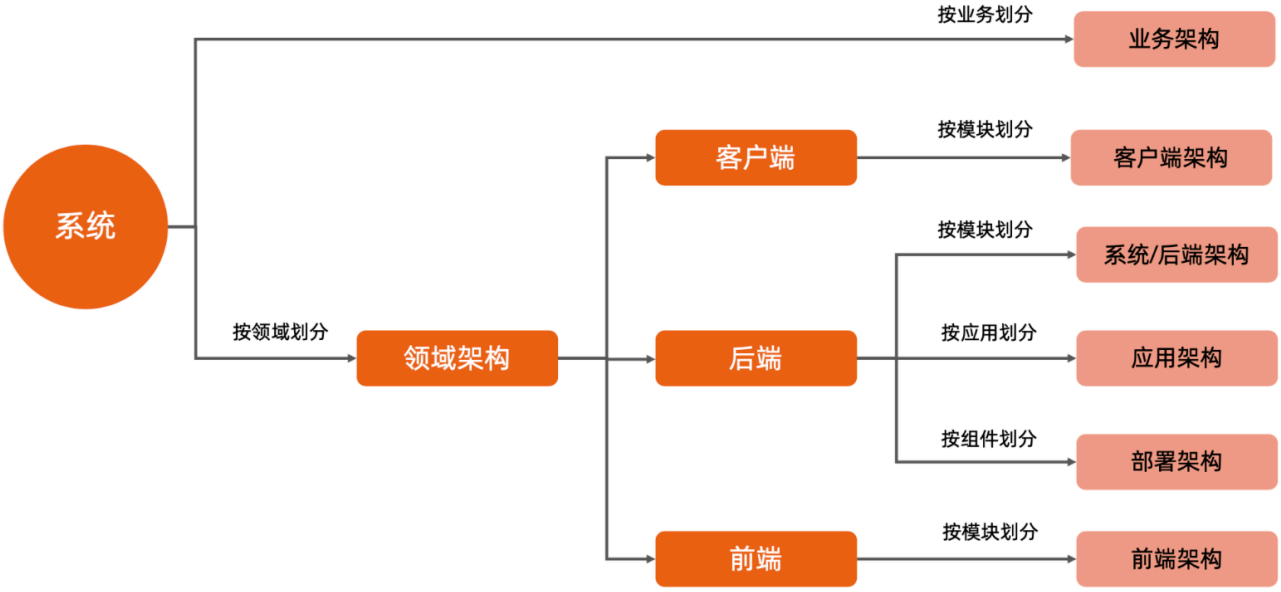
一。知识点梳理

1. 模块一

主要讲解了什么是架构以及如何做好架构设计。

1. 4R软件架构即：架构是分层的（Rank），定义了系统由哪些角色（Role）组成，角色之间的关系（Relation）和运作规则（Rule）。
2. 介绍了大厂常见的架构图和画法：
    1. 静态架构图/系统架构图，描述了 Role + Relation，大概分类：



    2. 系统序列图（动态架构图）：用 UML 序列图来画，这类图描述了核心业务功能 Rule
3. 介绍了常见的架构设计方法论：面向模式，面向风险，DDD，面向复杂度
4. 架构设计三原则：
    1. 合适原则：“合适优于业界领先”，考虑资源、时间、业务本身
    2. 简单原则：“简单优于复杂”，关键属性有可靠性、可扩展性和故障处理效率

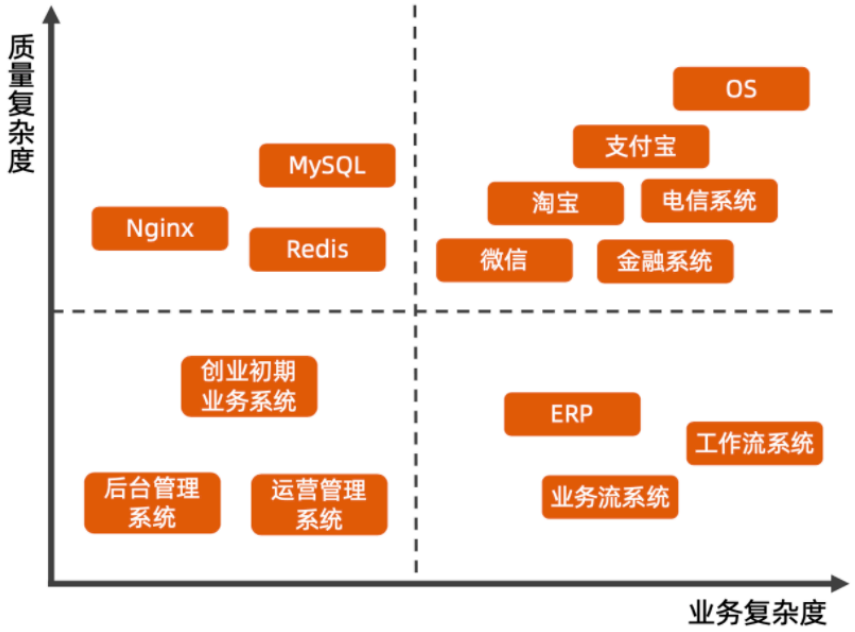
       复杂度分为 内部复杂度 和 外部复杂度。降低内部复杂度，同时可能会带来更多的外部复杂度。
    3. 演化原则：“演化优于一步到位”，要注意 创造、迭代优化和重构重写。

2. 模块二

主要讲了如何设计 可扩展架构、高性能架构和高可用架构，并总结了如何全面提升架构设计质量要注意的点。

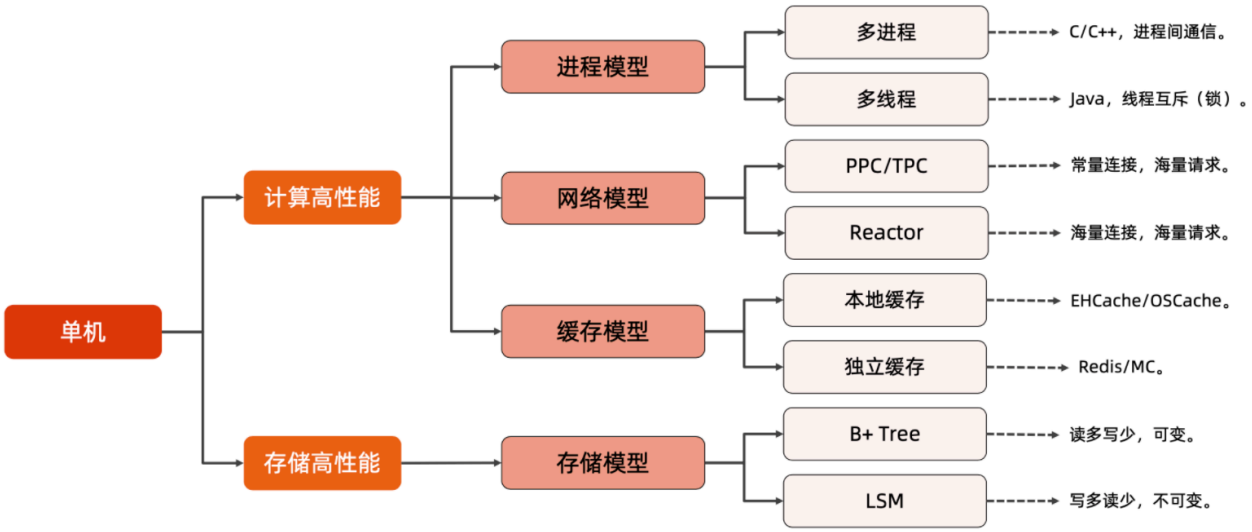
1. 如何设计可扩展架构

1. 明白业务复杂度和质量复杂度是正交的。业务复杂度是业务固有的复杂度，主要体现为难以理解、难以 扩展。例如业务数量多(微信)、业务流程长 (支付宝)、业务之间关系复杂(例如 ERP)。质量复杂度是高性能、高可用、成本、安全等质量属性的要求。



2. 可扩展复杂度模型
    1. 可伸缩（scalability）：系统通过添加更多资源（比如加机器等）来提升性能的能力
    2. 可扩展（extensibility）：系统适应变化的能力，包含可理解和可复用两个部分。
        1. 可理解：假如系统很难理解，那么要加新feature的话就会很困难。好理解的系统会比较容易修改。
        2. 可复用：要加新feature，需要改动很多个地方，这样可扩展的效率就会比较低
    3. 拆分要注意粒度，运用 内外平衡原则 和 先粗后细原则。
    4. 封装：预测变化的方式会决定封装模型的设计。预测要根据 2年原则 和 3次法则（1写2抄3封装）来进行。

2. 如何设计高性能架构
    1. 单机高性能



2. 集群高性能
    1. 任务分配架构：将任务分配给多个服务器执行。复杂度分析：
        1. 增加 “任务分配器” 节点，可以是独立的服务器，也可以是 SDK。
        2. 任务分配器需要管理所有的服务器，可以通过配置文件（e.g IP地址、端口、服务器等），也可以通过配置服务器(例如 ZooKeeper 做配置中心)。
        3. 任务分配器需要根据不同的需求采用不同的算法分配
    2. 任务分解架构：将服务器拆分为不同角色，不同服务器处理不同的业务。复杂度分析：
        1. 增加“任务分解器”节点，可以是独立的服务器，也可以是 SDK。
        2. 任务分解器需要管理所有的服务器，可以通过配置文件，也可以

            通过配置服务器(例如 ZooKeeper)。
        3. 需要设计任务拆分的方式，任务分解器需要记录“任务”和“服务器”的映射关系。
        4. 任务分解器需要根据不同的需求采用不同的算法分配。

3. 如何设计高可用架构：高可用的本质是冗余，所以其必然是“集群”方案。
    1. 计算高可用
        1. 任务分配，相比于高性能的任务分配，多了“状态检测，在故障时进行切换”。
        2. 任务分解，相比于计算高性能任务分解，多了“状态检测，在故障时进行切换”。
    2. 存储高可用
        1. 数据复制格式：
            1. 复制命令
            2. 复制数据本身
            3. 复制文件
        2. 数据复制方式：
            1. 同步复制：服务器1收到用户请求后，将相同的请求发给服务器2 和 3，当用户请求被所有服务器执行成功后，该请求才被认为是执行成功。适合节点比较少的情况，比如主备架构。
            2. 异步复制：服务器1收到用户请求后，执行成功后即可返回给客户端。之后，再发给服务器2 和 3 去执行。

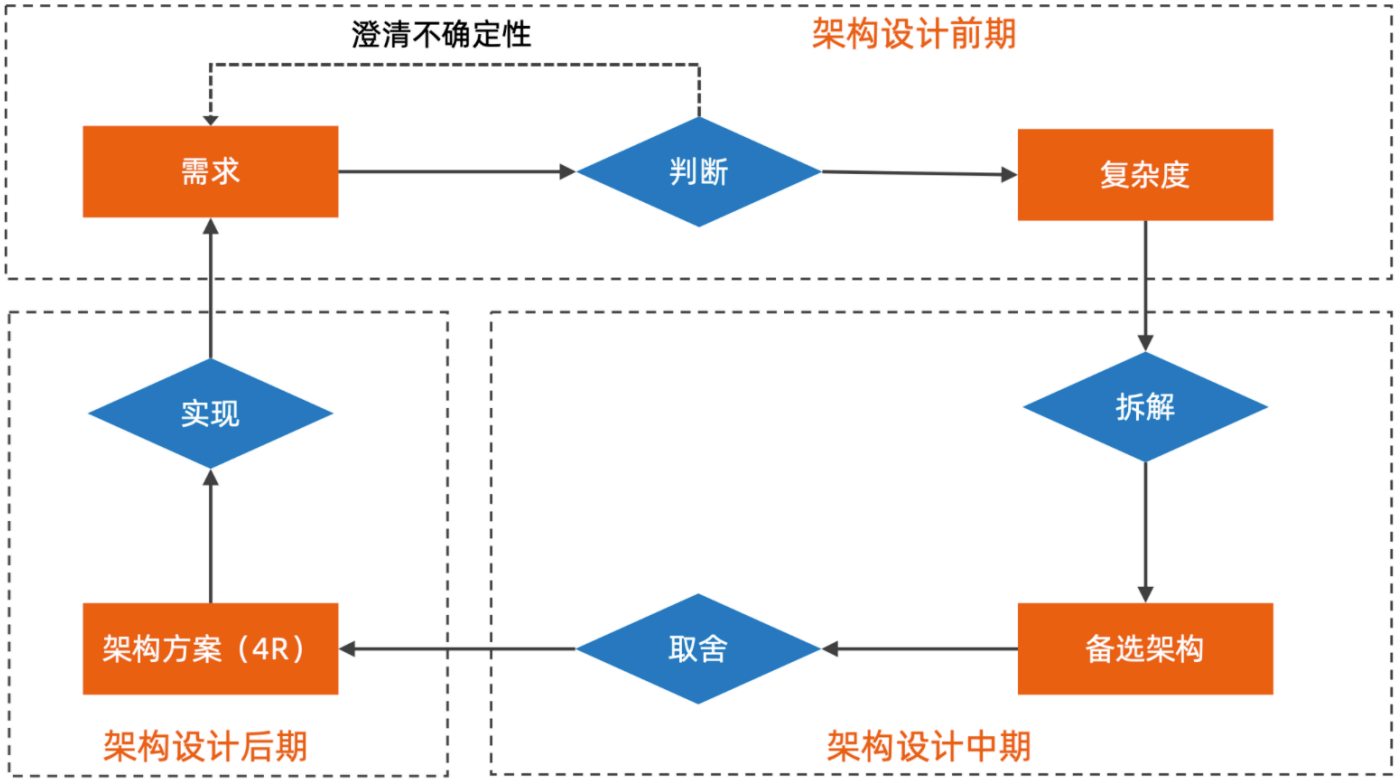
3. 半同步复制：服务器1收到用户请求后，选一个服务器并将相同的请求发给它，当用户请求被这两个服务器执行成功后，该请求就被认为是执行成功。之后，再异步的发给剩余的服务器做复制。
4. 多数复制：服务器1收到用户请求后，选多数（总服务器个数的1/2，例如total = 7个/3个，那多数=4 个/2个）服务器并将相同的请求发给它们，该请求就被认为是执行成功。之后，再异步的发给剩余的服务器做复制。 因为是要发给多数，所以一般都是用一致性的算法（e.g Paxos算法，Raft算法），就可以保证数据强一致性。

3. 状态决策
    1. 独裁式：
        1. 有一个独立的决策者来收集信息然后做决策。
        2. 但如果决策者挂了，整个集群就挂了，需要保证决策者的高可用。
        3. 上报者之间的复制并没有用很强的一致性算法来保证数据的一致性。
    2. 协商式：常应用于主备架构中。数据一致性弱：因为两个互相复制的话，会导致数据复制失败或者覆盖。
    3. 民主式/选举式：
        1. 决策过程复杂，决策逻辑复杂
        2. 可用性最高，数据一致性最强
        3. 可能出现“脑裂”问题（由于网络原因，分成了多派，每个派有自己的leader），可以采用 quorum 来控制（即多数来决定leader，e.g 5个节点，至少要有3个节点才能构成一个集群，然后选leader，如果只有2个节点就不要选了）

4. 如何全面提升架构设计质量：从低成本、安全性、架构质量属性（可测试性、可维护性、可观测性）来考虑。

3. 模块三

主要讲了架构设计的前期、中期、后期需要做啥。



1. 前期：对不同的利益关系人以及他们各自的诉求进行分析，并对诉求进行优先级排序，然后结合业务需求的核心场景，识别可能的复杂度，最终明确需求的复杂度。
2. 中期：备选架构方案设计、评估和选择。
3. 后期：基于选出的备选架构，继续进行详细的架构设计，并开始架构设计文档的写作。

4. 模块四

主要介绍了不同的存储架构模式 和 如何设计存储架构。

1. 数据库存储架构：
    1. 数据库读写分离：主机负责读写操作，从机只负责读操作，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。带来的复杂度：复制延迟，任务分解。
    2. 数据库分库分表
        1. 分库：把不同类型的数据分别各自放在一个DB上。带来的问题：Join 问题 和 事物问题。
        2. 分表：
            1. 垂直拆分: 将某个表按列拆分，优化单机处理性能，常见于2B领域超多列的表拆分。
            2. 水平拆分: 将某个表按行拆分，提升系统处理性能，常见于2C领域超多行的表拆分。
    3. 数据库分布式事务算法：2PC，3PC
2. 复制架构
    1. 主备复制：通过冗余来提升可用性。主机负责所有读+写请求，备机只负责数据备份，不接受任何请求。主机发生故障后，此时业务是中断的，需要人工将读写请求切换到备机后，才能继续进行业务请求处理。

2. 主从复制：主机负责所有写请求+部分读请求，备机既要负责读请求，又要负责数据备份。
3. 双击切换架构
    1. 主备切换：自动实现切换功能，不再需要人工干预
    2. 主从切换：整体和主备切换类似，差异点在于“切换阶段”，只有主机提供读写服务，如果此时出现大量读写，那么主机性能有风险。
4. 集群选举架构：集群中有很多台从/备机，从里面选一个来做新主机。

3. 分片架构：本质是通过叠加更多服务器来提升 写性能 和 存储性能。核心：数据按照什么规则分片，路由规则。分片架构高可用方案有 独立备份 和 互相备份。
4. 分区架构：本质是通过冗余 IDC 来避免城市级别的灾难，并提供就近访问。备份策略有 集中式、互备式、独立式，各个策略的成本、可扩展和复杂度均不同。
5. 如何设计存储架构：3步骤
    1. 性能需求估算：用户量预估，用户行为建模，存储性能需求计算
    2. 存储系统选择：选择存储架构，选择具体的存储系统
    3. 设计存储方案

5. 模块五

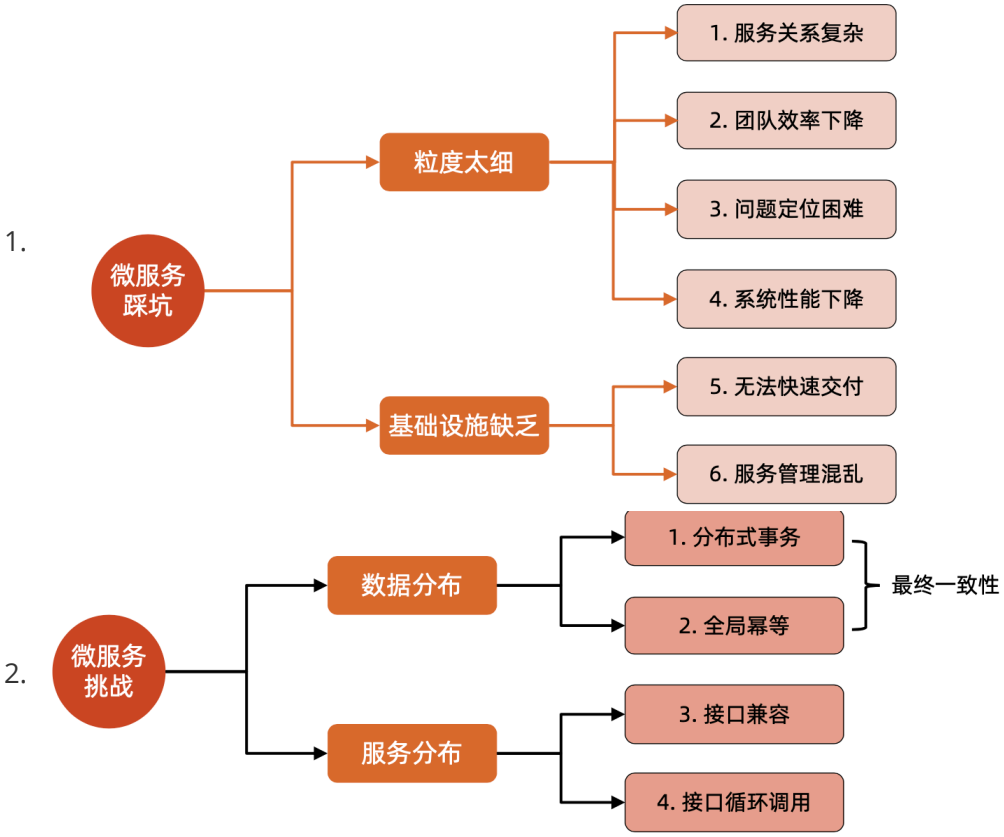
主要讲高性能高可用的计算架构设计，包括多级缓存、负载均衡、接口高可用等。

1. 多级缓存架构：
    1. Cache 的技术本质：空间换时间，凡是位于性能相差较大的两种系统之间，用于协调两者性能差异的结构，均可称之为 Cache。
    2. 缓存设计框架：3W1H --- 存啥，存多久，存在哪里，如何存
    3. 常见多级缓存架构有 5级缓存，4级缓存，3级缓存
2. 分布式缓存架构
    1. 2种模式：
        1. 数据缓存：计算系统优先从缓存系统中读，如果读不到，才从存储系统里面读，并且更新缓存系统中读数据
        2. 结果缓存：计算系统往缓存系统中写入计算结果，Client从缓存中读取该结果。
    2. 数据缓存架构一致性设计：读操作和写操作都会带来复杂度，复杂度本质是需要跨越缓存系统和存储系统实现分布式事务。
    3. 缓存架构通用3类问题：
        1. 缓存穿透
        2. 缓存雪崩：本质是请求太多
        3. 缓存热点
3. 负载均衡架构：整体架构，技术剖析，通用算法，常见业务负载均衡的技巧。
4. 接口高可用：架构决定系统质量上限，代码决定系统质量下限。常用策略有限流、排队、降级、熔断。

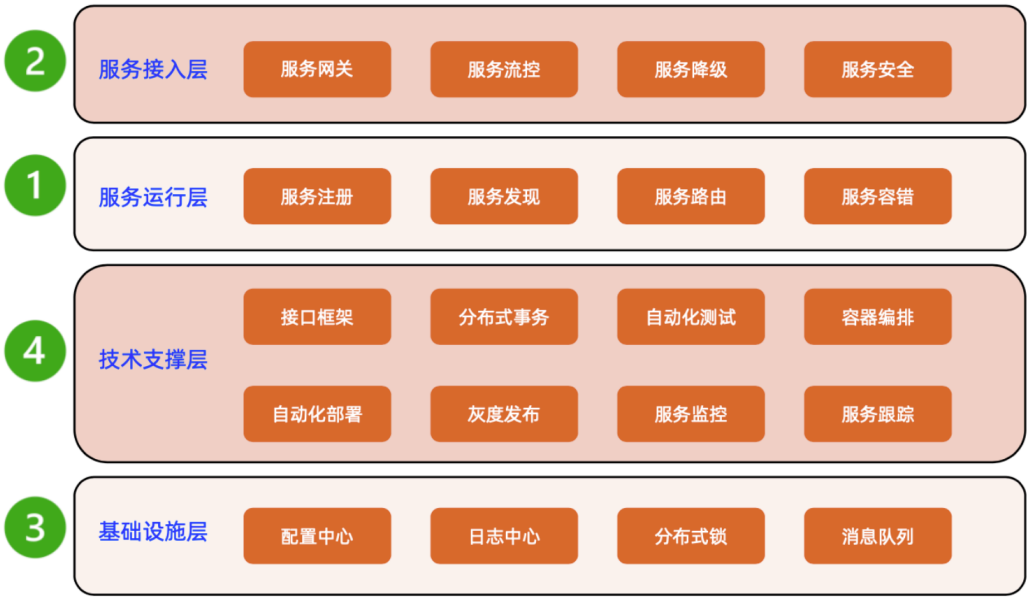
6. 模块六

主要讲了微服务架构和中台。

1. 微服务架构的历史，以及与 SOA 的对比。
2. 微服务架构的6大陷阱和4大挑战



3. 基础设施的全貌和优先级：服务运行层是微服务的灵魂，其他层只有在微服务本身的节点变多后才会显得重要



4. 拆分技巧：考虑3个方面：

1. 拆分方式
    1. 按业务拆分微服务
    2. 按质量拆分微服务
2. 基础设施要求
    1. 搭建完善基础设施
    2. 搭建核心基础设施，之后再逐步演进
3. 落地方式
    1. 一步到位
    2. 逐步落地

5. 中台：
    1. 业务中台：是用来应对业务复杂度，是将企业内多个相似业务的通用业务能力沉淀到平台，以减少重复建设，提升业务开发效率的一种架构模式。
    2. 数据中台：是将企业所有业务的数据沉淀到同一平台，支持业务间数据打通以及数据复用，提升企业运营效率的一种架构模式。

7. 模块七

主要讲高可用架构三大核心原理和灾备架构设计。

1. 高可用架构三大核心原理
    1. FLP 不可能原理：在基于消息传递的异步通信的场景下，即使只有一个进程失败，也没有任何确定性的算法能保证其他非失败的进程能达到一致性。
        1. FLP 三大限定条件：确定性协议，异步网络通信，所有存活节点
        2. FLP 可能的系统：SL 系统，SF 系统，LF 系统。S=Safety，L=Liveness， F=Fault Tolerance。
    2. CAP 定理：分布式数据存储系统不可能同时满足一致性、可用性和分区容忍性
    3. BASE 理论：BASE 是指基本可用(Basically Available)、软状态(Soft State)、最终一致性(Eventual Consistency)，核心思想是即使无法做到强一致性 (CAP 的一致性就是强一致性)，但应用可以采用适合的方式达到最终一致性。
2. FMEA：FMEA = Failure mode and effects analysis，故障模式与影响分析（又称为失效模式与后果分析、失效模式 与效应分析、故障模式与后果分析等）。用 FMEA 来检查设计出来的架构方案，去发现是否有需要改进的地方。
    1. 有11个分析维度。
3. 业务级灾备架构设计
    1. 同城多中心：可双中心或三中心。本质是一个逻辑机房。
    2. 跨城多中心：在邻近城市或者远端城市部署
    3. 跨国数据中心：全球部署；合规和监管。各国对数据监管对要求不同，比如对隐私保护的要求等；区域用户分区；不会做异地多活。
4. 异地多活：3种模式：
    1. 业务定制型：按照业务的优先级进行排序，优先保证核心业务异地多活。基于核心业务的流程和数据，设计

1. 业务定制型：按照业务的优先级进行排序，优先保证核心业务异地多活；基于核心业务的流程和数据，设计定制化的异地多活架构。比如电商业务里面做了一个异地多活架构，不一定能将其应用在社交业务上面。
2. 业务通用型：通过配套服务来支持异地多活，无需按照业务优先级排序来挑选某些业务实现异地多活，只需要判断当前业务是否能异地多活，如果能，配套服务就可以支撑。
3. 存储通用型：由于有的业务如果不满足 BASE 理论，就不能通过业务通用型架构实现异地多活。所以，引入与业务本身没有关系的存储通用型架构：采用本身已经支持分布式一致性的存储系统，架构天然支持异地多活。

5. 业务定制型异地多活
    1. 1个原理：CAP
    2. 3大原则：只保证核心业务，只能做到最终一致性，只能保证绝大部分用户
    3. 4个步骤落地：如果Top 3互相冲突，就尝试着保 Top 2，如果不行就保 Top 1。
        1. 业务分级：将业务按照某个维度进行 优先级排序，优先保证 TOP3 业务异地多活。
        2. 数据分类：分析 TOP3 中的每个业务的关键数据的特点，将数据分类
        3. 数据同步：针对不同的数据分类设计不同的数据同步方式
        4. 异常处理：针对极端异常的情况，考虑如何处理，可以是技术手段或非技术手段

8. 模块八

如果开源或者买到的已有的技术不能完全满足自己的业务，那么就要自己开发自己的轮子。他山之石可以攻玉。

1. 单机高性能网络模型
    1. 传统网络模型：PPC 和 prefork，TPC 和 prethread
    2. Reactor 网络模型：基于多路复用的事件响应网络编程模型
        1. 模式1 - 单 Reactor + 单进程/单线程
        2. 模式 2 - 单Reactor + 多线程
        3. 模式 3 - 多 Reactor + 多进程/线程
    3. Proactor 网络模型
2. 基于ZK实现高可用架构
    1. ZK 技术本质：At the heart of ZooKeeper is an atomic messaging system that keeps all of the servers in sync.
    2. ZK 实现主备切换架构
    3. ZK 实现集群选举：3种方案：最小节点获胜，抢建唯一节点，法官判决。
3. 复制集群架构设计
    1. Redis Sentinel： 基本架构 = Sentinel 节点集群 + Redis 节点集群
        1. Sentinel 架构模式有 双节点，三节点 和分离部署 3种。
    2. MongoDB Replication：基本架构：Primary 处理所有 Write 请求，Secondary 可以处理 Read 请求，或者只复制数据。新节点同步流程。架构技巧1 - Read Preference，技巧2 - Arbiter（Arbiter 节点是一个特殊的节点，只投票，不复制数据）
4. 分片架构设计

4. 分片架构设计
    1. Elasticsearch 集群分片架构设计
        1. 部署架构模式有：Master 和 Data 混合部署；Master 和 Data 分离部署；Coordinating 分离部署；Cross Cluster Replication。
    2. Redis Cluster 分片架构设计：数据分布和路由，每个节点都有所有key的分布信息，Client连接任意节点，由节点用 move指令来告诉实际的数据位置在另一个节点给 Client，然后Client 发起一个新请求给另一个节点。
    3. MongoDB/HDFS 分片架构设计
5. 常见集群算法：
    1. Gossip 协议：3种传播模式 -- Direct Mail，Anti-Entropy，Rumor mongering。
    2. Bully 选举算法：当一个进程发现协调者(或 Leader)不再响应请求时，就判定其出现故障，于是它就发起选举，选出新的协调者，即当前活动进程中进程号 最大/最小者。
    3. Raft 选举算法：Raft 是分布式一致性的算法，比Paxos 更容易理解。

9. 模块九

1. 架构重构：通过调整系统结构(4R，除了Rank)来修复系统质量问题而不 影响整体系统能力。目的是修复质量问题（性能、可用性、可扩展.....），不影响整体系统功能，架构本质没有发生变化，即并不会改变系统本身的能力。技巧有：
    1. 先局部优化、后架构重构
    2. 有的放矢，不要试图通过架构重构解决所有的问题，抓住关键问题
    3. 合纵连横，即说服利益干系人。
    4. 运筹帷幄，即将问题分类、排序，再逐一攻破。
2. 架构演进：通过设计新的系统架构(4R)来应对业务和技术的发展变化。架构演进是为了促进业务发展。
    1. 业务驱动的架构演进技巧：
        1. 主动演进：
            1. 做好预判：提前1年做好准备。
            2. 提前布局：团队和技术先行
        2. 被动演进
            1. 快速响应：熟悉什么就用什么
            2. 拿来主义：尽量用现成的方案
    2. 技术驱动的架构演进技巧
        1. 原则有：
            1. 新瓶旧酒：使用新的技术来解决老的问题或者老的复杂度，不要为了尝试新技术而演进
            2. 新技术要带来**典型**的价值才考虑演进
        2. 技巧：
            1. 说服老板：谈钱，别谈感情；谈竞争对手； 谈大环境
            2. 做好洞察，提前布局

3. IM架构设计之十万用户、百万用户、千万用户、亿级用户

3. MVC架构设计之千万用户，百万用户，千万用户，亿级用户

二。收获和感想

从纯技术方面，学到了如何从顶向下来考虑如何解决业务问题，而不像现在只是在底层扣代码的逻辑。拓宽了技术的知识面，习得了全新的mind set。

从软技能方面，学到了如何高效的和利益干系人沟通，如何写架构文档，如何话架构图等。

总的来说，这门课打开了新世界的大门，让我认识到了自己的不足，也更加明确了自己的成长方向，模块十给了很多关于软件工程师成长的建议，会细心采纳并认真实践。