

Spring Cloud Alibaba

从入门到实战

Spring Cloud Alibaba训练营合集



阿里云开发者社区 × 阿里巴巴云原生

ALIBABA CLOUD DEVELOPER COMMUNITY



阿里云开发者电子书系列



钉钉扫码加入
Spring Cloud Alibaba 交流群
如果发现电子书有任何问题，欢迎加群勘误



微信扫码关注
“阿里巴巴云原生公众号”



阿里云开发者“藏经阁”
海量免费电子书下载

大咖寄语



作为 Spring Cloud 的早期玩家，见证了 Spring Cloud 家族的成长与壮大。在 Netflix 体系不再继续提供新特性更新的大背景之下，Spring Cloud Alibaba 的出现，不仅提供了更符合中国开发者使用习惯的组件，也为 Spring Cloud 生态的其他使用者提供了更丰富的组件选择，承接了因 Netflix 体系不再更新导致的发展活力问题。相信在未来 Spring Cloud Alibaba 获得更多开发者的亲睐与应用，这也将成为 Java 开发者必不可少的技能之一。

——程序猿 DD 《Spring Cloud 微服务实战》作者



Spring Cloud Alibaba 脱胎于内部中间件，经受了阿里多年海量业务场景的考验，是目前最成熟、功能最丰富也最有前景的 Spring Cloud 实现。希望《Spring Cloud Alibaba 从入门到实战》电子书的发布有助于大家更加快速地上手，指导大家在项目中快速落地。

——周立（大目）《Spring Cloud 与 Docker 微服务架构实战》作者

| 目录

大咖寄语	3
基础知识篇	5
分布式配置	26
服务注册与发现	54
分布式服务调用	74
服务熔断和限流	89
分布式消息（事件）驱动	107
分布式事务	113

基础知识篇

前言

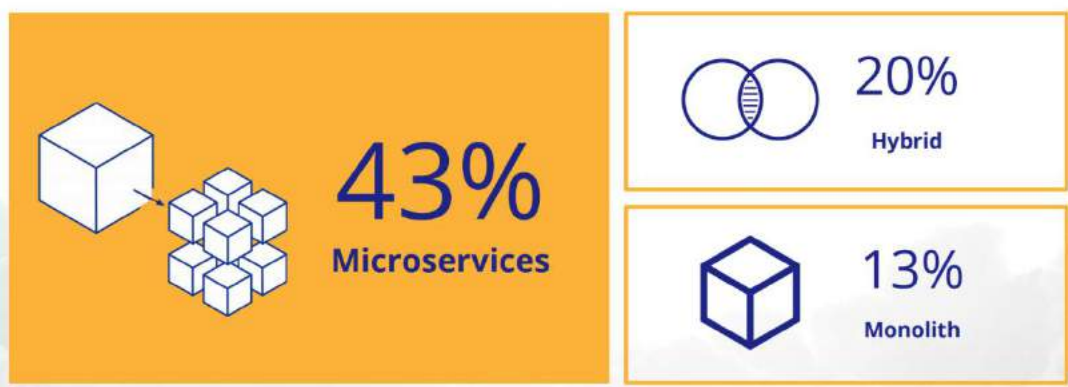
近些年随着云技术的发展,越来越多的用户选择使用云技术来代替将传统的 IT 基础设施。在云技术发展的早期,业界的关注点集中在虚拟化、分布式、存储等 IaaS 方面的技术。但是随着“云原生”概念的提出,大家的注意力开始转移到如何构建更加适合云环境运行的应用上来。

“什么样的架构才是适合在云环境中运行”是一个非常大的问题,在此先不展开讨论,而是到 CNCF 对云原生的定义中寻找答案:

云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中,构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段,云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。

从上文的定义中可以发现“微服务”在云原生技术中占有这非常重要的位置。在 jakarta.ee 2019 年的调研报告中也印证了这一点,超过 40% 公司选择采用微服务架构来构建云上系统:

Architectures for **implementing Java systems in the cloud**:



让我们继续把视角聚焦到 Java 语言下。Pivotal 依托于 Spring 这个 Java 语言下编程模型的事实标准，结合大量业界经验，为广大 Java 开发者带来了 Spring Cloud 框架。该框架是对云环境下微服务开发中需要解决的各种问题的标准化，以帮助开发者快速实现分布式系统中的某些常见模式（例如，配置管理，服务发现，断路器等）。

同时，基于 Pivotal 强大的标准制定能力与影响力，Spring Cloud 拥有一个强大的国际化社区，阿里巴巴作为这个社区里重要的成员，也贡献出了 Spring Cloud Alibaba 这套优秀的实现，这也是目前整个 Spring Cloud 体系下最完善并且在持续更新的实现方案。

学习目标

通过本次课的学习，你应该了解或掌握如下知识点：

- 微服务的发展历程
- 微服务的基本形式
- Spring 、 Spring Boot 、 Spring Cloud 的职责与关系
- Spring Cloud Alibaba 的功能与定位
- Java 工程脚手架的使用方法
- Sandbox 沙箱环境的使用发方法

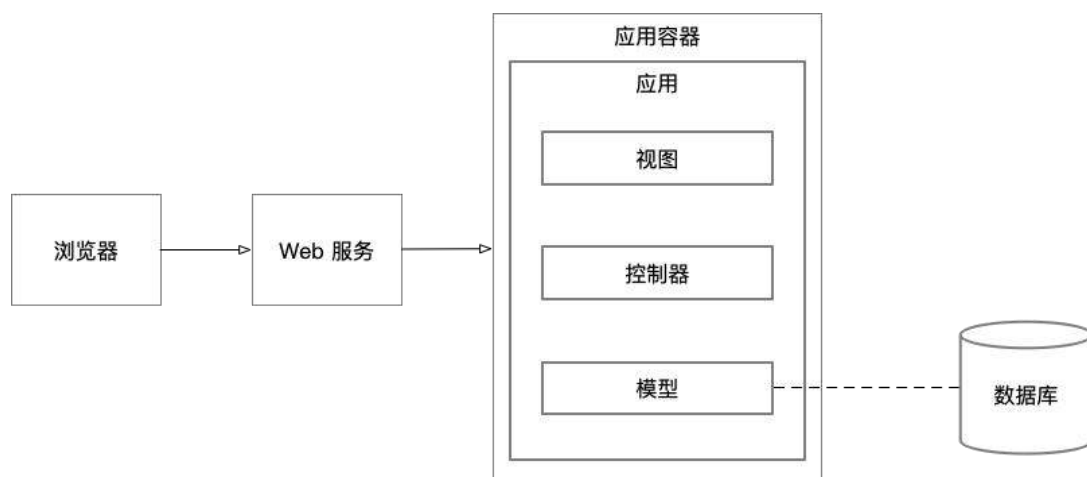
理论篇

俗话说，没有最好的架构，只有最合适的架构。微服务架构也是随着信息产业的发展而出现的最有普遍适用性的一套架构模式。通常来说，我们认为架构发展历史经历了这样一个过程：单体架构 -> SOA 面向服务架构 -> 微服务架构。

单体架构

在我们还是学生的年代，我们创建的绝大部分应用都属于单体应用。那个时候，我们几乎都是一个人在开发导师布置下来的各种实验。我们会把数据库连接、业务逻辑处理、展示逻辑等放在一起，甚至会在处理用户请求的地方直接连接数据库（多么美好的回忆啊 ^_^）。

后来，我们会学习到 MVC 架构以及由此衍生出来各种多层架构，由此便开启了应用的拆分之旅。多层架构的本质，是按照技术职责将应用做水平拆分，每一层解决的技术问题相对集中，层与层之间做单向依赖。这样做可以帮助我们更好的管理我们的代码，大大提升了后期的维护效率。但是，此时应用还是一个应用，部署时也是按照一个整体运行。我们看到的架构应该类似下面的样子：



在程序规模不大，开发人员很少的时候，下面的有点是非常显著的：

- 开发简单。单体应用的结构，天然决定了所有代码都集中在一起，开发者不需要在多个应用之间来回跳转来寻找其中的调用逻辑。
- 测试简单。所有代码都在一个应用里，测试人员可以很方便的做到端到端的测试（当然，很多时候测试人员就是开发者自己）。
- 部署简单。因为一个应用就是产品功能的全集，所以在部署的时候，只需要不是一款应用即可。即使是集群部署，也不会增加多少复杂度：只需要将应用部署多份即可。
- 开发迅速。上面的各种简单，带来的就是软件功能可以快速实现。很多时候，实现需求的速度是项目成功与否的决定性因素。

所以，在开发简单&独立的产品时，单体架构依然是第一优先选择。

如果故事可以一直这么简单就好了。

随着功能的持续增加、团队规模的不断扩大，我们很快就会发现单体应用的弊端：

- 应用膨胀。所有代码都在一个应用里，导致应用的代码量迅速上升，对于开发者来说，经常需要在海量的代码里找到自己需要维护的哪一行，这种体验往往是令人崩溃的。同时，对于 IDE 来说，一个应用内大量代码也会严重拖慢其运行效率。
- 团队合作冲突。这种冲突会体现在多个方面：开发阶段，很容易由于修改相同的代码导致代码冲突。部署阶段，又会因为“运行环境里跑的是谁的分支”而造成新的冲突。所有的这些冲突将会严重影响到团队的合作效率。
- 运行效率&稳定性。单体应用，由于逻辑都集中在一起，启动时需要完成所有的初始化工作；同时单一功能的问题也会因为运行在一个进程内，从而导致整个应用宕机。

单体架构原有的迅速、简单的优点，随着规模的扩大（功能、团队），会变得荡然无存。

为了能解决这些问题，我们自然而然就会想到分而治之的办法，即将原来的单体应用拆分开来。但是应用该怎么拆分？拆分后又会有哪些新的问题产生？如何解决这些新的问题？就留给下面的 SOA 架构来解答。

SOA 架构

SOA 是 Service-Oriented Architecture 的简写，直译为“面向服务的架构”，从命名上就可以看出“服务”是 SOA 架构里是非常重要的概念。SOA 的核心思想是“将系统的功能解构为一系列服务”：

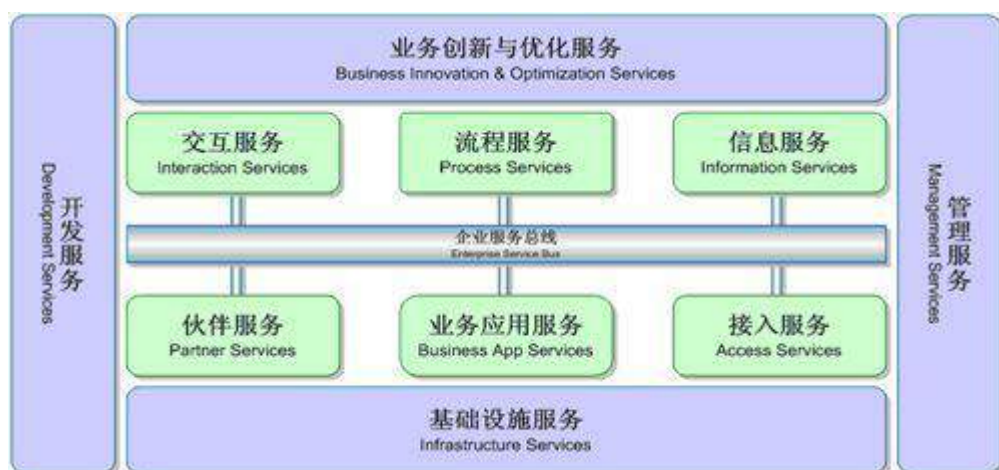
面向服务的架构（SOA）是一个组件模型，它将应用程序的不同功能单元（称为服务）进行拆分，并通过这些服务之间定义良好的接口和协议联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构件在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

与单体架构按照技术职责进行水平拆分不同，SOA 会按照业务领域对应用进行粗粒度的垂直拆分，至于拆分到什么程度，哪些领域可以放在一起等类似问题，可以参考一下康威定理。

应用从单体应用做了垂直拆分以后，就会变成一些相对独立的应用。此时，应用间的依赖、调用等相关问题自然而然的就会浮现出来。此时就需要下面这些技术方案来解决这些问题：

- XML – 一种标记语言，用于以文档格式描述消息中的数据。
- SOAP (Simple Object Access Protocol) – 在计算机网络上交换基于 XML 的消息的协议，通常是用 HTTP。
- WSDL (Web Services Description Language, Web 服务描述语言) – 基于 XML 的描述语言，用于描述与服务交互所需的服务的公共接口，协议绑定，消息格式。
- UDDI (Universal Description, Discovery, and Integration, 是统一描述、发现和集成) – 基于 XML 的注册协议，用于发布 WSDL 并允许第三方发现这些服务。
- ESB (Enterprise Service Bus, 企业服务总线) – 支持异构环境中的服务、消息，以及基于事件的交互，并且具有适当的服务级别和可管理性。

一个典型的 SOA 架构模式如下图：



SOA 看似解决了单体架构的所有问题，世界似乎都变得更加美好了 ^_^

但是…

SOA 并不完美，他也有很多问题的或者说是场景下的不适应。首先就是对 SOA 的解释缺乏统一标准，上文的引用的定义也只是众多解释中使用的较为通用的一种。甚至可以这么说：一千个人眼中，有一千种 SOA 。基于此，很多厂商便借用 SOA 的大旗来推广自己的产品和标准，这又进一步加剧了问题的严重性。

除此之外，SOA 还有很多其他的问题或不足：

- 高门槛。ESB 本身就是一套非常复杂的系统，通过 ESB 落地 SOA ，对开发人员的要求很高。甚至还会需要厂商参与；
- 厂商绑定。由于缺乏统一保准，不同厂商的解决方案之间很难做切换。
- 不适应云环境。在如今的互联网时代，速度就是一切。由此诞生了敏捷开发、持续集成等在不同节点提升业务上线速度的办法。但是方向是不一致的。
- 中心化。虽然应用本身实现了分布式与水平扩展，但是 ESB 却成了系统的中枢神经。

微服务架构

对于微服务架构，一直有一种说法，认为它是 SOA 架构的一种变体，或者是 SOA 的子集。关于这个问题，我们不去讨论他的对错（其实也没有对错之分），我们直接从这两者的区别入手来理解到底什么是微服务：

	传统 SOA	微服务
通信方式	基于 ESB，SOAP、WSDL 等重协议	点对点通信，开放式协议，如 RESTful、gRPC、或者是轻量级的二进制协议。
数据管理	全局数据模型以及共享存储	每个服务独立模型和存储
服务粒度	较粗	较细
诞生的背景	企业级应用	互联网
解决的问题	面向企业内，系统集成	面向最终产品，解决扩展，维护的问题

通信手段、数据等的不同只是表象，其本质区别还是由于两者诞生于不同历史时期，需要解决的问题域不同。SOA 解决的核心问题是复用，而微服务解决的核心问题是扩展。

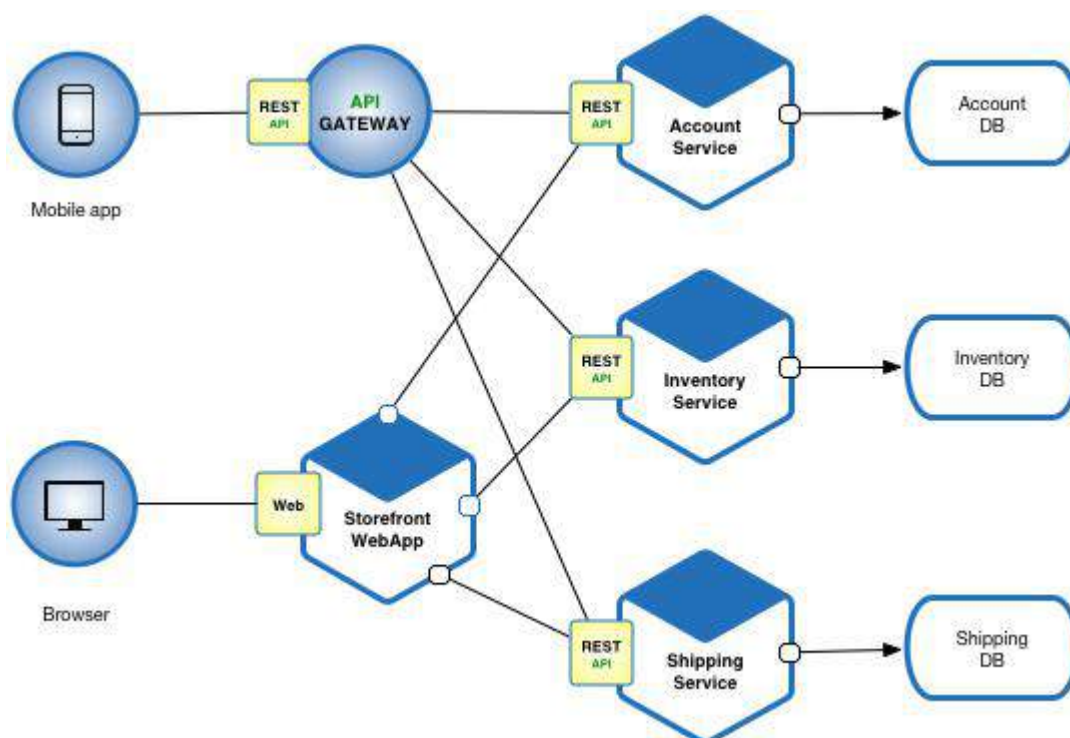
关于什么是微服务，Martin Fowler 有如下的定义（更多 MartinFowler 关于微服务的内容，可以参考[链接](#)）：

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.?

这里提到几个重要的概念：

- 一套小服务
- 独立进程
- 轻量级通信协议
- 可独立部署
- 多语言&不同储技术

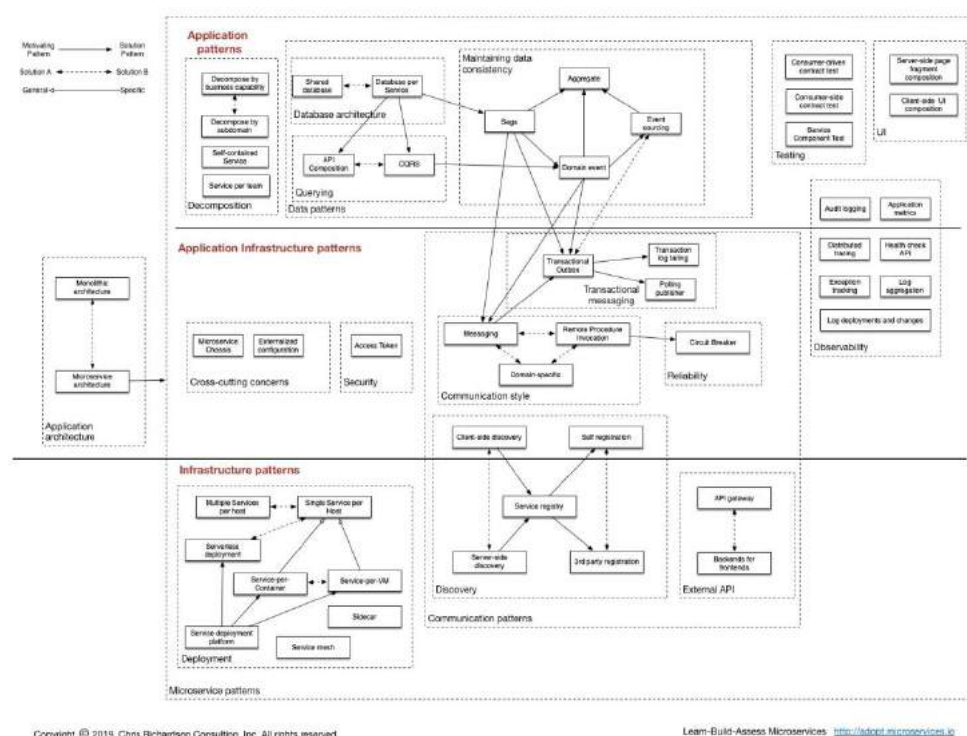
这个定义对微服务做了一个比较具象化较为易于理解的描述,通常来说我们看到的为服务架构如下图所示：



但是事实上，在实际生产环境中，微服务的架构要考虑的问题远比上面的示意图复杂的多，主要包括但不限于如下问题：

- 通过服务实现组件化
- 根据业务组织系统
- 做产品而不是做项目
- 简单高效的通信协议
- 自动化基础设施
- 面向失败的设计
- 具备进化能力的设计

纵然有 Martin Fowler 这样的大神在前面引路，但是我们依然认为“微服务”不是一个被设计出来的架构，而是在不断的尝试中总结出的一套适合在互联网行业使用的架构模式。以下的是微服务较为完整的架构图（出自[微服务架构模式](#)）。



今天我们所说的“微服务”是一个庞大且复杂的概念集合，它既是一种架构模式，也是实现这种架构模式时所使用的技术方案的集合。

“微服务”不是银弹

微服务并不是一劳永逸的解决了所有的问题，相反的，如果不能正确的使用微服务，则有可能被微服务自身的限制拖入另一个泥潭：

- 分布式的代价。原本在单体应用中，很多简单的问题都会在分布式环境下被几何级的放大。例如分布式事务、分布式锁、远程调用等，不光要考虑如何实现他们，相关场景的异常处理也是必须要考虑到的问题。
- 协同代价。如果你经历过一个项目上线需要发布十几个应用，而这些应用又分别由多个团队在维护。你就能深刻的体会到协同是一件多么痛苦的事情了。
- 服务拆分需要很强的设计功力。微服务的各种优势，其中一个重要的基础是对服务领域的正确切分。如果使用了不合适的切分粒度，或者是错误的切分方法，都会让服务不能很好的实现高内聚低耦合的要求。

框架篇

从 Spring 到 Spring Cloud

Spring

熟悉 java 语言的同学，对 Spring 框架应该都不陌生。从 2004 年 1.0 版本发布开始，便由于其灵活易用的特性受到了整个 Java 行业的广泛关注。经过十多年的发展，Spring 框架早已经成为 Java 语言下编程模型的事实标准。其所倡导的 IOC/AOP 概念也早已深入人心。

在 Spring 框架的早期，大家都喜欢称其为“轻量化”框架（现在好像早就没人提这个词了^_^），“轻量”是相对于 EJB 等企业级开发框架而言的。其“轻”的特性体现在：框架本身的大小很小，早期版本的 jar 包不超过 1MB；同时不依赖于运行容器，也就是说任何容器里都可以运行 Spring 框架；更加重要的是 Spring 是非侵入的，使用 Spring 开发的应用可以不完全依赖 Spring 的类。

Spring Boot

但是事情总会发生变化，随着 Spring 的不断发展，越来越多的组件被集成到了框架中。Spring 框架也从一个小巧精简的 IOC 容器框架变成了一套大而全的框架集合。开发者为了实现组件的整合工作，往往需要在大量的 xml 文件、java 注解 中完成各种 bean 的配置。曾经屠龙的少年，如今也变成了恶龙。

那个时候，很多比 Spring 更加简单小巧的 IOC 容器如雨后春笋般的出现。业界开始出现一种声音：Spring 是不是已经不行了，或者是在走下坡路了。就在这个时候 Pivotal 推出了 Spring Boot 来彻底的解决这些问题。

使用 Spring Boot 可以大大简化 Spring 应用的开发工作。在 Spring Boot 中无论是官方组件还是第三方框架都会提供各种“starter”来方便开发者进行依赖和集成。由于采用了“约定大于配置”的思想，开发者在引入“stater”以后只需要做少量的配置工作就可以完成框架集成工作。往往开发者只需要很少量的代码就可以实现以前大量配置文件才能做到的功能。

同时 Spring Boot 还是一套面向生产环境设计的框架。配置外化、运行情况检查功能，可以很方便的在系统外部实现对系统的管理。同时 Spring Boot 还是一个运行时容器。通过内嵌 Tomcat、Jetty 等使得程序的运行不在依赖传统的应用服务器。这一点在云原生时代意义尤其重大。

Spring 官方对 Spring Boot 特色定义如下：

- 创建独立的 Spring 应用程序
- 直接嵌入 Tomcat, Jetty 或 Undertow（无需部署 WAR 文件）
- 提供自以为是的“starter”依赖项，以简化构建配置
- 尽可能自动配置 Spring 和三方类库
- 提供可用于生产的功能，例如指标，运行状况检查和外部化配置
- 完全没有代码生成，也不需要 XML 配置

Spring Cloud

Spring Cloud 是什么，没有比官方的定义更能说明问题了：

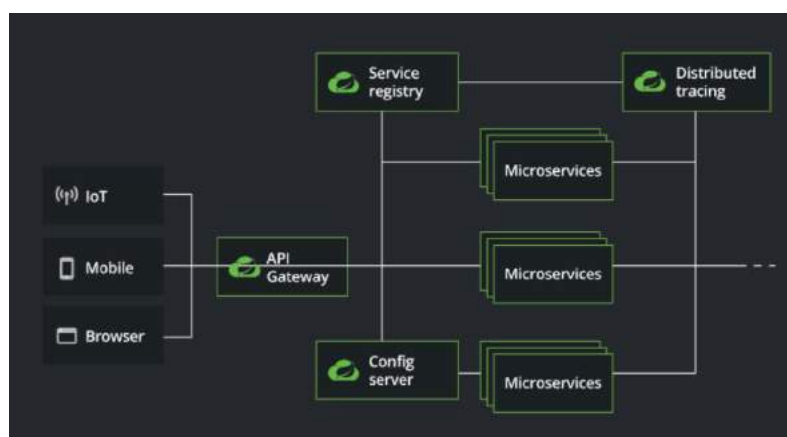
Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

这里面提到几个关键词：

- 分布式系统中的常见模式
- 任何分布式环境

“分布式系统中的常见模式”给了 Spring Cloud 一个清晰的定位，即“模式”。也就是说 Spring Cloud 是针对分布式系统开发所做的通用抽象，是标准模式的实现。

这个定义非常抽象，看完之后并不能知道 Spring Cloud 具体包含什么功能。再来看一下 Spring 官方给出的一个 High Light 的架构图，就可以对这套模式有更清晰的认识：

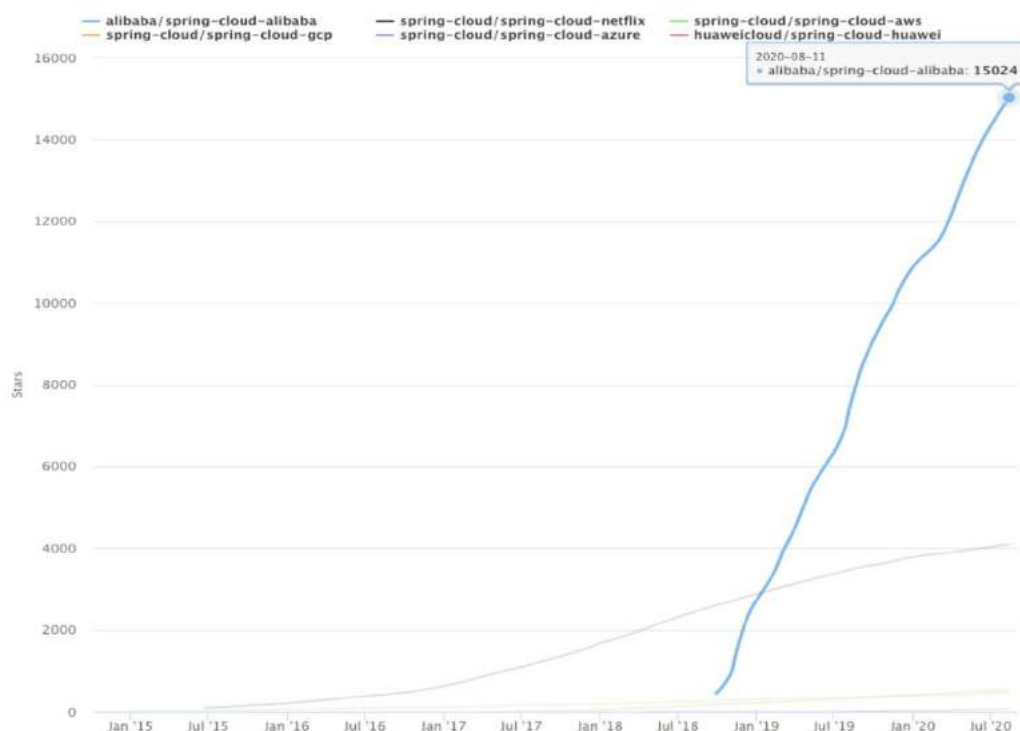


可以看到这个图中间就是各个 Microservice，也就是我们的这个微服务的实现，周边周围的话就是去围绕这个微服务来去做各种辅助的信息事情。例如分布式追踪、服务注册、配置服务等，都绕微服务运行时所依赖的必不可少的的支持性功能。我们可以得出这样一个结论：Spring Cloud 是以微服务为核心的分布式系统的一个构建标准。

Spring Cloud Alibaba

既然说 Spring Cloud 是标准，那么自然少不了针对标准的实现。参与这个标准实现的公司有很多，例如：Google 的 Spring Cloud GCP，Netflix 的 Spring Cloud Netflix，Microsoft 的 Spring Cloud Azure 等等。当然还有我们阿里巴巴的 Spring Cloud Alibaba。

Spring Cloud Alibaba 从 19 年初开始提交代码就获得了业界的广泛关注，从下面这张图中可以看到，在很短的时间之内，Spring Cloud Alibaba 就成为了 Spring Cloud 家族中最受关注的框架：



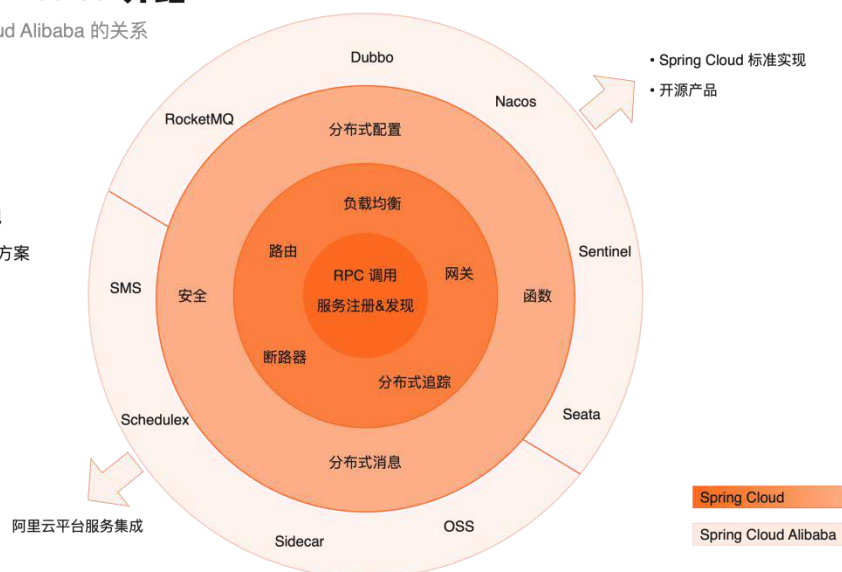
下面这张图很好的说明了 Spring Cloud Alibaba 的组成以及与 Spring Cloud 的关系：

Spring Cloud Alibaba 介绍

Spring Cloud & Spring Cloud Alibaba 的关系

Spring Cloud Alibaba 是：

- 对 Spring Cloud 的标准实现
- 以微服务为核心的整体解决方案
- 开源与平台服务分开维护



图中深色的部分，其实它就是 Spring Cloud 标准，一共有 3 层。中间颜色最深的部分就是及整个微服务最核心的内容，包括了“RPC 调用”以及“服务注册与发现”。第二层，也就是围绕着核心的这一圈，是一些辅助微服务更好的工作功能，包括了负载均衡、路由、网关、断路器，还有分就是追踪等等这些内容。再外层的话，主要是些分布式云环境里通用能力。

最外面这一圈，是 Spring Cloud Alibaba 对 Spring Cloud 的实现。右上部分是对 Spring Cloud 标准的实现。例如，我们通过 Dubbo 实现了 RPC 调用功能，通过 Nacos 实现了“服务注册与发现”、“分布式配置”，通过 Sentinel 实现了断路器等，这里就不一一列举了。左下部分是我们 Spring Cloud Alibaba 对阿里云各种服务的集成。可能很多同学会有这样的问题：为什么要加上这一部分呢？此时回头审视一下

Spring Cloud，它仅仅是一个微服务的一个框架。但是在实际生产过程中，单独使用微服务框架其实并不足以支撑我们去构建一个完整的系统。所以这部分是用阿里帮助开发者完成微服务以外的云产品集成的功能。

为什么要分成两个部分呢，这也是为了打消大家对于使用了 Spring Cloud Alibaba 以后就会被平台绑定的担忧。虽然在品牌商都叫做 SpringCloudAlibaba，但是在代码上，我们采用两个独立的项目维护，分别是 **Spring Cloud Alibaba** 和 **Aliyun Spring Boot**。

目前，Spring Cloud Alibaba 包含如下组件：

开源部分

- Sentinel: 把流量作为切入点, 从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。
- Nacos: 一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。
- RocketMQ: 一款开源的分布式消息系统, 基于高可用分布式集群技术, 提供低延时的、高可靠的消息发布与订阅服务。
- Dubbo: Apache Dubbo 是一款高性能 Java RPC 框架。
- Seata: 阿里巴巴开源产品, 一个易于使用的高性能微服务分布式事务解决方案。

平台服务部分

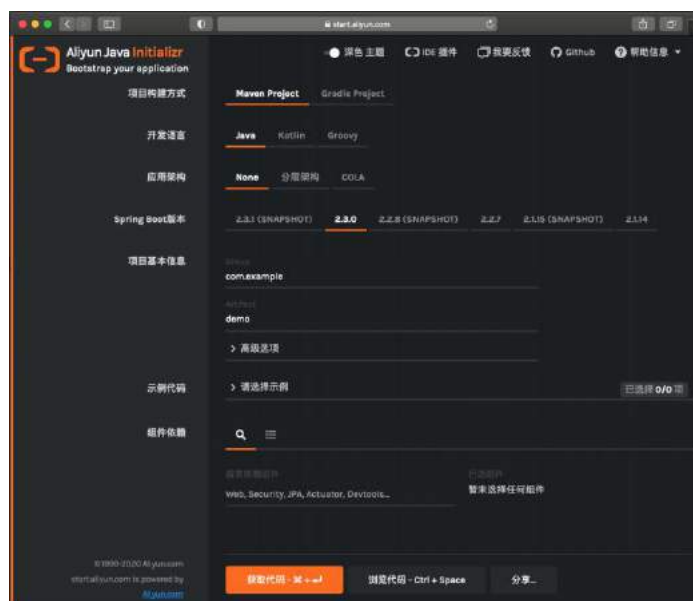
- Alibaba Cloud OSS: 阿里云对象存储服务 (Object Storage Service, 简称 OSS), 是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- Alibaba Cloud SchedulerX: 阿里中间件团队开发的一款分布式任务调度产品, 提供秒级、精准、高可靠、高可用的定时 (基于 Cron 表达式) 任务调度服务。
- Alibaba Cloud SMS: 覆盖全球的短信服务, 友好、高效、智能的互联化通讯能力, 帮助企业迅速搭建客户触达通道。

工具篇

Java 工程脚手架

Java 工程脚手架, 用于帮助开发者快速生成工程骨架。解决开发者在创建工程时的组件引入、解决版本依赖、基础配置、查询样例代码等繁琐问题。只需要简单的点点鼠标, 就可以生成一套标准工程骨架。

脚手架的访问地址是 <https://start.aliyun.com/bootstrap.html>, 打开后页面见下图:

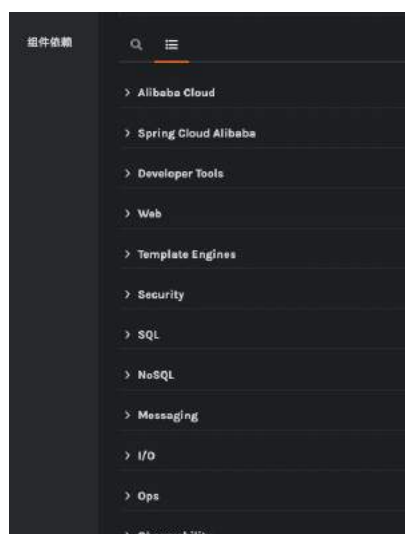


编译框架、坐标&名称、其他基础信息等，根据实际情况按需填写。当然，很多参数默认值就可以满足大部分需求。开发者重点关注的是下面 3 个部分：

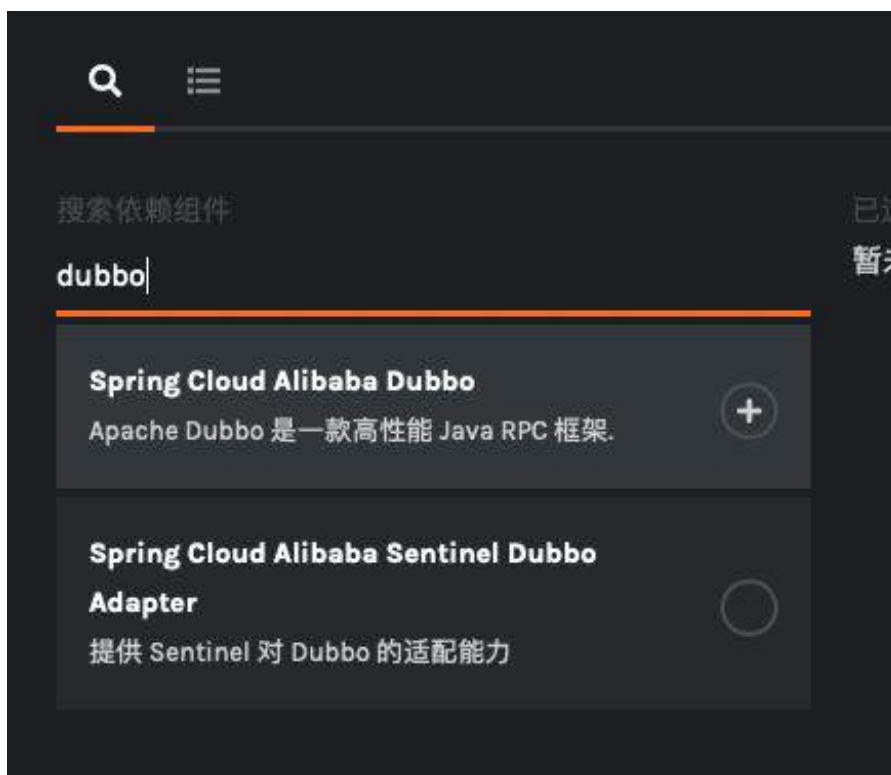
组件依赖

很少有开发者会使用语言最原始的 sdk 来实现所有功能。通常来说，大家都会使用各种技术产品的高级封装来实现相关的技术特性。这里就需要做 2 件事情：引入对应组件的 sdk、在应用中配置组件。而通过 Java 工程脚手架就可以很轻易的完成这些工作。

Java 工程脚手架中提供了 2 种寻找组件的方式：根据分类浏览&关键字搜索
这里根据组件分类寻找需要使用的组件：



也可以根据组件的关键字直接使用搜索功能寻找需要使用的组件：



无论通过哪种方式，都可以通过组件右侧的加好实现组件的选择。

应用架构

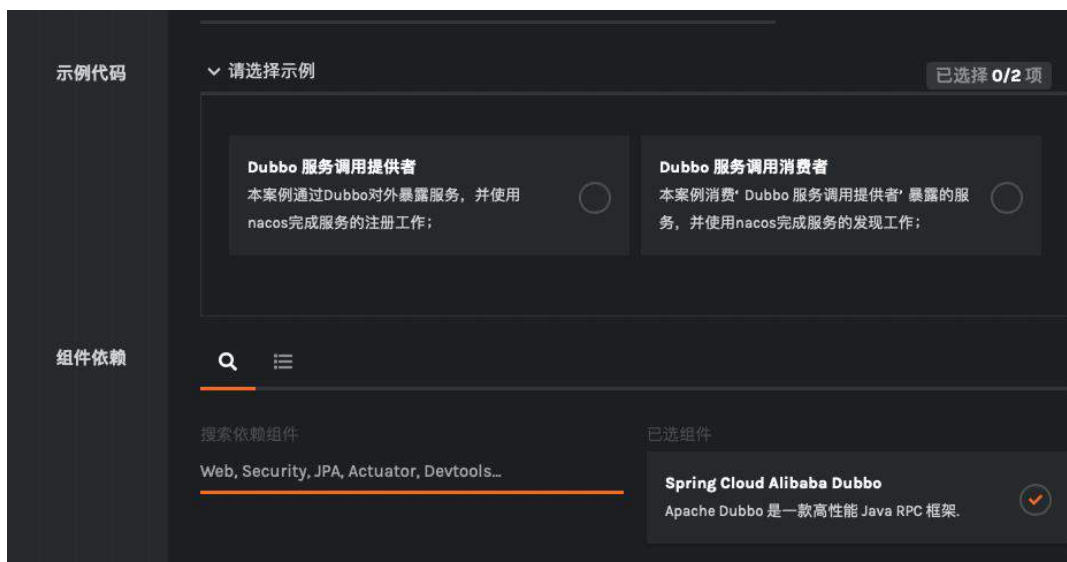
在生成的工程里，代码需要根据其逻辑职责进行分层，从而获得更好的代码组织与管理效果。在这里提供了 3 种应用架构供开发者选用。

- None：不做任何代码分层。
- 分层架构：标准的 4 层架构。分为 WEB 层、控制器层、服务层、持久化层。本架构参考了《Java 编码规约》的[应用分层](#)章节。
- COLA：领域驱动设计（DDD）的实现框架之一。采用了标准的六边形架构设计，并且辅之以一套灵活的扩展体系，可以有效提升复杂业务系统开发的效率。具体参考[COLA 的 Github 工程](#)。

示例代码

我们为组件准备了很多使用方法的参考样例，这样开发者就不需要选择外组件以后，再去别的搜索引擎寻找相关组件的使用方法了。

未选择任何组件时，是不会给出任何示例代码的。示例代码是在选择了组件依赖以后，才会出现于用户选择的组件相关的示例，如下图：



由于很多案例自身也依赖其他组件，所以在选择了某个案例以后，会多出一些案例，同时依赖的组件也会增多。

本次课程使用到的案例，都可以在这里寻找到。

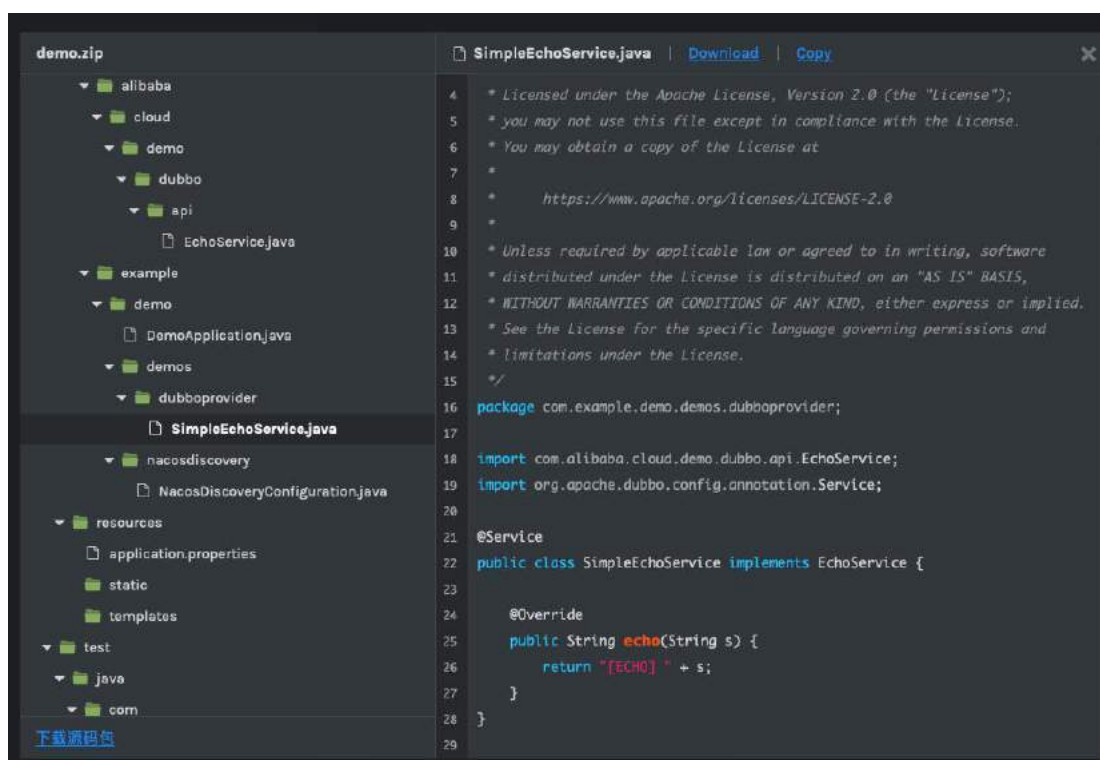
生成代码

仅仅完成项目配置是不够的，最终开发者需要的是项目的代码。so, show me the code

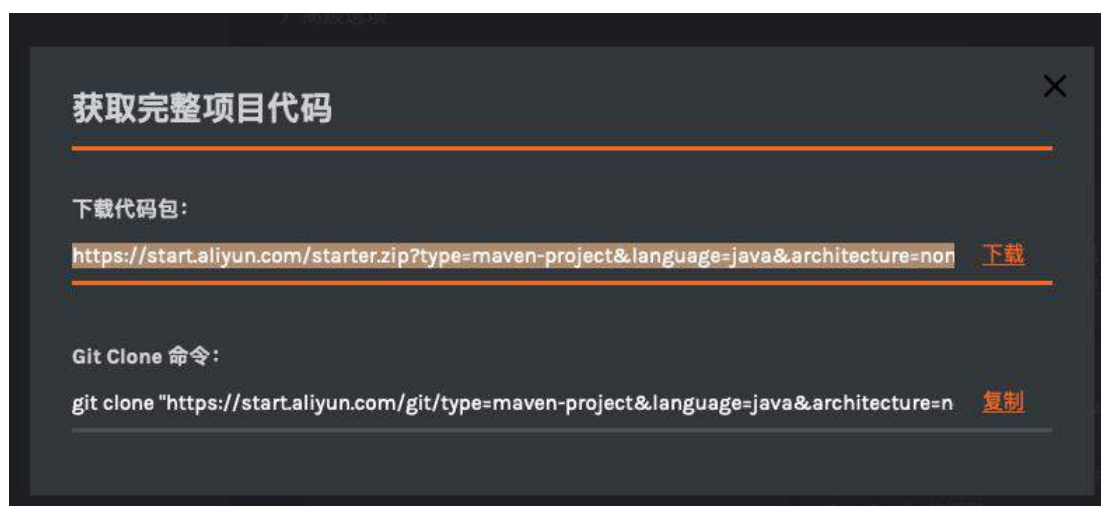
无论是出于查阅组件用法的目的，还是出于需要工程完整代码的目的，脚手架都可以很好的支持：



如果仅仅需要查阅代码，而不是下载完整工程，可以直接通过点击“浏览代码”来实现。点击该按钮以后，会打开一个包含了完整代码树以及允许查看每个文件的内容窗口：



如果需要获取所有代码内容，则可以通过点击“获取代码”来实现：

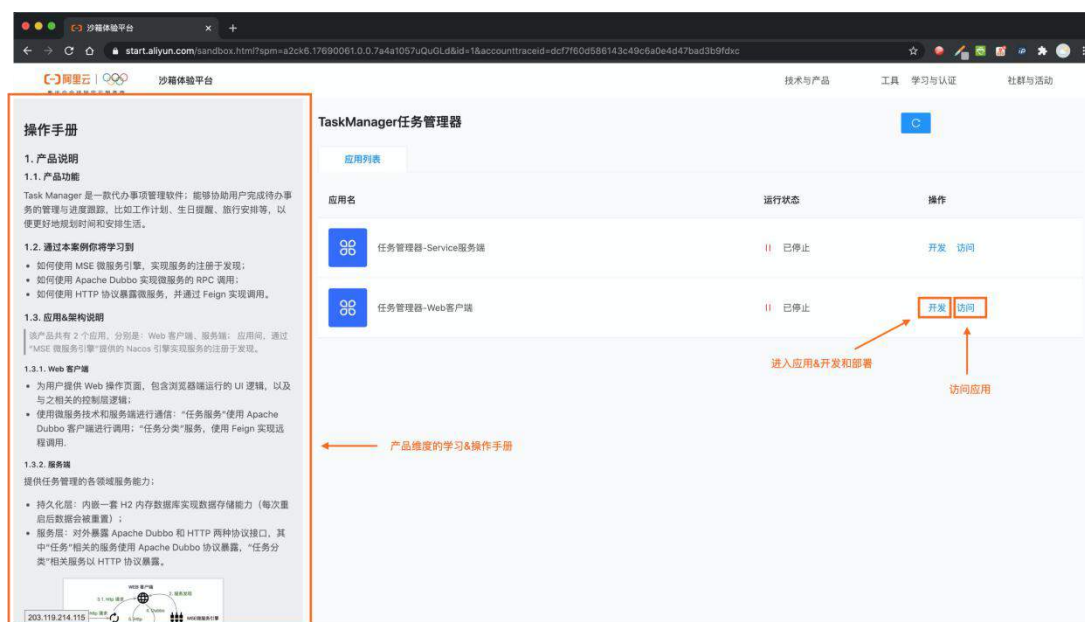


这里提供了 2 种获取代码的途径：直接下载代码包 & 通过 git 命令 clone 工程。如果选择使用 Git 命令来 Clone 工程，需要注意一下，这个仓库地址只能下载不能上传哦。

Sandbox 沙箱环境

Sandbox 沙箱环境，为开发者带来一套快速上手、免除任何环境依赖、免费、便捷的开发&运行环境。允许开发者在上面查看、修改、部署示例代码，并且由平台提供相关运行资源。

下面来看一下产品的界面：



左边是产品的手册&说明部分。这里会包含说当前项目的功能说明、应用架构，以及如何部署和访问这些应用的操作步骤等。一些项目中使用到的技术点以及这些相关知识，也都会在这里呈现给用户。这部分文档的目的，就是方便用户去学习和理解当前的案例。

右边的部分是应用列表。一个完整的产品，可能需要多个应用协同才能工作，这里就是用来陈列相关的应用列表，同时也是针对这些应用的操作入口。

图片中的案例是一个任务管理器产品，功能相对简单。但是麻雀虽小五脏俱全。这个产品包含两个应用：

- 服务端，的包含了这个任务管理器的所有业务逻辑，以及下层的持久化能力等。
- WEB 客户端，包含了所有前端页面逻辑、与前端通信的控制器层。

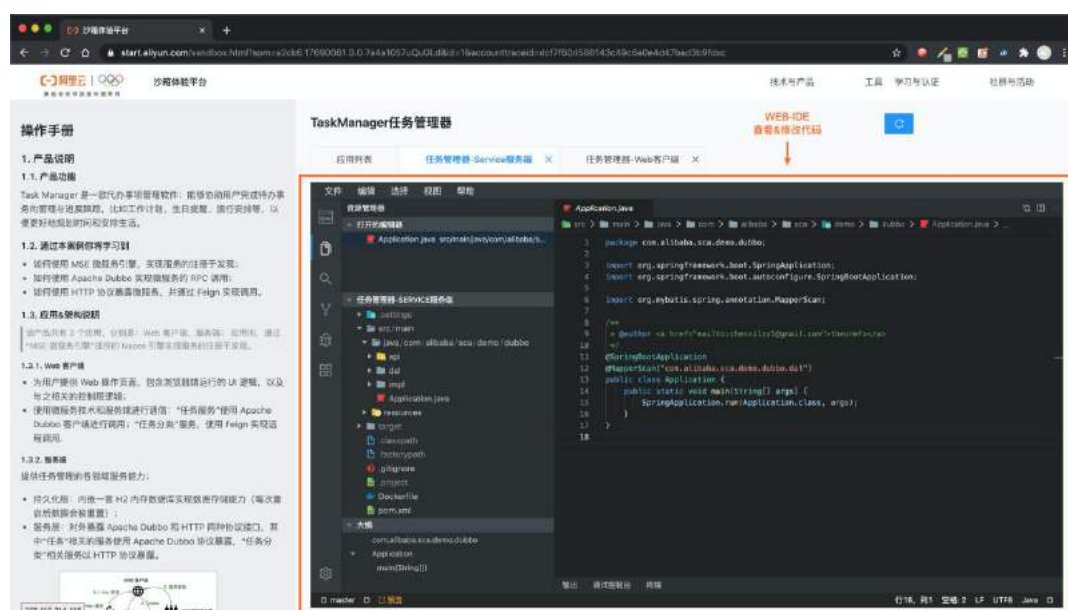
这两个应用通过一个注册中心来实现服务的注册&发现。最终实现一个完整的任务管理器产品。

这些东西都会放到这里面。这就是一个非常典型的一个一个应用拆分的一个方式，对不对？这里的话，其实业务应用上它有两个行动点，一个是开发和访问点，开发之后就会打开一个 IDE。

这里面就会有整个工程的代码。这些其实是我们预计好的，大家打开就能直接看到。如果它完全部署以后点了部署按钮，我也会直接访问到这个应用。

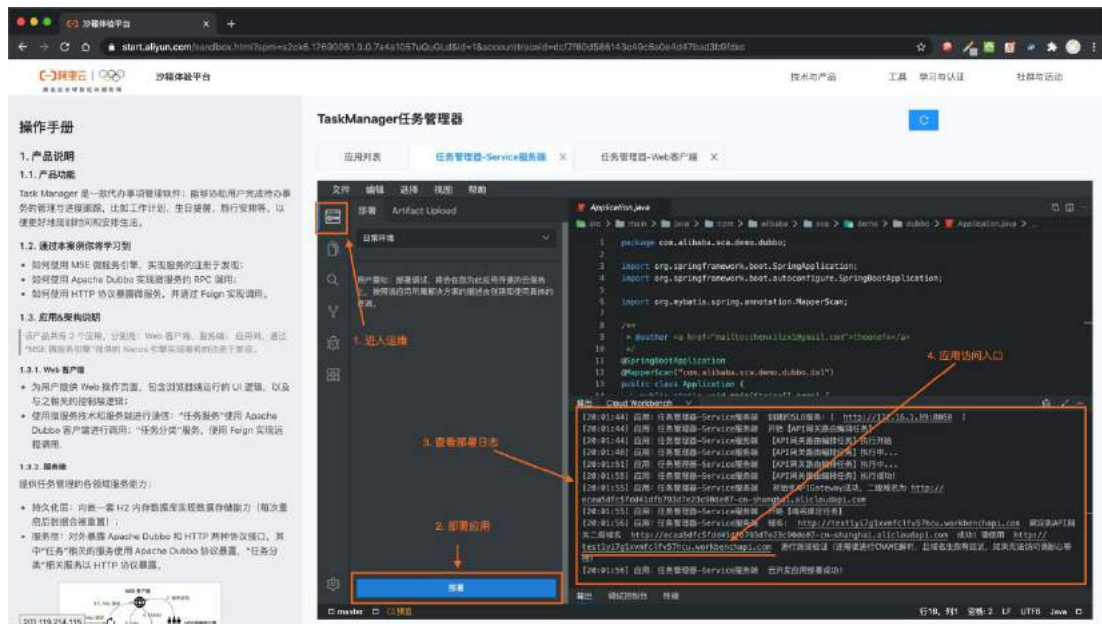
其暴露出来的这个访问接口，我们点开发之后会看到这样一个情况。对，这个就是我们的外包 id 。

开发者可以点击“开发”按钮，打开一个 WEB-IDE 来查看和修改对应应用的代码：



这个 WEB-IDE 和开发者日常使用的 IDE 是一样的，都是左侧代码树，右侧代码编辑器的标准布局。即使是不熟悉这个产品的用户，也可以非常快的上手，甚至不需要学习过程。

如果需要部署这个应用，只需要在“运维”功能下，点击“部署”按钮，此时只需要等待部署完成即可。在部署过程会有很多的日志输出，都可以通过“输出”窗体浏览：



部署完成以后，会向 WEB-IDE 返回一个访问地址，开发者只需要点击这和地址就可以访问这个应用。下图是实际的访问效果。可以看到，两个应用，一个是任务管理器的 web 操作页面、一个是后台数据库管理页面：

Theme example

这是一个简易任务管理器。你可以在这里查看、管理你的代办事项。

待办事项

全部

进行中

已完成

#	分类	创建时间	标题
1	生活	2020-08-06 02:53:35	陈女网友看电影
2	工作	2020-08-06 02:53:35	完成项目B的汇报PPT
3	默认	2020-08-06 02:53:35	读书：《一个演员的自我修养》
4	工作	2020-08-06 02:53:35	
5	生活	2020-08-06 02:53:35	

新建任务

任务标题

请在这里输入任务标题

任务分类

请选择任务分类

jdbc:h2~test

CATEGORIES

TASKS

INFORMATION_SCHEMA

Sequences

Users

H2 1.4.200 (2019-10-14)

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT * FROM TASKS

ID	GMT_CREATE	GMT_MODIFIED	TITLE	CONTENT	CATEGORY_ID	STATUS
1	2020-08-06 14:53:35.493	2020-08-06 14:53:35.493	陈女网友看电影	1周年纪念，千万不能忘	2	new
2	2020-08-06 14:53:35.493	2020-08-06 14:53:35.493	完成项目B的汇报PPT	快交付了，B项目客户很看重	3	new
3	2020-08-06 14:53:35.493	2020-08-06 14:53:35.493	电话拜访客户A	A客户可是VIP啊	3	completed
4	2020-08-06 14:53:35.493	2020-08-06 14:53:35.493	回家买蛋糕	...	2	completed
5	2020-08-06 14:53:35.493	2020-08-06 14:53:35.493	读书：《一个演员的自我修养》	好好学习	1	new

5 rows, 22 ms

Edit

通过上面的步骤，开发者可以将案例快速部署起来。先部署试用，然后去学习和修改代码，最后再部署验证。通过这样的循环，可以让开发者很快学习和理解案例的功能和相关技术点。

分布式配置

1. 简介

Nacos 提供用于存储配置和其他元数据的 key/value 存储，为分布式系统中的外部化配置提供服务器端和客户端支持。使用 Spring Cloud Alibaba Nacos Config，您可以在 Nacos Server 集中管理你 Spring Cloud 应用的外部属性配置。

Spring Cloud Alibaba Nacos Config 是 Config Server 和 Client 的替代方案，客户端和服务端的概念与 Spring Environment 和 PropertySource 有着一致的抽象，在特殊的 bootstrap 阶段，配置被加载到 Spring 环境中。当应用程序通过部署管道从开发到测试再到生产时，您可以管理这些环境之间的配置，并确保应用程序具有迁移时需要运行的所有内容。Nacos 的获取和启动方式可以参考 [Nacos 官网](#)。

注：如果读者是阿里云商业化组件 ANS 或 ACM 用户，请使用 Nacos Config 代替对应的组件。

2. 学习目标

- 使用 Nacos Config 作为 Spring Cloud 分布式配置
- 使用 Nacos Config 实现 Bean 动态刷新
- 了解 Nacos Config 高级配置

3. 详细内容

- 快速上手：使用 Nacos Config 作为外部化配置源
- 多文件扩展名支持：以 YAML 文件扩展名为例，讨论 Nacos Config 多文件扩展名支持
- 动态配置更新：演示 @RefreshScope 特性，实现 Bean 动态刷新
- 自定义扩展：自定义 namespace、Group 以及 Data Id 的配置扩展
- 运维特性：演示 Nacos Config 高级外部化配置以及 Endpoint 内部细节

4. 快速上手

4.1 如何引入 Nacos Config 支持分布式配置

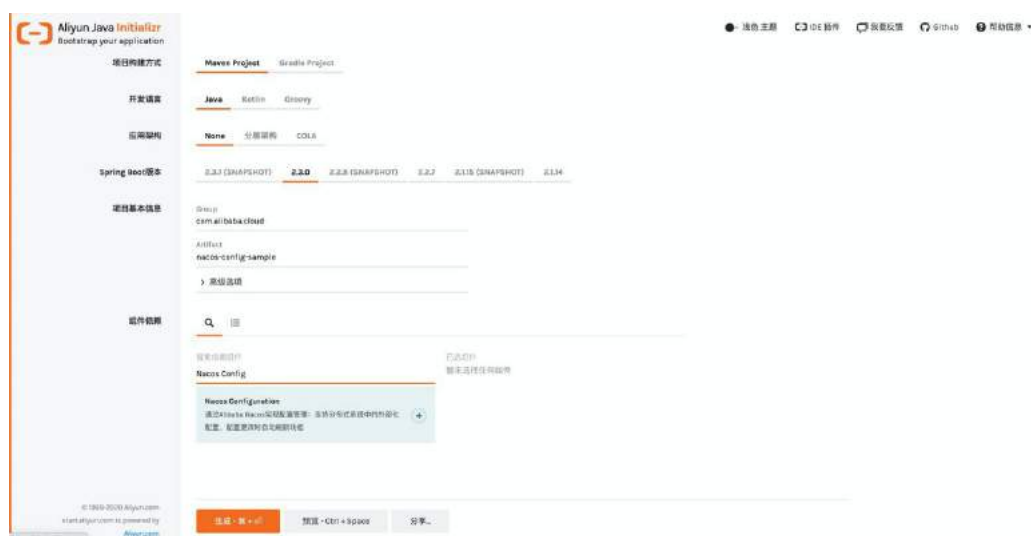
Nacos Config 引入的方式同样也有两种，即 Aliyun Java Initializr 引入和 Maven pom.xml 依赖。官方推荐使用 Aliyun Java Initializr 方式引入 Nacos Discovery，以便简化组件之间的依赖关系。

4.1.1 [简单] 通过 Aliyun Java Initializr 创建工程并引入 Nacos Config（推荐）

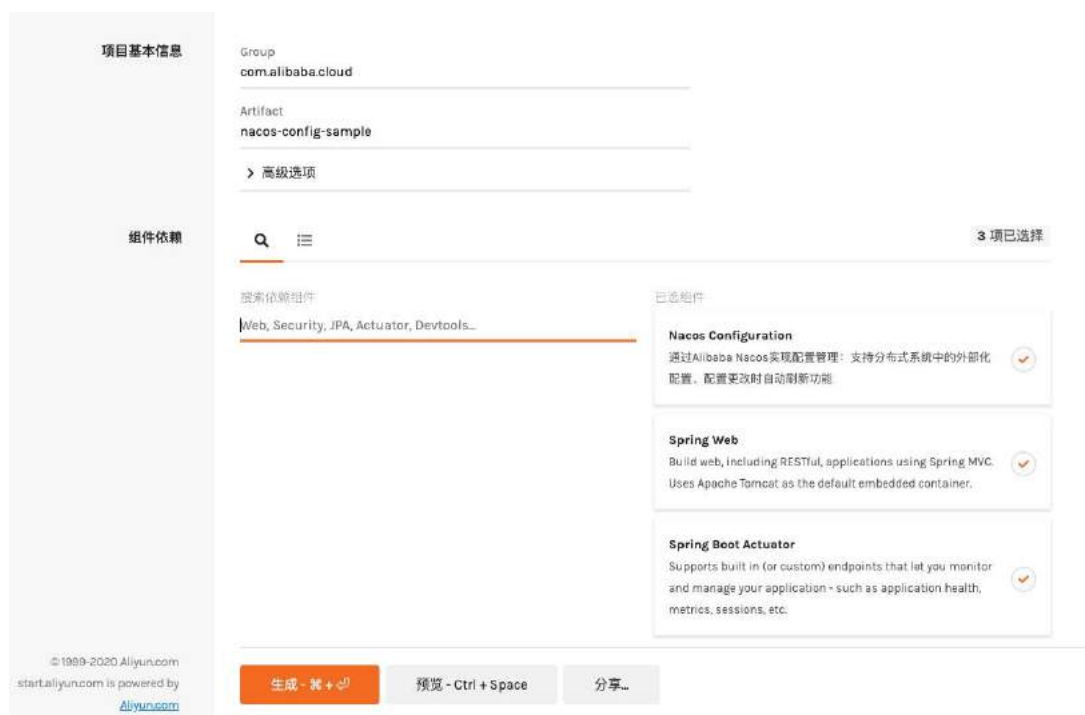
由于 Spring Cloud 组件的版本和依赖较为复杂，推荐读者使用 Aliyun Java Initializr 构建应用工程。

读者选择偏好的 Web 浏览器访问 Aliyun Java Initializr，其资源网址为：
<https://start.aliyun.com/bootstrap.html>

下文以 Google Chrome 浏览器为例，当网页加载后，首先，在 "项目基本信息" 部分输入 Group：“com.alibaba.cloud” 以及 Artifact：“nacos-config-sample”。然后，“组件依赖” 输入框搜索：“Nacos Config”，选择 "Nacos Configuration"，如下所示：



同上组件操作，增加 “Spring Web” 和 “Spring Boot Actuator” 组件：



组件选择后，点击“生成”高亮按钮。随后，平台将生成一个名为“nacos-config-sample.zip”的压缩文件，将其保存到本地目录，并解压该文件，工程目录将随之生成。打开目录下的 pom.xml 文件，不难发现 Nacos starter 声明其中（以下 XML 内容均来自于项目根路径中的 pom.xml 文件）：

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

不过该 starter 并未指定版本，具体的版本声明在 com.alibaba.cloud:spring-cloud-alibaba-dependencies 部分：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>${spring-cloud-alibaba.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${spring-boot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

其中，`${spring-cloud-alibaba.version}` 和 `${spring-boot.version}` 分别为 Spring Cloud Alibaba 和 Spring Boot 组件依赖的版本，它们的版本定义在 `<properties>` 元素中，即 2.2.1.RELEASE 和 2.3.0.RELEASE：

```
<properties>
  <java.version>1.8</java.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <spring-boot.version>2.3.0.RELEASE</spring-boot.version>
  <spring-cloud-alibaba.version>2.2.1.RELEASE</spring-cloud-alibaba.version>
</properties>
```

如果读者非常熟悉 Maven 依赖管理的配置方式，可以考虑 Maven pom.xml 依赖 Nacos Config。

4.1.2 [高级] 通过 Maven pom.xml 依赖 Nacos Config

如果要在您的项目中使用 Nacos 来实现服务注册/发现，使用 group ID 为 com.alibaba.cloud 和 artifact ID 为 spring-cloud-starter-alibaba-nacos-config 的 starter。

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId></dependency>
```

该声明方式同样需要声明 com.alibaba.cloud:spring-cloud-alibaba-dependencies，内容与上小节相同，在此不再赘述。下一节将讨论如何使用 NacosConfig 支持分布式配置。

4.1.3 [偷懒] 直接在沙箱里查看应用代码

点击 [链接](#)，直接访问沙箱环境，这里会有为你准备好的案例代码^_^。

4.2 使用 Nacos Config 实现分布式配置

使用 Nacos Config 实现分布式配置与 Spring Cloud Consul 和 Spring Cloud Zookeeper 的方式非常类似，仅需添加相关外部化配置即可工作。换言之，Nacos Config 同样不会侵入应用代码，方便应用整合和迁移，如果读者熟悉 Spring Cloud Consul 或 Spring Cloud Zookeeper 使用方式的话，通常需要将 Consul 或 Zookeeper 服务进程预先部署，Nacos Config 也如此。

4.2.1 启动 Nacos 服务器

我们为开发者提供了一套免费的 Nacos Server：进入 <http://139.196.203.133:8848/nacos/> 查看控制台(账号名/密码为 nacos-configuration/nacos-configuration)，选择 "配置管理/配置列表"：



由于服务是公共免费的，为了做好隔离，所以分布式配置的功能，请选择在 `sandbox-configuration` 的命名空间下操作。

具体启动方式参考 [Nacos 官网](#)。

关于更多的 Nacos Server 版本，可以从 [release 页面](#) 下载最新的版本。

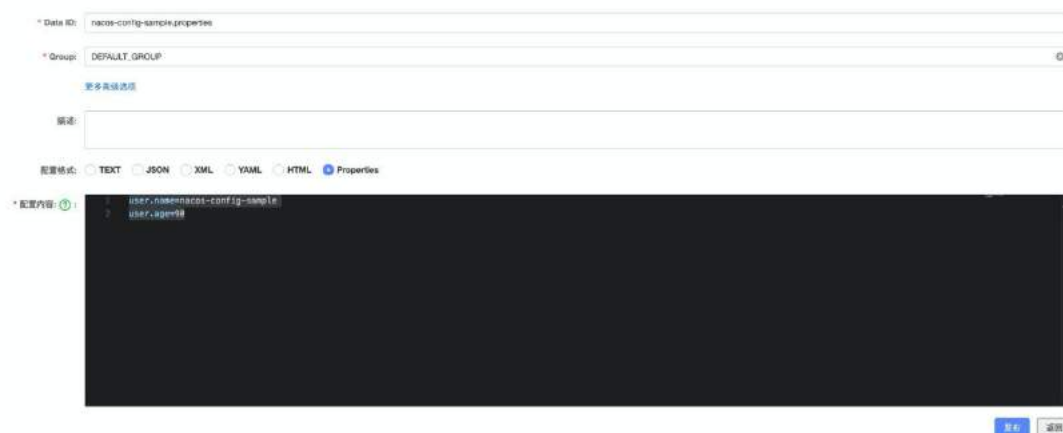
4.2.2 添加 Nacos 配置

点击“配置列表”页面右侧的“+”号（红色箭头所指）：



浏览器跳转新页面，并填充内容如下：

新建配置



其中，Data ID 由应用名（nacos-config-sample）+ 文件后缀名(.properties)组成，点击“发布”按钮（红色箭头所指），配置内容为：

```
user.name=nacos-config-sampleuser.age=90
```

发布成功后，控制台会出现提示弹出框。

特别提醒

图片中的 Data ID 中的应用名，使用的是 nacos-config-sample。如果使用我们提供的 Nacos Server，由于多个用户同时使用，可能存在应用名冲突的问题。建议修改为跟别人都不重复的应用名：

```
spring.application.name=xxxxxx
```

替换上面的 xxxxx 部分。

4.2.3 配置应用 Nacos Config Server 地址

回到应用 nacos-config-sample 工程，在 resources 目录下新建名为“application.properties”文件，并配置以下内容：

```
spring.cloud.nacos.config.server-addr=139.196.203.133:8848
spring.cloud.nacos.config.username=nacos-configuration
spring.cloud.nacos.config.password=nacos-configuration
spring.cloud.nacos.config.namespace=sandbox-configuration
```


注意，Nacos Server 地址必须配置在 application.properties 文件。

注意当你使用域名的方式来访问 Nacos 时，spring.cloud.nacos.config.server-addr 配置的方式为域名:port。例如 Nacos 的域名为 abc.com.nacos，监听的端口为 80，则 spring.cloud.nacos.config.server-addr=abc.com.nacos:80。注意 80 端口不能省略。

4.2.4 添加读取 Nacos Config 实现

修改 nacos-config-sample 引导类，如下所示：

```
@SpringBootApplicationpublic class NacosConfigSampleApplication {

    @Value("${user.name}")
    private String userName;

    @Value("${user.age}")
    private int userAge;

    @PostConstruct
    public void init() {
        System.out.printf("[init] user name : %s , age : %d\n", userName, userAge);
    }

    public static void main(String[] args) {
        SpringApplication.run(NacosConfigSampleApplication.class, args);
    }
}
```

4.2.5 启动 Nacos Config 应用

运行 nacos-config-sample 引导类 NacosConfigSampleApplication，观察控制台结果（截取关键日志信息）：

```
[init] user name : nacos-config-sample , age : 90
```

如果在沙箱内启动应用，由于暂时还无法看到运行时的日志，可以通过 WEB 控制器的方式查看参数，具体操作见 5.1 节。

5. 使用 Nacos Config 实现 Bean 动态刷新

Nacos Config 支持标准 Spring Cloud `@RefreshScope` 特性，即应用订阅某个 Nacos 配置后，当配置内容变化时，Refresh Scope Beans 中的绑定配置的属性将有条件的更新。所谓的条件是指 Bean 必须：

- 必须条件：Bean 的声明类必须标注 `@RefreshScope`
- 二选一条件：
 - 属性（非 static 字段）标注 `@Value`
 - `@ConfigurationProperties` Bean

除此之外，Nacos Config 也引入了 Nacos Client 底层数据变化监听接口，即 `com.alibaba.nacos.api.config.listener.Listener`。下面的内容将分别讨论这三种不同的使用场景。

Nacos Client: Nacos 客户端 API，也是 Nacos Config 底层依赖

5.1 使用 Nacos Config 实现 Bean `@Value` 属性动态刷新

基于应用 `nacos-config-sample` 修改，将引导类 `NacosConfigSampleApplication` 标注 `@RefreshScope` 和 `@RestController`，使得该类变为 Spring MVC REST 控制器，同时具备动态刷新能力，具体代码如下：

```
@SpringBootApplication
@RestController
@RefreshScope public class NacosConfigSampleApplication {

    @Value("${user.name}")
    private String userName;

    @Value("${user.age}")
    private int userAge;

    @PostConstruct
    public void init() {
        System.out.printf("[init] user name : %s , age : %d\n", userName, userAge);
    }

    @RequestMapping("/user")
    public String user() {
        return String.format("[HTTP] user name : %s , age : %d", userName, userAge);
    }

    public static void main(String[] args) {
        SpringApplication.run(NacosConfigSampleApplication.class, args);
    }
}
```

重启引导类 `NacosConfigSampleApplication`，控制台输出如故：

```
[init] user name : nacos-config-sample , age : 90
```

再通过命令行访问 REST 资源 `/user`：

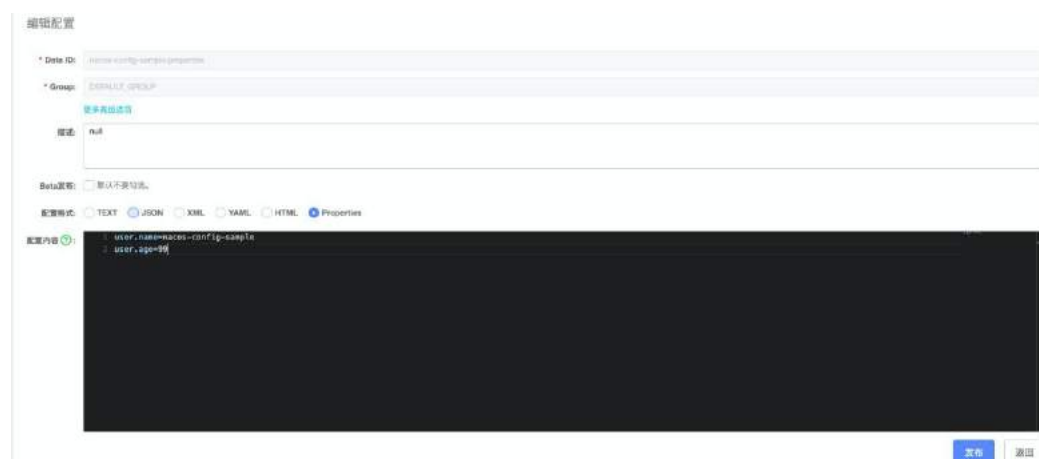
```
% curl http://127.0.0.1:8080/user
[HTTP] user name : nacos-config-sample , age : 90
```

如果使用沙箱环境，请直接点击应用列表的访问按钮，并在打开的浏览器窗口的地址栏中追加 `/user`，如下图：

本文中，其他的基于 http 访问的步骤类似，后面不在赘述。

本次请求结果中的 user name 和 age 数据与应用启动时的一致，因为此时 Nacos Server 中的配置数据没变化。

随后，通过 Nacos 控制台调整 nacos-config-sample.properties 配置，将 user.age 从 90 变更为 99：



点击“发布”按钮，观察应用日志变化（部分内容被省略）：

```
c.a.n.client.config.impl.ClientWorker : [fixed-127.0.0.1_8848] [data-received]
dataId=nacos-config-sample.properties, group=DEFAULT_GROUP, tenant=null,
md5=4a8cb29154adb9a0e897e071e1ec8d3c, content=user.name=nacos-config-sample
user.age=99, type=properties
o.s.boot.SpringApplication : Started application in 0.208 seconds (JVM running for
290.765)
o.s.c.e.event.RefreshEventListener : Refresh keys changed: [user.age]
```

- 第 1 和 2 行代码是由 Nacos Client 输出，通知开发者具体的内容变化，不难发现，这里没有输出完整的配置内容，仅为变更部分，即配置 user.age。
- 第 3 行日志似乎让 SpringApplication 重启了，不过消耗时间较短，这里暂不解释，后文将会具体讨论，只要知道这与 Bootstrap 应用上下文相关即可。
- 最后一行日志是由 Spring Cloud 框架输出，提示开发人员具体变更的 Spring 配置 Property，可能会有多个，不过本例仅修改一处，所以显示单个。

接下来，重新访问 REST 资源 /user：

```
% curl http://127.0.0.1:8080/user
[HTTP] user name : nacos-config-sample , age : 99
```

终端日志显示了这次配置变更同步到了 `@Value("${user.age}")` 属性 `userAge` 的内容。除此之外，应用控制台也输出了以下内容：

```
[init] user name : nacos-config-sample , age : 99
```

而该日志是由 `init()` 方法输出，那么是否说明该方法被框架调用了呢？答案是肯定的。既然 `@PostConstruct` 方法执行了，那么 `@PreDestroy` 方法会不会被调用呢？不妨增加 Spring Bean 销毁回调方法：

```
@SpringBootApplication
@RestController
@RefreshScope public class NacosConfigSampleApplication {

    @Value("${user.name}")
    private String userName;

    @Value("${user.age}")
    private int userAge;

    @PostConstruct
    public void init() {
        System.out.printf("[init] user name : %s , age : %d\n", userName, userAge);
    }

    @PreDestroy
    public void destroy() {
        System.out.printf("[destroy] user name : %s , age : %d\n", userName, userAge);
    }

    ...
}
```

再次重启引导类 `NacosConfigSampleApplication`，初始化日志仍旧输出：

```
[init] user name : nacos-config-sample , age : 99
```

将配置 user.age 内容从 99 调整为 18，观察控制台日志变化：

```
c.a.n.client.config.impl.ClientWorker    : [fixed-127.0.0.1_8848] [data-received]
dataId=nacos-config-sample.properties, group=DEFAULT_GROUP, tenant=null,
md5=e25e486af432c403a16d5fc8a5aa4ab2, content=user.name=nacos-config-sample
user.age=18, type=properties
o.s.boot.SpringApplication              : Started application in 0.208 seconds (JVM running for
144.467)
[destroy] user name : nacos-config-sample , age : 99
o.s.c.e.event.RefreshEventListener      : Refresh keys changed: [user.age]
```

相较于前一个版本，日志插入了 destroy()方法输出内容，并且 Bean 属性 userAge 仍旧是变更前数据 99。随后，再次访问 REST 资源 /user，其中终端日志：

```
% curl http://127.0.0.1:8080/user
[HTTP] user name : nacos-config-sample , age : 18
```

应用控制台日志：

```
[init] user name : nacos-config-sample , age : 18
```

两者与前一版本并无差异，不过新版本给出了一个现象，即当 Nacos Config 接收到服务端配置变更时，对应的 @RefreshScope Bean 生命周期回调方法会被调用，并且是先销毁，然后由重新初始化。本例如此设计，无非想提醒读者，要意识到 NacosConfig 配置变更对 @RefreshScope Bean 生命周期回调方法的影响，避免出现重复初始化等操作。

注：Nacos Config 配置变更调用了 Spring Cloud API ContextRefresher，该 API 会执行以上行为。同理，执行 Spring Cloud Acuator Endpoint refresh 也会使用 ContextRefresher。

通过上述讨论，相信读者已对 Nacos 配置变更操作相当的熟悉，后文将不再赘述相关配置。接下来继续讨论 @ConfigurationProperties Bean 的场景。

5.2 使用 Nacos Config 实现@ConfigurationProperties Bean 属性动态刷新

在应用 nacos-config-sample 新增 User 类，并标注 @RefreshScope 和 @ConfigurationProperties，代码如下：

```
@RefreshScope
@ConfigurationProperties(prefix = "user")public class User {
    private String name;
    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;
    }
    @Override
    public String toString() {
        return "User{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}
```

根据 @ConfigurationProperties 的定义，User 类的属性绑定到了配置属性前缀 user。下一步，调整引导类，代码如下：


```
@SpringBootApplication
@RestController
@RefreshScope
@EnableConfigurationProperties(User.class)
public class NacosConfigSampleApplication {

    @Value("${user.name}")
    private String userName;

    @Value("${user.age}")
    private int userAge;

    @Autowired
    private User user;

    @PostConstruct
    public void init() {
        System.out.printf("[init] user name : %s , age : %d\n", userName, userAge);
    }

    @PreDestroy
    public void destroy() {
        System.out.printf("[destroy] user name : %s , age : %d\n", userName, userAge);
    }

    @RequestMapping("/user")
    public String user() {
        return "[HTTP] " + user;
    }

    public static void main(String[] args) {
        SpringApplication.run(NacosConfigSampleApplication.class, args);
    }
}
```

较前一个版本 `NacosConfigSampleApplication` 实现，主要改动点：

- 激活 `@ConfigurationProperties` Bean `@EnableConfigurationProperties(User.class)`。
- 通过 `@Autowired` 依赖注入 User Bean。
- 使用 user Bean(`toString()` 方法替换 `user()` 中的实现。

下一步，重启应用后，再将 `user.age` 配置从 18 调整为 99，控制台日志输出符合期望：

```
[init] user name : nacos-config-sample , age : 18
.....
[fixed-127.0.0.1_8848] [data-received] dataId=nacos-config-sample.properties,
group=DEFAULT_GROUP, tenant=null, md5=b0f42fac52934faf69757c2b6770d39c,
content=user.name=nacos-config-sample
user.age=90, type=properties
.....
[destroy] user name : nacos-config-sample , age : 18
o.s.c.e.event.RefreshEventListener      : Refresh keys changed: [user.age]
```

接下来，访问 REST 资源 `/user`，观察终端日志输出：

```
% curl http://127.0.0.1:8080/user
[HTTP] User{name='nacos-config-sample', age=90}
```

User Bean 属性成功地变更为 90，达到实战效果。上小节提到 Nacos Config 配置变更会影响 `@RefreshScope` Bean 的生命周期方法回调。同理，如果为 User 增加初始化和销毁方法的话，也会出现行文，不过本次将 User 实现 Spring 标准的生命周期接口 `InitializingBean` 和 `DisposableBean`：

```
@RefreshScope
@ConfigurationProperties(prefix = "user")
public class User implements InitializingBean, DisposableBean {
    private String name;
    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;
}

@Override
public String toString() {
    return "User{" +
        "name='" + name + '\'' +
        ", age=" + age +
        '}';
}

@Override
public void afterPropertiesSet() throws Exception {
    System.out.println("[afterPropertiesSet()] " + toString());
}

@Override
public void destroy() throws Exception {
    System.out.println("[destroy()] " + toString());
}
}
```

代码调整后，重启应用，并修改配置(90 -> 19)，观察控制台日志输出：

```
[init] user name : nacos-config-sample , age : 90
.....
c.a.n.client.config.impl.ClientWorker    : [fixed-127.0.0.1_8848] [data-received]
dataId=nacos-config-sample.properties, group=DEFAULT_GROUP, tenant=null,
md5=30d26411b8c1ffc1d16b3f9186db498a, content=user.name=nacos-config-sample
user.age=19, type=properties
.....
[destroy()] User{name='nacos-config-sample', age=90}
[afterPropertiesSet()] User{name='nacos-config-sample', age=19}
[destroy] user name : nacos-config-sample , age : 90
.....
o.s.c.e.event.RefreshEventListener      : Refresh keys changed: [user.age]
```

不难发现，User Bean 的生命周期方法不仅被调用，并且仍旧是先销毁，再初始化。那么，这个现象和之前看到的 SpringApplication 重启是否有关系呢？答案也是肯定的，不过还是后文再讨论。

下一小节将继续讨论怎么利用底层 Nacos 配置监听实现 Bean 属性动态刷新。

5.3 使用 Nacos Config 监听实现 Bean 属性动态刷新

前文曾提及 com.alibaba.nacos.api.config.listener.Listener 是 Nacos Client API 标准的配置监听器接口，由于仅监听配置内容，并不能直接与 Spring 体系打通，因此，需要借助于 Spring Cloud Alibaba Nacos Config API NacosConfigManager (感谢小伙伴 liaochuntao 和 zkzlx 的代码贡献)，代码调整如下：

```
@SpringBootApplication
@RestController
@RefreshScope
@EnableConfigurationProperties(User.class)
public class NacosConfigSampleApplication {

    @Value("${user.name}")
    private String userName;

    @Value("${user.age}")
    private int userAge;
```

```
@Autowired
    private User user;

    @Autowired
    private NacosConfigManager nacosConfigManager;

    @Bean
    public ApplicationRunner runner() {
        return args -> {
            String dataId = "nacos-config-sample.properties";
            String group = "DEFAULT_GROUP";
            nacosConfigManager.getConfigService().addListener(dataId, group, new
AbstractListener() {
                @Override
                public void receiveConfigInfo(String configInfo) {
                    System.out.println("[Listener] " + configInfo);
                }
            });
        };
    }

    @PostConstruct
    public void init() {
        System.out.printf("[init] user name : %s , age : %d\n", userName, userAge);
    }

    @PreDestroy
    public void destroy() {
        System.out.printf("[destroy] user name : %s , age : %d\n", userName, userAge);
    }

    @RequestMapping("/user")
    public String user() {
        return "[HTTP] " + user;
    }

    public static void main(String[] args) {
        SpringApplication.run(NacosConfigSampleApplication.class, args); }
}
```

代码主要变化：

- `@Autowired` 依赖注入 `NacosConfigManager`。
- 新增 `runner()` 方法，通过 `NacosConfigManagerBean` 获取 `ConfigService`，并增加了 `AbstractListener` (`Listener` 抽象类) 实现，监听 `dataId = "nacos-config-sample.properties"` 和 `group = "DEFAULT_GROUP"` 的配置内容。

重启应用，并将配置 `user.age` 从 19 调整到 90，观察日志变化：

```
c.a.n.client.config.impl.ClientWorker : [fixed-127.0.0.1_8848] [data-received]
dataId=nacos-config-sample.properties, group=DEFAULT_GROUP, tenant=null,
md5=b0f42fac52934faf69757c2b6770d39c, content=user.name=nacos-config-sample

user.age=90, type=properties
[Listener] user.name=nacos-config-sample
user.age=90
.....
```

在第 1 行日志下方，新增了监听实现代码的输出内容，不过这段内容是完整的配置，而非变化的内容。读者请务必注意其中的差异。下一步要解决的是将配置映射到 Bean 属性，此处给出一个简单的解决方案，实现步骤有两个：

- 将 `String` 内容转化为 `Properties` 对象。
- 将 `Properties` 属性值设置到对应的 Bean 属性。

代码调整如下：

```
@SpringBootApplication
@RestController
@RefreshScope
@EnableConfigurationProperties(User.class)
public class NacosConfigSampleApplication {

    .....

    @Bean
    public ApplicationRunner runner() {
        return args -> {
```

```
String dataId = "nacos-config-sample.properties";
String group = "DEFAULT_GROUP";
nacosConfigManager.getConfigService().addListener(dataId, group, new
AbstractListener() {
    @Override
    public void receiveConfigInfo(String configInfo) {
        System.out.println("[Listener] " + configInfo);
        System.out.println("[Before User] " + user);

        Properties properties = new Properties();
        try {
            properties.load(new StringReader(configInfo));
            String name = properties.getProperty("user.name");
            int age = Integer.valueOf(properties.getProperty("user.age"));
            user.setName(name);
            user.setAge(age);
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("[After User] " + user);
    }
});
};
}

.....

}
```

重启应用，并将配置 user.age 从 90 调整到 19，观察日志变化：

```
[Listener] user.name=nacos-config-sampleuser.age= 19
[Before User] User{name='nacos-config-sample', age=90}
[After User] User{name='nacos-config-sample', age=19}
```


上述三个例子均围绕着 Nacos Config 实现 Bean 属性动态更新，不过它们是 Spring Cloud 使用场景。如果读者的应用仅使用 Spring 或者 Spring Boot，可以考虑 Nacos Spring 工程，Github 地址：<https://github.com/nacos-group/nacos-spring-project>，其中 @NacosValue 支持属性粒度的更新。

6. Nacos Config 高级配置

6.1 支持自定义 namespace 的配置

首先看一下 Nacos 的 Namespace 的概念，[Nacos 概念](#)用于进行租户粒度的配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

在没有明确指定 `spring.cloud.nacos.config.namespace` 配置的情况下，默认使用的是 Nacos 上 Public 这个 namespace。如果需要使用自定义的命名空间，可以通过以下配置来实现：

```
spring.cloud.nacos.config.namespace=b3404bc0-d7dc-4855-b519-570ed34b62d7
```

注：该配置必须放在 `bootstrap.properties` 文件中。此外 `spring.cloud.nacos.config.namespace` 的值是 namespace 对应的 id，id 值可以在 Nacos 的控制台获取。并且在添加配置时注意不要选择其他的 namespace，否则将会导致读取不到正确的配置

6.2 支持自定义 Group 的配置

在没有明确指定 `spring.cloud.nacos.config.group` 配置的情况下，默认使用的是 `DEFAULT_GROUP`。如果需要自定义自己的 Group，可以通过以下配置来实现：

```
spring.cloud.nacos.config.group=DEVELOP_GROUP
```

注：该配置必须放在 `bootstrap.properties` 文件中。并且在添加配置时 Group 的值一定要和 `spring.cloud.nacos.config.group` 的配置值一致。

6.3 支持自定义扩展的 Data Id 配置

Spring Cloud Alibaba Nacos Config 从 0.2.1 版本后, 可支持自定义 Data Id 的配置。关于这部分详细的设计可参考 [这里](#)。一个完整的配置案例如下所示:

```
spring.application.name=opensource-service-provider
spring.cloud.nacos.config.server-addr=127.0.0.1:8848
# config external configuration
# 1、Data Id 在默认的组 DEFAULT_GROUP, 不支持配置的动态刷新
spring.cloud.nacos.config.extension-configs[0].data-id=ext-config-common01.properties
# 2、Data Id 不在默认的组, 不支持动态刷新谢谢
spring.cloud.nacos.config.extension-configs[1].data-id=ext-config-common02.properties
spring.cloud.nacos.config.extension-configs[1].group=GLOBLE_GROUP
# 3、Data Id 既不在默认的组, 也支持动态刷新
spring.cloud.nacos.config.extension-configs[2].data-id=ext-config-common03.properties
spring.cloud.nacos.config.extension-configs[2].group=REFRESH_GROUP
spring.cloud.nacos.config.extension-configs[2].refresh=true
```

可以看到:

- 通过 `spring.cloud.nacos.config.extension-configs[n].data-id` 的配置方式来支持多个 Data Id 的配置。
- 通过 `spring.cloud.nacos.config.extension-configs[n].group` 的配置方式自定义 Data Id 所在的组, 不明确配置的话, 默认是 `DEFAULT_GROUP`。
- 通过 `spring.cloud.nacos.config.extension-configs[n].refresh` 的配置方式来控制该 Data Id 在配置变更时, 是否支持应用中可动态刷新, 感知到最新的配置值。默认是不支持的。

注: 多个 Data Id 同时配置时, 他的优先级关系是 `spring.cloud.nacos.config.extension-configs[n].data-id` 其中 `n` 的值越大, 优先级越高。

`spring.cloud.nacos.config.extension-configs[n].data-id` 的值必须带文件扩展名, 文件扩展名既可支持 `properties`, 又可以支持 `yaml/yml`。此时 `spring.cloud.nacos.config.file-extension` 的配置对自定义扩展配置的 Data Id 文件扩展名没有影响。

通过自定义扩展的 Data Id 配置, 既可以解决多个应用间配置共享的问题, 又可以支持一个应用有多个配置文件。

为了更加清晰的在多个应用间配置共享的 Data Id，你可以通过以下的方式来配置：

```
# 配置支持共享的 Data Id
spring.cloud.nacos.config.shared-configs[0].data-id=common.yaml
# 配置 Data Id 所在分组，缺省默认 DEFAULT_GROUP
spring.cloud.nacos.config.shared-configs[0].group=GROUP_APP1
# 配置 Data Id 在配置变更时，是否动态刷新，缺省默认 false
spring.cloud.nacos.config.shared-configs[0].refresh=true
```

可以看到：

- 通过 `spring.cloud.nacos.config.shared-configs[n].data-id` 来支持多个共享 Data Id 的配置。
- 通过 `spring.cloud.nacos.config.shared-configs[n].group` 来配置自定义 Data Id 所在的组，不明确配置的话，默认是 `DEFAULT_GROUP`。
- 通过 `spring.cloud.nacos.config.shared-configs[n].refresh` 来控制该 Data Id 在配置变更时，是否支持应用中动态刷新，默认 `false`。

6.4 配置的优先级

Spring Cloud Alibaba Nacos Config 目前提供了三种配置能力从 Nacos 拉取相关的配置。

- A: 通过 `spring.cloud.nacos.config.shared-configs[n].data-id` 支持多个共享 Data Id 的配置
- B: 通过 `spring.cloud.nacos.config.extension-configs[n].data-id` 的方式支持多个扩展 Data Id 的配置
- C: 通过内部相关规则(应用名、应用名+ Profile)自动生成相关的 Data Id 配置

当三种方式共同使用时，他们的一个优先级关系是：A < B < C

6.5 完全关闭配置

通过设置 `spring.cloud.nacos.config.enabled = false` 来完全关闭 Spring Cloud Nacos Config

6.6 更多高级配置

更多关于 Nacos Config Starter 的配置项如下所示:

配置项	Key	默认值	说明
服务端地址	<code>spring.cloud.nacos.config.server-addr</code>		Nacos Server 启动监听的 ip 地址和端口
配置对应的 DataId	<code>spring.cloud.nacos.config.name</code>		先取 prefix, 再取 name, 最后取 spring.application.name
配置对应的 DataId	<code>spring.cloud.nacos.config.prefix</code>		先取 prefix, 再取 name, 最后取 spring.application.name
配置内容编码	<code>spring.cloud.nacos.config.encode</code>		读取的配置内容对应的编码
GROUP	<code>spring.cloud.nacos.config.group</code>	<code>DEFAULT_GROUP</code>	配置对应的组
文件扩展名	<code>spring.cloud.nacos.config.fileExtension</code>	<code>properties</code>	配置项对应的文件扩展名, 目前支持 properties 和 yaml(yml)
获取配置超时时间	<code>spring.cloud.nacos.config.timeout</code>	<code>3000</code>	客户端获取配置的超时时间(毫秒)
接入点	<code>spring.cloud.nacos.config.endpoint</code>		地域的某个服务的入口域名, 通过此域名可以动态地拿到服务端地址

配置项	Key	默认值	说明
命名空间	<code>spring.cloud.nacos.config.namespace</code>		常用场景之一是不同的环境的配置的区分离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等
AccessKey	<code>spring.cloud.nacos.config.accessKey</code>		当要上阿里云时，阿里云上面的一个云账号名
SecretKey	<code>spring.cloud.nacos.config.secretKey</code>		当要上阿里云时，阿里云上面的一个云账号密码
Nacos Server 对应的 context path	<code>spring.cloud.nacos.config.contextPath</code>		Nacos Server 对外暴露的 context path
集群	<code>spring.cloud.nacos.config.clusterName</code>		配置成 Nacos 集群名称
共享配置	<code>spring.cloud.nacos.config.sharedDataIds</code>		共享配置的 DataId, "," 分割
共享配置动态刷新	<code>spring.cloud.nacos.config.refreshableDataIds</code>		共享配置中需要动态刷新的 DataId, "," 分割
自定义 Data Id 配置	<code>spring.cloud.nacos.config.extConfig</code>		属性是个集合，内部由 ConfigPOJO 组成。Config 有 3 个属性，分别是 dataId, group 以及 refresh

7. Nacos Config Actuator Endpoint

Nacos Config 内部提供了一个 Endpoint, 对应的 Endpoint ID 为 `nacos-config`, 其 Actuator Web Endpoint URI 为 `/actuator/nacos-config`。

注: 使用 Nacos Config Spring Cloud 1.x 版本的话, 其 URI 地址则为 `/nacos-config`。

其中, Endpoint 暴露的 json 中包含了三种属性:

- `NacosConfigProperties`: 当前应用 Nacos 的基础配置信息。
- `RefreshHistory`: 配置刷新的历史记录。
- `Sources`: 当前应用配置的数据信息。

由于 Aliyun Java Initializr 所生成的应用工程默认激活 Spring Boot Actuator Endpoints (JMX 和 Web), 具体配置存放在 `application.properties` 文件中, 同时, Actuator Web 端口设置为 8081, 内容如下:

```
management.endpoints.jmx.exposure.include=*
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
# Actuator Web 访问端口
management.server.port=8081
```

因此, 应用 `nacos-config-sample` 无需调整, 直接访问: <http://127.0.0.1:8081/actuator/nacos-config>, 服务响应的内容如下:

```
{
  "NacosConfigProperties": {
    "serverAddr": "127.0.0.1:8848",
    "username": "",
    "password": "",
    "encode": null,
    "group": "DEFAULT_GROUP",
    "prefix": null,
    "fileExtension": "properties",
    "timeout": 3000,
```

```
"maxRetry": null,
"configLongPollTimeout": null,
"configRetryTime": null,
"enableRemoteSyncConfig": false,
"endpoint": null,
"namespace": null,
"accessKey": null,
"secretKey": null,
"contextPath": null,
"clusterName": null,
"name": null,
"sharedConfigs": null,
"extensionConfigs": null,
"refreshEnabled": true,
"sharedDataids": null,
"refreshableDataids": null,
"extConfig": null,
"configServiceProperties": {
  "secretKey": "",
  "namespace": "",
  "username": "",
  "enableRemoteSyncConfig": "false",
  "configLongPollTimeout": "",
  "configRetryTime": "",
  "encode": "",
  "serverAddr": "127.0.0.1:8848",
  "maxRetry": "",
  "clusterName": "",
  "password": "",
  "accessKey": "",
  "endpoint": ""
}
},
"RefreshHistory": [

],
"Sources": [
  {
    "lastSynced": "2020-09-14 11:11:37",
    "dataId": "nacos-config-sample.properties"
  },
  {
    "lastSynced": "2020-09-14 11:11:37",
    "dataId": "nacos-config-sample"
  }
]
}]}
```


服务注册与发现

1. 简介

服务注册与发现是微服务架构体系中最关键的组件之一。如果尝试着手动的方式来给每一个客户端来配置所有服务提供者的服务列表是一件非常困难的事,而且也不利于服务的动态扩缩容。Nacos Discovery 可以帮助您将服务自动注册到 Nacos 服务端并且能够动态感知和刷新某个服务实例的服务列表。除此之外, Nacos Discovery 也将服务实例自身的一些元数据信息—例如 host, port, 健康检查 URL, 主页等内容注册到 Nacos。Nacos 的获取和启动方式可以参考 [Nacos 官网](#)。

2. 学习目标

- 掌握 Nacos Discovery 实现 Spring Cloud 服务注册和发现
- 掌握 Nacos Discovery 整合 Spring Cloud 负载均衡和服务调用
- 理解 Nacos Discovery 高级特性: 命名空间、安全控制、元数据、Nacos Watch 等

3. 详细内容

- 快速上手: 指导读者从使用 Nacos Discovery 进行服务注册/发现
- 服务调用整合: 实战 Nacos Discovery 整合 [@LoadBalanced](#) RestTemplate 以及 Open Feign
- 运维特性: 演示 Nacos Discovery 高级外部化配置以及 Endpoint 内部细节

4. 快速上手

4.1 如何引入 Nacos Discovery 进行服务注册/发现

Nacos Discovery 引入的方式通常有两种, 由易到难分别为: [Aliyun Java Initializr](#) 引入和 Maven pom.xml 依赖。官方推荐使用 [Aliyun Java Initializr](#) 方式引入 Nacos Discovery, 以便简化组件之间的依赖关系。

4.1.1 [偷懒] 直接在沙箱里查看应用代码

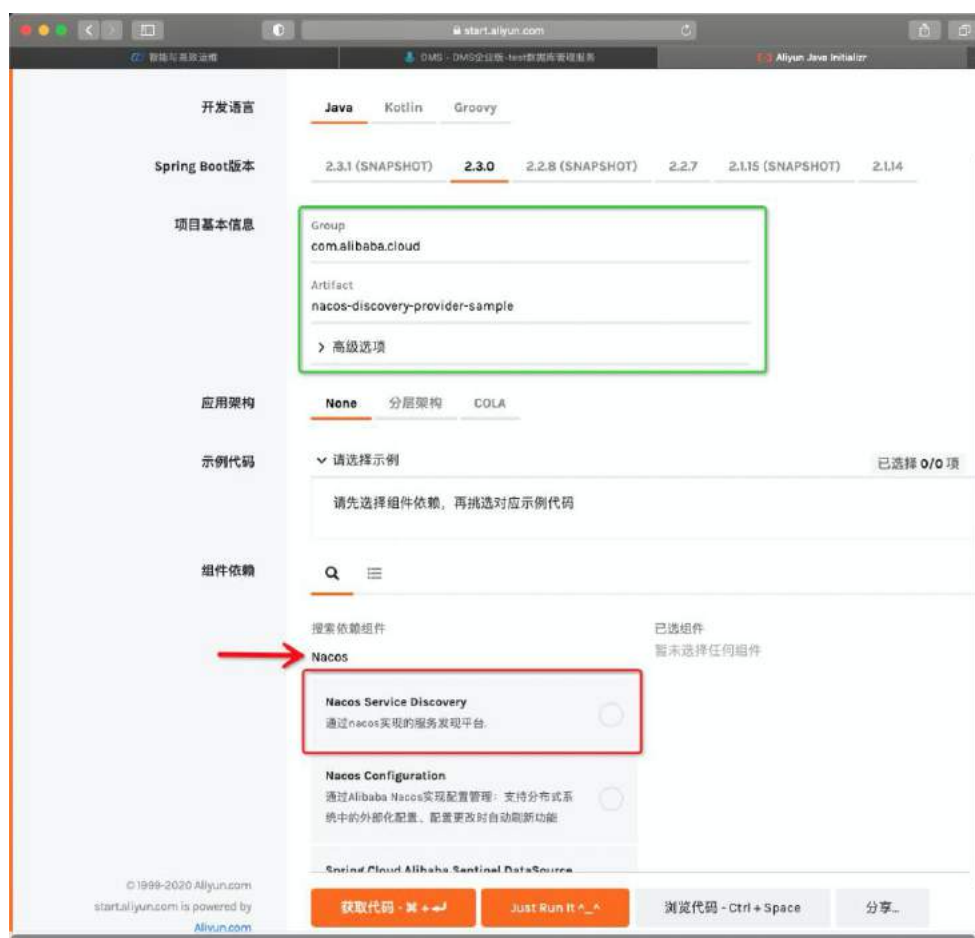
点击 [链接](#)，直接访问沙箱环境，这里会有为你准备好的案例代码^_^。

4.1.2 [简单] 通过 Aliyun Java Initializr 创建工程并引入 Nacos Discovery (推荐)

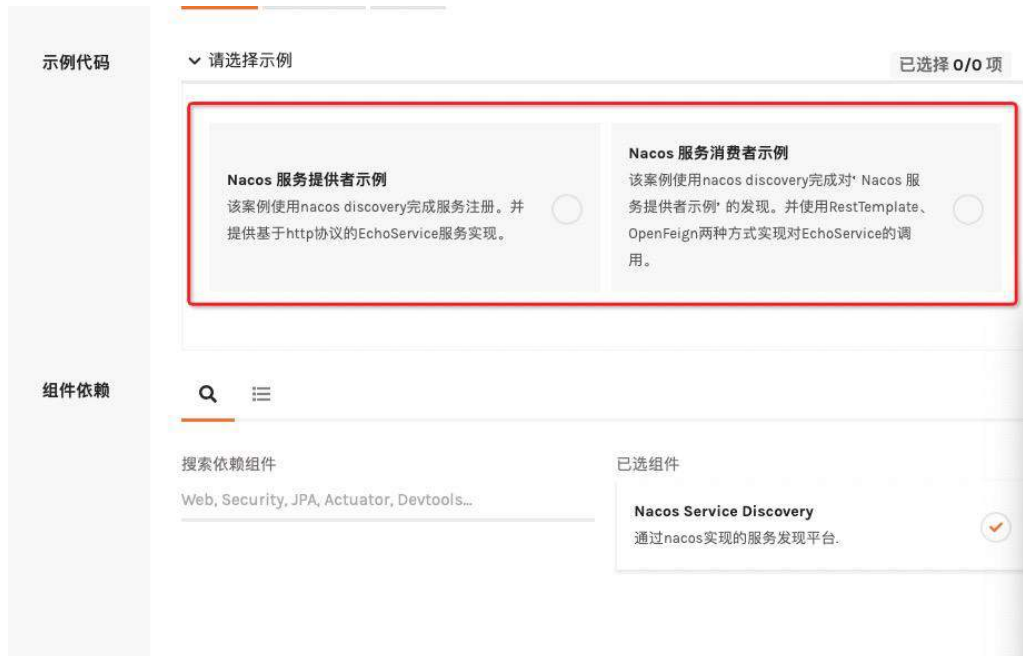
由于 Spring Cloud 组件的版本和依赖较为复杂，推荐读者使用 [Aliyun Java Initializr](#) 构建应用工程。

下文以 Google Chrome 浏览器为例，当网页加载后，首先，在 "项目基本信息" 部分输入 Group：“com.alibaba.cloud” 以及 Artifact：“nacos-discovery-provider-sample”（见下图绿框部分）。

然而，“组件依赖” 输入框搜索：“Nacos”（见下图红箭头部分），最后，选择 "Nacos Service Discovery"（见下图红框部分），如下所示：

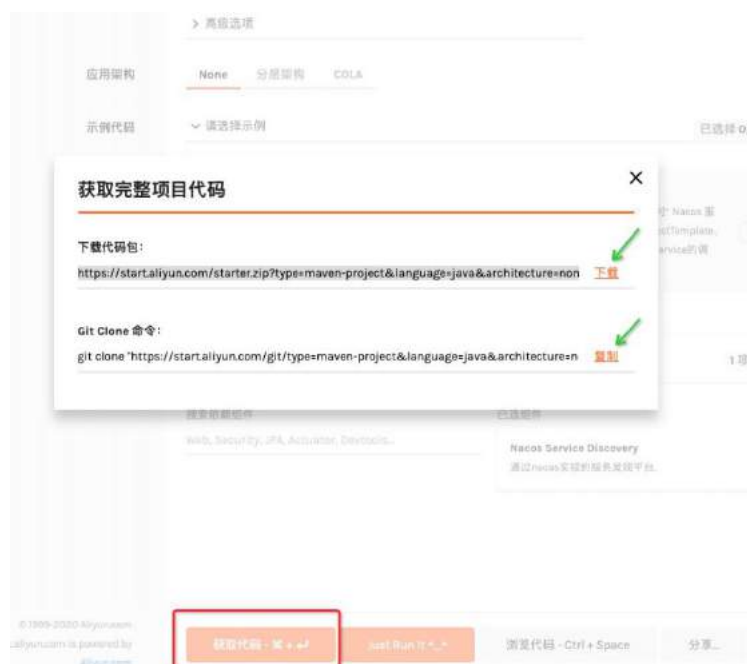


Nacos Service Discovery 组件选择后，展开“示例代码”部分，选择合适的示例代码：



由于“服务注册&发现”是两个独立的端，所以示例代码也被分为两个部分。

通过点击“获取代码”来获得由平台生成的代码：



点击下载按钮后，平台将生成一个名为 “nacos-discovery-provider-sample.zip” 的压缩文件，将其保存到本地目录，并解压该文件，工程目录将随之生成。

打开目录下的 pom.xml 文件，不难发现 Nacos Discovery starter 声明其中（以下 XML 内容均来自于项目根路径中的 pom.xml 文件）：

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

不过该 starter 并未指定版本，具体的版本声明在 com.alibaba.cloud:spring-cloud-alibaba-dependencies 部分：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>${spring-cloud-alibaba.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${spring-boot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

其中，`${spring-cloud-alibaba.version}` 和 `${spring-boot.version}` 分别为 Spring Cloud Alibaba 和 Spring Boot 组件依赖的版本，它们的版本定义在 `<properties>` 元素中，即 2.2.1.RELEASE 和 2.3.0.RELEASE：

```
<properties>
  <java.version>1.8</java.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <spring-boot.version>2.3.0.RELEASE</spring-boot.version>
  <spring-cloud-alibaba.version>2.2.1.RELEASE</spring-cloud-alibaba.version>
</properties>
```

如果读者非常熟悉 Maven 依赖管理的配置方式,可以考虑 Maven pom.xml 依赖 Nacos Discovery。

4.1.3 [高级] 通过 Maven pom.xml 依赖 Nacos Discovery

如果要在您的项目中使用 Nacos 来实现服务注册/发现,使用 group ID 为 com.alibaba.cloud 和 artifact ID 为 spring-cloud-starter-alibaba-nacos-discovery 的 starter。

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId></dependency>
```

该声明方式同样需要声明 com.alibaba.cloud:spring-cloud-alibaba-dependencies, 内容与上小节相同, 在此不再赘述。下一节将讨论如何使用 Nacos Discovery 进行服务注册/发现。

4.2 使用 Nacos Discovery 进行服务注册/发现

使用 Nacos Discovery 进行服务注册/发现与传统 Spring Cloud 的方式并没有本质区别, 仅需添加相关外部化配置即可工作。换言之, Nacos Discovery 不会侵入应用代码, 方便应用整合和迁移, 这归功于 Spring Cloud 的高度抽象。

如果读者熟悉 Spring Cloud 服务注册和发现的话, 通常需要将注册中心预先部署, Nacos Discovery 也不例外。

4.2.1 启动 Nacos 服务器

具体启动方式参考 [Nacos 官网](#)。

Nacos Server 启动后，进入 <http://ip:8848> 查看控制台(默认账号名/密码为 nacos/nacos)：



关于更多的 Nacos Server 版本，可以从 [release 页面](#) 下载最新的版本。

4.2.2 启动服务提供者 (Provider)

4.2.2.1 增加 Maven 依赖

回到之前构建的应用 `nacos-discovery-provider-sample`，在此基础增加 Spring WebMVC 以及 Spring Boot Actuator Starter 依赖：

```
<dependencies>

  <!-- Spring WebMVC Starter -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Boot Actuator Starter -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>

</dependencies>
```

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>

...

</dependencies>
```

4.2.2.2 增加 Nacos Discovery 外部化配置

Nacos 基本的配置需要添加到 application.properties (也可是 application.yml) 文件中。application.properties 文件已被 Aliyun Java Initializr 生成，内容如下：

```
spring.application.name=nacos-discovery-provider-sample
management.endpoints.jmx.exposure.include=*
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
# spring cloud access&secret config
# 可以访问如下地址查看: https://usercenter.console.aliyun.com/#/manage/ak
alibaba.cloud.access-key=****
alibaba.cloud.secret-key=****
# 应用服务 WEB 访问端口
server.port=8080
# Actuator Web 访问端口
management.server.port=8081
```

增加 Nacos Discovery 配置，如下所示：

```
spring.application.name=nacos-discovery-provider-sample
management.endpoints.jmx.exposure.include=*
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always

# spring cloud access&secret config
# 可以访问如下地址查看: https://usercenter.console.aliyun.com/#/manage/ak
alibaba.cloud.access-key=****
```



```
alibaba.cloud.secret-key=****  
  
# 应用服务 WEB 访问端口  
server.port=8080  
  
# Actuator Web 访问端口  
management.server.port=8081  
  
## Nacos 注册中心配置地址（无需配置 HTTP 协议部分）  
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848  
  
## Nacos 客户端认证信息（默认用户名和密码均为 nacos）  
spring.cloud.nacos.discovery.user-name=nacos  
spring.cloud.nacos.discovery.password=nacos
```

请注意，Nacos 服务器默认关闭认证，建议在生产环境开启，详情请参考此 [Blog](#)。

4.2.2.3 激活 Nacos Discovery 服务注册与发现

Aliyun Java Initializr 默认不会自动激活 Nacos Discovery 服务注册与发现，需要在引导类（main 方法所在类）标注 Spring Cloud 服务注册与发现标准注解 `@EnableDiscoveryClient`，代码如下所示：

```
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
  
@SpringBootApplication  
@EnableDiscoveryClient  
public class NacosDiscoveryProviderSampleApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(NacosDiscoveryProviderSampleApplication.class, args);  
    }  
}
```

4.2.2.4 启动引导类

启动引导类 `NacosDiscoveryProviderSampleApplication`，观察控制台输出（隐藏时间部分的内容）：

```
[      main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080
(http) with context path ''

[      main] c.a.c.n.registry.NacosServiceRegistry : nacos registry, DEFAULT_GROUP
nacos-discovery-provider-sample 30.225.19.241:8080 register finished

[      main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s):
8081 (http)

[      main] o.apache.catalina.core.StandardService : Starting service [Tomcat]

[      main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache
Tomcat/9.0.35]

[      main] o.a.c.c.C.[Tomcat-1].[localhost].[/] : Initializing Spring embedded
WebApplicationContext

[      main] o.s.web.context.ContextLoader : Root WebApplicationContext:
initialization completed in 176 ms

[      main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 18 endpoint(s) beneath
base path '/actuator'

[      main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8081
(http) with context path ''

[      main] .NacosDiscoveryProviderSampleApplication : Started
NacosDiscoveryProviderSampleApplication in 3.037 seconds (JVM running for 3.615)
```

按照日志的描述, 应用使用了 Tomcat 作为 Web 服务器, 并且将 8080 和 8081 作为应用服务和 Actuator Web 端口。同时, “c.a.c.n.registry.NacosServiceRegistry : nacos registry, DEFAULT_GROUP nacos-discovery-provider-sample 30.225.19.241:8080 register finished” 表明服务实例注册成功, 其 IP 为 30.225.19.241, 服务端口为 8080。下一步, 观察 Nacos 控制台注册情况。

4.2.2.5 观察 Nacos 控制台服务注册

打开 Nacos 控制台中的“服务列表”, 点击“查询”按钮, 观察页面的变化:



其中应用 `nacos-discovery-provider-sample` 出现在列表中，说明该应用已成功注册。至此，使用 Nacos Discovery 进行服务注册/发现演示完毕。接下来的示例将变得更为复杂，实现 Spring Cloud 服务调用。

如果不想使用 Nacos 作为您的服务注册与发现，可以将 `spring.cloud.nacos.discovery` 设置为 `false`。

5. Nacos Discovery 整合 Spring Cloud 服务调用

从应用架构上，Spring Cloud 服务调用通常需要两个应用，一个为服务提供者（Provider），一个为服务消费者（Consumer）。从技术上，传统的 Spring Cloud 服务通讯方式是基于 REST 实现的，包好两种内建实现方法，分别是 `@LoadBalanced RestTemplate` 以及 `Open Feign`，两者均作用于服务消费者，而服务提供者仅为 `WebMVC` 或者 `WebFlux` 应用（需注册到注册中心）。同时，还允许整合 Spring Cloud 负载均衡 API，实现自定义 REST 服务调用。至于，Spring Cloud Alibaba 引入 `Dubbo` 服务通讯方式，会在后续内容中单独讨论。

综上所述，首先，需要在服务提供者增加 Web 服务。

5.1 服务提供者添加 Web 服务

复用应用 `nacos-discovery-provider-sample`，在引导类 `NacosDiscoveryProviderSampleApplication` 同包下增加 `@RestController` 实现类：

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ServiceController {

    @GetMapping("/echo/{message}")
    public String echo(@PathVariable String message) {
        return "[ECHO] : " + message;
    }
}
```

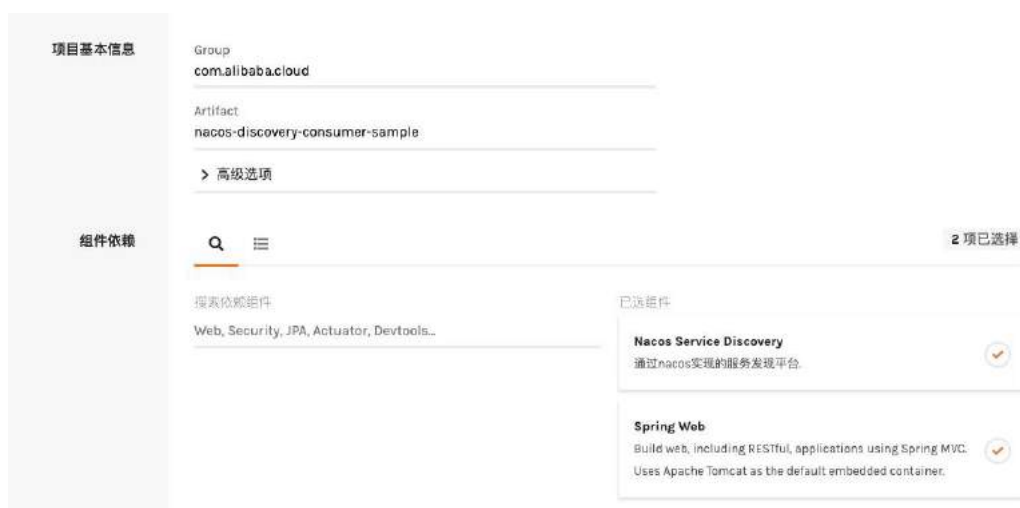
重启应用 `nacos-discovery-provider-sample`，测试该 Web 服务端口：

```
% curl http://127.0.0.1:8080/echo/Hello,World  
[ECHO] : Hello,World
```

结果符合期望，下一步增加消费者应用。

5.2 Nacos Discovery 整合 @LoadBalanced RestTemplate

继续使用 Aliyun Java Initializr 创建消费者应用 - `nacos-discovery-consumer-sample`，选择 Nacos Service Discovery 和 Spring Web 组件：



生成项目，并使用 IDE 导入工程。

5.2.1 增加 Nacos Discovery 外部化配置

与应用 `nacos-discovery-provider-sample` 配置类似，增加 Nacos Discovery 外部化配置。如果是本地部署的话，请调整应用服务和 Actuator 端口，以免与应用 `nacos-discovery-provider-sample` 端口冲突，完整配置如下：

```

spring.application.name=nacos-discovery-consumer-sample
management.endpoints.jmx.exposure.include=*
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
# spring cloud access&secret config
# 可以访问如下地址查看:https://usercenter.console.aliyun.com/#/manage/ak
alibaba.cloud.access-key=****
alibaba.cloud.secret-key=****
# 应用服务 WEB 访问端口
server.port=9090
# Actuator Web 访问端口
management.server.port=9091
## Nacos 注册中心配置地址（无需配置 HTTP 协议部分）
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
## Nacos 客户端认证信息（默认用户名和密码均为nacos）
spring.cloud.nacos.discovery.user-name=nacos
spring.cloud.nacos.discovery.password=naocs

```

5.2.2 服务消费者激活 Nacos Discovery 服务注册与发现

与应用 `nacos-discovery-provider-sample` 实现一样，在引导类上标注 `@EnableDiscoveryClient`，代码如下：

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
@EnableDiscoveryClient
@SpringBootApplication
public class NacosDiscoveryConsumerSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(NacosDiscoveryConsumerSampleApplication.class, args);
    }
}

```

5.2.3 服务消费者使用 `@LoadBalanced` RestTemplate 实现服务调用

前文提到 `@LoadBalanced` `RestTemplate` 是 Spring Cloud 内建的服务调用方式，因此需要在应用 `nacos-discovery-consumer-sample` 增加执行代码，消费应用 `nacos-discovery-provider-sample` REST 服务 `/echo/{message}`，故在引导类同包下新增 `RestController` 实现：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class RestTemplateController {

    @LoadBalanced
    @Autowired
    public RestTemplate restTemplate;

    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @GetMapping("/call/echo/{message}")
    public String callEcho(@PathVariable String message) {
        // 访问应用 nacos-discovery-provider-sample 的 REST "/echo/{message}"
        return restTemplate.getForObject("http://nacos-discovery-provider-sample/echo/" +
message, String.class);
    }
}
```

以上代码实现方式与传统 Spring Cloud 的方式无异，下一步启动引导类 `NacosDiscoveryConsumerSampleApplication`，并测试运行结果：

```
% curl http://127.0.0.1:9090/call/echo/Hello,World
[ECHO] : Hello,World
```

结果符合期望，说明 Nacos Discovery 整合 `@LoadBalanced` RestTemplate 的实现与标准 Spring Cloud 实现的差异仅体现在 Maven 依赖 starter 以及外部化配置上。接下来，应用 `nacos-discovery-consumer-sample` 将继续与 Spring Cloud OpenFeign 整合。

5.3 Nacos Discovery 整合 Spring Cloud OpenFeign

Spring Cloud OpenFeign 是 Spring Cloud 基于 REST 客户端框架 `OpenFeign` 而构建，使得服务发现和负载均衡透明，开发人员只需关注服务消费者接口契约。同时，Spring Cloud OpenFeign 可以与 `@LoadBalanced` RestTemplate 共存，因此，可在原有应用 `nacos-discovery-consumer-sample` 的基础上，增加 Maven 依赖和代码实现整合。

关于 Spring Cloud OpenFeign 的技术细节，可参考官方文档：
<https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/>

5.3.1 服务消费者增加 Spring Cloud OpenFeign Maven 依赖

在 `nacos-discovery-consumer-sample` 项目 `pom.xml` 中追加 Spring Cloud OpenFeign Maven 依赖：

```
<!-- Spring Cloud OpenFeign -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

下一步，则是新增 Spring Cloud OpenFeign 服务声明接口

5.3.2 服务消费者增加 Spring Cloud OpenFeign 服务声明接口

由于需要消费应用 `nacos-discovery-provider-sample` 提供的 REST 服务 `/echo/{message}`，根据 Spring Cloud OpenFeign 的要求，需要在消费者应用增加 REST 服务声明接口，即：

```
@FeignClient("nacos-discovery-provider-sample") // 指向服务提供者应用
public interface EchoService {

    @GetMapping("/echo/{message}")
    String echo(@PathVariable("message") String message);
}
```

不难发现，`echo(String)` 方法在 Spring MVC 请求映射的方式与 `nacos-discovery-provider-sample` 中的 `ServiceController` 基本相同，唯一区别在于 `@PathVariable` 注解指定了 `value` 属性 `"message"`，这是因为默认情况，Java 编译器不会将接口方法参数名添加到 Java 字节码中。

下一步，激活 Spring Cloud OpenFeign 服务声明接口。

5.3.3 服务消费者激活 Spring Cloud OpenFeign 服务声明接口

激活 Spring Cloud OpenFeign 服务声明接口的方法非常简单，仅需在引导类标注 `@EnableFeignClients`，如果声明接口与引导类不在同一个包的话，请使用 `basePackages` 属性指定。由于本例的 `EchoService` 与引导类位于同一包下，因此，无需指定：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients // 激活 @FeignClient
public class NacosDiscoveryConsumerSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(NacosDiscoveryConsumerSampleApplication.class, args);
    }
}
```

激活步骤就此完成，下一步为 Spring Cloud OpenFeign 服务接口增加 `RestController` 实现。

5.3.4 服务消费者使用 Spring Cloud OpenFeign 服务声明接口实现服务调用

新增名为 `OpenFeignController` 的实现类:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController public class OpenFeignController {

    @Autowired
    private EchoService echoService;

    @GetMapping("/feign/echo/{message}")
    public String feignEcho(@PathVariable String message) {
        return echoService.echo(message);
    }
}
```

重启引导类 `NacosDiscoveryConsumerSampleApplication` ,并测试 `/feign/echo/{message}` 结果:

```
% curl http://127.0.0.1:9090/feign/echo/Hello,World
[ECHO] : Hello,World
```

结果符合期望,说明 Nacos Discovery 整合 Spring Cloud OpenFeign 与传统方式也是相同的。

综上所述, Nacos Discovery 在 Spring Cloud 服务调用是无侵入的。

6. Nacos Discovery 更多配置项信息

更多关于 Nacos Discovery Starter 的配置项如下所示:

配置项	Key	默认值	说明
服务端地址	spring.cloud.nacos.discovery.server-addr		Nacos Server 启动监听的 ip 地址和端口
服务名	spring.cloud.nacos.discovery.service	\${spring.application.name}	注册的服务名
权重	spring.cloud.nacos.discovery.weight	1	取值范围 1 到 100，数值越大，权重越大
网卡名	spring.cloud.nacos.discovery.network-interface		当 IP 未配置时，注册的 IP 为此网卡所对应的 IP 地址，如果此项也未配置，则默认取第一块网卡的地址
注册的 IP 地址	spring.cloud.nacos.discovery.ip		优先级最高
注册的端口	spring.cloud.nacos.discovery.port	-1	默认情况下不用配置，会自动探测
命名空间	spring.cloud.nacos.discovery.namespace		常用场景之一是不同的环境的注册区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等
AccessKey	spring.cloud.nacos.discovery.access-key		当要上阿里云时，阿里云上面的一个云账号名

配置项	Key	默认值	说明
SecretKey	spring.cloud.nacos.discovery.secret-key		当要上阿里云时，阿里云上面的一个云账号密码
Metadata	spring.cloud.nacos.discovery.metadata		使用 Map 格式配置，用户可以根据自己的需要自定义一些和服务相关的元数据信息
日志文件名	spring.cloud.nacos.discovery.log-name		
集群	spring.cloud.nacos.discovery.cluster-name	DEFAULT	Nacos 集群名称
接入点	spring.cloud.nacos.discovery.endpoint		地域的某个服务的入口域名，通过此域名可以动态地拿到服务端地址
是否集成 Ribbon	ribbon.nacos.enabled	true	一般都设置成 true 即可
是否开启 Nacos Watch	spring.cloud.nacos.discovery.watch.enabled		

7. Nacos Discovery Actuator Endpoint

Nacos Discovery 内部提供了一个 Endpoint, 对应的 endpoint id 为 `nacos-discovery`, 其 Actuator Web Endpoint URI 为 `/actuator/nacos-discovery`

注：使用 Nacos Config Spring Cloud 1.x 版本的话，其 URI 地址则为 `/nacos-discovery`)

Endpoint 暴露的 json 中包含两种属性:

- subscribe: 显示了当前服务有哪些服务订阅者。
- NacosDiscoveryProperties: 当前应用 Nacos 的基础配置信息。

由于 Aliyun Java Initializr 所生成的应用工程默认激活 Spring Boot Actuator Endpoints (JMX 和 Web), 具体配置存放在 application.properties 文件中, 同时, Actuator Web 端口设置为 8081, 内容如下:

```
management.endpoints.jmx.exposure.include=*
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
# Actuator Web 访问端口
management.server.port=8081
```

因此, 应用 nacos-discovery-provider-sample 无需调整, 直接访问: <http://127.0.0.1:8081/actuator/nacos-discovery>, 服务响应的内容如下:

```
{ "subscribe": [
  {
    "jsonFromServer": "",
    "name": "nacos-provider",
    "clusters": "",
    "cacheMillis": 10000,
    "hosts": [
      {
        "instanceId": "30.5.124.156#8081#DEFAULT#nacos-provider",
        "ip": "30.5.124.156",
        "port": 8081,
        "weight": 1.0,
        "healthy": true,
        "enabled": true,
        "cluster": {
          "serviceName": null,
          "name": null,
          "healthChecker": {
            "type": "TCP"
          }
        }
      }
    ]
  }
]
```

```
        "defaultPort": 80,
        "defaultCheckPort": 80,
        "useIPPort4Check": true,
        "metadata": {

        }
    },
    "service": null,
    "metadata": {

    }
}
],
"lastRefTime": 1541755293119,
"checksum": "e5a699c9201f5328241c178e804657e11541755293119",
"allIPs": false,
"key": "nacos-provider",
"valid": true
}
], "NacosDiscoveryProperties": {
    "serverAddr": "127.0.0.1:8848",
    "endpoint": "",
    "namespace": "",
    "logName": "",
    "service": "nacos-provider",
    "weight": 1.0,
    "clusterName": "DEFAULT",
    "metadata": {

    },
    "registerEnabled": true,
    "ip": "30.5.124.201",
    "networkInterface": "",
    "port": 8082,
    "secure": false,
    "accessKey": "",
    "secretKey": ""
}
}
```

分布式服务调用

1. 简介

在《Spring Cloud Alibaba 服务注册与发现》篇中曾提到，Spring Cloud Alibaba Nacos Discovery 能无缝整合 Spring Cloud OpenFeign。换言之，Spring Cloud Alibaba 延续了 Spring Cloud 分布式服务调用的特性。除此之外，Spring Cloud Alibaba 引入了 Dubbo Spring Cloud，扩展了分布式服务调用能力，不仅能使 Apache Dubbo 和 OpenFeign 共存，还允许 Spring Cloud 标准调用底层通过 Dubbo 支持的通讯协议传输。无论开发人员是 Dubbo 用户还是 Spring Cloud 用户，都能轻松地驾驭，并以接近“零”成本的代价使应用向上迁移。Dubbo Spring Cloud 致力于简化 Cloud Native 开发成本，提高研发效能以及提升应用性能等目的。

2. 学习目标

- 使用 Dubbo Spring Cloud 实现 Spring Cloud 分布式服务调用
- 使用 Dubbo Spring Cloud 替换 Spring Cloud 分布式服务调用底层协议
- 理解 Dubbo Spring Cloud 高级特性：服务订阅、元数据、Actuator

3. 详细内容

- 快速上手：使用 Apache Dubbo
- 适配整合：使用注解 `@DubboTransported` 适配 Spring Cloud OpenFeign 和 `@LoadBalanced RestTemplate`
- 运维特性：演示服务订阅、元信息（服务、REST）以及 Actuator Endpoints

4. 功能特性

由于 Dubbo Spring Cloud 构建在原生的 Spring Cloud 之上，其服务治理方面的能力可认为是 Spring Cloud Plus，不仅完全覆盖 Spring Cloud 原生特性，而且提供更为稳定和成熟的实现，特性比对如下表所示：

功能组件	Spring Cloud	Dubbo Spring Cloud
分布式配置 (Distributed configuration)	Git、Zookeeper、Consul、JDBC	Spring Cloud 分布式配置 +Dubbo 配置中心
服务注册与发现 (Service registration and discovery)	Eureka、Zookeeper、Consul	Spring Cloud 原生注册中心?+Dubbo 原生注册中心
负载均衡 (Load balancing)	Ribbon (随机、轮询等算法)	Dubbo 内建实现 (随机、轮询等算法 + 权重等特性)
服务熔断 (Circuit Breakers)	Spring Cloud Hystrix	Spring Cloud Hystrix + Alibaba Sentinel ¹⁷ 等
服务调用 (Service-to-service calls)	Open Feign、Rest Template	Spring Cloud 服务调用 + Dubbo@Reference
链路跟踪 (Tracing)	Spring Cloud Sleuth?+ Zipkin	Zipkin、opentracing 等

4.1 高亮特性

4.1.1 Dubbo 使用 Spring Cloud 服务注册与发现

Dubbo Spring Cloud 基于 Spring Cloud Commons 抽象实现 Dubbo 服务注册与发现，无需添加任何外部化配置，就能轻松地桥接到所有原生 Spring Cloud 注册中心，包括：

- Nacos
- Eureka
- Zookeeper
- Consul

注：Dubbo Spring Cloud 将在下个版本支持 Spring Cloud 注册中心与 Dubbo 注册中心并存，提供双注册机制，实现无缝迁移。

4.1.2 Dubbo 作为 Spring Cloud 服务调用

默认情况，Spring Cloud Open Feign 以及 `@LoadBalanced`RestTemplate` 作为 Spring Cloud 的两种服务调用方式。Dubbo Spring Cloud 为其提供了第三种选择，即 Dubbo 服务将作为 Spring Cloud 服务调用的同等公民出现，应用可通过 Apache Dubbo 注解 `@Service` 和 `@Reference` 暴露和引用 Dubbo 服务，实现服务间多种协议的通讯。同时，也可以利用 Dubbo 泛化接口轻松实现服务网关。

4.1.3 Dubbo 服务自省

Dubbo Spring Cloud 引入了全新的服务治理特性 - 服务自省 (Service Introspection)，其设计目的在于最大化减轻注册中心负载，去 Dubbo 注册元信息中心化。假设一个 Spring Cloud 应用引入 Dubbo Spring Boot Starter，并暴露 N 个 Dubbo 服务，以“Dubbo Nacos 注册中心”为例，当前应用将注册 N+1 个 Nacos 应用，除 Spring Cloud 应用本身之前，其余 N 个应用均来自于 Dubbo 服务，当 N 越大时，注册中心负载越重。因此，Dubbo Spring Cloud 应用对注册中心的负载相当于传统 Dubbo 的 N 分之一，在不增加基础设施投入的前提下，理论上，使其集群规模扩大 N 倍。当然，未来的 Dubbo 也将提供服务自省的能力。

4.1.4 Dubbo 迁移 Spring Cloud 服务调用

尽管 Dubbo Spring Cloud 完全地保留了原生 Spring Cloud 服务调用特性，不过 Dubbo 服务治理的能力是 Spring Cloud Open Feign 所不及的，如高性能、高可用以及负载均衡稳定性等方面。因此，建议开发人员将 Spring Cloud Open Feign 或者 `@LoadBalanced`RestTemplate` 迁移为 Dubbo 服务。考虑到迁移过程并非一蹴而就，因此，Dubbo Spring Cloud 提供了方案，即 `@DubboTransported` 注解。该注解能够帮助服务消费端的 Spring Cloud Open Feign 接口以及 `@LoadBalanced`RestTemplateBean` 底层走 Dubbo 调用（可切换 Dubbo 支持的协议），而服务提供方则只需在原有 `@RestController` 类上追加 Dubbo `@Service` 注解（需要抽取接口）即可，换言之，在不调整 Feign 接口以及 `RestTemplateURL` 的前提下，实现无缝迁移。如果迁移时间充分的话，建议使用 Dubbo 服务重构系统中的原生 Spring Cloud 服务的定义。

5. 快速上手

5.1 如何引入 Dubbo Spring Cloud

Dubbo Spring Cloud 引入的方式通常有两种，由易到难分别为：[Aliyun Java Initializr](#) 引入和 Maven pom.xml 依赖。官方推荐使用 [Aliyun Java Initializr](#) 方式引入 Dubbo Spring Cloud，以便简化组件之间的依赖关系。

5.1.1 [偷懒] 直接在沙箱里查看应用代码

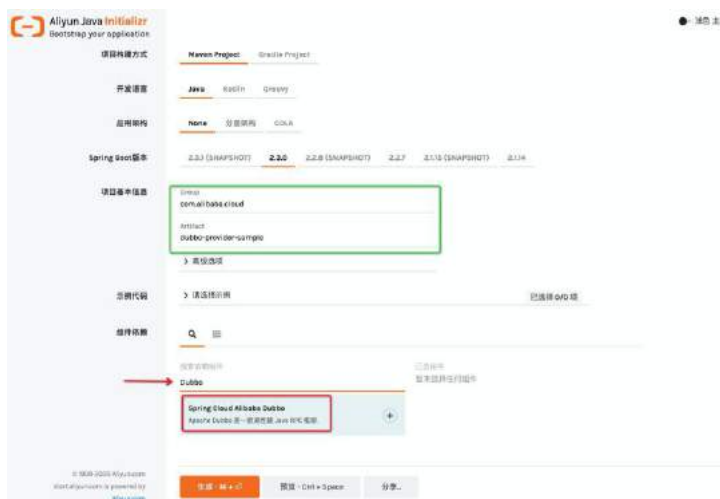
点击 [链接](#)，直接访问沙箱环境，这里会有为你准备好的案例代码^_^。

5.1.2 [简单] 通过 Aliyun Java Initializr 创建工程并引入 Dubbo Spring Cloud (推荐)

Dubbo Spring Cloud 组件的在整个 Spring Cloud Alibaba 版本和依赖最为复杂，推荐读者使用 Aliyun Java Initializr 构建应用工程。读者选择偏好的 Web 浏览器访问 Aliyun Java Initializr，其资源网址为：<https://start.aliyun.com/bootstrap.html>。

下文以 Google Chrome 浏览器为例，当网页加载后，首先，在“项目基本信息”部分输入 Group：“com.alibaba.cloud”以及 Artifact：“dubbo-provider-sample”（见下图绿框部分）。

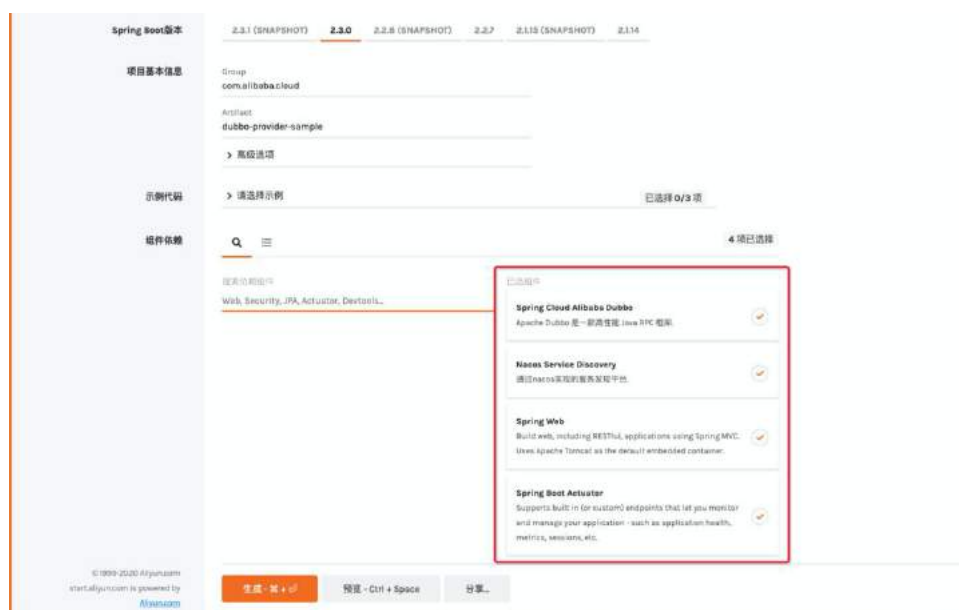
然而，“组件依赖”输入框搜索：“Dubbo”（见下图红箭头部分），最后，选择“Spring Cloud Alibaba Dubbo”（见下图红框部分），如下所示：



继续依赖其他组件：

- Nacos Service Discovery – 服务注册与发现组件
- Spring Web – Spring Web MVC 组件
- Spring Boot Actuator – Spring Boot Actuator 组件

完整组件汇总如下图所示：



点击“生成”高亮按钮，平台将生成一个名为“dubbo-provider-sample.zip”的压缩文件，将其保存到本地目录，并解压该文件，工程目录将随之生成。打开目录下的pom.xml文件，不难发现Dubbo Spring Cloud Starter声明其中（以下XML内容均来自于项目根路径中的pom.xml文件）：

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-dubbo</artifactId>
</dependency>
```

不过该starter并未指定版本，具体的版本声明在com.alibaba.cloud:spring-cloud-alibaba-dependencies部分：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>${spring-cloud-alibaba.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${spring-boot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

其中，`${spring-cloud-alibaba.version}` 和 `${spring-boot.version}` 分别为 Spring Cloud Alibaba 和 Spring Boot 组件依赖的版本，它们的版本定义在 `<properties>` 元素中，即 2.2.1.RELEASE 和 2.3.0.RELEASE：

```
<properties>
  <java.version>1.8</java.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <spring-boot.version>2.3.0.RELEASE</spring-boot.version>
  <spring-cloud-alibaba.version>2.2.1.RELEASE</spring-cloud-alibaba.version>
</properties>
```

如果读者非常熟悉 Maven 依赖管理的配置方式，可以考虑 Maven pom.xml 依赖 Dubbo Spring Cloud。

5.1.3 [高级] 通过 Maven pom.xml 依赖 Dubbo Spring Cloud

如果要在您 Dubbo Spring Cloud 的项目中使用 Nacos 来实现服务注册/发现, 可将两者 Stater 同时依赖:

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-dubbo</artifactId>
</dependency>
```

该声明方式同样需要声明 com.alibaba.cloud:spring-cloud-alibaba-dependencies, 内容与上小节相同, 在此不再赘述。下一节将讨论如何使用 Dubbo Spring Cloud 构建服务提供者。

5.2 使用 Dubbo Spring Cloud 构建服务提供者

按照传统的 Dubbo 开发模式, 在构建服务提供者之前, 第一个步骤是为服务提供者和服务消费者定义 Dubbo 服务接口。

为了确保契约的一致性, 推荐的做法是将 Dubbo 服务接口打包在第二方或者第三方的 artifact (jar) 中, 该 artifact 甚至无需添加任何依赖。

对于服务提供方而言, 不仅通过依赖 artifact 的形式引入 Dubbo 服务接口, 而且需要将其实现。对应的服务消费端, 同样地需要依赖该 artifact, 并以接口调用的方式执行远程方法。接下来的步骤则是创建 artifact。

5.2.1 创建 artifact – dubbo-sample-api

选择合适的文件目录, 通过 Maven 命令行工具构建 artifact dubbo-sample-api, 如下所示:

```
mvn archetype:generate -DgroupId=com.alibaba.cloud -DartifactId=dubbo-sample-api
-Dversion=0.0.1-SNAPSHOT -DinteractiveMode=false
```

命令执行后，名为“dubbo-sample-api”的项目目录生成，切换至该目录，并使用 tree 命令（macOS 命令工具）预览器内部接口：

```
% tree
.
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   ├── com
    │   │   └── alibaba
    │   │       ├── cloud
    │   │       └── App.java
    │   └── test
    │       ├── java
    │       │   ├── com
    │       │   └── alibaba
    │       └── cloud
    └── AppTest.java 11 directories, 3 files
```

其中，App.java 和 AppTest.java 文件并非需要文件，可将其删除。除此之外，当然最重要的是 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>dubbo-sample-api</artifactId>
  <packaging>jar</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>dubbo-sample-api</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies></project>
```

不难看出，Maven GAV 信息均按照之前的命令来设定，并没有依赖其他组件。接下来，为当前工程定义 Dubbo 服务接口。

5.2.2 定义 Dubbo 服务接口

Dubbo 服务接口是服务提供方与消费方的远程通讯契约，通常由普通的 Java 接口（interface）来声明，如 EchoService 接口：

```
package com.alibaba.cloud;  
  
public interface EchoService {  
    String echo(String message);  
}
```

该接口非常简单，仅有一个方法，接下来将 dubbo-sample-api 部署到本地 Maven 仓库。

5.2.3 部署 artifact - dubbo-sample-api

利用 Maven 命令，将 dubbo-sample-api 部署到本地 Maven 仓库：

```
% mvn clean install  
...  
[INFO] BUILD SUCCESS
```

注：如果读者所使用机器的 JDK 版本过高的话，可能会出现错误提示："不再支持源选项 5。请使用 7 或更高版本"。本例推荐选择 JDK 8 编译。

本地部署成功后，该 artifact 能被 Dubbo 服务提供者应用 dubbo-provider-sample 依赖。

5.2.4 依赖 artifact - dubbo-sample-api

将 artifact dubbo-sample-api 依赖信息添加到应用 dubbo-provider-sample 中的 pom.xml：

```
<!-- Dubbo 服务 artifact -->
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>dubbo-sample-api</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

依赖增加之后，下一步实现 Dubbo 服务 -EchoService？

5.2.5 实现 Dubbo 服务

在应用 `dubbo-provider-sample` 中的 `com.alibaba.cloud.dubboprovidersample` 包下创建实现类：

```
public class SimpleEchoService implements EchoService {

    @Override
    public String echo(String s) {
        return "[ECHO] " + s;
    }
}
```

其中，`@org.apache.dubbo.config.annotation.Service` 是 Dubbo 服务注解，仅声明该 Java 服务（本地）实现为 Dubbo 服务。因此，下一步需要将其配置 Dubbo 服务（远程）。

5.2.6 配置 Dubbo 服务提供方

在暴露 Dubbo 服务方面，推荐开发人员外部化配置的方式，即指定 Java 服务实现类的扫描基准包。

Dubbo Spring Cloud 继承了 Dubbo Spring Boot 的外部化配置特性，也可以通过标注 `@DubboComponentScan` 来实现基准包扫描。

同时，Dubbo 远程服务需要暴露网络端口，并设定通讯协议，完整的 `bootstrap.yaml` 配置如下所示：

```
dubbo:
  scan:
    # dubbo 服务扫描基准包
    base-packages: org.springframework.cloud.alibaba.dubbo.bootstrap
  protocol:
    # dubbo 协议
    name: dubbo
    # dubbo 协议端口（-1 表示自增端口，从 20880 开始）
    port: -1
  spring:
  cloud:
    nacos:
      # Nacos 服务发现与注册配置
      discovery:
        server-addr: 127.0.0.1:8848
```

以上 YAML 内容，上半部分为 Dubbo 的配置：

- `dubbo.scan.base-packages`: 指定 Dubbo 服务实现类的扫描基准包。
- `dubbo.protocol`: Dubbo 服务暴露的协议配置，其中子属性 `name` 为协议名称，`port` 为协议端口（-1 表示自增端口，从 20880 开始）。

下半部分则是 Spring Cloud 相关配置：

- `spring.application.name`: Spring 应用名称，用于 Spring Cloud 服务注册和发现。

该值在 Dubbo Spring Cloud 加持下被视作 `dubbo.application.name`，因此，无需再显示地配置 `dubbo.application.name`。

- `spring.cloud.nacos.discovery`: Nacos 服务发现与注册配置，其中子属性 `server-addr` 指定 Nacos 服务器主机和端口。

完成以上步骤后，还需编写一个 Dubbo Spring Cloud 引导类。

5.2.7 引导 Dubbo Spring Cloud 服务提供方应用

Dubbo Spring Cloud 引导类与普通 Spring Cloud 应用并无差别，如下所示：

```
@EnableDiscoveryClient@SpringBootApplicationpublic class DubboProviderSampleApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(DubboProviderSampleApplication.class);  
    }  
}
```

在引导 DubboProviderSampleApplication 之前，请提前启动 Nacos 服务器。当 DubboProviderSampleApplication 启动后，将应用 dubbo-provider-sample 将出现在 Nacos 控制台界面：



当 Dubbo 服务提供方启动后，下一步实现一个 Dubbo 服务消费方。

5.3 使用 Dubbo Spring Cloud 实现 Dubbo 服务消费方

由于 Java 服务仅为 EchoService、服务提供方应用 dubbo-provider-sample 以及 Nacos 服务器均已准备完毕。由于应用创建的步骤类似，构建消费方应用 dubbo-consumer-sample 的操作不再重复。

5.3.1 依赖 artifact - dubbo-sample-api

与服务提供方 Maven 工程类，需添加 artifact dubbo-sample-api 依赖，完整的组件 Maven 依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-dubbo</artifactId>
</dependency>

<!-- Dubbo 服务 artifact -->
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>dubbo-sample-api</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

5.3.2 配置 Dubbo 服务消费方

Dubbo 服务消费方配置与服务提供方类似, 当前应用 `dubbo-consumer-sample` 属于纯服务消费方, 因此, 所需的 `bootstrap.yaml` 文件配置更精简:

```
dubbo:
  cloud:
    subscribed-services: dubbo-provider-sample
  spring:
    cloud:
      nacos:
        # Nacos 服务发现与注册配置
      discovery:
        server-addr: 127.0.0.1:8848
```

对比应用 `dubbo-provider-sample`，除应用名称`spring.application.name`存在差异外，`dubbo-consumer-sample` 新增了属性`dubbo.cloud.subscribed-services`的设置。并且该值为服务提供方应用 “`dubbo-provider-sample`”。

`dubbo.cloud.subscribed-services`?: 用于服务消费方订阅服务提供方的应用名称的列表，若需订阅多应用，使用 “,” 分割。不推荐使用默认值为 “*”，它将订阅所有应用。

当应用使用属性 `dubbo.cloud.subscribed-services` 默认值时，日志中将会输出一行警告：

Current application will subscribe all services(size:x) in registry, a lot of memory and CPU cycles may be used, thus it's strongly recommend you using the externalized property 'dubbo.cloud.subscribed-services' to specify the services.

由于当前应用属于 Web 应用，它会默认地使用 8080 作为 Web 服务端口，如果 `dubbo-provider-sample` 和 `dubbo-consumer-sample` 在本地同步部署的话，两者 Web 端口会出现冲突，需调整 `dubbo-consumer-sample application.properties` 中的`server.port`和 `management.server.port`?:

```
# 应用服务 WEB 访问端口 server.port=9090
# Actuator Web 访问端口 management.server.port=9091
```

5.3.3 引导 Dubbo Spring Cloud 服务消费方应用

为了减少实现步骤，编辑引导类 `DubboConsumerSampleApplication`将 Dubbo 服务消费以及引导功能合二为一：

```
@EnableDiscoveryClient@EnableAutoConfiguration@RestControllerpublic class
DubboConsumerSampleApplication {

    @Reference

    private EchoService echoService;

    @GetMapping("/echo")

    public String echo(String message) {
```

```
        return echoService.echo(message);
    }

    public static void main(String[] args) {
        SpringApplication.run(DubboConsumerSampleApplication.class);
    }
}
```

运行该引导类，通过 `curl?` 命令访问 REST 资源 `/echo?`:

```
% curl "http://127.0.0.1:9090/echo?message=Hello,World"
[ECHO] Hello,World
```

不难发现，Dubbo 服务提供方应用 `dubbo-provider-sample` 的 `EchoService?` 计算结果返回到 `dubbo-consumer-sample` 中的 REST 资源 `/echo`。同时，在 `dubbo-provider-sample` 应用日志出现了以下内容：

```
[DUBBO] The connection of /x.x.x.x:64051 -> /x.x.x.x:20881 is established., dubbo version: 2.7.6,
current host: x.x.x.x
```

说明 Dubbo 服务消费端应用 `dubbo-consumer-sample`（端口：64051）向服务提供方应用 `dubbo-provider-sample` 建立连接（端口：20881）。

服务熔断和限流

前言：为什么需要流控降级

我们的生产环境经常会出现一些不稳定的情况，如：

- 大促时瞬间洪峰流量导致系统超出最大负载，load 飙高，系统崩溃导致用户无法下单
- “黑马”热点商品击穿缓存，DB 被打垮，挤占正常流量
- 调用端被不稳定服务拖垮，线程池被占满，导致整个调用链路卡死

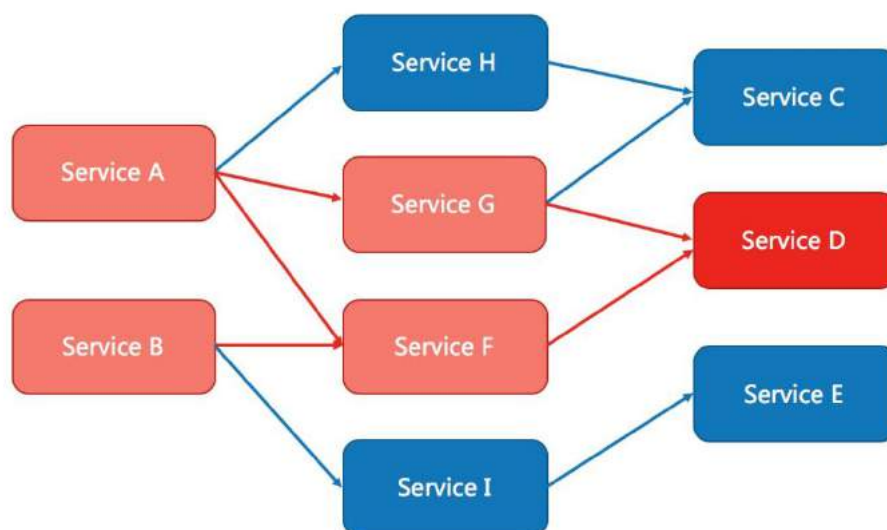
这些不稳定的场景可能会导致严重后果。大家可能想问：如何做到均匀平滑的用户访问？如何预防流量过大或服务不稳定带来的影响？这时候我们就要请出微服务稳定性的法宝——高可用流量防护，其中重要的手段就是流量控制和熔断降级，它们是保障微服务稳定性重要的一环。

为什么需要流量控制？

流量是非常随机性的、不可预测的。前一秒可能还风平浪静，后一秒可能就出现流量洪峰了（例如双十一零点的场景）。然而我们系统的容量总是有限的，如果突然而来的流量超过了系统的承受能力，就可能会导致请求处理不过来，堆积的请求处理缓慢，CPU/Load 飙高，最后导致系统崩溃。因此，我们需要针对这种突发的流量来进行限制，在尽可能处理请求的同时来保障服务不被打垮，这就是流量控制。

为什么需要熔断降级？

一个服务常常会调用别的模块，可能是另外的一个远程服务、数据库，或者第三方 API 等。例如，支付的时候，可能需要远程调用银联提供的 API；查询某个商品的价格，可能需要进行数据库查询。然而，这个被依赖服务的稳定性是不能保证的。如果依赖的服务出现了不稳定的情况，请求的响应时间变长，那么调用服务的方法的响应时间也会变长，线程会产生堆积，最终可能耗尽业务自身的线程池，服务本身也变得不可用。



现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路。以上的问题在链路调用中会产生放大的效果。复杂链路中的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。因此我们需要对不稳定的弱依赖服务进行熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。

Sentinel: 高可用护航的利器

Sentinel 是阿里巴巴开源的，面向分布式服务架构的高可用防护组件，主要以流量为切入点，从流量控制、流量整形、熔断降级、系统自适应保护、热点防护等多个维度来帮助开发者保障微服务的稳定性。**Sentinel** 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀、冷启动、消息削峰填谷、自适应流量控制、实时熔断下游不可用服务等，是保障微服务高可用的利器，原生支持 Java/Go/C++ 等多种语言，并且提供 Istio/Envoy 全局流控支持来为 Service Mesh 提供高可用防护的能力。

Sentinel 的技术亮点：

- 高度可扩展能力：基础核心 + SPI 接口扩展能力，用户可以方便地扩展流控、通信、监控等功能。
- 多样化的流量控制策略（资源粒度、调用关系、流控指标、流控效果等多个维度），提供分布式集群流控的能力。
- 热点流量探测和防护。

- 对不稳定服务进行熔断降级和隔离。
- 全局维度的系统负载自适应保护，根据系统水位实时调节流量。
- 覆盖 API Gateway 场景，为 Spring Cloud Gateway、Zuul 提供网关流量控制的能力。
- 实时监控和规则动态配置管理能力。

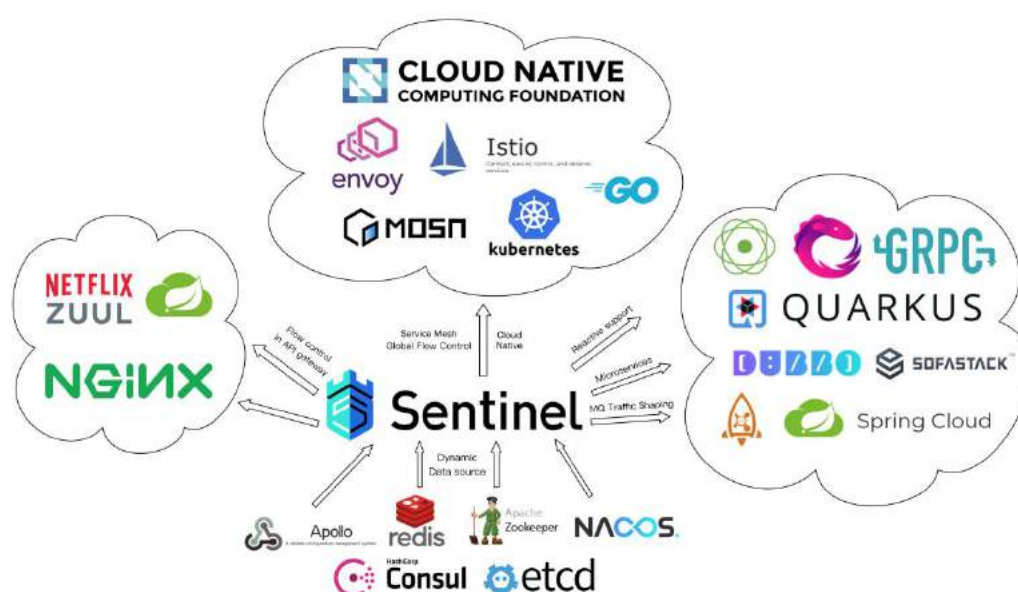


一些普遍的使用场景：

- 在服务提供方（Service Provider）的场景下，我们需要保护服务提供方自身不被流量洪峰打垮。这时候通常根据服务提供方的服务能力进行流量控制，或针对特定的服务调用方进行限制。我们可以结合前期压测评估核心接口的承受能力，配置 QPS 模式的限流，当每秒的请求量超过设定的阈值时，会自动拒绝多余的请求。
- 为了避免调用其他服务时被不稳定的服务拖垮自身，我们需要在服务调用端（Service Consumer）对不稳定服务依赖进行隔离和熔断。手段包括信号量隔离、异常比例降级、RT 降级等多种手段。
- 当系统长期处于低水位的情况下，流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。这时候我们可以借助 Sentinel 的 WarmUp 流控模式控制通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，而不是一瞬间全部放行。这样可以给冷系统一个预热的时间，避免冷系统被压垮。
- 利用 Sentinel 的匀速排队模式进行“削峰填谷”，把请求突刺均摊到一段时间内，让系统负载保持在请求处理水位之内，同时尽可能地处理更多请求。

- 利用 Sentinel 的网关流控特性，在网关入口处进行流量防护，或限制 API 的调用频率。

Sentinel 有着丰富的开源生态。Sentinel 开源不久就被纳入 CNCF Landscape 版图，并且也成为 Spring Cloud 官方推荐的流控降级组件之一。社区提供 Spring Cloud、Dubbo、gRPC、Quarkus 等常用微服务框架的适配，开箱即用；同时支持 Reactive 生态，支持 Reactor、Spring WebFlux 异步响应式架构。Sentinel 也在逐渐覆盖 API Gateway 和 Service Mesh 场景，在云原生架构中发挥更大的作用。



在原来的 Spring Cloud Netflix 系列中，有自带的熔断组件 Hystrix，是 Netflix 公司提供的的一个开源的组件，提供了熔断、隔离的这些特性，不过 Hystrix 在 2018 年 11 月份开始，就不再迭代开发，进入维护的模式。同年开源的 [Spring Cloud Alibaba](#) (SCA) 提供了一站式的解决方案，默认为 Sentinel 整合了 Spring Web、RestTemplate、FeignClient 和 Spring WebFlux。Sentinel 在 Spring Cloud 生态中，不仅补全了 Hystrix 在 Servlet、RestTemplate 和 API Gateway 这一块空白，而且还完全兼容了 Hystrix 在 FeignClient 中限流降级的用法，并且支持运行时灵活地配置和调整限流降级规则。同时 SCA 还集成了 Sentinel 提供的 API gateway 流控模块，可以无缝支持 Spring Cloud Gateway 和 Zuul 网关的流控降级。

Spring Cloud Alibaba Sentinel 服务限流/熔断实战

下面到了动手时间了！我们结合一个实例来对 Spring Cloud 服务限流/熔断进行实战。我们的实例项目由四个模块构成：

- service-api: 服务接口定义，供 consumer/provider 引用。
- dubbo-provider: Dubbo 服务端，对外提供一些服务。
- web-api-demo: Spring Boot Web 应用，其中的一些 API 会作为 consumer 来调用 dubbo-provider 获取结果。里面一共定义了三个 API path：

- /demo/hello: 接受一个 name 参数，会 RPC 调用后端的 FooService:sayHello(name) 方法。
- /demo/time: 调用后端的 FooService:getCurrentTime 方法获取当前时间；里面可以通过 slow 请求参数模拟慢调用。
- /demo/bonjour/{name}: 直接调用本地 DemoService 服务。
- demo-gateway: Spring Cloud Gateway 网关，作为整个项目的访问入口，将流量转发至后端服务或第三方服务。我们的入口 URL 访问都会经过该 API gateway。demo-gateway 的路由配置如下：

```
spring:
  cloud:
    gateway:
      enabled: true
      discovery:
        locator:
          # route ID 转化小写
          lower-case-service-id: true
      routes:
        - id: foo-service-route
          uri: http://localhost:9669/
          predicates:
            - Path=/demo/**
        - id: httpbin-route
          uri: https://httpbin.org
          predicates:
            - Path=/httpbin/**
          filters:
            - RewritePath=/httpbin/(?<segment>.*), /${segment}
```

这个路由配置包含两个路由：

- `foo-service-route`: 会将 `/demo/` 开头的访问路由到 `localhost:9669` 后端服务上面，即对应我们的 Web 服务。我们访问示例中的 API 都会经过这个路由，比如 `localhost:8090/demo/time`。
- `httpbin-route`: 这是一个示例路由，它会将 `/httpbin/` 开头的访问路由到 <https://httpbin.org> 这个示例网站上，比如 `localhost:8090/httpbin/json` 实际会映射到 `https://httpbin.org/json` 上面。

同时我们的环境也包含启动好的 Sentinel 控制台，可以直接访问并供各个服务接入。
对应的地址：TODO

下面我们来一步一步操作接入 SCA Sentinel 并通过控制台/Nacos 动态数据源配置流控降级规则来验证效果。

spring-cloud-alibaba-dependencies 配置

首先第一步我们在项目的父 pom 里面导入最新版本的 `spring-cloud-alibaba-dependencies`，这样我们在实际引入 SCA 相关依赖的时候就不需要指定版本号了：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>2.2.2.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies></dependencyManagement>
```

服务接入 SCA Sentinel

首先我们分别为三个服务模块引入 Spring Cloud Alibaba Sentinel 依赖：

```
<dependency>  
  <groupId>com.alibaba.cloud</groupId>  
  <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId></dependency>
```

starter 会自动对 Sentinel 的适配模块进行配置，只需要简单的配置即可快速接入 Sentinel 并连接到 Sentinel 控制台。

对于 Dubbo 服务，我们还需要额外引入 Dubbo 的适配模块。Sentinel 为 Apache Dubbo 提供开箱即用的整合模块，仅需引入 sentinel-apache-dubbo-adapter 依赖即可接入 Dubbo 自动埋点统计（支持 provider 和 consumer）：

```
<dependency>  
  <groupId>com.alibaba.csp</groupId>  
  <artifactId>sentinel-apache-dubbo-adapter</artifactId>  
  <version>1.8.0</version></dependency>
```

我们在 web-api-demo 和 dubbo-provider 两个应用的 pom 文件添加 adapter 依赖，这样两个应用的 Dubbo consumer/provider 接口就可以自动被 Sentinel 统计。

对于 Spring Cloud Gateway、Zuul 1.x 等网关，我们还需要在上面 SCA 依赖的基础上额外引入 spring-cloud-alibaba-sentinel-gateway 依赖：

```
<dependency>  
  <groupId>com.alibaba.cloud</groupId>  
  <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId></dependency>
```

这个依赖会自动为网关添加 Sentinel 相关的配置，从而可以让 API gateway 自动接入 Sentinel。我们在 demo-gateway 应用的 pom 文件里面添加这个依赖，这样我们的 gateway 应用就可以接入 Sentinel 了。

引入依赖之后，我们只需要进行简单的配置即可快速接入 Sentinel 控制台。我们可以在 `application.properties` 文件里面配置应用名和连接控制台的地址，以 `web-api-demo` 为例：

```
spring.application.name=foo-web
spring.cloud.sentinel.transport.dashboard=localhost:8080
```

其中 `spring.application.name` 相信大家都比较熟悉了，这里 Spring Cloud Alibaba Sentinel 会自动提取这个值作为接入应用的 `appName`。同时我们通过 `spring.cloud.sentinel.transport.dashboard` 来配置要连接的控制台地址和端口。

完成以上的配置后，我们可以依次启动 `dubbo-provider`、`web-api-demo` 和 `demo-gateway` 应用，并通过网关入口访问 `localhost:8090/demo/time` 获取当前时间。触发服务后，我们稍后可以在 Sentinel 控制台看到我们的三个应用，可以在监控页面看到访问信息，代表接入成功。



我们可以在每个应用的簇点链路页面看到当前应用的一些埋点调用，比如 Web 应用可以看到 Web URL 和 Dubbo consumer 调用：



流控规则

下面我们来配一条最简单的流控规则。在 Dubbo provider 端，我们进入簇点链路页面，针对 `com.alibaba.csp.sentinel.demo.dubbo.FooService:getCurrentTime(boolean)` 这个服务调用配置限流规则（需要有过访问量才能看到）。我们配一条 QPS 为 1 的流控规则，这代表针对该服务方法的调用每秒钟不能超过 1 次，超出会直接拒绝。



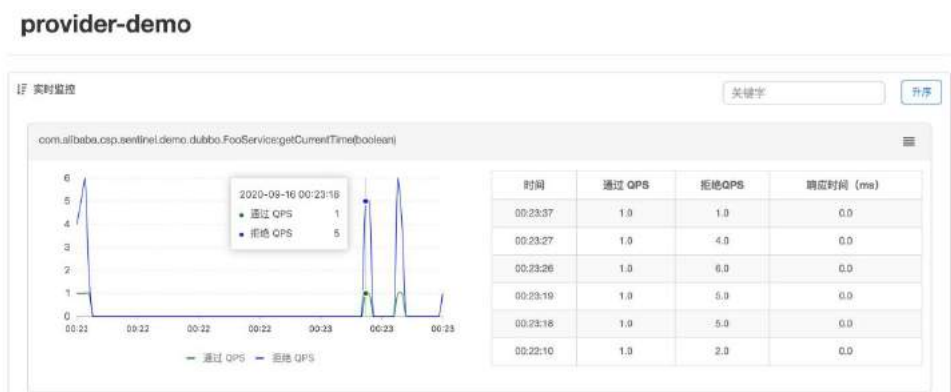
点击“新增”按钮，成功添加规则。我们可以在浏览器反复请求 `localhost:8090/demo/time`（频率不要太慢），可以看到会出现限流异常信息（Dubbo provider 默认的限流处理逻辑是抛出异常，该异常信息由 Dubbo 直接返回，并由 Spring 展示为默认 `error` 页面）：

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Sep 16 00:22:10 CST 2020
There was an unexpected error (type=Internal Server Error, status=500).
SentinelBlockException: FlowException

同时我们也可以在“实时监控”页面看到实时的访问量和拒绝量：



我们同样也可以在 Web API 处配置限流规则，观察效果。Spring Web 默认的限流处理逻辑是返回默认的提示信息（Blocked by Sentinel），状态码为 429。在后面的章节我们会介绍如何自定义流控处理逻辑。

了解了限流的基本用法，大家可能想问：生产环境我需要针对每个接口都去配置流控规则吗？阈值不会配怎么办？其实，限流降级的配置是需要结合容量规划、依赖梳理来做的。我们可以借助 JMeter 或 [阿里云 PTS](#) 等压测工具对我们的服务进行全链路压测，了解每个服务的最大承受能力，来确定核心接口的最大容量并作为 QPS 阈值。

网关流控规则

Sentinel 对 API Gateway 流控的场景进行了定制，支持针对网关的路由（如上面 gateway 定义的 foo-service-route）或自定义的 API 分组进行流控，支持针对请求属性（如某个 header）进行流控。用户可以在 Sentinel 控制台 自定义 API 分组，可以看做是一些 URL 匹配的组合。比如我们可以定义一个 API 叫 my_api，请求 path 模式为 /foo/** 和 /baz/** 的都归到 my_api 这个 API 分组下面。限流的时候可以针对这个自定义的 API 分组维度进行限流。



下面我们在控制台针对 gateway 配置一条网关流控规则。我们可以看到 API Gateway 的控制台页面与普通应用的页面有一些差异，这些就是针对网关场景的定制。Sentinel 网关流控规则支持提取某个 route 的请求属性，包括 remote IP、header、URL 参数、cookie 等，支持自动统计其中的热点值并分别进行限制，也支持针对某个具体值进行限制（比如给某个 uid 限量）。

我们给 foo-service-route 这个路由配一条针对请求属性的网关流控规则。这条规则会针对 URL 参数中提取出来的每个热点 uid 参数分别进行限制，每分钟的请求量最多允许 2 次。



保存规则后，我们可以构造一些向后端服务的请求，携带上不同的 uid 参数（即使没有用到），比如 `localhost:8090/demo/time?uid=xxx`。我们可以观察到，每个 uid 的访问每分钟超出两次后会出现限流页面。

关于 Sentinel 网关流控的详细配置指南和实现原理请参考 [网关流控文档](#)。

熔断降级规则

熔断降级通常用于自动切断不稳定的服务，防止调用方被拖垮导致级联故障。熔断降级规则通常在调用端，针对弱依赖调用进行配置，在熔断时返回预定义好的 fallback 值，这样可以保证核心链路不被不稳定的旁路影响。

Sentinel 提供以下几种熔断策略：

- 慢调用比例 (SLOW_REQUEST_RATIO): 选择以慢调用比例作为阈值，需要设置允许的慢调用 RT（即最大的响应时间），请求的响应时间大于该值则统计为慢调用。当单位统计时长（statIntervalMs，默认为 1s）内请求数目大于设置的最小请求数目，并且慢调用的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求响应时间小于设置的慢调用 RT 则结束熔断，若大于设置的慢调用 RT 则会再次被熔断。
- 异常比例 (ERROR_RATIO): 当单位统计时长内请求数目大于设置的最小请求数目，并且异常的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%。
- 异常数 (ERROR_COUNT): 当单位统计时长内的异常数目超过阈值之后会自动进行熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。

下面我们来在 Web 应用中针对 Dubbo consumer 来配置慢调用熔断规则，并模拟慢调用来观察效果。我们在 `web-api-demo` 中针对 `com.alibaba.csp.sentinel.demo.dubbo.FooService` 服务调用配置熔断降级规则。



控制台配置的统计时长默认为 1s。在上面的这条规则中，我们设定慢调用临界值为 50ms，响应时间超出 50ms 即记为慢调用。当统计时长内的请求数 ≥ 5 且慢调用的比例超出我们配置的阈值（80%）就会触发熔断，熔断时长为 5s，经过熔断时长后会允许一个请求探测通过，若请求正常则恢复，否则继续熔断。

我们的实例中 `/demo/time` API 可以通过 `slow` 请求参数模拟慢调用，当 `slow=true` 时该请求耗时会超过 100ms。我们可以用 `ab` 等压测工具或脚本，批量请求 `localhost:8090/demo/time?slow=true`，可以观察到熔断的返回：

```
+ ~ ./flow.sh "http://localhost:8090/demo/time?slow=true"
1600275239046
1600275239169
1600275239294
1600275239418
1600275239543
{"timestamp": "2020-09-16T16:53:59.654+0000", "status": 500, "error": "Internal Server Error", "message": "SentinelBlockException: DegradException", "path": "/demo/time"}
{"timestamp": "2020-09-16T16:53:59.715+0000", "status": 500, "error": "Internal Server Error", "message": "SentinelBlockException: DegradException", "path": "/demo/time"}
```

如果我们一直模拟慢调用，我们可以观察到熔断后每 5s 会允许通过一个请求，但该请求仍然是慢调用，会重新打回熔断，无法恢复。我们可以在触发熔断后，等待一段时间后手动发一个不带 `slow=true` 的正常请求，然后再进行请求，可以观察到熔断恢复。

需要注意的是，即使服务调用方引入了熔断降级机制，我们还是需要在 HTTP 或 RPC 客户端配置请求超时时间，来做一个兜底的防护。

注解方式自定义埋点

刚才我们看到的埋点都是 Sentinel 适配模块提供的自动埋点。有的时候自动埋点可能没法满足我们的需求,我们希望在某个业务逻辑的位置进行限流,能不能做到呢? 当然可以! Sentinel 提供两种方式进行自定义埋点: SphU API 和 @SentinelResource 注解,前者最为通用但是代码比较繁杂,耦合度较高;注解方式侵入性较低,但有使用场景的限制。这里我们来动手在 Web 应用的 DemoService 上添加注解,来达到针对本地服务埋点的目标。

在 DemoService 中我们实现了一个简单的打招呼的服务:

```
@Servicepublic class DemoService {

    public String bonjour(String name) {

        return "Bonjour, " + name;

    }

}
```

下面我们给 bonjour 这个函数添加 @SentinelResource 注解,注解的 value 代表这个埋点的名称 (resourceName), 会显示在簇点链路/监控页面。

```
@SentinelResource(value = "DemoService#bonjour")

public String bonjour(String name)
```

加上该注解后,再通过网关访问 /demo/bonjour/{name} 这个 API 的时候,我们就可以在簇点链路页面看到我们自定义的 DemoService#bonjour 埋点了。

web-demo

簇点链路 列表视图

簇点链路 30.225.184.60:8721 关键字 查询

资源名	通过QPS	拒绝QPS	线程数	平均RT	分钟通过	分钟拒绝	操作
sentinel_default_context	0	0	0	0	0	0	+ 流控 + 降级 + 热点 + 授权
sentinel_spring_web_context	0	0	0	0	0	0	+ 流控 + 降级 + 热点 + 授权
/demo/time	0	0	0	0	0	0	+ 流控 + 降级 + 热点 + 授权
/demo/bonjour/{name}	0	0	0	0	0	0	+ 流控 + 降级 + 热点 + 授权
DemoService#bonjour	0	0	0	0	0	0	+ 流控 + 降级 + 热点 + 授权

共 7 条记录, 每页 16 条记录

添加注解埋点只是第一步。一般在生产环境中，我们希望在这些自定义埋点发生限流的时候，有一些 fallback 逻辑，而不是直接对外抛出异常。这里我们可以写一个 fallback 函数：

```
public String bonjourFallback(Throwable t) {  
    if (BlockException.isBlockException(t)) {  
        return "Blocked by Sentinel: " + t.getClass().getSimpleName();  
    }  
    return "Oops, failed: " + t.getClass().getCanonicalName();  
}
```

我们的 fallback 函数接受一个 Throwable 参数，可以从中获取异常信息。Sentinel 注解的 fallback 会捕获业务异常和流控异常（即 BlockException 及其子类），我们可以在 fallback 逻辑里面进行相应的处理（如日志记录），并返回 fallback 的值。

注意：Sentinel 注解对 fallback 和 blockHandler 函数的方法签名有要求，具体请参考[此处文档](#)。

写好 fallback 函数的实现后，我们在 @SentinelResource 注解里面指定一下：

```
@SentinelResource(value = "DemoService#bonjour", defaultFallback = "bonjourFallback")  
public String bonjour(String name)
```

这样当我们自定义的 DemoService#bonjour 资源被限流或熔断的时候，请求会走到 fallback 的逻辑中，返回 fallback 结果，而不会直接抛出异常。我们可以配一个 QPS=1 的限流规则，然后快速请求后观察返回值：

```
? ~ curl http://localhost:8090/demo/bonjour/SentinelBonjour, Sentinel  
? ~ curl http://localhost:8090/demo/bonjour/SentinelBlocked by Sentinel: FlowException
```

注意：使用 @SentinelResource 注解要求对应的类必须由 Spring 托管（即为 Spring bean），并且不能是内部调用（没法走到代理），不能是 private 方法。Sentinel 注解生效依赖 Spring AOP 动态代理机制。

配置自定义的流控处理逻辑

Sentinel 的各种适配方式均支持自定义的流控处理逻辑。以 Spring Web 适配为例，我们只需要提供自定义的 `BlockExceptionHandler` 实现并注册为 bean 即可为 Web 埋点提供自定义处理逻辑。其中 `BlockExceptionHandler` 的定义如下：

```
public interface BlockExceptionHandler {  
    // 在此处处理限流异常，可以跳转到指定页面或返回指定的内容  
    void handle(HttpServletRequest request, HttpServletResponse response, BlockException e)  
        throws Exception;  
}
```

我们的 Web 应用中提供了 Web 埋点自定义流控处理逻辑的示例：

```
@Configurationpublic class SentinelWebConfig {  
    @Bean  
    public BlockExceptionHandler sentinelBlockExceptionHandler() {  
        return (request, response, e) -> {  
            // 429 Too Many Requests  
            response.setStatus(429);  
  
            PrintWriter out = response.getWriter();  
            out.print("Oops, blocked by Sentinel: " + e.getClass().getSimpleName());  
            out.flush();  
            out.close();  
        };  
    }  
}
```

该 handler 会获取流控类型并打印返回信息，返回状态码为 429。我们可以根据实际的业务需求，配置跳转或自定义的返回信息。

对于注解方式，我们上一节已经提到，可以指定 `fallback` 函数来处理流控异常和业务异常，这里不再展开讲解；对于 Dubbo 适配，我们可以通过 `DubboAdapterGlobalConfig` 注册 `provider/consumer fallback` 来提供自定义的流控处理逻辑；对于 Spring Cloud Gateway 适配，我们可以注册自定义的 `BlockRequestHandler` 实现类来为网关流控注册自定义的处理逻辑。

对 Spring Cloud 其他组件的支持

Spring Cloud Alibaba Sentinel 还提供对 Spring Cloud 其它常用组件的支持，包括 RestTemplate、Feign 等。篇幅所限，我们不展开实践。大家可以参考 [Spring Cloud Alibaba 文档](#) 来进行接入和配置。

如何选择流控降级组件

讲到这里，大家可能会有疑问：Sentinel 和其它同类产品（如 Hystrix）相比有什么优缺点？是否有必要迁移到 Sentinel？如何快速迁移？以下是 Sentinel 与其它 fault-tolerance 组件的对比：

	Sentinel	Hystrix	resilience4j
隔离策略	信号量隔离（并发控制）	线程池隔离/信号量隔离	信号量隔离
熔断降级策略	基于慢调用比例、异常比例、异常数	基于异常比例	基于异常比例、响应时间
实时统计实现	滑动窗口（LeapArray）	滑动窗口（基于 Rx Java）	Ring Bit Buffer
动态规则配置	支持多种数据源	支持多种数据源	有限支持
扩展性	多个扩展点	插件的形式	接口的形式
基于注解的支持	支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	有限的支持	Rate Limiter
流量整形	支持预热模式与匀速排队控制效果	不支持	简单的 Rate Limiter 模式
系统自适应保护	支持	不支持	不支持
多语言支持	Java/Go/C++	Java	Java

	Sentinel	Hystrix	resilience4j
Service Mesh 支持	支持 Envoy/Istio	不支持	不支持
控制台	提供开箱即用的控制台，可配置规则、实时监控、机器发现等	简单的监控查看	不提供控制台，可对接其它监控系统

总结

通过本教程，我们了解了流控降级作为高可用防护手段的重要性，了解了 Sentinel 的核心特性和原理，并通过动手实践学习了如何快速接入 SCA Sentinel 来为微服务进行流控降级。Sentinel 还有着非常多的高级特性等着大家去发掘，如热点防护、集群流控等，大家可以参考 [Sentinel 官方文档](#)来了解更多的特性和场景。

那么是不是服务的量级很小就不用进行限流防护了呢？是不是微服务的架构比较简单就不用引入熔断保护机制了呢？其实，这与请求的量级、架构的复杂程度无关。很多时候，可能正是一个非常边缘的服务出现故障而导致整体业务受影响，造成巨大损失。我们需要具有面向失败设计的意识，在平时就做好容量规划和强弱依赖的梳理，合理地配置流控降级规则，做好事前防护，而不是在线上出现问题以后再进行补救。

同时，我们也在阿里云上提供了 Sentinel 的企业版本 [AHAS Sentinel](#)，提供开箱即用的企业级高可用防护能力。与开源版本相比，AHAS 还提供以下的专业能力：

- 可靠的实时监控和历史秒级监控数据查询，包含接口维度的 QPS、响应时间及系统。load、CPU 使用率等指标，支持按照调用类型分类，支持同比/环比展示。
- Top K 接口监控统计，快速了解系统的慢调用和大流量接口；热力图概览，快速定位不稳定的机器。
- Java Agent 方式/K8s Java 应用零侵入快速接入，支持近 20 种主流框架和 API Gateway。
- 全自动托管、高可用的集群流量控制。
- Nginx 流量控制，支持规则动态配置、集群流控。

欢迎大家体验云上企业版本的 Sentinel，同时也欢迎大家多多参与社区贡献，一起帮助社区更好地进行演进。

分布式消息（事件）驱动

1. 简介

事件驱动架构(Event-driven 架构, 简称 EDA)是软件设计领域内的一套程序设计模型。这套模型的意义是所有的操作通过事件的发送/接收来完成。举个例子, 比如一个订单的创建在传统软件设计中服务端通过接口暴露创建订单的动作, 然后客户端访问创建订单。在事件驱动设计里, 订单的创建通过接收订单事件来完成, 这个过程中有事件发送者和事件接受者这两个模块, 事件发送者的作用是发送订单事件, 事件接受者的作用的接收订单事件。Spring Cloud Stream 是一套基于消息的事件驱动开发框架, 它提供了一套全新的消息编程模型, 此模型屏蔽了底层具体消息中间件的使用方式。开发者们使用这套模型可以完成基于消息的事件驱动应用开发。

2. 学习目标

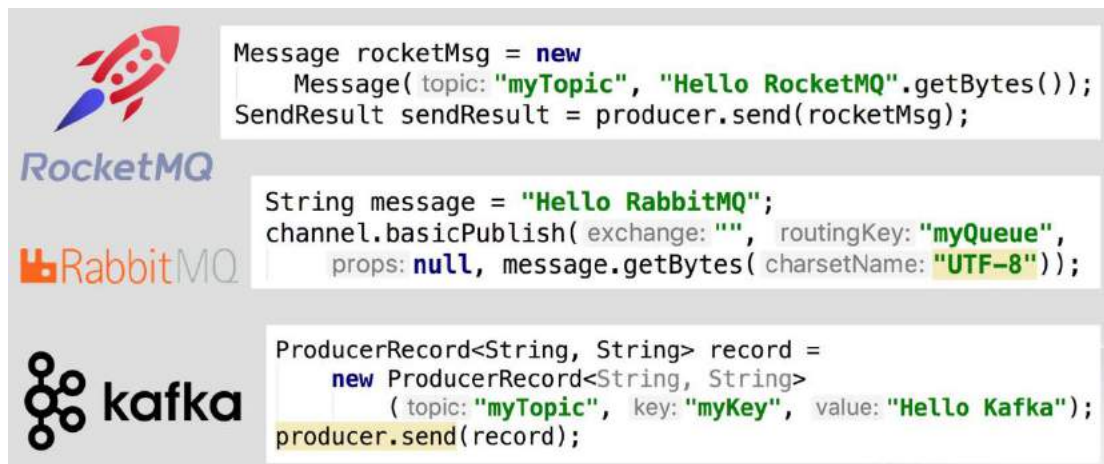
- 掌握 Spring 对消息的编程模型封装
- 掌握 RocketMQ 整合 Spring Cloud Stream 完成消息的发送和接收
- 掌握 RocketMQ 整合 Spring Cloud Bus 完成远程事件的发送和接收

3. 详细内容

- 概念理解: 指导读者理解 Spring 的消息编程模型
- 消息发送/接收: 实战 Spring Cloud Stream RocketMQ Binder
- 事件发送/接收: 实战 Spring Cloud Bus RocketMQ

4. 理解 Spring 消息编程模型

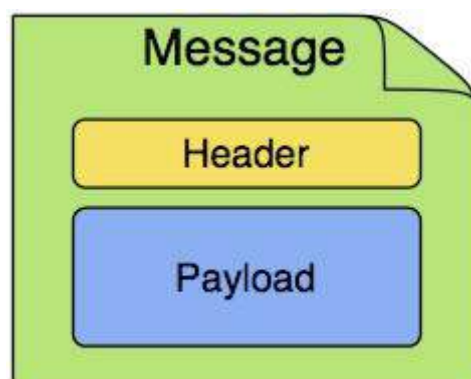
首先我们来看这个场景, 不同的消息中间件发送消息的代码:



每个消息中间件都有自己的消息模型编程，他们的代码编写方式都不一致。同样地，在消息的订阅方面，也是不同的代码。这个时候如果某天想把 Kafka 切换到 RocketMQ，必须得修改大量代码。

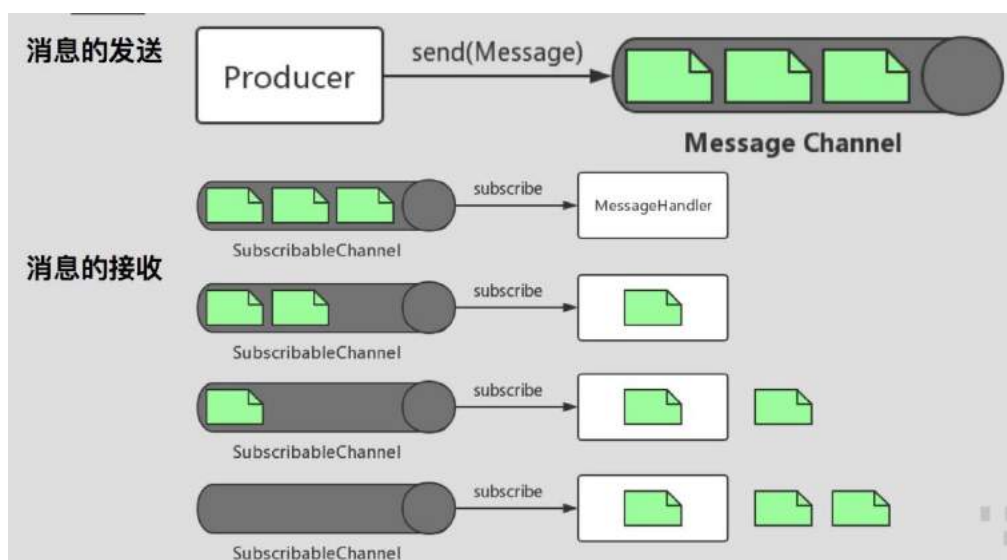
Spring 生态里有两个消息相关的模块和项目，分别是 spring-messaging 模块和 Spring Integration 项目，它们对消息的编程模型进行了统一，不论是 Apache RocketMQ 的 Message，或者是 Apache Kafka 的 ProducerRecord，都被统一称为 org.springframework.messaging.Message 接口。

Message 接口有两个方法，分别是 getPayload 以及 getHeaders 用于获取消息体以及消息头。如图所示，这也意味着一个消息 Message 由 Header 和 Payload 组成：

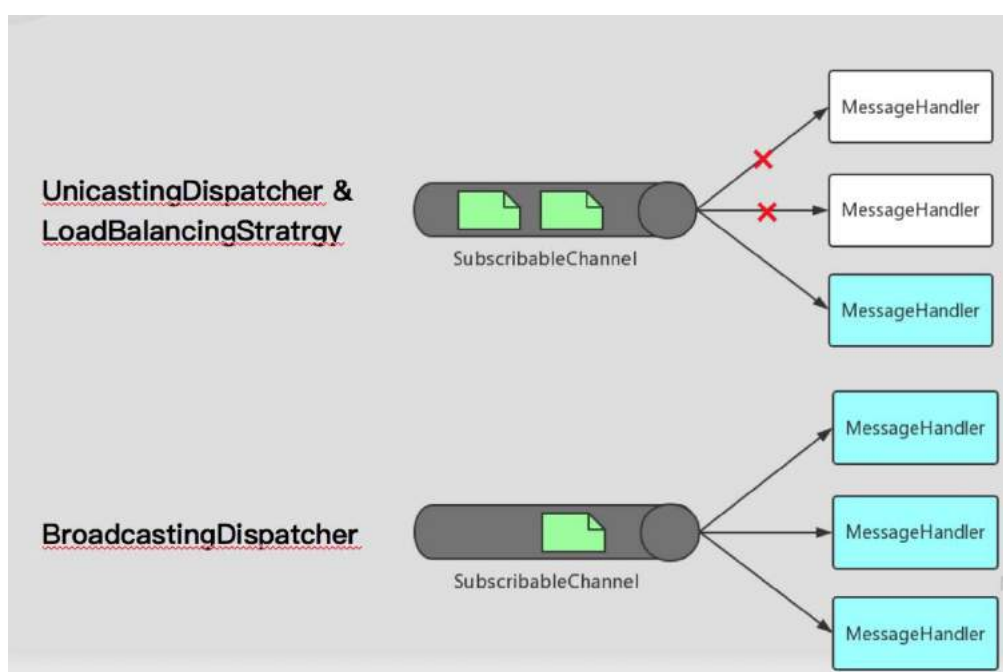


Payload 是一个泛型，意味是消息体可以放任意数据类型。Header 是一个 MessageHeaders 类型的消息头。

有了消息之后，这个消息被发送到哪里呢？Spring 提供了消息通道 MessageChannel 的概念。消息可以被发送到消息通道里，然后再通过消息处理器 MessageHandler 去处理消息通道里的消息：



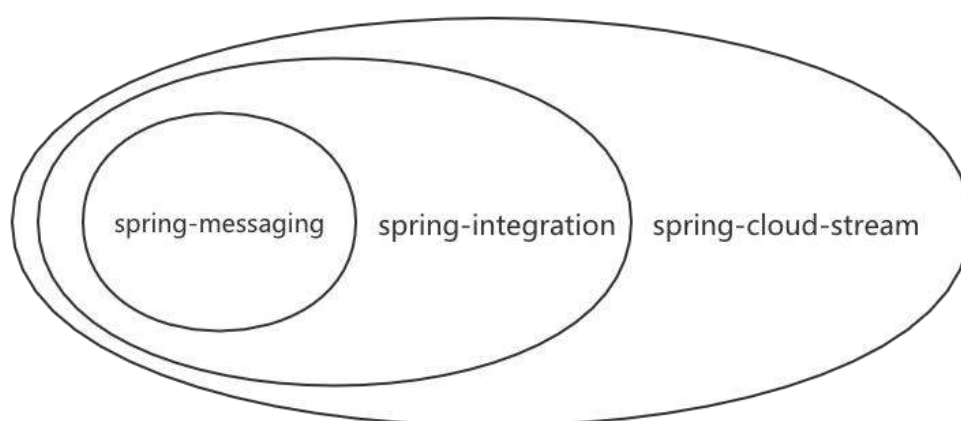
消息处理这里又会遇到一个问题。如果消息通道里只有 1 个消息，但是消息处理器有 N 个，这个时候要被哪个消息处理器处理呢？这里又涉及一个消息分发器的问题。UnicastingDispatcher 表示单播的处理方式，消息会通过负载均衡被分发到某一个消息处理器上，BroadcastingDispatcher 表示广播的方式，消息会被所有的消息处理器处理。



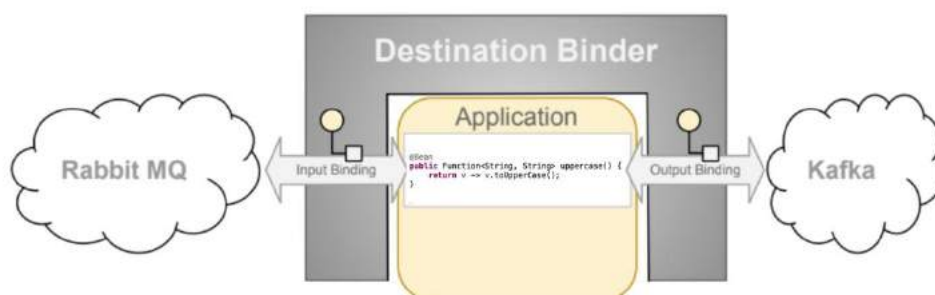
5. Spring Cloud Stream

Spring Cloud Stream 是一套基于消息的事件驱动开发框架。

Spring Cloud Stream 在 Spring Integration 项目的基础上再进行了一些封装，提出一些新的概念，让开发者能够更简单地使用这套消息编程模型。如图所示，这是三者之间的关系：



如下图所示，这是 Spring Cloud Stream 的编程模型。通过 RabbitMQ Binder 构建 input Binding 用于读取 RabbitMQ 上的消息，将 payload 内容转成大写再通过 Kafka Binder 构建的 output Binding 写入到 Kafka 中。图上中间的 [4]()行非常简单的代码就可以完成从 RabbitMQ 读取消息再写入到 Kafka 的动作。



以下代码是使用 Spring Cloud Stream 以最简单的方式完成消息的发送和接收：

```
@SpringBootApplication@EnableBinding({Source.class, Sink.class}) // @public class
SCSApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder().sources(SCSApplication.class)
            .web(WebApplicationType.NONE).run(args);
    }

    @Autowired
    Source source; // ②

    @Bean
    public CommandLineRunner runner() {
        return (args) -> {
            source.output().send(MessageBuilder.withPayload("custom payload").setHeader("k1",
"v1").build()); // ③
        };
    }

    @StreamListener(Sink.INPUT) // ④
    @SendTo(Source.OUTPUT) // ⑤
    public String receive(String msg) {
        return msg.toUpperCase();
    }
}
```

1. 使用 `@EnableBinding` 注解，注解里面有两个参数 `Source` 和 `Sink`，它们都是接口。`Source` 接口内部有个 `MessageChannel` 类型返回值的 `output` 方法，被 `@Output` 注解修饰表示这是一个 `Output Binding`；`Sink` 接口内部有个 `SubscribableChannel` 类型返回值的 `input` 方法，被 `@Input` 注解修饰表示这是一个 `Input Binding`。`@EnableBinding` 注解会针对这两个接口生成动态代理。
2. 注入 `@EnableBinding` 注解对于 `Source` 接口生成的动态代理。
3. 使用 `@EnableBinding` 注解对于 `Source` 接口生成的动态代理内部的 `MessageChannel` 发送一条消息。最终消息会被发送到消息中间件对应的 `topic` 里。
4. `@StreamListener` 注解订阅 `@EnableBinding` 注解对于 `Sink` 接口生成的动态代理内部的 `SubscribableChannel` 中的消息，这里会订阅到消息中间件对应的 `topic` 和 `group`。

5. 消息处理结果发送到 `@EnableBinding` 注解对于 Source 接口生成的动态代理内部的 MessageChannel。最终消息会被发送到消息中间件对应的 topic 里。

上述代码需要配置信息：

```
spring.cloud.stream.bindings.input.destination=test-input
spring.cloud.stream.bindings.input.group=test-input-binder
spring.cloud.stream.bindings.input.binder=kafka
spring.cloud.stream.bindings.output.destination=test-output
spring.cloud.stream.bindings.output.binder=rocketmq
```

这里的 Input Binding 对应的 topic 是 test-input, group 是 test-input-binder, 对应的 MQ 是 Kafka, Output Binding 对应的 topic 是 test-output, 对应的 MQ 是 RocketMQ。

所以这段代码的意思是以 test-input-binder 这个 group 去 Kafka 上读取 test-input 这个 topic 下的消息，把消息的内容转换成大写再发送给 RocketMQ 的 test-output topic 上。

当然，你也可以直接通过[沙箱环境](#)直接查看案例。

分布式事务

1. 简介

分布式一致性是分布式系统亟需解决的关键问题之一，根据过去一年的调查问卷，在微服务的实践中分布式事务是用户遇到的最大痛点。目前市面缺少经过洪荒流量验证的分布式事务组件，Seata 在阿里经济体内部经过了漫长的孵化，承载了双 11 洪荒流量，实践证明

Seata 是一款解决分布式数据一致性的优秀组件。Seata 于 2019 年正式对外开源，开源后就受到了大家的热情追捧，一度蝉联 GitHub 活跃排名榜首。Seata 除了提供了独创的 AT 事务模式外，还扩展了 TCC、Saga 和 XA 事务模式，满足大家对于不同业务场景中的需求。相关详细信息可参考其官网 [Seata 官网](#)。

2. 学习目标

- 理解分布式事务在业务中的核心使用场景和常用解决方案
- 理解 Seata AT 事务模式的核心原理
- 掌握 Seata 作为分布式事务组件与 Spring Cloud 的整合
- 如何扩展一个 RPC 框架
- Seata 实战

3. 为什么需要分布式事务？

分布式事务不是在新架构下产生的新问题，即使在单体应用中同样存在着分布式事务问题，典型的场景是单体应用执行方法中含有多个数据源。X/OPEN 对于这一问题，提出了含有三种角色的 DTP(Distributed Transaction Processing)模型并形成了 XA 规范来解决此问题。各厂商针对 XA 规范做了具体的实现，也就是大家常说的 XA 协议。在 Java 体系中基于 DTP 模型提出了 JTA 规范（参考 JSR 907），定义了分布式事务中的事务管理器(TM)与资源管理器(RM)、应用程序(AP)等的 Java 接口。在 Java EE 时代，应用服务器如 weblogic 充当了 TM 的角色，而传统关系数据库通过实现 XA 协议充当了 RM 的角色。

随着互联网的高速发展，庞大的用户群体和快速的需求变化已经成为了传统架构的痛点。在这种情况下，如何从系统架构的角度出发，构建出灵活、易扩展的系统来快速响应需求的变化，同时，随着用户量的增加，如何保证系统的稳定性、高可用性、可伸缩性等等，成为了系统架构面临的挑战。微服务基于此背景应运而生，微服务架构越来越成为一种架构趋势，其本质是分布式去中心化。但微服务架构绝不是银弹，它不一定是一种能支撑未来一二十年的架构，引入微服务架构时需要 we 根据业务场景，系统复杂性和团队规模有步骤的进行。微服务架构的引入使分布式数据一致性问题更为突出，由原来的单体应用拆分出来几十甚至上百个微服务，如何保证服务间的一致性？当在一条较长的微服务调用链中，位于中间位置的微服务节点出现异常，如何保证整个服务的数据一致性？

分布式一致性的引入，一定不可避免带来性能问题，如何更高效的解决分布式一致性问题，一直是我们致力于解决此问题的关键出发点。在“一切都正常”的情况下，我们可以认为我们并不需要分布式事务。但系统很难满足这种理想状态，系统可能因为一个非法的参数校验无法将服务链路继续向下调用下去，系统可能出现令人反感的超时问题，我们不清楚被调用的服务是否真正的执行了，被调用服务可能正在部署，网络抖动亦或者节点宕机导致接口无法继续调用。这些问题普遍存在于我们的系统中，业务的本质体现在数据上，数据不一致的直接后果是可能产生资损，更严重的是如果不一致的数据不能被及时发现，业务再次基于此数据的进行相关逻辑操作，会进一步导致数据错上加错，最终很难溯源。

4. 常见的分布式事务解决方案

从是否满足事务 ACID 特性上，我们可以将事务分为两大类：刚性事务和柔性事务。在常见解决方案中 XA 事务属于刚性事务解决方案，而其他的大多数解决方案如 TCC、Saga、消息最终一致性则属于柔性事务解决方案。以下将对几种常见的事务方案做简要的介绍：

消息最终一致性

消息最终一致性方案是在 Seata 问世之前，市面上应用最广泛的一种解决方案。它本身具有削峰填谷，可异步化的优点，更多的适应于可异步化的末端链路消息通知场景。但是它本身也存在着一些缺点：需要依赖可靠消息组件，消息的可靠性很重要，大多数的原生消息组件故障时很难降级；实时性比较差，要经过多次网络 IO 开销和持久化，遇到队列积压情形实时性不可控；无法保证隔离性，在已发送消息和消息消费之前，中间数据对外可见，

无法满足事务 isolate 特性；只能向前重试不可向后回滚，消息消费无法成功时无法回滚消息生产侧的数据；无法保证多条消息间的数据一致性。

XA

XA 标准提出后的 20 多年间未能得到持续的演进，在学术界有协议优化和日志协同处理等相关的研究，在工业界使用 XA 落地方案的相对较少，主要集中在应用服务器的场景。XA 方案要求相关的厂商提供其具体协议的实现，目前大部分关系数据库支持了 XA 协议，但是支持程度不尽相同，例如，MySQL 在 5.7 才对 xa_prepare 语义做了完整支持。XA 方案被人诟病的是其性能，其实更为严重的是对于连接资源的占用，导致在高并发未有足够的连接资源来响应请求成为系统的瓶颈。在微服务架构下 XA 事务方案随着微服务链路的扩展成为一种反伸缩模式，进一步加剧了资源的占用。另外 XA 事务方案要求事务链路中的 resource 全部实现 XA 协议方可使用，若其中某一资源不满足，那么就无法保证整个链路的数据一致性。

TCC

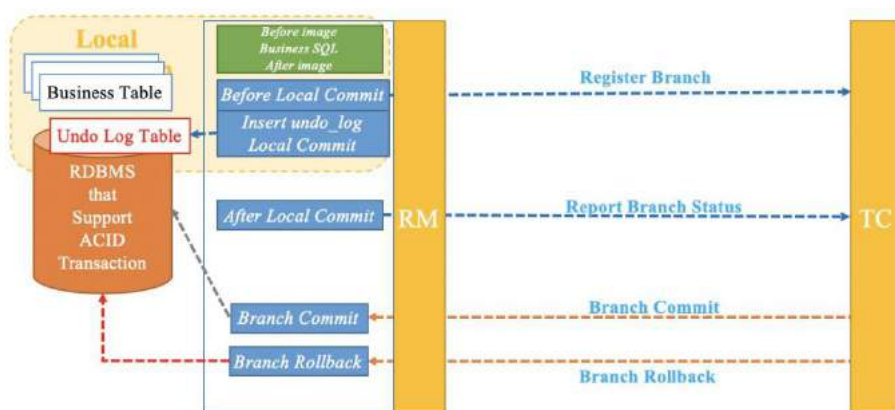
TCC 方案要求用户根据业务场景实现 try, confirm, cancel 三个接口，由框架根据事务所处的事务阶段和决议来自动调用用户实现的三个接口。从概念上 TCC 框架可以认为是一种万能框架，但是其难点是业务对于这三个接口的实现，开发成本相对较高，有较多业务难以做资源预留相关的逻辑处理，以及是否需要在预留资源的同时从业务层面来保证隔离性。因此，这种模式比较适应于金融场景中易于做资源预留的扣减模型。

Saga

有了 TCC 解决方案为什么还需要 Saga 事务解决方案？上文提到了 TCC 方案中对业务的改造成本较大，对于内部系统可以自上而下大刀阔斧的推进系统的改造，但对于第三方的接口的调用往往很难推动第三方进行 TCC 的改造，让对方为了你这一个用户去改造 TCC 方案而其他用户并不需要，需求上明显也是不合理的。要求第三方业务接口提供正反接口比如扣款和退款，在异常场景下必要的数据冲正是合理的。另外，Saga 方案更加适应于工作流式的长事务方案并且可异步化。

上面提到了 4 种常用的分布式事务解决方案，Seata 集成了 TCC、Saga 和 XA 方案。另外，Seata 还提供了独创的 AT 强一致分布式事务解决方案。下文将对 AT 方案进行简要的介绍。

6. AT 事务模式



一个分布式事务有全局唯一的 `xid`，由若干个分支事务构成，每个分支事务有全局唯一的 `branchId`。上图展示了在一个分支事务中 RM 与 TC 的交互过程。其中主要包含的交互动作如下：

• branchRegister

分布式事务一阶段执行，分支事务在 `commit` 之前与 TC 交互获取 全局锁 和返回 `branchId`。全局锁为 Seata 应用锁等同于修改数据记录的行锁，若获取锁失败将会进行锁重试，此处提供了两种重试策略是否持有数据库连接重试全局锁，默认为释放数据库连接。若成功，则抢占全局锁并返回 `branchId`，若重试到最大次数失败，则发起全局事务的回滚，对已完成的分支事务执行回滚。

• branchReport

分布式事务一阶段执行，本地事务 `commit` 之后与 TC 交互，上报本地事务已完成标识。目前 `branchReport` 动作已经在 1.0 版本做了相关的优化，本地事务 `commit` 不上报，本地事务 `rollback` 上报。经过优化分布式事务的整体性能在 `globalCommit` 场景下最低提升 25%，最高提升 50%。本地事务 `rollback` 上报可以帮助 TC 快速决策需要回滚的分支事务。

- **branchCommit**

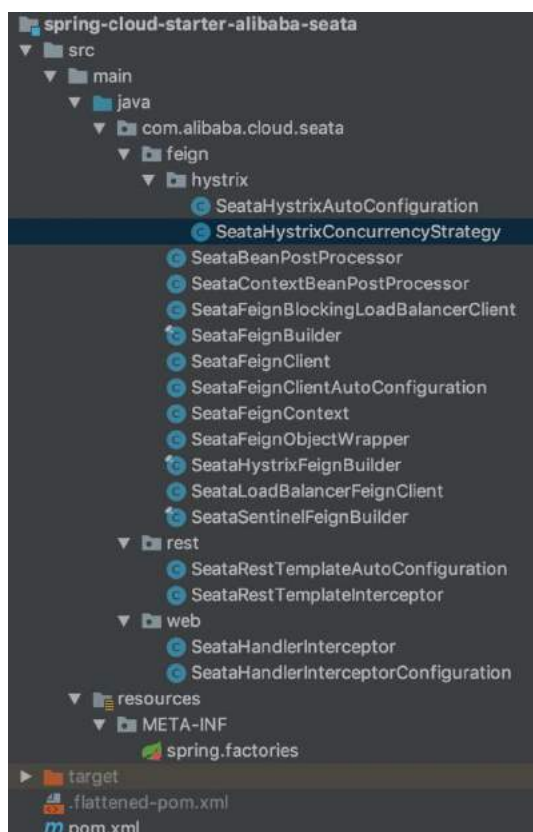
分布式事务二阶段执行，在形成 globalCommit 决议后执行。AT 模式中此步骤异步执行来提升其性能，可以认为分布式事务 globalCommit 决议提交到 TC 释放完全局锁就已经完成了整个分布式事务的处理。branchCommit 在 AT 模式主要用于删除一阶段的 undo_log，TC 下发到 RM 后并不是立即执行，而是通过定时任务+sql 批量合并的方式来提升其处理性能。

- **branchRollback**

分布式事务二阶段执行，在形成 globalRollback 决议后执行。RM 收到 branchRollback 请求，取 undo_log 表中对应的 branchId 记录解析 rollback_info 字段，对现有数据和 undo log 后镜像对比，对比不同则分支事务回滚失败。对比成功则根据前镜像构造 sql 并执行反向操作和删除 undo log。

详细处理过程和原理，可参考官网文档关于 AT 模式的介绍：<https://seata.io/zh-cn/docs/dev/mode/at-mode.html>

7. Seata 与 Spring Cloud 集成



如上图，Seata 与 Spring Cloud Alibaba 集成代码结构如上图所示。从代码上可以分为三大部分：rest、feign 和 web。AutoConfiguration 结尾的类是 @Configuration 类被 spring.factories 加载，负责创建 package 中所属的 bean。

rest

对应 restTemplate 调用，实现 ClientHttpRequestInterceptor 接口，将当前事务上下文包装到 HttpRequest header 中，加入到拦截器列表中。

feign

对应 openFeign 调用，这部分实现了事务上下文传递，与 Hystrix、Sentinel、Ribbon 组件集成功能。需要特别注意是 Hystrix 中跨线程的事务上下文传递。这部分代码大量使用了 Builder、Wrapper 模式，有兴趣的同学可深入阅读。

web

对应 Spring Web Servlet 中的处理，实现了 HandlerInterceptor 接口。在 preHandle 预处理中取 httpRequest header 中 Seata 的事务上下文并使用 API 绑定到当前线程的事务处理上下文中，这种后续的数据源操作就自动加入到了分布式事务的链路中；在 afterCompletion 中做了当前线程事务上下文的清除，防止事务上下文在线程中污染。

7. 如何扩展一个 RPC 框架？

在上一章节，我们讲到了 Seata 是如何与 Spring Cloud 相集成的。Seata 目前已经集成了 Spring Cloud、Alibaba Dubbo、Apache Dubbo、Motan、gRPC 和 sofa-RPC 等微服务调用。结合上一节与 Spring Cloud 的集成，扩展一个 RPC 框架我们需要做哪些工作呢？主要可以分为两大部分：

事务上下文传递

Seata 的事务上下文目前主要包含：xid 和调用服务的分支事务类型。xid 为一个分布式事务的全局唯一标识，类似于 Tracing 中的 traceId，只有将 xid 传递下去才可能加入到分布式事务的链路中。调用服务的分支事务类型主要用于非 AT 模式，例如在一个 TCC 分支事务中不能再嵌套 AT 分支事务。主流的 RPC 框架大多是基于 TCP 协议之上的私有协议封装或者是基于 HTTP 协议。Seata 的事务上下文标识都是简单的字符串，序列化由 RPC 框架直接完成，大多数 RPC 框架实现了 filter 或者 interceptor 接口，通过将 Seata 的事务上下文填充到协议中的 attachment 字段或者 http header 中，就可以简单的完成事务上下文的传递。

事务上下文绑定和清除

在 RPC 的 provider 端收到 consumer 的请求，将事务上下文取出，通过 API 绑定到当前的执行线程中，这样后续的业务处理都纳入到了 Seata 的分布式事务链路中。执行完业务处理后，需要对绑定到当前线程的事务上下文清除掉，防止产生线程事务上下文污染。

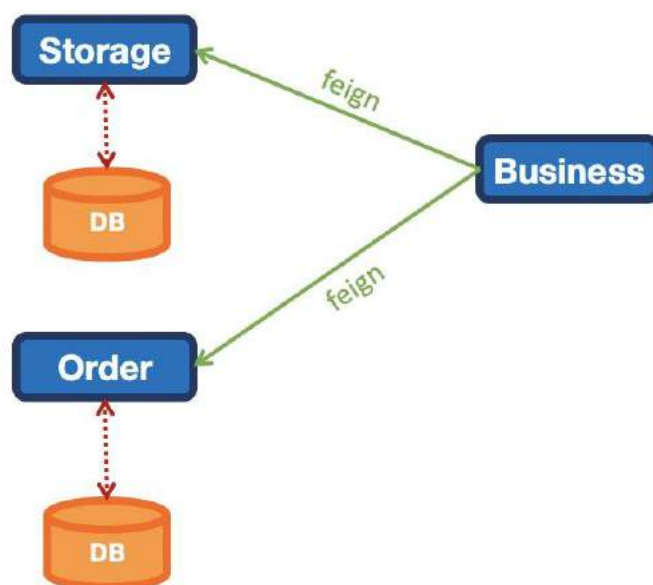
具体 API 可参考官网文档：<https://seata.io/zh-cn/docs/user/api.html>

请大家思考一下如何去与 Dubbo 集成的呢？<https://github.com/seata/seata/tree/develop/integration/dubbo>

8. Seata 实战

本章节将通过一个基于 Spring Cloud 的订单和库存服务进行 Seata 实战。

我们在沙箱环境里准备好了一套 seata 的实际应用案例，服务调用结构如下所示：



如图所示，服务链路中包含三个微服务：business（微服务入口）、order（订单服务）和 storage（库存服务）。

业务逻辑

通过 url 调用 business 服务，business 服务会通过 openFeign 的方式分别调用 storage 和 order 服务。每个服务调用后会根据正常（ok）或异常（fail）的返回值决定是否抛出异常，若返回结果不为 ok 那么 business 将抛出异常，触发整个事务回滚，预期数据至 business 方法执行前的数据并且库存总量和与初始值一致。当 order 和 storage 两个服务调用正常，用户可以根据 url 中的 mockException=true 或 false 来注入一个 mock 异常，当注入异常后，期望数据回滚至 business 方法执行前的数据并且库存总量和与初始值一致。

异常模拟

在 bussiness 中请求超过现有库存，通过参数 count 来指定要扣减的库存数量，当超出库存，将抛出异常进行事务回滚。

在 bussiness 中请求中加入 `mockException=true` 参数，触发事务回滚。

启动步骤

以下请求 url 中的 localhost 请根据实际值进行替换。

启动 order 和 storage 服务，最后启动 business 服务。服务的注册和发现，Seata 服务注册和发现 在这里使用了 Nacos 启动成功后，可以通过 Nacos 控制台的 sandbox-seata namespace 看到如下服务：



The screenshot shows the Nacos 1.3.2 console interface. The left sidebar contains navigation options: 配置管理, 服务管理, 服务列表, 应用管理, and 集群管理. The main area displays the '服务列表' (Service List) for the 'sandbox-seata' namespace. A table lists the registered services:

服务名	分组名称	权重	实例数	健康检查	配置保护策略	操作
seata-service	SEATA_GROUP	1	1	1	false	详情 注册 删除
storage-service	DEFAULT_GROUP	1	1	1	false	详情 注册 删除
order-service	DEFAULT_GROUP	1	1	1	false	详情 注册 删除
business-service	DEFAULT_GROUP	1	1	1	false	详情 注册 删除

访问 business 的 url：<http://localhost:8080/seata/check> 来检查初始化数据：



The screenshot shows the web application interface for the `seata/check` endpoint. The browser address bar shows `localhost:8081/seata/check`. The page content is as follows:

您当前操作的CommodityCode是: **C705346**

数据表:

order_tbl:

Id	user_id	commodity_code	count	money
80		C705346	100	

storage_tbl:

Id	commodity_code	count
80	C705346	100

数据一致性校验:

调用前库存:

初始化库存(storage_tbl): **100**

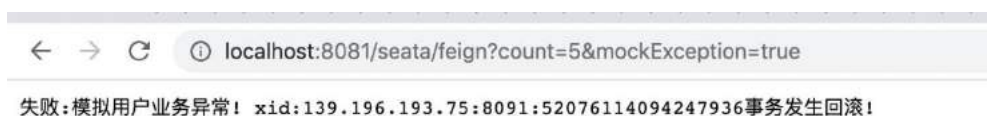
调用后库存:

已卖出库存数量(order_tbl): **0**

剩余库存(storage_tbl): **100**

结论: 库存量调用前后 **一致**

通过访问 business 的 url：<http://localhost:8080/seata/feign?count=5&mockException=true> 来触发业务逻辑：



其中 count 代表扣减库存的数量，当库存不足将触发事务回滚；mockException 代表是否触发业务异常，设置为 true 会触发业务异常，并通过分布式事务实现回滚。

每次操作完 business 请求后，访问 business 的 url：<http://localhost:8080/seata/check> 来检查操作后数据。

重复以上 3（根据实际需要请求，不必每次请求相同），4 步骤，最后再次通过访问 business 的 url：<http://localhost:8080/seata/check> 来检查结果数据：



使用总结

引入依赖。com.alibaba.cloud:spring-cloud-starter-alibaba-seata 中包含 io.seata:seata-spring-boot-starter 依赖。若于期望依赖 seata 的版本不一致，可手动排除重引入。com.alibaba.cloud 原生 [版本说明](#)。

```
parent: <dependencyManagement>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-alibaba-dependencies</artifactId>
        <version>${alibaba.cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
</dependencyManagement>
module:
    <dependencies>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
        </dependency>
    </dependencies>
```

添加配置文件，具体配置项说明参考[此处](#)。

```
seata:
  enabled: true
  application-id: business
  tx-service-group: my_test_tx_group
  config:
    type: nacos
    nacos:
      namespace: "sandbox-seata"
      serverAddr: 139.196.203.133:8848
      group: SEATA_GROUP
      username: xxx
      password: xxx
  registry:
    type: nacos
    nacos:
      application: seata-server
      serverAddr: 139.196.203.133:8848
      group: SEATA_GROUP
      namespace: "sandbox-seata"
      username: xxx
      password: xxx
```

1. 在所有业务库中创建 `undo_log` 表，不同的数据库类型脚本参考[此处](#)，示例中已自动完成创建。
2. 在需要纳入分布式事务链路的入口 `service` 方法（保证可使 Spring 切面生效的位置亦可）上添加 `@GlobalTransactional` 注解。

9. 其他

Seata 更多 sample 样例，请参考 [样例](#)。

Seata 社区联系方式，请参见 [联系我们](#)。

欢迎大家试用阿里云商业化产品 [GTS](#)，更高性能，更稳定，全面兼容 Seata。



钉钉扫码加入
Spring Cloud Alibaba 交流群
如果发现电子书有任何问题，欢迎加群勘误



微信扫码关注
“阿里巴巴云原生公众号”



阿里云开发者“藏经阁”
海量免费电子书下载