一、架构介绍

架构定义

软件架构指软件系统的顶层结构。

架构是经过系统性地思考、 权衡利弊之后，在现有资源约束下的最合理决策, 最终明确的系统骨架。 包括：子系统、模块、 组件， 以及他们之间协作关系、约束规范、 指导原则。 并由它来指导团队中的每个人思想层面上的一致。

架构本质

系统与子系统

系统：泛指由一群有关联的个体组成，根据某种规则运作，聚合模块和组件。

子系统：也是由一群关联的个体组成的系统，一般是大系统中的一部分。

模块与组件

都是系统的组成部分，从不同角度拆分系统而已。模块是逻辑单元，组件是物理单元。

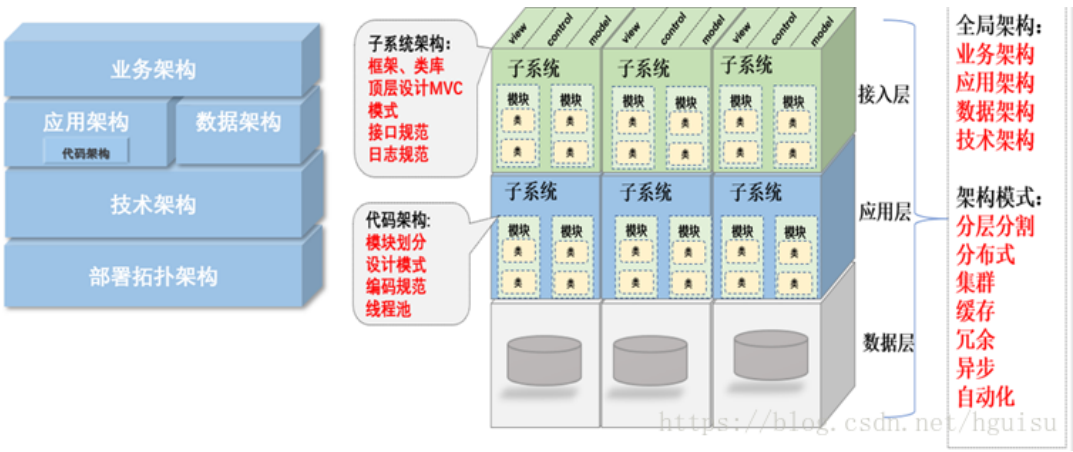
框架与架构

框架是组件的规范，如：MVC等；是提供基础功能的产品，如Spring、SpringMVC等。

框架是规范，架构是结构。

架构分类

架构可细分为业务架构、应用架构、技术架构、 代码架构、部署架构。



业务架构是战略，应用架构是战术，技术架构是装备。其中应用架构承上启下，一方面承接业务架构的落地，另一方面影响技术选型。

业务架构是生产力，应用架构是生产关系，技术架构是生产工具。业务架构决定应用架构，应用架构需要适配业务架构，并随着业务架构不断进化，同时应用架构依托技术架构最终落地。

业务架构

确定总体架构，核心流程，是最上层的战略架构。

包括业务规划，业务模块、业务流程，对整个系统的业务进行拆分，对领域模型进行设计，把现实的业务转化成抽象对象。

没有最优的架构，只有最合适的架构，一切系统设计原则都要以解决业务问题为最终目标，脱离实际业务的技术情怀架构往往会给系统带入大坑，任何不基于业务做异想天开的架构都是耍流氓。

所有问题的前提要搞清楚我们今天面临的业务量有多大，增长走势是什么样，而且解决高并发的过程，一定是一个循序渐进逐步的过程。 合理的架构能够提前预见业务发展1~2年为宜。这样可以付出较为合理的代价换来真正达到技术引领业务成长的效果。

应用架构

确定子系统的功能范围和划分解决方案

应用作为独立可部署的单元，为系统划分了明确的边界，深刻影响系统功能组织、代码开发、部署和运维等各方面. 应用架构定义系统有哪些应用、以及应用之间如何分工和合作。这里所谓应用就是各个逻辑模块或者子系统。

应用架构图关键

1、职责划分：明确应用（各个逻辑模块或者子系统）边界

逻辑分层、子系统和模块定义、关键类

2、职责之间的协作

1）接口协议：应用对外输出的接口。

2）协作关系：应用之间的调用关系。

应用分层方式

1、水平分（横向）

按照功能处理顺序划分应用，比如分为前台(前端)、中台（服务端）、后台（后端），这是面向业务深度的划分。

2、垂直分（纵向）

按照不同的业务类型划分应用，比如进销存系统可以划分为三个独立的应用，这是面向业务广度的划分。

说明：

应用的分偏向于业务，反映业务架构，应用的合偏向于技术，影响技术架

构。分降低了业务复杂度，系统更有序，合增加了技术复杂度，系统更无序。

应用架构的本质是通过系统拆分，平衡业务和技术复杂性，保证系统形散神不散。

业务复杂性（包括业务量大）必然带来技术复杂性，应用架构目标是解决业务复杂性的同时，避免技术太复杂，确保业务架构落地。

技术选型

技术调研, 确定系统核心技术点

1、技术选型

根据业务场景, 了解业内的玩法，能不能解决现有问题. 比如使用微服务构建，消息中间件选型：kafka、rabbitmq

2、评估技术成本

结合业内的玩法与自有技术体系的结合成本，评估使用成本，推广成本。

3、方案取舍

技术方案有多种，了解每种方案优缺点, 群里群策，选出最优、最合适的方案。

4、确定系统核心技术方案

技术架构要考虑系统的非功能性特征，对系统的高可用、高性能、可扩展、安全、隔离性、伸缩性、简洁等做系统级的把握。

其他架构

数据架构

数据架构指导数据库的设计。

架构总览图

这样能够帮助站在一个更高的角度去考虑架构的演变问题。如果是针对现存项目重新做架构，那么需要把现有项目架构梳理出来，作为我们上面思考过程中的一部分参考信息。

详细设计与实施

例子：

比如我们要设计一个微服务的订单系统:

业务: 确定业务流程: 确定订单关键功能点和流程

应用: 确定订单顶层设计,系统模块,对外暴露哪些接口. 接口协议形式.

技术: 确定使用哪些技术点: mysql, mongo,是否考虑分表分库. 使用哪些中间件.

数据: 如何设计表结构.

二、架构设计

架构不是像平常写代码一样，对就是对，错就是错，它并无对错之分，是一个取舍的过程。

我的经验步骤是：业务->功能->技术实现->架构综览图

架构设计的遵循

形成架构原则的过程

架构原则要SMART

	要求	不合适的	合适的
S	原则必须具体	尽快解决P1	在1天内解决P1
M	原则可以度量	无限扩张	不存在扩展的障碍
A	原则可以实现	可用性100%	可用性99.99%
R	原则可能达成	无Bug	100%测试覆盖
T	原则可以验证	N - 1设计	N+1设计

架构设计的关键原则

好的设计：

1）解决现有需求和问题

2）把控现实的进度和风险

3）预测和规划未来，不要过度的设计，从迭代中演进和完善。

六大原则

1、单一职责原则

一个模块或组件只有一个职责或功能，实现要单一。

功能要内聚，业务不要耦合。

这么设计架构是否合理，代码这么写是否合适。

2、开放封闭原则

对扩展开放，对修改关闭。

允许在不改变他源代码的前提下，变更他的行为。

未来好不好扩展，影不影响主流程、影不影响其他业务。

3、里氏替换原则

只要父类出现的地方，子类就可以出现，替换为子类后不会产生任何异常和错误。

约束子类的行为，最小成本替换一种策略实现。

解决继承不合理问题，不要随意继承、过多继承。

4、依赖倒置原则

面向接口编程

减少类与类之间的耦合、提高系统的稳定性。

接口和抽象类是实现扩展性的基石。

5、接口隔离原则

接口单一职责原则，抽象要单一。

使用多个专用接口，比使用一个单一接口要好。

防止接口臃肿、难以维护。

6、迪米特原则

最小（少）知识原则

一个组件或者是对象不应该知道其他组件或者对象的内部实现细节。

降低业务耦合

抽象三原则

1、DRY原则

Don't repeat yourself ， "不要重复自己"。

系统的每一个功能都应该有唯一的实现。不要重复开发。

2、YAGNI原则

You aren't gonna need it ， "你不会需要它"。

"极限编程"提倡的原则，指的是你自以为有用的功能，实际上都是用不到的。不要过度设计。

3、Rule Of Three原则

"三次原则"

当某个功能第三次出现时，才进行"抽象化"。

DRY原则和YAGNI原则的兼容并包。

其他原则

1、关注点分离

横向分层、纵向分域(区)

将应用划分为在功能上尽可能不重复的功能点。主要的参考因素就是最小化交互，高内聚、低耦合。但是，错误的分离功能边界，可能会导致功能之间的高耦合性和复杂性。

好处是把影响最小化，一部分发生变化，不会影响其他部分。

1) 将有关事务模块化，封装到单独的构件（例如子系统）中，并且调用其服务；
2) 使用AOP技术，比如AspectJ以对开发者透明的方式支持在编译之后将横切面关注织入代码。

首先，可以通过职责划分来分离关注点。面向对象设计的关键所在，就是职责的识别和分配。

其次，可以利用软件系统各部分的通用性的不同进行关注点分离。

2、最小化预先设计

只设计必须的内容。在一些情况，你可能需要预先设计一些内容。另外一些情况，尤其对于敏捷开发，你可以避免设计过度。如果你的应用需求是不清晰的，最好不要做大量的预先设计。

架构模式

分层

横向分层

层（layer）这个概念在计算机领域是个很重要的概念，无处不在。比如网络四层模型：应用层、传输层、网际层、网络接口层。

是对模型中同一抽象层次上的包进行分组的一种特定方式。通过分层，从逻辑上将子系统划分成许多集合，各层之间遵循一定的规则。可以限制子系统间的依赖关系，使各系统之间松耦合，从而更易于维护。

架构中常见的分层：产品层、应用层（业务逻辑）、核心层（可复用的平台或服务）、数据层

分层好处：

● 不需要去了解各层的实现细节。某一层内部实现发生变更，不会影响上面的层。

● 可以减少不同层之间的依赖。 容易制定出层标准。

分层缺点：

● 效率降低。层不可能封装所有的功能，一旦有功能变动，势必要波及所有的层。

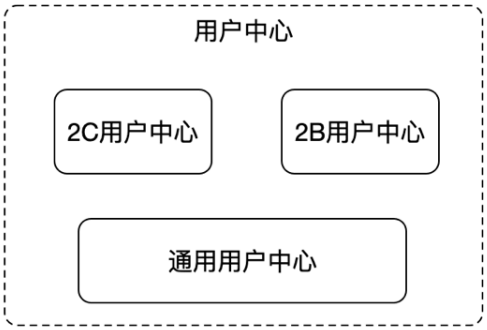
水平（横向）切分

通用、专用分离原则:通用性不同的单元划归不同子系统。

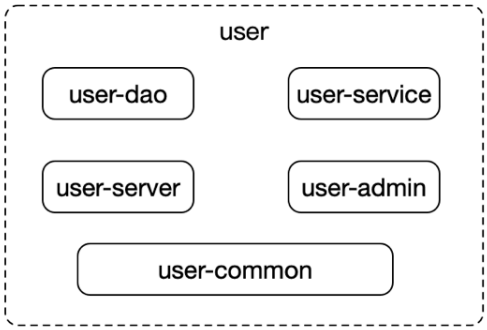
例：用户中心

按系统级别切分，按2C、2B分成两个用户中心系统，比如用户中心和企业中心

按模块级别切分，从数据底层到应用上层，分为dao、service、core、server



系统级



模块级

分层设计的三个常用原则：

向下依赖原则: 上层依赖下层

向上通知原则: 下层不直接与上层打交道，而是通过通知等方式

相邻通信原则: 近邻通信，非相关类用接口等隔离

分割

纵向分割

拆分功能和服务；将不同的功能和服务分割开来，包装成高内聚低耦合的模块单元。

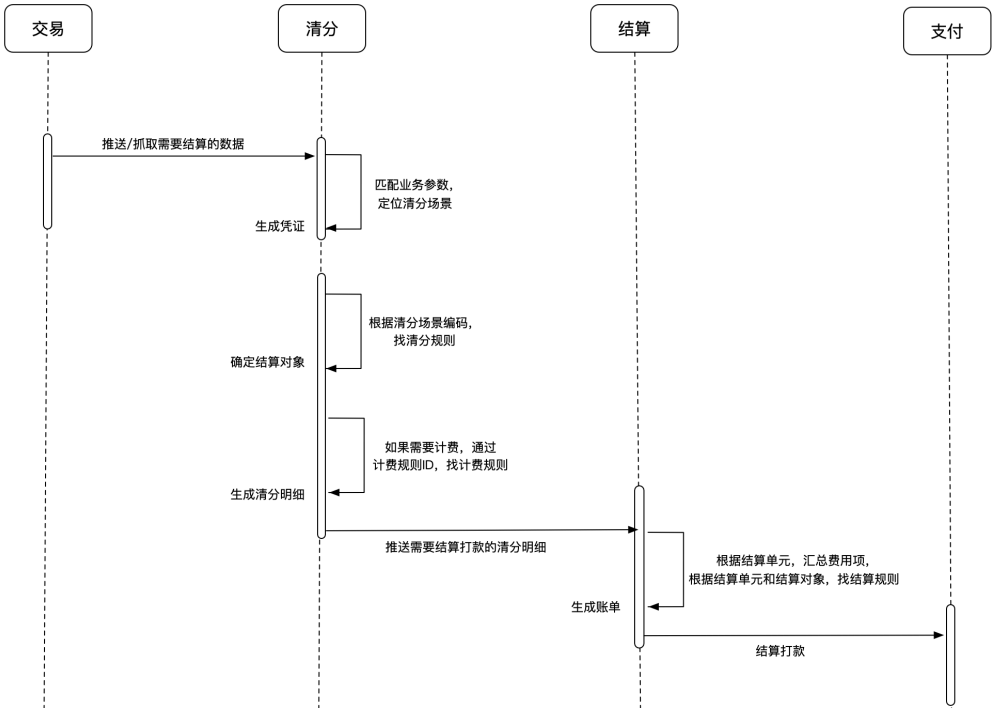
有助于各模块分布式部署，提供并发能力和功能扩展。

垂直（纵向）切分

职责分离原则: 职责不同的单元划归不同子系统

例：清结算

按领域（服务和功能）不同，划分为交易、清分、结算、支付等服务。



技术选型原则

合适原则

合适优于业界领先

简单原则

简单优于复杂

结构的复杂性和逻辑的复杂性。

演化原则

演化优于一步到位，架构设计没有完美银弹，勿过度设计。

1、首先满足当前需要,解决当前最核心问题.

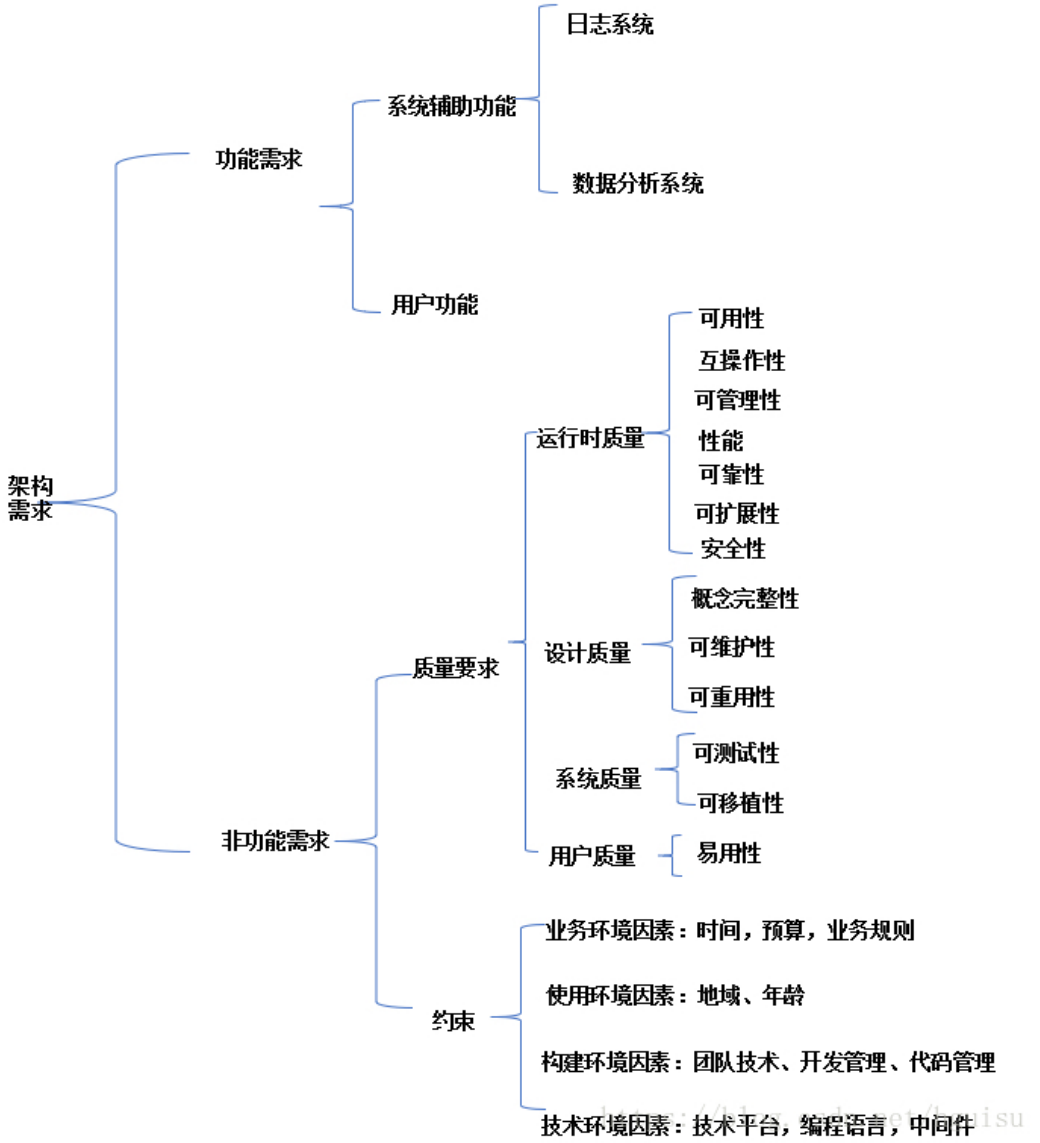
2、预测并并发未来可能存在的问题，不断迭代保留，不断完善，

3、业务变化时，架构扩展、重构、甚至重写。

三、架构需求分析

从需求向设计转化的关键

架构设计的本质目的是为了解决业务，架构设计也并不是面面俱到，而是识别问题有针对性的解决, 所以要先了解系统最需要解决的是什么。



推荐文章系列：https://blog.csdn.net/hguisu/article/category/5905793