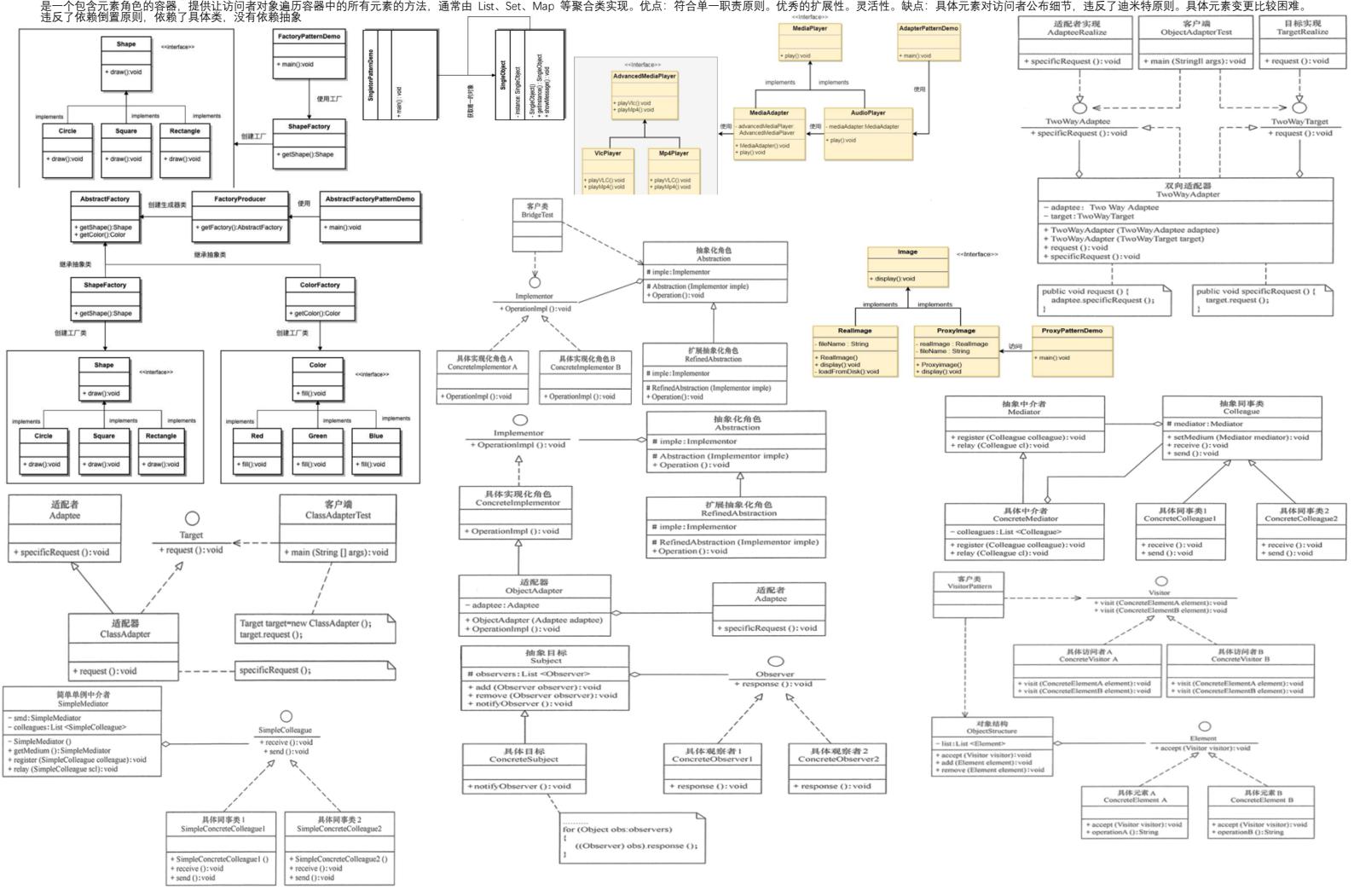
什么是系统体系结构 **软件架构**定义为：组件+连接件+拓扑结构+约束+性能 **软件架构**：提供了一个结构、行为和属性的高级抽象从一个较高层次来考虑组成系统的构件、构件之间的连接，以及由构件与构件交互形式形成的拓扑结构。这些要素应该满足一定的限制，遵循一定的设计规则，能够在一定的环境下进行演化。反映系统开发中具有重要影响的设计决策便于各类人员的交流，反映多种关注点。抵此开发的系统能够完成系统既定的功能和性能需求。**为什么需要软件系统体系结构**：对于大规模的复杂软件系统来说，对系统全局结构的设计比起对算法的选择和数据结构的设计明显重要得多。**软件系统体系结构的目标与作用**：软件架构关注的是：如何将复杂的软件系统划分为模块。如何规范模块的构成。如何将这些模块组成为完整的系统。以及保证系统的质量要求。主要目标：建立一个一致的系统及其视图集，并表达为最终用户和软件设计者需要的结构形式，支持用户和设计者之间的交流与理解。分为两方面：外向目标：建立清楚最终用户要求的系统需求。内向目标：建立清楚系统设计者需要以及易于系统实现、维护和扩展的系统结构构成。**作用**：交流手段：架构充当了设计者与用户、开发团队之间的沟通媒介。 可传递和复用的模型：一个好的软件架构不仅可以在当前项目中使用，还可以作为其他类似项目的参考，提高代码复用率。关键次要的体现 这些次要对系统质量有着最显著的影响，并且展示了各种折衷：性能与安全性之间的折衷。可维护性与可靠性之间的折衷。当前开发成本与未来开发成本之间的折衷。**意义**：SA 设计的成本和代价要低得多 正确有效的 SA 设计会给软件开发带来极大的便利 质量属性更多的是由系统结构和功能划分来实现的，而不再仅仅依靠所选择的算法或数据结构 **软件架构在软件生命周期中的作用** 项目规划：考虑项目的规模、复杂度、可行性等；利用 SA，支持用户、项目负责人、系统架构师、程序员、测试人员之间进行交流和协商；从不同视角审查备选的 SA，对相关的意见进行综合，找出合理的平衡方案；从用户角度考虑未来的需求会发生什么变化，并使 SA 能够提前支持这些变化。需求分析：参考经典 SA 风格，设计系统架构模型，推敲其存在的敏感点和替代方案，并进行评估；进一步细化 SA，并对定型后的 SA 作文档化工作。软件设计：各开发团队按照 SA 规定的“构件及其之间的相互关系”进行开发；保证最终得到的系统与最初的 SA 一致。软件实现：各开发团队按照 SA 规定的“构件及其之间的相互关系”进行开发；保证最终得到的系统与最初的 SA 一致。测试与评审：根据 SA 的约束条件，对软件的质量属性进行测试。维护与升级：把 SA 文档作为维护和升级的重要依据。架构的选择极大地影响部分**软件质量**，但不是全部 架构只为获得某个质量创造条件，但并不能保证肯定获得 我们能做的是，使架构 可文档化 易于理解 可度量 可复用 易于交流和执行 **系统体系结构与中间件**：中间件是一组应用于分布式系统的程序，主要作用是为系统屏蔽底层的通讯细节，并提供公共服务，确保系统的高可靠性、高可用性和高灵活性。它通常位于操作系统和应用程序之间，起到连接不同应用系统或独立系统的作用，尤其是在它们接口不同的情况下。**作用**：屏蔽异构性 异构性来源：包括计算机硬件差异（如 CPU 和指令集、硬件结构、驱动程序等）、操作系统差异（如不同 OS 的 API 和开发环境）、数据库差异（如不同的存储和访问格式）。实现互操作 异构性导致软件依赖于计算环境，不同软件之间可能不能直接通信或迁移和相对困难。共性凝练和复用随着软件应用领域的扩展，相同领域的应用系统之间往往共享基础功能和结构。提供一致、集成的开发与运行环境，简化分布式系统的设计、编程和管理。**分类**：应用服务类中间件：为应用系统提供一个统一的计算环境和支持平台，包括：对象请求代理（ORB）中间件事务监控交易中间件 Java 应用服务器中间件等 应用集成类中间件：提供各种不同网络应用系统之间的消息通信、服务集成和数据集成的功能，包括：常见的消息中间件企业集成 EAI 企业服务总线以及相配套的适配器等 业务架构类中间件：除了可以将底层共性技术的特征抽象到中间件外，还可以将业务共性抽象至中间件，形成应用模块，如：业务流程 业务模型 业务规则 交互应用**计算层的软件架构技术挑战**，**计算模式变革** 单机处理 并行计算 集群计算 分布式计算 传统垂直架构改造的核心就是要对应用进行服务化。 服务化改造使用到的核心技术就是分布式服务框架。业务纵向拆分：按照业务领域将系统拆分，每个业务域都独立部署到集群中。比如，将订单管理、商品管理、客户管理分别独立部署。 业务横向拆分：将核心业务拆分为多个服务，并通过分布式服务框架来协调这些服务。扩展性 高可用性 **分布式编程模型** 分而治之 计算向数据靠拢 **Map 函数**：将一个大的数据集拆分为多个小任务，每个小任务可以并行处理 **Reduce 函数**：汇总 Map 任务的输出，并得到最终的结果 Map 函数：输入：<k1, v1>（例如，<"文件名", "Hello World">）。输出：List(<k2, v2>)（例如，<"Hello", 1>，<"World", 1>）。作用：将输入数据解析为一系列<key, value>对，每个<k1, v1>会输出一批<k2, v2>。这就是 Map 任务的中间结果。Reduce 函数：输入：<k2, List(v2)>（例如，<"Hello", [1, 1, 1]>）。输出：<k3, v3>（例如，<"Hello", 3>）。作用：汇总 Map 任务的中间结果，得到最终的输出。**体系结构**：Client：用户编写的 MapReduce 程序通过 Client 提交到 JobTracker 用户可通过 Client 提供的一些接口查看作业运行状态 JobTracker：负责作业调度和资源监控，监控所有 TaskTracker 的状态，并在任务失败时重新分配任务 TaskTracker：负责执行 Map 或 Reduce 任务，并周期性向 JobTracker 报任务进展 Task：分为 Map Task 和 Reduce Task，由 TaskTracker 启动。**输入数据的分片（Split）**：将输入数据划分为多个分片，每个分片对应一个 Map 任务 Map 任务：并行执行 Map 任务，将输入数据转换为一系列<key, value>对 **Shuffle**：将 Map 任务的中间结果传递给相应的 Reduce 任务，进行排序和分区 Reduce 任务：汇总 Shuffle 过程的结果，得到最终的输出。重要概念：Split：MapReduce 处理单位的逻辑划分，通常与 HDFS 块大小相同（默认 64MB）**Shuffle**：Map 任务和 Reduce 任务之间的数据传输和排序过程。**Map 端的 Shuffle 过程** 在 Map 任务生成了中间结果后，Shuffle 过程就开始了步骤如下：输入数据和执行 Map 任务：Map 任务会读取输入数据（例如文件的某些分片）。并生成一系列键值对（key-value）写入缓存：Map 任务的输出数据会首先写入缓存 MapReduce 默认会为每个 Map 任务分配一个 100MB 的缓存 溢写：当缓存达到一定比例（默认是 0.8，即缓存使用 80%时），Map 任务就会将缓存中的数据写入磁盘，这个过程称为溢写 分区：MapReduce 会根据键值对的哈希值，将数据分配到不同的分区。不同分区的数据会被发往不同的 Reduce 任务 排序：Map 任务的输出会自动进行排序 这是 MapReduce 中的默认操作，目的是将相同的键聚集在一起，方便后续的 Reduce 操作。合并：在溢写过程中，如果有相同的键，Map 端可以调用一个本地的合并操作，将相同的键提前合并，减少传输的数据量和后续 Reduce 任务的工作量 文件归并：当 Map 任务的所有输出都写入磁盘后，Map 任务会将多个溢写文件归并成一个大的文件，减少文件碎片。**Reduce 端的 Shuffle 过程** Reduce 端的 Shuffle 过程从 Map 端输出的中间结果开始，并将这些结果传递给 Reduce 任务。具体步骤如下：领数据：Reduce 任务会通过 RPC 向 JobTracker 询问哪些 Map 任务已经完成，并从这些 Map 任务中获取中间结果。 Reduce 任务会将来自不同 Map 任务的中间结果存入自己的缓存中。归并数据：当 Reduce 任务从多个 Map 任务中获取到数据后，会先将这些数据进行归并，确保相同的键被聚集在一起。归并是排序的过程，不会改变数据的结构。如果数据较少，归并可以直接在缓存中进行；如果数据量大，则先写入磁盘，再进行归并。Reduce 任务：归并后的数据会输入给 Reduce 任务，Reduce 任务会根据相同的键进行处理，最终生成输出结果。Shuffle 是 MapReduce 中最复杂和关键的部分，它的主要作用是将 Map 任务的中间结果传递给相应的 Reduce 任务，并确保数据按照键值对进行分区和排序。整个 Shuffle 过程包括以下几个关键步骤：Map 端：将 Map 任务的输出进行分区、排序、合并和归并，生成中间结果文件。Reduce 端：从 Map 任务获取中间结果，进行数据归并，最终交给 Reduce 任务进行处理。**MapReduce 程序的流程及设计思路**：提交 Job：用户首先提交一个作业（job），该信息发送给 Job Tracker Job Tracker 管理：Job Tracker 是框架的核心，它负责与集群中的各个节点（机器）通信，分配任务、监控任务执行情况、处理失败任务并进行重启等操作 TaskTracker 角色：每台机器都有一个 TaskTracker，负责监控本机的资源情况（如 CPU、内存等），并运行当前机器上的 tasks（任务）。TaskTracker 通过心跳（heartbeat）定期向 Job Tracker 汇报状态 任务分配：Job Tracker 根据 TaskTracker 汇报的资源信息，将新提交的 job 分配到合适的机器上运行。**MapReduce on YARN 工作流程** 客户端提交任务：用户通过客户端向 ResourceManager 提交任务。ResourceManager 是 YARN 中的资源管理器，负责整个系统的资源调度。ResourceManager 包含两个关键组件：Applications Manager：负责管理应用的生命周期。Resource Scheduler：负责为每个应用分配资源。ApplicationMaster 启动：ResourceManager 接收到任务后，会为每个任务启动一个 ApplicationMaster。ApplicationMaster 是每个任务的管理者，负责为该任务申请资源，并监控任务的执行。申请资源（Resource Allocation）：ApplicationMaster 向 ResourceManager 申请运行任务所需的资源（如 CPU、内存等）。ResourceManager 分配资源，并将任务调度到集群中的各个节点上运行。任务执行（Task Execution）：每个节点上运行的 NodeManager 负责启动 Container。每个 Container 是一个独立的运行环境，负责执行具体的任务（Map Task 或 Reduce Task）。Container 会运行 Map 或 Reduce 任务，并定期向 ApplicationMaster 汇报任务执行状态。任务监控与容错（Task Monitoring and Fault Tolerance）：ApplicationMaster 负责监控每个任务的执行状态。如果某个任务失败，ApplicationMaster 会将失败的任务重新调度到其他节点上执行。ApplicationMaster 通过与 ResourceManager 和 NodeManager 的协作，确保任务的正常执行。结果汇总（Result Aggregation）：Map Task 完成后，结果会被传递到 Reduce Task。Reduce Task 将所有 Map 的结果汇总，并输出到目标文件系统。负载均衡：**DNS 负载均衡** DNS 负载均衡是一种通过 DNS 服务器解析，将请求分配到不同服务器的负载均衡方式。它是最简单和常见的负载均衡方法，通常用于地理级别的均衡。优点：简单、成本低：不需要额外的负载均衡设备。交由 DNS 服务器处理，提高访问速度。通过返回距离用户最近的服务器IP，加快访问速度。缺点：更新不及时：DNS 缓存时间较长，修改配置后仍有可能使用旧 IP。扩展性差：无法根据业务特性进行更细致的流量分配。灵活性不足：无法根据服务器的健康状态、不能根据负载状况智能分配。**硬件负载均衡** 硬件负载均衡是通过专用硬件设备（如 F5、Citrix）进行请求分发与流量管理的负载均衡方式。它通常具备强大的性能和丰富的功能，适合大规模商用。优点：功能强大：支持多种负载均衡算法，并支持会话负载均衡。性能优越：专门设计的硬件设备确保高性能处理。稳定性高：硬件设备经过严格测试和大规模应用，适合关键业务场景。缺点：成本高昂：硬件设备价格通常较高，适合大型企业或预算充足的场景。灵活性较差：配置和调整需要专业技术人员，扩展性受限。硬件设备不易根据业务需求进行扩展和定制。**软件负载均衡** 软件负载均衡通过软件程序来实现请求分配，通常部署在服务器上。不依赖于专用的硬件设备。它成本低、灵活性高，适合中小规模的应用场景。Nginx：一个 7 层负载均衡器，支持 HTTP 和其他应用层协议，灵活性较高。LVS：4 层负载均衡器，独立于协议，可以处理大多数应用场景。优点：简单易用：只需要 Linux 服务器和负载均衡软件，易于部署和维护，成本低；仅需服务器硬件，无需额外的专用设备。灵活性强：可以根据业务需求进行功能定制，例如通过 Nginx 插件实现特定的负载均衡策略。缺点：性能一般：如 Nginx 支持的并发数在 5 万左右，远低于硬件负载均衡。功能不及硬件负载均衡强大：例如缺乏防火墙和 DDoS 防护功能。**轮询** 轮询算法将请求按顺序分配到各服务器，不考虑服务器的当前负载和状态。特点：简单、易于实现，不需要复杂的负载监测。对服务器状态不敏感。例如，如果某台服务器繁忙或出错，负载均衡仍然会分配任务给它，可能导致性能问题。**加权轮询** 在轮询的基础上加入权重，根据服务器配置的权重分配任务。特点：可适应不同处理能力的服务器，适用于硬件性能差异的服务器。静态权重设置，无法动态调整以适应服务器实时负载变化。**负载最低优先** 将任务分配给当前连接数最少的服务器。特点：LVS 可以通过连接数判断压力，Nginx 可以用请求数评估服务器压力。适合需要动态评估服务器负载的场景。需要实时监控服务器状态，复杂性较高。可能导致负载均衡系统的性能下降。最低响应时间优先 根据服务器的响应时间进行任务分配，将任务分配给响应时间最短的服务器。特点：快速响应客户请求。需要监控和计算响应时间，增加复杂性。可能影响系统整体性能。**Hash 类** 根据请求的关键属性（如 IP 地址或 ID）进行 Hash 运算，将请求分配给特定服务器。类别：源地址 Hash：基于请求源 IP 地址分配，适合需要会话保持的场景。ID Hash：基于某种业务标识（如 session id）分配，确保用户在同一会话期间访问同一服务器。特点：能够确保会话一致性，适合需要会话保持的业务。Hash 算法需要设置合适的关键属性，可能不适合负载动态变化的场景。**分布式系统中的进程**：这些进程分散在不同的机器上，不像传统系统中那样共享内存来进行通信。由于底层的网络不可靠，所以分布式系统中的进程通信变得更具挑战性。**进程间通信的挑战**：分布式系统没有基于共享内存的原语，因此进程间的信息交换主要通过低级别的消息传递来完成。消息传递是通过底层网络来实现的，这种通信方式相比共享内存要复杂得多。**分布式系统的通信**：分布式系统的通信依赖于底层网络提供的支持，并且是通过低级别的消息传递来完成的，难度较高。**基于寻址的通信模型**：分为直接通信和间接通信。直接通信中，发送方和接收方通过明确地址进行通信，而间接通信通过中间媒介（如邮箱）传递消息。**基于阻塞**：分为同步通信和异步通信。同步通信中，发送方会阻塞等待接收者的确认，而异步通信中，发送方无需等待回复，可以继续执行其他任务。**基于缓存**：分为带缓冲（持久的）和不带缓冲（瞬态的）通信。根据消息是否需要中间存储来区分。**基于内容**：按通信内容的性质分类，包括事件、命令、数据和流。**基于确认**：根据通信的确认机制，分为无确认、确认和三次握手，保证消息接收的可靠性。**基于接收节点数**：按接收方的数量和范围进行分类，包括点对点、多播、任播、地域性群播和广播。**基于通信方向**：分为单向通信、双向半双工通信和双向全双工通信。主要描述了信息在发送方和接收方之间的流动方向和限制。 **基于发起方**：分为客户端发起的拉取模式（Pull）和服务器发起的推送模式（Push）。主要描述了谁首先发起通信。**基于消息存储**：分为持续通信和瞬态通信。主要描述了消息在传输过程中的存储方式，以及接收方是否需要实时在线。**消息队列（MOM）** MOM 是一种中间件技术 实现异步消息的传输 用于不同系统组件间的通信 主要特点：基于消息的通信机制 消息存储在队列中 消息服务器解耦客户端和服务端 支持异步处理 系统架构：客户端应用程序 本地消息队列 中央消息服务器 网络连接层 服务端应用程序 实现机制：消息队列：一对一传递发布-订阅：一对多传递 技术优势：解耦系统组件 提高可靠性 提升系统扩展性异步通信 提高效率高 应用场景：分布式系统通信跨平台数据传输应用程序集成负载均衡面向消息的中间件（Message-Oriented MiddleWare, MOM）是消息队列系统的一种重要类型，它通过提供持久异步通信来确保消息在发送方和接收方不必同时在线的情况下进行传输。MOM 提供了中间态的消息存储容量，支持消息传输即使需要数分钟也能正常进行。这种中间件不需要发送方或接收方在消息传输期间保持活动状态，确保了消息的可靠传递。MOM 的属性包括异步交互 客户端和服务器端是松散耦合的 消息被持久化 队列管理 适合应用集成场景。它支持可靠的服务交付，通过持久存储的队列保证消息不会丢失。同时，消息可以通过中间消息服务器进行处理，执行过滤、转换、记录等操作。支持多服务器网络的消息传递。此外，MOM 还天然地支持与数据库的集成，方便不同系统之间通过共享的队列中插入消息进行通信，消息从发送方经过一系列通信服务器后最终传递到接收方。接收方在消息发送时以处于离线状态，发送方和接收方无需同步运行。系统保证消息最终会被插入到接收方的队列中，但不能保证具体的插入时间。同步通信和紧耦合的通信方式，发送方在发送消息后会等待接收方的回应，直到收到回复。在这种模式中，发送方和接收方通常需要保持在线状态，以确保消息的实时交换。同步通信的优点是简单易实现，适用于需要即时反馈的场景，但缺点是发送方和接收方必须同时在线，且客户端会被阻塞，导致并行性较低。异步（持久）通信则是一种松散耦合的方式，发送和接收可以异步进行，消息通过队列传输。这种方式支持更高的并行性，发送方和接收方不需要同时在线，能够缓解速度差异，但也带来了复杂的同步问题。异步通信还存在潜在的缓冲区溢出风险，且发送方无法及时了解消息是否被成功接收 **队列消息传递模式**可以分为以下几类：One-to-One：一对一，即一个发送者对应一个接收者。这种模式通常是点对点（Point-to-Point）消息传递。One-to-Many：一对多，即一个发送者对应多个接收者。Many-to-One：多对一，即多个发送者对应一个接收者。Many-to-Many：多对多，即多个发送者对应多个接收者。这种模式通常应用于发布/订阅（Publish and Subscribe）模型。消息传递/点对点模式中，发送者将消息发送到队列管理器，由队列管理器将消息分别传递给不同的接收者。发布和订阅模式中，消息发送者发布消息到特定的主题，订阅者根据主题接收消息。这种模式支持多发送者和多接收者的通信模型，允许灵活的消息分发。请求/回复队列模式中，发送者通过请求队列发送请求，接收者通过回复队列发送对应的回复。客户端使用关联 ID 来匹配请求和对应的回复。**消息队列**是发送方和接收方之间的命名消息目的地，作为中间媒介允许进程独立执行并能容忍失败，掩盖进程失败和通信失败。消息队列的优先级机制可以通过**最高优先级优先**或**加权公平调度**来处理不同优先级的消息，确保资源合理分配。面向消息的中间件支持多种消息传递模型，提供队列管理、连接管理、服务质量（QoS）以及数据转换等功能。队列管理器是用于管理队列的专门数据库，提供队列的创建、删除、启动、停止功能，并可监控性能、故障和恢复情况。此外，队列管理器还能配置为将消息转发给其他队列管理器，实现消息代理和智能路由等高级功能，如基于内容的路由和流控制。通过队列的时间解耦和位置解耦，消息发送和接收可以在不同时间、不同位置独立进行，提高系统的灵活性和容错性。**分布式服务框架**：Service 层 定义：Service 层主要负责将远程服务的接口提供给消费者，通过动态代理封装远程调用逻辑，使得调用方无需关心底层实现。核心组件：Customer Service Interface (Spring Bean)：面向业务的接口，供服务消费者调用。Spring Bean 常用于配置和管理这些接口。Dynamic Proxy：动态代理通过 Java 反射或 CGLIB 技术，将远程调用接口的实现类在运行时动态生成。代理将调用包装成消息请求（包含接口名称、方法名、参数），并交给底层框架处理。Filter Chain (过滤链)：功能：处理拦截、超时重试、负载均衡等操作。主要模块：LoadBalance：根据预设策略（如随机、轮询、权重）选择目标服务实例。Timeout/Resend：对调用超时进行重试或失败转移。支持灵活扩展的过滤链模式，便于开发者定制请求处理流程。RPC 层 负责处理跨进程的远程调用逻辑（Remote Procedure Call），包括协议封装、序列化和网络传输。核心组件：Remoting Abstraction：提供统一的远程通信抽象层，屏蔽具体通信协议细节，支持灵活切换。通信协议：Binary Protocol：二进制协议，效率高，常用于高性能场景（如 Dubbo）。Text Protocol：文本协议，易读性强，多用于简单场景或调试。Web Services：基于 SOAP 协议的通信方式，适合跨平台应用。Restful：基于 HTTP 协议的轻量级通信模式，适合现代化微服务架构。Hessian：基于二进制序列化的高效协议，常用于 Java 分布式系统。Filter Chain：在 RPC 层，也可通过过滤器实现序列化、协议适配和连接池优化等功能。Transport 层定义：最底层的传输模块，负责实际的数据传输和连接管理。核心组件：NIO Framework：基于 Java 的非阻塞 I/O 框架（如 Netty），提供高性能的网络传输能力。支持多路复用、事件驱动，提高并发能力。HTTP Based Protocol：使用 HTTP 协议作为传输层，支持广泛的浏览器和客户端兼容性。适合与 Restful 或 Web Service 结合使用。原理与流程 调用方发送请求：业务代码通过 Customer Service Interface 发起服务调用。通过动态代理生成调用请求，打包成远程调用消息。拦截与过滤：Filter Chain 对请求进行拦截，执行超时、负载均衡等逻辑。协议处理与序列化：Remoting 层对消息进行序列化，通过指定协议（如 Binary Protocol、Restful）封装请求。数据传输：Transport 层通过 NIO 或 HTTP 框架进行数据传输，将请求发送至目标服务实例。服务端处理：服务端收到请求后，解码消息，调用实际的业务方法，并将结果返回给调用。响应经过相同的逆向流程，最终返回给调用方。特点与优势动态代理解耦：动态代理隔离调用方和服务实现，提高灵活性。可扩展过滤链：通过 Filter Chain 实现多种功能扩展，增强服务治理能力。协议与传输解耦：支持多种通信协议，满足不同场景需求。高性能传输：通过 NIO 等技术提供高效的数据传输，支持大规模并发。应用场景微服务架构：提供分布式系统中的服务调用能力。高并发系统：动态代理与 NIO 结合，满足高性能需求。异构系统集成：通过多协议支持，实现跨语言、跨平台的服务互通。**分布式服务框架主要功能：服务注册中心**：负责服务的发布和通知 **服务治理中心**：功能特性：服务订阅分布 服务路由 集群容错 服务调用 多协议 序列化方式 统一配置 幂等性概念 **分布式服务框架的功能特性**包括服务订阅分布 服务路由 集群容错 服务调用 多协议 序列化方式 统一配置 服务订阅分布功能使得服务的发布和引用可以通过配置化进行，支持 XML 配置，降低代码侵入性，同时提供服务自动发现机制，消费者无需手动配置服务提供者地址，支持在线注册和注销服务，实现动态管理。服务路由功能提供多种路由策略，包括随机、轮循和基于权重的路由，还支持粘滞连接，使请求总是发往同一服务提供者，并允许用户自定义路由策略以扩展平台功能。集群容错机制确保系统稳定性，常见的容错策略包括 Failover（自动切换到其他服务器）、Failback（自动恢复并重试请求）和 Failfast（快速失败，立即返回错误）。服务调用模式分为同步调用、异步调用和并行调用，分别用于不同的业务场景。同步调用要求消费者发起请求后等待响应，而异步调用则不阻塞，服务端异步返回结果，并行调用允许消费者同时向多个服务提供者批量发起请求，多协议支持是分布式服务框架的重要特性，既支持私有协议（如二进制协议、Thrift、Protocol Buffer 等）用于内部扩展，也支持公有协议用于外部对接。序列化方式包含二进制类和文本类方式。二进制类序列化如 Thrift 和 Protocol Buffer 提高了传输效率，而 JSON 和 XML 等文本类序列化方式则提升了兼容性和可读性，框架还支持统一配置，既可以实现静态配置，也支持动态配置的实时生效，方便运维和管理。**幂等性**是分布式系统中非常重要的概念，指的是无论同一操作执行多少次，产生的结果都是一致的，避免重复执行导致的副作用。幂等性常见于网络波动、消息消费、用户重复提交和未完成的重试机制等场景中。数据库操作的幂等性表现有所不同，查询类操作通常是幂等的，而更新和删除操作则有可能不是幂等的。幂等性解决方案包括数据库加锁、全局唯一 ID、去重表、多版本控制和状态检测等方法。数据库加锁可以避免并发问题，但可能引发效率和死锁问题。全局唯一 ID 通过标识重复请求，避免重复执行。去重表通过记录操作来检查重复操作，多版本控制则通过添加版本号确保数据一致性，状态检测适用于状态流转类业务场景，如订单支付中的成功与失败分辨。**微服务**：微服务是一种架构设计模式，将业务逻辑拆分成一系列松散耦合的分布式组件，每个组件独立运行，共同构成较大的应用。每个微服务具有单一职责，负责特定的功能任务。微服务间的调用关系是通过一个或多个微服务之间进行，完成具体任务。系统为任务提供统一的解决方案，负责应用程序的分布式部署和功能扩展。微服务的需求要点 负责任务功能。单独部署，每个微服务独立运行。包含一个或多个进程。拥有自己的数据存储。由一个团队维护。可替换和独立的服务单元。微服务架构与 SOA 架构的区别 组成大小：SOA 为大块业务逻辑，微服务为小块业务逻辑。公司架构：SOA 适合任何类型公司，微服务适合小型团队。管理方式：SOA 为集中管理，微服务为分散管理。目标：SOA 确保应用间的交互性，微服务注重快速部署和扩展。**微服务架构的服务集成** 点对点方式：直接调用其他微服务的 API 接口，适用于简单场景，但应用复杂时不推荐使用。API 网关方式：所有客户端和微服务的通信通过统一的 API 网关进行，提供安全、监控、负载均衡等功能，是微服务的广泛应用模式。消息代理方式：通过异步消息发布和订阅的模式实现微服务之间的通信，适合复杂的生产/消费模型。**微服务架构的服务发现** 客户端发现模式：客户端查询服务注册表，决定调用的服务实例，实现负载均衡。 服务端发现模式：负载均衡器查询服务注册表，将请求转发至可用的服务实例。**服务注册**：服务实例必须在服务注册表中注册和注销，支持自注册模式和第三方注册模式。**微服务架构的数据去中心化** 每个微服务有独立的数据库存储和访问权限，不能直接访问其他服务的数据库。数据去中心化确保各微服务间松耦合和自治，防止单一数据库故障。支持 SQL、NoSQL 等多种数据库存储类型。**Docker 引擎**：C/S 结构的应用程序，包含 Server 端（daemon）和 Client 端。Server：常驻进程，用于处理 Docker 命令与 API 请求。REST API：用于 client 和 server 间的交互。CLI：提供容器和镜像的管理命令行界面。虚拟机：基于 Hypervisor 的虚拟化，包含完整操作系统（Guest OS），资源消耗大。Docker：轻量化，直接与主机内核交互，不需要完整的操作系统层，资源利用率高，效率高。**Docker 的优势** 快速交付和部署：Docker 支持应用和服务的快速开发与交付。高效部署和扩容：几乎可在任意平台运行，方便实现负载管理和动态扩容。高资源利用率：一台主机可运行多个 Docker 容器，减少系统资源浪费。简便管理：修改直接发布更新，支持自动化管理。**Docker 的劣势** 资源隔离：cgroup 实现资源限制，但不如虚拟机强大。安全性问题：目前无法细粒度区分用户操作权限。版本依赖：依赖内核版本，对高版本内核兼容性较差。**Docker 架构** C/S 架构：通过 Client 与 Server 通信实现容器管理。镜像（Image）：只读的模板，用于创建容器，类似模板文件。仓库（Repository）：存放镜像的地方，有公共和私有仓库，类似 GitHub。容器（Container）：容器是从镜像启动的一个实例。容器的文件系统（Union File System）：Docker 用了一种叫“联合文件系统”的技术。**Docker 调度工具 Swarm**：Docker 的集群管理工具，支持将多个主机组成集群，进行容器的调度与管理。调度策略包括 random（随机选择节点）、spread（资源均分分配）、binpack（尽可能填满一个节点）。Apache **Mesos & Marathon**：提供一个高效、可扩展的系统，支持资源共享和多框架的兼容。**Kubernetes** 是一个用于自动部署、扩缩和管理容器化应用的开源系统。Kubernetes 集群由多个部分组成，其中 Pod 是最小的部署单元，通常包含一个或多个容器。这些容器共享同一个网络和端口，可以相互通信。Service 为一组 Pod 提供统一的访问接口，避免 Pod 动态变化带来的影响。Kubernetes 使用 Label 对对象进行分类管理，方便搜索和更新。DNS 服务为服务提供名称解析，Replication Controller 则确保集群中维持指定数量的 Pod 实例。Kubernetes 集群架构由 Master 节点和 Node 节点组成。Master 节点负责控制和管理集群，包括组件包括 kube-scheduler（调度器）、kube-apiserver（API 服务器）和 kube-controller-manager（控制器管理器）。Node 节点运行实际的容器工作负载，包括 Kube-proxy（网络代理）和 Kubelet（节点代理）。关键概念：Pod 生命周期管理：Pod 的状态包括 Pending、Running、Succeeded 和 Failed。Kube-scheduler 根据资源需求、亲和性规则等因素决定 Pod 被分配到哪个节点。调度器根据一系列规则来确保选择合适的节点，首先是通过调谐确保节点具备足够的资源和条件，然后通过优先级（Priorities）在多个可用节点中选择最合适的节点。**数据的定义**：一般而言，数据是指对客观事件进行记录并可以鉴别的符号，是对事物的性质、状态以及相互关系等进行记载的物理符号或这些物理符号的组合。它是可识别的、抽象的符号；表示形态上，数据可以是狭义上的数字，可以是有一定意义的文字、字母、数字符号的组合、图形、图像、视频、音频等，也可以是客观事物的属性、数量、位置及其相互关系的抽象表示；在计算机科学中，数据是指所有能输入到计算机并被计算机程序处理的符号的介质的总称，是用于输入电子计算机进行处理，具有一定意义的数字、字母、符号和模拟量等的通称。**数据的解释** 1.数据与信息：信息与数据既有联系，又有区别。数据是信息的表达、载体，信息是数据的内涵，是形与质的关系。数据本身没有意义，数据只有对实体行为产生影响时才成为信息。2.数据与语义：数据的表现形式还不能完全表达其内容，需要经过解释，数据和关于数据的解释是不可分的。数据的解释是指对数据含义的说明，数据的含义称为数据的语义，数据与其语义是不可分的。**数据架构演化**：单机 mysql，memcached+mysql+垂直分离，mysql 主从读写分离，分库分表+水平拆分+mysql 集群，nosql。**单机 MySQL 瓶颈**：1、数据量过大，单机存储不下 2、数据量过大，单机内存存不下数据索引，索引是加速数据库访问效率的一种机制，存储于内存，数据量越大，索引越大，内存消耗越多，每当加入数据，都需要维持索引，不做优化，数据访问将非常缓慢。3、访问量过大，在读写混合的情况下，一个实例不能承受。**Memcached+MySQL+垂直分离**：随着访问量的上升，几乎大部分使用 MySQL 架构的网站在数据库上都出现性能问题，程序员们开始大量的使用缓存技术来缓解数据库的压力。最开始比较流行的是通过文件缓存来缓解数据库的压力，但是当访问量继续增大的时候，多台 web 服务器通过文件缓存不能共享，大量的小文件缓存也带来了比较高的 IO 压力 Memcached(缓存)蕴意而生，管理缓存。缓存原则：频繁被访问的数据可以被放置于缓存当中，以供频繁访问。垂直分离指按业务划分将不同的数据放在不同的数据库里。**MySQL 主从读写分离**：主从复制：容灾备份，缓存备份，保证数据的完整性。读写分离：增删改是写，查为读。“读”的职能允许从被查询的数据库服务器读数据。“写”的职能允许向写数据的数据库服务器写数据。分工明确，结合缓存能实现性能的一大提升。**分库分表+水平拆分+MySQL 集群**：原因：数据库中的数据量不一定是可控的，在未进行分库分表的情况下，随着时间和业务的发展，库中的表会越来越多，表中的数据量也会越来越大，相应地，数据操作，增删改查的开销也会越来越大。另外，由于无法进行分布式部署，而一台服务器的资源（CPU、磁盘、内存、IO 等）是有限的，最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈。**NoSQL=Not Only SQL**：云时代对 DB 的新需求：高性能：对数据库高并发读写的需求，数据库并发负载非常高，往往要达到每秒上万次读写请求，硬盘 IO 无法承受。海量存储：对海量数据的高效率存储和访问的需求对海量数据进行 SQL 查询，效率极其低下不可忍受。高伸缩性与高可用性：对数据库的高可扩展性和高可用的需求，无法通过简单添加更多的硬件和服务节点来扩展性能和负载能力，需要停机维护和数据迁移。**数据层非功能需求与架构技术**：存储高性能，读写分离、数据缓存、分库分表、NoSQL；存储高可用主从、CAP 理论；存储高扩展，分库分表、NoSQL。**读写分离基本原理**：将数据库读写操作分散到不同的节点上，数据库服务器搭建主从集群，一主一从、一主多从都可以。数据库主机负责读写操作，从机只负责读操作。数据库主机通过复制将数据同步到从机，每台数据库服务器都存储了所有的业务数据。业务服务器将写操作发给数据库主机，将读操作发给数据库从机。**读写分离复制延迟**：主从复制延迟问题，如果业务服务器将数据写入到数据库主服务器后立刻进行读取，此时读操作访问的是从机，主机还没有将数据复制过来，到从机读取数据是读不到最新数据的，业务上就可能出现问题。**应对复制延迟的方案**：1、写操作后的读操作指定发给数据库主服务器，例如，注册账号完成后，登录时读取账号的读操作也发给数据库主服务器 2、读从机失败后再读一次主机 3、关键业务读写操作全部指向主机，非关键业务采用读写分离。**读写分离分配机制**：程序代码封装、中间件封装。**主备复制基本实现逻辑**：1、主机存储数据，通过复制通道将数据复制到备机 2、正常情况下，客户端无法读写操作，都发送给主机，备机不对外提供任何读写服务。3、主机故障情况下，客户端不会自动将请求发给备机，此时整个系统处于不可用状态，产能读写数据。但数据并没有全部丢失，因为备机上有数据。4、如果主机能够恢复（人工或自动），客户端继续访问主机，主机继续将数据复制给备机。5、如果主机不能恢复，则需要人工升级备机为主机，增加新备机，切换客户端访问新主机。成功写入主机但还没有复制到备机的数据会丢失，需要人工进行排查和恢复。也许有的数据因主机宕机丢失了，业务上需要考虑如何应对这种丢失风险。7、如果主备数据复制延迟，由于备机并不对外提供读写操作，主机的业务没有影响。但如果延迟太多，则此时主机宕机了，则可能丢失较多数据，因此对于复制延迟也不能掉以轻心。一般的做法是做复制延迟的监控措施，当延迟数据量较大时及时预警。由人工干预处理。注意，在此类方案中，备机只起到备份作用。**主备复制优点**：对于客户端来说，不需要感知备机的存在，即使宕难恢复后，原来的备机被人工修改为主机后，对于客户端来说，只是认为主机的地址发生了，无需知道是原来的备机升级为主机了。对于主机和备机来说，双方只需要进行数据复制即可，无须进行状态判断和主备倒换等复杂操作。**缺点**：备机仅是备份，不提供读写操作，硬件浪费。故障后需人工干预，无法自动恢复。**主从复制基本实现逻辑**：1、主机存储数据，通过复制通道将数据复制到从机。2、正常情况下，客户端写操作发送给主机，读操作可发送给主机也可以发送给从机(可以随机读，轮询读，只读主机)。3、主机故障情况下，客户端无法进行写操作，但可以将读操作发送给从机，从机继续响应读操作。此时和写操作相关的业务不可用，但是读操作相关的不受影响。4、如果主机能够恢复（人工或自动），客户端继续将写操作请求发送给主机，主机继续将数据复制给从机。5、如果主机不能恢复，则需要人工升级从机为主机，然后让客户端访问新主机，同时，为了继续保持主从结构，人工增加新机器作为从机。6、主机不能恢复的情况下，成功写入了主机但是还没有复制到从机的数据会丢失，需要人工进行排查和恢复，也许有的数据就永远丢失了，业务上需要考虑如何应对此类风险。7、如果主从间数据复制延迟，由于备机并不对外提供读写操作，主机的业务没有影响。但如果延迟太多，则此时主机宕机了，则可能丢失较多数据，因此对于复制延迟也不能掉以轻心。一般的做法是做复制延迟的监控措施，当延迟数据量较大时及时预警。**主从复制基本实现逻辑**：1、主机存储数据，通过复制通道将数据复制到从机。2、正常情况下，客户端写操作发送给主机，读操作可发送给主机也可以发送给从机(可以随机读，轮询读，只读主机)。3、主机故障情况下，客户端无法进行写操作，但可以将读操作发送给从机，从机继续响应读操作。此时和写操作相关的业务不可用，但是读操作相关的不受影响。4、如果主机能够恢复（人工或自动），

机器上，便于管理；便于实现动静分离、冷热分离的数据库表的设计模式；数据维护简单缺点：部分业务表无法关联（Join），只能通过接口方式解决，提高了系统的复杂度；受每种业务的不同限制，存在单库性能瓶颈，不易进行数据扩展和提升性能；事务处理复杂。**水平切分**不是将表进行分表，而是将其按照某个字段的某种规则分散到多个库中，在每个表中包含一部分数据，所有表加起来是全量数据。简言之，将数据按一定规律，按行切分，并分配到不同的库表里，表结构完全一样。优点：单库单表的数据保持在一定的量级，有助于性能的提高；切分的表的结构相同，应用层改造较少，只需要增加路由规则即可；提高了系统的稳定性和负载能力缺点：切分后，数据是分散的，很难利用数据库的 Join 操作。跨库 Join 性能差；拆分规则难以抽象；分片事务的一致性难以解决；数据扩容的难度和维护量极大。**垂直水平共同点**：存在分布式事务的问题；存在跨节点 Join 的问题；存在跨节点合并排序、分页的问题；存在多数据源管理的问题；垂直切分更偏向于业务拆分的过程；水平切分更偏向于技术性能指标。**分库分表引起的问题**：(1)扩容与迁移。通用的处理方法：1、按照新旧分片规则，对新旧数据库进行双写 2、将双写前按照旧分片规则写入的历史数据，根据新分片规则迁移写入新的数据库 3、将按照旧的分片规则查询改为按照新的分片规则查询 4、将双写数据库读操作按照旧分片规则写入数据 5、删除按照旧分片规则写入的历史数据。数据一致性问题：由于数据量大，通常会造成不一致问题，因此，通常先清理旧数据，洗完后再迁移到新规则的新数据库下，再做全量对比。还需要对比评估迁移过程中是否有数据更新，如果有需要迭代清洗，直至一致。如果数据量巨大，无法全量对比，需要抽样对比，抽样特征需要根据业务特点进行选取。注意：线上记录迁移过程中的更新操作日志、迁移后根据更新日志与历史数据共同决定数据的最新状态，以达到迁移数据的最终一致性。动静数据分离问题：对于一些动静敏感的数据，如交易数据，最好将动静数据分离。选取时间点对静历史数据进行迁移。(2)查询问题。查询问题解释：在分库分表以后，如果查询的标准是分片的主键，则可以通过分片规则再次路由并查询，但是对于其他主键的查询、范围查询、关联查询、查询结果排序等，并不是按照分库分表维度来查询的。**查询问题的解决方案：**1、在多个分片表查询后合并数据集（效率很低）2、按查询需求定义多分片维度，形成多张分片表（空间换时间）3、通过搜索引擎解决，如果有实时要求，还需要实时搜索。（难度大）(3)分布式事务，一致性（4）同组数据跨库 **缓存的概念**：用于存储数据的硬件或软件组件，以使得后续更快访问响应的数据。缓存中的数据可能是提前计算好的结果、数据的副本等。**缓存的作用**：主要解决高并发，热点数据访问的性能问题。提供高性能的数据快速访问。**本地缓存**指的是在应用中的缓存组件，其最大的优点是应用和 cache 是在同一个进程内部，请求缓存非常快速，没有过多的网络开销等，在单应用不需要集群支持或者集群情况下各节点无需互相通知的场景下使用本地缓存较合适；**本地缓存的缺点**：因为缓存跟应用程序耦合，多个应用程序无法直接的共享缓存，各应用或集群的各节点都需要维护自己的单独缓存，对内存是一种浪费。应用场景：缓存字典等常用数据。缓存介质：硬盘缓存：将数据缓存到硬盘，减少网络传输的消耗；内存缓存：直接将数据存储到本级内存中，通过程序直接维护缓存对象。**分布式缓存**指的是与应用分离的缓存组件或服务，其最大的优点是自身就是一个独立的应用，与本地应用隔离，多个应用可直接的共享缓存。应用场景：缓存经过复杂运算得出的数据；缓存存储系统中频繁访问的热点数据，减轻存储系统压力。**反向代理缓存**：反向代理位于应用服务器机房，处理所有对 WEB 服务器的请求。如果用户请求的页面在代理服务器上有缓冲的话，代理服务器直接将缓冲内容发送给用户。如果没有缓冲则先向 WEB 服务器发出请求，取回数据，本地缓存后再发送给用户。通过降低向 WEB 服务器的请求数，从而降低了 WEB 服务器的负载。应用场景：一般只缓存体积较小的静态文件资源，如 css、js、图片。**CDN：内容分发网络**：通过在现有互联网中增加一层新的网络架构（CDNS），将网站内容发布到最接近用户的网络'边缘'，使用户可以就近取得所需的内容。CDN 目标：解决由于网络带宽小、用户访问量大、网点分布不均等原因所造成的用户访问网站响应速度慢的问题。CDN：将一个服务器的内容平均分布到多个服务器上；智能识别服务器，让用户获取离用户最近的服务器，提高访问速度。分布式存储、负载均衡，网络请求的重定向，内容管理。**命中率：**命中率=返回正确结果数/请求缓存次数，命中率问题是缓存中的一个非常重要的问题，它是衡量缓存有效性的重要指标。命中率越高，表明缓存的使用率越高。**最大元素**缓存中可以存放的最大元素的数量，一旦缓存中元素数量超过这个值，那么将会触发缓存启动清空策略。根据不同的场景合理的设置最大元素值往往可以一定程度上提高缓存的命中率，从而更有效的使用缓存。**HTTP 再验证**原始服务器的内容可能会发生变化，缓存要不时对其进行检测，看看它们保存的副本是否仍是服务器上最新的副本，进行"新鲜度检测"。缓存可以在任意时刻，以任意的频率对副本进行再验证。但由于缓存中通常会包含数百万的文档，而且从部分缓存只有在客户端发起请求，并且副本旧得足以需要检测的时候，才会对副本进行再验证。为了有效地进行再验证，HTTP 定义了一些特殊的请求，不用从服务器上获取整个对象，就可以快速检测出内容是否最新的。HTTP 为我们提供了几个用来对已缓存对象进行再验证的工具，但最常用的是 If-Modified-Since 首部。将这个首部添加到 GET 请求中去，就可以告诉服务器，只有在缓存了对象的副本之后，又对其进行了修改的情况下，才发送此对象。**服务器收到 GET If-Modified-Since 请求时的 3 种情况**：再验证命中或缓慢命中：如果服务器对象未被修改，服务器会向客户端发送一个小的 HTTP 304 Not Modified 响应，只要缓存知道副本仍然有效，就会再次将副本标记为暂时新鲜的，并将副本提供给客户端。再验证未命中：如果服务器对象与已缓存副本不同，服务器向客户端发送一条普通的、带有完整内容的 HTTP 200 OK 响应，对象被删除：如果服务器对象已经被删除了，服务器就回送一个 404 Not Found 响应，缓存也会将其副本删除。**缓存清空策略**：FIFO(first in first out)：先进先出策略，最先进入缓存的数据在缓存空间不够的情况下，会被优先被清除掉。以腾出新的空间接受新的数据。策略算法主要比较缓存元素的创建时间。在数据实效性要求场景下可选择该类策略，优先保障最新数据可用。LFU(less frequently used)：最少使用策略，无论是否过期，根据元素的被使用次数判断，清除使用次数较少的元素释放空间。策略算法主要比较元素的命中次数。在保证高频数据有效性场景下，可选择这类策略 LRU(least recently used)：最近最少使用策略，无论是否过期，根据元素最后一次被使用的时间戳，清除最远使用时间戳的元素释放空间。策略算法主要比较元素最近一次被 get 使用时间。在热点数据场景下较适用，优先保证热点数据的有效性。数据缓存的更新策略：Cache Aside（旁路缓存）：失效：应用程序先尝试从缓存（cache）中取数据。如果缓存中没有，则从数据库取数据，然后放入缓存中，确保下次可以直接从缓存读取。命中：如果缓存中有数据，直接取出返回，不用访问数据库。更新：在更新数据时，先把数据写到数据库中，成功后再让缓存中的数据失效，下次再查询时会从数据库中重新加载。这个方法保证数据的准确性，因为数据库的数据是最新的，缓存只是一个暂时存储。如果数据发生更新，缓存会被清空，让新的数据重新加载进缓存。Read/Write Through（读/写通路）：Read Through：在查询操作中自动更新缓存。如果缓存失效了，缓存系统会自己去数据库取数据，然后更新缓存。应用程序始终从缓存中读取。Write Through：更新数据时，应用会同时更新缓存和数据库，确保缓存和数据库的数据一致。这种方法的好处是数据在缓存和数据库中总是一致的，避免了数据不一致的问题。Write Behind（写回缓存）：当更新数据时，先更新缓存，不直接更新数据库。然后把更新的操作批量地'写回'数据库。这样做的好处是数据库的操作压力会减少，因为数据更新的操作被延迟并批量处理。但是，这种方法会导致数据短时间内不一致。如果缓存系统发生故障，可能会丢失未写入数据库的数据。本地缓存实现方法：编程直接实现缓存 局部缓存（以成员变量实现）和静态缓存（以静态变量实现）**Ehcache** 是一种流行的 Java 开源缓存框架 **平滑迁移**的具体过程主要通过双写、切读、迁移历史数据和关闭旧集群的步骤来实现。最初，应用服务器同时写入旧集群和新集群，旧集群按原先的规则（如 Hash(key)%2）进行读写，而新集群则按新的分片规则（如 Hash(key)%4）进行写操作。这时，旧集群继续正常读写，新集群开始接收写入但不参与读取。接下来，逐步关闭部分新集群的写操作，确保部分数据只写入旧集群，从而减少新集群的写入压力。此时，应用服务器仍读取旧集群的数据，保证数据一致性。与此同时，历史数据从旧集群迁移到新集群，确保新集群能够拥有完整的数据副本。当历史数据迁移完成后，进行切读操作。应用服务器同时从新旧集群进行读写操作。确保新集群的读取功能开始生效，而旧集群仍然继续支持读操作，保证数据的可用性和一致性。最后一步是关闭双写，停止旧集群的写操作，此时应用服务器完全依赖新集群进行读写操作，旧集群不再参与写入过程。随着旧集群逐步下线，整个迁移过程结束，系统平滑地切换到新集群，完成了分布式缓存的平滑迁移。**一致性哈希算法**通过将整个哈希值空间组织成一个虚拟的环（哈希环），哈希值的范围从 0 到 \(2^{32}-1\)，整个空间按顺时针方向排列。每个服务器节点通过哈希函数计算出其在哈希环上的位置，通常使用服务器的 IP 或主机名作为输入。数据项（如缓存对象）也使用相同的哈希函数，计算出其在哈希环上的位置，并顺时针找到第一个服务器节点，数据便存储在该节点上。一致性哈希的优势在于，当节点发生增删时，只需重新分配少量的数据。例如，如果一个节点失效，只有那些顺时针最接近该节点的数据需要重新分配给下一个节点。而如果增加一个节点，只有新节点负责的少部分数据需要从其他节点迁移过来，其他数据不会受到影响。通过这种方式，一致性哈希保证了较高的容错性和扩展性。然而，当服务节点较少时，可能会出现数据分布不均的倾斜问题，为了解决这种问题，引入了虚拟节点机制。每个物理节点通过计算多个哈希值，映射到环上的多个位置，这样可以使数据更加均匀地分布在各个节点上，降低数据倾斜的风险。**停机迁移**的过程是停止应用后迁移历史数据，修改数据源指向新缓存集群，最后重启应用。缓存的问题 **数据一致性** 由于缓存是持久化数据的副本，因此不可避免地会出现数据不一致的问题，如读取不到缓存数据的情况。数据不一致通常是由于网络不稳定或节点故障导致的，常见的场景与解决方案：先写缓存，再写数据库：缓存写入成功，但数据库写入失败或延迟，导致下次读取时出现脏读。解决方案是将写操作改为先写数据库再写缓存的方式。先写数据库，再写缓存：数据库写入成功，但缓存写入失败，导致下次读取时无法读取到数据。解决方案是根据写入缓存的成功与否进行判断，若缓存写入失败则进行重试或异步写入。缓存异步刷新：指数据库操作和写缓存不同步，操作频繁时不能及时刷新缓存。解决方案是根据缓存中刷新数据的时间间隔，进行相应的操作。**缓存穿透**是指使用不存在的 key 进行大量高并发查询，导致每次请求都穿透到后端数据库系统，增加数据库压力甚至导致崩溃。解决方案：通常将空值缓存在缓存中，后续相同查询直接返回，避免穿透。对恶意查询可以设置查询条件规则，直接拒绝不符合条件的请求。**缓存并发**问题通常发生在高并发场景下，当某个缓存 key 过期时，多请求同时发现缓存过期，导致同时查询数据库并将结果写入缓存，增加数据库负载，导致性能下降甚至崩溃。分布式锁：确保同一 key 只有一个线程查询后端服务，其他线程等待锁释放，减轻并发压力。本地锁：通过限制一个线程访问数据库，其他线程等待，适用于单服务场景，较过期：设置缓存数据的失效时间，发现过期时延长其有效期，继续使用旧数据，避免并发。**缓存雪崩**是指大量缓存集中在某一时间内失效，导致业务系统需要重新生成缓存，瞬时增加数据库压力甚至导致数据库崩溃。解决方案：更新锁机制：对缓存更新操作进行加锁保护，确保只有一个线程能更新缓存。失效时间分片：对不同数据设置不同的失效时间，避免集体失效。后台更新机制：通过后台线程定时更新缓存，减少业务线程的压力。**缓存的高可用性**需要根据业务实际场景来定，不是所有业务都需要缓存的高可用，具体方案需结合业务设计。解决方案：分布式缓存：实现数据的海量缓存。复制缓存：实现缓存节点的高可用。缓存热点一些特定热点数据由于高并发访问，导致缓存服务器压力过大。解决方案：复制多份缓存副本，将请求分散到多个缓存服务器，减轻单台服务器压力。**创建型（隐藏创建逻辑）**：工厂、抽象工厂、单例、建造者、原型 **结构性（类和对象的组合）**：适配器、桥接、过滤器、组合、装饰器、外观、享元、代理 **行为型（关注对象之间的通信）**：中介者、观察者、访问者 J2EE关注表示层 **开闭原则**：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，需要使用接口和抽象类。**里氏代换原则**：LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。**依赖倒转原则**：这个原则是开闭原则的基础。针对接口编程，依赖于抽象而不依赖于具体。**接口隔离原则**：使用多个隔离的接口，比使用单个接口要好。客户端不应该依赖它 不需要的接口。降低类之间的耦合度。建立单一接口，尽量细化接口。接口中的方法尽量少。过大的话会增加耦合性，而过小的话会增加复杂性和开发成本。**迪米特法则**：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。降低系统的耦合度，使类与类之间保持松耦合状态。**合成复用原则**：复用类通过'继承'和"合成"两种方式来实现。尽量使用合成/聚合的方式，而不是使用继承。 继承的优点：容易实现并且容易修改和扩展继承来的内容。 继承的缺点：增加了类之间的依赖，继承是属于"白箱"复用，父类对子类来说是透明的，这破坏了类的封装性。合成复用存在的缺点就是在系统中会存在较多的对象需要管理。**单例模式**：只能有一个实例；自己创建自己的唯一实例；给所有其他对象提供这一实例。懒汉式、饿汉、双检锁、登记式 优点：只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例，避免对资源的多重占用。 缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。使用场景：要求生产唯一序列号。WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接。**工厂模式**是一种创建型设计模式，通过定义一个创建对象的接口，让子类决定实例化哪一个类，使对象的创建过程延迟到子类中。这种模式主要用于解决选择接口实现的问题，使得客户端无需知道对象的具体实现类，而只需使用工厂接口即可创建对象。优点：一个调用者想创建一个对象，只要知道其名称就可以了。扩展性高，如果想增加一个产品，只要扩展一个工厂类即可。屏蔽产品的具体实现，调用者只关心产品的接口。缺点：每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加。在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。使用场景：需要生成大量具有相同接口或基类的对象，但在不同条件下实例化不同子类时。日志记录器、数据库访问器、协议处理器等。**抽象工厂模式**通过定义一个创建一系列相关或相互依赖对象的接口，允许客户端在不指定具体类的情况下创建对象。这种模式常用于创建不同系列的产品，如同一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。 缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。模式结构抽象工厂（AbstractFactory）：定义创建一系列产品的接口。具体工厂（ConcreteFactory）：实现抽象工厂的接口，负责实例化具体的产品。抽象产品（AbstractProduct）：定义产品的接口。具体产品（ConcreteProduct）：实现抽象产品的接口。使用场景：当一个系统需要独立于其产品的创建、组合和表示时。当一个系统需要一系列相互关联或相互依赖的产品配合使用时。**适配器模式**主要用于将一个类的接口转换成客户希望的另一个接口，使原本由于接口不兼容而不能一起工作的类可以在一起工作。适配器模式通常作为两个不兼容的接口之间的桥梁，以满足新的环境需求。适配器模式的核心概念目标接口（Target Interface）：客户端期望的接口。适配器（Adapter）：将适配者的接口转换成目标接口，以满足客户端的需求。适配者（Adaptee）：需要适配的类，它的接口与目标接口不兼容。客户端（Client）：调用目标接口的对象。优点：可以让任何两个没有关联的类一起运行。提高了类的复用，增加了类的透明度。灵活性好。缺点：过多地使用适配器，会让系统非常零乱，不易整体进行把握。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。使用场景 当需要使用现有的类，但类的接口不符合需求时。当想要创建一个可以重用的类，并希望能够与其他不相关的类协同工作时。通过接口转换，将一个类插入到另一个类系统中。**桥接模式**通过将抽象部分与实现部分分离，使它们可以独立变化。桥接模式的主要意图是解耦，让抽象与实现不再是固定绑定的关系，从而提供更灵活的扩展方式。Abstraction（抽象化角色）：定义抽象类，包含对实现化对象的引用。Refined Abstraction（扩展抽象化角色）：是抽象化角色的子类，实现父类中的业务方法，并通过组合的关系调用实现化角色中的业务方法。Implementor（实现化角色）：定义实现化角色的接口，供扩展抽象化角色调用。Concrete Implementor（具体实现化角色）：实现 Implementor 接口，提供具体实现。优点：抽象和实现的分离。优秀的扩展能力。实现细节对客户透明。缺点：桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。使用场景：如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。**代理模式**：通过创建一个代理类，控制客户端对目标对象的访问。代理模式的目标是提供一个中间对象，以控制对其他对象的访问。这在客户端不适合直接访问目标对象，或出于安全性、性能、内存管理等原因时特别有用 静态代理优点可以做到在不修改目标对象的功能前提下，对目标功能扩展。缺点 因为代理对象需要与目标对象实现一样的接口，所以会有很多代理类,类太多。同时，一旦接口增加方法，目标对象与代理对象都要维护。JDK 动态代理：虽然相对于静态代理，动态代理大大减少了我们的开发任务，同时减少了对业务接口的依赖，降低了耦合度。但是 JDK 自带动态代理只能支持实现了 Interface 的类.优点：代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；代理对象可以扩展目标对象的功能；例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要打开已经封装好的委托类。代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度；缺点：在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；增加了系统的复杂度；使用场景：远程代理、虚拟代理、Copy-on-Write 代理、保护代理、Cache 代理、防火墙代理、同步化代理、智能引用代理 **中介者模式** 用于定义一个中介对象来封装一组对象之间的交互，使原有对象之间的耦合松散，通过中介者，每个对象不需要显式地引用其他对象，能够独立地修改它们之间的交互。抽象中介者（Mediator）角色：它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。具体中介者（ConcreteMediator）角色：实现中介者接口，定义一个 List 来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色。抽象同事类（Colleague）角色：定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能。具体同事类（Concrete Colleague）角色：是抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互。优点：降低了类的复杂度，将一对多转化成了一对一。各个类之间的解耦。符合迪米特原则。缺点：中介者会庞大，变得复杂难以维护。使用场景：系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。**观察者模式** 指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。抽象目标（Subject）角色：提供一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。具体目标（Concrete Subject）角色：实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。抽象观察者（Observer）角色：是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。优点：观察者和被观察者是抽象耦合的。建立一套触发机制。 缺点：如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。**访问者**：将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。抽象访问者（Visitor）角色：定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作 visit()，其参数类型标识了被访问的具体元素。具体访问者（ConcreteVisitor）角色：实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么。抽象元素（Element）角色：声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数。具体元素（ConcreteElement）角色：实现抽象元素角色提供的 accept() 操作，其方法体通常都是 visitor.visit(this)，另外具体元素中可能还包含本身业务逻辑的相关操作。对象结构（Object Structure）角色：是一个包含元素角色的容器，提供让访问者对象遍历容器中的所有元素的方法，通常由 List、Set、Map 等聚合类实现。优点：符合单一职责原则。优秀的扩展性。灵活性。缺点：具体元素对访问者公布细节，违反了迪米特原则。具体元素变更比较困难。违反了依赖倒置原则，依赖了具体类，没有依赖抽象