otto.de是德国的一家网上购物网站，本篇前半部分主要介绍了几个系统[**架构**](http://lib.csdn.net/base/architecture)以及它们的优缺点，后半部分主要讲解otto.de的[**微服务**](http://lib.csdn.net/base/microservice)架构。

　　在我们开始开发otto.de网上商店时，我们选择了分布式垂直架构。之前的工作经验告诉我们，一体化架构(monolithic architecture)不能够满足不断增长的需求。爆发式增长的数据，持续提高的负载和对系统的扩展，所有的这些强迫我们去重新思考网站的架构。

　　这篇文章将会描述我们的解决办法，还有我们这么做的原因。

　　一体化(Monoliths)

　　在项目刚开始的时候，团队通常会考虑使用什么编程语言和合适的架构。当谈到服务端应用时，[**Java**](http://lib.csdn.net/base/java)和[**spring**](http://lib.csdn.net/base/javaee)框架，Ruby on rails或者类似的框架通常会成为团队的选择。

　　选择了语言和框架后，经过一段时间的开发，一个简单的应用诞生了。与此同时，一体式架构(macro-architecture)毫无争议的成为了团队的选择。但是，这种架构的缺点也渐渐地浮出了水面：

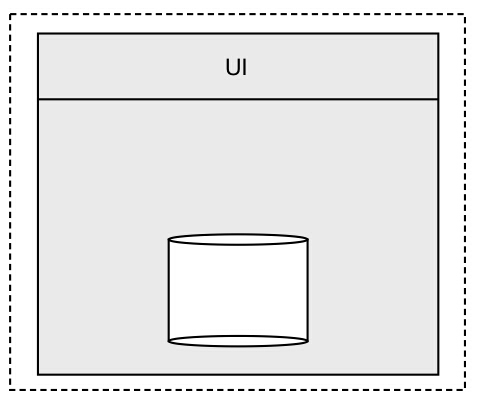
　　它导致了重量级微架构(a heavyweight Micro Architecture)

　　负载均衡限制了应用的可扩展性

　　系统的可维护性受到影响，尤其是那些大型应用

　　零停机部署(Zero downtime deployment)变得非常的困难，尤其是那些有状态的应用(stateful application)

　　多个团队开发效率低，并且需要额外的协调



　　当然，这并不是说一个新的应用一开始就变得巨大而混乱。在最开始的时候，新应用结构清晰，简单易懂，可扩展性也高，它能够很轻松的解决需求问题。在接下来的一段时间中，越来越多的代码被编写。为了应对日益增长的复杂度，系统被分层，抽象，模块，服务和框架被引入到系统中，最终变成了我们看到的样子。

　　即便是中型的应用(比如说一个50000的Java应用)，一体化架构也会渐渐地变得令人讨厌，更不用说那些对扩展性要求较高的应用了。

　　最终，曾经轻量简洁的应用将会变成下一代开发者的噩梦。

　　分而治之

　　问题的关键在于，如何避免这种类型的开发，并且将轻量应用好的那部分保留下来。换句话说，我们如何能够获得一个可持续发展的架构，这个架构在多年之后依旧能够让开发者保持高效开发。

　　在软件的开发过程中，有许多关于结构化代码的概念：函数，方法，类，库，框架等等。这些概念并不是程序运行所必须的，发明他们的原因是为了帮助开发者更好的理解他们的应用。

　　目前软件开发者已经理解了这些概念，一个问题接踵而来：为什么这些概念仅仅被应用于一个软件?是什么阻碍我们将应用拆分成多个低耦合的部分?

　　有三件事情我们需要牢记在心：

　　康威定律：软件开发最开始时仅有一个团队，根据康威定律，因此会产生一个应用。(译者注：可以参考图片理解)

　　初始消耗：部署应用，并让它运行起来似乎是一个非常简单的任务。实际上，你需要建立并管理VCS代码库，编译文件，构建管道，用于部署的程序，硬件，虚拟机，日志文件，监控软件等等。所有的这些都需要花费一定的时间去处理。

　　操作的复杂性：大型分布式系统比一个小型的负载均衡集群更难操作。

　　如果我们放手不管，由多个小型模块组成的系统并不会出现，一个巨大而混乱的系统将会取而代之。这时候，致命的问题已然出现，然而悔之晚矣。

　　正常情况下，一个系统是否需要被扩展，是否需要处理巨大的代码库在初期是非常清楚的。然后当你遇到以上提到的障碍，你要么没尝试解决他们，要么只能沉沦在无尽的苦果中。

　　在OTTO，在最开始的时候就花费了大量时间去建立4个跨职能的团队，根据前面提到的康威定律，一个项目属于4个团队，最终就能产生一个由4模块组成的应用，这样就避免了一体化应用的诞生。

　　因为我们之前操作过大型的一体化应用，操作的复杂性对于我们似乎是一个可以被解决的问题---操作200个一体化应用和操作200个更小型的系统没有太大的区别。

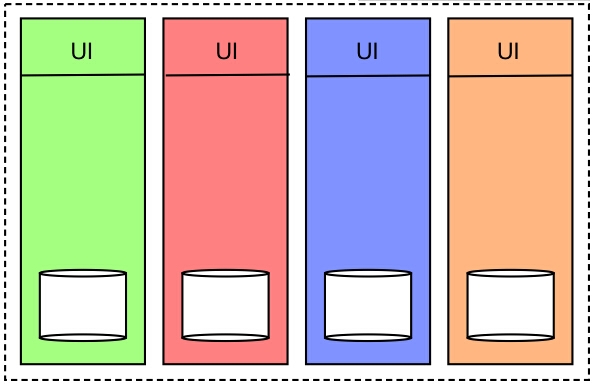
　　初始消耗可以通过标准化和自动化来克服。因为我们没有提到云服务，你还需要做相关的基础操作来启动自动化服务。虽然有些麻烦，但做过一次后，一切将自动化，你会获得巨大的便利。

　　可扩展性

　　如何将一体化应用转变为由多个小模块组成的应用?首先，让我们仔细想想一个应用能够从哪些维度进行扩展。

　　纵向分解(Vertical Decomposition)

　　纵向分解是一个非常自然而通用的方法，以至于常常被开发者所忽略。相比于把所有的功能集中到单一的应用中，我们将应用分解成了多个小模块，它们相互独立，互不影响。



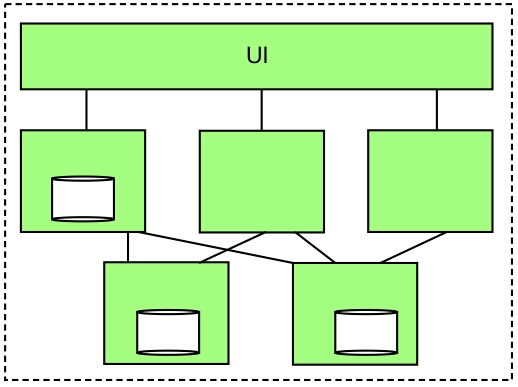
　　我们可以根据业务域来分解系统。举个例子来说，在otto.de我们就讲网上商城分解成了11个不同的垂直模块：后勤办公室，产品，订单等等。

　　每一个垂直模块属于一个单一团队，它们有独立的前端，后端和数据存储。在模块之间共享代码是严令禁止的。当然，在特殊的情况下，如果我们需要分享代码，我们会建立一个开源的项目来解决该问题。

　　因此，每个垂直模块是一个独立自主的系统，就像Stefan Tikov在Substainable Architecture中提到的那样。

　　分布式计算

　　一个垂直模块依旧可能成为一个相对大型的一体化应用，因此我们需要继续对垂直模块进行拆分。一种方法是将一个垂直模块分解成更多的垂直模块，另外一种方法是通过分布式计算将系统分解成多个模块，不同的是这些模块运行在他们自己的进程中，并且通过REST来传递信息。



　　在这种情况下，应用不仅仅被垂直分解，同时还会被水平分解。这种架构中，请求到达应用后，对请求的处理会被分布于多个模块中，然后每一模块产生的结果汇总成一个响应，发送回请求者。

　　这些模块并不会共享一个[**数据库**](http://lib.csdn.net/base/mysql)架构，因为这样做会导致模块间的紧密耦合：[**数据结构**](http://lib.csdn.net/base/datastructure)的改变会使得一个模块不能够被独立的部署。

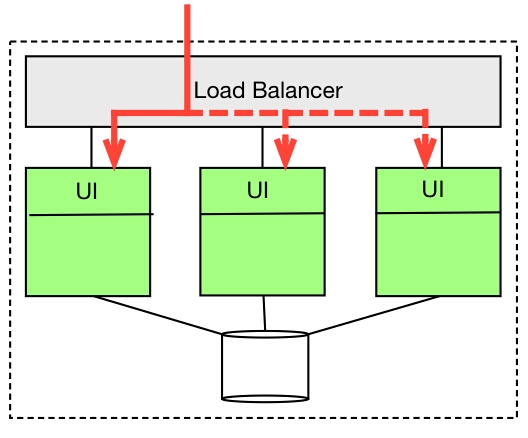
　　分片

　　当系统需要处理大量的数据，或者当一个分布式的应用被操作时，分片是很恰当的选择。比如说，分片非常适合向全球范围提供服务的应用。

　　因为我们暂时没有利用到分片这个概念，在这篇文章就不再详细描述了。

　　负载均衡

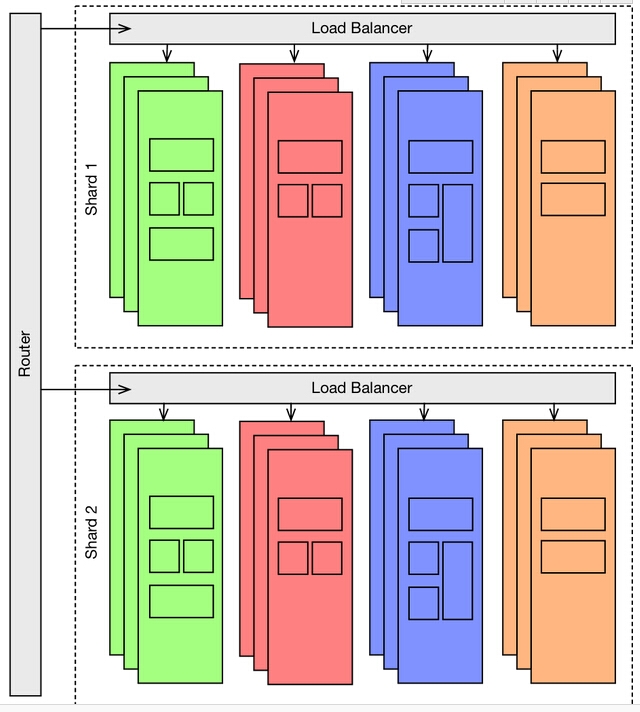
　　每当服务器承受不了巨大的负载压力时，负载均衡就会容重登场。通过对一个应用拷贝多次，同时利用负载均衡器将负载分解来缓解压力。



　　在负载均衡中，不同实例的应用通常会分时使用同一个数据库，数据库因此成为了系统的瓶颈，但是我们可以通过制定良好的扩展策略来避免这个问题。相比于关系数据库，NoSQL能够很好的处理扩展性的问题，这也就是NoSQL能够在软件世界中占据一亩三分地的原因之一。

　　最大的可扩展性

　　所有以上提到的办法能够组合在一起，能够达到任何级别的可扩展性。



　　当你没有相应的需求时，组合的结果会变得有点太复杂。幸运地是，开发者并不需要在一开始的时候就制定庞大的计划，相反，他们可以循序渐进，一步步朝着目标架构前进。

　　举个例子来说，在otto.de，架构一开始是4个垂直模块加上负载均衡，在过去的三年里，产生了更多的垂直模块。在此期间，某些模块变得非常的巨大笨重。因此，我们引入了微服务架构，同时通过扩展垂直模块来建立分布式计算。

　　微服务(Microservices)

　　微服务最近变得非常的流行。微服务是一种架构风格，它能够根据业务域将系统分解成多个细粒度，独立的模块。

　　在这种情况下，微服务可以是一个小的垂直模块，或者是分布式计算机构中的一个服务。与传统方法的不同之处在于应用的大小：一个微服务仅仅实现了一个业务域中的几个功能，它结构清晰，一个开发者能够很轻松的掌握它。

　　一个微服务非常的小，因此多个微服务能够运行在单一的服务器上。我们对“Fat JARs”有丰富的经验，通常能通过执行java –jar 来执行它们。如果需要的话，也能开启一个Jetty或者类似的服务器。

　　为了简化不同微服务的部署和操作问题，每一个服务器运行在独立的[**Docker**](http://lib.csdn.net/base/docker)容器中。

　　REST和微服务架构是一个很好的组合，它适合于构建大型的系统。一个微服务可以负责提供REST资源，超媒体(hypermedia)可以用来解决服务发现的问题，在涉及到接口的[**版本控制**](http://lib.csdn.net/base/git)，服务部署独立性的情况下，媒体类型(media type)有很大的帮助。

　　总而言之，微服务架构有许多的好处，比如说：

　　在微服务架构下进行开发是非常有趣的：每几周或者几个月，你就可能开始一个新的开发项目。

　　由于微服务非常小，微服务架构不需要重量级框架和过多的模板代码。

　　他们能够被独立的部署。因此持续交付或者持续部署变得非常的简单。

　　微服务架构能够支持多个独立的团体同时开发。

　　开发者能够为每一个服务选择最恰当的开发语言。不用担心对项目产生影响，开发者可以尝试新的语言或者框架。但是需要注意，这并不意味这你能够随意行使这项权利。

　　因为微服务足够小，只需消耗较少的资源就能将他们替换成新的项目。

　　这种架构的可扩展性相比于一体化架构显得非常的好，每一个服务都能被独立的扩展。

　　微服务架构遵守[**敏捷**](http://lib.csdn.net/base/agile)开发的原则。一个不能完全满足用户的新特性不仅可以被迅速的创建，而且还能够被快速的销毁。

　　宏架构和微架构(Macro- and Micro-Architecture)

　　在微服务架构中，哪一部分将难以改变?内部模块的扩展已经不再是关键问题，最难以改变的事情是有关微服务架构的决定，比如说如何将微服务整合到系统的方法，或者模块间传输信息协议的选择。

　　因此，otto.de严格区分了微架构(micro-architecture)和宏架构(macro-architecture)。微架构都是关于垂直架构或者微服务架构的内部结构，并且全部交由各自的团队全权处理。

　　但是，明确宏架构的大体方向是有价值的：

　　垂直分解：系统被分割成多个垂直模块，每一个模块完全属于一个特定的团队。模块与模块之间的信息传递禁止在用户请求的过程中进行，而必须在后台执行。

　　RESTful架构：不同服务之间的信息传递和整合只通过REST来执行。

　　零分享架构：服务间不会通过共享可变状态(mutable state)来进行信息交换或者分享信息。没有HTTP sessions，没有中央数据存储中心，没有共享代码。但是多个服务的实例之间有可能共享一个数据库。

　　数据管理：对于每一个数据节点，只有一个系统负责管理。其他的系统只能通过REST API读取数据，然后将需要的数据拷贝回自己的数据库。

　　我们的架构已经熬过了一轮软件开发周期，与此同时，我们开始标准化微服务使用的方式。

　　集成

　　目前为止，我已经详细说明了许多有关系统分解的内容。但是，用户体验是我们的系统的最终目标，我们希望我们的应用保持一致性，感觉就像是一个整体。

　　因此，问题来了：我们如何能够集成一个分布式的系统，同时让用户意识不到我们架构的分布特性。

　　超链接

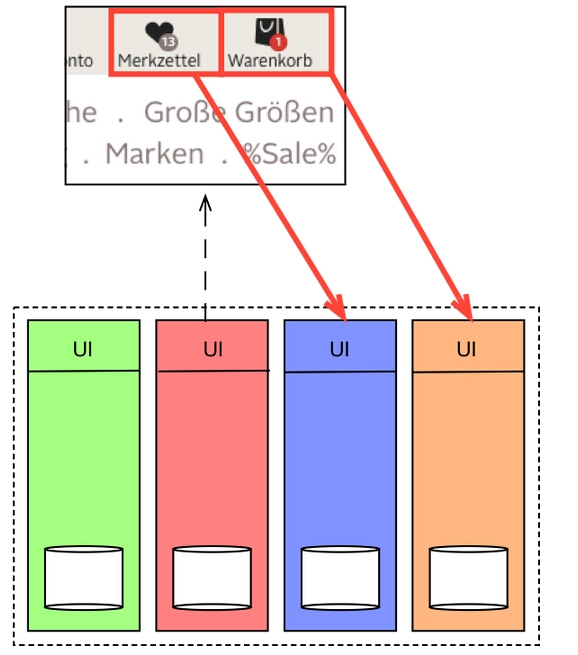
　　对于前端集成，最简单的办法就是使用超链接。

　　每一个服务负责不同的页面，页面的导航通过链接来实现。

　　AJAX

　　使用AJAX的目的也很明显，它能够通过[**JavaScript**](http://lib.csdn.net/base/javascript)重新加载页面的不同内容，并且将他们整合在特定的页面。

　　主要注意的是，服务之间涉及到的依赖非常的小，服务互相之间需要对使用的URL和媒体类型(media type)保持一致性。



　　资源服务器(Asset Server)

　　当然，图片在不同页面的显示也需要保持一致性。除此之外，分布式的服务需要对他们各自的Javascript库和版本保持一致。

　　为了保持一致性，在我们的系统中，静态资源，比如说CSS，JS和图片都是通过一个中央资源服务器来进行传输。

　　在垂直架构，共享资源的部署和版本控制是一个完全不同的话题，这需要一篇独立的文章来详细解释。在这里我们不做过多解释，我们只需要记住，共享资源的同时保持服务独立是具有非常大的挑战性。

　　Edge-Side Includes

　　有一种不太知名的方法，它能整合不同服务的资源到同一个站点。这种方法我们称之为Server-Side Includes或者是Edge-Side Includes。大多数的Web服务器或者反向代理都只支持这个功能。

　　这项技术非常的简单：一个服务插入插入一条带有URL的包含语句(include statement)。然后这个URL会被web服务器或者反向代理解析，代理根据URL获取到一个响应，然后用这个响应代替页面中的包含语句。

　　在我们的商城中，每一个页面都包含了来自搜索&导航服务(SAN)的导航信息：

　　...

　　...

　　反向代理(我们使用了Varnish)解析了页面，然后将URL分解出来，SAN根据URL提供相应的HTML片段。

　　然后Varnish代理用这个HTML片段取代包含语句，并将重新生成的页面发送回用户。

　　在这种方式下，用户根本意识不到页面是由多个来自不同服务的片段组成的。

　　数据拷贝(Data Replication)

　　以上提到的技术仅仅解决了前端集成问题，现在我们来谈谈服务端集成。不同的服务之间需要共同的数据，但是他们又不能共享同一个数据库，因此我们想了一些办法来处理这个问题。

　　数据拷贝就是一个办法。比如说，其他的服务需要关于产品的数据，它们就会定期的向负责产品数据的垂直模块(Product)请求数据，这样产品数据的更新能够很迅速的被其他服务检测到。

　　我们没有使用任何的消息队列来向客户端推送(push)数据。相反的，每当服务需要更新数据时，它们会轮询(poll)Atom Feed。

　　值得一提的是，某些不好的事情必须牺牲服务的可用性来避免，我们在开发过程中不得不面对这个矛盾。

　　没有远程服务调用(NO Remote Service Calls)

　　理论上来说，在某些情况下，我们可以避免数据的拷贝，这样服务就能够同步使用其他的服务了。一个购物篮并不需要保存额外的产品信息，相反它可以直接向产品模块请求数据。

　　我们并没有这么做，为什么呢?

　　当一个系统的主要功能依赖于其它系统时，系统的可[**测试**](http://lib.csdn.net/base/softwaretest)性受到影响。

　　一个缓慢的服务会影响到其它系统的请求，当请求越多，雪球越滚越大，最终影响到了整个系统的可用性。

　　系统的可扩展性受到了限制。

　　独立的服务部署变得非常的困难。

　　我们长期和垂直架构打交道，我们非常明确的知道，在早期的时候，严格的界限会使得微服务的开发，测试，迁移变得更加独立，方便。

　　经验教训

　　按照以上罗列的办法，经过三年的工作后，我们变得经验丰富。

　　回顾往事，如果我早点做这些事情，我们的一体化系统会变得更加的精细。现在，otto.de的未来属于微服务。