密级：　　保密期限：

北京郵電大學

硕士学位论文

题目：基于服务容器的
服务语义重构

学　　号：2015141002

姓　　名：贺路路

专　　业：计算机技术

导　　师：章洋

学　　院：网络技术研究院

2018 年 1 月 31 日

Confidentiality level:

Confidentiality period:

Beijing University of Posts and Telecommunications Master's Thesis

Title：Service Semantic Reconstruction Based On Service Container

No. : 2015141002

Name: He Lulu

Major: Computer Technology

Tutor: Zhang Yang

College: Institute of Network Technology

2018.01.31

独创性（或创新性）声明

本人声明所呈交的论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：_____ 日期：_____

关于论文使用授权的说明

本人完全了解并同意北京邮电大学有关保留、使用学位论文的规定，即：北京邮电大学拥有以下关于学位论文的无偿使用权，具体包括：学校有权保留并向国家有关部门或机构送交学位论文，有权允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，有权允许采用影印、缩印或其它复制手段保存、汇编学位论文，将学位论文的全部或部分内容编入有关数据库进行检索。（保密的学位论文在解密后遵守此规定）

本人签名：_____ 日期：_____

导师签名：_____ 日期：_____

基于服务容器的服务语义重构

摘　要

物联网服务的迅速发展要求必须保障物联网服务的安全性。在虚拟化平台中运行物联网服务，同时对物联网服务进行实时监控，对其运行时所产生的各类事件进行运行时验证，能够有效保障物联网服务的安全。其中，利用内存自省技术，从物联网服务的内存中取出数据，重构出服务语义，根据服务语义构造内存事件。物联网服务部署于 Java 服务容器中，因此要基于 Java 服务容器来进行语义重构。

基于已有的虚拟机内存自省技术，结合虚拟化和物联网服务特点，改进 Java 内存分析技术，本文提出了基于服务容器的服务语义重构系统。该系统基于虚拟机内存自省技术获取内存数据，根据 JVM 结构分析内存数据，重构出 Java 服务的函数调用和函数参数，构造内存事件，提供给物联网运行时验证系统。

本文从三个方面对基于服务容器的服务语义重构系统进行研究。首先，内存获取通过虚拟机内存自省技术，为内存分析提供了内存数据，包括共享库、子线程和栈内存。其次，内存分析根据 JVM 内存结构和共享库重构 JVM 数据结构。以 JVM 数据结构为基础，从 Java 函数栈中获取内存数据。根据解释运行的栈帧结构，分析内存数据重构出解释运行的函数和函数参数。根据编译运行的栈帧结构和编译生成的本地代码，分析内存数据重构出编译运行的函数和函数参数。根据前后栈帧的运行方式，得到下一栈帧的内存数据和函数调用。根据 Java 服务容器的结构，得到调用服务的参数。最后，根据函数调用、函数参数和服务参数构造内存事件并提供给物联网验证系统。

本文从需求分析、系统设计和系统实现出发，详细阐述了服务语义重构系统的功能需求、设计原理和系统实现。通过系统测试从功能和性能两方面验证了系统的可用性。

关键词 物联网服务 JVM 内存分析 解释运行 编译运行

SERVICE SEMANTIC RECONSTRUCTION BASED ON SERVICE CONTAINER

ABSTRACT

The development of Internet of things service requires the security of the Internet of things service. In the virtualization platform, we can run the Internet of things services, and real-time monitor the Internet of things services. We can verify all kinds of events generated during the runtime operation, which can guarantee the security of the Internet of things. Among them, the memory data fetched from service by virtual machine introspection reconstructs the service semantics. According to the service semantics, we can construct memory event. The Internet of things service is deployed in the Java service container, so the semantic reconstruction is based on the Java service container.

Based on the existing virtual machine introspection technology, combined with the characteristics of virtualization and Internet of things services, and improving the Java memory analysis technology, this paper proposes a service semantic reconstruction system based on service container. The system obtains memory data based on virtual machine introspection technology, analyzes memory data based on JVM structure, reconstructs function calls and function parameters of Java service, constructs memory events, and provides memory events to runtime verify system of Internet of things.

In this paper, service semantic reconstruction system based on service container is studied from three aspects. First, memory acquisition provides memory data for memory analysis, including shared libraries, child threads and stack memory, through virtual machine introspection technology. Secondly, the memory analysis reconstructs the JVM data structure based on the JVM memory structure and the shared library. Based on the JVM data structure, the memory data is obtained from the Java function stack. According to the stack frame structure of the interpreter running, the

analysis of memory data reconstructs the function and function parameters of the interpreter running. Based on the stack frame structure of the compiled run and the local code generated by compiler, the function and function parameters of the compiler running are reconstructed by analyzing the memory data. According to the running mode of the front and back frame, the memory data and function call of the next frame are obtained. According to the structure of the Java service container, the parameters of the invoke service are obtained. Finally, based on the function call, function parameters and service parameters, the memory events are constructed and provided to runtime verify system of Internet of things.

Based on the requirements analysis, system design and implementation, this paper introduces the functional requirements, design principles and system implementation of service semantic reconstruction system. System testing verifies the availability of the system from two aspects of function and performance.

KEY WORDS Internet of things service JVM memory analysis interpreter running compiler running

目录

第一章 绪论

1.1 研究背景

随着万物互联的物联网迅速发展，运行在物联网上的服务越来越多[1]。各个服务之间相互组合，相互调用，形成了非常庞大的物联网服务系统。任意一个服务都会影响与之相关的其他服务。因此，这些服务系统的安全性、可靠性和可用性必须得到保证。物联网服务可以使用各种技术进行开发，其中有为数不少的物联网服务是基于 Java 开发。Java 语言是现在世界上最流行的编程语言之一，已有的开发库等资源十分丰富。使用 Java 语言编写服务应用，开发便捷，功能完备，具有很好的可拓展性和可移植性，因此 Java 语言非常适合开发物联网服务。Java 服务通常部署在 Java 服务容器中。Java 服务容器是实现了 Java 容器规范（如 Java EE）的部署环境。Java 服务容器屏蔽了服务器平台的复杂性，使得服务可以方便快捷的部署运行和管理。Java 服务容器运行于 JVM 之中。本文所研究的物联网服务是部署在 Java 服务容器运行于 JVM 中的 Java 服务。

物联网服务大都应用在工业控制、环境监测、智能交通等领域[2]。因此，物联网服务的应用环境要求物联网服务的安全性和可靠性必须得到保证，以支持物联网服务长时间的提供可靠服务。为了保证物联网服务的运行安全，采用虚拟化技术，在虚拟化平台中运行物联网服务，同时对物联网服务进行实时监控，对其运行时所产生的各类事件进行运行时验证。而要对运行状态进行验证，不仅需要网络事件、串口事件，更需要具有高可信度的内存事件。服务语义重构直接从物联网服务的内存中取得数据，重构出内存中的栈帧，还原服务运行时的函数状态和服务语义，为运行时验证提供内存事件。

因此，要研究物联网 Java 服务容器的服务语义，对服务的内存语义重构，实现物联网运行时验证系统，关键是内存数据获取和内存数据分析。目前，对虚拟化平台的内存获取比较好的方案是 LibVMI 虚拟机内存自省库。该库提供了一系列的内存自省接口，对多种虚拟化平台和操作系统都有着良好的支持，可以方便的获取到虚拟化平台中运行的操作系统以及相关服务程序的内存数据。同时，本文所研究的物联网服务是 Java 语言编写的，物联网服务的内存数据是基于 JVM 内存结构的，无法直接通过内存的二进制数据来获取服务的语义，必须根据 JVM 的内存结构对内存数据进行分析，重构 JVM 的内存结构和内存数据在 JVM 中的语义，才能获取到服务的真实语义。在 Java 相关的内存分析工具中，JDK 包含的基于操作系统的调试接口进行分析的 Java SA（Java Serviceability Agent），可以获取运行在 JVM 中的 Java 程序的内存数据，对数据进行分析，然

后还原出运行的栈帧。但是，Java SA 只能对在同一操作系统中运行的 Java 程序进行内存获取和分析，无法对虚拟化平台中的 Java 服务进行内存获取和分析。因此，本文在 LibVMI 和 Java SA 的研究基础上，针对虚拟化平台和物联网 Java 服务的特性，实现了基于服务容器的服务语义重构系统，为物联网运行时验证系统提供函数轨迹和内存事件。

1.2 研究内容

本文所研究实现的系统是基于服务容器的服务语义重构系统。本系统的目的是通过读取虚拟化平台中运行的物联网 Java 服务内存数据，实时还原出 Java 服务的函数栈帧，获取函数调用轨迹和函数参数，得到调用服务的参数，为物联网运行时验证系统提供内存事件支持。

通过对虚拟化平台内存自省技术和 Java 内存分析的相关研究，对虚拟化平台的内存获取技术 LibVMI 只能获取到内存的二进制数据，而 Java 相关内存分析软件 Java SA 无法读取虚拟化平台的内存信息，从而无法进行内存分析，并且在分析过程中会对 Java 服务造成影响。因此，本系统将利用基于虚拟化平台的内存获取技术和 Java 内存分析技术，通过对两种技术的结合和拓展，实现在虚拟化平台下的基于服务容器的服务语义重构。

基于以上分析内容，本文的主要研究内容和工作分为以下三个部分：

1.2.1 基于虚拟化平台的内存获取

这部分主要研究如何通过基于虚拟化平台的内存获取技术来获取 Java 内存分析所需要的内存数据。Java 内存分析需要运行 Java 服务的 Java 虚拟机的多种内存数据来重构内存，包括运行时 JVM 内存，共享库（shared libraries），程序子线程等相关内存数据。

1.2.2 Java 内存分析

这部分主要研究如何根据已经获取的相关内存数据来重构出程序实际的栈帧状态，包括函数调用顺序，函数名称，函数参数和本地变量等信息，根据这些信息重构出调用物联网服务的参数语义。物联网 Java 服务是运行于 JVM 之中，要分析 Java 服务语义，就要根据 JVM 的内存结构和共享库数据结构，通过分析内存数据还原相应的 JVM 内存结构对应的类型、类型包含的成员、整型常量和长整型常量。为了提供代码的执行效率，JVM 在运行时有解释运行和编译运行两种代码执行方式。对于这两种不同运行方式的内存栈帧结构，有相应的基于解释运行的内存分析方法和基于编译运行的内存分析方法。在分析解释运行的栈帧时，根据 JVM 内存结构找到线程栈的起始地址，读取栈内存，根据解释型栈帧

的内存布局原理和栈帧指针，分析函数栈帧，得到函数名和本地变量。在分析编译运行的栈帧时，先得到线程栈的起始地址，读取栈内存，根据编译型栈帧的内存布局和操作数栈指针，分析函数栈帧，根据编译优化之后代码区的局部变量布局，得到函数名和本地变量。根据获取的栈内存，可以分析出在 Java 栈中的所有栈帧，就可以还原函数调用过程。根据 Java 服务容器的结构，利用函数调用过程和本地变量得到调用服务的参数。服务参数就是物联网 Java 服务的语义。

1.2.3 运行时验证内存事件

这部分主要研究如何根据已经获取的物联网服务语义来构造运行时验证所需要的内存事件。物联网运行时验证系统通过对网口，串口和内存等事件来源所产生的事件，进行运行时验证，来保障物联网服务的安全性和可靠性。构造运行时验证所需要的内存事件，主要包括函数调用和函数参数。通过配置文件，配置需要监控的物联网服务相关函数以及参数类型。根据相应配置，获取物联网服务的内存数据，构造包含相应函数和服务参数的字符串事件，并实时发送给物联网运行时验证系统。

最后，根据以上的研究开发了基于服务容器的服务语义重构系统。系统包含了内存获取模块、内存分析模块、内存事件模块、界面配置模块。内存获取模块实现内存获取功能。内存分析模块实现服务语义重构功能。内存事件模块实现事件构造和事件发送功能。界面配置模块实现系统界面以及配置功能。

1.3 论文组织

基于以上研究背景和研究内容，本论文的组织结构如下：

第一章为绪论，主要介绍了本文的研究背景，明确了研究的主要内容和论文组织结构；

第二章是相关技术概述，主要介绍了本系统的相关技术基础，包括基于虚拟机平台的内存自省技术，Java 以及 JVM 虚拟机相关技术；

第三章是系统的需求分析，包括内存获取、内存分析、内存事件和界面系统的功能需求和高性能、高可靠性的性能需求，为系统的设计与实现提供了目标；

第四章是系统设计，详细描述了与需求相对应的内存获取、内存分析、内存事件和界面系统等功能模块的原理、设计和流程；

第五章是系统实现，对第四章设计的模块和工作流程的实现详细说明；

第六章讨论系统的测试和验证，主要包括测试环境的描述、测试服务的部署以及测试用例和结果分析。

第七章是总结与展望，对本系统的研究设计和实现工作进行分析总结，并对

未来系统的后续研究和改进提供意见。

1.4 本章总结

本章概述了基于服务容器的服务语义重构系统的研究背景，确立了本文研究的主要内容和论述主题。同时，还介绍了本文的组织结构，使读者能够清楚更清楚的了解整个系统的设计与实现过程。

第二章 相关技术概述

2.1 内存取证

内存取证是对操作系统内存进行取证技术。内存取证作为计算机取证科学的重要分支，是指从计算机的物理内存和页面交换文件中查找、提取和分析易失性证据，是对基于外存文件系统取证的重要补充，是对抗网络攻击或网络犯罪的有力武器。当操作系统处于活动状态时，物理内存中保存着关于系统运行时状态的关键信息，比如解密密钥、口令、打开文件信息、进程信息、网络连接、系统状态信息等。进行内存取证，就是通过获取物理内存和页面交换文件的拷贝。对拷贝进行内存数据分析，重构出原先操作系统的状态信息。在本系统所使用的内存取证技术是针对虚拟化平台的内存取证，最重要的特点是实时性[3-7]。

2.1.1 虚拟机内存自省

LibVMI 是一个 C 库，它提供了对正在运行中的底层虚拟机的运行细节进行监控的功能。这种监视的功能是由获取内存和读取 CPU 寄存器来完成的。这种方式被称作虚拟机自省(virtual machine introspection)[8]。LibVMI 的工作原理如下：

1) VMI 应用请求查看系统内核符号表；
2) LibVMI 通过/boot/System.map 查找系统内核符号的虚拟地址；
3) 找到虚拟地址所对应的内核页目录和对应的页表；
4) 通过页表找到相应的数据；
5) 返回数据到 LibVMI；
6) LibVMI 返回数据给请求的 VMI 应用。

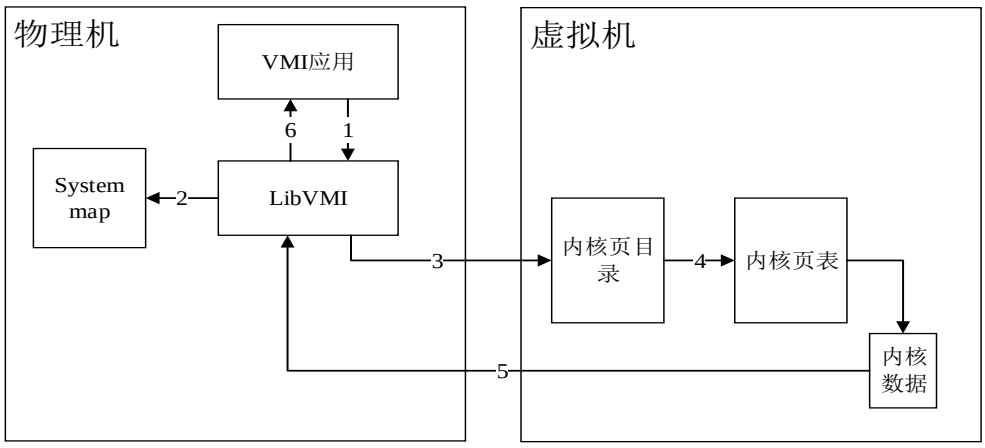

图 2-1 LibVMI 工作原理

Volatility Framework 是 Volatility Foudation 开源的内存分析框架[9]。它基于 Python 语言编写，方便进行模块集成，可以运行于 Windows，Linux，Mac 等任

何可以运行 Python 的操作系统。可扩展的和脚本化的 API 可以方便进行功能扩展，可以使用 Volatility 来构建定制的 Web 界面或 GUI，驱动恶意软件沙箱，执行虚拟机内省，或者以自动化方式开发内核内存。同时，Volatility 高效的算法也在进行内存分析时提供了很好的性能。

LibVIM 作为一个共享库文件，为许多编程语言都开发了接口进行支持。因此，LibVMI 提供了 pyvmi 作为 Python 编程和支持 Volatility 框架的适配器。在 Volatility 分析内存时，进程地址空间是必不可少的。通过进程地址空间，可以读取进程的内存数据。pyvmi 作为帮助 Volatility 实现实时内存自省的插件，为 Volatility 实现了 PyVmiAddressSpace 类用于表示虚拟机的进程地址空间。主要提供了 read()方法来读取进程的内存数据[10]。

在 Linux 系统中，进程是链表结构进行存储的[11]。通过 Volatility 的进程描述符 task_struct 结构的 tasks，可以得到进程描述符的双向链表。初始 task 是 init_task，它是 Linux 系统的第一个进程，之后的进程都是它的子进程。当新建进程时，就把新的进程加入到已有的进程链表当中，所有的进程都被链入链表。当已知要监控的进程的 pid，通过遍历 tasks，判断 task.pid 是否与 pid 相同，如果相同，当前 task 就是要监控的进程的进程描述，如果不同，就继续查找[12,13]。

在得到进程的进程描述符之后，就可以根据进程描述符得到进程地址空间。由于在获取和遍历进程时，Volatility 加载的都是 pyvmi 插件，因此进程地址空间就是 PyVmiAddressSpace 类的实例。通过调用 PyVmiAddressSpace 类的 read()方法，read()方法调用 LibVMI 库函数来获取内存数据。

2.1.2 共享库与符号表

库是一种可复用的可执行代码，提供了许多基础的共同的函数。共享库是一种动态加载的库。在程序加载阶段，程序将所需要的外部函数按照路径查找共享库位置，然后加载到内存中。因此，库地址包含在进程内存之中[14]。

在 Linux 系统中，/proc/<pid>/maps 文件中保存了正在运行的进程的库的相关信息，包括内存虚拟地址，执行权限，进程地址中偏移量，设备号，节点号以及文件路径。通过获取/proc/<pid>/maps，可以获得共享库在内存中地址。

共享库也是一种 ELF 文件。共享库中包含符号表，符号表用于表示函数名称，变量名称，通过符号表可以在链接和调试，可以获得函数的相关信息，从而链接正确的函数，调试时显示正确的函数名称和变量名称。通过 ELF 文件，可以得到符号相对于共享库的偏移量。在 Linux 中，Readelf 命令可以读取 ELF 文件，得到共享库的符号表相关信息[15]。

2.2 Java 相关技术

2.2.1 JVM

Java 语言是当前十分流行的面向对象编程语言之一，具有可移植性和高并发性等优点。Java 语言编写的代码，首先要经过 javac 编译进行编译成字节码指令的.class 文件。字节码指令是一种类似汇编语言的指令的指令集。JVM 是执行字节码的虚拟硬件，通过对不同操作系统平台编写相应的 JVM，字节码就可以在任何 JVM 上执行。JVM 在执行字节码的过程中会将其管理的内存划分为若干个不同的数据区域，包括程序计数器，Java 虚拟机栈，Java 堆，方法区等。其中，程序计数器，Java 虚拟机栈是线程私有的，而 Java 堆，方法区是线程共有的[6,8,16]。

Java 虚拟机栈是线程私有的，其生命周期与线程相同。因此，一旦线程创建，那么其虚拟机栈也会随之创建，并且不再改变，直到线程销毁。虚拟机栈是 Java 执行方法的内存模型。每一个方法在被执行时都会创建一个栈帧，栈帧中存储了局部变量表，操作数栈，动态链接，方法出口等信息。局部变量存放了函数的本地变量，包括各种基本数据类型（byte、char、short、int、long、float、double、boolean）、对象引用（reference 类型）以及 returnAddress 类型（指向字节码指令的地址）。在 64 位的 Java 虚拟机中，一个局部变量空间占 64 位，8 字节。每个本地变量都占用一个局部变量空间。当数据类型小于 64 位的时，头部使用 0 进行填充。局部变量表在编译期分配空间，因此，方法运行时，局部变量表的位置以及其中存储的变量地址就不会改变。在进行服务语义重构时，就要分析局部变量表的位置以及其中存储的变量[6,17]。

方法区是各线程共享的内存区域，用于存储已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译产生的代码等数据。因为大部分信息一经加载就不会再发生改变，所以方法区又称为永久代。在进行服务语义重构时，方法区的类信息中包含的方法信息以及即时编译器产生的代码信息，用于分析确定函数，查找代码编译优化信息确定函数参数偏移等[6]。

2.2.2 解释运行与编译运行

一般情况下，JVM 都包含解释器和即时编译器。少数厂商实现的 JVM 可能不包含解释器，也可能不包含编译。本文所研究的 Sun 公司开发的 Hotspot 虚拟机包括了解释器和即时编译器。所以，JVM 在执行代码时，可以选择解释执行或者编译执行。

在解释执行过程中，JVM 将字节码指令压入到 Java 虚拟机栈中，然后通过栈操作，执行字节码指令。在这样的执行方式下，大量的指令被用于压栈，出栈和取地址等操作，浪费了大量的 CPU 和执行时间。所以，Java 解释执行的效率不高。在解释执行的基础上，Java 又发展出了即时编译、编译执行的模式。在字节码执行的过程中，通过监测函数调用情况，对调用较多的函数和循环较多的结

构体，以函数为单位进行即时编译。当代码编译成本地代码之后，在使用栈上替换技术，将解释栈栈帧替换为新的编译型栈帧。当编译优化的条件不存在时，又会退化成解释型栈帧[6,18]。

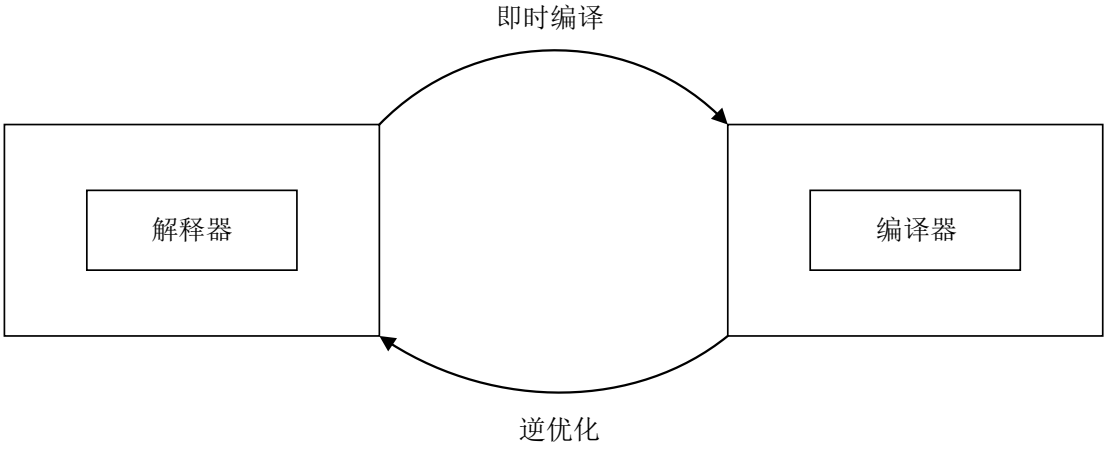

图 2-2 解释运行与编译运行的转换

在 Java 虚拟机运行时，默认采用混合模式，即解释器和编译器同时工作的模式进行。配置 Java 虚拟机的参数，可以让 Java 虚拟机采用解释模式，即只解释执行字节码。也可以让 Java 虚拟机采用变异模式，即优先使用编译执行，但是解释器会在无法编译的情况下执行代码[6,19]。

即时编译器在进行编译时会进行很多优化的手段，包括函数内联、公共子表达式消除、数组边界检查消除、逃逸分析等。当函数进行内联时，函数会共用相同的局部变量表，相同参数的地址也会相同。

2.2.3 SA-JDI

在 JDK 相关内存分析工具中，SA-JDI 是针对 Java 虚拟机结构基于操作系统调试接口的分析工具。SA-JDI 位于 JDK 的 lib/sa-jdi.jar 中。它可以对运行中的 JVM 进程进行分析，重构出 Java 虚拟机结构，还原出 JVM 的线程，函数，静态变量，实例变量，基本数据类型变量等相关信息。其原理是基于操作系统的调试接口，使进程进入调试暂停状态，获取进程内存信息和共享库信息，重构 JVM 结构，从而获取到线程，函数以及其他相关信息。在 Linux 平台中，SA-JDI 基于 ptrace 库实现。通过调用 ptrace_attach()函数，附着到 JVM 使 JVM 进入暂停状态，然后调用 ptrace()函数实现内存的读取[6,20]。

SA-JDI 对 JVM 进程的分析十分完善且便捷，但由于其所基于的分析原理要求 JVM 必须出于暂停状态，因而只能用于在程序出现问题时的排查分析，而不能用于程序的实时监控。

2.3 本章总结

本章详细的介绍了本系统相关的技术和开发基础，为后续设计和实现工作提供了相关技术支持。基于虚拟化技术的虚拟化平台是本系统的开发环境，基于虚拟化技术的内存取证技术是本系统所借助的技术手段。JVM 相关知识是物联网服务语义重构的理论基础。这些相关技术是之后系统设计和实现阶段必不可少的基础。

第三章 需求分析

本文所实现的基于服务容器的服务语义重构系统的目的是通过读取虚拟化平台中运行的物联网 Java 服务内存，还原出 Java 服务的函数栈帧，分析函数调用轨迹和函数调用参数，构造内存事件为物联网运行时验证系统提供支持。本系统通过界面操作和显示不仅可以配置要监控的服务，而且可以实时查看函数调用和内存事件相关信息。根据本系统所要实现的功能和物联网运行环境对本系统的要求，确定以下功能性需求和非功能性需求。

3.1 功能性需求

为了实现本系统的目的，通过将所有需要的功能进行仔细的划分，完成用户友好的服务语义重构系统。服务语义重构系统包括了内存获取，内存分析和内存事件功能。同时，还要有一个界面系统。整个系统的需求用例图如图 3-1 所示。

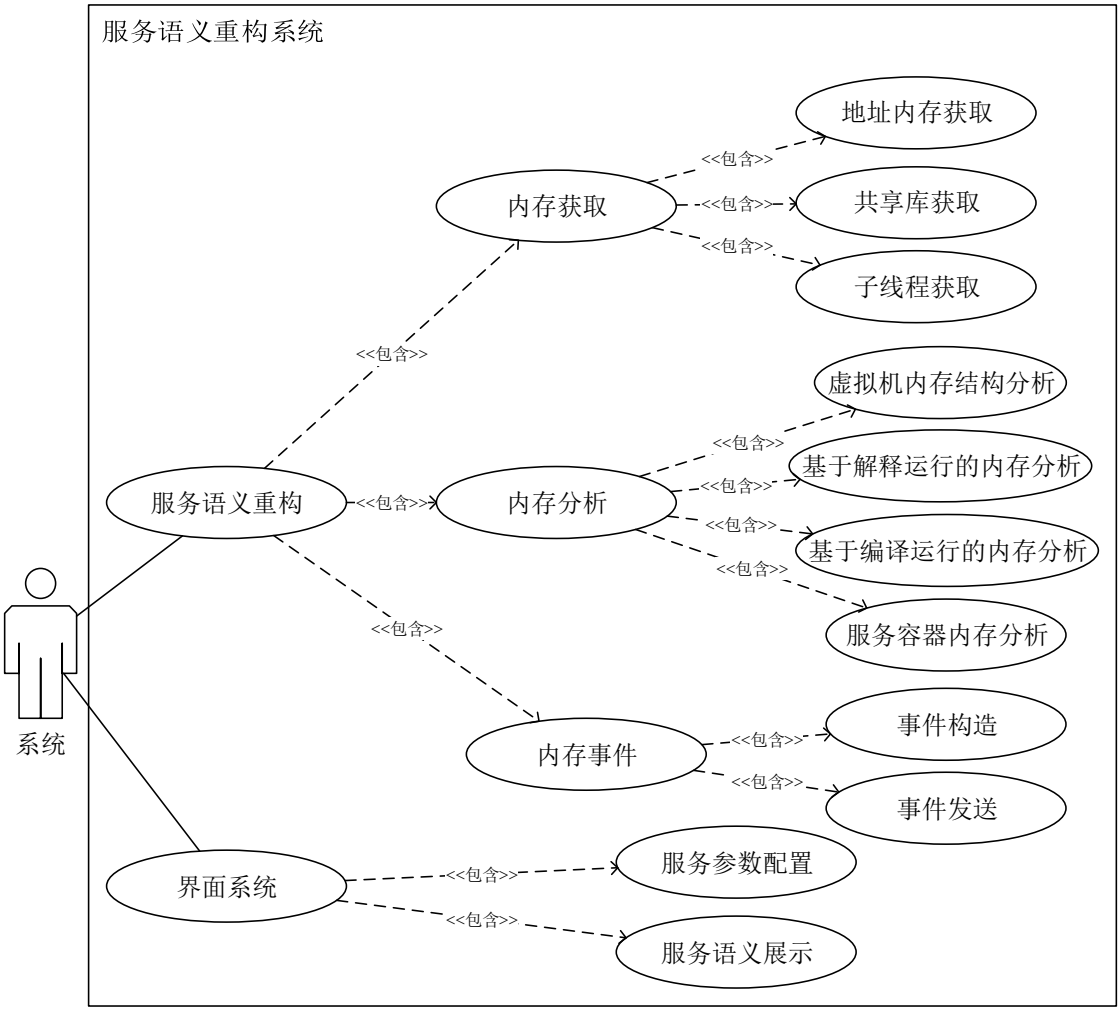

图 3-1 系统用例图

3.1.1 内存获取

内存获取是本系统必不可少的基础，语义重构所需要的内存数据都是由内存获取功能获取到的。为了物联网系统的安全性和可靠性，本系统所监控的物联网服务运行于虚拟化平台。在借助虚拟化技术构建的虚拟化平台中部署物联网服务，在物理机中部署服务语义重构系统。因此，内存获取功能借助虚拟机监控器实现，对虚拟化平台的运行没有任何影响，也不会影响在虚拟机中运行的物联网服务。为了实现内存分析和服务语义重构，内存获取功能需要实现地址内存获取，共享库获取和子线程获取功能。通过这些功能，系统可以实时的获取到物联网服务所在进程的内存数据。通过分析这些数据，从而实现服务语义重构。

地址内存获取，根据内存地址和需要的内存数据长度获取该地址到高地址相应长度的二进制数据。要分析内存，首先要能够获取到内存，因此需要实现地址内存获取功能。如表 3-1。

表 3-1 地址内存获取用例详细情况

参与者	系统
用例描述	地址内存获取
前置条件	系统启动，获得进程描述符
详细描述	1、根据进程描述符，获得进程地址空间 2、根据进程地址空间和给出的地址，得到地址相应的内存数据
后置条件	系统得到相应地址的内存数据

共享库获取，根据物联网服务所在的进程 pid 来获取到该进程所加载的共享库，从而得到共享库的名称和共享库在内存中的地址。系统根据 JVM 加载的共享库来重构 JVM 的内存结构。如表 3-2。

表 3-2 共享库获取用例详细情况

参与者	系统
用例描述	共享库获取
前置条件	系统启动，获得进程描述符
详细描述	1、根据进程描述符，获得虚拟地址映射 2、根据虚拟地址映射，获得共享库信息 3、根据共享库信息，获得共享库名称和在内存中的起始地址
后置条件	系统得到共享库名称和起始地址

子线程获取，根据物联网服务所在的进程 pid 来获取该进程所有子线程的线程 tid。如表 3-3。

表 3-3 子线程获取用例详细情况

参与者	系统
用例描述	子线程获取
前置条件	系统启动，获得进程描述符
详细描述	1、根据进程描述符，获得子线程 2、根据子线程信息，获得子线程 tid
后置条件	系统得到子线程 tid

3.1.2 内存分析

内存分析是实现服务语义重构的关键。只有物联网服务的二进制数据，不知道数据在内存中的结构，无法重构出服务的真实语义。本文所研究的物联网服务是基于 Java 语言开发的，因此，要分析内存数据，必须对 Java 服务的内存运行状态进行分析。因为 Java 服务容器都运行于 JVM 中，所以基于服务容器的服务语义重构要根据 JVM 的内存结构来分析获取到的内存数据。

Java 作为一门高级语言，有一些不同于其他编程语言的特性。Java 程序在编写完成之后，通过编译器来把 Java 代码编译成字节码（bytecode），然后把字节码加载到 JVM 中执行。字节码在 JVM 中执行有两种方式，一种是基于栈的解释器解释执行方式，另一种是 JIT 即时编译器编译执行方式。因此，对内存数据的分析分为基于解释执行的内存分析和基于编译执行的内存分析。同时，Java 服务运行于 Java 服务容器中，也要根据 Java 服务容器的结构进行内存分析。

对于 JVM 而言，内存分析先要分析的是 JVM 虚拟机内存结构。不论字节码是编译运行还是解释运行，JVM 初始化的内存结构是不变的。而且要获得程序的运行方式，先要重构出 JVM 的相关内存数据。所以，内存分析包括了虚拟机内存结构分析，基于解释执行的内存分析，基于编译执行的内存分析和服务容器内存分析四个部分。通过这些功能，系统能够重构出物联网服务的服务语义，为运行时验证系统提供内存事件。物联网服务要求实时性，在进行内存分析时，要保证内存分析能够实时完成。

虚拟机内存结构分析，根据内存数据，共享库信息和子线程信息重构出 JVM 虚拟机的内存数据结构。如表 3-4。

表 3-4 虚拟机内存结构分析用例详细情况

参与者	系统
用例描述	虚拟机内存结构分析
前置条件	内存获取模块初始化完成
详细描述	1、根据共享库信息和子线程信息，得到 JVM 内存结构地址 2、根据地址内存获取和 JVM 内存结构地址，得到 JVM 内存结构数据
后置条件	系统得到 JVM 内存结构数据

基于解释执行的内存分析，根据内存中的线程栈数据重构出线程所调用的栈帧信息，包括栈帧所代表的函数名称，本地变量表等。如表 3-5。

表 3-5 基于解释执行的内存分析用例详细情况

参与者	系统
用例描述	基于解释执行的内存分析
前置条件	虚拟机内存结构分析初始化完成，已读取栈内存
详细描述	1、根据 JVM 内存结构数据和栈内存，分析栈帧类型 2、根据解释运行的栈帧结构，分析栈帧，得到函数名称和函数参数 3、分析所有栈帧得到函数调用过程
后置条件	系统得到函数调用过程和函数参数

基于编译执行的内存分析，根据内存中的线程栈数据和虚拟机编译代码结构重构出线程所调用的栈帧信息，包括栈帧所代表的函数名称，本地变量表等。如表 3-6。

表 3-6 基于编译执行的内存分析用例详细情况

参与者	系统
用例描述	基于编译执行的内存分析
前置条件	虚拟机内存结构分析初始化完成，已读取栈内存
详细描述	1、根据 JVM 内存结构数据和栈内存，获得栈帧，分析栈帧类型 2、根据编译代码区，获得编译运行的栈帧结构 3、根据编译运行的栈帧结构，分析栈帧，得到函数名称和函数参数 4、分析其他栈帧得到函数调用过程
后置条件	系统得到函数调用过程和函数参数

服务容器内存分析，根据服务容器的内存数据利用基于解释运行的内存分析和基于编译运行的内存分析重构出服务容器线程池中执行线程所调用的栈帧信

息，通过栈帧信息分析出调用服务的请求，从而得到调用服务的相关参数。如表3-7。

表 3-7 服务容器内存分析用例详细情况

参与者	系统
用例描述	服务容器内存分析
前置条件	虚拟机内存结构分析初始化完成
详细描述	1、根据 JVM 内存结构数据，读取服务容器线程池中执行线程的内存 2、根据基于解释运行和编译运行的内存分析获取所有执行线程的栈帧信息 3、根据服务容器结构和栈帧信息，得到调用服务的请求 4、根据调用服务的请求得到调用服务的相关参数
后置条件	系统得到调用服务的相关参数

3.1.3 内存事件

内存事件是本系统的主要目的，通过内存获取和内存分析得到的物联网服务的相关信息需要构造成事件，发送给物联网运行时验证系统。物联网运行时验证系统是根据串口事件、网口事件和内存事件进行运行时验证来保障物联网运行的安全性和可靠性的系统。串口事件是虚拟机的物联网服务通过串口发送或接收的事件。网口事件是虚拟机的物联网服务通过网络端口发送或接收的事件。内存事件是通过服务语义重构从虚拟机的物联网服务内存中重构出的事件。由于内存事件是直接从虚拟化平台中运行的物联网服务进程中直接获取的，因此具有高可信性和高优先级。因此，内存事件是物联网运行时验证时系统必不可少的事件。

服务语义重构系统通过内存获取和内存分析重构出物联网服务的栈帧信息和服务调用，内存事件功能把重构出的栈帧的函数调用、函数信息和服务参数构造成事件，并发送给物联网运行时验证系统。内存事件包括了事件构造和事件发送两个功能。

事件构造，根据配置文件要求，把重构出的栈帧函数调用、函数信息和服务参数构造成字符串事件。如表3-8。

表 3-8 事件构造用例详细情况

参与者	系统
用例描述	事件构造
前置条件	系统已得到函数调用和函数参数
详细描述	1、读取配置文件，获取事件格式 2、根据事件格式、函数调用、函数参数以及服务参数，构造字符串
后置条件	系统得到字符串事件

事件发送，根据配置文件与物联网运行时验证系统建立连接，将构造好的字符串事件发送过去，发送频率同样根据配置文件来确定。如表 3-9。

表 3-9 事件发送用例详细情况

参与者	系统
用例描述	事件发送
前置条件	系统已得到字符串事件
详细描述	1、读取配置文件，获取发送端口 2、根据发送端口，建立发送服务 3、与物联网验证时系统建立 4、发送字符串事件到物联网验证时系统
后置条件	系统发送字符串事件到物联网验证时系统

3.1.4 界面系统

本文所实现的服务语义重构系统应当简单易用。因此，本系统需要一个功能完备、方便用户交互的界面系统。界面系统主要是用户使用，尽可能的简便是界面系统的设计目标。基于系统可定制化的要求，界面系统需要一个配置界面，用于配置系统的相关参数，包括虚拟机名称，物联网服务的进程 pid，所需要构造事件的函数名称以及参数数量和相应的参数类型。同时，为了方便用户查看服务语义重构的运行效果，需要有一个可以展示运行结果的界面。

界面系统分为服务参数配置和服务语义展示两个部分。服务参数配置功能通过配置服务的相关参数，使用户可以定制化的对不同的物联网服务进行监控。用户只要在界面中填写相关参数就可以实现定制监控服务，以满足用户不同的监控需求。如表 3-10 所示。

表 3-10 服务参数配置用例详细情况

参与者	用户
用例描述	服务参数配置
前置条件	系统启动
详细描述	1、用户配置参数 2、校验参数 3、如果参数正确，保存参数，提示用户成功 4、如果参数错误，提示用户错误
后置条件	系统提示用户成功或用户错误

服务语义展示用来展示用户所需要的服务调用，参数信息以及内存事件。如果相关信息没有界面进行展示，那么用户在使用过程中不能很好了解系统当前的运行状态。通过展示界面，用户可以很方便的看到物联网当前的函数调用栈，各个函数的本地变量以及发往物联网运行时验证系统的内存事件。如表 3-11 所示。

表 3-11 服务语义展示用例详细情况

参与者	用户
用例描述	服务语义展示
前置条件	系统启动，已有栈帧信息和内存事件
详细描述	1、读取配置文件，获取展示频率 2、读取栈帧信息和内存事件 3、显示栈帧信息和内存事件
后置条件	系统显示栈帧信息和内存事件

3.2 非功能性需求

本系统所面向虚拟化的物联网服务，应用场景一般为工业控制系统。物联网服务在工业控制中的应用具有高响应度和高可靠性的特点，这对本系统的性能和可靠性也提出了很高的要求。因此，高性能和高可靠性也是本系统非功能性的核心需求。

3.2.1 高性能

物联网服务具有高响应度的特点，服务调用在进程中是很迅速的和很频繁的。服务语义重构系统要通过虚拟化平台获取服务调用过程中的内存，就需要在内存获取时做到高性能，能够即时的获取到内存的变化。同时，物联网运行时验证系统同样是面向物联网服务的，也有高性能的要求。所以，在进行内存分析和发送构造内存事件时同样需要做到高性能。

3.2.1 高可靠性

物联网服务在整个物联网系统中具有高可靠性的特点。系统在生产环境中能够长期稳定的运行是一个系统必须满足的需求，也是判断系统优劣的核心技术指标之一。由于内存获取和内存分析都是比较底层的技术，而且物联网服务运行在虚拟化平台之中，因此在系统实现完成并成功运行之后，需要对系统进行反复的测试和调优，以使系统达到高可靠性的要求。如表 3-12 所示。

表 3-12 非功能性需求描述

功能需求描述	详细描述
高性能	运行效率高，响应时间短，符合物联网服务的要求
高可靠性	长时间稳定正确运行

3.3 本章总结

本章从整个系统的需求出发，通过对物联网服务以及服务语义重构详细的介绍，阐述了基于服务容器的服务语义重构系统的设计需求。在功能性需求方面，包括了内存获取、内存分析以及在前两个功能基础之上所应用的内存事件、界面系统。在非功能性需求方面，要求系统具备高性能和高可靠性。

第四章 系统设计

第三章详述了实现服务语义重构系统功能所需要完成的需求和目标。本章将阐述如何设计高性能高可靠性的服务语义重构系统。本章将从系统的设计角度出发，逐个说明内存获取，内存分析，内存事件和界面系统的设计思想，架构和流程。

4.1 系统架构

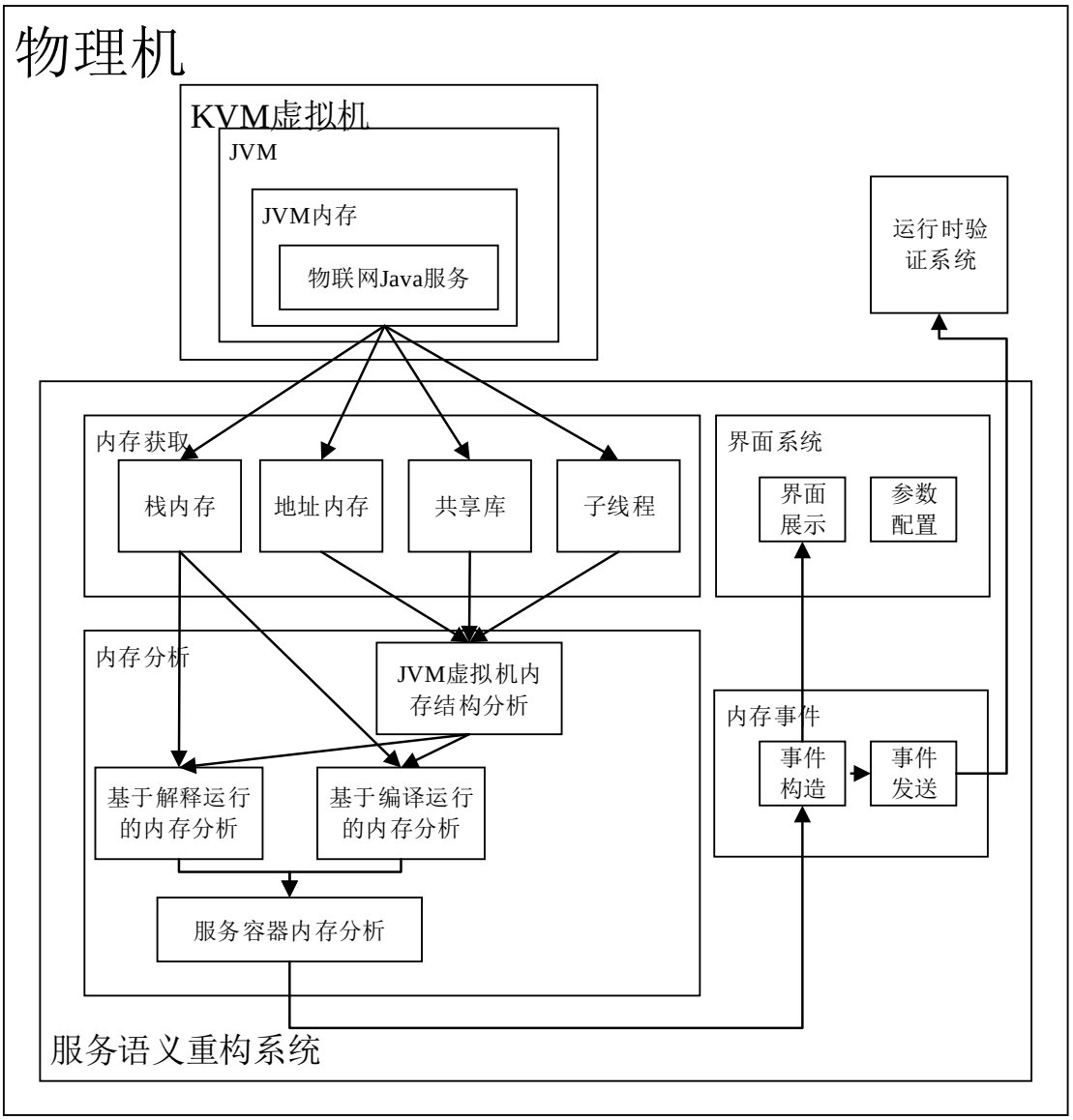

图 4-1 系统设计架构图

基于服务容器的服务语义重构系统通过虚拟化技术，实现了对 Java 服务的内存获取，内存分析，根据分析结果构造发送内存事件，通过界面系统，配置参

数实时展示 Java 服务语义。图 4-1 是本系统的系统设计架构图。根据图中描述，物联网服务部署在上层的虚拟化平台之中，服务语义重构系统部署在下层的物理机之中。物联网服务运行于 JVM 之中，JVM 内存是整个虚拟机内存的一部分。服务语义重构系统通过获取 JVM 内存，从而来重构服务语义。

用户需要在界面系统进行参数配置。然后，内存获取模块根据参数配置来获取 JVM 内存。当内存获取模块初始化完成之后，内存分析模块的 JVM 虚拟机结构分析子模块会调用内存获取模块的地址内存获取，共享库获取和子线程获取三个子模块来构建 JVM 虚拟机内存结构。之后根据 JVM 虚拟机栈帧的执行方式，基于解释运行的内存分析和基于编译运行的子模块调用栈内存获取子模块读取 JVM 栈内存，通过分析得到栈帧相关信息。服务容器内存分析会通过内存获取模块获取服务容器线程池执行线程的内存数据，通过基于解释运行和编译运行的内存分析内存数据来获取调用服务的栈帧，从而得到服务参数。内存事件模块的事件构造子模块根据栈帧相关信息和服务参数构造内存事件，并在界面中展示栈帧信息和内存事件。事件发送子模块把内存事件发送到运行时验证系统。

系统整体架构设计要保证系统简单易用，高性能和可靠性。根据软件工程的设计要求，系统应当做到高内聚低耦合，抽象模块化，减少系统瓶颈，提升系统短板，从而提高系统的整体性能。同时，尽可能的降低系统对 CPU 内存等资源的占用，提高系统的适应性。

4.2 内存获取

内存获取模块是进行语义重构的基础。基于内存分析模块的数据需要，内存获取模块分为栈内存获取、地址内存获取、共享库获取和子线程获取四个子模块。同时，这些功能需要按照参数配置文件来进行获取。在内存获取之前，先对参数配置进行解析，然后根据解析出来的 pid 参数，获取相应 pid 的虚拟机内存。在本系统中，pid 是运行物联网服务的 JVM 在虚拟机操作系统中的进程 id。

4.2.1 内存获取模块

在第二章中介绍了内存自省的相关技术，通过进程链表，可以得到运行物联网服务的 JVM 进程的进程描述符 task_struct。进程描述符 task_struct 包含了一个 Linux 进程的所有信息。task_struct 包含与内存获取模块相关的主要是获取进程地址空间的 get_process_address_space()方法，获取进程虚拟地址映射的 get_proc_maps()方法，和获取进程子线程 threads()方法。如表 4-1 所示。

表 4-1 task_struct 相关函数

函数名称	作用
get_process_address_space()	获取进程地址空间
get_proc_maps()	获取进程虚拟地址映射
threads()	获取进程子线程

进程地址空间 process_address_space 用于栈内存获取和地址内存获取子模块。栈内存获取通过栈起始地址获取进程的栈空间内存数据。地址内存获取根据给出的进程的某个地址获取该地址的内存数据。

在 Linux 中，可以根据内存地址在进程的进程地址空间获得内存地址对应的内存数据。利用进程地址空间 process_address_space 和虚拟机内存自省插件 pyvmi 的 read()方法，系统可以通过进程地址空间来获取进程的内存数据，从而实现了栈内存获取和地址内存获取功能。

通过查看 Linux 系统的/proc/<pid>/maps，可以查看进程的内存映射，包括进程加载的共享库的名称和映射的内存地址。通过 task_struct 的进程虚拟地址映射 proc_maps，可以读取到虚拟机内的/proc/<pid>/maps，从而可以得到运行物联网服务的 JVM 所加载的共享库。借助进程虚拟地址映射 proc_map 实现了共享库获取子模块，得到 JVM 所加载的共享库的名称和在内存中的起始地址。这些库的名称和起始地址可以计算出 JVM 内存结构的变量和常量在内存中的地址。

子线程 threads 通过进程描述 task_struct 的 thread_group 偏移量，来循环遍历获取进程所有的子线程信息。通过子线程 threads 实现子线程获取子模块。本系统的内存分析模块重构 JVM 内存结构时需要子线程的 tid。

栈内存获取和地址内存获取子模块在整个内存获取和内存分析过程中，会被频繁的调用，因此模块需要设计的高效可靠。由于共享库获取和子线程获取子模块只在初始化的过程加载一次，对运行效率的要求没有那么高。

4.2.2 内存获取流程

系统启动后初始化内存获取模块。内存获取流程图如图 4-2 所示。在内存获取流程中，系统主要会完成以下工作：

1) 启动服务语义重构系统；
2) 从参数配置文件中读取内存获取模块所需要的虚拟机名称，以及运行物联网服务的 JVM 进程 pid；
3) 根据 task_struct 的链表结构，遍历查找物联网服务进程 pid，获取 pid 所对应的 task_struct；
4) 初始化共享库获取子模块，利用 task_struct.mm（内存描述符）的 mmap

循环遍历 JVM 所加载的所有共享库信息，从虚拟机内存中读取共享库名称和起始地址，并保存到系统的 libs 列表之中；

5) 初始化子线程子模块，利用 task_struct 的 thread_groups 循环遍历物联网服务所在的进程的所有子线程信息，从中读取子线程 tid，并保存到系统的 threadsId 列表之中；

6) 初始化栈内存和地址内存获取子模块，利用 pyvmi 和 task_struct.mm（内存描述符）的 pdg 得到 directory_table_base，然后根据 directory_table_base 生成进程地址空间 process_address_space，之后在内存分析阶段，可以利用 process_address_space 和 pyvmi 的 read()方法来获取地址对应的内存数据，同时在得到栈起始地址的情况下，也可以获取栈内存。

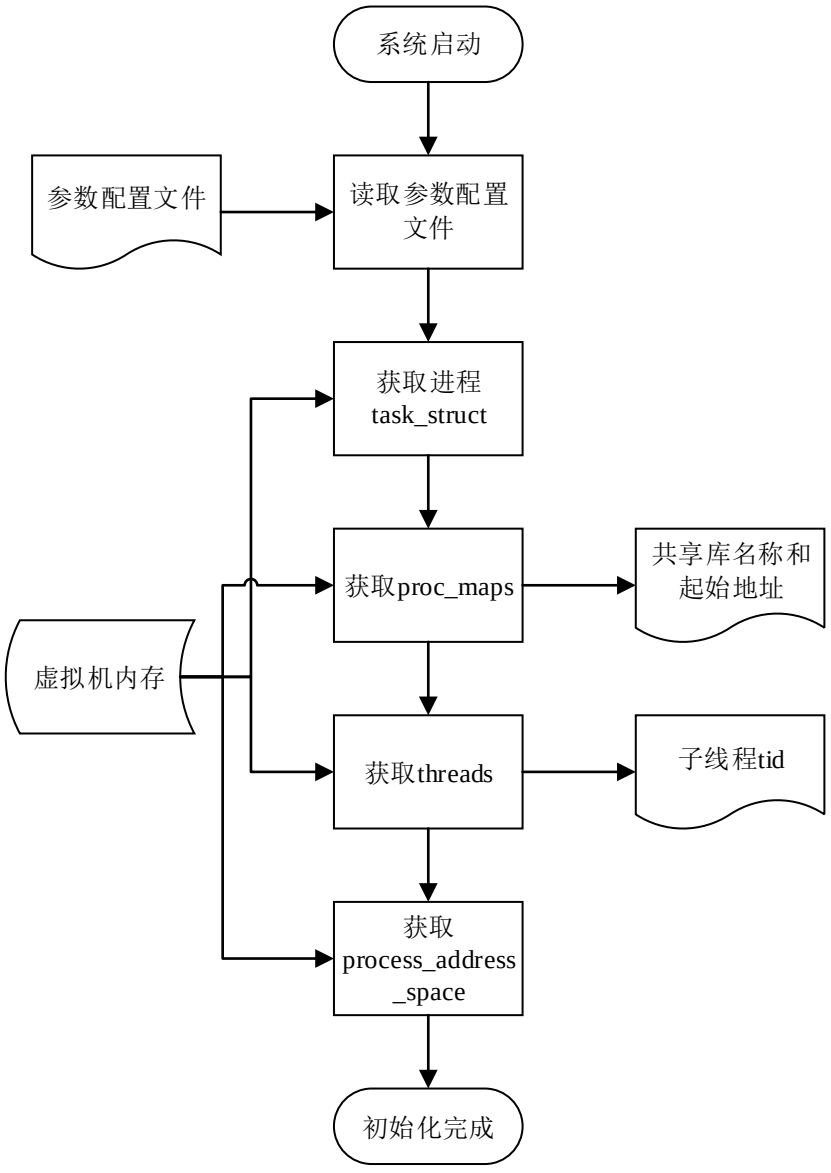

图 4-2 内存获取流程图

4.3 内存分析

内存分析模块是本系统的核心模块。根据 JVM 内存结构和内存事件模块的需求，本系统将内存分析模块分为 JVM 虚拟机内存结构分析、基于解释运行的内存分析、基于编译运行的内存分析和服务容器内存分析四个子模块。在内存获取模块初始化之后，系统获取 pid 对应的 JVM 进程所加载的共享库名称和起始地址以及子线程 tid 信息，同时，系统可以调用栈内存和地址内存获取子模块来读取内存。

4.3.1 JVM 虚拟机内存结构分析

JVM 虚拟机种类很多，本文研究的是 Hotspot 虚拟机。Hotspot 虚拟机的 VMTypeEntry，VMStructEntry，VMIntConstantEntry 和 VMLongConstantEntry 数据结构用来表示 JVM 中所有静态、非静态的变量和常量以及数据结构。如表 4-2 所示。

表 4-2 Hotspot 虚拟机数据结构

名称	说明
VMTypeEntry	表示一个虚拟机的数据类型
VMStructEntry	表示一个属于某个数据类型的成员变量
VMIntConstantEntry	表示一个整型的常量
VMLongConstantEntry	表示一个长整型的常量

系统调用虚拟机内存结构分析子模块初始化 JVM 内存结构数据。在内存分析模块初始化的同时，虚拟机内存结构分析子模块也随之初始化。根据 VMTypeEntry，VMStructEntry，VMIntConstantEntry 和 VMLongConstantEntry 数据结构从内存中读取数据，获取到 JVM 中所有的静态和非静态的变量。在 Linux 的 Hotspot JVM 源代码中，这四个数据结构被编译成了 libjvm.so 共享库。因为系统已经获取了所有共享库的名称和起始地址，所以我们可以得到 libjvm.so 的起始地址。通过 readelf 命令可以得到符号 symbol 在共享库 lib 中的偏移地址，根据共享库的起始地址和符号的偏移地址，可以得到符号 symbol 在内存中的地址。根据 JVM 源代码 vmStructs.cpp，在共享库 libjvm.so 中，VMTypeEntry，VMStructEntry，VMIntConstantEntry 和 VMLongConstantEntry 数据结构分别对应了 gHotSpotVMTypes，gHotSpotVMStructs，gHotSpotVMIntConstants，gHotSpotVMLongConstants 四个引用。如表 4-3 所示。因此，通过 readelf 命令读取 gHotSpotVMTypes，gHotSpotVMStructs，gHotSpotVMIntConstants，gHotSpotVMLongConstants 在 libjvm.so 中的偏移地址。根据 libjvm.so 的起始地

址可以计算出 gHotSpotVMTypes，gHotSpotVMStructs，gHotSpotVMIntConstants，gHotSpotVMLongConstants 在 JVM 内存中的地址。

表 4-3 libjvm.so 的相关引用

名称	说明
gHotSpotVMTypes	VMTypeEntry 在 libjvm.so 中的引用
gHotSpotVMStructs	VMStructEntry 在 libjvm.so 中的引用
gHotSpotVMIntConstants	VMIntConstantEntry 在 libjvm.so 中的引用
gHotSpotVMLongConstants	VMLongConstantEntry 在 libjvm.so 中的引用

VMTypeEntry 表示 JVM 数据类型的对象，有 typeName，superclassName，isOopType，isIntegerType，isUnsigned，size 六个属性。如表 4-4 所示。

表 4-4 VMTypeEntry 的相关属性

名称	说明
typeName	当前类型名称
superclassName	当前类型的父类型的名称
isOopType	当前类型是否是面向对象类型
isIntegerType	当前类型是否是整型
isUnsigned	当前类型是否是无符号的类型
size	当前类型所占的字节大小

只有 gHotSpotVMTypes 在内存的地址，无法得到 gHotSpotVMTypes 的六个属性值。因此，还要得到：

gHotSpotVMTypeEntryTypeNameOffset，

gHotSpotVMTypeEntrySuperclassNameOffset，

gHotSpotVMTypeEntryIsOopTypeOffset，

gHotSpotVMTypeEntryIsIntegerTypeOffset，

gHotSpotVMTypeEntryIsUnsignedOffset，

gHotSpotVMTypeEntrySizeOffset

六个偏移量。除此之外，VMTypeEntry 是一个链表结构，还有一个偏移量 gHotSpotVMTypeEntryArrayStride 用于表示下一个 VMTypeEntry 相对当前的偏移量。如表 4-5 所示。

表 4-5 gHotSpotVMTypes 的相关偏移量

名称	说明
gHotSpotVMTypeEntryTypeNameOffset	typeName 属性相对于 gHotSpotVMTypes 地址的偏移量
gHotSpotVMTypeEntrySuperclassNameOffset	superclassName 属性相对于 gHotSpotVMTypes 地址的偏移量
gHotSpotVMTypeEntryIsOopTypeOffset	isOopType 属性相对于 gHotSpotVMTypes 地址的偏移量
gHotSpotVMTypeEntryIsIntegerTypeOffset	isIntegerType 属性相对于 gHotSpotVMTypes 地址的偏移量
gHotSpotVMTypeEntryIsUnsignedOffset	isUnsigned 属性相对于 gHotSpotVMTypes 地址的偏移量
gHotSpotVMTypeEntrySizeOffset	size 属性相对于 gHotSpotVMTypes 地址的偏移量
gHotSpotVMTypeEntryArrayStride	下一个 VMTypeEntry 类型相对当前 gHotSpotVMTypes 的偏移量

根据 gHotSpotVMTypes 在内存的地址和六个属性的偏移量，可以得到当前 VMTypeEntry 指针六个属性的值。然后根据下一个 VMTypeEntry 的偏移地址得到下一个 VMTypeEntry 的起始地址。继续遍历从而得到所有的 VMTypeEntry。如图 4-3 所示。

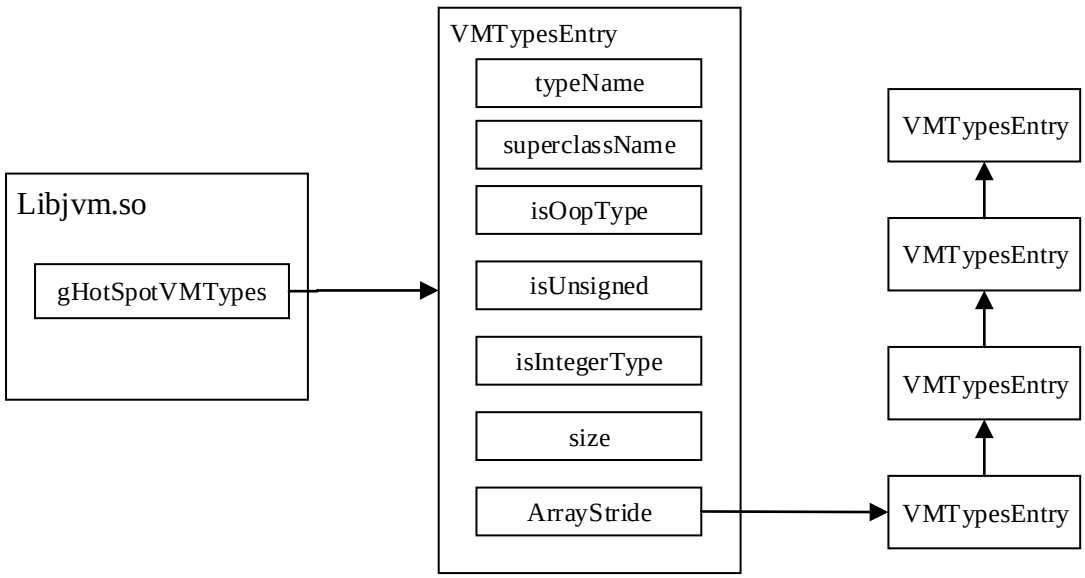

图 4-3 VMTypeEntry 结构图

VMStructEntry 表示一个属于某个数据类型的成员变量，有 typeName，

fieldName，typeString，isStatic，offset，address 六个属性。如表 4-6 所示。

表 4-6 VMStructEntry 的相关属性

名称	说明
typeName	当前成员变量所属于的类型名称
fieldName	当前成员变量名称
typeString	当前成员变量类型
isStatic	当前成员变量是否是静态的
offset	当前成员变量如果是非静态的，它在所属类型中的偏移量
address	当前成员变量如果是静态的，它在内存中的地址

typeName 表示当前类型名称，fieldName 表示当前类型的成员变量名称，typeStrng 表示成员变量的类型名称，isStatic 表示是否是静态成员变量，offset 表示非静态成员在当前类型中的偏移，address 表示静态成员变量在内存的地址。与 VMTypeEntry 类似，也有六个偏移量与六个属性相对应。除此之外，还有 gHotSpotVMStructEntryArrayStride 来表示下一个 VMStructEntry 相对当前的偏移量。同样的，循环遍历 gHotSpotVMStructs 可以得到所有的 VMStructEntry。

VMIntConstantEntry 和 VMLongConstantEntry 用于表示 JVM 内存中的一些整型和长整型的常量，都具有 name 和 value 两个属性和相应的偏移量。与之前类似，通过循环 gHotSpotVMIntConstants 和 gHotSpotVMLongConstants 来得到所有的 VMIntConstantEntry 和 VMLongConstantEntry。如表 4-7 所示。

表 4-7 VMIntConstantEntry 和 VMLongConstantEntry 的相关属性

名称	说明
name	当前常量的名称
value	当前常量的值

在获取和分析出所有的类型、成员变量、整型常量和长整型常量之后，系统初始化虚拟机的结构。在虚拟机内存结构中，每个线程拥有自己的程序计数器，虚拟机栈和本地方法栈，而 Java 堆和方法区则是线程共有的。因此要对栈内存进行分析，先要获取到相应的线程地址。在已经获取的 JVM 类型中，typeName 为 Threads 的类型是用来表示线程列表的，其中它的静态成员变量_thread_list 代表了线程列表的内存地址，存储了第一个线程的内存地址，通过这个地址可以得到第一个线程。类似的，typeName 为 JavaThread 的类型是用来表示一个线程的，其中它的非静态成员变量_next 表示存储下一个线程内存地址的偏移量。通过当前线程地址和_next 偏移量可以得到下一个线程的内存地址。循环遍历可以得到

所有的线程。通过 JavaThread 的 oop 类型非静态成员变量_threadObj 可以得到线程名称。单线程的 Java 服务被 main 线程的 main 函数调用，而多线程运行于 Java 服务容器的 Java 服务被服务容器线程池执行线程的 run 函数调用。根据系统配置，系统监控 Java 服务 main 函数所在的 main 线程或 run 函数所在的执行线程。JavaThread 的非静态属性_stack_base 表示存储栈起始地址的偏移量，通过线程的地址和_stack_base 的偏移量可以得到线程私有的栈地址。JavaThread 的非静态属性_stack_size 表示存储栈大小的偏移量，通过线程的地址和_stack_size 的偏移量可以得到线程私有栈的大小。如图 4-4 所示。

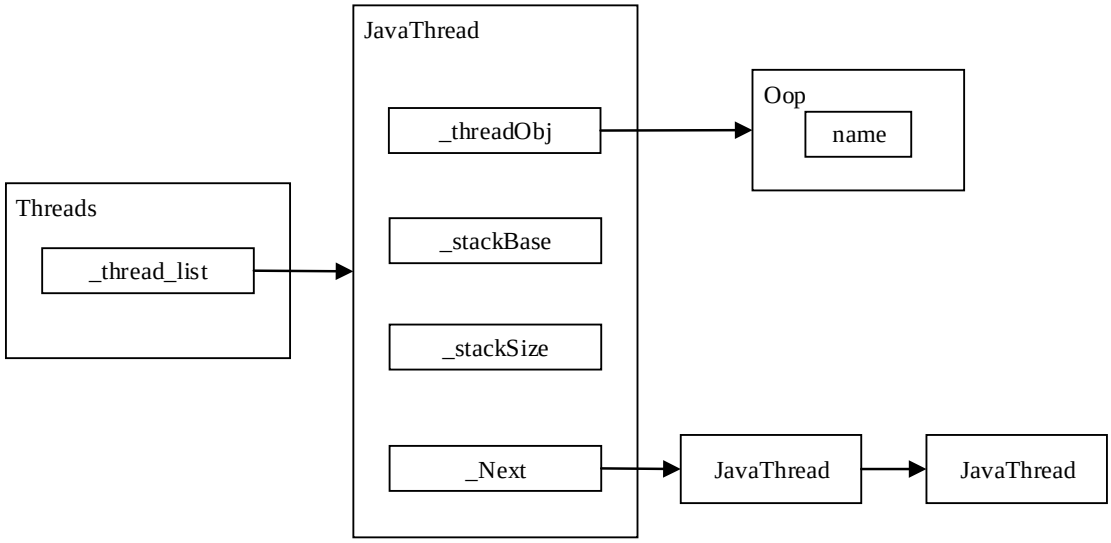

图 4-4 JVM 线程结构图

虚拟机内存结构分析子模块初始化完成之后，虚拟机内存结构和相关静态数据分析完成，这些数据存储在服务语义重构系统的相应变量和常量之中。

4.3.2 JVM 虚拟机内存结构分析流程

在内存获取模块初始化之后，系统开始 JVM 虚拟机内存结构分析流程。JVM 虚拟机内存结构分析流程图如图 4-5 所示。在 JVM 虚拟机内存结构分析流程中，系统主要会完成以下工作：

1) 利用 readelf 模块读取 libjvm.so 的相关偏移量；
2) 根据 gHotSpotVMTypes，gHotSpotVMStructs，gHotSpotVMIntConstants，gHotSpotVMLongConstants 变量的偏移量和 libjvm.so 的起始地址，计算出 gHotSpotVMTypes，gHotSpotVMStructs，gHotSpotVMIntConstants，gHotSpotVMLongConstants 在内存中的地址，根据六个属性的偏移量计算出属性的值；
3) 根据 ArrayStride 偏移量判断是否还有下一个数据，如果有，根据偏移量计算下一个数据的地址，执行 2)，否则执行 4)；
4) 读取 typeName 为 Threads 的静态成员变量_thread_list，得到第一个线程

JavaThread 的内存地址；

5) 通过 JavaThread 非静态成员变量_threadObj 可以得到线程名称，判断当前线程是否是 main 线程，如果是，执行 7)，否则执行 6)；

6) 根据 JavaThread 非静态成员变量_next 得到下一个线程 JavaThread 的内存地址，执行 5)；

7) 根据 JavaThread 非静态成员变量 stack_base 和_stack_Size，得到栈地址和大小。

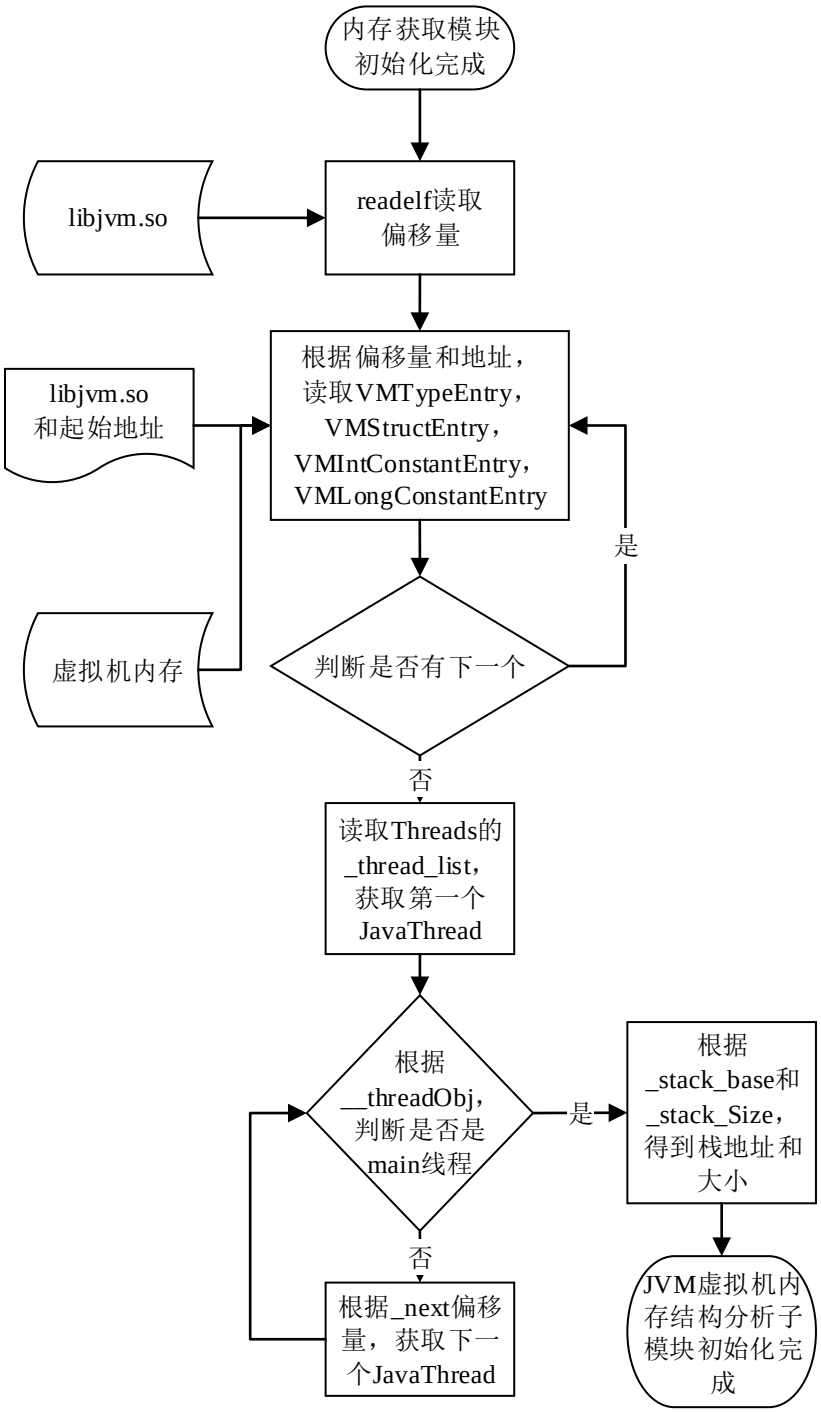

图 4-5 JVM 虚拟机内存结构分析流程图

4.3.3 基于解释运行的内存分析

在通过线程的栈地址和栈大小获取到线程的栈内存之后，系统根据 Java 栈帧的特点来进行分析。在已经获取 JVM 的类型中，typeName 为 JavaFrameAnchor 的类型表示 Java 的栈帧结构，它有三个属性_last_Java_fp，_last_Java_sp，_last_Java_pc，分别代表了当前运行的栈帧的 fp（栈帧指针），sp（操作数栈），pc（程序计数器）的偏移量。同时，JavaThread 的_anchor 表示当前线程的存储 JavaFrameAnchor 的地址的偏移量。所以，通过当前线程的地址和这些偏移量可以得到当前运行的栈帧的 fp，sp 和 pc。如图 4-6 所示。

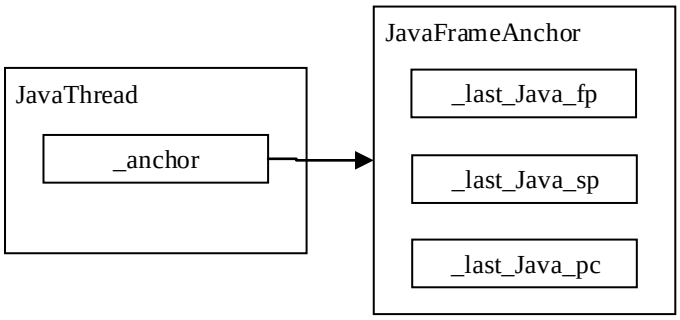

图 4-6 JavaFrameAnchor 结构图

在 64 位 JVM 中，地址大小的单位为 8，地址从高到低即压栈地址越来越低。在 JVM 进行压栈时，已经可以确定之前栈帧的所有信息，压栈的栈帧保存了之前栈帧的相关信息，以便在出栈时可以找到之前的栈帧。在 JVM 解释运行时，新栈帧的 fp 中保存着旧栈帧的 fp，新栈帧的 fp 前一个单位地址（fp＋8）中保存着旧栈帧的 pc，新栈帧 fp 的前两个单位地址（fp + 16）是旧栈帧的 sp。新栈帧的 unextendedSP 是栈帧后一个单位地址保存的值。结构如图 4-7 所示。

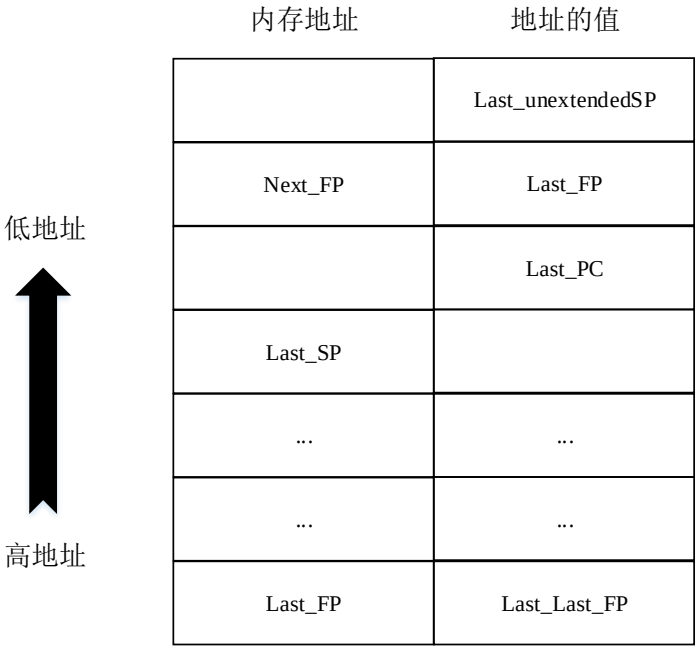

图 4-7 解释运行的 JVM 栈结构图

当前运行的栈帧是现在函数栈的栈顶栈帧。通过当前运行的栈帧的 fp，可以得到前一栈帧的 pc，sp 和 fp。循环遍历 fp，可以得到当前线程栈中的所有栈的 pc，sp 和 fp。由于 main 线程的 main 方法是 JVM 的起始方法，因此 main 函数是 main 线程栈底的栈帧。同样的，执行线程的 run 方法是执行线程的起始方法，因此 run 函数是执行线程栈底的栈帧。所以在 JVM 虚拟机内存结构分析子模块初始化之后，根据当前栈帧的 fp 可以得到 main 函数或者 run 函数的 fp。而 fp 是一个栈帧的栈底，因此得到了整个线程栈的栈底地址。

当线程暂停等待或函数暂停等待时，线程 JavaFrameAnchor 的_last_Java_fp，_last_Java_sp，_last_Java_pc 地址中才保存有最新的值。当栈顶函数不断变化时，它们的值为 0。因此在线程运行函数调用频繁时，无法通过_last_Java_fp 来重构栈帧，同时 sp 代表的操作数栈在函数可能进行不同分支时压入栈的字节码指令可能并不相同，因此即使是 main 函数或者 run 函数的栈帧，sp 也是在不断变化的，只能使用固定不变的栈底地址 fp 来重构栈帧。根据 main 线程 main 函数的 fp 地址或者执行线程 run 函数的 fp 地址，调用栈内存获取子模块，读取栈内存进行分析。

在读取栈内存的过程中，对地址和地址中存储的内存数据进行映射。因为 main 函数或者 run 函数的 fp 地址是存储在上一个栈帧的 fp 地址中的，所以根据地址中储存的内存数据来查找地址，可以找到上一个栈帧的 fp 地址。循环遍历可以找到所有的栈帧 fp。

找到所有的栈帧 fp 之后，需要对每一个栈帧进行分析。在解释运行的栈帧中，fp 决定了栈帧中函数引用和函数参数的位置。根据函数引用，可以得到当前栈帧代表的函数名称。在 JVM 进行类加载时，类和类的成员函数会加载到方法区，在之后的运行阶段地址都不会变化。通过函数引用可以得到方法区的函数名称等信息。函数引用存储在 fp 之前 3 个地址单位的位置（fp – 3 * 8），通过栈帧的 fp 地址可以得到函数指针的地址，最后得到函数名称。函数的本地变量都存储在栈帧的局部变量表中，包括 boolean，byte，char，short，int，float，long，double 和对象引用。首个本地变量存储在 fp 之前 6 个地址单位的位置（local = fp – 6 * 8），通过栈帧 fp 地址可以得到首个变量的地址，而下一个变量位置存储在前一个变量之前一个地址单位（local - 8）。通过循环遍历 local，可以得到所有的本地变量的地址。通过地址可以得到变量的值或者引用。如图 4-8 所示。

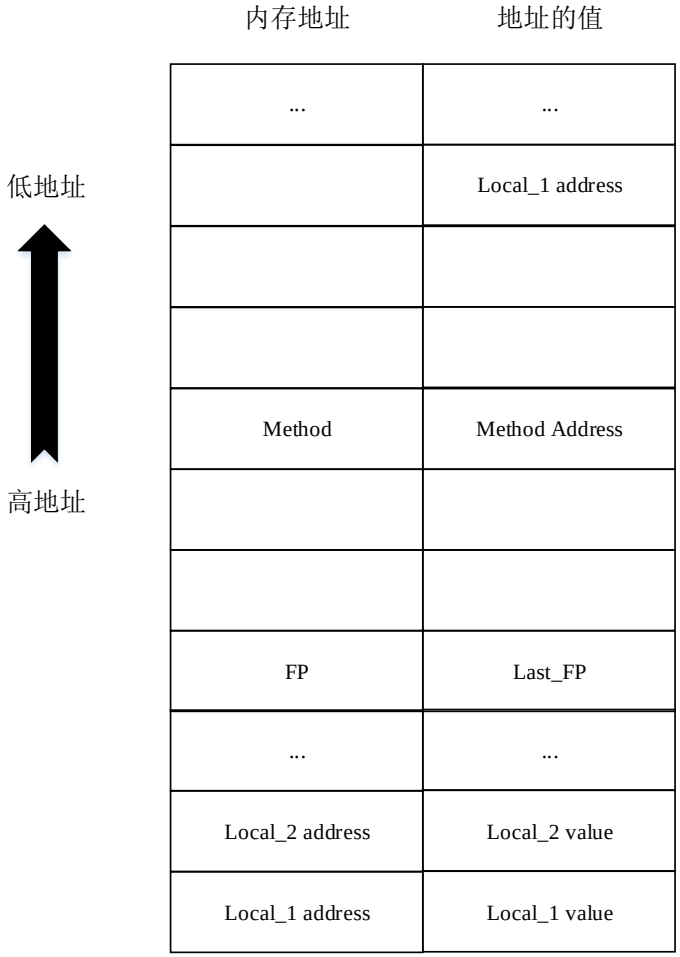

图 4-8 解释运行的栈帧函数和参数结构图

4.3.4 基于解释运行的内存分析流程

在 JVM 虚拟机内存结构分析子模块初始化之后，系统开始基于解释运行的内存分析流程。基于解释运行的内存分析流程图如图 4-9 所示。在基于解释运行的内存分析流程中，系统主要会完成以下工作：

1) 在线程暂停时，根据 JavaFrameAnchor 的_last_Java_fp 属性，得到 fp 在 JavaFrameAnchor 中偏移量，根据 JavaThread 的_anchor 属性，得到 JavaFrameAnchor 在当前 JavaThread 中偏移量，计算出当前栈帧的 fp 地址；
2) 根据 fp 计算出函数引用 method(fp – 3 * 8)，根据函数引用得到当前栈帧的函数名称，判断是否是 main 函数，如果是，执行 4)，否则，执行 3）；
3) 获取 fp 中存储的下一个栈帧的 fp，执行 2)；
4) 根据栈地址和栈大小利用栈内存获取子模块，读取 main 线程的栈内存；
5) 根据当前函数的 fp 地址，得到存储当前函数 fp 的 next 栈帧的 fp；
6) 根据 fp 计算出函数引用地址，得到 method address(fp – 3 * 8)；

7) 判断 method 是否函数，即这个引用是否指向了一个函数，如果是，执行 8)，否则，执行 11)；
8) 读取参数配置文件，得到要监控的函数名称，参数类型和参数个数；
9) 根据 method 得到函数名称，判断是否是要监控的函数，如果是，执行 10)，否则，执行 5)；
10) 根据 fp 计算出 local 地址(fp – 6 * 8)，下一个 local 是当前 local 的前一个地址（local - 8），根据函数个数，获取所有的 locals 地址，根据参数类型，还原 local 地址中的值，执行 5)；
11) 当前栈内存已经不是函数，栈帧分析完成，生成本次栈帧分析的信息，包括函数调用关系和被监控函数的参数值。进行下一次分析，执行 4)。

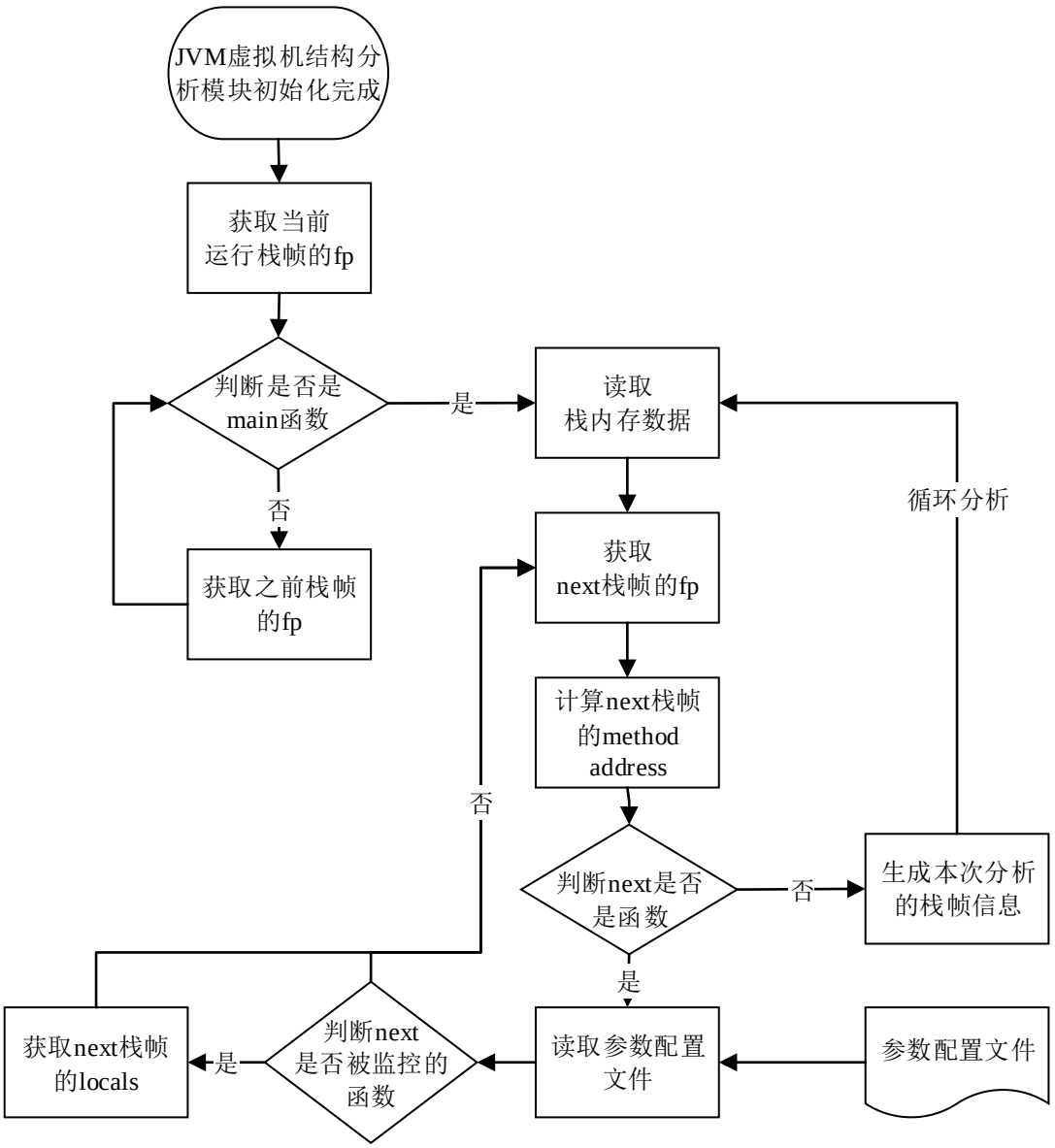

图 4-9 基于解释运行的内存分析流程图

4.3.5 基于编译运行的内存分析

在 JVM 运行过程中，为了提高程序的运行效率，在解释执行的同时，根据代码中函数的调用次数或者循环体的循环次数，判断代码是否热点代码。然后以函数为单位，调用编译线程将热点代码编译成本地二进制代码，然后执行二进制代码。因此，当 JVM 进入编译执行阶段时，函数栈的内存情况是不同于解释运行的。

在函数没有进入编译执行之前，函数是解释执行的，通过基于解释执行的内存分析可以得到线程的栈底。在编译执行时，函数的栈底是不变的。编译之后，执行代码变为本地代码，操作数等相关信息不会再压入栈中。在已经获取的 JVM 类型中，typeName 为 CodeCache 类型的静态成员变量_heap 和当前栈帧的 pc 可以得到属于当前栈帧的 typeName 为 CodeBlob 类型的地址。在编译型的栈帧里，用于表示这个地址的数据结构是 CodeBlob 类型的子类型，typeName 为 nmethod 的 NMethod 类型，根据它的非静态成员变量_scopes_data_offset 确定代码区 Scope。如图 4-10 所示。

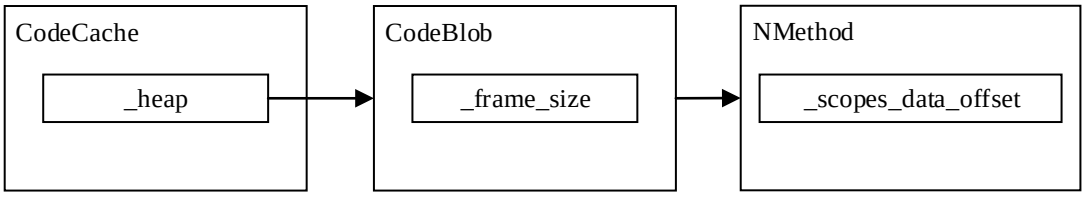

图 4-10 NMethod 结构图

Scope 实质是一系列的压缩数据，根据一定的顺序结构进行读取，得到代码区的相应情况。在编译型栈帧中，多个连续调用的编译型共享相同的 Scope。如表 4-8 所示。

表 4-8 Scope 的相关属性

名称	说明
senderDecodeOffset	相同 sp 的编译型栈帧的下一栈帧在 Scope 中的偏移，为 0 则代表为最后一个相同的编译型栈帧
method	当前栈帧的函数引用地址
localsDecodeOffset	当前栈帧的本地变量相关信息的偏移。根据偏移读取值，起始为参数数量，第一个值为参数类型，当参数类型为常量时，第二个值为常量的值或引用，当参数类型不为常量时，第二个值代表本地变量地址相对于 unextendedSP 的偏移量。之后依次类推。

当分析编译型栈帧时，在栈帧中重要的地址信息是 unextendedSP。通过上一个栈帧的 sp 和 CodeBlob 中的 frameSize 可以得到新编译栈帧的 unextendedSP（unextendSp = sp - frameSize），而新编译栈帧取决于新栈帧之后一个栈帧的运行状态，如果后一个栈帧是解释运行的，因为 unextendedSP 存储在前一个栈帧 fp 的前一个地址中（fp - 8），所以通过编译栈帧的 unextendedSP 可以得到后一个栈帧的 fp，再根据后一个栈帧的 fp，根据解释运行的栈帧变化规律，得到编译栈帧的 sp，fp，pc。如果后一个是编译型栈帧，连续的编译型栈帧共享相同的 fp，sp 和 pc，根据 Scope 的 senderDecodeOffset 来循环遍历判断有多少个相同 sp 的编译型栈帧。所以同样先计算出下一个调用的解释型栈帧的 fp，然后得到编译型所共同的 fp，sp 和 pc。如果当前编译型栈帧是最后一个被调用的栈帧，那么它的 unextendedSP 和 sp 相同，而 pc 存储在 sp 的后一个地址里，因此可以得到当前编译型栈帧的 pc，从而得到代码区 Scope。编译运行的栈帧结构如图 4-11 所示。

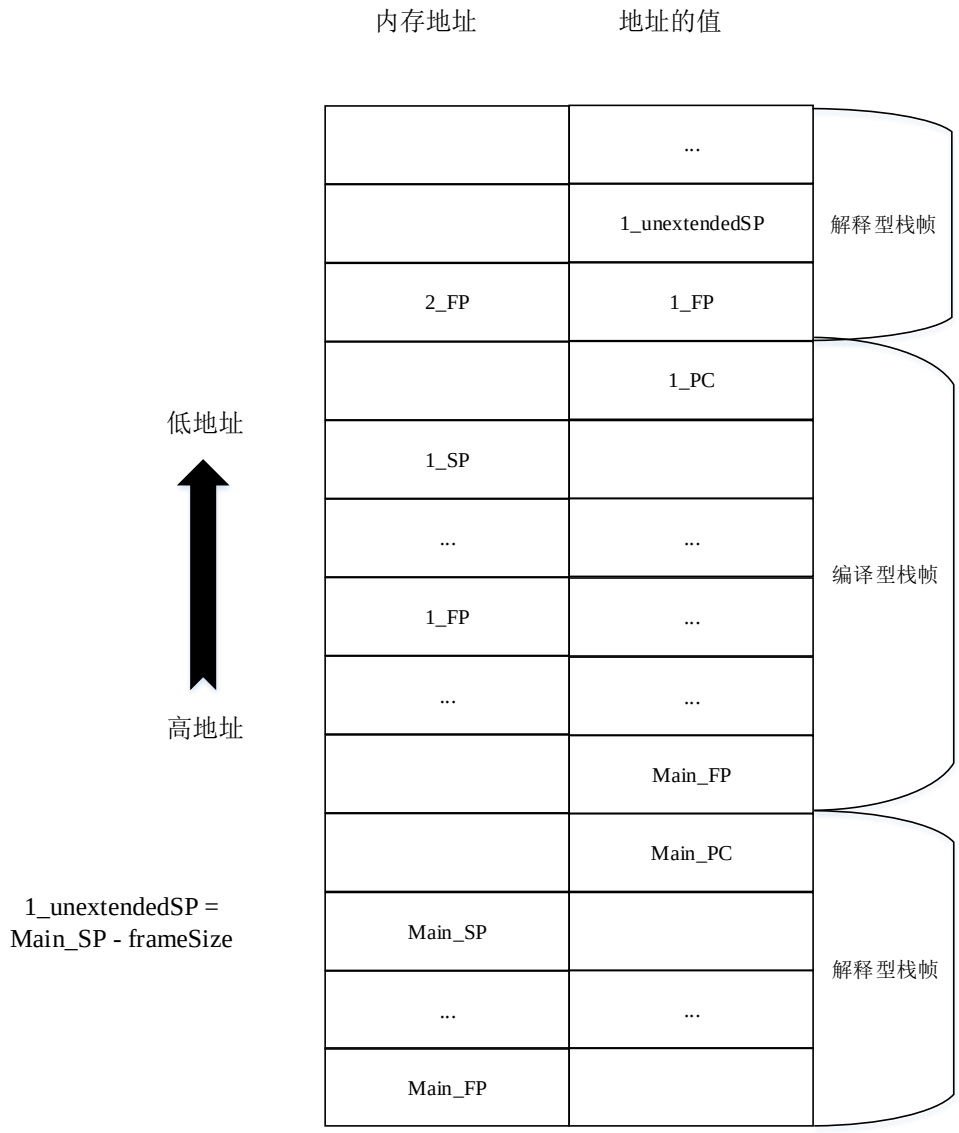

图 4-11 编译运行的栈帧结构图

在分析栈内存时，从栈底向上分析，因此在计算之前无法判断它的栈帧类型和 frameSize。因此，只能采用搜索的方式进行判定。先按照解释型栈帧进行分析，计算出函数引用，之后判断是否存在函数。如果不存在，再根据编译型栈帧进行分析。遍历所有 frameSize，计算栈帧 pc，然后跟 CodeBlob 进行匹配，直至匹配成功。若匹配失败，则已经不存在栈帧。

当得到当前编译型栈帧的 sp，fp 和 pc 之后，根据当前 Scope 来进行参数分析。编译运行有两个优化。首先，在解释型栈帧中，一个变量占一个地址单位，而在编译型中，优化了内存空间，因此有些变量占半个地址单位，有些变量占一个地址单位。其次，代码在进行编译的过程中，会进行函数内联等编译优化，因此会出现多个函数合成了一个函数共用栈帧的情况。这时候被内联函数的本地变量也会进行共享，即当前一个函数的本地变量传给被内联的函数会使用同一个地址的局部变量表存储。

因此，对编译型参数分析必须结合 Scope 代码区的代码优化情况进行分析。根据 Scope 的第二个数据代表的地址偏移，读取函数引用地址，获取当前栈帧的函数名称。根据 Scope 和 localsDecodeOffset 偏移量，从地址中可以读取参数的数量。在参数数量之后，可以读到参数的类型。在参数类型之后的数据根据参数类型的不同有不同的含义。如果本地变量在进行编译时被优化成了常量，那么这里的数据就是常量的值或者引用。如果没有被优化，那么这里的值经过计算后就是本地变量 local 地址相对于 unextendedSP 地址的偏移量（local = unextendedSP + offset）。通过 local 地址可以得到本地变量的地址。根据 senderDecodeOffset 循环读取 Scope，可以得到所有的本地变量的地址。通过本地变量的地址可以得到本地变量的值。图 4-12 是某种可能本地变量的情况。

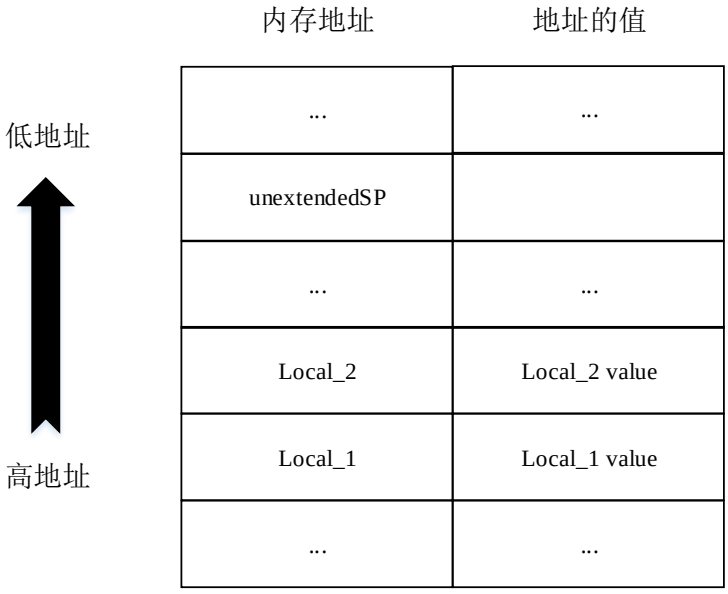

图 4-12 某种情况下编译运行的栈帧的结构图

4.3.6 基于编译运行的内存分析流程

在被监控的 Java 服务运行的 JVM 开始编译执行代码之后，系统开始基于编译运行的内存分析流程。基于编译运行的内存分析流程图如图 4-13 所示。在基于编译运行的内存分析流程中，系统主要会完成以下工作：

1) 在线程暂停时，根据 JavaFrameAnchor 的_last_Java_fp 属性，得到 fp 在 JavaFrameAnchor 中偏移量，根据 JavaThread 的_anchor 属性，得到 JavaFrameAnchor 在当前 JavaThread 中偏移量，计算出当前栈帧的 fp 地址；
2) 根据 fp 计算出函数引用 method(fp - 3 * 8)，根据函数引用得到当前栈帧的函数名称，判断是否是 main 函数，如果是，执行 4)，否则，执行 3)；
3) 获取 fp 中存储的下一个栈帧的 fp，执行 2)；
4) 根据栈地址和栈大小利用栈内存获取子模块,读取 main 线程的栈内存；
5) 根据当前栈帧的 fp 地址，得到存储当前栈帧 fp 的 next 栈帧的 fp；如果当前栈帧是编译型栈帧，则是根据当前栈帧的 unextendedSP 得到存储 unextendedSP 的地址，根据地址求出 next 栈帧的 fp；
6) 根据 fp 计算出函数引用地址，得到 method address(fp - 3 * 8)；
7) 判断 method 是否函数，即这个引用是否指向了一个函数，如果是，执行 8)，否则，执行 11)；
8) 读取参数配置文件，得到要监控的函数名称，参数类型和参数个数；
9) 根据 method 得到函数名称，判断是否是要监控的函数，如果是，执行 10)，否则，执行 5)；
10) 根据 fp 计算出 local 地址(fp - 6 * 8)，下一个 local 是当前 local 的前一个地址（local - 8），根据函数个数，获取所有的 locals 地址，根据参数类型，还原 local 地址中的值，执行 5)；
11) 当前栈帧不是解释型栈帧，进行编译型的分析；
12) 设置 frameSize，根据当前的 fp 和内存地址中存的值，得到存储 fp 的地址，先前两个地址单位得到 sp，根据当前 sp 和 frameSize 计算编译型栈帧的 unextendedSP，根据 unextendedSP 计算 sp，fp 和 pc；
13) 根据 pc 和 CodeCache 类型的静态成员变量_heap 可以得到 CodeBlob，判断得到的 CodeBlob 和 frameSize 是否匹配，如果可以得到且匹配，执行 14)，否则执行 17)；
14) 根据 CodeBlob 得到 NMethod，获取代码段 Scope；
15) 根据 Scope 的 localsDecodeOffset 偏移量得到代码段中 locals 的位

置，根据读取代码段的 local 类型和偏移量，得到 local 相对于 unextendedSP 的偏移量，从而得到 locals；

16) 根据 Scope 的 senderDecodeOffset 偏移量判断代码段中下一个编译栈帧是否存在，如果是，执行 14)，否则执行 5)；

17) 判断 frameSize 是否超过了设定的限制，如果是，那么说明在栈内存中无法再重构出函数，重新进行下一次分析，执行 4)，否则更新 frameSize，执行 12)。

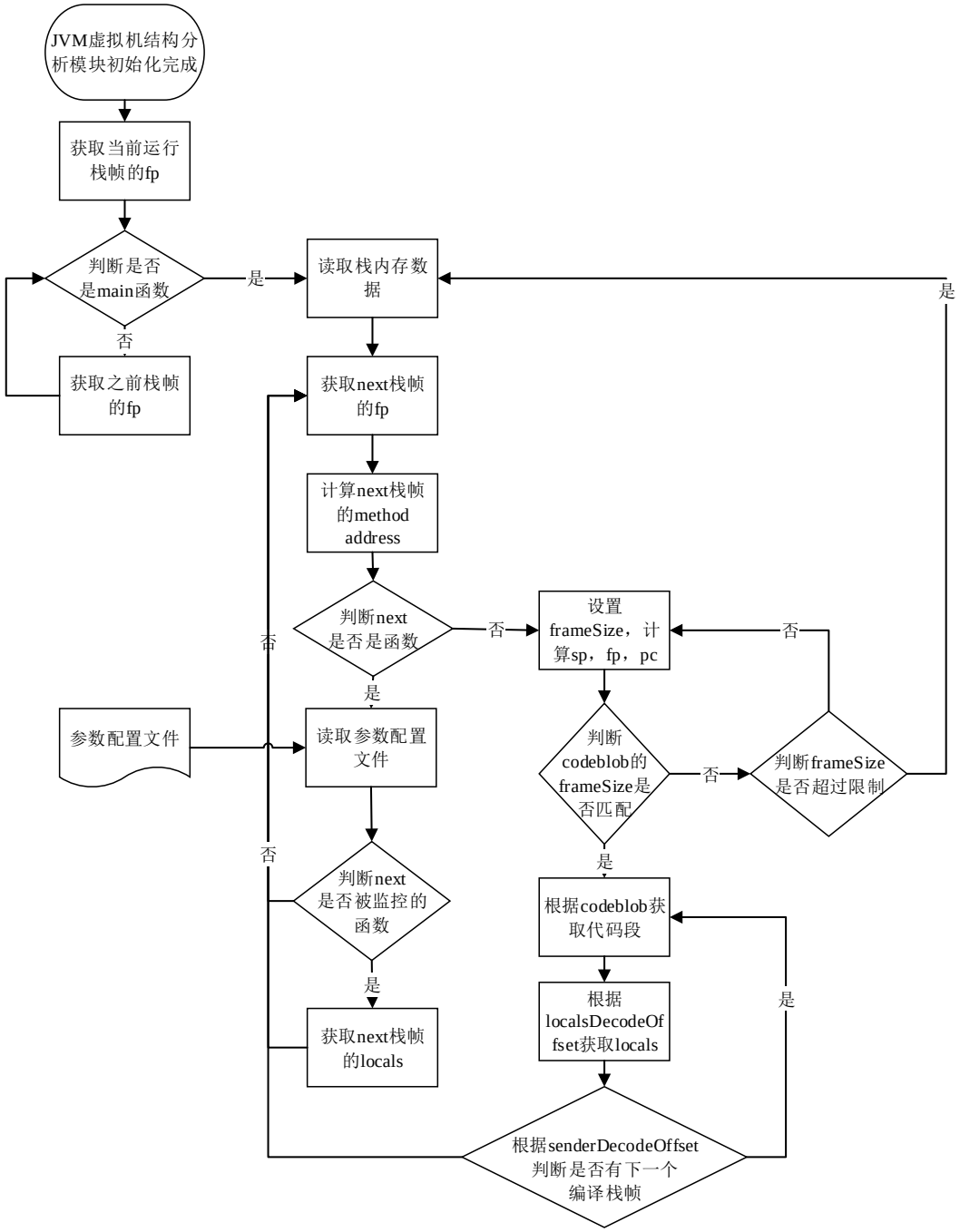

图 4-13 基于编译运行的内存分析流程图

4.3.7 服务容器内存分析

物联网服务运行于 Java 服务容器之中，因此要重构物联网服务语义就需要分析调用物联网服务的服务参数。Java 服务容器采用线程池技术来执行服务。在分析运行 Java 服务容器的 JVM 虚拟机内存结构时，通过 Threads 类型的静态成员变量_thread_list 和 JavaThrread 类型的_next 偏移量循环遍历得到 Java 服务容器线程池中所有的执行线程在内存中的栈地址和栈大小。

在得到栈地址和栈大小之后，通过内存获取模块获取所有执行线程的栈内存。执行线程的栈底函数是 run 函数，因此利用 run 函数栈底不变的特点，通过基于解释运行的内存分析和基于编译运行的内存分析，对执行线程的栈内存进行分析得到栈帧信息。根据 Java 服务容器的结构，对调用物联网服务的函数栈帧信息进行分析，得到调用服务的参数。

4.3.8 服务容器内存分析流程

当对服务容器进行监控时，系统执行服务容器内存分析流程。服务容器内存分析流程图如图 4-14 所示。在服务容器内存分析流程中，系统主要会完成以下工作：

1) 读取 typeName 为 Threads 的静态成员变量_thread_list，得到第一个线程 JavaThread 的内存地址；
2) 通过 JavaThread 非静态成员变量_threadObj 可以得到线程名称，判断当前线程是否是执行线程，如果是，执行 5)，否则执行 3)；
3) 根据 JavaThread 非静态成员变量_next 判断下一个 JavaThread 是否存在，如果是执行 4)，否则执行 9)；
4) 根据 JavaThread 非静态成员变量_next 得到下一个线程 JavaThread 的内存地址，执行 2)；
5) 根据 JavaThread 非静态成员变量 stack_base 和_stack_Size，得到栈地址和大小；
6) 通过内存获取模块，得到栈内存；
7) 通过基于解释运行的内存分析和基于编译运行的内存分析，分析栈内存，得到栈帧信息；
8) 根据栈帧信息和 Java 服务容器结构，得到调用服务的参数，然后执行 3)；
9) 所有线程分析完成，流程结束。

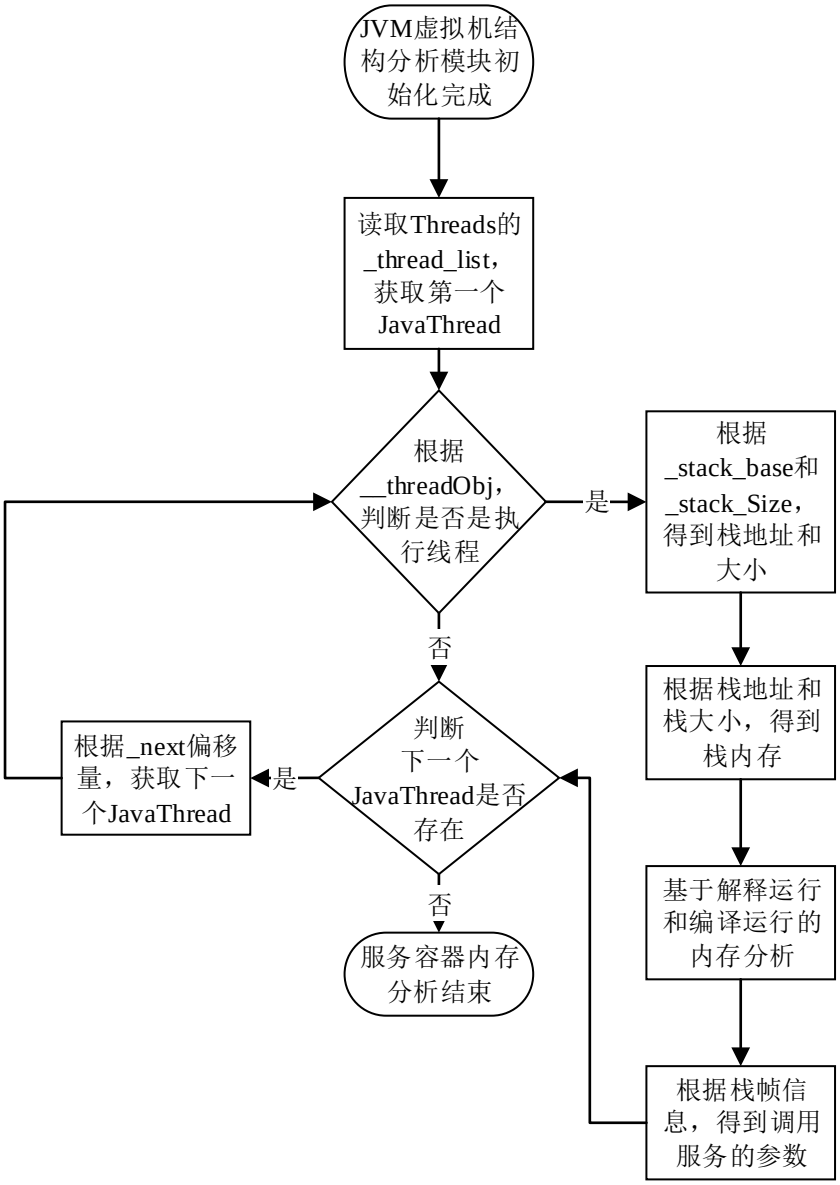

图 4-14 服务容器内存分析流程图

4.4 内存事件

4.4.1 内存事件模块

内存事件模块是本系统语义重构结果的应用方式之一。根据内存事件的需求，内存事件模块分为事件构造子模块和事件发送子模块。图 4-15 是内存事件模块的模块图。

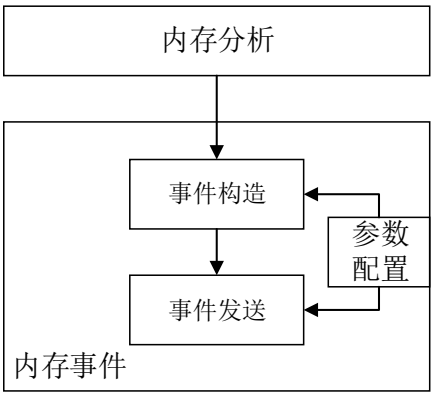

图 4-15 内存事件模块图

根据参数配置文件，事件构造子模块将内存分析的结果按照参数配置文件的要求构造成字符串，包括函数调用、被监控函数的参数和服务参数。事件发送子模块根据参数配置中的发送端口，建立事件发送服务，等待物联网运行时验证系统连接，然后将字符串事件发送给物联网运行时验证系统。由于物联网运行时验证系统对事件的响应速度和处理速度都有很高的要求，因此这两个子模块的设计应当是高性能和高可靠性的。同时，构造好的内存事件为了方便用户查看，还需要将其发送到界面系统来进行展示。

4.4.2 事件构造与发送流程

当内存分析完成生成栈帧信息之后，系统开始事件构造与发送流程。事件构造与发送流程图如图 4-16 所示。在事件构造与发送流程中，系统主要完成以下工作：

1) 读取参数配置文件，获取函数名称，事件格式和发送端口；
2) 根据发送端口，建立事件发送服务；
3) 根据栈帧信息和函数名称，判断栈帧是否有被监控函数，如果是，执行 4)，否则执行 6)；
4) 将栈帧信息按照事件格式进行构造成内存事件；
5) 当物联网运行时验证系统连接之后，将内存事件发送到物联网运行时验证系统；
6) 等待新的栈帧信息；
7) 当有新的栈帧信息时，循环执行 2)。

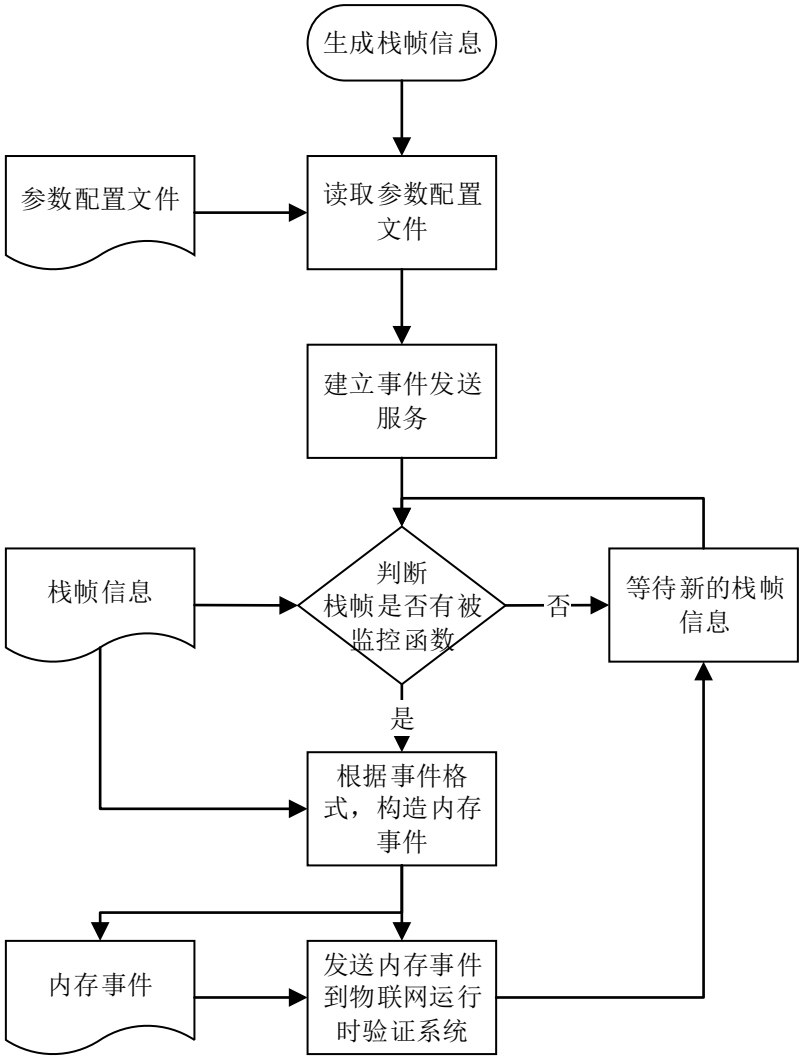

图 4-16 事件构造和发送流程图

4.5 界面系统

4.5.1 界面系统

界面系统用于用户操作系统和系统展示信息。为了方便用户使用本系统，监控本系统的运行状态，根据需求界面系统分为服务参数配置和界面展示两个子模块。界面系统模块图如图 4-17 所示。

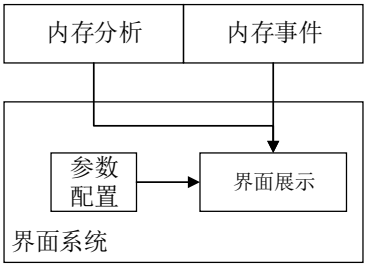

图 4-17 界面系统模块图

用户通过参数配置子模块，配置虚拟机名称，pid，监控的函数名称，函数参数数量和类型，事件格式和发送地址以及界面展示内容。然后系统将这些配置信息保存在文件中。当服务语义重构系统开始运行时，系统从参数配置文件中读取所需要的参数。用户可以根据不同的需求生成多个配置文件，在运行系统之前加载相应的配置文件。参数配置文件内容如表 4-9 所示。

表 4-9 参数配置文件

名称	含义
虚拟机名称	运行物联网服务的虚拟机名称
pid	运行物联网服务的 JVM 的进程 id
函数名称	监控的物联网服务的函数名称
参数数量	监控的物联网服务的函数参数的数量
参数类型	监控的物联网服务的函数所有参数的参数类型
事件格式	发送的内存事件所需要符合的格式
发送端口	与物联网运行时验证系统建立连接的端口
界面展示内容	界面展示子模块中需要展示的内容

界面展示子模块是为了方便用户了解整个服务语义重构系统的运行状态。主要展示内容来自内存分析的栈帧信息和内存事件，实时展示栈帧的函数调用，函数栈内参数和发送给物联网运行时验证系统的内存事件。不仅展示包含目标函数时的函数调用，同样也会展示不包含目标函数时的函数调用。

4.5.2 参数配置和界面展示流程

界面系统模块的设计目的是为了方便用户操作，因此参数配置和界面展示流程不能复杂。通过简单的鼠标点击和填写参数，用户可以完成参数配置和信息展示功能。由于参数配置是由用户输入的，为了保证系统安全性，防止出现错误和异常，在用户输入完成之后，对于用户输入的内容会进行校验。参数配置和界面展示流程图如图 4-18 所示。在参数配置和界面展示流程中，系统主要完成以下工作：

1) 启动界面系统；
2) 选择界面，如果选择参数配置界面，执行 3)，否则执行 7)；
3) 进入参数配置界面；

4) 填写参数，包括虚拟机名称，pid，函数名称等；
5) 校验参数，如果校验通过，执行 6)，否则，重新执行 4)；
6) 保存参数配置，生成参数配置文件，回到 2)；
7) 读取参数配置文件；
8) 根据参数配置文件中界面展示内容参数，显示相应的内容。默认显示栈帧信息和内存事件。

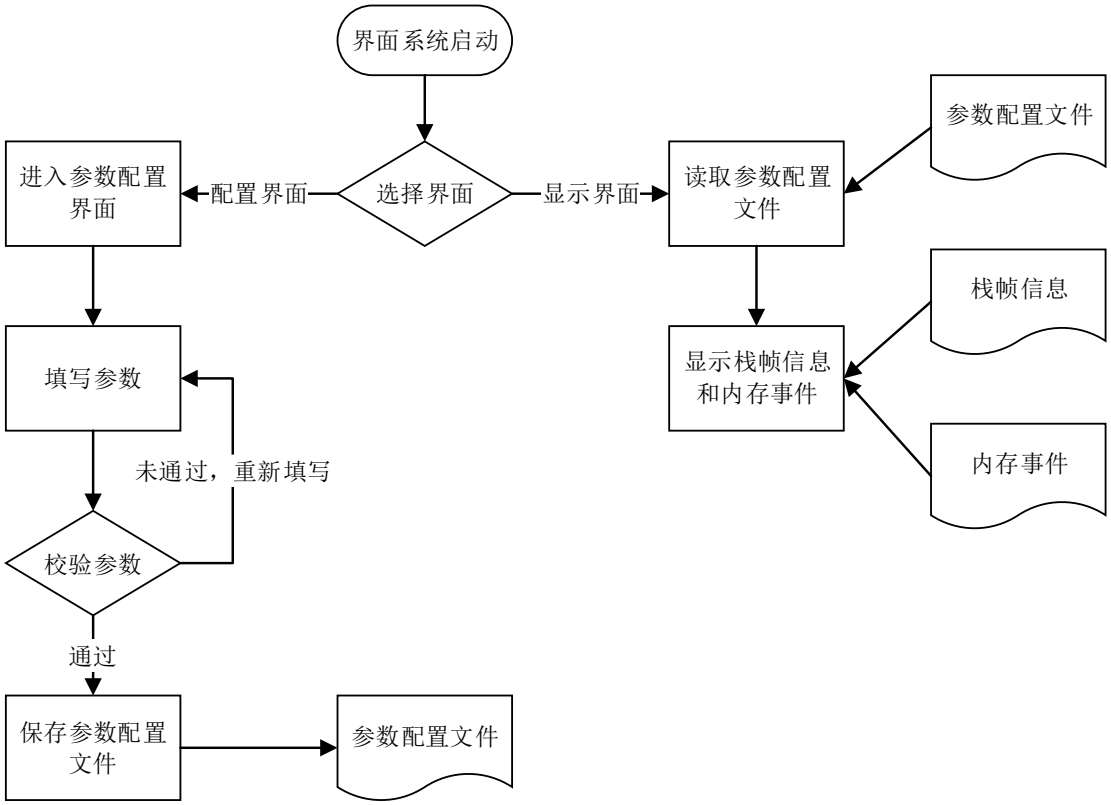

图 4-18 参数配置和界面展示流程图

4.6 本章总结

通过本章内容，详细梳理了服务语义重构系统所有模块的设计和工作流程。本章从整体架构开始，先设计了系统的整体架构，之后从系统的需求和软件工程设计要求出发，分别设计了内存获取模块，内存分析模块，内存时间模块和界面系统模块，详细介绍了内存获取，内存分析的设计原理和工作流程。其中，最为重要的是基于解释运行的内存分析和基于编译的内存分析的原理。然后，本章设计了整个系统所涉及的工作流程，包括内存获取流程，内存分析流程，事件构造和发送流程，参数配置和界面展示流程。读者通过阅读本章，可以对系统的整体和各个模块设计原理和功能有清晰的了解。

第五章 系统实现

第四章完成了本系统的系统设计内容。根据第四章中各个模块的设计原理和工作流程，完成系统实现。本章将详细介绍基于服务容器的服务语义重构系统是如何实现的。

5.1 系统结构

本系统由 Python 和 Java 实现，因此模块功能有两大部分。在 Python 实现的模块中，主模块完成系统的所有流程，Volatility 进程描述符和内存自省技术 pyvmi 用于内存获取模块，内存分析模块一部分 Python 实现，一部分调用 Java 实现。主模块调用界面系统和内存事件模块。在 Java 实现中，虚拟机结构分析和栈内存分析模块通过接口被 Python 调用，同时，它们通过接口调用内存获取模块获取内存数据。如图 5-1 所示。

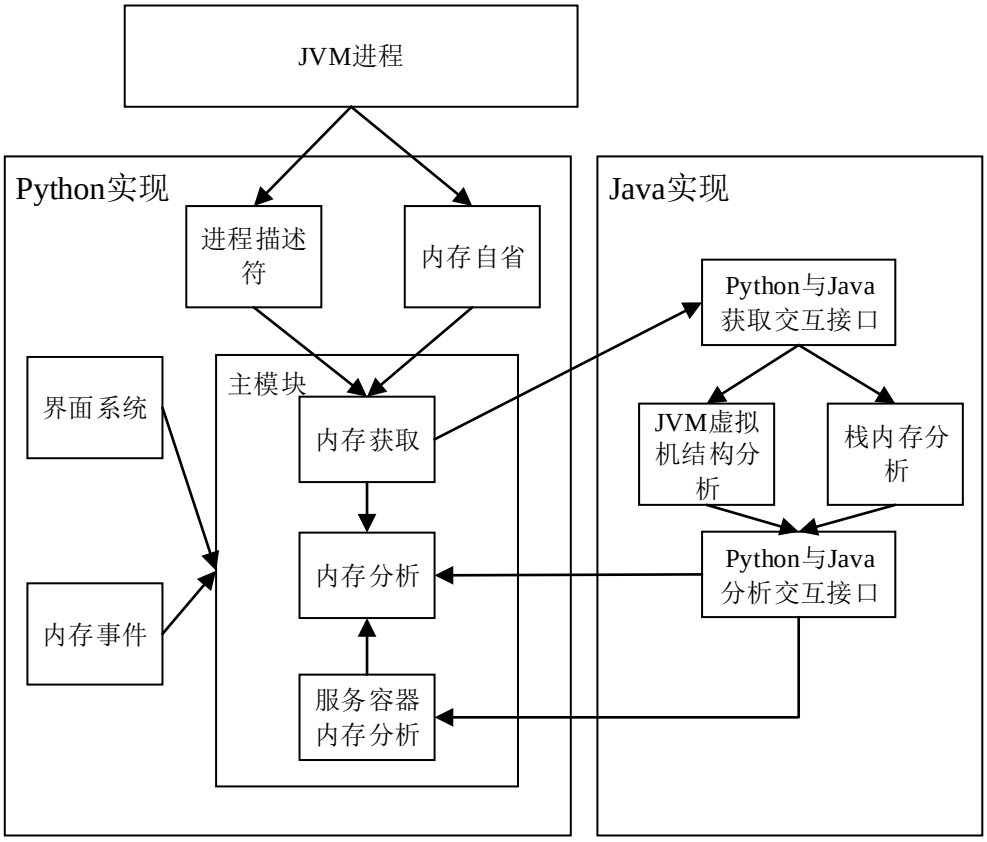

图 5-1 系统结构图

本系统由 Python 类和 Java 类组成。Python 类有 task_struct、linux_java、Frame、Conf 和 Event。Java 类有 PyDump、PyTool、PyBugSpotAgent、HotSpotTypeDataBase、LinuxDebuggerLocal、ReadResult、Threads、JavaThread、

JavaVFrame、Frame、PythonMethodInterface、ScopeDesc。类之间的关系如图 5-2 所示。

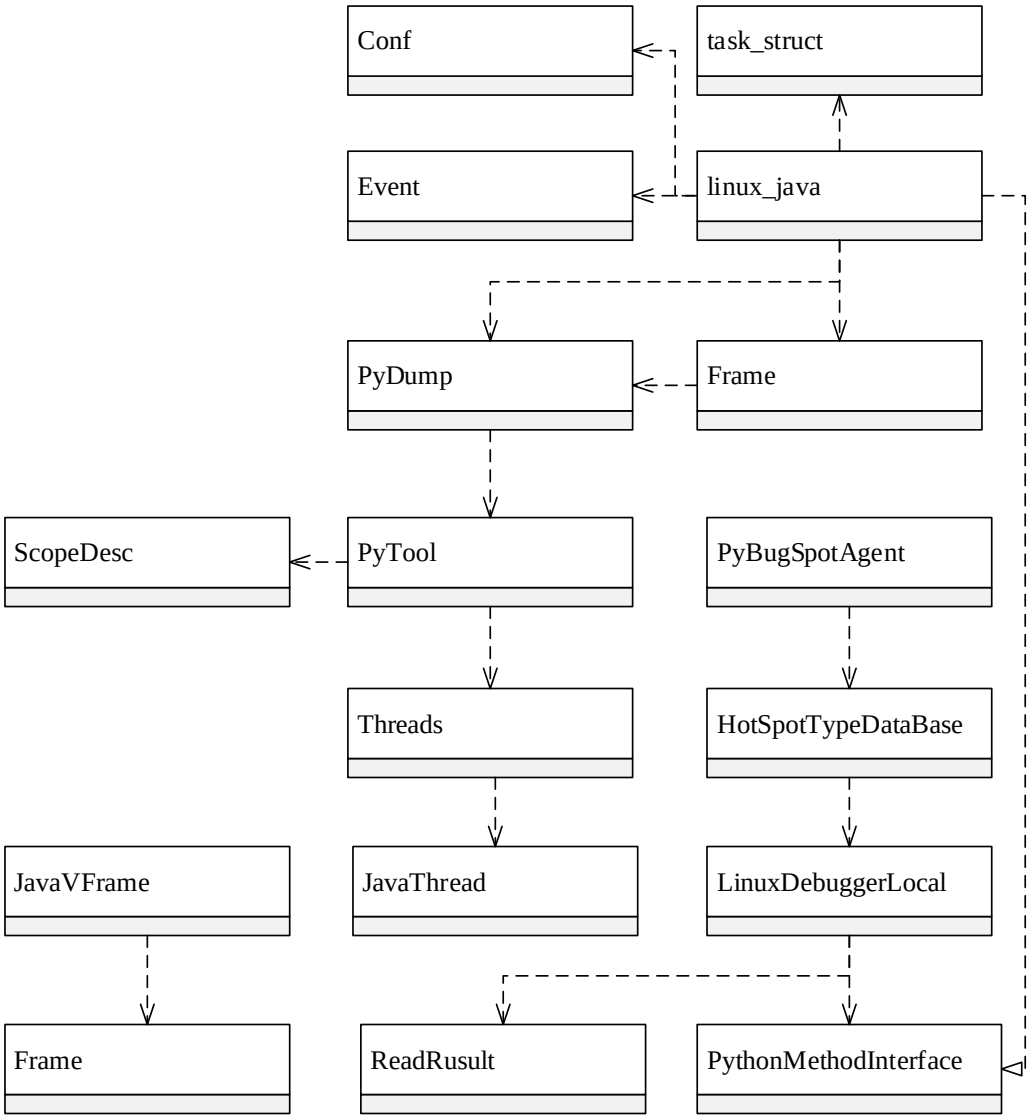

图 5-2 系统类图

5.2 内存获取

内存获取模块分为共享库获取、子线程获取、栈内存获取和地址内存获取四个子模块。本模块使用 Python 实现。共享库获取和子线程获取子模块调用进程描述符 task_struct 实现，地址内存获取和栈内存获取子模块调用进程描述符 task_struct 和内存自省插件 pyvmi 实现。如图 5-3 所示。

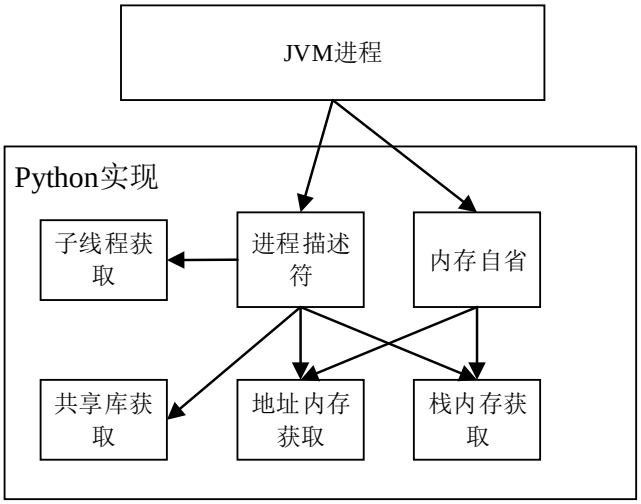

图 5-3 内存获取结构图

5.2.1 内存获取实现

内存获取是由类 linux_java 调用类 task_struct 来实现的。类 task_struct 来表示进程描述符。其中，get_proc_maps()方法用于获取共享库函数在进程中虚拟地址映射，threads()方法用于获取进程子线程，get_process_address_space()方法用于获取进程地址空间。

在类 linux_java 中，本地变量 PyDump 是调用 Java 方法的接口，列表 libnames 保存共享库名称，列表 libbases 保存共享库的起始地址，列表 threadsId 保存子线程 tid。getLibName()方法用于获取共享库名称，getLibBase()方法用于获取共享库起始地址，getThreadsId()方法用于获取子线程 tid，read()方法用于按地址读取内存数据的字节数组，readMemory()方法用于获取地址与内存数据字典映射，lookUpByName()和 readBytesFromProcess()方法是分析阶段 Java 程序调用 Python 方法的接口。如图 5-4 所示。

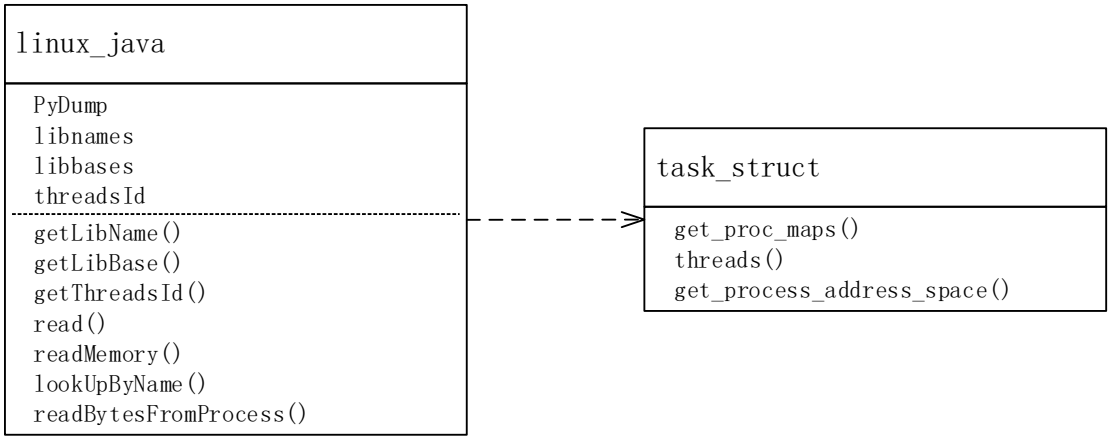

图 5-4 linux_java 类图

共享库获取子模块 getLibName()和 getLibBase()方法利用 task_struct 的

get_proc_maps()方法，来获取共享库信息。程序流程图如图 5-5 所示。声明列表 libnames 和 libbases，用于保存共享库和起始地址。调用 task_struct 的 get_proc_maps()方法得到 vma 列表。循环 vma，根据 vma 的 vm_name()方法得到共享库名称，如果名称为空则跳过。根据 vma 的 vm_start 变量，得到起始地址，然后将共享库名称和起始地址保存。getLibName()和 getLibBase()返回列表 libnames 和 libbases。

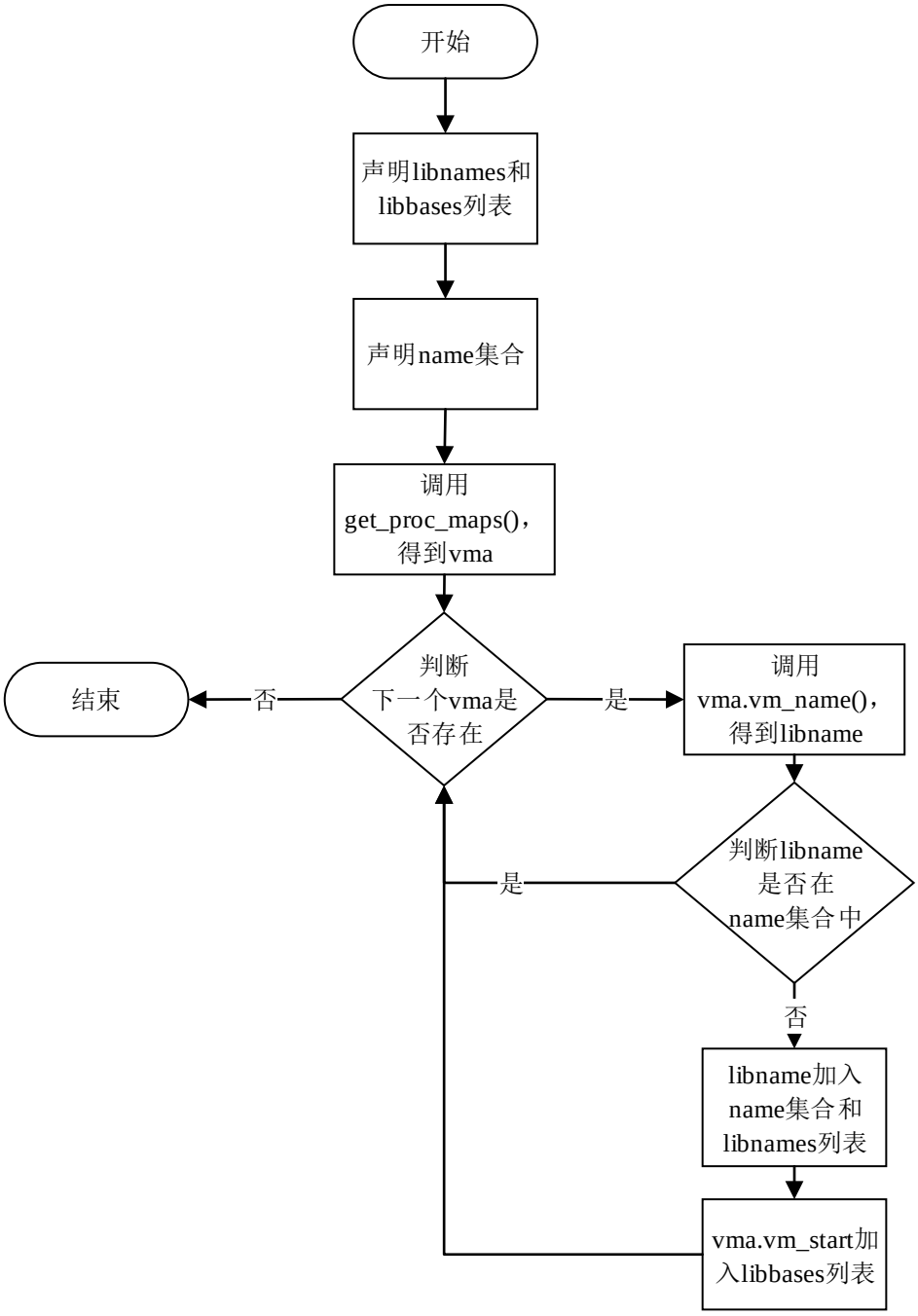

图 5-5 共享库获取程序流程图

子线程模块 getThreadsId()利用 task_struct 的 threads()方法，来获取进程的子线程。程序流程图如图 5-6 所示。声明列表 threadsId 来保存子线程 tid。调用

task_struct 的 threads()方法，得到 threads 列表，然后遍历 threads，根据 thread 的 pid 变量来得到子线程 tid，保存到 threadsId 中。getThreadsId()返回 threadsId 列表。

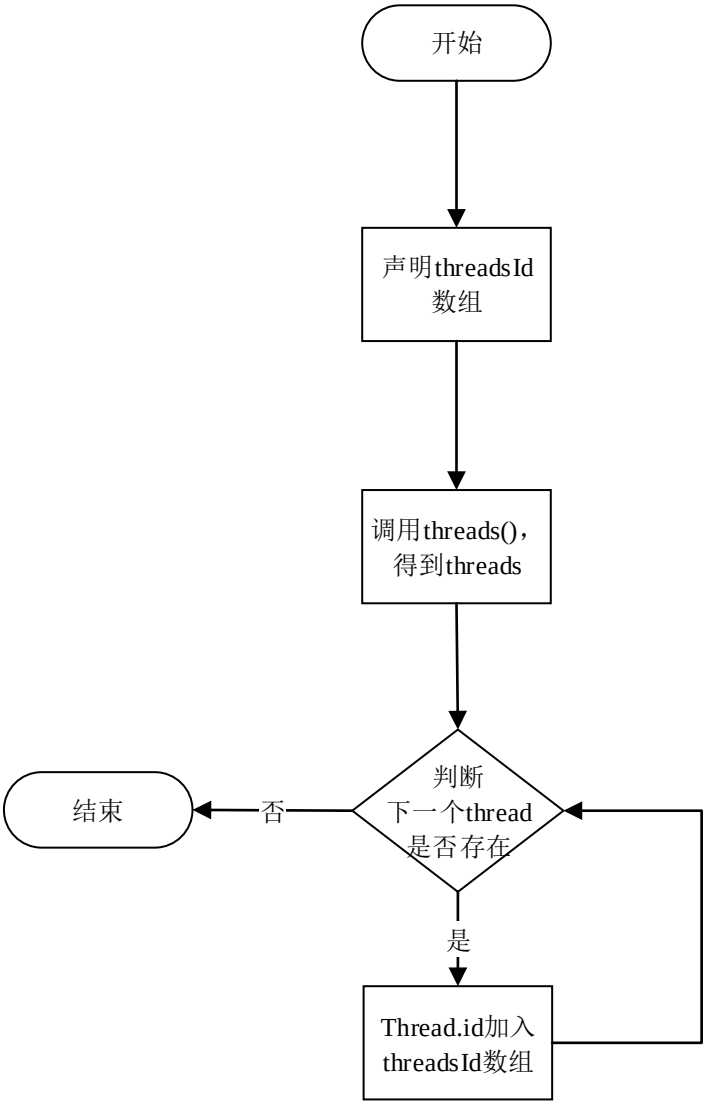

图 5-6 子线程获取程序流程图

地址内存获取子模块 read()方法利用 task_struct 的 get_process_address_space()方法得到进程地址空间，利用 pyvmi 和进程地址空间来读取内存。程序流程图如图 5-7 所示。read()方法输入参数进程描述符 task_space，地址和读取的内存大小 num，返回内存 bytes 数组。声明列表 ans 用于保存结果，调用 task_struct 的 get_process_address_space()方法得到进程地址空间，根据进程地址空间和起始地址，读取 8 字节的内存，将结果按照字节单位进行转换，保存到 ans 列表中，循环下一个 8 字节的地址，直到读取了 num 大小的内存。

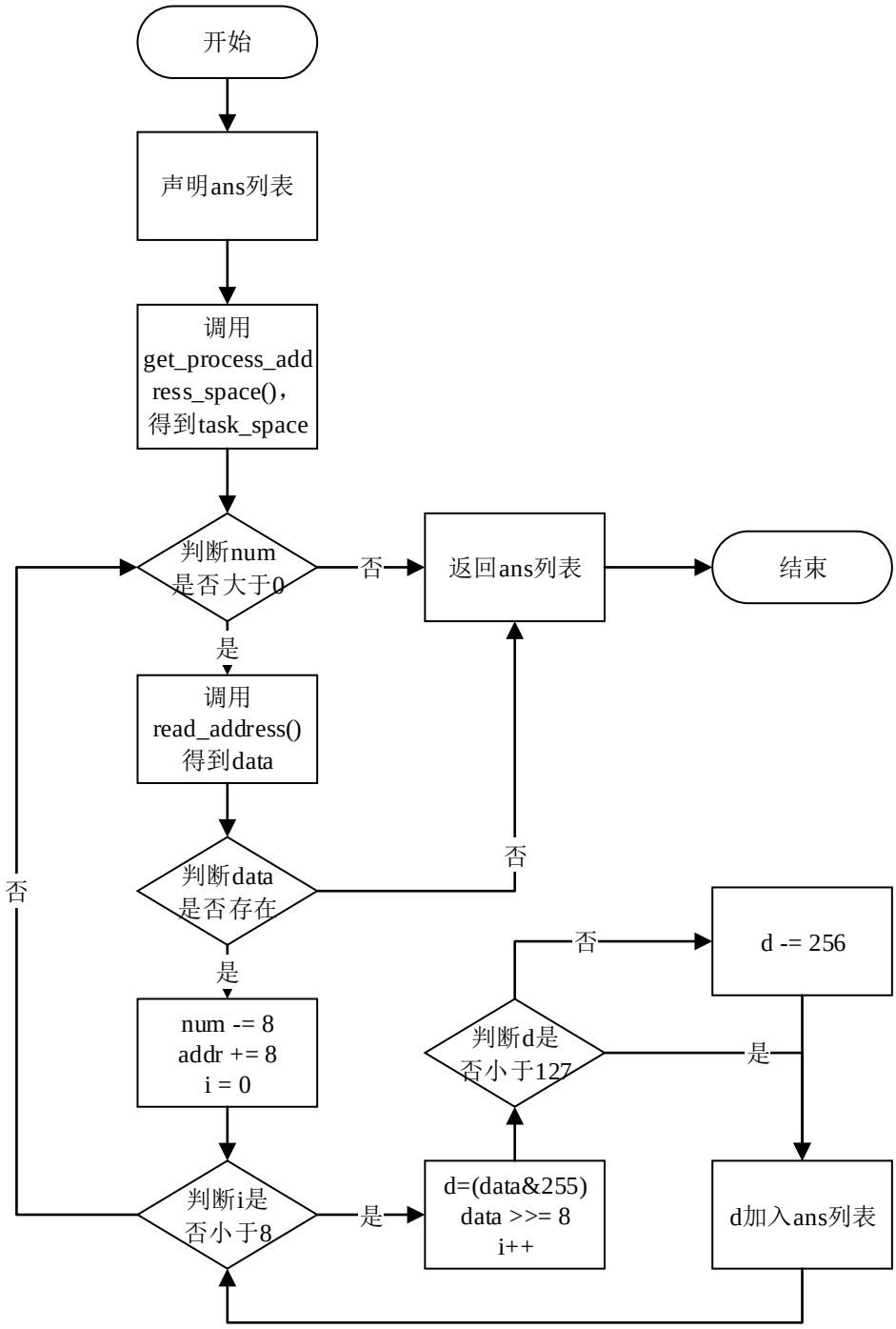

图 5-7 地址内存获取程序流程图

栈内存获取子模块 readMemory() 方法同样利用 task_struct 的 get_process_address_space()方法得到进程地址空间，利用 pyvmi 和进程地址空间来读取内存。不同的是，它还需要对地址和地址中内存进行映射，并返回映射。程序流程图如图 5-8 所示。声明字典 res1 和 res2 用于保存地址到内存的映射和内存到地址的映射。调用 task_struct 的 get_process_address_space()方法得到进程地址空间，根据进程地址空间和起始地址，读取 numBytes 大小的内存字符串结果，通过 Python 的 struct.unpack()方法转换出内存的整型结果。在转换过程中，

以 8 个地址为单位，对内存和内存地址进行映射，地址到内存的映射保存在 res1 中，内存到地址的映射保存在 res2 中。readMemory()方法输入地址和内存大小，返回地址到数据和数据到地址的字典映射。

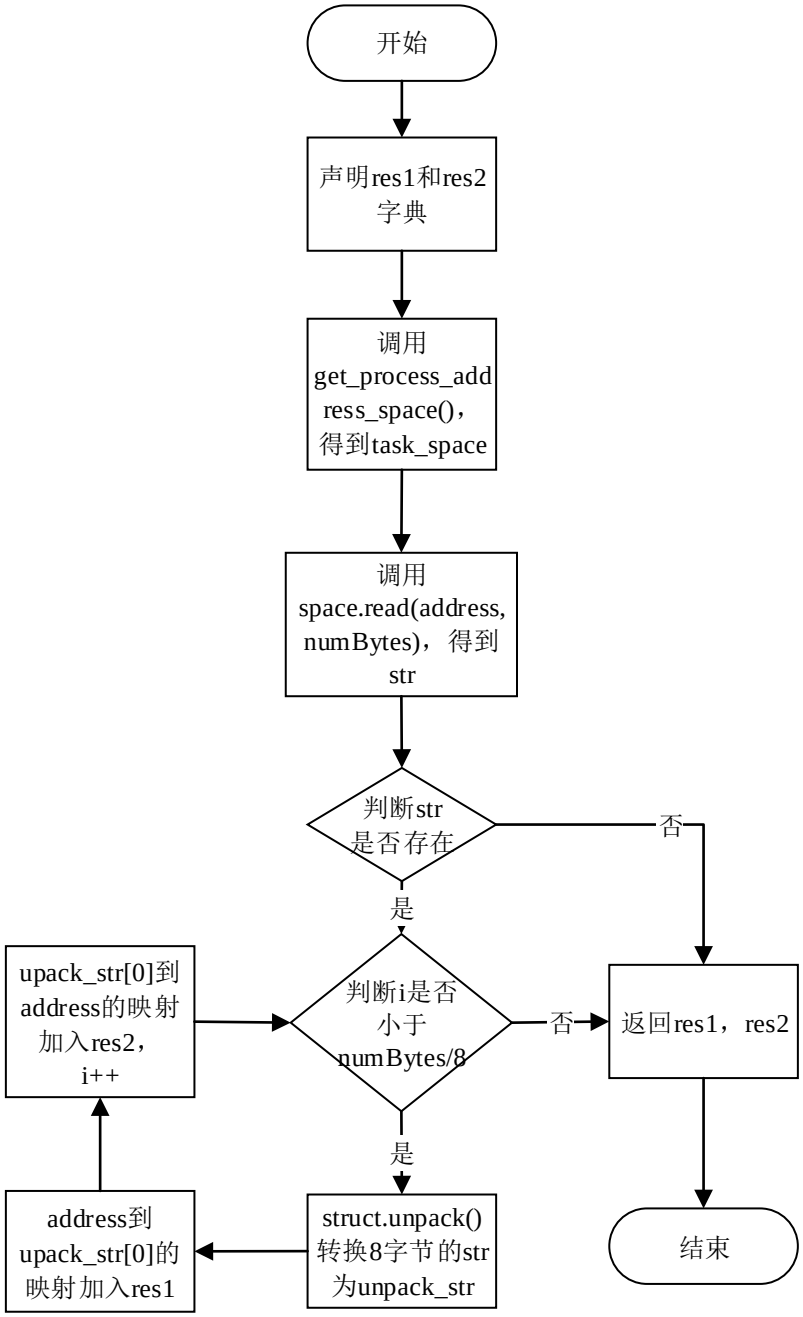

图 5-8 栈内存获取程序流程图

内存获取模块因为系统应用了 Volatility 框架，所以使用 Python 编写。而为了方便分析 Java 内存，内存分析模块采用了 Java 进行编写。由于内存分析模块是基于内存获取模块的结果进行分析，系统实现了 Java 方法和 Python 方法互相调用。

在本系统中，采用了开源的 jpype 模块实现 Python 调用 Java。通过 Python

代码实现了 Java 中的 getThreadsId()， getLibName()， getLibBase()，lookUpByName()，readBytesFromProcess()方法。如表 5-1 所示。

表 5-1 Python 调用 Java 代码实现

```
PyDump = jpype.JPackage('sun.tools.python').PyDump
self.PyDump = PyDump
method_dict = {
        'getThreadsId': self.getThreadsId,
        'getLibName': self.getLibName,
        'getLibBase': self.getLibBase,
        'lookUpByName': self.lookUpByName,
        'readBytesFromProcess': self.readBytesFromProcess
        }
jp = jpype.JProxy('sun.jvm.hotspot.debugger.linux.PythonMethodInterface', dict =
method_dict)

package sun.jvm.hotspot.debugger.linux;

public interface PythonMethodInterface {
      long[] getThreadsId();
      String[] getLibName();
      long[] getLibBase();
      long lookUpByName(String objectName, String symbol);
      byte[] readBytesFromProcess(long address, long numBytes);
}
```

getThreadsId()方法用于返回初始化阶段获取的 threadsId 数组。getLibName()用于返回初始化阶段获取的 libnames 数组。getLibBase()用于返回初始化阶段获取的 libbases 数组。lookUpByName()用于根据符号名称在共享库中查找符号在内存的地址。readBytesFromProcess()利用地址内存获取模块，根据地址和大小读取内存返回 byte 类型数组的内存数据。

lookUpByName()方法根据库名称 objectName 和符号名称 symbol 在所有的共享库中进行遍历，调用 readelf.read_sym_offset()方法根据库名称得到相应库的符号到偏移的字典映射 d，然后根据符号名称得到偏移，之后与库起始地址相加

后返回。如果没有找到，返回 0。程序流程图如图 5-9 所示。

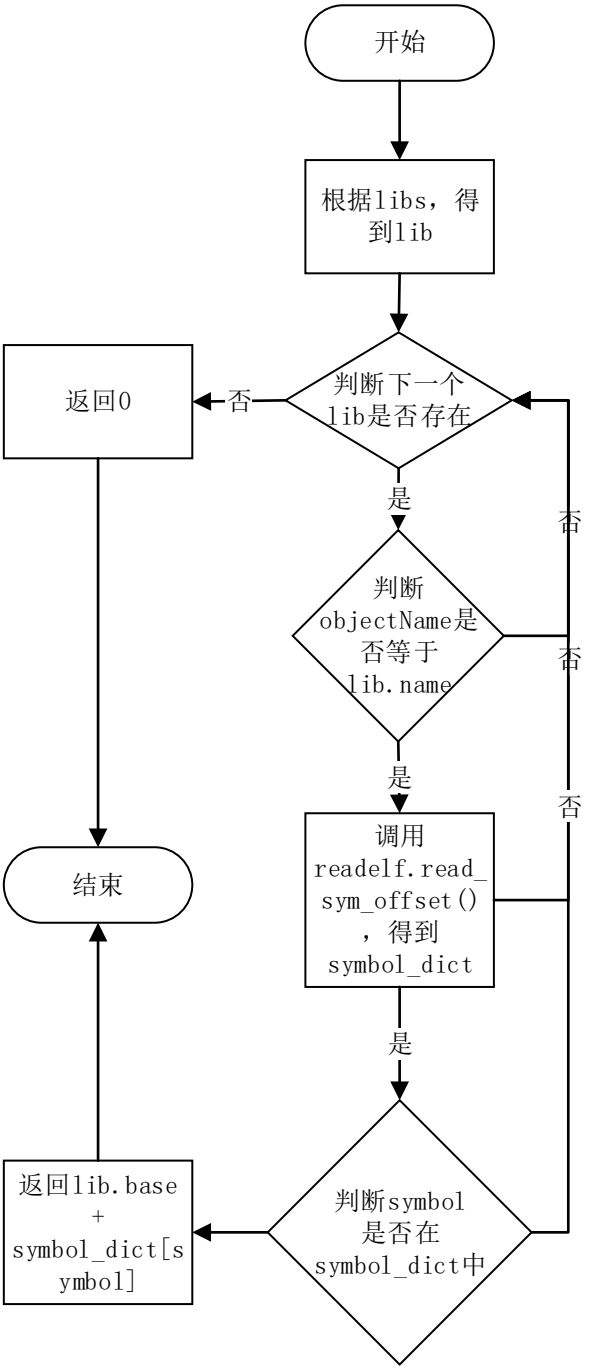

图 5-9 lookUpByName()程序流程图

5.2.2 内存获取序列图

内存获取模块的初始化过程如图 5-10 所示。linux_java 调用 get_proc_maps() 返回 libs 共享库，调用 threads()返回 threadsId，调用 get_process_address_space() 返回进程地址空间 task_space。

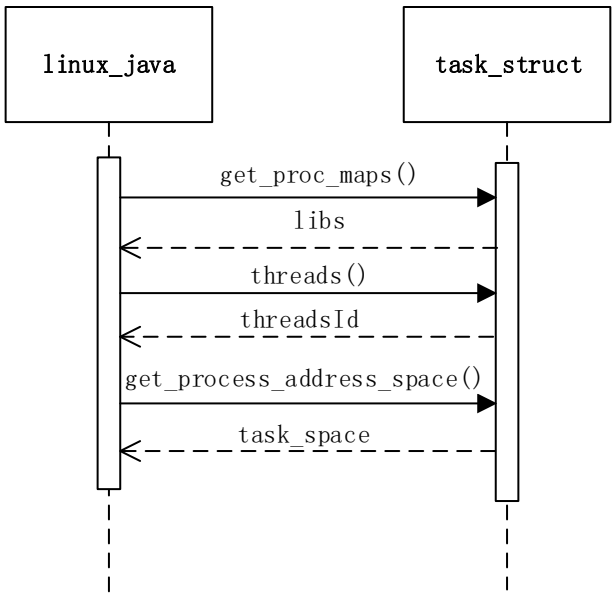

图 5-10 内存获取模块序列图

5.3 内存分析

内存分析模块分为 JVM 虚拟机内存结构分析、栈内存分析和服务容器内存分析三个子模块。本模块使用 Java 和 Python 混合编程的方式实现。JVM 虚拟机结构分析子模块通过 PyDump 接口调用 Java 实现的 JVM 虚拟机结构分析，栈内存分析子模块通过 PyDump 接口调用 Java 实现的栈内存分析，服务容器内存分析基于内存获取和栈内存分析子模块实现。Java 实现的 JVM 虚拟机结构分析和栈内存分析通过 PythonMethodInterface 类的实例 pmi 调用 Python 实现的内存获取模块。如图 5-11 所示。

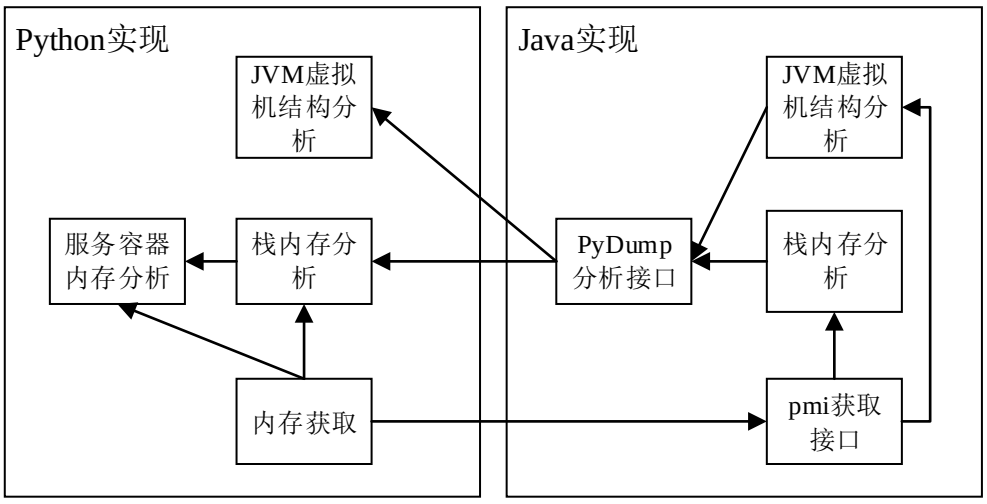

图 5-11 内存分析结构图

5.3.1 JVM 虚拟机内存结构分析实现

JVM 虚拟机内存结构分析子模块使用 Java 语言进行实现。涉及到 Java 与 Python 的调用，已经在上节表述。初始化阶段，使用了 PyBugSpotAgent 类和 PyTool 类来进行初始化，使用 PyDump 类与 Python 进行对接。在 PyDump 类中，PythonMethodInterface 类的 pmi 变量提供接口用于调用 Python 方法，initVM()方法用于初始化 JVM，initJavaFirstFPAddress()方法用于得到 main 函数的地址，getMethodName()方法用于获取函数名称，getNextCompliedSP()用于获取下一个编译型栈帧的 sp，getCompliedName()用于获得相同 sp 的多个编译型栈帧的名称数组，getCompliedLocals 用于获得相同 sp 的多个编译型栈帧的参数数组。在 PyTool 类中的方法是对 PyDump 中方法的实现。在 PyBugSpot 类中调用 attach()进行 JVM 虚拟机结构分析。

在 PyBugSpotAgent 类进行 JVM 虚拟机结构分析时，使用了 HotSpotTypeDataBase 类用于遍历和保存 JVM 的数据类型，LinuxDebuggerLocal 类用于调用 PythonMethodInterface 类提供的内存获取方法，ReadResult 类用于保存和转换内存数据。

在 HotSpotTypeDataBase 类中，利用构造函数调用 readVMTypes()、readVMStructs()、readVMIntConstants()、readVMLongConstants()方法读取 JVM 的类型、类型属性、整型常量和长整型常量进行初始化。在读取过程中 lookupInProcess()方法调用 LinuxDebuggerLocal 类的 lookUpByName()方法得到符号在内存中的地址。

在 LinuxDebuggerLocal 类中，init()和 attach()方法用于初始化。lookUpByName()，readBytesFromProcess()均是调用 Python 方法实现，调用 lookUpByName()方法得到符号在内存中的地址，调用 readBytesFromProcess()得到地址中存储的内存字节数组。

在 ReadResult 类中，getData()方法直接获取字节数组，其他方法根据需要的类型，将字节数组转换为相应类型的数据。如图 5-12 所示。

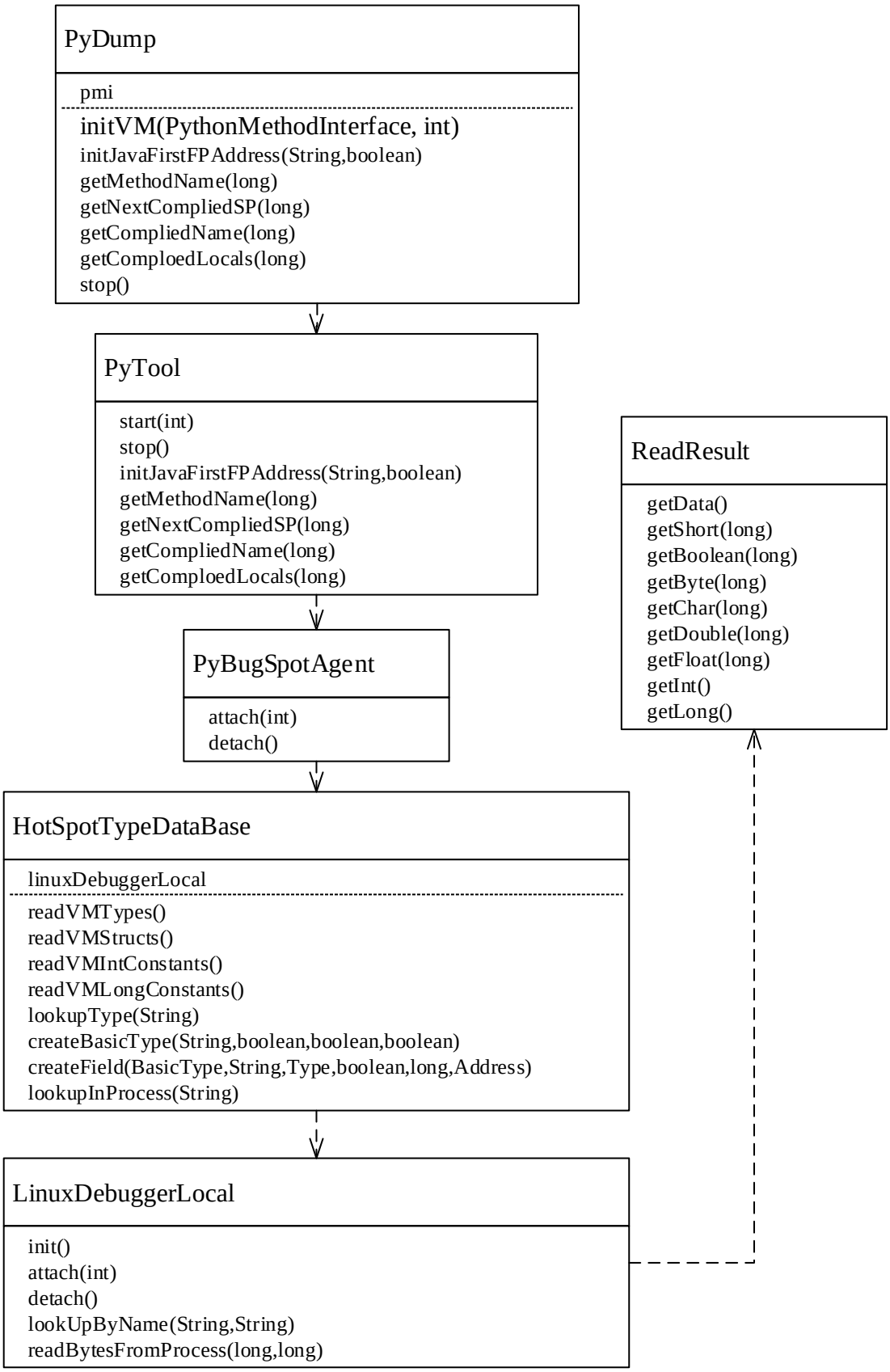

图 5-12 JVM 内存结构分析类图

在初始化阶段，最关键的是获取 JVM 的数据类型结构。由于 readVMTypes()，readVMStructs()， readVMIntConstants()， readVMLongConstants()方法类似，这

里以 readVMTypes()的代码为例。程序流程图如图 5-13 所示。调用 lookupInProcess()方法通过共享库和符号得到 VMTypes 入口地址的地址，根据属性相应的六个偏移量，获取当前 VMTypes 的六个属性和下一个 VMTypes 的偏移量。调用 createType()方法生成 BasicType 类型表示 JVM 类型。

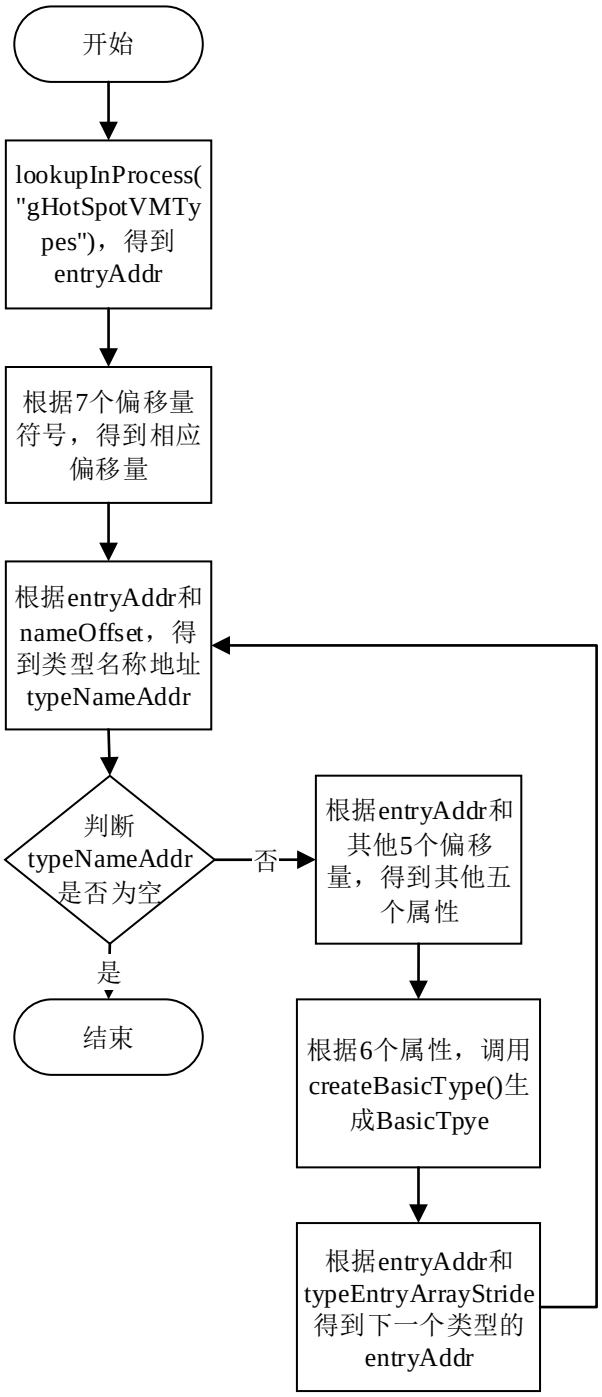

图 5-13 readVMTypes()程序流程图

BasicType 类是 Java 代码中用来表示 JVM 数据类型。它提供了一些方法用来读取数据类型的相关属性。Field 类是用来表示 JVM 数据类型的属性。它提供了一些方法用来读取属性的相关信息。

5.3.2 JVM 虚拟机内存结构分析序列图

在内存获取模块初始化完成之后，Python 通过 PyDump 类调用 initVM()方法开始 JVM 虚拟机内存结构分析子模块的初始化。initVM()调用 PyTool 类的 start()方法，start()方法调用 PyBugSpotAgent 类的 attach()方法是实现 JVM 内存结构初始化。attach()方法生成 HotSpotTypeDataBase 类遍历得到 JVM 的数据类型和常量。在 HotSpotTypeDataBase 类的构造函数中，调用 LinuxDebuggerLocal 类的 lookUpByName()方法得到符号在内存中地址，还调用 LinuxDebuggerLocal readBytesFromProcess()方法得到字节数组，同时利用 ReadResult 类转换字节数组为地址或值。如图 5-14 所示。

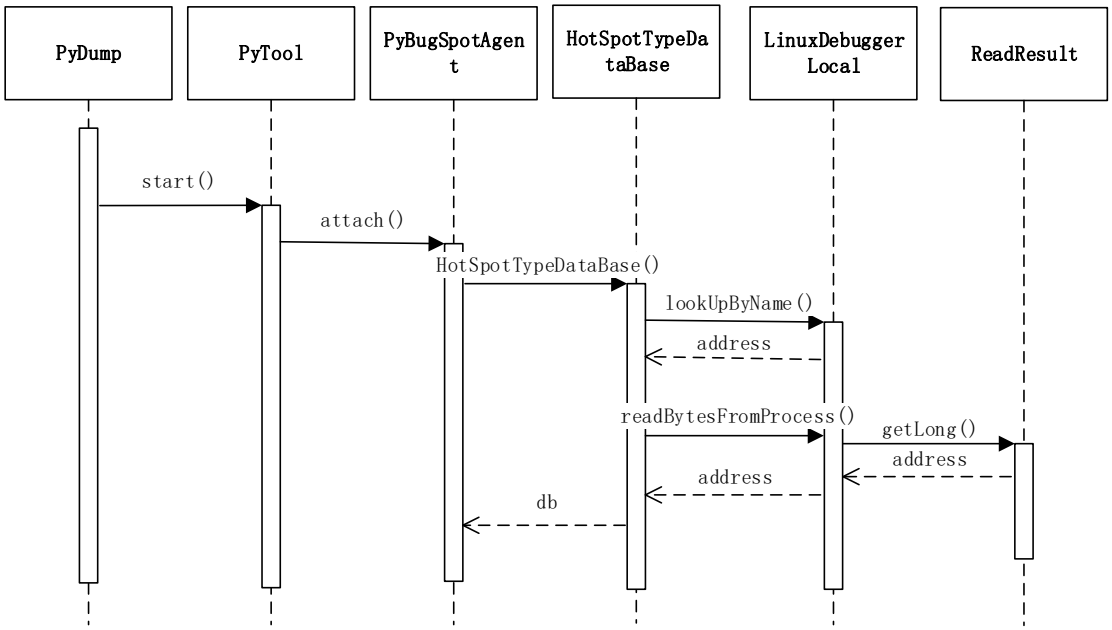

图 5-14 JVM 虚拟机内存结构分析序列图

5.3.3 栈内存分析实现

在 JVM 虚拟机内存结构分析子模块初始化之后，HotSpotTypeDataBase 类变量 db 用于获取 JVM 中相关变量、常量和类型，LinuxDebuggerLocal 类变量 debugger 用于调用内存获取模块来获取内存数据。按照流程调用 PyDump 的 initJavaFirstFPAddress()方法初始化。其中，Threads，JavaThread，JavaVFrame，Frame 类来表示相关的信息。

Threads 类在初始化时，读取 typeName 为 Threads 的静态成员变量_thread_list，通过 first()方法得到第一个线程 JavaThread 的内存地址。JavaThread 在初始化时，会读取 typeName 为 JavaThread 的非静态成员变量_next，通过 next()方法可以获取下一个线程的内存地址。同时，JavaThread 还提供了 getLastJavaVFrameDbg()方法通过_last_Java_fp，_last_Java_sp 和_last_Java_pc 生成当前运行的 JavaVFrame

类型的栈帧，而 JavaVFrame 类的 javaSender()方法通过 Frame 类的 sender()方法根据栈帧类型和内存数据获取下一个栈帧。如表 5-2 所示。

表 5-2 Threads，JavaThread，JavaVFrame，Frame 类说明表

类名	说明
Threads	first()方法用于获取第一个线程
JavaThread	next()方法用于获取下一个线程的内存地址，getLastJavaVFrameDbg()方法用于获取当前运行的 JavaVFrame 类型的栈帧
JavaVFrame	javaSender()方法获取下一个 JavaVFrame 栈帧
Frame	getSP()，getFP()，getPC()方法可以得到 sp、fp、pc；sender()方法用于获取下一个 Frame 的 sp，fp，pc

在 initJavaFirstFPAddress()初始化之后，系统得到了 main 函数或者 run 函数的 fp。调用内存获取模块的 readMemory()方法来读取内存。内存数据保存在字典 dict1 和 dict2 中，dict1 是地址到内存数据的映射字典，dict2 是内存数据到地址的映射字典。这里使用 Python 编写的 Frame 类来分析栈帧。如表 5-3 所示。

表 5-3 Python 的 Frame 类说明表

方法名	说明
getNextFrame()	根据当前栈帧的 fp，获取下一个栈帧
getName()	在解释型中，根据当前栈帧的 fp，获取当前栈帧的函数名称；在编译型中，根据当前栈帧的 Scope 获取函数名称
getLocals()	在解释型中，根据当前栈帧的 fp，获取当前栈帧的参数；在编译型中，根据当前栈帧 unextendedSP 和 Scope 来获取当前栈帧的参数
getVal()	根据内存数据和数据类型，重构出参数的值

getNextFrame()方法根据解释运行和编译运行的原理进行处理，程序流程图如图 5-15 所示。如果当前栈帧是解释型的，先根据 fp 和解释型结构计算下一个栈帧，如果无法得到下一个栈帧，再根据编译型进行计算。编译型的计算由 PyDump 类的 getNextCompliedSP()方法实现。如果当前栈帧是编译型的，根据 sp 和编译型结构计算下一个栈帧。

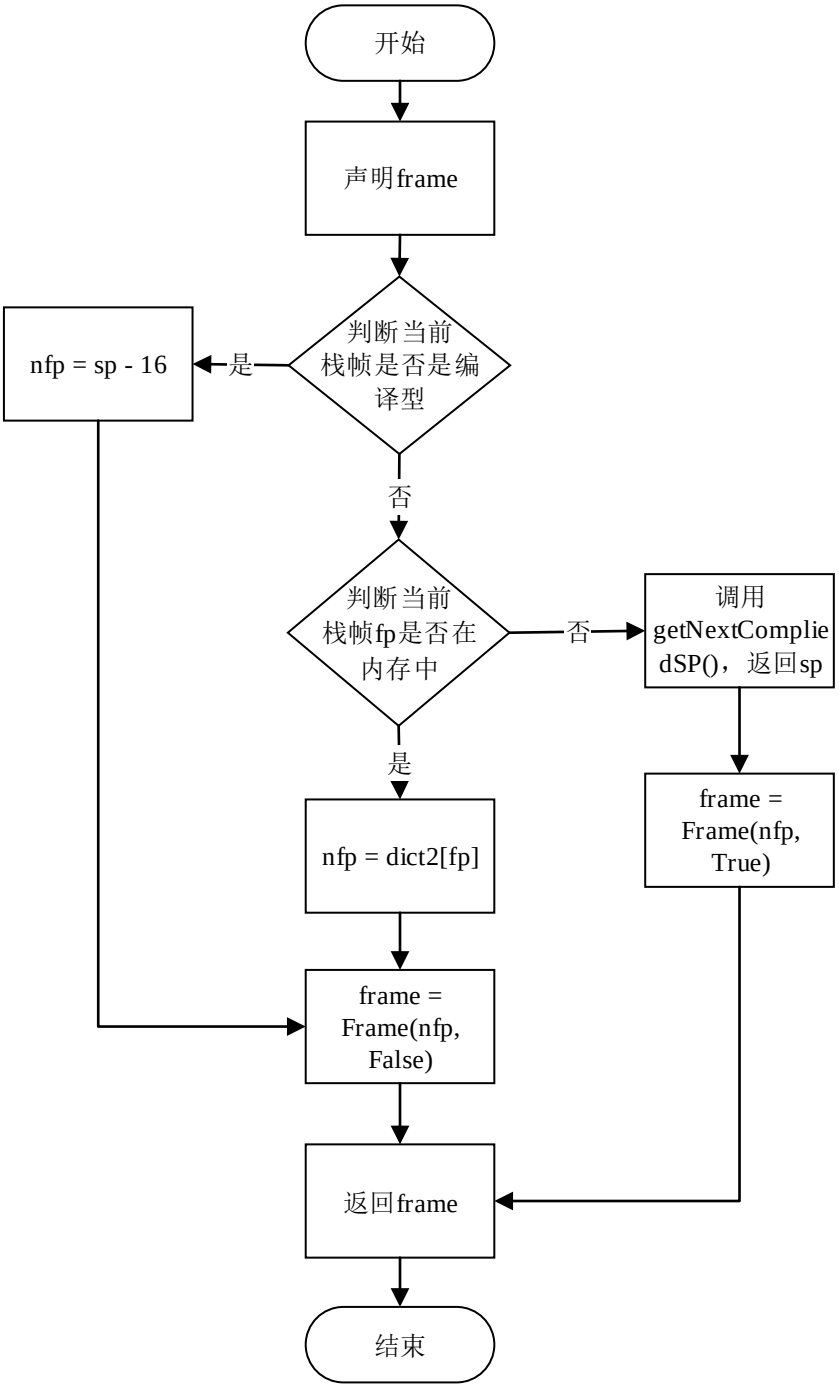

图 5-15 getNextFrame()程序流程图

getNextCompliedSP()方法用于搜索下一个编译型栈帧。因为下一个编译型栈帧的大小是未知的，所以需要匹配栈帧的大小。程序流程图如图 5-16 所示。设置 frameSize 初始值，根据 frameSize 和当前栈帧的 sp 求出下一个栈帧的 unextendedSP。先假设下一个栈帧不是最后一个栈帧，那么下一个的下一栈帧一定是解释型栈帧。根据 unextendedSP 和解释型结构的特点，求出下一栈帧的 pc，根据 pc 得到代码段 scopeDesc，调用 check()方法判断 frameSize 是否匹配。如果匹配，求出 sp 并返回。不匹配，再假设下一栈帧是最后一个栈帧，那么根据

unextendedSP 和编译型栈帧求出 pc，根据根据 pc 得到代码段 scopeDesc，调用 check()方法判断 frameSize 是否匹配。如果匹配，求出 sp 并返回。不匹配，更新 frameSize。

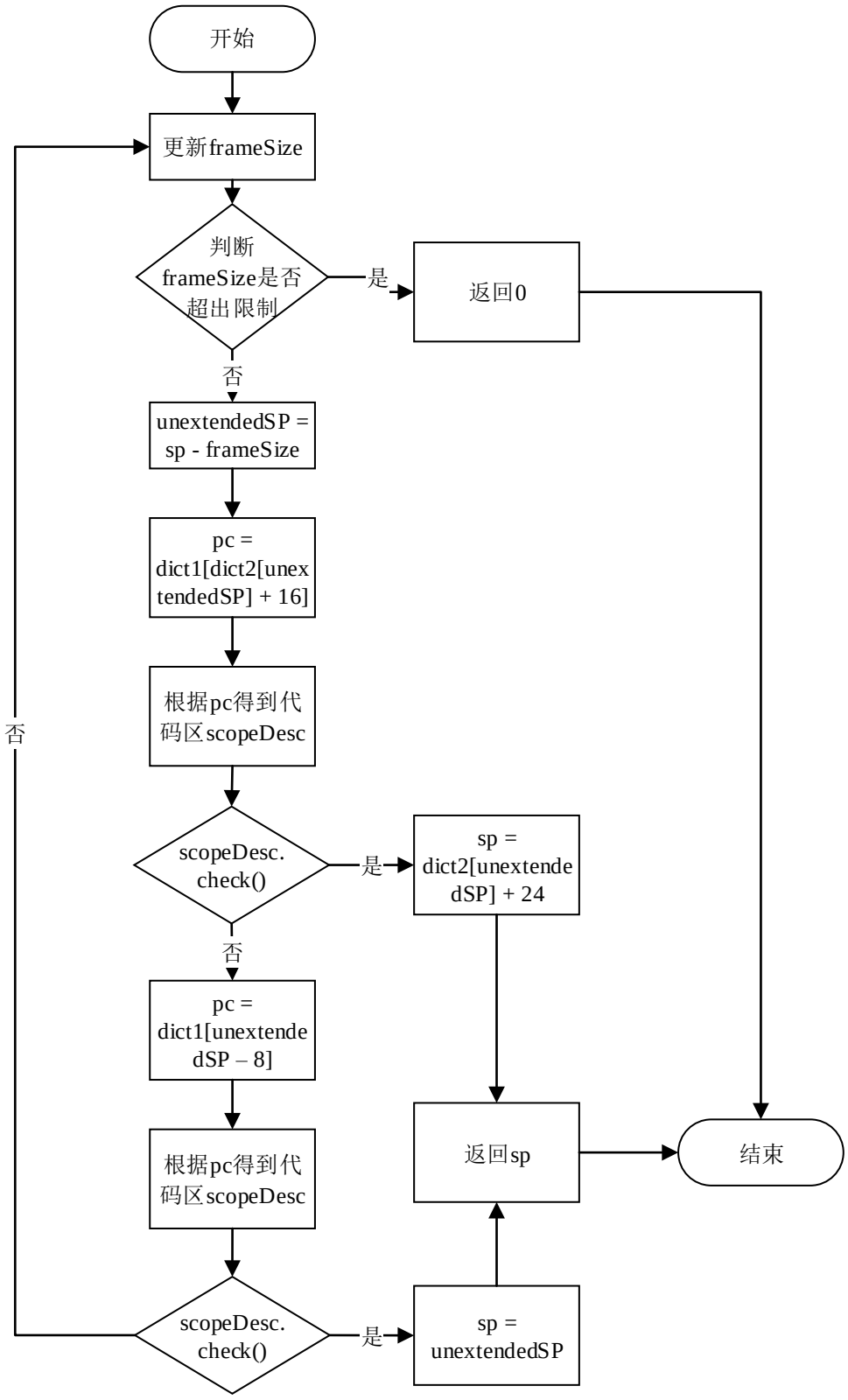

图 5-16 getNextCompliedSP ()程序流程图

getName()方法根据当前栈帧的类型进行处理，程序流程图如图 5-17 所示。如果当前栈帧是解释型的，调用 PyDump 的 getNameByAddress()方法得到函数名称。如果当前栈帧是编译型的，调用 PyDump 的 getCompieldName()方法可以得到相同 sp 的多个编译型栈帧的名称数组。

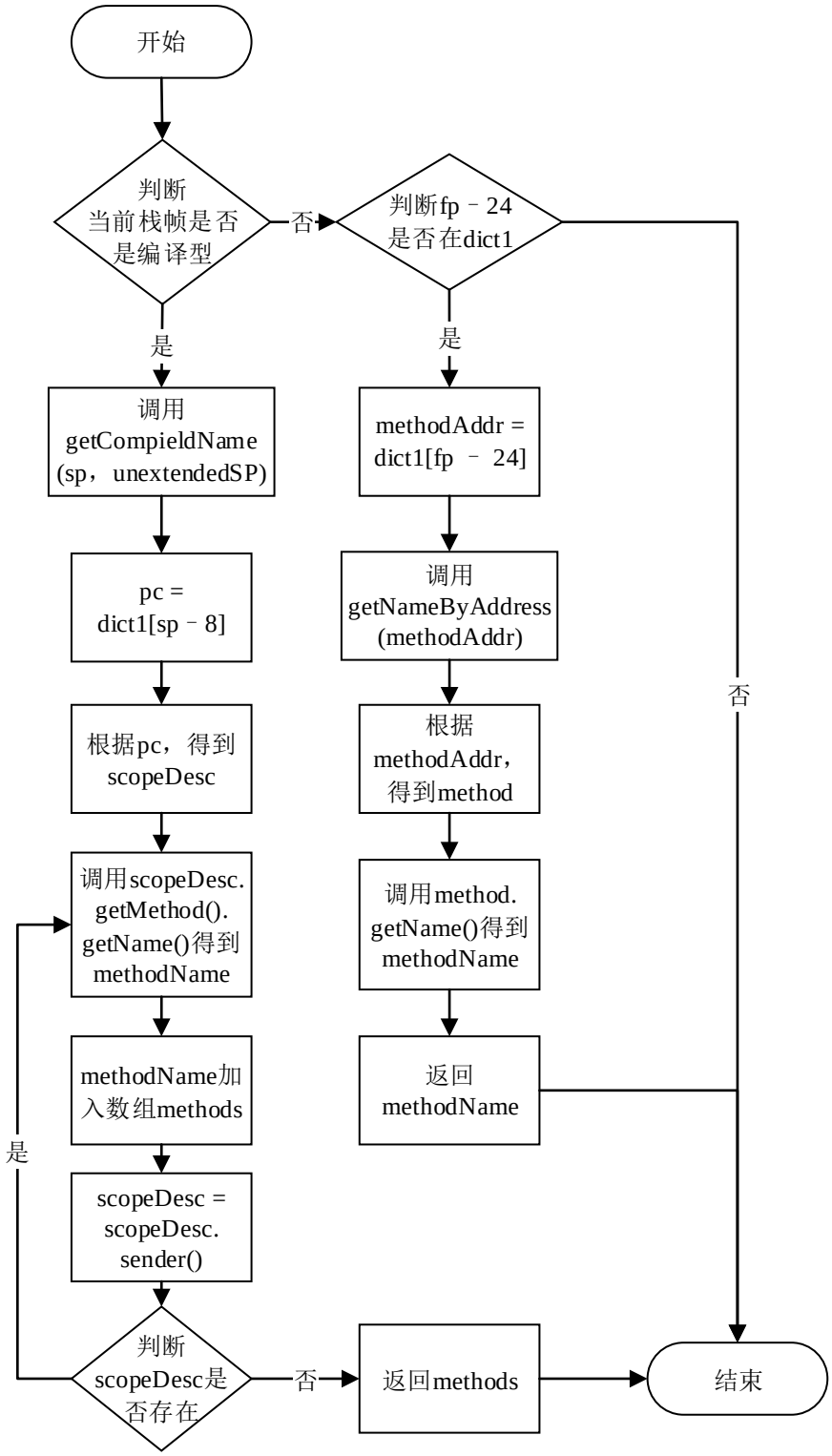

图 5-17 getName ()程序流程图

getLocals()方法根据当前栈帧的类型进行处理，程序流程图如图 5-18 所示。

如果当前栈帧是解释型的，直接根据解释型的原理进行计算。如果当前栈帧是编译型的，调用 PyDump 的 getCompieldLocals()方法可以得到一个二维数组，长度为相同 sp 的栈帧的数量，一维数组代表每个栈帧的参数值。

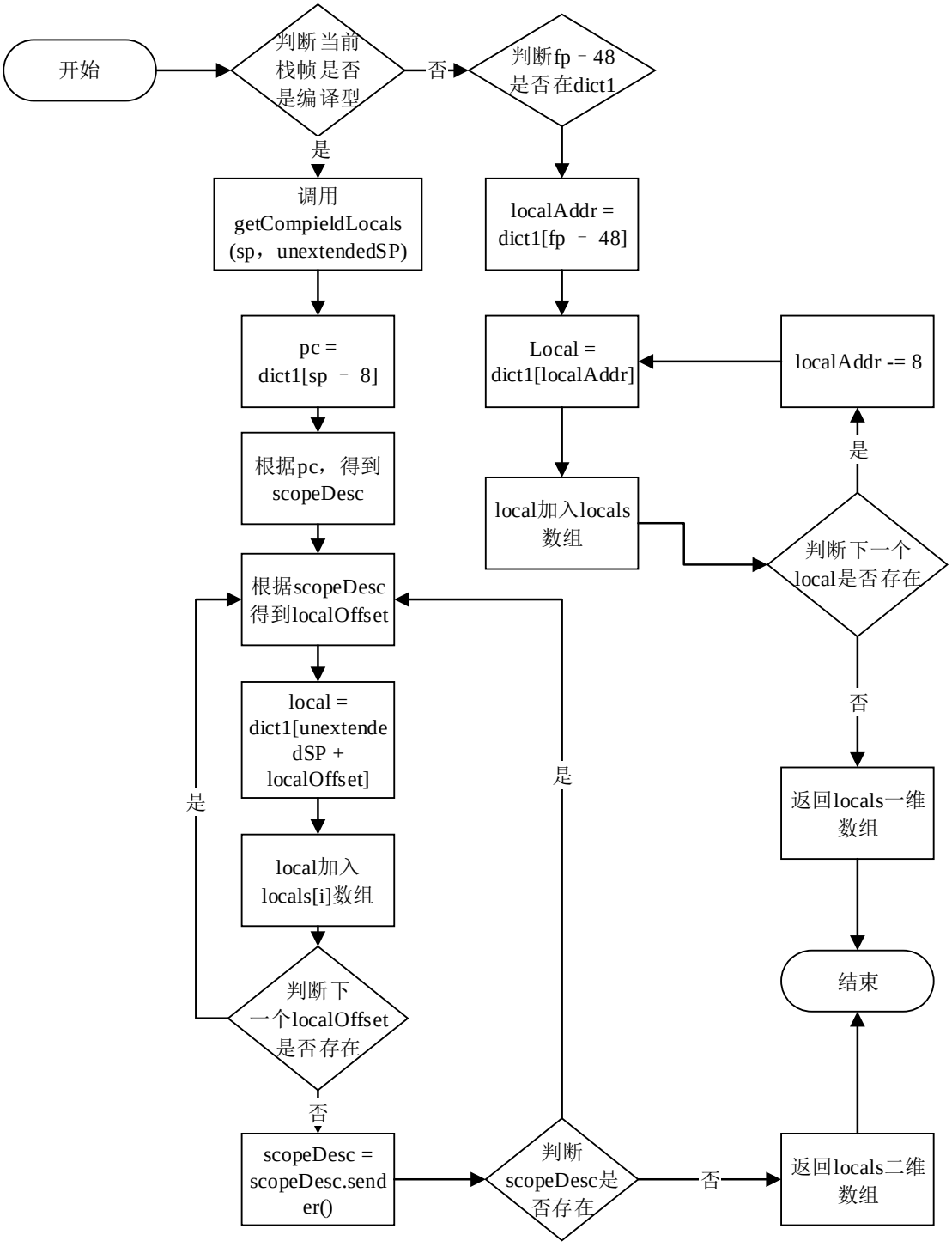

图 5-18 getLocals()程序流程图

在编译分析时，使用 ScopeDesc 类进行 Scope 代码区的计算，ScopeDesc 类的 check()方法用于判断当前代码区与 frameSize 是否匹配，getLocals()方法可以得到当前的函数参数偏移，getMethod()方法可以得到当前函数的引用，sender()方

法可以得到下一个相同 sp 的栈帧的 ScopeDesc。如图 5-19 所示。

图 5-19 ScopeDesc 类图

5.3.4 栈内存分析序列图

在调用 PyDump 的 initJavaFirstFPAddress()方法过程中，先调用了 Threads 类的 first()方法，然后调用了 JavaThread 的 next()方法和 getLastJavaVFrameDbg()方法，然后调用了 JavaVFrame 的 javaSender()方法和 Frame 的 sender()方法，当找到 main 函数时，调用 getFrame().getFp()方法得到 main 函数的 fp。如图 5-20 所示。

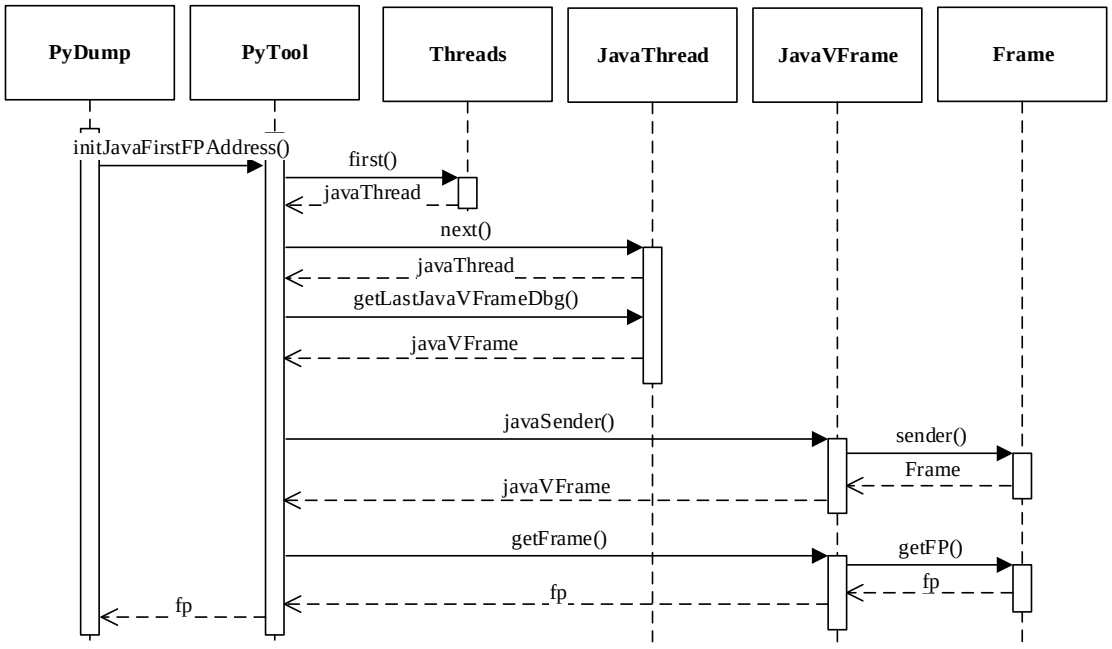

图 5-20 初始化 fp 序列图

在进行解释型栈帧分析时，先调用 readMemory()方法读取栈内存构造 main 函数栈帧然后调用 getName()方法得到名称，调用 getLocals()方法得到本地变量，最后调用 getNextFrame()获取下一个栈帧进行分析。如图 5-21 所示。

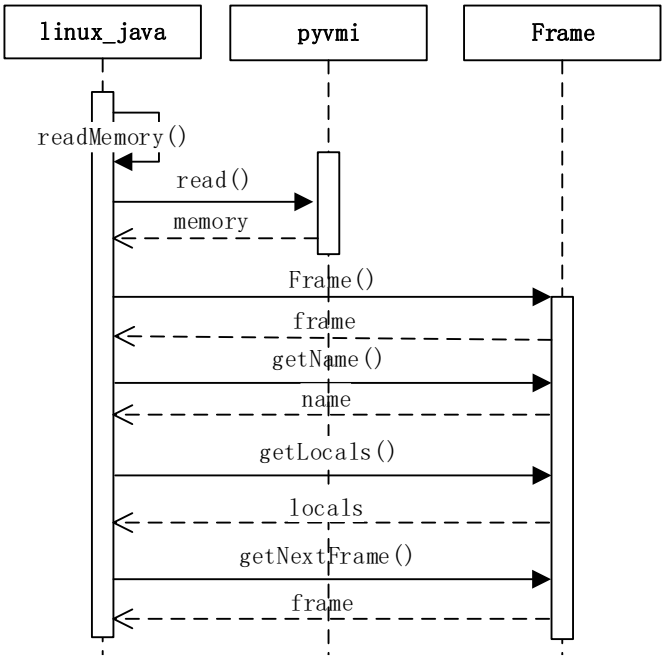

图 5-21 解释运行的内存分析序列图

在进行编译型栈帧分析时，先调用 getNextComplied()方法得到 sp，然后根据 sp 和 ScopeDesc 类的 getLocals()，getMethod()，sender()来获取所有相同 sp 的编译型栈帧的参数名称和参数值。如图 5-22 所示。

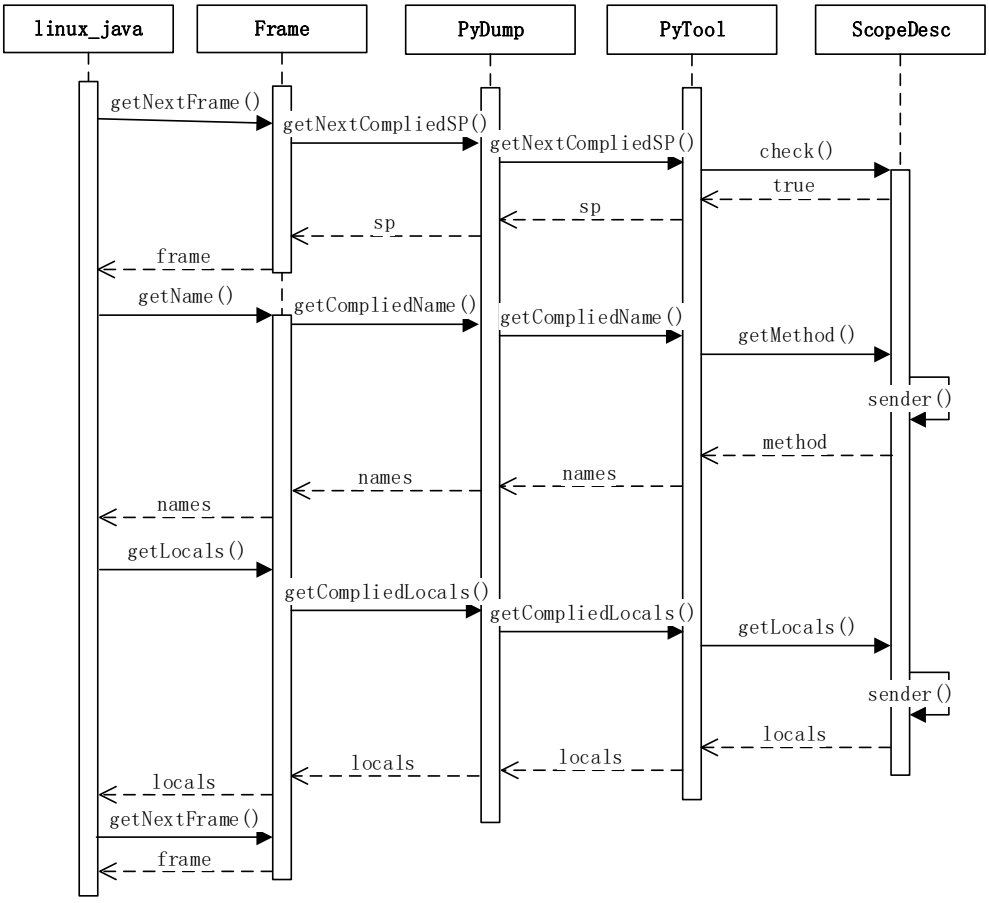

图 5-22 编译运行的内存分析序列图

5.3.5 服务容器内存分析实现

服务容器内存分析基于内存获取模块和栈内存分析模块实现。通过配置参数对执行线程 run 函数的分析。

首先，设置栈底函数名称 stack_base_name 为 run 函数，声明 addresses 列表用于保存所有执行线程的栈底内存地址。其次，调用 Threads 的 first 方法得到第一个线程，判断是否为执行线程，然后利用 JavaThread 的 next 方法循环遍历出所有执行线程的栈底内存地址。根据地址获取栈内存，利用内存分析模块分析出栈帧信息，获取到服务参数。程序流程图如 5-23 所示。

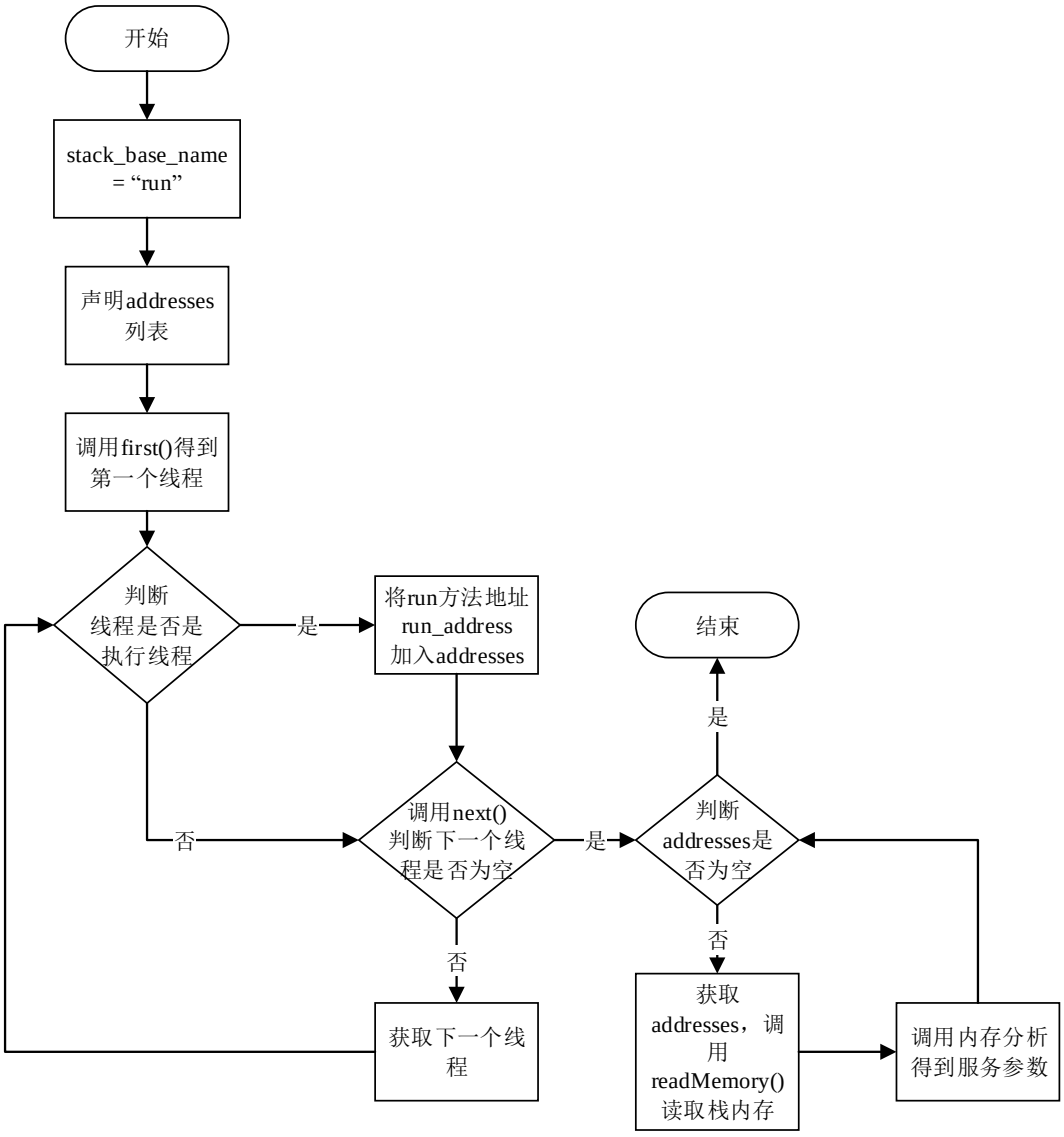

图 5-23 服务容器内存分析程序流程图

5.3.6 服务容器内存分析序列图

服务容器内存分析模块的序列图如图 5-24 所示。linux_java 调用 getAddresses()，通过 PyDump 类、PyTool 类、Threads 类和 JavaThread 类得到执

行线程的栈底内存地址 addresses。调用 readMemory()得到栈内存，调用 Frame 类来分析栈内存得到服务参数。

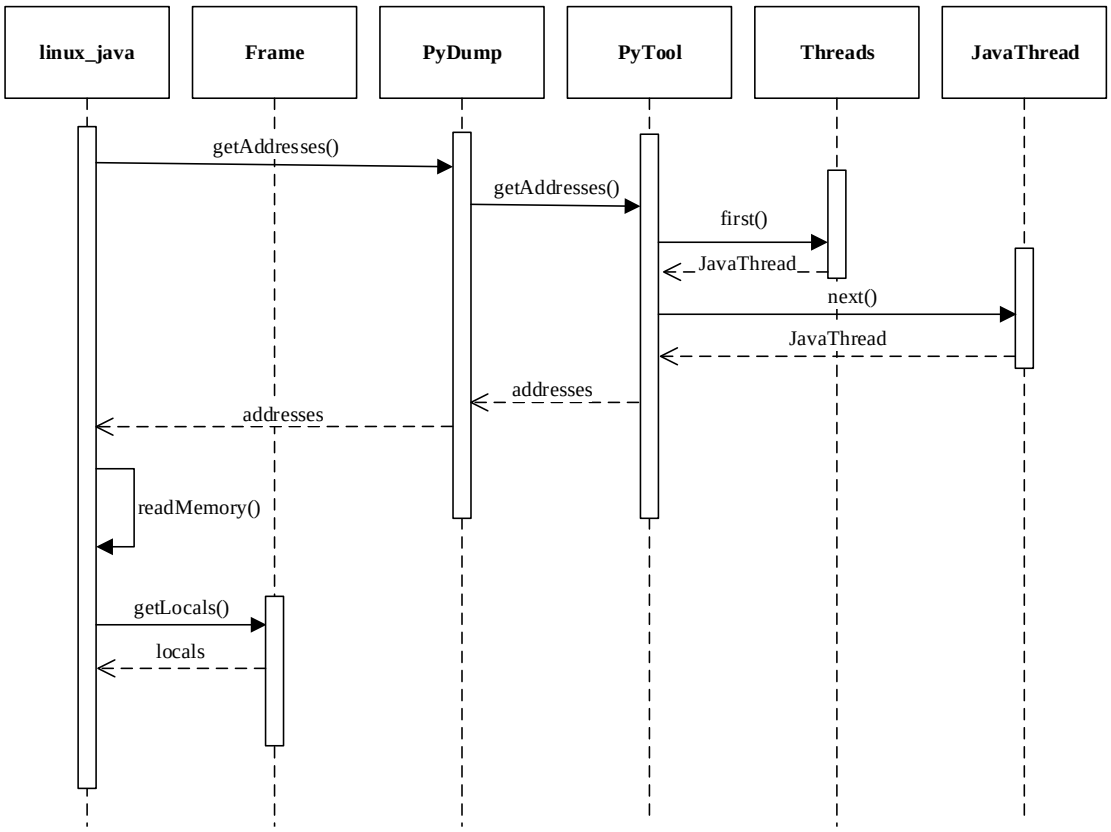

图 5-24 服务容器内存分析序列图

5.4 内存事件

内存事件模块分为事件构造子模块和事件发送子模块。采用 Python 编写。linux_java 事件接口通过内存分析得到栈帧信息和服务参数，调用事件构造子模块，构造内存事件，之后调用事件发送子模块，发送内存事件到物联网运行时验证系统。如图 5-25 所示。

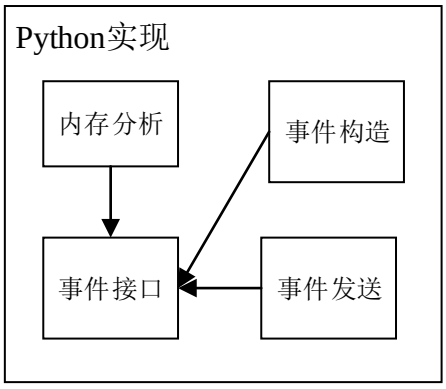

图 5-25 内存事件结构图

5.4.1 内存事件实现

在获取了栈帧信息和服务参数之后，系统要构造并发送事件。Event 类是用来实现上述功能的，init()方法用于初始化，setNameAndLocals()方法根据参数配置文件构造字符串事件，setupConnect()方法根据发送端口建立 UDP Server，sendEvent()方法发送构造好的事件。如图 5-26 所示。

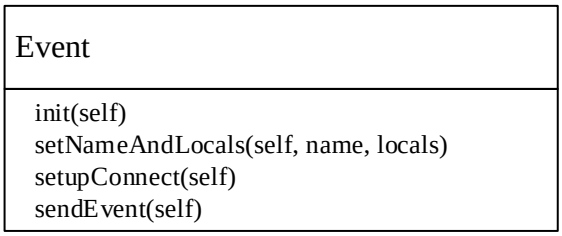

图 5-26 Event 类图

5.4.2 内存事件序列图

先调用 Event 类的 init()方法初始化读取参数配置文件，再调用 setNameAndLocals()方法构造事件，然后调用 setupConnect()方法建立 UDP Server，accept()方法监听发送端口。当物联网运行时验证系统建立连接之后，调用 sendEvent()方法，发送字符串事件。如图 5-27 所示。

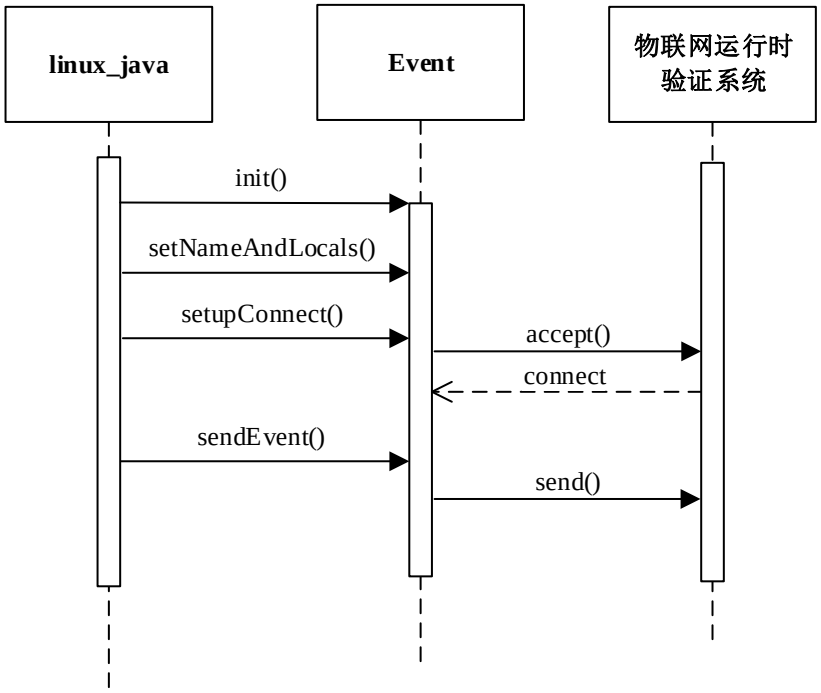

图 5-27 内存事件序列图

5.5 界面系统

界面系统分为服务参数配置和界面展示两个子模块。使用 Python 编写。linux_java 中的界面接口调用参数配置和界面展示子模块。如图 5-28 所示。

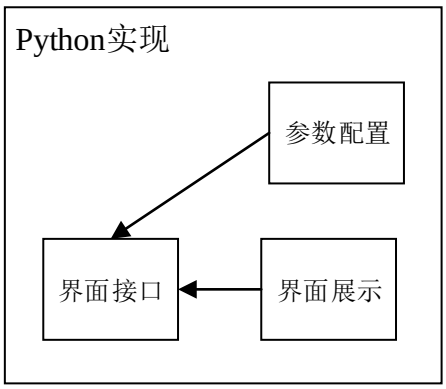

图 5-28 界面系统结构图

5.5.1 界面系统实现

服务参数配置功能主要通过配置服务的相关参数，使用户可以定制化的对不同的物联网服务进行监控。用户只需在界面中填写相关参数可以实现定制，以满足用户的特定需求。服务语义展示是用来展示用户所需要的服务调用，参数信息以及内存事件。类 Conf 用于参数配置，setConf()方法进行参数配置，check()方法用于校验参数。如图 5-29 所示。

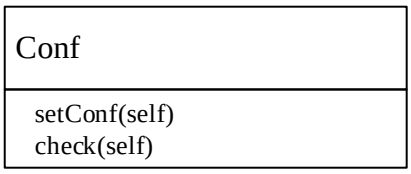

图 5-29 Conf 类图

5.5.2 界面系统序列图

界面系统调用 Conf 类的 setConf()方法设置参数，Conf 类调用 check()方法进行参数校验。界面系统调用 show()方法来展示栈帧信息和内存事件。如图 5-30 所示。

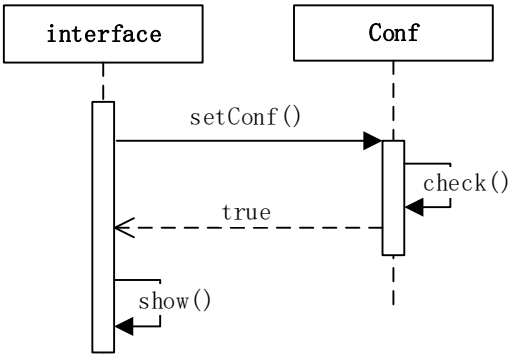

图 5-30 界面系统序列图

5.6 本章总结

本章讲述了服务语义重构系统的具体实现，从内存获取，内存分析到内存事件以及界面系统，详细介绍了各个模块的具体实现。通过阅读本章，可以从代码层面上了解服务语义重构系统是如何工作的，服务语义重构的原理是如何应用的。本系统基本满足了需求，实现系统设计的各个模块。

第六章 系统测试

本章对服务语义重构系统进行测试。测试分为两方面，功能测试和性能测试。功能测试从系统的功能模块出发，验证各个模块功能是否正确实现，同时模块间的调用和模块组成的系统是否正确实现。性能测试从系统可用性出发，主要测试各个模块的性能指标和系统整体的性能，用以判断系统是否适应于实际场景。本章从测试环境部署出发，设计相应的测试用例，对服务语义重构系统的功能和性能进行完备的测试。

6.1 测试环境

物联网服务部署于虚拟化平台之中，而服务语义重构系统部署于虚拟化平台的物理机之中。测试环境分为硬件环境和软件环境两部分。

6.1.1 硬件环境

部署服务语义重构系统的物理机配置如下：

1) 电脑型号 ：联想启天 B4360-B015 台式电脑
2) 处理器：英特尔 Pentium(奔腾) G2030 @ 3.00GHz 双核
3) 内存：4 GB
4) 主硬盘：希捷 500 GB
5) 网卡：瑞昱 RTL8168/8111/8112

部署物联网服务的虚拟化平台虚拟配置如下：

1) 处理器：虚拟的单核 CPU
2) 内存：1GB
3) 20G 可用硬盘

6.1.2 软件环境

由于服务语义重构系统使用 Python 和 Java 编写，且物联网服务运行于虚拟化平台中，因此服务语义重构系统软件环境版本如下：

1) Ubuntu16.04.01x64 桌面版
2) 内核版本 4.10.0-40-generic
3) Python 2.7.12
4) Java 1.7.0_91
5) QEMU-KVM 版本 1:2.5+dfsg-5ubuntu10.16

物联网服务软件环境版本如下：

1) Ubuntu16.04.01x64 桌面版

2) 内核版本 4.10.0-40-generic
3) Java 1.7.0_91
4) Tomcat 7.0

6.2 测试环境部署

要进行功能测试和性能测试，先要完成服务语义重构系统和物联网服务的部署。已实现服务语义重构系统的自动化部署。在服务语义重构系统的测试中，被监控的物联网服务需要能够全面的检验服务语义重构系统的各项功能。设计 Java 程序作为被监控的物联网服务，如表 6-1 所示。

表 6-1 物联网服务 Java 程序说明表

函数名称	说明
main	入口函数
func1	输入参数为 int 类型
func2	输入参数为 long 类型
func3	输入参数为 float 类型
func4	输入参数为 double 类型
func5	无输入参数

6.3 功能测试

设计测试用例测试服务语义重构系统的功能是否正确实现。在测试之前，确认测试的物联网 Java 服务已经正确运行。在测试过程中，默认物联网 Java 服务运行正常。

6.3.1 测试用例

根据功能测试要求，设计如下测试用例。

表 6-2 系统部署测试

测试编号：功能测试 1
测试项目：系统部署测试
测试目的：测试系统通过部署后能否正常运行
测试用例： 1) 启动物理机，安装配置服务语义重构； 2) 启动虚拟机，安装配置物联网 Java 服务； 3) 启动物联网 Java 服务； 4) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 5) 启动服务语义重构系统。
预期结果： 1) 服务语义重构系统部署成功； 2) 服务语义重构系统启动成功。

表 6-3 服务解释运行测试

测试编号：功能测试 2
测试项目：服务解释运行测试
测试目的：测试系统能否重构解释运行的 Java 服务并发送监控事件
测试用例： 1) 配置物联网 Java 服务，使其程序解释运行； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，查看界面显示栈帧信息和内存事件。
预期结果： 1) 服务语义重构系统运行正常，界面显示栈帧信息； 2) 当发生被监控函数调用时，界面显示内存事件，并发送内存事件到物联网运行时验证系统。

表 6-4 服务编译运行测试

测试编号：功能测试 3
测试项目：服务编译运行测试
测试目的：测试系统能否重构编译运行的 Java 服务并发送监控事件
测试用例： 1) 配置物联网 Java 服务，使其程序编译运行； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，查看界面显示栈帧信息和内存事件。
预期结果： 1) 服务语义重构系统运行正常，界面显示栈帧信息； 2) 当发生被监控函数调用时，界面显示内存事件，并发送内存事件到物联网运行时验证系统。

表 6-5 服务混合运行测试

测试编号：功能测试 4
测试项目：服务混合运行测试
测试目的：测试系统能否重构混合运行的 Java 服务并发送监控事件
测试用例： 1) 配置物联网 Java 服务，使其程序运行在混合模式； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，查看界面显示栈帧信息和内存事件。
预期结果： 1) 服务语义重构系统运行正常，界面显示栈帧信息； 2) 当发生被监控函数调用时，界面显示内存事件，并发送内存事件到物联网运行时验证系统。

表 6-6 服务进程暂停测试

测试编号：功能测试 5
测试项目：服务进程暂停测试
测试目的：测试系统能否重构进程暂停的 Java 服务
测试用例： 1) 启动物联网 Java 服务； 2) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 3) 启动服务语义重构系统，查看界面显示栈帧信息和内存事件； 4) 使物联网 Java 服务进程挂起； 5) 查看界面显示栈帧信息。
预期结果： 1) 服务语义重构系统运行正常，界面显示栈帧信息； 2) 当发生被监控函数调用时，界面显示内存事件，并发送内存事件到物联网运行时验证系统； 3) 当 Java 服务进程挂起，界面显示栈帧信息，栈帧信息不变。

表 6-7 服务进程退出测试

测试编号：功能测试 6
测试项目：服务进程退出测试
测试目的：测试系统能否检测 Java 服务进程退出
测试用例： 1) 启动物联网 Java 服务； 2) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 3) 启动服务语义重构系统，查看界面显示栈帧信息和内存事件； 4) 使物联网 Java 服务进程退出； 5) 查看界面显示栈帧信息。
预期结果： 1) 服务语义重构系统运行正常，界面显示栈帧信息； 2) 当发生被监控函数调用时，界面显示内存事件，并发送内存事件到物联网运行时验证系统； 3) 当 Java 服务进程退出，界面显示进程已经退出。

表 6-8 服务参数测试

测试编号：功能测试 7
测试项目：服务参数测试
测试目的：测试系统能否重构不同类型的参数信息
测试用例： 1) 启动物联网 Java 服务； 2) 配置 Java 服务 pid； 3) 把所有函数都配置为被监控函数，并配置相应的被监控函数名称、函数参数数量和类型（int、long、float、double）； 4) 启动服务语义重构系统，查看界面显示栈帧信息和内存事件。
预期结果： 1) 服务语义重构系统运行正常，界面显示栈帧信息，不同函数参数类型均能正确； 2) 当发生被监控函数调用时，界面显示内存事件，并发送内存事件到物联网运行时验证系统。

表 6-9 服务容器测试

测试编号：功能测试 8
测试项目：服务容器测试
测试目的：测试系统能否重构服务容器调用服务参数
测试用例： 1) 部署物联网 Java 服务于 Tomcat 中； 2) 启动 Tomcat； 3) 配置 Tomcat 服务的 pid； 4) 把服务函数配置为被监控函数，并配置相应的被监控函数名称、函数参数数量和类型； 5) 启动服务语义重构系统，查看界面显示栈帧信息和内存事件。
预期结果： 1) 服务语义重构系统运行正常，界面显示栈帧信息和调用服务的参数； 2) 当发生服务调用时，界面显示内存事件，并发送内存事件到物联网运行时验证系统。

6.3.2 测试结果说明及结果分析

在进行功能测试的过程中，编写设计了测试用例来测试服务语义重构系统的各项功能，包括系统部署测试、服务解释运行测试、服务编译运行测试、服务混

合运行测试、服务进程暂停测试、服务进程退出测试、服务参数测试和服务容器测试。

经过测试，根据测试结果与预期结果相比较，系统各项功能均能达到预期结果。如图 6-1 所示。

```
-> main() -> func3(7.7,9.9)
-> main() -> func2(100100010000,200200020000)
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6) -> func5()
-> main() -> func1(5,3) -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6) -> func5()
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6) -> func5()
-> main() -> func1(5,3)
-> main() -> func1(5,3) -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6)
-> main() -> func1(5,3) -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6) -> func5()
-> main() -> func3(7.7,9.9)
-> main() -> func4(4.4,6.6)
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9)
-> main() -> func4(4.4,6.6) -> func5()
-> main() -> func1(5,3) -> func2(100100010000,200200020000) -> func3(7.7,9.9)
-> main() -> func4(4.4,6.6)
-> main() -> func4(4.4,6.6) -> func5()
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6)
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9)
-> main() -> func1(5,3) -> func2(100100010000,200200020000) -> func3(7.7,9.9)
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6)
-> main() -> func3(7.7,9.9) -> func4(4.4,6.6)
-> main() -> func2(100100010000,200200020000)
-> main() -> func1(5,3) -> func2(100100010000,200200020000) -> func3(7.7,9.9)
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6) -> func5()
-> main() -> func2(100100010000,200200020000) -> func3(7.7,9.9) -> func4(4.4,6.6)
-> main() -> func3(7.7,9.9) -> func4(4.4,6.6)
-> main() -> func3(7.7,9.9)
-> main() -> func1(5,3) -> func2(100100010000,200200020000)
```

图 6-1 功能测试结果截图

6.4 性能测试

在实现各项功能的基础上，由于物联网服务的要求，性能也是服务语义重构系统的核心要求之一。在本系统中，对性能影响较大的主要是内存获取效率和内存分析效率。因此，要对内存获取效率和内存分析效率进行性能测试。同时，也要对系统整体的资源占用等性能指标进行测试。

6.4.1 内存获取性能测试

由于地址内存获取，共享库获取和子线程获取子模块均在初始化加载之后，就不再调用，因此内存获取的性能主要是栈内存获取子模块的性能。栈内存获取受到被监控程序的函数状态影响，因此，主要测试对不同参数数量、不同参数类型、不同栈帧数量、不同运行模式函数的栈内存获取效率。

表 6-10 不同参数数量的栈内存获取效率测试

测试编号：性能测试 1
测试项目：不同参数数量的栈内存获取效率测试
测试目的：测试系统对不同参数数量的物联网服务的栈内存获取效率
测试用例： 1) 分别配置函数参数数量为 1，2，4，8； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，记录不同参数数量的情况下栈内存获取时间； 5) 查看不同参数数量下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同参数数量的栈内存获取时间基本相同。

表 6-11 不同参数类型的栈内存获取效率测试

测试编号：性能测试 2
测试项目：不同参数类型的栈内存获取效率测试
测试目的：测试系统对不同参数类型的物联网服务的栈内存获取效率
测试用例： 1) 分别配置函数参数类型为 int、float、long、double； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，记录不同参数类型的情况下栈内存获取时间； 5) 查看不同栈帧数量下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同参数类型的栈内存获取时间基本相同。

表 6-12 不同栈帧数量的栈内存获取效率测试

测试编号：性能测试 3
测试项目：不同栈帧数量的栈内存获取效率测试
测试目的：测试系统对不同栈帧数量的物联网服务的栈内存获取效率
测试用例： 1) 启动物联网 Java 服务； 2) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 3) 启动服务语义重构系统，记录不同栈帧数量的情况下栈内存获取时间； 4) 查看不同参数数量下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同栈帧数量的栈内存获取时间基本相同。

表 6-13 不同运行方式的栈内存获取效率测试

测试编号：性能测试 4
测试项目：不同运行方式的栈内存获取效率测试
测试目的：测试系统对不同运行方式的物联网服务的栈内存获取效率
测试用例： 1) 分别配置物联网 Java 服务运行方式为解释运行、编译运行和混合运行； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，记录不同运行方式的情况下栈内存获取时间； 5) 查看不同运行方式下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同运行方式的栈内存获取时间基本相同。

根据以上测试用例，对栈内存获取模块进行测试。以下测试结果横坐标为获取次数，纵坐标为获取时间，单位为毫秒。如图 6-2 所示，不同参数数量的栈内存获取时间基本相同。

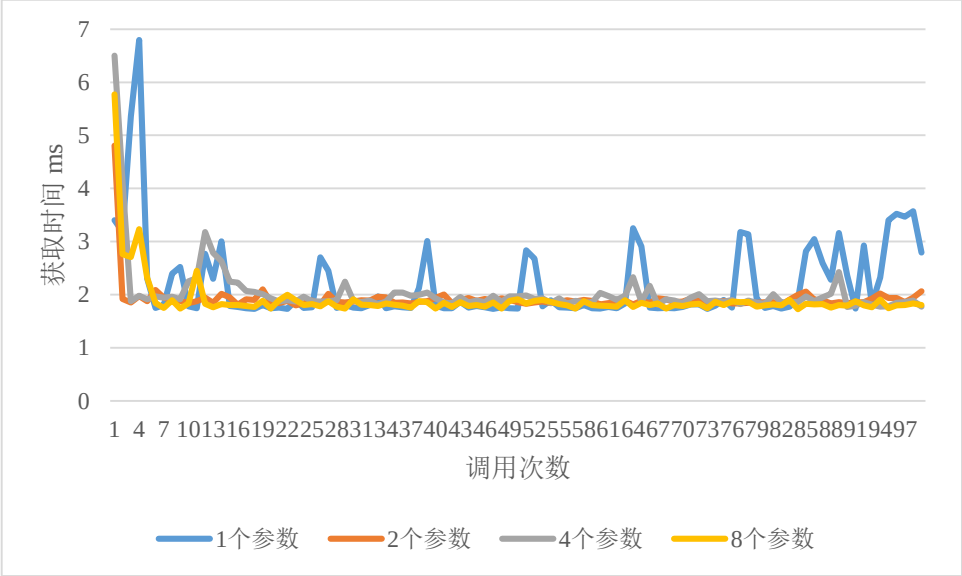

图 6-2 不同参数数量的栈内存获取时间

如图 6-3 所示，不同参数类型的栈内存获取时间基本相同。

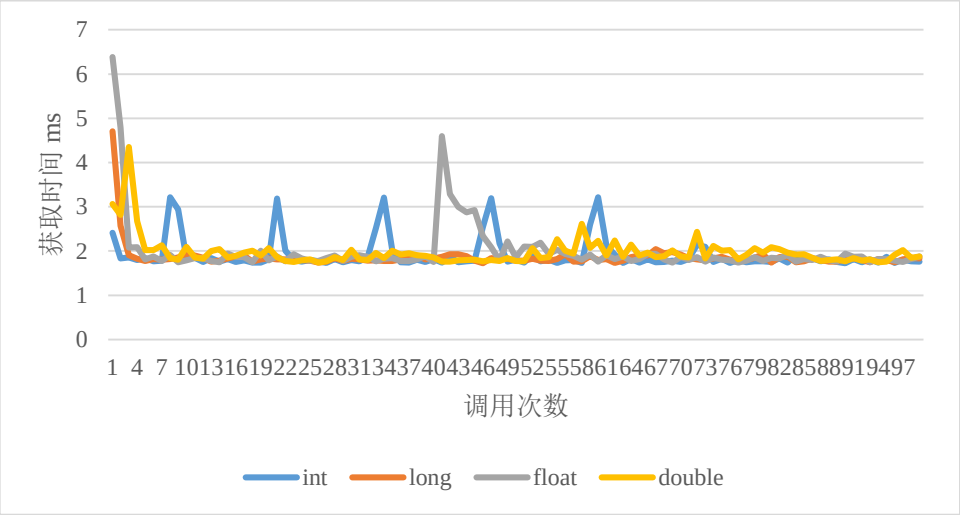

图 6-3 不同参数类型的栈内存获取时间

如图 6-4 所示，不同栈帧数量的栈内存获取时间基本相同。

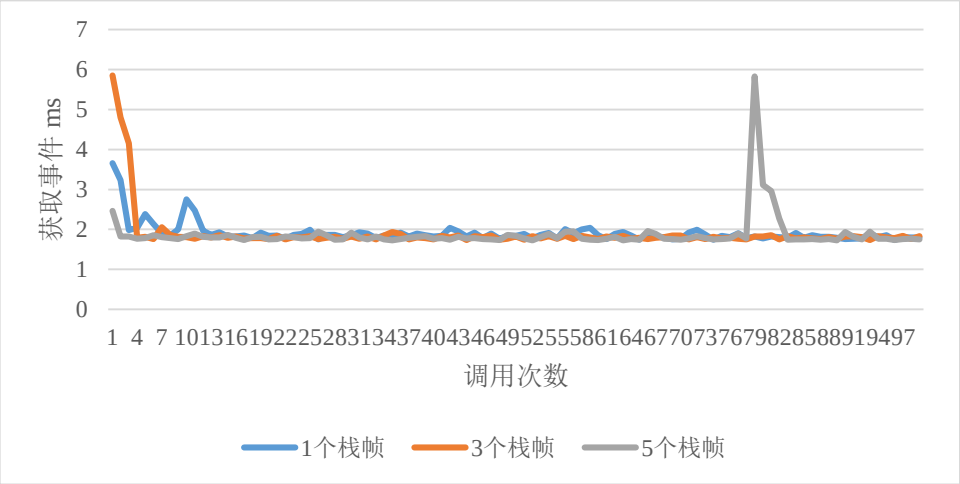

图 6-4 不同栈帧数量的栈内存获取时间

如图 6-5 所示，不同运行方式的栈内存获取时间基本相同。

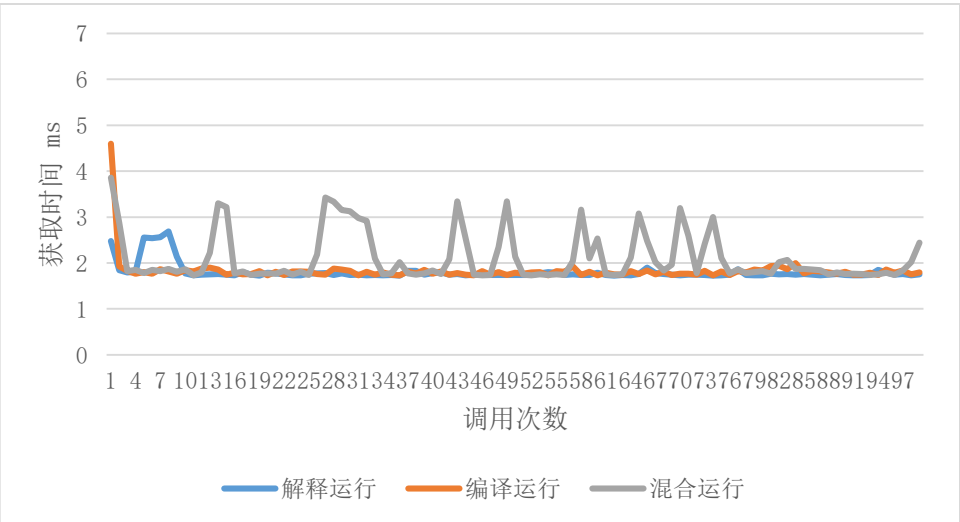

图 6-5 不同运行方式的栈内存获取时间

以上所有用例栈内存获取事件都基本相同，是因为在进行系统实现时，栈内存获取是先于栈帧分析进行的。根据栈底和栈大小获取栈内存，因此栈帧状态不影响栈内存获取时间。以测试程序为例，栈内存获取事件平均事件在 10ms 以内。

6.4.2 内存分析性能测试

在内存分析模块中，主要影响性能的是栈内存分析子模块。分析效率主要受到被监控程序的函数状态影响，因此，主要测试对不同参数数量、不同参数类型、不同栈帧数量、不同运行模式函数的栈内存分析效率。

表 6-14 不同参数数量的栈内存分析效率测试

测试编号：性能测试 5
测试项目：不同参数数量的栈内存分析效率测试
测试目的：测试系统对不同参数数量的物联网服务的栈内存分析效率
测试用例： 1) 分别配置函数参数数量为 1，2，4，8； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，记录不同参数数量的情况下栈内存分析时间； 5) 查看不同参数数量下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同参数数量的栈内存分析时间不同，参数数量与分析时间成正比。

表 6-15 不同参数类型的栈内存分析效率测试

测试编号：性能测试 6
测试项目：不同参数类型的栈内存分析效率测试
测试目的：测试系统对不同参数类型的物联网服务的栈内存分析效率
测试用例： 1) 分别配置函数参数类型为 int、float、long、double； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，记录不同参数类型的情况下栈内存分析时间； 5) 查看不同参数数量下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同参数类型的栈分析时间基本相同。

表 6-16 不同栈帧数量的栈内存分析效率测试

测试编号：性能测试 7
测试项目：不同栈帧数量的栈内存获取效率测试
测试目的：测试系统对不同栈帧数量的物联网服务的内存获取效率
测试用例： 1) 启动物联网 Java 服务； 2) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 3) 启动服务语义重构系统，记录不同栈帧数量的情况下栈内存分析时间； 4) 查看不同栈帧数量下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同栈帧数量的栈内存分析时间不同，栈帧数量与分析时间成正比。

表 6-17 不同运行方式的栈内存分析效率测试

测试编号：性能测试 8
测试项目：不同运行方式的栈内存获取效率测试
测试目的：测试系统对不同运行方式的物联网服务的内存获取效率
测试用例： 1) 分别配置物联网 Java 服务运行方式为解释运行、编译运行和混合运行； 2) 启动物联网 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统，记录不同运行方式的情况下栈内存获取时间； 5) 查看不同运行方式下的时间结果，汇总统计，对比结果，分析结果。
预期结果： 1) 不同运行方式的栈内存分析时间不同，编译运行时间最长，混合运行时间其次，解释运行时间最短。

根据以上测试用例，对栈内存分析模块进行测试。如图 6-6 所示，不同参数数量的栈内存分析时间与参数数量正相关，这是因为函数参数是遍历分析的，参数越多，分析事件就会越长。

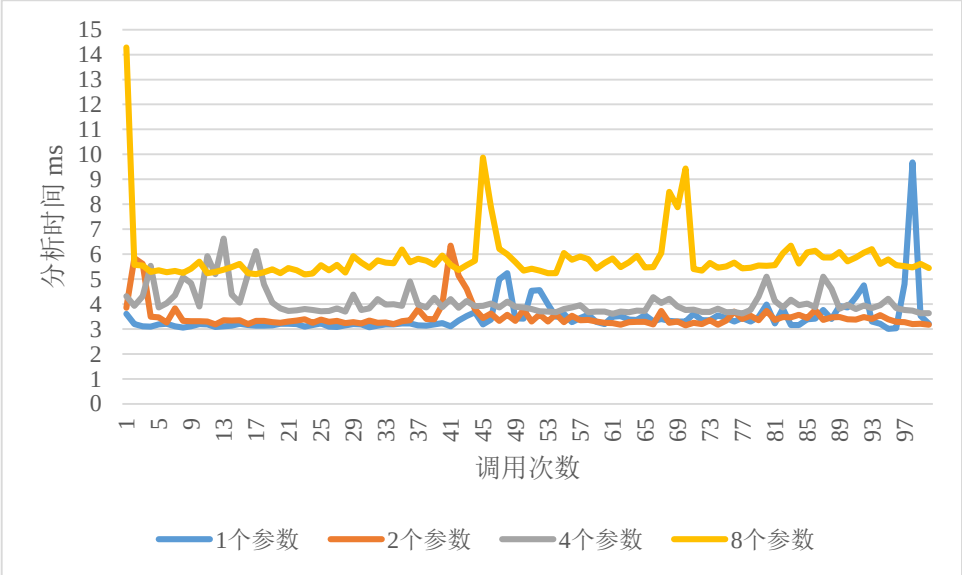

图 6-6 不同参数数量的栈内存分析时间

如图 6-7 所示，不同参数类型的栈内存分析时间中，int、long 类型比 float、double 类型分析时间短，这是因为内存二进制数据可以直接转为 int 和 long，而 float 和 double 需要浮点数转换增加了分析时间。

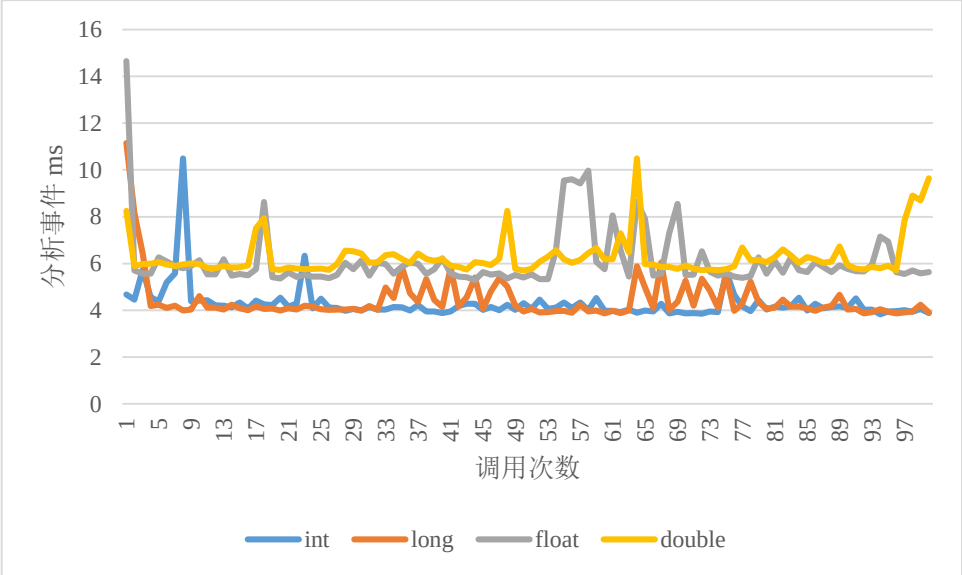

图 6-7 不同参数类型的栈内存分析时间

如图 6-8 所示，不同栈帧数量的栈内存分析时间不同，栈帧数量与分析时间成正相关，这是因为栈帧分析是遍历栈帧进行的，因此栈帧数量越多，遍历分析的时间就越长。

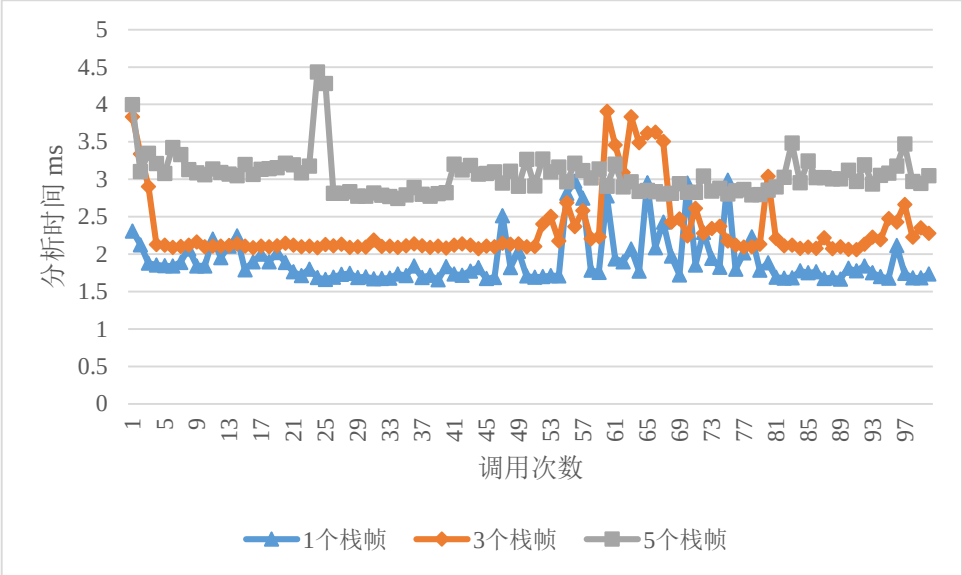

图 6-8 不同栈帧数量的栈内存分析时间

如图 6-9 所示，不同运行方式的栈内存分析时间不同，解释运行时间最短，混合运行次之，编译运行事件最长。这是因为编译运行需要循环遍历 frameSize 和本地代码区来查找下一个栈帧，因此分析事件较长。而混合运行有部分栈帧运行在编译模式，因此会增加分析事件。

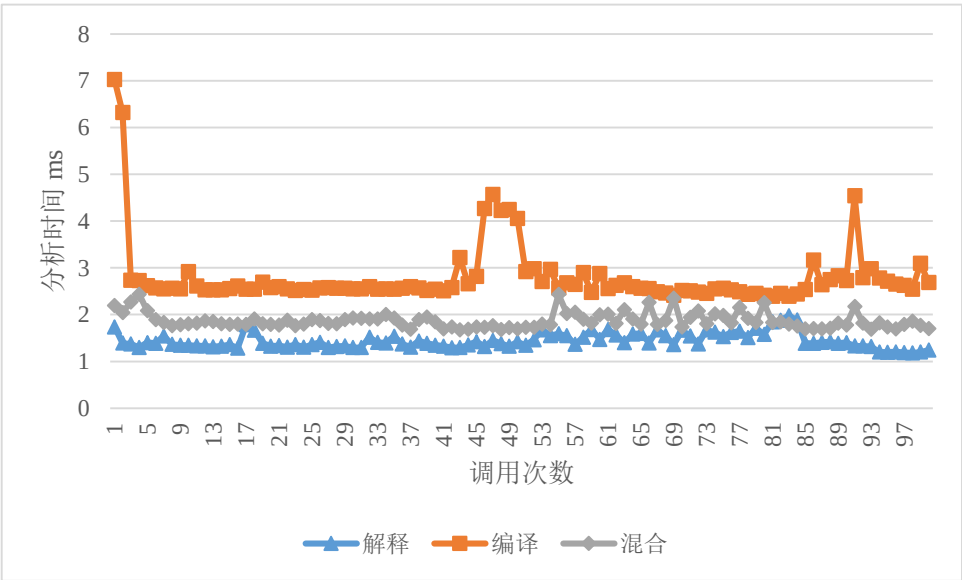

图 6-9 不同运行方式的栈内存分析时间

以上可知，栈内存获取因为功能实现方式受到函数状态的影响。以测试程序为例，栈内存分析的平均时间在 20ms 以内。

6.4.3 系统整体性能测试

除了内存获取和内存分析模块的性能测试，还需要对系统重构效率进行测试，记录不同内存获取频率在单位时间内重构出的函数调用次数与实际发送的函数调用次数，计算系统的重构效率。同时，也要测试系统的整体资源占用情况。

表 6-18 重构率测试

测试编号：性能测试 9
测试项目：重构效率测试
测试目的：测试系统服务语义重构效率
测试用例： 1) 配置记录内存获取重构的间隔 0.1s、0.2s、0.3s，0.5s，1s； 2) 启动 Java 服务； 3) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 4) 启动服务语义重构系统； 5) 以 300s 为时间单位记录栈内存重构获取次数； 6) 查看所有结果，汇总统计，对比结果，分析结果。
预期结果： 1) 重构效率随着调用重构间隔的增大而增大。

表 6-19 资源占用率测试

测试编号：性能测试 10
测试项目：资源占用率测试
测试目的：测试系统对 CPU 和内存资源的占用率
测试用例： 1) 启动物联网 Java 服务； 2) 配置 Java 服务 pid、被监控函数名称、函数参数数量和类型； 3) 启动服务语义重构系统； 4) 分别按照 0min，1min，2min，3min，4min，5min，6min，7min，8min 记录系统对 CPU 和内存的占用率； 5) 查看所有结果，汇总统计，对比结果，分析结果。
预期结果： 1) 系统资源占用率较低，占用率平稳。

根据以上测试用例，对系统整体进行测试。如图 6-10 所示，单位时间内，内存重构间隔越长，总获取次数越少，同时获取到参数的次数越少。但是在去除获取到的重复参数之后，获取到参数的次数基本相等。这是因为 libvmi 内存获取函数的效率和缓存等原因导致单位时间内能获取到的内存变化次数基本相同。

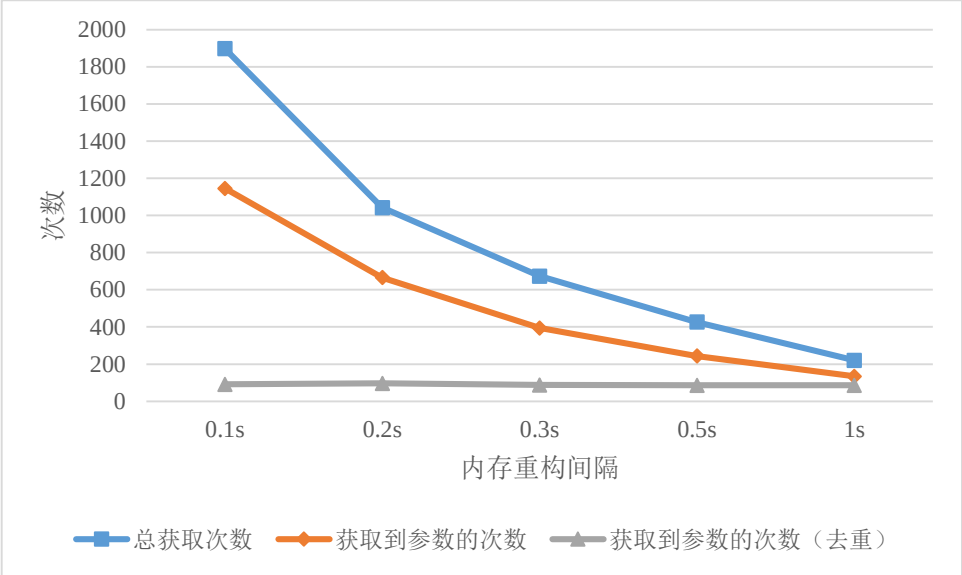

图 6-10 系统重构效率

如图 6-11 所示，系统资源占用率与预期相同，CPU 和内存资源占用都比较低。

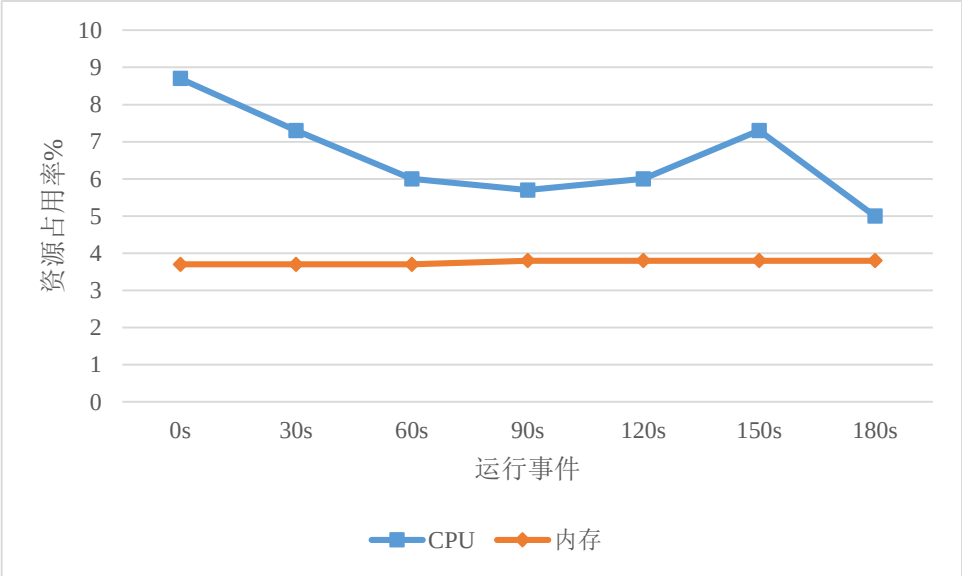

图 6-11 系统资源占用率

6.5 本章总结

本章根据需求分析设计了若干测试用例，通过系统测试，从功能和性能两方面验证服务语义重构系统。根据测试结果，系统很好的完成了各项功能，性能指标也基本满足物联网服务的要求。同时，资源占用率略高，需要改进。

第七章 总结和展望

本章对基于服务容器的服务语义重构系统的研究、设计、实现和测试工作进行了总结，同时对未来的研究工作进行了展望。

7.1 工作总结

本文的研究是在实验室已有研究工作的基础展开的。已有的基于虚拟化平台的物联网保障系统和运行时验证系统在语义重构方面还不完善，无法对 Java 服务进行语义重构。在已有的虚拟化平台内存获取技术的研究基础之上，经过大量调研分析，研究分析 JDK 以及相关 Java 内存分析工具，提出了对基于虚拟化平台的 Java 服务的内存分析技术。根据相关研究和技术，设计并实现了基于服务容器的服务语义重构系统。

该系统的目标是对物联网 Java 服务通过内存获取和内存分析实现函数调用和函数参数的重构，最终重构出服务语义。基于这个目标，首先研究了针对 JVM 内存分析的内存获取技术。基于已有的虚拟机内存自省技术，设计并实现了共享库获取、子线程获取、地址内存获取和栈内存获取。其次，根据对 JDK 以及相关 Java 内存分析工具的研究和分析，在内存分析中，先对 JVM 的整体数据结构进行分析，获取 JVM 所有的类型结构以及在内存中的地址。然后根据解释运行和编译运行的栈帧特点，分析栈内存。以 main 函数为栈底，自栈底到栈顶逐帧分析，先判断栈帧的类型，再根据相应的类型，重构函数名称和函数参数。根据栈帧分析的先后顺序，确定函数调用。最后，将函数调用和函数参数构造内存事件发送给物联网验证系统。同时，为方便用户配置和查看设计了界面系统。

由于能力的不足，本系统在 Java 服务语义重构还有一些未完成的工作，例如对多线程服务语义重构功能的完善等。

7.2 未来展望

本文主要研究实现了基于服务容器的服务语义重构系统，主要重构对象是 Java 服务。针对本文所实现的系统，未来有以下工作可以进一步改进：

1) 本系统主要针对函数调用在 main 函数中的单线程 Java 服务，对 Java 容器多线程的服务语义重构还存在一些问题，例如线程池对线程新建和销毁会影响服务语义重构程序的正常运行等。
2) 本系统实现了函数级的服务监控，基于物联网运行保障系统的进一步拓展，未来应当可以在监控函数的时候，对验证未通过的服务实现函数级

的控制。

参考文献

[1] 张硕，贺飞. Research Advance in Runtime Verification[J]. 计算机科学，2014.41(z2)

[2] 王珍，叶俊民，陈曙，辜剑，金聪. Research on Parameterized Runtime Monitoring[J]. 2014.41(11)

[3] Malik S. Runtime verification:a computer architecture perspective[A].Springer Berlin Heidelberg, 2012.49-62.

[4] Yellin F, Lindholm T. The Java Virtual Machine Specification[J]. Addison-Wesley, 1996.

[5] Laws S, Combellack M, Feng R, et al. Tuscany in Action[J]. MEAP Began, 2009.

[6] 周志明. 深入理解 Java 虚拟机: JVM 高级特性与最佳实践[J]. 2013.

[7] 张瑜, 刘庆中, 李涛, 吴丽华, 石春. 内存取证研究与进展[J]. 软件学报, 2015, 26(05): 1151-1172.

[8] 李文杰. Java 程序内存使用分析技术研究[D]. 中国矿业大学, 2016.

[9] 章婧, 卢凯, 周旭. Java 程序内存行为研究[J]. 小型微型计算机系统, 2011, 32(08): 1617-1621.

[10] 任国力. 基于 VMI 的入侵检测系统的研究与实现[D]. 华南理工大学, 2014.

[11] 李永波. 基于 KVM 的虚拟机自省系统设计与实现[D]. 西安电子科技大学, 2015.

[12] 范淋淋. 面向虚拟化的物联网可靠性保障系统网络保障方案的研究与实现[D]. 北京邮电大学, 2015.

[13] 戴弘扬. 面向虚拟化物联网服务系统运行时验证的方法研究与实现[D]. 北京邮电大学, 2017.

[14] Vladimir Šor, Satish Narayana Srirama. Memory leak detection in Java: Taxonomy and classification of approaches[J]. The Journal of Systems & Software, 2014.

[15] 欧阳强强. 基于工业物联网的高炉热负荷监测系统[D]. 安徽工业大学, 2016.

[16] 季杰. 工业物联网络安全技术[D]. 江南大学, 2012.

[17] 张立超. 基于 Java 的 IoC 容器的设计与实现[D]. 吉林大学, 2009.

[18] 赵明彪. 基于 J2EE 的 Java 应用程序安全[D]. 电子科技大学, 2006.

[19] 徐志强. Java 虚拟机内存管理及其实时性的研究[D]. 江南大学, 2009.

[20] 林巧民. 虚拟机相关技术研究及实践[D]. 河海大学, 2004.

致谢

光阴似箭，我在网络服务基础研究中心的两年半研究生生涯已经到了尾声，刚刚得知录取结果的那一刻仿佛还在眼前。两年半以来，收获了很多很多，这离不开许多人的帮助。在此，我想对所有帮助过我的老师们、同学们和朋友们表示最真诚的感谢。

感谢我的导师章洋副教授。章老师治学严谨，对待研究工作一丝不苟。在他的谆谆教导下，我对科研有了初步的了解和认识。在刚接手题目的那段时间，由于对相关内容不够了解，相关资料不够充分，无法给出很好的解决方法。在我困难重重的时候，章老师以极大的耐心引导和鼓励我不断的调研和探索。在尝试多种方法之后，才有了论文中所阐述的方案。在实验和论文阶段，当思路受阻遇到问题时，章老师总能指导我找到解决办法克服难题。

感谢戴弘扬学长和毛艳芳学姐以及其他学长学姐。戴弘扬学长指导我 Volatility 和 LibVMI 的相关使用方法和原理，毛艳芳学姐调研了很多 Java 相关的资料，也都倾囊相授。没有他们在之前研究打下的坚实基础，我也不可能在内存获取以及 Java 相关很快的入门。

感谢陈宽、韩波、孙华栋、李启波等同学。他们和我一起奋斗，共同科研，互相帮助，互相扶持，探讨学习，交流进步。陈宽同学的研究内容与我的相辅相成，经常给出很好的建议，使我的研究得以更顺利的进行。

感谢我的父母亲人和朋友对我的支持和鼓励。他们一直给与我无限的关爱和帮助，在最困难时刻帮助我重新树立信心。他们也是我努力前进的动力。

感谢各位参加评审和答辩工作的老师们在百忙之中抽出宝贵时间来评阅本文。

最后，再次对所有帮助、关心、鼓励、支持我的人致以最崇高的敬意和最真挚的谢意。