**软件架构** 定义：架构=构件+连接件+拓扑结构+约束+性能 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。**目标和作用：**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：建⽴满⾜最终⽤户要求的系统需求。内向⽬标：建⽴满⾜系统设计者需要以及易于系统实现、维护和扩展的系统构件构成。**作⽤：1.**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;2.可传递和复⽤的模型：在其他项目中可以使用，提⾼代码复⽤率;3.关键决策的体现：这些决策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。**软件⽣命周期**：项⽬规划、需求分析、软件设计、软件实现、测试与评审、维护与升级。**与其他软件活动的比较：**起点模糊：在用户需求尚没有明确时进行，需要交流与反复;高抽象层次：SA分析处理的是高层次的系统构件或子系统之间的关系，而非变量、函数等低层次的概念;分布决策：来自用户、架构师、测试人员等多方面;贯穿全局：SA设计活动在项目开始时进行，但其作用则贯穿整个项目周期，越往后越显出重要性。**架构师观念：**全局观 折中观 交流观 复用观。**软件质量因素：**性能、安全性、易用性、重用性、健壮性、可修改性、可测试性、集成性、移植性、兼容性、经济性、正确性、完备性、其他商业质量。**软件质量：**架构的选择极⼤地影响部分软件质量，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。**架构演化：**（系统=服务集群+中间件）单一服务器，应用服务和数据服务分离，缓存的使用，集群与并发，读写分离，CDN与反向代理，分布式数据库与文件系统，NoSQL，垂直划分业务服务，分布式服务。**软件中间件** 定义：是⼀组应⽤于分布式系统的程序，为系统屏蔽底层通讯并提供公共服务，并保障系统的高可靠性、高可用性、高灵活性。**作用：1.**屏蔽异构性：异构性来源：包括计算机硬件差异、操作系统差异、数据库差异。2.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相对困难；3.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基础功能和结构相似，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。**意义：**缩短开发周期；节约应用程序开发成本；降低运行成本；降低故障率；改善决策；应用系统群集/集成；减少软件维护；提高质量；改进技术；提高产品吸引力。**分类：1.**应用服务类中间件：为应用系统提供一个综合的计算环境和支撑平台；2.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；3.业务架构类中间件：将业务共性抽象至中间件，形成应用模式。**(Kruchten)4+1视图模型**：用例、逻辑、开发、进程、物理视图。**软件设计模式：定义：**最佳的实践、代码编制工程化、开发人员的共同平台。**分类：1.**创建型：在创建对象的同时隐藏创建逻辑的方式，而不是使用new 运算符直接实例化对象；程序在判断针对某个给定实例需要创建哪些对象时更加灵活。2.结构型：关注类和对象的组合，继承的概念被用来组合接口和定义组合对象获得新功能的方式。3.行为型：关注对象之间的通信。4.J2EE模式：关注表示层。**设计模式原则：**1.开闭原则：对扩展开放，对修改关闭；不修改原有的代码，实现一个热插拔的效果，使程序的扩展性好，易于维护和升级；使用接口和抽象类。2.里氏代换原则：任何基类可以出现的地方，子类一定可以出现；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骡就是抽象化，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。3.依赖倒置原则：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。4.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；注意适度原则，一定要适度，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。5.迪米特法则（最少知道原则）：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。6.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。优点:容易实现并且容易修改和扩展继承来的内容；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；在系统中会存在较多的对象需要管理。**创建型模式** **单例模式** 模式意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。解决问题：一个全局使用的类频繁地创建与销毁。使用时机:当你想控制实例数目，节省系统资源的时候。解决方案：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。关键代码:构造函数是私有的。种类：懒汉式（线程不安全和安全）、饿汉、双检锁、登记式/静态内部类。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：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** 模式意图:定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。解决问题:主要解决接口选择的问题。使用时机:明确地计划不同条件下创建不同实例时。解决方案:让其子类实现工厂接口，返回的也是一个抽象的产品。关键代码:创建过程在其子类执行。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** 模式意图:提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。解决问题:主要解决接口选择的问题 使用时机:系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。解决方案:在一个产品族里面，定义多个产品。关键代码:在一个工厂里聚合多个同类产品 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. **结构型模式** （**类、双向）适配器模式** 模式意图:将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作 解决问题:解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的 使用时机:系统需要使用现有的类，而此类的接口不符合系统的需要；想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口；通过接口转换，将一个类插入另一个类系中 解决方案:继承或依赖(推荐) 关键代码:适配器继承或依赖已有的对象，实现想要的目标接口 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。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** 模式意图:将抽象部分与实现部分分离，使它们都可以独立的变化 解决问题:在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。使用时机:实现系统可能有多个角度分类，每一种角度都可能变化。解决方案:把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。关键代码:抽象类依赖实现类 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 角色 抽象化角色 扩展抽象化角色 实现化角色 具体实现化角色 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。对干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 对于两个独立变化的维度，使用桥接模式再适合不过了 **代理模式** 模式意图:为其他对象提供一种代理以控制对这个对象的访问 解决问题:在直接访问对象时带来的问题， (比如对象创建开销很大，或者某些操作需要安全控制或者需要进程外的访问)，直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。使用时机:想在访问一个类时做一些控制。解决方案:增加中间层 关键代码:实现与被代理类组合 [**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** 总结: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) 实现Spring AOP的基础] 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; 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 **行为型模式** **中介者模式** 模式意图:用一个中介对象来封装一系列的对象交互使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互 解决问题:对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理 使用时机:多个类相互耦合，形成了网状结构 解决方案:将网状结构分离为星型结构，关键代码:对象之间的通信封装到一个类中单独处理。角色 抽象中介者角色 具体中介者角色 抽象同事类角色 具体同事类角色 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。**观察者模式** 模式意图:定义对象间的一种一对多的依赖关系，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。解决问题:一个对象状态改变给其他对象通知的问题，，而且要考虑到易用和低耦合，保证高度的协作。使用时机:一个对象(目标对象)的状态发生改变，所有的依赖对象(观察者对象)都将得到通知，进行广播通知。解决方案:使用面向对象技术，可以将这种依赖关系弱化。关键代码:在抽象类里有一个 ArrayList 存放观察者们, 角色类型 抽象目标角色 具体目标角色 抽象观察者角色 具体观察者角色 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:将这些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。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 注意事项:·1、JAVA 中已经有了对观察者模式的支持类。2、避免循环引用。3、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 **访问者模式** 模式意图:主要将数据结构与数据操作分离，解决问题:稳定的数据结构和易变的操作耦合问题,使用时机:需要对一个对象结构中的对象进行很多不同的并目不相关的操作，而需要避免让这些操作"污染"这些对象的类使用访问者模式将这些封装到类中，解决方案:在被访问的类里面加一个对外提供接待访问者的接口关键代码:在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。角色类型：抽象访问者 具体访问者 抽象元素 具体元素 对象结构 优点:符合单一职责原则。优秀的扩展性。灵活性。缺点:·1、具体元素对访问者公布细节，违反了迪米特原则·2、具体元素变更比较闲难 3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1、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。2、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、U1、拦截器与过滤器。**非功能性指标** 性能CPU速度 网络带宽 吞吐量 RT响应时间 网络延时 并发用户数 TPS QPS 扩展性（垂直、水平）可用性 **负载均衡 高性能集群的复杂性** 需要增加一个任务分配器（负载均衡器）；为任务选择一个合适的任务分配算法 **DNS负载均衡** 是最简单也是最常见的负载均衡方式 一般用来实现地理级别的均衡。优点:简单、成本低；就近访问，提升访问速度；缺点:更新不及时；扩展性差；分配策略比较简单；**硬件负载均衡** 通过单独的硬件设备来实现负载均衡功能 这类设备和路由器、交换机类似，目前业界典型的硬件负载均衡设备有两款F5 和 A10性能强劲、功能强大，但价格都不便宜 优点:功能强大；性能强大；稳定性高；支持安全防护；缺点：价格昂贵；扩展能力差；**软件负载均衡** 通过负载均衡软件来实现负载均衡功能 常见的有 Nginx 和 LVS Nginx 是7层负载均衡，LVS 是 Linux 内核的4层负载均衡 4层和7层的区别就在于协议和灵活性 Nginx 支持 HTTP、E-mail 协议 而 LVS 是4层负载均衡，和协议无关，几乎所有应用都可以做 软件和硬件负载均衡方法的最主要区别就在于性能 硬件负载均衡性能远远高于软件负载均衡性能。优点: 简单；便宜；灵活；与硬件负载均衡相比的缺点:性能一般；功能没有硬件负载均衡那么强大；一般不具备防火墙和防 DDoS 攻击等安全功能 **典型架构** 地理级别负载均衡；集群级别负载均衡；机器级别的负载均衡 **负载均衡算法** 任务数平分类（数量、比例）：轮询 负载均衡系统收到请求后，按照顺序轮流分配到服务器上，轮询是最简单的一个策略，无须关注服务器本身的状态，简单”是轮询算法的优点，也是它的缺点。加权轮询 负载均衡系统根据服务器权重进行任务分配 这里的权重一般是根据硬件配置进行静态配置的 加权轮询是轮询的一种特殊形式 其主要目的就是为了解决不同服务器处理能力有差异的问题 但同样存在无法根据服务器的状态差异进行任务分配的问题。负载均衡类（服务器的负载）：负载最低优先 负载均衡系统将任务分配给当前负载最低的服务器 可以根据业务特点来选择指标衡量系统压力 解决了轮询算法中无法感知服务器状态的问题，由此带来的代价是复杂度要增加很多 CPU 负载最低优先的算法要求以某种方式收集每个服务器的 CPU负载 不同业务最优的时间间隔是不一样的，时间间隔太短容易造成频繁波动，时间间隔太长又可能造成峰值来临时响应缓慢。负载最低优先算法基本上能够比较完美地解决轮询算法的缺点 可以感知服务器当前的运行状态 其代价是复杂度大幅上升 看起来很美好，但实际应用场景反而不如轮询(包括加权轮询) 性能最优类（服务器响应时间）：性能最优优先类算法则是站在客户端的角度来进行分配的优先将任务分配给处理速度最快的服务器，通过这种方式达到最快响应客户端的目的 负载最低优先类算法是站在服务器的角度来进行分配的 性能最优优先类算法本质上也是感知了服务器的状态 只是通过响应时间这个外部标准来衡量服务器状态而已 复杂度很高，主要体现在负载均衡系统需要收集和分析每个服务器每个任务的响应时间，在大量任务处理的场景下，这种收集和统计本身也会消耗较多的性能 工业上使用采样，并调优采样率 Hash 类（关键信息Hash，相同的放在一起）负载均衡系统根据任务中的某些关键信息进行 Hash 运算 将相同 Hash 值的请求分配到同一台服务器上，这样做的目的主要是为了满足特定的业务需求。源地址 Hash：将来源于同一个源IP 地址的任务分配给同一个服务器进行处理，适合于存在事务、会话的业务；ID Hash：将某个 ID 标识的业务分配到同一个服务器中进行处理，这里的 ID 一般是临时性数据的 ID(如 session id) **进程间通信** 规模大、底层网络不可靠、没有基于共享内存的原语、比用共享内存难 **通信模型** RPC：隐藏了大多数复杂的信息传递，理想的客户端/服务器应用程序；MOM：高级消息排队模型，类似于电子邮件；通信并不遵循相当严格的客户机/服务器交互模式。分类： 基于寻址类型的分类（直接（对称、非对称），间接（Intermediate Storage））；基于阻塞类型的分类（同步、异步）；基于缓存类型的分类（瞬态、持久）；基于内容类型的分类（事件、指令、数据、流）；基于确认类型的分类（未确认通信、确认通信、三次握手通信）；基于接收节点数的分类（点对点、多播、任播、地域性群播、广播）；基于通讯方向的分类（单向、双向半双工、双向全双工）；基于发起方的分类（拉取、推送）；基于消息存储的分类（持续通信：由通信中间件存储的消息，只要它需要传递它；收发双方无需同步，接收器将在下次运行时得到消息。瞬态通信：收发双方均运行时，才会存储消息；中间件传输中断或收件人无法传递消息，则丢弃该消息）。面向消息的中间件 **MOM**：异步消息传输的中间件，它是基于消息的通信、消息存储在消息队列中、消息服务器解耦了客户端和服务端、关于消息内容的各种假设。形式：消息排队、发布-订阅。重要性：1.面向消息的中间件服务的重要类别2.持久异步通信3.为消息提供中间形态的存储容量4.不需要发送方或接收方在消息传输期间保持活跃状态5.支持允许消息传输的时间开销。属性：异步交互、可靠服务交付、通过中间消息服务器处理消息、支持数据库集成。**消息队列模型**：应用程序通过在特定的队列中插入消息来进行通信、在最终被传递到目的地之前，通过一系列通信服务器转发的消息、接收双方无需同步、发送者可以保证其消息最终将被插入到收件人的队列中，但不能保证何时插入；作为发送和接收之间的命名消息目的地，允许进程独立执行和失败，可以掩盖进程失败和通信失败。分类：同步：不需要缓冲，发送方和接收方要同时运行；异步：高度并行，更快，可能缓冲溢出，依靠队列。消息优先级：最高优先级优先、加权公平调度。功能：支持多种消息传递模型、队列管理、连接管理、服务质量、数据转换；队列管理器：创建/删除队列，允许启动和停止队列，更改现有队列的属性，允许监视性能、故障和恢复，通常，队列管理器可以配置为将消息转发给其他队列管理器，消息代理，重新格式化数据，全局翻译，了解源代码和目标的结构/格式。解耦方式：按队列进行时间解耦、按队列划分的位置解耦。队列消息传递模式：一对一（点对点）、一对多、多对一、多对多。 **分布式服务框架** **原理**：1.Service层主要包括Java动态代理，消费者使用，主要用于将服务提供者的接口封装成远程服务调用；2.Java反射，服务提供者使用，根据消费者请求消息中的接口名、方法名、参数列表反射调用服务提供者的接口本地实现类；3.再向上就是业务的服务接口定义和实现类，具体服务逻辑内容由业务部门来实现，平台部分负责将业务接口发布成远程服务。**主要功能：1.**服务注册中心:负责服务的发布和通知，通常支持对等集群部署，服务注册中心宕机，只影响新服务的注册和发布，不影响已经发布的服务的访问。服务治理中心:通常包含服务治理接口和服务质量Portal，架构师、测试人员和系统运维人员通过服务治理Portal对服务的运行状态、历史数据、健康度和调用关系等进行可视化的分析和维护目标就是要持续化服务，防止服务架构腐化，保证服务高质量运行。**功能特性：1.**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；2.服务路由：路由策略、粘滞连接、路由定制；3.集群容错：Failover:失败自动切换Failback:失败自动恢复。Failfast:快速失败；4.服务调用：同步调用、异步调用、并行调用；5.多协议：私有协议、公有协议；6.序列化方式：二进制类序列化、文本类序列化；7.统一配置：本地静态配置、基于配置中心的动态配置。**幂等性概念** 用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。重要性:由于服务无状态的本质，对于业务的敏感性是很弱的。如果不支持幂等性的话，就会导致服务重复操作，对于业务数据进行违背业务逻辑的重复性操作。幂等场景: 网络波动、分布式消息消费、用户重复操作、未关闭的重试机制。幂等性的影响往往作用在数据上，而不同数据库操作对于幂等性的反应也不一样: 新增类请求不具备幂等性；查询具有天然幂等性；更新类请求，基于主键的计算式，不具备幂等性，基于主键的非计算式具备幕等性，基于条件查询的更新，不一定具备幂等性；删除类请求，基于主键的具备幂等性。数据库的幂等性解决方案：1.数据库加锁法:让关键资源的操作串行起来但是这会引入其他的问题，包括效率问题，死锁问题等；2.全局唯一ID法:根据业务的操作和内容生成一个全局ID，在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。该方案缺点实现起来困难，同时与服务的业务解绑有一定的冲突；3.去重表法:在本身具有唯一标识的业务场景下是非常好的方法，利用唯一的标识号，判断操作是否被重复执行；4.多版本控制法:为每一次操作添加一次版本号，以示区别，缺点在于版本号的管理，以及通常适用于更新操作，并且往往需要配合日志来完成数据最终一致性；5.状态机控制法:这种方法适合在有状态机流转的情况下。**微服务** 微服务是一种架构设计模式，在微服务架构中，业务逻辑被拆分成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。 每个微服务都在整体架构中执行着单独的任务，或负责单独的功能。每个微服务可能会被一个或多个其他微服务调用，以执行较大应用需要完成的具体任务。系统为任务执行提供了统一的解决处理方式，并限制应用内不同地方生成或维护相同功能的多个版本。特点：负责单个功能、单独部署、包含一个或多个进程、拥有自己的数据存储、一支小团队就能维护几个微服务、可替换的。与SOA架构的区别：组件大小：大块业务逻辑；单独任务或小块业务逻辑。耦合：通常松耦合；总是松耦合。公司架构：任何类型；小型、专注于功能交叉团队。管理：着重中央管理；着重分散管理。目标：确保应用能够交互操作；执行新功能、快速拓展开发团队。**SOA服务架构的服务集成**：通过企业服务总线，业务逻辑在中间层。**微服务架构的服务集成**：降低中心消息总线的依赖，将业务逻辑分布在每个具体的服务终端；基于轻量的消息总线或网关；点对点方式调用（不推荐）；API网关方式（最广泛），所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能；消息代理方式，异步场景，通过队列和订阅主题实现。**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。优点：相对直接，能适应不同的应用进行变化；缺点：客户端与服务注册绑定，要针对服务端用到的所有语言和框架来实现逻辑。 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。优点：客户端无需关注发现的细节；缺点：需要配置和管理。**微服务架构的服务注册**：服务注册表是服务发现的核心部分，是包含服务实例的网络地址的数据库。服务注册表需要高可用而且随时更新。客户端能够缓存从服务注册表中获取的网络地址，然而，这些信息最终会过时，客户端也就无法发现服务实例。因此，服务注册表会包含若干服务端，使用复制协议保持一致性。服务实例必须在注册表中注册和注销。注册和注销有两种不同的方法:自注册模式：服务实例负责在服务注册表中注册和注销，一个服务实例也要发送心跳来保证注册信息不会过时。优点：它相对简单，无需其它系统组件。缺点：把服务实例和服务注册表耦合，必须在每个编程语言和框架内实现注册代码。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。服务注册器会通过查询部署环境或订阅事件的方式来跟踪运行实例的更改。一旦侦测到有新的可用服务实例，会向注册表注册此服务。服务管理器也负责注销终止的服务实例。优点：服务与服务注册表解耦合，无需为每个编程语言和框架实现服务注册逻辑;相反，服务实例通过一个专有服务以中心化的方式进行管理。缺点：除非该服务内置于部署环境，否则需要配置和管理一个高可用的系统组件。**微服务架构的数据去中心化的核心要点**:1)每个微服务有自己私有的数据库持久化业务数据；2)每个微服务只能访问自己的数据库，而不能访问其它服务的数据库；3)需要在一个事务中更新多个数据库，不能直接访问其它微服务的数据库，而是通过对于微服务进行操作。意义：数据的去中心化，进一步降低了微服务之间的耦合度，不同服务可以采用不同的数据库技术。在复杂的业务场景下，如果包含多个微服务，通常在客户端或者中间层(网关)处理。**Docker** Docker引擎：是一个C/S结构的应用；Server是一个常驻进程；REST API实现了Client和Server间的交互协议；CLI实现容器和镜像的管理，为用户提供统一的操作界面。VS虚拟机： 虚拟机的Guest OS即为虚拟机安装的操作系统，它是一个完整操作系统内核;虚拟机的Hypervisor层可以简单理解为一个硬件虚拟化平台，它在Host OS是以内核态的驱动存在的，他们被Docker Engine层所替代；docker有着比虚拟机更少的抽象层，由于docker不需要Hypervisor实现硬件资源虚拟化，运行在docker容器上的程序直接使用的都是实际物理机的硬件资源，因此在CPU、内存利用率上docker将会在效率上有优势。优点：更快速的交付和部署（直接集成到开发流程中）；高效的部署和扩容（可以在任意平台运行）；更高的资源利用率；更简单的管理（以增量的方式分发更新，自动化高效管理）。缺点：资源隔离方面不如虚拟机，只能限制资源消耗的最大值，而不能隔绝其他程序占用自己的资源；安全性问题，Docker目前并不能分辨具体执行指令的用户；Docker目前还在版本的快速更新中，存在版本兼容问题。**Docker架构** C/S架构：Client 通过接口与Server进程通信实现容器的构建，运行和发布，Client和Server可以运行在同一台集群，也可以通过跨主机实现远程通信；Docker 镜像：一堆只读层的统一视角，重叠在一起，除了最下面一层，其它层都会有一个指针指向下一层，这些层是Docker内部的实现细节，并且能够在docker宿主机的文件系统上访问到；统一文件系统技术能够将不同的层整合成一个文件系统为这些层提供了一个统一的视角，这样就隐藏了多层的存在，在用户的角度看来，只存在一个文件系统，可以用来创建很多容器，提供了简单的机制来创建镜像或者更新现有的镜像；仓库：集中存放镜像文件的场所，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签,仓库分为公开仓库和私有仓库两种形式，最大的公开仓库是 Docker Hub，国内的公开仓库包括时速云、网易云等，用户也可以在本地网络内创建一个私有仓库，Docker 仓库的概念跟 Git 类似，注册服务器可以理解为GitHub 这样的托管服务；容器：被用来运行应用，是从镜像创建的运行实例，可以被启动、开始、停止、删除，每个容器都是相互隔离的、保证安全的平台，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，一个运行态容器被定义为一个可读写的统一文件系统加上隔离的进程空间和包含其中的进程，文件系统隔离技术使得Docker成为了一个非常有潜力的虚拟化技术，一个容器中的进程可能会对文件进行修改、删除、创建，这些改变都将作用于可读写层；**Docker调度工具** 容器没有操作系统或者 hypervisor，容器没有独立运作的能力，所以需要有自己的调度管理工具。它的主要任务就是负责在最合适的主机上启动容器，并且将它们关联起来。它必须能够通过自动的故障转移(fail-overs)来处理错误，并且当一个实例不足以处理/计算数据时，它能够扩展容器来解决问题。**Swarm：**是一个 Docker 的集群化工具，通过使用一个或者多个 Docker 主机来组成一个Swarm 集群；将容器打包到主机上，所以它能为更大的容器预留其他的主机资源，较之于随机地将容器调度到集群中的一个主机上，这种集群的组成方式能取得更加经济的伸缩性能；负责调度容器，Swarm使用了和Docker标准API一致的API，这意味着在Swarm上运行一个容器和在单一主机上运行容器使用相同的命令，开发者在使用Swarm的同时并不需要改变它的工作流程；由多个代理组成，把这些代理称之为节点，这些节点就是主机，这些主机在启动Docker daemon的时候就会打开相应的端口，以此支持Docker远程API；当启动Docker daemon时，每一个节点都能够被贴上一些标签，这些标签以键值对的形式存在，通过标签就能够给予每个节点对应的细节信息，当运行一个新的容器时，这些标签就能够被用来过滤集群。调度策略：random策略:随机选择节点，一般用于开发测试阶段；默认策略，swarm优先选择占用资源最少的节点，能保证集群中所有节点资源的均匀使用；binpack策略:与spread相反，它的目的是尽可能地填满一个节点，以保证更多空余的节点。**Mesos**：目的是建立一个高效可扩展的系统，支持很多各种各样的框架，在底部添加一个轻量的资源共享层，这个层使得各个框架能够适用一个统一的接口来访问集群资源，并不负责调度而是负责委派授权，取决于用户想要在集群上运行的作业类型，提供调度功能：约束，健康检查，服务发现和负载均衡。**K8S**：定义：一个开源的可以用来自动部署、伸缩和管理容器化应用的系统。基本组成部分: 1.Kubernetes成组地部署和调度容器，这个组叫Pod，常见的Pod包含一个到五个容器，它们协作来提供一个 Service；2.默认使用扁平的网络模式，通过让在一个相同Pod 中的容器共享一个 |P 并使用 localhost 上的端口，允许所有的 Pod 彼此通讯；3使用 Label 来搜索和更新多个对象；4搭设一个 DNS 服务器来供集群监控新的服务，然后可以通过名字来访问它们；5使用 Replication Controller来实例化的Pod，对运行的容器进行管理和监控。功能：1是Docker容器的编排系统，它使用label和pod的概念来将容器换分为逻辑单元，Pods是同地协作容器的集合，这些容器被共同部署和调度，形成了一个服务，这是Kubernetes和其他两个框架的主要区别，相比于基于相似度的容器调度方式(就像Swarm和Mesos)，这个方法简化了对集群的管理；2调度器的任务就是寻找那些PodSpec.NodeName为空的pods，然后通过对它们赋值来调度对应集群中的容器。相比于Swarm和Mesos，Kubernetes允许开发者通过定义PodSpec.NodeName来绕过调度器。调度器使用谓词和优先级来决定一个pod应该运行在哪一个节点上。谓词：谓词是强制性的规则，它能够用来调度集群上一个新的pod，如果没有任何机器满足该谓词，则该pod会处于挂起状态，直到有机器能够满足条件。优先级：如果调度器发现有多个机器满足谓词的条件，那么优先级就可以用来判别哪一个才是最适合运行pod的机器。优先级是一个键值对，key表示优先级的名字，value是该优先级的权重。**数据** 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，是对客观事物的性质、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，可以是具有一定意义的文字、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。解释：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。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。软件价值：数据（海量、离散）+语义（异构）+逻辑（复杂、多用户、多片段）=业务；代码+业务=软件应用系统（面向海量数据的高并发的复杂分布式异构系统）。**数据带来的架构变化** 单机MySQL：问题:数据量过大，且单机存储不下；单机内存存不下数据索引；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：用缓存来缓解数据库压力，Memcached(缓存)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：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：数据库中的数据量不可控，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，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，由于服务器资源有限，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）：（Redis，MongoDB）云时代对DB的新需求：高性能（对数据库高并发读写的需求）；海量存储（对海量数据的高效率存储和访问的需求）；高伸缩性与高可用性（对数据库的高可扩展性和高可用性的需求）。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。CAP 理论：核心特性：一致性、可用性、分区容错性**。**在分布式系统中，不可能同时满足，只能在三者中选择两个；不同的分布式系统会根据实际需求在 CAP 三者之间进行权衡。数据一致性：强一致性：数据的更新对所有节点立刻可见，所有读操作都能读取到最新的数据；最终一致性：数据的更新在一定时间后对所有节点可见。**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。复制延迟：主从复制延迟，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。方案：写操作后的读操作指定发给数据库主服务器；读从机失败后再读一次主机；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、中间件封装。**主备复制** （备份）基本实现逻辑：主机存储数据,通过复制通道将数据复制到备机；正常情况下,客户端无论读写操作,都发送给主机,备机不对外提供任何读写服务；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,不能读写数据,但数据并没有全部丢失,因为备机上有数据；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；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,切换访问链路；主机不能恢复的情况下,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,也许有的数据就永远丢失了,业务上需要考虑如何应对此类风险；如果主备间数据复制延迟,由于备机并不对外提供读写操作,因此对业务没有影响,但如果延迟较多,恰好此时主机又宕机了,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。优点：对于客户端来说,不需要感知备机的存在；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：主机存储数据，通过复制通道将数据复制到从机；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)；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；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；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；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；如果主从间数据复制延迟，则会出现主从读取数据不一致问题；如果主从间延迟较多，恰好此时主机又宕机了，则可能丢失较多数据一般的做法是做复制延迟的监控措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。优缺点（相对主备复制而言）:1主从复制在主机故障时,读操作相关的业务不受影响2主从复制架构的从机提供读操作,发挥了硬件的性能3主从复制要比主备复制复杂,主要体现在客户端需要感知主从关系,并将不同的操作发给不同的机器进行处理 同样的缺点,需要人工的干预处理故障,效率低。主备和主从复制的共性问题1主机故障后,无法进行写操作2如果主机无法恢复,需要人工指定新的主机。 关键的设计点 1主备间状态判断-状态传递渠道, 状态检测内容2倒换决策-倒换时机, 倒换策略, 自动程度3数据冲突。**主备/主从倒换常见架构** **互连式**:主备机直接建立状态传递的渠道 1可以是网络连接,也可以是非网络连接2可以是主机发送状态给备机,也可以是备机拉取主机的状态3可以和数据复制通道共用,也可以独立一条通道4状态传递通道可以是一条,也可以是多条,还可以是不同类型的通道混合。方案：1主备机共享一个对于客户端来说唯一的地址2客户端记录主备机各自的IP,备机具有拒绝服务的能力。缺点:1状态传递通道本身故障了,则备机会主动升级为主机2虽然可以通过多通道来降低通道故障的机率,但是通道越多,后续的状态决策越复杂,特别是容易收到多种矛盾的信息。**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。结构复杂,但是从状态传递和决策上更加简单:1连接管理更简单:主备机无须再建立和管理多种类型的状态传递连接通道,只要连接到中介即可,实际上降低了主备机的连接复杂度2状态决策更简单:无须考虑多种类型的连接通道获取状态信息如何决策的问题。 状态决策的步骤：1无论主机还是备机,初始状态都是备机,并且只要与中介断开连接,就将自己降级为备机,因此可能出现双备机的情况2主机与中介断连后,中介能够立刻告知备机,备机将被升级为主机3如果是网络中断导致主机与中介断连,主机自己会降级为备机,网络恢复后,旧的主机以新的备机身份向中介上报自己的状态4如果是掉电重启或者进程重启,旧的主机初始状态为备机,与中介恢复连接后,发现已经有主机了,保持自己备机状态不变5主备机与中介连接都正常的情况下,按照际的状态决定是否进行倒换。**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,向主机发起模拟的读写操作,根据读写操作的响应情况来判断主机的状态。模拟式相比互连式的优缺点:1实现更加简单,省去了状态传递通道的建立和管理工作2模拟式读写操作获取状态信息只有响应信息,没有互连式那么多样,基于有限的状态来做状态决策,可能出现偏差。**主主复制** 基本设计思路：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。1两台主机都存储数据,通过复制通道将数据复制到另一台主机2正常情况下,客户端可以将读写操作发送给任意一台主机3一台主机故障情况下,客户端只需要将读写操作发送给主机B即可,反之亦然4如果故障主机能够恢复,则客户端继续访问两台主机,两台主机继续相互复制对方数据5如果故障主机不能恢复,则需要人工操作,增加一台新的机器为主机6原有故障主机不能恢复的情况下,成功写入原有故障主机但没有复制到正常主机的数据会丢失7如果两台主机间复制延迟,则可能出现客户端刚写入的数据,在另一台主机上读取不到。优缺点:1两台主机,无倒换概念2客户端无须区分主备机身份3必须保证数据能够双向复制,然而很多数据无法双向复制。**数据集群** **数据集中集群：**称为一主多备/从，数据都只能往主机写,而读操作可以参考主备,主从的架构进行灵活变化。复杂度高:1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2多通道,情况不一,容易导致数据不一致,需要在备机之间进行数据一致性检查和修正3多备对单主状态的检测结果不一致,容易出现不同的判断和决策4单主多备,当主机宕机,如何重新选主,需要算法。**数据分散集群：**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。复杂度：在于如何将数据分配到不同的服务器上：1均衡性:保证数据分区基本均衡2容错性:部分服务器故障后,这些服务器上的数据分区需要分配给其他服务器3可伸缩性:当集群容量不够,扩充新的服务器后,算法能够自动将数据分区迁移到新服务器,并保证扩容后所有服务器的均衡性。必须要有一个角色来负责执行数据分配算法：可以是独立服务器,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。**集中VS分散：**写数据角色：1数据集中集群架构中,客户端只能将数据写到主机2数据分散集群架构中,客户端可以向任意服务器中读写数据。应用场景：1数据集中集群适合数据量不大,集群机器数量不多的场景2数据分散集群,由于其良好的可伸缩性,适合业务数据量巨大、集群机器数量庞大的业务场景。**数据库和数据库实例** 数据库是文件的集合，是依照某种数据模型组织起来并存放于二级存储器中的数据集合;数据库实例是程序，是位于用户和操作系统之间的一层数据管理软件，用户对数据库中的数据做任何的操作，包括数据定义、数据查询、数据维护、数据库运行控制等等都是在数据库实例下进行的，应用程序只有通过数据库实例才能和数据库进行交互;数据库是由文件组成(一般来说都是二进制文件)的，一般不可能直接对这些二进制文件进行操作，以实现数据库的SELECT、UPDATE、INSERT和DELETE操作，需要数据库实例来完成对数据库的操作。**分库分表** 本质是数据拆分，是对数据进行分而治之的通用概念；为了分散数据库的压力，采用分库分表将一个表结构分为多个表，或者将个表的数据分片后放入多个表，这些表可以放在同一个库里，也可以放到不同的库里，甚至可以放在不同的数据库实例上；数据拆分主要分为:垂直拆分:根据业务的维度，将原本的一个库(表)拆分为多个库(表)，每个库(表)与原有的结构不同。水平拆分:根据分片算法，将一个库(表)拆分为多个库(表)，每个库(表)依旧日保留原有的结构。基本概念阶段：单库单表；单库多表；多库多表。操作时机:如果在数据库中表的数量达到了一定量级，则需要进行分表，分解单表的大数据量对索引查询带来的压力，并方便对索引和表结构的变更;如果数据库的吞吐量达到了瓶颈，就需要增加数据库实例，利用多个数据库实例来分解大量的数据库请求带来的系统压力;如果希望在扩容时对应用层的配置改变最少，就需要在每个数据库实例中预留足够的数据库数量。**解决方案：客户端分片：**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，每个应用层都嵌入一个操作切片的逻辑实现，一般通过依赖Jar包来实现。实现方式：在应用层直接实现：直接在应用层读取分片规则，然后解析分片规则，据此实现切分的路由逻辑，从应用层直接决定每次操作应该使用哪个数据库实例、库、表等；需要侵入业务，但实现简单，适合快速上线，切分逻辑由开发者自行定义，容易调试维护，但要求开发者既要实现业务逻辑，还需要实现框架需求；该实现方式会让数据库保持的连接比较多，对整体应用服务器池的维护将造成压力；通过定制JDBC协议实现：可让开发者集中精力实现业务逻辑，无须关心分库分表的实现；通过定制JDBC协议来实现，也就是针对业务逻辑层提供与JDBC一致的接口，分库分表在JDBC的内部实现；开发者需要理解JDBC协议。通过定制ORM框架实现：分片规则实现到ORM框架中或者通过ORM框架支持的扩展机制来完成分库分表的逻辑。**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；应用层的开发人员不用关心分片规则，只需关心业务逻辑的实现，待业务逻辑实现之后，在代理层配置路由规则即可；代理层的引入增加了一层网络传输，对性能会造成影响；需要维护代理层，增加了人员和硬件的成本。**支持事务的分布式数据库：**对外提供可伸缩的体系架构，并提供一定的分布式事务支持，将可伸缩的特点和分布式事务的实现包装到分布式数据库内部，对使用者透明，使用者不需要直接控制这些特性，目前不太适用于交易系统，较多用于大数据日志系统、统计系统、查询系统、社交网络等。**切分方法**：**垂直切分**：是指按照业务将表进行分类或分拆，将其分布到不同数据库上（按业务进行分库、按业务进行分表）；不同业务模块的数据可以分散到不同数据库服务器；可以冷热分离，根据数据的活跃度将数据进行拆分；也可以人为将一个表中的内容划分为多个表；读多写少的冷数据库可以部署到缓存数据库上。优点:拆分后业务清晰，拆分规则明确；系统之间进行整合或扩展很容易；按照成本、应用的等级或类型等将表放到不同的机器上，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，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：**水平切分不是将表进行分类，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，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。路由过程：分库分表后，数据将分布到不同的分片表中，通过分库分表规则查找到对应的表和库的过程叫做路由。分片维度：按哈希切片：对数据的某个字段求哈希，再除以分片总数后取模，取模后相同的数据为个分片，这样将数据分成多个分片。好处:数据切片比较均匀，对数据压力分散的效果较好。缺点:数据分散后，对于查询需求需要进行聚合处理。按照时间切片：按照时间的范围将数据分布到不同的分片。**垂直、水平切分的共同点** 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。**分库分表引发的问题** **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：由于数据量大，通常会造成不一致问题，因此，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。查询问题：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査询、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。解决方案:1、在多个分片表查询后合并数据集(效率很低)；按查询需求定义多分片维度，形成多张分片表(空间换时间)；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)。分布式引发一致性问题，尽量把同一组数据放到同一台数据库服务器上。**分库分表的中间件简介** **Mycat**：是一个强大的数据库中间件，不仅仅可以用作读写分离、以及分表分库、容灾备份，而且可以用于多租户应用开发、云平台基础设施；后面连接的Mycat Server，就好像是MySQL的存储引擎，因此，Mycat 本身并不存储数据，数据是在后端的MySQL上存储的，因此数据可靠性以及事务等都是MySQL保证的。基本原理：Mycat拦截了用户发送过来的SQL语句，首先对SQL语句做一些特定的分析:如分片分析、路由分析、读写分离分析、缓存分析等，然后将此 SQL 发往后端的真实数据库，并将返回的结果做适当的处理，最终再返回给用户；当 Mycat收到一个SQL时，会先解析这个SQL，查找涉及到的表，然后看此表的定义，如果有分片规则，则获取到SQL里分片字段的值，并匹配分片函数，得到该SQL对应的分片列表，然后将SQL发往这些分片去执行，最后收集和处理所有分片返回的结果数据，并输出到客户端。应用场景：单纯的读写分离，此时配置最为简单，支持读写分离，主从切换；分表分库；多租户应用，每个应用一个库，但应用程序只连接Mycat，从而不改造程序本身，实现多租户化；报表系统，借助于Mycat 的分表能力，处理大规模报表的统计；替代Hbase，分析大数据；作为海量数据实时查询的一种简单有效方案，除了基于主键的查询，还可能存在范围查询或其他属性査询，此时 Mycat 可能是最简单有效的选择。**Sharding JDBC** 实现透明化数据库分库分表访问，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:可适用于任何基于Java的ORM框架；可基于任何第三方的数据库连接池；理论上可支持任意实现JDBC规范的数据库。定位为轻量Java框架，使用客户端直连数据库，以jar包形式提供服务，无proxy代理层，无需额外部署，无其他依赖，DBA也无需改变原有的运维方式。分片策略灵活，可支持等号、between、in等多维度分片，也可支持多分片键。SQL解析功能完善，支持聚合、分组、排序、limit、or等查询，并支持BindingTable以及笛卡尔积表查询。分片规则配置：分片逻辑非常灵活，支持分片策略自定义、复数分片键、多运算符分片等功能。除了支持等号运算符进行分片，还支持in/between运算符分片，提供了更加强大的分片功能。提供了spring命名空间用于简化配置，以及规则引擎用于简化策略编写。JDBC规范重写Sharding-JDBC对JDBC规范的重写思路是针对DataSource、Connection、Statement、PreparedStatement和ResultSet五个核心接口封装，将多个真实JDBC实现类集合纳入ShardingJDBC实现类管理。SQL解析作为分库分表类产品的核心，性能和兼容性是最重要的衡量指标。SQL改写分为两部分，一部分是将分表的逻辑表名称替换为真实表名称。另一部分是根据SQL解析结果替换一些在分片环境中不正确的功能。SQL路由：是根据分片规则配置，将SQL定位至真正的数据源。主要分为单表路由、Binding表路由和笛卡尔积路由。单表路由最为简单，但路由结果不一定落入唯一库(表)，因为支持根据between和in这样的操作符进行分片，所以最终结果仍然可能落入多个库(表)。Binding表可理解为分库分表规则完全一致的主从表。笛卡尔积查询最为复杂，因为无法根据Binding关系定位分片规则的一致性，所以非Binding表的关联查询需要拆解为笛卡尔积组合执行。查询性能较低，而且数据库连接数较高，需谨慎使用。SQL执行：路由至真实数据源后，Sharding-JDBC将采用多线程并发执行SQL，并完成对addBatch等批量方法的处理。结果归并：包括4类:普通遍历类、排序类、聚合类和分组类。每种类型都会先根据分页结果跳过不需要的数据。普通遍历类最为简单，只需按顺序遍历ResultSet的集合即可；排序类结果将结果先排序再输出，因为各分片结果均按照各自条件完成排序，所以采用归并排序算法整合最终结果。聚合类分为3种类型，比较型、累加型和平均值型。比较型包括max和min，只返回最大(小)结果；累加型包括sum和count；需要将结果累加后返回，平均值则是通过SQL改写的sum和count计算。分组类最为复杂，需要将所有的ResultSet结果放入内存，使用MapReduce算法分组，最后根据排序和聚合条件做相关处理，最消耗内存最损失性能的部分即是此，可以考虑使用limit合理的限制分组数据大。**数据缓存** 数据缓存存在于服务器层，位于**数据库/文件系统**和**客户端**之间。概念：1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。2缓存中的数据可能是提前计算好的结果、数据的副本等。作用：1主要解决高并发，热点数据访问的性能问题。2提供高性能的数据快速访问。原理：1将数据写入到读取速度更快的存储。2将数据缓存到离应用最近的位置。3将数据缓存到离用户最近的位置。存储介质：虽然从硬件介质上来看，无非就是内存和硬盘两种，但从技术上，可以分成内存、硬盘文件、数据库。内存:缓存于内存中是最快的选择，无需额外的I/0开销，但是内存的缺点是没有持久化到物理磁盘，一旦应用异常break down而重新启动，数据很难或者无法复原。硬盘:一般来说，很多缓存框架会结合使用内存和硬盘，在内存分配空满了或是在异常的情况下，可以被动或主动的将内存空间数据持久化到硬盘中，达到释放空间或备份数据的目的。数据库:非传统数据库，而是key-value存储结构的特殊数据库，响应速度和吞吐量都远高于关系型数据库等。**基本类型**：1本地缓存；2分布式缓存；3反向代理缓存；4 CDN缓存。**本地缓存**：本地缓存指的是在应用中的缓存组件，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。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。缓存介质：1硬盘缓存：将数据缓存到硬盘，减少网络传输的消耗.2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。实现方法：应用编码，中间件。**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。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：1缓存经过复杂运算得出的数据2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**：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：体积较小的静态文件资源。**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。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。1.分布式存储2.负载均衡3.网络请求的重定向4.内容管理。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。**术语**：命中率=返回正确结果数/请求缓存次数，是衡量缓存有效性的重要指标，越高，表示使用率越高，而且很难预测。最大元素（最大空间）：缓存中可以存放的最大元素的数量，一旦数量超过，就会触发缓存启动清空策略。**数据缓存的基本操作：数据缓存命中与验证**：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，而且网络带宽是很珍贵的，所以大部分缓存只有在户端发起请求，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是最新的；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；再验证未命中；对象被删除。**清洗策略**：缓存清空策略：在缓存的存储空间有限制，当缓存空间被用满时，既要保证稳定服务又要有效提升命中率。常见的一般策略有:FIFO: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(超出最大元素限制)会被优先被清除掉，以腾出新的空间接受新的数据。LFU: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。LRU: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。根据过期时间判断，清理过期时间最长的元素。根据过期时间判断，清理最近要过期的元素。随机清理。根据关键字(或元素内容)长短清理等。**更新策略**：Cache aside：失效:应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后放到缓存中。命中:应用程序从cache中取数据，取到后返回。更新:先把数据存到数据库中，成功后，再让缓存失效。Read/Write Through Pattern：Read Through 是在查询操作中更新缓存，当缓存失效的时候(过期或LRU换出)，Cache Aside是由调用方负责把数据加载入缓存，而Read Throuqh则用缓存服务自己来加载，从而对应用方是透明的；Write Through 在更新数据时发生，当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回，如果命中了缓存，则更新缓存，然后再由Cache自己更新数据库(这是一个同步操作)。Write Behind Caching Pattern：在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。优点：让数据的I/O操作飞快无比，因为异步(比如消息队列)，write back还可以合并对同一个数据的多次操作，所以性能的提高是相当可观的。缺点：在于数据的非强一致性，极可能造成数据的丢失。**本地缓存**：编程直接实现缓存：成员变量或局部变量实现，以局部变量map结构缓存部分业务数据，或者可以静态变量实现，减少频繁的重复数据库I/O操作，缺点仅限于类的自身作用域内，类间无法共享缓存。中间件Ehcache：Ehcache是现在最流行的纯Java开源缓存框架，配置简单、结构清晰、功能强大，是一个轻量级的缓存实现。核心定义：1 cache manager：缓存管理器，允许多实例2 cache：缓存管理器内可以放置若干cache，存放数据的实质，所有cache都实现了 Ehcache接口，这是一个真正使用的缓存实例；通过缓存管理器的模式，可以在单个应用中轻松隔离多个缓存实例，独立服务于不同业务场景需求，缓存数据物理隔离，同时需要时又可共享使用。**分布式缓存**：分片的基本原理与分库分表类似，不做赘述，参考分库分表。迁移：**平滑迁移**的关键步骤:双写（应用同时向旧节点和新节点写入数据）；迁移历史数据（将旧节点的历史数据批量迁移到新节点）；切读（优先从新节点读取数据，如果新节点无数据则回退到旧节点）；下线双写（完成所有数据迁移后，旧节点正式停止服务）。**一致性哈希**：一致性哈希将整个哈希值空间组织成一个虚拟的圆环(哈希环)，如假设某哈希函数H的值空间为0-2^32-1(即哈希值是一个32位无符号整形)，整个空间按顺时针方向组织，0和2^32-1在零点钟方向重合。将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置。将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。容错性和可扩展性:一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器(即沿着逆时针方向行走遇到的第一台服务器)之间数据，其它不会受到影响；如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器(即沿着逆时针方向行走遇到的第一台服务器)之间数据，其它数据也不会受到影响。一致性哈希算法在服务节点太少时，容易因为节点分布不均匀而造成数据倾斜问题，引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。**停机迁移**：1停机应用，先将应用停止服务2迁移历史数据，按照新的规则把历史数据迁移到新的缓存数据集群中3更改应用的数据源配置，指向新的缓存集群4重新启动应用。该方式简单，高效，能够有效避免数据的不一致但需要由业务方评估影响，一般在晚上访问量较小或者非核心服务的场景下比较适用。**缓存问题**：**数据一致性：**持久化数据的一个副本，会出现数据不一致问题，如脏读或读不到数据的情况，一般是因为网络不稳定或节点故障导致，正确做法：先写持久化介质再写缓存的方式；假如读缓存失败，先读数据库，再回写缓存缓存；根据日志中用户刷新数据的时间间隔，以及针对数据可能产生不一致的时间，进行同步操作。**缓存穿透**：对不存在的key进行大量的高并发查询，每次请求都要穿透到后端数据库系统进行查询。解决方法：1将空值缓存起来，再次接收到同样的查询请求时，若命中缓存并值为空，就会直接返回，不会透传到数据库，避免缓存穿透2对恶意的查询攻击，可以对查询条件设置规则，不符合条件产生规则的直接拒绝。**缓存并发**：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，因此多个请求会同时访问数据库来查询最新数据，并且回写缓存，这样会造成应用和数据库的负载增加，性能降低，由于并发较高，甚至会导致数据库崩溃。解决方案：分布式锁:使用分布式锁，保证对于每个key同时只有一个线程去查询后端服务；本地锁:与分布式锁类似，通过本地锁的方式来限制只有一个线程去数据库中查询数据，而其他线程只需等待，这种方法只能限制一个服务节点只有一个线程取数据库中查询，如果一个服务有多个节点，则会有多个数据库查询操作；软过期:指对缓存中的数据设置失效时间，就是不使用缓存服务提供的过期时间，而是业务层在数据中存储过期时间信息，由业务程序判断是否过期并更新，在发现了数据即将过期时，将缓存的时效延长，程序可以派遣一个线程去数据库中获取最新的数据，其他线程会先继续使用旧数据并等待，直至派遣线程获取最新数据后再更新缓存。也可以通过异步更新服务来更新设置软过期的缓存。**缓存雪崩**：指缓存服务器重启或者大量缓存集中在某个时间段内失效，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，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，保证只有一个线程能进行缓存更新；失效时间分片机制:对不同的数据使用不同的失效时间，甚至对相同的数据、不同的请求使用不同的失效时间；后台更新机制:由后台线程来更新缓存，并不是业务线程来更新缓存；缓存集群:可以做缓存的主从与缓存水平分片。**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。解决:复制多份缓存副本，把请求分散到多个缓存服务器上，减轻缓存热点导致的单台缓存服务器压力。