软件体系架构

1. 设计优良的架构

什么是架构

架构（Architecture）这个词语来自建筑学，放在软件领域中，相当于是一个隐喻，意味着它是一种High Level的设计指引、原则和具体的设计模型。架构有助于确保系统能够满足其利益相关人的关注点，在构想、计划、构建和维护系统时，架构有助于处理复杂性。

不同的人因为对软件系统的不同理解，对架构也有不同的定义。例如，IEEE对架构的定义为：

> 架构是以组件、组件之间的关系、组件与环境之间的关系为内容的某一系统的基本组织结构，以及指导上述内容设计与演化的原则。

而在BASS等人的著作Software Architecture in practice一书中，则将架构定义为：

> 某个软件或计算机系统的软件架构是该系统的一个或多个结构，每个结构均由软件元素、这些元素的外部可见属性、这些元素之间的关系组成。架构关注接口公开的一面，至于内部的实现细节则不属于架构的范畴。

《人月神话》的作者Frederick Brooks在著作Computer Architecture中则将计算机系统的架构定义为“一组最小的特征集，它们决定了哪些程序将运行，以及这些程序将得到什么结果。”

John Klein与David Weiss认为，待构建的对象或系统必须具有以下特征：

- 具备客户要求的功能
- 能够在要求的工期内安全地构建
- 性能足够好
- 可靠的
- 可用的，并且使用时不会造成伤害
- 安全的
- 成本是可以接受的
- 符合法规标准
- 将超越前人及其竞争者

满足上述要求的架构或许可以称之为接近完美的架构，但实际上很少有复杂系统能够很好地满足这些特征。那些追求完美主义的架构师，除了学院派的研究创作外，多数可能会在现实中碰壁，因为决定或约束软件系统的诸多因素在架构中是纠缠不清的，矛盾纠结的，因而无法做到尽善尽美。故而，架构的真谛是**trade off**，需要结合实际的用户场景权衡各种因素可能产生的影响，最终做出合理的决策。

> 错误的架构决策
>
> 我曾经做出一个错误的决策，是选择了MySQL作为我们大数据BI产品的数据集存储，幸好在产品研发前期我及时地发现了这个错误。因为我们的产品需要在各个层次（Web应用层，数据分析层，数据存储层）都需要支持水平伸缩，否则不足以应付海量数据存储以及实时分析高性能的需求。
>
> 由于我们选择了Spark作为产品的数据分析框架，最终，我们选择了设计为列式存储结构的Parquet文件作为数据集存储，它除了在容量大小与查询性能方面具备独特优势之外，还可以存储在HDFS中，满足了水平伸缩的需求。

《架构之美》认为：

> 架构观点中的常见思想是结构，每种结构都由各种类型的组件及其关系构成：它们如何组合、相互调用、通信、同步，以及进行其他交互。

架构师需要遵循“关注点分离”原则，在充分理解用户或利益相关者的需求前提下，对系统进行分解。分解的结构（可以是子系统、模块或组件）必须要遵循高内聚、低耦合的标准。这种分离可以是物理上的分离，也可以是逻辑上的分离，从而衍生出系统架构的物理架构与逻辑架构。

> 内聚与耦合
>
> 软件设计的关键品质是内聚和耦合。我们的目标是通过设计使系统的组件具备下列品质：
>
> - 高内聚（Strong cohesion）
>   内聚是一个测量指标，说明相关的功能如何聚集在一起，模块内的各部分作为一个整体工作得如何。内聚性是将模块粘成一个整体的胶水。弱内聚的模块是不良分解的信号。每个模块都必须具有清晰定义的角色，而不只是一堆不相干的功能。
>
> - 低耦合（Low coupling）
>   耦合是模块之间独立性的测量指标。在最简单的设计中，模块几乎没有什么耦合，所以彼此间的依赖关系较少。显然，模块不能完全解耦，否则它们将根本不能够一起工作！模块之间的联系有多种方式，有的是直接的，有的是间接的。模块可以调用其他模块中的函数，或被其他模块所调用。它可能使用其他模块提供的Web服务或设施，可能使用其他模块的数据类型，或提供某些数据让其他模块使用，如变量或文件。好的软件设计只提供绝对需要的依赖。

什么是优良的架构

简单的架构

> 让它尽可能简单，但不要过于简单。 ——爱因斯坦

架构是清晰的

简单的架构是容易理解容易维护的架构，这首先就要保证架构的清晰。就像设计一座城市一般，不能因为城市面积的庞大，人口的众多就失去其道路的清晰度，否则城市就会变成一座迷宫，让行人找不到正确的行进道路。混乱的架构体现为：

- 从每行程序、每个方法、每个组件看，代码都是混乱而粗糙地垒在一起的。
- 不存在一致性、不存在风格、也没有统一的概念能够将不同的部分组织在一起。
- 系统的控制流让人觉得不舒服，无法预测。
- 系统中有太多“坏味道”，整个代码集散发腐烂的味道。
- 数据很少放在使用它的地方。

缺乏清晰架构的系统是不可理解的，存在太高的复杂度，使得整个系统变得不可维护或者维护成本急剧增高。而“破窗效应”会导致架构日趋腐烂，最后变得无可挽回。

清晰的架构必然遵循了“高内聚、松耦合”的架构原则，每个子系统与模块职责都非常清晰，组件之间的通信机制是一致而统一的；从微观层面看，团队遵循了标准的编码规范，代码没有或者极少“坏味道”，至少这个“坏味道”是被隔离在极小的范围内。

小即是美

所谓的“小”，并非绝对的小，而是强调一种恰如其分的设计哲学。在开发过程中，每一次迭代的目标不宜设立过大，需小步前行，避免过度设计。在设计开发时，整个系统最好由松散耦合的细小模块组成。这些细小模块由于功能相对独立而单一，因而更易于理解。

在设计系统架构时，我们要注意克制做大做全的贪婪野心，尽力保证系统的小规模。

如何才能保证设计的系统足够小？首先，在设计思想上要确立“小即是美”的美学观，要清晰地辨别且能够欣赏小的**灵活之美**，**完整之美**与**轻盈之美**。只有在思想上认同它，你才能顺势而为；只有从心理上感受到这种美丽，你才能响应它的召唤。

灵活之美，在于它能快速地响应变化，这种变化可能是局部的，也可以是整个设计方向的改变。例如，在多数企业系统和互联网系统中，都需要分离Online和Offline任务，以指定不同的架构决策。又例如，我们可以设计独立的、具有最小功能子集的Batch Job来承担后台任务。这些Batch Job可以作为一个单独的应用程序执行在单独的进程中。一旦需求要求我们对设计做出改变，我们也能将修改控制在足够小的范围中，从而保证对整个系统不会带来巨大的影响。

若遵循EDA（Event Driven Architecture）模式，我们可以根据业务领域的不同，设计出功能最小完整的自治组件。组件之间的通信通过事件来传播，利用发布者/订阅者的方式解除组件之间的耦合；又或者利用消息传递来处理业务逻辑，例如在AKKA中，我们可以设计出灵活而小的Actor对象。而微服务（Micro Service）架构则从服务级别展现了设计的灵活之美。

轻盈之美，体现在它的功能并不臃肿，对外部的依赖较少，既容易在系统中快速引入，又不会使原有系统变得笨重，还能很方便地部署或者启动。专注的功能常能体现这种轻盈，例如Gradle专注于构建，Guice专注于依赖注入。

展现了轻盈之美的组件往往具有良好的可测试性。我们可以利用六边形架构将系统分隔为内、外两个边界，凡是系统对外的通信，皆通过端口（Port）和适配器（Adapter）完成，这样就能较好地解除对外部环境的依赖，提高系统的可测试性。而清晰的边界划分也是设计小组件的一种有效手段。

若对于框架或平台而言，则需要尽力降低框架或平台的侵入性。当年Rod Jonson之所以提出J2EE Without EJB，正是因为EJB的侵入性带来了诸多病症。

完整之美，在于它是自足的。完整并不意味着大而全，而在于它足够精简，没有冗余。当然，它同时应该是没有残缺的。残缺，意味着它无法在没有外部支持的情况下，完成自己应该完成的工作。这种美感符合“麻雀虽小，五脏俱全”的标准。Standalone的微服务，正好体现了这种自容器的完整之美。

小的益处还有一点，它可以使得我们在架构决策或技术选型时，可以变得更加从容。

譬如说，因为某些原因我们需要将整个企业系统从WebLogic上迁移到JBoss上，无疑，这是一个艰难的决定。即使在决策确定，要完成整个迁移也将是一个漫长的过程。如果系统是基于Micro Service的架构风格进行构建，每个服务根据各自情形选择自己的技术栈。倘若需要对某些服务进行技术栈迁移，相信这个问题不再变得棘手。——大象可以轻盈地跳舞，但付出的努力会百倍于一只敏捷的狐狸。

如今，Java已经发展到Java 8，引入的Lambda表达式等多个特性如此鲜嫩，让人垂涎不已。然而据我所知，国内多数企业的Java项目仍然停滞在JDK 6裹足不前。是JDK 8不够好吗？非也。盖因为求稳的他们仍然心存顾虑。即使Oracle号称这种JDK的迁移多么的平滑，多么的稳健，多数企业仍然不敢轻易做出迁移的决定。若因为迁移而带来未知的缺陷，可谓得不偿失。既然现在项目运行良好，何必冒此风险。

于是，我们这个行业因为系统的庞大而变得守旧老成，亦步亦趋。并非大家没有冒险的精神，实则是庞大的项目难以灵活地改变方向。倘若只是更新系统中的某一个库或框架，形势就截然不同了。记得在没有lambda的时代，当我们让客户看到了Guava的好处时，要引入Guava就轻而易举，真如顺水推舟了。

当我们发现某些功能具备独立和专注的特征时，都是可能做出小系统的机会。这些小系统并不一定是子系统或模块，它还可以是一个独立的应用或服务。

例如在一个税务系统中，需要生成复杂的税务报表。它的整个逻辑是相对独立的，不管是报表的动态生成，格式的转换，数据的查询以及流的处理和PDF文档的生成，都与系统其他部分关联不大。唯一可能与系统存在紧密关联的是数据库。但为了解决高峰期的性能问题，我们可以建立单独的数据提取器，又或引入流处理，定期将数据提取出来，放入内存数据库中。将这样相对独立的功能做成服务，就能够独立演化，并有效支持服务请求的可伸缩。这样的小型服务可以更灵活地应对变化。当我们发现内存数据库不能满足大量请求时，也可以轻而易举地将其迁移到NoSQL上，并根据数据的属性例如按照地域进行分区，支持水平扩展。

若要保证系统的小，我们还可以尝试使用脚本。在开发软件系统时，可以使用一些脚本语言来开发一些小工具，以应对灵活的需求变化，消除重复代码，实现某些步骤的自动化。例如用Groovy编写一些函数，用Ruby编写代码生成工具，又或者使用Gradle、SBT实现系统的自动部署，启动服务器等脚本。脚本语言具有很好的灵活性，而动态语言的特性也使得我们能够编写出短小精悍的超级小工具，甚至可以作为系统模块之间的粘合剂，如机器齿轮上的润滑油一般，让整个系统充满活力。

> **Unix的故事**

Unix的缔造者之一Dennis Ritchie就曾遭受过将系统做大做全的滑铁卢。他在贝尔实验室的第一个任务，是参与大项目Multics，即开发一个前所未有的、可以多人使用的、同时运行多个程序的操作系统。该项目由贝尔实验室、麻省理工学院和通用电气公司三方联合研制，但是由于设计过于复杂，迟迟拿不出成果，1969年贝尔实验室宣布退出。

痛定思痛，Dennis Ritchie和同事Ken Thompson之后在设计Unix时，就吸取了Multics设计复杂而导致失败的教训，提出了"保持简单和直接"（Keep it simple stupid）的原则，即所谓KISS原则。

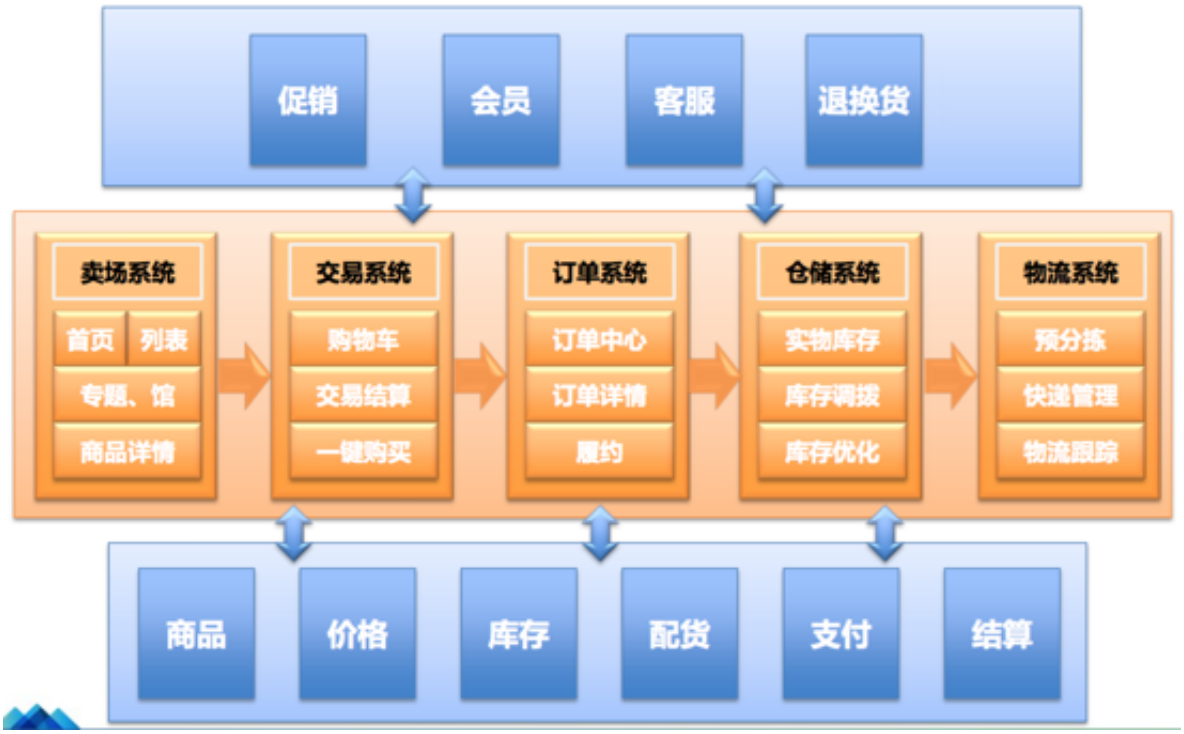
遵循KISS原则，整个Unix系统由许多小程序组成，每个小程序只能完成一个功能，任何复杂的操作都必须分解成一些基本步骤，由这些小程序逐一完成，再组合起来得到最终结果。表面上看，运行一连串小程序很低效。但是事实证明，由于小程序之间可以像积木一样自由组合，所以非常灵活，能够轻易完成大量意想不到的任务。而且，计算机硬件的升级速度非常快，所以性能也不是一个问题。另一方面，当把大程序分解成单一目的的小程序，开发变得容易，Unix在短短几个月内就问世。

案例：当当网的架构优化

随着业务逻辑越来越复杂，系统越来越多，相互依赖也越来越多。比如我的当当中就聚合了安全中心、用户、账户、订单、收藏夹、推荐等多维度的信息，需要调用多个系统服务。为了简化架构，采取的方案是将用户交互层面的前端页面与原有的后端系统拆分，并入前端的产品线，以便为用户提供更好的服务。

而后端系统之间的依赖关系也需要更为精细的分层定义，对于促销系统，需要会员系统、订单系统、价格系统提供基础数据；对于运费系统需要商品信息和配货数据，而在精准定位销售区域的前提下，库存只是配货的基础数据，配货系统负责判断是否有货，并根据配货结果计算预计送达时间。

事实上，这样的一种架构优化其实就是“关注点分离”的原则，而对于架构而言，则可以看做是一种纵向与横向的分解。横向的分解就是一种分层，更多是从应用的角度考虑；纵向的分解则是模块划分，采用的是业务的视角。



△ 当当的主要业务架构

针对当当的业务体系与质量属性需求，从纵向去分解，就可以去寻找一些公共的“关注点”，并将这些关注点往下压入到Infrastructure层，使其成为诸多业务均可以重用的平台或组件库，例如消息服务、安全服务、监控平台、日志平台、短信平台等。

横向地分解以业务划分为边界。例如区分核心业务和非核心业务，核心业务相对更加稳定。例如核心业务包括支付系统、交易系统、订单系统等。

针对业务对用户影响的程度不同，还对系统功能进行分级，不同级别的风险控制、资源占用、响应机制都各不相同。如下图所示：

级别	标准说明
一级系统	系统故障直接影响用户登录、浏览、检索、购买行为，影响下单支付，严重影响用户线上购买流程
二级系统	不影响订单收订完成，但对用户能否在承诺时间内顺利拿到购买商品产生重大影响的系统
三级系统	系统不稳定对线上业务系统无明显或即时的影响

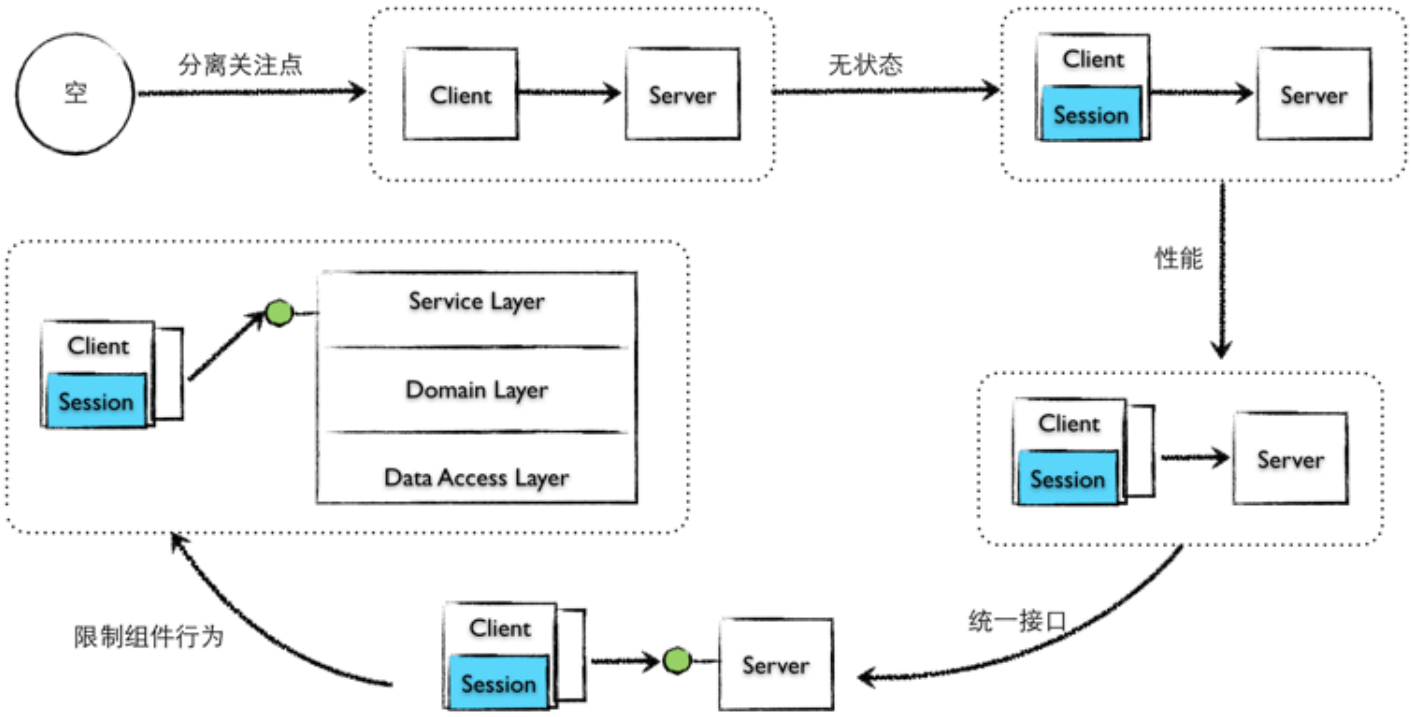
△ 当当网的系统分级

一致的架构

> 一致性是所有质量原则的根基。好的架构应该是直接的，人们掌握了部分系统后就可以推测出其他部分。 ——Brooks

要保证架构的一致性，可以考虑选择统一的架构风格，例如后面将要讲到的REST架构风格、基于消息的分布式架构风格、SOA架构风格等。当我们在进行设计决策时，任何决策都需要在整个设计的背景下做出，并满足系统的架构风格。

顶层设计的好风格和优雅会给详细设计带来好处，可以保证代码更加统一和整洁。与此同时，我们选择的技术平台和框架也会对系统设计带来规范与约束，使得设计变得更加地一致。下图就是不断地通过施加约束驱动出最终的架构设计风格：



△ 约束驱动出架构

一些框架自身就给出了好的设计原则，用在软件项目里就是好的设计指导。例如前端数据流控制框架Redux就设定了三条基本原则：

- 单一数据源：整个应用的 state 被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 store 中。
- State 是只读的：惟一改变 state 的方法就是触发 action，action 是一个用于描述已发生事件的普通对象。
- 使用纯函数来执行修改：为了描述 action 如何改变 state tree ，你需要编写 reducers。

架构的一致性还体现在对外暴露的服务或接口的一致性。接口要体现出抽象的意义，而一致的接口定义则有利于重用，并降低维护的成本。

> **Unix设计哲学**
>
> Doug McIlroy, Elliot Pinson和Berk Tague总结了两点Unix设计哲学：
>
> - 一个程序只做一件事情：这完全符合“单一职责原则”，每个程序变得很简单，而粒度小，通过组合可以完成更复杂的功能。
> - 程序的输入和输出都是统一的：如此，就能统一接口（uniform interface），支持自由的组合（composability），同时还能让程序之间完全解耦。
>
> 统一的接口是由输入与输出体现的。一个输入数据流stdin，两个输出数据流stdout与stderr，后者用于记录错误，以便于程序的诊断。如下图所示：

△ 统一的接口：输入流与输出流

在Unix中，这种统一接口可以被隐喻为文件（file），上图中的stdin、stdout与stderr都属于file descriptor，可以像操作文件那样读或写字节流（stream of bytes）。这里所谓的文件是一个宽泛的概念，可以是一个文件系统，也可以通过管道（pipe）将字节流传递到另一个进程，可以是Unix Socket，设备驱动，内核API以及TCP连接。