架构设计原则精华总结

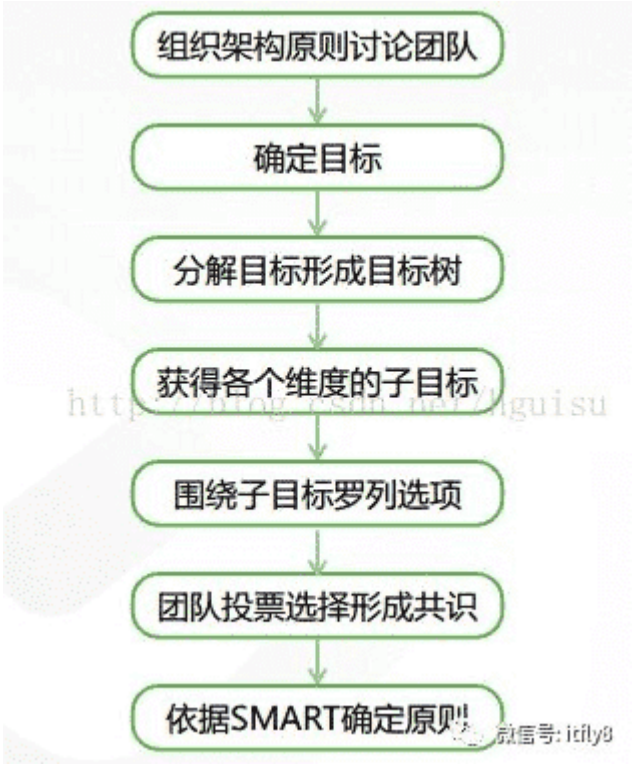
架构设计（2）-架构设计原则

如何设计出一个好的架构，不像数据公式或者定律，很难一概而就。很多时候是设计者（架构师）的各种设想，各种权衡折中而符合系统需求的智慧输出。但我们掌握前人总结的经验，让我们站在巨人的肩膀上高山远瞩。一些好的架构设计原则可以确保设计决策在一定程度上能够满足需求。

#一、形成架构原则的过程

形成架构原则的过程：

架构原则要SMART



#二、15条普适架构原则

《架构真经》这本书简单阐述了架构设计的一些常用的原则。罗列一些常用的原则，下面是15个具有普适价值架构原则：

#1、N+1设计：开发的系统在发生故障时，至少有一个冗余的实例

广泛地应用在从数据中心设计到应用服务的部署：

在发生故障时，系统至少要有一个冗余的实例。

必须确保一个为自己，一个为客户、一个为失败

#2、回滚设计：确保系统可以向后兼容。

1）如果很久才能修复服务，那么就要在一定的时间范围内完成回滚。

2）灾难性的事故，例如损坏客户数据，往往在部署后好几天才出现。

3）系统最好按照预先的设计，通过发布或回滚解决问题。

#3、禁用设计：可以关闭任何发布功能

当设计系统，特别是与其他系统或服务通讯的高风险系统时，要确保这些系统能够通过开关来禁用。这将为修复服务提供额外的时间，同时确保系统不因为错误引起诡异需求而宕机。

#4、监控设计 :在设计阶段就要考虑监控，而不是在部署完成后。

通过监控发现系统的可用性问题。

通过监控使系统自我诊断、自我修复成为可能。

通过监控确定系统可预留空间的使用情况。

通过监控掌握系统之间的交互关系，发现瓶颈

如果监控做的好，不仅能发现服务的死活，检查日志文件，还能收集系统相关的数据，评估终端用户的响应时间。如果系统和应用在设计和构建时就考虑好监控，那么即使不能自我修复，也至少可以自我诊断。

#5、多活数据中心设计

数据是否全部集中在一个数据中心?

读写是否分离?

是否所有的客户信息都共享同一个数据结构?

服务调用是否允许延时的存在

#6、采用成熟的技术

工程师倾向于学习和实施性感时髦的新技术。因为新技术可以降低成本、减少产品上市时间、提高性能。不幸的是，新技术也往往有较高的故障率。如果把新技术应用在架构的关键部分，可能会对可用性产生显著的影响。

最好争取在多数人采用该技术的时候进入，先把新技术用在对可用性要求不高的功能上，一旦证明它可以可靠地处理日常的交易，再将此技术移植到关键任务领域中去。

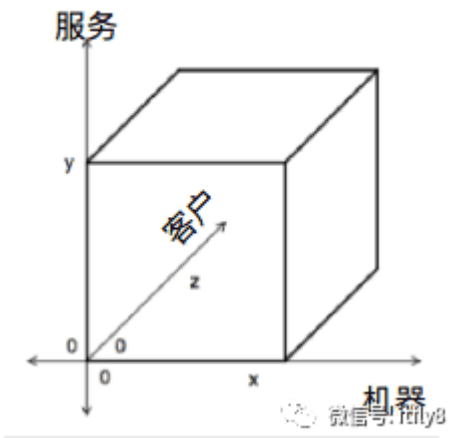
#7、故障隔离 ： 避免单一业务占用全部资源。避免业务之间的相互影响 2. 机房隔离避免单点故障。

不共享原则：理想情况是负载均衡、网络前端、应用服务器、数据库，绝不共享任何服务、硬件和软件。

不跨区原则：不同隔离区之间无通讯，所有服务调用必须发生在同一个故障隔离区。

#8、水平扩展

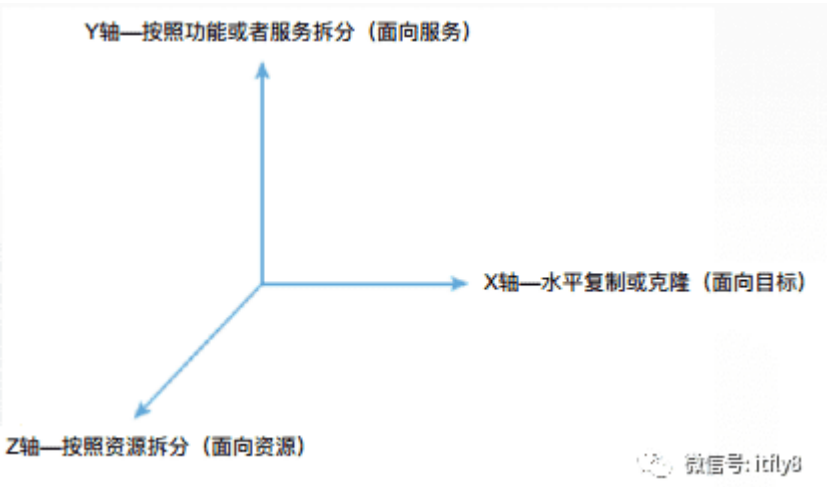
什么是水平可扩展？平台的水平扩展是指随着业务的发展，当需要扩大平台的服务能力时，不必重构软件系统，通过增加新的设备来满足业务增长的需要。

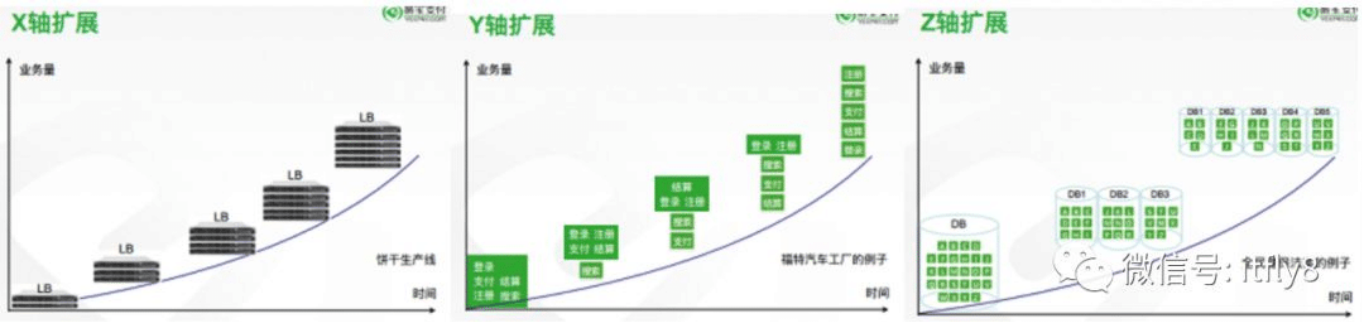


X轴扩展：服务器拆分。平台的服务能力可以在不改变服务的情况下，通过添加硬件设备来完成扩容。

Y轴扩展：数据库拆分。平台的服务能力通过不断地分解和部署服务来完成扩容。

Z轴扩展：功能拆分。平台的服务能力可以按照客户不断分解和部署来机器 完成容量的扩展。（比如按用户uid来分表分库等）





#9、非核心则购买

工程师往往有自己研发所有系统的冲动。

系统研发要投入资源，系统维护更要长期投入。

影响核心产品到市场的速度。

如果可以形成差异化的竞争优势，那么自己做，否则外购。

#10、使用商品化硬件

在大多数情况下，便宜的是最好的。

标准、低成本、可互换、易于商品化是商品化硬件的特征。如果架构设计得好，就可以通过购买最便宜的服务器轻松地实现水平扩展，前提是所有商品化硬件的总成本要低过高端硬件的总成本。

#11、快速迭代

小构建：小构建的成本较低，可以确保投资可以产生价值。

小发布：发布的失败率与变更数量相关，小发布失败率较低。

快试错：可依市场反馈，快速迭代，加快TTM，优化用户体验

#12、异步设计

同步系统中个别子系统出现故障会对整个系统带来影响。

同步系统中性能最慢的子系统成为整个系统性能的瓶颈。

同步系统中扩展性最差的子系统是整个系统扩展的瓶颈。

#13、无状态设计

1无状态的系统更利于扩展，更利于做负载均衡。

状态是系统的吞吐量、易用性、可用性、性能和可扩展性的大敌，要尽最大可能避免。

#14、前瞻性设计

Now： 目前正使用系统的架构、设计、能力、性能和扩展性。

Now+1: 下一代预研系统的架构、设计、能力、性能和扩展性。

Now+2: 下一代规划系统的架构、设计、能力、性能和扩展性

#15、自动化

设计和构建自动化的过程。如果机器可以做，就不要依赖于人. 人常犯错误，更令人沮丧的是，他们往往会以不同的方式多次犯同样的错误。

#三、应用服务拆分原则

应用拆分首先明确拆分目的和需求，然后制定拆分原则。

#1、拆分的目的

1)、人员的角度: 多人维护一个工程，从开发开发、测试、部署、上线，效率是极低的。项目一旦出现问题，都可能不知道问题是哪?

2)、 业务的角度：代码已经严重影响到业务的效率，每个业务有各自的需求，需要给自己应用部署，各自开发需求。

3)、从架构的角度：

应用已经无法满足非功能性需求：无法满足并发需求、安全性、扩展维护很麻烦需要梳理和抽取核心应用、公共应用，作为独立的服务下沉到核心和公共能力层，逐渐形成稳定的服务中心

总之，系统拆分是单体程序向分布式系统演变的关键一步，也是很重要的一步，拆分的好坏直接关系到未来系统的扩展性、可维护性和可伸缩性等，拆分工作不难理解，但是如何正确拆分、有什么样的方法和原则能帮助我们拆分得到一个我们理想中的系统：高可用、可扩展、可维护、可伸缩的分布式系统。

以下主要再从拆分需求、拆分原则和拆分步骤谈起：

#2、拆分需求

1、组织结构变化：从最初的一个团队逐渐成长并拆分为几个团队，团队按照业务线不同进行划分，为了减少各个业务系统和代码间的关联和耦合，几个团队不再可能共同向一个代码库中提交代码，必须对原有系统进行拆分，以减少团队间的干扰。

2、安全：这里所指的安全不是系统级别的安全，而是指代码或成果的安全，尤其是对于很多具有核心算法的系统，为了代码不被泄露，需要对相关系统进行模块化拆分，隔离核心功能，保护知识产权。

3、替换性：有些产品为了提供差异化的服务，需要产品具有可定制功能，根据用户的选择自由组合为一个完整的系统，比如一些模块，免费用户使用的功能与收费用户使用的功能肯定是不一样

的，这就需要这些模块具有替换性，判断是免费用户还是收费用户使用不同的模块组装，这也需要对系统进行模块化拆分。

4、交付速度：单体程序最大的问题在于系统错综复杂，牵一发而动全身，也许一个小的改动就造成很多功能没办法正常工作，极大的降低了软件的交付速度，因为每次改动都需要大量的回归测试确保每个模块都能正确工作，因为我们不清楚改动会影响到什么，所以需要做大量重复工作，增加了测试成本。这时候就需要对系统进行拆分，理清各个功能间的关系并解耦。

5、技术需求:

1）单体程序由于技术栈固定，尤其的是比较庞大的系统，不能很方便的进行技术升级，或者说对引入新技术或框架等处于封闭状态；每种语言都有自己的特点，单体程序没有办法享受到其它语言带来的便利；对应到团队中，团队技术相对比较单一。

2）相比于基于业务的垂直拆分，基于技术的横向拆分也很重要，使用数据访问层可以很好的隐藏对数据库的直接访问、减少数据库连接数、增加数据使用效率等；横向拆分可以极大的提高各个层级模块的重用性。

6、业务需求：由于业务上的某些特殊要求，比如对某个功能或模块的高可用性、高性能、可伸缩性等的要求，虽然也可以将单体整体部署到分布式环境中实现高可用、高性能等，但是从系统维护的角度来考虑，每次改动都要重新部署所有节点，显然会增加很多潜在的风险和不确定定性因素，所以有时候不得不选择将那些有特殊要求的功能从系统中抽取出来，独立部署和扩展。

#3.拆分原则

#3.1 业务原则

1、高内聚：满足单一职责原则：对于一个微服务而言，有限定的业务边界，可以帮助我们满足服务开发和交付的敏捷性；

2、服务粒度适中：以业务模型拆分、有适当的边界。

粗粒度优行原则：由服务提供方提供粗粒度的业务服务，封装数据及数据处理逻辑，屏蔽数据及业务规则，降低耦合度，提供更多业务价值；

适当的边界：关注微服务的功能范围，一个服务的大小应该等于满足某个特定业务能力所需要的大小；

3、业务分层原则：从整体规划上把业务分层，形成单向依赖，避免微服务之间的网状依赖关系；

4、可重用性拆分原则：将通用部分和专用部分分解为不同的应用。

1）若粗粒度服务不能满足重用需求，则拆分粗粒度服务，以增加重用；

2）非唯一依赖：至少被2个以上其它微服务依赖的功能模块，才有必要独立成一个微服务。

5、稳定性原则：将稳定部分和易变部分分离。将动态部分和静态部分分解为不同的元素；将机制和策略分离为不同的元素；将应用和服务分离。

#3.2 技术原则

1、低耦合：可独立部署

2、轻量级的通信机制

3、性能要求拆分原则：若粗粒度服务性能达不到性能需求，则适当拆分服务，以满足性能需求；

4、安全性拆分原则，若粗粒度服务所包含的所有处理不在同一个安全级别上，为满足安全性需求拆分服务形成细粒度服务；

#3.3 其他治理原则

1、演进式拆分

2、考虑团队人员结构

3、避免环形依赖和双向依赖

对于微服务组件拆分粒度应该是尽可能的拆小，但也不应该过分追求细粒度，要考虑适中不能过大或过小。按照单一职责原则和康威定律，在业务域、团队还有技术上平衡粒度。拆分后的代码应该是易控制，易维护的，业务职责也是明确单一的。

#四、架构设计的关键原则

一个好的设计：

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。

在开始设计之前，思考一下关键的原则，将会帮助你创建一个最小花费、高可用性和扩展性的架构。

分离关注点，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，

#职责单一：

每一个组件或者是模块应该只有一个职责或者是功能，功能要内聚。

#最小知识原则:

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。

不要重复你自己，你只需要在一个地方描述目的。例如，特殊的功能只能在一个组件中实现，在其他的组件中不应该有副本。

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。

低耦合、高内聚、防止变异（使用接口和适配器防止变异）、关注分离。

#1 关注分离

横向分层、纵向分区

(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) 使用装饰者，将所关注的事物（例如安全）置入Decrator对象中，Decorator对象包裹内部类并提取其服务，装饰者在EJB技术中被称为容器，EJB容器围绕内部对象的业务逻辑，在外部的装饰者中增添安全检查；

(3) 使用后便以和面向方面的技术（Aspect-oriented），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。

#2 关注点分离之道

好的架构设计必须把变化点错路有致地封装到软件系统的不同部分，为此，必须进行关注点分离。关注点相互分离，也就是说系统中的一部分发生变化，不会影响其他部分。即使需要改变，也能够清晰地识别出哪些部分需要改变。如果需要扩展架构，影响将会最小化，已经可以工作的每个部分都可以继续工作。

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。每个功能的完成，都是通过一系列职责组成的协作链条完成的，当不同职责被合理分离之后，为了实现新的功能只需构建新的协作链条，而需求变更页往往只会影响到少数职责的定义和实现。

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。不同的通用程度意味着变化的可能性不同，将通用性不同的部分分离有利于通用部分的重用，页便于对专用部分进行修改。

另外，还可以先考虑大粒度的子系统，而暂时忽略子系统是如何通过更小粒度的模块和类组成的。在实际中，软件架构师常常将系统划分为一组子系统，并为子系统定义明确的借口，其中的

细节将随其后的开发工作慢慢展开。

根据职责分离关注点、根据通用性分离关注点、根据不同粒度级别分离关注点是三种不同维度的思维方式。

#架构设计的非侵入性原则

那么什么是架构的侵入性呢?

所谓侵入性就是指的这个架构设计出来的部件对系统的影响范围，比如框架的侵入性就很高，因为在一个工程中引入一个框架，你的整个设计都必须围绕这个框架来进行，一旦使用了，框架的可替代性几乎为0，这样子就是搞侵入性。组件的侵入性就比较低,比如ibaties，他可以在任何java框架下使用，甚至可以和其他ORM组件共存，你仅仅需要引入，配置，然后就可以使用了，你也可以用其他的ORM替换他，所以.....这个体验应该是很愉快的。

所以话说回来说到如果我们在设计一个通用架构的时候就应该注意到这个一个非常重要的地方，除非我们只是自己拿来用用，否则我们不应该假设我们的设计的用户已经具备怎么怎么样的环境或者是需要做什么特殊的设计才能够使用。

这里打个比方，假如说我们在设计一个通用权限管理什么什么的时候我们就要想好，这是一个组件，还是框架，还是一个现成系统（复用通过改改代码实现，其实个人觉得这种设计很低级，虽然有的这样子的东西功能确实丰富）。确定了目标之后我们才好开始下一步，比如确定是一个框架的话可能发挥要自由一些，因为不需要高度的内聚，不过可能因为框架要设计的方方面面太多了，所以老是觉得个人的力量不足以搞这种东西出来。如果是组件的话就需要高度的内聚来实现非侵入式，比如引入DLL的时候还需要让所有页面继承自某个基类页就不算是一个good idear。

虽然话说得好听，不过我在自己做设计的时候还是常常因为功力不够造成一些侵入的现象，但是高内聚低耦合都是我们不断追求的目标，所以所有做设计的同学们一起努力吧

#参考文章

- https://blog.csdn.net/itfly8/article/details/105062228/