微服务架构简介

定义

微服务就是一些协同工作的小而自治的服务。深入阐述如下：

- 很小，专注于一件事情。在之前所说的单体系统中，会用模块来保证代码的内聚性，那么将这个理念应用在微服务设计上，“把因相同原因而变化的东西聚合在一起，而把因不同原因而变化的东西分离开来”，所以根据业务的边界来确定服务的边界。
- 自治性。一个微服务就是一个独立的实体，它可以独立地部署在PAAS 上，也可以作为一个操作系统进程存在。要尽量避免把多个服务部署到同一台机器上，这样能够大大简化分布式系统的构建。服务之间均通过网络调用进行通信，从而加强了服务之间的隔离性，避免了紧耦合。

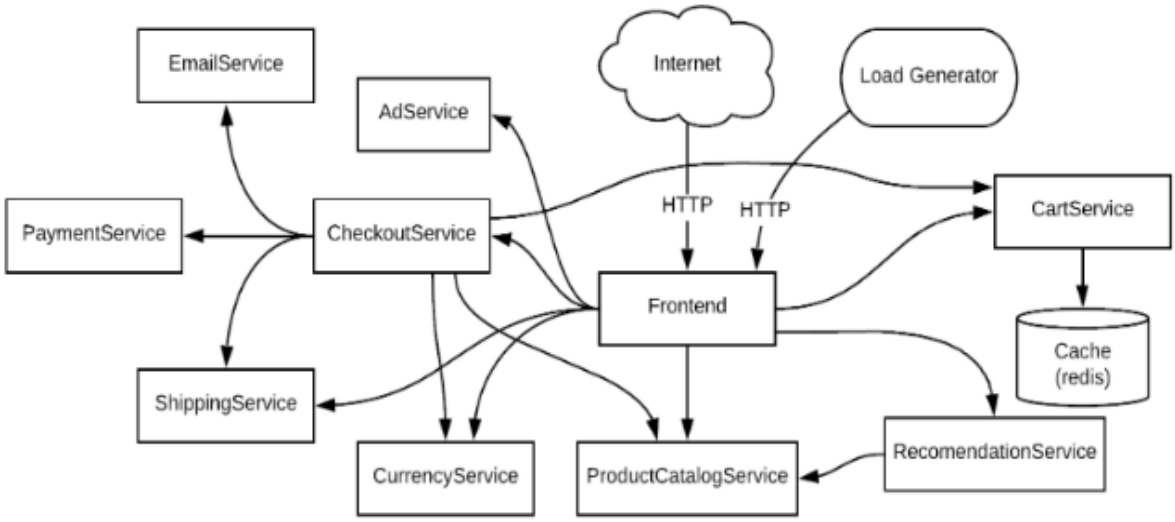
举个例子来说，考虑用来进行付款的服务。要成为微服务，它应该提供可以通过消息请求的付款功能，并将结果通过消息队列传递出去，但是它不应该有其他的功能，比如支付过程中要操作用户购买的库存，支付成功后要对订单进行操作。从技术角度上来说，微服务应该是概念上的部署的独立部件，彼此之间隔离 并且配备单独的持久化工具（数据库等）。 在微服务的概念基础上来定义微服务架构的概念：微服务架构是指一个分布式的应用程序，它的所有模块都是微服务。

优势

技术异构性

在一个由多个服务相互协作的系统中，可以在不同的服务中使用最适合该服务的技术。如果某一部分需要性能升级，可以使用性能更好的技术栈重新构建该部分。系统中的不同部分也可以使用不同的数据存储技术，比如对于社交网络来说，图数据库能够更好地处理用户之间地交互操作，但是对于用户发布的帖子而言，文档数据库可能是一个更好的选择。

而且微服务架构能帮助开发人员更快地使用新技术，并且理解这些新技术地好处。尝试这些新技术通常伴随着风险，尤其对于单体系统来说，采用一个新的语言、数据库或者框架都会对整个系统产生巨大的影响。而对于一个微服务架构 的系统来说，总存在一些地方可以去尝试新技术，在业务快速迭代的今天，能够快速的采用新技术的能力对于很多公司来说是很有价值的。如下图所示，这是 Google 开源的一个商城项目的架构，对于其中的每一个微服务都使用了不同的 语言和框架，Java、Python、Go、Scala、Erlang、C++、Ruby等等，充分体现了微服务架构在技术异构上的优势。



Service	Language
frontend	Go
cartservice	C#
productcatalogservice	Go
currencyservice	Node.js
paymentservice	Node.js
shippingservice	Go
emailservice	Python
checkoutservice	Go
recommendationservice	Python
adservice	Java
loadgenerator	Python/Locust

弹性

弹性工程学的一个关键概念是舱壁。如果系统中的一个组件不可用了，但并没有导致级联故障，那么系统的其他部分还可以正常运行。服务边界就是一个很显然的舱壁。在单块系统中，如果一个模块不可用，那么所有的功能都会不可用。对于单块服务的系统而言，可以通过将同样的实例运行在不同的机器上来降低功能完全不可用的概率，然而微服务系统本身就能够很好地处理服务不可用和功能降级问题。只要使用了分布式系统，网络和机器的故障就是不可避免的， 微服务架构的系统相对于传统分布式系统的优势是微服务架构的系统中会有服务的注册和发现中心，而且强调了每个服务调用其它微服务未成功时要有降级处理的措施，这个措施一般有多层，最后一层是不依靠网络的，能够一定程度上降低网络或机器故障对整体系统带来的影响。

可扩展性

使用较小的多个服务，当系统出现性能瓶颈时，只需要对需要扩展的服务进行扩展，这样就可以将那些不需要扩展的服务运行在更小的、性能稍差的硬件上， 这样可以更好的处理流量，而且也会节约成本。

简化部署

在微服务架构中，各个服务的部署是独立的，这样就可以更快地对特定部分的代码进行部署。如果真的出了问题，也会只影响一个服务，并且容易快速回滚， 这也意味着客户可以更快的体验到产品的新特性。这种架构也会更好的消除软件 发布过程中的种种障碍。

可组合性

面向服务的架构的主要优势是易于重用已有功能。而在微服务架构中，由于之前提到的技术异构性，这种优势进一步放大，根据不同的目的，人们可以通 过不同的方式使用同一个功能，在考虑客户如何使用软件这一点尤为重要，单纯考虑桌面网站或者移动 app 的时代已经过去了。现在需要考虑的应用程序种类包 括Web、客户端应用、移动端Web、移动端 app 和可穿戴设备等，针对每一种都应该考虑如何对已有的功能进行组合来实现这些应用。微服务架构可以辅助很好的实现这一点，每个微服务会开放很多接缝供系统内的其他微服务使用。当面对不同的情况时，可以使用不同的方法构建应用，而整体化的应用程序只提供一个粗粒度的接口表现在外部。

可替代性

即使最终通过微服务架构构建的系统的代码量还要超过一个单体应用，但是对于每个微服务来说，代码库可能只有几百行，所以重写或者移除一个或多个服务的阻碍也很小。

微服务设计指南

建模

什么是好的微服务

高内聚和松耦合在不同的上下文中被大量使用，尤其是在面向对象编程中。而这两个概念在微服务中有不同的含义。

松耦合

如果做到了服务之间的松耦合，那么修改一个服务就不需要修改另一个服务。使用微服务最重要的一点是，能够独立修改及部署单个服务而不需要修改系统的其他部分，这真的非常重要。什么会导致紧耦合呢？一个典型的错误是，使用紧耦合的方式做服务之间的集成，从而使得一个服务的修改会致使其消费者的修改。一个松耦合的服务应该尽可能少地知道与之协作的那些服务的信息。这也意味着，应该限制两个服务之间不同调用形式的数量，因为除了潜在的性能问题之外，过度的通信可能会导致紧耦合。

高内聚

我们希望把相关的行为聚集在一起，把不相关的行为放在别处。为什么呢？因为如果你要改变某个行为的话，最好能够只在一个地方进行修改，然后就可以尽快地发布。如果需要在很多不同的地方做这些修改，那么可能就需要同时发布多个微服务才能交付这个功能。 在多个不同的地方进行修改会很慢，同时部署多个服务风险也很高，这两者都是我们想要避免的。

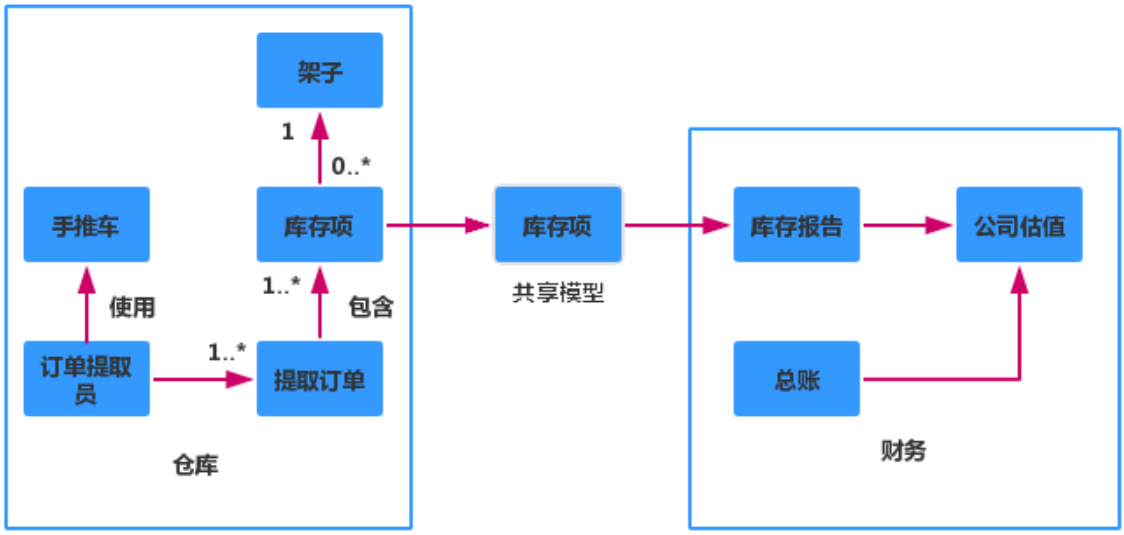
所以，找到问题域的边界就可以确保相关的行为能放在同一个地方，并且它们会和其他边界以尽量松耦合的形式进行通信。

限界上下文

Eric Evans的《领域驱动设计》一书引入的一个很重要的概念是限界上下文（bounded context）。作者认为任何一个给定的领域都包含多个限界上下文，每个限界上下文中的模型分成两部分，一部分不需要与外部通信，另一部分则需要。每个上下文都有明确的接口，该接口决定了它会暴露哪些模型给其他的上下文。另一个限界上下文的定义是：“一个由显式边界限定的特定职责。”如果你想要从一个限界上下文中获取信息，或者向其发起请求，需要使用模型和它的显式边界进行通信。使用细胞作为比喻：“细胞之所以会存在，是因为细胞膜定义了什么在细胞内，什么在细胞外，并且确定了什么物质可以通过细胞膜。”

举一个和运营相关的业务为例子。此业务涵盖了从仓储到前台、从财务到订单的所有元素。这些元素就是领域，尽管我们不一定要对所有的元素进行建模。我们首先尝试在领域中寻找限界上下文。在此例中，仓库负责管理发出去的订单（及退回的剩余产品），接收新到的库存，保证多个铲车能同时正常运行等。财务部的员工负责管理工资单和公司的账户，并生成重要的报表，这些报表的数量相当大。

对于此业务来说，财务部门和仓库就可以是两个独立的限界上下文。它们都有明确的对外接口（在存货报告、工资单等方面），也都有着只需要自己知道的一些细节（铲车、 计算器）。财务部门不需要知道仓库的内部细节。但它确实也需要知道一些事情，比如，需要知道库存水平以便于更新账户。下图展示了一个上下文图表示例。可以看到其中包含了仓库的内部概念，比如订单提取员、货架等。类似地，公司的总账是财务部必备的一部分，但是不会对外共享。为了算出公司的估值，财务部的雇员需要库存信息，所以库存项就变成了两个上下文之间的共享模型。然而，我们不会盲目地把库存项在仓库上下文中的所有内容都暴露出去。比如，尽管在仓库内部有相应的模型来表示库存项，但是我们不会直接把这个模型暴露出去。也就是对该模型来说，存在内部和外部两种表示方式。



有时候，同一个名字在不同的上下文中有着完全不同的含义。比如，退货表示的是客户退回的一些东西。在客户的上下文中，退货意味着打印运送标签、寄送包裹，然后等待退款。在仓库的上下文中，退货表示的是一个即将到来的包裹，而且这个包裹会重新入库。 退货这个概念会与将要执行的任务相关，比如我们可能会发起一个重新入库的请求。这个退货的共享模型会在多个不同的进程中使用，并且在每个限界上下文中都会存在相应的实体，不过，这些实体仅仅是在每个上下文的内部表示而已。

明白应该共享特定的模型，而不应该共享内部表示这个道理之后，就可以避免潜在的紧耦合风险。我们还识别出了领域内的一些边界，边界内部是相关性比较高的业务功能，从而得到高内聚。这些限界上下文可以很好地形成组合边界。

在同一个进程内使用模块来减少彼此之间的耦合也是一种选择。刚开始开发一个代码库的时候，这可能是比较好的办法。所以一旦你发现了领域内部的限界上下文，一定要使用模块对其进行建模，同时使用共享和隐藏模型。这些模块边界就可以成为绝佳的微服务候选。

一般来讲，微服务应该清晰地和限界上下文保持一致。如果服务边界和领域的限界上下文能保持一致，并且微服务可以很好地表示这些限界上下文的话，那么就已经迈出了高内聚低耦合的微服务架构的第一步。

很多时候直接使用微服务架构是不适合的，如果边界定义错误想要修正回来，会导致很多跨服务的修改，而这些修改的代价相当高。ThoughtWorks认为过早将一个系统划分成为微服务的代价非常高，尤其是在面对新领域时。很多时候，将一个已有的代码库划分成微服务，要比从头开始构建微服务简单得多。

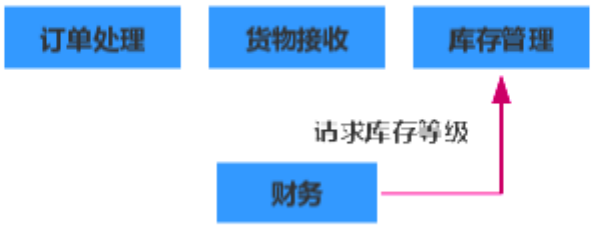
业务功能

当你在思考组织内的限界上下文时，不应该从共享数据的角度来考虑，而应该从这些上下文能够提供的功能来考虑。比如，仓库的一个功能是提供当前的库存清单，财务上下文能够提供月末账目或者为一个新招的员工创建工资单。为了实现这些功能，可能需要交换存储信息的模型，但是只考虑模型会导致贫血的、基于 CRUD（create，read， update，delete）的服务。所以首先要考虑“这个上下文是做什么用的”，然后再考虑它需要什么样的数据。

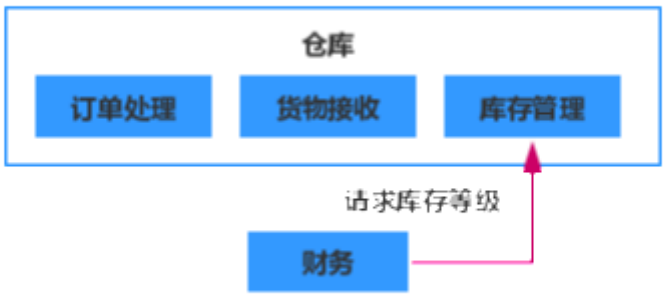
逐步划分上下文

一开始你会识别出一些粗粒度的限界上下文，而这些限界上下文可能又包含一些嵌套的限 界上下文。举个例子，你可以把仓库分解成为不同的部分：订单处理、库存管理、货物接 受等。当考虑微服务的边界时，首先考虑比较大的、粗粒度的那些上下文，然后当发现合适的缝隙后，再进一步划分出那些嵌套的上下文。到底是使用哪种方法呢。

如果订单处理、库存管理及货物接收是由不同的团队维护的，那么他们大概会希望这些服务都是顶层微服务。如下图所示。



另一方面，如果它们都是由一个团队管理的，那么嵌套式结构会更合理。其原因在于，组织结构和软件架构会互相影响。



关于业务概念的沟通

修改系统的目的是为了满足业务需求。我们会修改面向客户的功能。如果把系统分解成为限界上下文来表示领域的话，那么对于某个功能所要做的修改，就更倾向于局限在一个单 独的微服务边界之内。这样就减小了修改的范围，并能够更快地进行部署。

微服务之间如何就同一个业务概念进行通信，也是一件很重要的事情。基于业务领域的软件建模不应该止于限界上下文的概念。在组织内部共享的那些相同的术语和想法，也应该被反映到服务的接口上。以跟组织内通信相同的方式，来思考微服务之间的通信形式是非常有用的。事实上，通信形式在整个组织范围内都非常重要。

微服务集成

集成是微服务相关技术中最重要的一个。做得好的话，你的微服务可以保持自治性，你也可以独立地修改和发布它们。但做得不好的话会带来灾难。

理想的集成技术

微服务之间通信方式的选择非常多样化， 如SOAP、 RPC 、 REST 、 Protocol Buffers ， 理想的集成技术需要如下的特性。

- 避免破坏性修改

  有时候，对某个服务做的一些修改会导致该服务的消费方也随之发生改变。希望选用的技术可以尽量避免这种情况的发生。比如，如果一个微服务在一个响应中添加了一个字段，那么已有的消费方不应该受到影响。

- 保证API的技术无关性

  新的工具、框架、语言层出不穷，它们使我们的工作更高效。保证微服务之间通信方式的技术无关性是非常重要的。这就意味着不应该选择那种对微服务的具体实现 技术有限制的集成方式。

- 使服务易于消费方使用

  消费方应该能很容易地使用我们的服务。理想情况下，消费方应该可以使用任何技术来实现，从另一方面来说，提供一个客户端库也可以简化消费方的使用。但是通常这种库与其他我们想要得到的东西不可兼得。 举个例子，使用客户端库对于消费方来说很方便，但是会造成耦合的增加。

- 隐藏内部实现细节

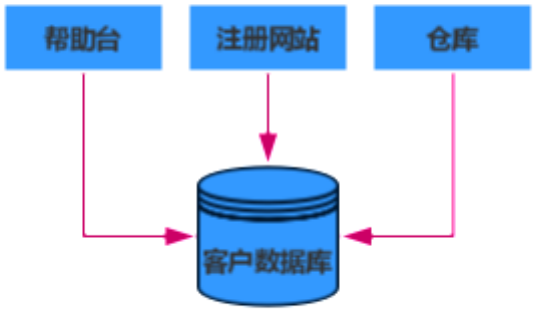
> 不希望消费方与服务的内部实现细节绑定在一起，因为这会增加耦合。与细节绑定意味着，如果想要改变服务内部的一些实现，消费方就需要跟着做出修改。这会增加修改的成本，而这恰恰是我们想要避免的。这也会导致为了避免消费方的修改而尽量少地对服务本身进行修改，而这会导致服务内部技术债的增加。所以，所有倾向于暴露内部实现细节的技术都不应该被采用。

我们举一个创建客户的业务作为例子帮助学习。创建客户这个业务，乍一看似乎就是简单的 CRUD 操作，但对于大多数系统来说并不止这些。添加新客户可能会触发一个新的流程，比如进行付账设置、发送欢迎邮件等。而且修改或者删除客户也可能会触发其他的业务流程。

共享数据库

目前为止，我和同事在业界所见到的最常见的集成形式就是数据库集成。使用这种方式时，如果其他服务想要从一个服务获取信息，可以直接访问数据库。如果想要修改，也可以直接在数据库中修改。这种方式看起来非常简单，而且可能是最快的集成方式，这也正 是它这么流行的原因。

下图展示了注册部分的用户界面，它直接使用 SQL 在数据库中创建用户。还可以看到， 呼叫中心应用程序可以直接运行 SQL 来查看和编辑数据库中的数据。仓库通过查询数据库 来显示更新后的客户订单信息。这是一种非常普通的模式，但实践起来却困难重重。



首先，这使得外部系统能够查看内部实现细节，并与其绑定在一起。存储在数据库中的数据结构对所有人来说都是平等的，所有服务都可以完全访问该数据库。如果我决定为了更好地表示数据或者增加可维护性而修改表结构的话，我的消费方就无法进行工作。数据库是一个很大的共享 API，但同时也非常不稳定。如果我想改变与之相关的逻辑，比如说帮助台如何管理客户，这就需要修改数据库。为了不影响其他服务，我必须非常小心地避免修改与其他服务相关的表结构。这种情况下，通常需要做大量的回归测试来保证功能的正确性。

其次，消费方与特定的技术选择绑定在了一起。可能现在来看，使用关系型数据库做存储是合理的，所以我的消费方会使用一个合适的驱动（很有可能是与具体数据库相关的）来 与之一起工作。说不定一段时间之后我们会意识到，使用非关系型数据库才是更好的选择。如果消费方和客户服务非常紧密地绑定在了一起，那么能够轻易替换这个数据库吗？ 正如前面所讨论的，隐藏实现细节非常重要，因为它让我们的服务拥有一定的自治性，从而可以轻易地修改其内部实现。

最后，让我们考虑一下行为。肯定会有一部分逻辑负责对客户进行修改。那么这个逻辑应该放在什么地方呢？如果消费方直接操作数据库，那么它们都需要对这些逻辑负责。对数据库进行操作的相似逻辑可能会出现在很多服务中。如果仓库、注册用户界面、呼叫中心都需要编辑客户的信息，那么当修复一个 bug 的时候，你需要修改三个不同的地方，并且对这些修改分别做部署。

还记得前面提到过的关于好的微服务的核心原则吗？没错，就是高内聚和低耦合。但是使用数据库集成使得这两者都很难实现。服务之间很容易通过数据库集成来共享数据，但是无法共享行为。内部表示暴露给了我们的消费方，而且很难做到无破坏性的修改，进而不 可避免地导致不敢做任何修改，所以无论如何都要避免这种情况。

在本章的剩余部分中，我们会介绍服务之间不同风格的集成方式，这些方式都可以保证服务的内部实现得以隐藏。

同步和异步

在介绍具体的技术选择之前，让我们先就服务如何协作这个问题做一些讨论。服务之间的通信应该是同步的还是异步的呢？这个基础性的选择会不可避免地引导我们使用不同 的实现。

如果使用同步通信，发起一个远程服务调用后，调用方会阻塞自己并等待整个操作的完 成。如果使用异步通信，调用方不需要等待操作完成就可以返回，甚至可能不需要关心这 个操作完成与否。

同步通信听起来合理，因为可以知道事情到底成功与否。异步通信对于运行时间比较长的 任务来说比较有用，否则就需要在客户端和服务器之间开启一个长连接，而这是非常不实 际的。当你需要低延迟的时候，通常会使用异步通信，否则会由于阻塞而降低运行的速 度。对于移动网络及设备而言，发送一个请求之后假设一切工作正常（除非被告知不正 常），这种方式可以在很大程度上保证在网络很卡的情况下用户界面依然很流畅。

这两种不同的通信模式有着各自的协作风格，即请求/响应或者基于事件。对于请求/响应来说，客户端发起一个请求，然后等待响应。这种模式能够与同步通信模式很好地匹配，但异步通信也可以使用这种模式。我可以发起一个请求，然后注册一个回调，当服务端操作结束之后，会调用该回调。对于使用基于事件的协作方式来说，情况会颠倒过来。客户端不是发起请求，而是发布一个事件，然后期待其他的协作者接收到该消息，并且知道该怎么做。我们从来不会告知任何人去做任何事情。基于事件的系统天生就是异步的。整个系统都很聪明，也就是说，业务逻辑并非集中存在于某个核心大脑，而是平均地分布在不同的协作者中。基于事件的协作方式耦合性很低。客户端发布一个事件，但并不需要知道谁或者什么会对此做出响应，这也意味着，你可以在不影响客户端的情况下对该事件添加新的订阅者。

远程过程调用（RPC）

远程过程调用允许你进行一个本地调用，但事实上结果是由某个远程服务器产生的。RPC 的种类繁多，其中一些依赖于接口定义（SOAP、Thrift、protocol buffers 等）。不同的技术栈可以通过接口定义轻松地生成客户端和服务端的桩代码。举个例子，我可以让一个 Java 服务暴露一个 SOAP 接口，然后使用 WSDL（Web Service Definition Language，Web 服务 描述语言）定义的接口生成 .NET 客户端的代码。其他的技术，比如 Java RMI，会导致服 务端和客户端之间更紧的耦合，这种方式要求双方都要使用相同的技术栈，但是不需要额 外的共享接口定义。然而所有的这些技术都有一个核心特点，那就是使用本地调用的方式 和远程进行交互。
有很多技术本质上是二进制的，比如 Java RMI、Thrift、protocol buffers 等，而 SOAP 使用 XML作为消息格式。有些 RPC 实现与特定的网络协议相绑定（比如 SOAP 名义上使用的 就是 HTTP），当然不同的实现会使用不同的协议，不同的协议可以提供不同的额外特性。 比如 TCP 能够保证送达，UDP 虽然不能保证送达但协议开销较小，所以你可以根据自己 的使用场景来选择不同的网络技术。
那些 RPC 的实现会帮你生成服务端和客户端的桩代码，从而让你快速开始编码。基本不用花时间，我就可以在服务之间进行内容交互了。

但是RPC的实现也确实存在一些问题。

- 技术的耦合

  有一些 RPC 机制，如 Java RMI，与特定的平台紧密绑定，这对于服务端和客户端的技术 选型造成了一定限制。Thrift 和 protocol buffers 对于不同语言的支持很好，从而在一定 程度上减小了这个问题的影响。但还是要注意，有时候 RPC 技术对于互操作性有一定的 限制。 从某种程度上来讲，这种技术上的耦合也是暴露内部实现细节的一种方式。

- 本地调用和远程调用并不相同

  RPC 的核心想法是隐藏远程调用的复杂性。但是很多 RPC 的实现隐藏得有些过头了，进 而会造成一些问题。使用本地调用不会引起性能问题，但是 RPC 会花大量的时间对负荷进 行封装和解封装，更别提网络通信所需要的时间。这意味着，要使用不同的思路来设计远 程和本地的 API。简单地把一个本地的 API 改造成为跨服务的远程 API 往往会带来问题。 最糟的情况是，开发人员会在不

知道该调用是远程调用的情况下对其进行使用。

你还需要考虑网络本身。即使客户端和 服务端都正常运行，整个调用也有可能会出错。这些错误有可能会很快发生，也有可能会 过一段时间才会显现出来，它们甚至有可能会损坏你的报文。你应该做出一个假设：有一 些恶意的攻击者随时有可能对网络进行破坏，因此网络的出错模式也不止一种。服务端 可能会返回一个错误信息，或者是请求本身就是有问题的。你能够区分出不同的故障模 式吗？如果可以，分别如何处理？如果仅仅是因为远程服务刚刚启动，所以响应才会有点慢，该怎么办？

如果要选用 RPC 这种方式的话，需要注意一些问题：不要对远程调用过度抽象，以至于网络因素完全被隐藏起来；确保你可以独立地升级服务端的接口而不用强迫客户端升级，所以在编写客户端代码时要注意这方面的平衡；在客户端中一定不要隐藏是在做网络调用这个事实。

REST

REST 是受 Web 启发而产生的一种架构风格。REST 风格包含了很多原则和限制，但是这里我们仅仅专注于，如何在微服务的世界里使用 REST 更好地解决集成问题。REST 是 RPC 的一种替代方案。

其中最重要的一点是资源的概念。资源，比如说 Customer，处于服务之内。服务可以根据请求内容创建 Customer 对象的不同表示形式。也就是说，一个资源的对外显示方式和内部存储方式之间没有什么耦合。举个例子，客户端可能会请求一个 Customer 的 JSON 表示形 式，而 Customer 在内部的存储方式可以完全不同。一旦客户端得到了该 Customer 的表示， 就可以发出请求对其进行修改，而服务端可以选择应答与否。

REST 本身并没有提到底层应该使用什么协议，尽管事实上最常用 HTTP。我以前也见过使用其他协议来实现 REST 的例子，比如串口或者 USB，当然这会引入大量的工作。 HTTP 的一些特性，比如动词，使得在 HTTP 之上实现 REST 要简单得多，而如果使用其他协议的话，就需要自己实现这些特性。

HTTP 本身提供了很多功能，这些功能对于实现 REST 风格非常有用。比如说 HTTP 的动词（如 GET、POST 和 PUT）就能够很好地和资源一起使用。事实上，REST 架构风格声明了一组对所有资源的标准方法，而 HTTP 恰好也定义了一组方法可供使用。GET 使用幂等的方式获取资源，POST 创建一个新资源。这就意味着，我们可以避免很多不同版本的 createCustomer 及 editCustomer 方法。相反，简单地 POST 一个 Customer 的表示到服务 端，然后服务端就可以创建一个新的资源，接下来可以发起一个 GET 请求来获取资源的表示。从概念上来说，对于一个 Customer 资源，访问接口只有一个，但是可以通过 HTTP 协议的不同动词对其进行不同的操作。

HTTP 周边也有一个大的生态系统，其中包含很多支撑工具和技术。比如 Varnish 这样的 HTTP 缓存代理、mod_proxy 这样的负载均衡器、大量针对 HTTP 的监控工具等。这些组件可以帮助我们很好地处理 HTTP 流量，并使用聪明的方式对其进行路由，而且这些操作基本上都对终端用户透明。HTTP 还提供了一系列安全控制机制供我们直接使用。从基本认证到客户端证书，HTTP 生态系统提供了大量的工具来简化安全性处理。

但同时，基于HTTP的REST也有其缺点。

从易用性角度来看，基于 HTTP 的 REST 无法帮助你生成客户端的桩代码.

有些 Web 框架无法很好地支持所有的 HTTP 动词。这就意味着你很容易处理 GET 和 POST 请求，但是 PUT 和 DELETE 就很麻烦了。

性能上也可能会遇到问题。基于 HTTP 的 REST 支持不同的格式，比如 JSON 或者二进制， 所以负载相对 SOAP 来说更加紧凑，当然和像 Thrift 这样的二进制协议是没法比的。在要求低延迟的场景下，每个 HTTP 请求的封装开销可能是个问题。

基于事件的异步协作方式

事件驱动的系统看起来耦合非常低，而且伸缩性很好。但是这种编程风格也会带来一定的复杂性，这种复杂性并不仅仅包括对消息的发布 订阅操作。举个例子，考虑一个非常耗时的异步请求 / 响应，需要考虑响应返回时需要怎 么处理。该响应是否回到发送请求的那个节点？如果是的话，节点服务停止了怎么办？如果不是，是否需要把信息事先存储到某个其他地方，以便于做相应处理？

需要确保各个流程有很好的监控机制，并考虑使用关联 ID，这种机制可以帮助你对跨进程的请求进行追踪。

部署

部署一个单块系统的流程非常简单。然而在众多相互依赖的微服务中，部署却是完全不同的情况。

持续集成

CI（持续集成）和CD（持续交付）已经出现很多年了，但是在微服务架构中要做CI 和 CD，还是有很多不同之处的。 当持续集成遇到微服务时，需要考虑的如何把 CI 的构建和每个微服务联系起来，即要在微服务、CI 构建和源代码之间建立合适的映射。现在比较主流的方法是让每个微服务都有自己的代码库，分别与相应的 CI 绑定。当对代码库进行修改时，可以只运行相关的构建及其中的测试。最终只会得到一个需要部署的构 建物，代码库与团队所有权的匹配程度也更高了。如果你对一个服务负责，就应该同时对相关的代码库和构建负责。 Jez Humble 和 Dave Farley 对持续交付是这样描述的：CD 能够检查每次提交是否达到了部署到生产环境的要求，并持续地把这些信息反馈给我们，它会把每次提交当初候选发布版本来对待。整个过程是一个流水线：编译及快速测试、 耗时测试、用户验收测试、性能测试，最后部署到生产环境。在微服务的世界，每个微服务的构建物都沿着上述的流水线进行移动，只是构建物的大小和形态会有很大的差别。

环境

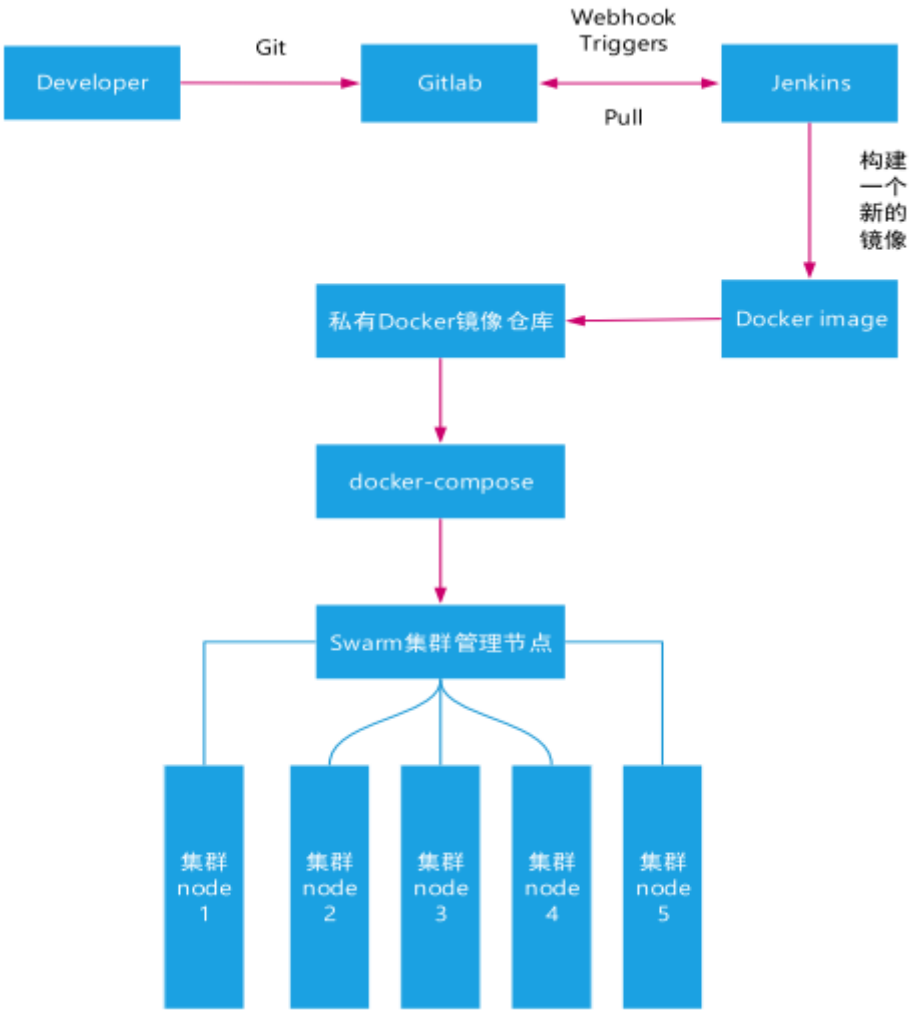
当软件在 CD 流水线的不同阶段之间移动时，它也会被部署到不同的环境中。如果考虑上图中所示的构建流水线，其中起码存在 4 个环境：一个用来运行耗时测试，一个用来做 UAT，一个用来做性能测试，另一个用于生产环境。我们的微服务构建物从头到尾都是一样的，但环境不同。至少它们的主机是隔离的，配置也不一样。而事实上情况往往会复杂 得多。这些环境之间的不同可能会引 起一些问题。管理单块系统的环境很具有挑战性，尤其是当你对那些很容易自动化的系统没有访问权的时候。当你需要对每个微服务考虑多个环境时，事情会更加艰巨。

服务配置

服务需要一些配置。理想情况下，这些配置的工作量应该很小，而且仅仅局限于环境间的不同之处，比如用来连接数据库的用户名和密码。应该最小化环境 间配置的差异。如果你的配置修改了很多服务的基本行为，或者不同环境之间的 配置差异很大，那么就只能在一套环境中发现某个特定的问题，这是很不合理的事情。而且服务的配置信息相对来说是容易改变的，可以使用配置服务器和 git 上的配置仓库结合来存储所有的服务配置信息，当配置信息有变更时，能够让服 务自动刷新配置，这样就可以避免配置信息改变使也需要重新部署服务。

云平台

Docker 是构建在轻量级容器之上的平台，可以处理大部分与容器管理相关 的事情，还可以缓解过多服务进行本地开发和测试的问题。Docker 并不能解决所有的问题，还需要一些工具来帮助跨多台主机管理 Docker 实例上的服务， 调度层的一个关键需求是，当你向其请求一个容器时会帮你找到相应的容器运行它。在这个领域，Kubernetes 已经成为了工业界的主流选择。



规模化微服务

使用微服务架构的很大一部分原因是因为它的可扩展性，随着微服务架构变 得越来越复杂，微服务的数量由几个变为成百上千之后，会面临很多问题。

拥抱故障

首先就是故障，硬盘可能会损坏，软件可能会崩溃，网络也是不可靠的，还有各种的偶然因素。尽管可以尽力去限制引起故障的因素，但是从统计学上看， 规模化后故障将成为必然事件。所以，既然买最好的工具，最昂贵的硬件也无法 避免它们会发生故障的事实，何不换一种思维去面对故障。为故障做好处理的准 备而不是过多的在意单节点的弹性。 构建一个弹性系统，尤其是当功能分散在多个不同的、有可能宕掉的微服务上时，重要的是能够安全地降级功能。举一个例子，在我自己开发的一个服务购 买商城的页面中，我需要几个微服务联合发挥作用。一个微服务显示出售服务的 详细信息，另一个显示用户信息，还有一个负责购物车的显示，还有一个是来给 用户推荐服务的推荐系统微服务。现在，如果这些微服务中的任何一个宕掉，都会导致整个Web页面不可用，那么该系统的弹性还不如只使用一个服务的系统。

因此需要做的是恰当地降级功能，功能降级的意思是在业务逻辑中包含了这个功 能不可用时应该如何做，特别是对于使用网络和操作数据库的行为来说，这是很有用的。比如购物车微服务不可用的话，可能会有很多事情无法完成，但是可以简单的将购物车部分的图标先替换为一个加载中的图标，然后将服务的列表的显示出来，这样的话对于用户来说，他不会得到一个无法使用的提示，还是可以进行其它的操作。对简单的单块应用程序来说，系统不是好的，就是坏的。但对于微服务架构，需要考虑更多微妙的情况，很多情况下，需要做的往往不是技术决策，而是一种业务上的考虑。当购物车微服务发生故障时，除非理解了业务的上下文，否则我们不知道该采取什么行动。比如，也许关闭整个网站，也许把用户界面上的购物车控件变成一个别的提示图片。所以对于产品经理来说，对于每一 个使用多个微服务的面向用户的界面来说，都要考虑其中的每个微服务故障时应该做出如何的业务处理。

Ariel Tseitlin 介绍了 Netflix 是如何运作的。Netflix 完全是基于 AWS 的基础设施的，它的经营规模也是非常庞大的。这两个因素意味着，它必须很好地应对故障。实际上 Netflix 通过引发故障来确保其系统的容错性。一些公司喜欢组织游戏日，在那天系统会被关掉以模拟故障发生，然后不同团队演练如何应对这种情况。Netflix 每天都在生产环境中通过编写程序引发故障。这些项目中最著名的是混乱猴子（Chaos Monkey），在一天的特定时段随机停掉服务器或机器。 知道这可能会发生在生产环境，意味着开发人员构建系统时不得不为它做好准备。混乱猴子只是 Netflix 的故障机器人猴子军队（Simian Army）的一部分。混乱大猩猩（Chaos Gorilla）用于随机关闭整个可用区（AWS 中对数据中心的叫法）， 而延迟猴子（Latency Monkey）在系统之间注入网络延迟。Netflix 已使用开源代 码许可证开源了这些工具。对许多人来说，你的系统是否真的健壮的终极验证是， 在你的生产环境上释放自己的猴子军队。通过让软件拥抱和引发故障，并构建系统来应对，这只是 Netflix 做的一部分事情。它还知道当失败发生后从失败中学习的重要性，并在错误真正发生时采用不指责文化。作为这种学习和演化过程的一部分，开发人员被进一步授权，他们每个人都需要负责管理他的生产服务。通过引发故障，并为其构建系统，Netflix 已经确保它的系统能够更好地规模化以及支持其客户的需求。

服务发现

拥有很多的微服务之后，就要考虑如何将这些微服务充分利用起来，想利用 这些微服务首先要知道它们在哪，这就是服务发现。服务发现有很多的用途，也许是想知道在特定环境下有哪些微服务在运行，据此才能知道哪些应该被监测；也许是以便其消费者知道在哪里能找到它；或许是想方便组织里的开发人员了解哪些 API 可用，以避免他们重新发明轮子。从广义上来说，上述所有用例都属于服务发现。与微服务世界中的其他问题类似，我们有很多不同的选项来处理它。目前有很多组织提供了解决方案，比如 Zookeeper、Consul、Eureka、Ectd 等等。它们都会把事情分成两部分进行处理。首先，它们提供了一些机制，让一个实例 注册并告诉所有人：“我在这里！”其次，它们提供了一种方法，一旦服务被注册就可以找到它。然后，当考虑在一个不断销毁和部署新实例的环境中，服务发现会变得更复杂。

自动伸缩

利用某些管理工具（比如 Kubernetes）完全自动化的创建虚拟主机和部署微服务是对微服务进行自动伸缩的基本条件。可能系统的负载高峰是从上午 9 点到下午 5 点，因此可以在早上 8：45 启动额外的实例，然后在下午 5：15 关掉这些不再需要规模化微服务的实例以节省开支。另一方面，可以响应式地进行负载 调整，比如在负载增加或某个实例发生故障时，来增加额外的实例，或在不需要时移除它们。关键是要知道一旦发现有上升的趋势，能够多快完成扩展。如果只能在负载增加的前几分钟得到消息，但是扩展至少需要10分钟，那么需要保持额外的容量来弥合这个差距。良好的负载测试套件在这里是必不可少的。可以使用它们来测试自动伸缩规则。如果没有测试能够重现触发伸缩的不同负载，那么只能在生产环境上发现规则的错误，但这时的后果不堪设想。新闻网站是一个很好的混合使用预测型伸缩和响应型伸缩的例子。能清楚地看到其日常趋势，从早晨一直到午餐时间负载上升，随后开始下降。这种模式每天都在重复，但在周末流量波动则不太明显。这呈现了相当明显的趋势，可以据此对资源进行主动扩容（缩容）。另一方面，一个大新闻可能会导致意外的高峰，在短时间内需要更多的容量。事实上相比响应负载，自动伸缩被更多应用于响应故障。AWS 允许你指定这样的规则：“这个组里至少应该有 5 个实例”，所以如果一个实例宕掉后，一个新的实例会自动启动。当有人忘记关掉这个规则时，就会导致一个有趣的打鼹鼠游戏（whack-a-mole），即当试图停掉一个实例进行维护时，它却自动启动起来了。响应型伸缩和预测型伸缩都非常有用，如果使用的平台允许按需支付所使用的计算资源，它们可以节省更多的成本。在大多数情况下，手头有多余的计算能力，比没有足够的计算能力要好得多。

监控

将系统拆分成更小的、细粒度的微服务会带来很多好处。然而，它也增加了生产系统的监控复杂性。

我们现在有多个服务需要监控，有多个日志需要筛选，多个地方有可能因为网络延迟而出现问题。该如何应对呢?

- 更多的日志

  使用专门的日志系统来专门获取日志而使日志能够集中在一起方便使用。

- 服务指标的选择

  我们用于无法得知什么数据是有用的，所以一个很好的做法是暴露一切数据然后依赖指标系统对它们进行处理。

- 级联故障的处理

  级联故障特别危险。想象这样一个情况，服务调用之间网络连接瘫痪了，服务本身是健康的，但它们之间无法交互。 如果只查看某个服务的健康状态，我们不会知道已经出问题了。使用合成监控会将问题暴露出来。但为了确定问题的原因，我们需要报告这一事实是一个服务无法访问另一个服务。
  因此，监控系统之间的集成点非常关键。每个服务的实例都应该追踪和显示其下游服务的健康状态，从数据库到其他合作服务。你也应该将这些信息汇总，以得到一个整合的画面。你会想了解下游服务调用的响应时间，并检测是否有错误。

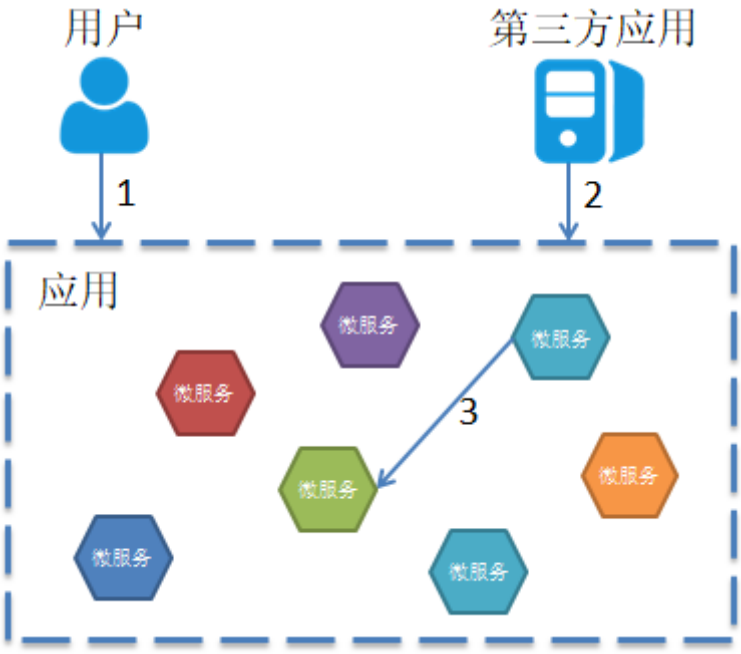
  你可以使用库实现一个断路器网络调用，以帮助你更加优雅地处理级联故障和功能降级，一些库，例如 JVM 上的 Hystrix，便很好地提供了这些监控功能。

- 标准化

  服务之间使用多个接口，以很多不同的方式合作为用户提供功能，你需要以整体的视角查看系统。所以应该以标准格式的方式记录日志。

安全

相对于传统单体应用，微服务架构下的认证和鉴权涉及到场景更为复杂，涉及到用户访问微服务应用，第三方应用访问微服务应用，应用内多个微服务之间相互访问等多种场景，每种场景下的认证和鉴权方案都需要考虑到，以保证应用程序的安全性。



一个完整的微服务应用是由多个相互独立的微服务进程组成的，对每个微服务的访问都需要进行用户认证。如果将用户认证的工作放到每个微服务中，存在下面一些问题：

- 需要在各个微服务中重复实现这部分公共逻辑。虽然我们可以使用代码库复用部分代码，但这又会导致所有微服务对特定代码库及其版本存在依赖，影响微服务语言/框架选择的灵活性。
- 将认证和鉴权的公共逻辑放到微服务实现中违背了单一职责原理，开发人员应重点关注微服务自身的业务逻辑。
- 用户需要分别登录以访问系统中不同的服务。

由于在微服务架构中以API Gateway作为对外提供服务的入口，因此可以在API Gateway处提供统一的用户认证，用户只需要登录一次，就可以访问系统中所有微服务提供的服务。

总结

总的来说，微服务的设计原则有如下几点：

- 围绕业务概念建模

  经验表明，围绕业务的限界上下文定义的接口，比围绕技术概念定义的接口更加稳定。针对系统如何工作这个领域进行建模，不仅可以帮助我们形成更稳定的接口，也能确保我们能够更好地反映业务流程的变化。使用限界上下文来定义可能的领域边界。

- 接收自动化文化

  微服务引入了很多复杂性，其中的关键部分是，我们不得不管理大量的服务。解决这个问题的一个关键方法是，拥抱自动化文化。前期花费一定的成本，构建支持微服务的工具是很有意义的。自动化测试必不可少，因为相比单块系统，确保我们大量的服务能正常工作是一个更复杂的过程。调用一个统一的命令行，以相同的方式把系统部署到各个环境是一个很有用的实践，这也是采用持续交付对每次提交后的产品质量进行快速反馈的一个关键部分。

  请考虑使用环境定义来帮助你明确不同环境间的差异，但同时保持使用统一的方式进行部署的能力。考虑创建自定义镜像来加快部署，并且创建全自动化不可变服务器，这会更容易定位系统本身的问题。

- 隐藏内部实现细节

  为了使一个服务独立于其他服务，最大化独自演化的能力，隐藏实现细节至关重要。限界上下文建模在这方面可以提供帮助，因为它可以帮助我们关注哪些模型应该共享，哪些应该隐藏。服务还应该隐藏它们的数据库，以避免陷入数据库耦合，这在传统的面向服务的架构中也是最常见的一种耦合类型。使用数据泵（data pump）或事件数据泵（event data pump），将跨多个服务的数据整合到一起，以实现报表的功能。

  请考虑使用 REST，它将内部和外部的实现细节分离方式规范化，即使是使用 RPC，你仍然可以采用这些想法。

- 让一切都去中心化

  为了最大化微服务能带来的自治性，我们需要持续寻找机会，给拥有服务的团队委派决策 和控制权。在这个过程初期，只要有可能，就尝试使用资源自助服务，允许人们按需部署 软件，使开发和测试尽可能简单，并且避免让独立的团队来做这些事。 在这个旅程中，确保团队保持对服务的所有权是重要的一步，理想情况下，甚至可以让团 队自己决定什么时候让那些更改上线。使用内部开源模式，确保人们可以更改其他团队拥 有的服务，不过请注意，实现这种模式需要很多的工作量。让团队与组织保持一致，从而 使康威定律起作用，并帮助正在构建面向业务服务的团队，让他们成为其构建的业务领域 的专家。一些全局的引导是必要的，尝试使用共同治理模型，使团队的每个成员共同对系 统技术愿景的演化负责。像企业服务总线或服务编配系统这样的方案，会导致业务逻辑的中心化和哑服务，应该避 免使用它们。使用协同来代替编排或哑中间件，使用智能端点（smart endpoint）确保相关的逻辑和数据，在服务限界内能保持服务的内聚性。

- 可独立部署

应当始终努力确保微服务可以独立部署。甚至当需要做不兼容更改时，我们也应该同时提供新旧两个版本，允许消费者慢慢迁移到新版本。这能够帮助我们加快新功能的发布速度。拥有这些微服务的团队，也能够越来越具有自治性，因为他们不需要在部署过程中不断地做编配。当使用基于 RPC 的集成时，避免使用像 Java RMI 提供的那种使用生成的桩代码，紧密绑定客户端 / 服务器的技术。 通过采用单服务单主机模式，可以减少部署一个服务引发的副作用，比如影响另一个完全不相干的服务。请考虑使用蓝 / 绿部署或金丝雀部署技术，区分部署和发布，降低发布出错的风险。使用消费者驱动的契约测试，在破坏性的更改发生前捕获它们。 请记住，你可以更改单个服务，然后把它部署到生产环境，无需联动地部署其他任何服务，这应该是常态，而不是例外。你的消费者应该自己决定何时更新，你需要适应他们。

- 隔离失败

  微服务架构能比单块架构更具弹性，前提是我们了解系统的故障模式，并做出相应的计划。如果不考虑调用下游可能会失败的事实，系统会遭受灾难性的级联故障，系统也会比以前更加脆弱。当使用网络调用时，不要像使用本地调用那样处理远程调用，因为这样会隐藏不同的故障模式。所以确保使用的客户端库，没有对远程调用进行过度的抽象。

  心中要持有反脆弱的信条，预期在任何地方都会发生故障，这说明正走在正确的路上。请确保正确设置超时，了解何时及如何使用舱壁和断路器，来限制故障组件的连带影响。如果系统只有一部分行为不正常，要了解其对用户的影响。知道网络分区可能意味着什么，以及在特定情况下牺牲可用性或一致性是否是正确的决定。

- 高度可观察

  不能依靠观察单一服务实例或一台服务器的行为来判断系统是否运行正常。相反，需要从整体上看待正在发生的事情。通过注入合成事务到你的系统，模拟真实用户的行为，从而使用语义监控来查看系统是否运行正常。聚合你的日志和数据，这样当你遇到问题时，就可以深入分析原因。而当需要重现令人讨厌的问题，或仅仅查看你的系统在生产环境是如何交互时，关联标识可以帮助你跟踪系统间的调用。

使用Spring Boot 开发一个微服务

SpingBoot简介

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者 。Spring Boot设计的重要一点就是想简化单个微服务的开发，它的精要部分有如下几点:

- 自动配置
- 起步依赖

  其实就是特殊的Maven依赖和Gradle依赖，利用了传递依赖解析，把常用库聚合在一起，组成了几个为特殊功能而定制的依赖。

  例如 spring-boot-starter-web
- 命令行界面（可选）
- Actuator

  提供运行时检视应用程序内部的能力，包含如下细节

  - Spring应用程序上下文里配置的Bean
  - Spring Boot的自动配置做的决策
  - app收到的环境变量、系统属性、配置属性和命令行参数
  - app里线程当前的状态

- 最近处理过的HTTP请求的追踪情况
- 各种和内存用量、垃圾回收、Web请求以及数据源用量相关的指标

如何开发一个微服务

1.在IDEA中使用Spring Initializr可以得到建立好的Maven/Gradle项目结构

2.XXXApplication是应用程序的启动引导类，也是主要的Spring配置类，初始只有一行配置代码@SpringBootApplication实际上是三个注解的组合

- @Configuration:表明该类使用Spring基于Java的配置
- @ComponentScan:启动组建扫描(所以使用@Component(或@Service,@Controller,@Repository)注解的组件才能被自动发现并且注册为Spring应用程序上下文里的Bean)
- Spring Boot的@EnableAutoConfiguration：开启Spring Boot自动配置

3.使用application.properties可以帮助细粒度的调整自动配置，例如server.port = xxxx改变默认的8080

起步依赖

- 起步依赖有很多，并不需要制定版本号,版本由使用的Spring Boot版本确定，起步依赖决定它们引入的传递依赖的版本。

  当然这些版本也可以由自己控制，有两种场景

  - 不需要这个构件，可以在maven中用来排除依赖
  - 需要使用另外一个版本，直接在pom.xml中表达诉求

自动配置

是一个程序启动时的过程，考虑众多因素才觉得配置应该用哪个，不该用哪个，例如会考虑Spring Security是不是在Classpath里，是的话就进行一个非常基本的Web安全配置。

会做接近200个这样的决定，涵盖安全、集成、持久化、Web开发等诸多方面。

如何实现

在向应用程序加入Spring Boot时，有个名为spring-boot-autoconfigure的JAR文件，其中包含了很多配置类，每个配置类都在应用程序的Classpath里，都有机会为应用程序的配置添砖加瓦。 *利用了Spring4.0的条件化配置，条件化配置允许配置存在于应用程序中，但是在满足某些特定条件之前都忽略这些配置。*

自定义配置

- 覆盖自动配置

  覆盖自动配置很简单，就当自动配置不存在，直接显式地写一段配置。

  实现

  扩展配置类并且覆盖里边的方法，自动配置类上的条件化配置会检测到已经有了配置的Bean,就会直接跳过

- 通过属性文件外置配置

  SB自动配置的Bean提供了300多个用于微调的属性，只需要调整设置，获取配置的优先级从高到低为

  - 命令行参数
  - java:comp/env里的JNDI属性

    - JVM系统属性
    - 环境变量
    - 随机生成的带random.*前缀的属性
    - 应用程序以外的application.properties
    - 打包在程序内的appliction.properties
    - 通过@PropertySource标注的属性源
    - 默认属性

  允许使用Profile配置不同的运行环境

- 可以定制应用程序错误界面让它更加有艺术性

测试

编写单元测试的时候Spring不需要介入，Spring鼓励松耦合、接口驱动的设计，这些都会帮助编写单元测试，但是在写单元测试时不需要用到Spring。

但是集成测试要用到Spring,如何生产程序使用Spring来配置并且组装组件，那么测试就需要用它来组装那些组件

注解

- `@SpringRunner` 可以在基于JUnit的应用程序测试里加载Spring应用程序上下文
- `@Runwith(SpringRunner.class)` 代表让测试运行于Spring测试环境
- `@SpringBootTest` 加在测试类上， `@Test` 加在方法上，不需要mian和启动就可以测试，`@Before` 代表在测试之前做的事情

测试Web应用程序

要恰当地测试一个Web应用程序，你需要投入一些实际地HTTP请求，确认它能正确地处理那些请求，SB有两个可选的方案

- Spring Mock MVC:在近似真实的模拟servlet容器里测试控制器，而不用实际启动应用服务器
- Web集成测试，在嵌入式Servlet容器（比如Tomcat或Jetty）里启动应用程序，在真正的应用服务器里执行测试

模拟**Spring** MVC

可以使用 `MockMvcBuilders` ，该类提供了两个静态方法

- `standaloneSetup` ： 构建一个Mock MVC，提供一个或多个收购创建并配置的控制器
- `webAppContextSetup` :使用Spring应用程序上下文来创建

示例

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class SuperadminApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void getTotal() throws Exception{
        mockMvc.perform(MockMvcRequestBuilders.get("/admin/getTotal"))
```

```
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("status").value("20000"))
                .andDo(MockMvcResultHandlers.print())
                .andReturn();
    }
}
```

`.perform()` 执行一个MockMvcRequestBuilders请求。其中 `.get()` 表示发送get请求（可以使用get、post、put、delete等）； `.contentType()` 设置编码格式； `.param()` 请求参数,可以带多个。 `andExpect()` 添加MockMvcResultMatchers验证规则，验证执行结果是否正确。 `status().isOk()` 校验返回200， `jsonPath` 可以对返回的JSON字段进行校验。 `.andDo()` 添加 MockMvcResultHandlers结果处理器,这是可以用于打印结果输出。 `.andReturn()` 结果还回，然后可以进行下一步的处理。

测试Web安全

使用Spring Security后要进行测试，要添加测试模块。

经过身份验证的请求的发起

- `@WithMockUser` 加载安全上下文，其中包含一个UserDetails对象，使用了给定的用户名、密码和授权
- `@WithUserDetails` 根据给定的用户名查找UserDetails对象，加载安全上下文

测试实际运行的应用程序

`@WebIntegerationTest` 声明启动嵌入式的Servlet容器，然后使用 `RestTemplate` 发起请求

Actuator

关键特性是在应用程序里提供众多的Web端点，通过他们能了解应用程序运行时的内部状况，要使用引入起步依赖spring-boot-starter-actuator

ID	描述	默认启用	默认公开
auditevents	公开当前应用程序的审计事件信息	Yes	No
beans	显示应用程序中所有Spring bean的完整列表	Yes	No
conditions	显示在配置和自动配置类上评估的条件以及它们是否匹配的原因	Yes	No
configprops	显示所有 @ConfigurationProperties 对照的列表	Yes	No
env	从Spring的 ConfigurableEnvironment 中公开属性	Yes	No
flyway	显示已应用的任何Flyway数据库迁移	Yes	No
health	显示应用程序健康信息	Yes	Yes
httptrace	显示HTTP跟踪信息（默认情况下，最后100个HTTP请求-响应交互）	Yes	No
info	显示任意应用程序信息	Yes	Yes
loggers	显示和修改应用程序中记录器的配置	Yes	No
liquibase	显示已应用的任何Liquibase数据库迁移	Yes	No
metrics	显示当前应用程序的“指标”信息	Yes	No
mappings	显示所有 @RequestMapping 路径对照的列表	Yes	No
scheduledtasks	显示应用程序中调度的任务	Yes	No
sessions	允许从Spring Session支持的会话存储中检索和删除用户会话	Yes	No
shutdown	让应用程序优雅地关闭	No	No
threaddump	执行线程转储	Yes	No

这些端点分为三大类，**配置端点**，**度量端点**，**其它端点**

配置

- /bean装配报告

```
{
  "bean": "readingListController",
  "dependencies": [                <------ Bean作用域
    "readingListRepository",
    "amazonProperties"
  ],
  "resource": "URL [jar:file:/../readinglist-0.0.1-SNAPSHOT.jar!
                          /readinglist/ReadingListController.class]",
  "scope": "singleton",
  "type": "readinglist.ReadingListController"
},
```

  Bean条目有五类信息

  - bean:Spring应用程序上下文中的名称或ID
  - resource:.class文件的物理位置，通常是一个URL，指向构建出的JAR。这会随着应用程序的构建和运行方式发生变化。
  - denpendencies:当前Bean注入的Bean ID列表
  - scope:Bean的作用域（通常是单例，也是默认作用域）
  - type:Bean的Java类型
- /autoconfig端点告诉你为什么会有这个Bean,或者为什么没有这个Bean
- /env会生成应用程序可用的**所有**环境属性的列表，无论这些属性是否用得到。
- /mappings端点提供了控制器的映射

度量

- 查看度量值 /metrics 是运行时度量状况的一个快照，对于评估应用程序的健康状况很有帮助。

分　类	前　缀	报告内容
垃圾收集器	gc.*	已经发生过的垃圾收集次数，以及垃圾收集所耗费的时间，适用于标记-清理垃圾收集器和并行垃圾收集器（数据源自java.lang.management.GarbageCollectorMXBean）
内存	mem.*	分配给应用程序的内存数量和空闲的内存数量（数据源自java.lang.Runtime）
堆	heap.*	当前内存用量（数据源自java.lang.management.MemoryUsage）
类加载器	classes.*	JVM类加载器加载与卸载的类的数量（数据源自java.lang. management.ClassLoadingMXBean）
系统	processors、uptime instance.uptime、systemload.average	系统信息，例如处理器数量（数据源自java.lang.Runtime）、运行时间（数据源自java.lang.management.RuntimeMXBean）、平均负载（数据源自java.lang.management.OperatingSystemMXBean）
线程池	threads.*	线程、守护线程的数量，以及JVM启动后的线程数量峰值（数据源自java.lang .management.ThreadMXBean）
数据源	datasource.*	数据源连接的数量（源自数据源的元数据，仅当Spring应用程序上下文里存在DataSource Bean的时候才会有这个信息）
Tomcat会话	httpsessions.*	Tomcat的活跃会话数和最大会话数（数据源自嵌入式Tomcat的Bean，仅在使用嵌入式Tomcat服务器运行应用程序时才有这个信息）
HTTP	counter.status.*、gauge.response.*	多种应用程序服务HTTP请求的度量值与计数器

- 追踪Web请求 /trace端点能够报告所有Web请求的详细信息，包括请求方法、路径、时间戳以及请求和响应的头信息

```
[
  ...
  {
    "timestamp": 1426378239775,
    "info": {
      "method": "GET",
      "path": "/metrics",
      "headers": {
        "request": {
          "accept": "*/*",
          "host": "localhost:8080",
          "user-agent": "curl/7.37.1"
        },
        "response": {
          "X-Content-Type-Options": "nosniff",
          "X-XSS-Protection": "1; mode=block",
          "Cache-Control":
                    "no-cache, no-store, max-age=0, must-revalidate",
          "Pragma": "no-cache",
          "Expires": "0",
          "X-Frame-Options": "DENY",
          "X-Application-Context": "application",
          "Content-Type": "application/json;charset=UTF-8",
          "Transfer-Encoding": "chunked",
          "Date": "Sun, 15 Mar 2015 00:10:39 GMT",
          "status": "200"
        }
      }
    }
  }
]
```

- 导出线程活动 /dump

```
[
  {
    "threadName": "container-0",
    "threadId": 19,
    "blockedTime": -1,
    "blockedCount": 0,
    "waitedTime": -1,
    "waitedCount": 64,
    "lockName": null,
    "lockOwnerId": -1,
    "lockOwnerName": null,
    "inNative": false,
    "suspended": false,
    "threadState": "TIMED_WAITING",
    "stackTrace": [
      {
        "classLoader": "java.lang.Thread",
        "fileName": "Thread.java",
        "lineNumber": -2,
        "methodName": "sleep",
        "nativeMethod": true
      },
      {
        "classLoader": "org.springframework.boot.context.embedded.
                        tomcat.TomcatEmbeddedServletContainer$1",
        "fileName": "TomcatEmbeddedServletContainer.java",
        "lineNumber": 139,
        "methodName": "run",
        "nativeMethod": false
      }
    ],
    "lockedMonitors": [],
    "lockedSynchronizers": [],
    "lockInfo": null
  },
  ...
]
```


- 健康情况（是否在运行） /health

其它

- /shutdown,关闭程序，默认关闭
- /info，展示希望发布的应用信息

基于Spring Cloud搭建的微服务框架

前言

Spring Boot和Spring Cloud为java开发者提供了一个将传统的、庞大的Spring应用迁移到可以部署到云的微服务应用的简单迁移路径。

Spring和微服务有什么关联

Spring 已经成为构建基于java应用程序事实上的开发框架。Spring 的核心是基于依赖注入的概念。在一个正常的 Java 应用里，应用被分解成各种各样的类。每个类经常被应用里其他的类明确的链接。链接就是一个类的构造函数中的代码直接调用。一旦代码被编译， 这些链接点是不能改发的。

这在一个大型项目中是有问题的，因为这些外部链接是脆弱的，迪行更改会导致其他代码的多个下游影响。依赖注入框架，如 Spring，让你通过在应用程序里使用约定（注释） 定义外部对象之间的关系，更方便的管理大型的 java项目，而不是在对象里硬编码对象之间的引用。在应用程序里不同的java 类之间，Spring 作为中介管理它们的依赖。

Spring 框架具有使人快速上手的优势，它迅速成为企业级应用 java开发人员寻找替代使用 J2EE 技术栈构建应用的一种更轻量级的方式。Spring 框架令人惊奇的是，它的社区生态能够持续发展并坚持自身。Spring 开发团现许多开发团队正在摒弃单体应用程序，这些应用程序的展示、业务和数据访问逻辑被打包在一起，作为单个工件部署。相反，他们正在转移到高度分布式的模型中，这些服务能够构建为可以轻松部署到云上的小型分布式服务，作为回应，Spring开发团队推出了两个项目：Spring Boot和Spring Cloud。

Spring Boot 是重新构想的 Spring 框架。然而它包含了 Spring 的核心特征，Spring Boot 剥离了 Spring 里面许多的“企业级”特性，提供了一个基于 java，面向REST微服务的框架。使用简单的注解，一个 java开发者就可以快速创建一个 REST 微服务，它可以被打包和部署而不需要外部的应用程序容器。

因为微服务已绊成为构建基于云的应用程序的更常见的架构模式之一，Spring 社区为我们提供了 Spring Cloud。Spring Cloud框架使得将微服务部署到私有云或者公有云变得更简单。Spring Cloud 把几种流行的云管理微服务框架整合到共同的框架下，在代码里使用注解使得这些技术的使用和部署变得更容易。

为什么要使用云

基于微服务架极的核心概念之一是每个服务都作为独立的组件被打包和部署。服务实例应该能够快速启动。服务的每个实例之间互不干扰，都应该与其他实例区分开来。

作为一位微服务的开发者，你将不得不决定是否将你的服务部署到如下的环境中：

- 物理服务器：虽然你可以构建和部署你的微服务到物理机，但很少有机构这样做，是因为物理服务器的有限制，你不能快速提升一台物理服务器的处理能力。在多台物理服务器上水平伸缩你的微服务是非常昂贵的。
- 虚拟机镜像：微服务一个关键的好处是它们具有快速启动和关闭服务实例以响应应用伸缩和服务故障事件处理的能力。虚拟机是主要的云计算提供商提供的核心服务。一个微服务可以被打包在一个虚拟机镜像。在任何一个 IaaS 私有云或公共云，服务的多个实例可以快速部署和启动。
- 虚拟容器：虚拟容器是在虚拟机镜像里部署微服务的自然延伸。与其将一个服务部署到一台完整的虚拟机，许多开发者将他们的服务部署到于上的Docker容器（或者等效的容器技术）。虚拟容器运行在一台虚拟机内部；使用虚拟容器，你可以将单个虚拟机分隔成一系列独立的、共享相同的虚拟机镜像的进程。

基于云的微服务的优势是围绍弹性的概念。云服务提供商能够为你提供在几分钟内快速地生成新的虚拟机和容器的能力。如果你的服务需求下降，你可以在不产生任何额外成本的情况下向下收缩虚拟服务器的数量。使用云提供商的设施部署你的微服务能够提高你应用横向扩展的能力（增加更多的服务器和服务实例）。服务器的伸缩性也意味着你的应用程序可以更具弹性。如果你的一个微服务有问题甚至宕机，生成新的服务实例，可以让你的应用程序处于更长的活动状态，使你的开发团队能够更好的解决问题。

所以Spring Cloud通常是与Docker容器结合使用，它的部署哲学如下：

- 简化基础设施管理

  IaaS云提供商给你提供对服务的最大的控制能力，使用简单的API调用，新的服务可以启动和停止。一个IaaS云解决方案，你只需为你使用的基础设施付费。

- 大规模的横向扩展能力

  IaaS云提供商能够让你快速地启动一个服务地一个或多个实例，这种能力意味着你可以快速伸缩服务和绕过有问题或已冗机的服务器。

- 通过地理分布实现高冗余

IaaS供应商有多个数据中心，通过IaaS云提供商部署你的微服务，你能获得 一个比在数据中心使用集群更高级别的冗余。

什么是Spring Cloud

从零开始微服务设计中所要关心的各种要素是一个惊人的工作量，幸运的是，Spring团队集成了大量进行充分测试的开源项目为Spring子项目，统称为Spring Cloud （**https://spring.io/projects/spring-cloud**） 。

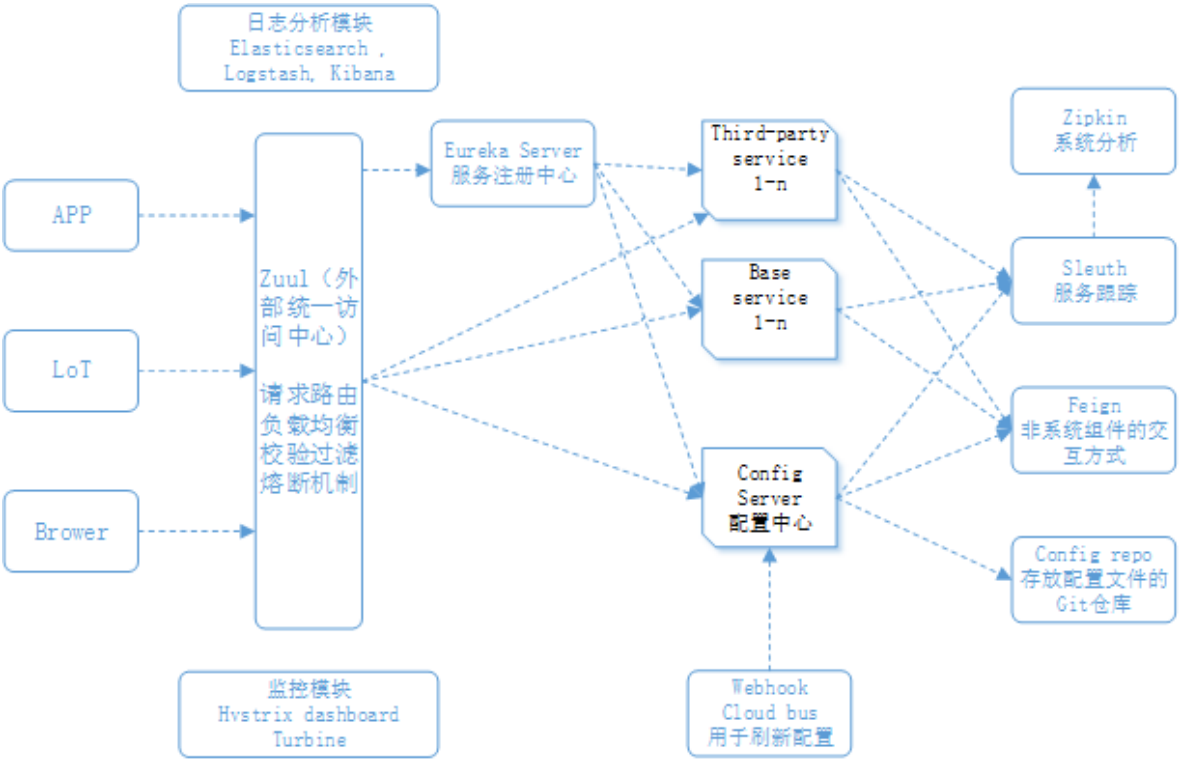
Spring Cloud把开源公司，如Pivotal，HashiCorp和Netflix交付的产品整合在一起。Spring Cloud 简化了设置和配置这些项目到你的 Spring 应用程序，使你专注于写代码，没有被掩埋在可以构建和部署一个微服务应用的所有基础设施的配置细节里。

Spring Cloud 部分项目总览如下：

项目名称	项目职能
Spring Cloud Config	Spring Cloud 提供的分布式配置中心，为外部配置提供了客户端和服务端的支持。
Spring Cloud Netflix	与各种Netflix OSS组件集成（Eureka，Hystrix，Zuul，Archaius等）。
Spring Cloud Bus	用于将服务和服务实例与分布式消息传递连接在一起的事件总线。用于跨群集传播状态更改（例如，配置更改事件）。
Spring Cloud Cloudfoundry	提供应用程序与 Pivotal Cloud Foundry 集成。提供服务发现实现，还可以轻松实现受SSO和OAuth2保护的资源。
Spring Cloud Open Service Broker	为构建实现 Open service broker API 的服务代理提供了一个起点。
Spring Cloud Cluster	提供Leadership选举，如：Zookeeper, Redis, Hazelcast, Consul等常见状态模式的抽象和实现。
Spring Cloud Consul	封装了Consul操作，consul 是一个服务发现与配置工具，与Docker容器可以无缝集成。
Spring Cloud Security	基于spring security的安全工具包，为你的应用程序添加安全控制。在Zuul代理中为负载平衡的OAuth2 rest客户端和身份验证头中继提供支持。
Spring Cloud Sleuth	Spring Cloud 提供的分布式链路跟踪组件，兼容zipkin、HTracer和基于日志的跟踪（ELK）
Spring Cloud Data Flow	大数据操作工具，作为Spring XD的替代产品，它是一个混合计算模型，结合了流数据与批量数据的处理方式。
Spring Cloud Stream	数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。
Spring Cloud CLI	基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件。
Spring Cloud OpenFeign	一个http client客户端，致力于减少http client客户端构建的复杂性。
Spring Cloud Gateway	Spring Cloud 提供的网关服务组件
Spring Cloud Stream App Starters	Spring Cloud Stream App Starters是基于Spring Boot的Spring 集成应用程序，可提供与外部系统的集成。
Spring Cloud Task	提供云端计划任务管理、任务调度。
Spring Cloud Task App Starters	Spring Cloud任务应用程序启动器是SpringBoot应用程序，它可以是任何进程，包括不会永远运行的Spring批处理作业，并且在有限的数据处理周期后结束/停止。

项目名称	项目职能
Spring Cloud Zookeeper	操作Zookeeper的工具包，用于使用zookeeper方式的服务发现和配置管理。
Spring Cloud AWS	提供与托管的AWS集成
Spring Cloud Connectors	便于云端应用程序在各种PaaS平台连接到后端，如：数据库和消息代理服务。
Spring Cloud Starters	Spring Boot式的启动项目，为Spring Cloud提供开箱即用的依赖管理。
Spring Cloud Contract	Spring Cloud Contract是一个总体项目，其中包含帮助用户成功实施消费者驱动合同方法的解决方案。
Spring Cloud Pipelines	Spring Cloud Pipelines提供了一个固定意见的部署管道，其中包含确保您的应用程序可以零停机方式部署并轻松回滚出错的步骤。
Spring Cloud Function	Spring Cloud Function通过函数促进业务逻辑的实现。 它支持Serverless 提供商之间的统一编程模型，以及独立运行（本地或PaaS）的能力。

一个基本的Spring Cloud项目的架构图如下所示：



Spring Cloud提供的能力有如下部分。

配置中心

Spring Cloud Config处理应用程序的配置数据的管理，通过一个集中的服务来使你的应用程序配置数据（尤其使你的环境的具体配置数据）从你的部署微服务分离干净，这确保了你无论扩展多少微服务实例，它们总会有相同的配置。

服务发现

Spring Cloud的服务发现能够将消费服务为的客户端从服务器部署的物理位置抽离出来，服务消费者通过逻辑名称而不是物理位置调用服务器的业务逻辑。Spring Cloud服务发现还负责在忘启动和停止的时候注册和注销服务实例。Spring Cloud的服务发现可以采用多个开源项目作为服务发现引擎，其中最常用的是Consul和Eureka。

容错

Spring Cloud在很大程度商集成了Netflix OSS套件来实现服务的弹性模式。Spring Cloud将Netflix的Hystrix和Ribbon项目集成在一起，在你的微服务中实施它们。

使用Hystrix，可以快速实现服务客户端的弹性模式，如断路器和舱壁模式。

Ribbon提供了来自服务消费者的服务调用的客户端负载均衡，这使得即使服务发现代理暂时不可用，客户端也可以继续进行服务调用。

服务网关

服务网关在微服务应用之间提供路由能力，它代理服务请求并确保目标服务调用之前，所有对你的微服务的调用都通过一个单一的”前门“。有了这种集中的服务调用，你可以执行标准的服务策略，如安全授权验证、内容过滤和路由规则。Spring Cloud中可以选择Zuul或者Gateway这两个组件来实现网关功能。

异步消息代理

Spring Cloud Stream是一门有利的技术，可以很容易将轻量级消息处理器整合到你的微服务，使用它你可以构建智能的微服务，它可以使用在你的应用程序发生的异步事件，通过Spring Cloud Stream，你能够快速将你的微服务与消息中间件集成，如RabbitMQ、RocketMQ、Kafka等。

链路追踪

Spring Cloud Sleuth允许你将唯一的跟踪标识符集成到HTTP调用和在你的应用程序使用的消息通道（RabbitMQ，Appache Kafka）。这些追踪号，有时被称为关联或者追踪ID，允许你跟踪一个交易，因为它在应用程序的不同服务之间传递，Sleuth会自动将追踪ID添加到在微服务中产生的日志。

另外，真正的优势是与日志聚合技术工具如Papertrail和跟踪工具如Zipkin集合。Papertail是一个日志平台，用于将不同微服务实时日志汇总到一个可查询的数据库。Zipkin使用Sleuth产生的数据，允许你可视化一个单一交易的有关联的服务调用流程

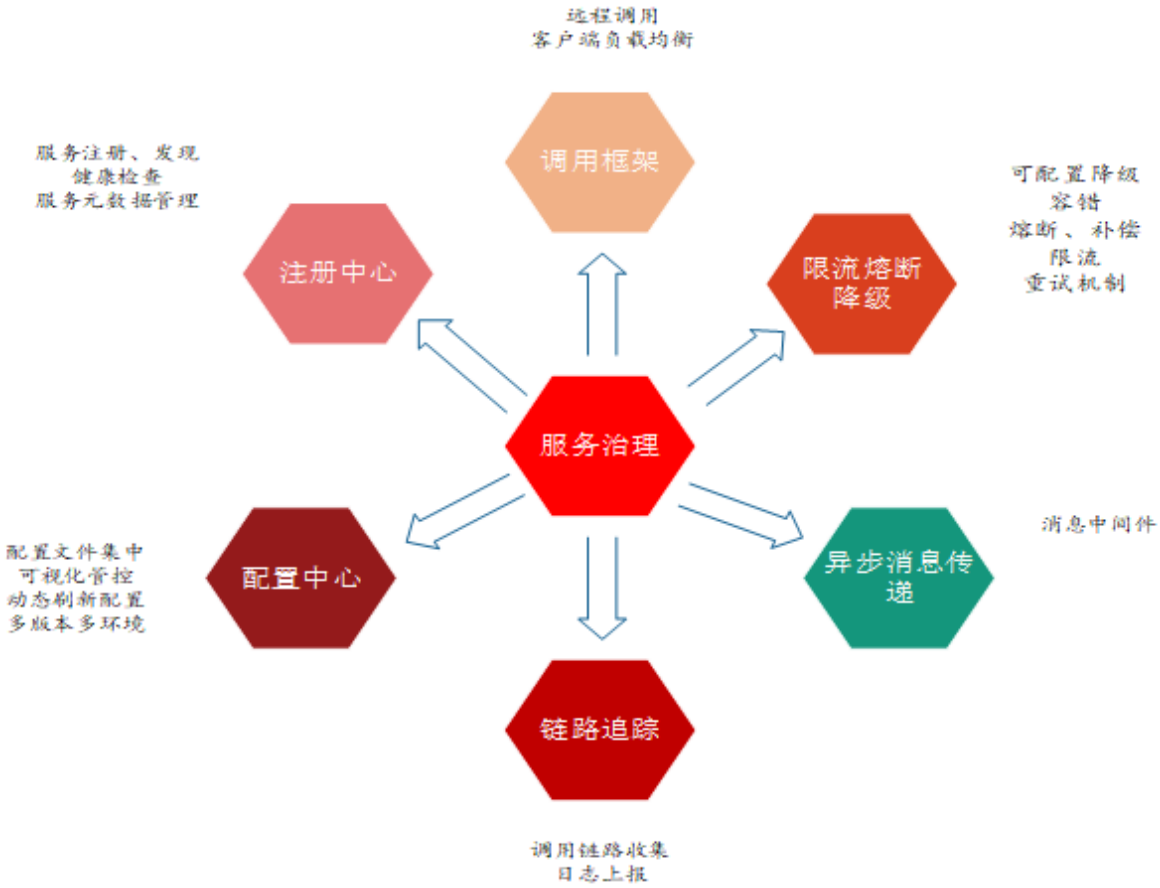
安全

Spring Cloud Security是一个身份验证和授权框架，它能控制谁可以访问你的服务和它们能为你的服务做些什么？Spring Cloud Security 是基于仓牌的，它允许服务通过一个由身份验证服务器发出的令牌互相通信。每个服务接收到一个调用，可以检查在 HTTP 调用中提供的仓牌，来验证用户的身份和它们访问服务的权限。另外，Spring Cloud Security 支持 JavaScript Web Token。

和其它微服务框架的对比

微服务框架是什么

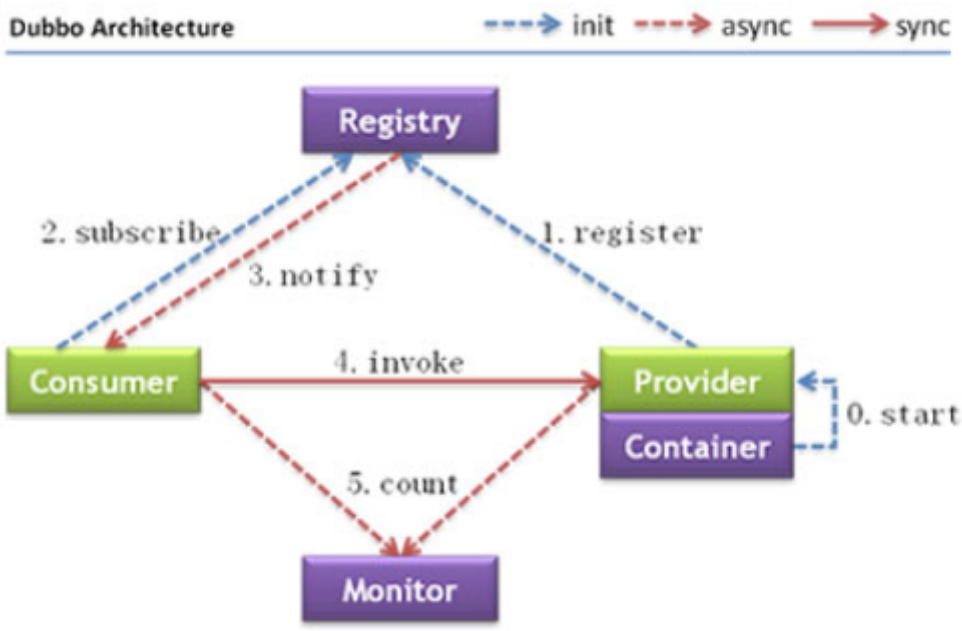
微服务架构开始变得火热之后，有了一系列的概念：服务注册与发现，请求链路追踪，服务熔断，服务限流，服务管控配置，服务预警，也有了一系列的开源工具：Eurake，Zuul，Ribbon，hystrix，zipkin，dubbo，Sleuth，Elastic Search，grafna，Promethues。而实际商这些工具都是为了解决一个问题：将分解而成的微服务治理好，而提到服务治理，有如下图中的几个概念。



而所谓的微服务框架就是集成了自己和其他公司的一些开源工具之后经过测试可以稳定运行然后发布出来供更多的开发人员使用的框架，可以看出Spring Cloud能够覆盖服务治理的六大基本模块。基本每个大型互联网公司都有自己的一套微服务框架，就开源项目而言，除了Spring Cloud目前比较活跃的还有Dubbo和Istio，接下来会将这两个项目和Spring Cloud进行对比。

Dubbo

Dubbo 是阿里开源的一个 SOA 服务治理解决方案，文档丰富，在国内的使用度非常高 。架构图和其中游戏额节点的信息说明如下。



节点角色说明

节点	角色说明
`Provider`	暴露服务的服务提供方
`Consumer`	调用远程服务的服务消费方
`Registry`	服务注册与发现的注册中心
`Monitor`	统计服务的调用次数和调用时间的监控中心
`Container`	服务运行容器

可以看出，Dubbo的功能特点如下

- 调用中间层变成了可选组件，消费方可以直接访问服务提供方；
- 服务信息被集中到 Registry 中，形成了服务治理的中心组件；
- 通过 Monitor 监控系统，可以直观地展示服务调用的统计信息；
- 服务消费者可以进行负载均衡、服务降级的选择。

它和Spring Cloud的功能对比如下：

功能名称	Dubbo	Spring Cloud
服务注册中心	ZooKeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

很明显Spring Cloud比Dubbo更加强大，涵盖面更广，而且作为Spring出品的框架，天生和Spirng的众多项目完美融合，使用Dubbo的话还需要去寻找能够弥补Dubbo在服务治理方面缺失功能的组件，这对于技术不是很成熟的公司是一个巨大的挑战。

实际上Dubbo是Spring Cloud功能的一个子集，Dubbo是RPC框架，Dubbo现在也支持集成在Spring Cloud中作为服务治理模块的一部分。

Istio

Istio 作为用于微服务服务聚合层管理的新锐项目，是 Google、IBM、Lyft 首个共同联合开源的项目，提供了统一的连接，安全，管理和监控微服务的方案。

目前首个测试版是针对 Kubernetes 环境的，社区宣称在未来几个月内会为虚拟机和 Cloud Foundry 等其他环境增加支持。 Istio 将流量管理添加到微服务中，并为增值功能（如安全性，监控，路由，连接管理和策略）创造了基础。

Istio的特点如下:

- Istio 适用于容器或虚拟机环境（特别是 k8s），兼容异构架构。
- Istio 使用 sidecard（边车模式）代理服务的网络，不需要对业务代码本身做任何的改动。
- HTTP、gRPC、WebSocket 和 TCP 流量的自动负载均衡。
- Istio 通过丰富的路由规则、重试、故障转移和故障注入，可以对流量行为进行细粒度控制；支持访问控制、速率限制和配额。
- Istio 对出入集群入口和出口中所有流量的自动度量指标、日志记录和跟踪。

从微服务的设计模式来说，Istio是最符合理想愿景的，不限制开发语言和框架，天生基于云平台开发，功能强大。但是它最大的缺点是不易于上手，对于开发人员来说，要学习很多运维方面的东西，而且Istio还没有能应用于生产环境的版本，对于需要实际应用的项目来说，不能保证稳定性是不可接受的事情。

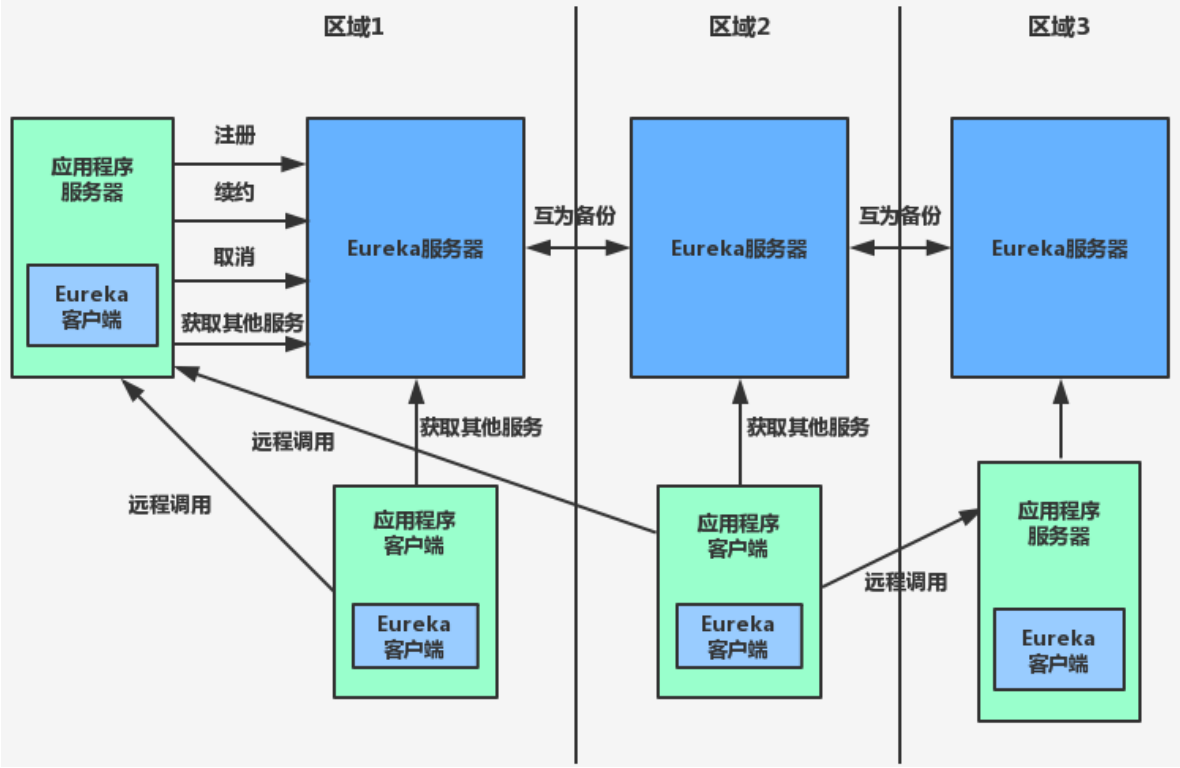
所以总结来说，Spring Cloud是Java领域最适合做微服务的框架，它对于微服务生态的支持力度最大，天然支持Spring Boot，便于业务落地。对于想用微服务架构而人手不足的中小公司更为友好。

框架中的组件介绍

我们在Spring Cloud的基础上选择了一套能满足微服务开发、管理且测试稳定的组件来作为自己的微服务框架，下面对其中的核心组件来进行介绍。

Eureka

Eureka作为我们微服务框架中的服务注册和发现模块，主要以云服务为支撑，提供服务发现并实现负载均衡和故障转移。Eureka提供了Java客户端组件，Eureka Client，方便与服务端的交互。客户端内置了基于round-robin实现的简单负载均衡。Netflix为Eureka提供更为复杂的负载均衡方案进行封装，以实现高可用，它包括基于流量、资源利用率以及请求返回状态的加权负载均衡。

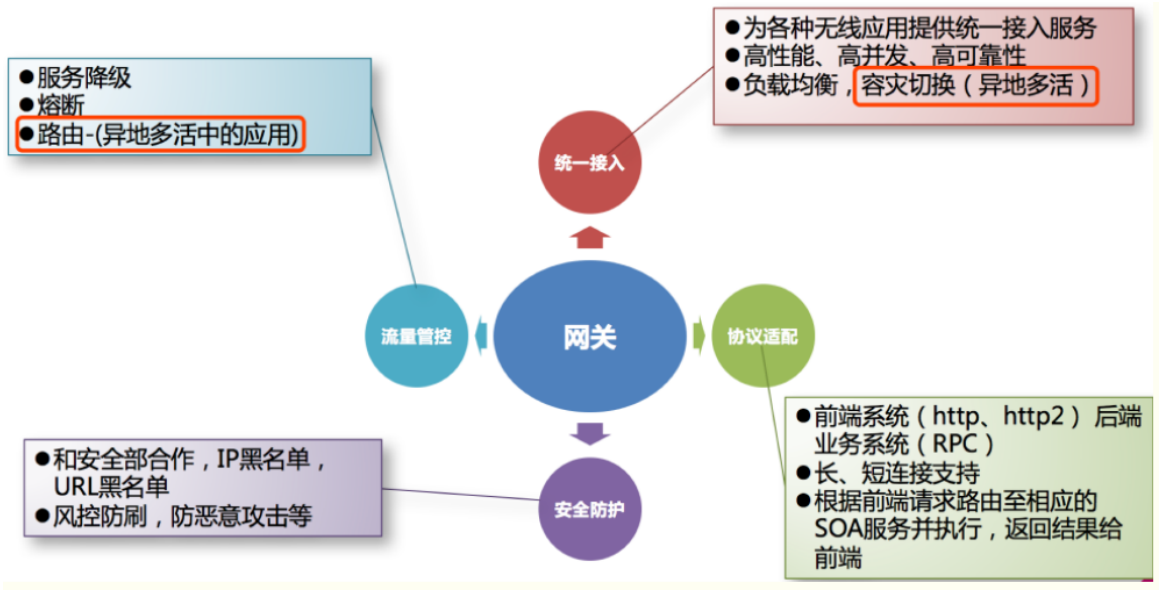


如架构图所示，在每个区域中都有一个eureka，同时每个分区至少有一个eureka服务器来处理本分区故障。

服务注册在Eureka上并且每30秒发送心跳来续租。如果一个客户端在几次内没有刷新心跳，它将在大约90秒内被移出服务器注册表。注册信息和更新信息会在整个eureka集群的节点进行复制。任何分区的客户端都可查找注册中心信息（每30秒发生一次）来定位他们的服务（可能会在任何分区）并进行远程调用。

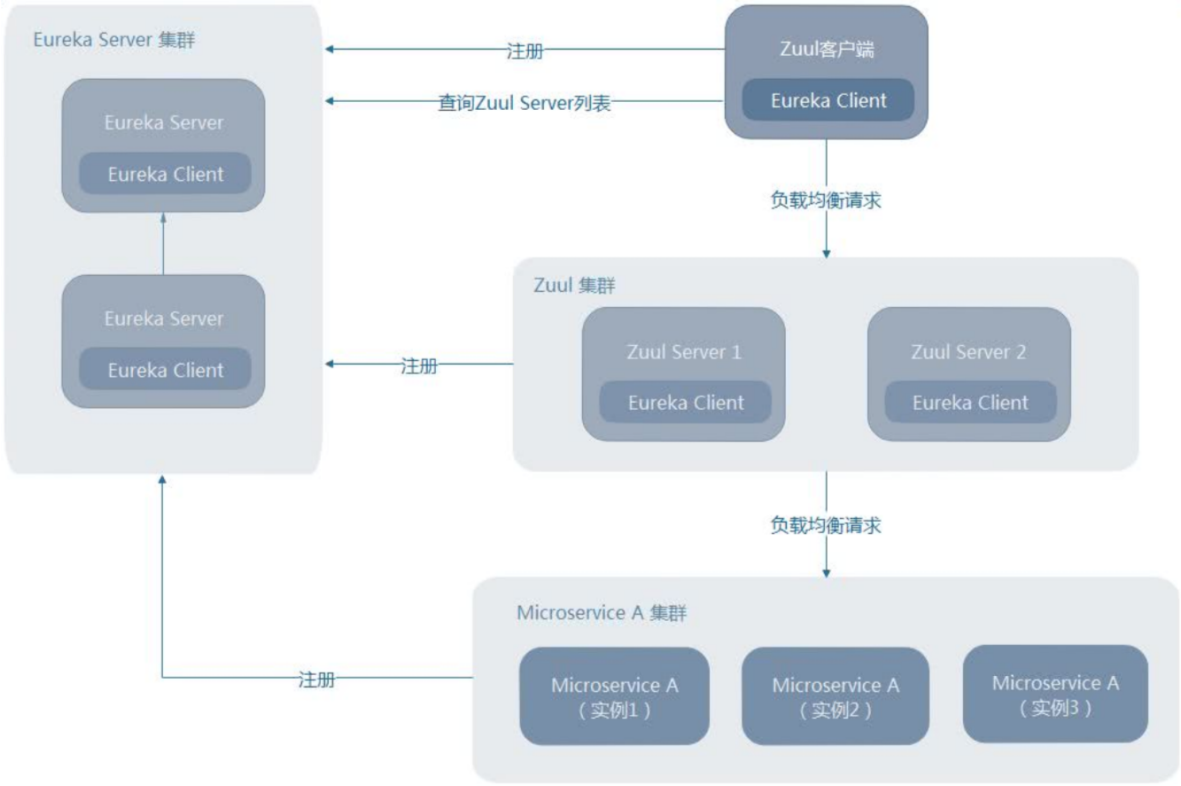
Zuul

可以从Spring Cloud组件架构图上看出，Zuul是所有外部设备请求的前门，它负责把请求转发到后端的真是微服务。所以作为网关它要具有如下图所示的这些功能点。



Zuul 是Netflix开源的一个网关服务器, 本质上是一个web servlet应用。Zuul 在云平台上提供动态路由，监控，弹性，安全等服务。

在Eureka存在的基础上，我们把Zuul也注册到Eureka服务器上，所以能够获取Eureka注册中心上的服务实例来进行请求转发。



Zipkin&&Sleuth

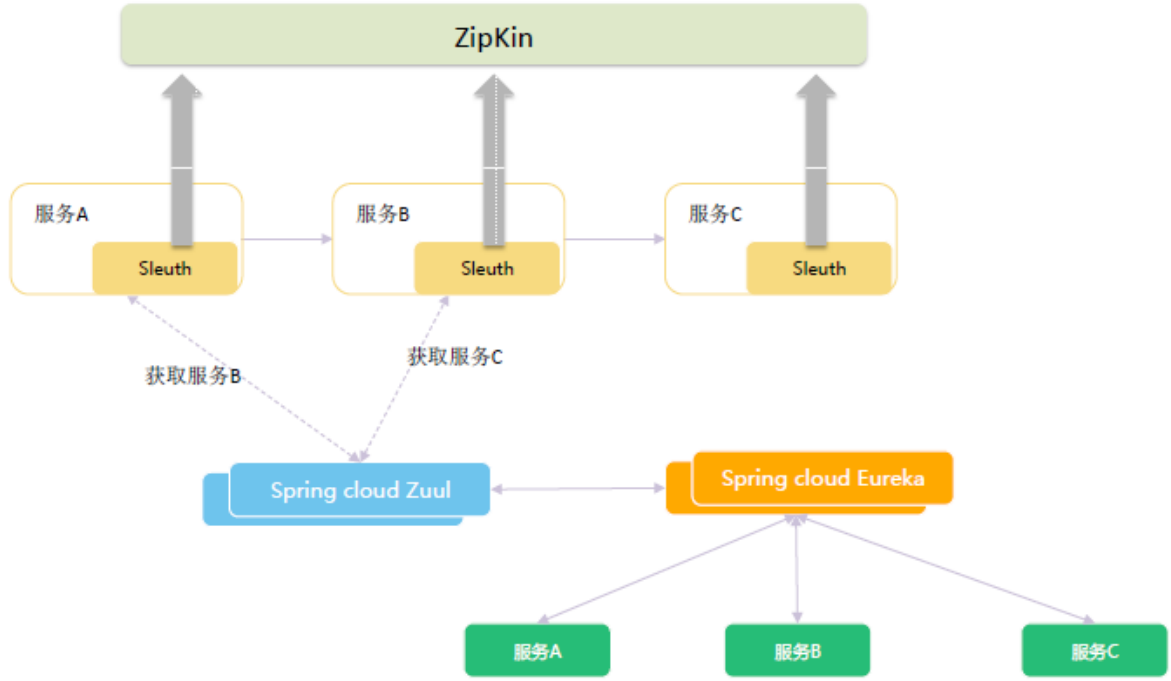
随着业务发展，系统拆分导致系统调用链路愈发复杂一个前端请求可能最终需要调用很多次后端服务才能完成，当整个请求变慢或不可用时，我们是无法得知该请求是由某个或某些后端服务引起的，这时就需要解决如何快读定位服务故障点，以对症下药。于是就有了分布式系统调用跟踪的诞生。

Spring Cloud Sleuth 为服务之间调用提供链路追踪。通过 Sleuth 可以很清楚的了解到一个服务请求经过了哪些服务，每个服务处理花费了多长。从而让我们可以很方便的理清各微服务间的调用关系。此外 Sleuth 可以帮助我们：

- 耗时分析: 通过 Sleuth 可以很方便的了解到每个采样请求的耗时，从而分析出哪些服务调用比较耗时;
- 可视化错误: 对于程序未捕捉的异常，可以通过集成 Zipkin 服务界面上看到;
- 链路优化: 对于调用比较频繁的服务，可以针对这些服务实施一些优化措施。

Zipkin 是 Twitter 的一个开源项目，它基于 Google Dapper 实现，它致力于收集服务的定时数据，以解决微服务架构中的延迟问题，包括数据的收集、存储、查找和展现。 我们可以使用它来收集各个服务器上请求链路的跟踪数据，并通过它提供的 REST API 接口来辅助我们查询跟踪数据以实现对分布式系统的监控程序，从而及时地发现系统中出现的延迟升高问题并找出系统性能瓶颈的根源。除了面向开发的 API 接口之外，它也提供了方便的 UI 组件来帮助我们直观的搜索跟踪信息和分析请求链路明细，比如：可以查询某段时间内各用户请求的处理时间等。Zipkin 提供了可插拔数据存储方式：In-Memory、MySql、Cassandra 以及Elasticsearch。

如图，在Eureka和Zuul存在的基础上，Sleuth和Zipkin用来解决微服务之间直接调用时的性能分析。



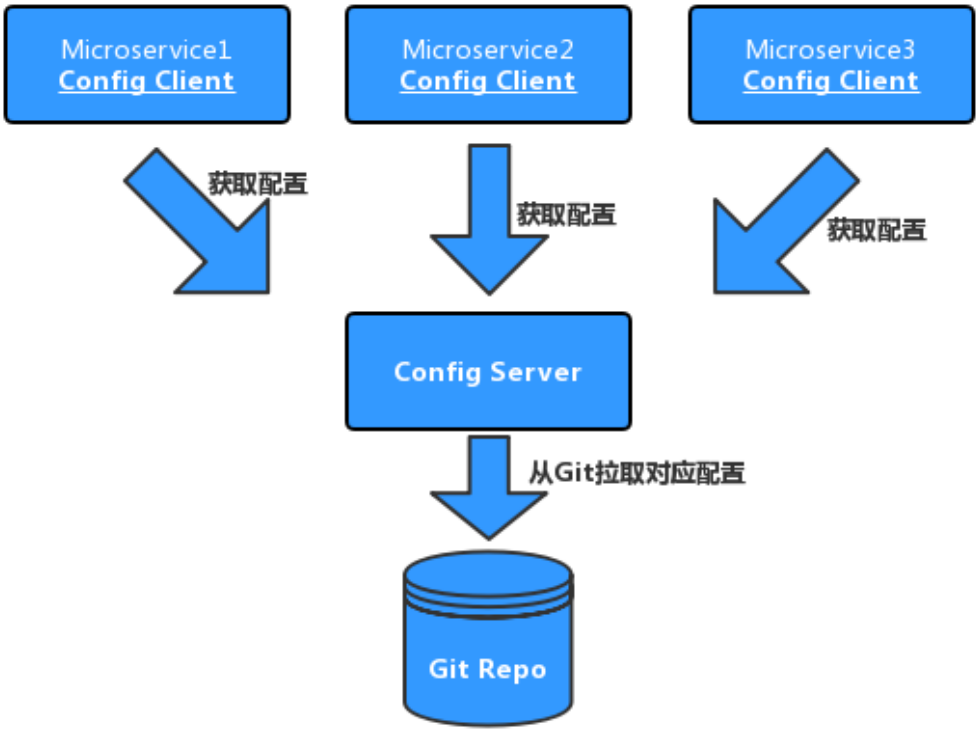
配置中心

Spring Cloud Config为分布式系统外部化配置提供了服务端和客户端的支持，它包括Config Server和Config Client两部分。

Config Server是一个可横向扩展、集中式的配置服务器，它用于集中管理应用程序各个环境下的配置，默认使用Git存储配置内容（也可使用Subversion、本地文件系统或Vault存储配置），因此可以很方便地实现对配置的版本控制与内容审计。

Config Client是Config Server的客户端，用于操作存储在Config Server中的配置属性。

Spring Cloud Config的架构图如下



所有微服务都指向Config Server。各个微服务在启动时，会请求Config Server以获取所需要的配置属性，然后缓存这些属性以提高性能。

微服务开发标准

在本节中，通过一个具体的项目实例，介绍该如何开发一个微服务项目。

项目介绍

功能点

paascloud（ https://github.com/paascloud/paascloud-master ）是模拟商城，有完整的购物流程、后端运营平台对前端业务的支撑，和对项目的运维，有各项的监控指标和运维指标

技术栈
核心框架：springcloud Edgware全家桶，Vye全家桶
安全框架：Spring Security Spring Cloud Oauth2
分布式任务调度：elastic-job
持久层框架：MyBatis、通用Mapper4、Mybatis_PageHelper
数据库连接池：Alibaba Druid
日志管理：Logback 前端框架：Vue全家桶以及相关组件
三方服务： 邮件服务、阿里云短信服务、七牛云文件服务、钉钉机器人服务、高德地图API#

目录结构说明

微服务架构目录形式

如果是采用Maven来进行构建的话，按照Maven分布式工程的基本架构来进行创建。

```
 microservice
---- microservice-common
---- microservice-modules
---- microservice-provider
---- microservice-provider-api
---- microservice-consumer
```

其中microservice为父工程模块，来定义整个工程的依赖版本，因为Spring Cloud中各个组件的版本都是有对应关系的，同时在根pom中定义项目中的模块。

```
<groupId>com.liuzm.paascloud</groupId>
<artifactId>paascloud-master</artifactId>
<version>1.0</version>
<modules>
    <module>paascloud-eureka</module>
    <module>paascloud-discovery</module>
    <module>paascloud-monitor</module>
    <module>paascloud-zipkin</module>
    <module>paascloud-gateway</module>
    <module>paascloud-provider-api</module>
    <module>paascloud-common</module>
    <module>paascloud-provider</module>
    <module>paascloud-generator</module>
</modules>
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <springfox.version>2.6.1</springfox.version>
    <commons-codec.version>1.10</commons-codec.version>
    <jjwt.version>0.7.0</jjwt.version>
    <codehaus-jackson.version>1.9.13</codehaus-jackson.version>
    <mybatis-starter-version>1.2.0</mybatis-starter-version>
    <mybatis.plus.version>2.0.7</mybatis.plus.version>
    <druid.version>1.0.29</druid.version>
    <mapper.version>3.4.0</mapper.version>
    <springframework.version>4.3.8.RELEASE</springframework.version>
    </properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
 </dependency>
</dependencies>
```

在下一级目录的Maven子工程中，要添加父pom的信息，

```
<parent>
    <groupId>com.liuzm.paascloud</groupId>
    <artifactId>paascloud-master</artifactId>
    <version>1.0</version>
</parent>
```

同理，在三级目录的maven子工程要依赖于二级maven工程的pom,这样利用Maven的依赖传递去控制所使用的构件的版本。

microservices-common包为公共模块，来定义一些公共的组件和配置类。

microservices-moudles是微服务架构中基础组件的代指（比如注册中心，配置中心，监控中心等等）。

provider为服务提供者，提供api供其它微服务使用，这两部分的api最好分包存放。

provider-api是提供api给其它微服务使用，这样做的目的是隐藏api的实现细节，重新定义两个微服务之间交互所需要使用的数据对象，应该要服务微服务建模时划分的限界上下文。

consumer为服务消费者，用于消费内部实现的微服务。实际上，提供者同时也可能是其它一些微服务的消费者，但是通常来说provider在命名定义的优先级更高，当使用其它微服务提供的api时，需要在pom中引入依赖，然后在services层实现自己服务的时候利用依赖注入注入其它微服务的api，利用Feign就像是像在本地操作数据库那样操作其它微服务的数据。以下是三步的演示。

```
<dependency>
    <groupId>com.liuzm.paascloud.provider.api</groupId>
    <artifactId>paascloud-provider-mdc-api</artifactId>
</dependency>
```

```
@Resource
private MdcProductQueryFeignApi mdcProductQueryFeignApi;
@Resource
private MdcProductFeignApi mdcProductFeignApi;
```

```
ProductDto product = mdcProductService.selectById(cartItem.getProductId());
```

此项目的目录结构

此项目选择了eureka作为注册中心，spring-cloud-config作为配置中心，hystrix dashboard和spring-boot-admin作为监控中心，netflix zuul作为网关，zipkin为日志采集中心。这些基础模块作为微服务架构中的服务治理部分。mdc、omc、opc、tpc、uac作为业务模块。

```
├─paascloud-master----------------------------父项目，公共依赖
|  |
|  ├─paascloud-eureka--------------------------微服务注册中心
|  |
|  ├─paascloud-discovery-----------------------微服务配置中心
|  |
|  ├─paascloud-monitor-------------------------微服务监控中心
|  |
|  ├─paascloud-zipkin--------------------------微服务日志采集中心
|  |
|  ├─paascloud-gateway-------------------------微服务网关中心
|  |
|  ├─paascloud-provider
|  |  |
|  |  ├─paascloud-provider-mdc-----------------数据服务中心
|  |  |
|  |  ├─paascloud-provider-omc-----------------订单服务中心
|  |  |
|  |  ├─paascloud-provider-opc-----------------对接服务中心
|  |  |
|  |  ├─paascloud-provider-tpc-----------------任务服务中心
|  |  |
|  |  └─paascloud-provider-uac-----------------用户服务中心
```

```
|  |
|  ├─paascloud-provider-api
|  |  |
|  |  ├─paascloud-provider-mdc-api------------------数据服务中心API
|  |  |
|  |  ├─paascloud-provider-omc-api------------------订单服务中心API
|  |  |
|  |  ├─paascloud-provider-opc-api------------------对接服务中心API
|  |  |
|  |  ├─paascloud-provider-tpc-api------------------任务服务中心API
|  |  |
|  |  ├─paascloud-provider-sdk-api------------------可靠消息服务API
|  |  |
|  |  └─paascloud-provider-uac-api------------------用户服务中心API
|  |
|  ├─paascloud-common
|  |  |
|  |  ├─paascloud-common-base-----------------公共POJO基础包
|  |  |
|  |  ├─paascloud-common-config-----------------公共配置包
|  |  |
|  |  ├─paascloud-common-core-----------------微服务核心依赖包
|  |  |
|  |  ├─paascloud-common-util-----------------公共工具包
|  |  |
|  |  ├─paascloud-common-zk-----------------zookeeper配置
|  |  |
|  |  ├─paascloud-security-app-----------------公共无状态安全认证
|  |  |
|  |  ├─paascloud-security-core-----------------安全服务核心包
|  |  |
|  |  └─paascloud-security-feign-----------------基于auth2的feign配置
|  |
|  ├─paascloud-generator
|  |  |
|  |  ├─paascloud-generator-mdc-----------------数据服务中心Mybatis Generator
|  |  |
|  |  ├─paascloud-generator-omc-----------------数据服务中心Mybatis Generator
|  |  |
|  |  ├─paascloud-generator-opc-----------------数据服务中心Mybatis Generator
|  |  |
|  |  ├─paascloud-generator-tpc-----------------数据服务中心Mybatis Generator
|  |  |
|  |  └─paascloud-generator-uac-----------------数据服务中心Mybatis Generator
```

组件使用说明

通过代码简单介绍Spring Cloud的组件应该如何使用，注意，这一切都是在Spring Boot的基础项目之上进行的操作。

Eureka

首先启动注册中心

- 添加maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

- 在启动类上添加 `@EnableEurekaServer` 注解
- 启动项目即可

接下来就可以把自己写的微服务添加到注册中心，

- 添加maven依赖

  ```
  <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  </dependency>
  ```

- 启动类上添加注解 `@EnableDiscoveryClient`
- 添加配置为注册中心的地址并且配置自己的实例名:

  ```
  eureka:
    client:
      serviceUrl:
        defaultZone: http://root:root@paascloud-eureka:8761/eureka/
    instance:
      instance-id:
  ${spring.application.name}:${spring.cloud.client.ipAddress}:${server.port}
  ```

  也可以设置一些连接参数如心跳时间等等。

这样在eureka提供的web界面上就可以看到注册的服务和它们的状态。

更高级的使用方法比如eureka的高可用部署模式以及提供的API这里不赘述。

config

同样也要启动配置中心

- 首先添加maven依赖

  ```
  <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
      </dependency>
  ```

- 在启动类添加注解 `@EnableConfigServer`
- 配置配置仓库的地址

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/paascloud/paascloud-config-repo.git
          username:
          password:
          search-paths: /*
          default-label: ${spring.profiles.active}
          basedir: /data/config/paascloud-config-repo
```

配置仓库是配置中心搜寻服务配置的地方，这里存放了所有的配置文件，配置中心根据客户端配置的文件名寻找它的配置文件并且发送给它。

- 启动即可

也可以将配置中心作为一个微服务同样注册到注册中心，这样在微服务中就不需要些配置中心的实际地址而只写逻辑名，假设使用了这种方法再将微服务接入到配置中心。

- 首先添加依赖

  注意，只要添加了此项依赖，如果不进行配置的话也会在本地8888端口寻找配置文件。

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

- 新建bootstrap的配置文件，配置配置中心所在的位置，这里直接写了逻辑名。所寻找的配置文件的名字是applicationname-label-profile。

```yaml
spring:
  cloud:
    config:
      fail-fast: true
      discovery:
        service-id: paascloud-discovery
        enabled: true
      label: ${spring.profiles.active}
      profile: ${spring.profiles.active}
      username: admin
      password: admin
```

高级应用是利用git的webhook功能和Spring Cloud Bus实现当配置刷新的时候自动更新程序的配置而不用将应用重新部署。

Monitor

监控中心使用了Hystrix的Dashboard监控微服务之间的调用关系，Hystrix的Turbine监控集群微服务的调用健康关系，spring-boot-admin监控单个服务的健康状态。

- 添加依赖

```xml
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
    <dependency>
```

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    </dependency>
    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-server</artifactId>
    </dependency>
    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-server-ui</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-server-ui-hystrix</artifactId>
    </dependency>
    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-server-ui-turbine</artifactId>
    </dependency>
```

- 添加注解

```
@EnableTurbine
@EnableHystrixDashboard
@EnableCircuitBreaker
@EnableAdminServer
```

- 启动

  就可以在这三个监控组件自带的前端界面上看到我们系统中微服务的健康情况。

  对于客户端的使用,spring-boot-admin的使用比较简单，添加依赖注解然后配置admin server的位置即可，hystrix可以集成在feign中使用，在后面介绍。

网关

网关也是一个单独的项目，这里使用Netflix的zuul

- 首先添加注解

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

- 启动类添加注解 `@EnableZuulProxy`
- 启动

这样是一个空的网关，需要在配置文件中添加路由规则（即访问路径和服务实例之间的对应关系，支持硬编码和服务化的路由规则），此外要实现权限、错误控制等功能需要添加filter来实现，这样实现网关虽然在功能的实现上会简单，但是会造成网关变成一个庞然大物，所以在设计网关的具体功能时还是要仔细考虑。

Feign

Spring Cloud使用Netflix的Hystrix做错误处理，Ribbon做负载均衡，而Feign就是这两者的综合，推荐在微服务之间的调用都使用Feign client代理，这样只需要很少的配置就可以实现两大功能。具体的使用说明如下:

- 添加依赖

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-feign</artifactId>
    </dependency>
```

- 在provider-api中有一个这样的方法

```
@Component
public class MdcAddressQueryFeignHystrix implements MdcAddressQueryFeignApi {

    @Override
    public Wrapper<AddressDTO> getById(final Long addressId) {
        return null;
    }
}
```

而自身有一个服务需要使用远程服务的api，所以使用 `@FeignClient` 注解，value代表要调用的服务实例，如果这个服务实例名会有多个会自动做负载均衡（Ribbon实现），fallback代表如果调用失败会调用此函数进行服务降级（Hystrix实现）。

```
@FeignClient(value = "paascloud-provider-mdc", configuration =
OAuth2FeignAutoConfiguration.class, fallback = MdcAddressQueryFeignHystrix.class)
public interface MdcAddressQueryFeignApi {

    /**
     * Select by id wrapper.
     *
     * @param addressId the address id
     *
     * @return the wrapper
     */
    @PostMapping(value = "/api/address/getById/{addressId}")
    Wrapper<AddressDTO> getById(@PathVariable("addressId") Long addressId);
}
```

这是最基本的实现，可以通过Hystrix设置超时的时间和重试的次数，对于服务降级需要根据实际的需求进行设计，服务降级可以进行多层，推荐服务降级的最后一层是不依赖于网络的，这样能够保证起码能够返回一些东西不至于被卡死。

链路追踪与Zipkin

Zipkin是一个收集信息的工具，它需要单独启动，可以和数据库配合做所搜集信息的持久化，这里不在赘述。

在每一个微服务中，

- 添加依赖

```
 <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

- 配置zipkin server所在的位置，默认是本地的9411

  Sleuth会将调用链路中的信息进行收集并且发送到zipkin，在Zipkin提供的面板中可以看到微服务之间的调用关系。