15个系统设计权衡关键点：构建高性能系统的黄金法则

360_go_php 2025-01-16 175 阅读6分钟 专栏：后端与系统架构

智能总结 复制 重新生成

这篇文章介绍了构建高性能系统的 15 个设计权衡关键点，包括横向与纵向扩展、数据一致性与可用性、同步与异步通信等，分析了每个权衡点的适用场景、优缺点，强调根据业务需求等做出最优决策以实现高效系统设计。

关联问题：单体应用如何扩展 微服务如何管理 怎样选择加密机制

基于该文章内容继续向AI提问

在系统设计中，性能是一个关键的考量因素，尤其是在面对大规模用户、复杂业务需求或高吞吐量的场景时。以下是构建高性能系统时，15个常见的设计权衡关键点，这些法则帮助架构师做出合理的决策，从而打造出高效、可扩展的系统：

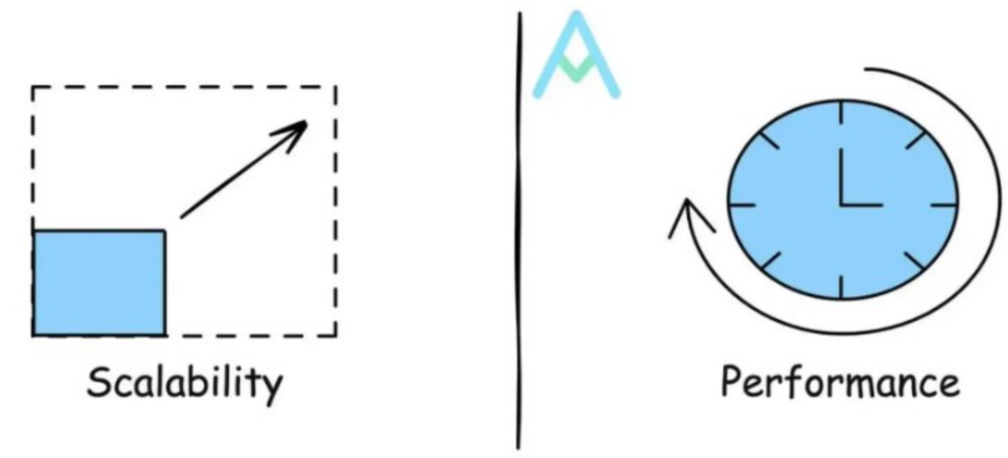
1. 横向扩展 vs. 纵向扩展

- **横向扩展**（Scale-out）是指通过增加更多的机器来扩展系统处理能力，通常适用于分布式系统。
- **纵向扩展**（Scale-up）是指增加单一机器的资源（如 CPU、内存等）来提升性能。纵向扩展成本较高，受物理硬件限制，通常适用于短期扩展或资源较为集中型的应用。

权衡点：横向扩展适合大规模系统，而纵向扩展适合一些小规模、对单一节点性能要求高的应用。

1、可扩展性与性能

可扩展性指的是系统在用户数量或数据量增长时，能否轻松扩展以处理更多的流量和数据。
性能则是指系统的响应速度，“对于用户的请求，系统能在多长时间内返回正确的响应给用户？”



向系统添加更多机器可以使其更具可扩展性，但是管理这些机器和协调任务的复杂性可能会导致性能的降低。
因此这两个指标通常相互矛盾，改进其中一个可能会影响另一个，因此需要您做出对系统最有利的选项，是牺牲一些性能以换取更好的扩展性？还是降低系统的可扩展性以换取系统更好的性能？这取决于具体的业务和系统。
例如，对于高频交易系统，要求极低的延迟，交易速度直接关系到利润，因此这些系统必须在毫秒级别内完成交易指令的处理，为了性能，可能会使用专用的硬件和网络基础设施，而在扩展性方面做出牺牲。
而对于Web应用、社交媒体平台、电商网站等大型互联网服务，这些系统需要能够轻松扩展，以应对突发的流量增长（例如淘宝双十一）。因此为了提高系统的可扩展性，可能会在不太重要的接口上牺牲一些性能。

2. 数据一致性 vs. 可用性

- 依据 **CAP 定理**，系统在分布式架构中面临数据一致性（Consistency）和可用性（Availability）之间的权衡。在选择时要评估业务需求，是否能容忍一定的延迟或不一致性。

360_go_php LV.4
后端资深技术专家 @360奇虎
作者榜No.8
148 文章 45k 阅读 123 粉丝
关注 私信

目录 收起

9. 事务处理 vs. 异步任务
10. 数据冗余 vs. 数据去重
11. 高并发 vs. 低延迟
12. 监控与日志 vs. 性能优化
13. 加密 vs. 性能
14. 短连接 vs. 长连接
15. 容错与容灾设计
总结：

相关推荐
在 Go语言中一个字段可以包含多种类型...
131阅读 · 0点赞
设计支持10W QPS的会员系统，如何做...
342阅读 · 4点赞
解决Cron定时任务中Pytest脚本无法发...
50阅读 · 1点赞
C# 对newlifex X 组件资源池的学习
44阅读 · 1点赞
gozero实现对接在线签名签署文件流程...
84阅读 · 3点赞

精选内容
排序算法稳定性解析与Java实现分析
Asthenia0412 · 27阅读 · 1点赞
链式数据分析引擎解析
冒泡的肥皂 · 22阅读 · 2点赞
电商秒杀场景下的布隆过滤器与布谷鸟...
Asthenia0412 · 51阅读 · 0点赞
基于SpringBoot养老院平台系统功能实...
115432031q · 52阅读 · 0点赞
基于SpringBoot养老院平台系统功能实...
115432031q · 22阅读 · 0点赞

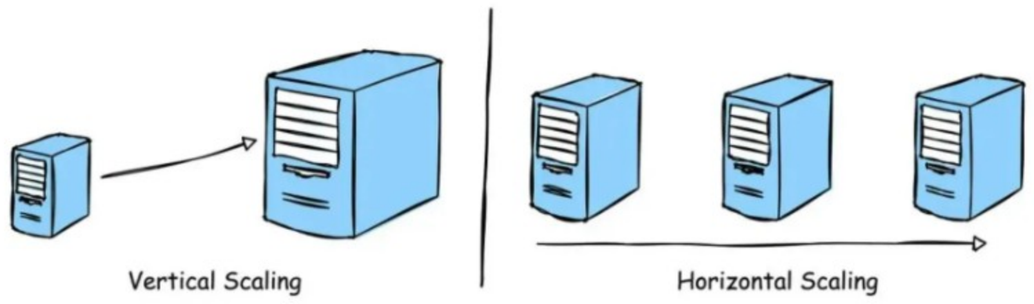
3. 同步 vs. 异步通信
4. 缓存 vs. 数据库查询
5. 强一致性 vs. 最终一致性
6. API 设计：REST vs. GraphQL
7. 负载均衡 vs. 服务器健康检查
8. 单体应用 vs. 微服务架构
9. 事务处理 vs. 异步任务

相关推荐
在 Go语言中一个字段可以包含多种类型...

权衡点：如果系统必须确保数据的一致性，可能牺牲可用性，反之亦然。

2、垂直扩展与水平扩展

垂直扩展指的是通过向现有服务器添加更多或性能更好的资源（如CPU、内存）来提升系统性能。
而**水平扩展**则是通过增加更多服务器来分担系统负载。



垂直扩展更简单，但单台计算机再怎么升级也是存在上限的，而且存在单点故障问题。如果机器发生故障，可能会导致整个系统都不可用。水平扩展允许几乎无限的扩展，但同时带来了管理分布式系统的复杂性。
初创公司可能会通过增加CPU和RAM来垂直扩展其服务器以应对增加的负载。但是随着公司业务的发展，后续一般会转向水平扩展，通过增加服务器来分散负载，这其中可能会需要将当前的服务架构进行重构，这也是为什么初期不采用水平扩展的原因之一。

编辑

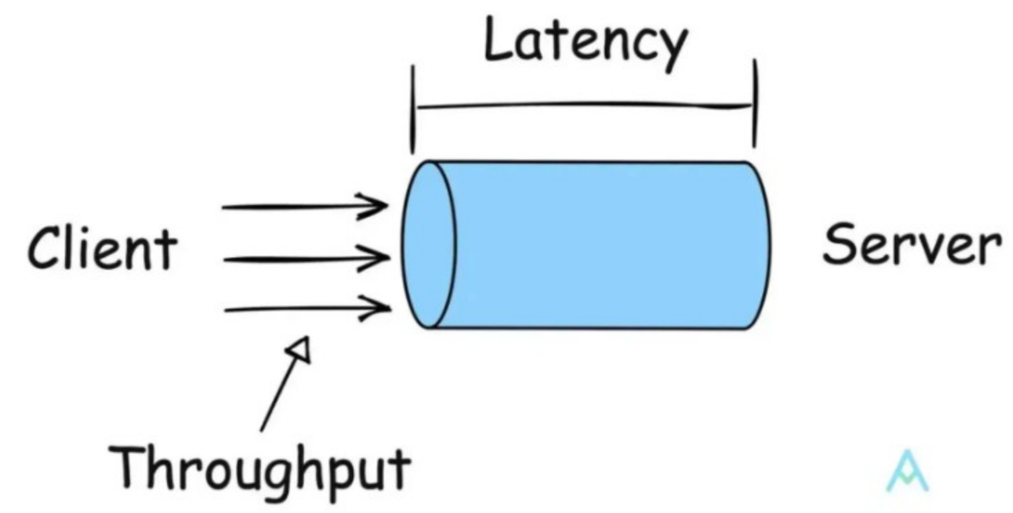
3. 同步 vs. 异步通信

- **同步通信**需要等待响应，通常适用于对实时性要求高的操作。
- **异步通信**则不需要等待响应，适用于高并发、需要解耦的场景，能提高系统的吞吐量和响应速度。

权衡点：同步通信简单且易于理解，但可能造成系统瓶颈。异步通信提高了系统的并发性，但增加了复杂性和延迟。

3、延迟与吞吐量

延迟是指从一个请求发出到收到响应所花费的时间，通常以毫秒（ms）为单位。延迟反映了系统对单个请求的响应速度。延迟越低，响应速度越快。
吞吐量是指在一定时间内系统能够处理的请求数量，通常以“每秒请求数”（requests per second, RPS）或“每秒事务数”（transactions per second, TPS）为单位。假设一个网站服务器每秒能够处理100个用户请求，那么这个服务器的吞吐量就是100 RPS。吞吐量反映了系统的整体处理能力。吞吐量越高，系统在单位时间内处理的请求就越多。



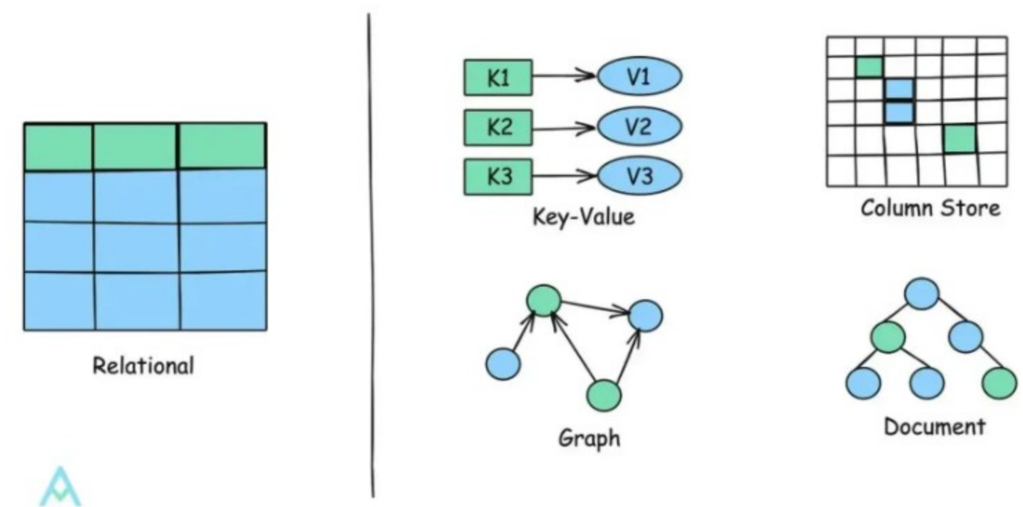
延迟和吞吐量通常是互相影响的。系统在高吞吐量下可能会增加延迟，因为处理大量请求会导致资源（如CPU、内存、网络带宽）被更多地占用，从而增加每个请求的响应时间。
低延迟系统对于实时应用至关重要，如果延迟太高，用户体验会不好，因为用户发出请求后需要等待很长时间才能拿到结果，例如，在线游戏要求低延迟，以确保玩家之间的互动是实时的。
对于视频流媒体服务（优酷、抖音），吞吐量则是关键。因为在高峰时段，可能有数千万用户同时观看视频。高吞吐量能够确保所有用户都能流畅播放视频，而不会出现卡顿。

4. 缓存 vs. 数据库查询

- 使用 **缓存**（如 Redis）能够极大提升系统读取性能，减少数据库的负载。
- 数据库查询保证了数据的一致性，但在高并发情况下可能成为瓶颈。

权衡点：可以使用缓存优化查询性能，但需要确保缓存失效和更新策略的正确性。

4、关系数据库 与 NoSQL 数据库



关系数据库建立在关系模型之上，将数据组织为由行和列构成的表格，并通过唯一的键（主键）来标识每一行。这些数据库结构严谨，提供强大的查询语言，非常适合处理复杂的查询和事务。例如，在一个电商平台上，用户信息可以存储在“用户”表中，订单信息可以存储在“订单”表中，这些表之间通过用户ID（主键）建立关系，来关联用户与其订单。
经常使用的MySQL就是一种关系数据库。然而，关系数据库在水平扩展上可能具有挑战性。
NoSQL（非关系数据库） 是一类不依赖固定表格模式的数据库，用于存储和处理大量的非结构化或半结构化数据。相对关系型数据库，NoSQL 数据库提供了更大的灵活性，并且更容易扩展，但通常会在查询功能和ACID事务上做出一些妥协。
NoSQL 遵循 BASE 属性，即基本可用、软状态、最终一致性……例如，它可能无法始终满足一致性，但最终会变得一致。因此，它牺牲了 ACID 属性，但并非完全牺牲！
常见的NoSQL数据库包括MongoDB、Cassandra、Redis和Couchbase。
在社交媒体应用中，用户生成的帖子、评论和点赞数据可以存储在一个NoSQL数据库中，如MongoDB，每个用户的动态数据可以以文档形式存储，不必像关系型数据库那样进行严格的结构定义。

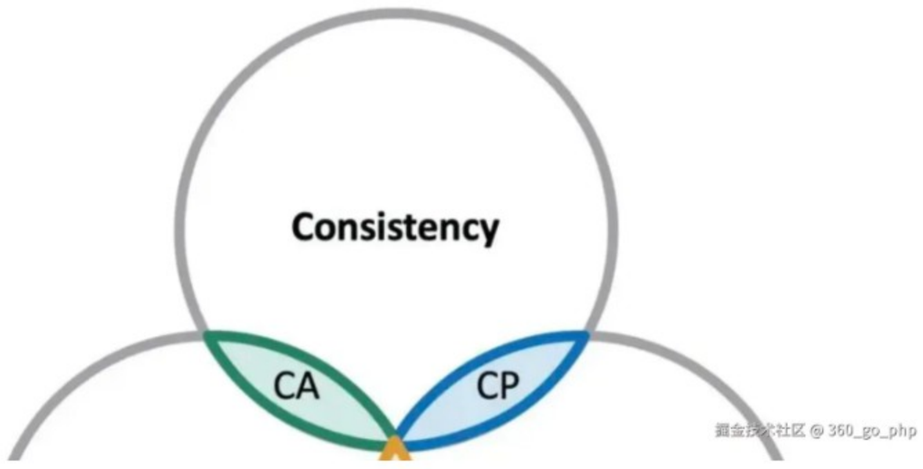
5. 强一致性 vs. 最终一致性

- 强一致性要求系统中的所有副本都在同一时间保持一致，适用于对数据准确性要求严格的场景。
- 最终一致性则允许一定程度的暂时不一致，系统最终会趋于一致，适用于可容忍短期不一致的系统。

权衡点：选择最终一致性能提高系统的可用性和性能，但牺牲了部分一致性。

5、一致性与可用性（CAP定理）

一致性指的是在一个分布式系统中，所有节点在同一时刻看到的数据是相同的。换句话说，当一个数据更新后，所有访问该数据的用户都会看到相同的最新值。
例如在金融系统中，每一笔交易都必须准确无误地反映在账户余额中，不能出现任何差错。当用户进行转账操作后，账户余额必须立即更新并一致地显示在所有系统中。
可用性就是确保系统始终正常运行，即使系统的某些部分出现问题。
例如在一个电商网站上购物，即使这个网站的部分服务出现问题，依然能够浏览商品、下单和付款，这就是系统的可用性。
根据CAP理论，在分布式系统中，只能保证一致性、可用性和分区容忍性中的两个。



6. API 设计：REST vs. GraphQL

- **REST** API简单、易于实现，适用于大多数应用场景，但可能在复杂查询时效率较低。
- **GraphQL** 允许客户端指定请求的字段，从而避免了过多的网络开销，适用于需要高效查询和响应的场景。

权衡点：REST API适合简单的资源操作，而GraphQL更适合复杂数据查询需求。

6、强一致性与最终一致性

强一致性意味着在一个分布式系统中，数据的更新在所有节点上同时生效，也就是说，一旦数据被更新，所有后续的读取操作一无论用户从哪个节点读取数据，都会得到相同的最新值。
假设在网上银行转账后，立即查看账户余额，强一致性确保看到的余额已经包含了刚刚的转账金额，无论从哪个设备或位置访问银行账户。
而最终一致性指的是在分布式系统中，数据的更新不会立即同步到所有节点，而是经过一段时间后，所有节点的数据最终会达到一致，在这之前允许短暂的数据不一致。
例如在社交媒体上，你的朋友修改了一条状态信息，最开始可能看到的还是他的旧状态，过来几秒钟后看到更新后的状态。这种情况下，系统是最终一致的：虽然在一开始数据不一致，但最终所有人看到的状态是一致的。
选择强一致性还是最终一致性，有以下两点参考：

- 强一致性：适合金融交易系统、银行账户管理和库存管理等场景，这些系统需要确保数据在所有节点上的一致性和实时性，任何延迟或不一致都可能带来严重后果。
- 最终一致性：适合社交媒体、电商购物车和内容分发网络等场景，这些系统允许在短时间内存在数据的不一致性，但最终会达成一致，以确保系统的高可用性和扩展性。

掘金技术社区 @ 360_go_php

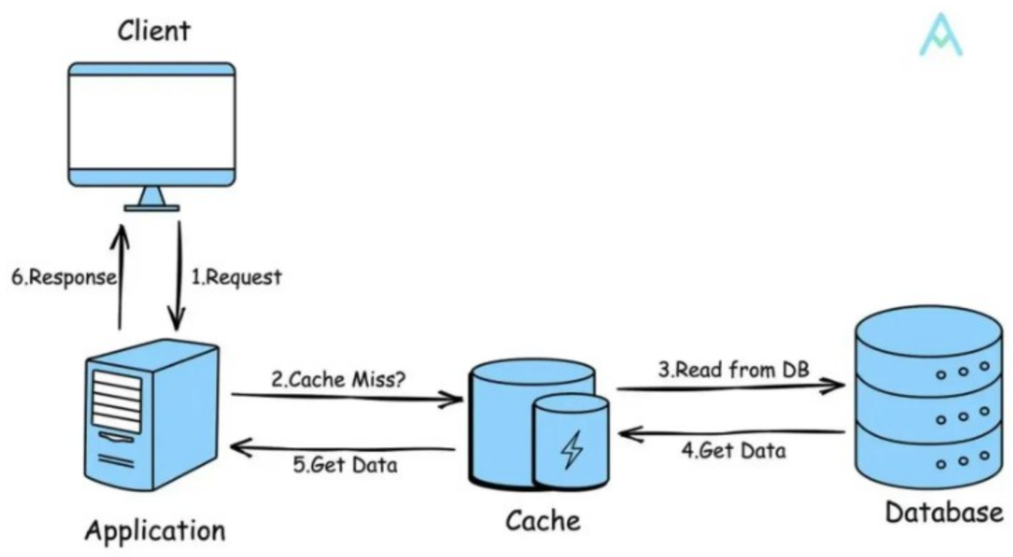
7. 负载均衡 vs. 服务器健康检查

- **负载均衡**将流量均匀分配给多个服务器，减少单个服务器的压力。
- **健康检查**机制会定期监测服务状态，防止请求被发送到故障的服务器。

权衡点：负载均衡提高了系统的可用性，但需要在健康检查机制上精心设计，避免服务不可用时依然转发请求。

7、Read-Through vs Write-Through Cache

缓存是一种通过将经常访问的数据存储在更快的存储介质中来加快数据访问的技术。说到缓存策略，“Read-Through”和“Write-Through”是两种常见的策略。
Read-Through策略： 客户端在请求数据时应用程序会先检查缓存。如果数据不存在（缓存未命中），则Cache会将数据从较慢的数据库中加载到自身中，然后再返回给应用程序。



它适用于读取频繁但不经常更新的应用程序。
例如一个新闻网站可能每天有数百万用户访问相同的文章，但这些文章的内容很少被更新。这种情况下，Read-Through模式是一种很好的选择，只有第一次读取文章的用户稍微慢点（因为第一次读取时不在缓存中，需要从数据库加载），此后所有的用户都能直接从缓存拿到文章数据了，这个性能是很高的。
Write-Through策略： 更新数据事同时将数据更新写入缓存和数据库，确保数据最新并降低数据丢失风险。

掘金技术社区 @ 360_go_php

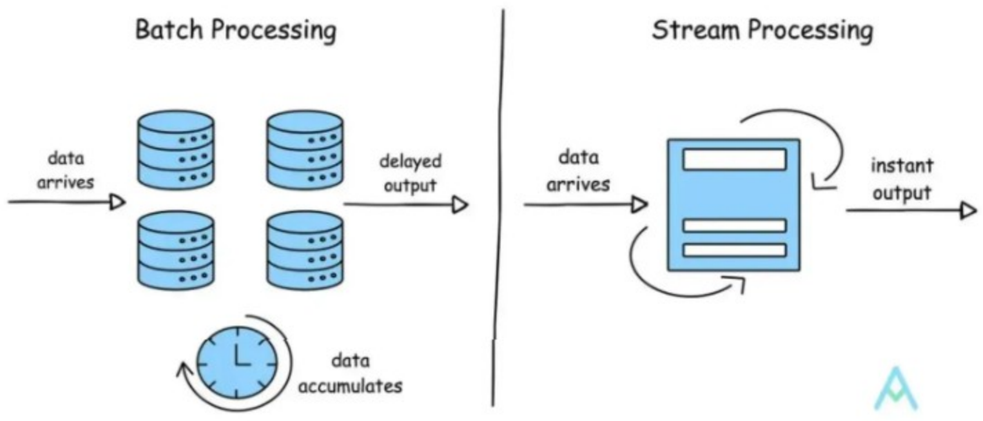
8. 单体应用 vs. 微服务架构

- **单体应用**通常较简单，适用于小型系统和初期开发，但在高并发和扩展性方面会遇到瓶颈。
- **微服务架构**通过拆分为多个小服务，每个服务独立扩展，适用于需要高可用、高扩展性的复杂系统。

权衡点：微服务架构需要更复杂的管理和协调，但在大规模应用中能提供更好的灵活性和可扩展性。

8、批处理与流处理

批处理是指将数据积累到一定量后，集中一次性处理。这种方式通常在预定的时间间隔内处理大量数据。
在电商平台上，每天或每周会对用户的浏览和购买行为进行分析，以便制定营销策略。数据量较大，不需要实时处理，因此使用批处理更为合适。
流处理是指对数据进行实时或近实时的处理。数据在生成后立即被处理，不需要等待积累到一定量。
假设在银行工作，系统需要实时监控所有的交易以检测是否存在欺诈行为。这些交易数据在发生时会立即被处理和分析，这就是流处理。



在设计系统时，需要根据具体的应用场景和需求，选择使用批处理还是流处理，或者在两者之间进行结合。

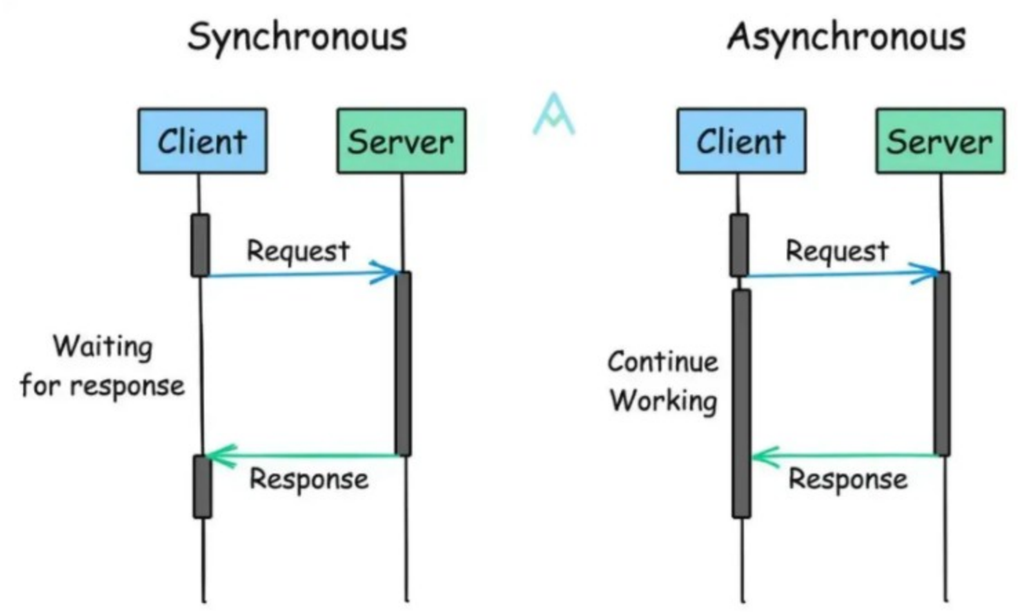
- 批处理：适合那些不需要实时性、可以集中处理大量数据的系统，比如数据仓库、离线分析、定期账单生成等。这些系统可以利用批处理的高效率，在预定时间处理积累的大量数据。
- 流处理：适合需要实时性、持续处理数据的系统，比如实时监控、在线推荐和欺诈检测系统。这些系统必须对数据进行实时处理，以快速响应用户或系统的需求。

9. 事务处理 vs. 异步任务

- **事务处理**确保数据的一致性，但可能影响性能，尤其是高并发时。
- **异步任务**通过消息队列等方式处理非核心业务，可以提高系统的吞吐量。

权衡点：对于不要求即时响应的任务，可以通过异步处理来提高系统的性能。

9、同步与异步处理



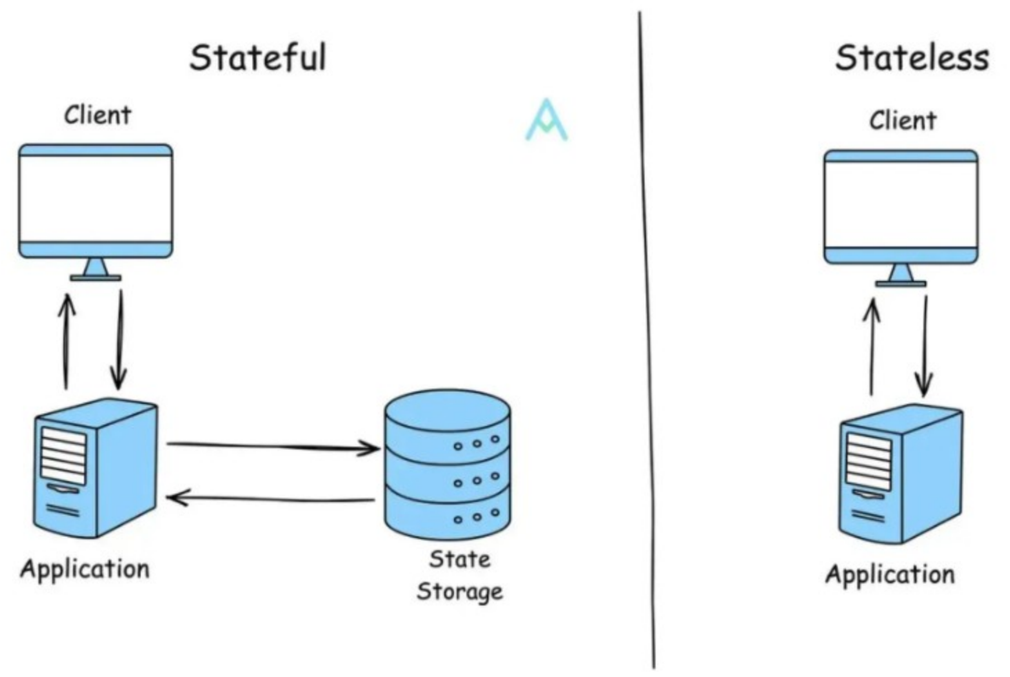
同步处理是指任务一个接一个地执行。必须完成一个任务才能开始下一个任务，系统会等待结果后再继续执行。
例如你去银行办业务，排队等候叫号，每个人必须等到前一个人办完业务，自己才能开始。这就是同步处理。
异步处理允许任务在后台运行，不需要等待其完成就可以开始本任务。
在社交媒体上上传照片是在后台异步进行的。您可以在照片上传时继续滚动或退出应用程序。

10. 数据冗余 vs. 数据去重

- **数据冗余**通过存储多个副本来提高数据的可用性和容错能力，但会增加存储成本。
- **数据去重**减少了数据存储的冗余，节省了存储空间，但可能影响数据恢复的速度。

权衡点：根据系统的容错需求和存储成本，选择适合的方式。

10、有状态系统与无状态系统



有状态系统是指服务在不同的请求之间保持状态信息，当前请求可能会依赖前一个请求的结果。
在线购物时，将商品添加到购物车时，网站会记住您的选择。如果离开以浏览更多商品，然后返回购物车，商品仍在那里，等待您结账。
无状态系统是指每个请求都是独立的，服务不需要保存前一个请求的状态信息。每个请求本身已经包含所有必要的信息，服务在处理完请求后不会保存任何数据。
例如搜索引擎，每次搜索请求都是独立的，系统不需要记住用户之前的搜索内容。

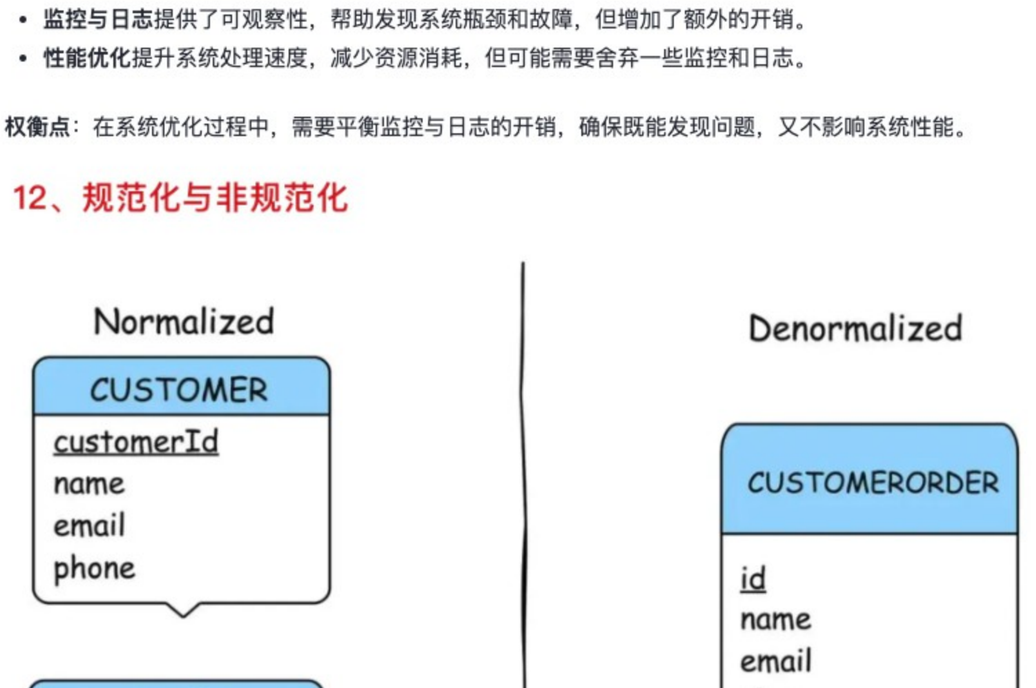
关于选择有有状态架构还是无状态架构，下面的例子可以作为参考：
哪些系统需要有状态架构？

11. 高并发 vs. 低延迟

- **高并发**优化了系统能够同时处理大量请求的能力。
- **低延迟**注重系统响应时间，减少等待时间。

权衡点：有些系统需要平衡两者，而有些系统则可以侧重其中一个，具体取决于应用场景（如实时游戏 vs. 批量处理系统）。

11、长轮询与 WebSockets



长轮询是一种技术，客户端向服务器请求数据，服务器保持请求打开，直到有新数据时才响应。客户端收到数据后再发起下一次请求，从而实现实时数据推送（更准确的讲只能做到准实时）。
这种方式技术实现较为简单，适用于简单的实时数据更新场景，例如社交媒体平台的通知系统。浏览器不断向服务器查询新通知，当出现新通知或发生超时时，服务器会做出响应。
WebSockets是一种真正的双向通信协议，客户端和服务器之间可以持续保持连接，双方都可以主动发送数据，而不必等待请求。这种方式更高效、实时性更强。
这种方式适合需要高实时性、持续数据交换的场景，例如在多人在线游戏中，WebSockets通过客户端和服务器之间的持久连接，实时共享玩家动作和游戏更新。

12. 监控与日志 vs. 性能优化

- **监控与日志**提供了可观察性，帮助发现系统瓶颈和故障，但增加了额外的开销。
- **性能优化**提升系统处理速度，减少资源消耗，但可能需要舍弃一些监控和日志。

权衡点：在系统优化过程中，需要平衡监控与日志的开销，确保既能发现问题，又不影响系统性能。

12、规范化与非规范化

数据库设计中的规范化涉及将数据拆分到相关表中，以确保每条信息只存储一次。其目的是减少冗余并提高数据完整性。
例如客户相关的信息可以有两个独立的表：一个用于客户详细信息，另一个用于订单，避免每个订单的客户信息重复。
另一方面，**非规范化**是将数据重新组合到更少的表中以提高查询性能。这通常会在数据库中引入冗余（重复信息）。

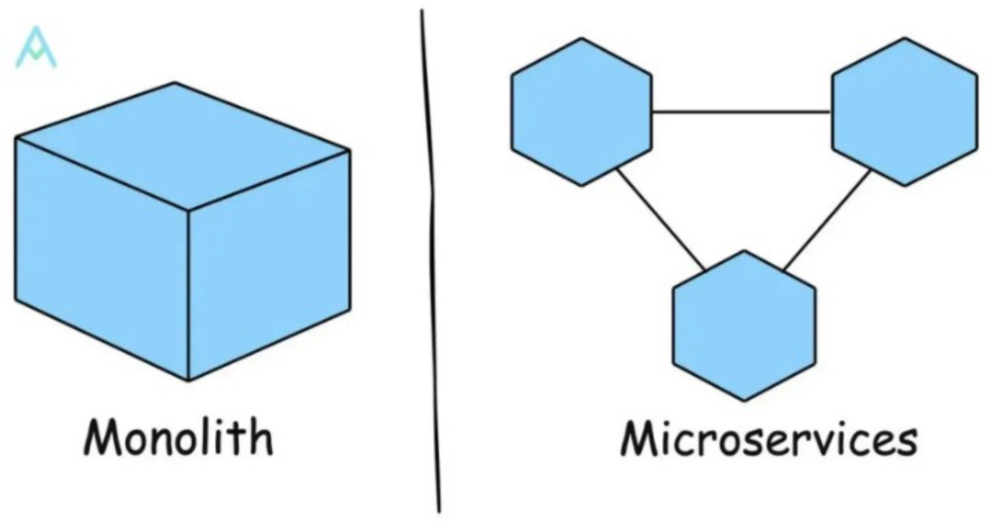
13. 加密 vs. 性能

- **加密**确保数据的安全性，但会增加额外的计算负担，影响系统性能。
- **未加密**减少计算资源的使用，但可能面临安全风险。

权衡点：选择合适的加密机制（如选择轻量级加密算法）来平衡安全性和性能。

13、单体架构与微服务架构

单体架构将应用的所有功能作为单一、不可分割的单元运行。所有功能都打包在一个进程中，通常作为单一代码库进行开发、测试和部署。
而**微服务**架构则是将应用分解为一组小型、松散耦合的服务，每个服务都专注于特定功能，并通过轻量级通信协议（如HTTP）相互交互。



单体架构简单易部署，适合小型应用程序或团队。但是，随着应用程序的增长，它可能会减慢开发速度并使可扩展性变得复杂。微服务架构提高了可扩展性和开发速度。但是，它引入了服务管理、数据一致性的复杂性，并增加了通信开销。
为简单起见，小型 Web 应用程序可能以单体式架构开始。随着规模的增长，它可以演变为微服务架构，拆分为更小、可独立扩展的服务，从而实现更好的灵活性和可扩展性。

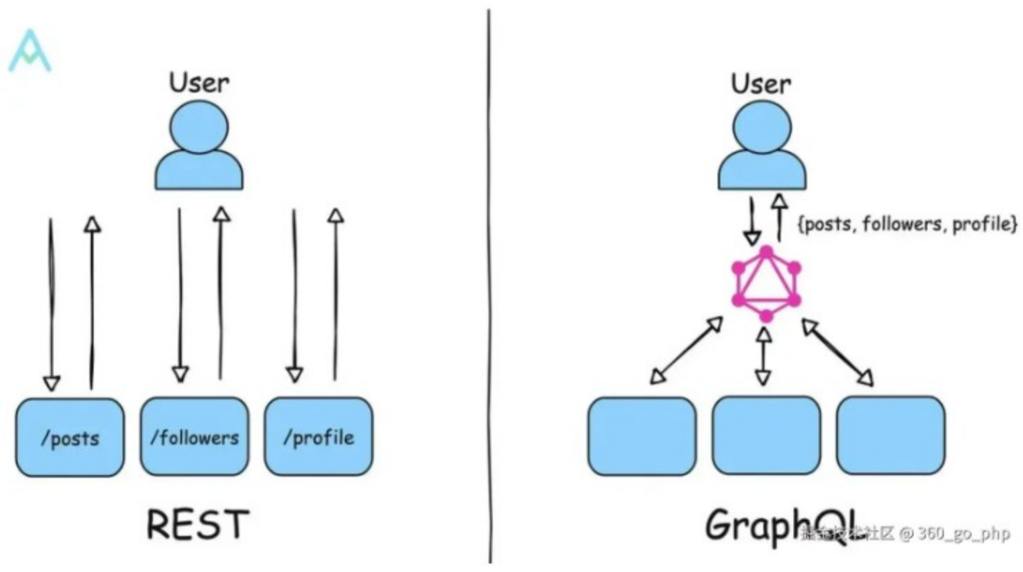
14. 短连接 vs. 长连接

- **短连接**每次请求结束后关闭连接，适用于低并发、请求量不大的场景。
- **长连接**保持连接活跃，适用于高并发、频繁请求的场景。

权衡点：长连接适用于需要频繁交互的系统，而短连接适合小流量系统，能够减少连接管理的开销。

14、REST 与 GraphQL

REST是一个成熟的API标准，提供了简洁性并支持多种格式。使用REST API时，您可以通过访问多个端点来收集数据。
GraphQL提供了更高效的数据获取，减少了请求次数，但需要更高的学习曲线和更多的前期设计。
在 GraphQL 中，可以向 GraphQL 服务器发送一个包含具体数据需求的查询。服务器然后返回一个JSON对象，其中包含了这些需求的数据。



15. 容错与容灾设计

- **容错设计**旨在在部分组件失效时，系统仍能继续正常运行。
- **容灾设计**通常是通过数据备份、冗余系统等手段，在系统出现灾难性故障时恢复系统。

权衡点：容错设计通常通过快速失败和重试来处理，但容灾设计能在灾难发生时快速恢复。

15、TCP 与 UDP

TCP 和 UDP 是TCP/IP协议中传输层的两种协议。
TCP（传输控制协议） 可确保您的消息完好无损地按照发送的顺序到达。它在发送方和接收方之间建立连接，检查数据是否正确接收，必要时会重新发送丢失的数据。TCP 非常适合用于对可靠性要求很高的应用程序，比如电子邮件服务。
UDP（用户数据报协议） 牺牲了可靠性来换取速度，适用于视频流等对时间敏感的应用程序，在这种情况下，传输过程中丢失一些数据也没关系。UDP发送数据时无需建立连接，也不检查数据是否接收或顺序是否正确。在线游戏和直播服务可能会选择 UDP，因为它的延迟较低，牺牲可靠性来换取速度。
每个权衡的选择因具体项目而异。
理解这些权衡可以帮助您作出明智的选择来设计高性能、可扩展和用户体验优秀的系统。
每个权衡的选择因具体项目而异。设计高性能、可扩展和用户体验优秀的系统。

总结：

构建高性能系统需要在多个设计维度上做出合理的权衡。通过评估业务需求、系统架构、性能目标和成本限制，可以做出最优的决策，从而实现高可用、高并发和低延迟的系统设计。每个权衡点都有其优缺点，了解并合理应用它们对于高效系统的设计至关重要。

标签： 面试 架构 Go　　话题： 每天一个知识点

本文收录于以下专栏　　1 / 3

后端与系统架构 专栏目录
整理与搜集后端知识点，不断更新架构知识
70 订阅 · 134 篇文章
订阅

上一篇 在 Go语言中一个字段可以包含多种...　　下一篇 go语言zero框架中在线截图chrome...

评论 0

登录 / 注册 即可发布评论！

暂无评论数据

为你推荐

软件系统架构黄金法则：解析软件架构的模式和风格
OpenChat | 1年前 | 42 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：可扩展性设计
OpenChat | 1年前 | 16 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：理解软件架构的角色和责任
OpenChat | 1年前 | 32 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：理解软件架构的角色和责任
OpenChat | 1年前 | 40 点赞 评论 后端

软件系统架构黄金法则：设计模式的应用
OpenChat | 1年前 | 29 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：如何设计高性能系统
OpenChat | 1年前 | 23 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则23：松耦合原则法则
OpenChat | 1年前 | 44 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：构建可扩展性的基石
OpenChat | 1年前 | 35 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：构建可扩展性的基石
OpenChat | 1年前 | 18 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则15：高性能搜索的架构法则
OpenChat | 1年前 | 13 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：深入解析软件架构的重构
OpenChat | 1年前 | 32 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：探讨软件架构的未来趋势
OpenChat | 1年前 | 20 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：团队协作与沟通
OpenChat | 1年前 | 14 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：解耦合的艺术与实践
OpenChat | 1年前 | 52 点赞 评论 后端 架构 人工智能

软件系统架构黄金法则：弹性设计的架构方法论
OpenChat | 1年前 | 16 点赞 评论 后端 架构 人工智能