架构设计的目的

> **架构设计的主要目的是为了解决软件系统复杂度带来的问题**

- 通过熟悉和理解需求，识别系统复杂性所在地方，然后针对这些复杂点进行架构设计
- 架构设计并不是要面面俱到，不需要每个架构都具备高性能、高可用、高扩展等特点，而是要识别出复杂点然后针对性解决问题
- 理解每个架构方案背后所需要解决的复杂点，然后才能对比自己业务复杂点，参与复杂点相似的方案

> NOTE：
>
> 所谓软件架构就是为两件事服务：业务架构和业务量级，这应该算是“软件系统复杂度带来问题”的具体化。
>
> 业务架构和业务量级都是从每个具体项目的实际应用场景中提炼出来的。
>
> 业务架构是对业务需求的提炼和抽象，开发软件必须要满足业务需求，否则就是空中楼阁。软件系统业务上的复杂度问题，可以从业务架构的角度切分工作交界面来解决。设计软件架构，首先是要保证能和业务架构对的上，这也是从业务逻辑转向代码逻辑的过程，所以软件架构的设计为开发指明的方向。另外架构设计也为接下来的开发工作分工奠定了基础。
>
> 业务量级表现在存储能力、吞吐能力和容错能力等，主要是软件运维期业务的复杂度。做软件架构设计，是要保证软件有能力托起它在业务量级上的要求，如果软件到运行使用期废了，前面所有的工作都付诸东流了。不同的业务量级，对应的软件的架构复杂度是不同的，所以对于不同的项目，业务量级不同，架构设计也不同。
>
> 做业务架构必须与其面向的实际应用场景相匹配，由于每个产品或项目的业务场景均有所不同，所以每次做新的软件开发前，必须先设计软件架构，试图不经分析直接套用先前的架构方案，十有八九会让当前系统在某个点上报出大问题导航推翻重来，更不要说直接拿别人的现成架构方案。
>
> 所以每个软件在开发前，都要结合自己的应用场景设计适配自身的软件架构，现成的架构方案只能借鉴，不能直接套用。
>
> 另外，由于业务架构和业务量级也会不断调用或长大，软件架构也不是一劳永逸，会随业务不断调整。

复杂度来源

高性能

软件系统中高性能带来的复杂度主要体现在两方面：

1. 单台计算机内部为了高性能带来的复杂度 ---- 单机复杂度；
    - 多进程
    - 多线程
    - 进程间通信
    - 多线程并发
2. 多台计算机集群为了高性能带来的复杂度 ---- 集群复杂度；
    - 任务分配
    - 任务分解

高可用

> 高可用定义 ：
>
> 系统**无中断**地执行其功能的能力，代表系统的可用性程序，是进行系统设计时的准则之一。

通过“冗余”来实现高可用。

复杂度介绍，举例：

1. 计算高可用
    - 需要增加任务分配器，需要综合考虑性能 ，成本，可维护性，可用性等因素
    - 任务分配器与业务服务器间的连接管理
    - 任务分配器需要增加分配算法
2. 存储高可用
    - 无论是正常情况下的传输延迟，还是异常情况下的传输中断，都会导致系统数据在某点是不一致的，而数据不一致又会导致业务问题
    - 存储高可用的难点不在于如何备份数据，而在于**如何减少或者规避数据不一致对业务造成的影响**
    - 存储高可用不可能同时满足“CAP”，最多满足其中两个，这就要求做架构设计时结合业务进行取舍

无论是计算高可用还是存储高可用，其基础都是“状态决策”，即系统需要能够判断当前的状态是正常还是异常，如果出现异常就采取行动来保证高可用。

几种常用的决策方式：

1. 独裁式
2. 协商式
3. 民主式

> NOTE：
>
> 如何做到高可用？
>
> 主要技术手段是服务与数据的冗余备份与失效转移。同一服务组件部署在多台服务器上；数据存储在多台服务器上互相备份。
>
> 通过上述技术手段，当任何一台服务器宕机或出现各种不可预期的问题时，就将相应的服务切换到其他可用的服务器上，不影响系统的整体可用性，也不会导致数据丢失。
>
> 高可用的解决方法不是解决，而是减少或者规避，这其实也是一个取值的过程 。

可扩展性

> 可扩展性指系统为了应对将来需求变化而提供的一种扩展能力，当有新的需求出现时系统不需要或者仅需要少量修改就可以支持，无须整个系统重构或者重建。

设计具备良好可扩展性的系统，有两个基本条件：正确预测变化，完善封装变化。

而对应复杂度也来自这块块，分析：

1. 预测变化
    - 不能每个设计点都考虑可扩展性
    - 不能完全不考虑可扩展性
    - 所有预测都存在出错的可能性
2. 应对变化
    - 系统需要拆分出变化层和稳定层

- 需要设计变化层和稳定层之间的接口

> NOTE:
>
> 一个具备良好可扩展性的架构设计应用符合开闭原则：对扩展开放，对修改关闭。
>
> 衡量一个软件系统具备良好可扩展性主要表现但不限于：（1）软件自身内部方面。在软件系统实现新增业务功能时，对现有系统功能影响较少，即不需要对现有功能作任何改动或者很少改动。（2）软件外部方面。软件系统本身与其他存在协同关系的外部系统之间存在松耦合关系，软件系统的变化对其他软件系统无影响，其他软件系统和功能不需要进行改动。
>
> **面向对象思想、设计模式**都是为了解决可扩展性而出现的方法与技术。
>
> 设计具备良好可扩展性的系统，有两个思考角度：（1）从业务维度。对业务深入理解 ，对可预计的业务变化进行预测。（2）从技术维度。利用扩展性好的技术，实现对变化的封装。

低成本 、安全和规模

低成本

低成本给架构设计带来的主要复杂度体现在，往往只有“创新”才能达到低成本目标。包括开创一个全新技术领域，也包括引入新技术。

安全

- 功能安全，如XSS攻击、SQL注入
- 架构安全，如防火墙、DDOS攻击

规模

> **"量变引起质变"**，当数量超过一定的阈值后，复杂度会发生质的变化。

- 功能越来越多，导致系统复杂度指数级上升
- 数据越来越多，系统复杂度发生质变

架构设计三原则

> 1. **合适原则** - “合适优于业界领先”
>
>   真正优秀的架构都是在企业当前人力、条件、业务等各种约束下设计出来的，能够合理地将资源整合在一起并发挥出最大功效，并且能够快速落地。
> 2. **简单原则** - “简单优于复杂”
> 3. **演化原则** - “演化优于一步到位”
>
>   软件架构需要根据业务发展不断变化。
>
>   首先，设计出来的架构要满足当时的业务需要；
>
>   其次，架构要不断地实际应用过程中迭代，保留优秀的设计，修复有缺陷的设计，改正错误的设计，去掉无用的设计，使得架构逐渐完善；
>
>   第三，当业务发生变化时，架构要扩展、重构、甚至重写；
>
> 架构师在进行架构设计时需要牢记这些原则，时刻提醒自己不要贪大求全，或者盲目照搬大公司的做法。应该认真分析当前业务的特点，明确业务面临的主要问题，设计合理的架构，快速落地以满足业务需要，然后在运行过程中不断完善架构，不断随着业务演化架构。
>
> 注：
>
> 合适原则第一考虑，优先满足业务需求；

> 简单原则第二考虑，挑选简单方案快速落地验证；
>
> 演进原则第三考虑，适当预测业务发展，感觉预测不准就不预测，等真出现问题时候演进即可；

架构设计流程

识别复杂度

> 以理解需求为前提，首要进行系统复杂度分析
>
> 将主要的复杂度问题列出来，然后根据业务、技术、团队等综合情况进行排序，优先解决当前面临的最主要的复杂度问题。
>
> 采用"排查法"分析

设计备选方案

> 对已存在技术和已验证过的架构模式非常熟悉，然后根据自己对业务的理解，挑选合适的架构模式进行组合，再对组合后的方案进行修改和调整。如高可用的主备方案、集群方案，高性能的负载均衡、多路复用，可扩展的分层、插件化等技术。
>
> 注意事项：
>
> - 备选方案的数量以3~5个为最佳。
> - 备选方案的差异要比较明显。
> - 备选方案的技术不要局限于已经熟悉的技术。
>   - 设计架构时，架构师需要将视野放宽，考虑更多可能性
>   - 避免"如果你手里一把锤子，所有的问题在你看来都是钉子"
> - 备选阶段关注的是技术选型，而不是技术细节。
>
> 误区：
>
> - 设计最优秀的方案
> - 只做一个方案
> - 备选方案过于详细

评估和选择备选方案

> 列出需要关注的质量属性点，然后分别从这些质量属性的维度去评估每个方案，再综合挑选适合当时情况的最优方案。
>
> 常见的方案质量属性点有：性能、可用性、硬件成本、项目投入、复杂度、安全性、可扩展性等。

详细方案设计

> 就是将方案涉及的关键技术细节给确定下来。

业界成熟架构模式

前面几章从各方面阐述了架构设计相关的理论和流程，包括架构设计起源、架构设计的目的、常见架构复杂度分析、架构设计原则、架构设计流程等，掌握这些知识是做好架构设计的基础。

在具体的实践过程中，为了更快、更好地设计出优秀的架构，除了掌握这些基础知识外，还需要掌握业界已经成熟的各种架构模式。大部分情况下，我们做架构设计主要都是基于已有的成熟模式，结合业务和团队的具体情况，进行一定的优化或者调整；即使少部分情况我们需要进行较大创新，前提也是需要对已有的各种架构模式和技术非常熟悉。

高性能架构模式

高性能数据库集群

读写分离

> 将访问压力分散到集群中的多个节点，但没有分散存储压力。
>
> 基本原理是将数据库读写操作分散到不同的节点上。
>
> 引入设计复杂度：主从复制延迟 和 分配机制
>
> 解决主从复制延迟的几种常见方法：
>
> 1. 写操作后的读操作，指定发给数据库主服务器
> 2. 读从机失败后再读一次主机
> 3. 关键业务读写操作全部指向主机，非关键业务采用读写分离
>
> 分配机制
>
> 1. 程序代码封装
> 2. 中间件封装

分库分表

> 既可以分散访问压力，又可以分散存储压力
>
> - 业务分库
>
>   按照业务模块将数据分散到不同的数据库服务器
>
>   复杂度：
>
>   - join
>   - 事务
>   - 成本
> - 分表
>   - 垂直分表
>
>   复杂度：表操作数据增加
>   - 水平分表
>
>   复杂度：
>
>   1. 路由 --> 范围路由、hash路由、配置路由
>   2. join操作
>   3. count操作
>   4. order by

高性能NoSQL

> 关系数据库存不足：
>
> - 存储的是行记录，无法存储数据结构
> - schema扩展不方便
> - 全文搜索功能比较弱

> NoSQL != No SQL 而是 NoSQL = Not Only SQL
>
> 常见NoSQL方案：
>
> - K-V存储，如Redis
> - 文档数据库，如MongoDB
> - 列式数据库，如HBase
> - 全文搜索引擎，如ElasticSearch

高性能缓存架构

> 基本原理是将可能重复使用的数据放在内存中，一次生成，多次使用，避免每次使用都去访问存储系统。
>
> 能带来性能的大幅提升
>
> 复杂度：
>
> - 缓存击穿：缓存中没有但数据库中有的数据（一般是缓存时间到期）
> - 缓存穿透：缓存和数据库中都没有的数据，而用户不断发起请求
> - 缓存雪崩：缓存中数据大批量到过期时间
>   - 更新锁
>   - 后台更新
>   - 双KEY策略
> - 缓存热点
>   - 复制多份缓存副本

单服务器高性能模式

> 做到高性能是一件很复杂很有挑战的事情，软件系统开发过程中的不同阶段都关系着高性能最终是否能够实现。站在架构师的角度，需要特别关注高性能架构的设计。
>
> 主要集中在两方面：
>
> - 尽量提升单服务器的性能，将单服务器的性能发挥到极致
> - 如果单服务器无法支撑性能，设计服务器集群文案

PPC与TPC

- PPC，即Process Per Connection，又称做Apache模型，是通过多进程来实现业务并发。每次有新连接就新建一个进程专门去处理该连接的请求
- TPC，即Thread Per Connection，是一种多线程并发处理模型。具体操作就是每次有新的连接就新建一个线程去专门处理这个连接的请求。

Reactor与Proactor

- Reactor
  - I/O多路复用统一监听事件，收到事件后分配 给某个进程 ，属于非阻塞同步网络模型
  - 核心组成包括Reactor和处理资源池(进程池或线程池)，其中Reactor负责监听和分配事件，处理资源池负责处理事件
  - 三种典型实现方案：
    1. 单reactor，单进程/线程
       - reactor对象通过select监控连接事件，收到事件后通过dispatcher进行分发
       - 若是连接建立事件，则由Acceptor处理，通过accept接受连接并创建一个Handler来处理连接后续的各种事件

> - 若非连接建立事件，则Reactor会调用连接对应的Handler来进行响应
>
> 适用于业务处理非常快速的场景，redis就是单reactor单进程

2. 单reactor，多线程

> - Handler只负责响应事件，不进行业务处理；Handler通过read读取到数据后人发给Processor进行业务处理
> - Processor会在独立的子线程中完成真正的业务处理，然后将响应结果发给主进程的Handler处理；Handler收到通过send将响应结果返回给client

3. 多Reactor，多进程/线程

> - 父进程中mainReactor对象通过select监控连接建立事件，收到事件后通过Acceptor接收，将新的连接分配给某个子进程
> - 子进程subReactor将mainReactor分配的连接加入连接队列进行监听，并创建一个Handler用于处理连接的各种事件
> - 当有新事件发生时，subReactor会调用连接对应的Handler来进行响应
> - Handler完成read-->业务处理-->send的完整业务流程
>
> Nginx：多Reactor多进程
>
> Netty/Memcache：多Reactor多线程

- Proactor

> Reactor，可以理解为“来了事件我通知你，你来处理”
>
> Proactor，可以理解为“来了事件我来处理，处理完了我通知你”
>
> “我” --> 操作系统内核
>
> “事件” --> 有新连接、有数据可读、有数据可写等I/O事件
>
> “你” --> 业务程序代码

高性能负载均衡

> 单服务器无论如何优化，无论采用多好硬件，总会有一个性能天花板，当单服务器的性能无法满足业务需求时，就需要设计高性能集群来提升系统整体的处理性能。
>
> 高性能集群的复杂性主要体现在需要增加一个任务分配器，以及为任务选择一个合适的任务分配算法。

分类及架构

> - 分类
>
> 1. DNS负载均衡，一般用来实现地理级别的均衡
> 2. 硬件负载均衡，如F5
> 3. 软件负载均衡，如LVS，Nginx
> - 典型架构
>
> 1. DNS --> 地理级别
> 2. 硬件 --> 集群级别
> 3. 软件 --> 机器级别

算法

> 根据算法期望达到的目的，可分为以下几类：
>
> 1. 任务平分类
> 2. 负载均衡类
> 3. 性能最优类

4. Hash类

常用算法有：

1. 轮询
2. 加权轮询
3. 负载最低优先
4. 性能最优类
5. Hash类

高可用架构模式

CAP

- Consistence 一致性

  > 对某个指定客户端来说，读操作保证能够返回最新的写操作结果

- Availability 可用性

  > 非故障的节点在合理的时间内返回合理的响应（不是错误和超时的响应）

- Partition Tolerance 分区容错性

  > 当出现网络分区后,系统能够继续"履行职责"

> 因为网络本身无法做到100%可靠,有可能出故障,所以分区是一个必须必然现象。故只能选择CP或者AP架构。
>
> **CAP关注的粒度是数据，而不是整个系统。**
>
> “C与A之间的取舍可以在同一系统内以非常细小的粒度反复发生，而每一次的决策可能因 具体的操作，乃至因为牵涉到特定的数据或用户而有所不同。”

FMEA方法

> FMEA,(Failure mode and effects analysis，故障模式与影响分析)
>
> 具体分析方法：
>
> - 给出初始的架构设计图
> - 假设架构中某个部件发生故障
> - 分析此故障对系统功能造成的影响
> - 根据分析结果，判断架构是否需要进行优化
>
> FMEA分析表：
>
> 1. 功能点
> 2. 故障模式，即故障现象
> 3. 故障影响
> 4. 严重程度
> 5. 故障原因
> 6. 故障概率
> 7. 风险程度
> 8. 已有措施
> 9. 规避措施
> 10. 解决措施
> 11. 后续规划

高可用存储架构

> 存储高可用方案的本质都是通过将数据复制到多个存储设备，通过数据冗余的方式来实现高可用，期复杂性主要体现在如何应对复制延迟和中断导致的数据不一致问题。
>
> 因此，对任何一个高可用存储方案，需要从以下几方面去进行思考和分析：
>
> - 数据如何复制?
> - 各个节点的职责是什么?
> - 如何应对复制延迟?
> - 如何应对复制中断?

双机架构

> - 主备
> - 主从
> - 双机切换
>
>> 问题：主备与主从存在两个共性问题：主机故障后无法进行写操作、若主机无法恢复，需要人工指定新主机角色
>>
>> 要实现一个完善切换方案，必须考虑的关键设计点：
>>
>> - 主备间状态判断
> - 切换决策
> - 数据冲突解决
>>
>> 常见架构：
>>
>> - 互连式
>>
>>> 在主备复制架构基础上，主机和备机多了一个“状态传递”的通道，该通道就是用来传递状态信息。
>>>
>>> 缺点：
>>>
>>> - 虽可以通过增加多个通道来增强状态传递的可靠性，但只是降低通道故障概率，并未从根本上解决问题。通道越多，后续的状态决策会更加复杂。
>>
>> - 中介式
>>
>>> 指在主备两者之外引入第三方中介，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。
>>>
>>> 关键点是如何实现中介本身的高可用。可通过ZK、Keepalived实现高可用。
>>>
>>> 见MongoDB
>>
>> - 模拟式
>>
>>> 批主备机之间并不传递任何状态数据，而是备机模拟成一个客户端，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。
>
> - 主主复制
>
>> 指两台机器都是主机，互相将数据复制给对方，客户端可以任意挑选其中一台机器进行读写操作。
>>
>> 复杂性表现在：如果采用主主复制架构，必须保证数据能够双向复制，而很多数据是不能双向复制的。

集群和分区

- 数据集群
    1. 数据集中集群

        > 如一主多从、一主多备，数据都只能往主机中写，而读操作可以参考主备、主从架构进行灵活多变。
        >
        > 问题：
        >
        > - 主机如何 将数据复制给备机
        > - 主机故障后，如何决定新的主机
        >
        > 目前开源的数据集中集群以Zookeeper为典型，其通过ZAB算法来解决上面问题。

    2. 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；
        >
        > 复杂度在于如何将数据分配到不同的服务器上，需要考虑均衡性、容错性、可伸缩性。
        >
        > 常见有的HDFS和ElasticSearch.

- 数据分区

    > 前面的存储高可用架构都是基于硬件故障的场景去考虑和设计的，主要考虑部分硬件可能损坏的情况 下系统如何处理，但对于大的灾难事故，有可能所有的硬件全部故障。因此需要基于地理级别的故障来设计高可用架构。
    >
    > 数据分区指将数据按照一定的规则进行分区，不同分区分布在不同的地理位置上，每个分区存储一部分数据，通过这种方式来规避地理级别的故障所造成的影响。
    >
    > 设计一个良好的数据分区架构，需要考虑：
    >
    > 1. 数据量
    > 2. 分区规则
    > 3. 复制规则
    >   - 集中式
    >   - 互备式
    >   - 独立式

高可用计算架构

> 计算高可用的主要设计目标是当出现部分硬件损坏时，计算任务能够继续正常运行。
>
> 计算高可用的本质是通过冗余来规避部分故障的风险，单台服务器是无论如何都达不到该目标，所以设计思想就是：通过增加更多服务器来达到计算高可用。
>
> 而设计复杂度主要体现在任务管理方面，即当任务在某台服务器上执行失败后，如何将任务重新分配到新服务器进行执行。
>
> 关键点有两点：
>
> 1. 哪些服务器可以执行任务
>   - 每个服务器都可以执行
>   - 只有特定服务器可以执行
> 2. 任务如何重新执行
>   - 对于已经分配的任务，即使执行失败也不做任何处理，系统只需要保证新的任务能够分配至非故障服务器上即可
>   - 由一个任务管理器来管理需要执行的计算任务，服务器执行完任务后，需要向任务管理器反馈任务执行结果，任务管理器根据任务执行结果来决定是否需要将任务重新分配到另外服务器上执行

常见计算高可用架构：

1. 主备
2. 主从
3. 集群

业务高可用保障

异地多活

> 异地多活架构的关键点就是异地、多活，其中异地就是指地理位置上不同的地方；多活就是指不同地理位置上的系统都能够提供业务服务，这里的“活”是活动、活跃的意思。
>
> 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。
>
> 复杂度：
>
> - 系统复杂度会发生质的变化，需要设计复杂的异地多活架构。
> - 成本会上升，毕竟要多在一个或者多个机房搭建独立的一套业务系统。

架构模式

1. 同城异区

   > 指的是将业务部署在同一个城市不同区的多个机房。
   >
   > 结合复杂度、成本、故障发生概率来综合考虑，同城异区是应对机房级别故障的最优架构。

2. 跨城异地

   > 指的是业务部署在不同城市的多个机房，而且距离最好要远一些。例如，将业务部署在北京和广州两个机房。
   >
   > 距离增加也带来一些问题，如：
   >
   > 1. 两个机房的网络传输速度会降低
   > 2. 中间传输的各种不可控因素较多
   >
   > 跨城异地距离较远带来的网络传输延迟问题，给异地多活架构设计带来了复杂性，如果要做到真正意义上的多活，业务系统需要考虑部署在不同地点的两个机房，在数据短时间不一致的情况下，还能够正常提供业务。这就引入了一个看似矛盾的地方：数据不一致业务肯定不会正常，但跨城异地肯定会导致数据不一致。
   >
   > 如何解决这个问题呢？重点还是在“**数据**”上，即根据数据的特性来做不同的架构。如果是强一致性要求的数据，例如银行存款余额、支付宝余额等，这类数据实际上是无法做到跨城异地多活的。
   >
   > 而对数据一致性要求不那么高，或者数据不怎么改变，或者即使数据丢失影响也不大的业务，跨城异地多活就能够派上用场了。例如，用户登录（数据不一致时用户重新登录即可）、新闻类网站（一天内的新闻数据变化较少）、微博类网站（丢失用户发布的微博或者评论影响不大），这些业务采用跨城异地多活，能够很好地应对极端灾难的场景。

3. 跨国异地

   > 相对于跨城异地，距离更远，数据同步延迟会更长。
   >
   > 适用场景：
   >
   > - 为不同地区用户提供服务
   > - 只读类业务做多活动

设计技巧

1. 保证核心业务的异地多活
2. 保证核心数据最终一致性
3. 采用多种手段同步数据

> 避免只使用存储系统的同步功能，可以将多种手段配合存储系统的同步来使用，甚至可以不采用存储系统的同步方案，改用自己的同步方案。

4. 只保证绝大部分用户的异地多活

NOTE：核心思想就是，采用多种手段，保证绝大部分用户的核心业务异地多活

设计步骤

1. 业务分级

> 按照一定标准将业务进行分级，挑选出核心的业务，只为核心业务设计异地多活，降低方案整体复杂度和实现成本 。
>
> 常见标准：
>
> - 访问量大的业务
> - 产生大量收入的业务

2. 数据分类

> 挑选出核心业务后，需要对核心业务相关数据进一步分析，目的在于识别所有的数据及数据特征。
>
> 常见分析维度：
>
> - 数据量
> - 唯一性
> - 实时性
> - 可丢失性
> - 可恢复性

3. 数据同步

> 确定数据特点后，可以根据不同的数据设计不同的同步方案。
>
> 常见方案有：
>
> - 存储系统同步
> - 消息队列同步
> - 重复生成

4. 异常处理

> 在出现部分数据异常时，系统将采取什么措施来应对。
>
> 常见措施：
>
> - 多通道同步
> - 同步和访问结合
> - 日志记录
> - 用户补偿

应对接口级故障

> 因系统压力太大、负载太高，导致无法快速处理业务请求，由此引发更多的后续问题。

故障原因：

- 内部原因，如程序bug、db慢查询等
- 外部原因，如黑客攻击、促销活动、第三方系统异常等

NOTE：解决接口级故障核心思想是，优先保证核心业务和优先保证绝大部分用户。

常见方式：

1. 降级

   > 丢车保帅，优先保证核心业务。

2. 熔断

   > 与降级区别是，降级的目的是应对系统自身的故障，而熔断是目的是应对依赖的外部系统故障。

3. 限流

   > 限流是从用户访问压力的角度来考虑如何应对故障。只允许系统能够承受的访问量进来，超出系统访问能力的请求将被丢弃。
   >
   > 常见方式有：基于请求限流 和 基于资源限流

4. 排队

   > 排队是让用户等待一段时间。

可扩展架构模式

> 如何避免扩展时改动范围太大，是软件架构可扩展性设计的主要思考点。
>
> 可扩展的基本思想就是“拆”，将原来大一流的系统拆分成多个规范小的部分，扩展时只修改其中一部分即可，无需每个系统到处都改，通过这种方式来减少改动范围，降低改动风险。
>
> 常见拆分思路及对应系统架构：
>
> - 面向流程 --> 分层架构
> - 面向服务 --> SOA、微服务
> - 面向功能 --> 微内核架构

分层架构

- 常见分类有

  1. C/S、B/S架构

     > 划分维度是用户交互，将和用户交互的部分独立为一层，支撑用户交互的后台作为另一层。

  2. MVC、MVP架构

     > 划分维度是职责，将不同的职责划分到独立层，但各层依赖关系比较灵活。

  3. 逻辑分层架构

     > 划分维度是职责，与MVC架构、MVP架构不同点在于，逻辑分层架构中的导是自顶向下依赖的，如操作系统内核架构。

> 无论采用何种分层维度，分层架构设计最核心的一点就是需要保证各层之间的差异足够清晰，边界足够明显。

> 分层架构之所以能够较好地支撑系统扩展，本质在于隔离关注点，即每个层中的组件只会处理本层的逻辑。

SOA

> 三个关键概念：
>
> 1. 服务
>
>   所有业务功能都是一项服务
>
> 2. ESB，企业服务总线
>
>   SOA使用ESB来屏蔽异构系统，对外提供各种不同的接口方式，以此来达到服务间高效的互通。
>
> 3. 松耦合
>
>   目的是减少各个服务间的依赖和互相影响

微服务

Small，Lightweight，Automated

银弹or焦油坑？

- SOA与微服务比较
  1. 服务粒度

     > 整体上来说，SOA的服务粒度要粗些，而微服务则更细一些

  2. 服务通信

     > SOA采用ESB作为服务间通信的关键组件，负责服务定义、服务路由、消息转换、消息传递，总体上是重量级实现。微服务推荐使用统一的协议和格式，如Restful协议、RPC协议。

  3. 服务交付

     > SOA对服务的交付并没有特殊要求，因为SOA更多考虑的是兼容已有的系统；
     >
     > 微服务的架构理念要求“快速交付”，相应地要求采用自动化测试、持续集成、自动化部署等敏捷开发相关的最佳实践。若没有这些基础能力支撑，微服务规模一旦变大，部署的成本 呈指数上升。

  4. 应用场景、

     > SOA更加适合于庞大、复杂、异构的企业级系统；
     >
     > 微服务更加适合于快速、轻量级、基于web的互联网系统。

- 微服务陷阱
  1. 服务划分过细，服务间关系复杂
  2. 服务数量太多，团队效率急剧下降
  3. 调用链太长，性能下降
  4. 调用链太长，问题定位困难
  5. 没有自动化支撑，无法快速交付
  6. 没有服务治理，微服务数量多了后管理混乱

方法篇 - 微服务最佳实战应该如何去做？

服务粒度

针对微服务拆分过细问题，建议基于团队规模进行拆分，如“两个披萨”理论。

常见服务拆分方式：

1. 基于业务逻辑拆分

   > 最常见的一种拆分方式，将系统中的业务模块按照职责范围识别出来，每个单独的业务模块拆分为一个独立的服务。

2. 基于可扩展拆分

   > 将系统中的业务模块按照稳定性排序，将已经成熟和改动不大的服务拆分为稳定服务，将经常变化和迭代的服务拆分为变动服务。稳定的服务粒度可以粗一些，即使逻辑没有强关联的服务也可以放在同一个子系统中。
   >
   > 这样拆分主要是为了提升项目快速效率。

3. 基于可靠性拆分

   > 将系统中的业务模块按照优先级排序，将可靠性要求高的核心服务和可靠性要求低的非核心服务拆分开来，然后重点保证核心服务的高可用。

4. 基于性能拆分

   > 将性能要求高或者性能压力大的模块拆分出来，避免性能压力大的服务影响其他服务，常见拆分方式和具体的性能瓶颈有关，可拆分web服务、数据库、缓存等。如电商的抢购，性能压力最大的是入口的排队功能，可以将排除功能独立为一个服务。

基础设施

建议按照优先级来搭建基础设施：

1. **服务发现、服务路由、服务容错**：这是最基本的微服务基础设施。
2. **接口框架、API网关**：主要是为了提升开发效率，接口框架是提升内部服务的开发效率，API网关是为了提升与外部服务对接的效率。
3. **自动化部署、自动化测试、配置中心**：主要是为了提升测试和运维效率
4. **服务监控、服务跟踪、服务安全**：主要是为了进一步提升运维效率

基础设施篇

- 自动化测试

  > 通过自动化测试系统来完成绝大部分测试回归的工作。

- 自动化部署

  > 包括版本管理、资源管理、部署操作、回退操作等功能。

- 配置中心

  > 包括配置版本管理、增删改查配置、节点管理、配置同步、配置推送等功能。

- 接口框架

  > 需要统一的接口协议和接口数据格式。

- API网关

  > API网关是外部系统访问的接口，所有的外部系统接入系统都需要通过API网关，主要包括接入鉴权(是否允许接入)、权限控制（可以访问哪些功能）、传输加密、请求路由、流量控制等功能。

- 服务发现

- 服务路由

  > 需要从所有符合条件的可用微服务节点中挑选出一个具体的节点发起请求。

- 服务容错

  > 常见的服务容错包括请求重试、流控和服务隔离。

- 服务监控

  > 一旦发生故障，需要快速根据各类信息来定位故障，因此需要服务监控系统来完成微服务节点的监控。
  >
  > 通常情况下，服务监控需要搜集并分析大量的数据，因此建议做成独立的系统。

- 服务跟踪

  > 服务监控可以做到微服务节点级的监控和信息收集，但若需要跟踪某一个请求在微服务中的完整路径。

- 服务安全

  > 系统拆分为微服务后，数据分散在各个微服务节点上。从系统连接角度来说，任意微服务都可以访问所有其他微服务节点；但从业务角度来说，部分敏感数据或者操作，只能部分微服务可以访问，因此需要设计服务安全机制来保证业务和数据的安全性。
  >
  > 主要分为三部分：接入安全、数据安全、传输安全。

微内核

微内核架构，也被称为插件化架构，是一种面向功能进行拆分的可扩展性架构，通过用于实现基于产品（指存在多个版本、需要下载安装才能使用，与web-based相对应）的应用。

基本架构

包含两类组件：

- 核心系统core-system

  > 负责和具体业务功能无关的通用功能，如模块架构、模块间通信等；

- 插件模块plugin modules

  > 负责实现具体的业务逻辑

核心系统Core System功能比较稳定，不会因为业务功能扩展而不断修改，插件模块可以根据业务功能的需要不断扩展。微内核架构的本质就是将变化封装在插件里面，从而达到快速灵活扩展的目的，而又不影响整体系统的稳定。

设计关键点

微内核的核心系统设计的关键技术有：插件管理，插件连接和插件通信。

1. 插件管理

   插件系统需要知道当前有哪些插件可用，如何加载这些插件，什么时候加载插件，常见的实现方法是插件注册表机制。 核心系统提供插件注册表（可以是配置文件，也可以是代码，还可以是数据库），插件注册表含有每个插件模块的信息，包括它的名字、位置、加载时机（启动就加载，还是按需加载等）

2. 插件连接

插件连接指插件如何连接到核心系统。通常来说，核心系统必须制定插件和核心系统的连接规范，然后插件按照规范实现，核心系统按照规范加载即可。 常见的连接机制有OSGi（Eclipse使用）、消息模式、依赖注入（Spring使用），甚至使用分布式的协议都是可以的，比如RPC或者HTTP Web的方式。

3. 插件通信

   通信必须经过核心系统，因此核心系统需要提供插件通信机制。

具体实现

常见的有两种微内核具体实现：OSGi和规则引擎。

架构实战

如何判断技术演进方向?

面对层出不穷的新技术，我们应该采取什么样的策略？

需要跳出技术的范畴，从一个更广更高的角度来考虑问题，这个角度就是企业的业务发展。

技术发展的主要驱动力是业务发展。

互联网技术演进模式

互联网业务发展一般分为几个时期：初创期、发展期、竞争期、成熟期。

不同时期的差别主要体现在两个方面：复杂性、用户规模。

> 互联网业务发展第一个主要方向就是“业务越来越复杂”；第二个方向就是“用户量越来越大”，用户量增大对技术影响主要是性能与可用性要求越来越高。

互联网业务驱动技术发展的两大主要因素是复杂性和用户规模，而本质其实就是“量变带来的质变”。

> 应对业务质变带来的技术压力，不同时期有不同的处理方式，但不管什么样的方式，其核心目标都是为了满足业务“快”的要求。
>
> “根据趋势预测下一个转折点，提前做好技术上的准备，是对技术人员的高要求。”

互联网架构模板

存储层技术

- SQL
- NoSQL
- 小文件存储，如淘宝的商品描述、Facebook的用户图片
- 大文件存储

开发层和服务层技术

- 开发层技术
  1. 开发框架
  2. web服务器
  3. 容器
- 服务层技术

  降低系统间相互关系的复杂度

  1. 配置中心
  2. 服务中心

3. 服务总线系统

网络层技术

- 负载均衡
    1. DNS
    2. Nginx/LVS/F5
- CDN
- 多机房
- 多中心

用户层和业务层技术

- 用户层技术
    1. 用户管理
    2. 消息推送
    3. 存储云、图片云
- 业务层技术

> 业务层面对的主要技术挑战是“复杂度”，原因是系统越来越庞大、业务越来越多。
>
> 解决方案就是
>
> - “拆” - 化整为零、分而治之，将整体复杂性分散到多个子业务或者子系统里面去。
> - “合” - 按照“高内聚、低耦合”原则，将职责关系比较强的子系统合并，然后通过网关对外统一呈现。

平台技术

- 运维平台

> 四大职责：配置、部署、监控、应急 四化：标准化、平台化、自动化、可视化

- 测试平台

> 用例管理、资源管理、任务管理、数据管理

- 数据平台

> 数据管理、数据分析、数据应用

- 管理平台

> 核心职责就是权限管理

架构重构

相比全新的架构设计来说，架构重构对架构师的要求更高，主要体现在：

- 业务已经上线 ，不能停下来
- 关联方众多，牵一发动全身
- 旧架构的约束

因此，架构重构对架构师的综合能力要求非常高，业务上要求架构师能够说明产品经理暂缓甚至暂停业务来进行架构重构；团队上需要架构师能够与其他团队达成一致的架构重构计划和步骤；技术上需要架构师给出让技术团队认可的架构重构方案。

有的放矢

> 架构师需要透过问题表象看到问题本质，找出真正需要通过架构重构解决的核心问题，从而做到有的放矢，即不会耗费大量的人力和时间投入，又能够解决核心问题。

合纵连横

> - 要想真正推动一个架构重构项目启动，需要花费大量的精力进行游说和沟通。
>
>> 在沟通协调时，将技术语言转换为通俗语言，以事实说话，以数据说话，是沟通的关键!
>
> - 有的重构还需要和其他相关或者配合的系统的沟通协调
>
>> “换位思考、合作双赢、关注长期”

运筹帷幄

> 架构师在识别系统关键的复杂度问题后，还需要识别为了解决这个问题，需要做哪些准备事项 ，或者还要先解决哪些问题。
>
> “分段实施”，将要解决的问题根据优先级、重要性、实施难度等划分为不同的阶段，每个阶段聚集于一个整体的目标，集中精力和资源解决一类问题。

再谈开源项目

- 选
    1. 聚焦是否满足业务
    2. 聚焦是否成熟
    3. 聚焦运维能力
- 用
    1. 深入研究，仔细测试
    2. 小心应用，灰度发布
        > 可以先非核心业务试点
    3. 做好应急，以防万一
- 改
    1. 保持纯洁，加以包装
    2. 发明你要的轮子

App架构演进

- Web App
    > 主要解决“快速开发”和“低成本”两个复杂度问题，架构设计遵循“合适原则”和“简单原则”。
- 原生 App
    > 要保证用户体验，采用原生App架构是最合适的，架构设计遵循“演化原则”。
- Hybrid App
    > 解决“快速开发”的复杂度问题，根据不同的业务要求选取不同的方案，如体验要求高的业务采用原生App实现，对体验要求不高的可以采用Web的方式实现。遵循“合适原则”。
- 组件化&容器化
    > Hybrid App能够较好的平衡“用户体验”和“快速开发”两个复杂度问题，但对于一些超级App来说，随着业务规模越来越大、业务越来越复杂，一个App承载了几十上百个业务。

> 这么多业务集中在一个App上，每个业务又在不断地扩展，后续又可能会扩展新的业务，并且每个业务就是一个独立的团队负责开发，因此整个App的**可扩展性**引入新的复杂度问题。
>
> “组件化&容器化”基本思想就是将超级App拆分为众多组件，这些组件遵循预先制定好的规范，独立开发、独立测试、独立上线。

- 跨平台方案

> 如React Native、Weex、Flutter，但还不成熟