架构篇：什么才是真正的架构设计？

架构营 2023-12-06 08:49 发表于上海

一. 什么是架构和架构本质

在软件行业，对于什么是架构，都有很多的争论，每个人都有自己的理解。此君说的架构和彼君理解的架构未必是一回事。因此我们在讨论架构之前，我们先讨论架构的概念定义，概念是人认识这个世界的基础，并用来沟通的手段，如果对架构概念理解不一样，那沟通起来自然不顺畅。

Linux有架构，MySQL有架构，JVM也有架构，使用Java开发、MySQL存储、跑在Linux上的业务系统也有架构，应该关注哪一个？想要清楚以上问题需要梳理几个有关系又相似的概念：系统与子系统、模块与组建、框架与架构:

1.1. 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，能完成个别元件不能独立完成的工作能力的群体。

子系统：也是由一群关联的个体组成的系统，多半是在更大的系统中的一部分。

1.2. 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。

模块就是从逻辑上将系统分解， 即分而治之， 将复杂问题简单化。模块的粒度可大可小， 可以是系统，几个子系统、某个服务，函数， 类，方法、 功能块等等。

组件可以包括应用服务、数据库、网络、物理机、还可以包括MQ、容器、Nginx等技术组件。

1.3. 框架与架构

框架是组件实现的规范，例如：MVC、MVP、MVVM等，是提供基础功能的产品，例如开源框架：Ruby on Rails、Spring、Laravel、Django等，这是可以拿来直接使用或者在此基础上二次开发。

框架是规范，架构是结构。

我在这重新定义架构：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。涉及四方面：

- 系统性思考的合理决策：比如技术选型、解决方案等。
- 明确的系统骨架：明确系统有哪些部分组成。
- 系统协作关系：各个组成部分如何协作来实现业务请求。
- 约束规范和指导原则：保证系统有序，高效、稳定运行。

因此架构师具备能力：**理解业务，全局把控，选择合适技术，解决关键问题、指导研发落地实施。**

架构的本质就是对系统进行有序化地重构以致符合当前业务的发展，并可以快速扩展。

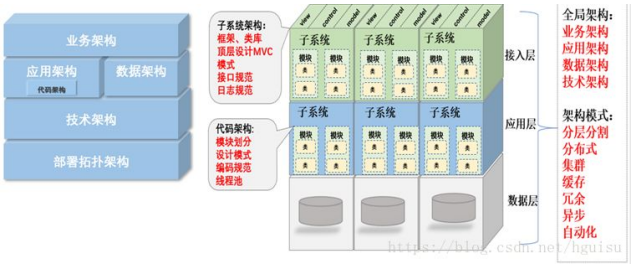
那什么样的系统要考虑做架构设计 技术不会平白无故的出和自驱动发展起来，而架构的发展和需求是基于业务的驱动。

架构设计完全是为了业务，

- 需求相对复杂.
- 非功能性需求在整个系统占据重要位置.
- 系统生命周期长,有扩展性需求.
- 系统基于组件或者集成的需要.
- 业务流程再造的需要.

二. 架构分层和分类

架构分类可细分为业务架构、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。

熟悉业务，形成业务架构，根据业务架构，做出相应的应用架构，最后技术架构落地实施。

如何针对当前需求，选择合适的应用架构，如何面向未来，保证架构平滑过渡，这个是软件开发者，特别是架构师，都需要深入思考的问题。

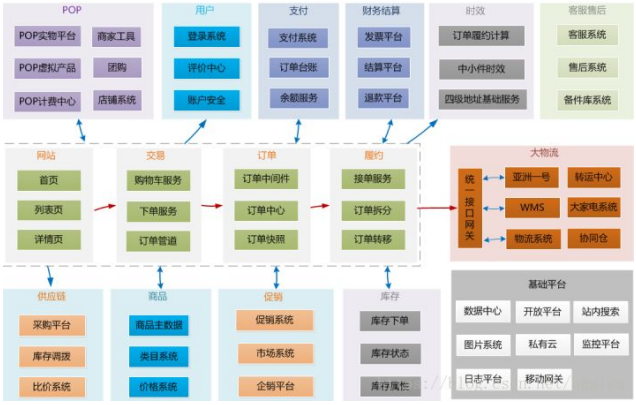
2.1. 业务架构（俯视架构）：

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。

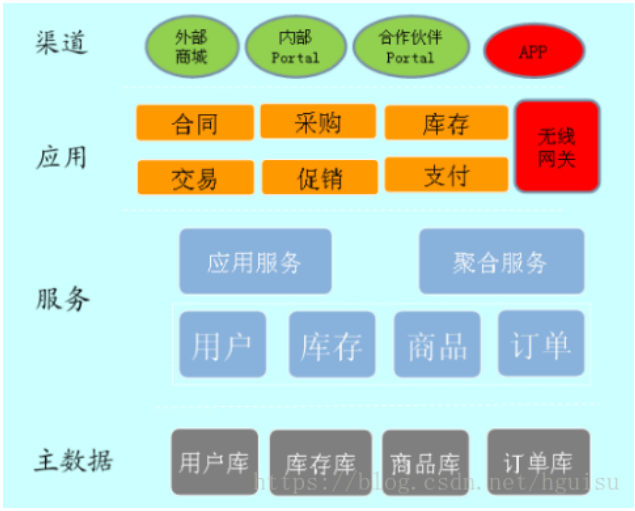
看看京东业务架构（网上分享图）：



2.2. 应用架构（剖面架构，也叫逻辑架构图）：

硬件到应用的抽象，包括抽象层和编程接口。应用架构和业务架构是相辅相成的关系。业务架构的每一部分都有应用架构。

类似：



应用架构：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。

应用架构图关键有2点：

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、模块定义。
- 关键类。

②. 职责之间的协作：

- 接口协议：应用对外输出的接口。
- 协作关系：应用之间的调用关系。

应用分层有两种方式：

- 一种是水平分（横向），按照功能处理顺序划分应用，比如把系统分为web前端/中间服务/后台任务，这是面向业务深度的划分。
- 另一种是垂直分（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。

应用的合反映应用之间如何协作，共同完成复杂的业务case，主要体现在应用之间的通讯机制和数据格式，通讯机制可以是同步调用/异步消息/共享DB访问等，数据格式可以是文本/XML/JSON/二进制等。

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。

系统采用什么样的应用架构，受业务复杂性影响，包括企业发展阶段和业务特点；同时受技术复杂性影响，包括IT技术发展阶段和内部技术人员水平。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。

2.3. 数据架构

数据架构指导数据库的设计. 不仅仅要考虑开发中涉及到的数据库，实体模型，也要考虑物理架构中数据存储的设计。

No.	考虑的方面	产出物	工具	说明
1	数据是集中还是分布存储的？如何考虑分布式存储？	数据架构图		
2	领域模型到数据库表的转换？表结构关系的设计？	逻辑模型 物理模型 ER图	Power Designer Visio	
3	实体如何设计？充血模型和贫血模型？	UML类图		
4	使用什么数据库？关系型还是非关系型？	选型结果		

2.4. 代码架构（也叫开发架构）：

子系统代码架构主要为开发人员提供切实可行的指导，如果代码架构设计不足，就会造成影响全局的架构设计。比如公司内不同的开发团队使用不同的技术栈或者组件，结果公司整体架构设计就会失控。

代码架构主要定义：

①. 代码单元:

- 配置设计
- 框架、类库。

②. 代码单元组织：

- 编码规范，编码的惯例。
- 项目模块划分
- 顶层文件结构设计，比如mvc设计。
- 依赖关系

No.	考虑的方面	产出物	工具	说明
1	分层结构设计	分层架构图（开发架构图）	各种绘图工具	好的分层结构支持自动化测试
2	开发技术选项	开发语言 开发框架 开发工具		考虑商用产品、开源框架、自研框架
3	模块划分	源码工程：Project目录结构； 分包(分库)		
4	开发规范	开发/编码规范文档；		
5	软件质量属性	分析和决策结果		考虑运行期和开发期软件质量属性，并权衡利弊进行决策。

2.5. 技术架构

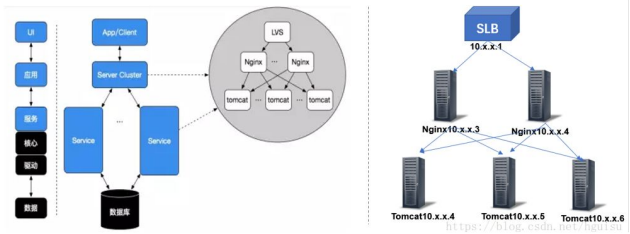
技术架构：确定组成应用系统的实际运行组件（lvs，nginx，tomcat，php-fpm等），这些运行组件之间的关系，以及部署到硬件的策略。

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

系统架构的设计要求架构师具备软件和硬件的功能和性能的过硬知识，这也是架构设计工作中最为困难的工作。

2.6. 部署拓扑架构图（实际物理架构图）：

拓扑架构，包括架构部署了几个节点，节点之间的关系，服务器的高可用，网路接口和协议等，决定了应用如何运行，运行的性能，可维护性，可扩展性，是所有架构的基础。这个图主要是运维工程师主要关注的对象。

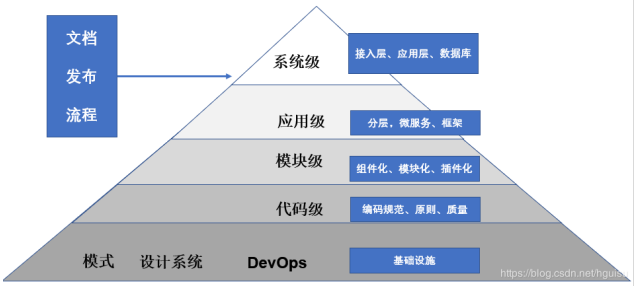


物理架构主要考虑硬件选择和拓扑结构，软件到硬件的映射，软硬件的相互影响。

	考虑的方面	产出物	工具	说明
1	网络方面：网络拓扑；网络设备；安全机制	拓扑图 安全规范		
2	性能方面：可靠性、可伸缩性	需要什么样设备性能		
3	部署方面：集中式还是分布式；组件部署	部署图		

三. 架构级别

我们使用金字塔的架构级别来说明,上层级别包含下层：



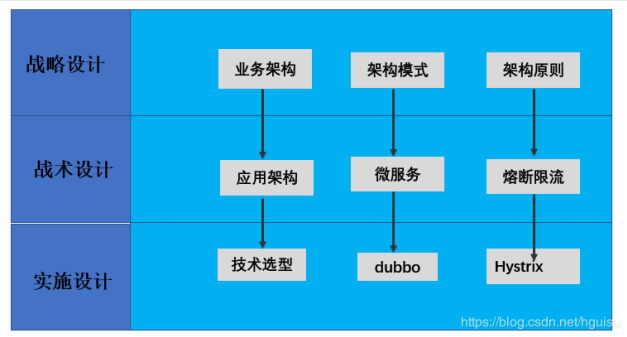
- **系统级**：即整个系统内各部分的关系以及如何治理：分层
- **应用级**：即单个应用的整体架构，及其与系统内单个应用的关系等。
- **模块级**：即应用内部的模块架构，如代码的模块化、数据和状态的管理等。

- **代码级**：即从代码级别保障架构实施。

战略设计与战术设计

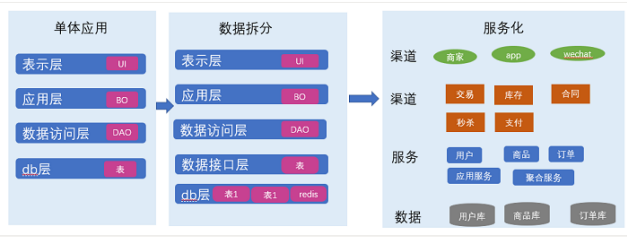
基于架构金字塔，我们有了系统架构的战略设计与战术设计的完美结合：

- **战略设计**：业务架构用于指导架构师如何进行系统架构设计。
- **战术设计**：应用架构要根据业务架构来设计。
- **战术实施**：应用架构确定以后，就是技术选型。



四. 应用架构演进

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。

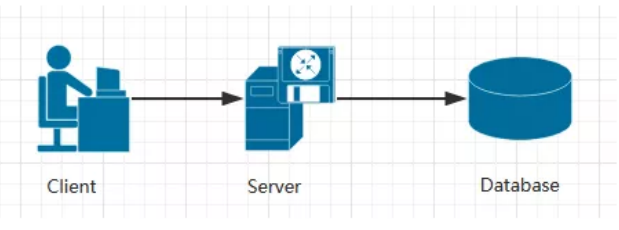


架构演进路程：单体应用→分布式应用服务化→微服务

4.1. 单体应用

企业一开始业务比较简单，只应用某个简单场景，应用服务支持数据增删改查和简单的逻辑即可，单体应用可以满足要求。

典型的三级架构，前端（Web/手机端）+中间业务逻辑层+数据库层。这是一种典型的Java Spring MVC或者Python Django框架的应用。其架构图如下所示：



单体架构 https://guisu.blog.csdn.net

针对单体应用，非功能性需求的做法：

- 性能需求：使用缓存改善性能
- 并发需求：使用集群改善并发
- 读写分离：数据库地读写分离
- 使用反向代理和cdn加速
- 使用分布式文件和分布式数据库

单体架构的应用比较容易部署、测试， 在项目的初期，单体应用可以很好地运行。然而，随着需求的不断增加， 越来越多的人加入开发团队，代码库也在飞速地膨胀。慢慢地，单体应用变得越来越臃肿，可维护性、灵活性逐渐降低，维护成本越来越高。下面是单体架构应用的一些缺点：

- **复杂性高**：以一个百万行级别的单体应用为例，整个项目包含的模块非常多、模块的边界模糊、 依赖关系不清晰、 代码质量参差不齐、 混乱地堆砌在一起。可想而知整个项目非常复杂。每次修改代码都心惊胆战， 甚至添加一个简单的功能， 或者修改一个Bug都会带来隐含的缺陷。
- **技术债务**：随着时间推移、需求变更和人员更迭，会逐渐形成应用程序的技术债务，并且越积 越多。 “ 不坏不修”， 这在软件开发中非常常见， 在单体应用中这种思想更甚。已使用的系统设计或代码难以被修改，因为应用程序中的其他模块可能会以意料之外的方式使用它。
- **部署频率低**：随着代码的增多，构建和部署的时间也会增加。而在单体应用中， 每次功能的变更或缺陷的修复都会导致需要重新部署整个应用。全量部署的方式耗时长、 影响范围大、 风险高， 这使得单体应用项目上线部署的频率较低。而部署频率低又导致两次发布之间会有大量的功能变更和缺陷修复，出错率比较高。
- **可靠性差**：某个应用Bug，例如死循环、内存溢出等， 可能会导致整个应用的崩溃。
- **扩展能力受限**：单体应用只能作为一个整体进行扩展，无法根据业务模块的需要进行伸缩。例如，应用中有的模块是计算密集型的，它需要强劲的CPU；有的模块则是IO密集型的，需要更大的内存。由于这些模块部署在一起，不得不在硬件的选择上做出妥协。
- **阻碍技术创新**：单体应用往往使用统一的技术平台或方案解决所有的问题， 团队中的每个成员 都必须使用相同的开发语言和框架，要想引入新框架或新技术平台会非常困难。

4.2. 分布式

随着业务深入，业务要求的产品功能越来越多，每个业务模块逻辑也都变得更加复杂，业务的深度和广度都增加，使得单体应用变得越来越臃肿，可维护性、灵活性逐渐降低，增加新功能开发周期越来越长，维护成本越来越高。

这时需要对系统按照业务功能模块拆分，将各个模块服务化，变成一个分布式系统。业务模块分别部署在不同的服务器上，各个业务模块之间通过接口进行数据交互。

该架构相对于单体架构来说，这种架构提供了负载均衡的能力，大大提高了系统负载能力，解决了网站高并发的需求。另外还有以下特点：

- **降低了耦合度**：把模块拆分，使用接口通信,降低模块之间的耦合度。

- **责任清晰**：把项目拆分成若干个子项目，不同的团队负责不同的子项目。
- **扩展方便**：增加功能时只需要再增加一个子项目，调用其他系统的接口就可以。
- **部署方便**：可以灵活的进行分布式部署。
- **提高代码的复用性**：比如Service层，如果不采用分布式rest服务方式架构就会在手机Wap商城，微信商城，PC，Android，iOS每个端都要写一个Service层逻辑，开发量大，难以维护一起升级，这时候就可以采用分布式rest服务方式，公用一个service层。
- **缺点**：系统之间的交互要使用远程通信，接口开发增大工作量，但是利大于弊。

4.3. 微服务

紧接着业务模式越来越复杂，订单、商品、库存、价格等各个模块都很深入，比如价格区分会员等级，访问渠道（app还是PC），销售方式（团购还是普通）等，还有大量的价格促销，这些规则很复杂，容易相互冲突，需要把分散到各个业务的价格逻辑进行统一管理，以基础价格服务的方式透明地提供给上层应用，变成一个微内核的服务化架构，即微服务。

微服务的特点：

- **易于开发和维护**：一个微服务只会关注一个特定的业务功能，所以它业务清晰、代码量较少。开发和维护单个微服务相对简单。而整个应用是由若干个微服务构建而成的，所以整个应用也会被维持在一个可控状态。
- **单个微服务启动较快**：单个微服务代码量较少， 所以启动会比较快。
- **局部修改容易部署**：单体应用只要有修改，就得重新部署整个应用，微服务解决了这样的问题。一般来说，对某个微服务进行修改，只需要重新部署这个服务即可。
- **技术栈不受限**：在微服务架构中，可以结合项目业务及团队的特点，合理地选择技术栈。例如某些服务可使用关系型数据库MySQL；某些微服务有图形计算的需求，可以使用Neo4j；甚至可根据需要，部分微服务使用Java开发，部分微服务使用Node.js开发。

微服务虽然有很多吸引人的地方，但它并不是免费的午餐，使用它是有代价的。使用微服务架构面临的挑战。

- **运维要求较高**：更多的服务意味着更多的运维投入。在单体架构中，只需要保证一个应用的正常运行。而在微服务中，需要保证几十甚至几百个服务服务的正常运行与协作，这给运维带来了很大的挑战。
- **分布式固有的复杂性**：使用微服务构建的是分布式系统。对于一个分布式系统，系统容错、网络延迟、分布式事务等都会带来巨大的挑战。
- **接口调整成本高**：微服务之间通过接口进行通信。如果修改某一个微服务的API，可能所有使用了该接口的微服务都需要做调整。
- **重复劳动**：很多服务可能都会使用到相同的功能，而这个功能并没有达到分解为一个微服务的程度，这个时候，可能各个服务都会开发这一功能，从而导致代码重复。尽

管可以使用共享库来解决这个问题（例如可以将这个功能封装成公共组件，需要该功能的微服务引用该组件），但共享库在多语言环境下就不一定行得通了。

五. 衡量架构的合理性

架构为业务服务，没有最优的架构，只有最合适的架构，架构始终以高效，稳定，安全为目标来衡量其合理性。

合理的架构设计：

5.1. 业务需求角度

- 能解决当下业务需求和问题
- 高效完成业务需求: 能以优雅且可复用的方式解决当下所有业务问题
- 前瞻性设计: 能在未来一段时间都能以第2种方式满足业务，从而不会每次当业务进行演变时，导致架构翻天覆地的变化。

5.2. 非业务需求角度

①. 稳定性。指标：

- **高可用**：要尽可能的提高软件的可用性，我想每个操作人都不愿意看到自己的工作无法正常进行。黑盒白盒测试、单元测试、自动化测试、故障注入测试、提高测试覆盖率等方式来一步一步推进。

②. 高效指标：

- **文档化**：不管是整体还是部分的整个生命周期内都必须做好文档化，变动的来源包括但不限于BUG，需求。
- **可扩展**：软件的设计秉承着低耦合的理念去做，注意在合理的地方抽象。方便功能更改、新增和运用技术的迭代，并且支持在适时对架构做出重构。
- **高复用**：为了避免重复劳动，为了降低成本，我们希望能够重用之前的代码、之前的设计。这点对于架构环境的依赖是最大的。

③. 安全指标

- **安全**：组织的运作过程中产生的数据都是具有商业价值的，保证数据的安全也是刻不容缓的一部分。以免出现XX门之类丑闻。加密、https等为普遍手段

六. 常见架构误区

开高走落不到实处

- 遗漏关键性约束与非功能需求
- 为虚无的未来埋单而过度设计
- 过早做出关键性决策
- 客户说啥就是啥成为传话筒
- 埋头干活儿缺乏前瞻性
- 架构设计还要考虑系统可测性
- 架构设计不要企图一步到位

常见误区

- **误区1——架构专门由架构师来做，业务开发人员无需关注**：架构的再好，最终还是需要代码来落地，并且组织越大这个落地的难度越大。不单单是系统架构，每个解决方案每个项目也由自己的架构，如分层、设计模式等。如果每一块砖瓦不够坚固，那么整个系统还是会由崩塌的风险。所谓“千里之堤，溃于蚁穴”。
- **误区2——架构师确定了架构蓝图之后任务就结束了**：架构不是“空中楼阁”，最终还是要落地的，但是架构师完全不去深入到第一线怎么知道“地”在哪？怎么才能落的稳稳当当。
- **误区3——不做出完美的架构设计不开工**：世上没有最好架构，只有最合适的架构,不要企图一步到位。我们需要的不是一下子造出一辆汽车，而是从单轮车→自行车→摩托车，最后再到汽车。想象一下2年后才能造出的产品，当初市场还存在吗？
- **误区4—— 为虚无的未来埋单而过度设计**：在创业公司初期，业务场景和需求边界很难把握，产品需要快速迭代和变现，需求频繁更新，这个时候需要的是快速实现。不要过多考虑未来的扩展，说不定功能做完，效果不好就无用了。如果业务模式和应用场景边界都已经比较清晰，是应该适当的考虑未来的扩展性设计。
- **误区5——一味追随大公司的解决方案**：由于大公司巨大成功的光环效应，再加上从大公司挖来的技术高手的影响，网站在讨论架构决策时，最有说服力的一句话就成了“淘宝就是这么搞的”或者“腾讯 就是这么搞的”。大公司的经验和成功模式固然重要，值得学习借鉴，但如果因此而变得盲从，就失去了坚持自我的勇气，在架构演化的道路上迟早会迷路。
- **误区6——为了技术而技术**：技术是为业务而存在的，除此毫无意义。在技术选型和架构设计中，脱离网站业务发展的实际，一味追求时髦的新技术，可能会将技术发展引入崎岖小道，架构之路越走越难。考虑实现成本、时间、人员等各方面都要综合考虑，理想与现实需要折中。

七. 架构知识体系

7.1. 架构演进

- 初始阶段：LAMP,部署在一台服务器
- 应用服务器和数据服务器分离

- 使用缓存改善性能
- 使用集群改善并发
- 数据库地读写分离
- 使用反向代理和cdn加速
- 使用分布式文件和分布式数据库
- 业务拆分
- 分布式服务

7.2. 架构模式

分层：横向分层：应用层，服务层，数据层

分割：纵向分割：拆分功能和服务

分布式

- 分布式应用和服务
- 分布式静态资源
- 分布式数据和存储
- 分布式计算

集群：提高并发和可用性

缓存：优化系统性能

- cdn
- 方向代理访问资源
- 本地缓存
- 分布式缓存

异步：降低系统的耦合性

- 提供系统的可用性
- 加快响应速度

冗余：冷备和热备，保证系统的可用性

自动化：发布，测试，部署，监控，报警，失效转移，故障恢复

安全：

7.3. 架构核心要素

高性能：网站的灵魂

- 性能测试

- 前端优化
- 应用优化
- 数据库优化

可用性：保证服务器不宕机，一般通过冗余部署备份服务器来完成

- 负载均衡
- 数据备份
- 自动发布
- 灰度发布
- 监控报警

伸缩性：建集群，是否快速应对大规模增长的流量，容易添加新的机器

集群

- 负载均衡
- 缓存负载均衡

可扩展性：主要关注功能需求，应对业务的扩展，快速响应业务的变化。是否做法开闭原则，系统耦合依赖

- 分布式消息
- 服务化

安全性：网站的各种攻击，各种漏洞是否堵住，架构是否可以做到限流作用，防止ddos攻击。

- xss攻击
- sql注入
- csr攻击
- web防火墙漏洞
- 安全漏洞
- ssl

八. 架构书籍推荐

1.《大型网站技术架构：核心原理与案例分析》

这是比较早，比较系统介绍大型网站技术架构的书，通俗易懂又充满智慧，即便你之前完全没接触过网站开发，通读前几章，也能快速获取到常见的网站技术架构及其应用场景。非常赞。

2.《亿级流量网站架构核心技术》

相比《大型网站技术架构》的高屋建瓴，开涛的这本《亿级流量网站架构核心技术》则落实到细节，网站架构中常见的各种技术，比如缓存、队列、线程池、代理……，统统都讲到了，而且配有核心代码。甚至连 Nginx 的配置都有！

如果你想在实现大流量网站时找参考技术和代码，这本书最合适啦。

3.《架构即未来》

这是一本“神书”啦，超越具体技术层面，着重剖析架构问题的根源，帮助我们弄清楚应该以何种方式管理、领导、组织和配置团队。

4.《分布式服务架构：原理、设计与实战》

这本书全面介绍了分布式服务架构的原理与设计，并结合作者在实施微服务架构过程中的实践经验，总结了保障线上服务健康、可靠的最佳方案，是一本架构级、实战型的重量级著作。

5.《聊聊架构》

这算是架构方面的一本神书了，从架构的原初谈起，从业务的拆分谈起，谈到架构的目的，架构师的角色，架构师如何将架构落地……强烈推荐。

不过，对于没有架构实践经验的小伙伴来讲，可能会觉得这本书比较虚，概念多，实战少。但如果你有过一两个项目的架构经验，就会深深认同书中追本溯源探讨的架构理念。

6.《软件架构师的12项修炼》

大多数时候所谓的“技术之玻璃天花板”其实只是缺乏软技能而已。这些技能可以学到，缺乏的知识可以通过决定改变的努力来弥补。

作者 | 规速

来源 | blog.csdn.net/hguisu/article/details/78258430

架构 211

架构 · 目录

< 上一篇

SpringBoot：SpEL让复杂权限控制变得很简单！

下一篇 >

什么是比较好的低代码产品？

喜欢此内容的人还喜欢

深入理解RBAC

架构营

“所有代码，都是技术债务！”

架构营

颜值吊打 Postman的开源 API 调试工具

架构营