软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(11):4061−4083 [doi: 10.13328/j.cnki.jos.006377]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于动态和静态分析的单体应用 FaaS 改造方法*

向麒麟[1,2], 彭 鑫[1,2], 赤坂居纱美[1,2], 李博文[1,2]

[1](复旦大学 计算机科学技术学院, 上海 201203)

[2](上海市数据科学重点实验室(复旦大学), 上海 201203)

通信作者: 彭鑫, E-mail: pengxin@fudan.edu.cn

摘 要: 作为 Serverless 架构的一种典型形态, 函数即服务(function as a service, FaaS)架构将业务抽象为细粒度的函数, 并且提供弹性的自动伸缩等自动化运维功能, 能够大幅降低运维成本. 当前, 许多在线服务系统中的一些高并发、高可用、灵活多变的业务(如支付、红包等)都已经迁移到了 FaaS 平台上, 但是大量传统单体应用还是难以利用 FaaS 架构的优势. 针对这一问题, 提出了一种基于动态和静态分析的单体应用 FaaS 改造方法. 该方法针对指定的单体应用 API, 通过动态分析和静态分析相结合的方式识别并剥离其实现代码和依赖, 然后按照函数模板完成代码重构. 针对函数在高并发场景下的冷启动问题, 该方法利用基于 IO 多路复用的主从多线程 Reactor 模型优化了函数模板, 提高了单个函数实例的并发处理能力. 基于该方法实现了针对 Java 语言的原型工具 Codext, 在开源 Serverless 平台 OpenFaaS 上, 面向 4 个开源单体系统进行了实验验证.

关键词: 单体应用; 无服务器架构; 函数即服务(FaaS); 轨迹分析; 动态分析; 静态分析; 冷启动

中图法分类号: TP311

中文引用格式: 向麒麟, 彭鑫, 赤坂居纱美, 李博文. 基于动态和静态分析的单体应用 FaaS 改造方法. 软件学报, 2022, 33(11): 4061−4083. http://www.jos.org.cn/1000-9825/6377.htm

英文引用格式: Xiang QL, Peng X, Akasaka I, Li BW. FaaS Migration Approach for Monolithic Applications Based on Dynamic and Static Analysis. Ruan Jian Xue Bao/Journal of Software, 2022, 33(11): 4061−4083 (in Chinese). http://www.jos.org.cn/1000-9825/6377.htm

FaaS Migration Approach for Monolithic Applications Based on Dynamic and Static Analysis

XIANG Qi-Lin[1,2], PENG Xin[1,2], AKASAKA Isami[1,2], LI Bo-Wen[1,2]

[1](School of Computer Science, Fudan University, Shanghai 201203, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 201203, China)

Abstract: As a typical form of the Serverless architecture, the function as a service (FaaS) architecture abstracts the business into fine-grained functions, and provides automatic operation and maintenance functionality such as auto-scaling, which can greatly reduce the operation and maintenance costs. Some of the high concurrent, high available, and high flexible services (such as payment, red packet, etc.) in many online service systems have been migrated to the FaaS platform, but a large number of traditional monolithic applications still find it difficult to take advantage of the FaaS architecture. In order to solve this problem, a FaaS migration approach for monolithic applications based on dynamic and static analysis is proposed in this study. This approach identifies and strips the implementation code and dependencies for the specified monolithic application API by combining dynamic and static analysis, and then completes the code refactoring according to the function template. Aiming at the cold-start problem of functions in high concurrency scenario, this approach uses the master-slave multithreaded Reactor model based on IO multiplexing to optimize the function template and improve the concurrency processing capability of a single function instance. Based on this approach, Codext, a prototype tool for Java language, is implemented and experimental verification is carried out on OpenFaaS, an open source Serverless platform, for four open source monolithic applications.

* 基金项目: 国家重点研发计划(2018YFB1004803)

收稿时间: 2020-12-22; 修改时间: 2021-02-24, 2021-04-21; 采用时间: 2021-05-18; jos 在线出版时间: 2021-12-24

Key words: monolithic application; Serverless; function as a service (FaaS); trace analysis; dynamic analysis; static analysis; cold start

传统的单体架构软件应用将所有代码一起打包、构建和部署. 随着需求的变化和业务的增长, 单体架构应用将变得越来越臃肿、代码复杂度不断提高、技术债务不断积累、维护成本大幅度提高, 从而导致新特性的交付周期变长、新技术难以应用、外部需求变化以及业务增长带来的性能需求难以满足[1,2]. 随着虚拟化技术和容器技术的发展, 微服务架构逐渐成为基于云的主流软件架构选择. 微服务架构将单体应用拆分成一组在独立进程中运行并通过轻量级机制进行通信的服务, 这些服务围绕业务功能构建, 可以灵活选择技术栈(如编程语言和数据存储技术), 并独立开发、部署和维护. 微服务架构更加灵活和敏捷, 但运维负担也更重, 并且需要更加完善的基础设施支持(如服务注册、发现、伸缩、监控和配置管理等). 随着微服务技术的发展和云计算基础设施的完善, Serverless 架构应运而生. Serverless 架构是对云计算基础设施各种能力的抽象, 将其通过更加简单的形式暴露给开发者, 并内置了丰富的功能, 例如应用的运行时监控、应用模板、自动伸缩、应用开发的常用组件等. 因此, 开发者只需要专注于业务代码, 就可以在对服务器无感知的情况对应用进行部署, 并且无需关注服务端的运维工作[3,4].

函数即服务(function as a service, FaaS)是 Serverless 架构的一种典型形态. FaaS 将应用的业务逻辑封装为在无状态的容器中运行、由事件触发、临时性(可能只一次调用即释放)并且完全由第三方云平台管理的函数[4]. 第三方云平台通常会为 FaaS 提供多语言的函数模板, 这些模板提供了除业务逻辑以外的所有必需的函数组件(应用服务器、函数接口的封装等). 函数基于函数模板进行开发和构建, 由平台自动托管部署、自动伸缩和运维管理. 这种方式进一步降低了应用构建、打包、部署的复杂性和运维成本, 同时缩短了应用开发生命周期、提高了业务敏捷性. DataDog 在 2020 年 2 月基于被广泛使用的 Serverless 平台 AWS Lambda 发布的“Serverless 现状”报告[5]显示: 一半以上的 AWS 用户都使用了 AWS Lambda 函数计算服务; 而大型环境中, 超过 3/4 的用户以及超过 80%的容器用户都使用了该服务.

为了应对业务增长和需求变化, 许多采用单体架构的企业都选择向更适应云计算特点的架构演进. 许多企业选择将单体应用重构拆分为微服务应用, 但这一过程十分困难. Netflix 公司花费了 7 年时间才完成从单体到微服务的架构迁移[6]. 微服务架构强调服务划分的合理性, 不正确、不合理的微服务架构反而会加大系统调试和故障定位的难度[7,8]. 微服务重构拆分理论复杂, 同时缺乏正确的建模语言和拆分工具支持[9]. 除此之外, 遗留单体应用因为文档缺失、技术老化、架构紊乱、规模庞大, 也增加了单体应用微服务化改造的难度. 事实上, 一个单体应用中经常只有部分模块或 API, 难以满足业务增长带来的压力, 因此可以只针对相关模块或 API 进行面向云计算平台的迁移[10−12]. FaaS 架构允许将 API 直接实现为一个相对独立、可托管运行的函数, 因此可以将应用的部分 FaaS 化改造作为一种替代方案. 其中的函数抽取需要剥离函数实现代码及其依赖, 同时按照函数模板进行重构. 此外, 函数在高并发场景下可能存在冷启动问题, 即函数还没来得及扩容流量, 波峰就已经过去. 为此, 还需要对函数进行相应的性能优化.

针对以上问题, 本文提出了一种基于动态和静态分析的单体应用 FaaS 改造方法. 该方法能够针对指定的 API 进行函数抽取和模板化改造, 并进行相应的性能优化. 该方法通过动态分析和静态分析相结合的方式获得指定 API 的完整方法调用执行轨迹, 其中, 动态分析通过运行测试用例获得动态执行轨迹, 静态分析补充未覆盖的执行分支和方法调用. 在此基础上, 通过测试驱动的方法确保指定 API 的实现代码及相关的配置和依赖都已实现完整剥离, 然后自动将相关实现代码及其配置和依赖迁移至函数模板. 此外, 我们利用基于 IO 多路复用的主从多线程 Reactor 模型[13,14]优化了函数模板, 提高了单个函数实例的并发处理能力, 减轻了高并发场景下函数实例频繁扩容导致的性能(冷启动[9])问题.

目前, FaaS 平台最流行的程序设计语言为 Python 和 NodeJS, 因为其语言特性能有效避免冷启动问题, 大量新的 FaaS 应用都使用了 NodeJS 和 Python. 但是在过去很长一段时间内, Java 为构建构建大型应用的主要语言, 很多遗留应用也为 Java 应用, 其中蕴含着大量的改造需求. 为了验证方法的可行性, 我们基于字节码增强技术(Javaagent/Javasist[15])、静态代码分析器 Javapaser[16]、高性能的事件驱动的 NIO 网络应用框架 Netty[17]

和开源 Serverless 平台 OpenFaaS[18]开发了针对使用 Maven 管理的 Java 单体应用的 FaaS 迁移辅助工具 Codext, 并且在多个开源单体应用上进行了验证实验. 实验结果表明: 该工具能够有效地从单体应用中抽取指定 API 的实现代码并迁移至 FaaS 平台, 极大地降低了开发人员的负担. 同时, 我们所实现的函数模板优化可以大幅度提升函数的性能, 有效地解决高并发场景下的性能问题.

1 背景及相关工作

单体应用的重构、改造和升级一直是软件工程领域的一个热门话题. 近几年来, 由于微服务架构的兴起, 越来越多的单体应用向微服务架构迁移[19]. Francesco 等人[10]报告了一项单体应用向微服务架构迁移的实证研究, 他们访谈了 18 位参与微服务架构迁移的工程师, 并且从架构迁移的 3 个方面(逆向工程、架构迁移和实现新系统)搜集了迁移过程的挑战, 总结了未来潜在相关的研究方向. Taibi 等人[11]通过一项实证研究总结一个由 3 部分组成的迁移过程框架, 分别为现有功能迁移、重新开发以及实现新功能. 这些工作聚焦于实证研究和方法论的研究, 主要为开发者在单体应用向微服务架构迁移过程中提供辅助性建议, 并不产生相关的工具或算法直接解决迁移过程中存在的问题.

目前, 已经有了很多关于单体应用的微服务化拆分的算法和工具. Abdullah 等人[20]使用日志和无监督学习的方法将系统自动分解为具有相似性能和资源要求的 URL 组, 将每组 URL 映射为一个微服务. 该方法将功能相似性作为唯一的标准, 没有考虑到横切性功能模块的拆分; 同时, 无监督学习可解释性较差, 无法提供充分的拆分理由. Service Cutter[21]是一个用于服务拆分的可视化工具, 通过用户输入自定义的接口操作和用例, 根据操作间的耦合程度对接口进行聚类. Mazlami 等人[9]提出了基于代码块的依赖信息的 3 种耦合策略和基于图的聚类算法, 从而辅助微服务拆分的决策. 以上两种方式武断地认为接口操作的耦合和代码耦合就可以反映服务的拆分结构, 并没有考虑到微服务围绕业务功能构建的特性. Jin 等人[22]通过监控系统动态收集代码的执行路径, 实现了一种面向功能的微服务抽取方法. Ding 等人[2]通过对运行时代码执行轨迹、数据库访问信息和数据表信息进行加权和聚合, 并且结合了代码静态分析, 提出了一种场景驱动的、自底向上的单体系统微服务拆分方法. 这两种方法考虑了微服务围绕业务功能构建的特性, 但该类型方法只给出了单体应用代码和数据表的拆分建议, 并没有提供相关的自动化工具, 以实现从单体应用到微服务的自动化拆分(比如代码和数据库的拆分).

由于微服务架构固有的复杂性, 微服务拆分过程需要考虑诸多因素, 以上所提到的研究都只是从某一个或几个方面来对单体应用进行微服务拆分, 缺少对微服务特性的全面分析. 微服务化至少需要对代码和数据库进行拆分, 需要解决前后台分离、服务通信方式重构、代码重构和迁移、数据表修改以及存储过程和数据库事务应该如何迁移等问题[1,10,11,23]. 除此之外, 微服务重构拆分的结果难以评估. 到目前为止, 大多数工具和算法都是从内聚和耦合度来评估拆分结果, 并没有一套综合全面的评价标准, 这也导致上述工具和算法难以被工业界大型软件应用使用.

除了单体系统向微服务架构迁移固有的复杂性和缺乏拆分评价标准以外, 企业还需要完善的基础设施和运维人员来支撑上层微服务系统的运行和维护, 这极大地提高了企业的成本. 为了节省成本, 越来越多的微服务系统选择使用云服务. Villamizar 等人研究发现: 使用云服务的微服务架构比单体应用节省 70%的成本, 使用 Serverless (AWS Lambda)的微服务系统比使用云服务自运维的微服务系统更加节约成本[24].

为了避免大量的运维成本开销和微服务拆分的复杂性, 已经有不少的单体应用开始向 Serverless 架构迁移. Goli 等人[25]将 CPU 和数据密集型的 FinTech 应用的 4 个 API 从单体架构迁移到了 Serverless 架构, 不仅降低了运维成本, 也大幅度提高了性能. 同时, 他们使用了非阻塞 IO(异步 HTTP 请求)对 API 进行了优化, 提高了性能, 同时也节约了成本. 该工作依赖人工将代码迁移至 Serverless 架构, 对于大型系统来说, 这个过程十分消耗时间和人力; 并且对于遗留系统来说, 文档缺失和人员流动也极大提高了迁移的风险; 并且该工作忽略了冷启动带来的影响, 在大流量场景下, 很可能出现服务冷启动期间拒绝服务的情况. Kaplunovich 等人[26]开发了一个将单体应用自动迁移到 Serverless 架构的工具 ToLambda, 该工具自动化地将 Java 应用转化成

NodeJS 应用并部署到云提供商的 Serverless 平台(AWS Lambda, Azure Function, Google/IBM Cloud Function). 通过性能实验, 发现迁移后的 Serverless 架构应用与原单体架构相比, 性能有了大幅度提升. 该工作通过静态代码分析将 Java 应用转换成 NodeJS 应用, 其中还存在很多问题: (1) Java 和 NodeJS 是两种不同风格的语言, Java 强调面向对象设计, 是一门命令式编程语言, 而 NodeJS 是一门函数式编程语言; (2) 该工作没有考虑 Java 代码向 NodeJS 代码迁移的特性, 比如运行时多态、继承、接口实现、同步语句、泛型等; (3) 该工作没有考虑应用配置和依赖的迁移, 一般大型应用都存在大量的配置(中间件配置、数据库配置和资源使用配置等)和第三方依赖; (4) 该工作也忽略了迁移至 Serverless 架构冷启动的问题. 阿里云等云提供商支持某些框架的函数迁移, 比如针对 Java 的 SpringBoot 框架、针对 Glang 的 gin 框架[27], 但只提供了相关框架的模板, 并未提供自动化迁移功能. 这些框架内置服务器都支持一定的并发, 某些框架也使用了 IO 多路复用原理. 而本方法使用的函数模板比框架更加轻量级, 相对于这些框架具有更少的内置 API 和不必要的组件, 编译打包后的包更小, 启动时间更短. 由于 FaaS 的应用场景多为简单场景, 没有必要使用框架, 而只需将业务代码进行迁移.

冷启动问题是 Serverless 不容忽视的, 冷启动时间过长将会带来各类问题[3,28]. 到目前为止, 已经有一些与冷启动相关的工作. Baird 等人[29]对比了 AWS 上热启动和冷启动的性能差异, 其结果表明, 热启动性能明显好于冷启动. McGrath 等人[30]分析了不同平台冷启动的问题, 他们报告了 Azure, Google, OpenShift, AWS 等大型平台在最后一次调用后大约 15 分钟将会面临冷启动问题. Cui 等人[31]对比了不同语言、内存分配和部署包大小对冷启动的影响, 发现 NodeJS 和 Python 的启动速度最快, 其他语言则相对较慢; 内存分配越多, 函数启动时间越短; 同时, 部署包越小, 启动时间也越短. Silva[32]等人提出了一种启动函数的技术, 该技术从以前执行的函数进程中恢复快照来加快函数启动的过程, 并且他们还基 CRIU 进程检查点和 Linux 恢复工具开发了该技术的原型. 该工作需要针对不同的函数进行恢复快照, 使用门槛较高, 并且不保证能优化所有的函数进程, 这并不是一个普适的解决方案, 难以被大范围使用. Daly 等人[33]开发了 Lambda Warmer 的工具来减轻冷启动的影响, 该工具通过 CloudWatch Event 工具周期性地向 Lambda 函数发送 ping 指令以保持函数的活性. 该工具虽然一定程度上减轻了冷启动的影响, 但是却带来了额外的访问事件和成本开销, 并且 ping 指令发送周期难以确定. OpenFaaS 建议了两种方式来解决冷启动的问题: (1) 将请求改造为异步调用; (2) 将需要使用的容器镜像提前拉取到相关的集群节点[34]. 将函数改造为异步调用的方式不能适用于所有的场景, 有一定局限性; 预拉镜像的方式可以部分解决冷启动问题, 但是对于自身启动时间较长的容器, 优化能力十分有限. 上述这些解决冷启动的方式可以总结为 3 类: 第 1 类方式为直接缩短容器从调度(schedule)到就绪(ready)的时间; 第 2 类方式为通过外部工具间歇性唤醒函数的方式来解决冷启动问题; 第 3 类方式为使用消息队列等方式将请求暂存, 然后异步进行处理. 前两种方式都没有考虑到在高并发场景下, 函数频繁水平扩容导致冷启动问题加剧的情况; 第 3 种方式对于一些需要同步返回请求的场景支持不足, 且需要对函数进行一定的改造.

本文提出了单体应用向 FaaS 迁移的方案, 充分参考了上述单体应用微服务化和 Serverless 化迁移工作的优缺点, 通过测试用例驱动得到运行时方法执行轨迹, 并结合静态代码分析的方式, 将有迁移需求的 API 相关代码迁移至 FaaS 函数模板, 并通过函数模板自动化生成容器镜像, 最后通过容器镜像实例化函数. 该迁移方法通过对指定 API 的迁移, 避免了微服务迁移的复杂性; 并通过平台托管治理维护的方式, 最大化降低了运维成本. 除此之外, 我们综合使用了代码依赖精简、基于 IO 多路复用的主从多线程 Reactor 的高性能函数模板等方式来降低冷启动带来的影响, 特别是在高并发场景下频繁扩容导致冷启动影响加剧的情况.

2 单体应用外部 API 迁移方法

几乎所有的软件应用系统中, 不管是使用命令式编程语言或者函数式编程语言编写, 其业务逻辑都封装在方法中. 一个对外提供服务的 API(下文称外部 API)通常包含多个内部方法, 一次对外部 API 的调用可能会经过部分或者全部这些内部方法. 这些方法可能会依赖其他代码文件中的定义的变量(常量)或者类型(比如 Java 语言中的 Class、Golang 和 C 语言中的 struct 等)以及第三方依赖提供的 API. 本文所提出的单体应用外部 API 迁移方法, 在测试环境中, 通过测试用例驱动指定外部 API 运行, 并通过运行时监控和静态代码分析得

到该外部 API 相关的内部方法调用; 然后以这些内部方法为主线, 分析其相关源码文件的抽象语法树(遍历抽象语法树, 对与外部 API 无关的代码元素进行剪枝); 最后, 通过剪枝后的抽象语法树生成该外部 API 相关的代码. 抽取出的代码通过人工调整和测试之后会被迁移到函数模板, 然后对迁移后的代码进行依赖精简, 最后通过函数模板构建函数容器镜像. 上述用到的动态分析、静态分析和依赖分析工具, 针对不同的语言有不同的实现, 因此本方法可以适用于不同的程序设计语言编写的应用程序.

本文所提出的方法适合于满足以下两个条件的单体应用后端的外部 API(即前端或第三方客户端可直接访问的 API): 无状态或具有状态管理机制(即状态可以迁移至外部存储); 具备较完备的 API 测试用例集(目的是驱动动态分析). 改造后的 FaaS 实现能够更好地应对突发或流量不可预测的场景, 并支持业务快速迭代.

方法流程如图 1 所示, 可以分为“外部 API 方法调用搜集”“外部 API 代码提取”“手工调整和测试”“函数模板构建和容器镜像生成”这 4 个部分. 第 1 部分“外部 API 方法调用搜集”涉及图 1 中的第 1 步–第 4 步,主要通过动态监控和静态代码分析(抽象语法树分析)得到外部 API 方法调用. 第 2 部分“外部 API 代码提取”涉及图 1 中的第 6–8 步, 这部分主要通过分析方法调用相关源码文件的抽象语法树, 然后通过剪枝去掉当前外部 API 未被依赖的抽象语法树元素, 最后通过剪枝后的抽象语法树生成源码. 第 3 部分“手工调整和测试”涉及图 1 中的第 9 步, 这部分主要为了保证函数在迁移前后语义一致. 主要步骤为在测试环境中的临时目录中反复修改第 8 步通过抽象语法树生成的函数代码, 直到其可以正确编译、构建并成功通过所有测试用例. 第 4 部分“函数模板构建和容器镜像生成”涉及图 1 中的第 10 步和第 11 步, 这部分主要将第 9 步手工调整和测试后的代码按照单体应用中的目录结构迁移至函数模板中, 并且精简第三方依赖, 剔除未使用的依赖. 然后通过镜像配置文件, 将函数模板中的函数构建、打包成容器镜像.

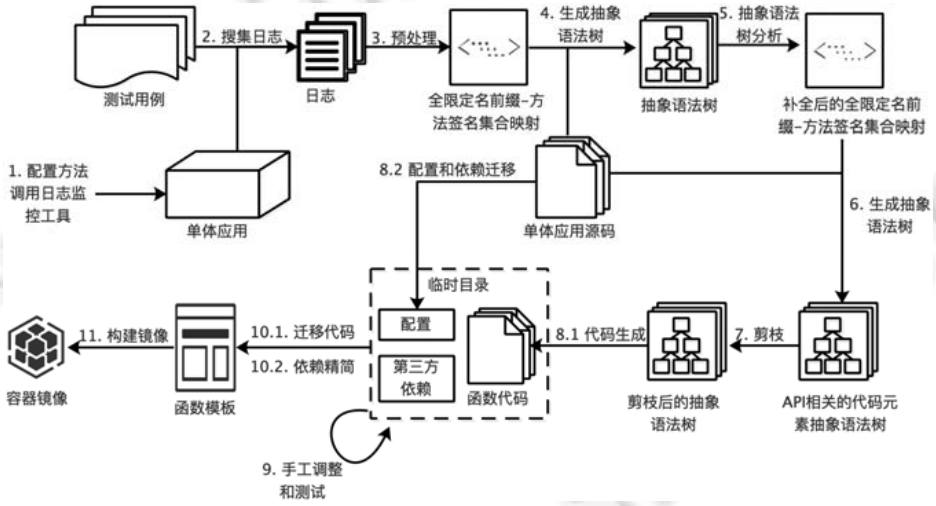


图 1 单体应用外部 API 迁移方法流程

2.1 外部API方法调用搜集

本文代码迁移的方法需要尽可能完整地搜集外部 API 依赖的方法. 目前主要存在两种方法搜集调用轨迹的方法, 即运行时动态分析和静态代码分析. 这两种方法都存在一定的缺陷: 动态分析过于依赖测试用例的完整性和分支覆盖度; 静态分析主要通过抽象语法树分析得到服务调用的拓扑结构, 难以分析程序运行时的行为, 比如动态绑定. 本方法充分考虑了动态分析和静态分析的缺陷, 使用测试用例驱动的运行时动态方法执行轨迹日志分析和静态代码抽象语法树分析相结合的方式搜集方法调用.

2.1.1 动态分析

动态分析大多数都是根据语言的特性, 采用不同的实现方式, 比如代码插桩、字节码增强等. 动态分析过程针对不同语言有不同工具, 配置过程和方法调用搜集过程也可能不同. 本文方法统一将其分别抽象为图 1

中的两个步骤，即第 1 步“配置方法调用日志监控工具”和第 2 步“搜集日志”. 如图 1 所示，首先，第 1 步先配置单体应用调用链监控工具(本文实现方式通过 javaagent/javasist 无侵入收集，参见第 4.1 节)，表 1 列举了针对主流编程语言可用的动态方法调用监控工具；第 2 步启动应用，运行需要迁移的 API 的测试用例，第 1 步配置好的监控工具会在运行时监控所有应用内部方法的调用情况，然后将方法调用信息记录在日志中. 方法调用信息包括日志前缀、全限定名前缀和函数签名. 日志前缀主要为了区别其他业务日志，全限定名前缀和方法签名用来唯一确定一个方法(参见第 4.1.1 节).

表 1　主流编程语言对应的动态调用监控工具

语言	工具	语言	工具
C	gprof/valgrind	C++	CodeViz/SourceInsight
Java	Java-callgraph	Python	pycallgraph
Golang	go-callvis	NodeJS	njsTrace

2.1.2　静态分析

静态分析主要通过分析代码抽象语法树补全动态分析中测试用例未覆盖的方法调用，主要基于动态调用轨迹，分析抽象语法树中动态分析未覆盖的方法调用，包括图 1 中的第 3 步“预处理”、第 4 步“生成抽象语法树”和第 5 步“抽象语法树分析”. 静态分析主要为抽象语法树分析，见表 2，不同语言有不同的抽象语法树分析工具.

表 2　主流编程语言对应的抽象语法树分析工具

语言	工具	语言	工具
C	Clang	C++	foonathan/cppast
Java	Javaparser	Python	内置 ast.py
Golang	fatih/astrewrite	NodeJS	ajaxorg/treehugger

图 2 左边虚线部分是第 3 步“预处理”的主要流程，包括过滤和生成两个步骤：预处理首先按行读取原始日志，然后通过日志前缀，从日志文件中过滤出监控工具打印的当前应用的方法调用日志，并以集合(图 2 中的“测试用例覆盖的方法调用集合”，后文称“方法调用集合”)的形式存入内存；然后将方法调用集合转化为以全限定名前缀名为键，方法签名集合为值的映射(图 2 中的“全限定名前缀-方法签名集合映射”，后文称“映射”). 一个全限定名前缀对应的源码文件可能声明多个方法，一次调用可能涉及其中的部分方法. 当分析一个被调用方法的声明的抽象语法树时，需要根据全限定名前缀和方法签名定位方法的源码文件，然后构建抽象语法树. 将属于同一源码文件的方法声明聚合，当分析用一个源码文件中多个方法声明时，只用构建一次抽象语法树，这样可以有效降低生成抽象语法树的次数，提高分析效率.

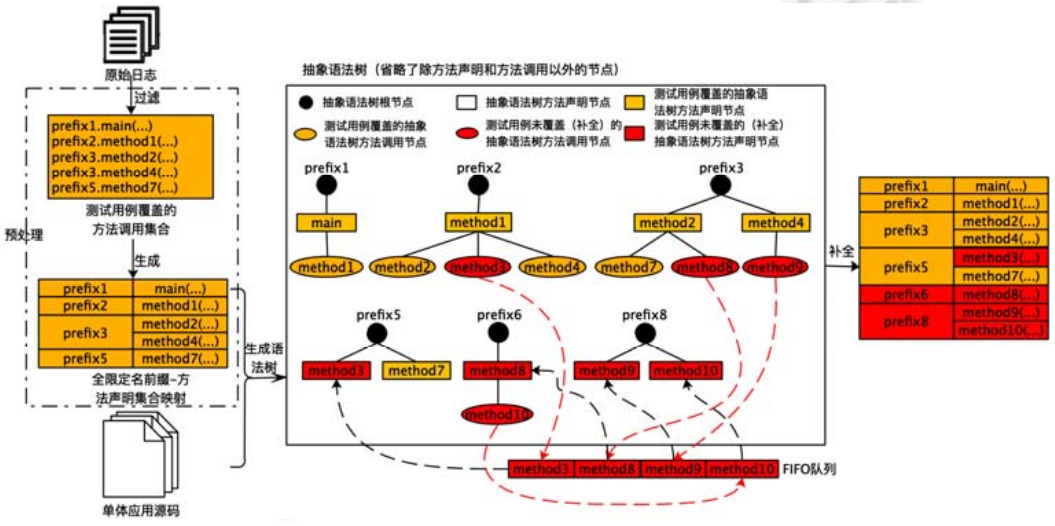


图 2　静态抽象语法树分析补全方法调用过程

算法 1 详细介绍了图 1 中第 4 步和第 5 步通过静态分析抽象语法树来补全测试用例未覆盖的方法调用过程, 具体过程如下.

(1) 遍历映射中的键, 通过键(全限定名前缀)和单体应用源码根目录 *RootPath* 定位源码路径, 然后生成对应的抽象语法树(算法 1 第 5–9 行).

(2) 使用算法 2 遍历第(1)步中抽象语法树的方法声明节点, 若方法声明对应的方法签名已经存在于映射的值(*allSignSet*)中, 则继续遍历该方法声明的方法调用子节点; 对于每一个方法调用节点, 若该方法调用节点对应的方法签名不存在于映射的值中, 说明该方法调用未被测试用例覆盖, 需要补全到映射中. 补全规则如下: 若该方法签名对应的全限定名前缀存在于映射的键的集合中, 则将该方法签名添加到该全限定名前缀对应的方法签名集合; 反之, 则插入一条以该全限定名前缀为键, 值为该方法签名集合的新记录(算法 1 第 8 行和第 15 行).

(3) 将第(2)步中测试用例未覆盖的方法调用节点加入 *FIFOQueue*(算法 1 第 11 行), 以便后续分析这些方法调用对应方法声明对其他方法的调用.

(4) 依次从 *FIFOQueue* 出队, 对于出队的每个方法调用节点, 得到其全限定名前缀和方法签名, 重复第(1)–(3)步, 直到 *FIFOQueue* 为空;

(5) 随后得到 *CompleteMap*, 即当前外部 API 依赖的每一个源码文件中的所有方法的签名.

算法 1. 调用链轨迹补全算法.

输入: 输入全限定名前缀-方法签名集合映射 *Map*(*Map*⟨*prefix*,*set*⟨*signature*⟩⟩)和源码根目录 *RootPath*.

输出: 补全后的全限定名前缀-方法签名集合映射 *CompleteMap*.

```
1:  function complete(Map,RootPath)
2:      CompleteMap←Map   //复制 Map 到 CompleteMap,存储最终补全的结果
3:      FIFOQueue   //记录测试用例未覆盖的方法调用
4:      根据 Map 得到键的集合 prefixSet 和所有方法签名的集合 allSignSet
5:      for p in prefixSet
6:          根据 p 和 rootPath 得到源码文件的路径 srcPath
7:          生成 srcPath 对应源码文件的抽象语法树 ast
8:          bfs(ast,allSigSet,prefixSet,FIFOQueue,CompleteMap)   //算法 2
9:      end for
10:     while fifoQueue≠∅
11:         methodCallExpr←fifoQueue.out(·)   //出队
12:         根据 methodCallExpr 获取对应方法的签名 signature 和全限定名前缀 prefix
13:         根据 prefix 和 rootPath 得到源码文件的路径 srcPath
14:         生成 srcPath 对应源码文件的抽象语法树 ast
15:         bfs(ast,allSigSet,prefixSet,FIFOQueue,CompleteMap)   //算法 2
16:     end while
17: return CompleteMap
```

算法 2 描述了基于广度优先遍历的方法调用轨迹补全算法, 该算法主要通过广度优先遍历(BFS)遍历抽象语法树的方法调用和方法声明语法树, 搜索 *CompleteMap* 中未覆盖的方法调用, 然后将未覆盖的方法调用添加到 *CompleteMap* 和 *FIFOQueue* 中. *CompleteMap* 用来记录所有代码文件和方法声明的映射关系; *FIFOQueue* 用来保存没有覆盖的方法调用, 以便后续分析其方法声明中其他的方法调用. 主要步骤如下.

(1) 遍历 ast 中所有的方法声明, 如果方法声明节点对应的方法签名存在于 *allSigSet* 中, 说明该方法被调用, 则继续分析其方法调用子语法树(理论依据: 在抽象语法树中, 未被测试用例覆盖的方法肯定是被测试用例覆盖的子树, 所有这里必须要求 ast 方法声明存在于 *allSigSet* 中).

(2) 遍历第(1)步中的方法调用子抽象语法树, 若该方法未存在于 *allSigSet*(已经覆盖的方法签名集合)中, 分两种情况: 若该方法所在文件已经有方法声明被覆盖, 则在 *CompleteMap* 中对应的 key 的 set 中加上一条该方法调用对应的方法签名; 若该方法所在文件没有有方法声明被覆盖, 则新加一条记录.最后将该方法调用加入 *FIFOQueue*.

算法 2. 基于广度优先遍历的方法调用补全算法.

输入: 抽象语法树 *ast*、方法签名集合 *allSignSet*、全限定性前缀集合 *prefixSet*, *FIFOQueue*, *CompleteMap*.

```
1:  function bfs(ast,allSignSet,prefixSet,FIFOQueue,CompleteMap).
2:    //遍历抽象语法树中的方法声明节点
3:    for methodDeclaration in ast.getMethodDeclaration(·)
4:      if allSigSet.contains(methodDeclaraton.getSignature(·))
5:        //遍历抽象语法树中的方法调用节点
6:       for methodCallExpr in methodDeclaration.getMethodCallExpr(·)
7:          根据 methodCallExpr 获取对应方法的签名 signature 和全限定名前缀 prefix
8:          if !allSigSet.contains(signature)   //方法调用未被测试用例覆盖
9:            //补全 MAP, 若存在 key 为 prefix, 则将 signature 加入 key 对应的方法签名集合
10:           if prefixSet.contains(signature)
11:             CompleteMap.get(prefix).add(signature)
12:             allSignSet.add(signature)
13:           //若不存在 key 为 prefix, 插入一条 key 为 prefix、值为包含 signaure 的集合
14:           else
15:             CompleteMap.put(prefix,new Set(·).add(signature)
16:             prefixSet.add(prefix)
17:           end if
18:           FIFOQueue.in(methodCallExpr)   //将新加入 MAP 的方法调用入队,以便后续分析其声明
19:         end if
20:       end for
21:     end if
22:   end for
23: return
```

图 2 中间实线框图形化展示了静态分析抽象语法树补全测试用例未覆盖方法调用的过程. 预处理部分首先将日志处理为映射形式加载进内存, 从映射可以看出, 测试用例覆盖了 4 个源码文件(prefix1–prefix3, prefix5)中的 5 个方法(main 和 method1, method2, method4, method7). 首先根据映射中的第 1 条记录生成 package1 抽象语法树, 遍历其方法声明子节点, package1 只有 main 方法被调用, 则继续遍历 main 方法声明中的方法调用节点, main 方法只调用了 method1 方法, 而 method1 存在于映射中, 所以不做任何处理. 继续遍历映射中的第 2 条记录, 生成 package2 抽象语法树, 遍历其方法声明节点, package2 中只有 method1 被调用, 继续分析方法调用子节点, method2 和 method4 都存在于映射中, 不作处理. method3 不存在于映射中(未被测试用例覆盖), 所以将其加入 *FIFOQueue* 并且添加进完整映射 *CompleteMap*. 依次类推, 直到遍历完映射. 然后将 *FIFOQueue* 依次出队, 然后依次分析每个出队的方法调用对应的抽象语法树方法声明(分析方式与前面方法声明分析方式一致), 直到 *FIFOQueue* 为空.

2.2 外部API相关代码提取

抽象语法树分析是外部 API 相关代码提取的核心部分, 如图 1 所示, 此部分主要包括第 6 步根据完整映射(算法 1 中的 *CompleteMap*)生成对应源代码文件的抽象语法树, 以及第 7 步根据完整映射分析抽象语法树中

所有方法声明节点依赖的变量和类型子节点并缓存入内存, 最后根据抽象语法树元素的依赖情况对抽象语法树进行剪枝, 最终得到迁移 API 的抽象语法树. 具体过程如算法 3 所示.

算法 3 的主要分为 3 步: (1) 通过 *CompleteMap* 生成抽象语法树, 并剪枝抽象语法树中为被调用方法的声明; (2) 分析抽象语法树中方法声明所依赖的变量和变量类型; (3) 并剪枝掉未被依赖的代码元素(变量、类型和导入语句).

算法 3. 抽象语法树剪枝算法.

输入: 补全的全限定名前缀-方法签名集合 *CompleteMap*, 源码文件根路径 *RootPath*.

输出: 剪枝后的抽象语法树集合 *astSet*.

```
1:  function pruning(CompleteMap,RootPath)
2:    fieldMap   //存储变量的映射, 键为全限定名前缀 prefix, 值为变量名集合 varSet
3:    typeMap   //存储变量类型, 键为全限定名前缀 prefix, 值为类型名集合 typeSet
4:    importSet   //存储导入语句
5:    //根据 CompleteMap 生成语法树集合 astMap, 键为全限定名前缀 prefix, 值为 AST
6:    astMap:= BuildASTs(CompleteMap)
7:    //去除未被调用方法的方法声明
8:    pruningMethods(ast,CompleteMap)
9:   for ast in astMap
10:     for methodDeclaration in ast.getMethodDeclaration(·)
11:       遍历 methodDeclaration 子语法树中变量相关的子语法树, 分析得到其变量名 var 和类型 type
12:       根据变量的类型分析导入(inport)语句, 得到其全限定名前缀 prefix
13:       fieldMap.put(prefix,var)
14:       typeMap.put(prefix,type)
15:       importSet.add(import)
16:     end for
17:     for classDeclaration in ast.getClassDeclaration(·)
18:       分析 classDeclaration 子语法树, 得到其以来的变量名 var 和类型 type
19:       根据变量的类型分析导入(inport)语句, 得到其全限定名前缀 prefix
20:       fieldMap.put(prefix,var)
21:       typeMap.put(prefix,type)
22:       importSet.add(import)
23:     end for
24:   end for
25:   //变量和类型的声明可能在其他源码文件中, 所以需要对这些变量和类型的声明重新构建 AST
26:   for k in fieldMap.keys(·) or typeMap.keys(·)
27:     if !CompleteMap.keys(·).contains(k)
28:       ast:=BuildAST(k)   //生成 k 对应源码文件的 ast 树
29:       astSet.add(ast)   //将 ast 加入 ast 树集合
30:     end if
31:   end for
32:   //分别对 field,type,import 进行剪枝
33:   pruningField(astMap,fieldMap)
34:   pruningType(astMap,fieldType)
```

35: *pruningImports*(*astMap*,*importSet*)

36: **return** *astSet*

外部 API 相关代码提取的最后一步(图 1 中的第 8 步)是基于剪枝后的抽象语法树生成对应的源代码. 具体步骤如下: (1) 根据抽象语法树包声明(*PackageDeclaration*)等信息创建与单体应用目录结构一样的目录结构; (2) 遍历剪枝后的抽象语法树, 通过抽象语法树工具在对应的源码文件中生成相关代码(参见第 4.14 节).

2.3 测试和人工调整

测试和人工调整主要是为了保证迁移后的函数代码质量和语义与迁移前外部 API 一致, 整个过程都在与单体应用一致的测试环境中进行. 在测试和人工调整前, 将生成的代码、相关配置和所有的依赖迁移至同一临时目录下, 并保持和原单体应用目录结构相对一致. 测试和人工调整的具体流程如图 3 所示. 首先编译构建源码, 若不成功, 则人工调整直至成功; 若成功, 则运行相关测试用例. 若测试用例通过, 则说明不需调整;若测试用例不通过, 则人工调整直至通过. 需要人工调整的代码主要为 Codext 工具不能完整迁移的代码, 以及原外部 API 入口参数格式和函数模板 API 入口函数参数不一致的地方.

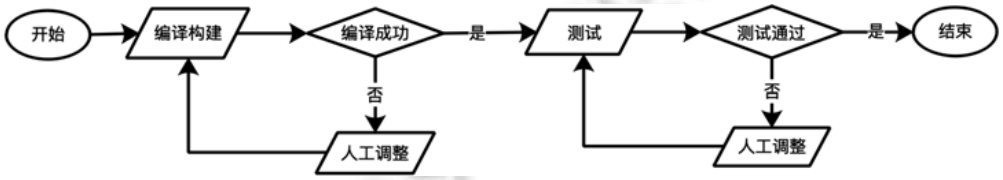


图 3 测试和人工调整流程

2.4 函数模板构建和容器镜像生成

此部分对应图 1 中的第 10 步和第 11 步, 具体步骤如图 4 所示.

(1) 根据单体应用的编程语言, 选择对应编程语言的函数模板. 将临时目录测试完毕的代码、配置迁移至函数模板的处理器目录下, 并保持目录结构相对一致, 然后调整函数启动入口代码, 即在函数入口中调用迁移的函数代码.

(2) 去除函数代码中未使用的第三方依赖. 本文实现主要针对 Java 语言, 基于依赖管理工具 Maven 的依赖分析插件 maven-dependency-plugin 对未使用的依赖进行分析, 然后根据依赖分析结果分析依赖配置文件, 去除未使用依赖的配置. 对于其他语言, 也具备相应的依赖分析工具, 比如 NodeJS 有 npm 工具、Python 有 pip 工具、Golang 有 Go module 工具.

(3) 将原单体应用的构建打包配置、运行时环境配置和函数启动配置填入函数模板的容器配置中.

(4) 根据容器配置, 首先将源码构建打包, 然后配置运行环境. 容器启动时, 将自动运行函数启动配置中的函数启动命令.

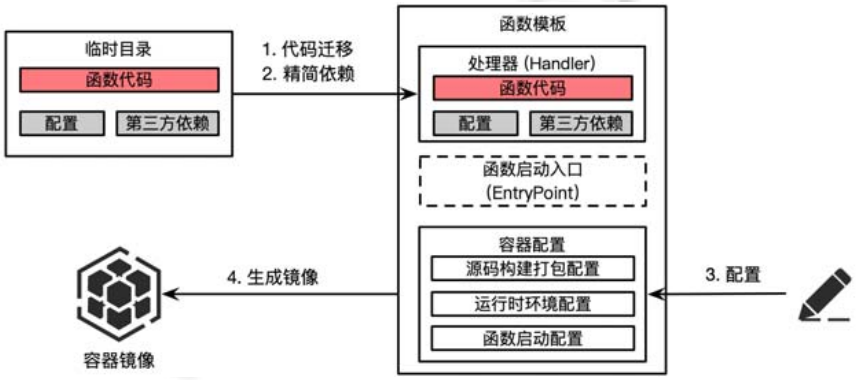


图 4 函数模板构建和容器镜像生成流程

3 高并发函数模板设计

本文使用基于 IO 多路复用的主从多线程 Reactor 模型实现了高并发的函数模板, 以减轻高流量场景下函数频繁水平扩容带来的冷启动影响. 如图 4 所示, 函数模板由处理器、函数启动入口和容器配置组成: 处理器封装了函数的业务逻辑; 函数启动入口实现了一个 HTTP 服务器, 该服务器用于处理函数请求, 该 HTTP 服务器使用了基于 IO 多路复用的主从多线程 Reactor 模型实现.

相对于基于阻塞 IO 和其他非阻塞 IO 的服务器, 主从多线程 Reactor 模型有更好的处理并发的能力. 如图 5 所示, 主从多线程 Reactor 模型将 IO 事件处理分为了两个部分: 主 Reactor 负责处理连接事件, 并将建立的连接注册到从 Reactor; 从 Reactor 负责处理主 Reactor 注册连接的各类 IO 事件, 即将这些 IO 事件分发给工作线程处理. Reactor 底层使用了 IO 多路复用机制, IO 多路复用通过 select 系统调用监听多个套接字(socket)连接, 任意数据就绪就返回, 然后通过 recvfrom 系统调用将数据从内核态拷贝至用户态, 避免了阻塞 IO 在处理请求接收事件和读写事件时带来的线程阻塞, 同时也避免了非阻塞 IO 需要在用户空间不断轮训操作系统内核数据是否准备就绪而带来的性能开销[35]. 主从 Reactor 模型可以高效利用多核 CPU 的处理能力, 大幅度提高了单个函数的并发能力.

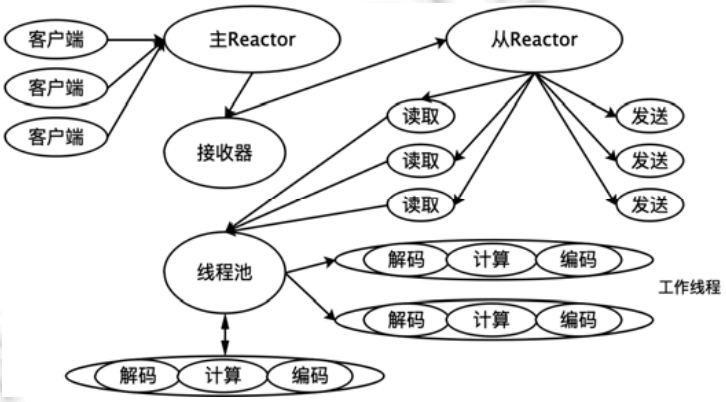


图 5 主从多线程 Reactor 模型

4 实 现

本文基于第 2 节单体应用外部 API 迁移方法实现了针对 Maven 工具管理的 Java 应用的的迁移工具 Codext, 基于第 3 节冷启动优化实现了基于 Netty 的主从多线程的 Reactor 模型的 Java 语言函数模板.

4.1 代码迁移工具Codext实现

如图 6 所示, Codext 工具主要由 5 部分组成: 方法调用搜集模块、日志预处理模块、抽象语法树分析模块、代码生成模块、依赖管理模块. 上述模块使用时, 需要提前进行配置, 配置如下.

(1) projectRoot: 单体应用源码根目录.

(2) pkgPrefix: 包名前缀(遵循 Java package 命名规范[36]). 报名前缀需要唯一标识当前项目的 package(通常为 groupId+artifactId, 遵循 Maven 规范[37]).

(3) outPath: 迁移目的地址, 即临时目录的路径.

(4) methodLogPath: 方法调用链搜集的日志路径.

(5) reservedFile: 迁移需要保留的配置文件(比如./src/main/resource, pom.xml, mybatis mapper 配置等).

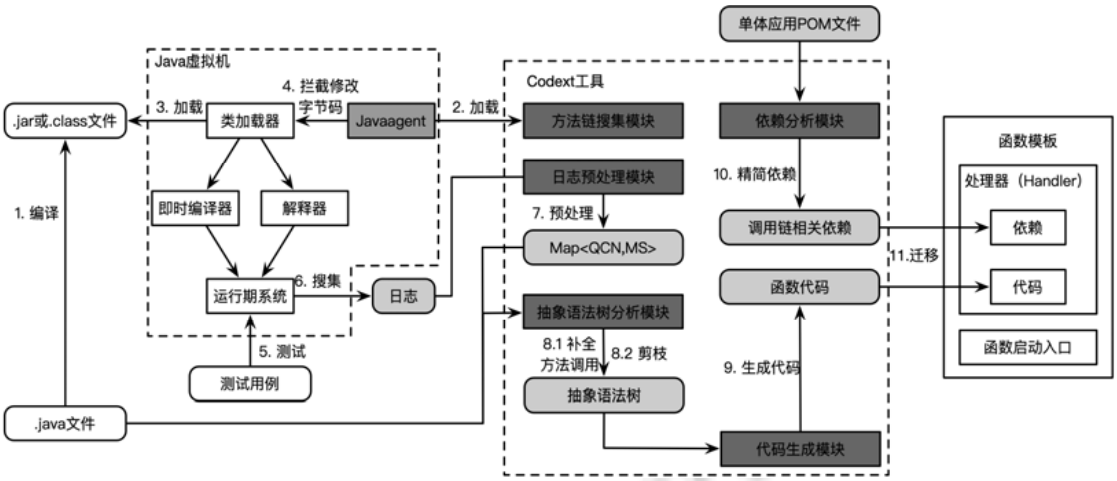


图 6　Codext 工具工作原理

4.1.1　方法调用搜集模块

方法调用链搜集模块基于 Javaagent 和 Javasist 的字节码增强技术实现了运行时无侵入式方法调用搜集, 该模块封装了修改字节码的逻辑, 该逻辑被打成 jar 包, 在应用启动时, 通过 JVM 参数-javaagent 加载. 具体流程如图 6 所示. 首先, 将.java 源码文件编译成为.jar 或者.class 可执行文件; 然后启动应用程序并运行相关测试用例. 当类被 JVM 加载后, JVM 会遍历所有加载类的字节码, 通过-javaagent 参数中配置的包名前缀过滤出当前应用的类, 通过 Javasist 工具修改字节码, 在该类的方法前加上一行日志用来搜集方法调用信息(如图 7 所示), 日志格式为“日志前缀+全类名+方法签名”, 其中, “日志前缀”为 callchain-collected, 主要用来区分其他业务日志; “日志前缀”后面的部分为“全类名+方法签名”, 用来唯一地确定一个方法调用.

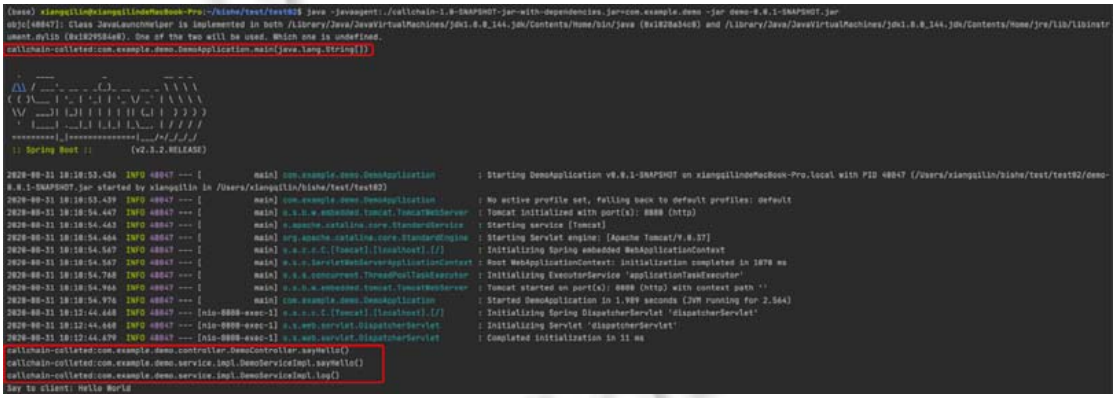


图 7　方法调用日志示例

4.1.2　日志预处理模块

日志预处理模块主要包含过滤和聚合两个过程: 首先, 通过日志前缀过滤出日志文件中与方法调用信息相关的行; 然后将其读入内存, 以 *Map*⟨*QulifiedClassName*,*Set*⟨*MethodSignature*⟩⟩的形式存储.

键 *QulifiedClassName* 为全类名, 值 *Set*⟨*MeghodSignature*⟩为该类中被调用方法的方法签名集合.

4.1.3　抽象语法树分析模块

抽象语法树分析模块基于 Javaparser 工具对抽象语法树进行分析, 主要用来补全测试用例未覆盖的方法调用以及对抽象语法树进行剪枝, 从而得到函数相关代码的抽象语法树, 主要流程如下.

1.　构建 AST: 遍历 *Map*⟨*QulifiedClassName*,*Set*⟨*MethodSignature*⟩⟩, 通过 Map 中的 *QualifiedClassName*

和配置的源码根目录定位到相应的源码文件 *projectRoot*, 并通过 JavaPaser 构建抽象语法树, 最后将生成的抽象语法树放入抽象语法树列表 *List*⟨*CompilationUnit*⟩(Javapaser 的抽象语法树实现中, *CompilationUnit* 为根节点).

2. 方法调用补全: 主要根据算法 1 分析遍历分析 *List*⟨*CompilationUnit*⟩中的每个抽象语法树中的方法声明子抽象语法树 *MethodDeclaration* 对其他方法的调用表达式(*MethodCallExpr* 和 *MethodReferenceExpr*), 从而得到补全的 *Map*⟨*QulifiedClassName*,*Set*⟨*MethodSignature*⟩⟩.
3. 抽象语法树代码元素依赖分析: 该步骤主要通过算法 2 从方法声明子语法树、类型声明子语法树两个层次对抽象语法树进行分析.
   (1) 方法声明子抽象语法树(*MethodDeclaration*)依赖分析: 搜集方法声明所依赖的代码元素, 主要包括成员变量和类型(类、接口、注解、枚举). 对类的分析主要包括表 3 中编号 1–编号 7、编号 9–编号 13、编号 16、编号 19、编号 21、编号 23 对应的元素类型. 对接口的分析主要包括表 3 中编号 1–编号 3、编号 5–编号 7、编号 10、编号 12、编号 13、编号 16、编号 23 对应的元素类型. 对注解的搜集主要分析 *NormalAnnotaionExpr*(表 3 中编号 8). 方法中对枚举和类的使用一样, 对枚举的分析包含在了对类的分析中.
   (2) 类型声明子语法树分析:搜集类型(类、接口、注解、枚举)中除了方法声明之外依赖(方法声明依赖的元素在(1)中已经分析)的变量和类型. 类的依赖分析主要包括表 3 中编号 1、编号 2、编号 5–编号 7、编号 9、编号 16、编号 17、编号 23 对应的元素类型. 接口的依赖主要包括表 3 中编号 1、编号 7、编号 8、编号 10、编号 16、编号 21、编号 23 对应的元素类型, 注解和枚举因为实现特殊, 本文实现不对其进行分析.

将所有依赖的类统一添加到 *Set*⟨*QualifiedClassName*⟩集合中, 依赖的成员变量(或常量)添加到 *Map*⟨*QualifiedClassName*,*FieldName*⟩.

表 3 Javaparser 抽象语法树分析涉及的类型和示例

编号	AST 元素类型	示例
1	ArrayType	*Object*[·][·]
2	ArrayCreationExpr	new Object[10]
3	CastExprt	(*Object*) *object*
4	CatchClause	*catch* (*Exception e*) {…}
5	ClassExpr	*Object.class*
6	ClassOrInterfaceDeclaration	*Class X*{·} *Interface X*{·}
7	ClassOrInterfaceType	*Object* *HashMap*⟨*String*,*Integer*⟩ *Java.util.Punchard*
8	NormalAnnotationExpr	*@Mapping*(…)
9	ObjectCreationExpr	*new Object*(·)
10	Parameter	*int foo*(*String str*)
11	ThrowStmt	*throw new Exception*
12	TypeExpr	*World*::*greet*
13	VariableDeclarator	*int x*=14 *List*⟨*String*⟩ *list=new ArrayList*⟨·⟩(·)
14	AssignExpr	*a*=5
15	FieldAccessExpr	*person.name*
16	FieldDeclaration	*private static Person*
17	ConstructionDeclaration	*X*{*X*(·){·}}
18	ImportDeclaration	*import com.github*…
19	InstanceOfExpr	*tool instanceof Drill*
20	MethodCallExpr	*list.size*(·)
21	MethodDeclaration	*public int foo*(·){…}
22	MethodReferenceExpr	*System.out*::*println*
23	TypeParameter	⟨*U*⟩ *U getU*(·){·} *X*⟨*Y*,*Z*⟩

4. 抽象语法树剪枝: 剪枝过程如算法 2 所示, 在 Java 实现中, 从类型语法树和根语法树两个层次进行剪枝, 最终得到剪枝后的函数相关的抽象语法树.

(1) 类型抽象语法树剪枝: 根据 *Map*⟨*QulifiedClassName*,*Set*⟨*MethodSignature*⟩⟩去除声明了但未被调用的方法声明. 根据 *Map*⟨*QualifiedClassName*,*FieldName*⟩去掉声明了但未被使用的成员变量声明. 最后保留所有的构造器和初始化块.

(2) 根抽象语法树剪枝: 根据 *Set*⟨*QualifiedClassName*⟩去除导入了但未被使用的导入声明(*ImportDeclaration*), 保留导入语句后缀为*的语句. 根据 *Set*⟨*QualifiedClassName*⟩去除未被使用的类型声明(类、枚举).

4.1.4 代码生成模块

代码生成模块将图 6 中第 8 步中剪枝后的抽象语法树转化为对应的代码, 并保证迁移前后的项目结构一致, 具体步骤如下.

(1) 根据临时目录路径配置 outPath, 生成临时目录.

(2) 根据 Maven 规范生成对应的源码目录结构和配置文件目录结构.

(3) 遍历剪枝后的每一个抽象语法树 *CompilationUnit* 节点, 根据包声明 *PackageDeclaration* 生成对应的目录结构, 并创建相应的.java 文件.

(4) 使用 Javaparser Lexical-Preserving 组件将相应的抽象语法树生成与单体应用源码中风格一致的 java 代码(如图 8 所示), 并写入步骤(3)中创建的.java 文件.

(5) 最后, 将配置项 reservedFiles 列表中的配置文件迁移至临时目录, 并保持项目结构与单体应用一致.

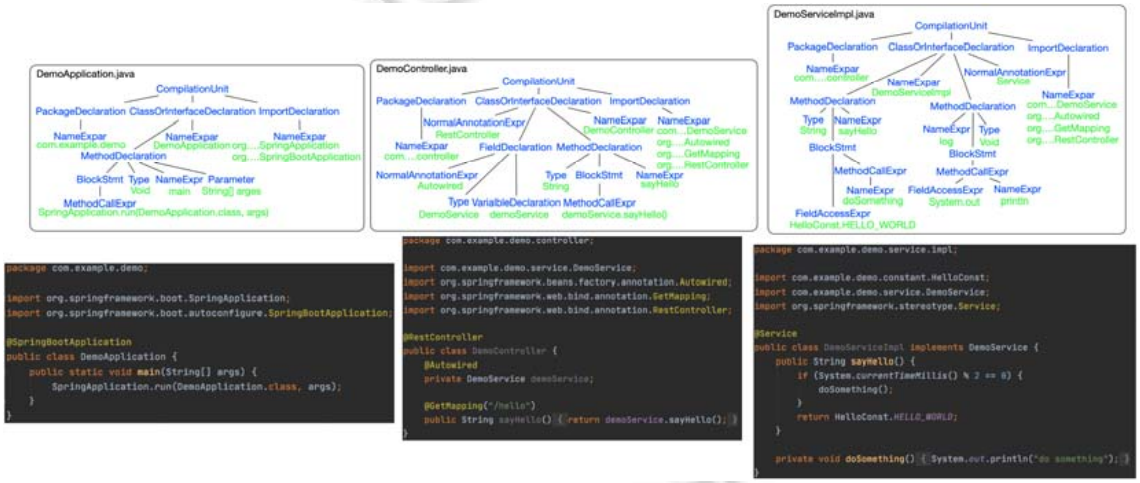


图 8 使用 Javaparser 生成的抽象语法树和对应的源码

4.1.5 依赖管理模块

依赖管理模块主要包含依赖分析和依赖精简两个部分.

- 依赖分析通过 Maven 插件 maven-dependency-plugin 分析临时目录中测试好的函数代码, 并生成相应的分析日志. 如图 9 所示, 分析日志中记录了 POM 文件中声明了但却未被使用的依赖.
- 依赖精简通过 maven-model 工具分析项目配置文件 POM, 并在 POM 文件中去除依赖分析日志中的声明了但未被使用的依赖配置.

图 9 依赖精简流程

4.2 高性能函数模板实现

根据第 3 节中基于 IO 多路复用的主从多线程 Reactor 模型函数原理，我们基于 Netty 和 OpenFaaS 模板规范[38]实现了轻量级的高性能的 Java 函数模板[39]. 图 10 详细描述了函数模板的组成和请求处理流程. 函数模板主要由 3 个部分组成，分别是:

(1) 函数业务逻辑代码: 此部分包含了所有业务代码及其相关的配置和依赖.

(2) 函数入口: 函数入口负责启动一个 HTTP 服务器，并负责请求的连接建立、解码、编码等一系列请求处理工作.

(3) HTTP 消息模型: 封装了 HTTP 的处理逻辑，以方便函数业务逻辑对 HTTP 对象的处理.

如图 10 所示，当一个客户端发送一个函数调用请求时, HTTP 服务器中 EventLoopGroup (主 Reactor)首先会监听到相关的连接事件(ServerSocketSocketChannel IO 事件)，然后通过 Acceptor 建立连接，并将相关连接的 IO 事件(SocketChannel IO 事件)注册到另外一个 EventLoopGroup (从 Reactor)上，并由该 EventLoopGroup 负责处理后续的 IO 操作. EventLoopGroup (从 Reactor)监听到读、写等 IO 事件后，会将其分配给相应的 Handler 进行处理. 一般来说，首先会对 HTTP 请求进行解码，然后入口调用 Handler 会根据请求信息设置 HTTP 消息模型，然后调用函数逻辑处理请求，最后将处理结果编码返回给客户端.

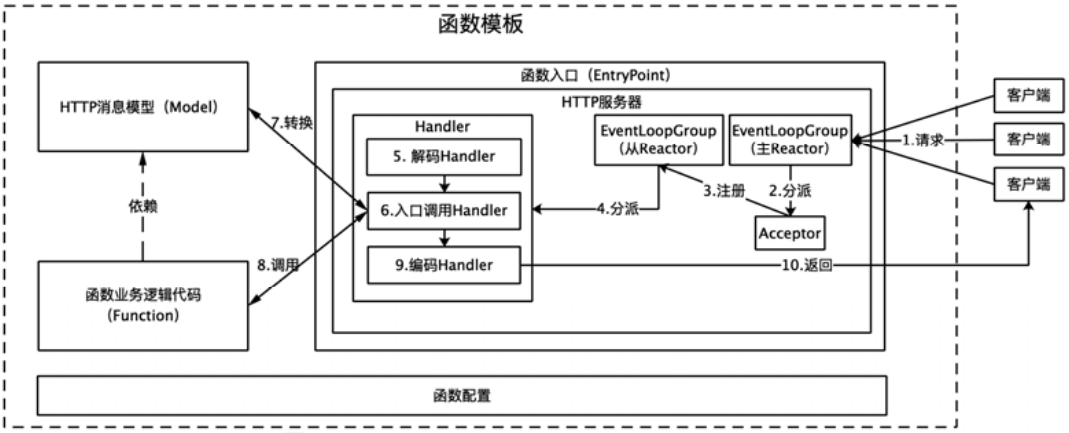


图 10 函数模板结构和运行时请求处理流程

5 实 验

我们设计了代码迁移工具 Codext 有效性实验和函数性能优化实验: 前者基于开源项目对 Codext 工具进行了功能和性能测试, 后者测试了依赖精简对冷启动的优化和大流量场景下高并发函数模板对函数实例频繁扩容带来的冷启动问题的优化.

5.1 代码迁移工具有效性实验

如表 4 所示, 本实验选取了 4 个开源项目作为实验对象, 分别是 PSS[40], StarChair[41], Newbee-mall[42]和 My-Blog[43], 这 4 个开源项目都使用了前后台分离的架构. 对于每个项目, 我们选取了 5 个 API(共 20 个)对 Codext 工具进行了功能和性能测试. 为了验证不同业务场景下 Codext 的工作情况, 这些被选取的 API 覆盖了登录、认证、增删改查、数据库访问等逻辑, 并且使用了各种开源社区热门的开发框架、ORM 框架和数据库.

本实验对比了开发者分别在对项目熟悉和陌生的情况下, 人工和使用 Codext 工具迁移指定 API 的效率; 同时定义了“类修改比”“方法修改比”和“代码行修改比”这 3 个指标来评估 Codext 工具的有效性.

表 4 实验项目信息

项目	API	技术栈	后端源码总行数
PSS	/student/login /login /choices-overview /choice /major	Framework: Springboot ORM: Spring-Data-JPA DB: MySQL Auth: JSON Web Token	1 719
StarChair	/register /applyConference /getUncheckedConference /changeApplicationStatus /getAllPassedMeetings	Framework: Springboot ORM: Spring-Data-JPA DB: H2 Auth: JSON Web Token	6 683
Newbee-mall	/login /goods/detail/{goodId} /shop-cart /shop-cart/{shopCartItemId} /personal/updateInfo	Framework: SpringBoot ORM: Mybatis DB: MySQL Auth: BasicAuth	7 519
My-Blog	/admin/login /admin/blogs/save /blog/{blogId} /admin/comments/delete /admin/blogs/update	Framework: SpringBoot ORM: Mybatis DB: MySQL Auth: BasicAuth	4 304

5.1.1 实验环境

本实验在单台 Linux 虚机上进行, 具体服务器、工具和环境信息配置见表 5 和表 6.

表 5 服务器配置信息

OS	CPU	内存	磁盘
CentOS Linux release 7.8.2003 (Core)	Intel(R) Xeon(R) CPU E5-2690 v2@3.00 GHz 8 核 8 线程	24 GB	107 GB

表 6 Java 和 Maven 信息

Java	JRE	JVM	Maven
java version “1.8.0_144”	Java(TM) SE Runtime Environment (build 1.8.0_144-b01)	Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)	Apache Maven 3.5.4

5.1.2 实验流程

我们选择了 4 名在校计算机相关专业的研究生作为实验人员, 此前他们并不熟悉将会被分配到的实验项目, 但熟悉上述实验项目的相关技术栈和 Docker 技术. 实验开始前, 首先对 4 位同学进行培训, 让他们了解 Codext 工具的使用流程, 然后基于测试项目进行练习, 直到能够熟练流畅使用 Codext 工具成功提取出单体应用的 API 代码, 并将其构建打包为 Docker 镜像. 完成培训后, 为每个实验人员分配一个开源项目, 并指定 5

个 API 完成 3 组实验: (1) 在完全陌生的条件下, 将指定的 5 个 API 人工迁移至函数模板, 并统计所用时间; (2) 给予实验人员对项目充分的熟悉时间, 然后再将指定的 5 个 API 人工迁移至函数模板, 并统计所用时间; (3) 使用 Codext 工具将指定的 5 个 API 迁移至函数模板, 并统计所用时间.

对于第 1 组和第 2 组人工迁移实验, 以及第 3 组需要人工调整代码的部分, 允许实验人员使用自己熟悉的集成开发环境和其他必要的工具. 实验完成后, 统计实验人员对 3 组实验的反馈.

5.1.3 实验结果

表 7 展示了 3 组实验的需要迁移的类和方法数量以及迁移耗时情况. 很明显, 第 1 组实验花费时间最长, 第 3 组实验花费时间最短.

表 7 3 组实验时间消耗对比

项目	API	迁移类数	迁移方法数	第 1 组耗时 (ms)	第 2 组耗时 (ms)	第 3 组耗时 (ms)
PSS	/student/login	11	33	1 975	1 063	568
	/login	9	55	1 673	1 007	839
	/choices-overview	12	33	2 573	1 090	780
	/choice	9	41	1 907	1 333	601
	/major	9	30	2 095	1 110	823
HardChair	/register	11	63	1 836	1 297	986
	/applyConference	14	54	3 791	1 764	922
	/getUncheckedConference	15	43	2 210	1 109	721
	/changeApplicationStatus	16	53	3 762	1 566	1 009
	/getAllPassedMeetings	15	44	2 427	1 221	863
Newbee-mall	/login	13	41	4 042	1 769	795
	/goods/detail/{goodId}	15	67	3 539	1 325	437
	/shop-cart	14	56	4 537	1 998	908
	/shop-cart/{shopCartItemId}	12	28	2 387	999	546
	/personal/updateInfo	14	47	2 555	1 079	653
My-Blog	/admin/login	14	34	2 075	1 264	975
	/admin/blogs/save	13	66	3 340	1 788	754
	/blog/{blogId}	16	97	4 329	2 130	638
	/admin/comments/delete	9	21	1 875	971	803
	/admin/blogs/update	13	70	3 339	1 434	764

图 11 展示了每个 API 的 3 组实验分别占实验总时长的比例, 其中, 第 1 组实验在对项目陌生情况下进行迁移, 花费时间占 3 组实验总时长的 44.57%–66.76%, 平均占比为 56.10%; 第 2 组实验在对项目完全熟悉的情况下进行迁移, 花费时间占 3 组实验总时长的 24.53%–34.70%, 平均占比为 27.68%; 第 3 组实验使用 Codext 工具的情况下进行迁移, 花费时间占 3 组实验总时长的 8.24%–23.94%, 平均占比为 16.31%. 从平均耗时占比来看: 在对项目陌生情况下, Codext 的效率是人工迁移的 3.5 倍; 在对项目完全熟悉的情况下, Codext 效率是人工迁移的 1.7 倍.

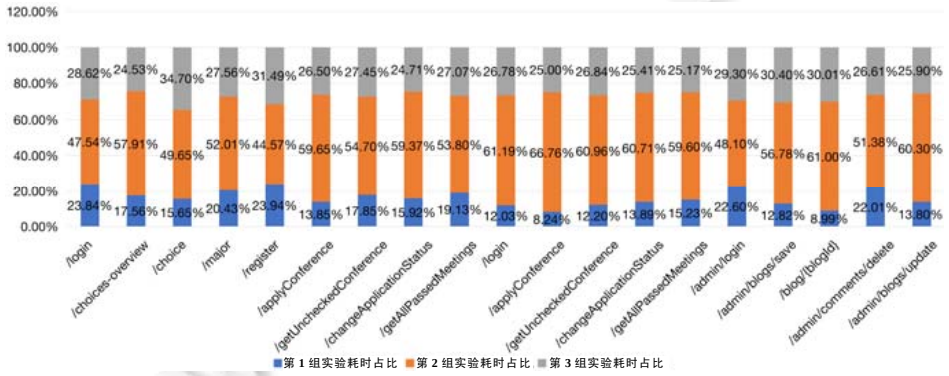


图 11 每个 API 的 3 组实验分别占实验总时长比例

根据实验人员的反馈, 他们认为, 在对项目陌生的情况下, 人工迁移和使用 Codext 工具迁移相比, 以下操作花费时间较多: (1) 寻找 API 起始代码的位置; (2) 某些隐式调用的方法难以确认, 花费时间较多; (3) 频繁的文件创建和复制粘贴. 在对项目完全熟悉的情况下, 他们认为, Codext 工具主要节约了代码文件创建和复制粘贴的时间. 同时, 他们还认为, Codext 工具可以自动化去除迁移后代码不需要的依赖, 这是人工迁移很难做到的.

我们定义了以下 3 个对代码的修改指标来判断 Codext 工具对 API 代码提取的有效性.

(1) 类修改比: 修改类的数量/最终函数代码中类的数量.

(2) 方法修改比: 修改方法数量/最终函数代码中方法的数量.

(3) 代码行修改比: 修改行数量/最终函数代码中代码行的数量.

统计规则: 若对任意一个类的非方法代码进行了任意代码行修改(包括增加和删除行), 则对修改类和修改代码行数量加 1; 若对任意一个方法进行了任意代码行修改, 则对修改类、方法和代码行数量加 1; 若对非类代码(包声明和导入语句)进行了修改, 则将修改代码行数量加 1. 所有的统计不包括配置文件和配置代码.

同时, 我们基于源码总行数和耗时来衡量 Codext 工具的性能. 实验结果见表 8, 20 个测试 API 中, 有 3 个 API 完全迁移, 不需要任何代码修改; 类修改比方面, 0–10%有 5 个, 10%–20%有 5 个, 20%–30%有 1 个, 30%–40%有 5 个, 40%–50%有 4 个, 最大修改比 55%有 1 个; 方法修改比方面, 0–10%有 10 个, 10%–20%有 4 个, 20%–30%有 2 个, 30%–40%有 2 个, 最大修改比 42%有 1 个; 代码行修改比方面, 所有 API 都低于 20%, 其中, 9 个 API 低于 5%, 8 个 API 位于 5%–10%之间. 对于某些 API, 类修改比和方法修改比较高. 这说明需要开发者修改多个类或者方法, 修改的覆盖范围较大. 但因为总体代码修改行数量较少, 仍在可接受范围之内. 比如 PSS 项目的/student/login 和 StarChair 项目的/register 的类修改比达到了 6/11 (54.5%)和 5/11 (45.4%), 但是总体代码修比例分别只有 34/599 (5.7%)和 164/910 (18.0%). 性能方面, 所有的分析都能在 5 s 以内出结果. 由此可见, Codext 工具能够有效地将单体应用中的 API 迁移至函数模板.

表 8 代码迁移有效性实验结果

项目	API	源码总行数	类修改比	方法修改比	代码行修改比	耗时(ms)
PSS	/student/login	1 719	6/11	9/33	34/599	809
	/login		3/9	23/55	30/507	822
	/choices-overview		4/12	10/33	71/551	1 011
	/choice		3/9	8/41	30/509	793
	/major		3/9	9/30	34/557	838
HardChair	/register	6 683	5/11	24/63	164/910	1 003
	/applyConference		5/14	6/54	49/927	910
	/getUncheckedConference		6/15	7/43	75/781	1 610
	/changeApplicationStatus		7/16	7/53	86/942	1 387
	/getAllPassedMeetings		6/15	7/44	75/774	1 218
Newbee-mall	/login	7 519	1/13	2/41	7/935	4 359
	/goods/detail/{goodId}		0/15	0/67	0/972	3 513
	/shop-cart		2/14	2/56	23/1,010	3 019
	/shop-cart/{shopCartItemId}		0/12	0/28	0/726	2 797
	/personal/updateInfo		0/14	0/47	0/876	2 842
My-Blog	/admin/login	4 304	2/14	6/34	59/675	4 209
	/admin/blogs/save		2/13	2/66	14/829	3 604
	/blog/{blogId}		4/16	1/97	19/1,227	4 206
	/admin/comments/delete		1/9	1/21	4/373	3 483
	/admin/blogs/update		1/13	1/70	4/831	4 904

上述部分代码未正确迁移的主要原因如下:

(1) 静态代码分析补全测试用例未覆盖的方法调用, 某些方法调用无法使用静态分析精确识别其方法签名和所属类(或对象), 所以难以定位这类方法调用对应的方法声明, 从而导致这类方法及其相关依赖无法被分析, 主要存在以下 3 种情况: (a) 方法调用为接口方法, 难以识别其动态绑定时调用的方法; (b) 方法调用为父类方法, 但是运行时绑定为子类, 未能识别子类; (c) Java8 新特性 Lambda 表达式中匿名类型对函数的调用, 不能识别匿名类型的调用.

(2) 一些框架使用了字节码增强技术导致某些类被声明, 但是在加载时被动态拦截修改字节码, 并且加载后的全类名发生了变化, 导致运行时并没有使用被声明的类, 最终导致 Codext 工具误判这部分类与当前 API 无关.

5.2 性能优化实验

性能实验在基于 Kubernetes[44]的 OpenFaaS[18](0.12.4 版本)环境下进行. Kubernetes 集群由一个主节点和 4 个从节点组成, 集群具体配置信息见表 9 和表 10. 性能优化实验使用 Jmeter[45] (5.3 版本)作为压测工具,证明了在高并发场景下, 高并发函数模板的性能和扩容情况, 主要与 OpenFaaS 官方 Java 函数模板对比.

实验所需的容器镜像已经提前构建好, 通过 docker pull 命令将其提前拉取到每个 Kubernetes 节点上, 并设置 Kubernetes 的镜像拉取策略为 IfNotPresent, 即如果本地存在某个镜像, 则不会从远程进行拉取. 这样避免了每次扩容都需要从远程拉取镜像带来的网络开销对实验的干扰.

表 9 Kubernetes 集群信息

Kubernetes 版本	主节点数量	从节点数量	网络插件	Docker 版本
v1.19.2	1	4	Calico v3.16.1	v19.03.13

表 10 Linux 服务器(Kubernetes 节点)配置信息

操作系统版本信息	处理器信息	内存	磁盘
CentOS Linux release 7.8.2003 (Core)	Intel(R) Xeon(R) CPU E5-2690 v2@3.00 GHz 8 核 8 线程	24 GB	107 GB

本实验总共分为 10 组, 每组实验使用 Jmeter 对集群进行持续 30 s 的压力测试, 并保持每组实验的并发数从 1000/s 递增至 10000/s. 函数实例在集群中配置了 CPU 核数为 50 m (0.05 个物理 CPU 核心)和 HPA 以实现自动伸缩, 伸缩范围为 1–999, 伸缩条件为 CPU 使用率超过 50%. 实验分别统计了高并发函数模板和 OpenFaaS 函数模板执行情况、返回时间和吞吐量等信息, 见表 11 和表 12.

表 11 自动伸缩条件下, 高并发函数模板在持续 30 s 的不同并发数下的性能指标

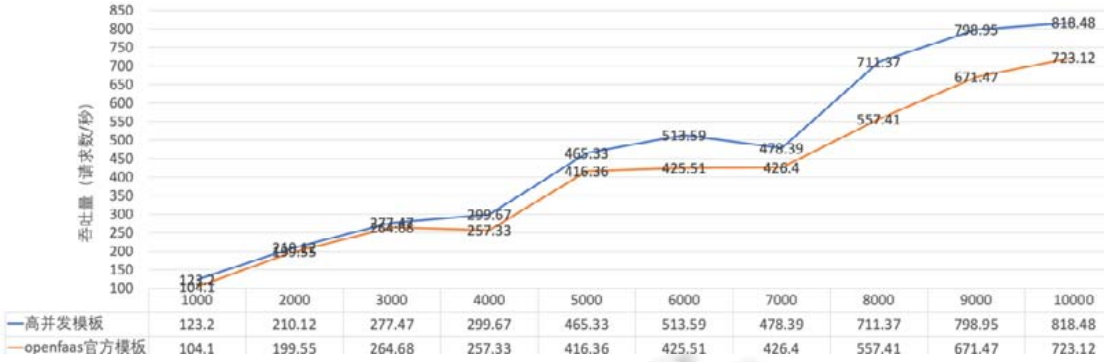
执行情况(持续 30 s)				返回时间(ms)							吞吐量
总请求数量	并发数	失败	失败率	平均	最小	最大	中位	P90	P95	P99	请求数/s
30 000	1000/s	909	3.03	6 447.06	–	139 070	2 699.00	5 400.90	7 601.00	15 366.00	123.20
60 000	2000/s	4 048	6.75	5 464.64	–	140 288	1 800.00	4 899.00	6 401.95	13 444.58	210.12
90 000	3000/s	5 846	6.50	7 797.00	–	139 515	1 505.00	5 160.90	6 199.00	18 298.00	277.47
120 000	4000/s	13 586	11.32	10 220.80	–	144 461	900.00	4 900.00	6 701.00	30 680.00	299.67
150 000	5000/s	40 098	26.73	6 647.58	–	144 100	300.00	2 698.00	3 501.00	131 019.91	465.33
180 000	6000/s	68 958	38.31	5 409.46	–	145 159	1 499.00	5 900.00	8 585.00	130 554.00	513.59
210 000	7000/s	102 572	48.84	7 144.61	–	174 080	1 900.00	5 201.00	6 399.00	13 394.63	478.39
240 000	8000/s	133 786	55.74	4 204.70	–	136 000	100.00	1 400.00	3 975.95	131 518.00	711.37
270 000	9000/s	161 478	59.81	3 864.31	–	140 288	203.00	3 803.00	9 728.95	131 023.44	798.95
300 000	10000/s	194 222	64.74	3 516.78	–	143 617	198.00	1 600.00	2 900.00	13 159.00	818.48

表 12 自动伸缩条件下, OpenFaaS 函数模板在持续 30 s 的不同并发数下的性能指标

执行情况(持续 30 s)				返回时间(ms)							吞吐量
总请求数量	并发数	失败	失败率	平均	最小	最大	中位	P90	P95	P99	请求数/s
30 000	1000/s	1 231	4.10	7 134.11	–	142 142	2 803.00	5 951.00	15 351.80	131 038.88	104.10
60 000	2000/s	4 925	8.22	6 626.76	–	132 395	1 945.00	5 103.00	7 203.00	131 067.00	199.55
90 000	3000/s	7 525	8.36	8 984.32	–	174 798	1 203.00	5 791.00	6 561.00	168 231.00	264.68
120 000	4000/s	15 896	13.25	11 351.22	–	193 376	1 056.00	4 590.00	8 655.00	128 509.00	257.33
150 000	5000/s	48 003	32.00	7 053.69	–	152 714	457.00	3 489.00	4 312.00	141 536.95	416.36
180 000	6000/s	70 481	39.16	7 746.08	–	184 319	1 580.00	6 590.00	9 560.00	169 531.54	425.51
210 000	7000/s	105 136	50.06	6 895.37	–	191 233	2 600.00	6 310.00	7 523.00	140 560.00	426.40
240 000	8000/s	144 909	60.38	8 573.21	–	187 904	200.00	1 900.00	3 621.00	161 057.00	557.41
270 000	9000/s	182 991	67.77	5 429.67	–	178 476	303.00	4 600.00	11 631.00	131 063.00	671.47
300 000	10000/s	213 091	71.03	4 823.03	–	157 963	259.00	1 893.00	3 251.00	145 312.00	723.12

图 12 展示了自动伸缩场景下的两种函数模板在不同并发数量下的吞吐量对比. 总体上看, 高并发函数模

板的吞吐量高于 OpenFaaS 模板，并且随着并发数量的不断增加，两种函数模板的吞吐量也都一直上升．这主要得益于自动伸缩横向扩展出多个实例，提高了吞吐量．



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
高并发模板	123.2	210.12	277.47	299.67	465.33	513.59	478.39	711.37	798.95	818.48
openfaas官方模板	104.1	199.55	264.68	257.33	416.36	425.51	426.4	557.41	671.47	723.12

图 12　自动伸缩场景下，两种函数模板在不同并发数量下的吞吐量对比

图 13 展示了自动伸缩场景下，两种函数模板在不同并发数量下的执行对比情况．可以看出，高并发函数模板的错误率优于 OpenFaaS 函数模板．因为高并发函数模板吞吐量的上升，在高并发场景下一个函数实例能处理更多的请求，所以能够有效缓解函数冷启动不能处理请求的情况，降低请求的错误率．

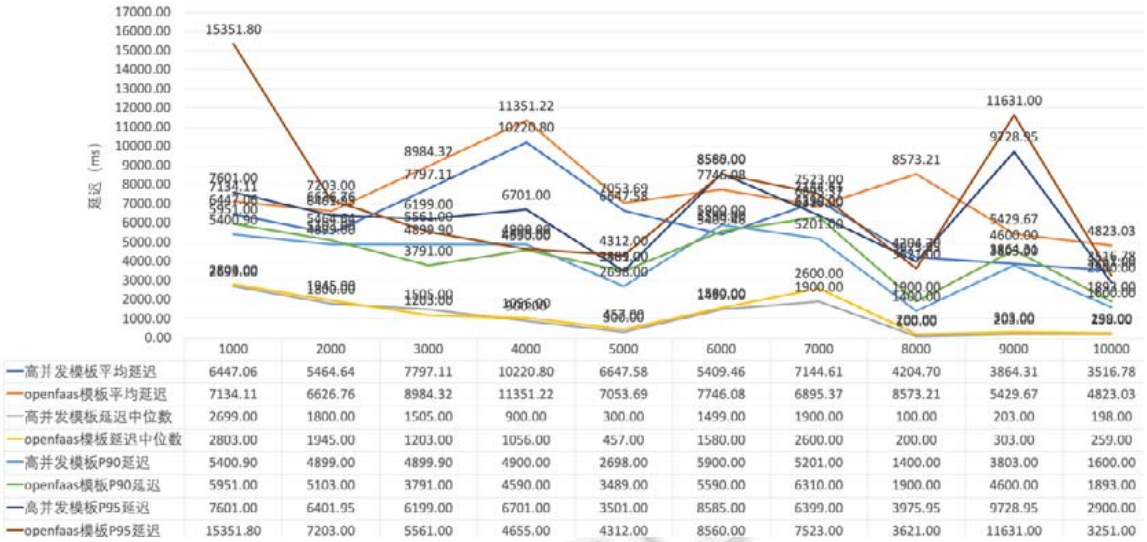
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
高并发模板请求失败数量	909.00	4048.00	5846.00	13586.00	40098.00	68958.00	102572.00	133786.00	161478.00	194222.00
openfaas模板请求失败数量	1231.00	4925.00	7525.00	15896.00	48003.00	70481.00	105136.00	144909.00	182991.00	213091.00
高并发模板请求失败比例	3.03%	6.75%	0.65%	11.32%	26.73%	38.31%	48.84%	55.74%	59.81%	64.74%
openfaas模板请求失败比例	4.10%	8.22%	8.36%	13.25%	32.00%	39.16%	50.06%	60.38%	67.77%	71.03%

图 13　自动伸缩场景下，两种函数模板在不同并发数量下的执行情况对比

图 14 展示了自动伸缩场景下，两种函数模板在不同并发下的 P90 和 P95 延迟对比．总体上，高并发函数模板的平均延迟明显低于 OpenFaaS 函数模板，说明高并发函数模板在持续的大流量场景下具有更高的吞吐量和更优秀的用户体验；两种函数模板的中位数接近重合，说明函数能够高效处理一般请求，且都明显低于平均数，说明一半的请求都能在较短时间内处理，另一半请求的延迟要远高于前一半请求．P90 和 P95 方面，高并发函数模板也明显好于 OpenFaaS 模板．说明在持续的大规模流量访问情况下，高并发函数模板都能保持较好的处理能力．

图 15 展示了自动伸缩场景下，两种函数模板在不同并发下 P99 延迟对比．高并发函数模板虽然随着并发升高存在较大的 P99 延迟波动，但是总体上，P99 延迟明显低于 OpenFaaS 函数模板，说明高并发函数模板在应对 99%的大规模流量时都具备更好的处理能力．

综上，高并发函数模板在大流量情况下总体并发量高于 OpenFaaS 函数模板，并且能够有效地避免流量峰值到来时，因为冷启动函数无法处理请求的问题．同时，请求延迟相对 OpenFaaS 函数模板有明显的降低，说明高并发函数模板能够更加高效地利用多核 CPU 的处理能力，在避免冷启动问题的同时提高函数性能．



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
高并发模板平均延迟	6447.06	5464.64	7797.11	10220.80	6647.58	5409.46	7144.61	4204.70	3864.31	3516.78
openfaas模板平均延迟	7134.11	6626.76	8984.32	11351.22	7053.69	7746.08	6895.37	8573.21	5429.67	4823.03
高并发模板延迟中位数	2699.00	1800.00	1505.00	900.00	300.00	1499.00	1900.00	100.00	203.00	198.00
openfaas模板延迟中位数	2803.00	1945.00	1203.00	1056.00	457.00	1580.00	2600.00	200.00	303.00	259.00
高并发模板P90延迟	5400.90	4899.00	4899.90	4900.00	2698.00	5900.00	5201.00	1400.00	3803.00	1600.00
openfaas模板P90延迟	5951.00	5103.00	3791.00	4590.00	3489.00	5590.00	6310.00	1900.00	4600.00	1893.00
高并发模板P95延迟	7601.00	6401.95	6199.00	6701.00	3501.00	8585.00	6399.00	3975.95	9728.95	2900.00
openfaas模板P95延迟	15351.80	7203.00	5561.00	4655.00	4312.00	8560.00	7523.00	3621.00	11631.00	3251.00

图 14　自动伸缩场景下，两种函数模板在不同并发下平均延迟、中位数、P90 和 P95 延迟对比

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
高并发模板P99延迟	15366.00	13444.58	18298.00	30680.00	131019.91	130554.00	13394.63	131518.00	131023.44	13159.00
openfaas模板P99延迟	131038.88	131067.00	168231.00	128509.00	141536.95	169531.54	140560.00	161057.00	131063.00	145312.00

图 15　自动伸缩场景下两种函数模板在不同并发下 P99 延迟对比

6　总　结

为了适应互联网时代用户量大、并发要求高等软件特性，传统的单体架构应用不得不向更高级的应用架构迁移. 单体应用向微服务架构迁移已经成了学术界和工业界一个热门的话题，但是其中还有很多难以克服的挑战，例如服务拆分、代码迁移、数据库迁移、分布式事务重构、需要完善的基础设施和高昂的运维成本等. 本文提出的基于动态和静态分析的单体应用 FaaS 改造方法规避了微服务拆分的复杂性，通过运行时轨迹和静态代码分析将单体应用的 API 迁移到 FaaS 架构，这种半自动化的方式能帮助开发者快速有效地将单体应用的 API 迁移至 FaaS 架构，特别是对于文档缺失的遗留系统. 除此之外，单体应用 API 迁移至 FaaS 架构充分利用了 Serverless 的优点，提高了应用的性能，降低了应用的维护成本. 同时，本文提出的函数模板优化方案通过实验验证，优于目前开源社区最优的 OpenFaaS 官方模板. 综上，本文提出的函数迁移和函数优化方案能够有效地将单体应用的 API 迁移至 FaaS 架构.

未来我们会将 Codext 工具在大规模范围内进行验证，提高对不同软件组件的支持(支持更多的语言、开发框架等)，将 Codext 做到更加自动化，例如自动化回归测试和接口测试，增强代码迁移的覆盖度，争取更少的依赖人工修改.

References:

[1] Rechardson C. Microservices Patterns: With Examples in Java. Manning Publications, 2018. 4–7.

[2] Ding D, Peng X, Guo XF, Zhang J, Wu YJ. Scenario-driven and bottom-up microservice decomposition for monolithic systems. Ruan Jian Xue Bao/Journal of Software, 2020,31(11): 145−164. http://www.jos.org.cn/1000-9825/6031.htm [doi: 10.13328/j.cnki.jos.006031]

[3] Roberts M. Serverless architectures. 2018. https://martinfowler.com/articles/serverless.html

[4] Fox CG, Ishakian V, Muthusamy V, Slominski A. Status of serverless computing and function-as-a-service (FaaS) in industry and research. arXiv:1708.08028, 2017. [doi: 10.13140/RG.2.2.15007.87206]

[5] DataDog. The state of serverless. 2020. https://www.datadoghq.com/state-of-serverless/

[6] Meshenberg R. Microservices at netflix scale: First principles, tradeoffs, lessons learned. https://gotocon.com/dl/goto-amsterdam-2016/slides/RuslanMeshenberg_MicroservicesAtNetflixScaleFirstPrinciplesTradeoffsLessonsLearned.pdf

[7] Zhou X, Peng X, Xie T, Sun J, Ji C, Li W, Ding D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. on Software Engineering, 2018, 47(2): 243−260. [doi: 10.1109/TSE.2018.2887384]

[8] Zhou X, Peng X, Xie T, Sun J, Ji C, Liu D, Xiang Q, He C. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). 2019. 683−694. [doi: https://doi.org/10.1145/3338906.3338961]

[9] Mazlami G, Cito J, Leitner P. Extraction of microservices from monolithic software architectures. In: Proc. of the IEEE Int'l Conf. on Web Services (ICWS). 2017. 524−531. [doi: 10.1109/ICWS.2017.61]

[10] Francesco PD, Lago P, Malavolta I. Migrating towards microservice architectures: An industrial survey. In: Proc. of the IEEE Int'l Conf. on Software Architecture (ICSA). IEEE, 2018. 29−39. [doi: 10.1109/ICSA.2018.00012]

[11] Taibi D, Lenarduzzi V, Pahl C. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 2017, 4(5): 22−32. [doi: 10.1109/MCC.2017.4250931]

[12] Fan C, Ma S. Migrating monolithic mobile application to microservice architecture: An experiment report. In: Proc. of the IEEE Int'l Conf. on AI & Mobile Services (AIMS). IEEE, 2017. 109−112. [doi: 10.1109/AIMS.2017.23]

[13] Reactor pattern. 2021. https://en.wikipedia.org/wiki/Reactor_pattern

[14] Doug Lea. Scalable IO in Java. http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf

[15] Javasist. 2021. https://www.javassist.org/

[16] Javaparser. 2021. https://javaparser.org/

[17] Netty. 2021. https://netty.io/

[18] OpenFaaS. 2021. https://www.openfaas.com/

[19] Ford N. The state of microservices maturity. 2018. https://www.oreilly.com/programming/free/the-state-of-microservices-maturity.csp

[20] Abdullah M, Iqbal W, Erradi A. Unsupervised learning approach for Web application auto-decomposition into microservices. Journal of Systems and Software, 2019, 151: 243−257. [doi: 10.1016/j.jss.2019.02.031]

[21] Gysel M, Kölbener L, Giersche W, Zimmermann O. Service cutter: A systematic approach to service decomposition. In: Proc. of the European Conf. on Service-oriented and Cloud Computing (ESOCC). Cham: Springer, 2016. 185−200. [doi: 10.1007/978-3-319-44482-6_12]

[22] Jin WX, Liu T, Zheng QH, Cui D, Cai YF. Functionality-oriented microservice extraction based on execution trace clustering. In: Proc. of the IEEE Int'l Conf. on Web Services (ICWS). IEEE, 2018. 211−218. [doi: 10.1109/ICWS.2018.00034]

[23] Lewis J, Fowler M. Microservices: A definition of this new architectural term. 2014. https://martinfowler.com/articles/microservices.html

[24] Mario V, Oscar G, Lina O, Harold C, Lorena S, Mauricio V, Rubby C, Santiago G, Carlos V, Angee Z, Mery L. Infrastructure cost comparison of running Web applications in the cloud using AWS Lambda and monolithic and microservice architectures. In: Proc. of the 16th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid). IEEE, 2016. 179−182. [doi: 10.1109/CCGrid.2016.37]

[25] Goli A, Hajihassani O, Khazaei H, Ardakanian O, Rashidi M, Dauphinee T. Migrating from monolithic to serverless: A FinTech case study. In: Proc. of the Companion of the ACM/SPEC Int'l Conf. on Performance Engineering (ICPE). 2020. 20–25. [doi: 10.1145/3375555.3384380]

[26] Kaplunovich A. ToLambda—Automatic path to serverless architectures. In: Proc. of the IEEE/ACM 3rd Int'l Workshop on Refactoring (IWoR). IEEE, 2019. 1–8. [doi: 10.1109/IWoR.2019.00008]

[27] Migrate Spring Boot Application to Function Computing. 2021 (in Chinese). https://help.aliyun.com/document_detail/160531.html?spm=a2c4g.11186623.6.829.203c3956LDQxwk

[28] Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A, Suter P. Serverless computing: Current trends and open problems. In: Proc. of the Research Advances in Cloud Computing. Singapore: Springer, 2017. 1–20. [doi: 10.1007/978-981-10-5026-8_1]

[29] Baird A, Huang G, Munns C, *et al*. Serverless architectures with AWS Lambda. https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf

[30] McGrath G, Brenner P. Serverless computing: Design implementation and performance. In: Proc. of the 37th IEEE Int'l Conf. on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017. 405–410. [doi: 10.1109/ICDCSW.2017.36]

[31] Cold start. 2021. https://en.wikipedia.org/wiki/Cold_start_(computing)

[32] Silva P, Fireman D, Pereira TE. Prebaking functions to warm the serverless cold start. In: Proc. of the 21st Int'l Middleware Conf. 2020. 1–13. [doi: 10.1145/3423211.3425682]

[33] Lambda warmer. 2021. https://github.com/jeremydaly/lambda-warmer

[34] Auto-Scaling. 2021. https://docs.openfaas.com/architecture/autoscaling/

[35] Stevens R. Unix Network Programming. 3rd ed., Prentice Hall, 1990. 202–237.

[36] Unique package names. 2021. https://docs.oracle.com/javase/specs/jls/se6/html/packages.html#7.7

[37] Guide to naming conventions on groupId, artifactId, and version. 2021. https://maven.apache.org/guides/mini/guide-naming-conventions.html

[38] OpenFaaS templates. 2021. https://github.com/openfaas/templates

[39] High-concurrency template. 2021. https://github.com/kylinxiang70/openfaas-template

[40] PSS. 2021. https://github.com/LeBW/preference-selection-system

[41] HardChair. 2021. https://github.com/FudanSELab/StarChair

[42] Newbee-Mall. 2021. https://github.com/newbee-ltd/newbee-mall

[43] My-Blog. 2021. https://github.com/ZHENFENG13/My-Blog

[44] Kubernetes. 2021. https://kubernetes.io/

[45] Jmeter. 2021. https://jmeter.apache.org/

附中文参考文献:

[27] 迁移到 SpringBoot 函数计算. 2021. https://help.aliyun.com/document_detail/160531.html?spm=a2c4g.11186623.6.829.203c3956LDQxwk

向麒麟(1994－), 男, 硕士, 主要研究领域为云原生与智能化运维.

赤坂居纱美(1998－), 女, 硕士生, 主要研究领域为云原生与智能化运维.

彭鑫(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能化软件开发, 云原生与智能化运维, 泛在计算软件系统.

李博文(1997－), 男, 硕士, 主要研究领域为云原生与智能化运维.