文章编号:1671-4598(2018)09-0232-10 DOI:10.16526/j.cnki.11—4762/tp.2018.09.050 中图分类号:TP311 文献标识码:A

基于性能分析的自适应插桩框架

王子鹏[1,2], 张树东[1,2], 任仲山[1,3], 胡建亚[1,2]
(1. 首都师范大学 信息工程学院, 北京 100048; 2. 北京市成像技术高精尖创新中心, 北京 100000;
3. 中国科学院软件研究所, 北京 100223)

摘要: 插桩技术用于跟踪获取软件系统的运行时信息, 是软件性能管理工具中不可或缺的一个部分; 目前, 存在的各类插桩工具所产生的插桩点在目标程序运行的过程中往往是不可改变的; 在实际的程序异常、错误检测中, 用户关心的程序代码的位置在不同阶段往往不同, 因此目标程序运行的不同阶段所需要获取信息的插桩点位置是不同的, 如果插桩点过多会导致插桩工具浪费系统资源, 产生系统资源消耗过大的问题; 插桩点过少则会导致无法准确定位目标软件发生异常位置; 针对以上问题, 使用基于线性回归和K—Means的分析模型, 分析目标软件的性能数据, 在其运行过程中动态地改变插桩点, 尽可能的减少资源的消耗; 另外, 采用朴素贝叶斯分类模型对插桩类进行筛选, 减少植入插桩点的类, 可降低插桩带来的资源消耗; 实验表明: 与传统的工具相比, 使用此插桩框架进行监控, 被测网页的平均响应时间减少6.88%, 同时对目标程序的干扰更小。

关键词: 插桩; 自适应; 性能数据; 机器学习

Adaptive Instrumentation Framework Based on Performance Analysis

Wang Zipeng[1,2], Zhang Shudong[1,2], Ren Zhongshan[1,3], Hu Jianya[1,2]
(1. College of Information Engineering, Capital Normal University, Beijing 100048, China;
2. Beijing Advanced Innovation Center for Imaging Technology, Beijing 100000, China;
3. Institute of Software, Chinese Academy of Sciences, Beijing 100223, China)

Abstract: Instrumentation is an indispensable part of the application performance management (APM) and used for tracking and obtaining the information of software system. Nowadays, there are various instrumentation tools, and the probes of the tools are unchanged during target application operation. However, in daily exception and bug detection, users require different code monitoring in different periods of a target application operation, and too many probes may have the result that the instrumentation tool wastes and consumes more system resources; too few probes may result that APM can't locate accurate positioning failure of target software. We proposes a new adaptation instrumentation framework based on analyzing performance data and carries out adaption instrumentation based on liner regression and K—Means, changes the probes dynamically during operation and reduces the resource waste as much as possible. What's more, this paper uses Naive Bayes to filter Java classes for reducing instrumentation classes and overhead; In comparison to the traditional tools, this paper leads to low overhead and disturbance through the experiment, average page response time reduced by about 6.88%.

Keywords: instrumentation; adaptive; performance; machine learning

0 引言

插桩是一种获取软件状态的方法, 是软件性能管理工具的核心部分, 常常使用于对大规模分布式系统的监控和跟踪[1], 一般实现方法是程序员对被监控软件系统进行代码指令注入, 这些被注入的代码可以实现各种自定义功能, 例如: 记录功能函数的执行时间、调用序列、植入回调函数、收集所需的信息并将数据记录到数据库中等。现如今大多数的插桩工具使用的是动态二进制插桩技术, 例如DTrace[2]、Pin[3]等工具, 但这些工具通常只提供插桩本身的实现, 并不会提供选择插桩点的和插桩粒度的工作, 这就导致当一些性能管理工具使用插桩技术时难以选择适宜的插桩点和控制插桩粒度。传统的性能管理工具选择全插桩的策略, 即将插桩点植入到被监控软件系统(下文称作目标程序)所有运行的类中, 这就导致了大量的插桩点产生了极大系统资源消耗, 并且插桩点在目标程序运行的过程中是完全不会改变的[4-5]。在对目标程序的实际监控中, 并不是所有的类都需要插桩, 如果对一个目标程序的插桩粒度过粗, 将可能导致达不到用户对目标程序性能管理的要求, 例如难以定位软件发生异常的具体位置。如果对一个目标程序的插桩粒度过细, 将产生大量的资源消耗, 甚至影响到目标程序本身的运行。所以一个根据实际需要, 动态自适应地改变插桩粒度的插桩框架不仅能够满足用户的需求, 而且能够把资源消耗减少到最小, 是十分必要的[6-7]。针对此问题, 本文提出了一种全新的自适应插桩框架。

本文针对Java应用程序, 提供一个基于机器学习的自

收稿日期:2018-02-27; **修回日期**:2018-04-03。
基金项目:国家重点研发计划项目(2017YFB1400800);北京市创新团队建设计划项目。
作者简介:王子鹏(1993-),男,甘肃定西人,硕士研究生,主要从事分布式系统监控、机器学习方向的研究。

适应插桩架构[8]，可以针对软件性能管理工具进行适当的插桩工作；本文可以识别不同来源的 Java 类，将插桩点植入到用户关心的类中。本文还提出插桩粒度矫正机制[9]，在被测程序运行时对插桩粒度进行动态控制，控制整个系统插桩资源的消耗。

1　研究动机

本文的插桩架构是为性能管理软件服务的组件，它可以在极小的资源消耗的情况的下进行插桩，得到跟踪数据。本文的主要目标有以下两点：1）对不同来源的 Java 类进行识别，将插桩点植入到用户编写类中。2）在被测程序运行中动态的对插桩粒度进行矫正。接下来，通过一个普通的例子，对问题进行建模。有一个简单的 Java 编写的 Web 客户端 Client，当此客户端运行的时候，性能管理工具就需要获取此程序的跟踪信息，此时就需要插桩工具对此程序进行插桩。为了方便表述，这里将相关的概念和术语进行引入和说明。

$C=\{c_1, c_2, \cdots, c_n\}$ 表示插桩类，即在被测程序中，要将跟踪代码注入到哪些类当中。

$M=\{m_1, m_2, \cdots, m_n\}$ 表示插桩方法，即是在 c 中，将跟踪代码注入到哪些方法中。

p 表示插桩点，即是在 m 中，将跟踪代码注入的具体位置。每两个 p，对应一个集合 $S_n=\{p_s, p_e\}$，S 是插桩点的作用域，表示这个插桩点的作用范围，p_s 为插桩起始位置，p_e 为插桩结束位置。一个 S_n 也可以看作一个插桩探针，$S=\{s_1, s_2, \cdots, s_n\}$。

$I=\{i_1, i_2, \cdots, i_n\}$ 表示插桩的内容，即是在插桩点上，被注入的具体代码。

C 中每个 c_n 包括一个集合 m，每个 m_n 包括一个集合 s，每个 s_n 包括一个集合 i。整个插桩信息的关系呈现出一个树状的关系，如图 1 所示。

当被测程序运行时，插桩要进行如下步骤：

1）确定被测程序中所有的插桩类集合 C；

2）筛选并确定 c_n 插桩方法的集合 M；

3）筛选并确定 m_n 中所有的插桩点作用域集合 S；

4）确定插桩点 s_n 的中的 p_s 和 p_e；

5）确定每个插桩点要注入的代码集合 I；

6）进行插桩。

当程序运行时，有：

$$\begin{aligned} C = \{ & c_1 = Client, \\ & c_2 = Java.net.Socket, \\ & c_3 = Java.io.BufferedReader, \\ & c_4 = Java.io.PrintWriter, \\ & c_5 = Java.io.InputStreamReader, \\ & c_6 = Java.io.IOException\} \end{aligned}$$

其中类 c_1 为用户自定义的类，c_2，c_3，c_4，c_5，c_6 属于 Java 提供的类库中的类。在传统的插桩方法中，不会区分用户自定义的类和非用户自定义类，选择全插桩的策略，即对所有类进行分析和插桩。但是，全插桩的策略会造成极大的资源消耗和浪费，而且用户在目标程序实际运行中，不关心其他类运行的情况。所以为了解决此问题，本文提出了一种基于朴素贝叶斯算法的分类模型，可以区分不同来源的 Java 类，将来自用户编写的类，JDK 中的类，第三方开发的工具类区分开。除此而外，本文能够在日常对目标程序的插桩过程中，依照目标程序的性能变化，自动地改变插桩的粒度，减少插桩的资源消耗。

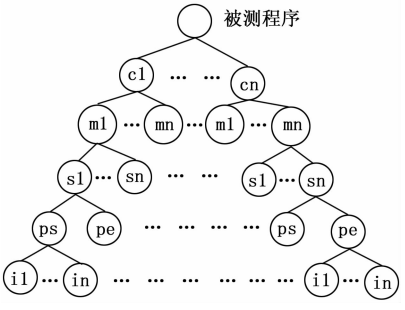

图 1　插桩的树形关系图

2　体系结构

在本章里，本文将主要介绍本文的体系结构以及进行插桩的过程。

2.1　自适应插桩的体系结构

图 2 显示了本文的高层体系结构以及插桩过程。如图所示，自适应插桩体系中，主要有 4 个部分，分别是目标程序，Java 虚拟机，Agent 以及客户端。

目标程序：指的是要被插桩的 Java 程序。

Java 虚拟机：Java 程序运行在 Java 虚拟机上，Agent 在插桩的过程中不直接与目标程序进行交互，而通过 Java 虚拟机完成对目标程序信息的获取以及对目标程序进行插桩。这样可以使得插桩对目标程序本身造成较小的干扰，保证了目标程序本身的独立性和安全性。在图中，JVMToolInterface 是 Java 虚拟机提供的一个工具接口，它是在 JavaSE5 提出的一个工具。JVMToolInterface 提供接口可以获取 Java 虚拟机的状态以及运行在 Java 虚拟机上的程序的状态信息，也可以控制这些程序的执行，具有广泛的用途。本文则通过 JVMToolInterface 来获取目标程序的信息。JavaInstrumentation 是 JavaSE6 的一个新特性，它最大的特点在于可以在目标程序运行时动态的将目标程序的类进行修改，本文则利用它的这一特点，对目标程序进行动态二进制插桩。

Agent：Agent 是整个插桩体系结构中最核心的部分，绝大部分的功能都在 Agent 中实现，主要包括了以下几个部分：

装载器：装载器主要与 Java 虚拟机进行交互，它的主要功能是提取运行在 Java 虚拟机中的程序的信息，主要是

提取目标程序要运行所需的全部类，并将这些信息写入目标程序类表。

分类器：分类器的作用就是将不同来源的三种 Java 类区分开，对用户关心的类进行插桩，这种方式可以极大地减少资源的消耗。分类器初步的确定了插桩类集合 C，分类器的具体实现将在第三章进行详细说明。分类器将判定为用户编写的类的写入用户类表中。

分析器：分析器的主要功能对分类器初步确定的插桩类集合 C 进行分析，得出每个类的插桩方法集合 M，作用域集合 S，插桩点集合 P，插桩内容集合 I。

插桩表：插桩表是本文 Agent 的核心数据表，它保存了在目标程序中所有的插桩点。除了分析器对它进行写入外，日常由自适应器对它进行操作。执行器会依照插桩表中的信息去实施插桩。

执行器：执行器的主要功能是依照插桩表中的内容进行实际的插桩工作。执行器是利用 Javassist 类库完成的[10]。它提供了对 Java 程序字节码进行操作和修改的方法，可以在 Java 程序运行的过程中对程序类进行修改，优点在于简单和快速。本文执行器使用 Javassist 类库，通过 JavaInstrumentation 接口对目标程序字节码进行修改，达到探针注入的目的。

数据回收器：数据回收器的主要工作就是回收目标程序的性能数据信息，数据回收器会依照一定的采样策略去回收数据。

自适应器：自适应器是本文 Agent 的核心部件，它的主要工作是动态地调整插桩粒度，达到减少资源负载目的。自适应器的工作时期是在插桩程序日常运行的阶段。自适应器是根据目标程序的性能数据反馈，利用基于机器学习的算法处理和分析性能数据信息，找出异常的程序执行路径，对其进行更细粒度的插桩，定位异常根源，具体的实现将在第四章中详细说明。

客户端：客户端的主要功能是向用户展示目标程序的性能数据。

2.2 插桩过程

本文将 Agent 对目标程序的插桩过程分为两个主要阶段：1）初始化阶段，2）自适应阶段。如图 2 所示，初始化阶段用点虚线表示，自适应阶段用实线表示，初始化与自适应都要进行的共同阶段用短划线表示。其中，初始化阶段的执行步骤用圆括号表示，例如：[2]，自适应阶段的执行步骤用尖括号表示，例如：＜1＞。

1）初始化阶段。

初始化阶段主要指的是当本文 Agent 第一次插桩一个目标程序的某个类所要经历的阶段，本文将这个过程称为初始化过程。当一个 Java 程序运行时，如果它的源代码是 Java 格式的文件，它的源代码将会编译成为 .class 格式的文件并加载在 Java 虚拟机上运行，如果是 .class 格式的文件，将直接加载在 Java 虚拟机上运行（步骤 [1]），这是所

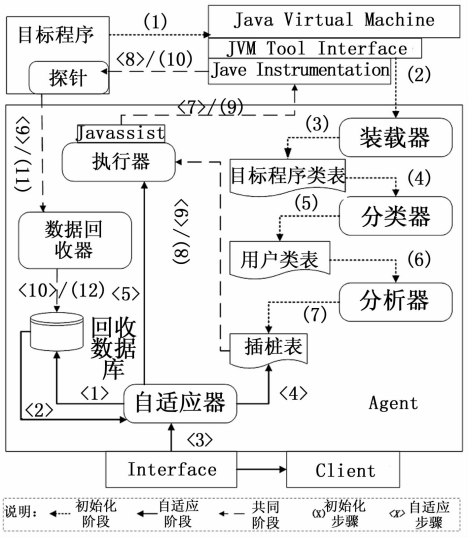

图 2 自适应插桩的高层体系结构以及插桩过程

有 Java 程序运行的必经阶段。当目标程序运行之后，本文 Agent 进入初始化阶段，本文 Agent 的装载器通过 JVMToolInterface 将目标程序的所有类信息获取（步骤 [2]），并存入目标程序类表（步骤 [3]）。当 Agent 获取到目标程序类之后，分类器开始工作，从目标程序类表中读出这些数据并进行分析（步骤 [4]）），分类器将利用基于机器学习的算法将这些类分类，并找出其中属于用户自定义编写的类，并把这些类的信息写入用户类表中（步骤 [5]）。然后分析器读出用户类表中的用户类信息（步骤 [6]）。此后，分析器对这些类进行分析，得出具体的插桩信息，这些信息包括插桩类集合 C，插桩方法集合 M，插桩探针集合 S，插桩点集合 P，以及插桩内容集合 I。并将这些信息数据写入插桩表中（步骤 [7]）。当得到插桩信息后（步骤 [8]），执行器开始工作，将通过 JavaInstrumentationInterface 把信息交给 Java 虚拟机（步骤 [9]），此时将重定义目标程序的类，如图 2 所示为步骤 [10]，探针被注入进目标程序。当目标程序进入到特定的阶段后，探针将返回数据到数据回收器（步骤 [11]），数据回收器通过一定的插桩策略进行采样，将目标程序的性能数据写入到回收数据库中（步骤 [12]）。此时则插桩的初始化阶段完成。

2）自适应阶段。

在完成初次插桩之后，为了更快、更准确、更高效地发现应用中潜在的问题，需要根据监测数据对插桩的粒度进行动态调整，这是因为当对一个目标程序的插桩粒度过粗，将导致无法准确定位发生异常的具体位置；如果对一个目标程序的插桩粒度过细，会导致应用性能严重下降，同时会带来大量的资源消耗。所以本文在目标程序正常启动运行后会进入自适应调整阶段，对插桩粒度进行动态的调整，达到满足性能管理需求的同时将插桩资源消耗减少的最小。

当 Agent 完成初始化阶段之后，当执行初始化阶段的

插桩函数时，将能够收集到目标程序的性能跟踪数据。此时，自适应器将请求读取并得到这些数据，如图 2 所示为步骤＜1＞，步骤＜2＞。自适应器将会对这些性能跟踪数据进行分析，通过基于机器学习的算法，会得出一个正常性能达标范围，当一段目标程序的跟踪路径的性能数据不在这个范围之内时，就说明这个跟踪路径可能出现了性能问题，此时自适应器会将此跟踪路径的插桩粒度变细，增加更多的插桩点。相反，对于一段执行路径的性能在设定时间内性能满足要求或当某些有问题的跟踪路径性能恢复正常，自适应器会通过去除一些插桩点的方法将该路径的插桩粒度变粗。自适应器将通过修改插桩表完成这些变更，如图 2 所示为步骤＜4＞，并通知执行器插桩信息发生了改变，如图 2 所示为步骤＜5＞，此后，执行器会执行调整后的插桩表，和初始化阶段中的步骤相同，为步骤＜6＞，步骤＜7＞，步骤＜8＞，步骤＜9＞，步骤＜10＞。除此而外，用户也可以通过客户端手动修改插桩粒度。

本文通过在应用执行中自适应调整插桩的粒度，实现了在最小的插桩资源消耗下，满足用户对目标程序的性能管理的需求。

3　实现

本文的插桩架构是针对 Java 程序，在本章内将详细介绍本文核心功能部分的实现，依次将介绍 1）基于朴素贝叶斯算法的分类器的实现[11]。2）插片式分析器的实现。3）基于机器学习组合算法的性能自适应器的实现[12-13]。

3.1　基于朴素贝叶斯算法的分类器

对于 Java 程序来说，其代码一般来自 3 个部分。开发人员自定义编写的代码，调用 JDK 类中的代码以及借助第三方提供的库。它们有各自的类，在 Java 程序运行时共同作用，形成完整的 Java 程序。但是，对于性能管理工具来说，在程序的实际运行中，用户通常只关心他们自己编写的类的性能表现，而传统的性能管理插桩策略是对所有的类进行插桩，这样就造成了大量的资源消耗。所以，在插桩前，将用户自定义编写的类，系统类库中的类以及第三方类库中的类区分开，只插桩用户自定义的类，这样可以很大的减少插桩资源负载，增大插桩工具的实用性，是十分有必要的。

3.1.1　采集数据

对于本文的类分类器来说，数据就是一个个类的信息，为了使数据能够有足够的代表性和得到广泛的认可，本文完成了对常用的 Java 开源类库中类信息数据的采集，分别对自定义类、系统类库、第三方插件库中的类进行了采集；包括第三方插件库 Maven、Solr、Commons Math、CommonConfiguration 等在内，一共收集到 77 000 多个类的信息。

3.1.2　特征选择

本文的目的是为了区分不同来源的 Java 类，为此将尽可能选取有明显区分意义的特征。

1）对于一个 Java 类来说，程序通过类名对其进行调用，类名也是对不同的类进行区分的最直接的标志，所以选取类名为第一个特征。

2）在 Java 程序的运行机制中，源代码被编译为 .class 格式的文件进行加载使用，每一个 .class 文件通常代表一个类，当程序需要使用某个类时，Java 由类加载器来加载此类。在 Java 的运行机制中，有双亲委派模型，由三种类加载器来加载不同类型的类，分别是启动类加载器，扩展类加载器，应用程序类加载器。启动类加载器加载路径＜JAVA _ HOME＞/lib 中的类，扩展类加载器加载路径＜JAVA _ HOME＞/lib/ext 中的类，应用程序类加载器加载其他类，由此可分辨出一个类是否为 JDK 中的类，其加载器的类型选取为第二个特征。

3）保存 Java 类信息的文件会存放在某个路径下，一般情况下，系统类存放在＜JAVA _ HOME＞下，第三方类库的类文件单独存放或者和用户自定义类存放在程序的目录下；由于不同类型的类文件一般存放在不同的路径下，因此本文将 Java 类存放的文件的路径选取为第三个特征。

4）在程序的目录下，大多数情况下，不管是自定义类文件还是第三方插件的类文件都会保存在 Jar 格式的压缩包中，但仍然有少数类文件不在 Jar 包中，此类型一般属于用户自定义的类，选取为第四个特征。

5）一般来说，对于一个存放类文件的 Jar 包，如果名字可以和程序的名称匹配或相似，这个 Jar 包中的类文件则是由用户自定义开发的。类文件所在 Jar 包的名称可以选取为第五个特征。

6）对于程序来说，生成项目会改变此程序的文件，从而改变了这些项目文件的最后修改时间，查看一个类文件的最后修改时间是否和程序一致，也可作为判断类文件是否属于程序的特征。

3.1.3　朴素贝叶斯分类模型

朴素贝叶斯分类模型是基于贝叶斯定理的分类模型，每个特征都相互独立。朴素贝叶斯分类器的优点在于算法简单，需要的训练数据量较少，且在实际的分类中处理中资源负载很小，它的分类过程如下：

1）对于一个数据样本 $X=\{x_1, x_2, \cdots, x_n\}$，其中 $x_1, x_2, \cdots\cdots, x_n$ 表示数据样本 X 的 n 个特征。$C=\{c_1, c_2, \cdots, c_n\}$ 表示 n 个类的集合，数据样本 X 可能属于的 C 中的某个类。对于未标记的数据样本 X，其属于类 c_k 的概率为 $P(C_k \mid x_1, x_2 \cdots, x_n)$。

2）根据贝叶斯定理：

$$P(C_k \mid x)=\frac{P(x \mid C_k)P(C_k)}{P(x)} \tag{1}$$

$$P(C_k \mid x_1, x_2, \ldots, x_n)=P(C_k)\prod_{i=1}^{n}P(x_i \mid C_k) \tag{2}$$

3）根据数据集可以训练得出：

$$P(x_1 \mid C_i), P(x_2 \mid C_i), \ldots, P(x_n \mid C_i) i \in$$

$\{1,\ldots,n\}$

4）对于未知样本数据 X，其分类为 y，$y \in C$

$$y = \text{argmax} P(C_k) \prod_{i=1}^{n} P(x_i \mid C_k) k \in \{1,\ldots,n\} \quad (3)$$

3.1.4 分类器的实验和评估

将采集的数据 67 000 余条用于学习，10 000 条用于评估。实验结果如表 1 所示。

表 1 贝叶斯分类器学习和预测结果

类别	学习个数	预测结果	标记结果	准确率/(%)
用户类	33 972	4 172	5 766	72.35
系统类	18 797	2 747	2 747	100
第三方类	8 756	3 081	1 487	48.26
总计	61 525	10 000	10 000	84.06

贝叶斯分类器对 Java 类进行区分时，正确预测用户类 4 172 个，系统类 2 727 个，第三方类，1 487 个，总计个数 8 406，准确率可达到 84%，对系统类的预测准确率较高，准确预测所有的系统类；可以将所有的第三方类标记成功，但是对用户类的预测率较低，5 766 个用户类只预测出4 172 个，真实预测准确率为 72. 35%，将一些用户类预测为第三方类。其原因主要为一些用户类的 Jar 包的名字并不与项目名称匹配或相似，且没有存放在目标程序的路径下，最后修改时间与主项目不一致。遇到此种情况时，在实际中，可以手动将一些 Jar 包信息加入用户类中。

3.2 插桩分析器的实现

Pinpoint 的是一个开源的软件性能管理软件[5]，在它的插桩部分，对软件不同的部分采用不同类型的插桩插件进行对应分析，一些常用组件都有与之对应的插件进行插桩分析，例如：http，mysql－jdbc，Spring 等插件。本文借鉴了 Pinpoint 的插桩思路，采用插件式的分析器组件。目标程序不同的部分都有与之对应的分析插件进行分析，所有插件分析的结果之和构成目标程序初始化阶段插桩分析的结果。一般情况下，在初始化分析阶段，本文倾向于使用尽可能粗的插桩粒度。

采用基于插件组合的插桩分析的策略，既能够保证插桩的专业性，也能够通过增加插件来实现扩展性，从而使得本文的方法具有很强的灵活性。

3.3 自适应插桩器的实现

3.3.1 插桩粒度

一个软件可以实现不同的功能，例如通过网络传递消息或者和数据库交互，软件实现不同功能的代码执行路径不同，插桩粒度的情况也不相同。实现不同功能都应该有不同的插桩粒度等级，本文采用插件式的插桩粒度等级，即不同插件的插桩粒度等级都有专门的划分。

3.3.2 消息传递模型的插桩粒度

接下来将通过一个用户请求浏览网页的例子，将说明 http 消息传递的插桩粒度等级。此粒度等级不仅适用于 http 消息传递程序的插桩，也适用于分布式环境下大部分程序的其他类型消息传递的插桩。

如图 3 所示，一个用户客户端请求访问一个网页，此网页由两部分的内容组成，文字和图片，分别来自不同的服务器。当一个客户端发送一个浏览请求 Request1 时，首先到达网页服务器，此时网页服务器会发现需要文字和图片的数据，于是发送请求 Request2 和 Request3 到文字服务器和图片服务器，文字服务器和图片服务器分别立即做出反应，将消息返还给网页服务器，分别是 Reply2 和 Reply3。然后网页服务器响应最初的请求，将请求返还到客户端 Reply1。

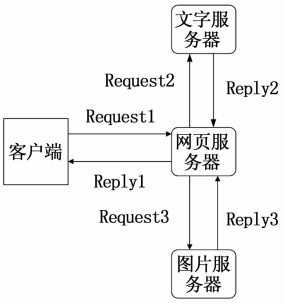

图 3 简单的 http 消息传递

以上是一个简单分布式消息传递过程，对于分布式程序的跟踪实现，Google 提出的分布式系统跟踪框架 Dapper 给予后来的开发者很大的启发[1]。在具体的分布式消息跟踪实现上[14-15]，本文也采用 Dapper 的分布式跟踪理念，将一个完整消息请求和返还链路称为一个 trace。

本文在对分布式系统消息传递插桩的最粗粒度为 trace 等级，即记录完成一个完整 trace 所需的时间和其他信息。

第二等级的插桩粒度是 server 等级，即记录一个服务器收到消息到发出消息的时间和其他信息。例如：图 3 中文字服务器接收到 Request2 时间到发出响应 Reply2 的时间。

第三等级的插桩粒度称为 method 等级，即记录一个方法执行所需的时间和其他信息。

以上三种等级的插桩粒度在目标程序的执行过程中随着程序的执行情况改变，在达到用户要求的情况下，确保最小的插桩资源负载。

3.3.3 特征选择

本文的性能自适应插桩器是利用目标程序的性能数据作为反馈，使用基于机器学习的组合算法进行处理，找出异常的性能数据，改变其执行路径的插桩粒度。本文希望可以在最小的插桩资源消耗下找出异常情况，即尽可能少的提取特征去判定。

特征提取：在分布式系统的消息传递中，消息的响应时间是衡量目标程序的某个执行路径是否出现异常的最主要因素。所以选取数据传输响应时间为主要特征。

假设一个 trace 的总响应时间为 T，这个 trace 的消息要经过 n 个节点的处理，那么有：

$$T = \sum_{i=1}^{n}(t_i + p_i) \tag{4}$$

其中：t_i 表示第 i 个节点传输数据的响应时间，p_i 表示第 i 个节点上程序处理请求所需要的时间。假设 $P = \sum_{i=1}^{n}(p_i)$，对于不同的 trace 来说 P 是不相同的，因此无法横向评估不同 trace 的时间数据。但是，对同一个 trace 来说，P 大致上相同的，本文则通过同一个 trace 的历史数据去评估此 trace 是否异常，此时有：

$$T = \sum_{i=1}^{n}(t_i) + P \tag{5}$$

P 对于同一个 trace 的历史数据来说可以看作为一个常量。

影响数据传输响应时间的因素有很多，例如服务器硬件性能，网络带宽等等。但是在同样的环境下，影响数据传输响应时间的因素主要为数据包的个数和大小，数据包越多，数据量越大传输所需要的时间越长。

在实际的传输过程中，传输时间受数据包大小和数据包个数的共同影响，数据包越少，传输时间越短。但是，并不能将所有数据都分在一个数据包内，因为最大数据包的大小受到最大传输单元的限制（MaximumTransmission-Unit），以下简称为 MTU，MTU 的单位为字节，当所传输的数据量大于一个 MTU 时，此时发送的数据就会被分为多个包，每个包的大小都不大于 MTU。

如图 4 所示，图中为 trace 总响应时间和数据包大小的关系，其中 X 轴代表数据包含有 MTU 的个数，Y 轴代表 trace 响应的时间。MTU _ s 是发送一个足够小的数据包需要的时间，MTU _ e 代表发送一个大小为 MTU 的数据包所需要的时间。当 x 的区间为（k，$k+1$）时，响应时间 Y 与 x 成正比例递增关系，每当数据量超出一个 MTU 时，响应时间就会因为多出一个数据包而增加 MTU _ s，趋势呈现阶梯形增长。

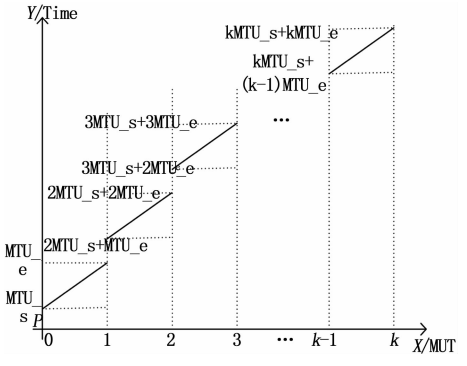

图 4　trace 总响应时间和包大小的关系

当一个 trace 中传输的数据量大小超过 MTU 时就会被分成 k 个尽可能大的子包，每个子包的大小不超过 MTU，其总响应时间在区间之间。

影响 trace 的响应时间的因素分别是数据量的大小，包的个数与 MTU 的大小。选择数据量的大小为第二个特征。MTU 的大小在一般的生成环境中是固定不变的，对其视为常量，不选为特征。

在实际的生产环境中，传输数据时会设有数据缓冲区，当缓存的数据足够接近一个 MTU 大小时才会发送。假设传输数据量为 $Size$，数据包的个数为 Num，则 Num 有以下式子表示：

$$Num = \left[\frac{Size}{MTU}\right] + 1 \tag{6}$$

虽然此时得到的 Num 会与实际 Num 有一定误差，但误差尚在可容忍范围内。若选取数据包数量为特征，则需要大量的插桩点获取数据，会造成很大的资源负载。综合考虑到实际插桩资源的消耗，尽可能的减少插桩点，所以不选数据包数量为特征。

3.3.4　性能数据线性回归模型

由上一节可知，响应时间和数据传输量是分布式系统消息传递的特征，采集到的 trace 的性能数据将是一个响应时间和数据传输量的数据集合，记为：

$$S = \{(x_1, y_1), (x_2, y_2), \cdots, (x_i, y_i)\}$$

其中：（x_i，y_i）表示当数据量大小为 x_i 个 MTU 时，其响应时间为 y_i。当 x 发生改变时，y 随着 x 的改变而改变，当 x 在区间为（k，$k+1$）内时，其中的正常数据点可以通过一元线性回归模型拟合成一条直线段，称为性能拟合回归线，如图 5 所示。

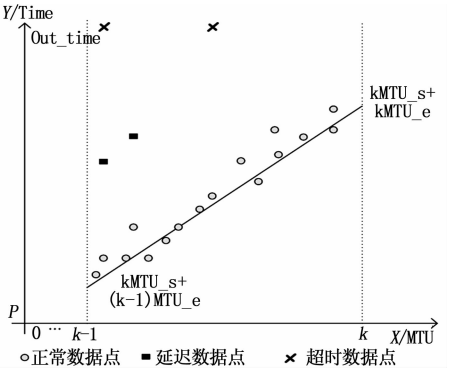

图 5　性能拟合回归线

线性回归模型的模型函数为：

$$y = \alpha + \beta x + \varepsilon, \varepsilon \sim (0, \sigma^2) \tag{7}$$

对以上参数利用最小二乘法进行估计：

$$\beta = \frac{\sum_1^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_1^n (x_i - \bar{x})^2} \tag{8}$$

$$\alpha = \bar{y} - \beta\bar{x} \tag{9}$$

其中：

$$\bar{x} = \frac{1}{n}\sum_1^n x_i \tag{10}$$

$$\bar{y} = \frac{1}{n}\sum_1^n y_i \tag{11}$$

有 r 表示样本 x 与 y 之间的相关系数，用来衡量拟合的

回归的好坏。

$$r=\frac{\sum_1^n(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_1^n(x_1-\bar{x})^2\sum_1^n(y_i-\bar{y})^2}} \tag{12}$$

回归方程的误差为 ε，$\varepsilon\sim(0,\sigma^2)$，利用离差平方和计算误差：

$$\sigma^2=\frac{\sum_i^n(y_i-\bar{y})^2}{n-2}(1-r^2) \tag{13}$$

3.3.5 噪声处理

如图 5 所示，trace 的性能数据不仅包括正常的数据点，而且包括出现性能异常的数据点。出现性能异常的情况一般包括两种：延迟和超时。延迟指出现性能问题，虽然能够成功传输数据，但是因为硬件环境、网络等缘故出现明显的响应时间过大情况。超时指不能够成功传输数据的情况，一般的超时时间远远大于发送单个数据包的时间。在图 5 中，除了正常的数据点外（在图中用圆形表示），还包括 2 个延时数据点（在图中用矩形表示），2 个超时数据点，（在图中用叉点表示）。超时时间 Out _ time>>MTU _ e>MTU _ s。

当用性能数据拟合成性能回归线时，由于异常数据点的存在，会对拟合成的线造成极大的影响，无法反应正常的情况，所以就需要将这些异常的点或者疑似异常的点去掉，由正常数据点的集合拟合成性能回归线。

K－Means 算法是一种经典的基于距离的聚类算法，采用距离作为相似性评价指标，可以将 N 个数据对象划分为 K 个类，同一类对象之中，对象之间的相似度较高，不同类对象之间的相似度较小。本文在 K－Means 算法的基础上对其进行改进，改变相似度的计算方式，工作流程如下：

1）从 N 个数据对象中随机选择 K 个质心。

2）对其他每个对象计算其到每个质心的相似度 $D(x)=\sqrt{\left(\frac{y_1}{x_1}-\frac{y_2}{x_2}\right)^2}$，并将其归入最相似的那个质心的类中。

3）重新计算每个类的质心

4）重复第 2 步和第 3 步，直到质心不在变化或者变化小于预先设定的阈值。

除此而外，使用改进的 K－Means 算法，简单、快速，造成的资源负载较小。

3.3.6 数据预处理

为了方便对性能数据进行判别，就需要对原始数据进行预处理。原始的性能数据将是一个响应时间和数据传输量的数据集合 O，$O=\{(size_1,time_1),(size_2,time_2),\cdots,(size_i,time_i)\}$。

$size$ 表示传输数据量的大小，$time$ 表示响应时间。根据上文可知，仅当数据量大小范围在 $(k,k+1)$ MTU 时，数据点可拟合为一条线段，则记 x 为含有 MTU 的个数。根据上文可知：

$$x=\left[\frac{size}{MTU}\right]+1 \tag{14}$$

x 在每个范围 $(k,k+1)$ 都能拟合成线段，而且拟合的线段斜率相等，因为其拟合线段的相似性，可以归纳得一般的模型，利于处理，此时将要求 x 的范围 (0，1)，相应地对 $time$ 项进行预处理得到：

$$y=time-xMTU\_e \tag{15}$$

最后，预处理后的性能数据集合为：

$$P=\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i)\},x\in(0,1)$$

3.3.7 性能数据处理过程

算法输入：原始性能数据集

$O=\{(size_1,time_1),(size_2,time_2),\cdots,(size_i,time_i)\}$

步骤 1：生成预处理性能数据集

$P=\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i)\}$，$x\in(0,1)$

步骤 2：使用 K－Means 算法对数据进行分类，SetK＝3，将原始数据分为三类，一类为正常数据，一类为延迟据，一类为超时数据

Get $P_1=\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i)\}$，$x\in(0,1)$ 为正常数据类

Get $P_2=\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i)\}$，$x\in(0,1)$ 为延迟数据类

Get $P_3=\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i)\}$，$x\in(0,1)$ 为延迟数据类

步骤 3：处理 P_1 数据集，将 P_1 数据集中 y 大于 MTU _ e 的数据点去除，得到数据集合 P_4。

步骤 4：使用 P_4 进行一元线性回归拟合，得到性能拟合回归线：

步骤 5：使用性能拟合回归线对 P_1、P_2 进行判断

```
设置异常数据集 R
foreach(x_i) do
{if) ;
elseR. add(); }
```

Get 为异常数据集合

算法输出：异常数据集合

$$R=\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i)\}$$

3.3.8 自适应器异常判断的实验和评估

本实验数据来源主要是对远程网路地址发送报文，并记录报文大小和响应时间。

经过多次实验，本文实验环境中，MTU 大小为 1 500 字节，MTU _ s 约为 2 ms，MTU _ e 约为 4 ms。向远程服务器发送不同大小的数据包，共发送 6 000 条报文，得到报文大小和响应时间，用基于组合机器学习算法的自适应器进行处理，得到表 2。

基于组合机器学习算法自适应器对性能数据进行预测，正确预测正常点 4 656 个，延迟点 327 个，超时点 344 个，总计个数为 5 327，正确率为 88.78%。其对超时数据点和延迟数据点的都能够完全准确判别。但是对数据的延迟十分敏感，将一些正常的数据点判别为延迟数据点，这是由于

表 2　性能数据处理结果

数据类型	标记结果	预测结果	准确率/(%)
正常数据点	5 329	4 656	88.87
延迟数据点	327	1 000	32.7
超时数据点	344	344	100
总计	6 000	6 000	88.78

在数据处理的过程中，为了处理噪音，尽可能的保留拟合回归线的数据点，得到的误差偏小，将一些波动较大的正常数据点判别为延迟数据点，这样会导致插桩点的数量变多。在实际的插桩过程中，可以根据实际的延迟容忍情况，可以在数据点判别时，适当增大误差值，减少插桩点。

4　实验和评估

在本章，本文将使用自适应插桩架构对的目标程序进行插桩，并与 Pinpoint 的插桩策略在同等的条件下的插桩情况进行对比，评估本文插桩架构对目标程序产生的影响和对系统产生的影响。

4.1　实验对象

为了验证本文插桩架构，实验的目标程序是基于 Java 开源电子商务网站架构 Broadleaf 的购物网站 HeatClinic[16]。HeatClinic 是一个多层架构的网站，主要使用 SpringBoot 框架开发。实验所使用的数据库为 MySQL，网站的部署在 Tomcat 应用服务器上。

4.2　实验设计

本文对 HeatClinic 网站的业务进行实验，将通过压力测试工具 JMeter 对插桩过后的 HeatClinic 网站进行测试，查看不同压力规模下，对目标程序和系统产生的影响。主要查看网页响应时间和吞吐量。网页的响应时间来衡量监控工具对目标程序的干扰情况，吞吐量来衡量插桩资源负载情况。压力测试中，线程在 100 ms 内完成启动。

本文实现两种不同情况的实验：1）正常情况下的实验，即网站没有发生异常。2）异常情况下的实验，即网站出现异常产生响应延时，此种情况可以由向网站注入线程延迟代码模拟。本文将在不同的方法中注入延迟时长不等的延迟点。

4.3　实验结果与分析

4.3.1　插桩对应用性能造成干扰的分析

图 6 显示了在相同并发用户数的情况下，不同插桩策略的网页平均响应时间随着发生异常的变化情况。在目标程序运行初时，Pinpoint 和本文分别对目标程序进行插桩，使目标程序正常运行一段时间，得到目标程序正常运行的性能数据范围。本文插桩架构完成初始化阶段并进入自适应阶段。在程序运行到一段时间后，图中为 a 时间点，动态的对相关方法注入延迟代码，造成异常，产生响应延迟 Δt。使用全插桩策略的 Pinpoint 和本文的插桩不会改变插桩粒度，目标程序网页的平均响应时间增加 Δt。本文的自适应插桩在目标程序发生异常后收集到性能数据，并分析得出程序发生异常，在 a 时后的一段时间，使程序的插桩粒度更细，增加插桩点。此时目标程序的网页平均响应时间增量超过 Δt，达到全插桩策略。在程序运行到 b 时，去除延迟代码，本文自适应插桩收集到性能数据并分析得出程序运行正常，调整插桩粒度，减少插桩点，对目标程序的干扰变小，平均响应时间减少。本文的自适应策略与 Pinpoint 全插桩策略相比，在对目标程序监控时，平均的响应时间更少。

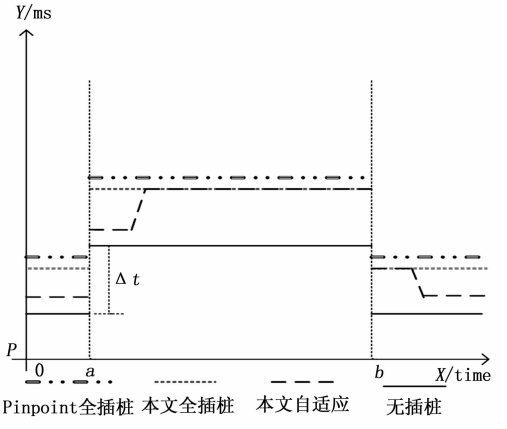

图 6　发生异常时的网页响应时间的变化情况

4.3.2　插桩资源负载分析

图 7 显示了在相同并发用户数的情况下，且计算资源充足，不同插桩策略的网页平均吞吐量随着发生异常的变化情况。在 a 时刻注入异常代码，随着响应时间增大，无插桩以及 Pinpoint 全插桩和本文全插桩的网页吞吐量会受到影响并减少，对于本文自适应策略来说，发生异常初始，吞吐量减少，但随着诊断出异常并增加插桩点后，需求系统资源变多，吞吐量增大。在 b 时刻移除异常后，各个策略的吞吐量均会恢复到正常水平。

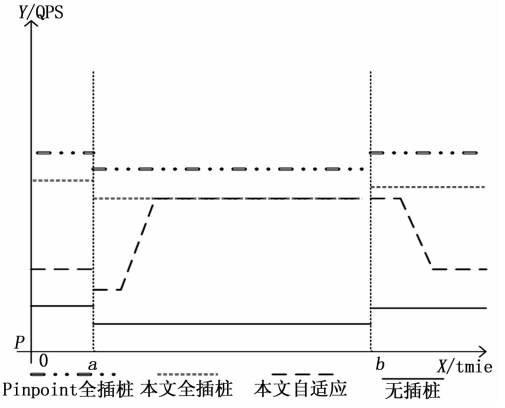

图 7　发生异常时网页吞吐量的变化情况

4.3.3　不同插桩策略对比分析

表 3 展示不同策略下目标程序网页的平均响应时间和吞吐量。使用 Pinpoint 和对目标程序进行插桩，10、50、

100 用户并发平均响应时间比不使用插桩分别多了 22.9%、15.38%、8.98%，使用本文的自适应插桩架构的情况为 13.63%，7.17%，2.81%。使用本文的自适应插桩比使用 Pinpoint 全插桩平均的响应时间要减小 6.88%，本文的插桩策略对目标程序造成的干扰更小。使用本文自适应插桩，在目标程序的运行过程中，平均网页吞吐量更小，平均插桩点比 Pinpoint 全插桩少，需要的系统资源更少。

表 3 不同插桩策略对比

类别/用户并发量	10	50	100
Pinpoint 全插桩	(131|106)	(225|154)	(388|213)
本文全插桩	(125|102)	(221|151)	(379|216)
本文自适应	(115|75)	(222|142)	(366|227)
无插桩	(110|73)	(225|137)	(356|224)

注：(avg | throughput) avg 表示平均响应时间，单位为 ms. Throughput 为平均吞吐量，单位为 QPS

5 相关工作

在分布式系统中，对于分布式应用出现的异常进行诊断和定位是十分重要的。在对这些异常进行诊断的过程中，现今研究主要目标是将插桩的资源消耗和对目标程序的干扰降低到最小，可以使插桩技术在实际的大规模分布式系统的中得到应用。虽然一些插桩工具被一些大公司使用在其生产环境，如 Twitter、Google、京东、阿里巴巴等，但是，到目前为止大部分的研究工作将重点放在了改进插桩过程的本身[17]，或者以不同的原理实现分布式跟踪[18-19]，如 Facebook 最新的工作[20]，达到优化插桩的目的。但是，插桩资源的主要消耗和对系统的干扰来自于产生和收集大量的性能数据，在插桩运行的过程中，自适应调节插桩点可以有效的减小插桩资源消耗与干扰。

现今仍然有一部分研究工作针对自适应插桩，AIM[6] 是针对于软件性能分析的自适应插桩框架，它能够在插桩过程中依靠指令增加和减少插桩点，但是指令却需要通过其客户端人工输入。DOBI[17]兼顾了插桩的精准性、完整性和性能消耗，使用基于方法执行时间的分析的模型进行自适应插桩，但是插桩粒度过细，产生大量消耗。RaceTrack[21]是针对. net 框架下程序异常诊断的自适应插桩框架，主要对线程进行分析，对异常线程加入更多的插桩点，问题在于在大规模的分布式系统中线程众多且复杂，并且难以收集综合分析。APMP[22] 对每个插桩点都产生权重，通过权重分析决策增减插桩点，缺点在于仍然需要大量的信息支持，会造成不少资源消耗。SSSM[24]提出了一种利用性能数据进行响应时间预测的策略，为了判定异常，采用了多个维度、细粒度的响应时间回收策略，进而产生大量的插桩点。本文认为在复杂多变的分布式环境下，很难通过一种或多种预先设定的策略或标准去判断发生性能延迟，使用机器学习算法，学习实际的数据，分析性能曲线，是一种客观有效的方法。

本文提出了一个基于机器学习的自适应插桩框架，在对目标程序运行的过程中，不仅能够动态地进行插桩，而且能够依据目标程序的性能数据动态的调整插桩粒度，在诊断异常的同时，将插桩资源消耗减少到最小。未来的研究方向主要是以下几个方面：1）将对现有性能数据处理的算法进行改进，主要提高对异常数据点识别的准确率和降低处理算法的资源消耗。2）完善自适应规则，对于不同类型的软件所产生的性能数据是不一样，所以就应该有相对应的、更专业的自适应规则去匹配，希望通过插件的形式可以完善自适应规则，保持本文的活力。3）采样策略，虽然本文在极力减少插桩点，但是这些插桩点也会产生很多性能数据，有些性能数据价值较大，有些价值较小[21]，如何在保持性能消耗不变甚至减小消耗的情况将有价值大数据点辨别出来并记录下来，抛弃无价值的数据点也是我们未来的工作之一[25]。

6 结束语

本文提出了一种全新的自适应插桩架构，其核心是利用机器学习算法分析性能数据并根据得到反馈对 Java 程序进行动态插桩，本文的方法依据目标程序实际运行情况动态调整插桩粒度。本文解决了用户在生产环境中应用执行状态追踪和性能监测中的两个主要的问题：第一，用户往往只关心他们自己编写的类的运行情况，而对于其他来源的类，插桩会造成较大的资源负载，所以本文提出基于朴素贝叶斯算法的分类器，只将用户自己编写的类找出并分析它们，减小了插桩资源消耗。第二个问题是传统的插桩工具往往不能令人满意，插桩粒度粗则会导致无法定位异常，插桩粒度细则会导致资源负载过大。本文将基于机器学习的组合算法引入性能数据处理，分析性能数据并动态调整插桩粒度。

参考文献：

[1] Sigelman BH, Barroso LA, Burrows M, et al. Dapper, a large－scale distributed systems tracing infrastructure [R]. Technical report, Google, Inc, 2010.

[2] Cantrill B, Shapiro MW, Leventhal AH. DynamicInstrumentationofProductionSystems [A]. USENIX Annual Technical Conference [C]. General Track. 2004: 15－28.

[3] Luk C, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation [J]. programming language design and implementation, 2005, 40 (6): 190－200.

[4] Zipkin. http://twitter.github.io/zipkin/ [EB/OL].

[5] Pinpoint. https://github.com/naver/pinpoint [EB/OL].

[6] Wert A, Schulz H, Heger C, et al. AIM: adaptable instrumentation and monitoring for automated software performance analysis [J]. automation of software test, 2015: 38－42.

[7] Madsen M, Tip F, Andreasen E, et al. Feedback－directed in-

strumentation for deployed Java Script applications [A]. Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference [C]. IEEE, 2016: 899 - 910.

[8] Salehie M, Tahvildari L. Self－adaptive software: Landscape and research challenges [J]. ACM Transactions on Autonomous and Adaptive Systems, 2009, 4 (2).

[9] Shende S, Malony AD, Morris A. Optimization of Instrumentation in Parallel Performance Evaluation Tools [J]. Lecture Notes in Computer Science, 2007, 4699: 440 - 449.

[10] Chiba S. Javassist—a reflection－based programming wizard for Java [A]. Proceedings of OOPSLA' 98 Workshop on Reflective Programming in C++ and Java, 1998, 174.

[11] Witten IH, Frank E. Data mining: practical machine learning-tools and techniques with Java implementations [J]. International conference on management of data, 2002, 31 (1): 76 - 77.

[12] Kianifard F. Applied linear regression models [J]. Technometrics, 1984, 26 (3): 352 - 353.

[13] Jain AK. Dataclustering: 50 years beyond K－means [A]. international conference on pattern recognition [C]. 2010, 31 (8): 651 - 666.

[14] 袁鑫晨，李海波，王　伟，等. 基于程序分析的分布式应用自动化追踪方法 [J]. 计算机系统应用，2016，25 (11)：35 - 40.

[15] Yang C, Wu S, Chan WK, et al. Hierarchical Program Paths [J]. ACM Transactions on Software Engineering and Methodology, 2016, 25 (3).

[16] Broadleaf online demo. http: //demo. broadleafcommerce.org/ [EB/OL].

[17] Zheng Y, Zhang C, Kell S, et al. Dynamic optimization of bytecode instrumentation [A]. ACM Work shop on Virtual Machines and Intermediate Languages [C]. ACM, 2013: 21 - 30.

[18] Zhang Y, Makarov S, Ren X, et al. Pensieve: Non－Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach [A]. Symposium [C]. 2017: 19 - 33.

[19] Li Z, Gan S, Jia R, et al. Capture－removal model sampling estimation based on big data [J]. Cluster Computing, 2017, 20 (2): 1 - 9.

[20] Kaldor J, Mace J, Bejda M, et al. Canopy: An End－to－End Performance Tracing And Analysis System [A]. The, Symposium [C]. 2017: 34 - 50.

[21] Yu Y, Rodeheffer T, Chen W. Race Track: efficient detection of data race conditions via adaptive tracking [A]. Twentieth ACM Symposium on Operating Systems Principles [C]. ACM, 2005: 221 - 234.

[22] Jeswani D, Natu M, Ghosh RK. Adaptive monitoring: A frame work to adapt passive monitoring using probing [J]. Network and Service Management. IEEE, 2012: 350 - 356.

[23] Ehlers J, Hoorn AV, Waller J, et al. Self－adaptives of twaresystem monitoring for performance anomaly localization [A]. ACM International Conference on Autonomic Computing [C]. ACM, 2011: 197 - 200.

[24] Karianakis N. Sampling Algorithms to Handle Nuisances in Large－Scale Recognition [J]. 2017.

[25] Li Z, Gan S, Jia R, et al. Capture－removal model sampling estimation based on big data [J]. Cluster Computing, 2017: 1 - 9.

(上接第 218 页)

果。通过结果分析，其中 2231、2232、2241、2242、2243、2244、9140 的检测结果良好，从聚类时间分析，六年的数据记录时间为 395.53 s。以此表示，从时间及结果方面，本文所研究的方法具有实际意义[8]。

5　结束语

地震前兆数据观测项较多，并且种类较为繁多，跨越的时间较短，来自于不同技术系统。另外，因为种种原因，导致目前实际数据集较为混乱。实现此数据的联合使用及长时间的数据分析，传统数据分析方法已经无法满足需求。大数据分析思路为使用前兆数据提供了全新的模式，通过此全新的思路，和地震前兆观测物理意义相互结合，能够从其中挖掘有用的规律及信息，对于前兆观测地震及研究其他地震问题都有重要的现实意义。通过大数据研究思想，能够对前兆数据传统使用及研究模式进行创新，不管是前兆数据推广使用，还是使用其进行科学研究，都是有意义的尝试。

参考文献：

[1] 梅莉军. 基于大数据挖掘的地震前兆观测研究 [J]. 科技与创新，2016 (8)：15 - 15.

[2] 王秀英，张　玲，张聪聪. 探讨地震前兆观测中的大数据挖掘与应用 [J]. 震灾防御技术，2015，10 (1)：39 - 45.

[3] 李正媛，熊道慧，刘高川，等. 基于大数据挖掘的地震前兆台网观测数据跟踪分析 [J]. 地震地磁观测与研究，2016，37 (3)：1 - 6.

[4] 王洪伟. 基于 Weka 大数据挖掘方法在地震前兆数据处理中的应用 [D]. 太原：太原理工大学，2017.

[5] 张聪聪. 地震前兆观测数据异常检测方法研究 [D]. 北京：中国地震局地壳应力研究所，2015.

[6] 屈　佳，郑　蕊，王宁. 地震行业“大数据”应用探讨 [J]. 城市与减灾，2014 (4)：24 - 26.

[7] 张晁军，陈会忠，李卫东，等. 大数据时代对地震监测预报问题的思考 [J]. 地球物理学进展，2015，30 (4)：1561 - 1568.

[8] 张　翔，王茂发，史鹏飞，等. 基于大数据技术的地震数据处理新思路 [J]. 地震地磁观测与研究，2017，38 (3)：191 - 195.