分布式服务的变革的原因 大型应用中业务需求的爆炸式增长 技术的不断演进导致系统异构化严重 业务与技术的沟通存在的鸿沟 **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。纵向拆分**按照业务进行梳理拆分,最终根据业务特性把应用拆开,不同的业务模块独立部署 **横向拆分** 将核心的、公共的业务拆分出来，通过分布式服务框架对业务进行服务化，消费者通过标准的契约来消费这些服务。服务提供者独立打包、部署和演进，与消费者解藕。本质上是把各个服务中公共和核心的部分拆分出来,各个核心直接去调用公共的业务,通过分布式服务框架(ESB或者服务网关) **服务治理要解决的主要问题如下**生命周期的管理 服务容量规划 运行时治 服务安全 **传统并行计算框架 MapReduce 集群架构/容错性** 共享式(共享内存/共享存储)容错性差 非共享式容错性好 **硬件/价格/扩展性** 刀片服务器、高速网、SAN， 价格贵，扩展性差 普通 PC 机便宜，扩展性好 **编程/学习难度** what-how，难 what，简单 **适用场景** 实时、细粒度计算、计算密集型 批处理非实时、数据密 集型 **MapReduce 模型简介** 抽象两个函数 Map 和 Reduce 编程容易 MapReduce 采用"分而治之"策略 设计理念是"计算向数据靠拢" MapReduce 框架采用了 Master/Slave 架构包括一个 Master 和若干个 Slave。Master 上运行 JobTracker，Slave 上运行 TaskTracker **MapReduce 体系结构**主要由四个部分组成，分别是：Client 、JobTracker、TaskTracker 以及 Task **client** 提交到 JobTracker 端查 看作业运行状态 **JobTracker** 负责资源监控和作业调度 健康状况跟踪任务的执行进度、资源使用量等信息告诉任务调度器（TaskScheduler） **TaskTracker** 周期性地通过"心跳" 使用"slot"等量划分本节点上的资源量一个 Task 获取到一个 slot 后才有机会运行 **Task** Map Task 和 Reduce Task 两种，均由 TaskTracker 启动 **MapReduce 工作流程概述** 不同的 Map Reduce 任务之间不会进行通信 用户不能显式地从一台机器向另一台机器发送消息 所有的数据交换都是通过 MapReduce 框架自身去实现的 **大概描述** 先放文件然后 split 切分一般一个切分片一个 mapslot Record Reader 解析成键值对(RR)后交给 Map,最后 map 完之后进行 shuffle,这个 shuffle 是 map 后放在缓存,然后缓存有一个溢写机制,也就是百分之多少会持久化写入多个分区最后文件在磁盘归并,归并后的数据作为 reduce 的输入(合并<'a',2>,归并<'a',<1,1>>) Reduce 任务通过 RPC 向 JobTracker 询问 Map 任务是否已经完成，若完成，则领取数据 •Reduce 领取数据先放入缓存，来自不同 Map 机器，先归并，再合并，写入磁盘 •多个溢写文件归并成一个或多个大文件，文件中的键值对是排序的 **MapReduce 程序的流程及设计思路** job，信息发给 Job Tracker ob Tracker 是框架的中心，定时通信管理 TaskTracker 是 MapReduce 集群中每台机器都有的部分，它主要监视自己 所在机器的资源情况 TaskTracker 需要把这些信息通过 heartbeat 发送给 JobTracker JobTracker 会搜集这些信息以给新提交的 job 分配运行机器 **主要的问题集中** JobTracker 是 MapReduce 的集中处理点，存在单点故障 完成了太多的任务，造成了过多的资源消耗 以 map/reduce task 的数目作为资源的表示过 于简单，没有考虑到 cpu/ 内存的占用情况 把资源强制划分为 map task slot 和 reduce task slot，如果当系统中只有 map task 或者只有 reduce task 的时 候，会造成资源的浪费 **MapReduce on YARN** ResourceManager 负责资源管理，NodeManager 负责资源监控和管理，而 ApplicationMaster 则负责具体任务的调度和运行 每个 MapReduce 任务会有独立的 ApplicationMaster 资源分配是动态的，ResourceManager 可以灵活分配资源 任务管理分给各个 ApplicationMaster，减少了单点故障的风险 **DNS 负载均衡**的核心原理是将一个域名解析为多个 IP 地址（服务器地址），当用户请求访问该域名时，DNS 服务器会将这些 IP 地址轮流或按某种策略返回给用户的 DNS 查询请求，使得用户最终访问到不同的服务器。这种方式在 DNS 解析阶段完成负载均衡，从而将请求分发到多个服务器上。**优点**：简单、成本低就近访问，提升访问速度 **缺点：**更新不及时:DNS 缓存的时间比较长 扩展性差：DNS 负载均衡的控制权在域名商那里，无法根据业务特点针对其做更多的定制化功能和扩展特性 分配策略比较简单：衡支持的算法少 **硬件负载均衡** 啥都好就是贵和扩展性差,无法定制 **软件负载均衡** Nginx 是 7 层负载均衡，LVS 是 Linux 内核的 4 层负载均衡 **优点** 简单:部署维护 便宜:只需 linux 服务器 灵活:定制 **缺点**性能一般 不具备防火墙和防 DDoS 攻击 **典型** 地理级别负载均衡：当用户访问时，DNS 会根据用户的地理位置来决定返回哪个机房的 IP 集群级别负载均衡：F5 收到请求后，进行集群级别的负载均衡机器级别的负载均衡：Nginx 收到用户请求后，将用户请求发送给集群里面的某台服务器 **负载均衡算法** 任务数平分类：轮询或者加权轮询(服务器差异,简单但服务器状态差异) 负载最低优先 连接数或者发 HTTP 请求 复杂度要增加很多 不同业务最优的时间间隔是不一样的，效果不好 性能最优优 站在客户端这边模拟请求但也花费了很多时间 Hash 类 **源地址 Hash** 基本同一台服务器，保持会话状态 **ID Hash** 适合临时性的、需要多次访问的 ID，例如用户的 session ID **消息队列**面向消息的中间件服务的重要类别 持久异步通信 为消息提供中间形态的存储容量 不需要发送方或接收方在消息传输期间保持活跃状态 支持允许消息传输的时间开销 **属性** 异步交互客户端和服务器之间松耦合，通过消息队列进行通信。可靠服务交 付消息被存储在持久化存储中，确保在系统或网络故障后，消息不会丢失，能够保证消息的可靠传递 通过中间消息服务器处理消息 中间消息服务器可以对消息进行过滤、转换、记录等操作，形成一个消息服务器网络，以便消息的传递和管理支持数据库集成 **消息队列基本理念** 在特定的队列中插入消息来进行通信 消息转发过程经过服务器 接收方可以不在线 发送方和接收方无需同步 消息送达保证(失踪在队列里面) **关键属性 作为发送和接收之间的命名消息目的地**：消息队列是一个有名称的消息目的地，用于在发送者和接收者之间充当中介。松散耦合的通信 **允许进程独立执行和失败**：消息队列允许发送和接收的进程独立运行，即使一方出现故障，另一方仍可以继续运行。**可以掩盖进程失败和通信失败**：通过消息队列，系统可以在某一部分失败或中断时继续运行，因为消息会保留在队列中，直到接收方能够处理，从而掩盖了进程或通信上的失败。**传递模式 Point-to-Point 专一性：**每条消息只能被一个消费者消费，确保消息不会被重复处理 **Publish/Subscribe** 广播机制 **松耦合**生产者和消费者之间不直接关联 **分布式服务框架通常包括两大功能**：服务治理中心和服务注册中心 **服务订阅分布** 配置化发布和引用服务 服务自动发现机制 服务在线注册和去注册 **集群容错** Failover Failback Failfast **幂等性** 也就是多次请求其实是同一操作但是会增加对服务器的访问量,增加负载 **数据库的幂等性** 查询类动作 基于主键的非计算式 Update 基于主键的 Delete 具备幂等性**解决** 数据库加锁法 全局唯一 ID 法 去重表法 多版本控制法 状态机控制法 **微服务**是一种架构设计模式拆分成一系列小而松散耦合的分布式组件，共同构成了较大的应用。每个组件都被称为微服务。每个组件单独,连接使用 GRPC 或者 Restful **SOA** 使用 ESB 连接 微服务使用点对点的 **Restful/API-网关/消息代理方式—消息队列的解耦 微服务发现:**客户端发现 客户端发现模式后返回一个服务注册表客户端使用负载均衡算法计算 服务端发现模式客户端通过负载均衡器向某个服务提出请求， 负载均衡器查询服务注册表，并将请求转发到 可用的服务实例 **微服务架构的注册** 自注册机制 服务实例负责在服务注册表中注册和注销 第三方注册模式 服务实例则不需要向服务注册表注册；相反，被称为服务注册器的另一个系统模块会处理。➔微服务的数据也会去中心化 docker 引擎是一个 c/s 结构的应用 l Server 是一个常驻进程 l REST API 实现了 client 和 server 间的交互协议 虚拟机的 Hypervisor 层被 Docker Engine 层所替代没有 Guest OS 更快 （Image）为只读模板 就是一堆只读层（read-only layer）的统一视角可互相公用 容器(container) 容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。**Docker 的调度工具(流行的容器编排系统) Swarm** 一个 swarm 主机相当于 masterNode,然后通过 docker line 和 DockerDaemon 调度(random,最少最多三种)其他的 docker 主机 **Mesos 工作流程**：**资源共享层**：Mesos 在集群的底层提供了一个轻量级的资源共享层，这个层允许不同的框架（如 Spark 或 Marathon）通过统一接口访问集群资源。**分配与委派**：Mesos 主要负责资源的分配和委派，而不是直接调度。各框架会按照自己的需求向 Mesos 请求资源，并根据得到的授权自行调度任务。**Marathon 调度**：作为 Mesos 的调度框架之一，Marathon 通过健康检查、服务发现等功能来管理容器的生命周期。ZooKeeper 则用于 Marathon 和 Mesos 的故障恢复。**Kubernetes 工作流程**：**Pod 与 Service**：Kubernetes 通过 Pod（同地协作的容器组）来封装应用实例，并使用 Service 来管理和暴露这些实例。**调度器**：Kubernetes 的调度器负责寻找未指定 Node 的 Pod，将它们分配到适当的节点上。调度策略包含强制性的谓词（如资源、节点选择等）和优先级（如最少资源请求的节点）。**复制控制器与服务发现**：Kubernetes 使用 Replication Controller 确保指定数量的 Pod 实例运行，并通过 DNS 服务实现服务发现，使 Pod 之间可以通过名称互相通信。**数据+语义+逻辑=业务 代码+业务=软件应用系统 1. 单机 MySQL** 最基础的数据库部署方式，所有数据都存储在一台 MySQL 实例中。适用于数据量小、并发访问量不高的场景，但当访问量增大时会遇到性能瓶颈。**2. Memcached + MySQL + 垂直分离** 引入缓存（如 Memcached）来减轻数据库的读压力，将频繁访问的数据存储在缓存中，以加快查询速度。同时进行垂直分离，即根据不同业务模块，将数据按功能分开存储在不同的 MySQL 实例中，例如用户数据和订单数据分开存储。缓解了单机 MySQL 的压力，但仍有限制，尤其是在写操作频繁的情况下。**3. MySQL 主从读写分离**实现主从架构，主库负责写操作，从库负责读操作，通过读写分离减轻主库的负担。适合读多写少的场景，但随着数据量和并发请求的增加，主库的写入压力依然可能成为瓶颈。**4. 分库分表 + 水平拆分 + MySQL 集群**数据进一步进行水平拆分，将同一个表的数据根据某种规则（如用户 ID、订单 ID 等）分配到不同的库或表中，减小单个库的压力。使用 MySQL 集群管理多个数据库实例，提高系统的扩展性和容错性，适合海量数据的场景。这种架构管理复杂度较高，需要专门的分片策略和集群管理。**NoSQL=Not Only SQL** 当关系型数据库难以满足业务需求时，可以转向 NoSQL 数据库（如 MongoDB、Cassandra）。具有**强大的水平扩展能力 读写分离的延迟问题**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 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 **分配机制程序代码封装**简单直接但是修改行差**中间件封装**简化了代码灵活但增大了开销 **主从/备倒换** 互连式：主备机直接建立状态传递的渠道(主机绵绵不断的给从机发送状态信息) 状态传递通道本身故障了，则备机会主动升级为主机 中介式：在主备机之间引入第三方中介，主备机之间不直接连接，而都 去连接中介，并且通过中介来传递状态信息。谁超时谁成备机 模拟式主备机之间并不传递任何状态数据，而是备机模拟成为一个客户端发送信息但是得到的回应有限 **主主复制**两台机器都是主机，互相将数据复制给对方，客户端可以任 意挑选其中一台进行读写操作。(考虑双向复制不及时可能存在的数据一致性问题) **数据集群** 一个主一群从机多备即多通道，增加了主机的复制压力，同时 增加了对正常读写的压力（实践中，需要考虑 降低该压力）多通道，情况不一，容易导致数据不一致。 多备对单主状态的检测结果不一致，不同的判断和决策。 单主多备，当主机宕机，如何重新选主。 **数据分散集群** 更大规模指多个服务器组成一个集群，每台服务器都会负责存储一部 分数据，同时，为了提升硬件利用率，每台服务器又会备份一部分数据。 一个角色来负责执行数据分配算法独立服务器，如 HDFS 架构也可以是集群选举出的服务器，也称之为"主机"，但职责完全不同(客户端可以向里面任意一个服务器写入)**分库分表**分库分表的本质是数据拆分，是对数据进行分而治之的通用概念 数据拆分主要分为：**垂直拆分(与原结构不同)和水平拆分(与原结构相同) 客户端分片**客户端分片就是使用分库分表的数据库的应用层直接操作分片逻辑，分片规则需要在同一个应用的多个节点间进行同步 一般三种在应用层直接实现，也就是自己写程序进行，通过定制 JDBC 协议实现,通过协议或者中间件 通过定制 ORM 框架实现如 mybatis 的 XML 文件 **代理分片** 代理分片就是在应用层和数据库层之间增加一个代理层，把分片的路由规则配置在代理层，代理层对外提供与 JDBC 兼容的接口给应用层。Cobar 和 Mycat 等 **支持分布式事务的数据库**：分布式数据库需要支持分布式事务，以确保在多个切片上的数据操作能够保持一致性。当前产品如 OceanBase 和 TiDB 对外提供了可伸缩架构和事务支持，简化了开发的复杂性。冷数据是变化更新频率低，查询次数多的数据,垂直切

分放缓存上但是切分后都不易关联,水平切分则是按照规则切(hash 切片—不易聚合但是均匀和时间切片) **扩容和迁移**按照新旧分片规则，对新旧数据库进行双写(双写指的是同时写入新旧数据) 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、将双写数据库逻辑从代码中下线，只按照新的分片规则写入数据 5、删除按照旧分片规则写入的历史数据**数据一致性问题**：数据量太大难以全做对比 **Mycat** 拦截 SQL 并对 SQL 做特定分析然后返回,如果有对应的分片规则也能直接执行 **Sharding-JDBC** Sharding-JDBC 直接封装 JDBC API，可以理解为增强版的 JDBC 驱动，旧代码迁移成本几乎为零,自定义分片规则,流程为分片规则配置,JDBC 规范重写,SQL 解析,SQL 改写,SQL 路由,SQL 执行,结果归并,对外暴露为一个数据库类型,内部修改 **数据缓存** 硬件,以使得后续更快访问响应的数据。提前计算好的结果和数据副本等 **主要解决高并发高性能 缓存基本类型** 本地缓存也就是缓存在客户端本地 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，缓存方式缓存字典等常用数据 Ehcache、 Guava Cache **分布式缓存**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多个应用可直接的共享缓存 Memcached Redis **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。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。解决网络上的问题 **缓存和 http 的命中与更新**,检测是否最新副本但是带宽珍贵需要副本非常久才行➔缓存发送 If-Modified-Since 回答 304 正确不修改 **缓存清洗策略** FIFO, LR(最近)U, LF(频率)U, 时间过期等 **更新策略 cacheside**(应用程序) 先把数据存到数据库中，成功后，再让缓存失效 **Read/Write Through Pattern(**缓存自身操作)写如果命中了缓存，则更新缓存，然后再由 Cache 自己更新数据库（同步）**Write Behind Caching Pattern** 写回法,调用类似于消息队列之类的主要在意数据的一致性 **Ehcache** 本地缓存中间件--两级缓存介质,支持缓存持久化,根据时间刷新的,时间越长缓存越多，内存占用也就越大，内存泄露的概率也越大。分布式缓存迁移和分片迁移一样,双写,读只读旧的,然后迁移数据,然后切读,最后下线双写， **分布式缓存的迁移-一致性哈希** 哈希环,顺时针移动到不同的节点注意考虑一些节点上线下线的影响—少数节点启用虚拟 ip 保证均匀 **缓存问题 数据一致性** 1. 先写缓存，再写数据库 2. 先写数据库，再写缓存 3.缓存异步刷新 **缓存穿透**指的是使用不存在的 key 进行大量的高并发 查询，这导致缓存无法命中，每次请求都要穿透到后 端数据库系统进行查询，使得数据库压力过大，甚至 导致数据库服务崩溃。**通常将空值缓存起来 缓存并发**的问题通常发生在高并发的场景下，当一个 缓存 key 过期时，因为访问这个缓存 key 的请求量较 大，多个请求同时发现缓存过期，分布式锁(数据库一个访问)本地锁(服务节点一个访问数据库)软过期(过期了不立马替换)**缓存雪崩**指缓存服务器重启或者大量缓存集中在某一 个时间段内失效，业务系统需要重新生成缓存，给后 端数据库造成瞬时的负载升高的压力，**失效时间分片机制**后台更新机制,而非业务更新 缓存是否**高可用**(不崩溃-分布式)，需要根据实际的场景而定，并不是 所有业务都要求缓存高可用 **缓存热点**一些特别热点的数据，高并发访问同一份缓存数据， 导致缓存服务器压力过大。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：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。注意事项：getInstance() 方法中需要使用同步锁 synchronize 防止多线程同时进入造成 instance 被多次实例化。工厂在创建对象时不会对客户端暴露创建逻辑，使用一个共同的接口来指向新创建的对象.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：日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。设计一个连接服务器的框架，需要三个协议，"POP3""IMAP""HTTP"可以把这三个作为产品类，共同实现一个接口。注意事项：作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。抽象工厂：接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。优点： 当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。使用场景：QQ 换皮肤，一整套一起换。生成不同操作系统的程序。注意事项：产品族难扩展，产品等级易扩展。适配器模式{继承或依赖（推荐）}涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能 类适配器模式（继承或实现关系），适配者类：它是被访问和适配的现存组件库中的组件接口。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。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。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：有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式。注意事项：适配器不是在详细设计时添加的，而是解决正在服役的项目的问题。桥接模式：抽象化角色、扩展抽象化角色、实现化角色、具体实现化角色。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。注意事项：桥接模式适用于两个独立变化的维度。代理模式：访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。静态代理：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：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。注意事项：不应当在职责混乱的时候使用。观察者：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使用场景：一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。一个对象必须通知其他对象，而并不知道这些对象是谁。需要在系统中创建一个触发链，A 对象的行为将影响 B 对象，B 对象的行为将影响 C 对象，可以使用观察者模式创建一种链式触发机制。注意事项：JAVA 中已经有了对观察者模式的支持类。避免循环引用。如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。访问者：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。使用场景：对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。注意事项：访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。**系统体系结构概述 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。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。关键决策的体现 折中 **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 **系统设计维度**系统规模与复杂度简单复杂 系统开放度封闭开放 模块粒度细粗 关注层面模块连接件 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。**作用**：软件中间件的作用 屏蔽异构性 实现互操作 共性凝练和复用 **Kruchten 4+1** 视图模型 用例视图 逻辑视图 开发视图 进程视图 物理视图