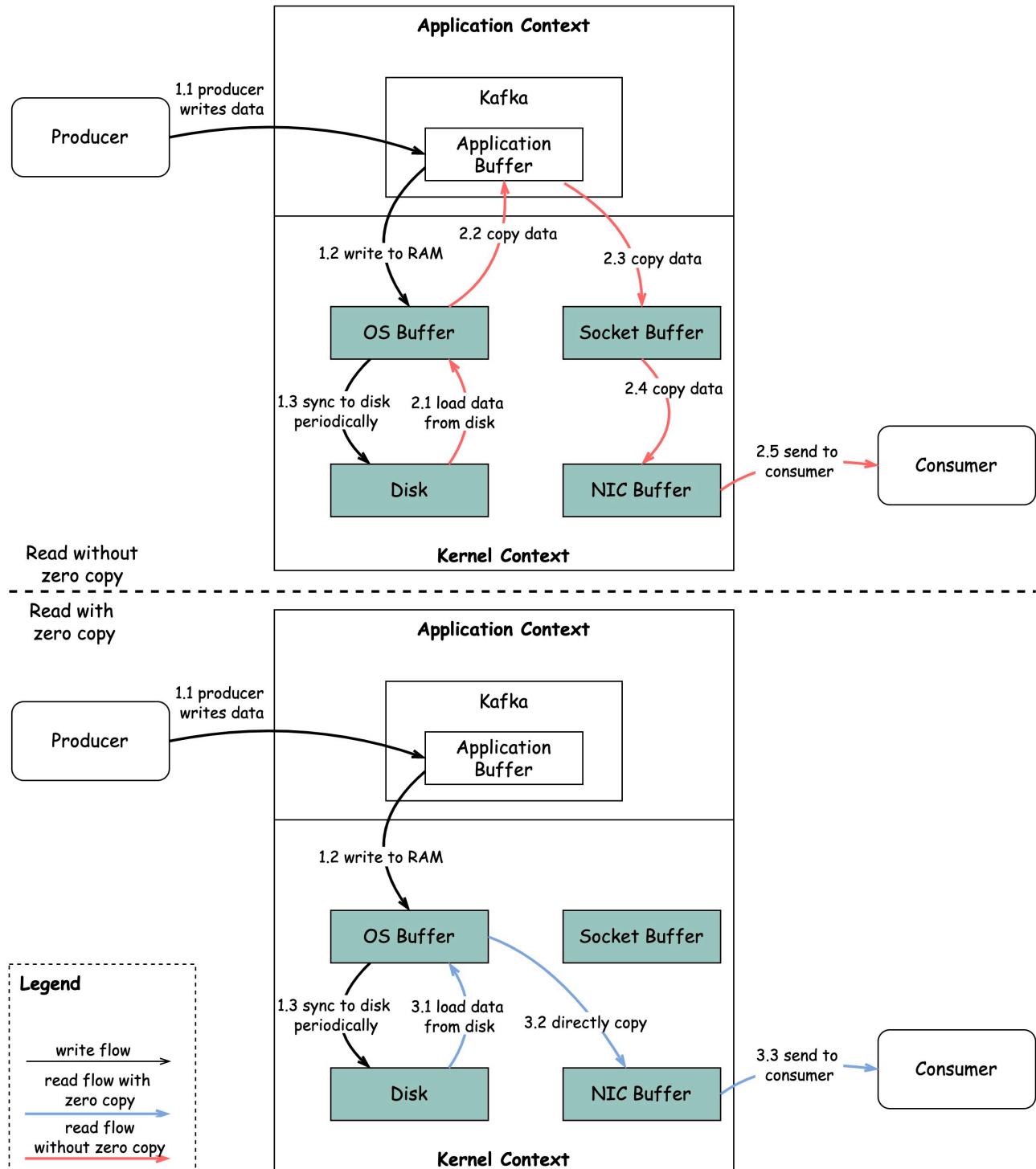
► 生产 (◆◆◆◆◆◆◆◆◆◆)

- NGinx 是负载均衡的常见选择。Cloudflare 则提供 CDN (Content Delivery Network)功能。
- API 网关 - 我们可以使用 spring boot 作为网关，并 Eureka/Zookeeper 进行服务发现。
- 在云上部署微服务。可以选择 AWS、Microsoft Azure 或者 Google GCP。缓存和全文搜索 —— 通常选择 Redis 来缓存键值对。使用 Elasticsearch 进全文搜索。
- 通信 - 为了使服务之间能够相互通信，我们可以使用 Kafka 或者 RPC。
- 持久化 - 可以使用 MySQL 或者 PostgreSQL 作为关系数据库，使用 Amazon S3 作为对象存储。如有必要，我们还可以使用 Cassandra 进行宽列存储。
- 管理和监控 - 为了管理如此多的微服务，常见的运维攻击包括 Prometheus、Elastic Stack 和 Kubernetes。

为什么 Kafka 快？

有许多设计决策为 Kafka 的性能做出了贡献。在这里，我们将关注其中两个设计决策。我们认为这两者最有分量。

Why is Kafka Fast?



1. 第一个是 Kafka 对顺序 I/O 的依赖。
2. 赋予 Kafka 性能优势的第二个设计决策是它对于效率的关注：零拷贝原则（zero copy principle）。

上图说明了数据是如何在生产者和消费者之间传输的，以及零拷贝的含义。

- 步骤 1.1 - 1.3: 生产者将数据写到磁盘中
- 步骤 2: 消费者无需零拷贝即可读取数据。
 - 2.1 从磁盘中将数据加载到操作系统缓存
 - 2.2 从操作系统缓存拷贝数据到 Kafka 应用
 - 2.3 Kafka 应用拷贝数据到 socket 缓存
 - 2.4 从 socket 缓存拷贝数据到网卡
 - 2.5 网卡发送数据给消费者
- 步骤 3: 消费者以零拷贝方式读取数据
 - 3.1 从磁盘加载数据到操作系统缓存
 - 3.2 操作系统缓存通过 `sendfile()` 命令，直接将数据拷贝到网卡
 - 3.3 网卡发送数据给消费者

零拷贝（Zero copy）是一种在应用程序上下文和内核上下文之间保存多个数据副本的快捷方式。

支付系统

如何学习支付系统？

How to Learn Payments?

 blog.bytebytego.com



 blog.bytebytego.com

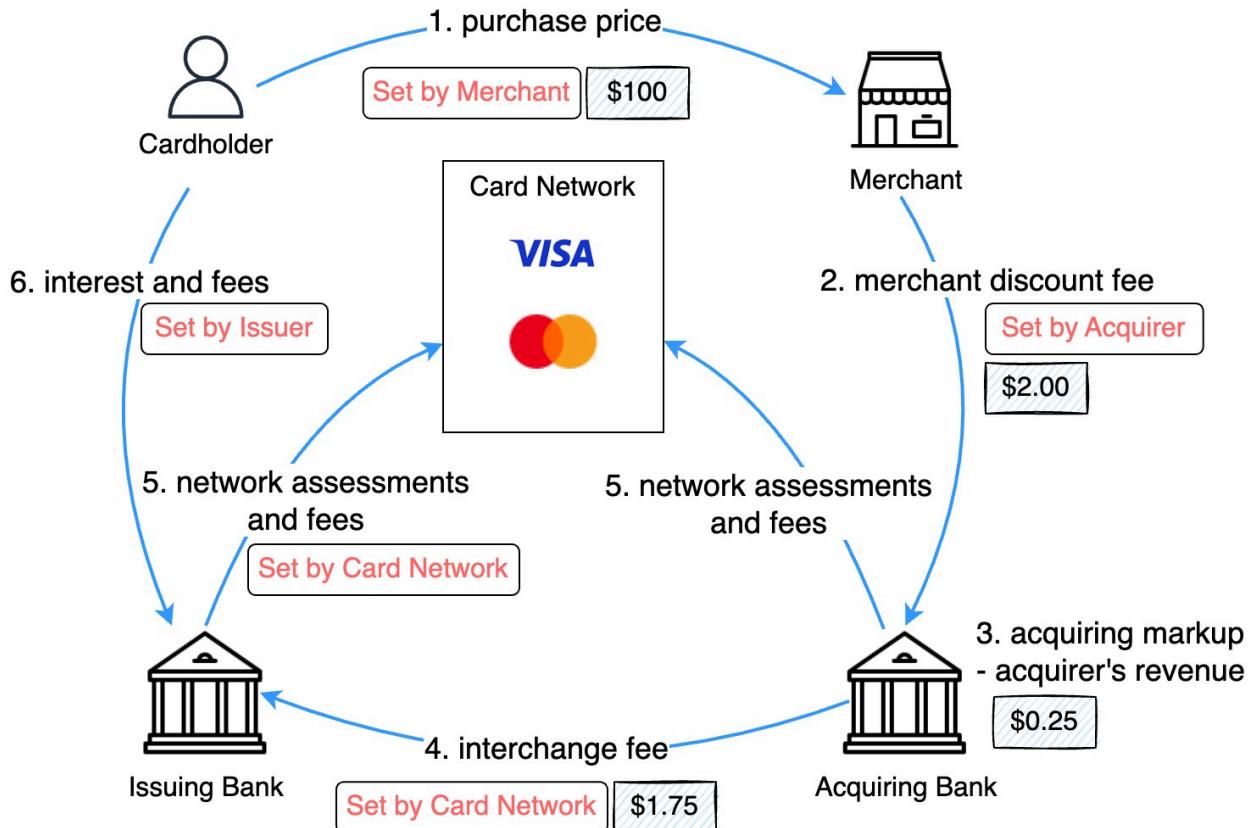
信用卡为何被称为“银行最赚钱的产品”？VISA/万事达卡是如何赚钱的？

下图显示了信用卡支付流程背后的经济学。

How does VISA Make Money?



blog.bytebytogo.com



$$\text{merchant discount fee} = \text{interchange fee} + \text{acquiring markup}$$

\$2.00

\$1.75

\$0.25

The merchant needs to compensate issuer and acquirer

1. 持卡人（cardholder）向商家（merchant）支付 100 美元以购买产品。
2. 商家从使用量较高的信用卡中获益，并且需要为发卡机构和卡网络（card network）提供的支付服务进行补偿。收单银行（acquiring bank）向商家收取一定费用，称为“商家折扣费”。

3 - 4. 收单银行保留 0.25 美元作为收单加价，并向发卡行 (issuing bank) 支付 1.75 美元作为互换费。商户折扣费应该覆盖互换费用。

互换费用由卡网络设定，因为每个商户与每个发卡行就费用进行谈判效率较低。

1. 卡网络与各银行建立网络评估和费用，银行每月支付卡网络服务费。例如，VISA 对每次刷卡收取 0.11% 的评估费，另加 0.0195 美元的使用费。
2. 持卡人向发卡行支付其服务费。

发卡行为何需要获得补偿呢？

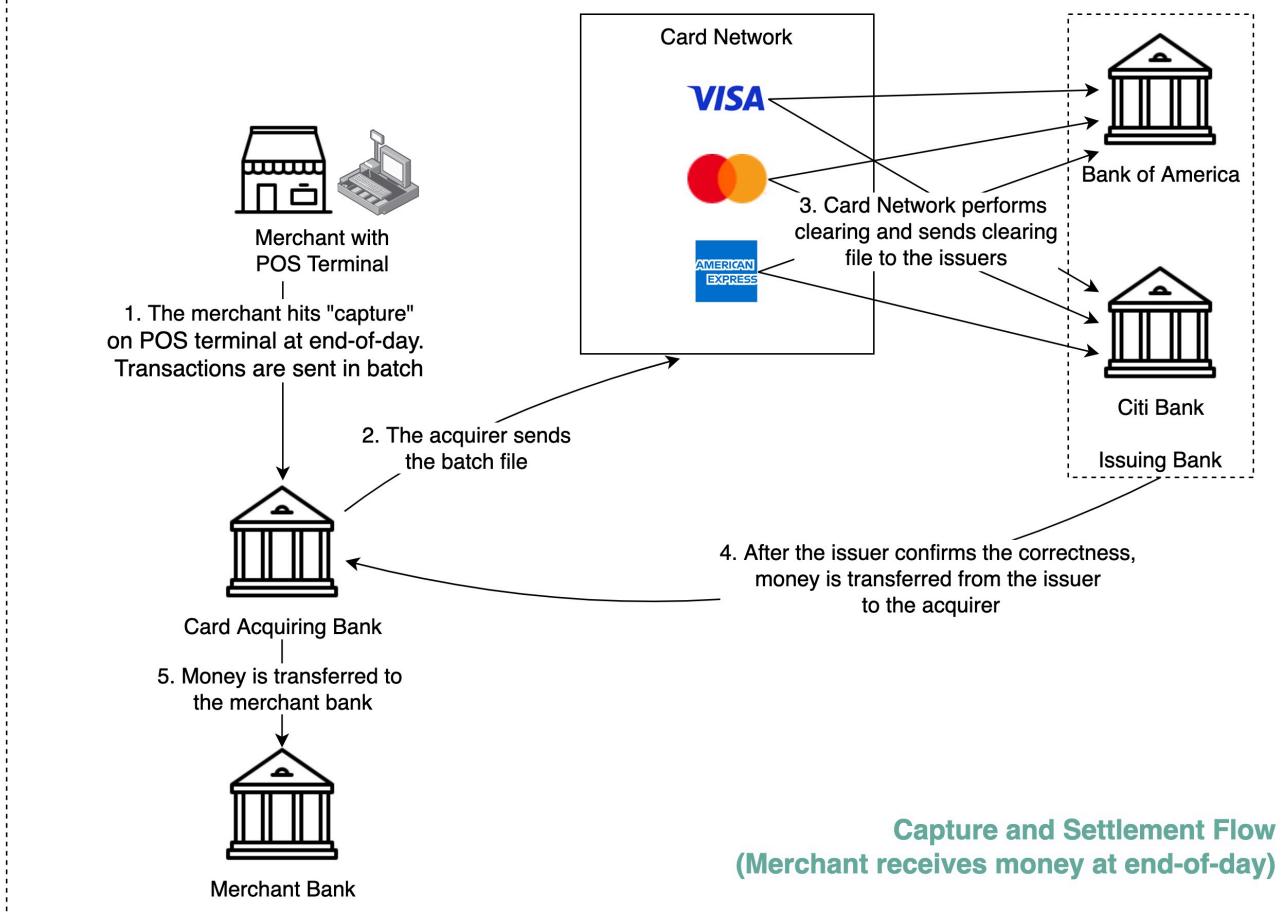
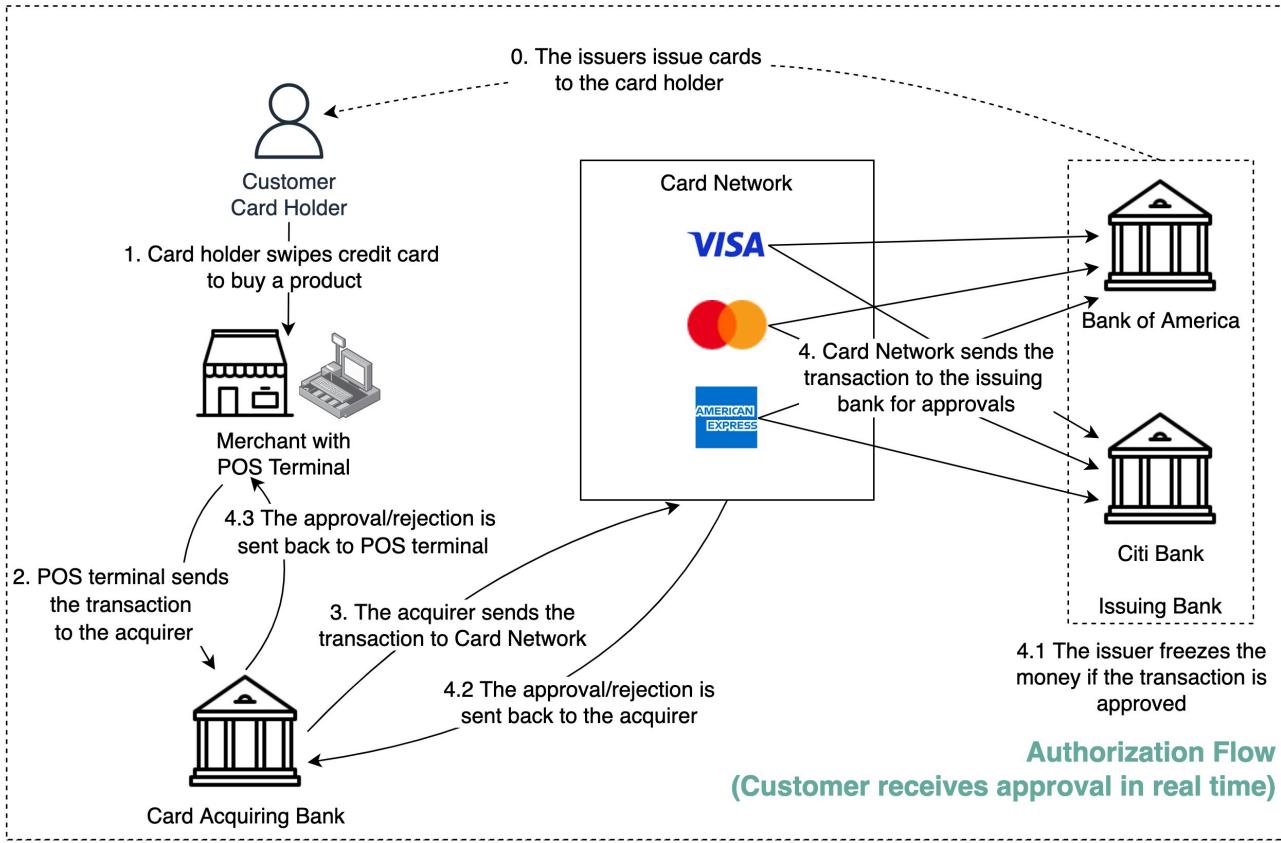
- 即使持卡人未能向发卡行支付款项，发卡行仍需支付商家。
- 发卡行在持卡人向其支付之前就已支付商家。
- 发卡行有其他运营成本，包括管理客户账户、提供对账单、欺诈检测、风险管理、清算等。

当我们在商家处刷卡时，VISA 是如何运作的？

How does VISA Work?



blog.bytebytego.com



VISA、Mastercard 和 American Express 充当资金清算和结算的卡网络。收单银行和发卡行可能（而且通常是）不同。如果银行要在没有中介的情况下一笔一笔地结算交易，那么每家银行都必须与所有其他银行结算交易。这是相当低效的。

上图显示了 VISA 在信用卡支付流程中的角色。涉及两个流程。当客户刷信用卡时就会发生授权流程（authorization flow）。当商家想要在一天结束时拿到钱时，就会发生捕获和结算流程（capture and settlement flow）。

- 授权流程

步骤 0：发卡行向其客户发放信用卡。

步骤 1：持卡人想购买产品，于是在商家店铺的销售点（POS）终端上刷信用卡。

步骤 2：POS 终端将交易发送给提供该 POS 终端的收单银行。

步骤 3 和 4：收单银行将交易发送到卡网络（也称为卡方案）。然后卡网络将交易发送到发卡行以获取审批。

步骤 4.1, 4.2 和 4.3：如果交易得到批准，发卡行会冻结资金。发送批准或拒绝的信息到收单银行和 POS 终端。

- 捕获和结算流程

步骤 1 和 2：商家希望在一天结束时收到款项，因此在 POS 终端上点击“捕获（capture）”。交易被批量发送到收单银行。收单银行会将带有交易的批处理文件发送到卡网络。

步骤 3：卡网络对从不同收单银行收集的交易进行清算，并将清算文件发送到不同的发卡行。

步骤 4：发卡行确认清算文件的正确性，并将资金转移给相关的收单银行。

步骤 5：收单银行然后将资金转移给商家的银行。

步骤 4：卡网络清理来自不同收单银行的交易。清算是一个互相抵消交易的过程，从而减少总交易数。

在此过程中，卡网络承担了与每家银行交涉的重任，并得到了服务费作为回报。

世界各地的支付系统系列（第一部分）：印度的统一支付接口（Unified Payments Interface, UPI）

什么是 UPI？UPI 是由印度国家支付公司开发的即时实时支付系统。

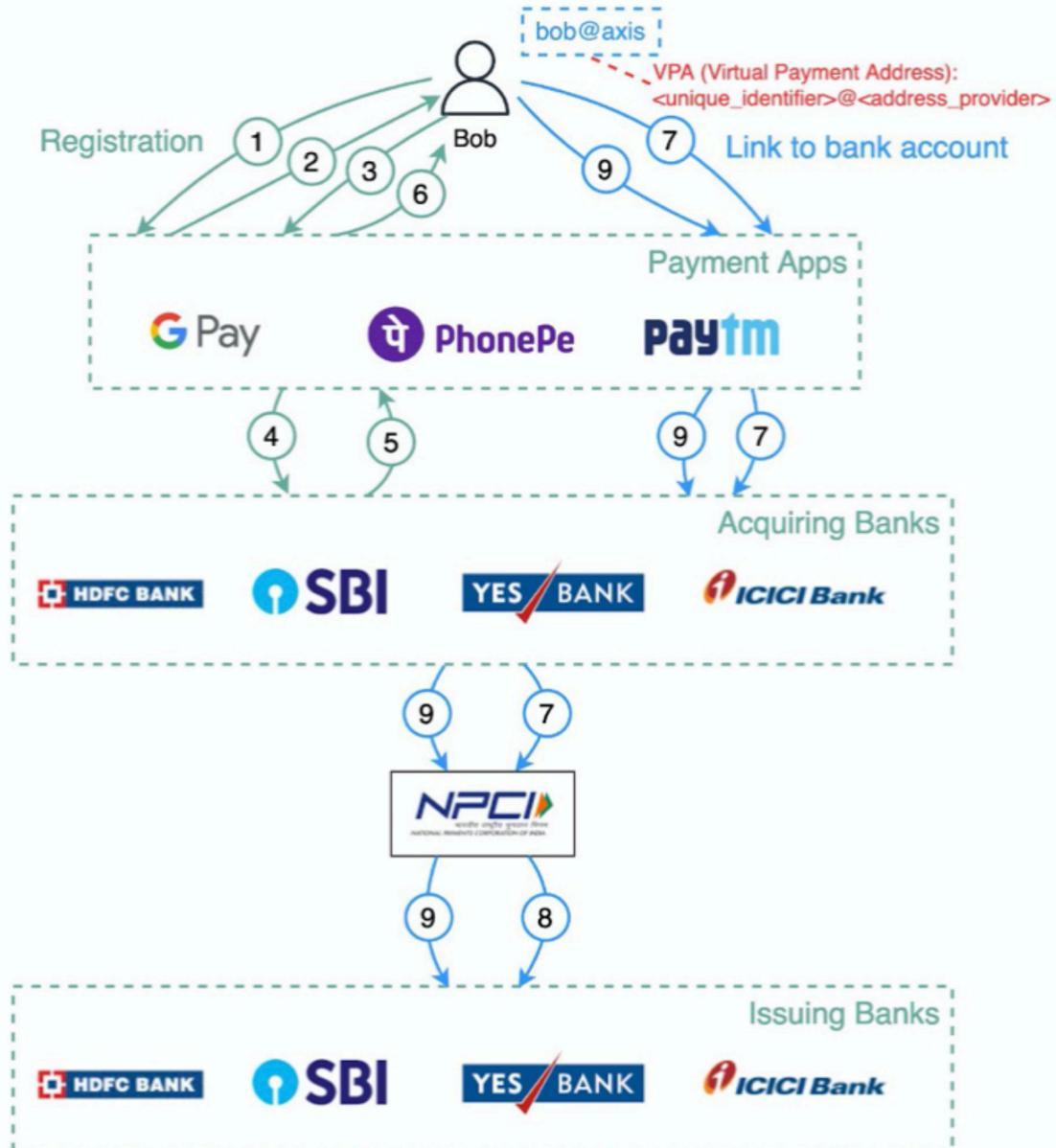
它占当今印度数字零售交易的 60%。

UPI = 支付标记语言 (payment markup language) + 可互操作支付标准 (standard for interoperable payments)

How does UPI Work?

 blog.bytebytogo.com

1. Registration & Link to Bank Account



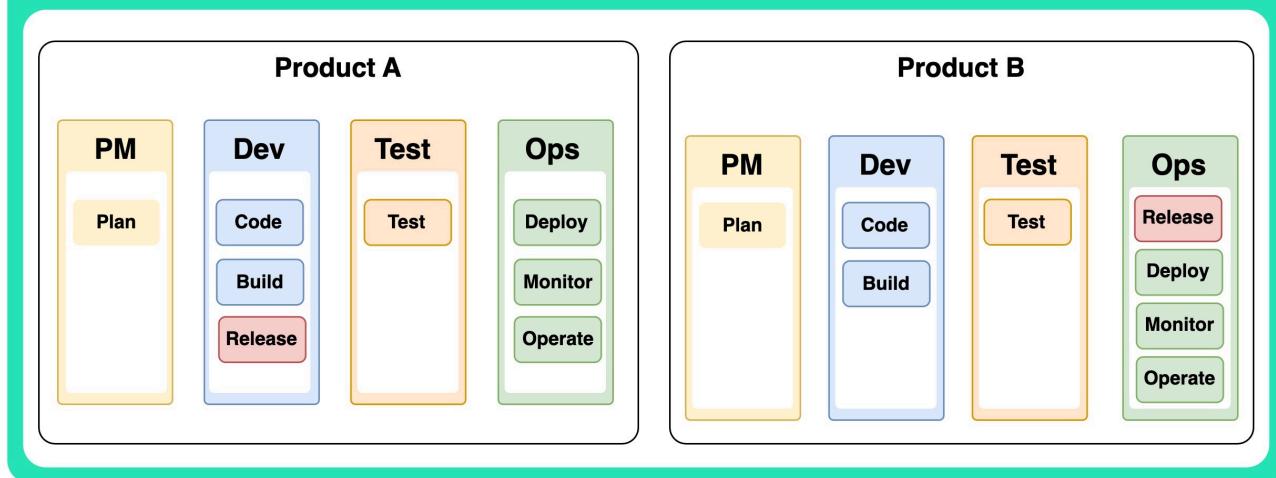
DevOps

DevOps vs. SRE vs. 平台工程（Platform Engineering）。有何不同？

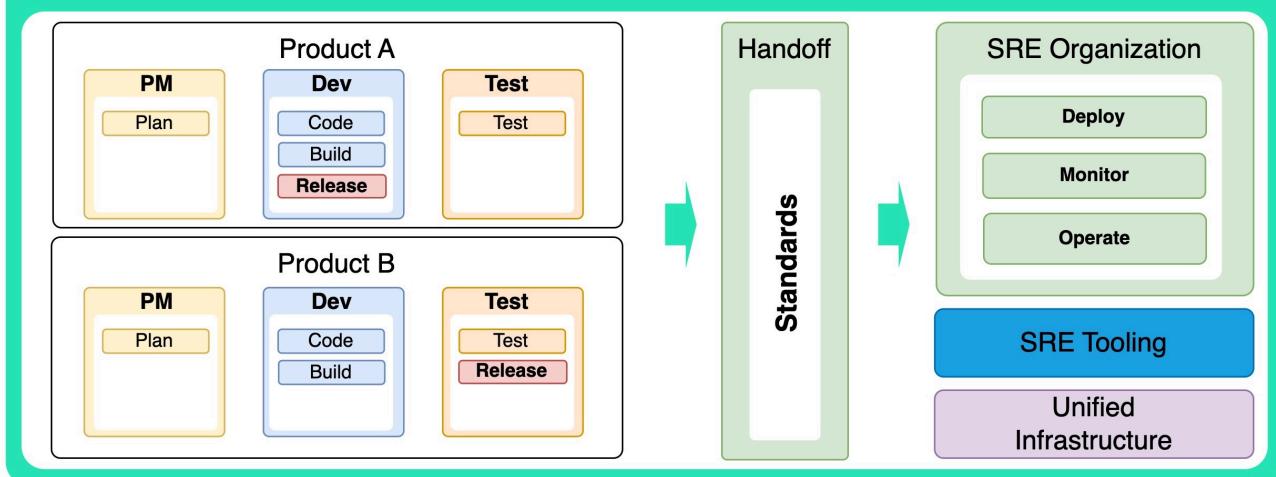
DevOps、SRE 和平台工程的概念是在不同时间出现的，并由不同的个人和组织开发。



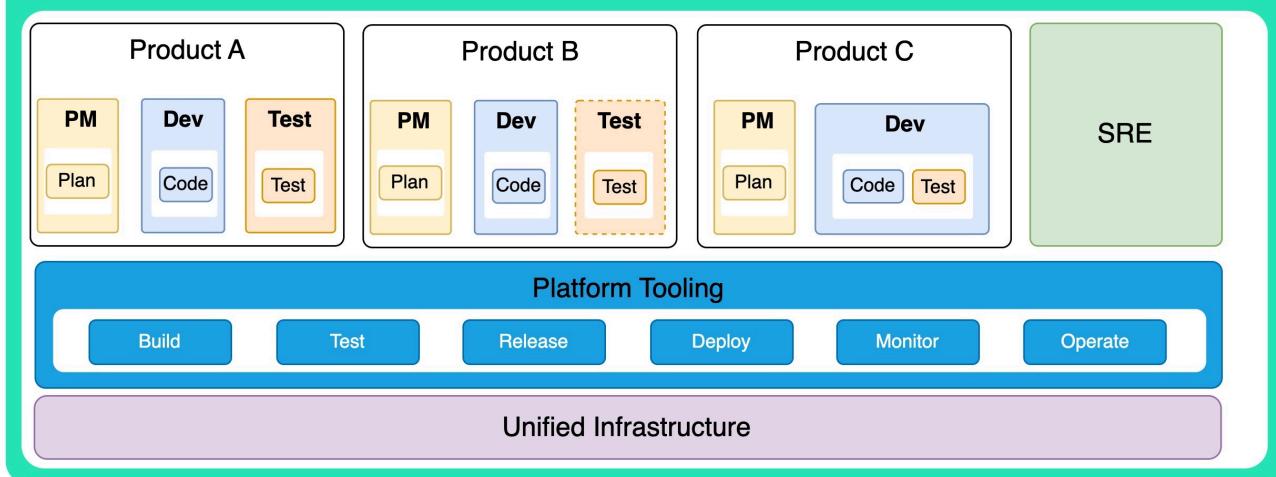
DevOps



SRE (Site Reliability Engineering)



Platform Engineering



DevOps 的概念是 Patrick Debois 和 Andrew Shafer 于 2009 年在敏捷大会上提出的。他们试图通过促进对整个软件开发生命周期的协作文化和共同责任，来弥合软件开发和运维之间的差距。

SRE (Site Reliability Engineering, 站点可靠性工程) 由谷歌在 2000 年代初首创，用以解决管理大规模复杂系统的运维挑战。谷歌开发了 SRE 实践和工具，例如 Borg 集群管理系统和 Monarch 监控系统，来提高其服务的可靠性和效率。

平台工程则是一个较新的概念，构建在 SRE 工程的基础上。平台工程的确切起源未明，但通常认为它是 DevOps 和 SRE 实践的延伸，专注于提供一个产品开发的综合平台，来支持整个业务视角。

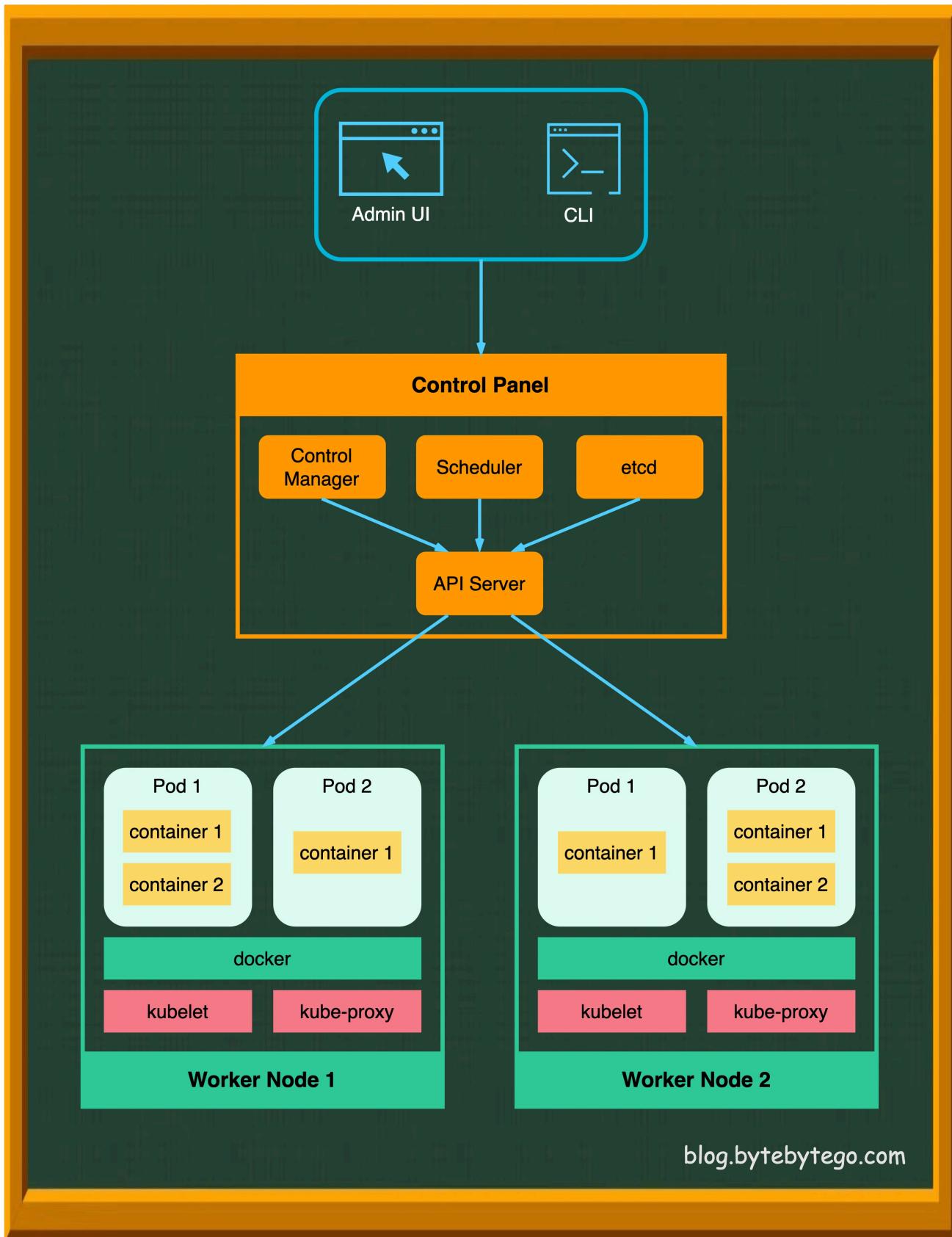
值得注意的是，虽然这些概念是在不同时期出现的，它们都与改善软件开发和运营中的协作、自动化和效率的更广泛趋势息息相关。

k8s (Kubernetes) 是什么？

K8s 是一个容器编排系统。用于容器部署和管理。其设计很大程度上受到 Google 内部系统 Borg 的影响。

What is k8s?

 blog.bytebybytego.com



blog.bytebybytego.com

一个 Kubernetes (k8s) 集群由一组称为节点 (node) 的工作机器组成，这些节点上运行着容器化应用程序。每个集群至少有一个工作节点 (worker node)。

工作节点托管 Pod，也就是应用工作负载的组件。控制平面 (control plane) 负责管理集群中的工作节点和Pod。在生产环境中，控制平面通常跨多台计算机运行，而一个集群通常运行多个节点，提供容错性和高可用性。

- 控制平面组件 (Control Plane Components)

1. API 服务器 (API Server)

API 服务器与 k8s 集群中的所有组件通信。Pod 上的所有操作都是通过与 API 服务器通信来执行的。

2. 调度器 (scheduler)

调度器监控 Pod 工作负载，并给新创建的 Pod 分配负载。

3. 控制器管理器 (controller manager)

控制器管理器运行控制器，包括节点控制器 (Node Controller)、作业控制器 (Job Controller)、EndpointSlice 控制器 (EndpointSlice Controller) 和 ServiceAccount 控制器 (ServiceAccount Controller)。

4. Etcd

etcd 是一个键值存储，用作 Kubernetes 所有集群数据的后备存储。

- 节点 (Node)

1. Pod

一个 Pod 是一组容器，也是 k8s 管理的最小单元。每一个 Pod 都有一个单独的 IP 地址，Pod 中的每一个容器共享这个 IP 地址。

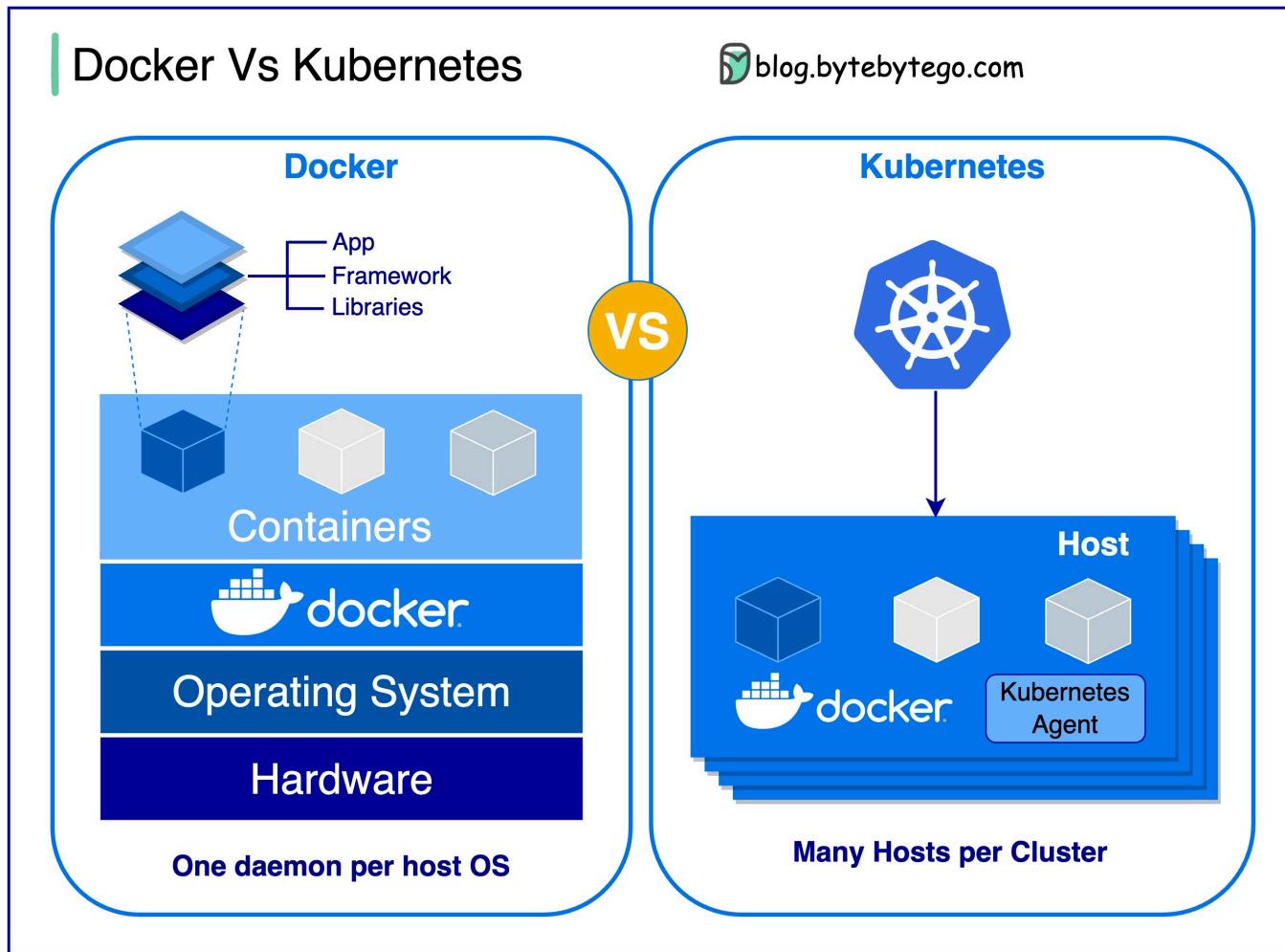
2. Kubelet

集群中每个节点上都运行的代理。它保证了容器在 Pod 中运行。

3. Kube Proxy

Kube-proxy 是一个网络代理，它运行在集群中的每一个节点上。它路由来自服务的流量到节点。它还将工作请求转发到正确的容器中。

Docker vs. Kubernetes。我们应该用哪一个？



Docker 是什么？

Docker 是一个开源平台，允许您打包、分发以及在隔离的容器中运行应用。它专注于容器化，提供封装了应用及其依赖的轻量级环境。

Kubernetes 是什么？

Kubernetes，通常称为 K8s，是一个开源的容器编排平台。它提供了一个框架，用于自动化部署、扩展、以及管理跨节点集群中的容器化应用。

它们两者之间有什么区别呢？

Docker: Docker 在单个操作系统主机上的单个容器级别上运行。

您必须手动管理每个主机，而为多个相关容器设置网络、安全策略和存储可能会变得复杂。

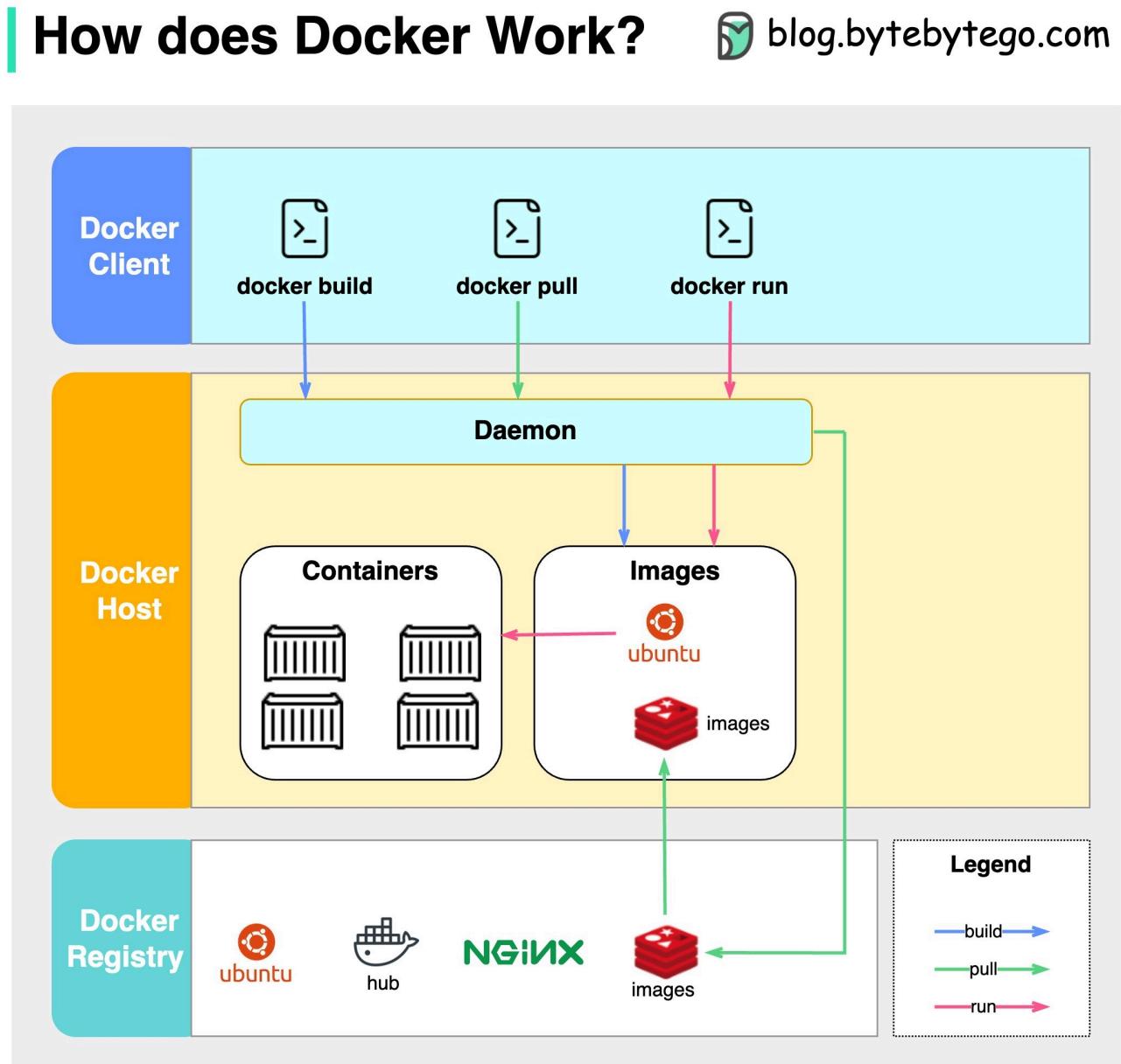
Kubernetes: Kubernetes 在集群级别上运行。它管理跨多个主机的多个容器化应用程序，提供

任务（如负载平衡、扩展）自动化，并确保应用程序所需状态。

简而言之，Docker 专注于容器化和在单个主机上运行容器，而 Kubernetes 则专注于跨主机集群大规模管理和编排容器。

Docker 的工作原理

下图显示了 Docker 的架构，以及当我们运行了“docker build”、“docker pull”和“docker run”时，它是如何工作的。



Docker 架构中有 3 个组件：

- Docker 客户端

Docker 客户端与 Docker daemon 通信。

- Docker 主机

Docker daemon 监听 Docker API 请求，管理 Docker 对象，例如镜像 (image)、容器、网络和卷。

- Docker registry

Docker registry 存储 Docker 镜像。Docker Hub 是一个人和人都可以使用的公共镜像存储库。

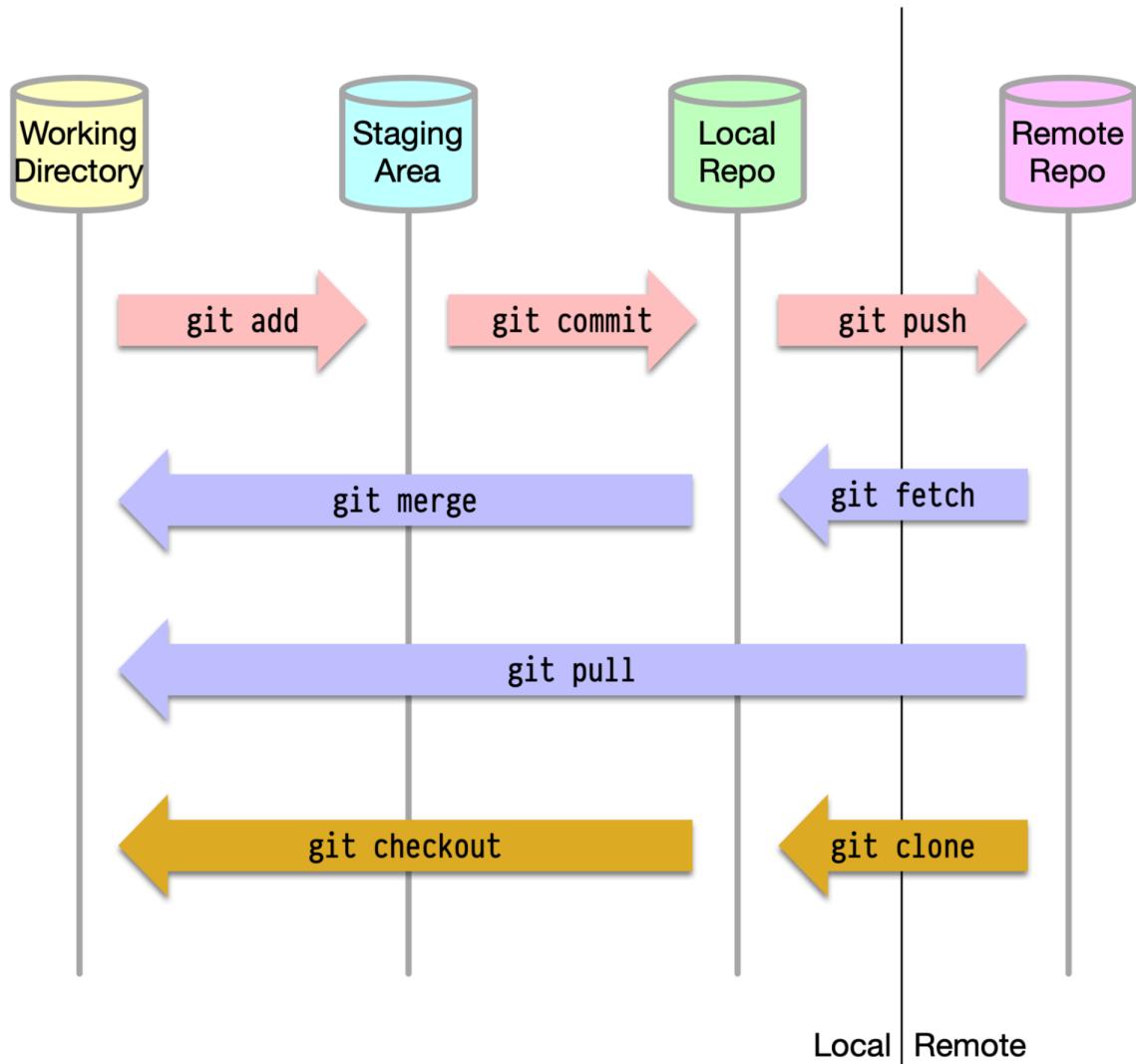
让我们以“`docker run`”命令为例。

1. Docker 从 registry 拉取镜像。
2. Docker 创建一个新的容器。
3. Docker 为容器分配一个读写文件系统。
4. Docker 创建一个网络接口，来将容器连接到默认网络上。
5. Docker 启动容器。

GIT

Git 命令是如何工作的？

首先，确定代码存储在何处是至关重要的。通常假设只有两个地方——一份代码在像 Github 这样的远程服务器上，而另一份则在我们的本地机器上。然而，这不完全正确。Git 在我们的机器上维护了三份本地存储，这意味着，我们的代码可以在四个地方找到：



- 工作目录 (Working directory) : 我们编辑文件的地方
- 暂存区 (Staging area) : 一个临时区域，代码保存在此处，以备下一次提交
- 本地仓库 (Local repository) : 包含已提交的代码
- 远程仓库 (Remote repository) : 存储代码的远程服务器

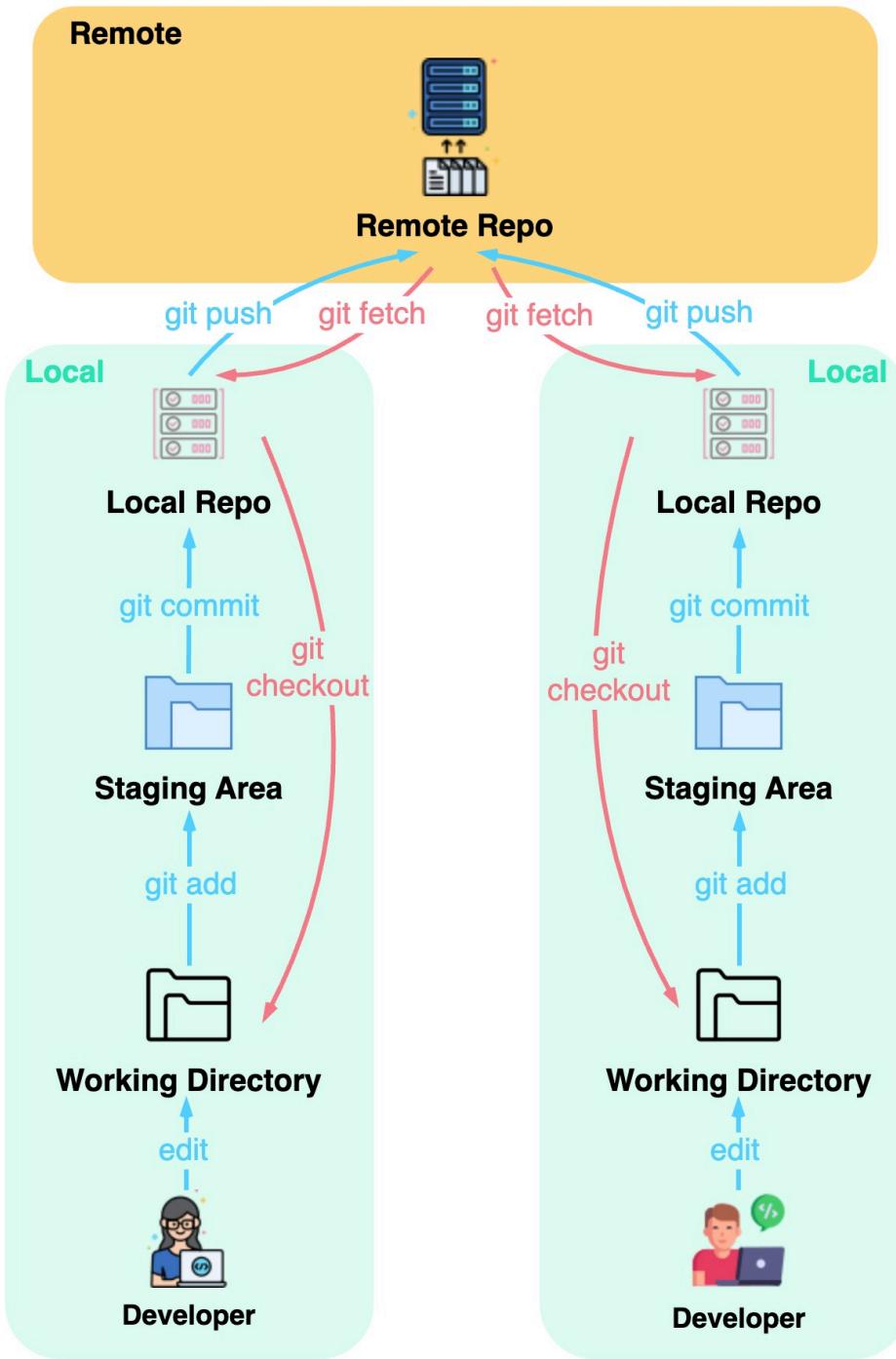
大多数的 Git 命令主要在这四个位置之间移动文件。

Git 的工作原理

下图显示了 Git 的工作流程。

How does Git Work?

 blog.bytebytego.com



Git 是一个分布式版本控制系统。

每一位开发者都维护了主仓库的一份本地拷贝，他们编辑以及提交到这个本地拷贝中。

提交操作是非常快的，因为此操作并不与远程仓库进行交互。

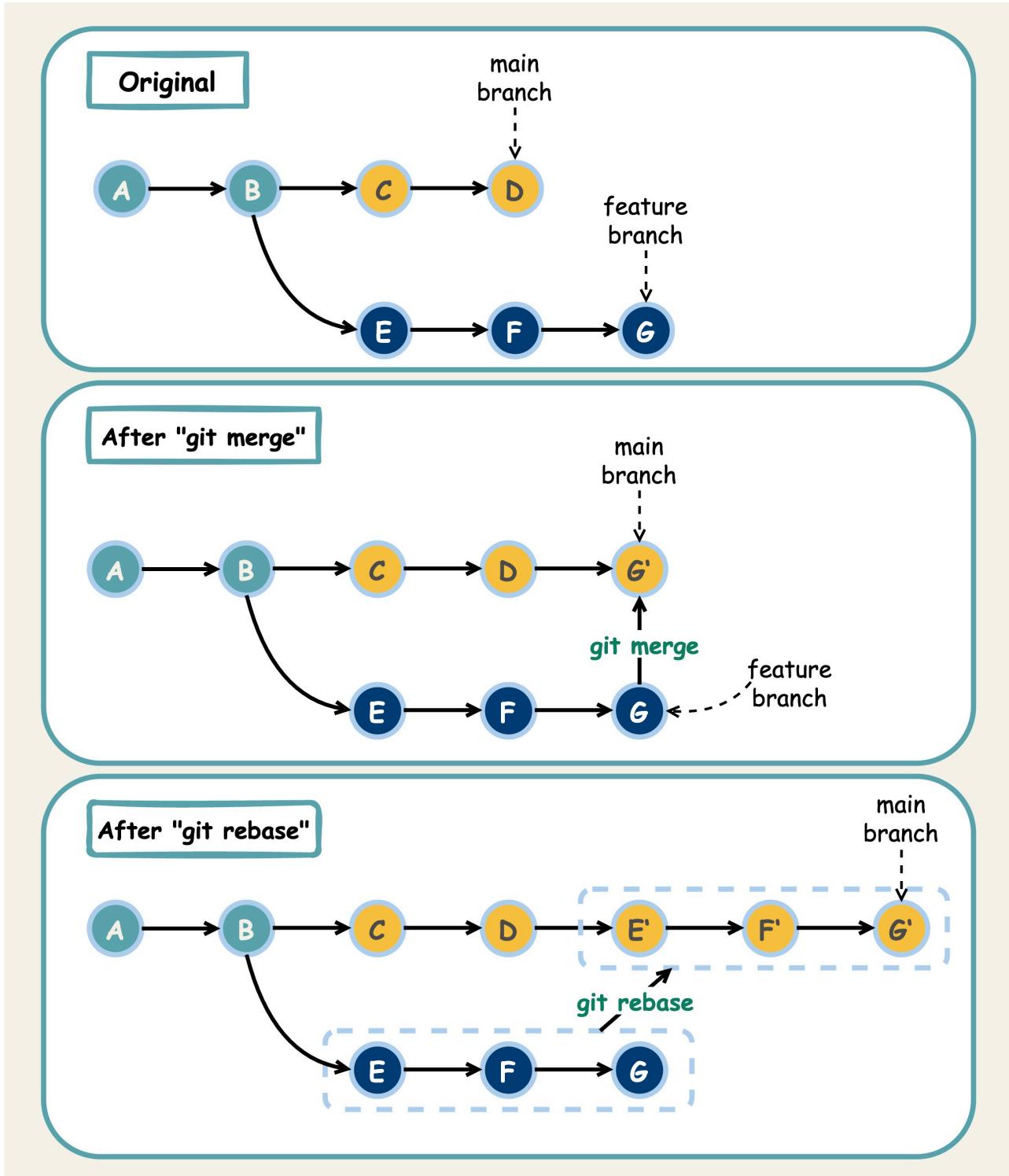
如果远程仓库挂了，可以从本地仓库恢复文件。

Git merge vs. Git rebase

有何不同？

Git Merge vs. Git Rebase

 blog.bytebybytego.com



当我们使用“git merge”或者“git rebase”命令将一个 Git 分支的更改合并到另一个分支时，我们可以选择不同的操作方式。下图展示了两个命令的工作原理。

Git merge

这在主分支创建了一个新的提交 G'。G' 将主分支和特性分支的历史记录连接在一起。

Git merge 是非破坏性的。主分支和特性分支都不会被修改。

Git rebase

Git rebase 将特性分支的历史记录移动到主分支的头部。它为特性分支中的每次提交，分别创建新的提交 E'、F' 和 G'。

rebase 的好处是，通过它，我们可以拥有一个线性的提交历史记录。

如果不遵循“git rebase 的黄金法则”，rebase 就有可能是一种危险操作。

git rebase 的黄金法则

永远不要在公共分支上使用它！

云服务

不同云服务的便捷速查表（2023 年版本）

Cloud Comparison Cheat Sheet

 blog.bytebybytego.com



-  Elastic Compute Cloud (EC2)
-  Elastic Kubernetes Service (EKS)
-  Lambda
-  Simple Storage Service (S3)
-  Elastic Block Store
-  Elastic File System
-  Virtual Private Cloud
-  Route 53
-  Elastic Load Balancing
-  Web Application Firewall
-  RDS
-  DynamoDB
-  Redshift
-  Elastic MapReduce
-  Kinesis
-  SageMaker
-  Glue
-  EventBridge
-  Simple Queuing Service
-  Simple Notification Service
-  CloudWatch
-  CloudFormation
- IAM



-  Virtual Machine
-  Azure Kubernetes Service (AKS)
-  Azure Functions
-  Blob Storage
-  Managed Disk
-  File Storage
-  Virtual Network
-  DNS
-  Load Balancer
-  Web Application Firewall
-  SQL Database
-  Cosmos DB
-  Synapse Analytics
-  HDInsight
-  Streaming Analytics
-  Machine Learning
-  Data Factory
-  Event Grid
-  Storage Queues
-  Service Bus
-  Monitor
-  Resource Manager
- Active Directory



-  Compute Engine
-  Google Kubernetes Engine (GKE)
-  Cloud Functions
-  Cloud Storage
-  Persistent Disk
-  File Store
-  Virtual Private Cloud
-  Cloud DNS
-  Cloud Load Balancing
-  Cloud Armor
-  Cloud SQL
-  Firebase Realtime Database
-  BigQuery
-  Dataproc
-  Dataflow
-  Vertex AI
-  Data Fusion
-  Eventarc
-  Pub/Sub
-  Firebase Cloud Messaging
-  Cloud Monitoring
-  Deployment Manager
- Cloud Identity



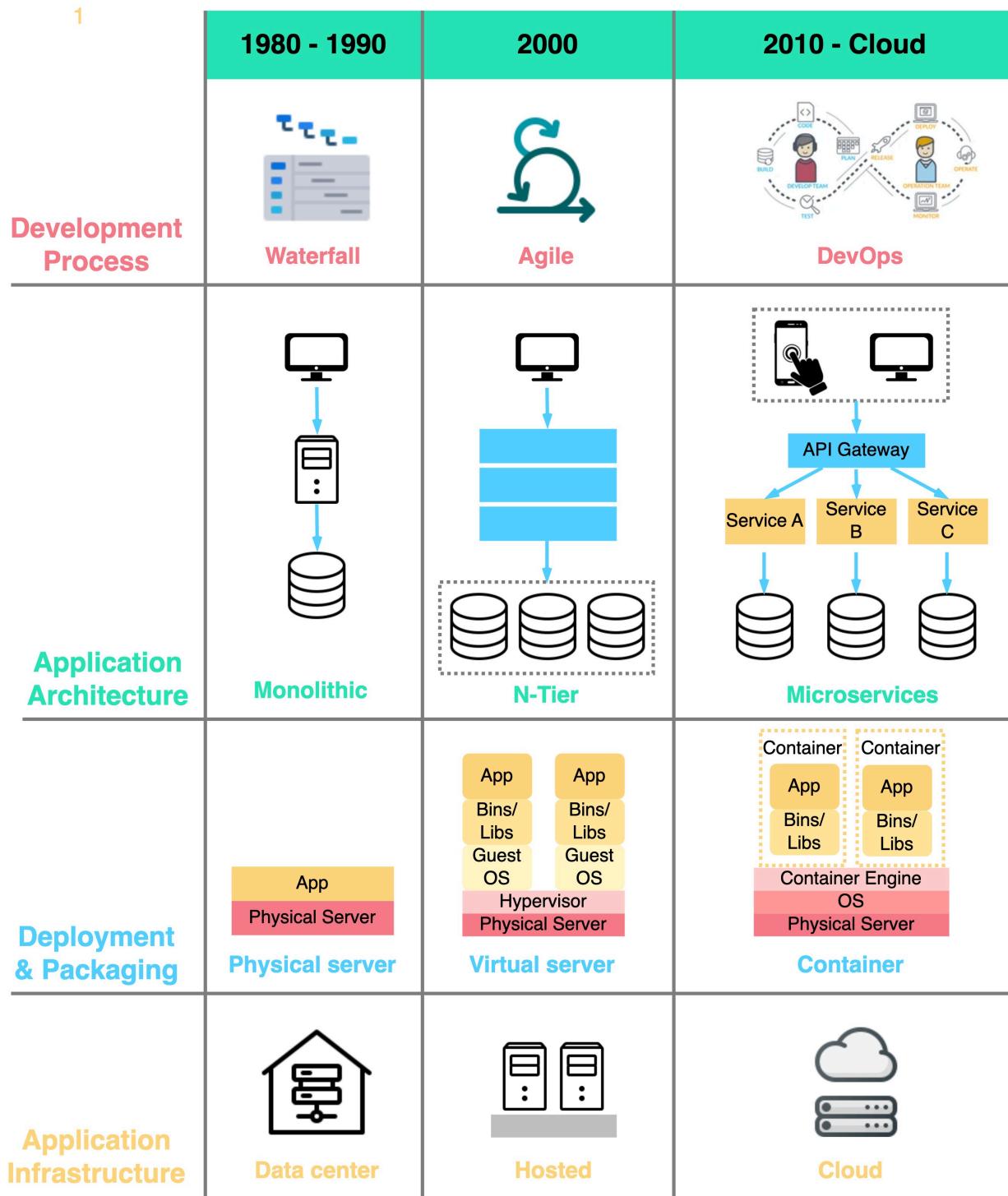
-  Virtual Machine
-  Oracle Container Engine
-  OCI Functions
-  Object Storage
-  Persistent Volume
-  File Storage
-  Virtual Cloud Network
-  DNS
-  Load Balancer
-  Web Application Firewall
-  ATP
-  NoSQL Database
-  Autonomous Data Warehouse
-  Big Data
-  Streaming
-  Data Science
-  Data Integration
-  Events
-  Streaming
-  Notifications
-  Monitoring
-  Resource Manager
- IAM
- Metrics

什么是云原生 (cloud native) ?

下图显示了自 20 世纪 80 年代以来架构和流程的演变。

What is Cloud Native?

 blog.bytebytego.com



Reference: <https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/>

通过使用云原生技术，组织可以在公有云、私有云和混合云上构建和运行可扩展的应用程序。

这意味着应用被设计成能够利用云功能，因此，它们具有负载弹性，并且易于扩展。

云原生包含 4 个方面：

1. 开发流程

它已经从瀑布流发展到敏捷，再到 DevOps。

2. 应用架构

架构已经从单体架构演变为微服务。每个服务都被设计得很小，从而适应云容器中的有限资源。

3. 部署和打包

过去，应用被部署到物理服务器上。然后，在 2000 年左右，那些对延迟不敏感的应用通常被部署到虚拟服务器上。对于云原生应用，它们被打包成 docker 镜像并部署在容器中。

4. 应用基础设施

这些应用被大量部署在云基础设施上，而不是在自托管服务器上。

开发者生产力工具

可视化 JSON 文件

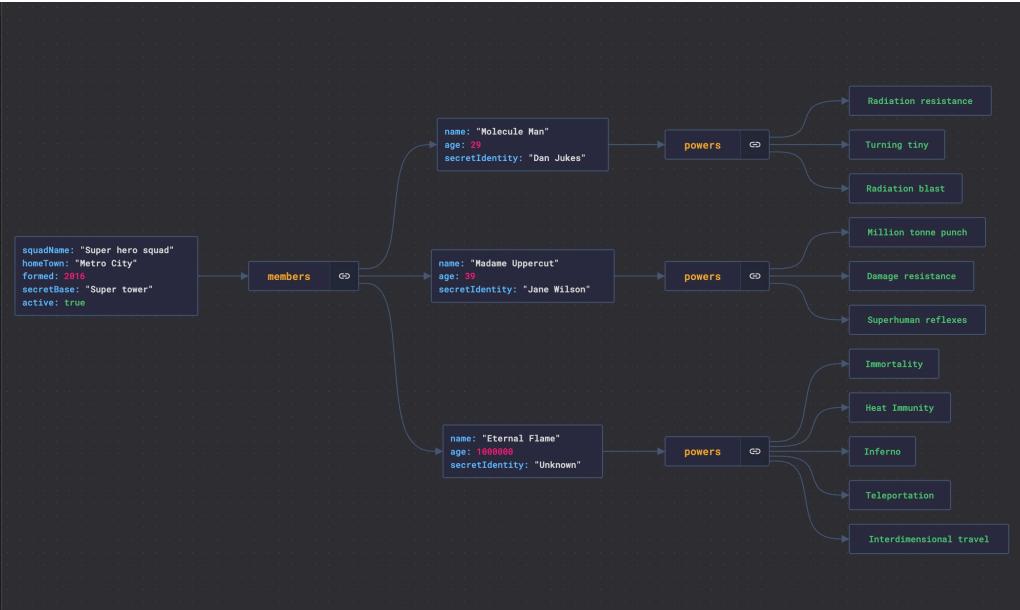
嵌套的 JSON 文件是难以阅读的。

JsonCrack 根据 JSON 文件生成图表，从而使其易于阅读。

此外，生成的图表可以下载为图片。

```

1  {
2    "squadName": "Super hero squad",
3    "homeTown": "Metro City",
4    "formed": 2016,
5    "secretBase": "Super tower",
6    "active": true,
7    "members": [
8      {
9        "name": "Molecule Man",
10       "age": 29,
11       "secretIdentity": "Dan Jukes",
12       "powers": [
13         "Radiation resistance",
14         "Turning tiny",
15         "Radiation blast"
16       ]
17     },
18     {
19       "name": "Madame Uppercut",
20       "age": 39,
21       "secretIdentity": "Jane Wilson",
22       "powers": [
23         "Million tonne punch",
24         "Damage resistance",
25         "Superhuman reflexes"
26       ]
27     },
28     {
29       "name": "Eternal Flame",
30       "age": 1000000,
31       "secretIdentity": "Unknown",
32       "powers": [
33         "Immortality",
34         "Heat Immunity",
35         "Inferno",
36         "Teleportation",
37         "Interdimensional travel"
38     }
39   ]
40 }
41 
```



自动将代码转换为架构图

Diagram as Code

```
from diagrams import Cluster, Diagram
from diagrams.aws.compute import ECS
from diagrams.aws.database import ElastiCache, RDS
from diagrams.aws.network import ELB
from diagrams.aws.network import Route53

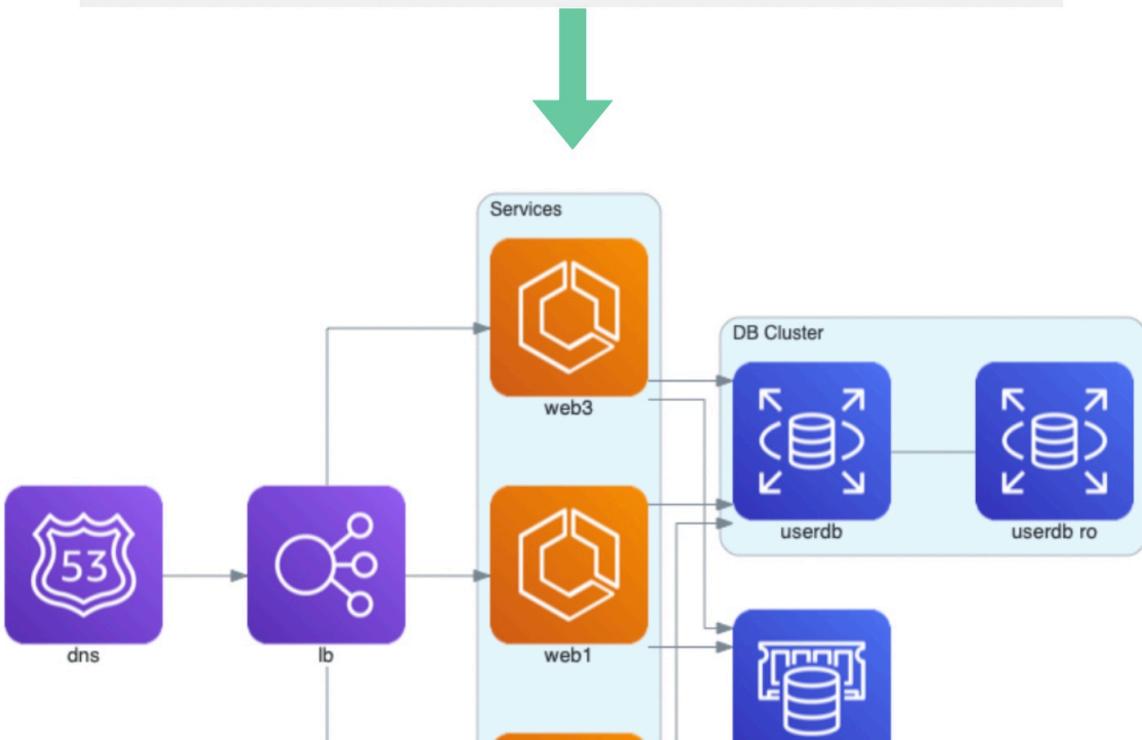
with Diagram("Clustered Web Services", show=False):
    dns = Route53("dns")
    lb = ELB("lb")

    with Cluster("Services"):
        svc_group = [ECS("web1"),
                     ECS("web2"),
                     ECS("web3")]

    with Cluster("DB Cluster"):
        db_primary = RDS("userdb")
        db_primary - [RDS("userdb ro")]

    memcached = ElastiCache("memcached")

    dns >> lb >> svc_group
    svc_group >> db_primary
    svc_group >> memcached
```



它的用处是什么？

- 通过编写 Python 代码绘制云系统架构。
- 也可以直接在 Jupyter Notebooks 中渲染图表。
- 无需设计工具。
- 支持以下提供商：AWS、Azure、GCP、Kubernetes、Alibaba Cloud、Oracle Cloud 等等。

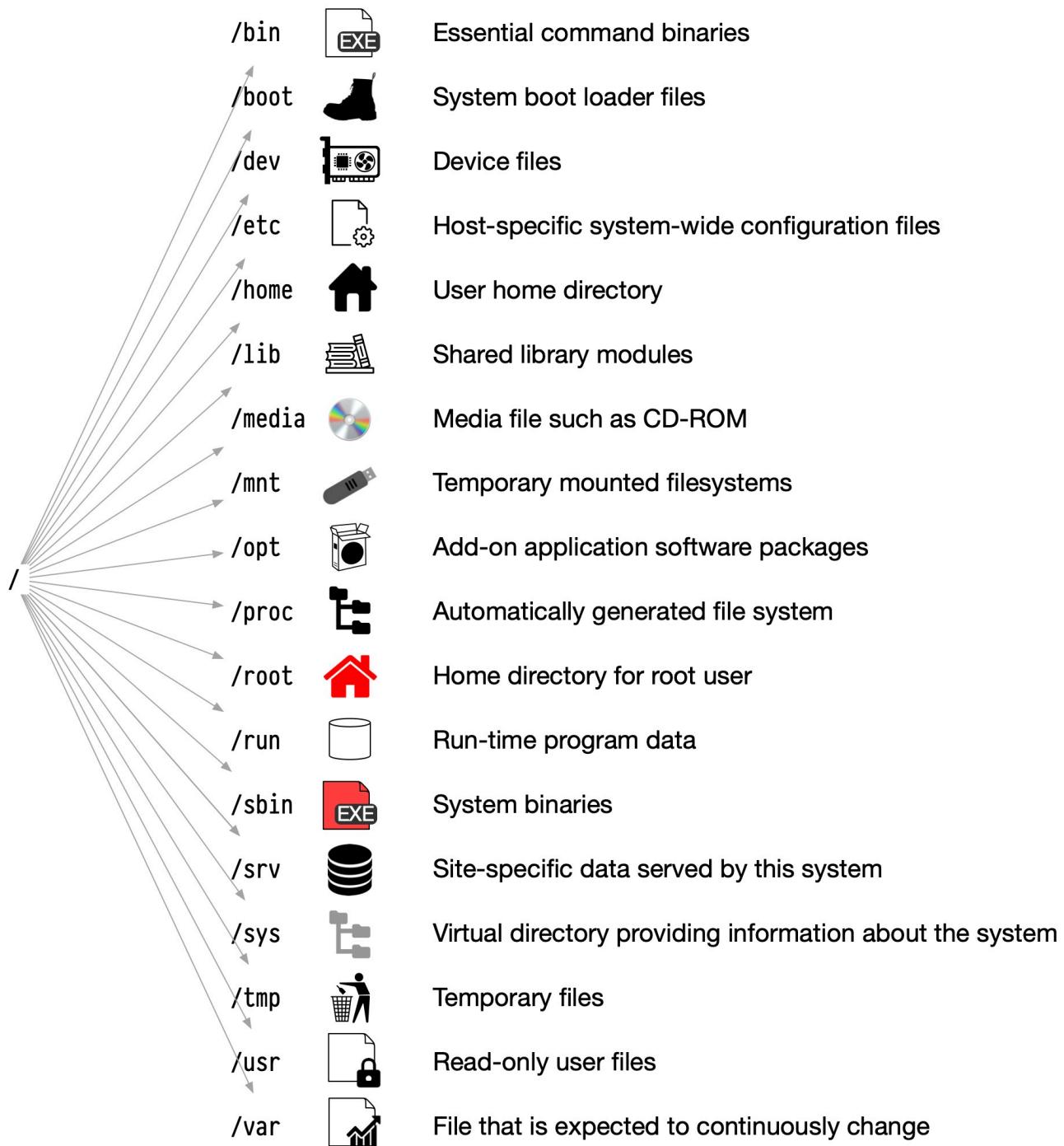
[Github 仓库](#)

Linux

Linux 文件系统解析

Linux File Systems

 ByteByteGo.com



过去，Linux 文件系统就像一个无组织的城镇，人们可以随心所欲地在他们喜欢的地方建造自己的房子。然而，在 1994 年的时候，文件系统层次结构标准（Filesystem Hierarchy Standard, FHS）被引入，从而为 Linux 文件系统带来了秩序。

通过实施诸如 FHS 这样的标准，软件可以确保在各种 Linux 发行版上的布局一致。尽管如此，并非所有 Linux 发行版都严格遵守此标准。它们通常会融入自己独特的元素或满足特定的要求。要精通此标准，您可以从探索它开始。使用“cd”等命令进行导航，使用“ls”等命令列出目录内容。将文件系统想象成一棵从根（/）开始的树。随着时间的推移，您将自然而然地使用它，从而使您成为熟练的 Linux 管理员。

你应该知道的 18 个最常用的 Linux 命令

Linux 命令是用来与操作系统交互的指令。它们帮助我们管理文件、目录、系统进程、以及系统的许许多多其他方面的内容。为了高效地导航和维护基于 Linux 的系统，您需要熟悉这些命令。

下图显示了流行的 Linux 命令：

18 Most-Used Linux Commands You Should Know

 blog.bytebytogo.com

ls

List files and directories

cd

Change current directory

mkdir

Create new directory

rm

Remove files or directories

mv

Move or rename files or
directories

chmod

Change file or directory
permission

cp

Copy files or directories

find

Search for files or
directories

grep

Search for a pattern in
files

vi

Edit files using text editor

cat

Display the content of files

tar

Manipulate tarball archive
files

ps

Display process
information

kill

Terminate process by
sending a signal

top

Display process and
resource usage

ifconfig

Configure network
interfaces

ping

Test network connectivity
between hosts

du

Estimate file space usage

- ls - 列出文件和目录
- cd - 更改当前目录
- mkdir - 创建一个新的目录

- rm - 删除文件或者目录
- cp - 拷贝文件或者目录
- mv - 移动或重命名文件或者目录
- chmod - 更改文件或者目录权限
- grep - 在文件中搜索指定模式的内容
- find - 搜索文件或者目录
- tar - 操作 tarball 归档文件
- vi - 使用文本编辑器编辑文件
- cat - 显示文件内容
- top - 显示进程和资源使用情况
- ps - 显示进程信息
- kill - 通过发送信号来终止进程
- du - 评估文件空间使用情况
- ifconfig - 配置网络接口
- ping - 测试主机之间的网络连通性

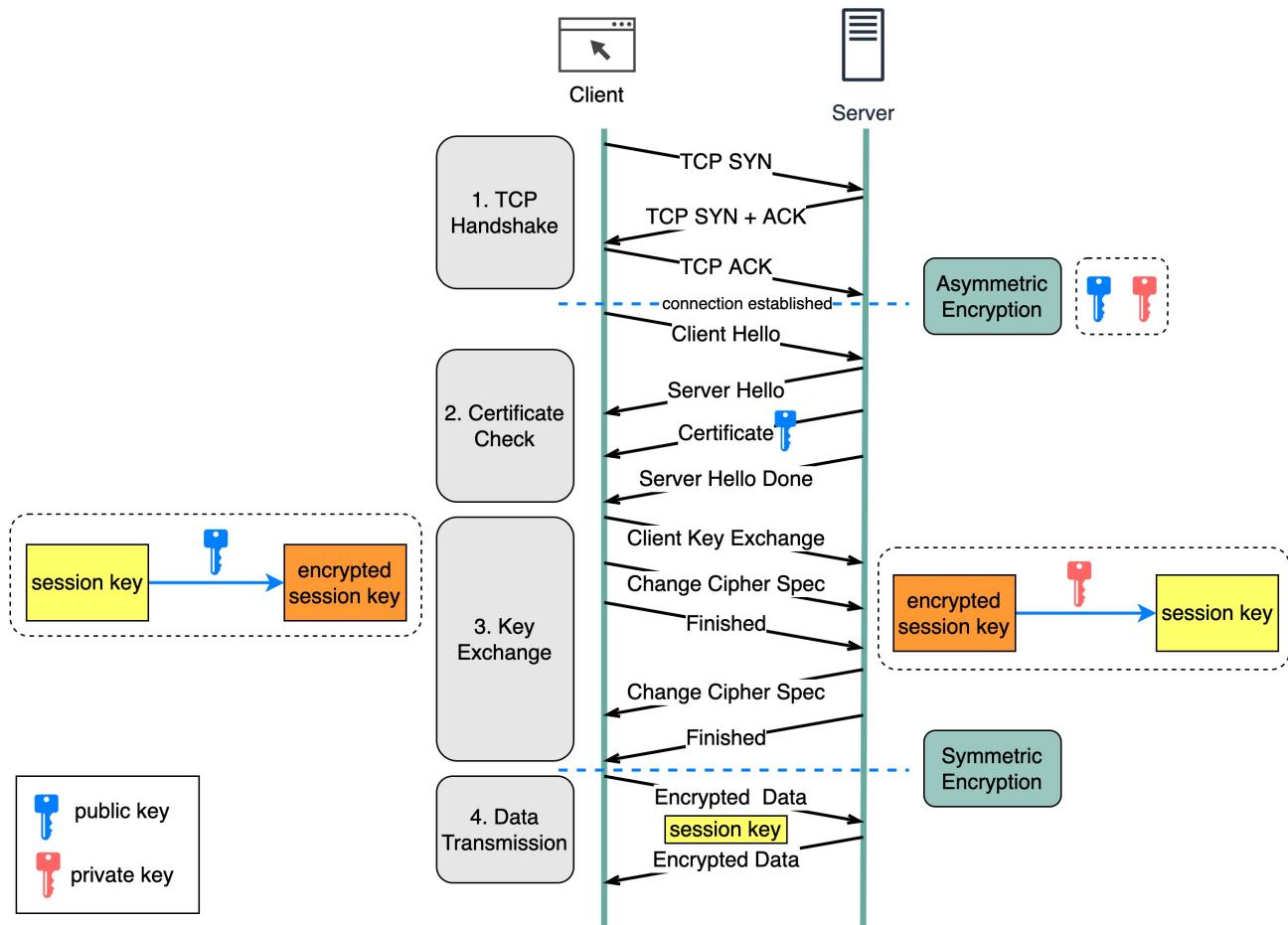
安全

HTTPS 是如何工作的？

超文本传输安全协议（Hypertext Transfer Protocol Secure, HTTPS）是超文本传输协议（Hypertext Transfer Protocol, HTTP）的扩展。HTTPS 使用传输层安全性（Transport Layer Security, TLS）传输加密数据。即使数据被在线劫持，劫持者得到的也只是二进制码。

How does HTTPS Work?

 blog.bytebytego.com



数据是如何被加密解密的？

步骤 1 - 客户端（浏览器）和服务器端建立一个 TCP 连接。

步骤 2 - 客户端发送一条“client hello”消息给服务器端。这条消息包含了一组必要的加密算法（密码套件）以及它支持的最新的 TLS 版本。接着，服务器端用一条“server hello”消息进行响应，以便浏览器知道它是否支持这些算法和 TLS 版本。

接着，服务器端发送 SSL 证书给客户端。证书包含公钥、主机名、过期日期等。客户端会验证此证书。

步骤 3 - 验证 SSL 证书后，客户端生成一个会话密钥，然后使用公钥对其进行加密。服务器端收到了这个加密的会话密钥后，使用私钥对其进行解密。

步骤 4 - 现在，客户端和服务器端都持有相同的会话密钥（对称加密），因此，加密数据就可以通过安全的双向通道进行传输了。

为什么在数据传输过程中，HTTPS 要切换到对称加密呢？有两个主要原因：

1. 安全性：非对称加密只能单向进行。这意味着，如果服务器端尝试将加密数据发送回客户端，那么任何人都可以使用公钥解密数据。
2. 服务器资源：非对称加密增加了相当多的数学计算开销。因此，它不适用于长会话中的数据传输。

简明扼要解释下 Oauth 2.0。

OAuth 2.0 是一个强大且安全的框架，允许不同应用代表用户安全地彼此之间进行交互，而无需共享敏感的凭据。

What is OAuth?

* Protocol for sharing user Authorization across systems.

Involves 3 entities



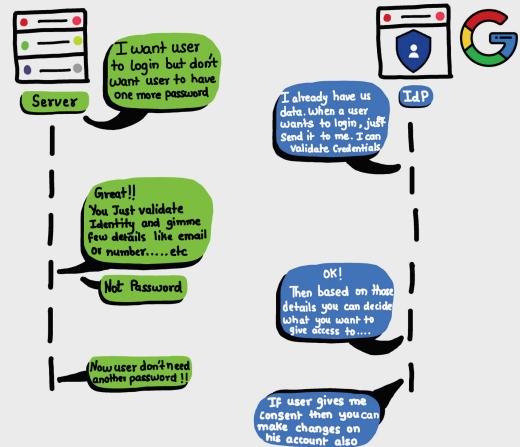
Entity who's authorization flows across the Systems. Also known as resource Owner



Service, User wants to access and Complete Authorization across.



Identity Provider which Stores user's identity and validate User's Credentials & Share user Authorization with other Services.



1.0 protocol designed for web browser only



2.0 protocol upgraded for browser App non browser App windows App mobile App APIs

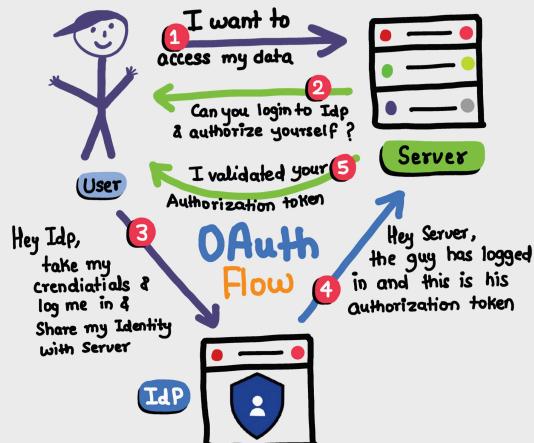


OAuth

@ Sec_#0 ByteByteGo

Open Authorization

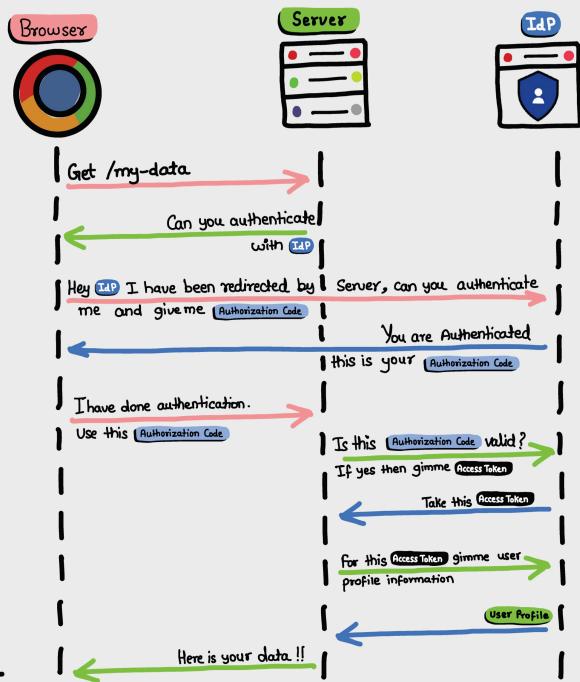
2.0



- ## 4 Types of OAuth Flows
- ① Authorization code
 - ② Client Credentials
 - ③ Implicit Code
 - ④ Resource owner Password

Authorization Code

Flow



OAuth 中涉及的实体包括用户、服务器和身份提供商（Identity Provider, IDP）。

OAuth 令牌（OAuth Token）能做什么？

使用 OAuth 时，你会获得一个代表你身份和权限的 OAuth 令牌。这个令牌可以执行一些重要的操作：

单点登录（Single Sign-On, SSO）：使用一个 OAuth 令牌，只要进行一次登录操作，你就可以上到多个服务或者应用，让生活轻松安全多了。

系统间授权（Authorization Across Systems）：OAuth 令牌让你可以在各个系统之间共享你的授权或者访问权限，因此，你不需要在每处单独登录。

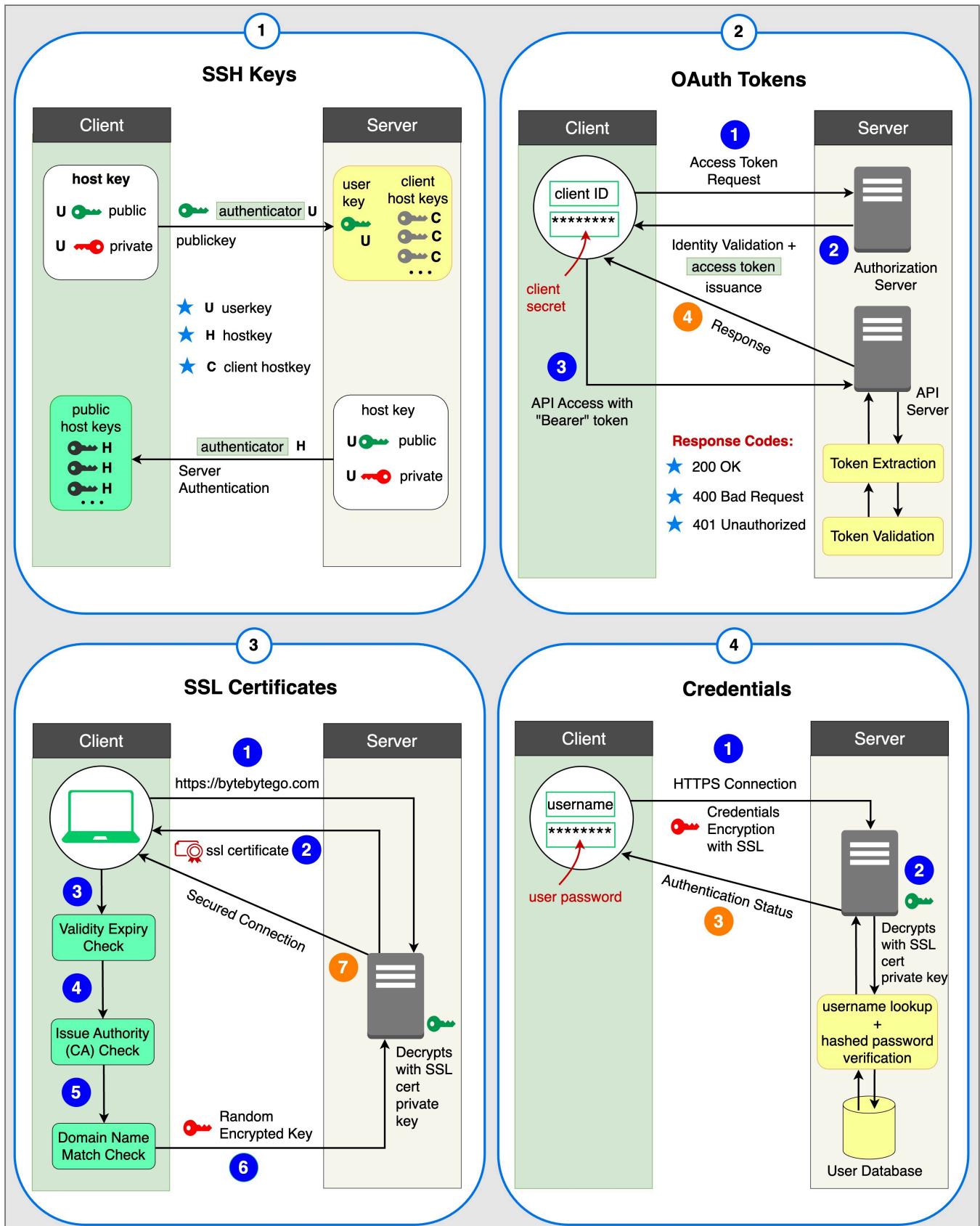
访问用户资料（Accessing User Profile）：拥有 OAuth 令牌的应用可以访问你允许的用户资料的特定部分内容，但不会看到所有内容。

记住，OAuth 2.0 的目标是在确保你和你的数据安全的同时，让你在不同的应用和服务之间的在线体验变得更加无缝和轻松。

四种最常见的身份认证机制

Top 4 Most Used Authentication Mechanisms

 blog.bytebytego.com



1. SSH 密钥:

使用加密密钥来安全访问远程系统和服务器。

2. OAuth 令牌:

令牌为第三方应用提供对用户数据的有限访问权限。

3. SSL 证书:

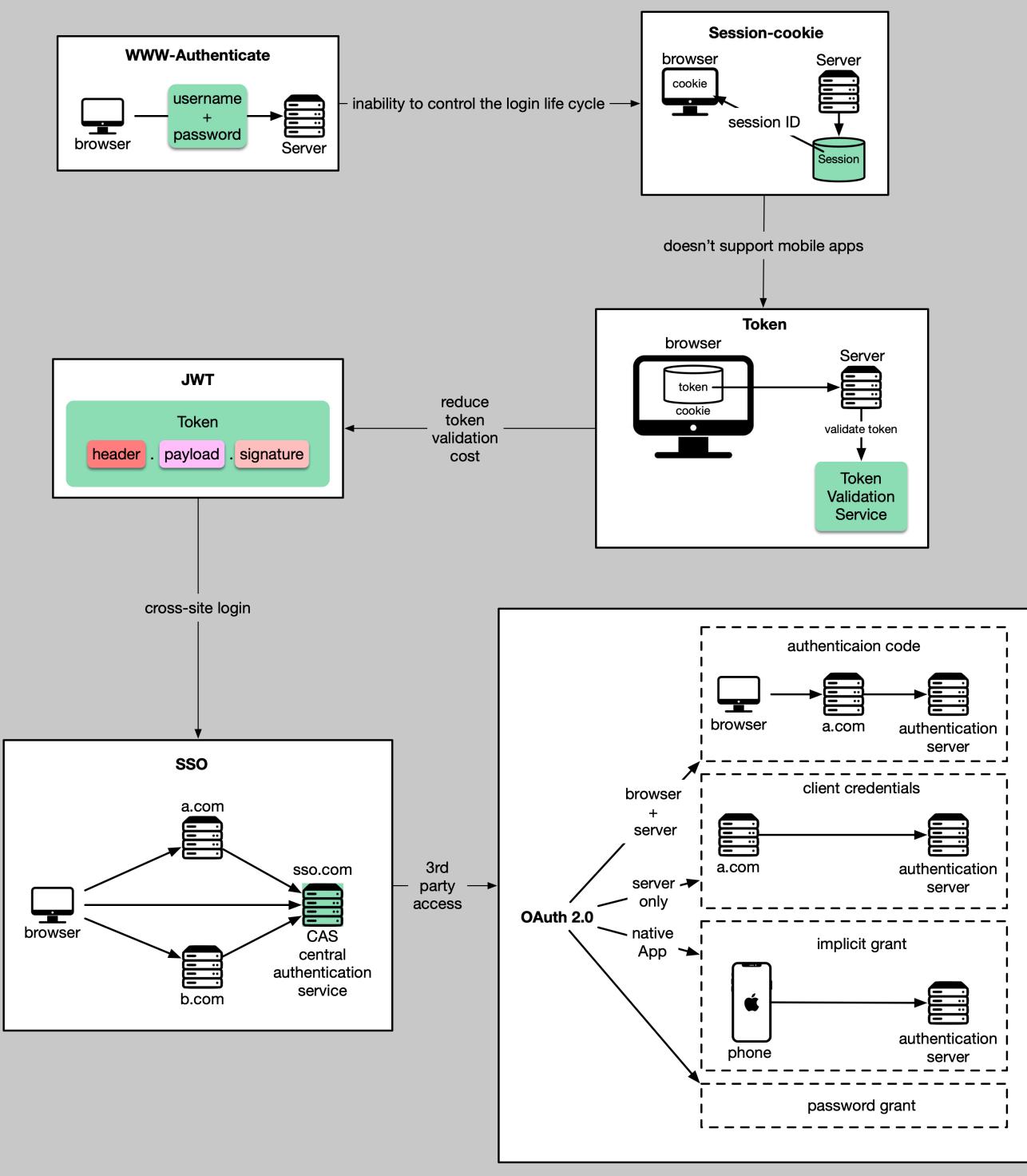
数字证书确保服务器端和客户端之间的安全和加密通信。

4. 凭据:

使用用户身份验证信息来验证并授予对各种系统和服务的访问权限。

会话、Cookie、JWT、令牌、SSO 和 OAuth 2.0 - 它们分别是什么？

这些术语都与用户身份管理相关。当你登录进一个网站时，你要声明自己是谁（身份验证，identification）。你的身份经过验证（认证，authentication），并被授予必要的权限（授权，authorization）。过去，许多解决方案被提出来，而这个解决方案列表还在不断增长中。



由简而繁，下面是我关于用户身份管理的理解：

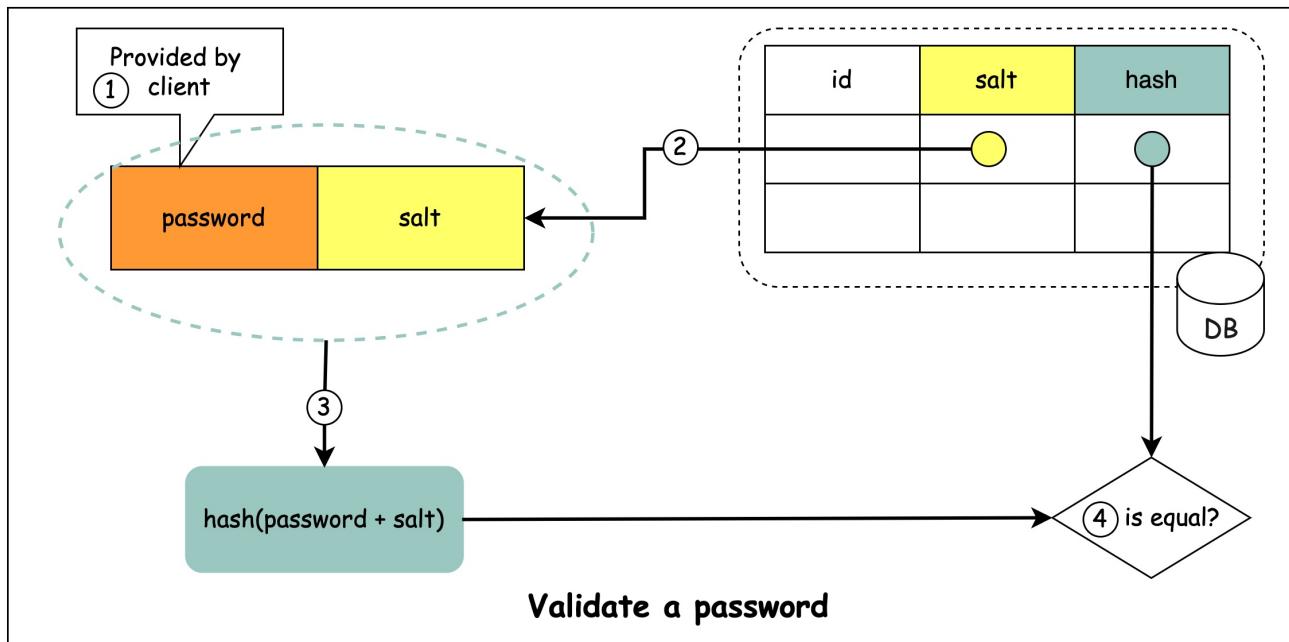
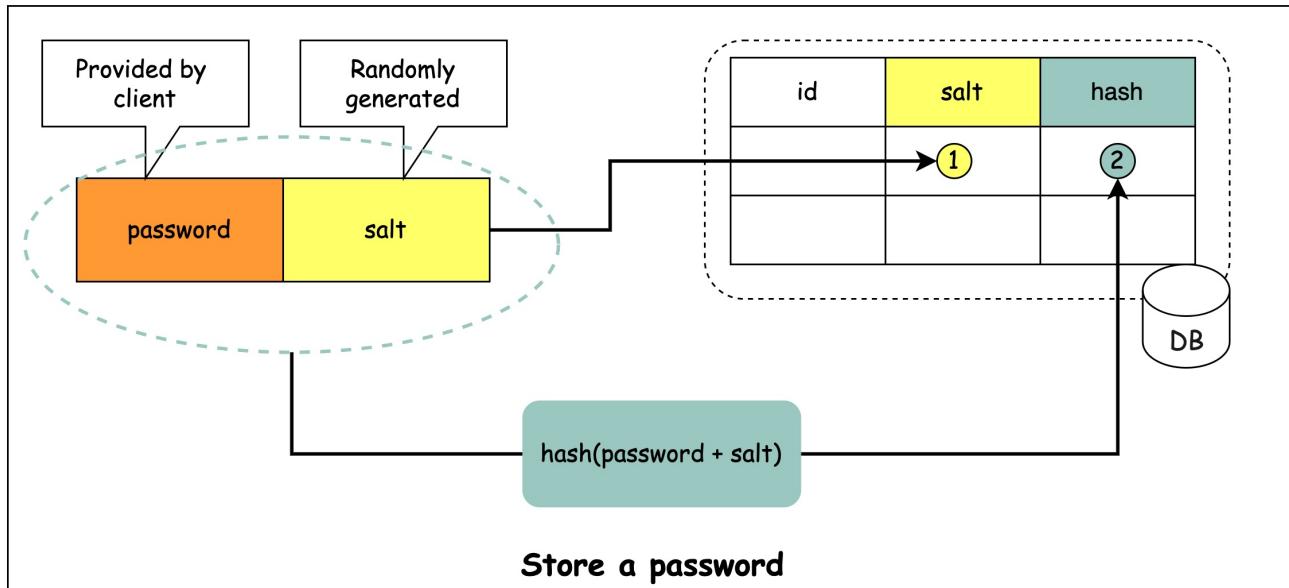
- WWW-Authenticate 是最基本的方法。浏览器要求你提供用户名和密码。由于无法控制登录生命周期，因此现今已经很少使用了。

- 对登录生命周期有更精细的控制的是 session-cookie。服务器维护会话存储，浏览器保留会话 ID。Cookie 通常仅适用于浏览器，不太适用于移动应用。
- 要解决兼容性问题，可以使用令牌。客户端将令牌发送到服务器，然后服务器验证令牌。这种方法的缺点是需要对令牌进行加密和解密，这一过程可能会耗时。
- JWT 是表示令牌的标准方式。由于可以对这些信息进行数字签名，因此它们可以被验证和信任。由于 JWT 包含签名，因此无需在服务器端保存会话信息。
- 通过使用 SSO（单点登录），您只需进行一次登录操作，即可登录到多个网站。它使用CAS（central authentication service，中央认证服务）来维护跨站点信息。
- 通过使用 OAuth 2.0，你可以授权一个网站访问你在另一个网站上的信息。

如何将密码安全地存储到数据库中，以及如何验证密码？

How to store passwords in DB?

 blog.bytebytogo.com



不要做的事

- 明文存储密码并非明智之举，因为任何一个具有内部访问权限的人都可以看到它们。
- 直接存储密码哈希值并不够，因为它容易受到预计算攻击（precomputation）

attack) 的影响，例如彩虹表 (rainbow table)。

- 要缓解预计算攻击，我们使用 salt 来加密密码。

什么是 salt？

根据 OWASP 指南，“salt 是在散列过程中添加到每个密码中的随机生成的唯一字符串”。

如何存储密码和 salt？

1. 哈希结果对于每个密码都是唯一的。
2. 密码可以以以下格式存储在数据库中：hash(password + salt)。

如何验证密码？

要验证一个密码，可以经过以下过程：

1. 客户端输入密码。
2. 系统从数据库中获取对应的 salt。
3. 系统将 salt 附加到密码后，然后对结果进行哈希操作。让我们将哈希过的值称为 H1。
4. 系统对比 H1 和 H2（数据库中存储的哈希值）。如果二者相等，那么密码就是有效的。

向一个十岁小孩解释 JSON Web Token (JWT)

1 {"what": "JSON"}

* A file format to store data in key:value format

Key has to be string
Value can be string, or list of strings or JSON
Nested JSON, [JSON, JSON] or list of JSON
Just a data structure :)

2 JWT Structure 3 parts

Header: { "alg": "HS256", "type": "JWT" }
data: { "key": "foo" }
Signature: Base64 encode of Header + data

3 Working?

1. login: username & password → Server (Validate credentials, Create & sign JWT with secret). Now I don't need to store session info. Light weight implementation.

2. Store JWT locally.

3. Authorization: Bearer JWT → Server (Validates Signature, OK).

4 Signing Alg

1 Public key

Sign JWT with private key → Signed JWT + Signature → JWT consumer (Validate with public key).

* RS256
* ES256 etc

2 Symmetric key

Sign JWT with Shared Key → Signed JWT → JWT consumer (Validate with Shared Key).

* HMAC
* HS256

SecurityZines.com In Collaboration with ByteByteGo

想象一下，你有一个特殊的盒子，叫做 JWT。这个盒子里有三个部分：一个头部、一个数据体和一个签名。

头部就像盒子外部的标签。它告诉我们盒子的类型以及如何保护它。通常以 JSON 的格式（只是一种使用花括号“{}”和冒号“:”组织信息的方式）编写。

数据体就像你想要发送的实际消息或者信息。可以是你的名字、年龄或者任何你想要分享的数

据。它也是以 JSON 格式编写的，因此，易于理解和处理。

而签名则是让 JWT 保持安全的东西。它就像一种特殊的印章，只有发送者知道如何创建它。这个签名是使用一种秘密代码生成的，有点像密码。它确保了没有人能够在发送者不知情的情况下篡改 JWT 的内容。

当你想把 JWT 发送给服务端时，你把头部、数据体和签名放进箱子里。然后把它发送到服务器。服务器可以轻松读取头部和数据体，以了解你是谁，以及你想要做什么。

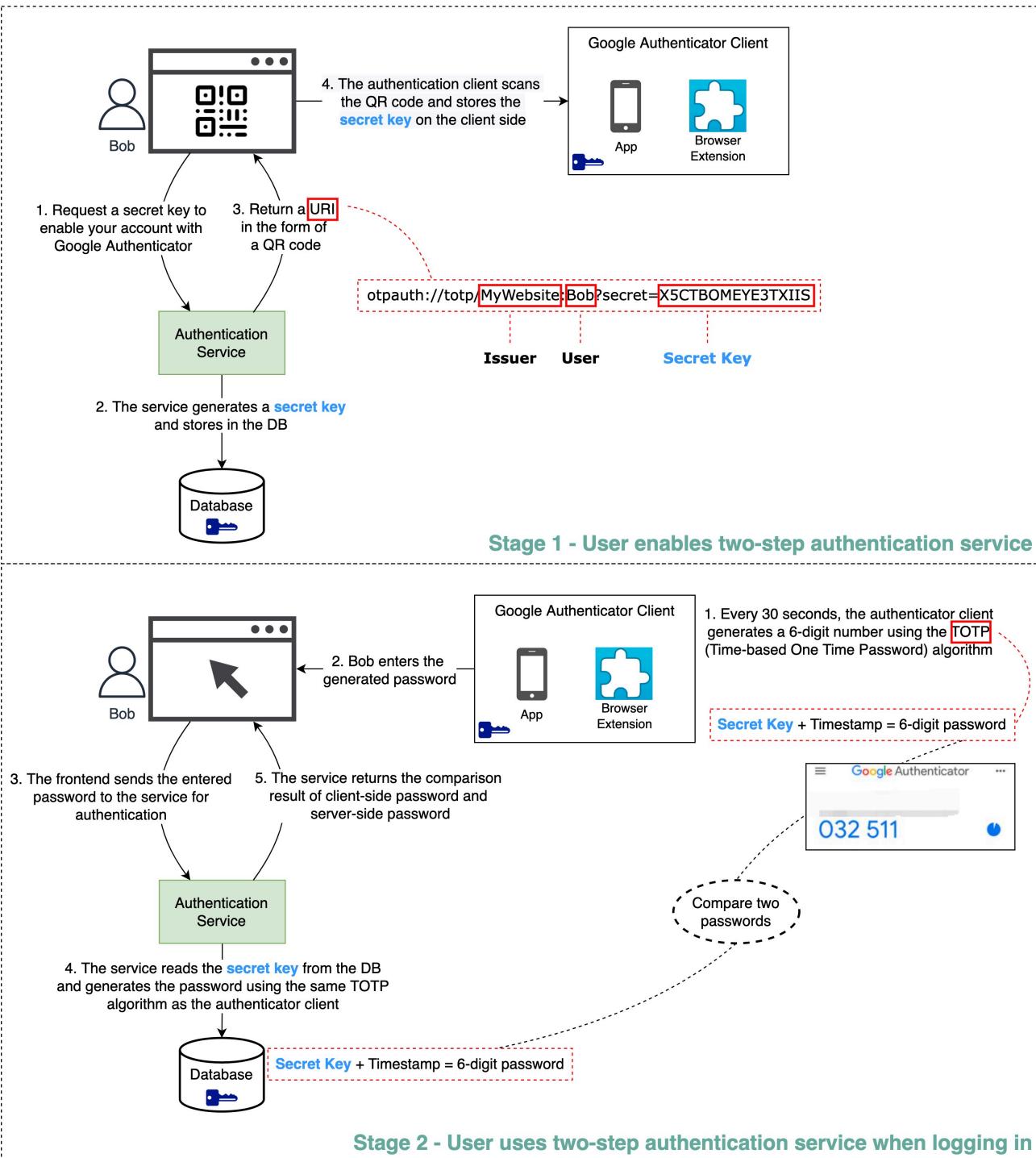
Google Authenticator（或者其它类型的两步认证器）是如何工作的？

Google Authenticator 通常用于在已启用两步验证（2-factor authentication）的情况下登录我们的账户。那么，它是如何保证安全的呢？

Google Authenticator 是一种基于软件的身份验证器，它实现了两步验证服务。下图提供了详细信息。

How does Google Authenticator Work?

 blog.bytebytogo.com



涉及两个阶段：

- 第一阶段 - 用户启用 Google 两步验证。
- 第二阶段 - 用户使用身份验证器进行登录等操作。

让我们看看这两个阶段。

第一阶段

步骤 1 和 2: Bob 打开网页来启用两步验证。前端请求一个密钥。身份验证服务为 Bob 生成密钥并将其存储在数据库中。

步骤 3: 身份验证服务返回一个 URI 给前端。这个 URI 由密钥颁发者、用户名和密钥组成。这个 URI 以二维码的形式显示在网页上。

步骤 4: 然后, Bob 使用 Google Authenticator 扫描生成的二维码。密钥存储在身份验证器中。

第二阶段 步骤 1 和 2: Bob 想要使用 Google 两步验证登录网站。为此, 他需要密码。Google Authenticator 每30秒使用 TOTP (Time-based One Time Password, 基于时间的一次性密码) 算法生成一个6位数的密码。Bob 使用这个生成的密码来登录网站。

步骤 3 和 4: 前端将 Bob 输入的密码发送到后端进行身份验证。身份验证服务从数据库中读取密钥, 然后使用与客户端相同的 TOTP 算法生成一个6位数的密码。

步骤 5: 身份验证服务比较客户端和服务器分别生成的两个密码, 并将比较结果返回给前端。只有在两个密码匹配时, Bob 才能继续登录过程。

这种身份验证机制安全吗?

- 其他人能获取密钥吗?

我们需要确保密钥是使用 HTTPS 传输的。身份验证器客户端和数据库存储密钥, 因此, 我们需要确保这些密钥是加密的。

- 黑客能猜测这个6位数密码吗?

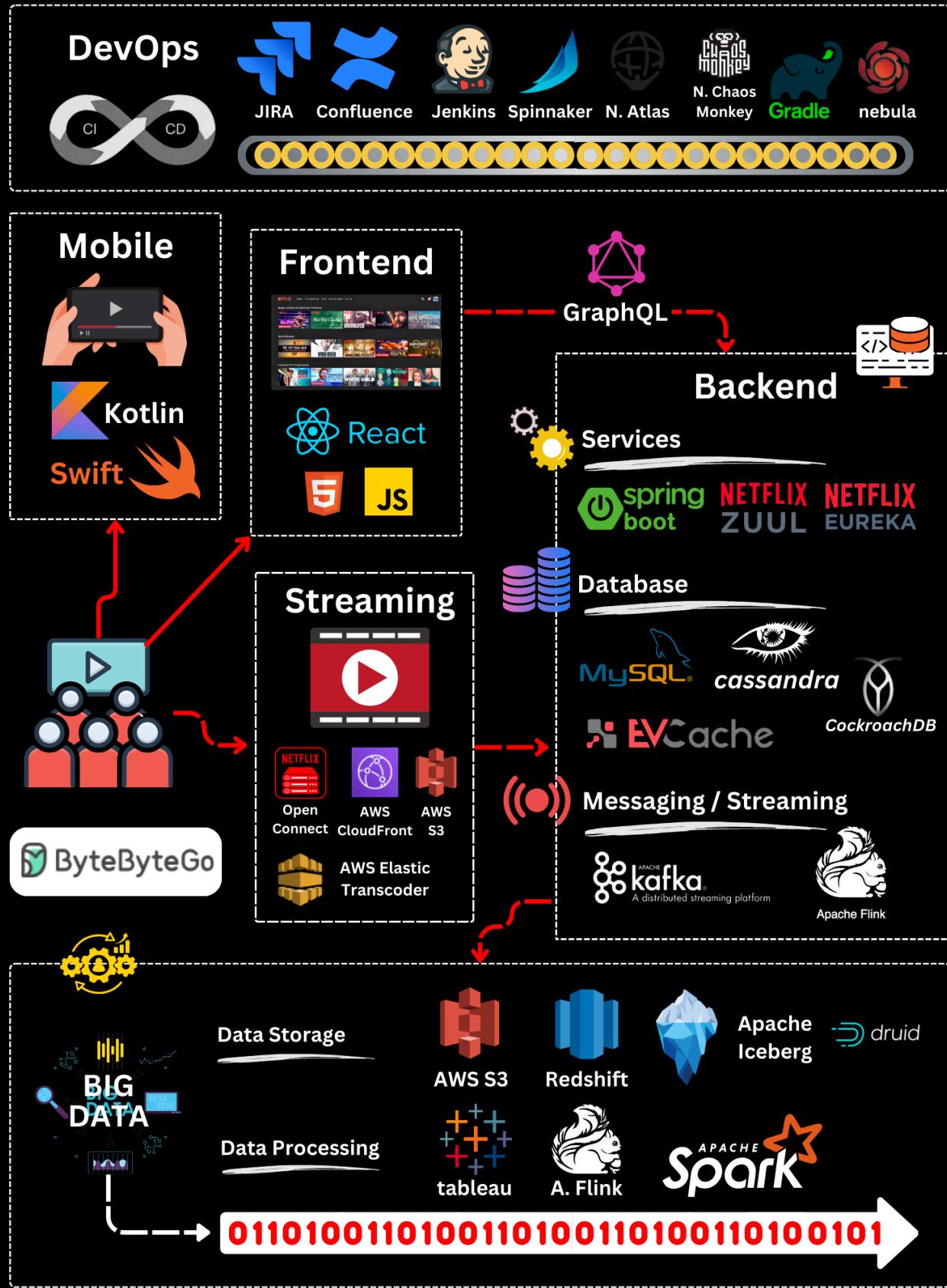
不能。密码有6位数字, 因此, 生成的密码有100万种可能的组合。此外, 密码每30秒更改一次。如果黑客想要在30秒内猜测出密码, 那么他们需要每秒输入30,000个组合。

真实案例学习

Netflix 的技术栈

本文基于许多 Netflix 工程博客和开源项目的研究。若有任何不当之处, 请随时告诉我们。

NETFLIX Tech Stack



移动和 Web: Netflix 采用 Swift 和 Kotlin 来构建原生移动应用。而对于 Web 应用，它使用 React。

前端和服务器之间的通信: Netflix 使用 GraphQL。

后端服务: Netflix 依赖 ZUUL、Eureka、Spring Boot 框架和其他技术。

数据库: Netflix 利用 EV Cache、Cassandra、CockroachDB 和其他数据库。

消息传递/流处理: Netflix 使用 Apache Kafka 和 Flink 进行消息传递和流处理。

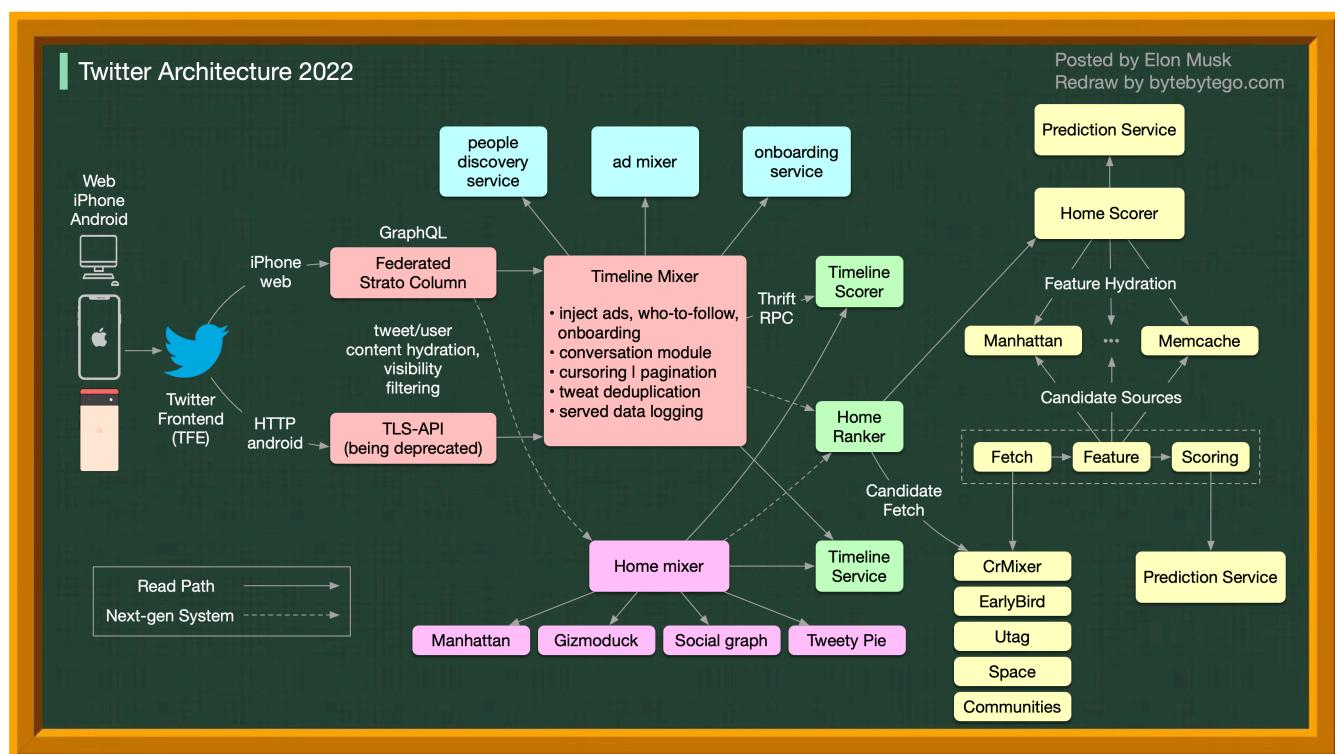
视频存储: Netflix 使用 S3 和 Open Connect 进行视频存储。

数据处理: Netflix 利用 Flink 和 Spark 进行数据处理，然后使用 Tableau 进行可视化。Redshift 被用于结构化数据仓库信息的处理。

CI/CD: 对于 CI/CD 流程，Netflix 使用各种工具，例如 JIRA、Confluence、PagerDuty、Jenkins、Gradle、Chaos Monkey、Spinnaker、Atlas 等。

Twitter 架构 2022

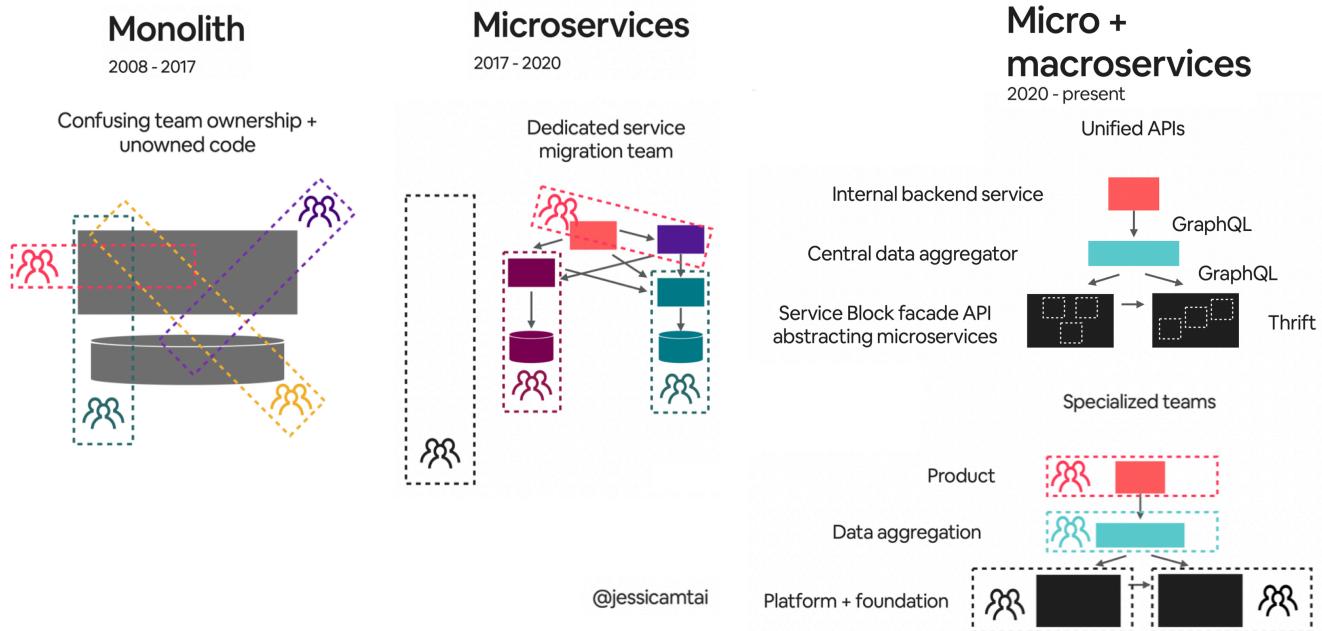
是的，这就是真正的 Twitter 架构。它由 Elon Musk 发布，我们对其进行了重绘以便于更好的阅读。



过去 15 年 Airbnb 微服务架构的演进之路

Airbnb 的微服务架构经历了三个主要阶段。

Airbnb's Microservice Architecture



单体架构 (Monolith) (2008 - 2017)

Airbnb 最初只是一个简单的房东房客市场。它使用 Ruby on Rails 构建 —— 单体架构。

有何挑战？

- 团队所有权混乱 + 无主代码
- 部署慢

微服务 (Microservices) (2017 - 2020)

微服务旨在解决上述挑战。在微服务架构中，关键的服务包含：

- 数据获取服务
- 业务逻辑数据服务
- 写工作流服务
- UI 聚合服务
- 每一个服务都由一个团队专门负责

有何挑战？

对人类来说，数百个服务和依赖是难以管理的。

微服务 + 宏服务 (Micro + macroservices) (2020 - present)

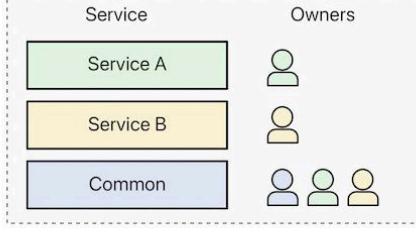
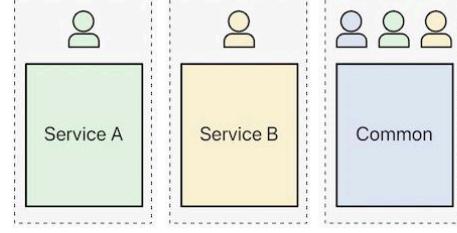
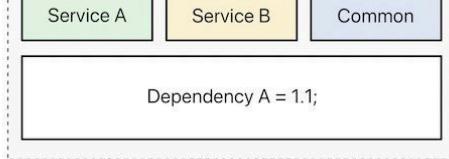
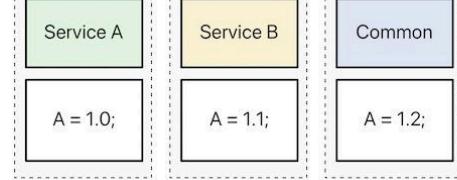
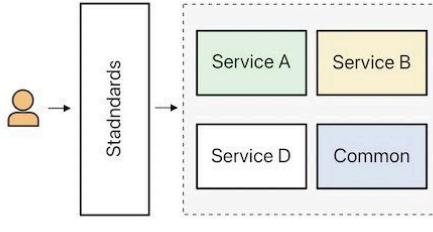
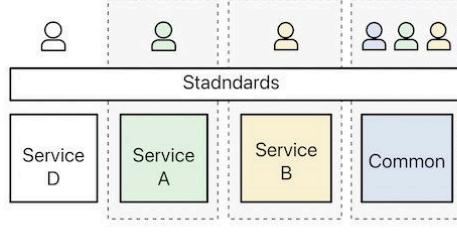
这是 Airbnb 现在正在努力实现的架构。微服务和宏服务混合模型侧重于 API 的统一。

Monorepo vs. Microrepo.

何为最佳之选？为什么不同的公司选择不同的选项？

Monorepo vs Microrepo: which is the best?

 ByteByteGo.com

	Monorepo	Microrepo														
	     	    														
	 <table border="1"> <tr> <td>Service</td> <td>Owners</td> </tr> <tr> <td>Service A</td> <td>1 person</td> </tr> <tr> <td>Service B</td> <td>1 person</td> </tr> <tr> <td>Common</td> <td>3 people</td> </tr> </table> <p>Services work under the same repository.</p>	Service	Owners	Service A	1 person	Service B	1 person	Common	3 people	 <table border="1"> <tr> <td>Service A</td> <td>1 person</td> </tr> <tr> <td>Service B</td> <td>1 person</td> </tr> <tr> <td>Common</td> <td>3 people</td> </tr> </table> <p>Service owners work under separate repositories.</p>	Service A	1 person	Service B	1 person	Common	3 people
Service	Owners															
Service A	1 person															
Service B	1 person															
Common	3 people															
Service A	1 person															
Service B	1 person															
Common	3 people															
	 <table border="1"> <tr> <td>Service A</td> <td>Service B</td> <td>Common</td> </tr> <tr> <td colspan="3">Dependency A = 1.1;</td> </tr> </table> <p>Service share the same dependency.</p>	Service A	Service B	Common	Dependency A = 1.1;			 <table border="1"> <tr> <td>Service A</td> <td>Service B</td> <td>Common</td> </tr> <tr> <td>A = 1.0;</td> <td>A = 1.1;</td> <td>A = 1.2;</td> </tr> </table> <p>Service choose their own dependency.</p>	Service A	Service B	Common	A = 1.0;	A = 1.1;	A = 1.2;		
Service A	Service B	Common														
Dependency A = 1.1;																
Service A	Service B	Common														
A = 1.0;	A = 1.1;	A = 1.2;														
	 <table border="1"> <tr> <td>Standards</td> <td>Service A</td> <td>Service B</td> </tr> <tr> <td></td> <td>Service D</td> <td>Common</td> </tr> </table> <p>Services share the same standard.</p>	Standards	Service A	Service B		Service D	Common	 <table border="1"> <tr> <td>Standards</td> <td>Service D</td> <td>Service A</td> <td>Service B</td> <td>Common</td> </tr> </table> <p>Services set their own standard.</p>	Standards	Service D	Service A	Service B	Common			
Standards	Service A	Service B														
	Service D	Common														
Standards	Service D	Service A	Service B	Common												
	 <table border="1"> <tr> <td>Bazel</td> <td>Buck</td> </tr> <tr> <td>NuGet</td> <td>Lerna</td> </tr> </table>	Bazel	Buck	NuGet	Lerna	 <table border="1"> <tr> <td>Gradle</td> <td>nebulia</td> <td>Maven™</td> </tr> <tr> <td>NPM</td> <td>CMake</td> </tr> </table>	Gradle	nebulia	Maven™	NPM	CMake					
Bazel	Buck															
NuGet	Lerna															
Gradle	nebulia	Maven™														
NPM	CMake															

Monorepo 并不是一个新东西；Linux 和 Windows 都是用 Monorepo 创建的。为了提升可伸缩性和构建速度，谷歌开发了内部专用的工具链，以便更快地对其进行扩展，并制定了严格地编码质量标准，以保持代码一致性。

Amazon 和 Netflix 是微服务理念的主要簇拥者。这种方法自然地将服务代码分开存储在不同的仓库中。它能更快地扩展，但后期可能会导致治理方面的问题。

在 Monorepo 中，每个服务都是一个文件夹，每一个文件夹都有一个 BUILD 配置和 OWNERS 权限控制。每一个服务成员负责他们自己的文件夹。

另一方面，在 Microrepo 中，每一个服务负责它们自己的仓库，而构建配置和权限通常是为整个仓库设置的。

在 Monorepo 中，无论业务是什么，整个代码库都共享依赖项，因此，当有版本升级时，每一个代码库都会升级它们的版本。

在 Microrepo 中，每一个仓库控制自己的依赖项。业务根据自己的安排选择何时升级版本。

Monorepo 有提交标准。Google 的代码评审以其高标准而闻名，确保 Monorepo 具有一致的质量标准，而无论业务是什么。

Microrepo 可以设置自己的标准，也可以通过采纳最佳实践来使用共享标准。它可以更快地为业务进行扩展，但代码质量可能会有所不同。Google 工程师构建了 Bazel，而 Meta 构建了 Buck。还可以用一些其他的开源工具，包括 Nx、Lerna 等。

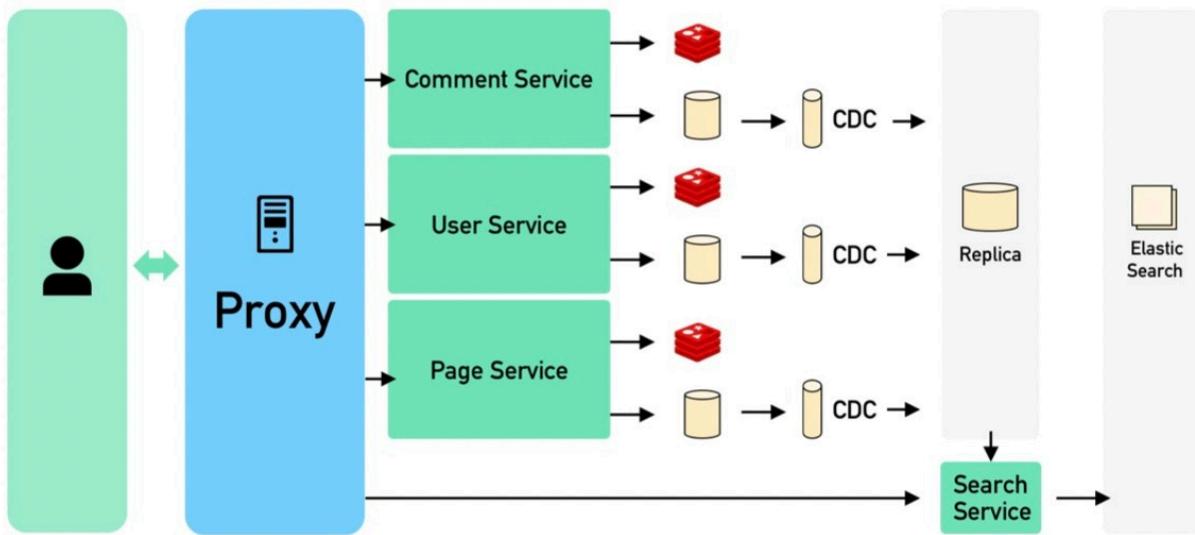
多年以来，Microrepo 拥有了更多支持工具，其中包括 Java 的 Maven 和 Gradle、NodeJS 的 NPM 以及 C/C++ 的 CMake for C/C++。

如果是你，你要如何设计 Stack Overflow 网站？

如果你的答案是本地服务器（on-premise server）和单体架构（位于下图底部），那么你有可能面试失败，但这正是它现实中的构建方式！

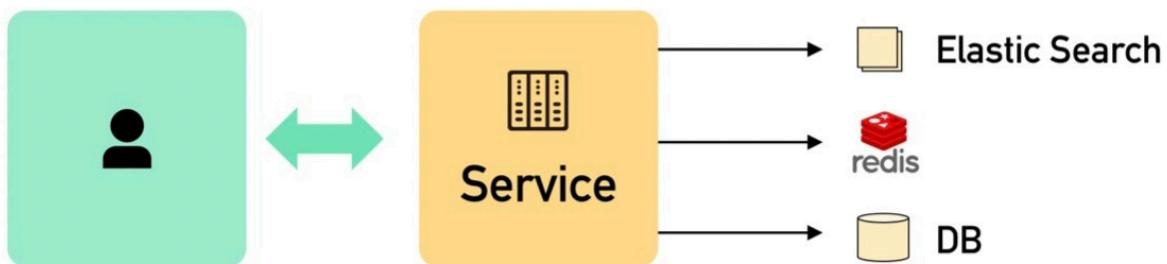
What people think it looks like

1. Microservice based
2. Event sourcing (CQRS)
3. Eventual consistency
4. Sharding
5. Heavy use cache
6. ...



What it actually is

1. Monolithic
2. Only 9 web servers



人们觉得它应该是什么样子的呢

面试官可能期待类似于上图顶部那样子的回答。

- 使用微服务来将系统分解为小组件。
- 每个服务都有自己的数据库。重度使用缓存。
- 服务是分片的。

- 服务之间通过消息队列异步通信。
- 服务是使用 CQRS (Command Query Responsibility Segregation) 和 Event Sourcing 来实现的。
- 展示分布式系统方面的知识，例如最终一致性 (eventual consistency) 、CAP 定理等。

实际上呢

Stack Overflow 仅用九台本地服务器即可满足所有流量需求，而且它是单体的！它拥有自己的服务器并且并没有在云上运行。

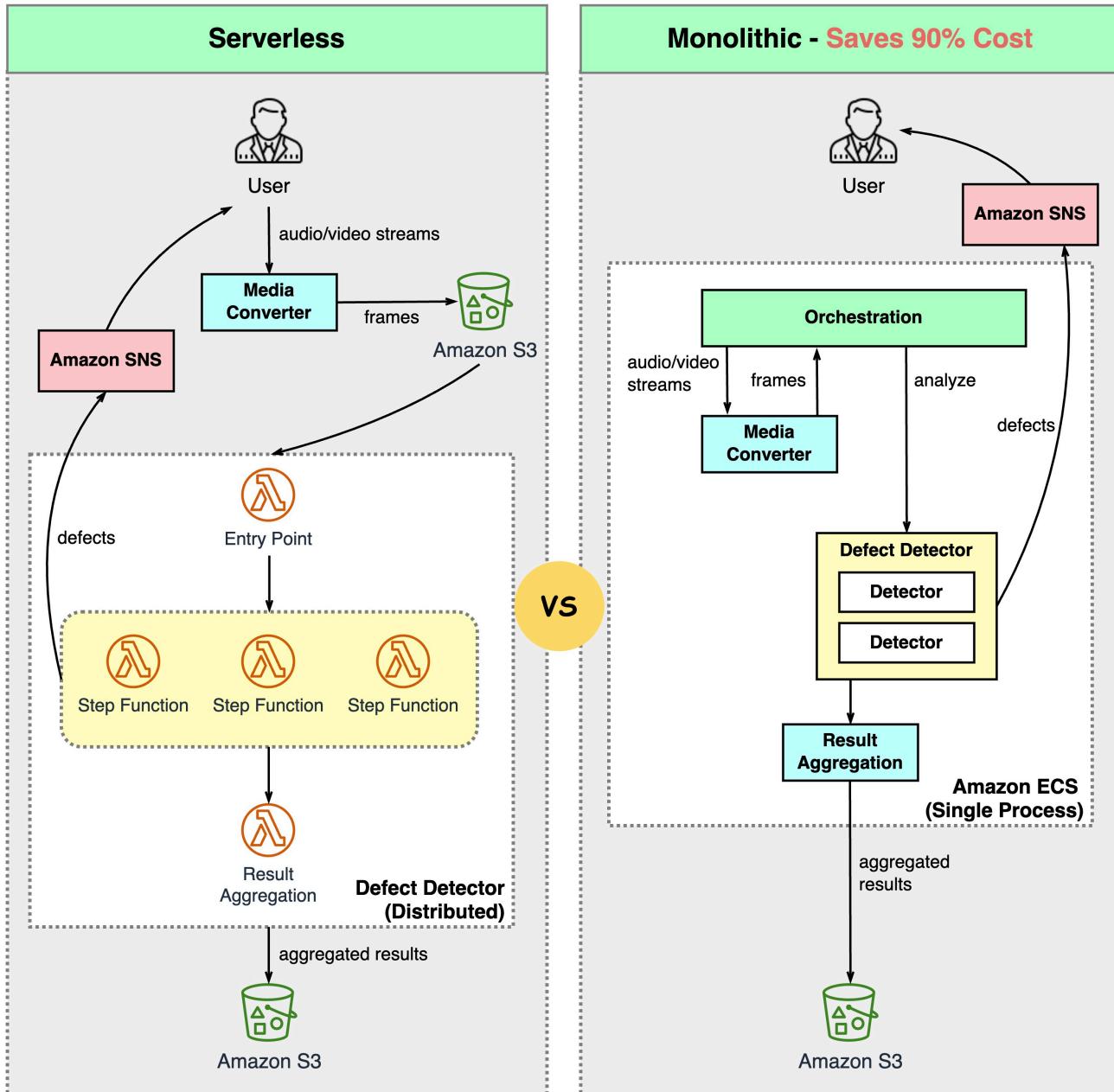
这与我们当下所有流行的信念背道而驰。

为什么 Amazon Prime Video 监控从无服务 (Serverless) 转向了单体架 (Monolithic) ？它是怎样节省九成成本的呢？

下图是迁移前后的架构对比。

Amazon Prime Video monitoring - From Serverless to Monolithic

 blog.bytebytego.com



Based on: <https://primevideotech.com/>

Amazon Prime Video 监控服务是什么？

Prime Video 服务需要监控数千个直播流的质量。这个监控工具自动实时分析流，识别诸如块损坏、视频冻结和同步问题这样的质量问题。这对于提高客户满意度来说，是一个重要的过程。

共有三步：媒体转换器（media converter）、缺陷检测器（defect detector）和实时通知（real-time notification）。

- 旧架构有什么问题？

旧架构基于 Amazon Lambda，适用于快速构建服务。然而，当大规模运行此架构时，就成本而言，并不划算。其中两个最贵的操作是：

1. 编排工作流 —— AWS Step Functions 按状态转换收费，而编排每秒会执行多次状态转换。
 2. 在分布式组件之间传递数据 —— 中间数据存储在 Amazon S3 中，以便下一个阶段可以下载。当数据量很大的时候，下载操作可能很昂贵。
- 单体架构节约了九成成本

单体架构旨在解决成本问题。在此架构中，仍然有 3 个组件，但是，媒体转换器和缺陷检测器部署在同一进程中，从而节省了通过网络传递数据所带来的开销。令人惊讶的是，这种部署架构的变更节省了九成成本！

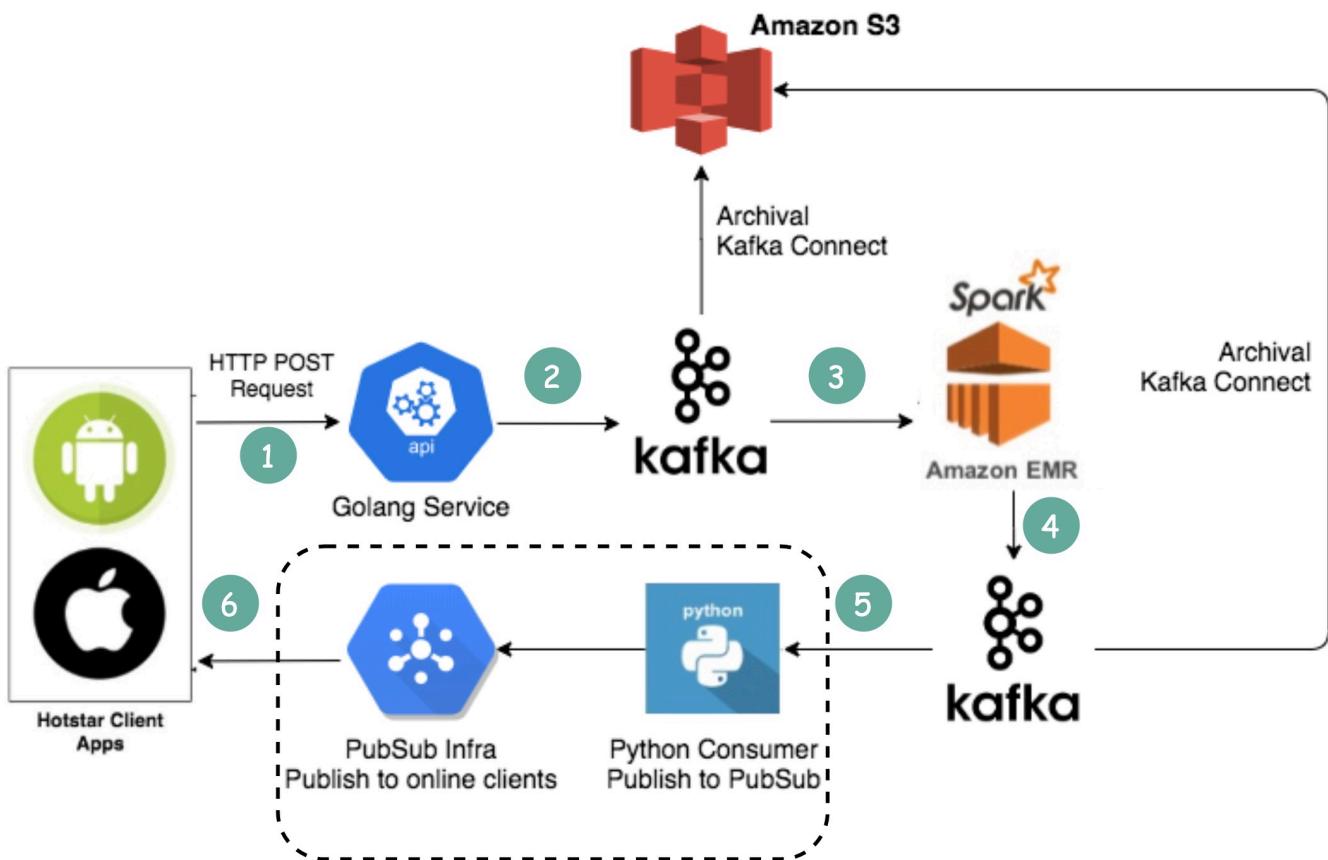
这是一个有趣且独一无二的案例研究，因为微服务已经成为了科技行业的时尚首选。很高兴看到我们对架构的演变进行了更多的讨论，并且对其利弊也进行了更加诚实的讨论。将组件分解为分布式微服务是有成本的。

- Amazon 的领导人对此有何评论？

Amazon 的 CTO Werner Vogel：“构建可进化的软件系统是一种策略，而不是一种宗教。以开放的心态审视你的架构是必须的。”

前 Amazon 可持续性副总裁 Adrian Cockcroft：“Prime Video 团队走的是我称之为无服务优先（Serverless First）的道路…我并不主张仅无服务（Serverless Only）”。

Disney Hotstar 是如何在锦标赛期间捕获 50 亿个表情符号的？

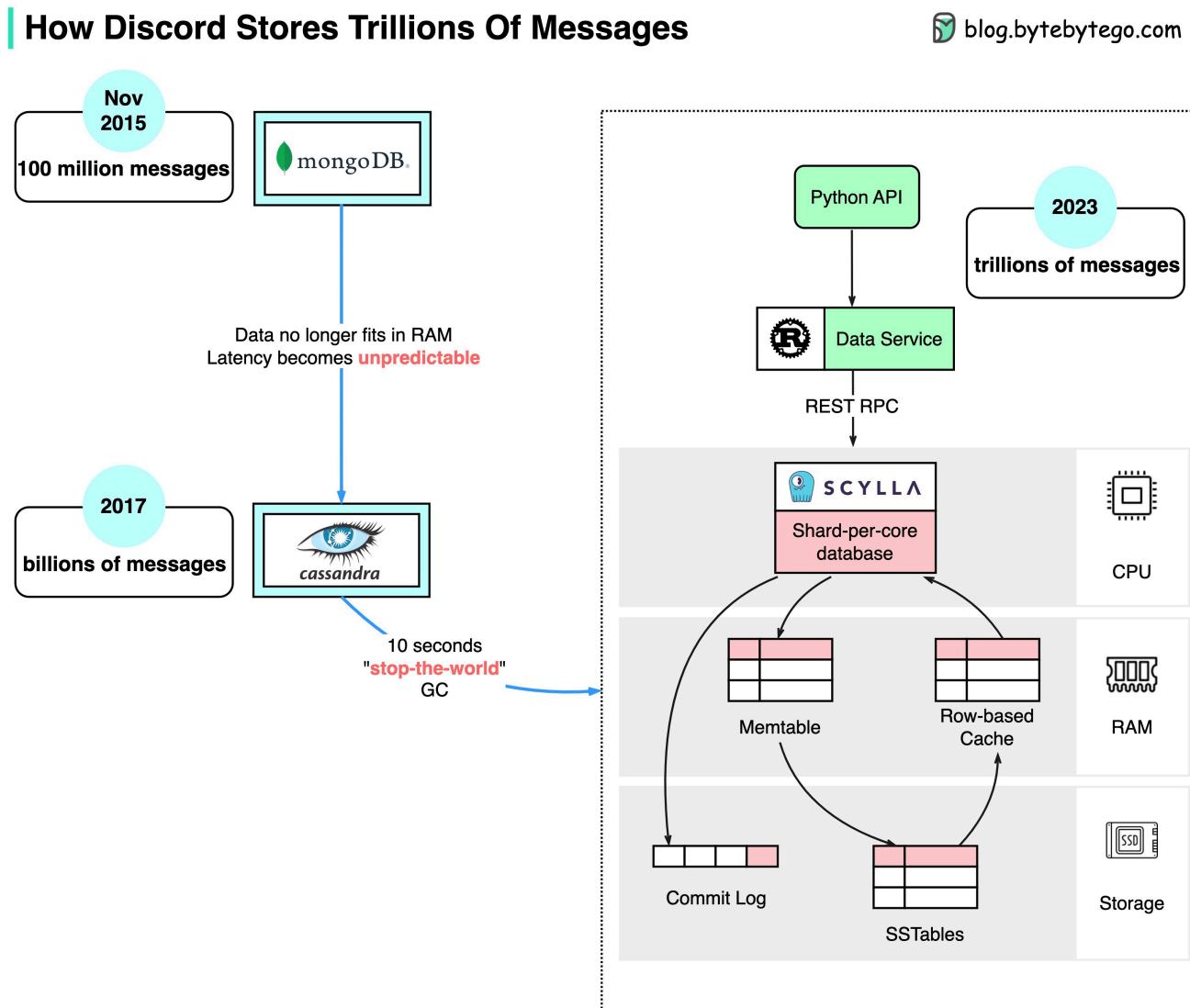


1. 客户端通过标准的 HTTP 请求发送表情符号。你可以将 Golang 服务看成一个典型的 Web 服务器。选择 Golang 是因为它很好的支持了并发。线程在 Golang 中是轻量的。
2. 由于写入量非常大，Kafka（消息队列）被用作缓冲区。
3. 表情符号数据由一个名为 Spark 的流处理服务来进行聚合。它每 2 秒聚合一次数据，这个时间是可配置的。根据时间间隔需要进行权衡。较短的间隔意味着，表情符号将更快地被传递给其他客户端，但同时也意味着需要更多的计算资源。
4. 将被聚合的数据写入到另一个 Kafka。
5. PubSub 消费者从 Kafka 拉取聚合的表情符号数据。
6. 通过 PubSub 基础设施，表情符号被实时传递给其他客户端。PubSub 基础设施很有意思。Hotstar 考虑了以下协议：Socketio、NATS、MQTT 和 gRPC，最终选择了 MQTT。

LinkedIn 也采用了类似的设计，每秒流式传输一百万个赞。

Discord 是怎样存储数万亿条消息的

下图显示了 Discord 的消息存储的演进之路：



MongoDB → Cassandra → ScyllaDB

在 2015 年，Discord 的第一个版本建立在单个 MongoDB 副本之上。到了 2015 年 11 月左右，MongoDB 已经存储了 1 亿条数据，此时，RAM 再也无法容纳数据和索引了。延迟变得不可预测。消息存储需要移到另一个数据库。Cassandra 被选中。

在 2017 年，Discord 拥有 12 个 Cassandra 节点，存储了数十亿条消息。

到了 2022 年初，它拥有 177 个节点，存储了数万亿条消息。此时，延迟再次不可预测，而维护操作也成本过高以致无法进行。

这个问题的出现有几个原因：

- Cassandra 的内部数据结构使用了 LSM 树。读取比写入更昂贵。在具有数百用户的服务器上可能会有许多并发读取操作，从而导致热点问题。
- 维护集群，例如紧凑的 SSTable，这会影响性能。
- 垃圾回收暂停会导致显著的延迟波动

ScyllaDB 是一种兼容 Cassandra 的数据库，用 C++ 编写。Discord 重新设计了其架构，使其具有一个单体 API，一个用 Rust 编写的数据服务以及基于 ScyllaDB 的存储。

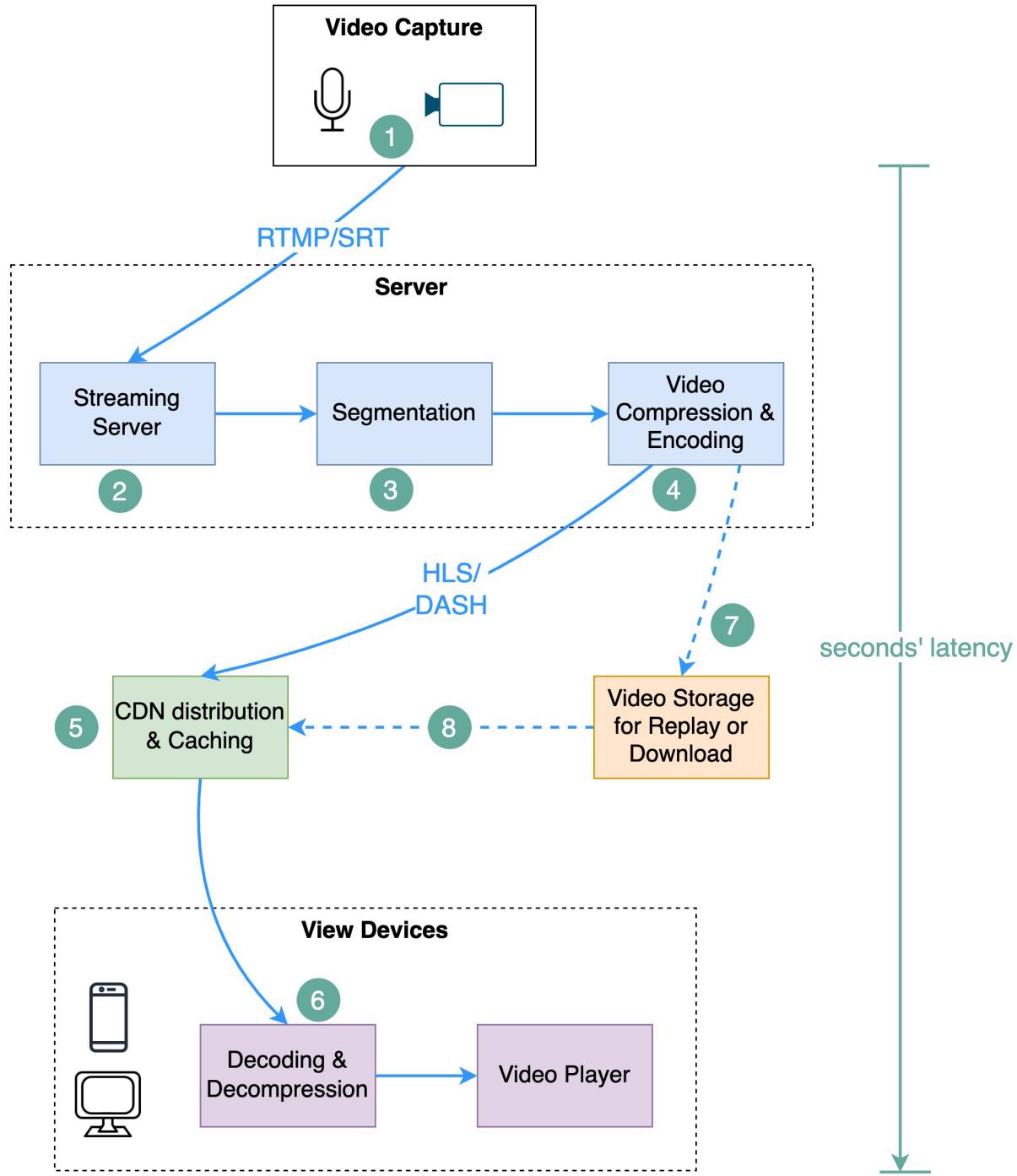
ScyllaDB 中的 p99 读取延迟为 15 毫秒，而 Cassandra 则为 40-125 毫秒。p99 写入延迟为 5 毫秒，而 Cassandra 则为 5-70 毫秒。

YouTube、TikTok Live 或 Twitch 上的视频直播是如何工作的呢？

直播与常规的流媒体不同，因为直播的视频内容是通过互联网实时传输的，通常延迟只有几秒钟。

下图解释了使其成为可能的幕后工作流。

How does Live Streaming Work? blog.bytebytego.com



步骤 1：麦克风和摄像头捕获原始视频数据。数据稍后被发送到服务器端。

步骤 2：视频数据被压缩和编码。例如，压缩算法将背景与其他视频元素分离。压缩后，视频被编码为诸如 H.264 之类的标准。在此步骤之后，视频数据的大小会小得多。

步骤 3：编码数据被分成更小的段，每一段长度通常为几秒，因此，大大缩短了下载或者流式传输的时间。

步骤 4：发送分段数据到流媒体服务器。流媒体服务器需要支持不同的设备和网络条件。这称为“自适应比特率流式传输（Adaptive Bitrate Streaming）”。这意味着我们在步骤2和步骤3中需要生成具有不同比特率的多个文件。

步骤 5：推送直播流数据到由 CDN（Content Delivery Network，内容传输网络）支持的边缘服务器（edge server）。数百万观众可以从附近的边缘服务器观看视频。CDN 显著降低了数据传输延迟。

步骤 6：观众的设备解码和解压视频数据，并在视频播放器中播放视频。

步骤 7 和 8：如果需要存储视频以供重播，那么编码数据会被发送到存储服务器（storage server），观众可以稍后从中请求重播。

直播流媒体的标准协议包括：

- RTMP（Real-Time Messaging Protocol，实时消息传输协议）：该协议最初由 Macromedia 开发，用于在 Flash 播放器和服务器之间传输数据。现在，它用来通过互联网传输视频数据。注意，视频会议应用程序（如Skype）使用 RTC（Real-Time Communication，实时通信）协议来获得更低的延迟。
- HLS（HTTP Live Streaming，HTTP实时流传输）：它要求 H.264 或 H.265 编码。Apple 设备仅接受 HLS 格式。
- DASH（Dynamic Adaptive Streaming over HTTP，基于HTTP的动态自适应流传输）：DASH不支持Apple设备。
- HLS 和 DASH 都支持自适应比特率流传输。

许可

This work is licensed under CC BY-NC-ND 4.0 