微服务性能测试利器 Locust (蝗虫)

范亚敏 高级系统架构师

课程详情　完整文稿

你好，我是范亚敏。

在软件开发领域，有三句话深入人心:

- 不能度量就不能管理 (If you can't measure it, you can't manage it)
- 不能度量就不能证明 (If you can't measure it, you can't prove it)
- 不能度量就不能提高 (If you can't measure it, you can't improve it)

这几句话就能看出来，我们必须要度量微服务性能的能力，而度量最有效的手段就是性能测试。

性能测试的目的是要解决三个问题:

- 系统及服务能承受的最大负载是多少?
- 有没有性能瓶颈?
- 如果有性能瓶颈，瓶颈在哪里?

这三个问题需要从测试的结果中分析得出，其中最重要的性能指标有以下三点:
1）响应时间
2）吞吐量
3）成功率

$$\text{吞吐量} = \frac{1s}{\text{响应时间}}$$

想要准确的得出测试结果，度量出这三个关键指标，我们要做好微服务的性能测试，也就需要一款可以实现目的并且称手的测试工具，方便我们对微服务进行维护和扩展。

性能测试的具体实践

JMeter 当然功能强大，也有一定的扩展性，但在这节课我并不想用 JMeter，原因有两点，一是因为 JMeter 是资源消耗黑洞， 每个任务 / 用户都要使用一个线程，二是因为 JMeter 是基于配置的。

相比之下，新一代的性能测试工具 Locust（蝗虫）是基于编程来实现测试用例的性能测试工具，它更加灵活。而且，它使用 Python 代码来定义用户行为，用写 Python 代码的方式来写测试脚本，远离了复杂的配置脚本和图形界面，利用强大而丰富的 Python 类库可以轻松支持各种协议，简单易用，扩展方便。

所以，我选择用 Locust 这个测试工具来给你演示怎么做性能测试。这里以一个帐户管理的微服务为例, 这个例子的源码请参见 https://github.com/walterfan/mdd/blob/master/account/account_service.py, 它是用 Python 基于 Flask 框架写的, 你可以用你熟悉的任何语言来实现它，也就是一个对于 Account 的 Create、Retrieve,、Update ,、Delete 测试。

那我们先上手试一下如何用 Locust 来做个简单的性能测试。Locust 要先写一个脚本文件 ， 它就是模拟 HTTP Client 进行若干个 CRUD 测试，并且测试集选用 SequentialTaskSet， 通过 @task(n) 来指定测试用例的执行顺序

create_account: POST /api/v1/accounts

retrieve_account: GET /api/v1/accounts/{accountId}

update_account: PUT /api/v1/accounts/{accountId}

delete_account: DELETE /api/v1/accounts/{accountId}

让我们来快速实现这几个测试用例，写一个 Python 代码文件，文件名为 account_load_test.py

复制代码

```
import os
import string
import time
import json
# 引入一个自己的工具类库
import utils
from queue  import Queue
# 引入 locust 的相关类库
from locust import HttpUser, between, task
from locust import SequentialTaskSet

PERF_THRESHOLD_MS = 500

BASE_DIR = os.path.dirname(os.path.realpath(__file__))
DATE_FORMAT = '%Y-%m-%dT%H:%M:%S%z'
TEST_USER = "walter"
TEST_PASSWORD = "pass"

logger = utils.init_logger("account-load-test")
# 检查响应时间, 并记录那些响应时间太长的请求
def check_response_time(start, end, name, trackingId):
    duration = end - start
    if duration > PERF_THRESHOLD_MS:
        logger.info("%s response too slow: %d, %s", name, duration, trackingId)

# 测试用例集

class AccountTestSuite(SequentialTaskSet):

    account_queue = Queue()
    auth_headers = utils.getAuthHeaders(TEST_USER, TEST_PASSWORD)

    def on_start(self):
        logger.info("on_start")

    def on_stop(self):
        logger.info("on_stop")

    def list_account(self):
        self.client.get("/api/v1/accounts")

```

```
45
46          http_headers = utils.get_common_headers()
47          http_headers.update(self.auth_headers)
48
49          post_dict = utils.build_account_request()
50
51          post_data = json.dumps(post_dict)
52          logger.info("http_headers: %s", json.dumps(http_headers))
53          logger.info("http body: %s", post_data)
54          start = time.time()
55          response = self.client.post("/api/v1/accounts", headers=http_headers, data=post_data)
56          check_response_time(start, time.time(), "create_account", http_headers['TrackingID'])
57          if (200 <= response.status_code < 300):
58              siteName = post_dict['siteName']
59              logger.info("siteName: %s" % siteName)
60              self.account_queue.put(siteName)
61
62          return response
63
64      #  调用获取帐号的 API 进行测试
65      @task(2)
66      def retrieve_account(self):
67          http_headers = utils.get_common_headers()
68          http_headers.update(self.auth_headers)
69
70          if not self.account_queue.empty():
71              siteName = self.account_queue.get(True, 1)
72              logger.info("retrieve_account by siteName %s", siteName)
73              start = time.time()
74              response = self.client.get("/api/v1/accounts/" + siteName, headers=http_headers,
75                                         name="/api/v1/accounts/siteName")
76              check_response_time(start, time.time(), "retrieve_account", http_headers['TrackingID'])
77              logger.info("retrieve_account's response: %d, %s", response.status_code, response.text)
78              self.account_queue.put(siteName)
79          else:
80              logger.warn("not account to retrieve")
81      #  调用修改帐号的 API 进行测试
82      @task(3)
83      def update_account(self):
84          http_headers = utils.get_common_headers()
85          http_headers.update(self.auth_headers)
86
87          if not self.account_queue.empty():
88              siteName = self.account_queue.get(True, 1)
89              post_dict = utils.build_account_request()
90
91              put_data = json.dumps(post_dict)
92              start = time.time()
93              response = self.client.put("/api/v1/accounts/" + siteName, headers=http_headers, data=put_data,
94                                         name="/api/v1/accounts/siteName")
95              check_response_time(start, time.time(), "update_account", http_headers['TrackingID'])
96              logger.info("response: %d, %s", response.status_code, response.text)
97              self.account_queue.put(siteName)
98          else:
99              logger.warn("not account to update")
100     #  调用删除帐号的 API 进行测试
101     @task(4)
102     def delete_account(self):
103         http_headers = utils.get_common_headers()
104         http_headers.update(self.auth_headers)
105
106         if not self.account_queue.empty():
107             siteName = self.account_queue.get(True, 1)
108             start = time.time()
109             response = self.client.delete("/api/v1/accounts/" + siteName, headers=http_headers,
110                                           name="/api/v1/accounts/siteName")
111             check_response_time(start, time.time(), "delete_account", http_headers['TrackingID'])
112             logger.info("response: %d, %s", response.status_code, response.text)
113         else:
114             logger.warn("not account to delete")
115
116 # 模拟真实用户进行测试
117 class LocustTestUser(HttpUser):
118     # 指定测试任务集
119     tasks = [AccountTestSuite]
```

这段代码其中除了引用标准库，还导入了一个工具库文件 utils.py, 源码附后。

整个测试文件不长，也就一百多行，它的结构非常简单。你只要记住三点：

第一，创建一个继承自 HttpUser 类的 LocustForScenarioGroup 来模拟真实的用户，它要执行的测试任务是 AccountTestSuite。

第二，AccountTestSuite 类也就是我们的测试用例集，它继承自 SequentialTaskSet， 因为我们的 CRUD 是需要按顺序执行的，没有添加之前是不能修改和删除的，所以我们在各个测试用例上标注 "@task(N)" , 这里的 N 是执行的顺序。

第三，各个测试用例中就可以用 Locust 内置的 Http Client 来调用 API 进行测试了

到这里，咱们写好了微服务和对应的测试脚本，接下来就可以做性能测试了。

首先要启动被测试的微服务，为简单起见，就在本机上启动这个微服务，侦听端口为 5000

复制代码

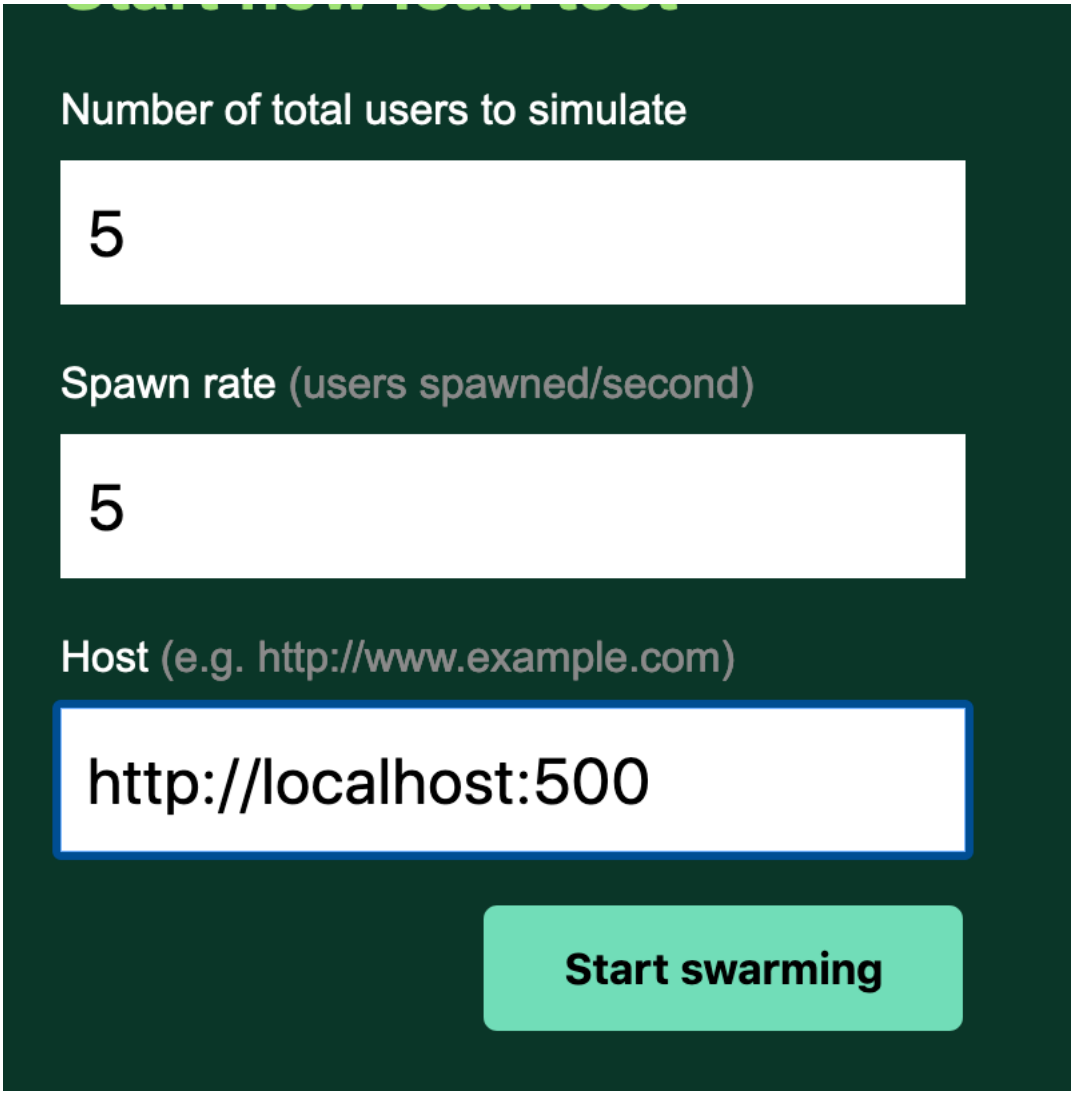
```
python account_service.py
```

启动 Locust 工具来针对这个微服务做性能测试，命令如下

复制代码

```
$ locust -f account_load_test.py
```

打开 http://localhost:8089，你就能看到一个启动界面，我们可以指定模拟的用户数，以及孵化率（每秒增加的用户数），以及所要测试的服务器地址



点击“Start swarming”，就可以启动测试了，它会像蝗虫一样蜂拥而至，纷纷请求微服务的 API。

这是 Locust 的图形界面启动方式，你可以随时点击右上角的停止和启动按钮进行不同的压力测试。当然了，你也可以用另一种方法，使用命令行的方式进行梯度加压：

复制代码

```
$ locust -f account_load_test.py --host=http://localhost:5000 --no-web c 100 -r 100 -t 30m --csv=100.csv
locust -f account_load_test.py --host=http://localhost:5000 --no-web c 200 -r 200 -t 30m --csv=200.csv
locust -f account_load_test.py --host=http://localhost:5000 --no-web c 400 -r 400 -t 30m --csv=400.csv
locust -f account_load_test.py --host=http://localhost:5000 --no-web c 800 -r 800 -t 30m --csv=800.csv
```

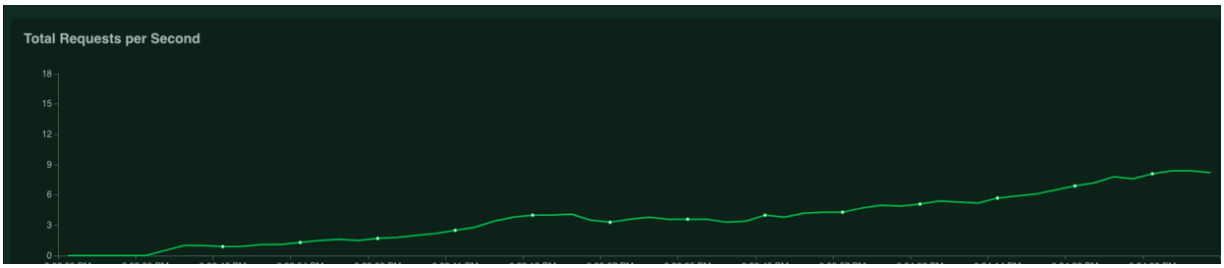
当性能测试开始了，这时候，你可以去倒杯咖啡，去茶水间和同事聊聊天了，过一会儿回来点击性能测试页面，可以看到一个性能测试报告页面，通过这些统计信息，你就能了解到测试的统计数据情况：

Statistics Charts Failures Exceptions Download Data

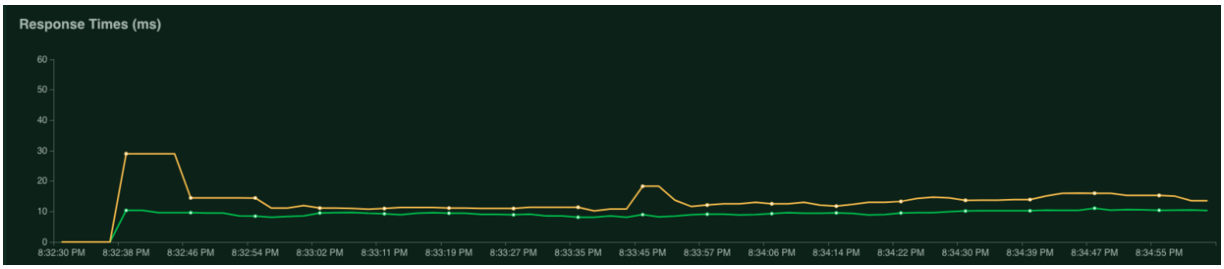
Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/api/v1/accounts	51	0	10	10	8.09812545763672	29.01005744934082	123	1.3
DELETE	/api/v1/accounts/siteName	34	0	10	10	8.04591178894043	11.317014694213867	0	0.8
GET	/api/v1/accounts/siteName	24	0	8	9	6.8359375	14.590978622436523	123	0.3
PUT	/api/v1/accounts/siteName	34	34	7	7	5.584955215454102	8.59212875366211	0	1.4
	Total	143	34	9	9	5.584955215454102	29.01005744934082	65	3.8

这样，微服务性能测试最看重三点性能指标：响应时间、吞吐量、成功率对应于 API 的响应时间、吞吐量、成功率，我们点击“chart”标签就能够看到前两个关键指标：

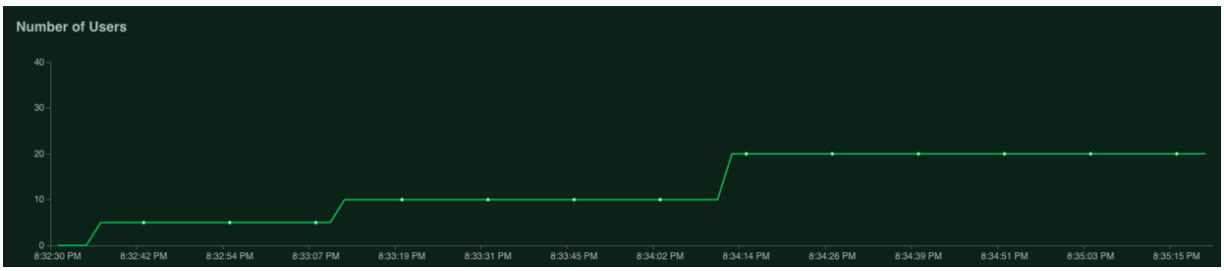
RPS(Request Per Second): 每秒请求数的变化，它反映了吞吐量



Response Time 响应时间的变化



Number of Users, 用户并发数量的变化, 它反映了系统能承受的并发请求吞吐量



至于第三个指标——成功率，我们在“Statistics”和“Failure ”标签页中都可以看到。

通过 Locust 生成的测试报告，不仅可以在 Web 页面上看到，你还可以导出 csv 文件，对报告内容进行详细分析，再结合微服务的各种度量数据进行分析。

另外，值得一提的是 Locust 可以生成从测试工具这一客户端角度的性能测试报告，而我们在服务器端需要通过日志、内置的度量数据进行分析，根据 Locust 报告中记录的性能拐点、慢响应的 TrackingId，进行有针对性的分析。（这段老师没有读）

这里有三个要点，你一定要了解：

的日志中，在后期分析的时候通过 TrackingId 和服务器端的相关日志以及度量数据关联起来，分析慢在哪里。

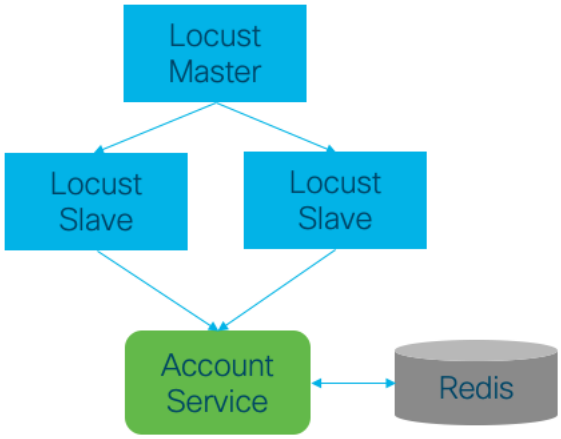
第二点，在 Locust 生成的请求的 HTTP 头里需要带上 HTTP 头域“Authorization：$token”，这里我只演示简单的 HTTP Basic 认证方式，现实世界中常用 OAuth Token 这一认证方式，Token （令牌）是有时效期的，所以我们需要用一个定时器在 Token 过期之前重新申请 Token。这个方法可以放在 "Account_Load_test.py" 的 on_start 方法中。

复制代码

```
from threading import Timer

    def get_access_token(self):
        logger.info("Token is refreshing, next is after %s seconds", token_refresh_time)
        self.auth_headers = oauth_client.getAuthHeaders()
        Timer(token_refresh_time, self.get_access_token).start()
```

第三点， 实际工作中一台测试机器产生的压力往往不够时，我们可以使用多台服务器来加压，就像这张图给我们展示的这样，有 一台 Master Server，若干台 Slave Server ，由 Master Server 控制若干台 Slave Server 发送海量的请求进行压力测试。



总结

好了，讲到这里，今天的分享也就结束了，最后我来给你总结一下。今天我们首先了解了为什么要做性能测试，阐述了性能测试的 3 个关键指标 – 响应时间，吞吐量，成功率。然后，通过演示 Locust 这个测试工具的操作方法，讲解了怎么做微服务性能测试，轻松得出想要的性能测试结果。我们知道，性能测试必须结合监控与度量，有了 Locust 生成的客户端性能测试数据，再结合服务器端的日志和度量指标，我们就能有的放矢地做好性能优化。

所有源代码可以在这里下载： https://github.com/walterfan/mdd/blob/master/account

好，我是范亚敏，希望我的分享可以帮助到你，也希望你在视频下方的留言区和我讨论。

hunkier

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Command + Enter 发表 0/2000字 提交留言