一种微服务框架的实现①

张 晶[1], 王琰洁[1], 黄小锋[2]

[1](北京中电普华信息技术有限公司, 北京 100192)
[2](中国电建集团国际工程有限公司, 北京 100048)

摘 要: 相对于传统单块架构, 微服务框架具有技术选型灵活、独立按需扩展、可用性高等优点, 更适合当前互联网时代需求. 但微服务架构的应用也会引入新的问题, 如跨进程通讯、服务注册发现、分布式 Session 管理等. 本文在对传统框架和微服务框架进行分析比较的基础上, 给出了微服务框架的一种实现方案. 该方案设计了微服务框架的功能架构, 对微服务框架引入的关键问题给出了解决方案. 采用该实现方案进行业务系统开发, 开发人员只需要关注微服务内部业务功能的开发, 微服务之间的注册、发现、监控和 Session 管理由微服务框架完成, 简化了系统开发的难度, 提高开发效率.

关键词: 微服务框架; 服务注册; 服务发现; Session 管理

Implementation of Microservice Architecture

ZHANG Jing[1], WANG Yan-Jie[1], HUANG Xiao-Feng[2]

[1](Beijing China Power Information Technology Co. Ltd., Beijing 100192, China)
[2](Power China International Group Limited, Beijing 100048, China)

Abstract: Compared with traditional monolithic architecture, microservice architecture has many advantages, such as flexible technology selection, independent scalability, high availability and so on, and is more suitable for the current needs of the Internet age. But microservice architecture also introduces new problems, such as cross-process communication, service registration, service discovery, and distributed session management. On the basis of analysis and comparison between the traditional framework and microservice framework, this paper shows one implementation method of microservice framework. First, we design a scheme of microservices architecture framework and the functional framework, and then give the solutions of some key issues that microservice architecture introduces. With this scheme, developers only need to focus on the development of business functions, service registration, discovery, monitoring and session management provided by the framework to simplify the development and improve development effectiveness.

Key words: microservice architecture; service registration; service discover; session management

软件的架构设计是决定应用系统是否能够被正确、有效实现的关键要素之一. 架构设计描述了在应用系统的内部, 如何根据业务、技术、组织, 以及灵活性、可扩展性、可维护性等多种因素, 将应用系统划分成不同的部分, 并使这些部分相互协作, 从而为用户提供某种特定价值的方式[1,2].

传统信息化系统的典型架构是单块架构, 即将应用程序的所有功能都打包成一个应用, 每个应用是最小的交付和部署单元, 应用部署后运行在同一进程中. 典型的单块架构应用, 如基于传统 J2EE 平台所构建的产品或者项目, 它们存在的形态一般是 WAR 包或者 EAR 包. 当部署这类应用时, 通常是将整个包作为一个整体, 部署在同一个 WEB 容器, 如 Tomcat 或者 Jetty 中. 当这类应用运行起来后, 所有的功能也都运行在同一个进程中. 单块架构应用具有 IDE 友好、易于测试和部署等优势, 但是, 随着互联网的迅速发展和企业应用范围的扩展, 单块架构表现出一些不足[3]:

① 代码庞杂, 理解困难, 新人上手也困难;

② 维护困难, 一般由一个团队维护, 应用越大, 则维护的人越多, 团队管理成本也越高, 团队效率也会

① 收稿时间:2016-07-21;收到修改稿时间:2016-08-18 [doi:10.15888/j.cnki.csa.005684]

越低下;

③ 功能升级困难: 每一次小改动都需要重新部署整个应用, 进行全面测试;

④ 技术堆栈固化: 尝试新技术的代价太高, 导致技术僵化;

⑤ 扩展困难: 应用某些部分偏 IO 密集型、某些部分却偏 CPU 密集型, 但应用却只部署在一台机器上, 很难用单一硬件来满足应用各部分对硬件资源的不同要求.

如何找到一种更有效的、更灵活、更适应当前互联网时代需求的系统架构方式, 成为大家关注的焦点. 随着微服务架构的出现以及在国内外的成功应用, 基于微服务框架构建系统应用成为一种新的选择.

1 微服务架构

微服务架构是一种架构模式, 采用一组服务的方式来构建一个应用, 服务独立部署在不同的进程中, 不同服务通过一些轻量级交互机制来通信, 例如 RPC、HTTP 等, 服务可独立扩展伸缩, 每个服务定义了明确的边界, 不同的服务甚至可以采用不同的编程语言来实现, 由独立的团队来维护[4].

相对于传统的单体应用架构, 微服务架构通过将功能分解到各个离散的服务实现对应用系统的解耦, 具有明显的优势[5,6]:

① 复杂度可控: 每一个微服务专注于单一功能, 并通过定义良好的接口清晰表述服务边界, 体积小、复杂度低, 提高了系统的可维护性和开发效率.

② 独立部署: 微服务具备独立的运行进程, 可以独立部署. 当某个微服务发生变更时无需编译、部署整个应用. 由微服务组成的应用相当于具备一系列可并行的发布流程, 使得发布更加高效, 同时降低对生产环境所造成的风险, 最终缩短应用交付周期.

③ 技术选型灵活: 每个团队可以根据自身服务的需求和行业发展的现状, 自由选择最适合的技术栈.

④ 容错: 在微服务架构下, 故障被隔离在单个服务中. 可通过重试、平稳退化等机制实现应用层面的容错, 避免全局性的不可用.

⑤ 扩展: 每个服务可以根据实际需求独立进行扩展.

采用微服务架构也会引入新的问题: 基于微服务架构的应用是分布式系统, 服务独立运行在不同的进程中, 需要有进程间通讯机制来支撑服务之间的交互; 应用由多个服务构成, 每个服务可以有多个实例, 当一项服务存在于多个主机节点时, 需要一套服务发现机制, 使服务调用端可以获取正确的服务地址; 微服务应用是分布式系统, 需要解决分布式系统中 Session 管理的问题. 这些都是在微服务框架实现时需要解决的问题.

2 微服务框架设计

通过对微服务框架进行分析, 确定了微服务框架的总体架构, 如图 1 所示.

图 1 架构图

微服务框架包括服务注册发现、日志管理、序列化、服务配置、服务容器、服务通信、限流容错、管理接口和服务安全等组件.

其中, 服务容器是微服务运行的容器, 通过内嵌服务器的方式实现. 包含界面展现组件、IOC 组件、数据持久化组件支撑业务系统的功能开发. 界面展现组件提供用于界面设计的通用组件, 支持主流浏览器兼容性、可视化编程、控件拖拽等功能. IOC 组件提供依赖注入功能, 管理程序依赖. 数据持久化组件负责微服务数据的持久化存储.

日志管理: 记录重要的框架层日志、度量和调用链数据, 同时将日志、度量等接口暴露出来, 让业务层能根据需要记录业务日志数据.

序列化: 支持将业务逻辑以 REST 或者 RPC 方式暴露出来, 支持可定制的序列化机制. 对浏览器, 框架支持输出 Ajax 友好的 JSON 消息格式; 而对无线设备上的 Native App, 框架支持输出性能高的 Binary 消息格式.

服务配置: 支持文件方式配置, 集成动态运行时配置, 能够在运行时针对不同环境动态调整服务的参数和配置.

服务通信: 提供基于 REST/RPC 的同步请求响应

模式和基于消息的异步通信模式.

限流和容错: 集成限流容错组件, 能够在运行时自动限流和容错, 保护服务. 同时和动态配置相结合, 实现动态限流和熔断.

管理接口: 提供管理接口, 可以在线查看框架和服务内部状态, 同时还可以动态调整内部状态, 对调试、监控和管理提供快速反馈.

服务安全: 以插件方式封装安全和访问控制逻辑, 业务服务根据需要加载相关安全插件.

2.1 服务注册发现

微服务架构是由一系列职责单一的细粒度服务构成的分布式网状结构, 一个服务可以多实例并存, 这就引入了服务注册发现问题, 服务提供者要注册发布服务地址, 服务消费者要能发现目标服务, 同时服务提供者一般以集群方式提供服务, 也就引入了负载均衡问题. 目前主要的服务注册发现方案有两种:

① 提供者服务发现

在服务提供者和服务消费者之间有一个独立的负载均衡器, 负载均衡器上有所有服务的地址映射表. 当服务消费者调用某个目标服务时, 向负载均衡器发送请求, 负载均衡器根据策略做负载均衡后将请求转发到目标服务. 这种方式实现简单, 服务消费者无需实现服务发现逻辑, 但系统中需要维护一个高可用的负载均衡组件. 另外, 负载均衡组件在服务消费者和服务提供者之间增加了一跳, 有一定性能开销.

② 消费者服务发现

服务消费者要访问某个服务时, 通过内置的负载均衡组件向服务注册表查询目标服务地址列表, 然后以某种负载均衡策略选择一个目标服务地址, 最后向目标服务发起请求. 这一方案需要一个服务注册表配合服务注册和发现, 一般采用高可用、分布式一致的组件如 Zookeeper、Consul、Etcd 等来实现. 原理如图 2 所示.

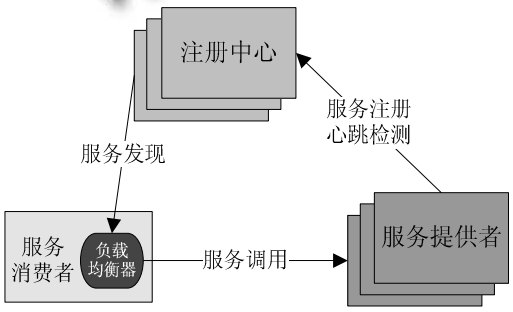

图 2　消费者服务发现

和方案一相比, 该方案将服务发现能力分散到每一个服务消费者的进程内部, 同时服务消费方和服务提供方之间是直接调用, 没有额外开销, 性能比较好.

微服务框架在实现时采用消费者服务发现的方式.

2.2 会话管理

基于微服务框架的系统是一个分布式系统, 用户的一个请求可能跨越多个微服务, 而且一个服务可部署多个实例, 因此需要实现分布式环境下的 Session 管理, 以解决 Session ID 共享、Session 数据复制及 Session 生命周期管理等问题.

在分布式环境中, Session 管理通常有三种方式: Session 复制、粘性 Session 和缓存集中式管理. Session 复制, 将一台机器上的 Session 数据广播复制到集群中其它机器上; 粘性 Session, 当用户访问集群中某台机器后, 强制指定后续所有请求均落到此机器上; 缓存集中式管理, 将 Session 存入分布式缓存中, 当用户访问不同节点时先从缓存中获取 Session 信息. 这三种方式的对比如表 1 所示.

表 1　三种方式对比

	Session Replication	Session Sticky	缓存集中式管理
使用场景	机器较少, 网络流量较小	机器数适中、对稳定性要求不高	集群中机器数多、网络环境复杂
优点	实现简单、配置较少、当网络中有机器宕机时不影响用户访问	实现简单、配置方便、没有额外网络开销	可靠性好
缺点	广播式复制有延时, 带来一定网络开销	网络中有机器宕机时用户 Session 丢失、容易造成单点故障	实现复杂、稳定性依赖于缓存的稳定性、Session 信息放入缓存时要有合理的写入策略

为适应复杂的网络环境, 提高 Session 管理的可靠性, 在微服务框架实现时采用缓存集中式管理方式进行 Session 的管理.

3　框架实现

3.1 基础框架

目前开源的微服务框架主要有 Dropwizard 和 SpringBoot. 两个框架在实现技术上存在一定差异[7], 如表 2 所示.

表 2 Dropwizard 和 SpringBoot 对比

框架	HTTP	REST	JSON	Metrics	Health Check
Spring Boot	Tomcat Jetty	Spring	Jackson GSON Json-simple	Spring	Spring
Dropwizard	Jetty	Jersey	Jackson	Dropwizard Metrics	Dropwizard

Spring boot 聚焦于 Spring 应用, 可以借力 Spring 家族体系的其它成员, 完成通信、数据访问等功能, 体系庞大沉重; DropWizard 从前端网页、核心服务、资料库存取到资源监控, 提供了一个轻量级的开发架构, 更适用于云开发环境. 为了保持微服务轻量级特性及向云环境部署迁移, 应用框架在实现上选取了 DropWizard 框架.

DropWizard 是一个后台服务开发框架, 内置 jetty 服务器, 封装 jersey 容器, 帮助开发者快速的打造一个 Rest 风格的后台服务, 同时集成 hibernate4、log4j、slf4j、jackson 等开源组件. DropWizard 结构的微服务主要包括四部分[8]:

① Configuration/config.yml: 用于微服务配置信息的定义和读取.

② Application: 该服务的主入口, 定义服务使用的配置文件, 开放 Resource, 创建数据库连接对象等.

③ Resource: 定义一个资源, 包括如何获取该资源, 对该资源进行 get、post 请求时对应的业务逻辑.

④ HealthCheck: 随时检测当前服务是否可用.

Dropwizard 框架本身缺少依赖注入的支持, 在微服务框架中对 Dropwizard 进行改造, 引入轻量级依赖注入框架 Google Guice[9], 简化系统开发的难度. 整合代码如表 3 所示.

表 3 整合 Google Guice 代码

```
injector = createInjector(configuration);
injector.injectMembers(this);
//将注入的实例添加到dropwizard的管理中
addHealthChecks(environment, injector);
addProviders(environment, injector);
addInjectableProviders(environment, injector);
addResources(environment, injector);
addTasks(environment, injector);
addManaged(environment, injector);
```

3.2 服务注册发现

消费者服务发现需要一个服务注册表, 采用开源组件 Consul 来实现. 相对于 Zookeeper、Etcd 等其它服务注册组件, Consul 具有以下优点[10]: 支持多数据中心下分布式高可用的服务发现和配置共享; 成员管理和消息广播采用 Gossip 协议, 去中心化; 支持健康检查, 允许键值对存储; 支持 ACL 访问控制. 服务注册发现的架构如图 3 所示.

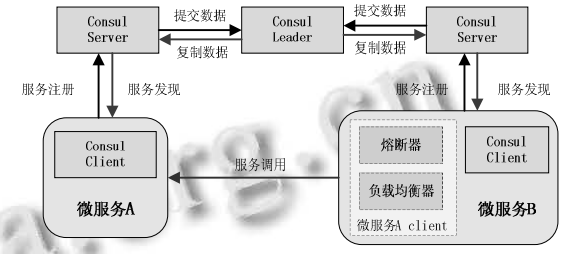

图 3 服务注册发现架构图

微服务启动时, 将服务信息注册到 consulClient, consulClient 将注册信息提交给 consulServer, consulServer 将信息提交给 consulLeader, consulLeader 将自身的数据复制给其他的 consulServer, 实现所有服务信息同步. 当微服务 B 访问微服务 A 时, 首先从 consulServer 上获取微服务 A 所有可用的服务地址, 根据负载均衡策略选择一个进行访问, 访问的过程中通过熔断器来进行超时容错处理.

Consul 采用 Go 语言进行编写, 引入开源的 consul-client 库实现 Java 对 Consul 的访问. 在微服务启动的时候进行服务注册, 注册代码如表 4 所示.

表 4 服务注册代码

```
AgentClient agentClient = consul.agentClient();
try {
    agentClient.register(prop.getServicePort(),
        URI.create(prop.getHealthUrl()).toURL(),
        prop.getHealthInterval(),
        prop.getServicename(),
        prop.getServicename(), // serviceId:
        prop.getServiceTag());
} catch (MalformedURLException e) {
    logger.error ("服务注册异常: " + e.printStackTrace());
}
```

3.3 Session 管理

为了解决分布式环境中 Session 共享的问题, 微服务框架在实现时选择分布式缓存方式进行 Session 的管理. 通过对网络 I/O 模型、内存管理、数据一致性、存储方式等方面对开源的分布式缓存组件进行对比,

最终选择 redis 作为缓存组件.redis 是一个开源的 Key-Value 存储系统, 具有快速、支持数据类型丰富、所有操作都是原子操作等优点. 同时使用 Haproxy 组件实现负载均衡和 redis 故障迁移, 保证 Session 信息的高可用性. 架构如图 4 所示.

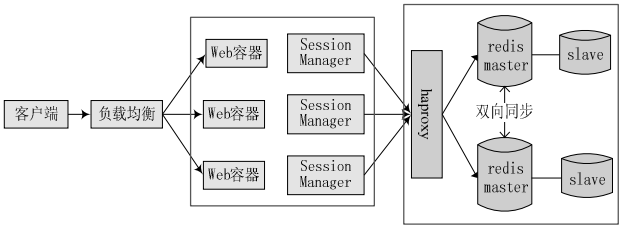

图 4 Session 管理架构图

当 redis master 故障时, 通过 haproxy 设置 redis slave 为临时 master, redis master 重新恢复后, 再切换回去. redis-master 与 redis-slave 双向同步, 解决 redis 单点问题.

Session 共享原理: 借助 HttpServletRequest Wrapper 对 HttpServletRequest 对象进行包装, 覆盖 getSession()方法, 接管创建和管理 Session 的工作. 通过 Filter, 完成请求包装操作, 并触发事件完成 Session 的保存. Session 保存流程如图 5 所示.

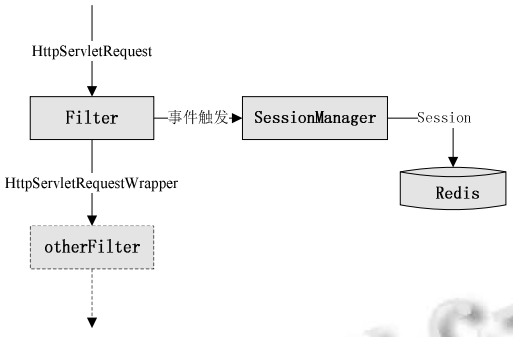

图 5 Session 保存流程

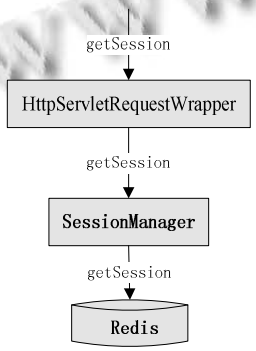

图 6 Session 获取流程

由于对 HttpServletRequest 进行了包装, 当系统中获取 Session 时, 调用包装类的 getSession 方法从 redis 缓存中获取 Session, 流程如图 6 所示.

4 结语

微服务框架已经在国家电网统一应用开发平台 V2.9.0 版本中实现, 该版本经过第三方安全、性能测试, 于 2016 年 7 月 27 日正式发版. 目前, 已选中两个项目组进行框架的试点应用.

本文对目前流行的微服务框架进行分析, 提出了一种微服务框架的实现方案. 该方案设计了微服务框架的功能架构, 重点分析了微服务架构所带来的服务注册、服务发现、Session 管理等问题, 并给出了实现方案. 基于该微服务框架进行业务系统开发, 开发人员只需要关注微服务内部业务功能的开发, 微服务之间的注册、发现、监控和 Session 管理由微服务框架完成, 简化了系统开发难度, 提高开发效率.

参考文献

1 温昱.软件架构设计:程序员向架构师转型必备.北京:电子工业出版社,2012.
2 王磊.解析微服务架构(一)单块架构系统以及其面临的挑战 [2015-05-11].http://www.infoq.com/cn/articles/analysis-the-architecture-of-microservice-part-01.
3 Richardson C. Introduction to Microservices [2015-05-19]. https://www.nginx.com/blog/introduction-to-microservices/
4 Fowler M, Lewis J. Microservices. [2014-03-25]. http://martinfowler.com/articles/microservices.html.
5 Parmar K. Microservice Architecture-A Quick Guide [2014-06]. http://www.kpbird.com/2014/06/microservice-architecture-quick-guide.html.
6 Richardson C. Introduction to Microservices [2015-05-19]. https://www.nginx.com/blog/introduction-to-microservices/.
7 Ullah R. Dropwizard vs Spring Boot—A Comparison Matrix. https://dzone.com/articles/dropwizard-vs-spring-boot?utm_source=tuicool&utm_medium=referral,2015-02-02.
8 Yammer.Getting Started.http://www.dropwizard.io/0.9.1/docs/getting-started.html.
9 Sameb. Getting Started. [2014-07-08]. https://github.com/google/guice/wiki/GettingStarted.
10 HashiCorp. Introduction to Consul. https://www.consul.io/intro/index.html.