架构共性：
构件：一组基本的构成要素
连接件：这些要素之间的连接关系
物理分布：这些要素连接之后形成的**拓扑结构**
约束：作用于这些要素或连接关系上的限制条件
性能：质量

软件架构(SA)：提供了一个结构、行为和属性的高级抽象；从一个较高的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，便于各种人员的交流，反映多种关注，据此开发的系统能完成系统既定的功能和性能需求
作用：1. 交流的手段:在软件设计者、最终用户之间方便的交流；2. 可传递的、可复用的模型(可重复利用的、可转移的系统抽象):用到其它的项目、提高大规模重复利用率；3.关键决策的体现:折衷，关于性能与安全性、可维护性与可靠性，当前开发费用和未来开发代价
意义：SA是软件开发过程初期的产品，在开发的早期阶段就考虑系统的正确设计与方案选择，为以后开发、测试、维护各个阶段提供了保证

中间件是一组程序，应用于分布式系统各应用之中，为系统屏蔽底层通讯和提供公共服务，并保障系统的高可靠性、高可用性、高灵活性
分布式应用借助中间件在不同技术之间共享资源。中间件位于客户机/ 服务器的操作系统之上，管理计算机资源和网络通讯。通过中间件，应用程序可以工作于多平台或OS 环境
作用：1. 屏蔽异构性：异构性表现在计算机的软硬件差异，包括硬件（CPU和指令集等，操作系统、数据库（不同存储和访问格式等2. 实现互操作：因为异构性，产生的结果是软件依赖于计算环境，使得各种不同软件之间在不同平台之间不能移植，或者移植困难。而且，因为网络协议和通信机制不同，这些系统不能有效相互集成3. 共性凝练和复用：软件应用领域越来越多，相同领域的应用系统之间许多基础功能和结构是有相似性的。通过中间件提供简单、一致、集成的开发和运行环境，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

4+1视图模型
用例视图：描述系统的典型场景与功能，主要包括用例图等。
逻辑视图：系统的抽象概念与功能(类、接口等)类图,协作图,时序图等;
开发视图：系统中的子系统,模块,文件,资源及其之间的关系，组件图、包图等;
进程视图：系统的进程及其之间的通信协作关系,活动图,时序图等
物理视图：系统如何被安装,部署,配置在分布式的物理环境下，部署图

软件架构风格：描述用以组织一类软件系统的惯用模式，反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。定义一些构件和连接件类型，施加一组约束描述组合方式
软件架构风格分类
数据流风格：批处理；管道/过滤器；过程控制;
调用/返回风格：主程序/子程序；面向对象；分层结构
独立构件风格：事件系统;
虚拟机风格：解释器；基于规则的系统;
以数据为中心的风格：仓库；黑板;
其他架构风格：MVC；P2P；Grid；SOA。

面向对象构件
面向对象可以做到信息隐藏，内部的设计决策被封装
特性
- 封装：限制对某些信息的访问
- 交互：通过过程调用或类似的协议
- 多态：在运行时选择具体的操作
- 继承：对共享的功能保持唯一的接口
- 动态绑定：运行时决定实际调用的操作
- 复用和维护

构件：类和对象
连接件：对象之间通过函数与过程调用实现交互
优点：复用和维护；反映现实世界；容易分解一个系统
缺点：管理大量的对象；必须知道对象的身份标识；继承引起复杂度，关键系统中慎用

主程序-子过程风格
非结构化程序：所有的程序代码均包含在一个主程序文件中
缺陷：逻辑不清；无法复用；难以与其他代码合并；难于修改;难以测试特定部分的代码
结构化程序：逐层分解
- 基于"定义-使用"关系
- 用过程调用作为交互机制
- 主程序的正确性依赖于它所调用的子程序的正确性

构件：主程序、子程序
连接器：调用-返回机制
拓扑结构：层次化结构

本质：将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能。
优点：已被证明是成功的设计方法，可以被用于较大程序
缺点：程序超过10万行，表现不好；程序太大，开发太慢，测试越来越困难
把大系统**分解为模块的四大原则**:
模块独立性：高聚合、低耦合
模块规模适中性：过大分解不充分难理解;过小开销大接口复杂
模块复用性：高扇入+低扇出
作用域与控制域适当性：作用域要包含在控制域之中
- C的控制域包括C自己和C下属(调用)的模块
- 作用域：一个模块里有判断语句，这个语句会影响多个模块，多个模块依赖于这个判定条件，那么影响的范围就是作用域
- C控制域是CEF，作用域CEFD，作用域>控制域，一旦C做了修改，你就要去找D的位置，因为你的修改可能影响D，一旦修改了你还要去找D在哪（因为它不受你控制），会带来负面的影响，这是一种耦合性，会对维护带来很大的麻烦

数据流风格
处理操作：数据到达即被激活，无数据时不工作
特征：数据的可用性决定着处理<计算单元>是否执行；系统结构：数据在各处理之间的有序移动；在纯数据流系统中，处理操作之间除了数据交换，没有任何其他的交互
构件：数据处理
- 构件接口：输入端口和输出端口，从输入端口读取数据，向输出端口写入数据
- 计算模型：从输入端口读数，经过计算/处理，然后写到输出端口

连接件：数据流
- 单向，通常是异步、有缓冲
- 接口角色：reader和writer
- 计算模型：把数据从一个处理的输出端口传送到另一个处理的输入端口

拓扑结构：任意拓扑结构的图

管道-过滤器风格
适用场景：数据源源不断的产生，系统需要对这些数据进行若干处理。
解决方案：把系统分解为几个顺序的处理步骤，这些步骤之间通过数据流连接，一个步骤的输出是另一个步骤的输入；每个处理步骤由一个过滤器构件(Filter)实现；处理步骤之间的数据传输由管道(Pipe)负责。
构件：过滤器，处理数据流
连接件：管道，连接一个源和一个目的过滤器。连接器定义了数据流的图，形成拓扑结构

管道数据
作用：在过滤器之间传送单向流；可能具有缓冲区；数据缓冲区可以是文件、数组、字典、树等集合类型；管道形成数据传输图；管道的先后顺序不影响输出的结果；不同的管道中流动的数据流，可能具有不同的数据格式。
从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

过滤器数据
目标：将源数据递增的变换成目标数据
从"数据流" -> "数据流"的变换：通过计算和增加信息来丰富数据；通过浓缩和删减来精炼数据；通过改变数据表现方式来转化数据；将数据分解为多个流；将多个数据流合并为一个
读取与处理数据流方式：递增的读取和消费数据。数据到来时便处理，不是收集完然后处理，即在输入被完全消费之前输出便产生了
其他特征：无上下文信息；不保留状态；对上下游的其他过滤器无任何了解
过滤器状态：1. 停止状态：处于待启动状态，外部启动过滤器后，过滤器处于处理状态。2. 处理状态：正处理输入数据队列中的数据。3. 等待状态：输入数据队列为空，此时过滤器等待，当有新的数据输入时，过滤器处于处理状态。

优点：允许对一些如吞吐量、死锁等属性的分析；支持并行执行；使得系统中的构件具有良好的隐蔽性和高内聚、低耦合的特点；
- 支持软件复用：系统的行为是多个过滤器的行为的简单合成；在两个过滤器之间提供适合数据，任何两个过滤器都可被连接起来。
- 系统维护和增强系统性能简单：新的过滤器可以添加到现有系统中，旧的可以被改进的换掉；

缺点：
- 通常导致进程成为批处理的结构：每个过滤器是一个完整的从输入到输出的转换；
- 不适合处理交互的应用：当需要增量地显示改变时，这个问题尤为严重；
- 因为在数据传输上没有通用的标准，每个过滤器都增加了解析的和合成数据的工作，这样就导致了系统性能下降，并增加了编写过滤器的复杂性。

事件系统风格
显式调用：各个构件之间的互动是由显性调用函数或过程完成的。调用的过程与次序是固定的、预先设定的。
隐式调用：不直接去invoke一个过程
- Event Source：一个组件可以广播一些事件(发布事件到EvtManager)
- Event Handlers：系统中的其它构件可以注册自己感兴趣的事件，并将自己的某个过程与相应的事件进行关联
- Event Manager：当一个事件被发布，系统自动调用在该事件中注册的所有过程(负责调用所有注册到该事件的EventHandler)

风格主要特点：事件的触发者并不知道哪些构件会被这些事件影响，相互保持独立
- 不能假定构件的处理顺序，甚至不知道哪些过程会被调用
- 各个构件之间彼此之间无连接关系，各自独立存在，通过对事件的发布和注册实现关联

构件：对象或过程，并提供如下两种接口
- 过程或函数，充当事件源或事件处理器的角色；
- 事件

连接器：事件-过程绑定
- 事件处理器(事件的接收和处理方)的过程向特定事件注册
- 事件源构件发布事件
- 当某些事件被发布时，向其注册的过程被隐式调用

调用的次序是不确定的
> 在某些情况下，连接件可以是事件-事件的绑定——一个事件也可能触发其他事件，形成事件链

优点：支持实现交互式系统；异步执行，不必同步等待执行结果；为软件复用提供了强大的支持。当需要将一个构件加入现存系统中时，只需将它注册到系统的事件中；为系统动态演化带来了方便。构件替换时，不会影响到其它构件的接口；对事件的并发处理将提高系统性能；健壮性：一个构件出错将不会影响其他构件。
缺点：1. 分布式的控制方式使得系统的同步、验证和调试变得异常困难：关于正确性的推理则难以保证。传统的基于先验和后验条件的验证变得不可能。2. 数据交换的问题：在具有独立调度模块的事件系统中，数据则需要经过调度模块的传递，全局性能和资源管理成为了系统的瓶颈。

事件调度策略
无独立（非集中式）调度模块的事件管理器：称为"被观察者/观察者"（观察者模式）
- 每一个模块都允许其他模块向自己所能发送的某些事件表明兴趣
- 当某一模块发出某一事件时，它自动将这些事件发布给那些曾经向自己注册过此事件的模块

带有独立调度模块的事件管理器：事件调度模块
- 负责接收到来的事件并分发它们到其它模块。调度器要决定怎样分发事件：两种策略
  - 全广播式：调度模块将事件广播到所有的模块，但只有感兴趣的模块才去取事件并触发自身的行为；无目的广播，靠接受者自行决定是否加以处理或者简单抛弃。
  - 选择广播式：调度模块将事件送到那些已经注册了的模块中。
    - 点对点模式：基于消息队列。
    - 发布-订阅模式：一个事件可以被多个订阅者消费；事件在发送给订阅者之后，并不会马上从topic中删除，topic会在事件过期之后自动将其删除。

解释器风格
解释器是一个用来执行其他程序的程序。解释器针对不同的硬件平台实现了一个虚拟机，将高抽象层次的程序翻译为低抽象层次所能理解的指令，以消除在程序语言与硬件之间存在的语义差异。通常用来在程序语言定义的计算和有效硬件操作确定的计算之间建立对应和联系
解释器和编译器：编译器不会执行输入的源程序代码，而是将其翻译为另一种语言，通常是可执行的机器码或目标码，并输出到文件中以便随后链接为可执行文件并加以执行；在解释器中，程序源代码被解释器直接加以执行。
- 解释器的执行速度要慢于编译器产生的目标代码的执行速度，但是可能低于编译器"编译+链接+执行"的总时间
- 解释器通常省略了链接与编译的步骤，从而降低编程时间
- 解析器执行速度之所以慢，是因为每次解释执行的时候，都需要分析程序的结构，而编译代码则直接执行而无需重复编译
- 解释器对内存的分配是在解释时才进行的；而编译器则是在编译时进行，因此运行时直接将程序代码装入内存并执行即可

解释器风格分类
传统解释器：纯粹的解释执行
基于字节码的：编译 -> 解释执行。源代码首先被"编译"为高度压缩和优化的字节码，但并不是真正的机器目标代码，因而与硬件平台无关；编译后得到的字节码然后被解释器加以解释；
Just-in-Time (JIT)编译器：编译||解释执行。
只有当某个函数要被执行时，才被编译，因此称为JIT，而且，JIT并不是编译全部的代码，而是只编译那些被频繁执行的代码段，如被执行多次的方法、包含多次循环的方法
第一步是编译得到字节码；字节码被配置到目标系统中；当字节码被执行时，运行环境下的编译器将其翻译为本地机器码
使解释器、字节码解释器和编译器之间的边界模糊

MVC风格
关注点分离：模型、视图和控制器
从开发者的角度看，实现model与view的解耦
构件：
- Model：负责数据存取、负责业务逻辑实现、可能负责数据验证
- View：负责获取用户输入、向controller发送处理请求、接收来自Controller的反馈并将model的处理结果显示给用户。一个model可能有多个View
- Controller：负责接收来自客户的请求、调用model执行业务逻辑、调用View显示执行结果

连接件：隐式调用、显式调用、或者其他机制Http
两个分离原则：展示与模型分离；控制器与视图分离
优点：代码易开发易维护；同一信息可以有不同的显示方式；业务逻辑更易测试

分层原则：分离关注(尽量减少功能重叠)；抽象(删技节留主干)；隐藏(只暴露需要访问的接口)

分层结构
构件：各层次内部包含的构件
连接件：层间的交互协议
拓扑结构：分层
拓扑约束：对相邻层间交互的约束(集中式/分布式部署)

分层模式：
- 严格分层：上层仅能与直接下层交互
- 松散分层：上层可以与所有其下层交互，即 可以跨层交互
- 横切关注：在严格分层的基础上，每一层都可以与另外的一个组件交互

某一层中的构件一般只与同一级别中的对等实体或较低级别中的构件交互，这种单向交互有助于减少不同级别中的构件之间的依赖性。

三层C/S结构
在客户端与数据库服务器之间增加了一个中间层
中间层可能为：事务处理监控服务器、消息服务器、应用服务器等
中间层负责消息排队、业务逻辑执行、数据中转等功能

表示层
- 用户接口部分，担负着用户与应用之间的对话功能；
- 检查用户的输入，显示应用的输出；
- 通常使用GUI；
- 在变更时，只需要改写显示控制和数据检查程序，而不影响其他层；
- 不包含或包含一部分业务逻辑。

功能层
- 应用系统的主体，包括大部分业务处理逻辑 (通常以业务构件的形式存在，如JavaBean/EJB/COM等)；
- 从表示层获取用户的输入数据并加以处理；
- 处理过程中需要从数据层获取数据或向数据层更新数据；
- 处理结果返回给表示层。

数据层
- DBMS；
- 接受功能层的数据查询请求，执行请求，并将查询结果返回给功能层；
- 从功能层接受数据存取请求，并将数据写入数据库；
- 请求的执行结果也要返回给功能层。

B/S结构
浏览器/服务器(B/S)是三层C/S风格的一种实现方式。
- 表现层：浏览器
- 逻辑层：Web服务器、应用服务器
- 数据层：数据库服务器

优点
- 基于B/S架构的软件，系统安装、修改和维护全在服务器端解决，系统维护成本低：
  - 客户端无任何业务逻辑，用户在使用系统时，仅仅需要一个浏览器就可运行全部的模块，很容易在运行时自动升级。
  - 良好的灵活性和可扩展性：只要对业务逻辑层实施相应的改变，就能够达到目的。
- 较好的安全性：在这种结构中，客户应用程序不能直接访问数据，应用服务器不仅可控制哪些数据被改变和被访问，而且还可控制数据的改变和访问方式 。
- 三层模式成为真正意义上的"瘦客户端"，从而具备了很高的稳定性、延展性和执行效率。
- 三层模式可以将服务集中在一起管理，统一服务于客户端，从而具备了良好的容错能力和负载平衡能力。

缺点：客户端浏览器以同步的请求/响应模式交换数据，每请求一次服务器就要刷新一次页面；受HTTP协议"基于文本的数据交换"的限制，在数据查询等响应速度上，要远远低于C/S架构；数据提交一般以页面为单位，数据的动态交互性不强，不利于在线事务处理(OLTP)应用；受限于HTML的表达能力，难以支持复杂GUI 。

DNS 负载均衡：DNS 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：交给 DNS 服务器处理，无须自己开发、维护负载均衡设备。➢就近访问，提升访问速度：DNS 解析时可以根据请求来源 IP，解析成距离用户最近的服务器地址，加快访问速度，改善性能
缺点：更新不及时：DNS 缓存的时间比较长，修改 DNS 配置后，有可能会继续访问修改前的 IP访问失败，影响正常使用业务。➢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：DNS 负载均衡支持的算法少；不能区分服务器的差异；也无法感知后端服务器的状态。
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。如F5 和 A10
优点：功能强大：全面支持各层级的负载均衡，支持全面的负载均衡算法，支持全局负载均衡。性能强大：软件负载均衡支持到 10 万级并发已经很厉害，硬件负载均衡可以支持 100 万以上的并发。稳定性高：商用硬件负载均衡，经过了良好的严格测试，经过大规模使用。支持安全防护：具备防火墙、防 DDoS 攻击等安全功能
缺点：价格昂贵：最普通的一台 F5十多万，好一点近百万元 。扩展能力差：硬件设备，可以根据业务进行配置，但无法进行扩展和定制

软件负载均衡：通过负载均衡软件来实现负载均衡功能。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(百万级)远远高于软件负载均衡性能(万级)
优点：简单：无论是部署还是维护都比较简单；便宜：只要买个 Linux 服务器，装上软件即可；灵活：4 层和 7 层负载均衡可以根据业务进行选择；也可以根据业务进行比较方便的扩展
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。
负载均衡典型架构
地理级别负载均衡：DNS 会根据用户地理位置决定返回哪个机房的 IP
集群级别：F5 收到请求后，进行集群级别的负载均衡
机器级别：Nginx 收到用户请求后将请求发送给集群里面的某台服务器

沟通模型分类
寻址：直接、间接
阻塞：同步、异步
缓存：有、无
消息内容：事件、命令、数据、流
确认：无确认、有确认、三次握手
接收方数量：点对点、多播、任意播、组播多播、广播
沟通方向：单向、全双工、半双工
初始化方式：客户端初始化拉取、服务器初始化推送
队列：发送者和接受者之间扮演一个中间地址目标，各个进程独立执行（发送）和失效，各个进程间的失效和交流失败可忽略(发生了也没关系)——实现了时间解耦、位置解耦
信息优先级：最高优先级优先、权重优先发

特点：异步交互、客户端服务器松散耦合、可靠送达(数据持久存储)、利于应用集成、消息暂存在队列里、进程通过中间消息服务器来传递信息、对于数据库集成很自然
消息模式：一对一、一对多、多对一、多对多(发布/订阅)模式

消息中间件

负载均衡算法
任务数平分类：负载均衡系统将收到的任务平均分配给服务器进行处理。绝对数量的平均或权重上的平均
- **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：负载均衡系统根据服务器的负载来进行分配，再连接数、I/O 使用率、网卡吞吐量等来衡量系统的压力
- **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。解决了轮询算法中无法感知服务器状态的问题，代价是复杂度要增加很多。CPU 负载最低优先的算法要求收集每个服务器的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，时间间隔太长可能造成峰值来临时响应缓慢

性能最优类：负载均衡系统根据服务器的响应时间来进行任务分配，优先将新任务分配给响应最快的服务器。
站在客户端的角度来进行分配，优先将任务分配给处理速度最快的服务器，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。此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。
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

Hash 类：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。例如，当我们通过浏览器登录网上银行时，会生成一个会话信息，这个会话是临时的，关闭浏览器后就失效。网上银行后台无须持久化会话信息，只需要在某台服务器上临时保存这个会话就可以了，但需要保证用户在会话存在期间，每次都能访问到同一个服务器，这种业务场景就可以用源地址 Hash来实现。
- **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）。例如，上述的网上银行登录的例子，用session id hash 同样可以实现同一个会话期间，用户每次都是访问到同一台服务器的目的

冗余高可用计算
集群方案
主备架构和主从架构简析
架构简单：通过冗余一台服务器来提升可用性
问题：人工操作效率低、容易出错、不能及时处理故障
高可用集群方案：可用性要求更加严格的场景中，自动完成切换操作
根据集群中服务器节点角色的不同，分为两类：
- 对称集群（也叫负载均衡集群）
- 非对称集群

主备：主机执行所有计算任务。例如，读写数据、执行操作等。当**主机故障**（例如主机宕机）时，任务分配器不会自动将计算任务发送给备机，此时系统处于不可用状态。如果主机能够恢复，任务分配器继续将任务发送给主机。如果主机不能够恢复(如机器硬盘损坏短时间内无法恢复)：人工操作备机升为主机；任务分配器将任务发送给新的主机；人工增加新的机器作为备机。
根据备机状态的不同，主备架构又可以细分
- 冷备架构：备机上的程序包和配置文件都准备好，但备机上的业务系统没有启动（注意：备机的服务器是启动的），主机故障后，需要人工手工将备机的业务系统启动，并将任务分配器的任务请求切换发送给备机。
- 温备架构：备机上的业务系统已经启动，只是不对外提供服务，主机故障后，人工只需要将任务分配器的任务请求切换发送到备机即可。
冷备可以节省一定的能源，但温备能够大大减少手工操作时间，因此一般情况下推荐用温备的方式。
优点：简单：主备机之间不需要进行交互，状态判断和切换操作由人工执行，系统实现很简单
缺点：需要"人工操作"
主备架构也比较适合内部管理系统、后台管理系统这类使用人数不多、使用频率不高的业务，不太适合在线的业务

主从：主从架构中的从机也要执行任务的，任务分配器需要将任务进行分类，确定哪些任务可以发送给主机执行，哪些任务可以发送给备机执行。正常情况下，主机执行部分计算任务，备机执行部分计算任务。**主机故障时**：
任务分配器不会自动将原本发送给主机的任务发送给从机，而是继续发送给主机，不管这些任务执行是否成功。
如果主机能够恢复(不管是人工恢复还是自动恢复)，任务分配器继续按照原有的设计策略分配任务，即计算任务 A 发送给主机，计算任务 B 发送给从机。
如果主机不能够恢复，则需要人工操作，将原来的从机升级为主机，增加新的机器作为从机，任务分配器继续按照原有的设计策略分配任务。
优点：主从架构的从机也执行任务，发挥了从机的硬件性能
缺点：主从架构需要将任务分类，任务分配器会复杂一些。

异地多活
判断一个系统是否符合异地多活，需要满足两个标准：
正常情况下，用户无论访问哪一个地点的业务系统，都能够得到正确的业务服务
某个地方业务异常的时候，用户访问其他地方正常的业务系统，能够得到正确的业务服务

根据**地理位置上的距离来划分，异地多活架构**可以分为：
同城异区：机房火灾这类问题可以很好地解决；支付宝等金融相关的系统，只能采用同城异区这种架构
跨城异地：解决类似新奥尔良水灾这种问题，适用于对数据一致性要求不那么高的情形。
跨国异地：解决类似美加大停电，更多适用于只读类业务或为不同地区用户提供服务

对称集群：集群中每个服务器角色一样，都可以执行所有任务正常情况下，任务分配器采取某种策略（随机、轮询等）将计算任务分配给集群中的不同服务器。当集群中的某台服务器故障后，任务分配器不再将任务分配给它，而是将任务分配给其他服务器执行。当故障服务器恢复，任务分配器重新将任务分配给它执行。
负载均衡集群的设计关键点在于两点：任务分配器需要选取分配策略
- 任务分配器需要检测服务器状态。

任务分配策略比较简单，轮询和随机基本够用
状态检测稍微复杂：既要检测服务器的状态，例如服务器是否宕机、网络是否正常等；同时还要检测任务的执行状态，例如任务是否卡死、是否执行时间过长等。常用的做法是任务分配器和服务器之间通过心跳来传递信息，包括服务器信息和任务信息，然后根据实际情况来确定状态判断条件。

并行计算架构
并行层次与代码粒度
- 任务level—大粒度—程序
- 控制level—中粒度—函数（线程）
- 数据level—细粒度—语句（编译器）
- multiple issue—非常细粒度—硬件指令（CPU）

并行范式
1. Task-Farming
2. Single-Program Multiple-Data (SPMD)
3. 流水线：取指译码执行写回互相交叉，充分利用资源
4. 分治：分离、计算和合并
5. 预测并行
6. 参数计算模型
   - Master-Worker Model：Master把内容给分解成小的任务，分发给workers，并且汇聚最后产生的结果
   - work stealing：每个worker维护一个任务队列，为空时随机去其他的worker的队列中stealing任务

阿姆达尔定律：忽略所有系统或通信开销的情况下，N处理器并行的加速比（r_s, r_p分别表示串行、并行执行的比例）

$$Speedup = \frac{1}{r_s + \frac{r_p}{N}}$$

非对称集群：集群中的服务器分为多个不同的角色，不同的角色执行不同的任务。例如最常见的 Master-Slave 角色。部分任务是 Master 服务器才能执行，部分任务是 Slave 服务器才能执行。
集群会通过某种方式来区分不同服务器的角色。
例如，通过 ZAB 算法选举，或者简单地取当前存活服务器中节点 ID 最小的服务器作为 Master 服务器。
任务分配器将不同任务发送给不同服务器。
当指定类型的服务器故障时，需要重新分配角色：Master 服务器故障后，需要指定一个 Slave 服务器为 Master 服务器；Slave 服务器故障，不需要重新分配角色，只需要将其剔除即可
非对称集群相比负载均衡集群，设计复杂度主要体现在两个方面：1.任务分配策略更加复杂：需要将任务划分为不同类型并分配给不同角色的集群节点。2.角色分配策略实现比较复杂：例如，可能需要使用 ZAB、Raft 这类复杂的算法来实现 Leader 的选举。
以 ZooKeeper 为例：
任务分配器：ZooKeeper 中不存在独立的任务分配器节点，每个 Server 都是任务分配器，Follower 收到请求后会进行判断，如果是写请求就转发给 Leader，如果是读请求就自己处理。
角色指定：ZooKeeper 通过 ZAB 算法来选举 Leader，当Leader 故障后，所有的 Follower 节点会暂停读写操作，开始进行选举，直到新的 Leader 选举出来后才继续对 Client 提供服务

设计并行算法的四个阶段
1. **划分**：分解成小的任务，开拓并发性。先域分解再功能分解使数据集和计算集互不相交；
2. **通讯**：确定诸任务间的数据交换，监测划分的合理性。四种通讯模式：局部/全局通讯、结构化/非结构化通讯、静态/动态通讯、同步/异步通讯
3. **组合**：依据任务的局部性，组合成更大的任务。合并小尺寸任务，减少任务数
4. **映射**：将每个任务分配到处理器上，提高算法的性能

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
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。

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；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

主备复制
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，客户端无论读写操作，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，不能读写数据，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，切换访问链路。
6. 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。
7. 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。 在此类方案中，备机只起到备份作用

优点：1. 对于客户端来说，不需要感知备机的存在，即使灾难恢复后，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换了，无需知道是原来的备机升级为主机了。2. 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
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，则会出现主从读取数据不一致问题
8. 如果主从间延迟较多，恰好主机又宕机了，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

优缺点（相对主备复制而言）：主从复制在主机故障时，读操作相关的业务不受影响。主从复制架构的从机提供读操作，发挥了硬件的性能。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。

主从倒换与主备倒换
主备和主从复制的共性问题：主机故障后，无法进行写操作。如果主机无法恢复，需要人工指定新的主机。
关键的设计点：主备间状态判断、倒换决策、数据冲突

常见架构
互联式：主备机直接建立状态传递的渠道。**缺点**：状态传递通道本身故障了，则备机会主动升级为主机；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。结构复杂了，但是从状态传递和决策上更加简单：
- 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。
- 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。决策步骤见下一页

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：实现更加简单，省去了状态传递通道的建立和管理工作。模拟式读写操作获取状态信息只有响应信息（如HTTP404、超时等），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。

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，备机将主机升级为主机。
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，按照实际的状态决定是否进行倒换，如响应超时

数据集群
数据集中集群：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：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；多通道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；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；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：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。
4. 数据分散集群中，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

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。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。正常情况下，客户端可以将读写操作发送给任意一台主机。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。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。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。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。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。
相较之下的**优缺点**：两台主机，无倒换概念；客户端无须区分主备机身份；必须保证数据能够双向复制，然而很多数据无法双向复制，如售票

集中VS分散
- 写数据角色
  - 数据集中集群架构中，客户端只能将数据写到主机
  - 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
  - 数据集中集群适合数据量不大，集群机器数量不多的场景，如ZooKeeper 集群，一般个位数机器
  - 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，如Hadoop 集群，可达上千台服务器

分库分表

操作时机
- 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

1. 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。具实现方式分为三种：
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。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。如mybatis

2. 代理分片：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。而且需要维护代理层增加了人员和硬件的成本

3. 支持事务的分布式数据库：有很多产品如OceanBase等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

垂直切分：按照业务将表进行分类或分拆，将其分布到不同数据库上。1. 不同业务模块的数据可以分散到不同数据库服务器；2. 也可以冷热分离，根据数据的活跃度将数据进行拆分；3.也可以人为将一个表中的内容划分为多个表，例如将查询较多，变化不多的字段拆分成一张表放在查询性能高的服务器，而将频繁更新的字段拆分并部署到更新性能高的服务器。

- 冷数据：变化更新频率低，查询次数多的数据。(冷数据的数据库可以使用MyISAM引擎，查询性能好
- 热数据：变化更新频率高活跃的数据。(热数据的数据库可以使用InnoDB存储引擎，更新性能好)
- 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(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。即将数据按一定规律，按行切分，并分配到不同的库表里，表结构完全一样。
- 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。
在设计表时需要确定对表按照什么样的规则进行分库分表。例如，当生成新用户时，程序得确定将此用户的信息添加到哪个表中。同样，在登录时我们需要通过用户的账号找到数据库中对应的记录。
- 分片维度：1）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一个分片，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。
2）按照时间切片：按照时间的范围将数据分布到不同的分片

水平切分优点：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
不同点：垂直更偏向于业务拆分的过程，水平更偏向于技术性能指标

分库分表引起的问题
1. 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。
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）

2. 分布式事务问题：多库多表分布式所引发的一致性问题。
3. 同组数据跨库问题：要尽量把同一组数据放到同一台数据库服务器上，不但在某些场景下可以利用本地事务的强一致性，还可以使这组数据实现自治。

4. 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，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。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。

分布式事务
CAP理论：一个分布式系统不能同时满足CAP需求，最多只能同时满足两个。
- Consistency(一致性)：多点的数据是一致的；
- Availability(可用性)：系统随时都是可用的；
- Tolerance of Network Partition(分区容忍性)：在出现网络分区的情况下，分离的系统也能正常运行
对于分布式存储系统而言，P是基本需求，只有两种模式的选择。
- CP模式：保证分布在网络上不同节点数据一致性，但对可用性支持不足。
- AP模式：以实现“最终一致性”来确保可用性和分区容忍性，但弱化了数据一致性要求

事务路由：无论使用哪种分布式事务处理方法，都需要对分库分表的多个数据源路由事务。如果更新操作在一个数据库实例内发生，便可以使用数据源的事务来处理。对于跨数据源的事务，可通过在应用层使用最大努力保证模式和事务补偿机制来达成事务的一致性。这就需要通过编写程序来选择数据库的事务管理器，即事务路由。
- 自动提交事务路由
- 可编程事务路由
- 声明式事务路由

分片后的事务处理机制
1. 最大努力保证模式：适用于对一致性要求并不十分严格，但是对性能要求较高的场景。具体实现方法：在更新多个资源时，将多个资源的提交尽量延后到最后一刻处理，如果业务流程出现问题，则所有资源更新都回滚，保持事务一致。以消息队列消息消费和更新数据库为例：1、开始消息事务；2、开始数据库事务；3、接收消息；4、更新数据库；5、提交数据库事务；6、提交消息事务
2. 事务补偿机制：在数据库分库分表后，如果涉及的多个更新操作在某一个数据库范围内完成，则可以使用数据库内的本地事务保证一致性。
- 对于跨库的多个操作，可通过补偿和重试，使其在一定时间窗口内完成操作
- 这样既保证了事务的最终一致性，又突破了事务遇到问题就回滚的传统思想。
- 如果采用事务补偿机制，则在遇到问题时，需要记录遇到问题的环境、信息、步骤、状态等，后续通过重试机制使其达到最终一致性。

3和4见下一页

3. 两阶段提交协议
1. 请求阶段(表决)：事务协调者通知每个参与者准备提交或取消事务，然后进入表决过程，参与者要么在本地执行事务，写本地的redo和undo日志，但不提交。请求阶段，参与者将告知协调者自己的决策：同意(事务参与者本地作业执行成功)或取消（本地作业执行故障）
2. 提交阶段(执行)：在该阶段，写调整将基于第一个阶段的投票结果进行决策：提交或取消。当且仅当所有的参与者同意提交事务，协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务，参与者在接收到协调者发来的消息后将执行响应的操作。
- 缺点：同步阻塞问题；单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）
- 数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这会导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。

4. 三阶段提交协议：针对“单点故障”问题，在第一、二阶段间加入“准备阶段”，当协调者故障后，参与者可以通过超时提交来避免一致阻塞。
(1) canCommit阶段：3PC的canCommit阶段其实和2PC的准备阶段很像。协调者向参与者发送commit请求，参与者如果可以提交就返回yes响应，否则返回no响应
(2) preCommit阶段：协调者根据参与者canCommit阶段的响应来决定是否可以继续事务的preCommit操作。 a) 协调者从所有参与者得到的反馈都是yes：那么进行事务的预执行，协调者向所有参与者发送preCommit请求，并进入prepared阶段。参与者接收到preCommit请求后会执行事务操作，并将undo和redo信息记录到事务日志中。如果一个参与者成功地执行了事务操作，则返回ACK响应，同时开始等待最终指令 b) 协调者从所有参与者得到的反馈有一个是No或是等待超时之后协调者都没收到响应：那么就要中断事务，协调者向所有的参与者发送abort请求。参与者在收到来自协调者的abort请求，或超时后仍未收到协调者请求，执行事务中断。
(3) doCommit阶段：协调者根据参与者preCommit阶段的响应来决定是否可以继续事务的doCommit操作。 a) 协调者从参与者得到了ACK的反馈：协调者接收到参与者发送的ACK响应，那么它将从预提交状态进入到提交状态，并向所有参与者发送doCommit请求。参与者接收到doCommit请求后，执行正式的事务提交，并在完成事务提交之后释放所有事务资源，并向协调者发送haveCommitted的ACK响应。那么协调者收到这个ACK响应之后，完成任务。 b) 协调者从参与者没有得到ACK的反馈, 也可能是接收者发送的不是ACK响应，也可能是响应超时：执行事务中断。

数据缓存的基本操作
- 命中与验证：HTTP 再验证 If-Modified-Since
- 基本清洗策略：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。策略：FIFO、最少使用策略、最近最少使用策略
- 其他清洗策略：根据过期时间判断，清理过期时间最长的元素/清理最近要过期的元素；随机清理；根据关键字长短清理等

数据缓存

- 更新策略
- Cache aside：1. 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。2. 命中：应用程序从cache中取到数据后返回。3. 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，也就是说，当缓存失效的时候，Cache Aside是由调用方负责把数据加载入缓存，而Read Through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(同步操作)。
- Write Behind Caching Pattern：俗称写回，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。**好处**是让数据的I/O操作飞快，因为异步，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。**问题**在于数据的非强一致性，极可能造成数据的丢失，一旦挂了，可能造成数据丢失且无法恢复。

数据一致性：因为缓存属于持久化数据的一个副本，所以不可避免的会出现数据不一致问题。一般是因为网络不稳定或节点故障导致。常见3个场景以及解决方案：
1. 先写缓存，再写数据库：缓存写成功，但写数据库失败或响应延迟，则下次读取(并发读)缓存时，就出现脏读；【解决】这个写缓存的方式，本身就是错误的，需要改为先写持久化介质，再写缓存的方式。2. 先写数据库，再写缓存：写数据库成功，但写缓存失败，则下次读取（并发读）缓存时，则读不到数据；【解决1】根据写入缓存的响应来进行判断，如果缓存写入失败，则回滚数据库操作。该方法增加了程序的复杂度；【解决2】缓存使用时，假如读缓存失败，先读数据库，再回写缓存。3. 缓存异步刷新：指数据库操作和写缓存不在一个操作步骤中，比如在分布式场景下，无法做到同时写缓存或需要异步刷新；【解决】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：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，使数据库压力过大甚至崩溃。解决：将空值缓存，再次接收到同样的查询请求时，若命中缓存并值为空就直接返回，不会透传到数据库；对恶意的查询攻击，可以对查询条件设置规则，不符合的直接拒绝
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，性能降低甚至崩溃。解决：
- 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，通过本地锁限制只有一个线程去数据库中查询数据。节点数量较多时并未完全解决缓存并发的问题
- 软过期：对缓存中的数据设置失效时间，不使用缓存服务提供的过期时间。

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，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

缓存问题

缓存高可用：缓存是否高可用，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，需要结合具体业务，具体情况进行方案设计，例如临界点是否对后端的数据库造成影响。解决方案：
- 分布式：实现数据的海量缓存
- 复制：实现缓存数据节点的高可用
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。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

数据缓存

本地缓存：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
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。
1. 静态变量实现：通过静态变量一次获取到缓存内存中，进程内可共享，缓存的实时性稍差。编程直接实现缓存，优点是能直接在heap区内读写，最快也最方便；缺点同样是受heap区域影响，缓存的数据量非常有限，同时缓存时间受GC影响。
2. 中间件Ehcache实现
分布式缓存：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

缓存作用：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。**原理**：将数据写入到读取速度更快的存储；将数据缓存到离应用最近的位置；将数据缓存到离用户最近的位置

分布式缓存的迁移
平滑迁移：双写->迁移历史数据->切读->下线双写。具体过程与分库分表的扩容与迁移相同
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）。将各个服务器使用Hash(服务器的ip或主机名)进行一个哈希，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。
数据倾斜问题：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。此时必然造成大量数据集中到一个节点上。**解决**：引入虚拟节点机制，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。迁移历史数据；2、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。

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。
CDN缓存：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

面向异步内容加载的前端架构技术
- 问题：加载渲染内容多，一次性加载，带宽限制了响应速度
- 解决方案：单任务分解为多任务，多任务异步处理
- 技术方案：1、浏览器要求加载动态内容。2、浏览器渲染静态内容。3、浏览器引擎要求不同服务器给数据。4、各服务器异步返回数据。5、浏览器引擎通知浏览器数据到了。6、浏览器异步渲染返回的数据
- AJAX：是一种异步请求数据的web开发技术，对于改善用户的体验和页面性能很有帮助
面向自适应的前端框架技术
- 问题：多屏适配之痛
- Bootstrap：基于 HTML、CSS、JAVASCRIPT 的快速开发 Web 应用程序和网站的前端框架。移动设备优先、浏览器支持、容易上手、响应式设计
前端组件化——抽象与提炼：
- 把每个组件当做一个独立产品来设计考虑空状态、极端情况、尺寸变化尽可能灵活适应各使用场景
- 统一组件与组件、组件与散件之间的组合方式
- 之前所有的内容，最终汇集成设计规范
- 轻度组件化：同一个组件使用相同的html结构和class名，用同一段css代码定义样式，用同一个js函数定义交互。
- 重度组件化：每个组件的html结构、css样式、js交互都独立封装管理，定义好框架和加载方式，内容在加载时从外部填充。
- 改变开发的协作方式。大家不再是按页面分工，而是按组件来分工。弱化了相互间的依赖关系
- 重在维护：组件化不是一件轻松的工作，在项目初期的准备工作会增加一定工作量，但随时间推移会发挥出巨大的优势。

软件架构的评审方法

系统的质量属性
- 终端用户的视角：性能、可靠性、可用性、安全性
- 商业视角：上市时间、成本和收益、项目周期、目标市场、与原有系统的集成性
- 开发者的视角：可维护性、轻便性、复用性、可测试性
三类主要的评审方式
- **基于调查问卷的评审方式**。评估过程：多个评估专家考察系统，然后回答问卷中的问题；对多个评估结果进行综合，得到最终的结果。**优点**：自由、灵活，可评估多种质量属性，也可在SA设计的多个阶段进行。**缺点**：由于评估的结果很大程度上来自评估人员的主观推断，因此不同的评估人员可能会产生不同甚至截然相反的结果；评估人员对领域的熟悉程度、是否具有丰富的相关经验也成为评估结果是否正确的重要因素。
- **基于场景的评审方式**。分析SA对系统应用的典型场合的支持程度，从而判断该SA对这一场景所代表的质量需求的满足程度。此方式由SEI首先提出：
  - 软件体系结构分析方法(SAAM) - 体系结构权衡分析方法(ATAM)
    - ATAM输入：场景（用例场景-常规应用场合、变化性场景-可预料到的变化、探测性场景-极端变化）、需要关注的质量属性及对其的提问（性能、可修改性等）、待评价的SA的设计方案
    - 中间结果：效用树-提供了一种自顶向下的机制来将系统的业务驱动要素翻译为具体的质量属性及其场景
    - 输出：风险、非风险、关键点、折中点
- **基于度量的评审方式**。设计质量指标的具体度量方式(数学公式)。在度量时，可采用自动化的工具，对质量属性进行计算，得到结果。三个基本活动：1、从软件体系结构文档中获取度量信息并计算得到结果；2、建立质量属性和度量结果之间的映射原则，即确定怎样从度量结果推出系统具有什么样的质量属性；3、根据映射原则分析推导出系统的某些质量属性。
优点：结果比较客观、精确。**缺点**：很多质量属性无法给出具体的计算公式；能给出计算公式的质量属性，往往是针对源代码级别的质量属性(如代码行数等)，而这些属性对SA的评价往往缺乏足够的意义；需要在SA设计基本完成以后才能进行；需要评估人员对待评估的体系结构十分了解。

软件架构测试方法
性能测试：负载测试、压力测试、容量测试、网络性能测试
性能测试步骤：制定目标和分析系统、选择测试度量的方法、选择相关技术和工具、制定评估标准、设计测试用例、运行测试用例、分析测试结果
性能测试指标：响应时间、内存、磁盘、处理器、网络