# 基于模型的云应用动态配置框架

梁超超 陈 伟 魏 峻 许舒人
(中国科学院软件研究所 北京 100190)

**摘 要** 云应用是云计算技术在应用层的一种重要体现形式,通常由分布式异构组件构成,且组件相互依赖,配置参数众多。组件依赖导致配置参数间存在关联,使应用运行时弹性扩展难以确定组件实例的配置顺序并保证关联参数的一致性,导致应用扩展后的系统故障和服务不可用。针对这一问题,提出了一种基于模型的云应用动态自配置方法,实现运行时组件实例配置顺序的自动协调,保障配置参数的一致性,提高应用运行时弹性扩展的可靠性。首先提出一种部署配置模型STM(Service-based Topology Model),该模型采用声明式的方法刻画云应用的部署拓扑结构,并基于服务的方式描述组件信息,实现组件间关联关系的分离,涵盖运维部署、扩展、运行时服务状态的监测。然后基于该模型,提出了一个云应用动态自配置协议,其基于服务注册发现机制实现组件间强依赖关系的解耦,保证动态调整应用实例时组件配置变化的一致性,实现组件部署配置的并行化。基于上述方法实现了一个原型系统,通过对分布式应用BookStore-TPCW的部署配置和运行时弹性扩展来验证方法的有效性。

**关键词** 云应用,自配置,模型,运行时扩展
**中图法分类号** TP319 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.04.011

## Model-based Runtime Configuration Framework for Cloud-based Applications

LIANG Chao-chao CHEN Wei WEI Jun XU Shu-ren
(Insititute of Software, Chinese Academy of Sciences, Beijing 100190, China)

**Abstract** Cloud-based application is one of the most important paradigms of cloud computing at application layer. These applications are usually constituted of a set of distributed components. Due to the dependencies between the components, there are correlations between the configuration parameters, which makes it difficult to configure the components correctly and to ensure the parameter consistencies at runtime, making system failed or service unavailable. To address this issue, we proposed a model-based self-configuration method for cloud-based applications, which realizes the automatic choreography for component configuration, ensures configuration consistency and improves the reliability of runtime scale. Firstly, this method proposes a model to specify application topology in a declarative way. This model describes application components in form of services, and makes the separation of components and their relations. Then this method designes a self-configuration protocol to support component configurations at runtime. Based on service registration and service discovery, this protocol decouples the component dependencies and ensures the consistencies of changing the configuration values. We also implemented a prototype based on this method and evaluated the effectiveness of this method with a typical application BookStore-TPCW.

**Keywords** Cloud-based application, Self-configuration, Model, Runtime scale

## 1 引言

云计算以虚拟化技术为基础、以网络为载体提供基础设施、平台、软件等服务。作为云计算在应用层的重要体现形式,云应用越来越多地被应用服务提供商(Application Service Provider, ASP)所采用。根据国际数据咨询公司(International Data Corporation, IDC)的预测[1],公有IT云服务投入将在2018年增长到1270亿美元。在2014年的公有IT云服务投入中,SaaS服务投入占比70%。面对不断增长的用户规模、业务规模和系统规模,云应用的部署维护工作呈爆炸式增长,管理压力急剧增加,如何有效地部署与管理云应用成为云计算领域面临的挑战之一。

到稿日期:2015-11-30 返修日期:2016-02-28 本文受自然科学基金(61402453),863项目(2013AA041301),国家科技支撑(2015BAH18F02)资助。

梁超超(1991—),男,硕士,主要研究方向为网络分布式计算与软件工程,E-mail:liangchaochao13@otcaix.iscas.ac.cn;陈 伟(1980—),男,副研究员,主要研究方向为网络分布式计算与软件工程,E-mail:wchen@otcaix.iscas.ac.cn;魏 峻(1970—),男,研究员,主要研究方向为分布式计算与软件工程,E-mail:weijun@iscas.ac.cn;许舒人(1961—),男,高级工程师,主要研究方向为网络分布计算和软件工程,E-mail:shuren@iscas.ac.cn。

目前存在的主要问题有如下几点：1)云应用的复杂度和规模不断扩大，系统参数众多；2)云应用通常由异构组件组成，系统参数间存在相关性，导致组件参数在配置顺序上存在约束，各个组件初始部署的安装启动均需参照一定的顺序；3)云应用通常采用多租户模式，系统的负载频繁变化，应用系统需要根据负载的变化进行组件的弹性伸缩。由于组件参数依赖的原因，增减组件实例的同时需更改其他已有组件实例的配置参数取值。为应对上述问题，已有相关工作[2-5]提出了一套动态配置协议，用于协调组件间相互依赖的参数设置，完成应用的初始部署。但是这些工作无法解决运行时应用组件动态增减、服务失效引起的诸多问题，包括：1)组件实例配置参数取值的运行时动态调整；2)组件实例故障导致的应用系统部分参数设置失效。另一方面，已有工作在运行时组件服务可用性状态检测机制方面存在不足，难以正确判断服务状态并做出相应的正确操作。如 MySQL 启动后需加载应用特定的数据库初始化脚本，在此过程中虽然 MySQL 服务可访问，但因应用的数据库表并未完全建立，此时 Web 应用访问数据库建立连接仍会失败。

针对上述问题，本文提出了一种基于模型的云应用动态自配置方法。该方法不仅支持初次部署时组件间依赖参数的动态设置，还支持运行时弹性扩展动态配置，自动协调组件实例配置顺序，保障配置参数的一致性，提高云应用运行时弹性扩展的可靠性。同时该方法加入了组件运行时的状态检测，以监测已部署组件的服务是否可用，当组件服务不可用时，自动更改应用已部署组件的配置，提高云应用运行时的可用性，避免弹性扩展时失效配置参数的影响。本文的主要贡献如下：

1)提出部署配置模型 STM。该模型以声明式的方法刻画云应用的部署拓扑结构，采用基于服务的方式描述组件信息与组件状态监测信息，实现组件与组件间关联关系的分离，涵盖应用运维的部署、扩展、运行时组件服务状态的监测。

2)基于部署配置模型 STM 提出一个云应用动态自配置协议，基于服务注册/服务发现机制支持应用组件的并行部署和自动配置，自动协调组件实例的配置参数设置和实例启动顺序，提高部署效率和配置的正确性。该协议基于组件模型定义的服务检测命令自动检测应用组件的服务状态，避免因为组件启动命令的执行与组件服务可用间存在的时间差问题导致的应用部署配置失败。基于服务状态信息，该动态自配置协议能够解决组件运行时部分组件故障导致的配置参数失效的问题，缓解单一组件故障引起的整个应用服务不可用的问题。

本文第 2 节介绍描述云应用部署拓扑结构的声明式模型。基于该模型，第 3 节介绍云应用动态配置协议，阐述该协议实现应用组件实例动态扩展的工作机制以及协议中各个角色的交互流程。基于上述模型和协议，第 4 节介绍系统的原型实现，描述了原型的整体架构，并在第 5 节的实验中对本文方法的有效性进行实验评估。第 6 节介绍本文的相关工作。最后总结本文的工作并介绍未来的工作方向。

## 2 应用拓扑描述模型

混合云模式越来越被企业采纳，各大云提供商提供的部署方式多样，如何将分布式的应用部署在多个云提供商的平台上以及如何在多种部署方式间进行切换已经成为企业迫切关心的问题。应用部署的复杂度与应用的结构紧密相关，组件采用管理工具、环境的异构性更增大了这一难题的复杂度，比如虚机的创建需调用 Amazon 的 API，而虚机上的软件则依赖于配置管理系统如 Puppet 等。为增强应用的跨 Iaas 移植及应用的可扩展性，需要一种统一的模型来描述应用。

### 2.1 TOSCA

OASIS 提出了云应用程序的拓扑结构和编排规范 TOSCA(The Topology and Orchestration Specification for Cloud Applications)[6]。TOSCA 采用 YAML 或者 XML 描述云应用组件属性、组件关系以及组件的行为操作(包括安装、卸载等)，创建应用部署拓扑和应用部署任务流程，简化云应用的生命周期管理。

如图 1 所示，一个服务模板(Service Template)对应一个完整的应用。服务模板包含两部分：描述应用组件拓扑结构关系的拓扑模板(Topology Template)，以及采用流程模型如 BPEL 和 BPMN 等描述的云应用生命周期中包括部署、配置、卸载等在内的管理任务流程的执行计划(Plan)。

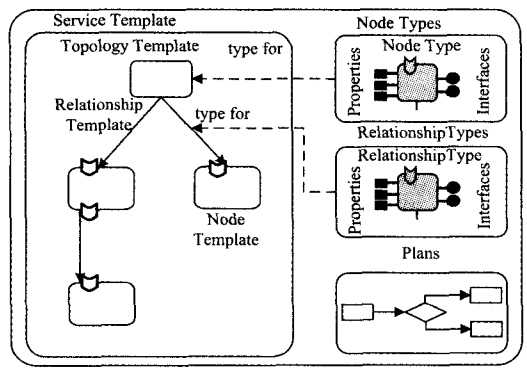


图 1 TOSCA 模型

拓扑模板是一个定义应用结构的有向图，其中节点为 Node Template，边为 Relationship Template。Node Template 描述应用组件或依赖的软件系统如操作系统、MySQL 数据库等。Relationship Template 定义 Node Template 节点之间的关系，如 MySQL 必须安装在操作系统之上。Node Template 与 Relationship Template 分别为 Node Type 与 Relationship Type 的实例，类似于面向对象语言中的类与对象的概念。Node Type 是可重用的软件组件的抽象，定义组件的属性(Properties)、组件的行为(Interfaces)以及组件的可输出属性(Attributes)。Properties 描述节点的属性如 Tomcat 安装路径，Interfaces 指定对该节点可执行的操作，如安装、卸载等，Attributes 描述 Properties 中可用于其他组件类型的 Interfaces 脚本的输入参数。

Plans 是基于拓扑模板中定义的节点的组件行为(Interfaces)，采用流程模型如 BPEL 和 BPMN 等描述的云应用生命周期的管理任务流程。因不同组件的管理操作间存在先后依赖关系，需要一个统一的调度器来控制应用组件的部署执行顺序，流程控制复杂，实现困难，易形成单点故障。

然而 TOSCA 模型无法定义组件间参数的依赖关系以及

组件的服务状态、服务状态检测命令，这也是 STM 模型与 TOSCA 最大的不同之处。STM 模型基于配置模板描述组件间的关系、配置参数间的依赖，同时 STM 模型不需要描述应用的 Plans，可基于 STM 模型的描述自动化地构建管理任务。

### 2.2 部署配置模型 STM

为协调应用组件间相互依赖的参数的配置，本文将组件的参数中被其他组件依赖的部分抽离，定义为服务信息，其他组件基于该服务信息与其自身的配置文件模板自动化更新自己的配置文件。组件的服务信息有两种状态：可用和非可用。状态的检测基于用户自定义的命令，从而可分离应用的部署启动命令的成功执行与组件服务可用间的逻辑关联，启动命令的成功执行并不代表组件服务信息，组件提供的服务是可用的，解决了因二者之间存在的时间差导致的应用部署失败的问题。并且当组件发生故障时，可将服务的状态标记为不可用，从而解决配置参数失效问题，实现应用组件配置参数的自我修复。

为实现运行时组件实例配置顺序的自动协调，保障配置参数的一致性，提高云应用运行时弹性扩展的可靠性，本文提出一个满足运行时再配置需求的应用拓扑描述模型，即 STM (Service-based Topology Model)。模型抽象语法结构如图 2 所示。

```
ServiceTemlate::=name=>string
    description=>string
    nodeTypes=>NodeType+
    relationshipTypes=>RelationshipType+
    topologyTemplate=>TopologyTemplate

NodeType::=name=>string
    properties=>Property+
    capabilities=>Capability+
    requirements=>Requirement+
    services=>Service+
    attributes=>Attribute+
    interfaces=>Interface+

Property::=name=>string
    defaultValue=>string
    required=>boolean

Capability::=name=>string
    source=>string

Requirement::=name=>string
    target=>string

Service::=ip=>string
    port=>string
    check=>string
    name=>string

Interface::=name=>string
    params=>InterfaceParam+

InterfaceParam::=name=>string
    value=>string

Attribute::=key=>string
    value=>string

RelationshipType::=name=>string
    target=>string
    source=>string
    templates=>Template*

Template::=template=>string
    configurationfile=>string
    command=>string

TopologyTemplate::=
    nodetemplates=>
        NodeTemplate+
    relationshiptemplates=>
        RelationshipTemplate*

NodeTemplate::=name=>string
    type=>string
    min=>integer
    max=>integer
    [properties=>Property+]
    [services=>Service+]
    [attributes=>Attribute+]
    [interfaces=>Interface+]

RelationshipTemplate::=name=>string
    type=>string
    [target=>string]
    [source=>string]
    [templates=>Template*]
```

图 2　STM 文法定义

该模型的主要信息如下：

1）描述组件的拓扑结构。隔离应用的高层抽象与具体平台实现细节，实现与配置管理工具（Configuration Management Tool，CMT）的集成，支持声明式模型和系统初始部署与维护管理协作脚本间的自动转换，解耦组件间部署行为的依赖。

2）描述组件的服务信息以及服务状态的检测命令。用于验证部署的正确性，分离应用部署的启动命令的成功执行与组件服务可用间的逻辑关联，启动命令执行成功并不代表该服务是可用的，解决了因二者之间存在的时间差导致的应用部署失败的问题；同时可用于实时检测应用层面的服务失效问题。

3）描述组件间的参数依赖关系。通过配置模板中的配置参数与组件的服务信息构建整个应用的关系依赖图。通过组件服务的注册与发现检测应用组件的增减，组件的服务状态检测组件的服务失效，配置参数的注册与发现检测应用组件配置参数的更改。

本模型定义以 ServiceTemplate 表示云应用的拓扑模型，以 TopologyTemplate 表示应用的拓扑结构。该模型主要包含如下几个部分。

1）组件类型（NodeType）

组件类型是对应用可重用的组件的抽象，描述组件的属性参数、关联参数以及服务信息等。图 3 给出 Tomcat 组件的 NodeType 定义。

```
-name:nodetype, tomcat
 description:tomcat type
 properties:
 -name:catalina_base
  defaultValue:
/usr/local/tomcat
  required:false
 attributes:
 -name:port
  defaultValue:8080
  required:true
 capabilities:
  -name:host
   source:container
 requirements:
  -name:host
  target:container
 services:
  -name:tomcat_service
 serviceId:
 ip:
 port:8080
 check:
  interfaces:
  -name:install
 params:
  -name:catalina_base
   value:/usr/local/tomcat
   -name:start
   -name:stop
   -name:delete
   attributes:
    -name:port
    value:8080
```

图 3　Tomcat 组件类型的定义

其中主要包含如下几个部分。

①属性参数（Properties）：组件自身的属性参数，只与组件自身相关，属性参数既不会因其他组件参数的变化而变化，也不会影响其他组件的参数值。如图 3 所示的 catalina_base 指定 Tomcat 组件的安装路径，其中的 required 表明实例化该 NodeType 时用户必须指定其值。

②关联参数（Attributes）：被其他组件依赖的参数，组件 Attribute 的变化会引起其他组件的配置文件发生更改。如图 3 所示的 attributes 下指定的 port，当 Tomcat 的访问端口更改后，会导致 Nginx 的 upstream 配置文件更改 members 的定义。

③服务信息(Services):描述该组件提供的服务的名字、IP、访问端口、服务检测(check)信息。check 一般定义为一条命令,格式可扩展为 URL 地址,用于检测该服务是否可用。check 是可选的,默认为检测服务所在虚机是否宕机。

④能力与需求描述(requirements,capabilities):用于验证关系的正确性,如 Tomcat 具备应用访问能力和数据库访问的需求。Nginx 具有应用访问需求。当关系以 Tomcat 为起点连接至 Nginx 时,则关系建立失败,因为 Nginx 并不具备 DBMS 能力,不满足 Tomcat 的数据库访问需求;反之,以 Nginx 为关系起点则成立。

2)关系类型(RelationshipType)

关系类型是对云应用组件关系的抽象,将组件之间的依赖关系从组件类型定义中分离。本模型将组件间的依赖关系抽象为一组配置文件模板对其他组件的关联参数或服务的依赖,在配置模板中进行关联参数依赖的声明。图 4 为一个基于 Consul Template 描述 Web 应用的配置模板,其中 username(第 8 行)和 password(第 9 行)依赖于 MySQL 组件的关联参数(mysql_user,mysql_passwd)。关系类型包含该关系类型的源组件(source)与目的组件(target),以及两组件间配置依赖的定义。配置依赖的定义包括如下几大部分。

①配置模板定义(template):描述配置模板文件的路径;

②配置文件路径(configurationfile):描述基于配置模板文件生成的新配置文件存放的路径;

③配置载入命令(command):描述新配置文件生成后,应用组件载入新配置的命令。

```
1. 〈Resource
2. name="jdbc/bench4q"
3. auth="Container"
4. type="javax. sql. DataSource"
5. driverClassName="com. mysql. jdbc. Driver"
6. maxIdle="30"
7. maxWait="5000"
8. username="{{key_or_default"db/mysql_user" "root"}}"
9. password="{{key_or_default"db/mysql_passwd" "root"}}"
10. {{range service "TPCWMySQL"}}
11. url = "jdbc: mysql://{{. Address}}: {{. Port}}/bench4q"
{{end}}
12. maxActive="1024"/〉
```

图 4 配置模板示例

3)应用组件拓扑结构图(TopologyTemplate)

TopologyTemplate 是由组件类型实例(NodeTemplate)和依赖关系实例(RelationshipTemplate)定义的应用拓扑结构图,描述整个应用的部署信息。

①组件实例(NodeTemplate):表示组件类型(NodeType)的具体实例,设置组件类型中 Properties 和 Attributes 的值以及云应用初始部署时的最小实例数(min)、最大可扩展实例数(max)。

②关系实例(RelationshipTemplate):表示关系类型(RelationshipType)的具体实例,用于描述图中两个 NodeTemplate 之间的关系,包括用户指定关系的模板文件以及配置文件的相对路径等。

图 5 所示即为一个 Web 应用 BookStore-TPCW 的 TopologyTemplate 定义,其中 BookStore-TPCW 是一个基于Java实现的满足 TPC-W 基准的购物电商网站,包含 3 个 NodeTemplate:bookstore_mysql,bookstore_tomcat1,bookstore_nginx,其中 bookstore_tomcat1 为图 3 定义的 Tomcat NodeType 的实例;2 个 RelationshipTemplate:tomcat_connectTo_mysql,nginx_connectTo_tomcat。

```
topologytemplate:
  nodetemplates:
  -name:bookstore_mysql
   type:cloudeploy. nodetype. mysql
   min:1
   max:1
  -name:bookstore_tomcat1
   type:cloudeploy. nodetype. tomcat
   min:1
   max:10
  -name:bookstore_nginx
   type:cloudeploy. nodetype. nginx
   min:1
   max:1
  relationshiptemplates:
  -name:tomcat_connectTo_mysql
   type:template_relationship
   source:bookstore_tomcat1
   target:bookstore_mysql
   templates:
       -name:context_config_template
        type:tomcat_db_relation
        template:tpcw_db. ctml
        configurationfile:/usr/local/tomcat/webapps/bookstore-
                          tpcw/META-INF/context. xml
        command:
       -name:nginx_connectTo_tomcat
        type:nginx_lb_relation
```

图 5 TopologyTemplate 示例

## 3 配置协议

基于提出的应用拓扑模型,本节提出了支持云应用初始部署和运行时弹性扩展的自配置协议。该协议基于服务注册与服务发现机制,以用户建模时的组件关联和配置参数信息为依据,通过发布/订阅机制实现应用组件间配置参数取值的自动调整,并保证其最终一致性,实现部署配置流程的解耦;基于自治和协调方式确保应用组件间参数配置的正确性,实现应用的部署和运行时的弹性伸缩。

该协议中包含 Manager,ServiceCluster 和 Agent 3 类角色。本节主要介绍和讨论协议中各个角色的职责以及在初始化部署、弹性伸缩或组件服务不可用几种场景下的交互流程。

### 3.1 角色定义

角色划分如图 6 所示。

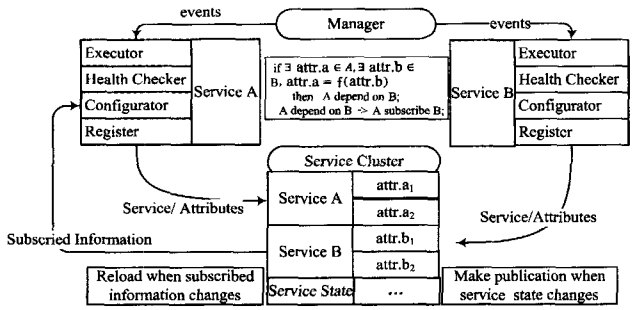


图 6　角色划分

根据图 6，该协议中的主要角色分为以下 3 种。

1）Manager，负责解析应用拓扑模型，自动生成应用各组件的初始化脚本，还负责部署脚本的分发和管理脚本的执行。对于给定目标应用的拓扑模型，Manager 遍历拓扑模型中的组件类型与组件实例，从组件脚本库中获取对应组件类型的安装、卸载、服务启动等脚本，将拓扑模型图转化成一组独立的任务脚本。

2）Service Cluster，其功能分为 3 部分：①接收 Agent 的服务注册，存储 Agent 上组件的 Attributes 和 Services 信息；②服务发现以及服务的订阅推送，类似一个发布订阅的中间件。假设组件 B 依赖于组件 A，组件 B 会向 Service Cluster 订阅组件 A 对应的服务信息，当组件 A 的服务状态发生更改后，Service Cluster 负责通知组件 B；③服务状态监测，定时检测服务状态，收集已注册服务的状态信息。

3）Agent，主要分为 4 部分：部署执行器、动态配置器、服务注册器、健康诊断器。每个虚机或者容器有且仅有一个 Agent。

①部署执行器（Executor）：负责脚本的解释执行，将组件拓扑模型描述中相应的信息发送给其他组件，将组件服务的 check 信息提交给健康诊断器，组件的 RelationshipTemplate 信息提交给动态配置器，组件的 Service 信息提交给服务注册器。

②服务注册器（Register）：负责应用组件的 Service 和 Attributes 信息注册。

③健康诊断器（Health Checker）：负责检测本机已部署的应用组件服务是否可用。健康诊断器主要管理拓扑模型中定义的 Service 的 check 信息。健康诊断器定时运行 check 命令，将 check 的结果发送至 Service Cluster。当 check 失败后，Service Cluster 将对应的应用组件注册的服务更改为不可用，并通知依赖该服务的其他组件的动态配置器。

④动态配置器（Configurator）：负责动态更改配置。动态配置器管理应用拓扑模型中的 relationtemplates 定义的配置模板以及配置文件目标地址信息。当接收到新的配置参数后，动态配置器自动参照配置模板生成新的配置文件，替换目标文件，重新启动应用组件。动态配置器将组件的安装与配置之间的强依赖和强顺序转换为松耦合的、顺序无关的动态自配置，不同组件的安装顺序是并行的，组件的安装顺序是不确定的，通过动态配置器和 Service Cluster 间的交互流程能保证最后组件的状态。假设组件 B 依赖于组件 A，若组件 B 先于组件 A 启动动态配置器，动态配置器向 Service Cluster 订阅组件 A 的服务信息以及属性信息，此时 Service Cluster 中并没有组件 A 的服务信息，组件 B 的动态配置器会处于等待状态，只有当组件 A 启动后，通过服务注册器将自身的服务信息发布到 Service Cluster 中，组件 B 的动态配置器才会生成配置文件，启动组件 B。动态配置器分离脚本中配置部署流程的控制，简化脚本的自动化生成和系统的流程控制逻辑。

### 3.2　角色交互

下面分别介绍在初始部署、发生故障、增删节点 3 个场景下各个角色间的交互过程。

#### 3.2.1　初始部署

图 7 描述了某一应用组件初始部署时各个角色间的交互流程。首先 Manager 解析拓扑模型，创建已预安装 Agent 的虚机，自动生成部署脚本，将脚本发送至 Agent 的部署执行器；部署执行器接收到 Manager 的执行请求后，执行脚本，安装组件，检测该组件是否依赖于其他组件，如果是则将该组件动态更改的配置文件路径以及对应的配置模板信息提交给动态配置器；动态配置器从 Service Cluster 中获取组件依赖的其他组件的 Service 和 Attributes 信息，实时更新组件的配置文件的所有依赖参数获取完毕后启动组件，部署执行器将组件的 Service 和 Attributes 信息提交给服务注册器，由服务注册器发送至 Service Cluster 供其他组件查询，同时将状态检测命令提交给健康诊断器；健康诊断器开始检测组件的服务状态。

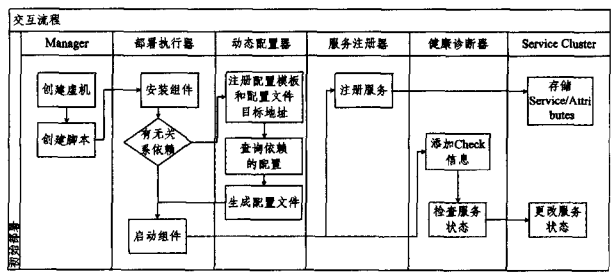


图 7　组件初始部署流程图

图 8 描述了 BookStore-TPCW 应用初始部署时可能出现的一个时序图。应用不同组件的部署是并行的，组件的安装顺序是不确定的，但最后各组件的状态是确定的，即整个 Web 应用能正确访问。首先 Manager 以 Nginx→MySQL→Tomcat 的顺序初始化安装该 Web 应用的 3 个组件，Nginx 安装完成之后，无法从 Service Cluster 中查询到 Tomcat 组件的 Service 和 Attributes 信息，处于等待状态。MySQL 安装完毕后，检测无配置模板设置，启动 MySQL 应用，将自己的 Service和 Attributes 信息发送至 Service Cluster 中。Tomcat 安装完毕后，检测 db. property 配置文件指定了配置模板，读取配置模板，从 Service Cluster 中查询到 MySQL 的配置信息，通过配置模板在指定路径下生成配置文件，启动 Tomcat 的同时将 Service 和 Attributes 信息发送至 Service Cluster 中；Service Cluster 将信息推送至 Nginx 的动态配置器，载入模板，生成对应的配置文件启动 Nginx，启动后将 Nginx 的信息发送至 Service Cluster 中。

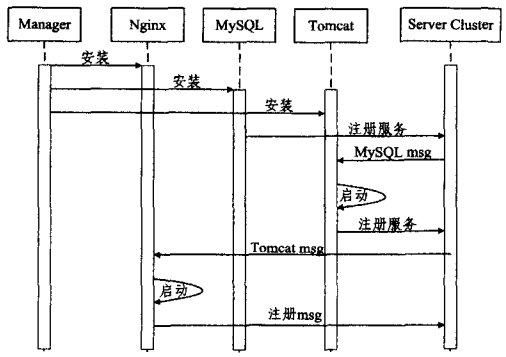


图 8 BOOKSTORE-TPCW 部署时序图

### 3.2.2 动态配置

动态配置指在应用运行过程中对应用的组件进行再配置的操作。本文重点阐述应用组件的增减和应用组件的服务不可用引起的应用组件的重配置。而针对应用某一组件的配置的更改类似于应用组件的增减，本文不再赘述。

1)应用组件故障

与文献[4]不同，本文将安装组件的虚拟机宕机、动态配置器或者整个 Agent 的故障、网络故障以及应用组件自身的故障等统一归结为组件服务不可用故障。应用组件服务状态的检测依赖应用拓扑模型中对应 NodeType 或者 NodeTemplate 指定的 check 属性，check 属性指定检测应用组件健康状态的命令，如 Tomcat 组件可定义 check 命令为 netstat-ntlp|grep 8080。默认故障 check 属性值为提到的检测虚机、物理机故障以及网络故障，通过心跳机制与 Service Cluster 交互。本文后续的故障均代指服务不可用故障。

①故障检测

如图 9 所示，应用组件初始化时，部署执行器会将拓扑模型中定义的组件 check 信息提交给健康诊断器。健康诊断器以一定的频率将组件的状态信息发送至 Service Cluster，当时间延迟超过一定阈值或者健康诊断器诊断出应用组件服务不可用时，Service Cluster 将该组件的 Service 状态更改为不可用，其他依赖该组件的组件就会感知到该信息，从而启动动态配置器，重新更改配置参数。

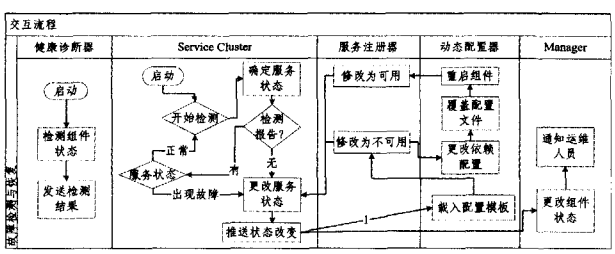


图 9 故障检测与故障恢复流程图

②故障恢复

当组件产生故障并被检测到时，Service Cluster 将该组件的 Service 与 Attributes 值标记为不可用，同时通知监测该组件信息的动态配置器和 Manager，重新载入模板生成新的配置文件，重新启动组件。组件间参照依赖关系描述，自动协调配置顺序，某一组件收到更新配置通知后，会将自身的状态先更改为不可用，从而使依赖该组件的其他组件的动态配置器一直处于等待状态，直到完成重新配置。Manager 则将信息反馈给运维管理人员，由运维管理人员进一步恢复。

2)应用组件弹性扩展

①增加组件

当应用组件增加节点时，与初始部署场景下最后被部署的组件类似，各个角色的交互流程如下：应用组件先从 Service Cluster中获取配置模板中依赖的参数，生成配置文件，启动组件，之后将 Service 与 Attributes 信息存入 Service Cluster 中，由 Service Cluster 通知监听该组件信息的其他组件的动态配置器重新生成新配置文件，重新启动组件。与故障恢复类似，此时其他组件配置的先后顺序由各个组件的动态配置器自动协调。

②删除节点

删除节点与故障恢复类似，在删除应用组件对应的某一节点前，由消息交互组件通知 Service Cluster 删除与该节点相关的 Service 等信息，同时通知依赖该组件的其他组件，启动重新配置过程。

## 4 系统实现

基于前述的应用拓扑模型与动态配置协议，本文基于Java 实现了面向云应用的部署配置和管理服务系统 Cloudeploy，其能够通过 WebUI 实现对应用系统的部署配置建模，并集成 CMT(Configuration Management Tool)工具 Puppet[7] 以及服务发现工具 Consul[8] 以实现云平台应用的自动部署、运行时的动态配置、弹性扩展。该系统的原型系统架构如图 10 所示，主要包括 4 个部分：WebUI、Manager 集群、Service 集群、Agent 代理。

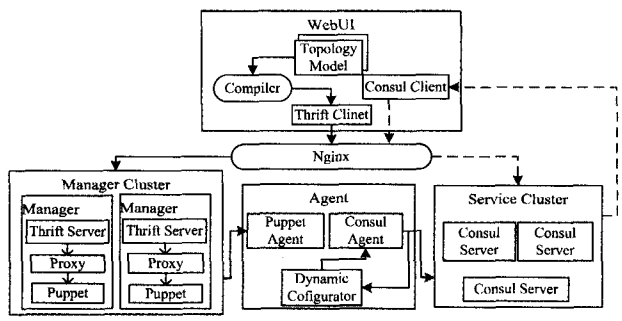


图 10 系统原型系统架构

1)WebUI。其负责模型部署建模与应用状态管理。基于 JsPlumb 提供图像化的建模工具，系统默认提供了大量常用应用组件的 NodeType 实现，如 Tomcat，MySQL，Nginx 等。用户只需拖拽图像化的组件、修改参数以实例化组件类型，即可建立应用的拓扑模型。

2)Manager 集群。Manager 主要包含 3 个组件：①Thrift Server，用于接收 WebUI 端的部署操作请求；②Proxy，用于处理部署请求，验证部署请求中执行脚本的正确性，将脚本传递给 Puppet Master，触发 Agent 端 Puppet Agent 与 Manager 的 Puppet 组件交互；③Puppet，用于编译部署脚本，将部署脚本发送至 Agent 代理中的 Puppet Agent 和接收脚本执行报告。WebUI 的部署请求之间相互独立，请求会随机转发至 Manager集群中的任意一个 Manager 节点。Manager 节点是无状态的，不保存任何用户主机信息，便于扩展。

3)Service 集群。Service 集群由一组 Consul Server 组

成，负责存储 Agent 上已安装组件的 Service 以及 Attributes 参数，监测 Agent 端组件的状态，实时更新组件服务状态，具体流程参照配置协议。

4）Agent 代理。Agent 代理主要包含 3 部分：①Puppet Agent，负责部署脚本的执行；②Consul Agent，负责注册服务，报告服务状态；③Dynamic Configurator，负责组件的动态配置，管理组件模板，通过 Consul Agent 查询服务信息，动态地更新配置。

如图 11 所示，用户通过 WebUI 界面创建应用的拓扑模型。WebUI 层编译器基于该模型确定目标系统中各个组件的初始实例数，在系统默认的组件仓库中检测与拓扑模型中 NodeTemplate 对应的 NodeType，最后将拓扑模型自动转化为 Puppet 脚本，自动安装应用的各个组件并启动 Agent 端的动态配置器完成应用的动态配置。该系统的动态配置依赖于服务发现工具 Consul 搭建服务发现和 keyValue 存储集群，每个组件的 Service 以及 Attributes 信息将通过 Agent 端的 ConsulAgent 发送至 Consul 的 Server 集群。

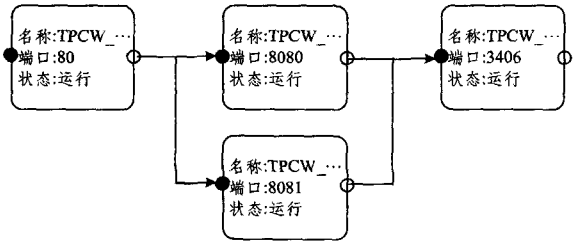


图 11　BookStore-TPCW 拓扑建模

## 5　实验

本实验主要为验证本文解决的 3 个问题而设计，步骤如下：1）定义 BookStore-TPCW 的 STM 模型如图 11 所示，单个 Tomcat 节点，单个 MySQL 节点，包含 50 万条图书数据，单个 Nginx 节点。在 Nginx 的 upstream 配置模板中声明对 BookStore-TPCW 服务信息的依赖：{{range service "bookstore-tpcw"}} server {{. Address}}:{{. Port}} max_fails=0 fail_timeout=10s; {{end}}，其中{{}}定义特定的操作，range service 代表遍历所有可用 BookStore-TPCW 服务信息，每条服务信息生成一条 server 配置信息。在 BookStore-TPCW 的 context 配置模板中声明对 MySQL 服务信息的依赖，部署应用并观察压力测试平均响应时间，以检测应用是否部署成功。2）运行时为应用增加 Tomcat 实例，观察平均响应时间的变化。3）使用配置模板中的 max_fails 与 fail_timeout 参数关闭 Nginx 默认健康检测，减少 Tomcat 节点，对比采用动态配置协议前后错误事务的比率与时长，检测组件宕机的自我配置修复。

本实验的目的是验证动态配置框架的有效性。假设应用的性能会随节点的增减而变化，因吞吐率以及平均响应时间与底层硬件资源和应用的配置紧密相关，故本实验只关注应用吞吐率在增减节点后的变化。

### 5.1　实验配置

实验待测系统采用 BookStore-TPCW。BookStore-TPCW 应用各个组件的版本如下：Tomcat 版本为 8.0.9，Nginx 版本为 1.8.0，MySQL 版本为 5.6.25。实验环境采用软件所软件工程中心研发的云平台 OnceCloud，测试虚机的配置为 1 核 CPU，1GB 内存，压力测试工具采用 JMeter2.13。

### 5.2　实验结果与分析

1）系统将模型转化为脚本并分发至 3 个虚机节点，分别对应应用的 3 个组件，启动动态配置协议。MySQL 节点等待 50 万条图书数据初始完毕，将注册 MySQL 服务信息，触发等待中的 Tomcat 节点配置 context 文件并启动，此时应用已正确部署。如图 12 所示，前 1 分 57 秒在 500 并发下系统能正确响应，此时 Nginx 的配置如图 13 所示。2）如图 12 所示，在未中断服务的情况下，在 1 分 57 秒时系统自动构建扩展脚本扩展 Tomcat 节点，自动修改 Nginx 的 upstream 配置文件，触发 Nginx 重新载入配置文件，此时平均响应时间在同样负载下较单节点更为稳定，此时 Nginx 的配置如图 14 所示。3）在 Nginx 关闭默认健康检测功能运行场景下，启动压力测试 30 秒后，关闭其中一个 Tomcat 节点模拟组件宕机故障，此时 Nginx 平均失败事务数占比 37.89%，如图 15 所示，节点宕机后，失败事务数一直高于 0。

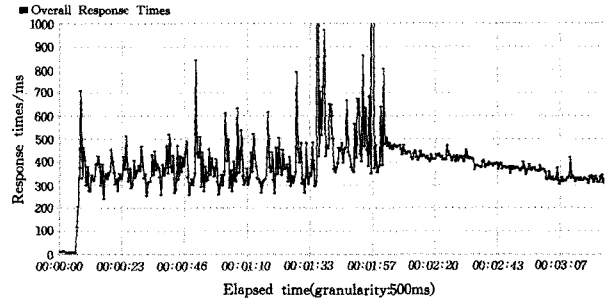


图 12　增加 Tomcat 节点前后的平均响应时间

```
upstream bookstore-tpcw{
# ip_hash;
  keepalive 20;
  server 133.133.134.78:8081 fail_timeout=10s;
}
```

图 13　单节点后 Nginx 配置

```
upstream bookstore-tpcw{
# ip_hash;
  keepalive 20;
  server 133.133.134.77:8080 fail_timeout=10s;
  server 133.133.134.77:8081 fail_timeout=10s;
}
```

图 14　多节点 Tomcat，Nginx 配置

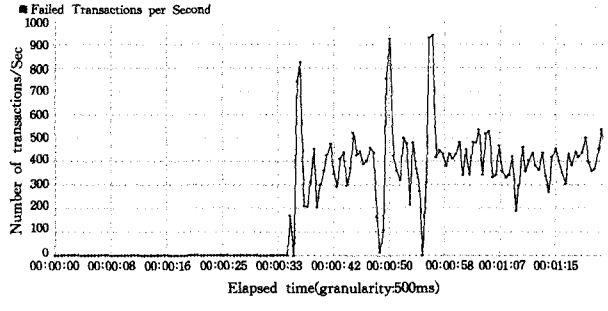


图 15　平均失败事务

启动自动配置机制，重新模拟宕机，此时 Service Cluster 中对应的 BookStore-TPCW 服务状态变为不可用，Nginx 节点的动态配置器接收到事件通知，upstream 模板中的{{range bookstore-tpcw}}命令只能获取到一条可用的 BookStore-TPCW 的服务信息，重新生成配置文件并加载。平均失败事务数降至 6.99%，如图 16 所示，系统将 Nginx 配置自我修复至图 14 所示状态，失败事务重新降为 0。

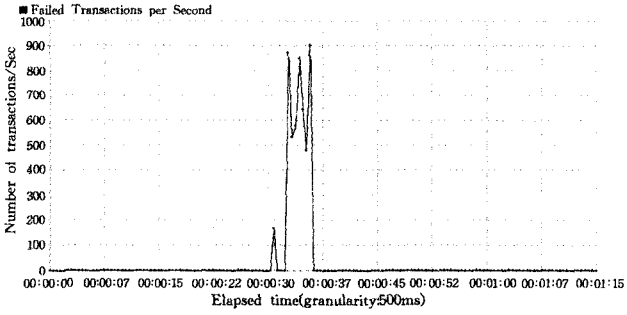


图 16 平均失败事务(采用动态配置协议)

以上实验表明，本系统能支持应用系统的初始化部署、弹性伸缩以及修复组件宕机产生的失效配置参数。

## 6 相关工作

为提高运维工作效率，降低应用管理的成本，目前工业界已经提出了很多商业的或者开源的配置管理工具 CMT(Configuration Management Tool)，如 Puppet，Chef，Ansible，Salt Stack。它们能够解决大规模集群配置管理问题，但是局限于企业或者机构内部使用，同时需要用户了解底层运维细节、软件部署的顺序、软件参数之间的依赖关系，无法分离应用部署模型与具体实现，且软件的升级迭代会导致软件部署顺序以及参数依赖关系处于不可知状态，使得 CMT 对软件运行时再配置的支持并不完善。

当前已有的 PaaS 平台在描述云应用高层拓扑结构及参数依赖关系方面存在不足，没有将用户需求与系统实现细节完全分离，难以降低应用部署配置和运维管理的复杂度，使用场景和适用范围受到限制。例如，Google App Engine 提供基于 YAML 描述用户上传的应用的 App Config Tool，目前支持 Java，Python，GO 等编程语言实现的多层 Web 应用，该描述模型将实现与需求描述紧密结合，用户无法跨平台迁移。Microsoft Azure 支持的应用的类型在实现技术上存在局限性，主要支持采用 Microsoft 技术实现的云应用系统。

在模型方面，如 AWS CloudFormation，OpenStack Heat，Terraform 等主要针对 IaaS 层资源的创建与管理，基于配置文件创建 IaaS 层资源，依旧无法分离用户的部署需求和部署的实现细节。TOSCA 能方便地描述复杂应用的拓扑结构以及配置描述，但 TOSCA 过于复杂，对用户有一定学习成本，并且与已有配置管理工具结合时，同样需要用户了解应用各个组件的运维细节，需自定义每个组件的安装、卸载等一系列操作。基于 TOSCA 模型的 Cloudify 虽然已经能部分图形化，可是所有配置参数均由中心的 Manager 在解析 TOSCA 模型时计算出，一旦某一组件出现故障，所有的参数需重新计算，容易形成单点故障。文献[9]将应用组件的具体部署实现与应用的需求描述分离，提出应用描述模型并基于 Puppet 实现模型到部署实现的转换，但是其与 Cloudify 一样，所有的配置参数在编译阶段计算出，无法动态地再配置，可扩展性不强，无法应对运行时组件服务不可用引起的组件参数失效问题。

文献[3,5]将参数的计算从编译阶段抽离，提出了自配置的动态协议，在部署过程中动态地设置应用组件间的依赖参数，可是文献[3,5]的自配置动态协议局限于初始配置，假定应用在初始部署完成后，配置参数不再更改。文献[4]在文献[3,5]的基础上加入错误检测，提高了协议的可靠性，可是文献[4]中假定的虚机层面的故障与应用自身无关，本文在此基础上添加应用自身服务不可用故障的监测。

云应用后期维护管理任务如均衡负载增删节点、节点故障移除恢复、升级某一组件是很常见的。文献[2]提出了一个在运行时虚拟机的增加与删除的管理协议，该协议的运行依赖于一个支持消息发布订阅的信息系统；同时该协议局限于虚机层面的增加与删除，而非应用组件层面的增加与删除，即使是虚机层面的故障也没有可靠的机制保证系统的正确性。文献[4]提出的协议与文献[2]类似，增加了应用层面组件的启动与停止时应用的动态配置，以及增加了虚机或网络故障发生后应用动态配置时一致性的考虑(如已启动的组件不应该依赖于一个已停止的组件)。但是文献[4]的协议忽略了应用组件层面的故障。本文与文献[4]的不同之处在于，配置信息存储于分布式的服务发现集群中，考虑了运行时组件服务状态监测，能应对部署成功后部分组件服务不可用的问题。文献[5]中组件间的配置依赖参数依赖于各个虚机上的 Agent 相互交互信息，各个虚机上的 Agent 之间的初始关系的建立以及虚机的故障检测完全依赖于中心的管理器处理，容易形成单点故障。

**结束语** 本文首先提出一种部署配置模型 STM，该模型采用声明式的方法刻画云应用的部署拓扑结构，并基于服务的方式描述组件信息，实现组件间关联关系的分离。然后，本文基于该模型提出了一个云应用动态自配置的协议，基于服务注册发现机制实现组件间强依赖关系的解耦，保证动态调整应用实例时组件配置变化的一致性，实现组件部署配置的并行化，分别从初始部署、故障检测与恢复等几个场景出发描述了协议中各个角色的交互流程。本文基于上述方法和已有的配置管理工具 Puppet 与服务发现工具 Consul 实现了一个系统原型，描述了系统原型的整体架构以及架构中各个组件的功能，并通过对典型分布式应用 BookStore-TPCW 的部署配置、故障模拟和运行时弹性扩展来验证方法的有效性。未来将通过更多应用实例来进一步验证该协议的可靠性，同时考虑加入基于规则的弹性扩展以及 Docker 容器的支持，基于虚机或者容器 CPU 等资源的利用率，自动化地扩展应用。

## 参 考 文 献

[1] IDC. Worldwide and Regional Public Cloud IT Services 2014-2018 Forecast[OL]. http://www.idc.com/getdoc.jsp?containerId=prUS25219014.

[2] ABID R, SALA N G, BONGIOVANNI F, et al. Verification of a Dynamic Management Protocol for Cloud Applications [M]//

Automated Technology for Verification and Analysis. Springer, 2013:178-92.

[3] SALA N G, ETCHEVERS X, DE PALMA N, et al. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud [M]// Assurances for Self-Adaptive Systems. Springer, 2013:60-79.

[4] ETCHEVERS X, SALA N G, BOYER F, et al. Reliable self-deployment of cloud applications[C]// Proceedings of the 29th Annual ACM Symposium on Applied Computing. ACM, 2014.

[5] ETCHEVERS X, COUPAYE T, BOYER F, et al. Self-Configuration of Distributed Applications in the Cloud[C]// 2011 IEEE International Conference on Proceedings of the Cloud Computing (CLOUD). 2011:4-9.

[6] OASIS. Topology and Orchestration Specification for Cloud Applications Simple Profile in YAML Version 1.0[S]. 2015.

[7] Puppet[OL]. https://puppetlabs.com.

[8] Consul[OL]. https://www.consul.io.

[9] BREITENB CHER U, BINZ T, K PES K, et al. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA [C]// IC2E IEEE. 2014.

(上接第 23 页)

比较热门的研究方向。这方面比较有代表性的研究是 Bachmann 等人提出的正则表达式提取法[8]。该方法使用正则表达式提取 commit 记录中的缺陷标识来建立追踪关系，进而通过专家分析等方式验证追踪关系的正确性。实验证明，这种追踪关系建立方法可以提高追踪关系的准确率，但不适用于那些 commit 记录中缺省了所修复缺陷信息的情况。基于正则表达式构建追踪关系的研究还有很多[9-11]。

Ashish Sureka 等人[5]首次将 Fellegi-Sunter 模型应用到开源领域追踪关系的建立中。该研究证明 Fellegi-Sunter 模型可以有效建立开源软件缺陷修复追踪关系。本文在此基础上做了一些改进，首先引入正则表达式来提取关键数据，初步建立追踪关系，再利用 Fellegi-Sunter 模型对初步建立的追踪关系进行筛选，最后采用该模型全面挖掘可能的缺陷修复关联并建立追踪关系。相比 Ashish Sureka 等人的研究，本文引进正则表达式提取关键信息，并利用 FS 模型对初步建立的追踪关系进行验证筛选是本文的创新之处，且文中所提方法提高了缺陷修复追踪关系的质量。这种缺陷修复追踪关系构建方法为研究开源项目制品间的追踪关系提供了经验参考[12]。

**结束语**　在开源软件领域，软件制品可追踪性研究还处于起步阶段。本文在了解了开源社区软件开发过程、缺陷追踪系统以及版本控制系统等相关知识之后，对 Mozilla 项目 Bugzilla 缺陷追踪系统和 Mercurial 版本控制系统中的相关数据进行了深入的分析研究，并选取 Mozilla 项目中最大的软件产品 Firefox 浏览器的缺陷数据和代码提交记录作为基础研究数据集进行缺陷修复追踪关系的挖掘和建立。

本文采用 Fellegi-Sunter 模型对在 Firefox 浏览器版本 4 到版本 5 演化期间的缺陷修复关联关系进行挖掘，找到约 55.61%的缺陷修复追踪关系。建立开源项目缺陷和版本制品之间的缺陷修复追踪关系对于评估项目发展进度、发现制约项目发展的瓶颈以及开发中的异常现象、指导软件测试工作、提高缺陷发现率、改善软件质量管理版本更新以及预测 Bug 数和分派资源等具有重要的意义。本文对于 Firefox 浏览器缺陷修复追踪关系的建立方法可以为开源项目其他追踪关系的建立提供一定的经验参考。此外，在 Firefox 浏览器版本 4 到版本 5 演化期间，剩余 44.39%的缺陷修复关联没有找到，还需要更加深入的研究。

**致　谢**　本文工作是在北京大学软件研究所教授金芝老师和周明辉老师的指导下完成的，在此致以诚挚的感谢！

## 参 考 文 献

[1] CoEST: Center of excellence for software traceability[OL]. http://www.CoEST.org.

[2] BISSYANDE T F, THUNG F, WANG S, et al. Empirical Evaluation of Bug Linking[C]// European Conference on Software Maintenance & Reengineering. 2013:89-98.

[3] D'AMBROS M, LANZA M, ROBBES R. Evaluating defect prediction approaches: a benchmark and an extensive comparison [J]. Empirical Software Engineering, 2012, 17(4/5):531-577.

[4] FELLEGI I P, SUNTER A B. A Theory for Record Linkage [J]. Journal of the American Statistical Association, 1969, 64 (328):1183-1210.

[5] SUREKA A, LAL S, AGARWAL L. Applying Fellegi-Sunter (FS) Model for Traceability Link Recovery between Bug Databases and Version Archives[C]// 2011 18th Asia Pacific Software Engineering Conference (APSEC). IEEE, 2011:146-153.

[6] BETTENBURG N, WEISS C, JUST S, et al. What Makes a Good Bug Report? Revision 1.1[J]. Fse, 2008, 36(5):618-643.

[7] Bugzilla official website[OL]. http://www.bugzilla.mozilla.org.

[8] BACHMANN A, BERNSTEIN A. Data retrieval, processing and linking for software process data analysis: Technical Report IFI-2009.0003[R]. Department of Informatics, University of Zurich, May 2009.

[9] SCHRÖTER A, ZIMMERMANN T, PREMRAJ R, et al. If your bug database could talk[J]. Proceedings of International Symposium on Empirical Software Engineering, 2006, 7(5):18-20.

[10] SLIWERSKI J, ZIMMERMANN T, ZELLER A. When do changes induce fixes?[J]. ACM Sigsoft Software Engineering Notes, 2005, 30(1):1-5.

[11] ZIMMERMANN T, PREMRAJ R, ZELLER A. Predicting Defects for Eclipse[C]// Proc International Workshop on Predictor Models in Software Engineering. 2007:9.

[12] SHIHAB E, IHARA A, KAMEI Y, et al. Studying re-opened bugs in open source software[J]. Empirical Software Engineering, 2013, 18(5):1005-1042.