11 | 技术架构：作为开发，你真的了解系统吗？

2020-03-16 王庆友

架构实战案例解析 进入课程 >

讲述：王庆友
时长 17:32　大小 16.06M

你好，我是王庆友。从今天开始，我们就进入了技术架构模块，所以，这一讲，我想先跟你聊聊技术架构要解决什么问题。

对于开发人员来说，我们每天都在用技术。但你要知道，我们写的代码，其实只是系统的一小部分，我们了解的技术，也只是系统用到的一小部分。要深入掌握技术架构，我们就需要了解整体的系统。

面对一个复杂的系统，我想你可能经常会有以下困扰：

1. 不清楚系统整体的处理过程，当系统出问题时，不知道如何有针对性地去排查问题。

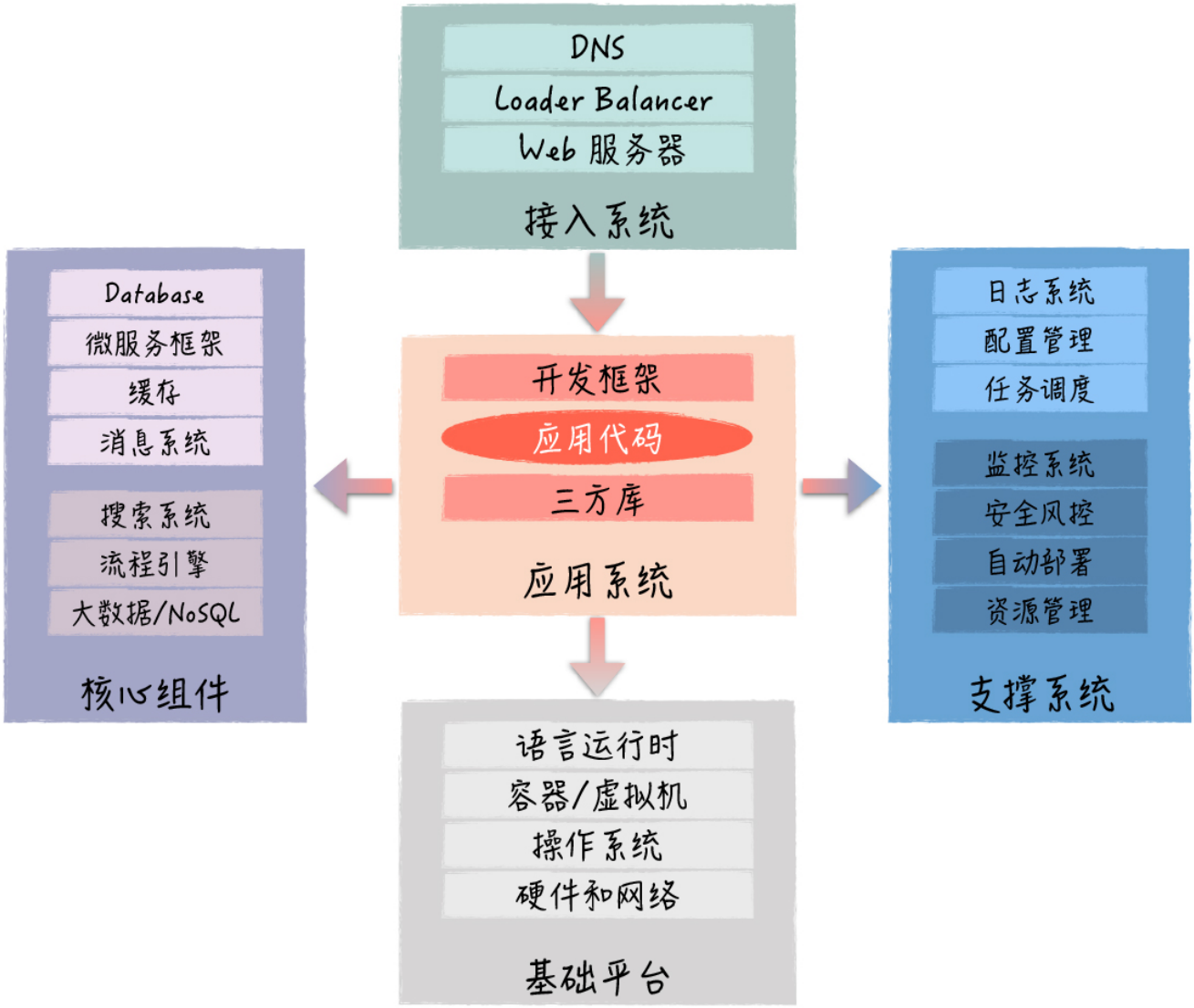
2. 系统设计时，经常忽视非业务性功能的需求，也不清楚如何实现这些目标，经常是付出惨痛的教训后，才去亡羊补牢。

不知你是否还记得，在第一讲“架构的本质”中，我已经说过，技术架构是从物理层面定义系统，并保障系统的稳定运行。那么今天，我会先分析下系统在物理上由哪些部分组成，让你可以从全局的角度看一个系统；然后再和你一起讨论，技术架构会面临哪些软硬件的挑战，以及它都有哪些目标，让你能够深入地了解技术架构。

系统的物理模型

对于大部分开发人员来说，我们主要的工作是写业务相关的代码，保证业务逻辑正确、业务数据准确，然后，这些业务代码经过打包部署后，变成实际可运行的应用。但我们写的代码只是系统的冰山一角，为了保证应用能正常运行，我们需要从**端到端系统的角度**进行分析。

我们先看下一个系统的具体组成，这里我为你提供了一个简化的系统物理模型，你可以了解一个系统大致包含哪些部分。



从用户请求的处理过程来看，系统主要包括五大部分。

首先是**接入系统**，它负责接收用户的请求，然后把用户的请求分发到某个 Web 服务器进行处理，接入系统主要包括 DNS 域名解析、负载均衡、Web 服务器这些组件。

接下来，Web 服务器会把请求交给**应用系统**进行处理。一般来说，我们是基于某个开发框架来开发应用的，比如 Java 应用一般是基于 Spring MVC 框架。

这个时候，开发框架首先会介入请求的处理，比如对 HTTP 协议进行解析，然后根据请求的 URL 和业务参数，转给我们写的业务方法。接下来，我们的应用代码，会调用开发语言提供的库和各种第三方的库，比如 JDK 和 Log4j，一起完成业务逻辑处理。在这里，我们会把开发框架、应用代码，还有这些库打包在一起，组成一个应用系统，作为独立的进程在 Web 服务器中进行部署和运行。

到这里，整个系统要做的事情就完了吗？

还没有呢，在我们的应用系统底下，还有**基础平台**，它由好几个部分组成：首先是各个语言的运行时，比如说 JVM；然后是容器或虚拟机；下面还有操作系统；最底下就是硬件和网络。

接入系统、应用系统、基础平台就构成一个最简单的系统。

在大多数情况下，应用系统还要借助大量外部的中间件来实现功能和落地数据，比如数据库、缓存、消息队列，以及 RPC 通讯框架等等。这里，我统称它们为**核心组件**，它们也是系统不可缺少的一部分。

除此之外，还有大量周边的**支撑系统**在支持应用的正常运行，包括日志系统、配置系统，还有大量的运维系统，它们提供监控、安全、资源调度等功能，它们和核心组件的区别是，这些系统一般不参与实际的用户请求处理，但它们在背后默默保障系统的正常运行。

到这里，你可以发现，一个端到端的系统是非常复杂的，它包含了大量的软硬件。为了保障我们的应用代码能够正常运行，我们就需要保证这里的每个组件不出问题，否则一旦组件出问题，很可能就导致系统整体的不可用。

技术架构的挑战

应用代码怎么组织（比如模块划分和服务分层），那主要是业务架构的事，这部分在前面我们已经讨论过很多了；而**技术架构的职责，首先是负责系统所有组件的技术选型，然后确保这些组件可以正常运行。**

我们知道，系统是由硬件和软件组成的。接下来，我们就分别从软硬件的角度来看下，技术架构都会面临什么挑战，我们需要如何应对。

硬件的问题

硬件是一个系统最基础的部分，负责真正干活的，但它有两方面的问题。

首先是硬件的处理能力有限。 对于服务器来说，它的 CPU 频率、内存容量、磁盘速度等等都是有限的。虽然说按照摩尔定律，随着制造工艺的发展，大概每隔 18 个月，硬件的性能

可以提升一倍，但还是赶不上快速增长的系统处理能力的要求，特别是目前许多互联网平台，面向的都是海量的 C 端用户，对系统处理能力的要求可以说是没有上限的。

从技术架构的角度，提升硬件的处理能力一般有两种方式。

Scale Up

也就是垂直扩展，简单地说就是**通过升级硬件来提升处理能力**。CPU 不够快，升级内核数量；内存不够多，升级容量；网络带宽不够，升级带宽。所以说，Scale Up 实际上是提升硬件的质量。

Scale Out

也就是水平扩展，**通过增加机器数量来提升处理能力**。一台机器不够，就增加到 2 台、4 台，以及更多，通过大量廉价设备的叠加，增强系统整体的处理能力。所以说，Scale Out 是提升硬件的数量。

垂直扩展是最简单的方式，对系统来说，它看到的是一个性能更强的组件，技术架构上不需要任何改造。如果碰到性能有问题，垂直扩展是我们的首选，但它有物理上的瓶颈或成本的问题。受硬件的物理限制，机器的性能是有天花板的；或者有时候，硬件超出了主流的配置，它的成本会指数级增长，导致我们无法承受。

水平扩展通过硬件数量弥补性能问题，理论上可以应对所有服务器处理能力不足的情况，并实现系统处理能力和硬件成本保持一个线性增长的关系。

但水平扩展对于系统来说，它看到的是多个组件，比如说多台 Web 服务器。如何有效地管理大量的机器，一方面，使得性能上可以实现类似 1+1=2 的效果；另一方面，要让系统各个部分能够有效地衔接起来，稳定地运行，这不是一件容易的事情。我们需要通过很复杂的技术架构设计来保障，比如说，通过额外的负载均衡，来支持多台 Web 服务器并行工作。

硬件的第二个问题是，**硬件不是 100% 的可靠，它本身也会出问题**。

比如说，服务器断电了，网络电缆被挖断了，甚至是各种自然灾害导致机房整体不可用。尤其是一个大型系统，服务器规模很大，网络很复杂，一旦某个节点出问题，整个系统都可能

受影响，所以，机器数量变多，也放大了系统出故障的概率，导致系统整体的可用性变差。**我们在做技术架构设计时，就要充分考虑各种硬件故障的可能性，做好应对方案。**比如说针对自然灾害，系统做异地多机房部署。

软件的问题

接下来我们说下软件的问题，这里的软件，主要说的是各种中间件和系统级软件，它们配合我们的应用代码一起工作。

软件是硬件的延伸，它主要是解决硬件的各种问题，软件通过进一步封装，给系统带来了两大好处。

首先是弥补了硬件的缺陷。比如 Redis 集群，通过数据分片，解决了单台服务器内存和带宽的瓶颈问题，实现服务器处理能力的水平扩展；通过数据多副本和故障节点转移，解决了单台服务器故障导致的可用性问题。

其次，封装让我们可以更高效地访问系统资源。比如说，数据库是对文件系统的加强，使数据的存取更高效；缓存是对数据库的加强，使热点数据的访问更高效。

但软件在填硬件的各种坑的同时，也给系统挖了新的坑。举个例子，Redis 集群的多节点，它解决了单节点处理能力问题，但同时也带来了新的问题，比如节点内部的网络有问题（即网络分区现象），集群的可用性就有问题；Redis 数据的多副本，它解决了单台服务器故障带来的可用性问题，但同时也带来了数据的一致性问题。

我们知道，分布式系统有个典型的 CAP 理论，C 代表系统内部的数据一致性，A 代码系统的可用性，P 代表节点之间的网络是否允许出问题，我们在这三者里面只能选择两个。对于一个分布式系统来说，网络出问题是比较常见的，所以我们首先要选择 P，这意味着我们在剩下的 C 和 A 之间只能选择一个。

CAP 理论只是针对一个小的数据型的分布式系统，如果放大到整个业务系统，C 和 A 的选择就更加复杂了。

比如有时候，我们直接对订单进行写库，这是倾向于保证数据一致性 C，但如果数据库故障或者流量太大，写入不成功，导致当前的业务功能失败，也就是系统的可用性 A 产生了问题。如果我们不直接落库，先发订单数据到消息系统，再由消费者接收消息进行落库，这样

即使单量很大或数据库有问题，最终订单还是可以落地，不影响当前的下单功能，保证了系统的可用性，但可能不同地方（比如缓存和数据库）的订单数据就有一致性的问题。

鱼和熊掌不能兼得，系统无法同时满足 CAP 的要求，我们就需要结合具体的业务场景，识别最突出的挑战，然后选择合适的组件，并以合理的方式去使用它们，最终保障系统的稳定运行，不产生大的业务问题。

技术架构的目标

好，现在你已经了解了系统的复杂性和软硬件的问题，那技术架构就要选择和组合各种软硬件，再结合我们开发的应用代码，来解决系统非功能性需求。

什么是系统非功能性需求呢？这是相对于业务需求来说的，所谓的业务需求就是保证业务逻辑正确，数据准确。比如一个订单，我们要保证订单各项数据是准确的，订单优惠和金额计算逻辑是正确的。而一个订单页面打开需要多少时间，页面是不是每次都能打开，这些就和具体的业务逻辑没有关系，属于系统非功能性需求的范畴。产品经理在一般情况下，也不会明确提这些需求。非功能性需求，有时候我们也称之为系统级功能，和业务功能相区分。

那对于一个系统来说，技术架构都要解决哪些非功能性需求呢？

系统的高可用

可用性的衡量标准是，系统正常工作的时间除以总体时间，通常用几个 9 来表示，比如 3 个 9 表示系统在 99.9% 的时间内可用，4 个 9 表示 99.99% 的时间内可用，这里的正常工作表示系统可以在相对合理的时间内返回预计的结果。

导致系统可用性出问题，一般是两种情况：

一种是软硬件本身有故障，比如机器断电，网络不通。这要求我们要么及时解决当前节点的故障问题，要么做故障转移，让备份系统快速顶上。

还有一种是高并发引起的系统处理能力的不足，软硬件系统经常在处理能力不足时，直接瘫痪掉，比如 CPU 100% 的时候，整个系统完全不工作。这要求我们要么提升处理能力，比如采取水平扩展、缓存等措施；要么把流量控制在系统能处理的水平，比如采取限流、降级等措施。

系统的高性能

我们这里说的高性能，并不是指系统的绝对性能要多高，而是系统要提供合理的性能。比如说，我们要保证前端页面可以在 3s 内打开，这样用户体验比较好。

保证合理的性能分两种情况：

一种是常规的流量进来，但系统内部处理比较复杂，我们就需要运用技术手段进行优化。比如针对海量商品的检索，我们就需要构建复杂的搜索系统来支持。

第二种是高并发的流量进来，系统仍旧需要在合理的时间内提供响应，这就更强调我们做架构设计时，要保证系统的处理能力能够整体上做水平扩展，而不仅仅是对某个节点做绝对的性能优化，因为流量的提升是很难准确预计的。

系统的可伸缩和低成本

系统的业务量在不同的时间点，有高峰有低谷，比如餐饮行业有午高峰和晚高峰，还有电商的大促场景。我们的架构设计要保证系统在业务高峰时，要能快速地增加资源来提升系统处理能力；反之，当业务低谷时，可以快速地减少系统资源，保证系统的低成本。

高可用、高性能、可伸缩和低成本，这些技术架构的目标不是孤立的，相互之间有关联，比如说有大流量请求进来，如果系统有很好的伸缩能力，它就能通过水平扩展的方式，保证系统有高性能，同时也实现了系统的高可用。如果系统的处理能力无法快速提升，无法保证高性能，那我们还是可以通过限流、降级等措施，保证核心系统的高可用。

我在前面也提到，这些目标很多时候会冲突，或者只能部分实现，**我们在做技术架构设计时，不能不顾一切地要求达到所有目标，而是要根据业务特点，选择最关键的目标予以实现。**

比如说，一个新闻阅读系统，它和订单、钱没有关系，即使短时间不可用，对用户影响也不大。但在出现热点新闻时，系统要能支持高并发的用户请求。因此，这里的设计，主要是考虑满足高性能，而不用太过于追求 4 个 9 或 5 个 9 的可用性。

总结

系统比我们想象的要复杂得多，这里，我和你分享了系统的物理模型，相信你不再局限于我们自己写的代码，而是对系统的整体结构有了更清晰的认识。

你还记得吗？在前面介绍业务架构时，我和你分享的是**系统 = 模块 + 关系**，而在这里介绍技术架构时，我和你分享的是**系统的物理模型**。

因为**业务架构解决的是系统功能性问题**，我们更多的是**从人出发**，去更好地理解系统；而**技术架构解决的是系统非功能性问题**，我们在识别出业务上的性能、可用性等挑战后，更多的是**从软硬件节点的处理能力出发**，通过合理的技术选型和搭配，最终实现系统的高可用、高性能和可伸缩等目标。通过这一讲的介绍，相信你现在对技术架构的目标和常见的解决手段，已经有了更深入的理解。

当然，针对这些不同的目标，技术架构处理的原则和手段也是不一样的。后面的几讲中，我会针对每个目标，为你具体展开介绍。

最后，给你留一道思考题：技术架构除了我在课程中说的几个目标之外，还有哪些目标呢？

欢迎在留言区和我互动，我会第一时间给你反馈。如果觉得有收获，也欢迎你把这篇文章分享给你的朋友。感谢阅读，我们下期再见。

精选留言 (5)

写留言

夜空中最亮的星（华仔...
2020-03-17

人也很重要

展开

孙同学
2020-03-16

除了高可用，高性能，可伸缩，低成本，应该还要易部署，好管理吧

孙同学
2020-03-16

https://www.processon.com/view/link/5e51378ce4b0c037b5f9d1e3 学习整理更新

Jeff.Smile
2020-03-16

要点总结：
业务架构解决的是系统功能性问题，我们更多的是从人出发，去更好地理解系统；
而技术架构解决的是系统非功能性问题，我们在识别出业务上的性能、可用性等挑战后，更多的是从软硬件节点的处理能力出发，通过合理的技术选型和搭配，最终实现系统的高可用、高性能和可伸缩等目标。...

展开

Mandalorian
2020-03-16

要相对低成本，可以横向的弹性伸缩。

展开