# 实验十一

实验十一 软件体系结构设计（三）

实验目的：

1. 深入理解体系结构的设计和评估改进
2. 完成SAD

实验内容：

1. 继续完成上周的实验任务 （有同学反映上周实验内容较多）

1.学习、检索课本5.17参考文献及以下推荐的参考书或网上检索新的有关软件体系结构的资料。小组分工，每位成员选择自己关注的部分专题学习并写出学习报告（笔记）（附到最终提交的SAD）。

**邢乐凯——事件驱动架构学习报告**

伴随企业数字化进程进入“深水区”，企业面临着日益复杂的 IT 系统和业务流程，不同系统间的壁垒导致企业运转效率下降以及协同摩擦增加。而事件驱动架构（Event-Driven Architecture，EDA）已成为解决这些问题的关键技术。

Gartner 将事件驱动架构（EDA）列为十大战略技术趋势之一，并强调事件驱动架构 (EDA) 是技术和软件领域发展的主要驱动力。EDA 是实时敏捷数字业务的核心，通过“监听”物联网 (IoT) 设备、移动应用程序、生态系统以及社交和业务网络等事件源，以数字形式实时捕获真实世界的业务事件。

过去，以 API 为中心的应用程序设计架构显著提升了业务的敏捷性，但随着数字化业务场景越来越复杂，企业对于实时智能、敏捷响应、上下文自适应等需求增加。仅依靠 API 为中心的请求驱动模型就显得力不从心。随着越来越多数字化系统的接入，应用程序变得更难拓展，连接的 API 网络更难管理，不同系统间的数据壁垒越筑越高，系统的耦合度越来越高。而依托事件驱动架构异步、松耦合的特性，将各个业务系统解耦，降低系统间的依赖程度，最大程度提升企业数字敏捷性。

**什么是事件?**

事件，本质上就是运动、变化，跟“函数”、“消息”、“操作”、“调用”、“算子”、“映射”等概念全息。在计算机领域里指：可以被控件识别的操作，如按下确定按钮，选择某个单选按钮或者复选框。每一种控件有自己可以识别的事件，如窗体的加载、单击、双击等事件，编辑框（文本框）的文本改变事件，等等。

事件有系统事件和用户事件。系统事件由系统激发，如时间每隔24小时，银行储户的存款日期增加一天。用户事件由用户激发，如用户点击按钮，在文本框中显示特定的文本。事件驱动控件执行某项功

能。触发事件的对象称为事件发送者；接收事件的对象称为事件接收者。事件就是用户对窗口上各种组件的操作。使用事件机制可以实现：当类对象的某个状态发生变化时，系统将会通过某种途径调用类中的有关处理这个事件的方法或者触发控件事件的对象就会调用该控件所有已注册的事件处理程序等。

在.net框架中，事件是将事件发送者（触发事件的对象）与事件接受者（处理事件的方法）相关联的一种代理类，即事件机制是通过代理类来实现的。当一个事件被触发时，由该事件的代理来通知（调用）处理该事件的相应方法。

C#中事件机制的工作过程如下：

1、将实际应用中需通过事件机制解决的问题对象注册到相应的事件处理程序上，表示今后当该对象的状态发生变化时，该对象有权使用它注册的事件处理程序。

2、当事件发生时，触发事件的对象就会调用该对象所有已注册的事件处理程序。

**什么是事件驱动架构?**

以下是一个经典的事件驱动用例：

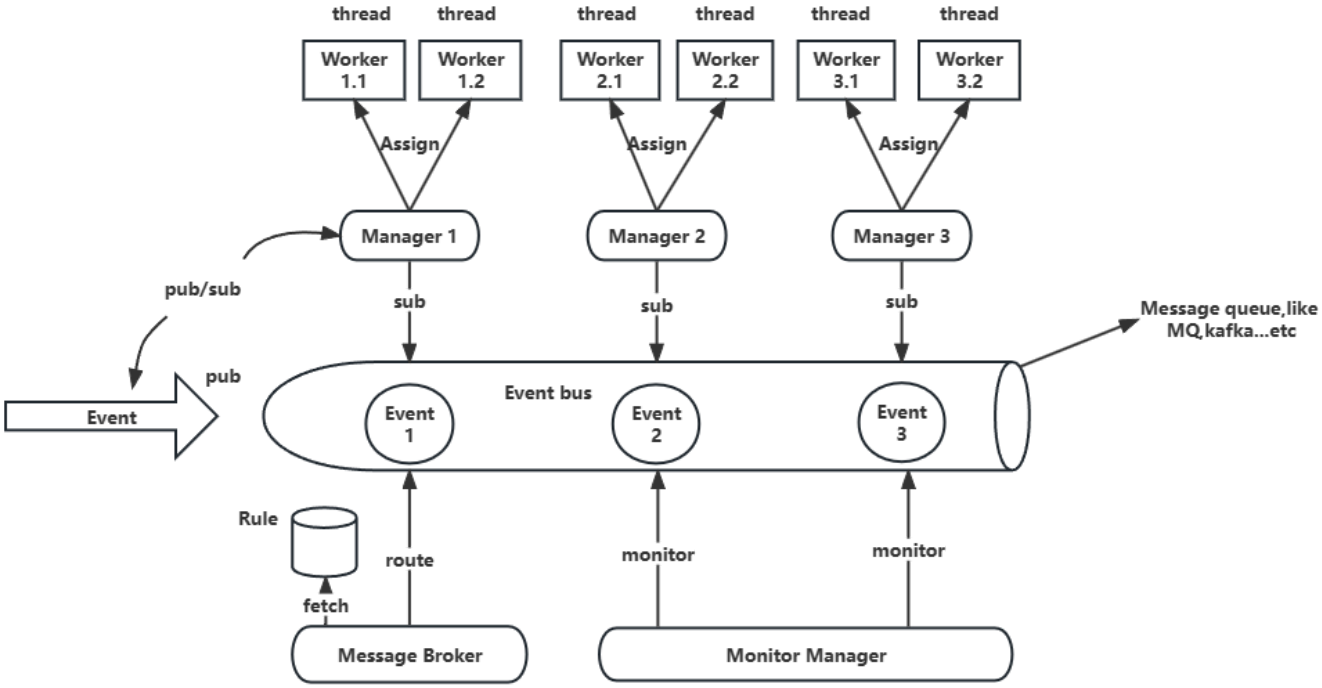
当你的公司需要招聘一名新员工，基础流程包括：发布职位广告→面试→录用→人事准备→入职程序→工作安排。这是一个基础的招聘流程，在这个流程中，每一个步骤都是基于一个特定的事件来进行的。例如，招聘过程是基于公司需要新员工这个事件的触发来进行的；面试是基于收到应聘者申请这个事件的触发来进行的；录用是基于面试考核通过这个事件的触发来进行的。

该用例是企业标准的招聘流程或系统，企业管理者只需要搭建好整套招聘流程和拟定标准，以及每个节点所需要的人员。在管理者需要招聘新员工时，只需要发布指令，该招聘系统则会自动依照流程运作，直至返回招聘结果。因此，以上述场景为例，可以对事件驱动架构进行如下描述：

事件驱动架构模式是一种非常流行的分布式异步架构模式，经常被用与构建高可伸缩性的应用程序。当然它也适合小型应用，复杂应用和规模比较大的应用。这种架构模式由一系列高度解耦的、异步接收和处理事件的单一职责的组件所组成，可以理解为架构层面的观察者模式。

事件驱动架构主要分为以下7个核心对象，具体的协同模式可以参考下方绘制的原理图。

- 事件 (Event)：即要被处理的对象，它可以是离散的也可以是有序的；其格式既可以是JSON，也可以是XML或者银行专用的8583报文；
- 事件巴士（Event bus）：负责接收从外部推送过来的event并作为同一个event在不同manager之间流转的载体；一般的选型可以使用MQ、Kafka或者Redis（当然Redis的pub/sub模式不能消息持久化）；
- Worker Manager：负责把订阅主题拿到的event分配给Worker；
- Worker：作为执行者对event进行业务处理并做出响应；
- MonitorManager：作为监控者负责监控event的处理，可以看作是event的守护线程。
- Message Broker：作为一个消息中介，负责分配和协调多个worker（或者我们可以称之为工序）针对该event的处理顺序，且该broker负责维护该处理顺序所用到的路由规则。




执行的大致流程为：

- Event作为一个事件被发布到Event bus；
- Message Broker和Rule是搭配干活的。所有的事件对自己的“我从哪里来”和“我要到哪里去”是完全不清楚的，由Message Broker根据Rule所定义的路由规则进行分派给对应的WorkManager；
- 当WorkManager在订阅的主题里面发现有对应的事件后，则会根据当时的worker的负载情况进行工作分配；接着，worker就会执行对应的业务处理流程；
- 贯穿整个过程，会由MonitorManager负责监听每个事件的状态。具体的实现是（以Kafka为例）可以约定所有的WorkManager针对处理异常的事件全部丢到一个死信主题，然后由MonitorManager负责订阅监听该主题并进行相应的告警处理。

**事件驱动跟消息驱动机制比较**

通常，我们写服务器处理模型的程序时，有以下几种模型：

(1)每收到一个请求，创建一个新的进程，来处理该请求；

(2)每收到一个请求，创建一个新的线程，来处理该请求；

(3)每收到一个请求，放入一个事件列表，让主进程通过非阻塞I/O方式来处理请求

上面的几种方式，各有千秋，

第一种方法，由于创建新的进程的开销比较大，所以，会导致服务器性能比较差,但实现比较简单。

第二种方式，由于要涉及到线程的同步，有可能会面临死锁等问题。

第三种方式，在写应用程序代码时，逻辑比前面两种都复杂。

综合考虑各方面因素，一般普遍认为第三种方式是大多数网络服务器采用的方式

在UI编程中，常常要对鼠标点击进行相应，首先如何获得鼠标点击呢？

方式一：创建一个线程，该线程一直循环检测是否有鼠标点击，那么这个方式有以下几个缺点：

CPU资源浪费，可能鼠标点击的频率非常小，但是扫描线程还是会一直循环检测，这会造成很多的CPU资源浪费；如果扫描鼠标点击的接口是阻塞的，又会出现下面这样的问题，如果我们不但要扫描鼠标点击，还要扫描键盘是否按下，由于扫描鼠标时被堵塞了，那么可能永远不会去扫描键盘；另外，如果一个循环需要扫描的设备非常多，这又会引来响应时间的问题；所以，该方式是非常不好的。

方式二：就是事件驱动模型

目前大部分的UI编程都是事件驱动模型，如很多UI平台都会提供onClick()事件，这个事件就代表鼠标按下事件。事件驱动模型大体思路如下：

有一个事件(消息)队列；当鼠标按下时，往这个队列中增加一个点击事件(消息)；有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如onClick()、onKeyDown()等；事件(消息)一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；

**事件驱动架构的适用范围**

1、整个交易处理链路较长的准实时或非实时场景，例如票据管理，在复杂的模式匹配种，事件可能被串在一起以推断更复杂的事件。

2、基于fan-out的广播类型场景，例如移动商城抢购后需要跟进的一系列动作（短信通知、发货申请、更新订单状态等），这类场景一般是非实时，是time-tolerance（接受一定时间容差）；

3、削峰填谷的场景。例如，上游应用系统推送了大量的系统日志至ELK，ELK更多是进行入库和统计分析，不需要对上游做实时响应的。

4、如果业务模式的整个主流程不强调强一致性且流程变化很快的，则可以适当的考虑这种架构。如不透明的消费者生态系统：生产者通常不了解消费者的情况。后者甚至可能是短暂的过程，可以在短时间内来去匆匆！

5、命令查询职责分离。CQRS 是一种将数据存储的读取和更新操作分开的模式。实施 CQRS 可以提高应用程序的可扩展性和弹性，但需要权衡一些一致性。这种模式通常与 EDA 相关联。

注：由于EDA是通过管道进行异步通信的，如果系统是那些对交易实时性要求较高的或者是跟2C端页面交互强关联的，则不太建议使用该异步架构；

**EDA的好处和优势**

1、在这种模式下，系统一般被分解成多个独立又互相有一定关联关系的服务或模块，这种模式真正体现高内聚低耦合，很好的体现Y轴扩展。例如一个票据处理系统就是这种EDA架构，每个worker只负责一个工序（满足高内聚），当需要新增额外工序的时候只需要继承基类新增新类型的worker，并配套新rule即可。

2、高内聚带来的好处就是，每当新增功能时大概率只需要改动某个节点的worker，改动的影响可以被限定在一定范围内（即某个worker内部）。

3、worker理论上可以无限水平扩展以便支持大规模的业务量；当manger变成瓶颈后，也可以适当把manager从单实例扩展成集群；

4、基于事件（event）实际上持久化到Event bus ，因此便于做差错处理，提升了系统整体的可运维性。例如，event1在manager2处理失败后即不会继续往后处理，方便IT人员排查并修复后把该event重新路由至同一个manager下进行处理。

5、提供缓冲和容错。事件可能以与其生产不同的速率被消费，生产者不能放慢速度让消费者赶上。

6、生产者和消费者解耦，避免笨拙的点对点集成。向系统添加新的生产者和消费者很容易。只要遵守约束事件记录的合同/模式，更改生产者和消费者的实现也很容易。

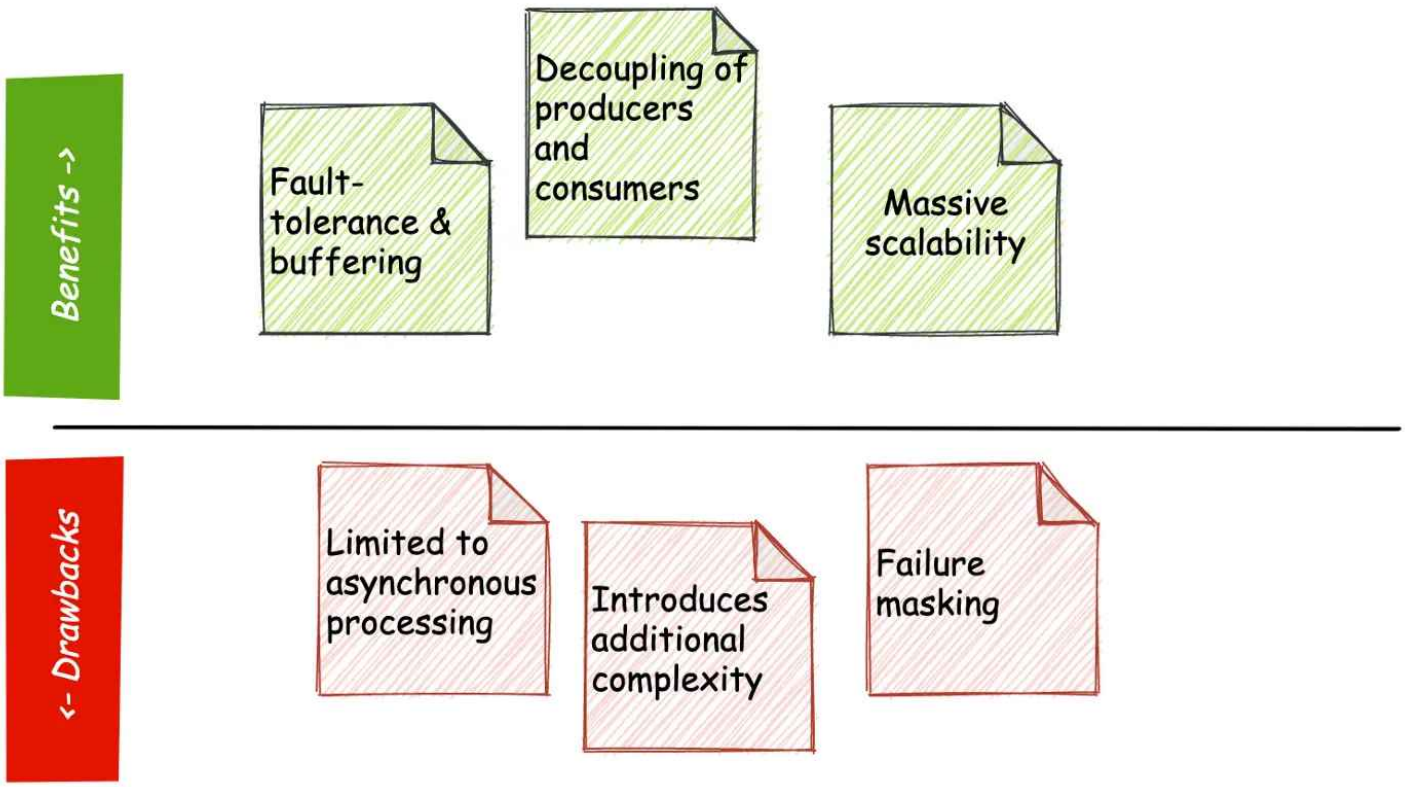
7、大规模的可扩展性。通常可以将事件流划分为不相关的子流并并行处理这些子流。如果事件积压增加，我们还可以扩展消费者数量以满足负载需求。像 Kafka 这样的平台能够以严格的顺序处理事件，同时允许跨流的大规模并行性。

**EDA的缺点**

1、仅限于异步处理。虽然 EDA 是一种用于解耦系统的强大模式，但它的应用仅限于事件的异步处理。EDA 不能很好地替代请求-响应交互，其中发起者必须等待响应才能继续。

2、引入了额外的复杂性。传统的客户端-服务器和请求-响应式计算仅涉及两方，而采用 EDA 则需要第三方——代理来调解生产者和消费者之间的交互。

3、故障屏蔽。这是一个特殊的问题，因为它似乎与解耦系统的本质背道而驰。当系统紧密耦合时，一个系统中的错误往往会迅速传播，并经常以痛苦的方式引起我们的注意。在大多数情况下，这是我们希望避免的：一个组件的故障应该对其他组件的影响尽可能小。故障掩蔽的另一面是它无意中隐藏了本应引起我们注意的问题。这是通过向每个事件驱动组件添加实时监控和日志记录来解决的，但这会增加复杂性。



**EDA幂等性**

既然使用到EDA这种事件触发型的架构模式，势必会面临一个以下常见的场景：

- 由于路由规则错误导致的同一个event被重复多次路由到同一个manager进行处理；
- event被重复消费（例如可能来自于kafka的再平衡）；
- 人工从死信主题中被重新捞出来处理。

因此，幂等性设计在这种架构下就显得尤为必要。所有的业务流程或操作在数据库视野上归根结底就是事物状态的变更和查询两大类。如果是查询类的操作，那幂不幂等这个无关重要。如果是变更类的操作，那就需要考虑幂等的设计。一般来说，幂等性可以通过token、状态码、乐观锁等方式实现。

幂等性在接口层面非常关键，许多系统都会出现由于幂等性设计不足导致一些生产故障。以下通过查询资料对于如何解决这一问题进行总结：

1、去重表

基本原理：通过设计一张独立的表，该表拥有一个唯一索引（可以是独立索引或联合索引），然后当请求进来时，通过数据库的唯一性约束，如果插入正常的则继续执行，失败则返回失败。

具体做法：

在服务端定义一个接口，在接口中要求客户端上送客户端自己的UUID_1，然后服务端在处理的时候会生成服务端自己的UUID_2，并在该独立表中把UUID_1和UUID_2作为唯一索引进行insert操作。这样的话，就算是因为用户在前端重复提交，但在服务端都会因执行insert操作失败（DuplicateKeyException）而被成功拒绝掉。

这个方案不好的地方在于有数据库操作，因此在常规并发不高的情况下可以考虑，毕竟综合成本相对较低（复杂度和成本）,这才符合KISS原则。

2、token+redis机制

token 机制其实就是服务端提供两个接口：①提供token的接口；②真正的业务接口。

首先，客户端先去服务端申请token，服务端返回token并缓存至redis用于后面的校验；接着，客户端带着token去调用服务端的业务接口，服务端先尝试删除redis中的token：如果删除成功，则往下处理业务逻辑；如果删除失败，则代表相同接口被重复调用过，即这请求为重复请求，服务端直接拒绝。

这个方案的成本必要性：首先，这个机制要求客户端每次都需要调用两次接口，但是重复的情况肯定不是常态。那就是说为解决那1%的问题要99%的请求都要按照这种模式。

3、状态码机制

状态码其实就是在交易表或主表中增加一个针对该交易的状态（前提是该表有个交易的全局唯一流水），针对提交进来的具有相同交易流水的交易通过该状态避免重复提交的情况。

这种做法一般在非2C的系统用得比较普遍，因为大部分情况下不太需要考虑流量、并发等因素。

4、乐观锁机制

具体做法：在要更新的表中，通过增加一个字段（这里可以使用#version或者#timestamp）作为一个版本号字段。

5、分布式锁机制

基本原理：

这里以redis实现的方式为例。如果基于redis，主要使用SETNX+EXPIRE实现分布式锁（但是锁超时时间要取决于业务场景，或者说一般要大于调用方的超时时间）。这样的话，在锁超时时间内，同一个接口的重复提交（当然指的是接口参数是一样的情况下）会因为在调用SETNX失败而成功拒绝掉重复提交。

具体做法：

自定义一个幂等注解，然后配合AOP进行方法拦截，对拦截的请求信息(包括方法名+参数名+参数值)根据固定的规则去生成一个key，然后调用redis的setnx方法，如果返回ok，则正常调用方法，否则就是重复调用了。这样可以保证重复请求接口在一定时间内只会被成功处理一次。

具体的SETNX实现分布式锁还是有一些因key超时带来的锁释放的细节问题。

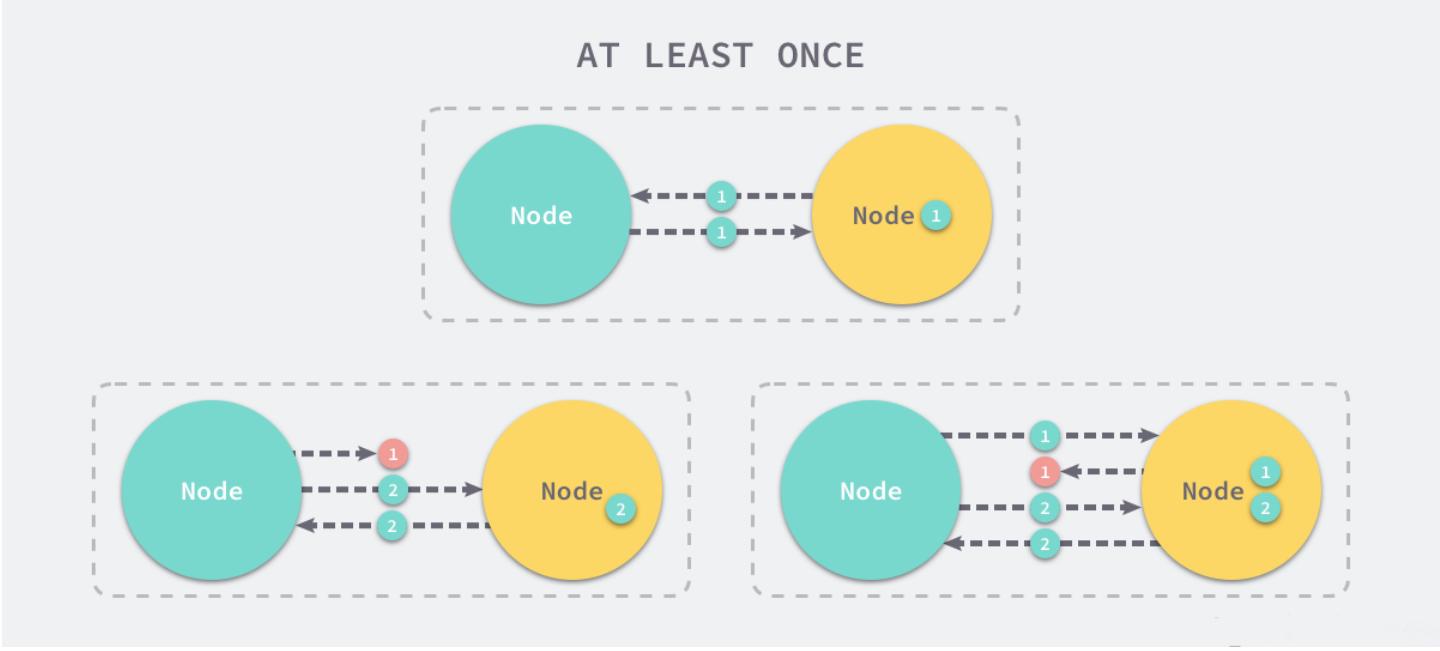
以上几种方式其实都是有个共通点，通过给某个业务请求生成或赋予一个唯一对应的令牌（如token，或乐观锁的#version），然后服务端针对业务接口的调用进行令牌校验，如果不能满足规则则拒绝处理。当然，上面的每种方法也有其局限性，因此在用于生产的设计方案中一般都是以上两种或多种方法的组合体。最关键的是，具体每个方案怎么组合或者组合到哪种程度是要与实际业务场景相结合的，总不能为了所谓的技术追求而犯了教条主义的错。

**EDA最终一致性**

EDA架构是通过实现可靠事件模式来达成业务层的最终一致性的。什么是可靠事件模式？可靠事件其实就是保证事件（event）能够被成功投递、接收及处理，简直TCP链接的增强版。其中可靠性通过以下三个维度进行可靠性保证。Talk is cheap, show u the pic。

1、投递可靠性

首先，EDA架构下的消息巴士一般采用各种消息中间件作为消息传递的桥梁，而主流的开源消息中间件（例如RabbitMQ/RocketMQ/Kafka）使用的都是At least Once的投递机制（即每个消息必须投递至少一次），简单点说就是消息发送者（这里指“事件巴士”）发送消息至消息接受者（这里指“下游”）并且要监听响应，如果在规定指定时间内没有收到，则消息发送者会按某种频次重新发送该消息直到收到响应为止。当然，如果是“上游”投递至“事件巴士”也需要从上游的应用层面做可靠性的容错处理。




2、传输可靠性

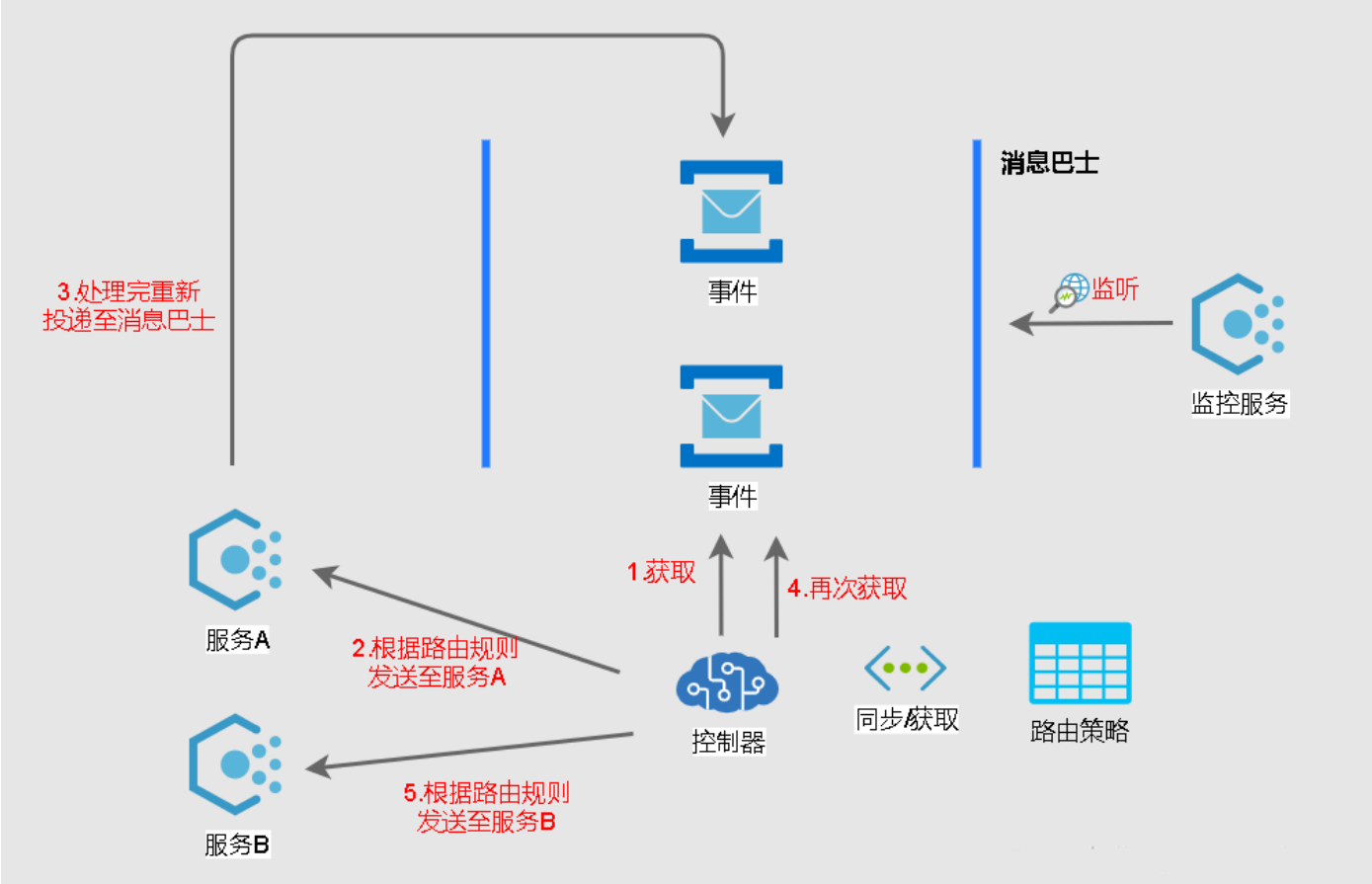
因为架构中使用消息中间件，目前大部分中间件都有对应的消息持久化机制，保证数据在没有被下游成功确认收到前不会丢失，哪怕是中间件本身宕机重启。当然，这个功能因中间件而异，部分中间件是放开给客户端控制的。

当然，中间件本身也有相应的数据容错策略。举个例子，Kafaka通过分区复制的策略保证数据不丢失。具体大致逻辑是由生产者（producer）首先找到领导者（leader）（这里的leader是Broker1上的Partition0）并把消息写到leader分区，然后leader分区会通过内部管道把消息复制到其它broker上的分区，这就是所谓的分区复制。

3、处理可靠性

这里的处理可靠性更多指的是应用层的消息路由逻辑。就是说，当一个事件（event）被一个节点（worker）处理完后，会按照路由策略表严格指定该事件（event）的下一个节点（worker）是谁。我认为它的可靠性是相对平时的代码接口调用或者过程式代码的这种风格而言的，这也正是它的可靠性所在。

- 路由规则：这套路由规则被抽象出来作为核心资产进行统一管理，因为它是定义整个业务流转规则，明确-简单-严格；
- 异常处理规则：某个节点处理出现异常，处理过程会被终止。保证经过人工介入才能重新触发继续往后处理；
- 监控规则：某个事件或业务流程要整体成功跑完所有按路由规则要求的所有节点，否则监控会进行告警并触发人工介入。




**EDA监控**

EDA的这种架构还有一个突出的特点，就是因为每个节点都是解耦的，所以哪个节点都不清楚进来的每一个event当前的状态是怎样的，究竟是已经处理完毕呢还是被丢到死信主题呢。这就好像流水线上的工人，个人只会完成自己的工序并再放回到流水线上。

当然，我们可以通过定义每个节点的worker的异常处理逻辑（即发生异常时指定错误码并顺带进行告警），但是这种方法有两个弊端：

- 业务流程处理跟告警处理耦合在一起；如果这种告警是通过API接口调用的话就更麻烦，因为如果告警系统有任何问题且大面积的event出现异常时候分分钟拖死你这个worker，继而耗尽线程资源导致系统假死；
- 缺乏系统的整体错误情况看板；

因此，需要定义单独的monitor对这种异常进行监控并告警。如上图，MonitorManager和worker的协同方式一般可以有以下几种方式。

- 由worker指定错误码后，并同时生成一个告警event及推倒告警主题；
- MonitorManager监听该告警主题，在发现有event后做响应告警处理；但因为MonitorManager一般只负责监控告警，且问题解决后MessageBroker后续还是得把它重新路由到之前的worker重试，所以一般使用fanout模式；

**事件驱动对企业的价值**

Gartner 报告指出，掌握“事件驱动的 IT”和以事件为中心的数字业务战略将成为大多数全球企业 CIO 的首要任务。企业严重依赖技术来构建可扩展、敏捷和高度可用的业务。事件驱动架构正在成为支持

现代企业实时运营、快速适应变化并做出明智业务决策的关键基石。

- 降低系统之间的依赖性

在传统的系统中，不同的组件需要在代码层面进行耦合，例如通过函数调用或共享变量等方式。这种紧密的耦合会导致组件之间高度依赖，当一个组件需要修改时，往往需要同时修改其它组件的代码，这样会导致系统的不稳定性和难以维护。相比之下，事件驱动架构中，组件之间通过事件进行通信，而不是直接调用代码或共享变量。当一个组件完成某个任务时，它将触发一个事件，然后将这个事件发送给系统中的其他组件。其他组件可以根据自己的需要选择是否要监听这个事件，如果监听则可以响应事件并执行相应的操作。这种机制使得系统中的组件可以相对独立地进行开发和维护，减少了代码之间的耦合度。

这种松耦合方式使得系统具有更高的可维护性和可扩展性。当需要修改某个组件时，仅需要对该组件进行修改，而不需要同时修改其他组件的代码。这不仅简化了开发过程，同时也使得系统更加健壮和灵活，因为不同的组件可以独立地进行部署和升级。同时，这种事件驱动架构的松耦合特性也使得系统更加容易进行扩展和集成，因为新的组件可以很容易地添加到系统中而不会影响到其他组件的运行。

- 提高可用性和可靠性

传统的系统通常采用紧耦合的方式，即各个组件之间紧密依赖，一个组件的故障很容易影响到其他组件的正常运行，从而导致系统的故障和不可用。在事件驱动架构中，当某个组件出现故障时，其他组件可以选择是否监听该事件，不受影响的组件可以继续执行自己的任务而不受影响。因此，这种松散的耦合方式可以提高系统的可用性和可靠性。

此外，事件驱动架构还可以通过异步事件处理的方式来提高系统的可用性和可靠性。异步事件处理的特点是事件的处理是非阻塞的，即组件不需要等待事件的处理结果就可以继续执行自己的任务。这种处理方式可以降低系统的延迟，并且能够处理大量的并发请求，从而提高系统的性能和可靠性。

事件驱动架构还可以通过实现事件溯源机制来提高系统的可靠性和容错性。事件溯源机制是指将事件的历史状态记录下来，并且可以回溯到任何一个事件的状态。这种机制可以帮助企业快速恢复系统，以及减少因故障而导致的数据丢失。通过记录和回放事件，企业可以更加容易地发现和解决系统的问题，从而提高系统的可靠性和容错性。

- 敏捷开发

事件驱动架构可以帮助开发人员更快地开发和部署应用程序。由于每个组件都是独立的，因此可以并行地开发和测试每个组件，从而缩短应用程序的开发时间。在部署方面，企业可以选择只部署需要的组件，而不是整个应用程序，这将进一步加快部署速度。

- 支持灵活的架构演进

企业可以通过向系统中添加新的事件、更改现有事件的结构或添加新的组件来扩展其功能。通过这种方式，企业可以实现较小的增量更改，同时仍保持整个系统的稳定性和可靠性。

另外，事件驱动架构还可以通过使用适当的事件类型来促进组件之间的松散耦合。例如，企业可以定义通用事件类型来处理跨多个组件的通信和协作。这种松散耦合使得企业可以更容易地更改系统的某些部分而不影响整个系统。

在灵活的架构演进方面，事件驱动架构还支持多种部署选项。企业可以选择在本地、云端或混合部署中使用事件驱动架构，以适应其业务需求和可用资源。

- 实时的数据处理与分析

在事件驱动架构中，事件是系统的核心，组件通过触发和处理事件来进行通信与协作。因此，事件驱动架构非常适合处理实时数据，并且可以轻松地将数据从一个组件传输到另一个组件。企业可以构建实时数据处理系统，从而更快地对业务数据做出决策和反应。例如，企业可以使用事件驱动架构来监控网站流量、分析用户行为、检测异常等。在这种情况下，组件可以通过触发和处理事件来快速响应这些实时数据，并执行必要的操作。

此外，事件驱动架构也可以用于构建复杂的数据流处理系统，例如处理大数据、实现实时分析等。这些系统需要处理大量的数据，并且需要快速而准确地处理数据。事件驱动架构可以通过使用分布式事件处理系统来实现这些目标，同时保持系统的可扩展性和可靠性。这些优势可以帮助企业更好地了解自己的业务，优化业务流程，并提高企业的竞争力。

**结论**

微服务架构范式是构建更具可维护性、可扩展性和健壮性的软件系统的更广泛难题的一部分。从问题分解的角度来看，微服务非常棒，但它们留下了很多棘手的问题；一个这样的问题是耦合。与你开始的地方相比，一个随意分解成几个微服务的单体实际上可能会让你处于更糟糕的状态。我们甚至有一个术语：“分布式单体”。为了帮助完成这个难题并解决耦合问题，事件驱动架构EDA被提出了。EDA 是一种通过使用生产者、消费者、事件和流的概念对交互进行建模来减少系统组件之间耦合的有效工具。一个事件代表一个感兴趣的动作，并且可能被甚至不知道彼此存在的组件异步发布和消费。EDA 允许组件独立运行和发展，是任何成功的微服务部署的基本要素。

**胡峻岩——软件质量属性**

**一、软件质量属性内容**

软件质量属性的定义和概念：

软件质量属性是指衡量软件产品质量特征的量化指标，也被称为质量特性或质量维度。它们是衡量软件产品是否满足用户需求和预期的标准。不同的软件质量属性可以相互影响和交叉影响，因此需要综合考虑多个软件质量属性来评估软件产品的总体质量。

常见的质量属性包括功能性、可用性、可修改性、性能、安全性、可测试性、易用性、可移植性、兼容性、可靠性、可维护性

**1.1 功能性**

定义：在指定条件下使用时，产品或系统提供满足明确和隐含要求的功能的程度。功能性只关注功能是否满足明确和隐含要求，而不是功能规格说明。其包括以下内容：

① 功能完备性：功能集对指定的任务和用户目标的覆盖程度

② 功能正确性：产品或系统提供具有所需精度的正确的结果的程度

③ 功能合适性：功能促使指定的任务和目标实现的程度

④ 功能性的依从性：产品或系统遵循与功能性相关的标准、约定或法规UI及类似规定的程度

### 1.2 可用性

可用性的一般场景：所关注的方面包括系统故障发生的频率、出现故障时会发生什么情况、允许系统有多长时间非正常运行、什么时候可以安全地出现故障、如何防止故障的发生以及发生故障时要求进行哪种通知。

1、可用性战术将会组织错误发展为故障，或者至少能够把错误的影响限制在一定范围内，从而使系统恢复成为可能。可用性是指系统正常运行时间的比例，是通过两次故障之间的时间长度或在系统崩溃情况下能够恢复正常运行的速度来衡量的。

2、维护可用性的方法：

（1）错误检测——用来检测故障的某种类型的健康监视

（2）自动恢复——检测到故障时某种类型的恢复

（3）错误预防——阻止错误演变为故障

3、可用性战术分类

错误检测、恢复监测和修复、重新引入、预防

### 1.3 可修改性

可修改性是指系统或软件能够快速地以较高地性价比对系统进行变更地能力。对于一个网站，我们要修改它某一板块地UI界面，当我们对界面进行修改时是否引起对另一个UI模块地影响，是否会引起后台控制，业务逻辑代码地变更，是否会引起整个网站地崩溃，这就体现了一个网站地整个架构是否具备可修改性。

引起可修改性地因素包括用户需求和系统内在需求。用户需求例如用户对软件某些图标的更改，而淘宝网后期对数据库体系架构的更新和完善则来源于系统内在的需求。

可修改性地战术分析：

1、局部化变更

局部化意味着实现“模块化”思想，也就是设计模式中的“单一职责原则”的设计原则。通俗的来讲就是一个模块只完成一个部分，使每一个模块责任单一，防止职责过多引起整体变更时的繁琐，复杂，主要表现在类、函数、方法和接口的时候，实现“高内聚，低耦合”。

2、防止连锁反应

所谓连锁反应就是我们平时编程，无论是写函数还是写类，都需要被其他类还是函数调用，修改此函数或类就会影响到调用他的函数，这就是连锁反应。

防止连锁反应：

- 信息隐藏：信息隐藏就是把某个实体地责任分解为更小地部分并选择哪些信息成为共有，哪些信息成为私有的。

- 维持现有的接口：该战术的模式包括添加接口、添加适配器、提供一个占位程序。
- 限制通信路径：限制与一个给定的模块共享的模块，减少联系，一旦变更影响会小很多。
- 使用仲裁者：插入仲裁者来管理依赖之间的关系，就比如数据库的使用，通过数据库来管理不同的数据信息。

3、推迟绑定时间

将有可能的修改，尽量用配置文件，或者其他后期让非开发人员可调整的方式实现。

### 1.4 性能

性能反映的是系统的响应能力，表现在三个方面，速度、吞吐量和持续高速性。性能战术的目标是对一定的时间限制内到达系统的事件生成一个响应，这些事件可以是消息到达、定时器到时，系统状态的变化。

影响响应时间的因素，包括资源消耗和闭锁时间。资源包括CPU、数据存储、网络通信带宽和内存等；资源消耗是指实际需要耗费的时间；比如：数据的增删改查，CPU进行大量的加减运算等等。由于资源争用、资源不可用或长时间计算导致事件无法处理，这是指计算机可能等待的时间。最常见的就是多个进程同时操作一个数据，写操作正操作此数据，读操作只能等待，必须等写操作解锁，读操作才能进行，这就会产生等待时间。

性能战术的三大分类：

（1）资源需求——分析影响性能的资源因素。提高计算效率，减少计算开销，管理事件率，控制采样频率。

（2）资源管理——提高资源的应用效率。引入并发维持多个副本，增加可用资源。

（3）资源仲裁——解决资源的争用。调度策略，先进/先出，固定优先级，动态优先级，静态调度。

### 1.5 安全性

安全性是指在确保用户正常使用系统的情况下，软件抵御攻击的能力。

其包括如下子特性：

① 保密性：产品或系统确保数据只有在被授权时才能被访问的程度

② 完整性：系统、产品或组件防止未授权访问、篡改计算机程序或数据的程度

③ 抗抵赖性：活动或事件发生后可以被证实且不可被否认的程度

④ 可核查性：实体的活动可以被唯一地追溯到该实体的程度

⑤ 真实性：对象或资源的身份表示能够被证实符合其声明的程度

⑥ 信息安全性的依从性：产品或系统遵循与信息安全性相关的标准、约定或法规以及类似规定的程度

提高安全性的方法主要分为三大类：抵抗攻击，检测攻击，从攻击中恢复

1、抵抗攻击：防止攻击对系统和数据造成影响乃至破坏

（1）用户的证实：通过账号密码，指纹等识别手段，确定现在正在访问系统的人是真正的用户

（2）用户的授权：管理用户权限，确认用户的操作在自己的权限内

（3）维持数据的保密性：数据传输时进行加密

（4）维持数据的完整性：利用MD5码等校验文件是否未修改。MD5码是和文件内容相关的字符串，文件的任何一点改动理论上都会导致MD5码变化

（5）减少暴露：关闭不安全的或没必要的端口，自启动服务和无线路由SSID等

（6）访问控制：通过白名单，限制可以访问的用户或其他主机

2、检测攻击：尽早发现正在进行地攻击，确保能够及时作出回应如关闭主机等。系统监测一般是软件和人工结合，既有自动检测系统检测异常，也有安全专家人工复核或检查

3、从攻击中恢复：在被攻击并产生了影响后，尽快地从异常情况中恢复

（1）恢复状态：采用提高可用性的一些手段，如备份等，提高恢复速度

（2）攻击者的识别：找出并确定攻击者，震慑潜在的攻击者

**1.6 可测试性**

定义：软件可测试性是指软件系统的一种质量属性，表示软件系统的代码、设计、结构和功能是否易于进行测试。软件可测试性对于软件开发过程中的测试、调试和维护都具有重要的影响。

软件可测试性的具体内容和特性包括：

① 可观察性：软件系统应该具备能够观察和监测系统内部状态和行为的特性，包括日志记录、调试信息输出等。可观察性能够帮助开发人员识别问题并进行调试和测试。

②可控性：软件系统应该具备能够控制和操作系统内部状态和行为的特性，包括通过接口或命令行控制系统行为和状态。可控性能够帮助开发人员构建测试用例和模拟各种场景和条件进行测试。

③可测试性的设计：软件系统应该具备能够支持测试的设计特性，包括模块化设计、接口设计、参数化设计、高内聚低耦合设计等。可测试性的设计能够提高软件的可测试性，降低测试成本和测试时间。

④可变性：软件系统应该具备能够变化和配置的特性，包括可配置参数、可替换组件、可扩展性等。可变性能够帮助开发人员快速适应各种测试和调试环境和条件。

⑤可复用性：软件系统应该具备能够复用和重用的特性，包括可重用模块、可复用代码等。可复用性能够帮助开发人员减少测试和调试的工作量，提高测试效率。

⑥可维护性：软件系统应该具备易于维护的特性，包括易于理解的代码结构、清晰的注释、良好的文档等。可维护性能够帮助开发人员在测试和调试过程中快速定位问题并进行修复。

**1.7 易用性**

易用性涉及用户完成任务的容易程度以及所提供的用户支持类型。易用性可以划分为几个模块：学习系统功能、有效使用系统、最小化错误影响、系统适应用户需求和提高用户信息和满意度。

易用性既可从它的子特性角度当前产品质量特性来进行指定和测量，也可以直接通过测度来进行指定和测量 ，其包括的具体内容如下：

① 可辨别性：用户能够辨识产品或系统是否适合他们的要求的程度

1、可辨识性将取决于通过对产品或系统的初步印象和/或任何相关文档来辨识产品或系统功能的能力

2、产品或系统提供的信息可包括演示、教程、文档或网站的主页信息

② 易学性：在指定的使用周境中，产品或系统在有效性、效率、抗风险和满意度特性方面为了学习使用该产品或系统这一指定的目标可为指定用户使用的程度

③ 易操作性：产品或系统具有易于操作和控制的属性的程度

④ 用户差错防御性：系统防御用户犯错的程度

⑤ 用户界面舒适性：用户界面提供令人愉悦和满意的交互的程度

⑥ 易访问性：在指定的使用周境中，为了达到指定的目标，产品或系统被具有最广泛的特性和能力的个体所使用的程度，能力的范围包括与年龄有关的能力障碍

⑦ 易用性的依从性：产品或系统遵循与易用性相关的标准、约定或法规以及类型规定的程度

**1.8 可移植性**

定义如下：软件可移植性是指软件系统可以在不同的硬件和操作系统平台上运行和使用的特性，是产品或系统能够有效地、有效率地适应不同的或演变的硬件、软件或者其他运行（或使用）环境的程度，实用性包括内部能力：如屏幕域、表、事务量、报告格式等，的伸缩性

可移植性包括如下的子特性：

① 适应性：产品或系统能够有效地、有效率地适应不同的或演变的硬件、软件或者其他运行（或使用）环境的程度，实用性包括内部能力：如屏幕域、表、事务量、报告格式等，的伸缩性

② 易安装性：在指定环境中，产品或系统能够成功地安装和/或鞋子的有效性和效率的程度（若最终产品会被最终用户安装，那么易安装性会影响功能合适性和易操作性）

③ 易替换性：在相同的环境中，产品能够替换另一个相同用途的指定软件产品的程度

1、软件产品的新版本的易替换性在升级时对用户来说是重要的

2、易替换性可保留易安装性和适应性的属性

3、易替换性将降低锁定风险：因此其他软件产品可以替代当前产品，例如按标准文档格式使用

④ 可移植性的依从性：产品或系统遵循与可移植性相关的标准、约定或法规以及类似规定的程度

**1.9 兼容性**

定义：在共享相同的硬件或软件环境的条件下，产品、系统或组件能够与其他产品、系统或组件交换信息，和执行其所需的功能的程度，具体包括如下内容：

① 共存性：在与其他产品共享通用的环境和资源的条件下，产品能够有效执行其所需的功能并且不会对其他产品造成负面影响的程度

② 互操作性：两个或多个系统、产品或组件能够交换信息并使用已交换的信息的程度

③ 兼容性的依从性：产品或系统遵循与兼容性相关的标准、约定或法规以及类型规定的程度

### 1.10 可靠性

定义：系统、产品或组件在指定条件下、指定时间内执行指定功能的程度，可靠性的种种局限是有需求、设计和实现中的故障或周境的变化所致。

其子特性如下：

① 成熟性：系统、产品或组件在正常运行时满足可靠性要求的程度

② 可用性：系统、产品或组件在需要使用时能够进行操作和访问的程度，可通过系统、产品或组件在总时间中处于可用状态的百分比进行外部评估。可用性是成熟性、容错性和易恢复性的组合

③ 容错性：尽管存在硬件或软件故障，系统、产品或组件的运行符合预期的程度

④ 易恢复性：在发生中断或失效时，产品或系统能够恢复直接受影响的数据并重建期望的系统状态的程度，在失效发生后，计算机系统有时会宕机一段时间，这段时间的长短由其易恢复性决定

⑤ 可靠性的依从性：产品或系统遵循与可靠性相关的标准、约定或法规以及类似规定的程度

### 1.11 可维护性

定义：产品或系统能够被预期的维护人员修改的有效性和效率的程度。修改包括纠正、改进或软件对环境、需求和功能规格说明变化的适应，维护性包括安装更新和安装升级，维护性可以被解释为便于维护活动的一种产品或系统固有能力，或者为了产品或系统维护的目标维护人员所经历的使用质量

具体的子特性如下：

① 模块化：由多个独立组件组成的系统或计算机程序，其中一个组件的变更对其他组件的影响最小的程度

② 可重用性：资产能够被用于多个系统，或其他资产建设的程度

③ 易分析性：可以评估预期变更对产品或系统的影响。诊断产品的缺陷或失效原因、识别待修改部分的有效性和效率的程度

④ 易修改性：产品或系统可以被有效地、有效率地修改，且不会引入缺陷或降低现有产品质量的程度

1、实现包括编码、设计、文档和验证的变更

2、模块化和易分析性会影响易修改性

3、易修改性是易改变性和稳定性的组合

⑤ 易测试性：能够为系统、产品或组件建立测试准则，并通过测试执行来确定测试准则是否被满足的有效性和效率的程度

⑥ 维护性的依从性：产品或系统遵循与维护性相关的标准、约定或法规以及类似规定的程度

## 二、提高软件质量的方法

要想提高软件质量，可以从软件质量的每个性能着手。

### 2.1 提高软件的功能性

- 确定清晰的需求规格：对于软件开发过程中的需求分析和规格化非常关键，要确保需求清晰、详尽，并且具备可验证性。
- 采用适当的开发方法和技术：软件开发过程中的方法和技术选择对于软件的功能性至关重要，需要根据实际情况选择最适合的方法和技术。
- 使用测试驱动的开发方法：测试驱动的开发方法可以帮助开发人员在开发过程中不断验证软件的功能性，确保软件系统满足用户需求。
- 做好版本控制：版本控制可以帮助开发人员追踪软件系统的变更历史，确保每个版本的功能都符合用户的期望。

**2.2 提高软件的可靠性**

- 设计健壮的错误处理机制：软件系统需要处理各种异常和错误情况，要设计健壮的错误处理机制，确保软件系统在发生错误时能够进行恰当的处理。
- 进行适当的测试：测试是提高软件可靠性的关键步骤，要进行全面、有效的测试，包括单元测试、集成测试、系统测试、性能测试等。
- 采用自动化测试工具：自动化测试工具可以帮助开发人员快速、准确地执行测试用例，提高测试效率和准确性。
- 避免过度设计：过度设计会增加软件系统的复杂度和错误风险，要避免过度设计，采用简洁、清晰的设计方案。

**2.3 提高软件的可修改性**

- 采用模块化的设计：模块化的设计可以帮助开发人员将软件系统划分为独立的模块，降低模块之间的耦合度，提高系统的可修改性。
- 遵循良好的编码规范：良好的编码规范可以提高代码的可读性和可维护性，减少后续修改和维护的难度。
- 提供清晰的文档和注释：清晰的文档和注释可以帮助开发人员快速了解软件系统的结构和实现细节，便于后续修改和维护。
- 采用自动化构建工具：自动化构建工具可以帮助开发人员自动构建和打包软件系统，提高软件交付的质量和效率。

**2.4 提高软件的性能**

- 优化算法和数据结构：优化算法和数据结构可以提高软件系统的效率和性能，减少资源的浪费。
- 使用高效的编程语言和工具：选择高效的编程语言和工具可以提高软件系统的运行效率和性能。
- 进行性能测试和分析：进行全面、准确的性能测试和分析可以帮助开发人员了解软件系统的性能瓶颈和问题，制定相应的优化方案。
- 采用分布式架构和负载均衡技术：分布式架构和负载均衡技术可以提高软件系统的可扩展性和容错能力，从而提高性能。

**2.5 提高软件的安全性**

- 采用安全的编码规范：采用安全的编码规范可以减少代码漏洞和安全隐患，提高软件系统的安全性。
- 进行安全测试和评估：进行全面、有效的安全测试和评估可以发现软件系统中存在的安全漏洞和风险，制定相应的安全方案。
- 采用多层次的安全防护措施：采用多层次的安全防护措施可以提高软件系统的安全性，包括网络安全、数据加密、身份认证等。
- 进行定期的安全更新和维护：定期的安全更新和维护可以及时修复软件系统中存在的安全漏洞和风险，保护系统和数据的安全。

**2.6 提高软件可测试性**

- 使用单元测试、集成测试和系统测试等测试方法，确保软件质量。
- 设计易于测试的代码和界面，避免代码和界面的复杂性和耦合性。
- 使用自动化测试工具和框架，例如JUnit和Selenium，可以提高测试效率和测试覆盖率。
- 遵循良好的软件工程实践，例如模块化、低耦合和高内聚等，可以提高软件的可测试性。
- 培养团队的测试意识和测试技能，例如测试计划的制定、测试用例的编写和缺陷报告的撰写等。

**2.7 提高软件易用性**

- 进行用户研究和用户体验设计，了解用户需求和使用场景。
- 确定用户需求和使用场景，设计符合用户认知和心理特点的界面和交互方式。
- 提供良好的反馈机制和帮助文档，让用户在使用过程中可以得到及时的帮助和支持。
- 进行用户测试和反馈收集，不断优化和改进用户体验。

**2.8 提高软件可移植性**

- 使用标准化的编程语言、API和数据格式，避免使用与操作系统或硬件相关的代码和功能。
- 避免使用平台特定的API和库，尽可能地使用跨平台的工具和框架。
- 做好跨平台的适配和测试工作，确保软件在不同平台上运行和使用的稳定性和兼容性。

**2.9 提高软件兼容性**

- 了解兼容性要求：在设计软件时，要充分了解目标用户的设备、操作系统和其他软件环境，以确定软件需要支持哪些版本和配置，以及兼容性要求。
- 使用标准技术和协议：使用标准技术和协议可以提高软件的兼容性。例如，使用标准的网络协议和数据格式可以确保软件在不同的网络环境下能够正常工作。
- 进行充分的测试：在开发软件时，需要进行充分的测试来确保软件在各种环境下的兼容性。测试可以包括功能测试、性能测试、安全测试和兼容性测试等多个方面。
- 提供兼容性选项：如果软件不能在某些环境下正常工作，可以提供兼容性选项，允许用户在软件中进行一些设置来提高兼容性。

- 更新软件：随着硬件、操作系统和其他软件环境的更新，软件也需要更新以提高兼容性。定期更新软件，确保其能够兼容最新的硬件和软件环境。

**2.10 提高软件的可靠性**

- 设计时考虑可靠性：在软件设计的早期阶段就要考虑可靠性，例如使用稳定、可靠的设计模式和架构，以及避免设计上的缺陷。
- 使用自动化测试：使用自动化测试可以在软件开发的早期阶段就发现并修复软件中的缺陷和错误，提高软件的可靠性和质量。
- 定期进行代码审查：定期进行代码审查可以帮助发现代码中的潜在缺陷和错误，并及时进行修复，提高软件的可靠性。
- 保持简单和清晰：尽可能地保持软件的简单和清晰，避免过度设计和复杂性，这可以提高软件的可靠性和易于维护性。

**2.11 提高软件的可维护性**

- 优秀的代码结构和设计：好的软件设计能够降低维护成本。使用模块化的设计，使得每个模块的功能单一、内部关系紧密，尽量避免代码之间的相互依赖。合理的代码结构和命名规范也有助于提高代码的可维护性。
- 版本控制：使用版本控制工具来管理软件代码。这可以让您跟踪软件的变化并在需要时回退到以前的版本。还可以将多个开发人员的工作集成到一个代码库中。
- 文档：编写清晰、详细的文档可以帮助其他开发人员更好地理解代码，从而降低维护成本。文档应该包括代码的结构、设计、算法、数据结构、错误处理等方面的详细信息。
- 单元测试：编写单元测试可以发现代码中的错误并防止它们在代码库中累积。在修改代码时运行单元测试可以快速发现和解决问题。
- 代码审查：进行代码审查可以发现代码中的潜在问题和错误。这可以在代码进入代码库之前及时发现和纠正问题。
- 合理的注释：适当的注释可以使代码更容易理解。注释应该解释代码的设计原理、算法、数据结构等方面的信息，同时还应该提供更深入的背景信息。

**三、软件质量属性的评估方法**

**SAAM评估方法**

SAAM是一种通过场景来评估软件架构的方法，它的主要步骤包括场景定义、场景分析、架构设计和评估。在SAAM中，场景是指一个具体的使用情况或者用户故事，场景分析用来识别场景中涉及的元素（比如模块、接口等）以及它们之间的关系，架构设计是根据场景分析的结果对架构进行调整和设计，评估则是针对设计后的架构进行评估和反馈。

1、评估目的

SAAM的目的是验证基本的体系结构假设和原则，评估体系结构固有的风险。SAAM指导对体系结构的检查，使其主要关注潜在的问题点，如需求冲突。SAAM不仅能够评估体系结构对于特定需求的使用能力，也能被用来比较不同的体系结构。

2、主要输入

问题描述、需求说明、架构描述文档

3、评估参与者

风险承担者、记录人员、软件体系结构设计师

4、评估活动或过程

SAAM分析评估体系结构的过程包括六个步骤：场景开发、架构描述、场景的分类和优先级确定、单个场景评估、场景相互作用的评估、总体评估。

5、评估结果

SAAM评估的主要有形输出包括：

- 把代表了未来可能做的更改的场景与架构对应起来，显现出架构中未来可能会出现的比较高复杂性的地方，并对每个这样的更改的预期工作量做出评估。
- 理解系统的功能，对多个架构所支持的功能和数量进行比较。

如果所评估的是一个框架，SAAM评估将指明框架中未能满足其修改性需求的地方，有时还会指出一种效果更好的设计。SAAM评估也能对两个或者三个备选架构进行比较，明确其中哪一个能够较好的满足质量属性需求，而且做得更改更少、不会再未来导致太多复杂的问题。

**ATAM评估方法**

ATAM是一种通过场景和质量属性来评估软件架构的方法，它的主要步骤包括定义架构、场景选择、属性评估、风险识别和迭代。在ATAM中，定义架构是通过讨论和文档来获取架构的高层次设计，场景选择是选择能够覆盖架构关键决策的场景，属性评估则是通过对场景进行分析来评估各个质量属性的影响，风险识别是对架构的不确定性进行评估并识别可能的风险，迭代则是在评估的基础上进行架构设计的优化和迭代。

1、评估目的

ATAM的评估目的是依据系统质量属性和商业需求评估设计决策的结果。ATAM希望揭示出架构满足特定质量目标的情况，使我们更清楚的认识到质量目标之间的联系，即如何权衡多个质量目标。

2、评估参与者

评估小组，该小组是所评估架构项目外部的小组，通常由3-5人组成。该小组的每个成员都要扮演大量的特定角色。他们可能是开发组织内部的，也可能是外部的。

项目决策者，对开发项目具有发言权，并有权要求进行某些改变，包括项目管理人员、重要的客户代表、架构设计师等。

架构涉众，包括关键模块开发人员、测试人员、用户等。

3、评估活动或过程

整个ATAM评估过程包括九个步骤：描述ATAM方法、描述商业动机、描述体系结构、确定体结构方法、生成质量属性效用树、分析体系结构方法、讨论和分级场景、描述评估结果，其中分析体系结构方法需要执行两次。

4、评估结果

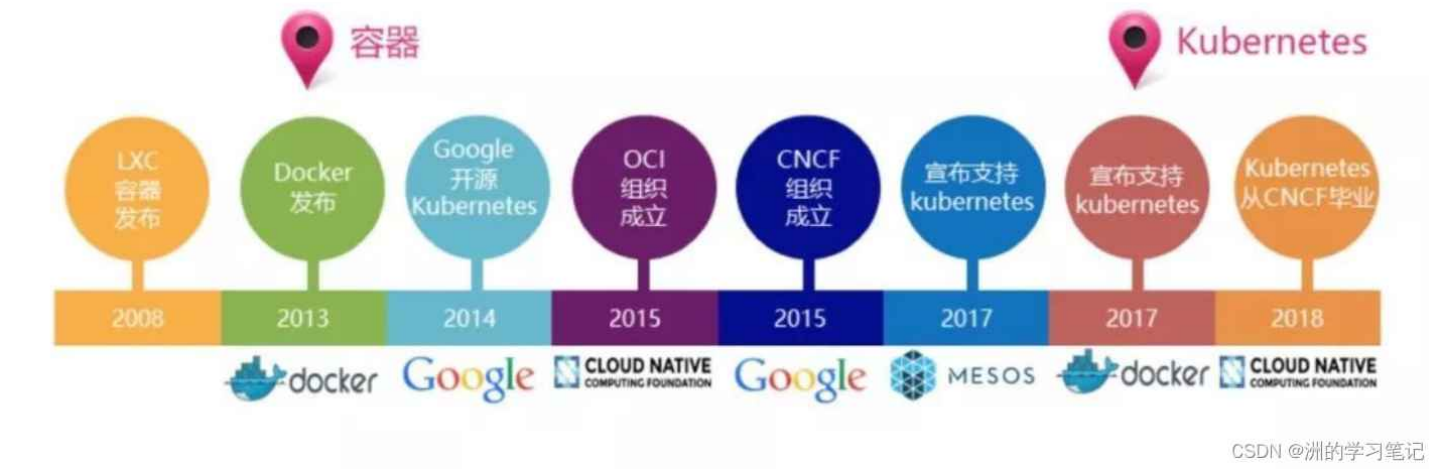
一个简洁的构架表述；表述清楚的业务目标；用场景集合捕获的质量需求；构架决策到质量需求的映射；所确定的敏感点和权衡点集合；有风险决策和无风险决策；风险主题的集合。

**于一帆——云原生架构**

**一、介绍**

云原生架构是一种用于构建和运行云原生应用程序的软件架构，它通过将应用程序设计为微服务，并使用容器化技术和自动化部署来实现高度可扩展性、弹性和可靠性。云原生架构的设计目标是让应用程序能够快速、可靠地部署和运行在云环境中，从而能够更好地应对云计算环境的复杂性和变化性。

2014年谷歌发布了一篇论文，其中介绍了他们内部的容器化编排平台——Borg，这就是云原生架构的雏形。在随后的几年里，容器化和微服务架构的出现和发展进一步加速了云原生架构的发展，使得它逐渐成为了一种主流的软件架构。云原生架构是一种创新的软件开发方法，专为充分利用云计算模型而设计。它使组织能够使用微服务架构将应用程序构建为松散耦合的服务，并在动态编排的平台上运行它们。因此，基于云原生应用程序架构构建的应用程序是可靠的，可提供规模和性能，并缩短上市时间。





云原生架构的核心理念是将应用程序设计为微服务，这些微服务是松耦合的、独立可部署的、可组合的组件。每个微服务都应该只关注自己的业务逻辑，而不涉及其他微服务的业务逻辑。这样，当需要修改一个微服务时，只需要修改该微服务，而不需要修改整个应用程序，这样可以提高应用程序的可维护性、可伸缩性和可重用性。

容器化技术是实现云原生架构的关键技术之一。它可以将应用程序打包成容器镜像，并将这些镜像部署到容器编排平台上。容器化技术可以提供应用程序的隔离性、可移植性和可重复性，从而使得应用程序可以更加高效地部署和管理。

自动化部署是另一个重要的技术，它可以通过自动化测试、构建、部署和监控，实现快速和可靠的应用程序交付。自动化部署可以帮助开发团队更快地响应用户需求，减少发布错误，并提高应用程

序的可靠性和稳定性。

服务网格是云原生架构的另一个重要组成部分，它可以帮助解决微服务架构中服务之间的通信问题。服务网格是一个由一系列专用网络代理组成的基础设施层，它可以自动化地处理服务之间的通信、负载均衡、服务发现、故障恢复和安全等问题。服务网格可以提高应用程序的可观察性和可操作性，从而更容易进行故障排除和性能优化。

云原生和微服务不同，微服务是应用程序的软件架构，可以是单体式和微服务式。 微服务是基于分布式计算的。 应用程序即使不采用微服务架构也可以是云原生的，例如分布式的，但效果没有微服务好。 如果是单体式的，云原生就基本发挥不出什么优势。 另外微服务的程序也可以不是云原生的。它们虽然是两个不同的东西，但云原生和微服务是天生良配，相得益彰，相辅相成。 而且很多云原生的工具都是针对微服务架构设计的。

云原生架构的基石包括基础设施即代码、不可变基础设施和声明式API。

基础设施即代码将基础设施的定义和配置保存在代码库中，然后通过代码管理工具自动化部署和管理基础设施。这种方法可以确保基础设施的一致性和可重复性，并且可以实现快速部署和回滚。基础设施即代码帮助开发人员快速、可靠地部署应用程序所需要的基础设施，同时也帮助管理基础设施的状态和版本。

不可变基础设施：基础设施的状态在创建后就不能被改变，任何的修改都需要重新创建新的基础设施。这种方法可以确保基础设施的可靠性和一致性，同时也可以降低维护成本和风险。不可变基础设施确保应用程序的环境在任何时候都是可重复的，从而保证应用程序的稳定性和可靠性。

声明式API以声明的方式描述应用程序的状态和行为，而不是以指令的方式告诉计算机如何去实现。这种方法可以让开发人员更加专注于应用程序的业务逻辑，而不是关注底层实现细节。声明式API让开发人员更加方便地描述应用程序所需要的资源和服务，并且可以实现自动化的部署和管理。同时，声明式API也可以降低应用程序的复杂度，从而提高应用程序的可维护性和可扩展性。

**二、云原生架构原则**

云原生架构的原则包括服务化原则、弹性原则、可观测原则、韧性原则、所有过程自动化原则、零信任原则、架构持续演进原则。

服务化原则强调将应用程序拆分为较小、自治的服务单元，以便每个单元都可以独立开发、部署和维护。这种服务化的方式使得开发人员可以更快地交付新功能、更容易地扩展应用程序，并且可以降低整个系统的复杂性。服务化的方式还可以支持更好的分布式计算，这在云环境下尤其重要。

弹性原则指应用程序应该设计成可以快速适应不同的负载，从而实现高可靠性和可伸缩性。这种弹性可以通过自动扩展、自动缩放和自动恢复来实现。在云环境下，自动化弹性可以帮助应用程序在面对不断变化的负载时保持高可用性。

可观测原则指开发人员可以在不影响应用程序性能的情况下，获得应用程序的完整、准确的状态信息。这种可观测性可以通过日志、指标、跟踪和分布式跟踪等手段来实现。可观测性可以帮助开发人员快速诊断和解决问题，提高应用程序的可靠性和可维护性。 今天大部分企业的软件规模都在不断增长，原来单机可以对应用做完所有调试，但在分布式环境下需要对多个主机上的信息做关联，才可能回答清楚服务为什么宕机、哪些服务违反了其定义的SLO、目前的故障影响哪些用户、最近这次变更

对哪些服务指标带来了影响等等，这些都要求系统具备更强的可观测能力。可观测性与监控、业务探索、APM等系统提供的能力不同，前者是在云这样的分布式系统中，主动通过日志、链路跟踪和度量等手段，让一次APP点击背后的多次服务调用的耗时、返回值和参数都清晰可见，甚至可以下钻到每次三方软件调用、SQL 请求、节点拓扑、网络响应等，这样的能力可以使运维、开发和业务人员实时掌握软件运行情况，并结合多个维度的数据指标，获得前所未有的关联分析能力，不断对业务健康度和用户体验进行数字化衡量和持续优化。

韧性原则指应用程序应该能够在面对各种故障和错误时继续正常运行。这种韧性可以通过容错、重试、降级和优雅停机等机制来实现。韧性非常重要，因为云环境不可靠，可能出现各种故障和错误。当业务上线后，最不能接受的就是业务不可用，让用户无法正常使用软件，影响体验和收入。韧性代表了当软件所依赖的软硬件组件出现各种异常时，软件表现出来的抵御能力，这些异常通常包括硬件故障、硬件资源瓶颈、业务流量超出软件设计能力、影响机房工作的故障和灾难、软件 bug、黑客攻击等对业务不可用带来致命影响的因素。

所有过程自动化原则指在云原生架构中，所有的过程都应该自动化。这包括自动化部署、自动化测试、自动化构建、自动化监控等。自动化可以提高开发效率，缩短开发周期，并且可以减少人为错误。

零信任原则是一种安全模型，认为所有的请求都应该受到验证和授权，即使它们来自内部网络。这种零信任模型可以保护应用程序免受内部和外部的威胁。在云原生架构中，零信任原则可以通过身份验证、访问控制、加密等方式来实现。

架构持续演进原则：云原生架构是一种持续演进的架构，它需要不断适应变化的业务需求和技术发展。这个原则强调了在云原生架构中，架构设计应该是灵活、可扩展和可维护的。

**三、云原生架构相比传统架构的优势**

1、弹性和可扩展性

云原生架构通过容器编排和基础设施自动化实现了更好的弹性和可扩展性。容器是轻量级的、可移植的应用程序运行时环境，可以快速启动、停止和调整规模。容器编排平台可以自动管理容器的生命周期，根据应用程序的负载自动调整容器的数量和资源使用率。这样，应用程序可以更好地应对突发的负载压力，同时也能够充分利用基础设施资源，提高资源利用率。

2、更快的交付速度

云原生架构通过基础设施即代码和自动化流程实现了更快的交付速度。开发人员可以通过声明式API快速部署应用程序，而无需手动配置基础设施。基础设施即代码将基础设施的配置和管理变成了代码，开发人员可以像管理代码一样管理基础设施。自动化流程可以自动化测试、构建、部署和监控等流程，从而减少人工干预，提高交付速度和质量。

3、更高的可靠性和韧性

云原生架构通过服务化、自我修复和故障隔离等方式实现了更高的可靠性和韧性。服务化将应用程序拆分成小的、可重用的服务，每个服务都有自己的生命周期管理和故障恢复机制。这样，应用程序可以更容易地适应变化，也可以更容易地进行测试和维护。自我修复可以在容器出现故障时自动重启容器或迁移容器到其他可用的节点，从而保证应用程序的高可用性。故障隔离可以避免单点故障影

响整个系统，每个服务都有自己的容器和资源，即使一个服务出现问题，也不会影响整个系统的稳定性。

4、更低的成本

云原生架构可以通过容器共享基础设施、基础设施自动化和自我修复等方式降低成本。容器可以在同一台主机上运行多个应用程序实例，共享主机资源，提高资源利用率。基础设施自动化可以减少人工干预，降低人力成本。自我修复可以在出现故障时自动重启容器或迁移容器到其他可用的节点，降低人工干预和停机时间，从而降低维护成本。

5、更好的安全性

云原生架构实现了“零信任”原则，这意味着只有经过身份验证和授权的用户才能访问应用程序和数据。此外，云原生架构还可以帮助您更好地保护应用程序和数据，例如使用容器隔离技术和自动化安全检查。

6、让开发人员以最好的状态工作

云原生架构的自动化和可编程性可以让开发人员更快地构建、测试和部署应用程序。开发人员不再需要手动执行复杂的操作，例如手动构建和部署代码，而可以使用自动化工具和基于代码的流程快速迭代应用程序。这使开发人员能够专注于创新和业务需求，而不是处理常规任务。

7、协调运营和业务

云原生架构使运营人员和开发人员之间的沟通更加紧密，从而实现更好的协作。开发人员可以在开发过程中考虑应用程序在生产环境中的表现，而运营人员可以更好地理解应用程序的设计和实现。此外，云原生架构的自动化和可编程性还可以减少部署和维护过程中的冲突和错误，从而提高运营效率。

8、以应用为中心

云原生架构是以应用为中心的，这意味着整个架构都是为应用程序而设计的。开发人员可以使用容器化技术轻松地打包应用程序和依赖项，并使用声明式API和基础设施即代码来定义应用程序的配置和要求。运营人员可以使用自动化工具和监控来管理应用程序的性能和可靠性，从而使应用程序成为云原生架构的核心组件。

9、更快的迭代和创新

使用云原生架构，开发人员可以更快地迭代和创新，因为他们可以使用自动化工具和基于代码的流程来快速测试、部署和更新应用程序。此外，云原生架构还提供了更多的灵活性和可扩展性，使开发人员能够更快地响应变化的市场需求。

**四、云原生应用和传统企业应用的区别**

| 云原生应用 | 传统企业应用 |
| --- | --- |
| **可预测**。 云原生应用相对于传统企业应用更具有可预测性，这是由于云原生应用采用了更为自动化和标准化的技术和工具，例如自动化部署、自动扩展、自动恢复等。 | **不可预测。**传统企业应用通常采用的是单块式架构逻辑都集中在一个单独的代码库中。这种架构下的很难预测，因为应用程序很可能由多个模块组成， |

| | |
|---|---|
| | 间的依赖关系非常复杂。这就导致了修改一个模块<br>模块，从而使得应用程序难以预测和维护。 |
| **操作系统抽象化。** 云原生应用架构要求开发人员使用平台作为一种方法，从底层基础架构依赖关系中抽象出来，从而实现应用的简单迁移和扩展。实现云原生应用架构最有效的抽象方法是提供一个形式化的平台。 | **依赖操作系统。** 传统的应用架构允许开发人员在应<br>系统、硬件、存储和支持服务之间建立紧密的依赖<br>赖关系使应用在新基础架构间的迁移和扩展变得复<br>险，与云模型相背而驰。 |
| **快速恢复。**云原生具有弹性和高可用性的特性，这些特性使得云原生应用在面临故障或部分组件失效时，能够快速适应并恢复。云原生应用采用分布式架构和容器化技术，可以将应用程序分割成多个微服务，并在多个容器中运行，这些容器可以在不同的节点上进行部署和管理，从而实现高可用性和弹性。同时，云原生应用还可以使用自动化工具来进行监控、管理和修复，这样就能够快速检测到故障并进行快速恢复。 | **恢复缓慢。**主要是由于传统企业架构架构单一、不<br>可用性的特性所致。传统企业应用通常采用单体架<br>构，整个应用程序部署在一个单一的节点上，一旦<br>障，整个应用程序就会崩溃。即使采用了一些高可<br>如主从复制、负载均衡等，也很难保证在故障发生<br>复，往往需要手动介入或调试。此外，传统企业应<br>护工作也很繁琐，需要人工管理和维护，缺乏自动<br>持，导致恢复速度较慢。 |
| **协作。** 云原生可协助 DevOps，从而在开发和运营职能部门之间建立密切协作，将完成的应用代码快速顺畅地转入生产。 | **孤立。** 传统 IT 将完成的应用代码从开发人员“隔<br>营，然后由运营人员在生产中运行此代码。企业的<br>重以至于无暇顾及客户，导致内部冲突产生，交付<br>工士气低落。 |
| **持续交付。** IT 团队可以在单个软件更新准备就绪后立即将其发布出去。快速发布软件的企业可获得更紧密的反馈循环，并能更有效地响应客户需求。持续交付最适用于其他相关方法，包括测试驱动型开发和持续集成。 | **瀑布式开发。** IT 团队定期发布软件，通常间隔几<br>实上，当代码构建至发布版本时，该版本的许多组<br>就绪，并且除了人工发布工具之外没有依赖关系。<br>的功能被延迟发布，那企业将会错失赢得客户和增<br>会。 |

## 五、主要架构模式

云原生架构的主要架构模式有服务化架构模式、Mesh架构模式、Serverless模式、分布式事务模式、可观测架构。

1、服务化架构模式

服务化架构模式是云原生架构的核心架构模式之一，它将应用程序分解成一组小而相互独立的服务，每个服务都可以被独立地开发、部署、扩展和管理。服务化架构模式的主要优点包括：

灵活性。服务可以被独立地开发和部署，这使得应用程序更加灵活和可扩展，能够更好地应对变化和不断增长的需求。

可维护性。服务化架构模式可以使应用程序更容易维护。由于每个服务都是相对独立的，开发人员可以更容易地修改和更新它们，而不必担心对整个系统的影响。

可伸缩性。服务化架构模式允许开发人员针对每个服务进行独立的扩展，以满足变化的负载和需求。

2、Mesh架构模式

Mesh架构模式是一种面向微服务的架构模式，它强调网络拓扑、负载均衡、服务发现、安全性等方面。Mesh架构模式主要有以下优点：

负载均衡。Mesh架构模式支持负载均衡，并提供了多种路由算法，可以根据请求的属性、服务的负载和位置等来选择最优的服务实例。

服务发现。Mesh架构模式提供了服务发现机制，可以自动发现和注册服务，提供服务的可用性和稳定性。

安全性。Mesh架构模式提供了丰富的安全机制，包括数据加密、身份验证和访问控制等。

3、Serverless模式

Serverless模式是一种极端的云原生架构模式，它将应用程序从底层的基础设施中解耦出来，并将应用程序部署到由云服务提供商管理的虚拟环境中。Serverless模式主要有以下优点：

成本效益：Serverless模式按需付费，只有在使用时才需要付费，这可以有效地降低应用程序的成本。

自动伸缩：Serverless模式可以根据请求的数量和负载自动伸缩，这使得应用程序更加灵活和可扩展。

高可用性：Serverless模式自动处理应用程序的高可用性，由云服务提供商来负责应用程序的管理和运维。

4、分布式事务模式

在分布式系统中，一些事务可能涉及到多个服务之间的数据交互。分布式事务模式旨在处理这种情况。它使用一种称为“两阶段提交”（Two-Phase Commit）的协议，确保所有涉及的服务都在事务提交前完成它们的操作。在两阶段提交中，所有服务都参与到一个事务中。第一阶段，事务协调器要求每个参与者准备提交它们的操作。如果所有参与者都准备好了，那么在第二阶段，事务协调器会通知所有参与者提交它们的操作。如果有任何一个参与者没有准备好或者提交失败，那么整个事务将被回滚。

5、可观测架构

可观测架构是指一种可以轻松监控和调试的架构。它旨在使开发人员更容易地了解系统的运行状况，从而快速发现和解决问题。可观测架构包括以下方面：

日志记录：记录系统中发生的事件和异常情况。

指标：收集关键指标，如请求次数、响应时间和错误率等。

分布式跟踪：记录跨服务调用的路径和性能。

警报：基于指标和日志记录，设置警报以通知开发人员和运营人员发生问题时。

**六、架构模式特点**

云原生架构的模式特点包括现收现付、自助服务基础设施、分布式架构、管理服务、自动放缩、自动恢复。

现收现付是指云原生架构中，资源使用时按需付费，不会有大量资本投入。这是云原生架构的重要特点之一。传统IT架构的投入要比云原生架构更多，因为需要购买大量的硬件、网络和软件，而且这些资源不一定会被充分利用。而云原生架构中的现收现付模式可以使企业按照实际需求使用云服务，使IT成本更加合理。

自助服务基础设施是指企业可以使用API、控制面板等方式来管理和操作基础设施。云原生架构的自助服务基础设施能够帮助企业将基础设施作为服务来管理，自动化运维，并且实现按需扩展和缩小的能力。这样一来，开发团队可以通过API和控制面板来获取和管理基础设施，大大降低了部署和维护的成本和时间。

分布式架构是云原生架构中的另一个特点。在分布式架构中，应用程序的不同模块可以分布在不同的服务器或容器中，实现高可用、可伸缩性和容错性。由于云原生架构的分布式架构可以实现应用程序的高度并发和水平扩展，所以它比传统的单体应用程序更具有优势。

管理服务是指在云原生架构中，管理应用程序的各种服务的技术。这些服务可以包括日志记录、监控、自动化测试和安全管理等。云原生架构中的管理服务可以提高开发和部署的效率，减少错误和故障的发生，同时也能够提高应用程序的可靠性和安全性。

自动放缩是指云原生架构中的应用程序可以自动根据负载变化来调整容器或虚拟机的数量。自动放缩可以确保应用程序始终具有所需的容量，不会因为负载的变化而导致应用程序的中断。自动放缩还可以提高应用程序的效率，减少不必要的资源浪费。

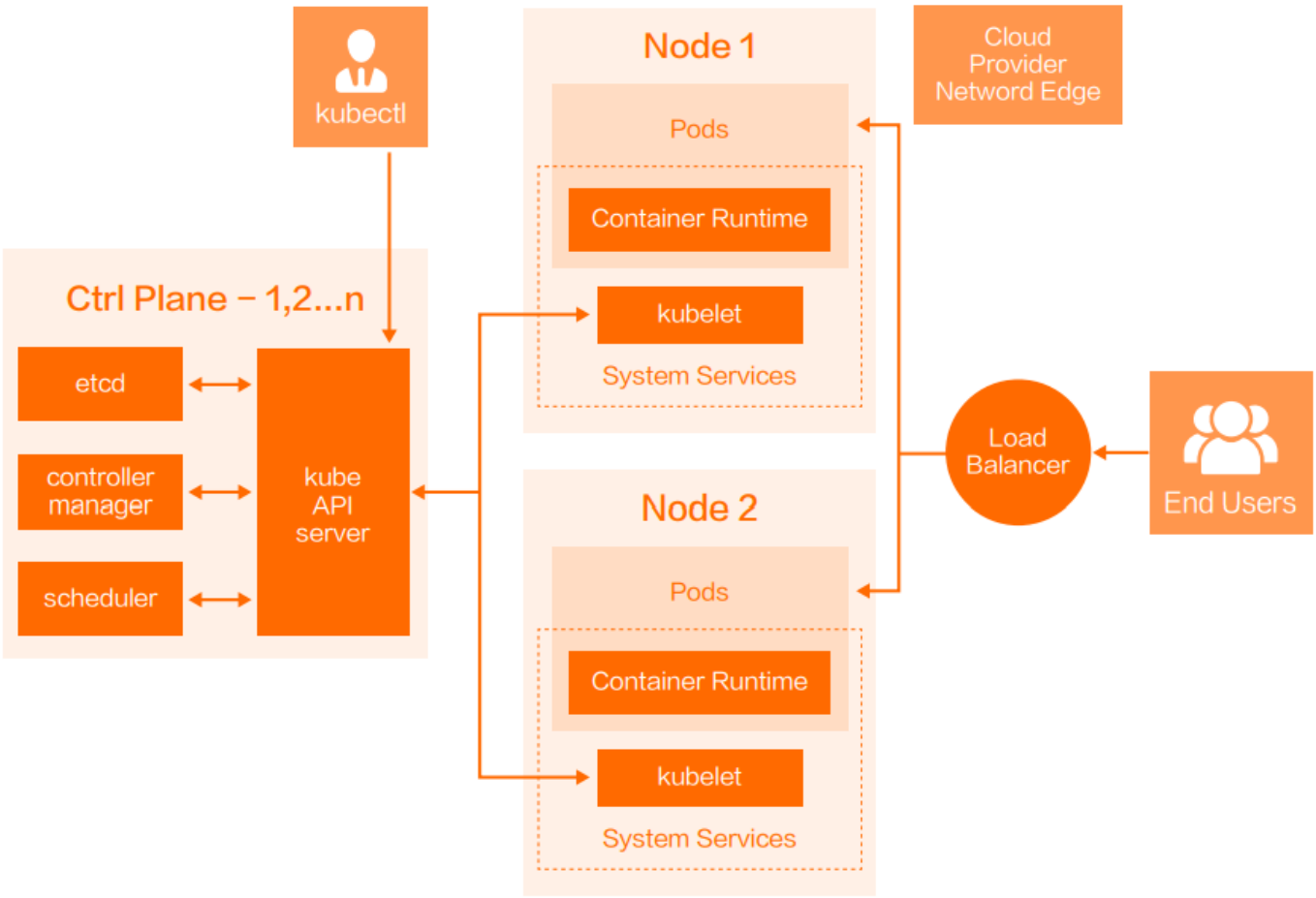
自动恢复是指在云原生架构中，应用程序可以自动恢复在发生故障时恢复运行。自动恢复可以帮助应用程序快速恢复正常运行状态，减少对业务的影响。云原生架构中的自动恢复能力可以提高应用程序的可靠性，确保服务的连续性和可用性。

**七、容器技术**

容器技术是云原生架构中非常重要的一部分，它使得应用程序可以更加高效地运行和部署。在传统的应用部署模式中，应用程序通常被打包成一个完整的虚拟机镜像，这使得应用程序的启动和停止过程比较慢，且需要占用较多的系统资源。而容器技术则提供了一种更轻量级的部署方式，容器可以在操作系统层面提供隔离和资源管理，同时具有更快的启动时间和更小的系统资源占用。

云原生架构中常用的容器技术包括Docker和Kubernetes。Docker是一种轻量级的容器引擎，它可以将应用程序和其依赖项打包成一个可移植的容器镜像，使得应用程序可以在不同的环境中运行。Docker还提供了一套完整的命令行工具，可以帮助用户方便地创建、部署和管理容器。

Kubernetes是一个开源的容器编排平台，它可以管理多个Docker容器，使其可以在不同的主机上自动扩展和负载均衡。Kubernetes还提供了一套完整的API，可以用于管理容器集群、自动伸缩、滚动更新、监控和日志管理等操作。Kubernetes的设计理念是以容器为中心，为应用程序提供高可用、自动化的运行环境。




**八、云原生微服务**

云原生微服务是云原生架构中的一个核心概念，它是指将一个应用程序划分为多个小型的、自治的服务单元，这些服务单元可以独立开发、测试、部署和扩展。云原生微服务通常使用轻量级的通信协议进行通信，并使用标准化的接口定义语言进行描述和交互。

云原生微服务的特点是自治、可独立部署、可扩展、可编排。它们的自治性意味着它们能够独立地运行和管理，并且不会被其他服务单元的故障所影响。此外，云原生微服务通常是以容器的形式部署，这使得它们可以在不同的环境中（如本地开发环境、测试环境、生产环境）进行部署和运行，并且可以方便地扩展。

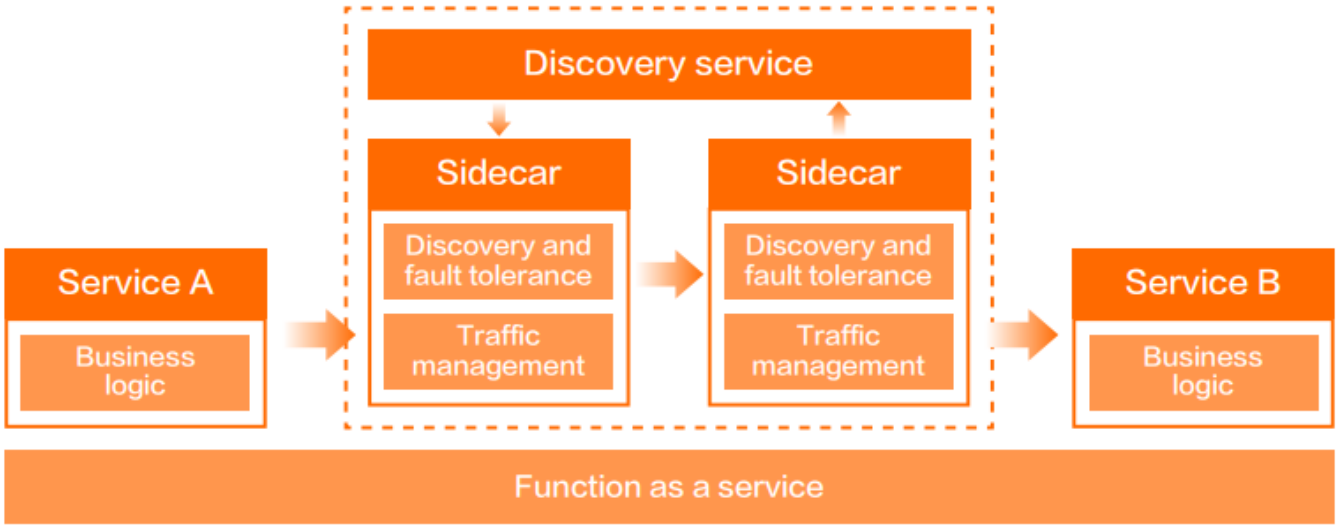
云原生微服务具有高度的可维护性和可测试性。由于微服务的独立性和自治性，每个微服务都可以独立地进行测试和维护。此外，微服务的小型化也使得代码变得更加清晰和可读，使得开发人员能够更轻松地理解和修改代码。

云原生微服务的优势包括更快的开发和交付速度、更高的可靠性和可扩展性、更灵活的架构和更好的适应性。通过将应用程序划分为多个小型的、自治的服务单元，开发人员可以更加专注于每个服务单元的开发和测试，并且能够更快地部署和交付整个应用程序。此外，微服务的自治性使得它们更加可靠，并且能够更好地适应变化的需求和环境。

Apache Dubbo作为源自阿里巴巴的一款开源高性能RPC框架，特性包括基于透明接口的RPC、智能负载均 衡、自动服务注册和发现、可扩展性高、运行时流量路由与可视化的服务治理。经过数年发展已是国内使用最广泛的 微服务框架并构建了强大的生态体系。为了巩固Dubbo生态的整体竞争力，2018年阿里巴巴陆续开源了 SpringCloud Alibaba(分布式应用框架)、Nacos(注册中心&配置中心)、

Sentinel(流控防护)、Seata(分布式事务)、 Chaosblade(故障注入)，以便让用户享受阿里巴巴十年沉淀的微服务体系，获得简单易用、高性能、高可用等核心 能力。Dubbo 在v3中发展Service Mesh，目前Dubbo协议已经被Envoy支持，数据层选址、负载均衡和服务治理方面的工作还在继续，控制层目前在继续丰富Istio/Pilot-discovery 中。

下图为第三代微服务架构




**九、云原生架构的应用**

1、云原生数据库

云原生数据库是在云原生架构下部署和管理的数据库系统，具有高可用性、弹性、可扩展性等特点。相比于传统的关系型数据库系统，云原生数据库更注重分布式存储和计算，适合于云原生应用的场景。

例如，云原生数据库ArangoDB就是一款开源的多模型数据库，支持图形数据库、文档数据库和键值存储等多种模型，能够满足不同场景下的数据存储需求。其采用分布式架构，支持数据自动分片和副本机制，保证了高可用性和可扩展性。

2、实时数据处理

实时数据处理是指对实时数据流进行实时处理和分析，以快速响应和处理各种数据。在云原生架构中，实时数据处理需要具备高可靠性和高可扩展性等特点，因此采用了分布式架构和微服务架构等技术

例如，云原生实时数据处理平台Apache Flink就是一款开源的分布式流处理框架，具有低延迟、高吞吐量、容错性强等特点，能够处理实时数据流并提供复杂事件处理、数据流聚合和数据流分析等功能。

3、消息传递系统

消息传递系统是指在分布式系统中，不同的服务之间通过消息传递来进行通信。在云原生架构中，消息传递系统采用了微服务架构和事件驱动架构等技术，能够快速响应和处理各种消息。

例如，云原生消息传递系统Apache Kafka就是一款开源的分布式消息队列系统，具有高可用性、高扩展性、高吞吐量等特点。它支持异步消息传递和批量处理，能够快速响应和处理大量的消息。

4、人工智能和机器学习

云原生架构在人工智能和机器学习领域的应用越来越广泛，因为这些领域需要大量的计算能力和数据处理能力。云原生架构的弹性和可伸缩性可以帮助机器学习模型在需要时快速部署和扩展。此外，云原生架构也可以通过服务化架构模式来解耦和扩展不同的服务，从而提高机器学习和人工智能系统的可维护性和可靠性。

例如，Google在人工智能和机器学习领域的应用中采用了云原生架构。Google Cloud AI平台提供了多种云原生服务，包括TensorFlow，Kubernetes和Cloud TPUs。这些服务可以帮助开发人员轻松构建和部署机器学习模型，同时保证高可用性和可伸缩性。

亚马逊AWS的机器学习服务，其中包括云原生的数据存储、处理和分析服务。亚马逊AWS还提供了SageMaker，一种云原生的机器学习平台，可以帮助开发人员构建、训练和部署机器学习模型，同时支持自动扩展和自动调整。

**十、未来发展**

随着云计算的普及和发展，云原生架构已经成为了云计算领域的热门话题。它是一种基于容器、微服务、自动化、持续交付等技术的新一代应用程序开发和部署架构。云原生架构具有高效、灵活、可扩展和可靠性高等特点，在当前云计算领域的发展趋势中占有重要的地位。

1、容器化技术的普及

容器化技术是云原生架构的重要组成部分，可以将应用程序打包成一个可移植、自足的单元，具有快速部署、弹性伸缩、跨云平台等优势。目前，容器化技术已经得到了广泛的应用和支持，包括开源容器引擎Docker、Kubernetes等。未来随着容器化技术的普及和完善，云原生架构也将更加流行和广泛应用。

2、自动化运维技术的发展

自动化运维技术是云原生架构的又一重要组成部分，可以自动化地部署、扩展和管理应用程序。随着自动化运维技术的不断发展和完善，未来云原生架构的自动化程度将越来越高，大大提高了应用程序的可靠性和稳定性。

3、大数据技术的应用

随着云原生架构的应用场景越来越多样化，大数据技术在云原生架构中的应用也越来越广泛。大数据技术可以帮助应用程序实现实时数据处理、分析和挖掘，提高应用程序的智能化和自动化水平，未来大数据技术的应用将会越来越广泛。

4、安全性和可观测性的提升

云原生架构的安全性和可观测性一直是云计算领域的重要话题。未来云原生架构将会加强对安全性和可观测性的支持和应用，包括自动化安全检测和修复、安全日志分析和监控、全链路跟踪等技术。

5、Kubernetes的普及。

Kubernetes是一个容器编排系统，可以协调和管理多个容器化应用程序的部署、扩展和运行。Kubernetes解决了云原生应用的一些挑战，例如自动化部署、自动化伸缩、自动化负载均衡和自动化

故障恢复。因此，它被广泛认为是构建云原生应用的必要工具之一。Kubernetes已经成为云原生技术栈中不可或缺的一部分，并逐渐成为云原生架构的标准之一。

6、云原生安全

随着云原生应用的部署和运行环境变得越来越复杂，保证云原生应用的安全性也变得越来越困难。云原生安全需要从应用程序、容器、集群、网络等各个层面上进行保护。未来，随着云原生应用的广泛应用，云原生安全将成为一个重要的研究方向和市场需求。

未来发展方向包括更好的自动化和AI集成。随着自动化技术和人工智能技术的不断发展，未来的云原生架构将更加注重自动化和AI集成。通过自动化和AI集成，云原生应用将能够更加智能化地管理和运行，实现更高效的资源利用和更优秀的性能表现

许鑫——微服务架构

**一、总论**

所谓微服务架构，就是将一个应用程序拆分为多个小型、自治的服务。每个服务都具有特定的功能和数据处理职责，并通过轻量级通信协议（比如RESTful）进行交互。这样做可以有效提高应用程序组件之间的耦合度，增强系统的可维护性和灵活性。

**二、微服务架构的优点：**

1、首先，由于微服务架构将一个大型应用程序拆分为多个小型自治服务，每个服务各司其职、相互独立，因此更容易实现代码的复用和重构。同时，开发者只需要关注所负责的微服务，而不必了解整个应用程序的所有细节。这样就能够更好地促进团队之间的合作和协作，提高项目实现效率。

2、与传统的单体式应用相比，微服务架构还具有更强的系统灵活性。例如，当需要在应用中增加或减少功能时，只需修改对应的微服务即可，无需修改整个系统。这就使得应用软件研发过程更加敏捷快速，满足市场变化需求。

3、微服务架构的另一个优点是易于发布和部署。在采用微服务架构方式后，我们可以将不同的服务模块部署在不同的主机上，避免了单机环境的限制问题。而且每个微服务都是一个独立的运行单元，可以部署在云端或本地服务器上。这样便于实现自动化部署、预测性伸缩以及弹性容错等特性，使系统更加稳定、高效。

4、一项常被提到的微服务优点是，每个服务可由不同团队开发和维护。嵌入式产品设计公司Harman International Industries便使用微服务架构来实现跨多个团队协作，推出据称是"纯粹数字化车辆平台"的车联网解决方案。他们选择采用Kubernetes和Docker作为基本的容器技术架构，并将每个微服务放置在独立的容器上，在不同团队的同事之间共享代码和部署工具，更高效地完成以往需要花费几周来适配单个产品特性的任务。

5、另外一个重要的微服务架构特点是，它能够使整个系统变得更加弹性。当系统采用微服务架构时，一旦某个微服务无法正常工作，其余的服务仍然可以继续运行。这样就能够保证业务的持续进行，同时快速检测和修复故障，最小化因故障导致的停机时间，更直接地提高整个系统的安全性、健壮性、灵活性等。

**三、微服务架构的缺点**

微服务架构是一种基于分布式系统思想的软件设计风格，它通过将整个应用拆分为可独立开发和部署的小型服务，来提高系统的可扩展性、灵活性、可维护性和可测试性等方面。尽管微服务在很多情况下可以带来巨大的好处，但是在实践中也有不少挑战和问题需要我们关注。以下是微服务架构的缺点：

1、系统复杂度：由于微服务架构的核心理念是将一个单olithic应用拆分成多个网络化的服务，因此会导致系统整体的复杂度增加。如何管理这些服务以及保证它们之间的通信，会变得更加困难。

2、部署和升级：由于单一的微服务被分解成了许多小容器，一旦要对微服务进行升级，就需要为每个服务单独执行部署，系统的部署复杂性因此大幅上升，一套服务如果要更新，所有服务都要重新部署，运维的工作量也会增多。

3、运维监控：在微服务架构中，各个服务之间的依赖关系可能比较复杂，当服务出现故障时，排查问题也比较难，需要有完善的监控手段来应对各种问题。

4、原有架构和技术体系的转型：如果企业原先使用的是传统的单一架构或整体部署方式，那么采用微服务架构就需要进行许多调整转变。光是从技术角度考虑，如何确保多个微服务之间的协同和正常运行至关重要。同时，从组织架构层面出发，微服务需要依赖专业分工团队解决及时问题，这也意味着与既存架构逐渐淘汰相关的人们将会失去其职业能力的市场价值。

5、数据管理：在微服务架构中，不同的服务之间需要共享数据，但跨服务的数据库管理和事务处理会带来更大的挑战。之前单态架构所输入的数据单元不能用于新微服务需求，并且在不同servicers状态下的错误还相当容易带来更大范围、毁灭性的后果。保证流程命令权益的互斥与共享，在微服务架构中都被逐渐聚焦到语言框架和设计与实现方法上。

6、调试和测试复杂度：在微服务架构中，各个服务之间的依赖较为复杂，这导致了在开发和测试过程中往往会出现问题。与此同时，难以维护约束或冗余空格将会带来外部调试难度。对微服务架构进行调试和测试，需要各种不同的技术特性和测试手段，通常也很难对整个系统的封闭和功能进行上下文特定的测试或模拟。

7、安全性：由于微服务架构系统较为分散，并且多节点、多人访问的特点，因此，从安全角度考虑维护一个稳健的环境是很有必要的。服务其间的协作依赖关系、授权验权、漏洞和服务管理等领域都存在一些重大的威胁，它们在故意攻击、数据恢复操作、由数据流跟踪露出的漏洞、基于底层物理排列资源的共通的安全策略等方面更容易遭到利用。

这些都是微服务架构的缺点所体现出来的，当然了，也存在各种解决方法和实践经验，这其中就需要一个专业的团队来推进实践、解决这些问题和逐渐提高企业运维水平。

此外，还可以考虑使用API网关来简化客户端与多个微服务之间的通信。API网关可以将所有微服务的HTTP API封装在一起，并且提供统一的入口点，从而提高整体系统的安全性和可维护性。

最后，在实际应用开发中，我们可以借助第三方框架和平台来加速微服务架构的实现。比如Spring Cloud、Kubernetes、Istio等等都是非常流行的微服务框架。

微服务架构是一种基于分布式系统思想的软件设计风格，它采用多个小型操作单元来拆解复杂的应用程序，从而实现了软件的高度可扩展性、灵活性、可靠性和可测试性。

**四、微服务架构的应用**

在当今快速发展的互联网行业中，微服务架构正变得越来越流行和普及。它因为其分割程度更细腻、功能组合更精细，仿佛具备可竞速性和接受多种客户端服务请求能力的特征，正在被越来越多的企业所使用。 以下是微服务架构的应用场景：

1、大型企业级应用：在大型企业中，一个庞大的单态架构应用程序往往非常难以管理、扩展和更新。这时候微服务架构就显得特别有优势，因为它可以将整个应用拆分成许多小部分，然后将每个小部分独立开发和部署，便于组合、更新和维护。

2、云原生应用系统：云原生应用系统是一种新的应用开发模式，它利用云计算的核心技术，在云平台上运行多个敏捷轻便的微服务组件，随着云计算技术的发展和应用，微服务架构也成为了云原生应用系统的首选技术。

3、满足高并发和用户流量要求：在高并发和用户流量加载场景下，单态架构可能承受不住这种压力，如电商平台、在线支付平台等。为了解决这个问题，一些企业开始采用微服务架构来优化其应用程序，并且使其具有更高的响应速度和更强的可扩展性，以满足大量用户同时访问的需求。

4、特定的信誉和可靠性场景：在某些需要特别高可靠性和可信度的场景下，微服务架构可以避免整体故障或错误对整个应用程序产生影响。例如，银行和保险公司通常需要保证他们开放的API接口运行准确、可信和高效，那么采用微服务架构可以使得每个服务都独立运行，并通过有效的灵活故障恢复机制或调换机制来降低人为错误的发生率。

5、多语言的开发环境：最后，微服务架构可以帮助企业整合多种代码库和开发环境，实现混合语言开发，也就是微服务之间，其对应的技术框架不一定要保持完全相同。这种特点可以使得企业采用自己擅长的语言和技术栈开发外部功能组件，提高开发效率和质量。

需要注意的是，虽然微服务架构带来了很多好处，但它并不是所有场景都适用，企业在考虑转型时应该结合自身实际情况进行思考选择，是否尝试转向微服务架构。此外，企业在使用微服务架构时需要投入更多的人力和资源，尤其是建立专业团队以及制定过程、管理准则，才能够取得好的效果和成果。

**五、微服务架构的未来**

随着互联网技术的发展和应用，微服务架构已经成为现代软件开发领域中不可或缺的一部分。未来，微服务架构将继续发挥其强大的优势，并在多个领域得到广泛应用和进一步优化。

以下是四个关键方面，展示微服务架构的未来。

1、自动化运维：在微服务架构下，开发人员可以建立独立的、具体功能组合的微服务模块，这些模块需要依赖基础设施和开发工具支撑。现代自动化运维平台可以与微服务结合，消除手动进行快速配置、部署和监测操作的繁琐，协同分布于不同客户端并且基本独立部署的各种微服务模块，并轻松解决监控报警和故障恢复等问题。

2、云原生的应用发展趋势：随着云计算技术的快速发展和云计算服务的普及，云原生也让微服务架构更加流行。云计算平台上,微服务模块级别的按需扩展能力和智能故障恢复机制提供了非常大的灵活性和可扩展性， 组合这些微服务可以让大规模应用被快速构造。据了解,云原生技术已经逐渐成为企业应用研发的重要趋势。

3、笛卡儿接口独立编写、测试和交付能力：在微服务架构下，每个微服务都有其特定的由外部继承和调用接口。而这些微服务之间的耦合度非常低，甚至可以使用不同的开发语言和相应的技术栈编程实现。因此，以微服务为基础的软件架构被赋予了更高的选择性能，设计者可以只专注于接口 API 的定义，从而提高软件的可维护性和拓展性

4、Micro Frontends：长大多分布式应用中，前端视图层也是一个非常特殊但至关重要的部分。Micro frontends 特指前端使用微服务架构思路，将应用按组件或模块划分，从而实现类似于后端微服务的前端组合模式。每个模块拥有独立的渲染、数据获取等管理机制，这种方法可以有效提高核心代码质量并且保证前端 ui 体验一致性。

总的来说，微服务架构作为一种高度可扩展性的应用架构方式，将在未来继续发挥其优势和不断演化。大量的垂直细分和逐步组合下，微服务将慢慢从一个受众较小、有限领域低调地扫过来，成为通用型云计算基础设施的核心模块之一。同时，随着人工智能等新技术的不断发展和应用，微服务架构也将迎来更多新的变革和突破，并在未来的数字经济中扮演更重要的角色。

**付召帅——面向服务体系结构**

面向服务体系结构（SOA）是一种常见的软件设计范式，其核心思想是将应用程序拆分成相互独立的服务，并使用面向服务的方法来开发和部署它们，从而可以实现高效的重用和集成。SOA演变到Web Service 阶段后, 更加的规范化SOAP协议的普及化使得各个服务得以自主发布和选择， SOA 也在未来几年内成为企业应用程序开发的第一选择。

对于SOA，需要理解以下关键概念：

1、服务: 系统中相互独立的软件部分，该部分仅在需要时才会被调用。

2、服务描述： 描述服务特性和功能的元数据，如WSDL和Service Registry

3、服务协议：约定服务之间如何交换信息。如SOAP/RESTful, 通信模型等。

4、服务组合：将服务聚合成较大、更高级别的服务或应用程序。

5、服务治理： 实现服务租赁、安全控制和线程管理等。

尽管SOA 是为各个服务应用程序提供了一些很好的协商机会，但是如果不加以规范框架的控制容易出现以下问题:

1、大量冗余信息: SOA 中的滥用代码或服务过多会导致系统混乱且难以维护。

2、过度重用: 可能存在某些服务被过度使用而变得难以维护，同时可能存在服务的争用和不一致的版本问题。

3、总体复杂性：面向服务带来的整体复杂性使得系统集成和维护整体困难。

总之，面向服务体系结构是一个非常灵活和可扩展的架构范式，可以有效地促进业务流程自动化和复杂性管理，同时也需要注意其架构设计的规范化和框架的选择。

2.对比书上各种软件体系结构风格和视图特点，思考自己项目属于哪种设计风格?

网上搜索最新的软件体系结构资料，如MVC、Kruchten 4+1视图等。

我们的博客项目，采用了分层和微服务结合的方式。分层结构可以帮助我们将表示层、业务逻辑层和数据访问层分离，便于维护和扩展。同时，采用微服务架构可以将应用程序分解为一组独立的服务，这些服务可以独立开发、部署和扩展，有利于实现高并发、高可用性和敏捷开发。

MVC：

MVC（Model–View–Controller）模式是软件工程中的一种软件架构模式，它把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

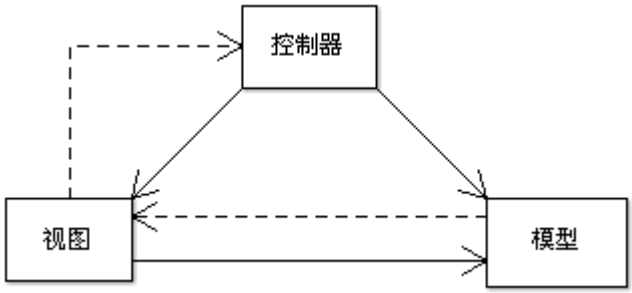
MVC 模式的目的是实现一种动态的程序设计，简化后续对程序的修改和扩展，并且使程序某一部分的重复利用成为可能。除此之外，MVC 模式通过对复杂度的简化，使程序的结构更加直观。软件系统在分离了自身的基本部分的同时，也赋予了各个基本部分应有的功能。专业人员可以通过自身的专长进行相关的分组：

**模型（Model）**：用于封装与应用程序业务逻辑相关的数据以及对数据的处理方法。Model 有对数据直接访问的权力，例如对数据库的访问。Model 不依赖 View 和 Controller，也就是说， Model 不关心它会被如何显示或是如何被操作。但是 Model 中数据的变化一般会通过一种刷新机制被公布。为了实现这种机制，那些用于监视此 Model 的 View 必须事先在此 Model 上注册，由此，View 可以了解在数据 Model 上发生的改变。（如，软件设计模式中的“观察者模式”）；

**视图（View）**：能够实现数据有目的的显示（理论上，这不是必需的）。在 View 中一般没有程序上的逻辑。为了实现 View 上的刷新功能，View 需要访问它监视的数据模型（即 Model），因此应该事先在被它监视的数据那里注册；

**控制器（Controller）**：起到不同层面间的组织作用，用于控制应用程序的流程。它处理事件并作出响应。“事件”包括用户的行为和数据 Model 上的改变。

MVC 模式的描述如下图所示：




Kruchten4+1视图：

Kruchten 4+1视图由Philippe Kruchten于1995年提出。这种设计方法旨在通过多个视图来描述一个软件系统的体系结构，使得不同的利益相关者能够更容易地理解和沟通系统的设计。

Kruchten 4+1视图包括以下五个视图：

**1.逻辑视图**（Logical View）：这个视图关注系统的功能性。它展示了系统中的主要功能模块、组件和它们之间的关系。逻辑视图通常使用UML类图和包图来表示。

**2.开发视图**（Development View）：这个视图关注系统的实现，主要描述了软件代码的组织。它展示了系统的模块、子系统、文件、文件夹等，并展示了它们之间的依赖关系。开发视图通常使用UML组件图来表示。

**3.过程视图**（Process View）：这个视图关注系统的运行时行为，关注点在于系统的并发、性能和可伸缩性。它展示了系统中的进程、线程以及它们之间的交互。过程视图通常使用UML活动图、顺序图和状态图来表示。

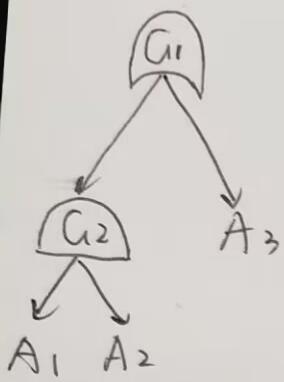
**4.物理视图**（Physical View）：这个视图关注系统的部署，描述了系统在硬件和网络环境中的分布。它展示了软件组件如何映射到硬件组件，以及它们之间的通信关系。物理视图通常使用UML部署图和节点图来表示。

**5.场景（用例）视图**（Scenarios, or Use Case View）：这个视图是前四个视图之间的“胶水”，用于驱动和验证其他视图。它展示了系统的关键功能需求和用例，并通过描述系统的典型运行情况来展示其他视图之间的关系。场景视图通常使用UML用例图和顺序图来表示。

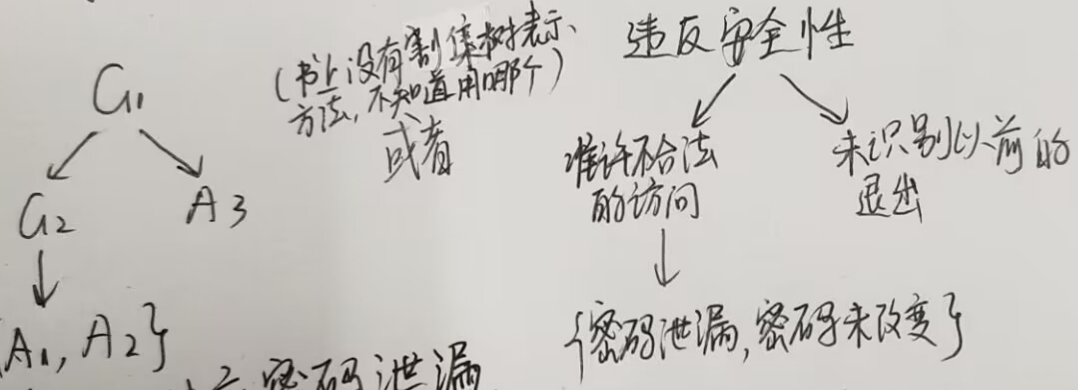
3.第五章课后习题14，故障树转割集树练习。针对自己项目分析、描绘故障树，分解为割集树 （附到最终提交的SAD）

课后题14：

5-11对应的故障树如下：




转换成割集树如下：




{A1, A2}

其中A1对应密码泄漏，

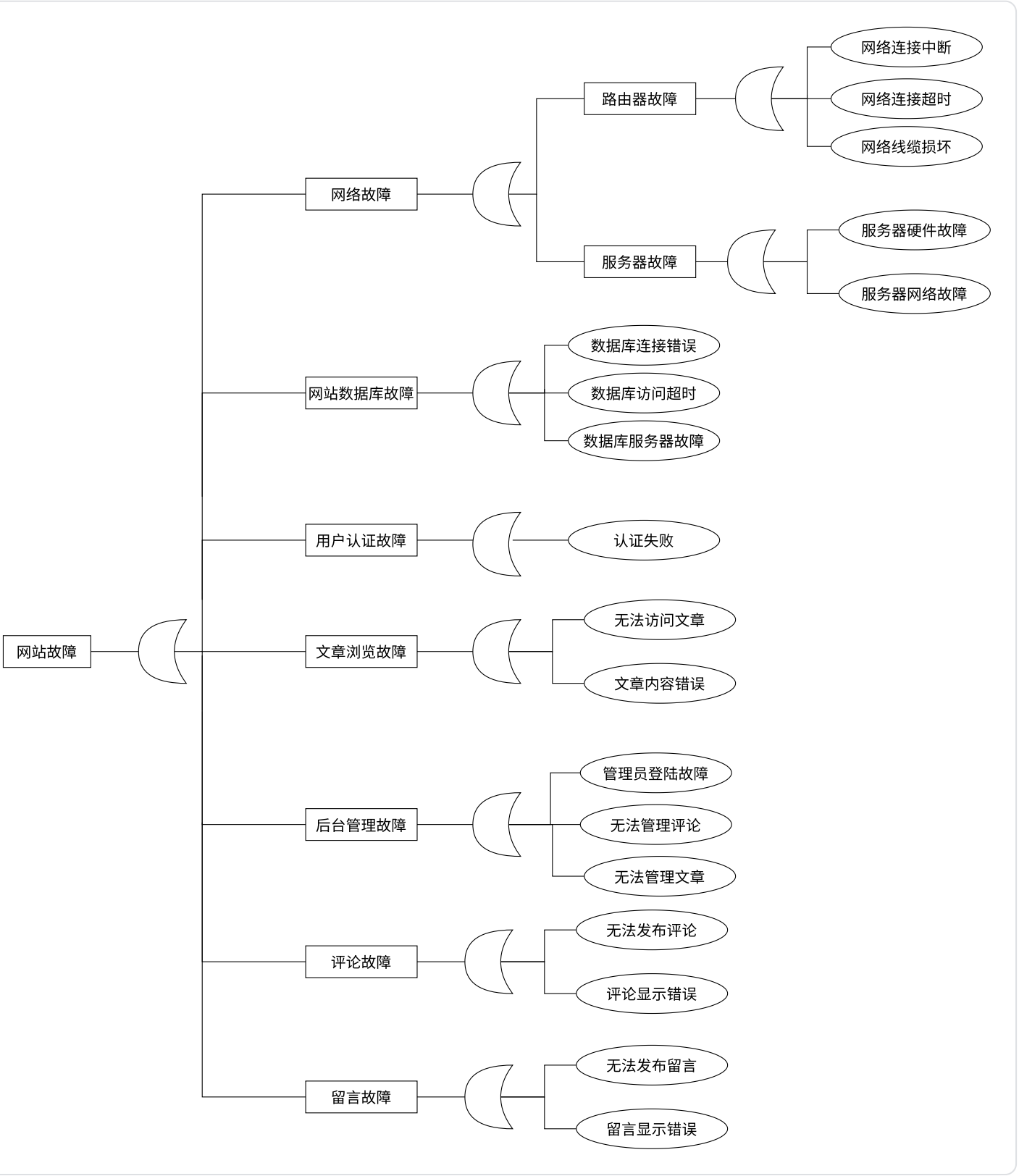
A2对应密码未改变，

A3对应未识别以前的退出。

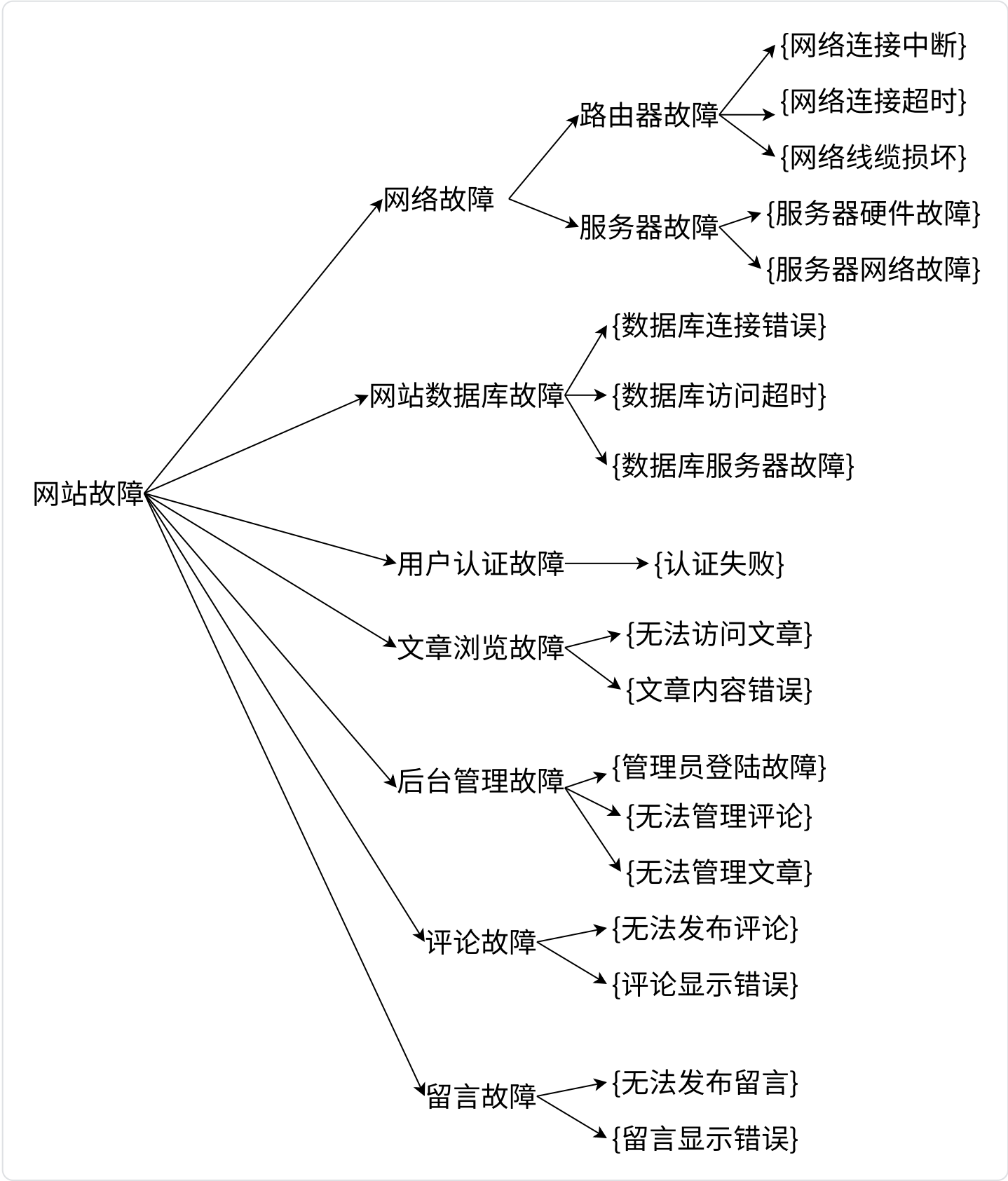
可以看到割集为{A1, A2}，{A3}



项目的故障树分析：


网站故障
网络故障
路由器故障
网络连接中断
网络连接超时
网络线缆损坏
服务器故障
服务器硬件故障
服务器网络故障
网站数据库故障
数据库连接错误
数据库访问超时
数据库服务器故障
用户认证故障
认证失败
文章浏览故障
无法访问文章
文章内容错误
后台管理故障
管理员登陆故障
无法管理评论
无法管理文章
评论故障
无法发布评论
评论显示错误
留言故障
无法发布留言
留言显示错误

分解为割集树：




4.参阅课本和网上资料，研究经典软件体系结构案例KWIC。An Introduction to Software Architecture，4.1节针对KWIC和自己项目，参考课本ch5 表5-3，小组成员每人给几种不同的体系结构风格设计打分，评最佳

1、**数据流风格: 批处理序列; 管道/过滤器。**

批处理序列：适用于一些需要对一批数据进行批量处理的场景，例如图像处理或音频处理等。但是，对于一个博客网站，每个请求都需要即时响应，因此批处理序列可能不太适合。

管道/过滤器：适用于一些需要处理大量数据的场景，例如日志处理或数据分析等。对于博客网站，管道/过滤器可以被用于处理某些复杂查询，但是这种风格可能过于复杂，不太适合博客网站项目。

2、**调用/返回风格：主程序/子程序；面向对象风格；层次结构。**

主程序/子程序：主程序/子程序是一种简单的结构，适用于需要按照固定顺序执行一系列步骤的场景。对于博客网站项目，这种风格可以被用于处理某些事务性操作，例如用户注册或登录等。

面向对象风格：面向对象编程提供了一种基于对象和类的编程模式，可以帮助组织代码并提高代码复用性。对于博客网站项目，面向对象编程可以被用于设计数据模型和实现用户认证等功能。

层次结构：层次结构风格是一种将系统分解为多个层次的设计方法，每个层次负责不同的任务。对于博客网站项目，层次结构可以被用于分解系统为多个模块，并确保每个模块只负责一个特定的任务。

3、**独立构件风格：进程通讯；事件系统。**

进程通讯：进程通讯是一种将系统分解为多个进程并通过进程间通讯进行通信的设计方法。对于博客网站项目，进程通讯可以被用于将系统分解为多个独立的模块，并确保模块之间的通信是安全和可靠的。

事件系统：事件系统是一种基于事件驱动的设计方法，通过事件触发系统的不同模块进行响应。对于博客网站项目，事件系统可以被用于处理用户的不同操作，例如用户发布文章或评论等。

4、**虚拟机风格：解释器；基于规则的系统。**

解释器：解释器是一种将代码解释为可执行代码的设计方法。对于博客网站项目，解释器可以被用于处理某些复杂业务逻辑，例如搜索引擎或分析引擎等。

5、**仓库风格：数据库系统；超文本系统；黑板系统。**

数据库系统：数据库系统是一种将数据组织为表格的设计方法，通过SQL语句进行查询和操作。对于博客网站项目，数据库系统是必不可少的，用于存储用户数据和文章内容等。

超文本系统：超文本系统是一种将文本组织为超链接的设计方法，通过超链接将不同的文本链接起来。对于博客网站项目，超文本系统可以被用于实现文章之间的链接和文章分类等。

黑板系统：黑板系统是一种基于共享状态的设计方法，通过将系统状态存储在共享的“黑板”上，各个模块可以通过读取和修改黑板上的状态进行通信和协作。对于博客网站项目，黑板系统可以被用于处理一些需要协作的业务逻辑，例如文章审批和发布等。

6、**过程控制环路**

过程控制环路是一种将系统的各个组件分解为一个个小的过程，并且每个过程都能相互通信和协作的设计方法。对于博客网站项目，过程控制环路可以被用于分解系统为多个小的组件，并确保这些组件能够相互通信和协作，从而实现复杂的业务逻辑。

7、**C/S风格**

C/S风格是一种将系统分解为客户端和服务器端两部分的设计方法。对于博客网站项目，C/S风格可以被用于将用户界面和后台逻辑分离，并确保用户界面和后台逻辑的高效通信和协作。

8、**B/S风格**

B/S风格是一种将系统分解为浏览器和服务器两部分的设计方法。对于博客网站项目，B/S风格是一种常用的设计方法，可以通过浏览器向服务器请求数据并显示结果，从而实现博客网站的功能。

| 体系结构风格 | 易整合性 | 易于复用 | 易于改变功能 | 易测试性 | 易于改变算法 | 有效数据表示 | 易于改变数据表示 | 性能 | 模块化 | 易理解性 | 安全性 | 易使用性 | 总 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 数据流风格 | 5 | 3 | 1 | 1 | 3 | 4 | 2 | 2 | 1 | 1 | 0 | 0 | 2 |
| 调用/返回风格 | 4 | 4 | 3 | 3 | 2 | 0 | 1 | 1 | 3 | 5 | 0 | 0 | 2 |
| 独立构件风格 | 2 | 4 | 1 | 2 | 2 | 3 | 4 | 4 | 1 | 3 | 0 | 0 | 2 |
| 虚拟机风格 | 1 | 1 | 3 | 3 | 5 | 3 | 3 | 5 | 1 | 2 | 1 | 2 | 2 |
| 仓库风格 | 3 | 3 | 2 | 2 | 1 | 5 | 4 | 3 | 1 | 3 | 1 | 1 | 2 |
| 过程控制环路 | 1 | 1 | 4 | 2 | 4 | 2 | 4 | 4 | 1 | 1 | 0 | 0 | 2 |
| C/S风格 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| B/S风格 | 5 | 5 | 2 | 2 | 2 | 3 | 3 | 4 | 3 | 3 | 3 | 5 | 4 |

2. 完成自己项目的SAD