Spring Boot with Redis

摘自简书

Contents

Spring boot

> Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Boot 致力于在蓬勃发展的快速应用开发领域（rapid application development）成为领导者。

多年以来，Spring IO 平台饱受非议的一点就是大量的 XML 配置以及复杂的依赖管理。在去年的 SpringOne 2GX 会议上，Pivotal 的 CTO Adrian Colyer 回应了这些批评，并且特别提到该平台将来的目标之一就是实现免 XML 配置的开发体验。Boot 所实现的功能超出了这个任务的描述，开发人员不仅不再需要编写 XML，而且在一些场景中甚至不需要编写繁琐的 import 语句。在对外公开的 beta 版本刚刚发布之时，Boot 描述了如何使用该框架在 140 个字符内实现可运行的 web 应用，从而获得了极大的关注度，该样例发表在 tweet 上。

然而，Spring Boot 并不是要成为 Spring IO 平台里面众多 “Foundation” 层项目的替代者。Spring Boot 的目标不在于为已解决的问题域提供新的解决方案，而是为平台带来另一种开发体验，从而简化对这些已有技术的使用。对于已经熟悉 Spring 生态系统的开发人员来说，Boot 是一个很理想的选择，不过对于采用 Spring 技术的新人来说，Boot 提供一种更简洁的方式来使用这些技术。

在追求开发体验的提升方面，Spring Boot，甚至可以说整个 Spring 生态系统都使用到了 Groovy 编程语言。Boot 所提供的众多便捷功能，都是借助于 Groovy 强大的 MetaObject 协议、可插拔的 AST 转换过程以及内置的依赖解决方案引擎所实现的。在其核心的编译模型之中，Boot 使用 Groovy 来构建工程文件，所以它可以使用通用的导入和样板方法（如类的 main 方法）对类所生成的字节码进行装饰（decorate）。这样使用 Boot 编写的应用就能保持非常简洁，却依然可以提供众多的功能。

安装 Boot

从最根本上来讲，Spring Boot 就是一些库的集合，它能够被任意项目的构建系统所使用。简便起见，该框架也提供了命令行界面，它可以用来运行和测试 Boot 应用。框架的发布版本，包括集成的 CLI（命令行界面），可以在 Spring 仓库中手动下载和安装。一种更为简便的方式是使用 Groovy 环境管理器（Groovy enVironment Manager，GVM），它会处理 Boot 版本的安装和管理。Boot 及其 CLI 可以通过 GVM 的命令行 gvm install springboot 进行安装。在 OS X 上安装 Boot 可以使用 Homebrew 包管理器。为了完成安装，首先要使用 brew tap pivotal/tap 切换到 Pivotal 仓库中，然后执行 brew install springboot 命令。

要进行打包和分发的工程会依赖于像 Maven 或 Gradle 这样的构建系统。为了简化依赖图，Boot 的功能是模块化的，通过导入 Boot 所谓的 “starter” 模块，可以将许多的依赖添加到工程之中。为了更容易地管理依赖版本和使用默认配置，框架提供了一个 parent POM，工程可以继承它。Spring Boot 工程的样例 POM 文件定义如程序清单 1 所示。

程序清单 1

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <!-- Inherit defaults from Spring Boot -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.0.0.RC1</version>
    </parent>

    <!-- Add typical dependencies for a web application -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/libs-snapshot</url>
        </repository>
    </repositories>

    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/libs-snapshot</url>
        </pluginRepository>
    </pluginRepositories>
```

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

为了实现更为简单的构建配置，开发人员可以使用 Gradle 构建系统中简洁的 Groovy DSL，如程序清单 1.1 所示。

程序清单 1.1

```
buildscript {
  repositories {
    maven { url "http://repo.spring.io/libs-snapshot" }
    mavenCentral()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.0.RC1")
  }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

repositories {
  mavenCentral()
  maven { url "http://repo.spring.io/libs-snapshot"  }
}

dependencies {
  compile 'org.springframework.boot:spring-boot-starter-actuator:1.0.0.RC1'
}
```

为了快速地搭建和运行 Boot 工程，Pivotal 提供了称之为 “Spring Initializr” 的 web 界面，用于下载预先定义好的 Maven 或 Gradle 构建配置。我们也可以使用 Lazybones 模板实现快速起步，在执行 lazybones create spring-boot-actuator my-app 命令后，它会为 Boot 应用创建必要的工程结构以及 gradle 构建文件。

开发 Spring Boot 应用

Spring Boot[1] 在刚刚公开宣布之后就将一个样例发布到了 Twitter 上，它目前成为了最流行的一个应用样例。它的全部描述如程序清单 1.2 所示，一个非常简单的 Groovy 文件可以生成功能强大的以 Spring 为后端的 web 应用。

这个应用可以通过 spring run App.groovy 命令在 Spring Boot CLI[2] 中运行。Boot 会分析文件并根据各种 “编译器自动配置（compiler auto-configuration）” 标示符来确定其意图是生成 Web 应用。然后，它会在一个嵌入式的 Tomcat 中启动 Spring 应用上下文，并且使用默认的 8080 端口。打开浏览器并导航到给定的 URL，随后将会加载一个页面并展现简单的文本响应：“hello”。提供默认应用上下文以及嵌入式容器的[3] 这些过程，能够让开发人员更加关注于开发应用以及业务逻辑，从而不用再关心繁琐的样板式配置。

Boot 能够自动确定类所需的功能，这一点使其成为了强大的快速应用开发工具。当应用在 Boot CLI 中执行时，它们在使用内部的 Groovy 编译器进行构建，这个编译器可以在字节码生成的时候以编码的方式探查并修改类。通过这种方式，使用 CLI 的开发人员不仅可以省去定义默认配置，在一定程度上甚至可以不用定义特定的导入语句，它们可以在编译的过程中识别出来并自动进行添加。除此之外，当应用在 CLI 中运行时，Groovy 内置的依赖管理器，“Grape”，将会解析编译期和运行时的类路径依赖，与 Boot 编译器的自动配置机制类似。这种方式不仅使得框架更加对用户友好，而且能够让不同版本的 Spring Boot 与特定版本的来自于 Spring IO 平台的库相匹配，这样一来开发人员就不用关心如何管理复杂的依赖图和版本结构了。另外，它还有助于快速原型的开发并生成概念原型的工程代码。

对于不是使用 CLI 构建的工程，Boot 提供了许多的 “starter” 模块，它们定义了一组依赖[4]，这些依赖能够添加到构建系统之中，从而解析框架及其父平台所需的特定类库。例如，spring-boot-starter-actuator 依赖会引入一组基本的 Spring 项目，从而实现应用的快速配置和即时可用。关于这种依赖，值得强调的一点就是当开发 Web 应用，尤其是 RESTful Web 服务的时候，如果包含了 spring-boot-starter-web 依赖，它就会为你提供启动嵌入式 Tomcat 容器的自动化配置，并且提供对微服务应用有价值的端点信息，如服务器信息、应用指标（metrics）以及环境详情。除此之外，如果引入 spring-boot-starter-security 模块的话，actuator 会自动配置 Spring Security，从而为应用提供基本的认证以及其他高级的安全特性。它还会为应用结构引入一个内部的审计框架，这个框架可以用来生成报告或其他的用途，比如开发认证失败的锁定策略。

---

[1] On their friendship, [see @Campbell_1987 p. 314-318; @Duckworth_1956 p. 281-316]; that the two were not friends by the time of the *Odes*, [see @Thomas_2001 p. 60] who argues that Horace and Vergil were only acquaintances and [@Moritz_1969 p. 13] who believes that the friendship was strained by the publication of the *Odes*. For a response to such readings, see Margheim 2012.

[2] Horace’s poetry provides the sole basis for positing a friendship. Vergil never mentions Horace by name in his poetry, and no other contemporary or near-contemporary sources ascribe *amicitia* to the two poets, although by 380 St. Jerome assumes a friendship. Horace names Vergil ten times throughout his corpus (*Sat*. 1.5.40, 48, 1.6.55, 1.10.45, 81; *Odes* 1.3.6, 1.24.10, 4.12.13; *Ep*. 2.1.247; *A.P.* 55), five times in the *Satires* alone, where Vergil consistently appears as a friend and colleague.

[3] Though there is some debate whether the Vergilius of 4.12 is Virgil the poet, the *opinio communis* today asserts this identification (see below, p. 10).

[4] For readings of *Odes* 1.24, [see @Commager_1995 p. 287-90; @Khan_1967; @Nisbet-Hubbard_1970 p. 279-89; @Lowrie_1994 p. 377-94; @Putnam_1992; @Horace_1995 p. 112-15]. For the Epicurean, and specifically Philodeman, influence on the ode, [@Thibodeau_2003 pp. 243-56, and @Armstrong_2008 p. 97-99].

延伸阅读

Spring Boot 团队已经编写了完整的指导和样例来阐述框架的功能。Blog 文章、参考资料以及 API 文档都可以在 Spring.IO 网站上找到。项目的 GitHub 页面上可以找到示例的工程，更为具体的细节可以阅读 Spring Boot 的参考手册。SpringSourceDev YouTube 频道有一个关于 Spring Boot 的 webinar，它概述了这个项目的目标和功能。在去年在伦敦举行的 Groovy & Grails Exchange 上，David Dawson 做了一个使用 Spring Boot 开发微服务的演讲。

关于作者

Daniel Woods 是 Netflix 的高级软件工程师，负责开发持续交付和云部署工具。他擅长 JVM 栈相关的技术，活跃在 Groovy、Grails 和 Spring 社区。可以通过电子邮件地址 danielpwoods@gmail.com 或 Twitter @danveloper 联系到 Daniel。

浅析如何设计一个亿级网关

背景

什么是 API 网关

API 网关可以看做系统与外界联通的入口，我们可以在网关进行处理一些非业务逻辑的逻辑，比如权限验证，监控，缓存，请求路由等等。

为什么需要 API 网关

RPC 协议转成 HTTP

由于在内部开发中我们都是以 RPC 协议 (thrift or dubbo) 去做开发，暴露给内部服务，当外部服务需要使用这个接口的时候往往需要将 RPC 协议转换成 HTTP 协议。

请求路由

在我们的系统中由于同一个接口新老两套系统都在使用，我们需要根据请求上下文将请求路由到对应的接口。

统一鉴权

对于鉴权操作不涉及到业务逻辑，那么可以在网关层进行处理，不用下层到业务逻辑。

统一监控

由于网关是外部服务的入口，所以我们可以在这里监控我们想要的数据，比如入参出参，链路时间。

流量控制，熔断降级

对于流量控制，熔断降级非业务逻辑可以统一放到网关层。

有很多业务都会自己去实现一层网关层，用来接入自己的服务，但是对于整个公司来说这还不够。

统一 API 网关

统一的 API 网关不仅有 API 网关的所有的特点，还有下面几个好处:

统一技术组件升级

在公司中如果有某个技术组件需要升级，那么是需要和每个业务线沟通，通常几个月都搞不定。举个例子如果对于入口的安全鉴权有重大安全隐患需要升级，如果速度还是这么慢肯定是不行，那么有了统一的网关升级是很快的。

统一服务接入

对于某个服务的接入也比较困难，比如公司已经研发出了比较稳定的服务组件，正在公司大力推广，这个周期肯定也特别漫长，由于有了统一网关，那么只需要统一网关统一接入。

节约资源

不同业务不同部门如果按照我们上面的做法应该会都自己搞一个网关层，用来做这个事，可以想象如果一个公司有 100 个这种业务，每个业务配备 4 台机器，那么就需要 400 台机器。并且每个业务的开发 RD 都需要去开发这个网关层，去随时去维护，增加人力。如果有了统一网关层，那么也许只需要 50 台机器就可以做这 100 个业务的网关层的事，并且业务 RD 不需要随时关注开发，上线的步骤。

统一网关的设计

异步化请求

对于我们自己实现的网关层，由于只有我们自己使用，对于吞吐量的要求并不高所以，我们一般同步请求调用即可。

对于我们统一的网关层，如何用少量的机器接入更多的服务，这就需要我们的异步，用来提高更多的吞吐量。对于异步化一般有下面两种策略:

Tomcat/Jetty+NIO+servlet3

这种策略使用的比较普遍，京东，有赞，Zuul，都选取的是这个策略，这种策略比较适合 HTTP。在 Servlet3 中可以开启异步。

Netty+NIO

Netty 为高并发而生，目前唯品会的网关使用这个策略，在唯品会的技术文章中在相同的情况下 Netty 是每秒 30w+ 的吞吐量，Tomcat 是 13w+, 可以看出是有一定的差距的，但是 Netty 需要自己处理 HTTP 协议，这一块比较麻烦。

对于网关是 HTTP 请求场景比较多的情况可以采用 Servlet，毕竟有更加成熟的处理 HTTP 协议。如果更加重视吞吐量那么可以采用 Netty。

全链路异步

对于来的请求我们已经使用异步了，为了达到全链路异步所以我们需要对去的请求也进行异步处理，对于去的请求我们可以利用我们 rpc 的异步支持进行异步请求所以基本可以达到下图:

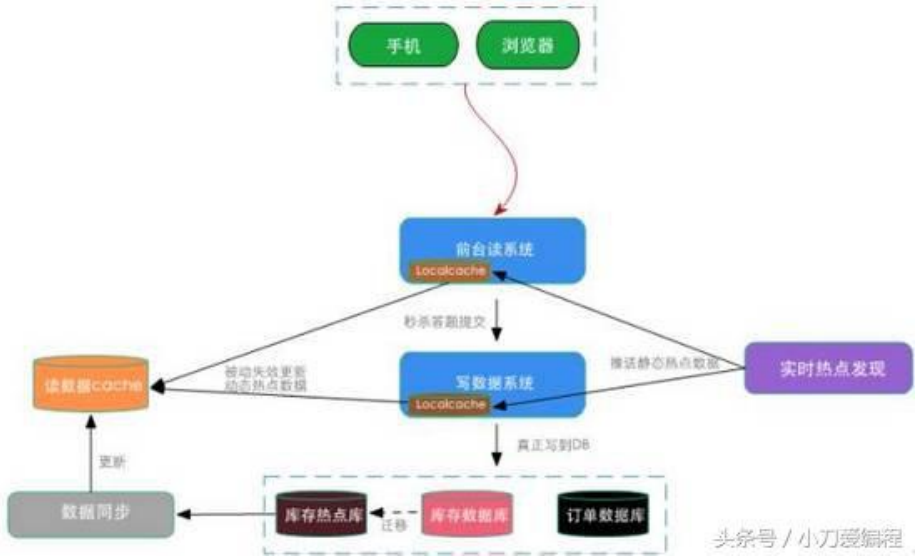

Figure 1: la lune

由在 web 容器中开启 servlet 异步，然后进入到网关的业务线程池中进行业务处理，然后进行 rpc 的异步调用并注册需要回调的业务，最后在回调线程池中进行回调处理。

链式处理

在设计模式中有一个模式叫责任链模式，他的作用是避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。通过这种模式将请求的发送者和请求的处理者解耦了。在我们的各个框架中对此模式都有实现，比如 servlet 里面的 filter，springmvc 里面的 Interceptor。

在 Netflix Zuul 中也应用了这种模式，如下图所示:

这种模式在网关的设计中我们可以借鉴到自己的网关设计:

- preFilters：前置过滤器，用来处理一些公共的业务，比如统一鉴权，统一限流，熔断降级，缓存处理等，并且提供业务方扩展。
- routingFilters: 用来处理一些泛化调用，主要是做协议的转换，请求的路由工作。
- postFilters: 后置过滤器，主要用来做结果的处理，日志打点，记录时间等等。
- errorFilters: 错误过滤器，用来处理调用异常的情况。

这种设计在有赞的网关也有应用。

业务隔离

上面在全链路异步的情况下不同业务之间的影响很小，但是如果在提供的自定义 FiIlter 中进行了某些同步调用，一旦超时频繁那么就会对其他业务产生影响。所以我们需要采用隔离之术，降低业务之间的互相影响。

信号量隔离

信号量隔离只是限制了总的并发数，服务还是主线程进行同步调用。这个隔离如果远程调用超时依然会影响主线程，从而会影响其他业务。因此，如果只是想限制某个服务的总并发调用量或者调用的服务不涉及远程调用的话，可以使用轻量级的信号量来实现。有赞的网关由于没有自定义 filter 所以选取的是信号量隔离。

线程池隔离

最简单的就是不同业务之间通过不同的线程池进行隔离，就算业务接口出现了问题由于线程池已经进行了隔离那么也不会影响其他业务。在京东的网关实现之中就是采用的线程池隔离，比较重要的业务比如商品或者订单都是单独的通过线程池去处理。但是由于是统一网关平台，如果业务线众多，大家都觉得自己的业务比较重要需要单独的线程池隔离，如果使用的是 Java 语言开发的话那么，在 Java 中线程是比较重的资源比较受限，如果需要隔离的线程池过多不是很适用。如果使用一些其他语言比如 Golang 进行开发网关的话，线程是比较轻的资源，所以比较适合使用线程池隔离。

集群隔离

如果有某些业务就需要使用隔离但是统一网关又没有线程池隔离那么应该怎么办呢？那么可以使用集群隔离，如果你的某些业务真的很重要那么可以为这一系列业务单独申请一个集群或者多个集群，通过机器之间进行隔离。

请求限流

流量控制可以采用很多开源的实现，比如阿里最近开源的 Sentinel 和比较成熟的 Hystrix。

一般限流分为集群限流和单机限流: - 利用统一存储保存当前流量的情况，一般可以采用 Redis，这个一般会有一些性能损耗。- 单机限流: 限流每台机器我们可以直接利用 Guava 的令牌桶去做，由于没有远程调用性能消耗较小。

熔断降级

这一块也可以参照开源的实现 Sentinel 和 Hystrix，这里不是重点就不多提了。

泛化调用

泛化调用指的是一些通信协议的转换，比如将 HTTP 转换成 Thrift。在一些开源的网关中比如 Zuul 是没有实现的，因为各个公司的内部服务通信协议都不同。比如在唯品会中支持 HTTP1,HTTP2, 以及二进制的协议，然后转化成内部的协议，淘宝的支持 HTTPS,HTTP1,HTTP2 这些协议都可以转换成，HTTP,HSF,Dubbo 等协议。

泛化调用

如何去实现泛化调用呢？由于协议很难自动转换，那么其实每个协议对应的接口需要提供一种映射。简单来说就是把两个协议都能转换成共同语言，从而互相转换。

一般来说共同语言有三种方式指定:

- json：json 数据格式比较简单, 解析速度快，较轻量级。在 Dubbo 的生态中有一个 HTTP 转 Dubbo 的项目是用 JsonRpc 做的，将 HTTP 转化成 JsonRpc 再转化成 Dubbo。

  比如可以将一个 www.baidu.com/id = 1 GET 可以映射为 json：

  代码块

  { “method”: “getBaidu” “param” : { “id” : 1 } }

- xml:xml 数据比较重，解析比较困难，这里不过多讨论。
- 自定义描述语言: 一般来说这个成本比较高需要自己定义语言来进行描述并进行解析，但是其扩展性，自定义个性化性都是最高。例:spring 自定义了一套自己的 SPEL 表达式语言

对于泛化调用如果要自己设计的话 JSON 基本可以满足，如果对于个性化的需要特别多的话倒是可以自己定义一套语言。

管理平台

上面介绍的都是如何实现一个网关的技术关键。这里需要介绍网关的一个业务关键。有了网关之后，需要一个管理平台如何去对我们上面所描述的技术关键进行配置, 包括但不限于下面这些配置:

- 限流
- 熔断
- 缓存
- 日志
- 自定义 filter
- 泛化调用

总结

最后一个合理的标准网关应该按照如下去实现:

参考

- 京东:http://www.yunweipai.com/archives/23653.html
- 有赞网关:https://tech.youzan.com/api-gateway-in-practice/
- 唯品会:https://mp.weixin.qq.com/s/gREMe-G7nqNJJLzbZ3ed3A
- Zuul:http://www.scienjus.com/api-gateway-and-netflix-zuul/

如何选择适合你的微服务-API 网关

微服务 API 网关有什么作用?

让我们先来看下微服务 API 网关的作用，下图是一个简要的说明

API 网关并非一个新兴的概念，在十几年前就已经存在了，它的作用主要是作为流量的入口，统一的处理和业务相关的请求，让请求更加安全、快速和准确的得到处理。它有以下传统的功能:

- 反向代理和负载均衡，这和 Nginx 的定位和功能是一致的;
- 动态上游、动态 SSL 证书和动态限流限速等运行时的动态功能，这是开源版本 Nginx 并不具备的功能;
- 上游的主动和被动健康检查，以及服务熔断;
- 在 API 网关的基础之上进行扩展，成为全生命周期的 API 管理平台。

在最近几年，业务相关的流量，不再仅仅是由 PC 客户端和浏览器发起，更多的来自手机、IoT 设备等，未来随着 5G 的普及，这些流量会越来越多，同时，随着微服务架构的结构变迁，服务之间的流量也开始爆发性的增长。在这种新的业务场景下，催生了 API 网关更多、更高级的功能:

- 云原生友好，架构要变得轻巧，便于容器化;
- 对接 Prometheus、Zipkin、Skywalking 等统计、监控组件;
- 支持 gRPC 代理，以及 http 到 gRPC 之间的协议转换，把用户的 http 请求转为内部服务的 gPRC 请求;
- 承担 OpenID Relying Party 的角色，对接 Autho、okta 等身份认证提供商的服务，把流量的安全作为头等大事来对待;
- 通过运行时动态执行用户函数的方式来实现 serverless，让网关的边缘节点更加灵活;
- 不锁定用户，支持混合云的部署架构;
- 最后就是网关节点要状态无关，可以随意的扩容和缩容。

一个微服务 API 网关具备了上述十几项功能，就可以让用户的服务只关心业务本身，而和业务实现无关的功能，比如服务发现、服务熔断、身份认证、限流限速、统计、性能分析等，就可以在独立的网关层面来解决。从这个角度来看，API 网关既可以替代 Nginx 的所有功能，来处理南北向的流量，也可以完成 Istio 控制面和 Envoy 数据面的角色，来处理东西向的流量。

备选的 API 网关有哪些?

正因为微服务 API 网关的地位如此重要，所以它一直处于兵家必争之地，传统的 IT 巨头在这个领域很早就都有布局，比如谷歌、CA、IBM、红帽、salesforce、以及 AWS、阿里云等公有云厂商。

这些闭源的商业产品，它们的功能都很完善，覆盖了 API 的设计、多语言 SDK、文档、测试和发布等全生命周期管理，并且提供 SaaS 服务，有些还与公有云做了集成，使用起来非常方便，但同时也带来两个痛点：- 平台锁定。API 网关是业务流量的入口，它不像图片、视频等 CDN 加速的这种非业务流量可以随意迁移，API 网关上会绑定不少业务相关的逻辑，一旦使用了闭源的方案，就很难平滑和低成本的迁移到其他平台。- 无法二次开发。一般的大中型企业都会有自己独特的需求，需要定制开发，这时候你就只能依靠厂商，而不能自己动手去做二次开发。

所以我们更偏重于开源的 API 网关方案，比如 Kong、APISIX 和 Trk 等。这些 API 网关是从云原生软件基金会（CNCF）的全景图中摘选的:

对比选型的依据

部署和维护成本

- 是可以在单机就能完整部署，还是需要多个节点配合才能使用?
- 是否有外部的数据库依赖? 比如 MySQL、Postgres?
- 是否有 web 控制台可以操作整个集群?

开源还是闭源

- 你是否可以编写自己的插件来扩展 API 网关的功能?
- 当你使用了某个 API 网关后，是否可以平滑而且低成本的迁移到其他 API 网关?
- 是否会被锁定在特定的平台上?

能否私有化部署

- 是否支持部署在用户自己的内部服务器中?
- 是否支持多云、混合云的部署模式?

功能

- 是否支持动态上游、动态 SSL 证书、主动/被动健康检查这些基本的功能
- 能否对接 Prometheus、Zipkin、Skywalking 等统计、监控组件
- 是否可以通过 HTTP REST API 和 yaml 配置文件这两种方式，来控制网关配置

社区

- 使用者能否通过 Github、QQ 群、Stack Overflow 等方式联系到社区的开发者?
- 开源许可证是否友好?
- 是否可以方便的提交自己的修改到主线版本?
- 背后是否有商业公司支持?

商业支持和价格

- 开源版本和商业版本差异是否很大?
- 商业版本是按照 API 调用次数还是订阅方式收费?

API 网关对比

下面是各个 API 网关多个角度的对比结果:

从中我们可以看出，Kong 和 APISIX 都是非常好的选择。如果你有其他推荐的 API 网关，或者有更多的观点，欢迎留言。

阅读原文