目錄

Microservices-Book

由微服务的概念入门，从浅至深，逐步学习 Spring Boot/Cloud，理解其中的各种关系，笔记电子书。

Spring Boot Version：2.1.4.RELEASE

Spring Cloud Version：暂无

Github：iqeq00

Gitee：iqeq00

GitBook：从这里开始阅读

Email：iqeq00@gmail.com

QQ：24761297

Overview

本章主要针对微服务的理论学习，了解理论，便于实践。

序号	标题	文件
4-5	微服务架构和单体应用架构	MicroserviceAndMonolithic.md
6	微服务架构特征	MicroserviceCharacteristics.md
7	微服务治理与去中心化	DecentralizedGovernance.md
8	微服务演进式设计与优缺点	EvolutionaryDesign.md
9	微服务宏观把控与深入	MacroControl.md

微服务架构和单体应用架构

微服务架构

微服务是一种软件架构风格，由 Martin Fowler 提出来。

软件开发领域没有银弹，一个架构或者框架有好的一面，一定也有不好的一面。

微服务架构定义

- 开发单个应用的一种开发模式。
- 单个应用作为小型服务套件的一部分。
- 每个应用都运行在自己的进程中。
- 应用于应用之间的通信，是通过轻量级的机制，通常是 HTTP 的资源 API。
- 服务通常围绕业务能力进行构建。
- 服务可以独立部署。
- 服务之间没有中央化的管理模式。
- 每个服务可以通过不同的编程语言来编写，可以使用不同的数据存储技术。

微服务变得复杂的地方

- 事务

  事务处理会变得非常复杂。

- 部署

  进行细粒度的功能拆分以后，会多出多个项目，每个都项目需要部署。

- 调用链追踪

  为了快速定位可能产生的问题，需要对整个调用链非常清楚。

传统单体架构

单体应用

一个单体应用通常构建为一个独立的单元。

企业应用

企业应用通常是构建在三个主要部分之上。

- 客户端用户界面

  包含 html、js

- 数据库

  包含表、关系数据库、数据库管理系统

- 服务端应用

  处理 http 的请求，执行领域逻辑，获取、更新数据库的数据，选择并且装配 html 视图，并且把视图发送给浏览器。服务端应用是一个单体的可执行的逻辑。对于系统的任何修改，都涉及到构建、部署服务端应用的一个新版本。

两种架构风格对比

这种单体的服务器，是一种这样系统自然而然的构建方式。用于处理一个请求的所有的逻辑都会运行在一个单个的进程当中，这样的话，就可以使用编程语言的基本特性，将应用分解成类、函数、命名空间。可以水平的对单体应用进行可伸缩的处理（扩容、减容），方式就是在一个负载均衡器后面运行多个实例。

单体应用非常成功，但是逐渐地人们开始对它们感受到沮丧，特别是有更多的应用被部署到云端的时候，这种变化都联系到了一起，对于一个应用任何一个非常很小的修改和改变，都需要让整个单体应用进行一次重新构建和部署。随着时间的推移，我们很难保持很好的模块化结构，这个使得保持只对一个模块的变更变得越来越困难了。可伸缩就要求整个应用进行可伸缩，而不是应用的部分进行可伸缩，这种做法就需要更多的资源。

比如：我只想对商品详情页面进行更好的可伸缩，而公告、投诉这些模块不需要，要是单体应用就只有整个应用一起可伸缩了，不能只针对一个具体的模块进行可伸缩。

单体应用的可伸缩

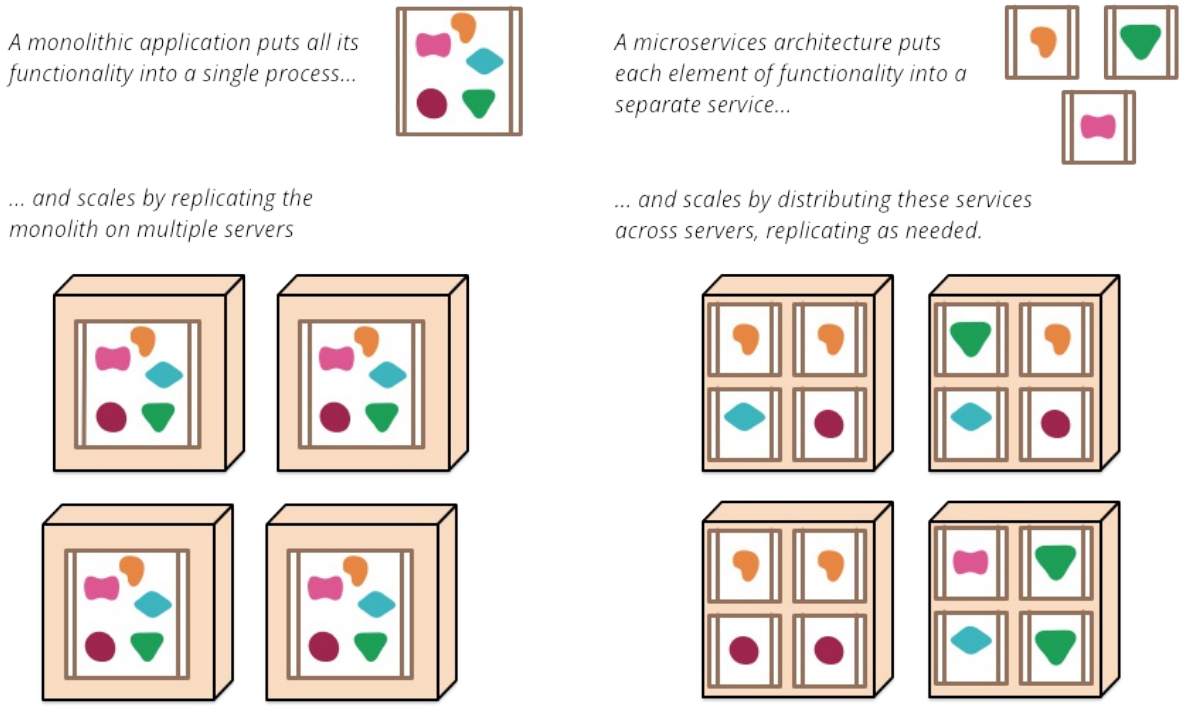
一个单体应用会将它所有的功能都放置在一个进程中，单体应用典型的表现形式。通过复制这个单体应用在多个机器上面，比如要部署4个单体应用，就需要复制成 4 份，分别放置在 4 台机器上。通过在多个服务器上去复制一个单体应用的实例来完成可伸缩，而且每一台机器的实例内容都是一样的。

微服务应用的可伸缩

微服务架构会将每一个功能元素放置在单独的服务当中，服务与服务之间是完全独立的。通过跨服务器来分发这些服务，在需要的时候才进行复制。非常的灵活，完全根据需要来复制。而且每一台机器的实例内容可以一样，也可以完全不一样。

可伸缩对比

如下图所示（左边为单体应用，右边为微服务应用）：



这些沮丧导致了微服务架构风格的出现，构建应用做为服务的套件，服务是独立的、可部署的、可伸缩的。每一个服务都提供了一个坚实的模块边界，甚至允许不同的服务可以用不同的编程语言来编写，它们还可以由不同的团队来进行管理。

我们不会去表明微服务架构风格是非常创新卓越的，它可以追溯到 Unix 的设计原则。但是我们认为，并没有太多的人去考虑过这种微服务架构。并且我们也会认为，很多软件开发如果使用了微服务架构，都会从中获益匪浅。

参考资料

Martin Fowler

微服务架构特征

我们不能说存在这种微服务架构风格的正式定义，我们可以尝试去描述我们所看到的对于这种架构风格通用、共同的特征。并不是所有的微服务架构都会拥有所有的特征，但是我们希望大多数微服务架构都可以展现出我们所描述的大多数特征。我们并不会给出一些强制遵循的标准，微服务只是一种风格，而不是标准。给出指导原则，但不是强制的。

组件化和服务

我们希望系统的构建方式，是通过不同的组件之间来可插拔搭配完成的，这种方式和现实世界中所看到的很多事物都是类似的。

当我们谈论组件的时候，其实陷入了一种很困难的定义，到底是什么构成了一个组件。我们的定义：组件就是一个软件单元，它是独立的、可替换的、可升级的。

微服务架构会使用到库，但是它们主要的组件化方式是将软件分解成一个个的服务形态。我们将库定义成组件，它们可以连接到程序中，也可以通过内存当中的函数调用来进行调用。而服务形态是进程外的组件，跨进程的，它们通信机制是通过 web service、remote procedure call（远程过程调用）。与面向对象程序中的服务对象来比，是一个完全不同的概念。

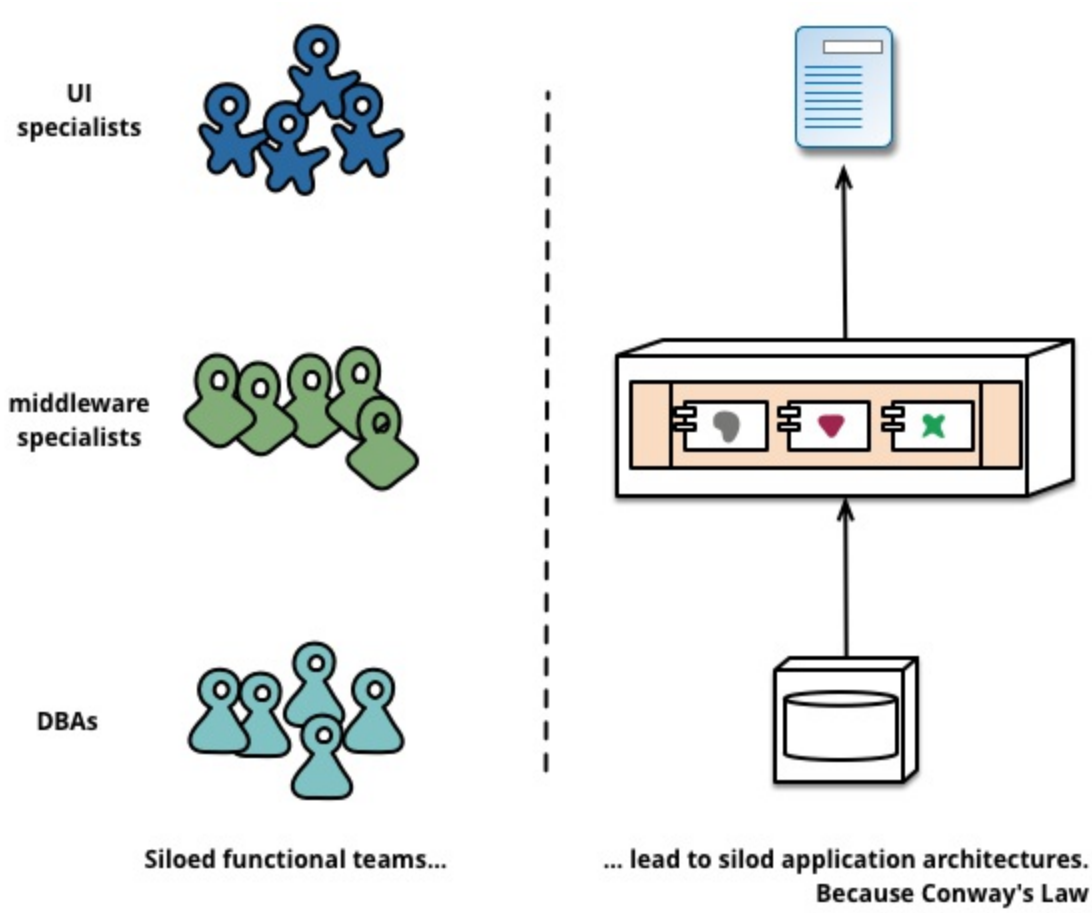
使用服务作为组件，而非库的一个主要原因是，服务是可以独立部署的。如果你拥有一个应用，它是由一个进程当中的多个库组成的。对于任何一个组件的修改，都会导致整个应用需要重新部署。但是如果这个应用是被分解成多个服务的话，就可以期望很多单个服务的改变，只会需要重新部署被修改的单个服务。这并不是绝对的，一些更新还会改变服务接口本身，比如这个服务对外提供的接口从三个参数变成了四个参数，这就会影响到所有调用这个服务的其他微服务，也需要跟着修改，这种修改可能会很麻烦，会涉及到很多其他微服务的改动、部署。只有只修改了服务内部实现，而服务对外提供的接口或者锲约没有发生什么改变，这种情况才只需要重新部署被修改的微服务本身就可以了。但是一个好的微服务架构目标是，通过明确的服务边界和演化机制来把这种改变最小化。

将服务作为组件的另一个结果更加的显示组件接口。大多数语言都没有提供一个良好的机制用来定义一个显示的发布接口的一种方式。通常这并不仅仅是文档和原则性的问题，来去防止客户端去破坏一个组件的封装原则。而且这会导致过于紧密的耦合，使组件间的耦合性过于紧密。通过服务的方式可以使得这个问题变得容易，避免这一点，方式是使用显示的远程调用机制。

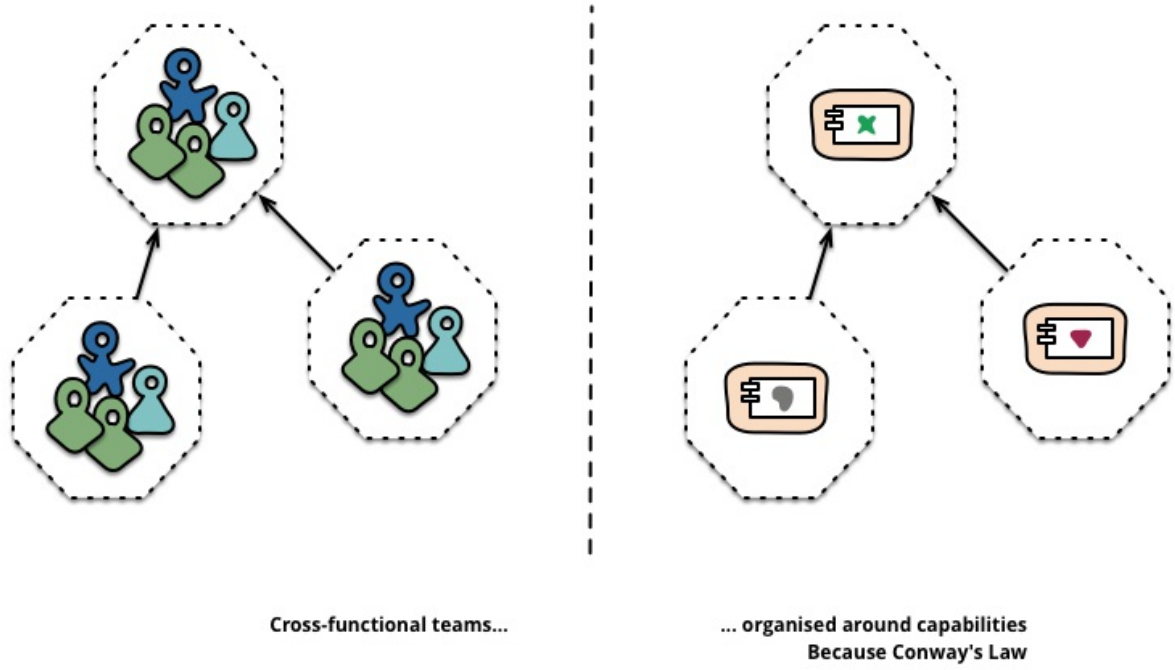
按照这种方式来使用服务也有一些缺点，远程调用通常要比进程内的调用成本更高。而且远程 API 需要粗粒度的设计，通常用起来感觉很笨拙。如果你需要改变组件间的职责分配，如果要跨越进程边界，行为的移动是更加困难的，比如把很多逻辑从 A 服务放置到 B 服务里。

我们可以观察到服务是可以映射成运行期的进程，但是这只是一种大致的描述，一个服务是可以包含多个进程。这些进程总是一起被开发，部署的，这样的应用进程和数据库只是被这个服务自己所独有的。

当我们去观察一个大型应用拆分若干个部分的时候，通常管理会关注在技术层面上，会导致UI团队、服务端逻辑团队、数据库团队等等。当这些团队按照服务线的方式进行软件架构风格，即使是一个小小的改变，都会导致一个跨团队的项目的沟通、批准。一个很聪明的团队通常会对其进行优化，减少这种情况的发生，会强制这种逻辑放置到他们应该能访问到的应用当中。



如上图，如果我们把架构设计成单体应用，大家都各辞其职。UI 做 UI、中间件做中间件、DBA 做 DBA。在某一个业务当中，这个服务可能同时涉及到 UI、中间件、DBA 这几种工作。会糅合多种人员在同一个业务内，会使得团队与团队之间的沟通不像以前那么简单而又直接，可能会出现一些修改一个地方涉及到多个部门通力配合的情况。



微服务的方式则和上面的单体应用不一样，它会将服务按照业务能力来组织。这种服务会接受一个针对业务领域的软件实现，包括用户界面、持久层存储、额外的一些协作。结果导致团队是跨越功能，包含用户界面、数据库、项目管理等完整技能的团队。

微服务架构是围绕业务能力展开的，和单体应用差距很大。

大型的单体应用总是可以围绕业务进行模块化，虽然这并不是常见的情况。我们可以督促一个大型团队按照业务线来分解、构建单体应用。主要的问题是他们趋向于围绕着太多的上下文。如果单体应用可以按照模块化来拆分的话，那么对于每一个团队成员来说都很难去短期适应组织。此外我们还会看到模块化这种方式需要很好的一个纪律去遵循，这种必要性会增加服务组件边界清晰化的困难。

微服务治理与去中心化

产品并非项目

我们所能看到的大多数应用程序开发都使用一个项目模型：目标是交付软件，目的是交付一些软件，然后将其视为已完成。完成后，软件将移交给维护组织，构建它的项目团队将被解散。

微服务支持者倾向于避免这种模型，而是倾向于认为团队应该在其整个生命周期内拥有产品。 对此的一个共同启示是亚马逊的概念“你构建，运行它”，开发团队对生产中的软件负全部责任。 这使开发人员能够日常接触他们的软件在生产中的行为，并增加与用户的联系，因为他们必须承担至少一些支持负担。

产品心态，与业务能力的联系紧密相连。 不是将软件视为一组要完成的功能，而是存在一种持续的关系，其中的问题是软件如何帮助其用户增强业务能力。

没有理由为什么这种方法不能与单一应用程序一起使用，但较小的服务粒度可以使创建服务开发人员与其用户之间的个人关系变得更加容易。

智能终端和哑管道

当构建不同进程之间的通信结构时，我们会看到许多产品和方法都强调将重要的智慧放入通信机制本身。 一个很好的例子是企业服务总线（ESB），其中 ESB 产品通常包括用于消息路由，编排，转换和应用业务规则的复杂工具。

而微服务社区倾向于采用另一种方法：智能终端和哑管。 从微服务构建的应用程序旨在尽可能地解耦和内聚 - 它们拥有自己的领域逻辑，看起来像传统、经典的 Unix 上的过滤器 - 接收请求，适当地应用逻辑并产生响应。 这些是使用简单的 RESTish 协议而不是复杂的协议（如 WS-Choreography 或 BPEL 或中央工具的编排）编排的。

最常用的两种协议是 HTTP 请求 - 响应资源 API 和轻量级消息传递。 第一个最好的表达方式是 web 的一部分，但没有隐藏在 web 后面。

微服务团队使用万维网（在很大程度上，Unix）构建的原则和协议。 经常使用的资源可以通过开发人员或操作人员的非常小的努力来缓存。

常用的第二种方法是通过轻量级消息总线进行消息传递。 选择的基础设施通常是愚蠢的（如同仅作为消息路由器那样愚蠢） - 像 RabbitMQ 或 ZeroMQ 这样的简单实现不仅仅提供可靠的异步结构 - 智能仍然存在于生成和消费消息；在服务中。

在一个单体应用当中，组件在进程中执行，它们之间的通信是通过方法调用或函数调用。 将单体应用变为微服务的最大问题在于改变通信模式。 从内存中方法调用到 RPC 的简单转换导致繁琐的通信，这些通信效果不佳。 相反，您需要用粗粒度的方法替换细粒度的通信。

注意：跨进程的调用，一定要把接口定义的粗粒度，一次调用尽可能拿到所有需要的数据。相反，传统进程内的调用，越细越好。

去中心化的治理

中心化治理的后果之一是在单一技术平台上实现标准化的趋势。 经验表明，这种方法是有限的 - 不是每个平台都是一样的，也不是每个解决方案都是一致的。 我们推荐使用正确的工具来完成正确的工作，然而传统单体应用程序可以在一定程度上利用不同的语言，但这种做法并不常见。

将单体应用的组件划分成服务之后，我们在构建每个服务时都有选择。您想使用 node.js 来支持一个简单的报告页面吗？去争取它。C++ 用于一个特别接近实时的组件？好的。您想换一种更适合一个组件的读取行为的不同风格的数据库吗？我们有技术来重构他。

当然，仅仅因为你可以做某事，并不意味着你应该这么去做 - 但以这种方式对系统进行划分意味着你可以选择。

构建微服务的团队也喜欢使用不同的标准方法。与其使用一套写在纸上的已定义标准，他们更愿意开发出有用的工具，其他开发人员可以使用这些工具来解决与他们所面临的问题类似的问题。这些工具通常是从实现中获得的，并与更广泛的组共享，有时，但不是仅使用内部开放源代码模型。现在 Git 和 Github 已经成为事实上的版本控制系统的选择，开源实践在企业内部越来越普遍。

Netflix 是遵循这一理念的组织的一个很好的例子。 共享有用的，最重要的是经过实战考验的代码，因为库鼓励其他开发人员以类似的方式解决类似问题，但如果需要，可以选择不同的方法。 共享库往往侧重于数据存储，进程间通信的常见问题，我们将在下面进一步讨论基础架构自动化。

对于微服务社区来说，管理费用特别缺乏吸引力。这并不是说社区不重视服务合同。恰恰相反，因为往往会有更多。只是他们正在寻找管理这些合同的不同方式。像容忍阅读器和消费者驱动的合同这样的模式通常应用于微服务。这些援助服务合同独立发展。在构建过程中执行消费者驱动的合同可以增强信心，并提供有关您的服务是否正常运行的快速反馈。事实上，我们知道澳大利亚的一个团队通过消费者驱动的合同推动新服务的建设。他们使用简单的工具来定义服务合同。在编写新服务的代码之前，这将成为自动构建的一部分。然后，该服务仅在满足合同的情况下构建 - 这是在构建新软件时避免 'YAGNI' 困境的优雅方法。这些技术和围绕它们成长的工具通过减少服务之间的时间耦合来限制中央合同管理的需要。

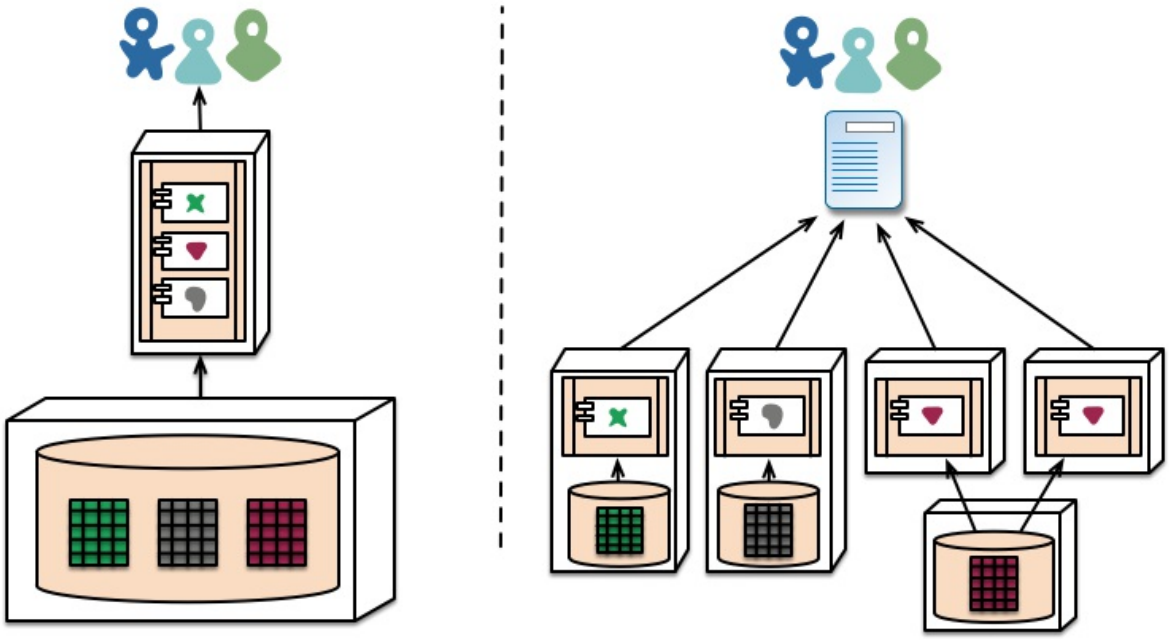
也许分散治理的最高点是建立它/运行它由亚马逊推广的精神。 团队负责他们构建的软件的所有方面，包括全天候运行软件。 这种责任水平的下放绝对不是常态，但我们确实看到越来越多的公司将责任推向开发团队。 Netflix 是另一个采用这种精神的组织。 每天晚上凌晨 3 点被您的寻呼机唤醒，无疑是在编写代码时专注于质量的强大动力。 这些想法与传统的集中治理模式相差甚远。

去中心化的数据管理

数据管理的去中心化以多种不同的方式呈现。 在最抽象的层面上，它意味着世界的概念模型在不同系统之间会有所不同。 在整合大型企业时，这是一个常见问题，客户的销售视角将与支持视角不同。 在销售视角中称为客户的某些内容可能根本不会出现在支持视角中。 那些做的可能具有不同的属性和（更糟糕的）具有微妙不同语义的共同属性。

此问题在应用程序之间很常见，但也可能在应用程序中发生，特别是在将应用程序划分为多个单独的组件时。 一种有用的思考方式是有界上下文的领域驱动设计概念。DDD 将复杂域划分为多个有界上下文，并映射出它们之间的关系。 此过程对单体应用和微服务体系结构都很有用，但服务和上下文边界之间存在自然关联，这有助于澄清，正如我们在业务功能部分中所述，强化了分离。

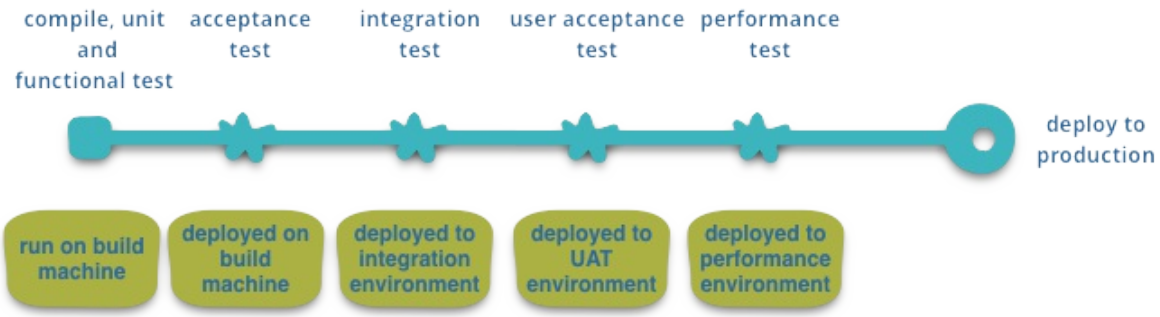
除了关于概念模型的分散决策之外，微服务还分散了数据存储决策。 虽然单一应用程序更喜欢使用单个逻辑数据库来存储持久性数据，但企业通常更喜欢在一系列应用程序中使用单个数据库 - 其中许多决策是通过供应商围绕许可的商业模型来实现的。 微服务更喜欢让每个服务管理自己的数据库，可以是同一数据库技术的不同实例，也可以是完全不同的数据库系统 - 这种方法称为 Polyglot Persistence。 您可以在整体中使用多语言持久性，但它在微服务中更常出现。



基础设施自动化

基础设施自动化技术在过去几年中发生了巨大变化 - 特别是云和 AWS 的发展降低了构建，部署和运行微服务的操作复杂性。

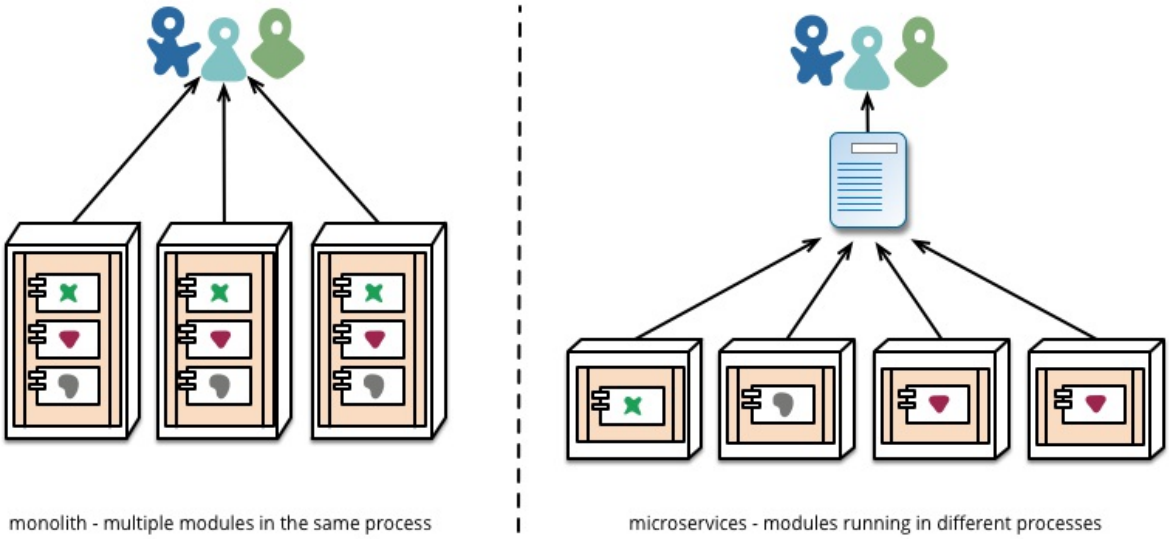
许多使用微服务构建的产品或系统都是由具有丰富的持续交付经验的团队构建的，并且是前身的持续集成。 以这种方式构建软件的团队广泛使用基础设施自动化技术。 这在下面显示的构建管道中说明。



由于这不是关于持续交付的文章，我们将在这里引起注意几个关键功能。 我们希望尽可能多的信心使我们的软件正常工作，因此我们进行了大量的自动化测试。 推广工作软件“向上”管道意味着我们自动部署到每个新环境。

一个单体应用程序将在这些环境中非常随意地构建、测试和推送。事实证明，一旦您投资于自动化一个整体的生产路径，那么部署更多的应用程序就不再那么可怕了。记住，持续交付的目标之一是使部署变得简单，因此无论是它的一个或三个应用程序，只要它仍然简单，就不重要了。

我们看到团队使用广泛的基础设施自动化的另一个领域是管理生产中的微服务。 与我们上面的断言相反，只要部署很简单，单体应用和微服务之间没有太大的区别，每个部署的运营环境可能会截然不同。



针对于失败来设计

使用服务作为组件的结果是，需要设计应用程序以便它们能够容忍服务的失败。 由于供应商不可用，任何服务调用都可能失败，客户必须尽可能优雅地对此做出响应。 与单体应用设计相比，这是微服务一个缺点，因为它引入了额外的复杂性来处理它。 结果是微服务团队不断反思服务失败如何影响用户体验。Netflix 的 Simian Army 在工作日引发服务甚至数据中心的故障，以测试应用程序的弹性和监控。

这种生产中的自动化测试足以让大多数运维团队在休息一周之前就会发抖。 这并不是说整体式建筑风格不具备复杂的监控设置 - 在我们的经验中它不常见。

由于服务可能随时发生故障，因此能够快速检测故障并在可能的情况下自动恢复服务非常重要。 微服务应用程序非常重视应用程序的实时监控，检查架构元素（数据库每秒获得多少请求）和业务相关度量（例如每分钟收到多少订单）。 语义监控可以提供出现问题的早期预警系统，从而触发开发团队跟进和调查。

注意：熔断器概念就是因为这个原因出来的，Spring Cloud Hystrix 的由来。

微服务演进式设计与优缺点

演进式设计

微服务实践者通常来自演进式设计背景，并将服务分解视为进一步的工具，使应用程序开发人员能够控制应用程序中的更改，而不会降低变更速度。 变更控制并不一定意味着改变 - 通过正确的态度和工具，您可以对软件进行频繁，快速和良好控制的更改。

每当您尝试将软件系统分解为组件时，您就面临着如何划分各个部分的决定 - 我们决定将应用程序分割的原则是什么？ 组件的关键属性是独立替换和可升级性的 - 这意味着我们寻找可以想象在不影响其协作者的情况下重写组件的点。 实际上，许多微服务组织通过明确期望许多服务被废弃而不是长期发展来进一步考虑这一点。

Guardian 网站是一个设计和构建为整体的应用程序的一个很好的例子，但是已经在微服务方向上发展。 单体应用仍然是网站的核心，但他们更喜欢通过构建使用单体应用 API 到微服务的演进。 这种方法对于本质上是临时的功能尤其方便，例如处理体育赛事的专用页面。 网站的这一部分可以使用快速开发语言快速组合在一起，并在事件结束后删除。 我们在金融机构看到了类似的方法，为市场机会添加新服务，并在几个月甚至几周后丢弃。

这种对可替换性的强调是模块化设计的更一般原则的一个特例，即通过变化模式驱使我们朝着模块化方向演进。 您希望在同一模块中保持同时更改的内容。 很少变化的系统部分应该与目前正在经历大量流失的系统处于不同的服务中。 如果您发现自己一再改变两项服务，那就表明它们应该合并。

将组件放入服务中可以为更细粒度的发布计划添加机会。 对于单体应用，任何更改都需要完整构建和部署整个应用程序。 但是，使用微服务，您只需要重新部署您修改的服务。 这可以简化并加快发布过程。 缺点是您必须担心一项服务的变化会其消费者出现问题。 传统的集成方法是尝试使用版本控制来解决这个问题，但微服务领域，不得已才采用版本控制这种方式。我们可以通过将服务设计为对供应商变更尽可能宽容来避免大量版本控制。

注意：上面的“版本控制”指代的是 api 的版本，比如某个 APP 从 2.0 升级到了 4.0。内部的 api 也跟着升级了，从 2.0 升级到 4.0，为了保证用户不更新 app 也能正常使用，所以这里 api 就会维护多个版本。

微服务是未来么？

我们写这篇文章的主要目的是解释微服务的主要思想和原则。 通过花时间来做到这一点，我们清楚地认为微服务架构风格是一个重要的想法 - 值得认真考虑企业应用程序。 我们最近使用这种方式构建了几个系统，并了解其他已经使用并支持这种方法的系统。

我们知道谁在某种程度上开创了架构风格，包括亚马逊，Netflix，The Guardian，the UK Government Digital Service，realestate.com.au，Forward and comparethemarket.com。 2013年的会议电路充满了一些公司的例子，这些公司正在转向可以归类为微服务的公司 - 包括 Travis CI。 此外，有很多组织长期以来一直在做我们称之为微服务的东西，但没有使用过这个名字。 （通常这被标记为 SOA - 尽管如我们所说，SOA 有许多相互矛盾的形式。 [15]）

然而，尽管有这些积极的经验，但我们并不认为我们确信微服务是软件架构的未来发展方向。 虽然到目前为止我们的经验与单体应用相比是积极的，但我们意识到没有足够的时间让我们做出充分的判断。

通常，您的架构决策的真正后果只有在您制作它们几年后才会明显。 我们已经看到一个项目，一个优秀的团队，对模块化的强烈渴望，已经建立了一个多年来已经腐朽的单体应用架构。 许多人认为微服务不太可能出现这种衰退，因为服务边界是明确的，很难修补。 然而，在我们看到足够多的系统、足够多的时间之前，我们无法完全认为微服务架构是已经成熟了。

人们可能会期望微服务成熟得很好。 在组件化的任何努力中，成功取决于软件在组件中的适用程度。 很难弄清楚组件边界的确切位置。 演进式的设计会认识到正确边界的困难，因此很容易重构它们的重要性。 但是当您的组件是具有远程通信的服务时，那么重构比使用进程内库要困难得多。 跨服务边界移动代码很困难，需要在参与者之间协调任何接口更改，向后兼容也会变得复杂，并且测试变得更加复杂。

上面加粗的地方就是微服务架构的主要缺点！

另一个问题是如果组件没有清晰地组成，那么您所做的就是将复杂性从组件内部转移到组件之间的连接。 这不仅仅是移动复杂性，而是将其移动到一个不那么明确且难以控制的地方。 当你在一个小而简单的组件内部查看时，很容易认为事情会更好，同时缺少服务之间的混乱连接。（复杂性的转移）

最后，还有团队技能的因素。 新技术往往被更熟练的团队所采用。 但对于技能更高的团队来说，一种更有效的技术并不一定适用于技能较低的团队。 我们已经看到很多不太熟练的团队构建混乱的单体应用架构，但是当微服务发生这种混乱时，需要花时间看看会发生什么。 一个糟糕的团队总是会创建一个糟糕的系统 - 很难说微服务是否可以减少这种情况下的混乱或使情况变得更糟。（微服务架构是跟团队中每一个人的能力是息息相关的）

我们听到的一个合理的论点是，您不应该从微服务架构开始。 相反，从单体应用开始，保持模块化，并在单体应用成为问题时将其拆分为微服务。（虽然这个建议并不理想，但是好的进程内接口通常不是一个好的服务接口。）

因此，我们谨慎乐观地写下这一点。 到目前为止，我们已经看到了足够多的微服务风格，觉得它是一条值得走的路。 我们无法确定最终会在哪里结束，但软件开发的挑战之一是您只能根据您当前必须提供的不完善信息做出决策。

微服务宏观把控

架构演进过程

1. 单体应用架构模式
2. 单体应用优化架构模式
3. ESB 企业总线架构模式

   SOA 架构主要针对企业级、采用 ESB 服务（ESB企业服务总线），非常笨重，需要序列化和反序列化，采用 XML 格式传输。ESB 也可以说是传统中间件技术与 XML、Web Service 等技术相互结合的产物。

4. 微服务架构模式

   微服务架构主要用于互联网公司，轻量级、小巧、独立运行，基于 Http、REST、JSON 格式传输。

单体架构的缺点

- 复杂性逐渐变高
- 技术债务逐渐上升
- 部署速度逐渐变慢
- 阻碍技术创新
- 无法按需伸缩

什么是微服务

Martin Fowler：简而言之，微服务架构风格这种开发方法，是以开发一组小型服务的方式来开发一个独立的应用系统。其中每个小型服务都运行在自己的进程中，并经常采用 **HTTP** 资源 **API** 这样轻量的机制来相互通信。这些服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署。这些微服务可以使用不同的语言来编写，并且可以使用不同的数据存储技术。对这些微服务我们仅做最低限度的集中管理（也就是去中心化）。

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务于服务间采用轻量级的通信机制互相沟通（通常是基于 HTTP 的 RESRful API）。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生产环境、类生产环境等。另外，应尽量避免同一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。

微服务架构风格特点

- 服务组件化
- 服务围绕业务

- 产品开发模式
- 轻量级通信机制

  REST API 或者是 RPC
- 去中心化处理
- 去中心化数据设计
- 故障处理设计
- 演进式设计
- 基础设施自动化

微服务的优点与挑战

优点

- 开发简单

  易于开发和维护。
- 技术栈灵活

  技术栈不受限制。
- 服务独立

  启动较快，局部修改容易部署。
- 按需扩容

  按需伸缩。
- DevOps

  开发要参与运维工作。

挑战

- 运维复杂

  对运维要求较高，也需要开发人员在一定程度上参与其中。
- 数据一致性问题
- 分布式的复杂性
- 接口调整成本高
- 集成测试复杂
- 重复代码
- 监控困难

微服务设计原则

- 单一职责原则
- 服务自治原则
- 轻量级通信原则
- 接口明确原则

Overview

序号	标题	文件
10-13	SOA理论与概念	Service-orientedArchitecture.md
13	维基百科微服务	Microservices.md
14	微服务和SOA对比	MicroservicesAndSOA.md

SOA理论与概念

描述

SOA（Service-Oriented Architecture），面向服务的架构，它是通过应用组件的方式向其他组件提供服务的一种软件设计的风格，提供服务的方式是基于网络的通信协议。SOA 的基本原则是独立于供应商、产品和技术。服务是一个独立的功能单元，可以远程访问并独立操作和更新，例如在线检索信用卡声明。

根据 SOA 的许多定义之一，服务具有四个属性：

1. 它逻辑上代表具有指定结果的业务活动。
2. 它是独立的。
3. 它是消费者的黑盒子。
4. 它可能包含其他基础服务

1998年，SOA 首先被称为基于服务的体系结构，由一个团队开发集成的基础管理服务，然后是基于工作单元的业务流程类型服务，并使用 CORBA 进行进程间通信。

可以结合使用不同的服务来提供大型软件应用程序的功能，SOA 与模块化编程共享的原则。 面向服务的体系结构集成了分布式，单独维护和部署的软件组件。 它通过技术和标准实现，这些技术和标准有助于组件通过网络进行通信和协作，尤其是通过 IP 网络。

概述

在 SOA 中，服务使用描述如何使用描述元数据传递和解析消息的协议。 该元数据描述了服务的功能特征和服务质量特征。 面向服务的体系结构旨在允许用户将大块功能组合在一起，形成纯粹由现有服务构建并以临时方式组合的应用程序。 服务为请求者提供了一个简单的接口，它抽象出作为黑盒子的底层复杂性。 其他用户也可以在不了解其内部实现的情况下访问这些独立服务。

定义概念

相关的流行语面向服务促进了服务之间的松散耦合。SOA 将功能分为不同的单元或服务，开发人员可以通过网络访问这些单元或服务，以便允许用户在应用程序的生产中组合和重用它们。 这些服务及其相应的消费者通过以明确定义的共享格式传递数据或通过协调两个或更多服务之间的活动来相互通信。

2009 年 10 月发布了一份面向服务架构的宣言。其中提出了六个核心价值观，如下所示：

1. 商业价值比技术战略更重要。
2. 战略目标比项目特定的利益更重要。
3. 本质互操作性比定制集成更重要。
4. 共享服务比特定用途实现更重要。
5. 灵活性比优化更重要。
6. 演进式的精化比追求初始完美更重要。

SOA 可以被视为连续体的一部分，其范围从分布式计算和模块化编程的旧概念到 SOA，再到当前的 mashup ，SaaS 和云计算实践（有些人认为是 SOA 的后代）。

原则

尽管许多行业来源已经发布了自己的原则，但业界没有与面向服务架构的确切的相关行业标准。 包括以下内容：

- 标准化服务合同 服务遵循标准通信协议，由一组给定服务中的一个或多个服务描述文档共同定义。
- 服务引用自治（松散耦合的一个方面） 服务之间的关系被最小化到他们只知道它们存在的水平，而不知道怎么实现的，底层又是什么，都是不知道的，只知道有这个东西。
- 服务地点透明度（疏松耦合的一个方面） 无论网络位于何处，都可以从网络中的任何位置调用服务。
- 服务长寿 服务应该设计为长期存在的。 在可能的情况下，如果您不需要新功能，服务应该避免强迫消费者进行更改。如果您今天调用服务，您明天应该能够调用相同的服务。
- 服务抽象 服务类似黑盒一样，他们的内在逻辑对消费者是隐藏的。
- 服务自治 服务是独立的，从设计期和运行期的角度控制它们封装的功能。
- 服务无状态 服务本身是无状态的，即返回请求的值或抛出异常，从而最大限度地减少资源使用。

  补充：用户的 session 信息是放在后端 java 端好，还是前端 node 端好？

  推荐放在 **node** 层，通过 **node** 访问 **redis** 集群获取，这样做的好处是 **java** 的后端服务层是没有状态的，这种状态更好的应该是由前端层来处理。
- 服务粒度 确保服务具有足够规模和范围的原则。 服务向用户提供的功能必须是相关的。
- 服务规范化 服务被分解或合并（标准化）以最小化冗余。 在某些情况下，这可能无法完成，这些是需要性能优化，访问和聚合的情况。
- 服务可组合性 服务可用于组成其他服务。
- 服务发现 服务补充了交流元数据，通过它可以有效地发现和解析它们。
- 服务可重用性 逻辑分为各种服务，以促进代码的重用。
- 服务封装 许多最初未在 SOA 下计划的服务可能会被封装或成为 SOA 的一部分。

模式

每个 SOA 构建块都可以扮演以下三种角色中的任何一种：

- 服务提供者

  它创建 Web 服务，并将其信息提供给服务注册。每个提供商都会讨论大量的方法，以及为什么要公开哪些服务，哪些更重要：安全性或易用性，提供服务的价格等等。提供商还必须决定应该为给定的代理服务列出服务的类别以及使用该服务需要哪种贸易伙伴协议。

- 服务代理，服务注册或服务仓库

  其主要功能是使任何潜在请求者都能获得有关Web服务的信息。无论谁来实施这个代理，都要决定代理的范围。公开代理随处可见，但私有代理只能向有限的范围开放。UDDI 是一种早期的，不再主动支持的尝试来提供 Web 服务发现。

- 服务请求者/消费者

它使用各种查找操作在代理注册表中查找条目，然后绑定到服务提供者以调用其中一个 Web 服务。无论服务消费者需要哪种服务，他们都必须将其纳入到服务代理当中，将其与相应的服务绑定，然后使用它。如果服务提供多种服务，他们可以访问多种服务。

服务消费者 - 提供者之间关系由标准化服务合同进行管理，其中包括业务部分，功能部分和技术部分。

服务组合模式有两种广泛的高级架构风格：编排和编排。不受特定架构风格约束的低级企业集成模式在 SOA 设计中仍然具有相关性和合格性。

实现方法

面向服务的体系结构可以通过Web服务实现。这样做是为了使功能构建块可通过独立于平台和编程语言的标准 Internet 协议访问。这些服务既可以代表新应用程序，也可以代表现有遗留系统的包装，使其具备网络功能。

实施者通常使用 Web 服务标准构建 SOA 。一个例子是 SOAP ，它在 2003 年 W3C（万维网联盟）推荐 1.2 版之后获得了广泛的行业认可。这些标准（也称为 Web 服务规范）也提供了更大的互操作性和一些保护。锁定专有供应商软件。但是，也可以使用任何其他基于服务的技术（如 Jini ，CORBA 或 REST ）实现 SOA 。

架构可以独立于特定技术运行，因此可以使用多种技术实现，包括：

- Web services based on WSDL and SOAP
- Messaging, e.g., with ActiveMQ, JMS, RabbitMQ
- RESTful HTTP, with Representational state transfer (REST) constituting its own constraints-based architectural style
- OPC-UA
- WCF (Microsoft's implementation of Web services, forming a part of WCF)
- Apache Thrift
- SORCER

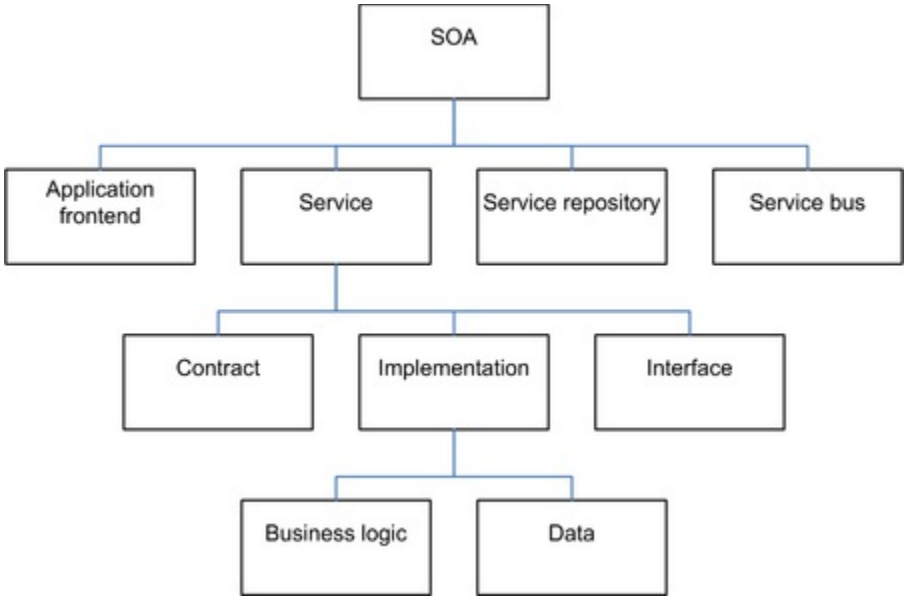
实现可以使用这些协议中的一个或多个，例如，可以使用文件系统机制来遵循符合 SOA 概念的进程之间的定义的接口规范来传送数据。关键是具有已定义接口的独立服务，可以调用这些接口以标准方式执行其任务，而无需服务预先知道调用应用程序，并且应用程序不需要或不需要知道服务如何实际执行其任务。SOA 支持开发通过组合松散耦合和可互操作的服务构建的应用程序。

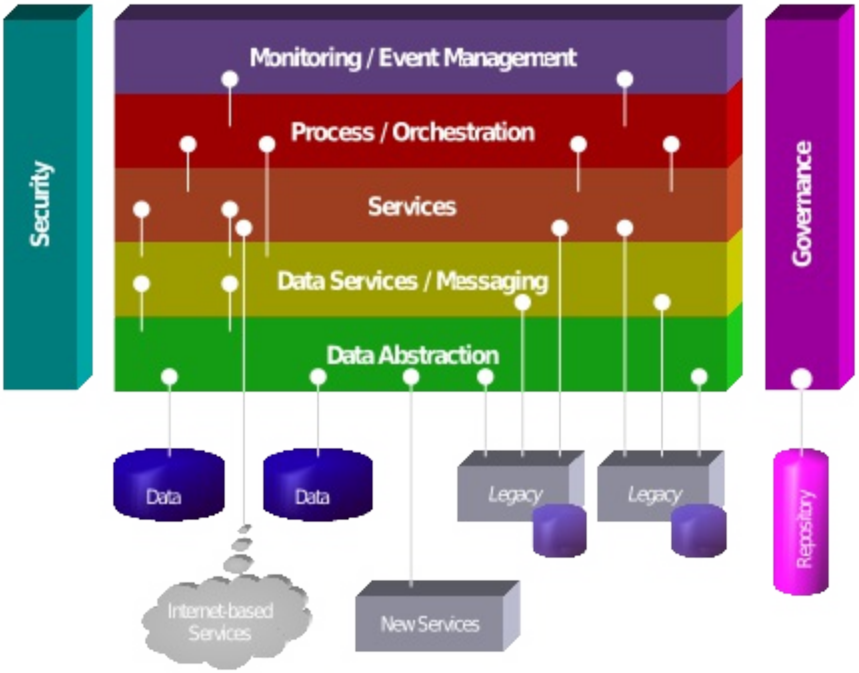
这些服务基于独立于底层平台和编程语言的正式定义（或合同，例如 WSDL ）进行互操作。接口定义隐藏了特定于语言的服务的实现。因此，基于 SOA 的系统可以独立于开发技术和平台（例如 Java ，.NET 等）运行。例如，在运行于 .NET 平台上的 C＃ 和用 Java EE 平台上运行的 Java 编写的服务编写的服务都可以由公共复合应用程序（或客户端）使用。在任一平台上运行的应用程序也可以使用在另一平台上运行的服务作为便于重用的 Web 服务。托管环境还可以包装 COBOL 遗留系统并将其作为软件服务提供。

诸如 BPEL 之类的高级编程语言和诸如 WS-CDL 和 WS-Coordination 之类的规范通过提供一种定义和支持将细粒度服务编排成更粗粒度的业务服务的方法来扩展服务概念，架构师可以反过来并入在复合应用程序或门户中实现的工作流和业务流程。

面向服务的建模是一个 SOA 框架，它可以识别指导 SOA 从业者对其面向服务的资产进行概念化，分析，设计和构建的各种规程。面向服务的建模框架（SOMF）提供了一种建模语言和一个工作结构或“映射”，描述了有助于成功的面向服务的建模方法的各种组件。它说明了识别服务开发方案的“做什么”方面的主要元素。该模型使从业者能够制定项目计划并确定面向服务的计划的里程碑。SOMF 还提供了一种通用的建模符号，以解决业务和 IT 组织之间的一致性问题。

图解组成部分





组织利益

一些企业架构师认为，SOA 可以帮助企业更快，更经济地响应不断变化的市场条件。这种架构提倡宏观（服务）级别的重用，而不是微观（类）级别的重用。它还可以简化现有 IT（传统）资产的互连和使用。

使用 SOA，我们的想法是组织可以从整体上看待问题。企业拥有更多的整体控制权。从理论上讲，不会有大量的开发人员使用任何工具集可能会让他们满意。但他们将编码为业务中设定的标准。他们还可以开发企业级 SOA，封装面向业务的基础架构。SOA 也被描述为为汽车驾驶员提供效率的高速公路系统。关键在于，如果每个人都有车，但在任何地方都没有高速公路，那么事情就会受到限制和混乱，无论是试图快速或有效地到达任何地方。IBM Web 服务副总裁 Michael Liebow 表示，SOA“建设高速公路”。

在某些方面，SOA 可以被视为架构演变而不是革命。它捕获了以前软件架构的许多最佳实践。例如，在通信系统中，很少开发使用真正静态绑定与网络中的其他设备通信的解决方案。通过采用 SOA 方法，此类系统可以将自己定位为强调定义明确，高度可互操作的接口的重要性。SOA 的其他前身包括基于组件的软件工程和远程对象的面向对象分析和设计（OOAD），例如，在 CORBA 中。

服务包括仅通过正式定义的界面可用的独立功能单元。服务可以是某种易于生产和改进的 “纳米企业” 。服务也可以是作为下属服务的协调工作而构建的 “大型企业” 。 SOA 的成熟部署有效地定义了组织的 API。

将服务实施视为大型项目的单独项目的原因包括：

1. 分离将业务概念推广到业务，即服务可以快速独立地从组织中常见的较大且移动较慢的项目中提供。 业务开始了解呼叫服务的系统和简化的用户界面。 这提倡敏捷。 也就是说，它促进了业务创新并加快了产品上市时间。
2. 分离促进了服务与消费项目的脱钩。 这样可以鼓励良好的设计，因为服务的设计不需要知道消费者是谁。
3. 服务的文档和测试构建较大项目的详细信息中。 当服务需要稍后重用时，这很重要。

SOA 承诺间接简化测试。服务是自治的，无状态的，具有完全记录的接口，并且与实现的横切关注点分开。如果组织拥有适当定义的测试数据，则会构建相应的存根，以便在构建服务时对测试数据做出反应。还会为服务捕获一整套回归测试，脚本，数据和响应。该服务可以使用与其调用的服务相对应的现有存根作为 “黑盒子” 进行测试。可以构建测试环境，其中原始和超出范围的服务是存根，而网格的其余部分是完整服务的测试部署。由于每个接口都有完整的文档，并附有完整的回归测试文档，因此可以轻松识别测试服务中的问题。测试演变为仅仅根据其文档验证测试服务是否运行，并且发现环境中所有服务的文档和测试用例存在差距。管理幂等服务的数据状态是唯一的复杂性。

实例可能有助于将服务记录到有用的级别。Java Community Process 中的一些 API 文档提供了很好的示例。由于这些是详尽无遗的，工作人员通常只使用重要的子集。JSR-89 中的 'ossjsa.pdf' 文件举例说明了这样一个文件。

批评

SOA 已与 Web Services 混淆; 但是，Web Services 只是实现构成 SOA 风格的模式的一种选择。 在没有本地的或二进制形式的远程过程调用（RPC）的情况下，应用程序可能运行得更慢并且需要更多处理能力，从而增加了成本。大多数实现都会产生这些开销，但 SOA 可以使用不依赖于远程过程调用或通过转换的技术（例如，Java Business Integration（JBI），Windows Communication Foundation（WCF）和数据分发服务（DDS））来实现 XML。 与此同时，新兴的开源 XML 解析技术（如VTD-XML）和各种 XML 兼容的二进制格式有望显着提高 SOA 性能。 使用 JSON 而不是 XML 实现的服务不会受到此性能问题的影响。

有状态服务要求消费者和提供者共享相同的特定于消费者的上下文，该上下文包含在提供者和消费者之间交换的消息中或由其引用。 如果服务提供者需要为每个消费者保留共享上下文，则该约束的缺点在于它可能降低服务提供者的整体可伸缩性。 它还增加了服务提供商和消费者之间的耦合，使交换服务提供商更加困难。最终，一些评论家认为 SOA 服务仍然受到它们所代表的应用程序的限制。

面向服务的体系结构面临的主要挑战是管理元数据。 基于 SOA 的环境包括许多彼此之间进行通信以执行任务的服务。由于设计可能涉及多个服务一起工作，因此应用程序可能会生成数百万条消息。 进一步的服务可能属于不同的组织甚至是竞争公司，造成巨大的信任问题。 因此 SOA 治理进入了事物的计划。

SOA 面临的另一个主要问题是缺乏统一的测试框架。 没有工具可以提供在面向服务的体系结构中测试这些服务所需的功能。 困难的主要原因是：

1. 异构和解决方案的复杂性。
2. 由于自主服务的集成，大量的测试组合。
3. 包含来自不同和竞争供应商的服务。
4. 由于新功能和服务的可用性，平台不断变化。

微服务

微服务是对用于构建分布式软件系统的面向服务的体系结构（**SOA**）的现代解释。 微服务架构中的服务是通过网络彼此通信以实现目标的进程。 这些服务使用技术不可知的协议，这有助于封装语言和框架的选择，使他们的选择成为服务内部的一个问题。 微服务是 **SOA** 的一种新的实现和实现方法，自 2014 年（以及 DevOps 引入之后）开始流行，并且还强调持续部署和其他敏捷实践。

微服务没有一个达成一致的定义。 以下特征和原理可在文献中找到：

- 细粒度接口（可独立部署的服务）
- 业务驱动的开发（例如域驱动设计）
- IDEAL 云应用程序架构
- 多语言编程和持久性
- 轻量级容器部署
- 分散的持续交付
- DevOps 提供全面的服务监控

参考资料

维基百科

维基百科微服务

微服务是一种软件开发技术 - 面向服务架构（SOA）架构风格的变种，它将应用程序分解为松散耦合服务的集合。在微服务架构中，服务是细粒度的，协议是轻量级的。将应用程序分解为不同的较小服务的好处是它可以提高模块性。 这使得应用程序更易于理解，开发，测试，并且对架构变得更具弹性，可伸缩。 它通过使小型自治团队能够独立开发，部署和扩展各自的服务来实现开发的并行化。 它还允许通过连续重构来显示单个服务的体系结构。基于微服务的架构支持持续交付和部署。

特性

- Per Martin Fowler和其他专家认为，微服务架构（MSA）中的服务通常是通过网络进行通信以使用与HTTP等技术无关的协议实现目标的过程。
- 微服务架构中的服务可独立部署。
- 服务以细粒度的业务能力为基础。微服务的粒度很重要 - 因为这是这种方法与**SOA**不同的关键。微服务强调的是细粒度，而**SOA**则强调的是粗粒度。
- 服务可以使用不同的编程语言，数据库，硬件和软件环境来实现，具体取决于最适合的。这并不意味着单个微服务是用混杂的编程语言编写的。虽然几乎可以肯定的是，服务组成的不同组件将需要不同的语言或API（例如，Web服务器层可能使用Java或Javascript，但数据库可能使用SQL与RDBMS进行通信），这实际上反映了与整体式建筑风格的比较。如果将整体应用程序重新实现为一组微服务，那么各个服务可以选择自己的实现语言。因此，一个微服务可以为Web层选择Java，而另一个微服务可以选择基于Node.js的实现，但在每个微服务组件中，实现语言将是统一的。
- 服务规模小，启用消息，受上下文限制，自主开发，可独立部署，分散，并通过自动化流程构建和发布。

质疑

微服务方法受到许多问题的批评：

- 服务形成信息障碍。
- 与整个服务流程中的进程内调用相比，网络上的服务间调用在网络延迟和消息处理时间方面的成本更高。
- 测试和部署更加复杂。
- 在服务之间转移责任更加困难。它可能涉及不同团队之间的沟通，用另一种语言重写功能或将其融入不同的基础设施。
- 当内部模块化的替代方案可能导致更简单的设计时，将服务的大小视为主要的结构化机制可能会导致太多的服务。
- 在基于微服务的体系结构中，两阶段提交被视为一种反模式，因为这会导致事务中所有参与者的耦合更加紧密。然而，缺乏这种技术会导致尴尬的情况，为了保持数据的一致性，所有事务参与者都必须实现这些要求。
- 如果使用不同的工具和技术构建，许多服务的开发和支持将更具挑战性 - 如果工程师经常在项目之间移动，这尤其是一个问题。

参考资料

维基百科

微服务与**SOA**对比

IBM

如果您在IT领域工作，您可能已经听过SOA与微服务的争论。毕竟，现在每个人都在谈论微服务和敏捷应用程序。

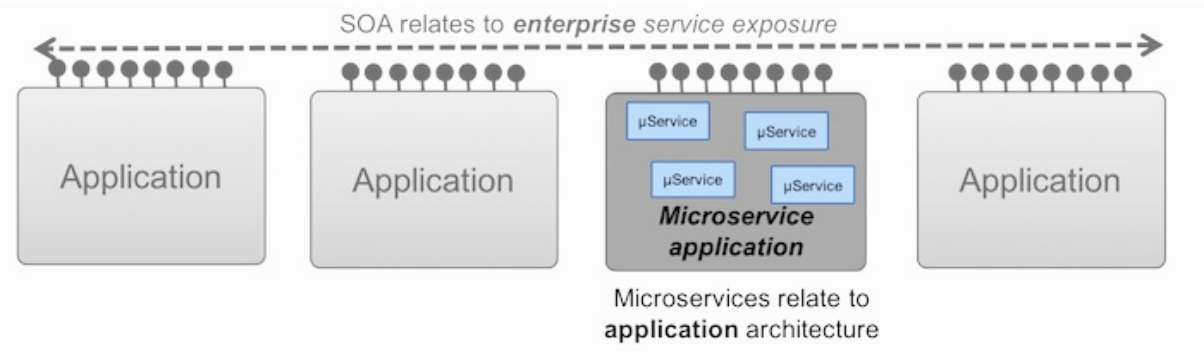
乍一看，这两种方法听起来非常相似。在某些方面，他们是。两者都不同于传统的单片架构，因为每项服务都有自己的责任。两者都受益于一定程度的解耦。

主要区别归结为范围。简而言之，面向服务的体系结构（SOA）具有企业范围，而微服务体系结构具有应用范围。

注：**SOA**范围大，着眼于整个企业，而微服务范围小，着眼于应用。

以下是每种情况的一些非常基本的定义：

- SOA是一项企业级范围的计划，旨在创建可重用，同步可用的服务和API。 这有助于开发人员更快速地创建应用程序，并更轻松地合并来自其他系统的数据。
- 微服务架构是一种用于构建单个应用程序的选项，使该应用程序更加灵活，可扩展且具有弹性。



SOA是与企业服务的公开性密切相关的，关注的范围更大，是应用与应用之间的通信、服务公开。

微服务是与应用架构紧密相关的，关注的范围小，只关注应用本身的范围。

重要性

当你忽视这种差异时，每种方法的许多核心原则都会变得不兼容。 如果您接受他们在范围的差异，您可能很快意识到这两者可能相互补充而不是竞争关系。

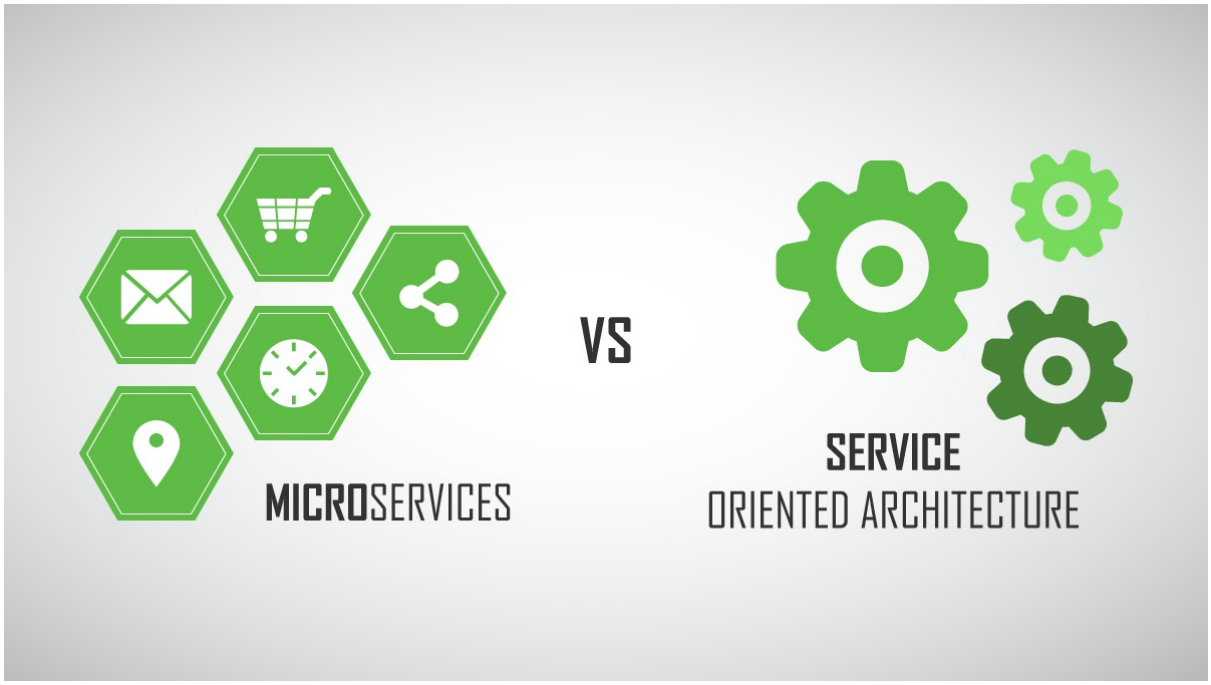
以下是这种区别发挥作用的几种情况：

- 重用性。在SOA中，集成的重用性是主要目标，在企业级别，努力实现某种程度的重用性是至关重要的。在微服务架构中，创建在整个应用程序中在运行时重用的微服务组件会导致依赖性降低敏捷性和弹性。微服务组件通常更喜欢通过复制重用代码并接受数据复制以帮助改善解耦。
- 同步调用。 SOA中的可重用服务可在整个企业中使用，主要使用RESTful API等同步协议。但是，在微服务应用程序中，同步调用会引入实时依赖性，从而导致弹性丧失。它还可能导致延迟，从而影响性能。在微服务应用程序中，基于异步通信的交互模式是优选的，例如事件源，其中使用发布订阅模型使微服务组件能够保持最新发生在另一组件中的数据发生的变化。
- 数据重复。在SOA中提供服务的明确目标是让所有应用程序直接在其主要源上同步获取和更改数据，从而减少维护复杂数据同步模式的需要。在微服务应用程序中，每个微服务理想地具有对其所需的所有数据的本地访问，以确保其与其他微服务以及实际上来自其他应用程序的独立性，即使这意味着在其他系统中存在一些数据重复。当然，这种重复增加了复杂性，因此必须与敏捷性和性能的提升相平衡，但这被认为是微服务设计的现

实。

DZone

最近，关于这两种架构之间的差异，或者是否存在任何差异，已经有很多大惊小怪。 为了深入研究引发数百次辩论的这个问题，我将首先简要地定义SOA和微服务架构及其起源，然后我们将对它们进行比较，看看我们如何才能最好地区分它们。

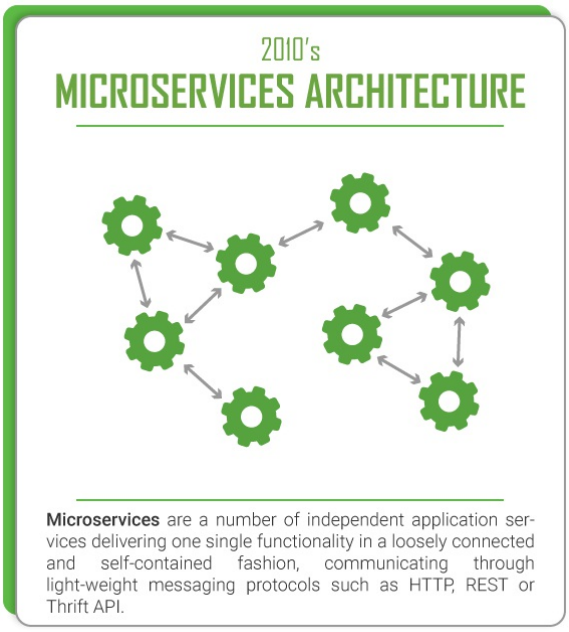


Service-Oriented Architecture (SOA)

面向服务的体系结构（SOA）是一种软件体系结构，其中应用程序的不同组件通过网络上的通信协议向其他组件提供服务。通信可以涉及简单的数据传递，或者涉及彼此协调连接服务的两个或更多个服务。这些不同的服务执行一些小功能，例如验证付款，创建用户帐户或提供社交登录。

面向服务的体系结构（**SOA**）更少的去关注如何模块化应用程序，而更多的是去关注如何通过集成分布式，单独维护和部署的软件组件来组合应用程序。它通过技术和标准实现，使组件更容易通过网络进行通信和协作，尤其是IP网络。

SOA中有两个主要角色：服务提供者和服务使用者。软件代理可以扮演两种角色。消费者层是用户（人，应用程序的其他组件或第三方）与SOA交互的点，提供者层由SOA中的所有服务组成。



SOA首先在90年代中期得名，当时一家名为Gartner Group的公司认识到软件架构中的这一新兴趋势，采用它并在全球范围内推广它。 通过这样做，他们成功地大大加快了这种架构模式的采用和进一步发展。 但是，使用分布式服务作为软件架构的第一个记录可以追溯到80年代早期。

Microservices

从某种程度上讲，微服务是面向服务架构（SOA）演化的下一个阶段。基本上说，这种架构类型是开发软件，Web或移动应用程序作为独立服务套件的一种特殊方式 - 又叫做微服务。创建这些服务仅用于一个特定的业务功能，例如用户管理，用户角色，电子商务购物车，搜索引擎，社交媒体登录等。此外，它们彼此之间完全相互独立，这意味着它们可以写入不同的编程语言并使用不同的数据库集中式服务管理几乎不存在，微服务使用轻量级HTTP，REST或Thrift API进行相互通信。

这个词本身源于2011年5月在威尼斯附近举办的软件架构师研讨会。他们首次使用“微服务”这一术语来描述参与者所看到的许多人最近一直在探索的共同建筑风格。 2012年5月，同一小组决定将“微服务”作为最合适的名称。然而，微软，亚马逊，Netflix和Facebook等主要科技公司已经使用微服务十多年了。在提出普遍接受的名称之前，其他人称他们为“微网络服务”或“细粒度SOA”。

是的，乍一看，我们似乎在谈论与SOA相同的事情。但是，我将请求微软服务领域的先驱Martin Flower的话，他曾经说过我们应该将SOA视为微服务的超集。

So, Where’s the Difference?

我们可以说两种架构都有相似之处而不是差异，但是，最终，它们是两种不同类型的架构。 为了支持我的主张，首先，我将以表格的形式介绍将它们区分开来的架构的特定部分。 稍后，我将更详细地阐述其中的一些内容。 开始了！

SOA架构	微服务架构
最大化应用服务的可重用性	关注于解耦
系统的变化需要修改整体结构	系统的变化是创造一种新的服务
DevOps和持续交付正在变得流行，但不是主流	专注于DevOps和持续交付

专注于业务功能的可重用性	更加重视“边界上下文”的概念
对于通信，它使用企业服务总线（ESB）	对于通信使用不那么复杂和简单的消息系统
支持多种消息协议	使用轻量级协议，如HTTP，REST或Thrift API
为部署到它的所有服务使用通用平台	应用服务器并未真正使用，通常使用云平台
使用容器（如Docker）不太受欢迎	容器与微服务一起工作得很好
SOA服务共享数据存储	每个微服务可以具有独立的数据存储
共同治理和标准	轻松治理，更加注重团队协作和选择自由度

我将在上表中显示的某些方面进一步详细说明，并进一步解释其中的差异：

- 开发 - 在这两种体系结构中，可以使用不同的编程语言和工具开发服务，从而为开发团队带来技术多样性。可以在多个团队中组织开发，但是，在SOA中，每个团队都需要了解常见的通信机制。另一方面，通过微服务，服务可以独立于其他服务运行和部署。因此，更容易经常部署新版本的微服务或独立扩展服务。您可以在此处进一步了解微服务的这些优点。
- “绑定上下文” - SOA鼓励共享组件，而微服务试图通过“绑定上下文”最小化共享。绑定上下文指的是将组件及其数据作为单个单元耦合，具有最小的依赖性。由于SOA依赖于多种服务来满足业务请求，因此基于SOA构建的系统可能比微服务慢。
- 通信 - 在SOA中，ESB可能成为影响整个系统的单点故障。由于每个服务都通过ESB进行通信，如果其中一个服务速度变慢，它可能会阻塞ESB请求该服务。另一方面，微服务在容错方面要好得多。例如，如果一个微服务有内存故障，那么只有那个微服务会受到影响。所有其他微服务将继续定期处理请求。
- 互操作性 - SOA通过其消息传递中间件组件促进多个异构协议的使用。微服务试图通过减少集成选择的数量来简化架构模式。因此，如果要在异构环境中使用不同协议集成多个系统，则需要考虑**SOA**。如果可以通过相同的远程访问协议访问所有服务，那么微服务对您来说是更好的选择。
- 大小 - 最后但并非最不重要的是，**SOA**和微服务之间的主要区别在于大小和范围。微服务中的前缀**“微”**指的是内部组件的粒度，这意味着它们必须比**SOA**趋向于小得多。微服务中的服务组件通常只有一个目的，他们做得很好。另一方面，在**SOA**中，服务通常包含更多的业务功能，并且它们通常作为完整的子系统实现。

结论

人们不能简单地说一个架构比另一个架构好。 它主要取决于您正在构建的应用程序的目的。 **SOA**更适合需要与许多其他应用程序集成的大型复杂企业应用程序环境。 话虽这么说，较小的应用程序不适合**SOA**，因为它们不需要消息传递中间件组件。 另一方面，微服务更适合于较小且分区良好的基于**Web**的系统。 此外，如果您正在开发移动或**Web**应用程序，那么微服务可以让您作为开发人员获得更大的控制权。 最后，我们可以得出结论，因为它们用于不同的目的 - 微服务和**SOA**确实是不同类型的架构。

参考资料

IBM

DZone

Overview

序号	标题	文件
15	Spring Boot 应用起步与配置	SpringInitializr.md
16	Spring Boot 应用配置分析	Configure.md
17	Spring Boot 打包文件内容与结构	JarDirectoryStructure.md
18	使用 Gradle 构建 Spring Boot 应用	GradlePlugin.md
19	Spring Boot 参数自动装配	Autowired.md
20	Jar 文件规范	JarFileSpecification.md
19-21	Spring Boot Loader 源码分析	Loader.md
20	反射扩展	Reflect.md
22	JDWP 远程调试	JDWP.md
23	使用 JDWP 调试 Spring Boot Loader 源码	JDWPLoader.md

Spring Boot 应用起步与配置

快速起步

Spring Initializr

通过勾选的方式，快速搭建一个项目的骨架，eclipse 和 idea 都有相关插件。

Starters

起步依赖，Spring Boot 引入的理念，通过它可以很方便的把 Spring Boot 以及需要的第三方插件集成到应用当中。

Version

因为有 starter 的存在，其实 Spring Boot 的版本号在 dependency 里是省略掉的，由 starter 统一管理，Spring Cloud 也是如此。

启动方式

Spring Boot 应用的启动方式一共有 4 种。

1. 在 @SpringBootApplication 所注释的类里直接调用 run 启动。
2. 通过 Spring Boot 启动指令

```
gradle bootRun
或者
gradlew bootRun
```

3. 双击 Idea 里 Gradle 任务栏里面的 bootRun 任务。
4. 还可以找到这个应用打好的 jar 包，通过 Java 运行 jar 包的方式运行。

```
java -jar microservices-0.0.1-SNAPSHOT.jar
```

前三种是开发阶段常用的启动方式（建议使用第二、三种方式启动，第一种在工程有问题的时候，有时候也可以正常运行成功，例如：没有成功下载所有的 **jar** 包），最后一种是部署服务器时常用的启动方式。

Spring Boot 启动是通过一个 main 方法，以 jar 包方式运行。web 服务器是以嵌入的方式存在于应用当中，所有的配置、三方依赖都在 jar 包内。其实 Tomcat 就是内嵌在 Spring Boot 应用内的（嵌入式服务器），这也是为什么 Tomcat 的一些配置可以通过 Spring Boot 的配置文件进行修改。

理念

约定优于配置

题外话

尝试下 Visual Studio Code，对于前端和后端的开发都有比较好的支持。

Spring Boot 应用配置分析

UTC

一种时间格式，国际化的项目，最好采用 utc 格式，因为 utc 的格式是没有时区的问题，在日期传递过程中效果是最好的。

Banner

控制台会打印 Spring Boot Logo 以及版本信息，这个操作可以通过配置实现自定义。

例如

进程运行

从控制台的输出信息可以知道 Spring Boot 都是运行在进程内的，如何查看？

指令：`lsof -i:8080` 或 `ps -ef | grep java`

Profile

一般模式：一套配置文件，基于开发、测试、线上等多种环境时，构建多个整套配置文件，在项目启动的时候，通过一个参数来实现配置的切换。目的解决代码和配置分离，但不是最好的方式。

比较好的方式：代码和配置完全隔离。项目工程当中是看不到任何配置信息的，配置信息是集中的放在 Spring Cloud Config 中，可以对其加密。

更好的方式：配置中心。比如：携程的 Apollo。

Gradle

推荐学习。

启动类

每个 Spring Boot 应用都有一个启动类。

启动类需要以下几个条件：

- 由 @SpringBootApplication 注解修饰
- 启动类当中有一个 main 方法，也只有一句启动代码

```
SpringApplication.run(xxx.class, args);
```

配置文件

Spring Boot 提供了两种类型的配置文件形式，但是都是遵循约定优于配置，都叫做 application 这个名字，虽然可以，尽量不要修改这个名字。

properties 文件格式

传统的经典文件格式。

application.properties

```
server.port=8080
server.address=127.0.0.1
server.context-path=/
```

yml 文件格式

YAML（Yet Another Markup Language），一种新的文件格式。

application.yml

```
server:
    port: 8080
    address: 127.0.0.1
    context-path: /
```

感觉上 yml 的方式要更简洁一点，可以少写前缀 server 很多次。

注意事项：

- 当 yml 配置的最后一层，也就是实际值和前面对应的 key 后面的冒号之间，有一个空格。
- 缩进不允许使用tab只能使用空格。

例如：port: 8080，冒号和 8080 之间必须有空格，可以观察 idea 或者 md 的引用代码区域是否正常高亮显示来判断书写正确与否。

Spring Boot 打包文件内容与结构

Gradle 任务

指令：gradle tasks

会显示出当前工程所有可以使用的 gradle 的 task，并且给每一个 task 一个说明。

```
> Task :tasks

------------------------------------------------------------
Tasks runnable from root project
------------------------------------------------------------

Application tasks
-----------------
bootRun - Runs this project as a Spring Boot application.

Build tasks
-----------
assemble - Assembles the outputs of this project.
bootJar - Assembles an executable jar archive containing the main classes and their dependencies.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.

Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.

Documentation tasks
-------------------
javadoc - Generates Javadoc API documentation for the main source code.

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root project 'microservices'.
components - Displays the components produced by root project 'microservices'. [incubating]
dependencies - Displays all dependencies declared in root project 'microservices'.
dependencyInsight - Displays the insight into a specific dependency in root project 'microservices'.
dependencyManagement - Displays the dependency management declared in root project 'microservices'.
dependentComponents - Displays the dependent components of components in root project 'microservices'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'microservices'. [incubating]
projects - Displays the sub-projects of root project 'microservices'.
properties - Displays the properties of root project 'microservices'.
tasks - Displays the tasks runnable from root project 'microservices'.

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.
```

有些任务是 gradle 自带的，有些则不是。

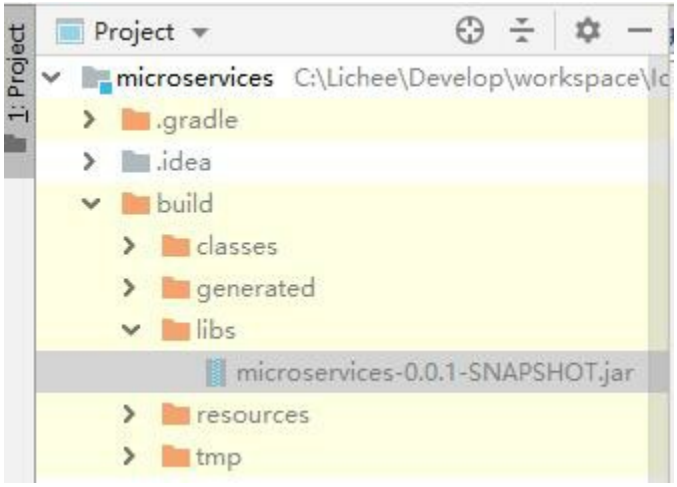
bootJar 任务

gradle 提供的一个任务，作用是将 Spring Boot 打包成一个独立的 jar 包。不依赖于任何其他的依赖、容器等，是一个自包含的独立 jar 包。这个 jar 包内包含运行这个工程的一切，也就是说它是一个可执行的 jar 包。

老版本的 gradle 插件命令 ：packageJar。

注意：基于 **Spring Boot** 的应用，打包方式都是使用 **bootJar** 命令，而不是 **jar** 命令。**jar** 命令只是把工程打成一个 **jar** 包，但 **bootJar** 命令才是打包成一个可执行 **jar** 文件。这两种打包方式，打包出来的文件结构的差距也是非常大的。

打包生成目录：/build/libs/microservices-0.0.1-SNAPSHOT.jar。



运行 **bootJar** 打包的文件

指令：java -jar microservices-0.0.1-SNAPSHOT.jar，和普通的运行 jar 的方式一样。

注意：这种方式其实是 **Spring Boot** 应用部署的标准方式，先把工程通过 **bootJar** 命令打包成一个可运行的 **jar** 文件。然后通过 **java -jar** 命令来执行这个 **jar** 文件。

当然 Spring Boot 应用也支持把工程打包成 war 文件，不过要把 内嵌 tomcat 相关的 jar 包都删除掉，再部署到外部的 tomcat 容器中。

bootJar 文件分析

通过 bootJar 打包后会生成一个 jar 文件，而 这个 jar 文件本身就是一个压缩文件。

解压缩 **jar**

java 解压缩 **jar**

通过 java 自带的 jar 指令解压： jar -xvf microservices-0.0.1-SNAPSHOT.jar

这种方式不好的原因在于，会把解压后的所有文件都直接放在当前目录，而不能把解压后的所有文件放在一个指定的目录内。

unzip 解压缩 **jar**

通过 mac 自带的 unzip 指令解压：unzip microservices-0.0.1-SNAPSHOT.jar -d ./microservices

解压缩后的目录结构

BOOT-INF

传统应用开发中没有的目录结构，Spring Boot 特有的目录，里面有 2 个子目录。分别是：

classes

当前工程编译好的结果文件（也就是 class 字节码文件和相关配置文件）都在这里，包含 src/main/java 和 src/main/resources 下的所有文件。

lib

当前工程依赖的所有 jar 文件（第三方 jar 包）都在这里，居然有 36 个 jar 文件，而我们在 build.gradle 里只引入了一个 starter 的依赖。

META-INF

只有一个文件：MANIFEST.MF。

通过 jar 命令把一个或者若干个 class 文件打包成 jar 包时通常要指定一个文件作为清单文件。当中描述的是可执行 jar 包的基本信息。

```
Manifest-Version: 1.0
Start-Class: com.lichee.microservices.MicroservicesApplication
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Version: 2.1.4.RELEASE
Main-Class: org.springframework.boot.loader.JarLauncher

```

Manifest-Version

清单文件版本。

Start-Class

Spring Boot 特有的属性（com.lichee.microservices.MicroservicesApplication），被 @SpringBootApplication 所修饰且包含 main 方法的启动类。

按照一般的经验来猜想，这个类才应该是 Main-Class 才对，但实际并不是。

Spring-Boot-Classes

就是 BOOT-INFO/classes 下的文件。

Spring-Boot-Lib

BOOT-INF/lib 下的文件。

Spring-Boot-Version

Spring Boot 的版本号。

Main-Class

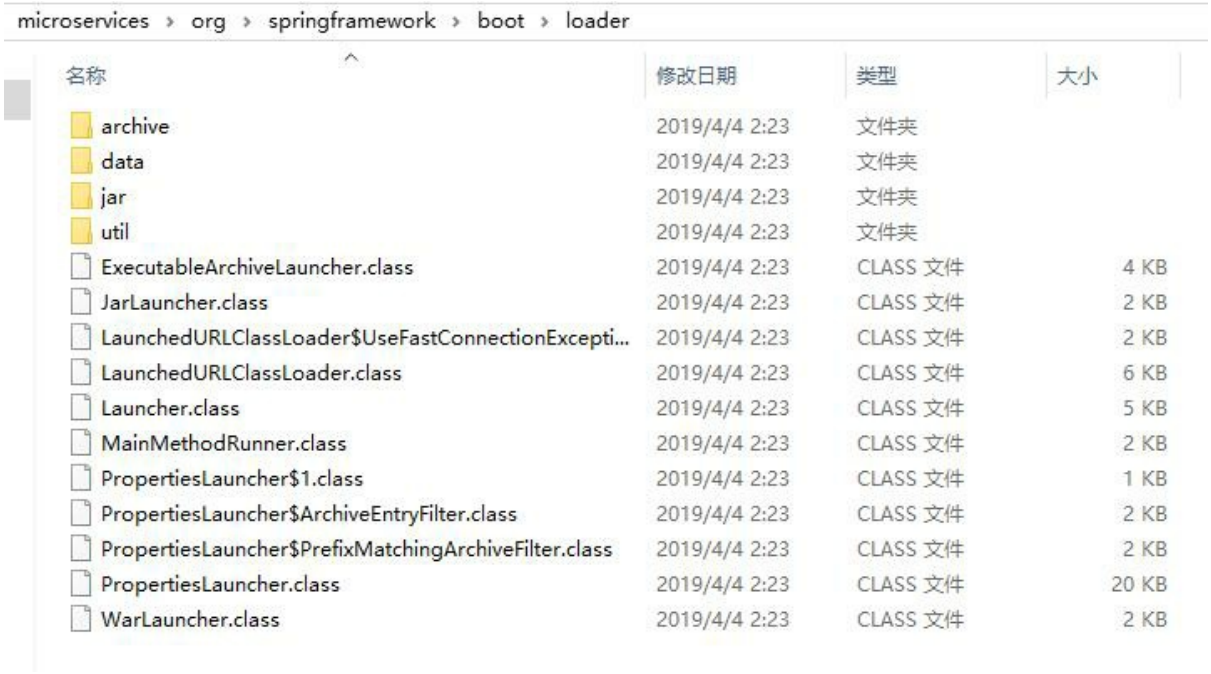
一个可执行 jar 的入口文件，也就是 main 方法所在的类，jar 包文件执行的入口类。居然是 Spring Boot 提供的类（org.springframework.boot.loader.JarLauncher），而不是我们自己写的。只要我们打包方式是 jar 包方式，那么不管什么应用的 Main-Class 永远都是这个 JarLauncher。

JarLauncher：jar 文件的装载器。

注意：有一个细节，在这个清单文件的最后，也就是在 **Main-Class** 文件之后，一定要有一个回车换行，**Spring Boot** 采用的是两个回车换行，可以观察上图。

org

由 Spring Boot 提供的一堆字节码文件，入口类 org.springframework.boot.loader.JarLauncher 就在这里。



使用 **Gradle** 构建 **Spring Boot** 应用

关键插件配置

Spring Boot Gradle Plugin

```
plugins {
    id 'org.springframework.boot' version '2.1.4.RELEASE'
}
```

Spring Boot Gradle Plugin 为 Spring Boot 提供了对 Gradle 的支持，允许你将打包可执行的 jar 或者 war archives。

有了这个插件配置以后，就会有 bootRun、bootJar、bootWar 等额外的任务。

注意：Spring Boot Gradle 插件需要 Gradle 4.4 或更高版本

io.spring.dependency-management Plugin

```
apply plugin: 'java'
apply plugin: 'io.spring.dependency-management'
```

当使用了 io.spring.dependency-management 依赖管理插件，Spring Boot 的插件会自动地从你使用的 Spring Boot 版本里导入 spring-boot-dependencies bom。允许你在声明依赖的时候忽略掉版本号，使用这项功能，只需要正常的声明依赖，不用写版本号就可以了。

例如：

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-web'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

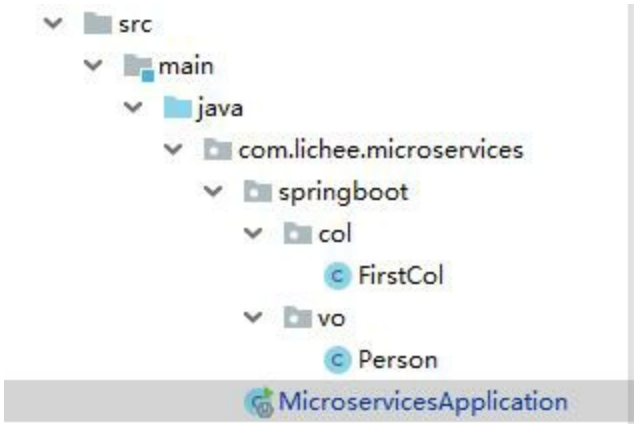
主类

Spring Boot 应用都有 1 个被 @SpringBootApplication 注释所标识的 java 类，这个类一般被称为 Spring Boot 应用的主类。

最佳实践

Spring Boot 应用主类所放置的位置，一般是在 package 的最顶层。这是因为 @SpringBootApplication 注释的包扫描机制所决定的，扫描机制：扫描当前 package，以及当前 package 下的所有子 package ，来进行自动装配和类型加载。而且这个主类所在的 package 里最好不要放置其他任何文件，只放置这个主类。

如下图：


src
main
java
com.lichee.microservices
springboot
col
FirstCol
vo
Person
MicroservicesApplication

Spring Boot 参数自动装配

Spring Boot 自带配置项

Spring Boot 默认自带了很多配置项，可以根据自己的需要对 application.properties 或者 application.yml 进行修改，这里对几种常见的做一个演示。

默认配置项可以参考：Spring Boot Docs

```
spring:
  application:
    name: lichee
  mandatory-file-encoding: UTF-8
  http:
    encoding:
      enabled: true
      charset: UTF-8
```

- name

  应用的名字。

- mandatory-file-encoding

  应用程序必须使用预期的字符编码。

- enabled

  启动 http 的编码支持。

- charset

  HTTP请求和响应的字符集。

自定义配置项

配置文件方式

Spring Boot 不但可以修改它自带的属性，也可以自定义需要的属性，只要 key 不重复就好。也是根据自己的需要对 application.properties 或者 application.yml 进行属性添加。

```
config:
  lichee:
    name: 里奇
    age: 30
```

使用方式和以前传统的 Spring 模式一样，通过 **@Value** 注释来获取。

```
@Value("${config.lichee.name}")
private String name;

@Value("${config.lichee.age}")
private int age;
```

配置类方式

Spring Boot 的配置类必须被 @Configuration 注释所标记，Spring 通过 package 扫描的机制就可以识别出这个类是配置类，就会通过配置的信息进行属性、对象等注入。

```
@Configuration
public class Config {

    @Bean
    public Lichee initLichee() {

        return new Lichee();
    }
}
```

这个 initLichee 方法上面的 @Bean 注释是关键，只有标记了这个注释后，才能把 Lichee 这个实例交给 Spring 管理的 Bean 对象。也就是 initLichee 这个方法返回的对象是 Spring 管理的，Spring 才会进行属性、对象注入。

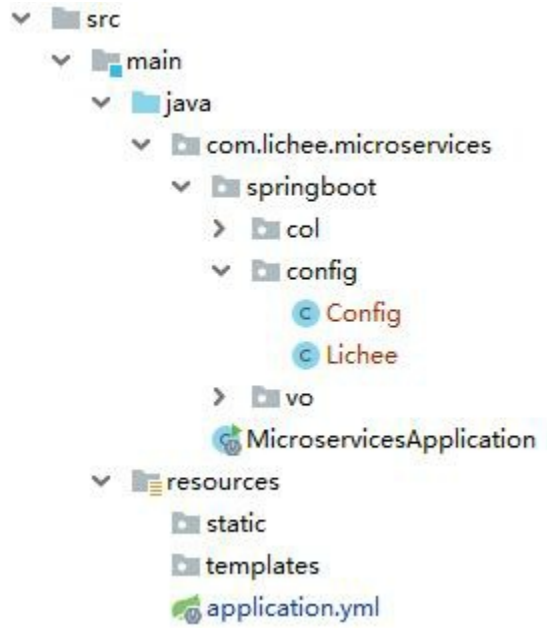
```
public class Lichee {

    @Value("${config.lichee.name}")
    private String name;

    @Value("${config.lichee.age}")
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

上面的操作完成以后，就可以在需要的地方进行调用了。

```
    @Autowired
    private Lichee lichee;
```

配置类方式最佳实践

在项目中，专门分配一个名为 config 的 package，把项目相关的 Java Config 都放在这个 package 里。



几年前 Java Config 这种做法是不对的，而现在提倡这样。

Jar 文件规范

为了更好的理解后续内容，先了解下 Jar 文件规范。

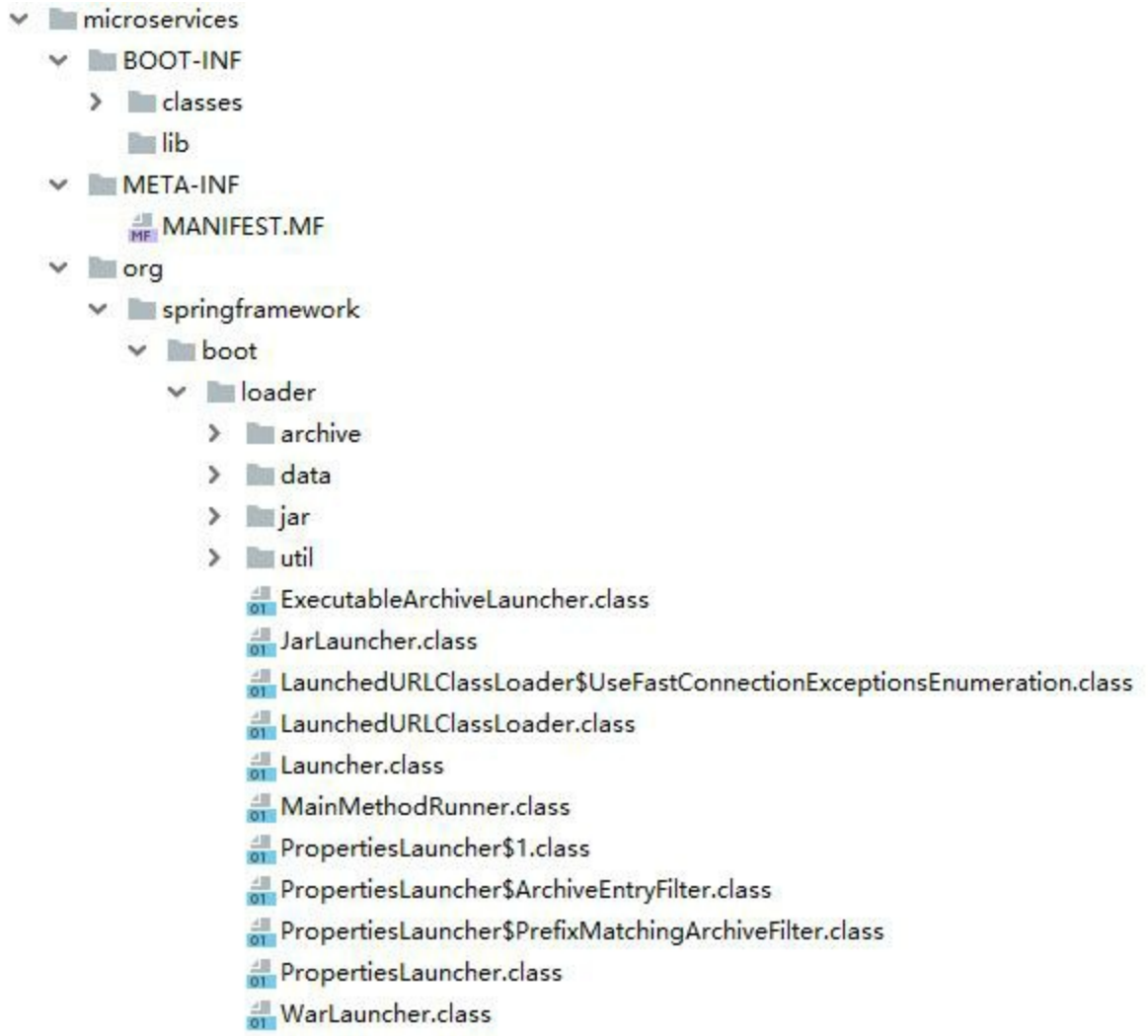
Main-Class 位置

对于一个标准的 java jar 归档文件，规定一旦指定了 Main-Class 类信息，那么这个类连同它的包结构就必须放在这个 jar 归档文件的最顶层目录下，相当于整个应用工程的代码都放在了 jar 文件的最顶层目录下。这个类不允许嵌套，如果出现了嵌套（jar 文件内嵌套了 jar文件内的 Main-Class 类），会导致 Main-Class 类不能执行，这个类只有在 jar 归档文件的最顶层目录下才能被执行。

比如：Spring Boot 打包后的 jar。

```
1  Manifest-Version: 1.0
2  Start-Class: com.lichee.microservices.MicroservicesApplication
3  Spring-Boot-Classes: BOOT-INF/classes/
4  Spring-Boot-Lib: BOOT-INF/lib/
5  Spring-Boot-Version: 2.1.4.RELEASE
6  Main-Class: org.springframework.boot.loader.JarLauncher
7
8
```

这里指定了 **Main-Class** 类信息，那么 **JarLauncher** 这个入口类连同它的包结构 **org.springframework.boot.loader** 就必须在 **Spring Boot** 打包后的 **jar** 的最顶层目录下。



这就明白了，为什么 Spring Boot 不把 spring-boot-loader.jar 这个依赖传递给实际的 Spring Boot 应用（这种就是规范里说的不允许出现 jar 文件嵌套），而是采用把这个依赖的内容打包进实际的 Spring Boot 应用。

Jar 文件嵌套

标准规范里 jar 文件是不允许嵌套的，不能在一个 jar 文件内嵌套另外一个 jar 文件。Java 类加载器是加载不了嵌套 jar 文件的情况，但是有时为了合理的规避规范问题，有两种做法：

传统方式

通过 maven 的一些打包插件，比如：maven-shade-plugin 等相关插件。

假如一个应用工程依赖了 10 个三方依赖 jar，maven-shade-plugin 为了迎合 jar 文件规范。除了把应用工程自身的 class 文件打包到应用工程 jar 文件里，还把这 10 个三方依赖 jar 进行解压缩，把三方依赖 jar 的所有 class 文件同时打包到应用工程 jar 文件里。这样的话，应用工程 jar 文件里就没有嵌套其他 jar 文件，就符合了 jar 文件规范。

虽然解决了规范问题，但是也包含几个问题：

- 文件混乱

  应用工程自身的 class 文件和三方依赖 jar 里的 class 文件都放在了一个 jar 内，会导致打包后 jar 内的文件非常的混乱。

- 文件覆盖

  这 10 个三方依赖 jar 里的文件，出现了文件同名，比如：配置文件名字、java 类名、package 包名、handlers 文件名、schemas 文件名等同名情况。

  那么就会产生文件覆盖问题，这就会导致应用无法生产使用。

遗留问题：**maven-assembly-plugin** 这些插件是不是也可以做到把三方 **jar** 放在一个指定的目录来打包？

不寻常的 Spring Boot

通过上面的内容，发现 Spring Boot 实际打包出来的 jar 是不符合规范的。应用类加载器只会加载符合 jar 文件规范的类或 jar 文件，只有 spring-boot-loader.jar 解压后的所有类可以被加载。

源码包路径

Spring Boot 只是把 spring-boot-loader.jar 这个依赖的包内容解压后，放在应用工程 jar 的最顶层，而应用工程自身的 class 文件却放在了 BOOT-INF/classes 内，这种目录结构内的类是不能被应用类加载器加载的。

依赖包路径

Spring Boot 还把三方依赖 jar 放在 BOOT-INF/lib 内，这就产生了嵌套问题，jar 文件内嵌套了其他三方依赖 jar。这种目录结构内的 jar 包是不能被类应用加载器加载的。

解决办法

Spring Boot 创建了一个自定义的 Spring Boot Loader 类加载器，来加载 BOOT-INF/classes 目录下的类文件和 BOOT-INF/lib 目录下的 jar 文件。

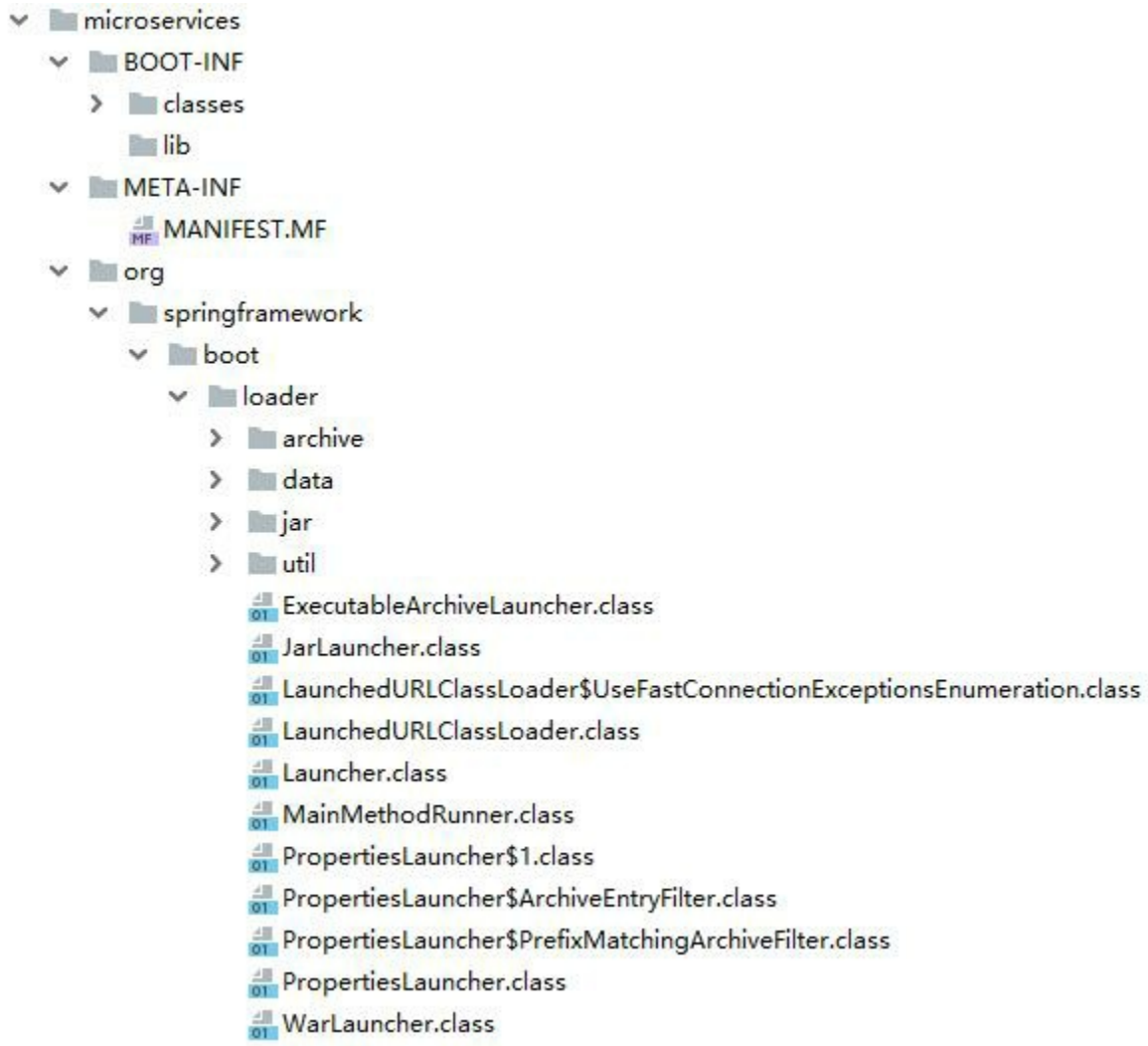
FatJar

类似于 Spring Boot 这种 jar 打包方式，jar 文件内嵌套了其他的 jar 文件，称为 FatJar。

Spring Boot Loader 源码分析

Spring Boot jar

我们之前就发现 Spring Boot 打包出来的 jar 解压后，有三个目录。



BOOT-INF

classes

当前工程编译好的结果文件，包含 src/main/java 和 src/main/resources 下的所有文件。

lib

当前工程依赖的所有 jar 文件（第三方 jar 包）。

META-INF

只有一个文件：MANIFEST.MF。

```
Manifest-Version: 1.0
Start-Class: com.lichee.microservices.MicroservicesApplication
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Version: 2.1.4.RELEASE
Main-Class: org.springframework.boot.loader.JarLauncher

```

org

由 Spring Boot 提供的一堆字节码文件，入口类就在这里。

```
org.springframework.boot.loader.JarLauncher
```

org package 是怎么来的

BOOT-INF 是通过编译工程文件，META-INF 是打包时自动生成，那么 org package 是怎么来的？

```
plugins {
   id 'org.springframework.boot' version '2.1.4.RELEASE'
   id 'java'
}
```

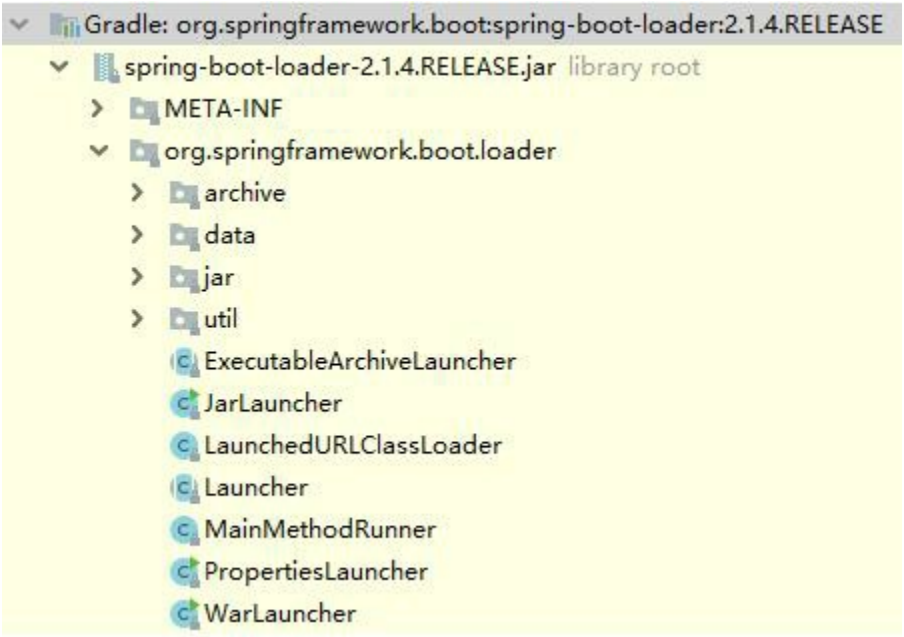
还记得我们之间配置的 gradle plugin 么？既然打包是由 org.springframework.boot 这个插件完成，那么答案肯定在这里。

根据 org 的目录结构，尝试增加一个依赖到工程里。

org.springframework.boot:spring-boot-loader （在开发阶段，因为有 **gradle** 插件的存在，原则上这个依赖是不引入的，这里是研究需要）

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-web'
   implementation 'org.springframework.boot:spring-boot-loader'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

再观察 spring-boot-loader.jar，发现和上面的 org package 一模一样。



结论：说明 Spring Boot Gradle Plugin 在用 bootJar 任务对 Spring Boot 工程进行打包的时候，是把 spring-boot-loader.jar 这个 jar 解压，随后把解压内容放置在 Spring Boot 的应用 jar 里。而 spring-boot-loader.jar 这个依赖，肯定是在 Spring Boot Gradle Plugin 里被依赖的。

疑问：为什么不直接把这个依赖传递给实际的 Spring Boot 应用，而是采用这么麻烦的方式，把这个依赖的内容打包进实际的 Spring Boot 应用。

JarLauncher 启动类

根据 MANIFEST.MF 文件的描述。

Spring Boot 的启动类就是

```
Main-Class: org.springframework.boot.loader.JarLauncher
```

而 Spring Boot 的应用主类是

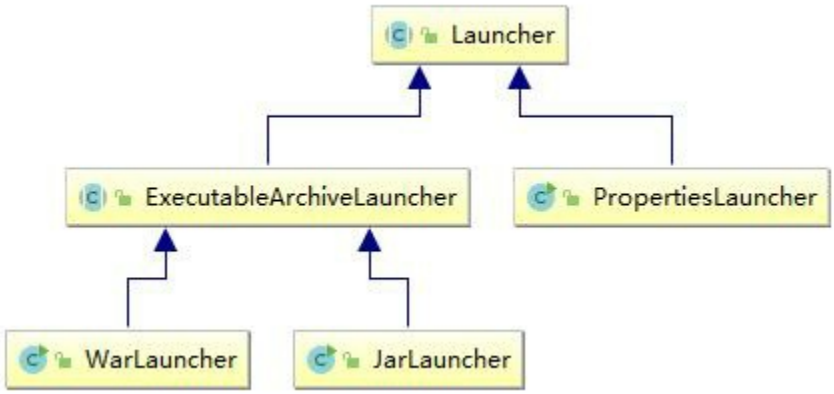
```
Start-Class: com.lichee.microservices.MicroservicesApplication
```

源码

```
package org.springframework.boot.loader;

import org.springframework.boot.loader.archive.Archive;

/**
 * {@link Launcher} for JAR based archives. This launcher assumes that dependency jars are
 * included inside a {@code /BOOT-INF/lib} directory and that application classes are
 * included inside a {@code /BOOT-INF/classes} directory.
 *
 * @author Phillip Webb
 * @author Andy Wilkinson
 */
public class JarLauncher extends ExecutableArchiveLauncher {
```

```
    static final String BOOT_INF_CLASSES = "BOOT-INF/classes/";

    static final String BOOT_INF_LIB = "BOOT-INF/lib/";

    public JarLauncher() {
    }

    protected JarLauncher(Archive archive) {
        super(archive);
    }

    @Override
    protected boolean isNestedArchive(Archive.Entry entry) {
        if (entry.isDirectory()) {
            return entry.getName().equals(BOOT_INF_CLASSES);
        }
        return entry.getName().startsWith(BOOT_INF_LIB);
    }

    public static void main(String[] args) throws Exception {
        new JarLauncher().launch(args);
    }

}
```

类关系图



从图中可以看出来，最顶层的启动器是 Launcher，其次是 ExecutableArchiveLauncher。最终的实现分别两种模式，一个是 JarLauncher（针对于 jar 归档文件），另一个是 WarLauncher（针对于 war 归档文件）。从这里也就说明了为什么 Spring Boot 的应用可以通过 jar 和 war 两种方式运行。

源码分析

通过代码可以发现，实际程序的入口是在：

```
new JarLauncher().launch(args);
```

实际调用的 launch 方法是 Launcher.java 里的 launch 方法。

```
/**
 * Launch the application. This method is the initial entry point that should be
 * called by a subclass {@code public static void main(String[] args)} method.
```

```
     * @param args the incoming arguments
     * @throws Exception if the application fails to launch
     */
    protected void launch(String[] args) throws Exception {
        JarFile.registerUrlProtocolHandler();
        ClassLoader classLoader = createClassLoader(getClassPathArchives());
        launch(args, getMainClass(), classLoader);
    }
```

这里一共 3 个步骤，最重要的就是第 2 步 ClassLoader 类加载器。

registerUrlProtocolHandler

注册URL协议处理器，这个不重要，可以忽略。

```
JarFile.registerUrlProtocolHandler();
```

classLoader

最重要的就是自定义类加载器这个阶段。

```
ClassLoader classLoader = createClassLoader(getClassPathArchives());
```

getClassPathArchives

返回所有符合条件的 jar 或 工程文件，并包装成一个类型为 Archive 的 List 对象。这里的符合条件是指在 BOOT-INF/lib/ 下的 jar 文件，和在 BOOT-INF/classes/ 下的所有工程文件，用来构建 classpath。

```
@Override
protected List<Archive> getClassPathArchives() throws Exception {
    List<Archive> archives = new ArrayList<>(
            this.archive.getNestedArchives(this::isNestedArchive));
    postProcessClassPathArchives(archives);
    return archives;
}
```

createArchive

定位当前执行的具体 jar 文件或者文件目录，通过磁盘上的绝对路径来定位，返回一个 Archive 对象。

```
protected final Archive createArchive() throws Exception {
    ProtectionDomain protectionDomain = getClass().getProtectionDomain();
    CodeSource codeSource = protectionDomain.getCodeSource();
    URI location = (codeSource != null) ? codeSource.getLocation().toURI() : null;
    String path = (location != null) ? location.getSchemeSpecificPart() : null;
    if (path == null) {
        throw new IllegalStateException("Unable to determine code source archive");
    }
    File root = new File(path);
    if (!root.exists()) {
        throw new IllegalStateException(
                "Unable to determine code source archive from " + root);
    }
    return (root.isDirectory() ? new ExplodedArchive(root)
            : new JarFileArchive(root));
}
```

getNestedArchives

返回与指定过滤器（EntryFilter）所匹配的嵌套归档文件。

```
@Override
public List<Archive> getNestedArchives(EntryFilter filter) throws IOException {
    List<Archive> nestedArchives = new ArrayList<>();
    for (Entry entry : this) {
        if (filter.matches(entry)) {
            nestedArchives.add(getNestedArchive(entry));
        }
    }
    return Collections.unmodifiableList(nestedArchives);
}
```

isNestedArchive

EntryFilter 过滤器，判断 entry 所指定的文件（具体 jar 文件或者文件目录）是否满足条件，满足条件的应该添加到 classpath 里，每个指定的文件都会调用一次。

条件：在 BOOT-INF/classes/ 下的工程文件或者在 BOOT-INF/lib/ 下的第三方 jar 包。

注意：其实这里就是在判断，要执行的 **jar** 文件是否是按照 **Spring Boot** 特有的目录结构来放置工程文件，以及所依赖的第三方 **jar** 包。只有满足条件的工程文件或所依赖的第三方 **jar** 包才会进入下一步，也就是通过自定义类加载器来加载这些满足条件的文件，这里就需要 **jar** 文件规范相关知识。

```
static final String BOOT_INF_CLASSES = "BOOT-INF/classes/";
static final String BOOT_INF_LIB = "BOOT-INF/lib/";

@Override
protected boolean isNestedArchive(Archive.Entry entry) {
    if (entry.isDirectory()) {
        return entry.getName().equals(BOOT_INF_CLASSES);
    }
    return entry.getName().startsWith(BOOT_INF_LIB);
}
```

postProcessClassPathArchives

是一个空方法，事后处理方法，回调方法。

```
/**
 * Called to post-process archive entries before they are used. Implementations can
 * add and remove entries.
 * @param archives the archives
 * @throws Exception if the post processing fails
 */
protected void postProcessClassPathArchives(List<Archive> archives) throws Exception {
}
```

createClassLoader

上面的所有方法，都是为了准备 List 对象，所有符合条件的 **jar**（**BOOT-INF/lib/** ）和工程文件（**BOOT-INF/classes/** ），并包装成一个类型为 **Archive** 的 **List** 对象。

创建一个针对指定归档文件（Archive）的自定义类加载器，也就是用来加载 getClassPathArchives 所返回的集合（jar 或者 工程文件），应用类加载器（也可以叫做系统类加载器）是加载不了这些文件的，就必须自己创建一个新的类加载器，用来加载这些存在于自定义目录内的文件。

这个方法是把传入的 List 对象转撑一个 List 对象，URL 表示文件在磁盘上的绝对路径。

```
**
 * Create a classloader for the specified archives.
 * @param archives the archives
 * @return the classloader
 * @throws Exception if the classloader cannot be created
 */
protected ClassLoader createClassLoader(List<Archive> archives) throws Exception {
   List<URL> urls = new ArrayList<>(archives.size());
   for (Archive archive : archives) {
      urls.add(archive.getUrl());
   }
   return createClassLoader(urls.toArray(new URL[0]));
}
```

LaunchedURLClassLoader

Spring Boot 提供的自定义类加载器，urls 表示所有需要加载文件的 url（jar 文件的绝对路径），getClass().getClassLoader() 表示父加载器（也就是应用类加载器）。

注意：在创建一个类加载器的时候，一定要指定它的父加载器 **getClass().getClassLoader(**），这个父加载器其实就是应用类加载器。

这个方法创建一个针对指定归档文件（URL）的类加载器。

```
/**
 * Create a classloader for the specified URLs.
 * @param urls the URLs
 * @return the classloader
 * @throws Exception if the classloader cannot be created
 */
protected ClassLoader createClassLoader(URL[] urls) throws Exception {
   return new LaunchedURLClassLoader(urls, getClass().getClassLoader());
}
```

launch

最后阶段，通过反射来完成工程应用启动。

```
launch(args, getMainClass(), classLoader);
```

getMainClass

返回应该被加载的主类。从 Manifest 对象种，获取 Start-Class 属性，这个 Start-Class 是什么呢？

MANIFEST.MF 文件中定义 Start-Class: com.lichee.microservices.MicroservicesApplication

```
@Override
protected String getMainClass() throws Exception {
   Manifest manifest = this.archive.getManifest();
   String mainClass = null;
   if (manifest != null) {
```

```
            mainClass = manifest.getMainAttributes().getValue("Start-Class");
        }
        if (mainClass == null) {
            throw new IllegalStateException(
                    "No 'Start-Class' manifest entry specified in " + this);
        }
        return mainClass;
    }
```

launch

通过给定的归档文件和全新的 classloader 来启动应用。

```
    /**
     * Launch the application given the archive file and a fully configured classloader.
     * @param args the incoming arguments
     * @param mainClass the main class to run
     * @param classLoader the classloader
     * @throws Exception if the launch fails
     */
    protected void launch(String[] args, String mainClass, ClassLoader classLoader)
            throws Exception {
        Thread.currentThread().setContextClassLoader(classLoader);
        createMainMethodRunner(mainClass, args, classLoader).run();
    }
```

第一行代码把自定义的 classloader 设置到当前线程上下文类加载器，在默认情况下，当前线程上下文类加载器就是 AppClassLoader。通过这种方式就把当前线程上下文的默认类加载器换成了 Spring Boot 自定义的类加载器。

转换：**AppClassLoader --> LaunchedURLClassLoader**

现在是把类加载器放置进去，在未来某处肯定会从当前线程中取出这个上下文类加载器，然后进行类加载。

createMainMethodRunner

创建 MainMethodRunner，用于启动和加载应用。

其实这里的 classLoader 并没有用到。

```
    /**
     * Create the {@code MainMethodRunner} used to launch the application.
     * @param mainClass the main class
     * @param args the incoming arguments
     * @param classLoader the classloader
     * @return the main method runner
     */
    protected MainMethodRunner createMainMethodRunner(String mainClass, String[] args,
            ClassLoader classLoader) {
        return new MainMethodRunner(mainClass, args);
    }
```

MainMethodRunner

使用当前线程上下文类加载器，加载一个包含了 main 方法的主类，然后调用这个主类的 main 方法。主要看这个 run 方法。

```
/**
 * Utility class that is used by {@link Launcher}s to call a main method. The class
```

```
 * containing the main method is loaded using the thread context class loader.
 *
 * @author Phillip Webb
 * @author Andy Wilkinson
 */
public class MainMethodRunner {

   private final String mainClassName;

   private final String[] args;

   /**
    * Create a new {@link MainMethodRunner} instance.
    * @param mainClass the main class
    * @param args incoming arguments
    */
   public MainMethodRunner(String mainClass, String[] args) {
      this.mainClassName = mainClass;
      this.args = (args != null) ? args.clone() : null;
   }

   public void run() throws Exception {
      Class<?> mainClass = Thread.currentThread().getContextClassLoader()
            .loadClass(this.mainClassName);
      Method mainMethod = mainClass.getDeclaredMethod("main", String[].class);
      mainMethod.invoke(null, new Object[] { this.args });
   }

}
```

run

当我们使用下面这个指令的时候，就会启动 Spring Boot 的应用，原理就在这个 run 方法。

```
java -jar microservices-0.0.1-SNAPSHOT.jar
```

Thread.currentThread().getContextClassLoader() 获取当前线程上下文类加载器，其实也就是获取我们之前已经设置好的 LaunchedURLClassLoader。之前从 MANIFEST.MF 文件中把 Start-Class 取出来，也就是我们的主类：com.lichee.microservices.MicroservicesApplication，通过 loadClass 方法就把主类加载到虚拟机中，于是得到主类的一个 Class 对象。

通过反射主类 Class 对象的方式，拿到主类的 main 方法。

```
Method mainMethod = mainClass.getDeclaredMethod("main", String[].class);
```

通过反射方式直接调用 main 方法。

```
mainMethod.invoke(null, new Object[] { this.args });
```

总结下来三个步骤：

1. 通过当前线程上下文类加载器加载 Start-Class 定义的主类到虚拟机中，拿到 Class 对象
2. 通过 Class 对象反射拿到 main 方法
3. 通过反射执行 main 方法

思考

为什么 **Spring Boot** 规定入口类（被 **@SpringBootApplication** 注释的类）的启动方法是 **main** 方法？

通过上面的源码分析，不需要非要 main 方法才可以，在反射阶段，完全可以换成另外一个普通的实例方法，只要在反射的时候，更换方法名就可以。

```
Method mainMethod = mainClass.getDeclaredMethod("main", String[].class);
```

原因在于为了方便 **Spring Boot** 应用开发阶段的运行、调试，但是类加载器不一样。

两种运行方式

Spring Boot 可以通过以下两种方式运行，好处就是便利了日常开发、调试、测试阶段。

但是类加载器会有明显的区别，修改代码，把类加载器打印出来。

```
@SpringBootApplication
public class MicroservicesApplication {

    public static void main(String[] args) {

        System.out.println(MicroservicesApplication.class.getClassLoader());
        SpringApplication.run(MicroservicesApplication.class, args);
    }

}
```

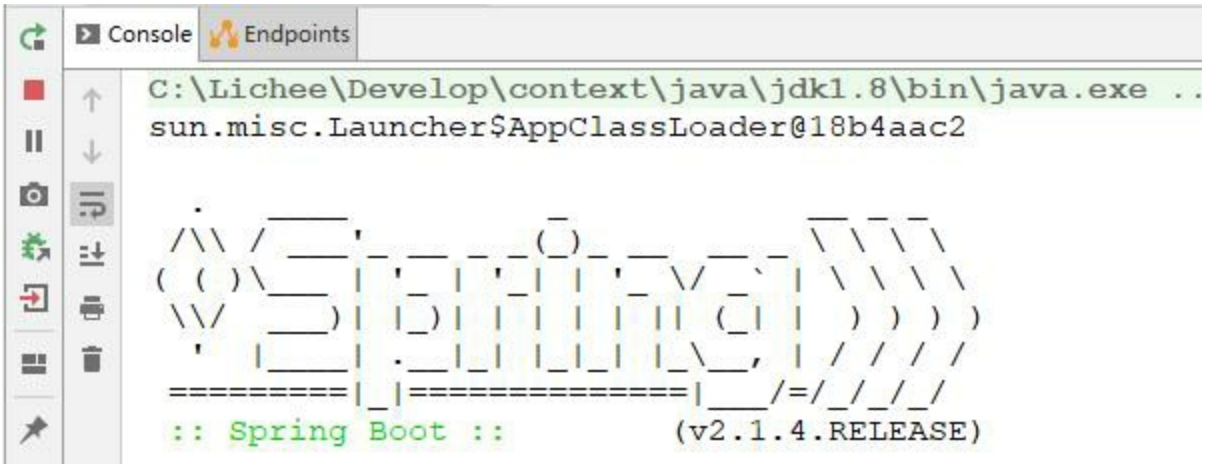
jar

通过指令运行。从结果可以看出来是采用的 **LaunchedURLClassLoader**，也就是采用 **Spring Boot** 提供的全新类加载器。

```
java -jar microservices-0.0.1-SNAPSHOT.jar
```

main

直接在开发阶段的 ide（eclipse、idea）里右键 run 方式运行。从结果可以看出来是采用的 **AppClassLoader**，也就是采用应用类加载器。



总结

打包机制

Spring Boot 打包机制，通过 gradle plugin 的 org.springframework.boot 插件对应用打成 jar 包，在 jar 包的根目录放置 spring-boot-loader.jar 这个 jar 包解压缩后的所有文件。自定义 BOOT-INF 和 META-INF 两个目录，BOOT-INF/classes 放置工程文件，BOOT-INF/lib 放置工程三方依赖，MANIFEST.MF 文件中定义 Start-Class 和 Main-Class 属性。

运行机制

Spring Boot 运行机制，首先应用加载器（系统加载器）加载 org.springframework.boot.loader.JarLauncher。在加载 JarLauncher 的同时，创建一个 Spring Boot 特有的类加载器 LaunchedURLClassLoader，用这个特有的类加载器加载 BOOT-INF 下的工程文件和三方依赖。最后通过反射调用 Start-Class 应用入口类的 main 方法启动应用程序。

上面所讲的是 jar 包形式运行，开发阶段直接在 ide 里右键 run 运行工程，则是直接调用系统的 AppClassLoader 类加载器。

优雅方式

Spring Boot 通过自定义类加载器这种方式，优雅的解决了 jar 文件规范问题。至于把 spring-boot-loader.jar 这个 jar 包里的文件原封不动的拷贝过来，是为了给应用类加载器（系统加载器）一个程序入口。先加载 JarLauncher 到内容中，再通过自定义类加载器加载自己的应用。

这也回答了之前的问题，为什么不把 **spring-boot-loader.jar** 这个依赖传递给工程应用，而是在打包的时候，直接打包进工程应用的最顶层。

反射扩展

Spring Boot Loader 源码分析中，有这么一行代码，通过反射调用 main 方法。

```
mainMethod.invoke(null, new Object[] { null });
```

invoke

invoke 方法一般会接收 2 个参数，第 1 个是被调用方法所在类的对象，第 2 个参数被调用方法所接收的参数。

而 Spring Boot 这里为什么第 1 个参数传递的是 null ？

源码

```
@SpringBootApplication
public class MicroservicesApplication {

    public static void main(String[] args) {

        SpringApplication.run(MicroservicesApplication.class, args);
    }

}
```

原因

这里反射调用的是 main 方法，main 方法是静态方法。原则上，静态方法是不归属于它所在的具体类，它只是把静态方法寄存在它所在的具体类。java 中所有的方法，都需要依托一个具体的类才能存在。寄存在这个类，其实也就相当于依托这个类。

静态方法不归属于它所在的类，而是归属于当前所在类对应的 class 对象。

所以这里是可以传 null，也可以传递具体的对象。

实践

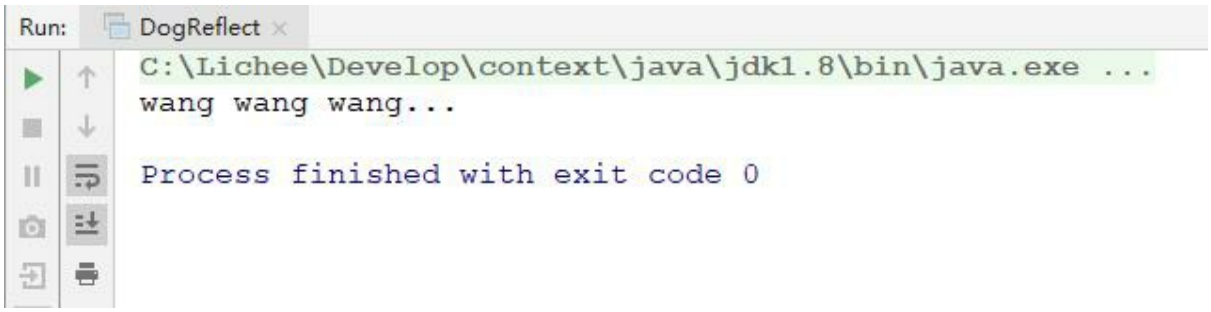
先定义 Dog。

```
public class Dog {

    public static void main(String[] args) {

        System.out.println("wang wang wang...");
    }
}
```

再定义 DogReflect，通过反射调用 Dog 的 main 方法。

```
public class DogReflect {

    public static void main(String[] args) throws NoSuchMethodException, InvocationTargetException, IllegalAccessExce
ption {

        Class<?> mainClass = Dog.class;
        Method mainMethod = mainClass.getDeclaredMethod("main", String[].class);
        mainMethod.invoke(null, new Object[] { null });
    }
}
```

正常执行，说明调用是没问题的。



通过一个实例来调用，而不用null，结果也是正确的。

```
mainMethod.invoke(new Dog(), new Object[] { null });
```

JDWP 远程调试

背景

在日常开发中，大家借助 IDE 都会经历调试程序的情况，除 IDE 以外，其实 Java 还提供了一种调试程序的协议，JDWP。

典型的场景，代码在本机运行正常，发布到服务器后，就出现了各种问题。这个时候如果想定位问题位置，就变得不是那么简单了。一种是打日志，就需要把相关的日志都打印出来，还牵扯到重新部署等等问题，会非常麻烦。另外一种比较好的方式是远程调试，程序还是在服务器，把服务器的程序与本机源代码做关联，在本机可以根据断点单步调试来定位问题。

定义

Java Debug Wire Protocol， Java 调试协议。

无论以 main 方法运行，还是以 jar 包运行，都可以远程调试。本质来说，是通过 socket 和端口号把服务器和客户端连接。连接建立成功以后，就可以通过断点定位到需要调试的程序。

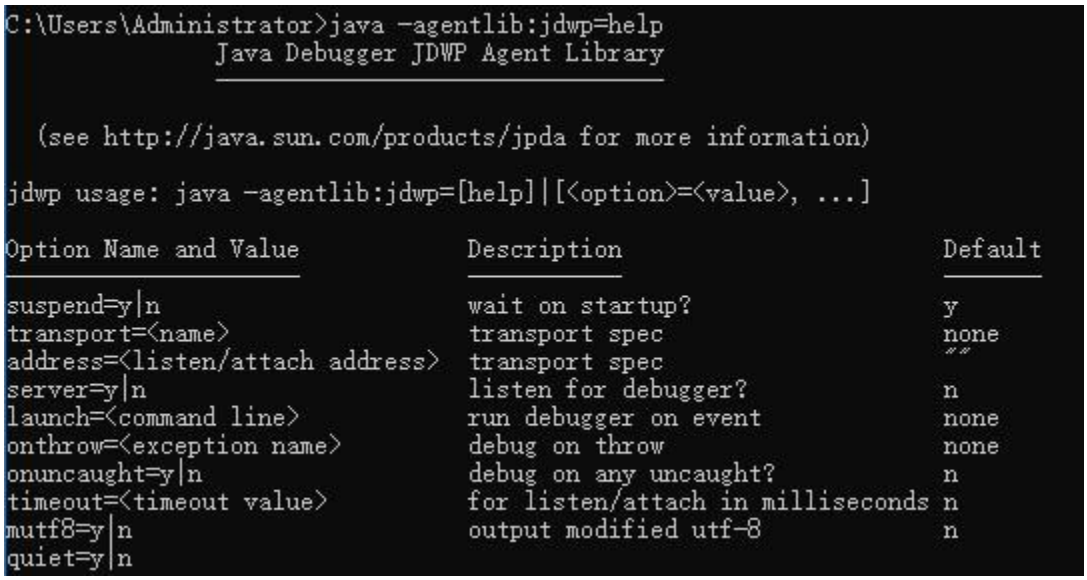
观察

打开命令行，输入 `java` 命令。

再次输入 agentlib 指令。

```
java -agentlib:jdwp=help
```

主要是前 4 个选项。



suspend

是否在启动的时候就等待，表示程序一启动就停下，等待远程调试 socket 和它建立连接。

transport

传输规范，用 JDWP 调试程序一般叫做：**dt_socket**。

address

地址，表示的是需要调试的地址。

server

是否监听调试器，需要改成 y，要监听调试器。

launch

当事件发生时运行调试器（用不到）。

onthrow

抛出异常时（用不到）。

onuncaught

没有捕获异常时（用不到）。

timeout

监听超时时间。（用不到）。

mutf8

（用不到）。

quiet

（用不到）。

使用

使用 JDWP 协议进行远程调试的时候，有两个 socket，一个是服务器，另一个是客户端。服务器优先启动，然后等待客户端来连接，客户端也称为调试器（debugger）。

关联源代码

首先把想要调试的源代码和 IDE 进行关联，并在 IDE 内打好断点。

题外话：直接在 idea 里面关联了 spring-boot-loader.jar 源码后，打好断点。右键 debug 来调试是无法进入 spring-boot-loader.jar 源码的，原因还是因为类加载器的问题。右键使用的是系统类加载器，并没有使用到 Spring Boot 提供的类加载器。

启动服务端 **socket**

监听调试器，启动好服务器端的 socket。

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5050 -jar microservices-0.0.1-SNAPSHOT.jar
```

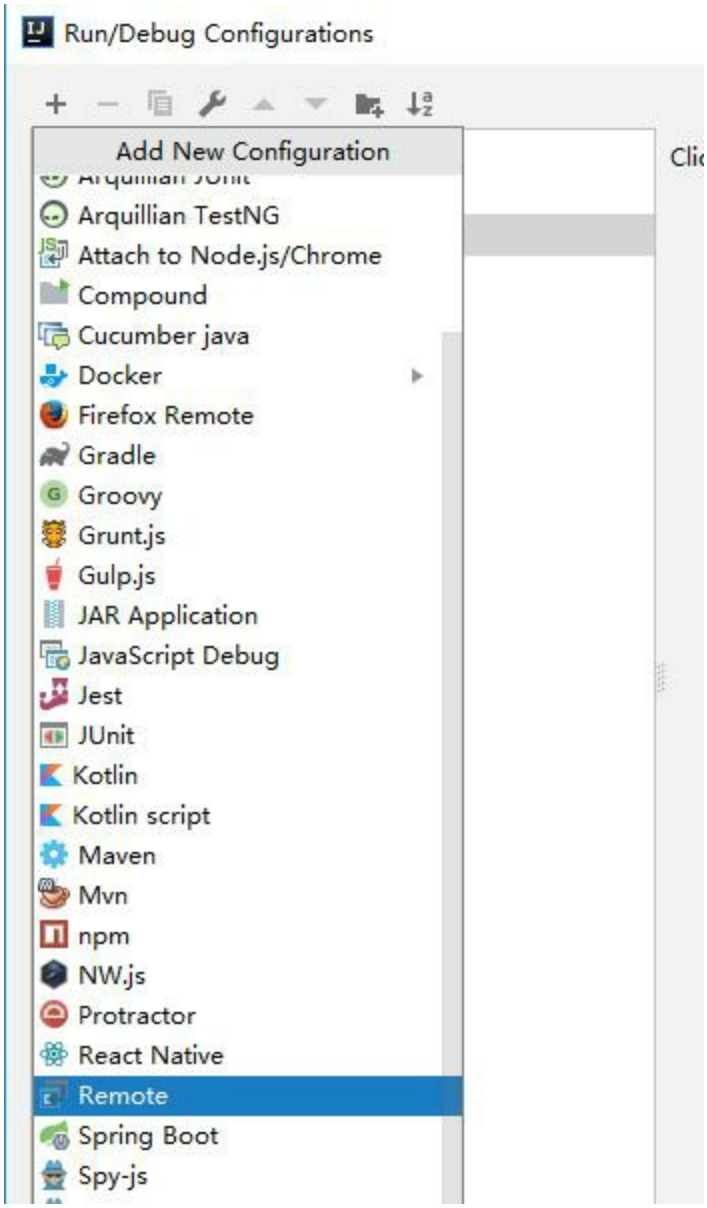
这里的 address 不用写 ip 地址，只需要些端口号就可以了。因为这里是启动服务器，只有客户端来连接服务器的时候，才需要指定服务器的 ip ，而服务器启动只需要指定端口就可以了。

```
D:\>java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5050 -jar microservices-0.0.1-SNAPSHOT.jar
Listening for transport dt_socket at address: 5050
```

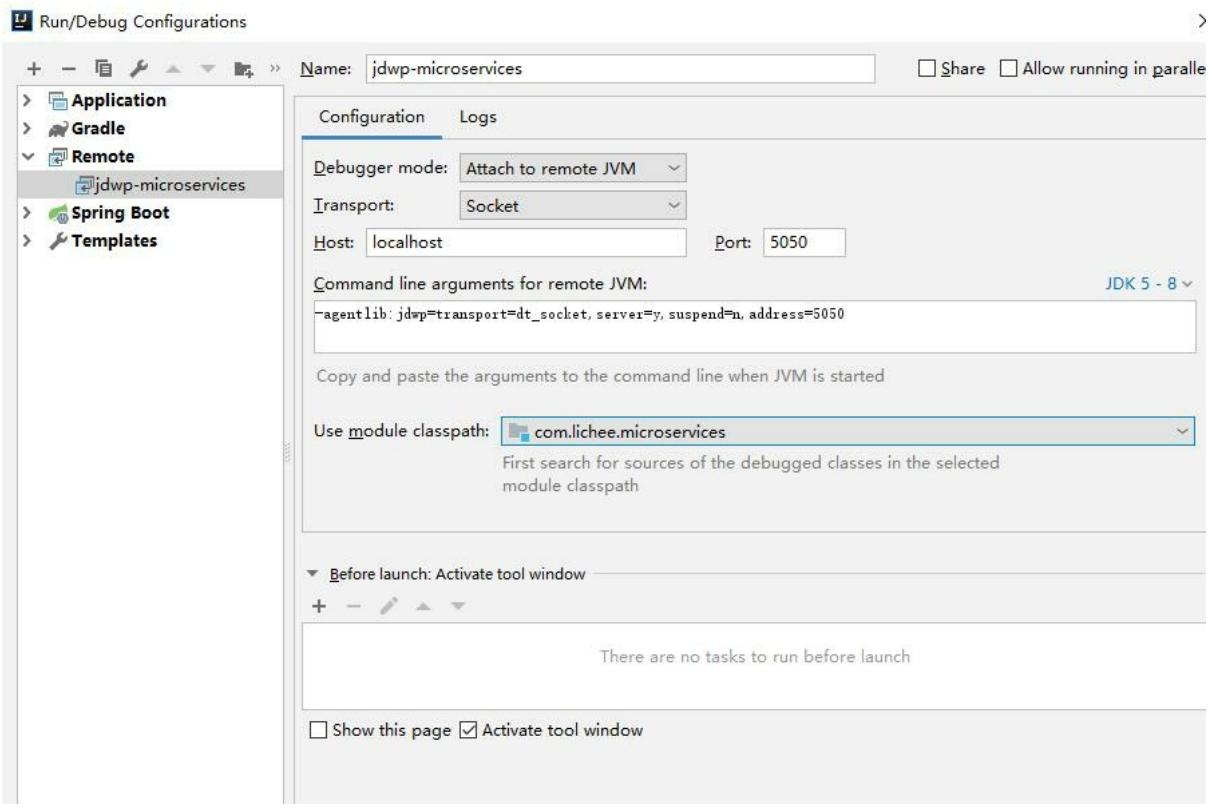
当出现 Listening for transport ... 的时候就说明服务端的 socket 已经启动成功了，在本地 5050 这个端口号上等待客户端 socket 的连接。

启动客户端 **socket**

首先在 Idea 的 Run/Debug 里面找到 Remote 选项。


Run/Debug Configurations
Add New Configuration
Arquillian TestNG
Attach to Node.js/Chrome
Compound
Cucumber java
Docker
Firefox Remote
Gradle
Groovy
Grunt.js
Gulp.js
JAR Application
JavaScript Debug
Jest
JUnit
Kotlin
Kotlin script
Maven
Mvn
npm
NW.js
Protractor
React Native
Remote
Spring Boot
Spy-js

随便取一个名字。

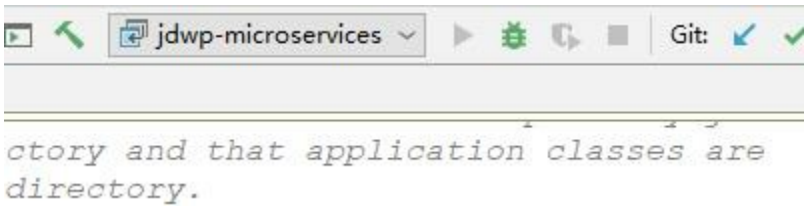


Debugger mode 选择 Attach to remote JVM，表示附加到远程的 JVM 上。Transport 选择 Socket 方式，Host 为本机，Port 修改为 5050，module 选择我们自己的模块。

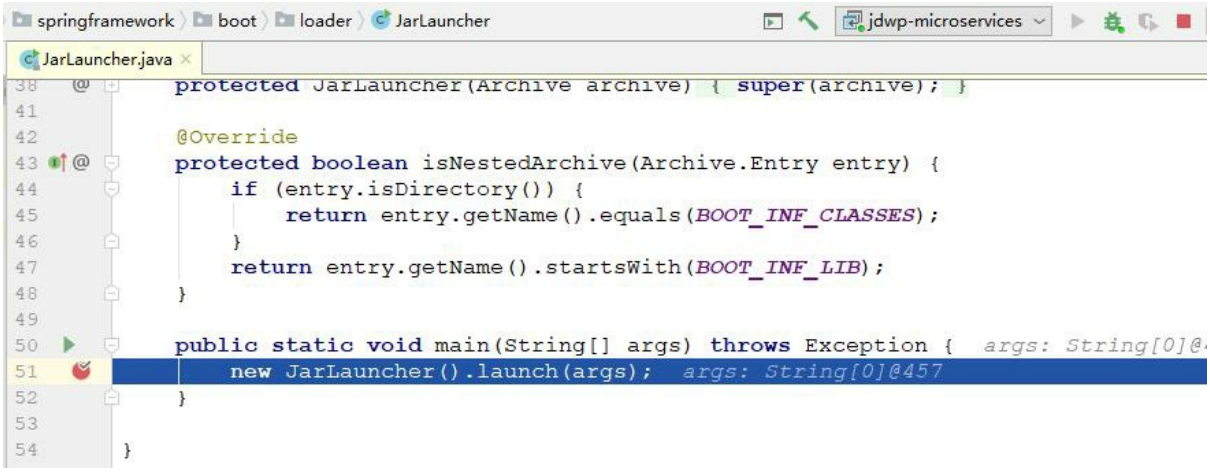
会发现下面出现了命令行指令

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5050
```

address 没有 ip 的原因是调试本机 localhost 可以省略，连接远程的话就必须把 ip 写全。比对没什么错误后，点击 OK，就可以在 Idea 里面直接运行了，点击 debug 小虫虫。



神奇般的进入了之前在 JarLauncher 里面打的断点。



现在就可以用调试的方式，来逐步阅读 spring-boot-loader.jar 源码了，对于特别复杂的框架源码来说，实时的查看每一步执行的结果，这样的效率提升大大降低了阅读源码的困难性。

注意

以上是基于 jar 运行方式的远程调试，如果目标应用是 web 应用，那么需要针对 tomcat 做一些 JDWP 的配置，就可以远程调试 web 应用了。一旦客户端断开了调试，那么服务器也需要重新启动 **socket**。

使用 JDWP 调试 Spring Boot Loader 源码

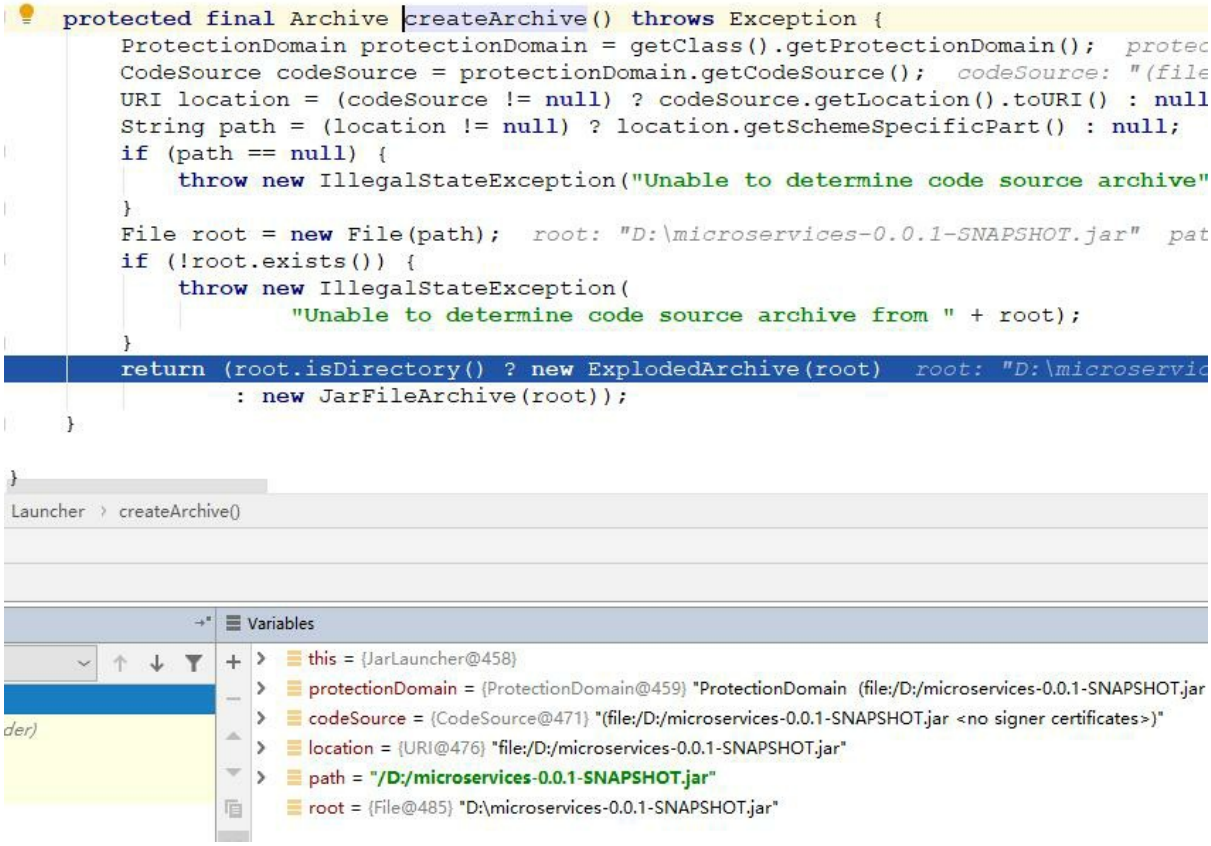
准备工作

按照之前 JDWP 远程调试 的内容，搭建好一个调试环境。

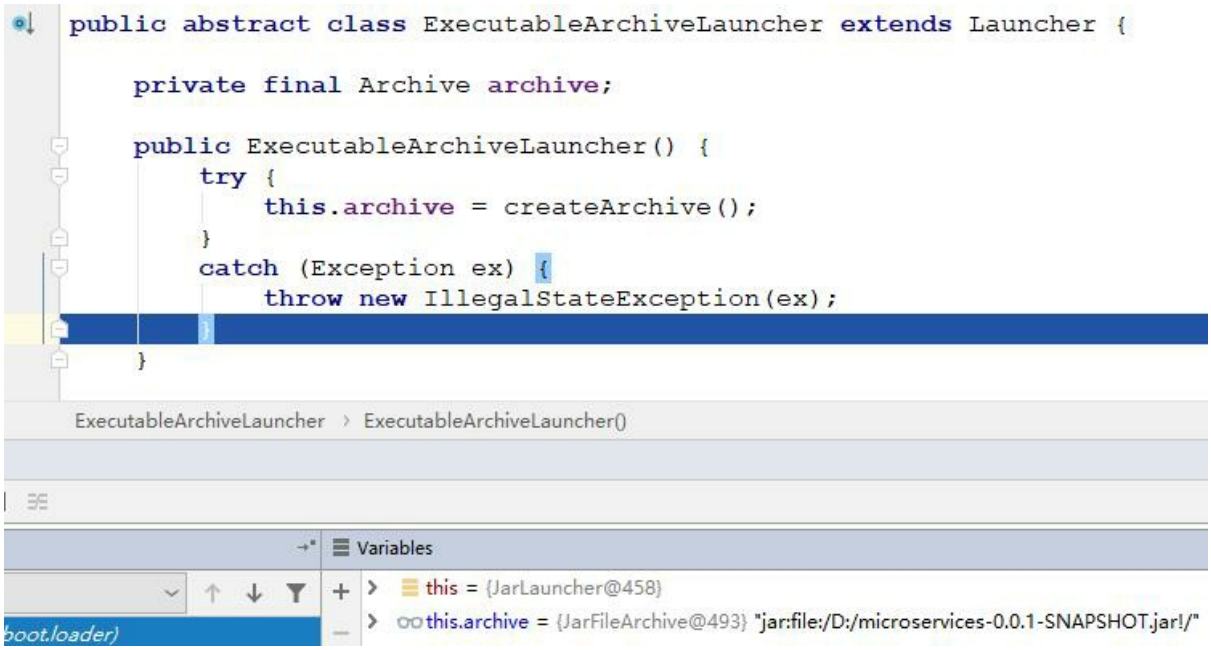
new JarLauncher

```
new JarLauncher();
```

在启动的第一步，子类 JarLauncher 构造器虽然没做什么，父类 ExecutableArchiveLauncher 在构造器中会调用 createArchive 方法来构造一个 Archive。其实这个 Archive 返回的就是被执行 jar 的绝对路径，并包装成一个 JarFileArchive。



观察 archive 的值。



launch(args)

```
/**
 * Launch the application. This method is the initial entry point that should be
 * called by a subclass {@code public static void main(String[] args)} method.
 * @param args the incoming arguments
 * @throws Exception if the application fails to launch
 */
protected void launch(String[] args) throws Exception {
   JarFile.registerUrlProtocolHandler();
   ClassLoader classLoader = createClassLoader(getClassPathArchives());
   launch(args, getMainClass(), classLoader);
}
```

registerUrlProtocolHandler

注册 url 协议处理器，不是特别重要。

createClassLoader

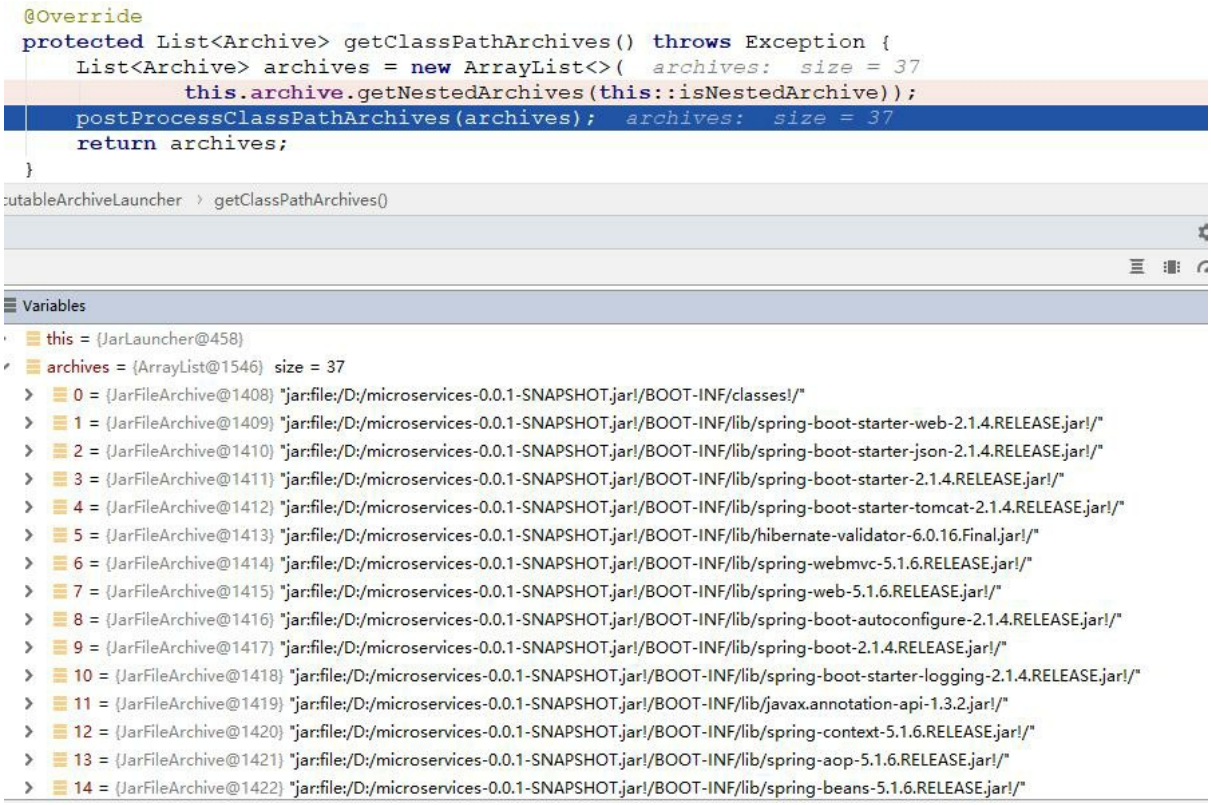
创建自定义加载器，相当重要。

getClassPathArchives

获取 jar 路径的归档文件集合。

```
@Override
protected List<Archive> getClassPathArchives() throws Exception {
   List<Archive> archives = new ArrayList<>(
         this.archive.getNestedArchives(this::isNestedArchive));
   postProcessClassPathArchives(archives);
   return archives;
}
```

返回了一个 List 集合，这里面装的是什么呢？先看结果，再分析。



返回的 archives 包含的就是 BOOT-INF/classes 这个目录 和 BOOT-INF/lib 下的所有三方 jar 包，也就是我们应用程序编译生成的 class 文件以及所依赖的三方库。

isNestedArchive

```
@Override
protected boolean isNestedArchive(Archive.Entry entry) {
   if (entry.isDirectory()) {
      return entry.getName().equals(BOOT_INF_CLASSES);
   }
   return entry.getName().startsWith(BOOT_INF_LIB);
}
```

它会判断当前 classes 目录或者 三方依赖 jar 文件，是不是位于 BOOT-INF 下。起判断作用。

getNestedArchives

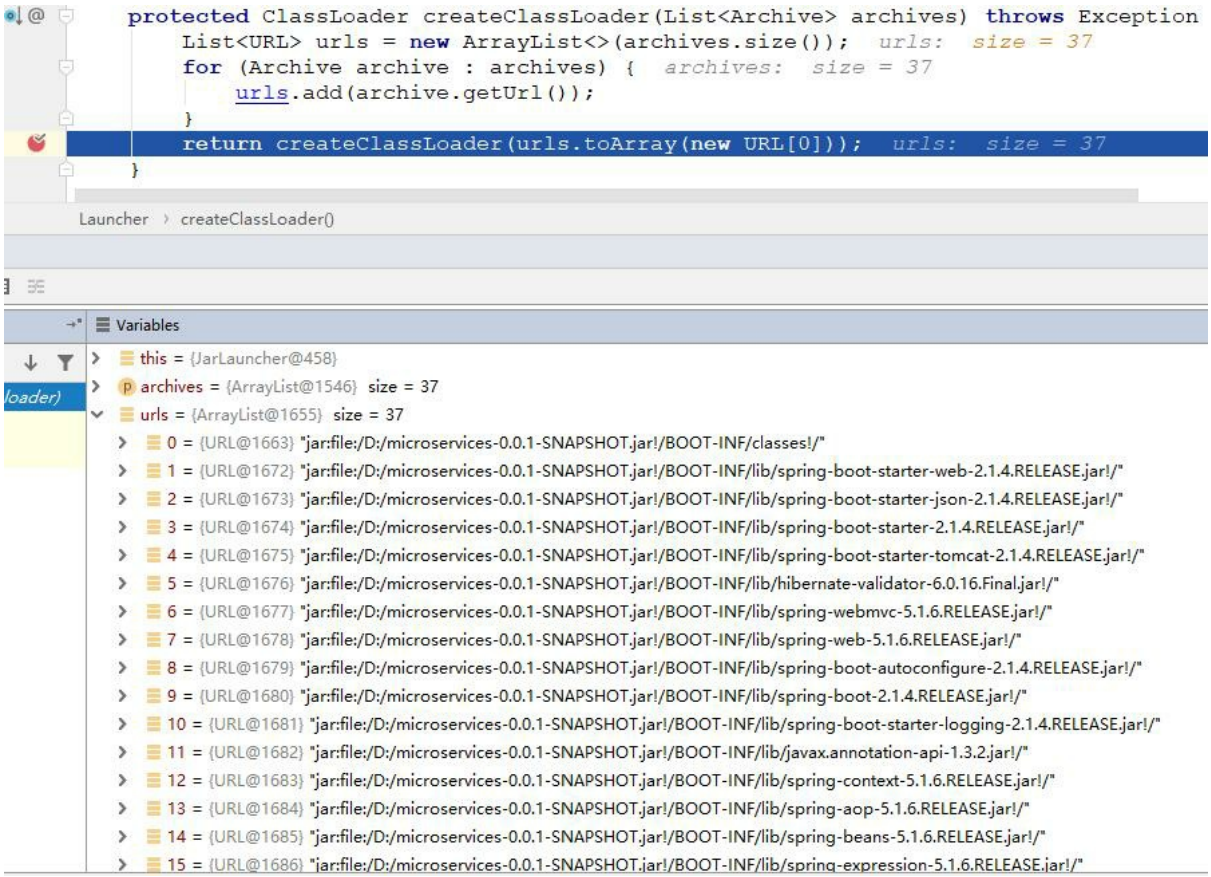
```
@Override
public List<Archive> getNestedArchives(EntryFilter filter) throws IOException {
   List<Archive> nestedArchives = new ArrayList<>();
   for (Entry entry : this) {
      if (filter.matches(entry)) {
         nestedArchives.add(getNestedArchive(entry));
      }
   }
   return Collections.unmodifiableList(nestedArchives);
}
```

根据 isNestedArchive 方法提供的判断，来构建 List。这里的 **for** 循环，是把整个被执行 **jar** 文件的每一个文件和目录都循环一次，当满足 **isNestedArchive** 要求（目录等于 **BOOT_INF_CLASSES** 或者文件等于 **BOOT_INF_LIB** 下的每一个三方依赖 **jar**）的 **entry** 都会被加入到集合当中，并且返回。

createClassLoader

```
/**
 * Create a classloader for the specified archives.
 * @param archives the archives
 * @return the classloader
 * @throws Exception if the classloader cannot be created
 */
protected ClassLoader createClassLoader(List<Archive> archives) throws Exception {
   List<URL> urls = new ArrayList<>(archives.size());
   for (Archive archive : archives) {
      urls.add(archive.getUrl());
   }
   return createClassLoader(urls.toArray(new URL[0]));
}
```

进行一次 for 循环，把 List 转换成 List，再观察 List 的值，这里转换成 **List** 的原因是，**Spring Boot** 自定义的类加载器 **LaunchedURLClassLoader**，其实也是继承 **URLClassLoader**，最终通过 **URL** 来进行类加载器的加载。



继续跟踪。

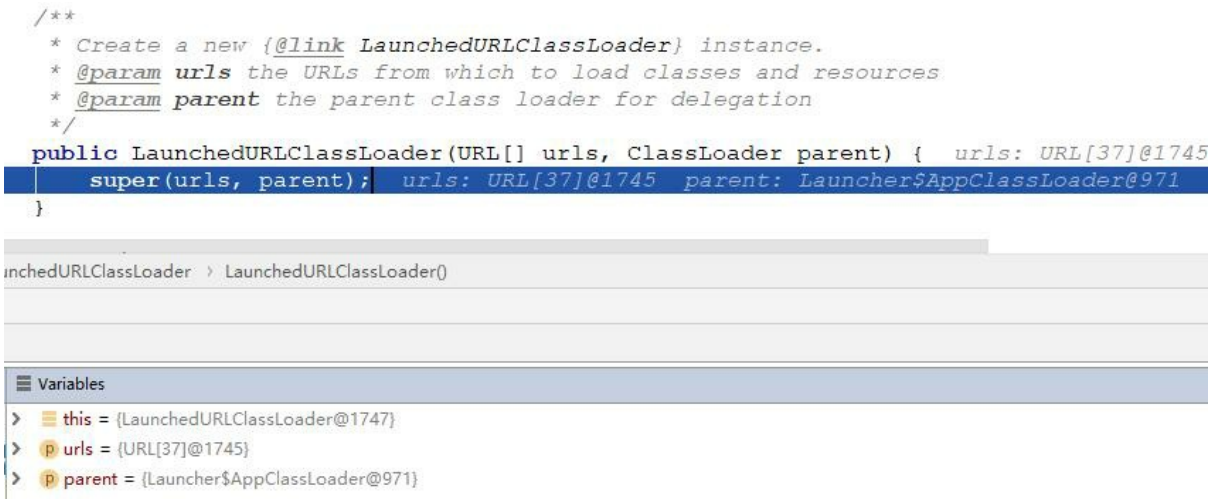
```
/**
 * Create a classloader for the specified URLs.
 * @param urls the URLs
```

```
     * @return the classloader
     * @throws Exception if the classloader cannot be created
     */
    protected ClassLoader createClassLoader(URL[] urls) throws Exception {
        return new LaunchedURLClassLoader(urls, getClass().getClassLoader());
    }
```

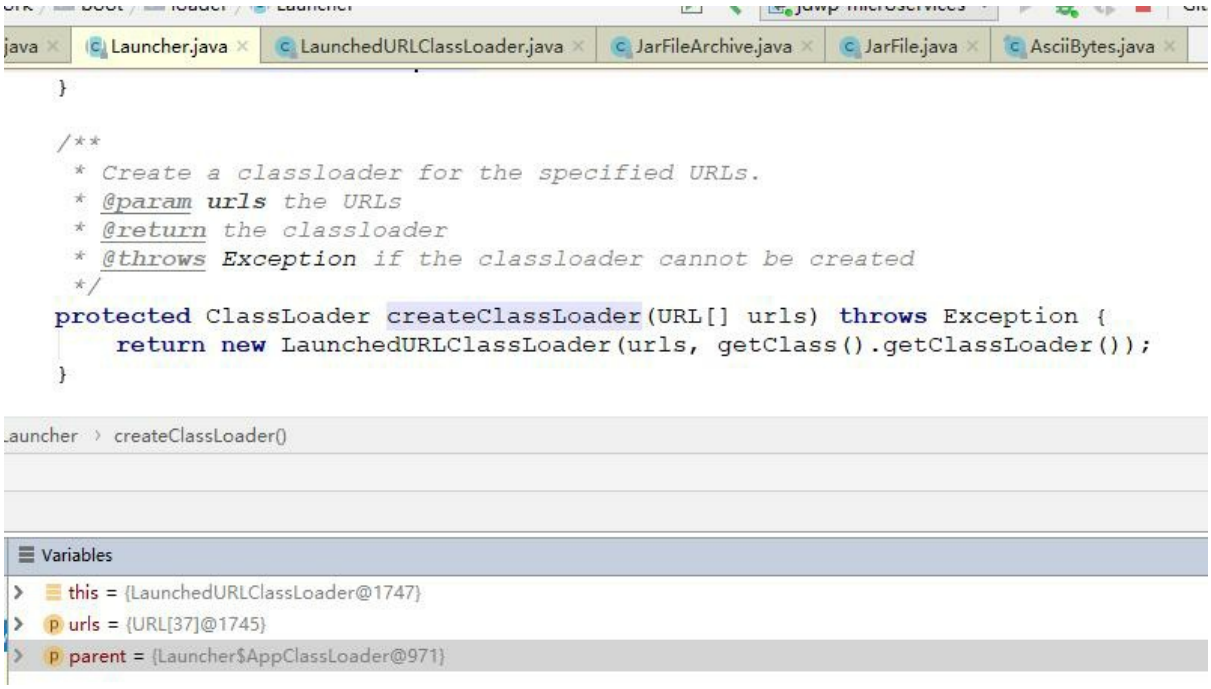
到了这里，就发现了 LaunchedURLClassLoader，Spring Boot 自定义的全新类加载器。这里面的 **getClass().getClassLoader()** 这个参数非常重要，当我们要创建一个新的类加载器的时候，一定要指定当前被创建类加载器它的父加载器。

LaunchedURLClassLoader

类加载器的构造方法，我们观察一下它的父加载器是什么？其实就是 **Launcher$AppClassLoader**。



这个父加载器的参数是从 **Launcher** 类传递过来的，所以名字叫做 **Launcher$AppClassLoader**，这也说明了，其实 **Launcher** 这个类是由 **AppClassLoader** 系统类加载器加载的。



上面的过程，也解释了为什么 Spring Boot 要把 spring-boot-loader.jar 整个 jar 包直接解压到 Spring Boot 应用 jar 包的最顶层，而不是传递依赖。通过这种方式，由系统类加载器来加载 Launcher 这些类，而后在由自定义的 LaunchedURLClassLoader 来加载 BOOT-INF 下面的工程文件和三方依赖 jar 文件。

到了这一步，自定义的类加载器 LaunchedURLClassLoader 已经准备好了。

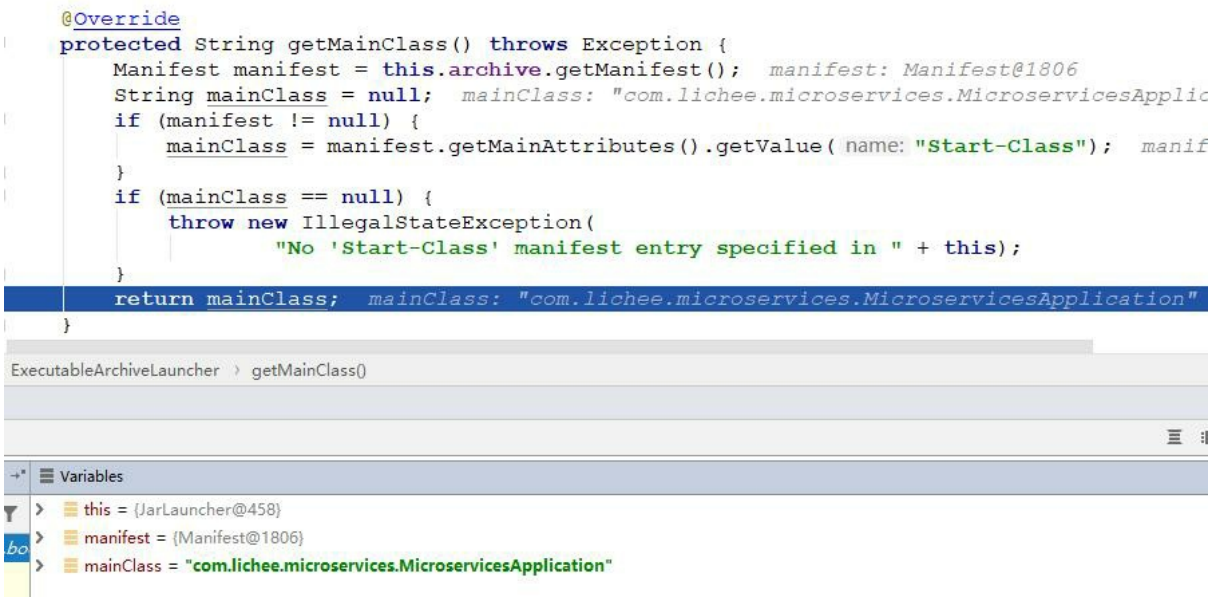
launch

getMainClass

通过这个方法来获得，在 MANIFEST.MF 文件中定义 Start-Class

```
@Override
protected String getMainClass() throws Exception {
   Manifest manifest = this.archive.getManifest();
   String mainClass = null;
   if (manifest != null) {
      mainClass = manifest.getMainAttributes().getValue("Start-Class");
   }
   if (mainClass == null) {
      throw new IllegalStateException(
            "No 'Start-Class' manifest entry specified in " + this);
   }
   return mainClass;
}
```

观察实际值，就是在 MANIFEST.MF 文件中定义 Start-Class。



launch

```
/**
 * Launch the application given the archive file and a fully configured classloader.
 * @param args the incoming arguments
 * @param mainClass the main class to run
 * @param classLoader the classloader
 * @throws Exception if the launch fails
 */
```

```
protected void launch(String[] args, String mainClass, ClassLoader classLoader)
      throws Exception {
   Thread.currentThread().setContextClassLoader(classLoader);
   createMainMethodRunner(mainClass, args, classLoader).run();
}
```

第一行代码把自定义的 classloader 设置到当前线程上下文类加载器，在默认情况下，当前线程上下文类加载器就是 AppClassLoader。通过这种方式就把当前线程上下文的默认类加载器换成了 Spring Boot 自定义的类加载器。

那么在调用 setContextClassLoader 之前，当前线程上下文类加载器是什么呢？

是 **Launcher$AppClassLoader** 类加载器。如果不去设置前线程上下文类加载器，在默认情况下，当前线程上下文类加载器就是 **AppClassLoader**。

createMainMethodRunner

创建 MainMethodRunner，用于启动和加载应用。

其实这里的 classLoader 并没有用到。

```
/**
 * Create the {@code MainMethodRunner} used to launch the application.
 * @param mainClass the main class
 * @param args the incoming arguments
 * @param classLoader the classloader
 * @return the main method runner
 */
protected MainMethodRunner createMainMethodRunner(String mainClass, String[] args,
      ClassLoader classLoader) {
   return new MainMethodRunner(mainClass, args);
}
```

run

```
public void run() throws Exception {
   Class<?> mainClass = Thread.currentThread().getContextClassLoader()
         .loadClass(this.mainClassName);
   Method mainMethod = mainClass.getDeclaredMethod("main", String[].class);
   mainMethod.invoke(null, new Object[] { this.args });
}
```

Thread.currentThread().getContextClassLoader() 获取当前线程上下文类加载器，其实也就是获取我们之前已经设置好的 LaunchedURLClassLoader。

可以观察 mainClass 是什么？就是我们应用的入口。

```
public void run() throws Exception {
    Class<?> mainClass = Thread.currentThread().getContextClassLoader()  mainClass: "cla
            .loadClass(this.mainClassName);
    Method mainMethod = mainClass.getDeclaredMethod( name: "main", String[].class);  main
    mainMethod.invoke( obj: null, new Object[] { this.args });
}
```

```
ainMethodRunner › run()

Variables
  this = {MainMethodRunner@1819}
  mainClass = {Class@1834} "class com.lichee.microservices.MicroservicesApplication"... Navigate
  this.args = {String[0]@1820}
  this.mainClassName = "com.lichee.microservices.MicroservicesApplication"
```

再观察反射获得的 method。

```
public void run() throws Exception {
    Class<?> mainClass = Thread.currentThread().getContextClassLoader()  mainClass: "cl
            .loadClass(this.mainClassName);
    Method mainMethod = mainClass.getDeclaredMethod( name: "main", String[].class);  mai
    mainMethod.invoke( obj: null, new Object[] { this.args });  mainMethod: "public stat
}
```

```
MainMethodRunner › run()

Variables
  this = {MainMethodRunner@858}
  mainClass = {Class@873} "class com.lichee.microservices.MicroservicesApplication"... Navigate
  mainMethod = {Method@875} "public static void com.lichee.microservices.MicroservicesApplication.main(java.lang.String[])"
  this.args = {String[0]@859}
```

最后，开始启动了。

```
D:\>java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5050 -jar microservices-0.0.1-SNAPSHOT.jar
Listening for transport dt_socket at address: 5050

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.4.RELEASE)

2019-06-04 11:25:32.922  INFO 27564 --- [           main] c.l.m.MicroservicesApplication           : Starting Micr
oservicesApplication on PC-20181120YWKI with PID 27564 (D:\microservices-0.0.1-SNAPSHOT.jar started by Lichee in D
:\)
2019-06-04 11:25:32.983  INFO 27564 --- [           main] c.l.m.MicroservicesApplication           : No active pro
file set, falling back to default profiles: default
2019-06-04 11:25:49.674  INFO 27564 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initia
lized with port(s): 9090 (http)
2019-06-04 11:25:50.224  INFO 27564 --- [           main] o.apache.catalina.core.StandardService   : Starting serv
ice [Tomcat]
2019-06-04 11:25:50.225  INFO 27564 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Serv
let engine: [Apache Tomcat/9.0.17]
2019-06-04 11:25:52.224  INFO 27564 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing
Spring embedded WebApplicationContext
2019-06-04 11:25:52.224  INFO 27564 --- [           main] o.s.web.context.ContextLoader            : Root WebAppli
cationContext: initialization completed in 18600 ms
2019-06-04 11:25:56.693  INFO 27564 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing
ExecutorService 'applicationTaskExecutor'
2019-06-04 11:25:59.771  INFO 27564 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat starte
d on port(s): 9090 (http) with context path ''
2019-06-04 11:25:59.802  INFO 27564 --- [           main] c.l.m.MicroservicesApplication           : Started Micro
servicesApplication in 33.794 seconds (JVM running for 322.403)
```

总结

Spring Boot Loader 采用非常精巧、整洁的设计，让应用能以不符合 jar 包规范的方式运行，也就是 FatJar。同时还能以传统的 `java -jar` 这种方式去启动执行，通过自定义类加载器的方式，非常优雅且合理的规避了很多 Jar 文件规范 问题。

借助于 Spring Boot Loader，可以让应用能以 FatJar 形式打成 jar 包，并部署。