基于程序分析的分布式应用自动化追踪方法①

袁鑫晨[1,2], 李海波[3], 王 伟[2], 唐 震[1,2], 任仲山[1,2], 郑莹莹[1,2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院软件研究所 软件工程技术研究开发中心, 北京 100190)
[3](中国电子技术标准化研究院, 北京 100176)

摘 要: 提出了一种零配置的端到端细粒度追踪的方法—EasyTrace, 应用于分布式系统发生性能降级时的问题诊断. 传统工具往往存在监测粒度与服务组件源码紧耦合或配置信息复杂、修改代价高等不足, EasyTrace 能够做到零配置下性能问题的准确定位. 基于开源电子商务系统的实验结果表明, EasyTrace 相对传统监测工具对系统造成的扰动很小.

关键词: 分布式系统; 端到端追踪; 性能问题; 插桩

Auto-Tracing in Distributed Applications Based on Program Analysis

YUAN Xin-Chen[1,2], LI Hai-Bo[3], WANG Wei[2], TANG Zhen[1,2], REN Zhong-Shan[1,2], ZHENG Ying-Ying[1,2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[3](China Electronics Standardization Institute, Beijing 100176, China)

Abstract: A zero-configuration fine-grained end-to-end tracing approach, namely EasyTrace, is proposed, which is applied to the diagnosis of distributed system with performance degradation problem. Compared with the traditional tracing tools, in which granular control and service component source code are tightly coupled or which have complex configuration and high cost of modification, EasyTrace achieves accurate performance problem locating in the premise of zero-configuration. Extensive experiments that use EasyTrace to trace an open source e-commerce system show that compared with traditional monitoring tools, the disturbance caused by EasyTrace is low.

Key words: distributed system; end-to-end tracing; performance problem; instrumentation

1 引言

Amazon 统计数据显示, 网页打开慢 0.1 秒, 客户活跃度降 1%, 页面打开速度每增加 1 秒页面浏览量减少 11%, 用户满意度减少 16%, 网站转化率下降 8%. 如果 Amazon 的网页加载时间增加 1 秒, 每年销售额将损失 16 亿美元[1]. 上层应用的用户访问请求需要经过多种服务组件才能完成. 例如: Google Search 在用户输入检索关键词之后会调用拼写检查、自动补全、及时搜索、广告等一系列服务才会将结果反馈给用户[2]. 这些服务组件由不同的开发团队维护, 部署在数以百记的服务器上, 甚至跨越数个集群. 当用户请求的响应时间急剧增大时, 问题诊断变得极其困难[3]. 问题可能发生在单一服务组件内部或者多个服务的交互过程中. 当线上系统出现抖动时, 如何帮助开发和运维人员快速定位和修复成了亟待解决的问题.

通过对请求添加标记来实现端到端追踪是一种帮助定位性能降级的方法, 也是工业界监测线上系统普遍采用的, 由此诞生了如 Dapper[3]、Zipkin[4]等系统. 开发和运维人员可以通过用户界面查询线上系统状态, 及时发现性能异常和进行异常的诊断. 监测的粒度可以是节点, 也可以是组件甚至是方法级别, 粗粒度的监测有着更好的性能, 细粒度则能更快的诊断修复性能问题.

但传统工具在监测粒度上的调整往往依赖服务开

① 收稿时间:2016-03-09;收到修改稿时间:2016-04-08 [doi:10.15888/j.cnki.csa.005449]

发人员利用 API 对组件代码进行修改或者进行复杂的配置, 这一方面依赖开发人员对于系统的理解, 另一方面同样也耗费人力. 如使用了第三方库, 细粒度监测甚至变得极为困难, 因为第三方库的代码通常不支持修改.

为了解决上述问题, 本文提出了一种零配置的自动化追踪方法——EasyTrace. 其基本步骤为: 1)为 JVM 加载的类创建数据结构追踪插桩过程; 2)分析服务入口字节码中的方法调用指令, 并进行标记; 3)根据标记结果进行追踪代码的插桩. 由此实现了服务入口执行路径的自动化追踪, 避免了人工配置可能引入的错误和维护配置的代价.

本文第 2 节介绍问题模型; 第 3 节介绍 EasyTrace 的主要算法; 第 4 节通过实验验证 EasyTrace 的有效性; 第 5 节介绍并比较相关工作; 第 6 节则对本文的工作以及未来的工作方向进行了总结.

2 问题模型

2.1 相关术语介绍

为了便于后文表述, 这里就相关术语和概念进行说明.

E={e_1, e_2, ...,e_n}表示服务入口, 即服务进程供外界访问的方法, 通常是使用了注解或是某些接口的实现类, 如 JAX-RS[5]注解@Get、@Post 等定义了 REST 的服务入口; Thrift[6] RPC 服务的入口实现了 Iface 或 AsyncIface 接口.

F={f_1, f_2, ...,f_n}表示系统内所有方法组成的集合, 有E ⊂ F.

P={p_1, p_2, ...,p_n},p=(f_1,f_2,...,f_n)表示服务路径集合. 请求到达服务组件到执行结束所涉及的调用序列.

$IF \subset F$表示系统内所有注入监控探针的方法, 对∀if, if ∈ IF，∃p ∈ P使得 if 在 p 上。

I={invokestatic, invokespecial, invokevirtual, invokeinterface}表示 Java 编译器产生的 4 中方法调用指令.

SI={invokestatic,invokespecial}静态绑定方法调用指令, 可以确定唯一的插桩方法.

DI={invokevirtual,invokeinterface}动态绑定方法调用指令, 需要根据根据插桩情况和类加载情况进行动态插桩.

ClassRepository={cm_1,cm_2,…,cm_n}, 存储类元信息, 追踪插桩状态.

ABN={abn_1,abn_2,…,abn_n}表示性能异常集合.

Probe={$probe_1$,probe,,…,$probe_n$}注入的探针的集合, 探针主要包括两个方法调用, before()和 after(), 使用字节码工具嵌入在 if ∈ F方法体前后, 用来生成时间戳, 标记分布式消息和记录异常等.

2.2 服务执行监测建模

对于服务执行路径, p=(f_1,f_2,...,f_n), 我们通常使用调用图来表示路径上方法的调用关系, 并利用注入的探针获取监测数据并在数据库中进行关联.

对于监测获得的路径, p′=(f_1',f_2',...,f_n'),f_i'∈IF, 因为监测粒度的原因, 通常无法获取完整的路径.

调用上下文树(Calling Context Tree, CCT), 使用 CCT 对于监测数据进行表示, 即在节点上记录方法执行的时间戳和执行时间, 除此之外也会附带异常数据库查询的注解, 如图 1. 这里主要关注耗时操作和程序异常, 我们将其称为性能异常.

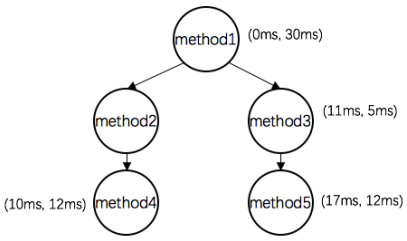

图 1　调用上下文树

对于性能异常, 在粗粒度追踪下, 监测工具并不能准确定位问题所在方法. 例如图 1 中如果 method5 和 method3 都没见植入监测探针, 当 method5 出现性能异常, 监测工具只能将异常标记在 method1 上. 这意味这开发人员需要对 method1、method3、method5 进行审查和测试才能定位异常. 而要使监测探针注入 method5, 通常依赖于开发人员对于系统有着良好的理解, 并使用监控工具的 API 为探针标记位置或者对监测工具进行配置. EasyTrace 的目标即是不依赖开发人员, 即可实现高效的探针注入.

3 基于程序分析的自动化追踪

3.1 方法原理

为了实现细粒度追踪, 通常追踪工具可以提供配置选项, 由用户去声明需要注入探针的方法名、类名或者包名. 这种人工选择的方式乏味, 且配置的复杂

度与应用的复杂度密切相关, 因此极易引入错误. 进行细粒度追踪伴随着更高的性能开销, 甚至是冗余的追踪信息.

更好的方式是, 用户无需声明植入探针的方法名称, 而是定义植入探针的服务入口, 这实际上是分布式应用进行事务追踪在组件纬度的下钻. 对于根方法所代表的任务, 开发和运维人员可能需要将其分解成多个子任务才能对问题进行定位, 甚至子任务又可以继续进行分解. 这正是调用关系图插桩所解决的问题, 并且根方法执行过程中不涉及的方法不会进行探针的注入, 这在一定程度上避免了性能开销.

然而对于 Java 语言, 任意方法的调用关系图上的方法进行识别和插桩并不是简单. 一种通用方式是对程序进行静态分析, 扫描所有的字节码. 如果程序中使用了反射, 静态分析无法得知运行时绑定的方法. 除此之外, 如果程序中出现了动态绑定的方法调用, 必须进行类层次分析(CHA)或者对整个程序进行分析. 文献[7]总结了这些技术, 并指出这些技术都会对很多程序并不会调用的方法进行插桩. 简单的 CHA 方法在数量上可能会造成 10-20 倍方法的插桩.

如果我们只能对方法进行一次插桩, 也就是类文件被加载到 JVM 过程之前, 但此时利用服务入口的调用关系图进行插桩是十分困难的. 但是 JVM 提供的 Java Instrumentation Interface 允许我们在 JVM 在加载类文件的时候对其二进制内容进行修改, 并且 hotswapping 机制在不添加删除字段和不添加删除方法的前提下, 完成类的重新定义. 在这基础上, 形成了 EasyTrace 的方法.

3.2 自动化追踪算法

当一个方法满足如下条件时, 才会对其进行插桩:

(1) m()所在的类 C 已被 JVM 加载.

(2) CHA 分析认为 m()是可达的.

(3) 方法 m1()会直接调用 m(), 而且 m1()已插桩.

(4) 方法 m1()即将被第一次执行.

相比静态的类层次分析, 条件 1 对目标进行了限制, 只对已被 JVM 加载的 class 进行插桩.

条件(2)是层级分析的体现, 用来解决动态绑定的问题.

条件(3)、(4)是考虑到并不是所有被 JVM 加载的 class 都会出现方法调用, 使用惰性的方式来减少潜在的字节码分析和迭代过程.

利用 Java Instrumentation Interface 来实现动态插桩的过程如图 2 所示.

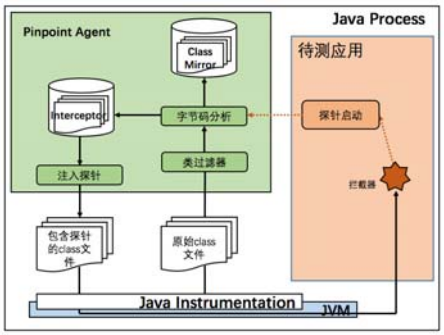

图 2 动态插桩过程

对于每一个正在被加载的类文件, 将类文件输入算法 1, 可以得到注入探针的类文件, 再由类加载器加载.

算法 1 对类文件进行插桩

输入: 类 c 原始文件的字节数组 bytes
输出: 添加探针的字节数组 i_bytes
create ClassMirror for c
i_bytes ← bytes
if bytes has service_entry in E **then**
  create probe for service_entry in i_bytes
  for each method invoke instruction ins in service_entry
    if ins is in SI **then**
      create probe for ins' method
    if ins is in DI **then**
      create probe for ins' method in subclasses and concreateclasses of ins' class
if superclass or interface have probe **then**
  create probe for that method in i_bytes

probe 是一个对象, 当 probe 的 before()方法被第一次调用时, 会分析探针所在方法的字节码, 执行与算法描述中一样的过程. ClassMirror 用来维护类之间的继承关系和方法是否已经添加探针等信息,

对 $e \in E$, 执行下述算法过程.

1) 当所在类 C 被 JVM 加载时, 为这个类创建 ClassMirror(一个简单的数据结构), 由类元数据仓库进行管理, 主要包括当前类定义的字节数组, 方法插桩状态.

2) 对入口方法进行插桩, 并在 C 的 ClassMirror 中标记.

3) 发生在探针的 before()中, 检查 m()是否是第一次执行, 如果是的话, 对 m()执行步骤 4.

4) 分析方法 m()的字节码, 分析所有的方法调用

指令:

a) 对 si ∈ SI ,在所在类的 ClassMirror 中将该方法标记为“可达未插桩”.

b) 对 di ∈ DI , 在所在类的 ClassMirror 中将该方法标记为可达未插桩, 并对当前所有 JVM 加载的 DI 的子类或实现类的 ClassMirror 中将该方法标记位为“可达未插桩”.

当步骤 4 执行完时, 我们对 m()直接调用的所有方法都进行了标记, 然后利用 hotswapping 机制对这些方法进行插桩, 然后标记为“已插桩”.

5) 当一个类 C 被加载时. 如果类元数据仓库没有 C 的 ClassMirror, 则创建一个. 然后检查 C 的所有父类和接口类的 ClassMirror 中是否有“可达未插桩”或“已插桩”的方法, 如果有, 检查 C 是否重写了这些方法, 标记它们为“可达未插桩”.

最后, 对 C 中所有标记为“可达未插桩”的方法进行插桩.

4 系统实现

Pinpoint[8]是 2015 年开源的分布式追踪系统, 其插桩的思路是面向组件的, 以插件的形式提供了对于 Tomcat、Jetty、Memcached 等组件的支持[9]. 本文的方法在其 1.5.0 版本做了实现.

Pinpoint 使用 Javassist[10]实现底层的字节码修改, Pinpoint 上层使用了 ClassPool、InstrumentClass、InstrumentMethod、InstrumentContext、Interceptor 等抽象描述插桩过程, 它们作为 API 以实现 Pinpoint 插件的开发. 扩展这些 API 即可在插桩过程提供字节码分析的功能, 再对插件源码进行简单的修改即可实现 EasyTrace.

ClassPool: 与 ClassRepository 概念类似, 扩展了此接口, 存储当前加载类的字节码.

InstrumentClass: 重写了 toByteCode(), 在生成字节码的时候写入 ClassPool, 追踪 EasyTrace 运行状态.

InstrumentMethod: 添加了一个接口方法 codeAnalyse(), 用于分析方法的字节码指令, 进行自动追踪, 可用于插件的开发. 并利用此 API 修改了 Pinpoint 官方的 Spring 插件.

InstrumentContext: 扩展了此接口以利用 hotswapping 机制来实现动态插桩.

EasyTraceInterceptor: 探针类. 与普通探针不同的是, before()方法被调用时可能产生新的探针, 同时会根据运行时的统计信息主动关闭, 避免产生过多监测数据.

5 实验验证

为了验证文本的方法, 使用开源的电子商务框架——Broadleaf 搭建了一个多层架构的购物网站 Heat Clinic[11]. 主要是为了验证 1)EasyTrace 能对异常进行更准确的定位; 2)同时量化相对于 Pinpoint 对系统造成的扰动.

5.1 实验环境

如图所示, 实验环境使用 MySQL 数据库作为数据层, 业务逻辑层部署在 Tomcat 应用服务器上, Pinpoint Agent 和故障注入器作为 Java Agent 注入监控探针和会产生异常的代码, 使用 JMeter 作为负载生成器.

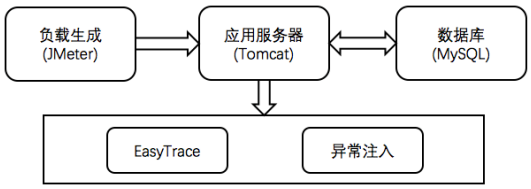

图 3　实验架构图

5.2 异常定位的准确性

故障注入不是本文讨论的范围, 我们使用了几种典型的异常[12]并且使用典型的注入方法[13]. 实验中使用运行时和耗时操作这两种故障, 即.

运行时异常: 运行时异常可能会造成服务异常终止, 所以也是监测数据的一部分, 在注入点抛出 RuntimeException.

耗时操作: 因为监测数据是服务执行时间, 所以使用线程睡眠来模拟耗时操作.

表 1　运行时异常例子

```
public class SolrSearchServiceImpl implements SearchService,
InitializingBean, DisposableBean {
  ...
  public SearchResult findSearchResultsByQuery(String query,
SearchCriteria searchCriteria) throws ServiceException {
    //运行时异常
    throw new RuntimeException();
    List<SearchFacetDTO> facets = getSearchFacets();
```

```
        query = "(" + sanitizeQuery(query) + ")";
        return findSearchResults(query, facets, searchCriteria, null);
    }
    ...
}
```

我们主要对 Heat Clinic 的几个方法进行了故障注入，因篇幅所限，就运行时异常和耗时操作分别进行一个用例研究.

① 搜索业务

搜索业务的服务入口是 Servlet 所在 Controller 层 SearchController 的 search 方法，实验时选择 CCT 上较高的位置来注入性能异常，即向 SolrSearchServiceImpl 的 findSearchResultsByQuery 方法添加运行时异常. 使用 Pinpoint 获取的监测如图 4.

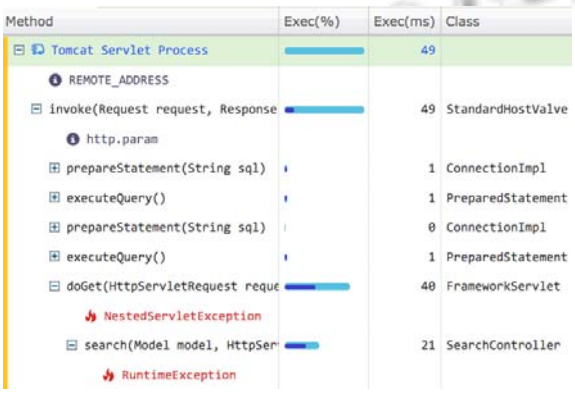

图 4 Pinpoint 监测下的 Search 业务

Pinpoint 的拦截器对于异常的处理方式是捕获并记录，再重新抛出，所以在监测路径的上层可以看到下层抛出的异常，但是在缺少配置的情况下，Pinpoint 不能准确定位异常位置. 开发人员可以从日志信息获取 StackTrace，从而定位问题，所以这种故障的调试相对容易. 使用 EasyTrace 获取的监测情况如图 5.

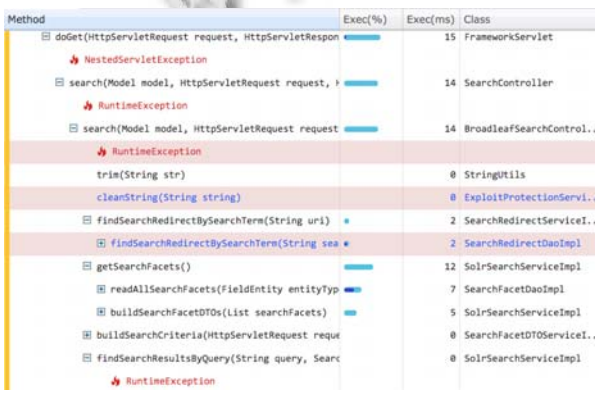

图 5 EasyTrace 监测下的 Search 业务

② 购物车业务

与搜索业务类似，购物车业务的服务入口为 CartController 的 addJson 方法，与搜索业务不同的是实验时在 OrderServiceImpl 的 addItem 方法添加耗时操作. 使用 Pinpoint 获取的监测如图 6.

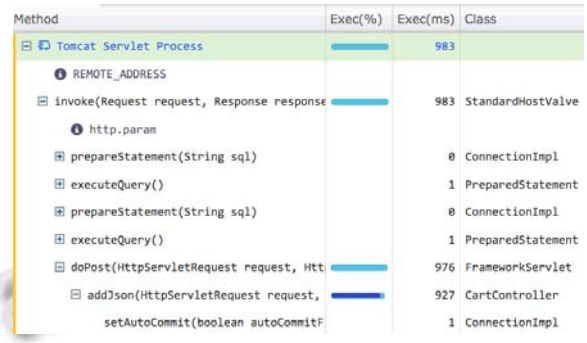

图 6 Pinpoint 监测下的 Cart 业务

与搜索业务类似，没有进行配置的 Pinpoint 不能发现耗时操作的准确位置，在缺乏信息的情况下开发人员对此进行调试十分困难. EasyTrace 同样准确定位到问题所在，如图 7.

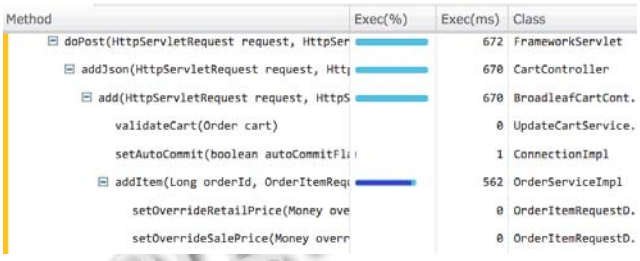

图 7 EasyTrace 监测下的 Cart 业务

5.3 性能开销

本节主要对比 EasyTrace 相对 Pinpoint 对系统造成的扰动，通过 JMeter 脚本形成访问首页、搜索、添加购物车等业务的混合负载，并进行不同压力规模的测试. 线程的思考时间设定为 100 毫秒，设计多组对比实验.

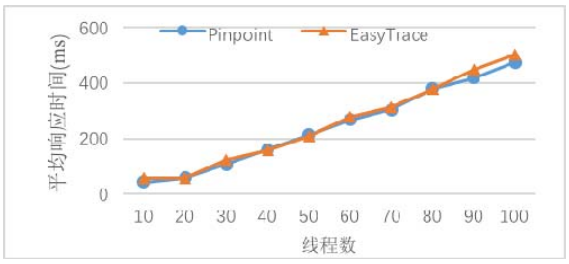

图 8 不同负载下的应用响应时间

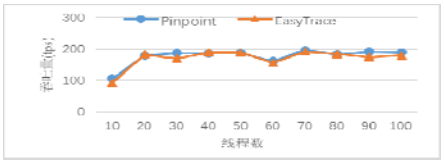

图 9 不同负载下的服务吞吐量

在向 Heat Clinic 进行实验设定的压力测试之前, 使用 JMeter 发起 1 个虚拟用户的负载进行预热以让 EasyTrace 的动态插桩趋于稳定. 实验数据表明使用 EasyTrace 相对于 Pinpoint 的性能开销很低, 不会影响到系统的正常服务能力.

6 相关工作

性能诊断一直是高性能计算领域和大型分布式系统研究的重点. 主要的研究方法有 2 类, 基于黑盒模式的统计方法和基于插桩的请求追踪. 本文属于后者.

在选择性插桩方法的研究工作中, 文献[14]利用了 JVM 热交换特性, 提出了一种细粒度插桩的方法, 用来对程序根方法的调用图进行侧写.

在利用选择性插桩工具建立应用的资源使用和性能模型的问题上, 文献[15]提出了一种自适应的插桩和监测方法. 但是其依赖于已有工作的一种插桩抽象描述, 并且不适用于分布式系统的线上长期监测.

性能数据的产生和对系统性能的扰动依赖于插桩代码的植入粒度. 文献[16]总结了控制插桩代码粒度的机制, 并开发了 TAU 工具用于高性能计算的并行程序的性能监测.

7 结语

对大型分布式系统进行性能诊断是当前研究是热点之一. 系统出现性能降级时, 系统开发与维护人员需要借助工具来对系统行为进行理解, 并对问题进行定位和修复. 针对当前工具监测粒度依赖开发人员的问题, 本文提出了一个插桩配置自动化的方法——EasyTrace. EasyTrace 对已有服务入口方法字节码指令进行分析, 利用 JVM 热交换机制完成运行时插桩. 实验证明 EasyTrace 相比已有工具, 能够对问题进行更准确的定位, 同时自动化程度也更高.

需要说明的是 EasyTrace 对于程序启动的影响会随着程序的复杂度而提升, 虽然这种影响不会在程序运行时得到体现, 但是会影响服务程序在集群环境下的部署. 并且动态插桩过程要求应用预热.

本文下一阶段主要的研究方向包括: 1)对插桩过程并行化的研究; 2)对服务入口的自动识别; 3)完善插桩规则的定义.

参考文献

1 The Cost of Poor Web Performance. http://blog.smartbear.com/web-performance/the-cost-of-poor-web-performance-infographic/.
2 How Search Works. http://www.google.com/insidesearch/howsearchworks/thestory/.
3 Sigelman BH, Barroso LA, Burrows M, et al. Dapper, a large-scale distributed systems tracing infrastructure. Google Research, 2010.
4 Zipkin. http://twitter.github.io/zipkin/.
5 Java API for RESTful Web Services. https://zh.wikipedia.org/zh/JAX-RS.
6 Apache Thrift. http://thrift.apache.org/.
7 Rountev A, Milanova A, Ryder BG. Fragment class analysis for testing of polymorphism in Java software. IEEE Trans. on Software Engineering, 2004, 30(6): 372–387.
8 Pinpoint . https://github.com/naver/pinpoint.
9 Pinpoint supported modules. https://github.com/naver/pinpoint-supported-modules.
10 Chiba S. Javassist—a reflection-based programming wizard for Java. Proc. of OOPSLA'98 Workshop on Reflective Programming in C++ and Java, F. 1998.
11 broadleaf online demo. http://demo.broadleafcommerce.org/.
12 Pertet S, Narasimhan P. Causes of failure in web applications (cmu-pdl-05-109). Parallel Data Laboratory, 2005, 48.
13 Jiang G, Chen H, Yoshihira K, et al. Ranking the importance of alerts for problem determination in large computer systems. Cluster Computing, 2011, 14(3): 213–227.
14 Idmitriev M. Profiling Java applications using code hotswapping and dynamic call graph revelation. ACM SIGSOFT Software Engineering Notes, 2004, 29(1): 139–50.
15 Wert A, Schulz H, Heger C. AIM: Adaptable instrumentation and monitoring for automated software performance analysis. International Workshop on Automation of Software Test. IEEE. 2015. 38–42.
16 Shende S, Malony AD, Morris A. Optimization of instrumentation in parallel performance evaluation tools. Applied Parallel Computing State of the Art in Scientific Computing, Springer, 2006: 440–449.