一种基于微服务的应用框架①

张 晶[1], 黄小锋[2]

[1](北京中电普华信息技术有限公司, 北京 100192)
[2](中国水电顾问集团国际工程有限公司, 北京 100101)

摘 要: 采用组件化方式, 可以使应用系统的结构更加清晰, 简化大型系统开发和部署的难度. 然而传统的依靠 JAR 包或 OSGI 模块实现组件化的方式具有成本高、系统扩展性差和资源浪费等问题. 针对这些问题, 本文实现了一种基于微服务的应用框架, 通过将业务功能分解到各个离散的微服务中实现对系统功能组件的解耦. 基于该框架, 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、调用和监控由应用框架完成. 基于微服务的应用框架的使用可以简化系统开发难度, 降低代码修改、测试、打包以及部署的成本和风险; 功能模块按需扩展, 提升大型复杂业务系统运行期动态扩展能力; 将故障隔离在微服务内部, 提升系统的容错性.

关键词: 组件化; 微服务; 服务注册; 服务发现

Application Framework Based on Microservice

ZHANG Jing[1], HUANG Xiao-Feng[2]

[1](Beijing China Power Information Technology Co. Ltd, Beijing 100192, China)
[2](HydroChina International Engineering Co. Ltd., Beijing 100101, China)

Abstract: Componentization can make the structure of the application system more clearly, simplify the development, deployment and upgrades of large-scale systems. However, the traditional methods of componentization have some drawbacks like high cost, poor system scalability,thewaste of resources and so on. In order to solve the above problems, an application framework based on microservice is realized to functionally decompose the application into a set of collaborating services. Using this framework, developers just need to focus on the business function development, and service registration, service discovery, access and monitoring is done by the framework. This framework can simplify the development; reduce costs and risks of code changes, testing, packaging and deployment; enhance capacity of dynamic need expansion; improve fault inside isolation.，enhance the fault tolerance of the system.

Key words: componentization; microservice; service registration; service discover

随着企业级应用系统的规模和复杂度不断增加, 系统变得更加笨重和庞大, 这对系统的开发和部署提出了新的挑战, 对企业应用架构的高效性和扩展性提出了更高的要求.

组件化是解决复杂性问题的重要工具, 可以把庞大的应用程序分割成多个组件, 每个组件完成独立的功能, 组件之间协同工作[1]. 这些组件可以进行单独开发、单独编译、单独测试, 把所有的组件组合在一起得到了完整的系统. 采用组件化的应用结构, 使系统的结构更加清晰, 简化大型应用系统在开发、部署和升级的难度, 实现多种业务功能的分离[2].

传统实现组件的方式是通过库(library)或者 OSGI 框架.

采用库的方式是将通用的工具类打成 JAR 包放到 Web 应用相应的目录下, 如 J2EE 平台的 WEB-INF/lib 目录. 通过库实现组件化的方式, 只是对后端工具类代码进行了封装, 无法根据业务功能进行完整切分, 各个业务功能的代码混杂在一起, 调试和部署难度大,

① 收稿时间:2016-01-14;收到修改稿时间:2016-02-29 [doi: 10.15888/j.cnki.csa.005347]

一个或多个 JAR 包的修改, 需要重启整个应用.

OSGI 是一种动态模块化标准, 它为模块化应用的开发定义了一个基础架构[3]. 基于 OSGI 架构的应用程序由一系列 OSGI 模块组成, 在开发期, 可以实现模块的物理分离, 每个业务模块独立开发和调试, 但是在部署运行时, 仍需要将多个模块耦合在一起发布到 Web 容器上. 这种方式带来如下问题: 首先, 全部模块在一个 Web 应用中加载启动, 对资源要求高, 需要高配置的服务器或集群来支撑系统的运行, 部署成本高, 而且不利于向云环境扩展. 另外, 不同的业务模块所需资源不同, 有些可能是 CPU 密集型的, 另一些可能是 IO 密集型的, 在这种架构模式下, 各个模块不能独立部署运行, 当出现性能问题时只能扩展整个业务系统, 而不能针对模块扩展其资源能力, 系统扩展性差, 而且相同的模块在不同业务系统中需要重复部署, 造成了资源浪费.

随着微服务架构的出现以及在国内外的成功应用, 基于微服务实现组件化成为系统组件化的新选择. 通过微服务来实现组件, 将应用拆分为一系列的服务, 组件的边界更加清晰, 组件之间的耦合度降低. 服务运行在不同的进程中, 单一服务的变化只需重新部署相应的服务, 系统的扩展性得到提升.

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.

微服务架构的应用也会引入新的问题: 微服务应用是分布式系统, 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; 一个微服务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上时, 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. 这些都是在采用微服务实现组件化时需要解决的问题.

2 应用框架设计

为了降低大型复杂业务系统的开发难度, 解决传统依靠 JAR 包或 OSGI 模块实现组件化成本高、系统扩展性差和资源浪费的问题, 本文实现了一种基于微服务的应用框架. 该框架采用一组微服务的方式来构建一个应用, 通过将业务功能分解到各个离散的微服务中实现对系统功能组件的解耦. 基于该框架, 业务系统的开发被分解成若干微服务的开发, 开发人员只需要关注微服务内部业务功能的逻辑实现, 微服务之间的注册、发现和远程调用由微服务框架完成. 基于微服务的应用框架实现企业级应用系统功能组件服务化, 实现组件分离、共享、动态扩展的目标. 其中关键部分是应用框架的设计、微服务的动态注册及透明化的服务通信.

整个应用框架包括两部分: 微服务运行时容器和本地服务注册中心. 应用框架结构如图 1 所示.

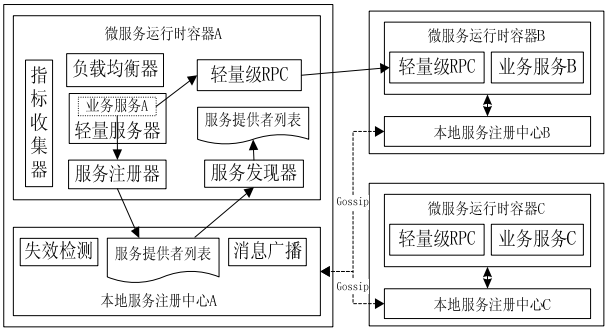

图 1　应用框架结构图

微服务运行时容器为微服务的运行提供支撑; 本地服务注册中心负责微服务的注册发现, 注册中心之间消息同步和事件广播, 微服务集群组建及集群中物理机的状态监测.

2.1 运行时容器

微服务运行时容器主要包括轻量级服务器, 运行指标收集器、服务发现器、服务注册器、负载均衡器、轻量级 RPC 等部件. 轻量级服务器是一个以嵌入式方式运行的应用服务器, 使微服务可以直接启动, 无需再部署到应用服务器. 负载均衡器采用软负载均衡及容错机制, 可替代 F5 等硬件负载均衡器, 降低成本. 轻量级 RPC 提供基于接口方法的透明远程过程调用, 就像调用本地方法一样调用远程方法, 只需简单配置, 没有任何 API 侵入.

2.2 本地注册中心

传统的分布式服务框架多采用中心化的架构, 服务提供者将服务注册到注册中心服务器, 服务消费者从注册中心服务器取得服务列表, 一旦注册中心服务器出现问题, 服务注册和发现无法生效, 整个系统就会瘫痪; 而且所有服务向一个注册中心服务器进行远程注册, 当服务数量过大时, 会消耗大量的网络资源并带给注册中心服务器很大的压力. 为解决上述问题, 在基于微服务的应用框架中设计了一种去中心化的方案, 提供本地化的服务注册和发现功能, 以提高系统的容错能力, 减少网络资源的消耗.

本地服务注册中心作为独立的可执行程序, 在集群节点的物理机上运行, 提供服务注册、服务发现和服务刷新功能. 注册中心之间通过 Gossip 协议进行通讯, 实现不同物理机之间服务信息的同步. 本地服务注册中心维护服务提供者列表, 服务提供者列表是一个服务接口名和服务地址的映射表, 保存了业务系统内所有微服务对外提供的服务信息.

2.3 注册发现机制

2.3.1 服务注册

在微服务容器启动时, 服务注册器向本地注册中心发送注册请求, 请求参数是微服务访问地址、提供的服务列表、消费的服务列表. 本地服务注册中心接收到请求后, 在注册中心本地的哈希表中添加该注册信息, 并返回注册成功与否的状态信息. 服务注册器收到应答信息后, 通过状态信息判断是否注册成功.

2.3.2 服务发现

微服务运行时容器通过服务发现器, 向本地服务注册中心发送服务发现请求, 获取所有的微服务注册信息. 服务注册中心接收到该请求后, 从本地的哈希表中获取所有微服务的注册信息, 同时生成 MD5 校验码, 将注册信息和校验码一并返回. 服务发现器接收返回数据后, 保存服务注册信息和 MD5 校验码.

2.3.3 服务刷新

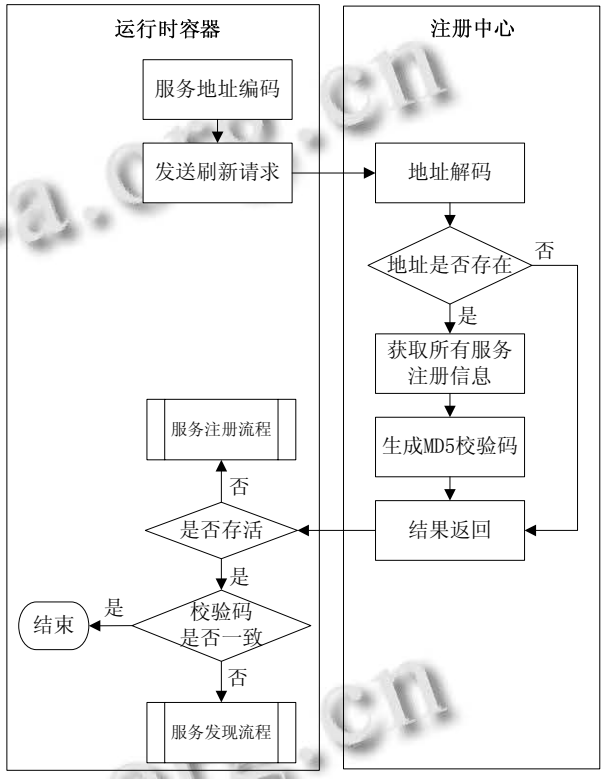

图 2 服务刷新流程图

微服务运行时容器通过服务刷新器周期性的进行服务刷新, 服务刷新的流程如图 2 所示. 服务刷新器将微服务的访问地址进行 Base64 编码后发送到服务注册中心. 服务注册中心接收到请求后, 对服务地址解码, 去本地的哈希表中查询该地址是否存在, 如果存在, 则服务注册中心认为该微服务存活, 否则认为微服务已经掉线. 当微服务处于存活状态时, 服务注册中心从本地的哈希表中获取所有微服务的注册信息并计算 MD5 校验码, 最终将存活状态信息和校验码一并返回. 服务刷新器接收返回数据后, 判断存活状态为掉线时, 调用服务注册器重新注册. 否则将本地保存的校验码与返回的校验码比对, 如果不一致, 表明服务注册中心的注册信息发生变化, 运行时容器调用

服务发现器去注册中心重新获取, 保证微服务容器的服务提供者列表是最新的数据.

3 应用框架实现

3.1 运行时框架

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[7,8], 如表 1 所示.

表 1 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[9]:

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 框架本身缺少对 RPC 远程过程调用和依赖注入的支持, 而且没有服务注册发现机制, 因此在运行时框架中对 Dropwizard 进行改造, 加入轻量级 RPC 框架、轻量级依赖注入框架和服务注册发现机制, 并在微服务启动时完成这三部分的加载. 加载代码如表 2 所示.

表 2 加载代码

```
// 加载MicroIoc
ioc.scanPackages(environment.getApplicationContext().getClassLoader(),
        this.getClass().getPackage().getName());
// 初始化Micro-Service RPC服务器
MicroServiceServlet mss = new MicroServiceServlet(ioc);
environment.servlets().addServlet("MicroService-Servlet",
mss).addMapping(
        MicroServiceServlet.CONTEXT_PATH + "/*");
// 初始化服务发现代理
environment.lifecycle().addServerLifecycleListener(
         new DiscoveryListener(ioc, config, environment));
launch(config, environment);
```

服务发现代理用于实现服务的注册、路由获取及服务定时刷新功能.

表 3 服务发现代理代码

```
try {
for (final Connector connector : server.getConnectors()){
   if (APPLICATION_CONNECTOR.equals(connector.getName())
          &&HTTP_PROTOCOL.equals(
             connector.getDefaultConnectionFactory().getProtocol())){
      final ServerSocketChannel channel = (ServerSocketChannel)
connector.getTransport();
      appPort = ((InetSocketAddress) channel.getLocalAddress()).getPort();
      logger.info("Using '{}' as application port", appPort);
      break;
      }
}
} catch (IOException e) {
  logger.error("Error getting local Port addresses", e);
  return;
}
String appAddr =    "http://" + appIp + ":" + appPort +
MicroServiceServlet.CONTEXT_PATH;
proxy = new DiscoveryProxy(client, bluedAddr, appAddr);
microApp = buildMicroApp(appAddr);
// 注册
proxy.register(microApp);
// 取路由表
fetch();
// 定时刷新
refresh();
```

3.1 服务注册中心

微服务应用是一个分布式系统, 具有集群特性, 系统中的节点通过 Serf 进行管理和同步. Serf 是一个

去中心化的分布式组件, 采用 Go 语言实现, 基于 Gossip 协议, 主要用于集群成员管理(Membership)、失败检测(Failure detection and recovery)和客户事件扩展(Custom event propagation)等功能, 轻量、高可用, 同时具备容错的特性[10].

Serf 实现了对集群节点的管理, 但缺少服务管理功能. 因此服务注册中心在 Serf 的基础上通过 Gin 框架实现服务注册、服务发现等服务管理的功能. Gin 是基于 GO 的 WEB 开发框架, 它具有类似 Martini 风格的 API 接口, 可以方便的编写后端服务代码, 对外提供 Restful 服务, 而且能够提供高性能的访问速度. 通过 Gin 框架可以快速创建 HTTP 服务器, 代码如表 4 所示.

表 4 HTTP 服务器创建代码

```
func NewHttpServer(addr string, logger *log.Logger, repo
*msd.DiscoverdRepo) *HttpServer {
        rs := msd.NewServiceResource(repo, logger)
        router := gin.Default()
        router.PUT("/msd/register", func(c *gin.Context) {
                rs.RegMicroApp(c)
        })
        router.GET("/msd/refresh/:addr", func(c *gin.Context) {
                rs.Refresh(c)
        })
        router.GET("/msd/fetch/:addr", func(c *gin.Context) {
                rs.GetRouterTable(c)
        })
        return &HttpServer{
                addr:     addr,
                logger: logger,
                server: router,
        }
}
```

通过该服务器对外发布服务注册、服务刷新、服务发现功能, 服务注册、服务刷新和服务发现对应的后端逻辑如表 5 所示.

表 5 服务注册、刷新及发布

```
func (sr *ServiceResource) RegMicroApp(c *gin.Context) {
        var as api.MicroApp
        if !c.Bind(&as) {
                c.JSON(http.StatusBadRequest, api.NewError("problem
decoding body"))
                return
        }
        sr.repo.Register(&as)
        c.JSON(http.StatusCreated, nil)
}
func (sr *ServiceResource) Refresh(c *gin.Context) {
        addr, err :=
base64.StdEncoding.DecodeString(c.Params.ByName("addr"))
        if err != nil {
                c.JSON(http.StatusBadRequest, api.NewError("error
decoding addr"))
                return
        }
        appStatus := sr.repo.Refresh(string(addr))
        c.JSON(http.StatusAccepted, appStatus)
}
func (sr *ServiceResource) GetRouterTable(c *gin.Context) {
        addr, err :=
base64.StdEncoding.DecodeString(c.Params.ByName("addr"))
        if err != nil {
                c.JSON(http.StatusBadRequest, api.NewError("error
decoding addr"))
                return
        }
        rt := sr.repo.GetRouterTable(string(addr))
        c.JSON(http.StatusOK, rt)
}
```

4 结语

本文实现了一种基于微服务的应用框架, 通过微服务实现企业级应用系统功能组件化、组件共享和组件动态扩展的目标. 该框架解决了微服务架构所带来的服务发现、服务通信、服务监控等问题, 简化了系统开发调试的难度, 提高了开发效率. 基于该框架开发的业务系统功能模块按需扩展, 系统水平扩展能力强; 服务单独部署, 部署成本和维护成本低; 功能模块粒度更细、职责更单一, 利于软件资产复用, 降低成本.

参考文献

1 李品升.基于组件技术的软件系统模型研究与实现[硕士学位论文].沈阳:沈阳师范大学,2012.
2 宋海荣.组件技术的研究及在商标审查系统中的应用[硕士学位论文].北京:北京邮电大学,2013.
3 梁小江.基于 OSGi 的构件库系统设计与实现[硕士学位论

文].西安:西安电子科技大学,2010.

4 Fowler M, Lewis J. Microservices. http://martinfowler.com/articles/microservices.html. [2014-03-25].

5 肖勤.微服务架构实践经验分享. http://www.csdn.net/article/2015-08-07/2825412.[2015-08-07].

6 Richardson C. Introduction to Microservices. https://www.nginx.com/blog/introduction-to-microservices/.[2015-05-19].

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 Zhitnitsky A. Java Bootstrap: Dropwizard vs. Spring Boot. http://blog.takipi.com/java-bootstrap-dropwizard-vs-spring-boot/?utm_source=tuicool&utm_medium=referral. [2015-03-02].

9 Yammer. Getting Started. http://www.dropwizard.io/0.9.1/docs/getting-started.html.

10 HashiCorp. Introduction to Serf. https://www.serfdom.io/intro/index.html.