

基于 OpenStack 的大规模云负载测试平台研究

晋文明¹, 李昌建¹, 钱 巨^{1,2}

(1. 南京航空航天大学 计算机科学与技术学院, 江苏 南京 210016;

2. 江苏省软件新技术与产业协同创新中心, 江苏 南京 210023)

摘 要:为保障大型线上软件系统服务质量的可靠性,有必要对其有效地进行大规模负载测试。然而,现有测试工具存在支持的负载生成机制不够丰富、测试资源分配不够经济优化等问题,导致大规模负载测试不易开展。鉴于此,研究了多类型的负载生成、智能化测试资源分配和分布式负载同步控制技术,实现了一款基于 OpenStack 的大规模云负载测试平台。平台支持协程等负载并发机制,结合多种类型的测试脚本以生成大规模负载;基于负载测试的资源智能预测和多目标优化分配方法,实现面向云负载测试的资源优化分配;使用同步控制算法来保证不同测试主机上网络活动的并行性。该测试平台为测试人员实施大规模负载测试提供了一个功能丰富、经济易用的平台,可有效降低大规模负载测试的实施难度。

关键词:负载测试;测试脚本;协程;测试资源分配;负载同步控制

中图分类号:TP311

文献标识码:A

文章编号:1673-629X(2020)10-0047-06

doi:10.3969/j.issn.1673-629X.2020.10.009

Study on Large-scale Cloud Load Testing Platform Based on OpenStack

JIN Wen-ming¹, LI Chang-jian¹, QIAN Ju^{1,2}

(1. School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics,

Nanjing 210016, China;

2. Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210023, China)

Abstract: In order to guarantee the reliability of service quality of large-scale online software system, it is necessary to carry out a large scale load test effectively. However, some limitations in the existing test tools, such as inflexible load generation mechanisms and uneconomic and non-optimized test resource allocation, make it difficult to conduct large-scale load testing easily. To address these limitations, we study the technology of multi-type load generation, intelligent test resource allocation and distributed load synchronous control, and develop a large-scale cloud-based load testing platform on the ground of OpenStack. The platform supports coroutine-based concurrent load generation and multiple type test script languages. An optimized resource allocation for cloud load testing is implemented by incorporating intelligent resource demand prediction and multi-objective resource allocation optimization. The platform also adopts a synchronous control algorithm to enforce the parallel execution of loads on different test hosts. The whole test system provides a flexible and economical platform easy to use for large-scale load testing, which can effectively reduce the difficulty of conducting large-scale load testing.

Key words: load testing; test script; coroutine; test resource allocation; load synchronous control

0 引 言

随着互联网的普及,以电子商务网站(eBay、亚马逊、淘宝)和典型云服务(如Gmail、OneDrive)为代表的许多大型软件系统承受越来越多的访问流量^[1],这给软件系统的服务质量带来了一定的不确定性。负载测试作为软件测试的一种,是为了检测系统在负载方面的相关问题从而对系统进行评估的过程^[2]。因而,为了保障大型软件系统服务质量的可靠性,有必要对

其进行有效的大规模负载测试^[3]。人们研发了LoadRunner^[4]、JMeter^[5]、httpperf^[6]等一系列负载测试工具^[7],但这些测试工具一般存在两个问题。一是支持的脚本类型过于单一化,二是仅支持采用进程或线程并发方式发起负载,负载生成机制不够丰富,负载生成消耗不够优化,在资源受限的情况下难以生成较大规模的负载。同时,这些测试工具在分布式环境下的部署流程较为复杂,加大了测试的难度。云测试能够降

收稿日期:2019-12-19

修回日期:2020-04-20

基金项目:中国人民解放军总装备部装发部共性预研共用技术基金(170441402030)

作者简介:晋文明(1994-),男,硕士研究生,研究方向为软件分析与测试;钱 巨,硕士,副教授,博士,研究方向为软件分析与测试。

低负载测试的实施难度,在云负载测试服务方面,以阿里云 PTS^[8] 为代表的云测试系统同样存在负载生成不够优化的问题,执行大规模负载测试需要大量云资源作为支撑,增加了测试成本。在云测试工具的相关研究中,文献[9]实现了一个 IaaS 云平台测试系统,该测试系统的负载生成依赖于 Apache JMeter 来模拟并发负载,只支持以线程并发方式执行 JMeter 测试脚本,难以在有限资源下生成较大规模的负载。文献[10]实现了一种自动化云测试平台,基于测试工具 Selenium 执行测试脚本以实现自动化测试。但 Selenium 引擎执行开销非常大,导致能够发起的负载规模十分有限,并且该平台主要用于功能测试。文献[11]设计了一个基于云计算的软件测试系统框架,使用动态优先权调度算法实现测试任务的资源分配与执行。其资源分配没有考虑测试任务的资源特征,仅以测试执行时间为优化目标,可能导致测试资源分配不够优化。文献[12]也设计了一个云性能测试工具,支持分布式测试服务器间的同步控制功能,其工作流程为:由控制服务器向测试服务器发送测试执行命令,测试命令保证了所有接收到命令的测试服务器同步执行测试。但是该测试工具只能保证负载测试在初始状态的同步性,无法保证整个测试在不同阶段的同步性。因而,现有研究的测试工具往往存在负载生成机制不够丰富、测试资源分配不够优化等问题,导致大规模负载测试成本过高且不易实施。

为了能够高效地实施大规模负载测试,该文研究了多类型的负载生成、智能化测试资源分配和分布式负载同步控制技术,设计和实现了一种基于 OpenStack^[13] 的大规模云负载测试平台。该平台支持进程、线程和协程负载并发机制,结合多类型测试脚本以生成客户端负载;能够预测负载测试任务的资源需求,并为其确定云测试主机(OpenStack 中部署了测试执行引擎的虚拟节点)资源,实现智能化测试资源分配;采用同步控制算法保证不同测试主机间测试进度的同步性。该平台为大规模负载测试提供了一个功能丰富、经济易用的平台,辅助以智能化测试资源分配、分布式负载同步控制等功能保证了负载测试的执行效果,同时可帮助降低大规模负载测试的实施难度。

1 系统结构

基于 OpenStack 的大规模云负载测试平台的总体界面如图 1 所示。平台通过导入测试脚本,对待测 Web 应用进行负载测试,收集相关指标数据,并据此来分析承载 Web 应用的被测云服务设施的相关表现情况。该平台支持云负载测试执行、分布式负载同步控制、智能化测试资源分配等主要功能,此外,还支持

测试脚本管理、测试结果管理等辅助功能。



图 1 平台总体界面

为保证灵活性,云测试平台采用如图 2 所示的物理结构,整个系统由前端 Dashboard、测试控制中心 CloudTest、测试执行引擎 TestAgent、监控器 VMMonitor 四大基本模块构成。

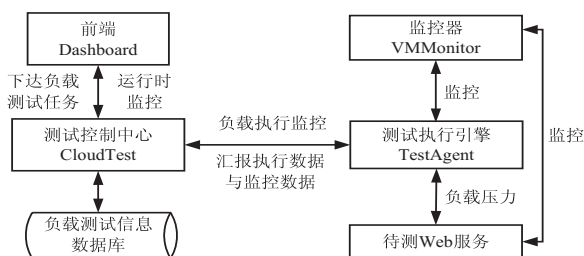


图 2 系统物理结构

前端 Dashboard 是一个 Web 前端页面,在该页面下,包含测试集群、云应用、云主机、测试脚本、云应用性能测试、测试结果等主页面。测试管理中心 CloudTest 一方面通过暴露的 RESTful 接口与 Dashboard 交互;另一方面与测试执行引擎交互,接收实时负载执行数据;与监控器交互,接收主机的实时监控数据。测试执行引擎是执行负载测试的主要模块,对用户配置的负载测试任务进行解析,按照预设的负载变化策略在指定的时间节点上生成相应规模的负载,发起对被测目标应用的网络调用,并统计每个负载的执行数据,汇报至测试控制中心。监控器周期性地采集宿主机的资源使用信息汇报至测试控制中心。其中,测试控制中心、测试执行引擎、监控器等模块均支持部署在私有云 OpenStack 上,亦可部署在公有云环境中。

2 支持协程的负载并发机制

由于进程或线程并发是多个子例程通过操作系统时间片轮转来实现,占据资源大、任务切换代价较高,导致进程或线程并发方式下单机的负载执行性能一般。协程是一种程序组件^[14],其高性能主要体现在如下几个方面:(1)协程间的切换由用户空间控制,无需操作系统参与,上下文切换的开销极小;(2)协程本身是轻量级的,资源占用小,基本不存在数量限制;(3)

协程具有非阻塞异步 I/O 模型的特性,在 IO 密集型程序中执行性能非常高。因而,协程可模拟大规模负载。

该云测试平台在支持进程、线程负载并发机制的基础上,引入了协程负载并发机制。Quasar 为 Java 提供了高性能的轻量级线程,使用 Quasar Fiber 可实现 Java 协程,平台利用 Quasar Fiber 和异步 Apache Http 组件实现并发客户端负载的生成,其主要流程如图 3 所示。

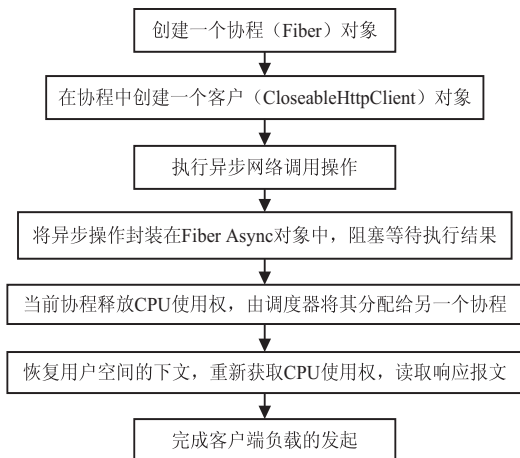


图 3 协程执行流程

如图 4 所示,平台利用云环境(如,OpenStack)中的测试主机来发起大规模负载测试,通过分布式并行调用产生对目标应用的并发客户端负载。测试主机上具体由访问被测试应用的任务进程、任务进程内的任务线程或任务协程来完成网络访问。不同任务进程、任务线程以及任务协程模拟了不同的虚拟客户端,总的任务进程、任务线程和任务协程的数量大致对应了所生成客户端负载的总体规模。

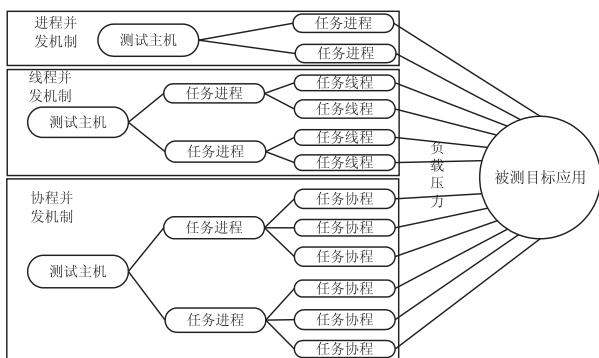


图 4 负载并发机制

3 基于多类型脚本的负载生成

传统负载测试工具支持的脚本过于单一化,导致无法很好地适用于不同的测试目标和目的。文中的云测试平台支持执行 Selenium、JMeter、Java 等类型的测试脚本,基于多类型测试脚本来生成客户端负载。脚本间比较如表 1 所示。

表 1 不同类型的测试脚本

脚本类型	支持并发机制	优点	缺点	负载生成性能
Selenium	进程并发 线程并发	测试的 Web 应用类型较广	执行开销非常大	较差
JMeter	进程并发 线程并发	适用性强、执行开销较小	灵活性较弱	尚可
Java	进程并发 线程并发 协程并发	执行开销极小、执行效率高	手工编写、制作代价高	较好

如 Selenium 脚本示例所示的脚本能够描述 Web 页面的打开、点击等行为,可表达复杂动态页面 Web 应用上的动作,灵活性最强,能测试的 Web 应用类型最广。但是脚本的执行开销非常大,并且依赖 Python 引擎支撑,而当前大部分 Python 引擎的并发执行性能都较弱。因而,在资源较为一般的测试主机上,很难发起较大规模的负载。

```

from selenium import webdriver
st = tester()
class Addbook(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://192.168.1.130/"
    def test_addbook(self):
        driver = self.driver
        driver.get(self.base_url + "/BookManager? method = list")
        driver.find_element_by_link_text(u"添加图书").click()
if __name__ == "__main__":
    unittest.main()
  
```

如 JMeter 脚本示例所示的脚本是一段 XML 配置,描述了一组针对 Web 应用的网络访问报文。利用 JMeter 脚本执行引擎可以提取 JMeter 脚本中所描述的网络报文,对其进行重放。JMeter 脚本执行引擎的一个优势是可以直接利用其他工具的脚本。相对 Selenium 脚本,JMeter 脚本的负载生成能力明显更强。然而,JMeter 脚本不能像 Python 脚本那样随意修改,轻松插入新的动作代码,因此,灵活性弱于 Selenium 脚本。

```

<? xml version="1.0" encoding="UTF-8"? >
<jmeterTestPlan version="1.2" properties="3.1" jmeter="3.3
r1808647">
  <hashTree>
    <hashTree>
  
```



```

<HTTPSampler Proxyguiclass=" HttpTestSampleGui" test
class=" HTTPSamplerProxy" testname=" Open blazemeter.
com" >
    <elementProp name=" HTTPSampler. Arguments" elem
entType=" Arguments" >
        <collectionProp name=" Arguments. arguments" />
    </elementProp>
    <stringProp name=" HTTPSampler. domain" >
weibo. com</stringProp>
    <intProp name=" HTTPSampler. port">80</intProp>
    <stringProp name=" HTTPSampler. path" >/<
</stringProp>
    <stringProp name=" HTTPSampler. method" >GET
</stringProp>
</HTTPSamplerProxy>
</hashTree>
</hashTree>
</jmeterTestPlan>

```

如 Java 脚本示例所示的脚本使用 Apache HTTP 库通信,对网络的访问较为直接,避免 JMeter 中的脚本解释开销。Java 脚本支持协程等高性能机制,依赖于协程并发方式执行脚本,相对于其他测试脚本,其资源开销最低,执行效率最高,在同样的 CPU 和内存配置下,脚本能够更大规模的并发。然而,此种脚本需要手工编写,且需要事先编译,制作代价较高。

```

import java. util. * ;
public class TestCase implements Runnable {
private static String URL = " http://www. baidu. com" ;
public void run() {
    ResultStatus status = ResultStatus. PASSED;
    try {
        Response response = Request. Get( URL). execute();
        StatusLine statusLine = response.
returnResponse(). getStatusLine();
        int statusCode = statusLine. getStatusCode();
        if( statusCode != 200) {
            status = ResultStatus. FAILED;
        }
    }
}
}

```

4 智能化测试资源分配

在实施大规模负载测试的过程中,由于单台机器的硬件资源能力较为有限,难以生成较大规模的负载。为了生成足够规模的负载,往往需要多台测试主机同时提供测试服务。分配的主机资源如果过少,将会导致无法发起相应规模的负载;分配的主机资源如果过多,将会造成测试资源的冗余。因而,如何为负载测试任务分配资源显得十分关键。

文中的云测试平台支持智能化测试资源分配功能。平台以每一处理周期内用户下达的负载测试任务的序列 $T = \langle t_1:T_1, t_2:T_2, \dots, t_n:T_n \rangle$ 为处理对象。首先使用机器学习方法预测出 T 中各测试任务 T_i 的资源需求 R_i 。在资源需求的基础上,基于遗传演化的多目标资源分配算法^[15],确定提供测试服务的云测试主机等所需资源的分配方案,依据分配方案,将各个负载测试任务分配到对应的 OpenStack 云测试主机上执行。

4.1 测试资源预测

对于目标负载测试任务序列 T 中的每个测试任务 T_i ,云测试平台按照如下流程预测得到其资源需求向量 $R = \langle R_1, R_2, \dots, R_n \rangle$ 。

首先,从平台的测试脚本池中选择目标负载测试任务(假设目标负载规模为 5 000)的测试脚本(如, edit_order. jar 脚本)作为训练的对象,平台支持为其自动配置一个较小负载规模(一般为 500)的测试活动,并在单台测试主机(一般配置为 2 CPU 核心和 4 GB 内存,其资源向量可用 CPU 核心数和内存大小的二维向量(2, 4)表示)上自动执行该测试活动,完成预热执行工作。

第二步,在预热执行过程中,部署在测试主机上的测试执行引擎收集实时负载执行数据 $L(t) = \{(t, \text{load})\}$,监控器实时收集测试主机的资源使用数据 $R(t) = \{(t, R)\}$,使用处理函数 $M(X, Y)$ 将负载执行数据和资源使用数据融合,使其在时间节点上对应,得到负载_资源数据 LR,可表示为:

$$LR = M(L(t), R(t)) = \{(\text{load}, R)\}$$

第三步,以负载资源数据 LR 作为训练模型的输入,通过回归学习方法,抽取 edit_order. jar 脚本的资源模型(可用映射 $f: \text{load} \rightarrow R$ 表示,其输入为目标负载规模 load,输出为负载测试任务所需资源向量 R)。如图 5 所示,其中 CPU 资源模型为 $f_{\text{cpu}}(x) = 0.284 \ln x + 0.07$,内存资源模型为 $f_{\text{ram}}(x) = 0.013x^{2/3} + 0.29$,从而预测负载测试任务的资源需求为(2.49, 4.09),则至少需要 3 台配置为 1 核心 CPU 和 2 GB 内存的云测试主机提供测试服务。重复上述步骤,预测出目标负载测试任务序列中每个测试任务的资源需求向量。

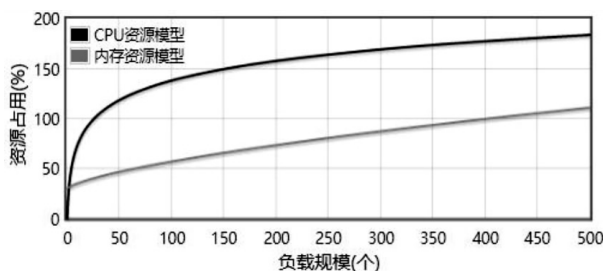


图5 资源模型

4.2 测试资源分配

对于待分配资源的负载测试任务序列 T , 基于 4.1 节所述的方法可预测其资源需求向量为 R 。云测试平台采用一种基于遗传演化的多目标资源分配算法为 T 分配云测试资源, 其总体执行流程如图 6 所示。在资源分配的过程中, 为了减少资源的浪费, 降低测试的成本开销, 并保证测试的执行效率, 算法以最小资源冗余、最低测试执行成本为一组优化目标, 关于目标函数的定义这里不再赘述。

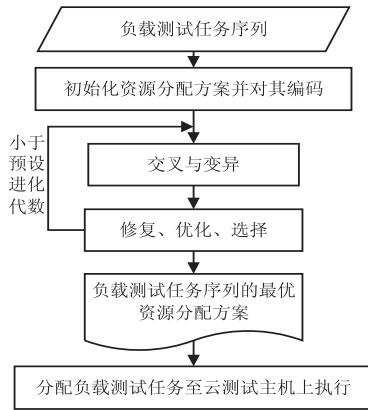


图 6 算法执行流程

首先, 平台支持获取当前云测试环境 OpenStack 的网络拓扑结构图 (物理节点、虚拟节点以及路由节点间的网络通信结构)。依据网络拓扑结构以及虚拟节点可用资源等信息生成负载测试任务序列 T 的一组初始化资源分配方案, 并按照向量 S 所示编码方式对资源分配方案进行编码。

$$S = \begin{bmatrix} \{vm_{p_{11}}, vm_{p_{12}}, \dots, vm_{p_{1c_1}}\} \\ \{vm_{p_{21}}, vm_{p_{22}}, \dots, vm_{p_{2c_2}}\} \\ \dots \\ \{vm_{p_{m1}}, vm_{p_{m2}}, \dots, vm_{p_{mc_m}}\} \end{bmatrix} = \begin{bmatrix} \{p_{11}, p_{12}, \dots, p_{1c_1}\} \\ \{p_{21}, p_{22}, \dots, p_{2c_2}\} \\ \dots \\ \{p_{m1}, p_{m2}, \dots, p_{mc_m}\} \end{bmatrix}$$

然后, 对编码后的资源分配方案进行交叉、变异操作, 形成初步子代种群。对初步子代种群中的资源分配方案进行修复并优化, 确保分配方案的可行性与优越性; 父子种群合并, 将合并后的种群中的优良分配方案选择进入新的种群, 使得分配方案的最小资源冗余、最低测试执行成本等目标朝着更优的方向进化, 从而完成一次进化。重复上述过程直至预设进化代数, 最终得到目标负载测试任务序列的最优资源分配方案。

最后, 平台依据最优资源分配方案, 将负载测试任

务序列中的各测试任务分配到对应的 OpenStack 云测试主机上, 并由平台对云测试主机进行统一化管理。分配完成后, 即可开始目标负载测试任务的执行, 平台收集实时负载执行数据以对测试进行运行时监控, 从而形成对各负载测试执行的性能评估。

给定如图 7 所示的云测试环境 G 和一组资源需求已知的负载测试任务 $T = \langle A, B, C \rangle$, 在文中的云测试平台上分别采用多目标资源分配和随机资源分配方法为其分配测试资源。实验结果如图 7 所示, 在相同规模的云测试环境下, 对于相同的负载测试任务序列, 多目标资源分配方法相较于传统的随机分配方法, 能够减少云测试主机的占用数, 降低测试执行成本。

实验环境	云测试环境 G	虚拟节点数	物理节点数	路由节点数
	资源需求向量	80	10	4
实验结果	测试任务 A	测试任务 B	测试任务 B	测试任务 B
	(3.62, 6.87)	(4.52, 10.23)	(2.17, 4.13)	
	资源分配方式	占用云测试主机数	测试执行成本	
	多目标资源分配	11	35	
	随机资源分配	13	48	

图 7 实验环境与实验结果

5 分布式负载同步控制

随着测试的执行, 不同测试主机上的测试进度可能差别越来越大, 最终导致不同机器上对待测 Web 应用的访问可能不是按预期来并发的, 所产生的负载压力与预期设定不符。

文中的云测试平台支持分布式负载同步控制功能。对每个负载测试配置, 测试平台支持为其设定一组同步集合点, 典型的同步策略包括按比例同步, 即当一定比例的任务进入同步点后, 已经进入同步点的任务即可继续向下执行; 以及按绝对数量同步, 即当某指定数量的任务进入同步点后, 已经进入同步点的任务即可继续向下执行。

在为负载测试配置设定了同步集合点后, 可以在测试脚本中添加同步集合点进入原语, 如含同步集合点的测试脚本示例中的 `st.rendezvous("dosearch")`。测试脚本执行到该同步集合点语句时, 会进入等待状态, 直到退出同步集合点的条件得到满足。该同步集合点可以使得该集合点后的语句都尽可能在同一时刻得到执行。

```

st = tester();
class Baidu(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.base_url = "https://www.baidu.com/"
    def test_baidu(self):
        driver = self.driver
  
```

```

        driver.get(self.base_url + "/" )
    st.rendevous("dosearch")
        driver.find_element_by_id("su").click()
    if __name__ == "__main__":
        unittest.main()

```

同步集合点仅对负载变化策略中同一时刻点上发起的负载有效。对于原定测试计划中理论上应发生于同一时刻的客户端负载,平台采用同步控制算法以确保持同步集合点能够有效发挥作用,保证不同测试主机间测试进度的同步性。同步控制算法的总体流程如下:

```

input:待执行的负载测试活动
output:每个步骤上的有效负载规模的集合  $\varnothing(\text{load})$ 
begin
    获取负载变化策略,测试集群的主机数量 testClusterSize 和负载
    变化步骤数 steps;
    while step<steps
        读取当前步骤上预设的负载规模 loadScale;
    whiletrue
        为每个负载启动一个负载执行器;
        在负载执行器内执行测试脚本;
        脚本进入 rendezvous 语句后,向测试主机发出 enterRendezvous
        消息;
        当前等待执行的脚本数 actualLoad++;
        if actualLoad==loadScale(测试主机满足退出同步集合点的条
        件) then
            测试主机向测试控制中心发出 enterRendezvous 的消息;
            等待执行的测试主机数 agentCount++;
            actualLoad→ $\varnothing(\text{load})$ ;
            while agentCount<testClusterSize
                阻塞等待执行的测试主机上进入同步集合点的脚本;
            end while
            测试控制中心向测试集群中各测试主机发出 exitRendezvous;
            测试主机将同步退出消息转发给各个正在等待的脚本;
            脚本收到消息后,继续向下执行测试脚本中的后续语句;
        end
    end while
end while
return  $\varnothing(\text{load})$ ;
end

```

6 结束语

研究了多类型负载生成、智能化测试资源分配和分布式负载同步控制技术,实现了一种基于 OpenStack 的大规模云负载测试平台。该平台具有如下特性:支持进程、线程和协程负载并发机制,结合多类型测试脚本生成客户端负载,相较于已有测试工具,客户端负载生成更为高效;支持智能化测试资源分配功能,实现面向云负载测试任务的资源优化分配;支持分布式负载同步控制功能,保证不同测试主机上网络活动的并行

性。该测试平台为测试人员实施大规模负载测试提供了一个功能丰富、经济易用的平台,能够降低大规模负载测试的难度。

参考文献:

- [1] CHEN T H, SYER M D, SHANG W, et al. Analytics-driven load testing: an industrial experience report on load testing of large-scale systems [C]//2017 IEEE/ACM 39th international conference on software engineering: software engineering in practice track (ICSE-SEIP). Buenos Aires: IEEE, 2017: 243-252.
- [2] JIANG Z M, HASSAN A E. A survey on load testing of large-scale software systems [J]. IEEE Transactions on Software Engineering, 2015, 41(11): 1091-1118.
- [3] SCHULZ H, ANGERSTEIN T, HOORN A V. Towards automating representative load testing in continuous software engineering [C]//Companion of the 2018 ACM/SPEC international conference on performance engineering. New York: ACM, 2018: 123-126.
- [4] LING Q L, FENG W Q. Research on automatic test of WEB system based on loadrunner [C]//2018 13th international conference on computer science & education (ICCSE). Piscataway: IEEE, 2018: 1-4.
- [5] HALILI E H. Apache JMeter [M]. Birmingham: Packt Publishing, 2008: 52-74.
- [6] MOSBERGER D, JIN T. Httpperf-a tool for measuring web server performance [J]. ACM SIGMETRICS Performance Evaluation Review, 1998, 26(3): 31-37.
- [7] ABBAS R, SULTAN Z, BHATTI S N. Comparative analysis of automated load testing tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege [C]//2017 international conference on communication technologies (ComTech). Piscataway: IEEE, 2017: 39-44.
- [8] 曹 阳. 基于云平台的软件性能测试技术研究 [J]. 电子技术与软件工程, 2016(17): 67-69.
- [9] 马喜刚. IaaS 云平台测试基准研究及负载生成模块设计与实现 [D]. 西安: 西安电子科技大学, 2017.
- [10] 马亚明. 基于 selenium 的前端自动化云测试平台 [D]. 南京: 南京大学, 2015.
- [11] 杨本生, 袁祥梦, 黄晓光. 基于云计算的软件测试系统框架研究 [J]. 计算机测量与控制, 2014, 22(6): 1683-1686.
- [12] 周 悦. 基于 WEB 系统的云性能测试工具设计与实现 [D]. 北京: 北京工业大学, 2014.
- [13] 李立耀, 赵少卡, 王 焱, 等. Dandelion: OpenStack 云平台的快速部署机制 [J]. 计算机应用, 2015, 35(11): 3070-3074.
- [14] 杨济运, 刘建勋, 姜 磊, 等. 基于协程模型的分布式爬虫框架 [J]. 计算技术与自动化, 2014, 33(3): 126-133.
- [15] 梅志伟. 多目标进化算法综述 [J]. 软件导刊, 2017, 16(6): 204-207.