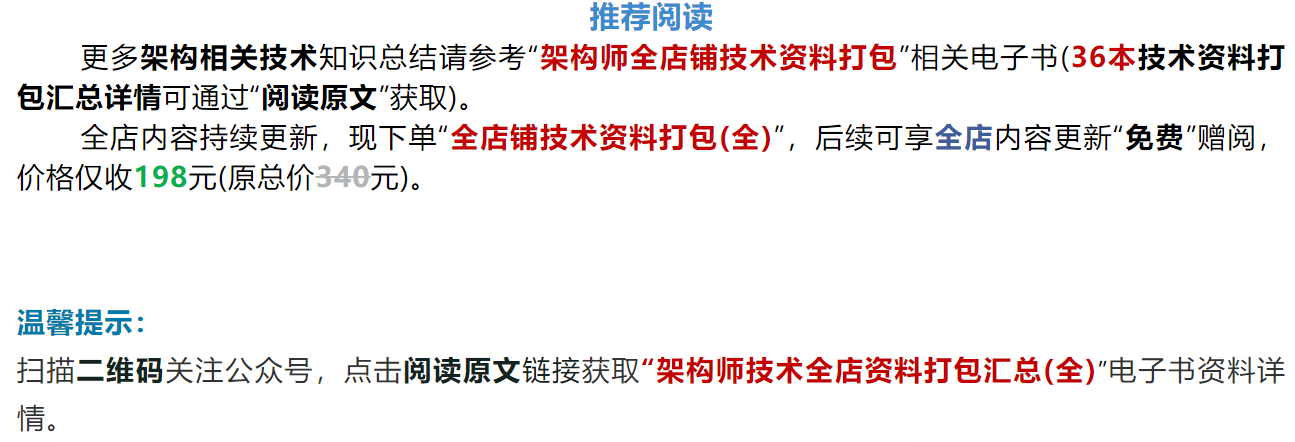
# 架构方法论

4C

## 架构制图

架构制图：工具与方法论

<https://zhuanlan.zhihu.com/p/260068315>



# ToGAF

<https://zhuanlan.zhihu.com/p/35930267>

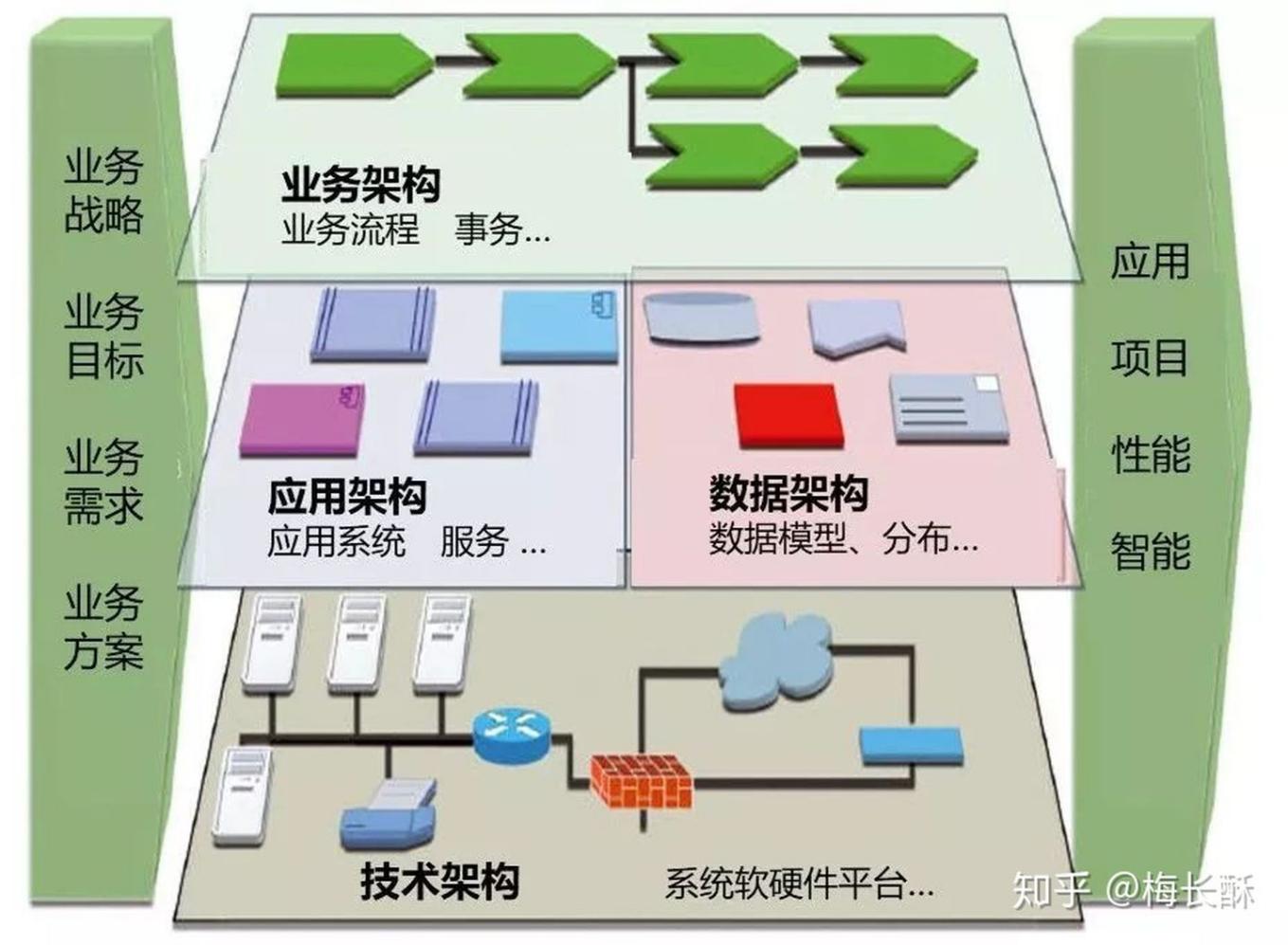
## 业务架构

[https://www.infoq.cn/article/m\*aorjcu6gqiibpcvtok](https://www.infoq.cn/article/m*aorjcu6gqiibpcvtok)

企业架构(Enterprise Architecture，EA)是记录在企业内所有信息系统、它们的相互关系已经它们如何完成企业使命的蓝图。企业架构可以分为两大部分：业务架构和IT架构，大部分企业架构方法都是从IT架构发展而来的。业务架构是把企业的业务战略转化为日常运作的渠道，业务战略决定业务架构，它包括业务的运营模式、流程体系、组织结构、地域分布等内容。IT架构则指导Ⅱ投资和设计决策的Ⅱ框架，是建立企业信息系统的综合蓝图，包括数据架构(又叫信息架构)、应用架构和技术架构三部分。

<https://zhuanlan.zhihu.com/p/35930267>

TOGAF 进一步认为企业架构分为两大部分：业务架构和 IT 架构，大部分企业架构方法都是从 IT 架构发展而来的。业务架构是把企业的业务战略转化为日常运作的渠道，业务战略决定业务架构，它包括业务的运营模式、流程体系、组织结构、地域分布等内容。TOGAF 强调基于业务导向和驱动的架构来理解、分析、设计、构建、集成、扩展、运行和管理信息系统，复杂系统集成的关键，是基于架构（或体系）的集成，而不是基于部件（或组件）的集成。



## 架构公众号

架构师技术联盟

武汉二手房不是按照成交价贷款的，估计这也是武汉二手房交易比较低迷的原因之一吧

## 分布式数据中心网络互联技术实现

[https://mp.weixin.qq.com/s/8nznkIGba6RA05BssGINvQhttps://mp.weixin.qq.com/s/8nznkIGba6RA05BssGINvQ](https://mp.weixin.qq.com/s/8nznkIGba6RA05BssGINvQhttps:/mp.weixin.qq.com/s/8nznkIGba6RA05BssGINvQ)

# 架构笔记

**架构设计**

**<https://zhuanlan.zhihu.com/p/296963761>**

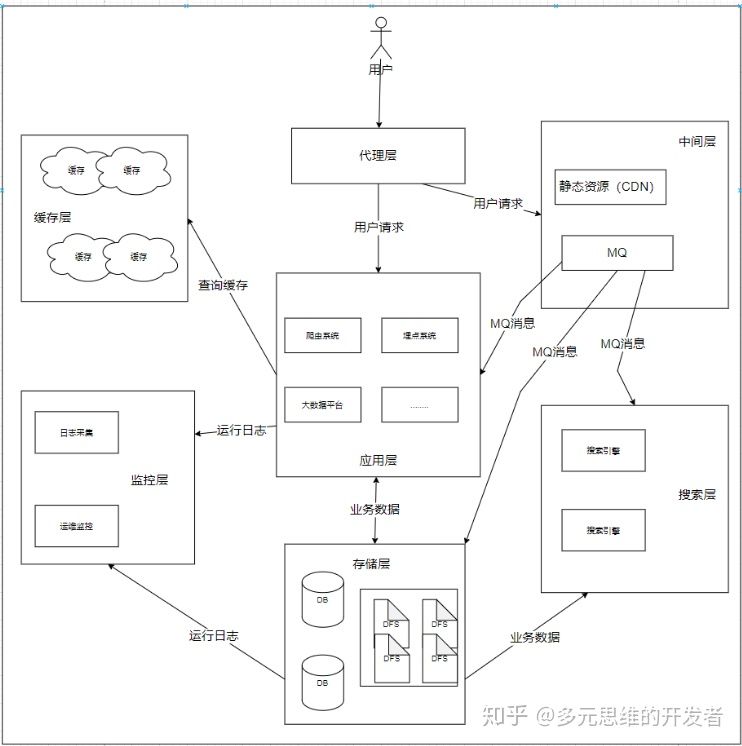
**1、架构演进：**

|  |  |
| --- | --- |
| 1、单体 | 这是最原始的架构，将系统的所有功能都集中在一个工程中。对于小应用，这是非常快捷；但是耦合强。 |
| 2、垂直架构 | 垂直架构与单体一样，都是将所有的功能都集中在一个系统项目中，一旦遇到系统的瓶颈，就在物理层面进行优化。 |
| 3、SOA | SOA：面向服务架构。以系统提供的服务为单位，设计、开发系统，将大系统按照服务进行分模块。 |
| 4、微服务 | 微服务是SOA的强化版，其核心思想也是面向服务。而微服务就是体现在一个微字，将系统服务原子化，单一职责化。尽可能的让一个模块、或子系统只负责一个功能或者是一类功能。 |

**2、架构考虑因素：**

|  |  |
| --- | --- |
| 1、正确性 | 确保基于当前架构设计，功能输出结果是正确的。当然，也有些业务场景是不需要绝对的正确，例如：天气预测结果 |
| 2、安全性 | 考虑系统可能面临或者常见的安全攻击，尤其是对于一些涉及金钱、商业机密数据，更是需要考虑；还有反爬虫，请求过滤之类等等。例如：在哪些场景需要加密通讯，数据库的哪些业务数据需要脱敏、以及常见的CSRF、XSS和SQL攻击；定制网关，进行安全管理、风险控制。 |
| 3、高可用 | 定义系统可用级别（多少个9），而高可用一般都是依靠集群来保证的。然后，分析架构中哪一层需要集群化，最后进行压力测试、稳定性测试（连续、周期性、突发性测试）。还可以对集群进行负载测试、性能测试，看看集群的运行是健康，资源最大化利用。 |
| 4、高性能 | 高性能最直观的感受和衡量标准就是RT了，请求响应速度越快越好，提高用户体验。 |
| 5、扩展性 | 扩展性，指如何在不改变原有代码、模块的基础上，对系统进行功能扩展。这也就是开放闭合原则。 |
| 6、伸缩性 | 伸缩性指的是集群的扩容和缩容，如何无感改变集群规模或者将扩容缩容的影响程度降到最低。这个对线上突发的洪峰具有很大的意义，能够及时缓解集群负载。 |
| 7、可维护性 | 可维护性，一般是指对当前系统进行功能无差异维护。主要体现在代码层面、代码的可读性、类的设计、模块的划分都对维护性具有相当的影响。 |
| 8、其他 | 除了上面考虑因素之外，还可能需要考虑搭建部署成本和难易程度，系统规模和复杂度以及技术选型和搭配等 |

**3、架构分层：**



|  |  |
| --- | --- |
| 代理层 | 用于反向代理，负载均衡。负责对请求进行重定向和分流作用。常用：Nginx，以及高级一点就是lvs + Nginx实现高可用的代理层甚至DNS也可以纳入此中 |
| 应用层 | 主要搭建平台、部署应用系统，对内外提供业务服务，一般地应用服务都以集群形式存在 |
| 存储层 | 数据落地存储，持久化；可以建立数据仓库用于数据挖掘，搜索层和缓存层的数据源。常用MySQL等关系型数据库、HBase等非关系型数据库以及分布式文件系统DFS |
| 缓存层 | 用于常查询业务数据的缓存，提高查询响应速度。数据的来源主要是存储层，常用Redis。 |
| 搜索层 | 提供搜索服务，从存储层或MQ队列中的数据建立索引，提供应用层进行搜索。常用：elasticsearch、solar |
| 中间层 | 中间层是介于代理层和应用层之间，所以我称它中间层。主要提供静态资源请求和MQ消息缓存服务。1、静态资源请求服务：主要体现在图片、视频、文件等不会轻易变化的静态资源的请求处理。这些静态资源会有相应的应用提供处理服务：例如FastDFS。2、MQ：主要用于削峰、失败隔离、异步调用的作用。例如：kafka、RabbitMQ |
| 监控层 | 主要负责各个层的系统运行日志的采集、处理，形成直观的监控数据并展示。常用的：ELK、Flume和logstash |

在大多数的公司，主要是对代理层、MQ层、缓存层、应用层、存储层的搭建和开发。对于监控层和搜索层，要么使用集成的，要么就是没有应用场景。其中，最主要还是对应用层的开发。首先会对应用层的业务进行模块划分、然后集群化提高系统可用性。

**4、架构设计原则：**

|  |  |
| --- | --- |
| 合适原则 | 这个主要体现在技术选型和搭配上，系统的搭建和开发主要是需要结合业务场景和技术或语言特性来综合考虑，甚至还需要对团队人员、项目进度和资源调配来快抉择；并不是业界主流才是最优最好的选择。 |
| 简单原则 | 这个简单实际的意思是：架构层次分明易懂、易替换、易扩展、模块或组件之间依赖关系简单。可以归类为：1、结构简单：模块数量适中、依赖关系简单、架构层次分明2、逻辑简单：模块、组件系统单一职责化 |
| 演化原则 | 演化就是指系统会随着业务场景的变化而不断调整架构，因为业务场景不可能一成不变，很多时候也很难考虑以后可能会出现的场景；因此，系统的架构也不可能永远不变。能做到的就是，分析好当前的业务场景、并尽可能考虑到未来（1~3年）业务场景变化趋势，从而设计出适合当前场景的架构系统；并结合时间推进，对系统的架构持续跟进和调整（有用的保留和优化、无用的去除、留有应对变化的plan B）。 |

<https://zhuanlan.zhihu.com/p/296969974>

## **设计模式：**

* **1、定义：**

设计模式，是一种解决某类问题的可重复使用方案。设计模式的诞生离不开开发的设计原则，在大部分的设计模式的背后都能感受到其背后的原则。而这些模式最终目的，是为了提高程序的高类聚低耦合。

* **2、分类：**
* **2.1、从功能上：**

从功能上对设计模式进行划分，可以分成熟知的三类：

|  |  |
| --- | --- |
| 1、创建型 | 单例、工厂、建造者、原型 |
| 2、结构型 | 代理、门面、桥接、适配器、享元、装饰器、组合 |
| 3、行为型 | 策略、备忘录、迭代器、访问者、观察者、解释器、命令、模板方法、责任链、中介者、状态 |

* **2.2、从方式上：**

从方式上可以分为两类：

|  |  |
| --- | --- |
| 1、类模式 | 以继承的方式实现的模式，属于静态的。 |
| 2、对象模式 | 以组合的方式实现的模式，属于动态。 |
| 在设计模式中，模板方法和迭代器属于类模式 |  |

* **3、设计模式背后的设计原则：**

在很多设计模式中，都能体现其背后的设计原则，不过有些是为了体现现实规则，例如：单例。

|  |  |  |
| --- | --- | --- |
| 模式 | 说明 | 背后的原则 |
| 桥接模式 | 将两类事物通过桥接的方式实现关联，不管两边如何改变，只要遵循桥接的规范，那么还是可以继续关联 | 依赖倒置、开放闭合 |
| 适配器 | 在不改变已有的接口的情况下，通过新增类，组合方式来兼容两个互不相同的接口通讯 | 组合 |
| 装饰器 | 在不改变已有类的情况，对该类的功能进行扩展 | 组合、开放闭合 |
| 迭代器 | 统一遍历的方式 | 迪米特原则 |

* **4、设计模式的意义：**

虽然设计模式的出现是为了解决重复问题，设计出具有良好结构的程序。GOF也提到过：在开发中不要为了使用设计模式而用设计模式。设计模式只是一种方案，在实现上和数量上也不是一成不变的。所以，现在也出现了很多新的设计模式，经典的设计模式也不要持有永远适用的心态。不过万变不离其宗；要记住，设计模式只是方案，其背后诞生的问题背景、设计原则和目的才是我们需要熟悉和牢记的。我们开发时需要清楚现实的问题和背景、了解怎样的程序才是好程序，怎样的代码才算是好代码。

例如：好的代码简洁、高效、不重复，但是又能适用大部分的场合；即使需求改变，也能少改代码、甚至不用改动代码就能进行扩展。这样，我们在设计程序的时候，即使不知道这些设计模式，在开发的时候会自发的设计出和设计模式类似，甚至是高于设计模式的设计方案。所以，不要刻意使用设计模式，要结合自身面临的问题具体分析，然后再结合良好的代码风格，设计出良好的程序。设计模式能够很好的辅助我们进行架构的设计，设计出良好的系统和程序。当然，我们自己也可以根据业务场景，并在原有的模式上，创新出属于自己的设计模式来解决新的问题。

## **面向对象：**

* **1、定义：**

对象就是具备状态、行为和标识的事物，而面向对象是一种处理问题的思维方式；就是看待一类问题，应该想着是由哪些对象来进行处理，然后实现这个对象的处理逻辑和过程。

其实，我觉得面向对象和面向过程不是完全独立的，互不交集；而是相互协作，是宏观和微观上的关系。在宏观上，我们将现实的一类问题的处理方案由对象来负责提供；在微观上，对象将这类问题的处理过程具体实现。例如：一个人受伤了（对象调用者），他应该首先想到是找医生接受治疗（对象的治疗功能），而不是自己来学习治疗方法来治疗自己（这就是分析问题时，面向对象和面向过程区别于出发点的不同）；然后医生的治疗过程，便是面向过程的具体实现。这样受伤的人不用学习治疗的方法，只需要找医生就可以了，由医生来学习具体的治疗过程。

* **2、面向对象编程特征：**

|  |  |
| --- | --- |
| 1特征 | 说明 |
| 封装 | 将对象的功能实现细节进行封装隐藏，以及访问的控制。遵循“最少知道”原则。 |
| 继承 | 如果两个类之间具有is-a的关系，那么都可以使用继承。提高接口和属性的重用 |
| 多态 | 持有父类类型引用，根据传入的具体的子类类型，从而表现不同的行为 |

* **3、面向对象设计原则：**

|  |  |
| --- | --- |
| 原则 | 说明 |
| 开闭原则（OCP） | 对扩展开放，对修改关闭。装饰器模式就是很好的实践例子，通过组合的方式，新增类来扩展功能。 |
| 依赖倒置原则（DIP） | 高层模块不依赖低层模块，而是依赖一个抽象模块。桥接模式就是一个例子，或者说JDBC。这个桥梁就是抽象接口，而桥的两边便是高层模块和底层模块，只要它们遵循接口的协议，那么就不用在乎桥的两边如何变化 |
| 里式替换原则（LSP） | 只要父类出现的地方，都可以传入子类对象进行替换，并不会改变整体的功能。这也是验证继承关系是否合理的一个重要原则之一。 |
| 单一职责原则（SRP） | 一个类只有一个原因引起它的变化，也就是一个类只负责一个或一类功能。 |
| 接口隔离原则（ISP） | 相关的接口聚合在一起，不相关的就分离；同时也不让客户知道他们不需要的方法。 |
| 迪米特原则（LKP） | 迪米特原则也叫最少知道原则，这是体现封装思想。就是让一个类只暴露需要被调用的方法，不过多暴露其他细节。 |

<https://zhuanlan.zhihu.com/p/296977046>

## **架构的生命周期：**

什么是架构？架构在互联网中被描述为：系统由什么组件组成，各个组件之间的关系是怎样。而一个系统在其生命周期中会演化出不同的架构，就像一个人从小孩到长大成人，身体、智力和精神会出现不同程度的变化，从而来适应每个阶段的变化和挑战。

因此，一个良好的架构不是指一上来就设计出一个大型网站的架构，而是结合业务背景、面临的问题和挑战，来综合考虑并设计适合当前状况的架构。并且随着的时间的推移，场景的变化来不断的审核目前的架构是否还合适，如果不合适，是因为什么原因，应该怎么办？如果还合适，那么以目前的发展趋势，当前的架构还能支撑到什么时候？简而言之，抓到老鼠就是好猫。因为好的架构不一定合适，但合适的架构一定是好的架构。

**1、架构为什么会有生命周期？**

架构和人的成长一样，会有各阶段成长历程，这也就是架构的生命周期。之所以会出现架构的生命周期，离不开环境的改变，架构需要适应新的变化和挑战。下面简单罗列普遍的架构生命周期：

|  |  |  |
| --- | --- | --- |
| 环境背景 | 面临的挑战 | 架构 |
| 系统从0到1的开辟 | 快速搭建系统，快速实现需求，面向用户 | 单体架构，应用服务器、存储、网关都集中在一台机器或者分布在不同机器。 |
| 系统用户数量快速增长 | 高并发，大流量 | 分布式集群架构，系统应用集群化，提高整体的吞吐量。 |
| 系统用户数量继续增长、用户诉求增多、需求快速变更 | 业务种类繁多、庞大，部署效率低、高并发、高可用 | SOA或微服务架构，对系统进行功能分层和业务分割，就系统根据功能和业务进行划分，并单一职责化，分开并集群部署，相互隔离但又互相联系。 |
| 系统海量数据累积 | 海量数据的管理（存储、检索和计算） | 大数据架构，采集-计算-存储 |

**2、架构的设计手段：**

不同的阶段有不同的架构，而这些架构的设计都离不开这些核心手段：

**2.1、从作用对象来看：**

|  |  |
| --- | --- |
| 1、垂直伸缩 | 提高单台物理机器的性能，例如：cpu核数、内存大小、磁盘容量、网络带宽。容易受到物理限制，并且存在一个提升的临界点。过了这个临界点，之后的提升与成本之间不再是1:1的关系。 |
| 2、水平伸缩 | 提高物理机器的数量，不会受到物理限制，可以无限伸缩。 |

**2.2、水平伸缩的扩展：**

在大多数情况下，都是基于水平伸缩对系统进行优化和扩展。

|  |  |
| --- | --- |
| 分层 | 根据功能进行水平分层。例如：存储、搜索、缓存、应用、代理、中间件等 |
| 分割 | 根据业务进行垂直分割。以电商为例：商品、支付、评论、登录等 |
| 分布式 | 将不同的功能模块、业务模块分开独立部署和通讯。 |
| 集群 | 将同类功能、业务模块集群化。 |

**3、架构模式：**

模式就是用于解决某类问题的可复用解决方案，而架构模式自然就是一种重复使用的架构方案。例如：大家都耳熟能详的 主从、集群和哨兵。这三种也许在Redis中出现就以为是Redis特有，但其实不是，这是非常典型的架构模式，而这些模式都是基于分布式的基础上实现的。所以为什么分布式和集群有着密切关系，就是这样了。

|  |  |
| --- | --- |
| 主从模式 | 一主多从的模式、例如：MySQL的读写分离 |
| 集群模式 | 与主从模式的区别在于，集群没有主。例如：应用集群、Redis集群。经常与主从混为一谈。 |
| 哨兵模式 | 这种是基于主从模式，加入监听者的一种模式。目的是为了确保集群有Master |
| 大数据模式 | 也就是前面提到的：采集-计算-存储，甚至还可以加入搜索。例如：典型的ELK |

**4、架构的衡量因素：**

|  |
| --- |
| 高可用 |
| 高性能 |
| 扩展性 |
| 伸缩性 |
| 安全性 |

# 微服务

聊聊用于构建微服务的工具和技术

<https://cloud.tencent.com/developer/article/1149186>

# 领域模型和数据模型

<https://mp.weixin.qq.com/s/jlLcSUxpflpPAaDQ87FBVg>

# 整洁面向对象分层架构

<https://github.com/alibaba/COLA>

Todo 待体验。