

08 | 案例：手把手教你编写最简单的性能脚本

2020-01-01 高楼

性能测试实战30讲

[进入课程 >](#)



讲述：高楼

时长 29:09 大小 26.70M

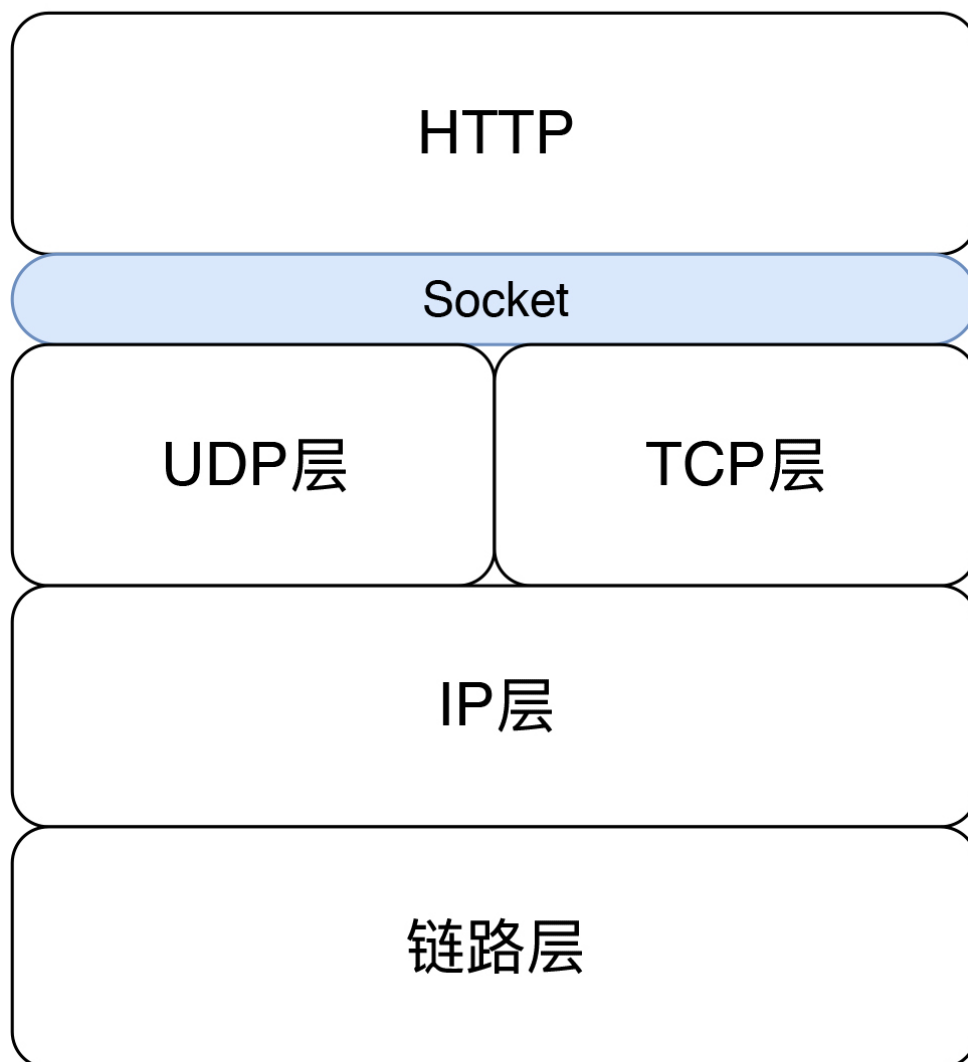


通常我们会遇到要手写脚本的时候，就要针对一些接口编写脚本。这时候，我们需要知道接口规范和后台的数据是什么。而有些性能测试工程师写脚本时，并不知道后端的逻辑，只知道实现脚本，事实上，只知道实现脚本是远远不够的。

在这一篇文章中，我不打算讲复杂的内容，只想针对新手写一步步的操作，描述最简单的脚本编写。如果你已经具有丰富的脚本编写经验，会觉得本文很简单。

我没有打算把 JMeter 的功能点一一罗列出来，作为一个性能测试的专栏，不写一下脚本的实现似乎不像个样子。在脚本实现中，我们最常用的协议就是 HTTP 和 TCP 了吧，所以在今天的内容里，我简单地说一下如何编写 HTTP 和 TCP 脚本，以应测试主题。

我先画个图说明一下。



这样的图做性能的人一定要知道，相信很多人也画的出来。

我们知道 HTTP 是应用层的协议之一，现在很多场景都在用它，并且是用的 HTTP1.1 的版本，对应的是 RFC2616，当然还有补充协议 RFC7231、6265。

HTTP 中只规定了传输的规则，□规定了请求、响应、连接、方法、状态定义等。我们写脚本的时候，必须符合这些规则。比如为什么要在脚本中定义个 Header？Header 里为什么要那样写？这些在 RFC 中都说得明明白白了。

还有一点也需要注意，HTTP 是通过 Socket 来使用 TCP 的，Socket 做为套接层 API，它本身不是协议，只规定了 API。

而我们通常在 JMeter 中写 TCP 脚本，就是直接调用 Socket 层的 API。TCP 脚本和 HTTP 脚本最大的区别就是，TCP 脚本中发送和接收的内容完全取决于 Socket server 是怎么处理的，并没有通用的规则。所以脚本中也就只有根据具体的项目来发挥了。

手工编写 HTTP 脚本

服务端代码逻辑说明

我们先自己编写一小段服务端代码的逻辑。现在用 Spring Boot 写一个示例，其实就是分分钟的事情。我们做性能测试的人至少要知道访问的是什么东西。

Controller 关键代码如下：

 复制代码

```
1 @RestController
2 @RequestMapping(value = "pa")
3 public class PAController {
4
5     @Autowired
6     private PAService paService;
7
8     // 查询
9     @GetMapping("/query/{id}")
10    public ResultVO<User> getById(@PathVariable("id") String id) {
11        User user = paService.getById(id);
12        return ResultVO.<User>builder().success(user).build();
13    }
14 }
```

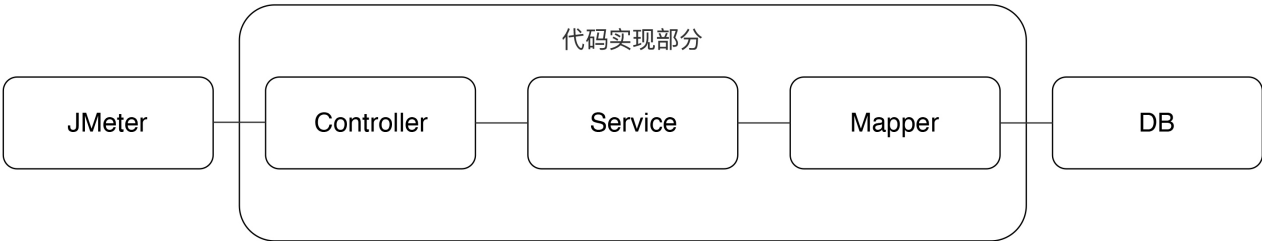
Service 关键代码如下：

 复制代码

```
1 public User getById(String id) {
2     return mapper.selectByPrimaryKey(id);
3 }
```

用 MyBatis 组件实现对 Mapper 的操作。由于不是基础开发教程，这里只是为了说明逻辑，如果你感兴趣的话，可以自己编写一个接口示例。

逻辑调用关系如下：



数据库中表的信息如下：

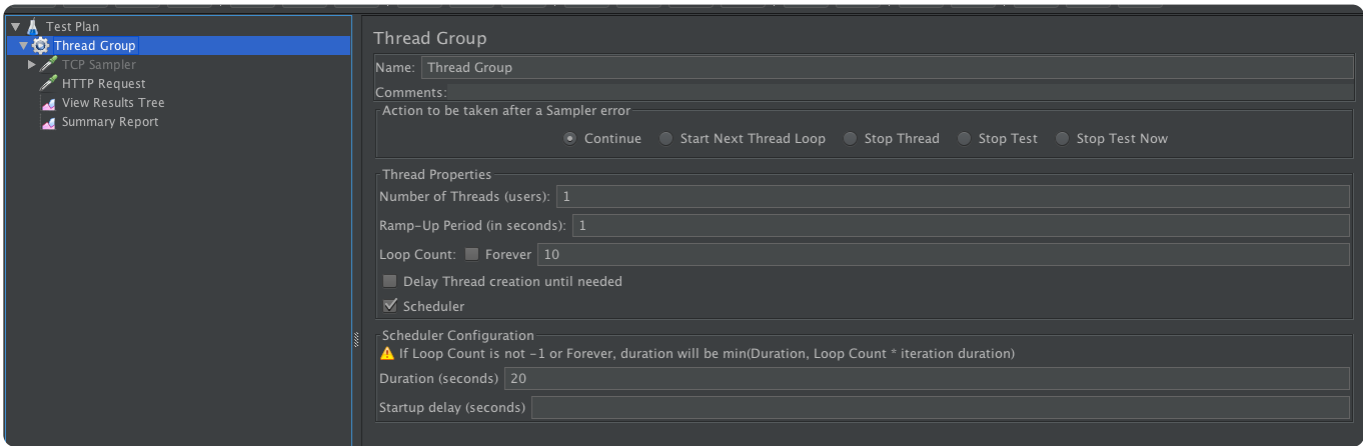
id	user_number	user_name	org_id	email	mobile	create_time
0808050c-0ae0-11ea-af5f-00163e124cff	00009496	Zee_pabc9495	NULL	test9495@dunshan.com	17600009498	2019-11-19 23:19:51
0a3ce698-0c2f-11ea-af5f-00163e124cff	00009496	李六	NULL	test9495@dunshan.com	17600009498	2019-11-21 15:17:56

我们先看这个接口的访问逻辑：JMeter——SprintBoot 的应用——MySQL。

1. 编写 JMeter 脚本

1.1 创建线程组

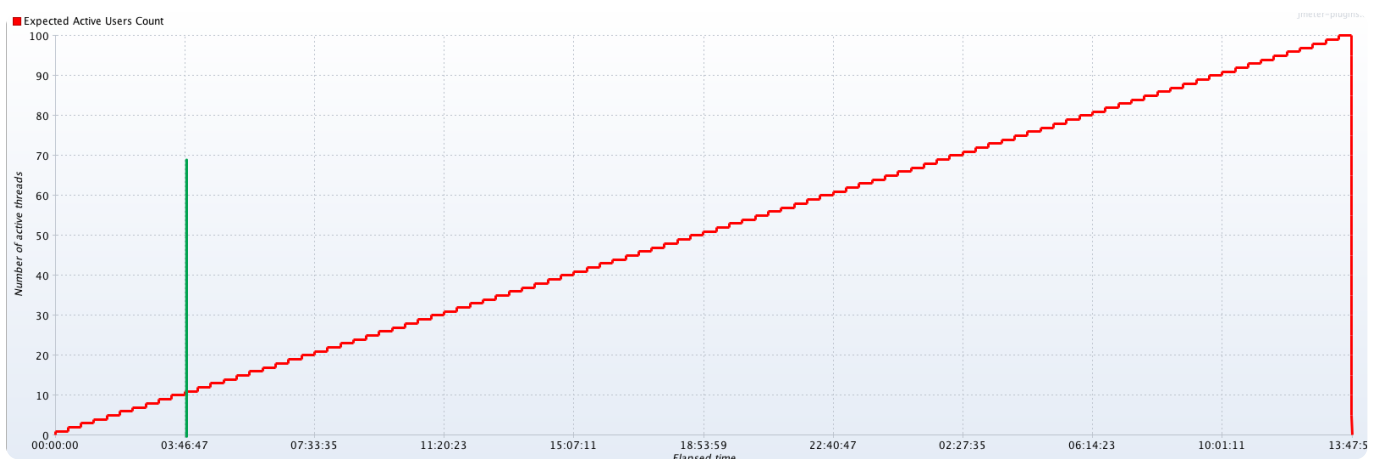
首先创建一个线程组，配置如下：



在这个线程组中，有几个关键配置，我来一一说明一下。

Number of Threads(users): 我们都知道这是 JMeter 中的线程数，也可以称之为用户数。但是在 [第 2 篇文章](#) 中，我已经说得非常明确了，这个线程数是产生 TPS 的，而一个线程产生多少 TPS，取决于系统的响应时间有多快。所以我们用 TPS 这个概念来承载系统的负载能力，而不是用这里的线程数。

Ramp-up Period(in seconds): 递增时间，以秒为单位。指的就是上面配置的线程数将在多长时间内会全部递增完。如果我们配置了 100 线程，这里配置为 10 秒，那么就是 $100/(10s*1000ms)=1$ 线程 /10ms；如果我们配置了 10 线程，这里配置为 1 秒，则是 $10/1000=1$ 线程 /10ms。这时我们要注意了哦，在 10 线程启动的这个阶段中，对服务器的压力是一样的。示意图如下： 什么意思？



Loop Count 这个值指的是一个线程中脚本迭代的次数。这里你需要注意，这个值和后面的 Scheduler 有一个判断关系，下面我们会提到。

Delay Thread creation until needed: 这个含义从字面看不是特别清楚。这里有一个默认的知识点，那就是 JMeter 所有的线程是一开始就创建完成的，只是递增的时候会按照上面的规则递增。如果选择了这个选项，则不会在一开始创建所有线程，只有在需要时才会创建。这一点和 LoadRunner 中的初始化选项类似。只是不知道你有没有注意过，基本上，我们做性能测试的工程师，很少有选择这个选项的。选与不选之间，区别到底是什么呢？

如果不选择，在启动场景时，JMeter 会用更多的 CPU 来创建线程，它会影响前面的一些请求的响应时间，因为压力机的 CPU 在做其他事情嘛。

如果选择了的话，就会在使用时再创建，CPU 消耗会平均一些，但是这时会有另一个隐患，就是会稍微影响正在跑的线程。这个选项，选择与否，取决于压力机在执行过程中，它

能产生多大的影响。如果你的线程数很多，一旦启动，压力机的 CPU 都被消耗在创建线程上了，那就可以考虑选择它，否则，可以不选择。

Scheduler Configuration: 这里有一句重要的话，If Loop Count is not -1 or Forever, duration will be $\min(\text{Duration}, \text{Loop Count} * \text{iteration duration})$ 。举例来说，如果设置了 Loop Count 为 100，而响应时间是 0.1 秒，那么 $\text{Loop Count} * \text{iteration duration}$ (这个就是响应时间) = $100 * 0.1 = 10$ 秒。

即便设置了 Scheduler 的 Duration 为 100 秒，线程仍然会以 10 秒为结束点。

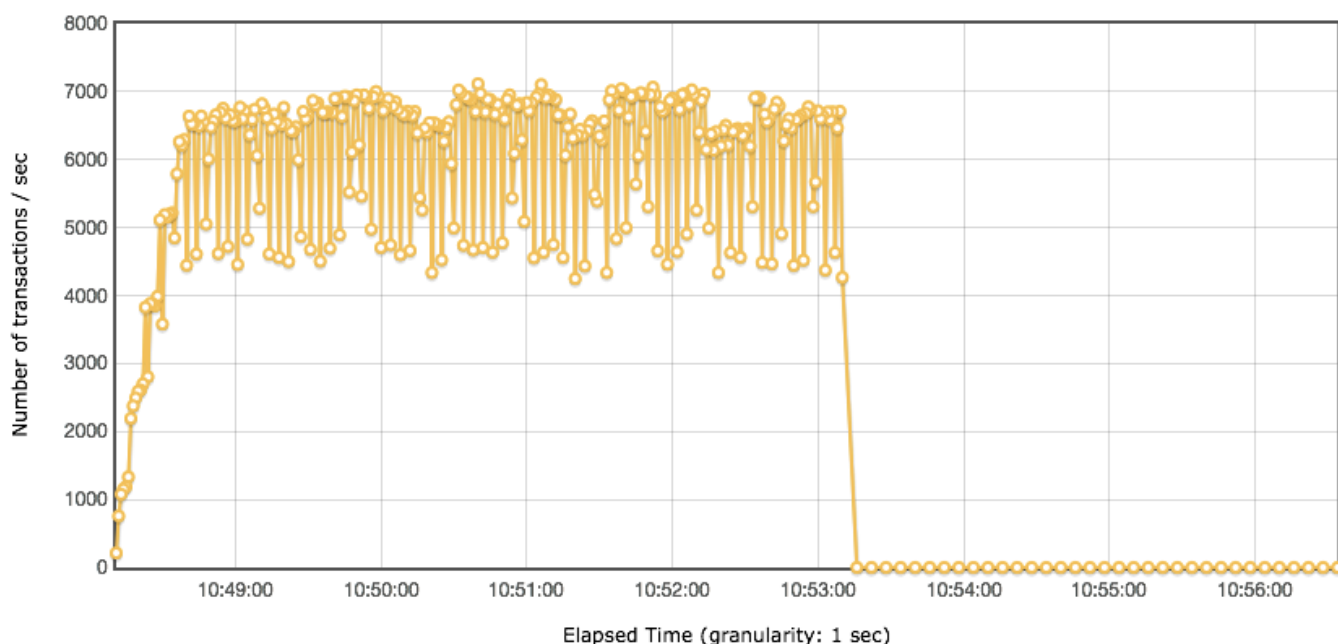
如果没有设置 Scheduler 的 Duration，那么你会看到，在 JMeter 运行到 10 秒时，控制台中会出现如下信息：

[复制代码](#)

```
1 2019-11-26 10:39:20,521 INFO o.a.j.t.JMeterThread: Thread finished: Thread G
```

有些人不太理解这一点，经常会设置迭代次数，同时又设置 Scheduler 中的 Duration。而对 TPS 来说，就会产生这样的图：

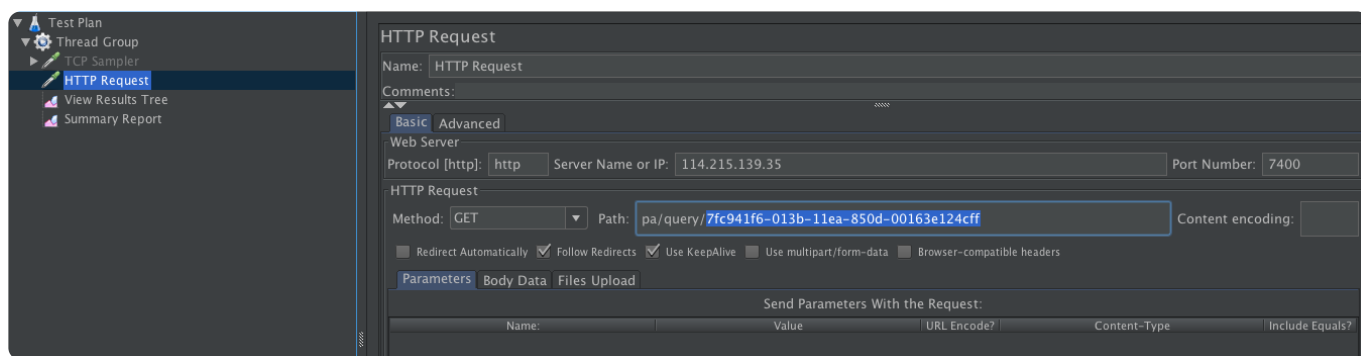
Transactions Per Second



场景没执行完，结果 TPS 全掉下去了，于是开始查后端系统，其实和后端没有任何关系。

1.2 创建 HTTP Sampler

1.2.1 GET 接口



看上图，我将 Method 选择为 GET。为什么要选择它？往上看我们的接口注解，这是一个 GetMapping，所以这里要选择 GET。

再看 path 中，这里是 /pa/query/0808050c-0ae0-11ea-af5f-00163e124cff，对应着 “/query/{id}”。

然后执行：

```
1 User user = paService.getById(id);
```

复制代码

返回执行结果：

```
1 return ResultV0.<User>builder().success(user).build();
```

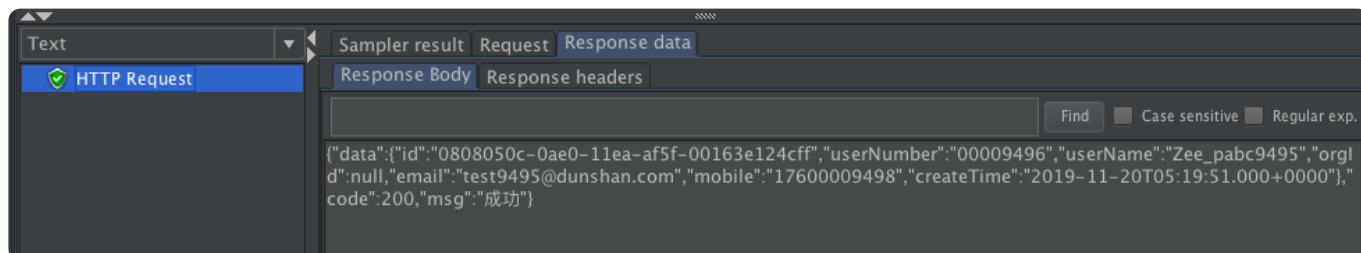
复制代码

为什么要解释这一段呢？

做开发的人可能会觉得，你这个解释毫无意义呀，代码里已经写得很清楚了。事实上，在我的工作经历中，会发现很多做性能测试脚本的，实际上并不知道后端采用了什么样的技术，实现的是什么样的逻辑。

所以还是希望你可以自己写一些 demo，去了解一些逻辑，然后在排除问题的时候，就非常清楚了。

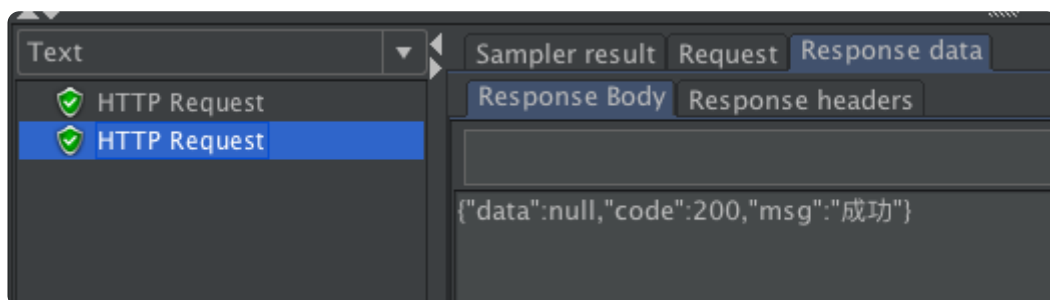
接着我们执行脚本，就得到了如下结果：



这样一个最简单的 GET 脚本就做好了。

前面我们提到过，URL 中的 ID 是 0808050c-0ae0-11ea-af5f-00163e124cff，这个数据来自于数据库中的第一条。

如果我们随便写一个数据，会得到什么结果呢？



你会看到，结果一样得到了 200 的 code，但是这个结果明显就不对了，明明没有查到，还是返回了成功。

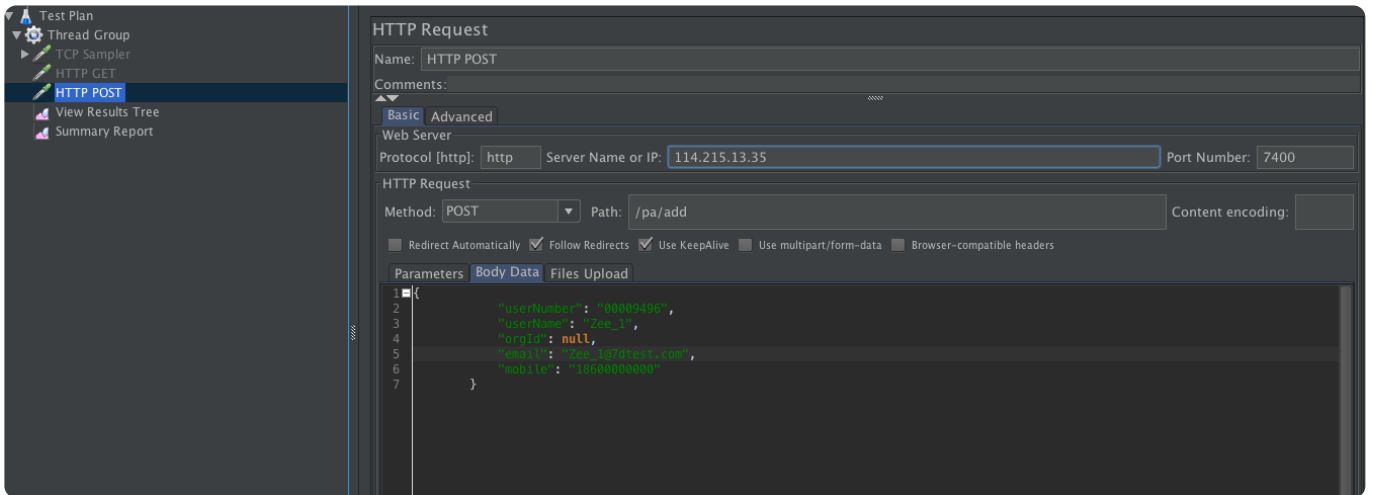
所以说，业务的成功，只能靠业务来判断。这里只是查询成功了，没返回数据也是查询成功了。我将在后面给你说明如何加断言。

1.2.2 POST 接口

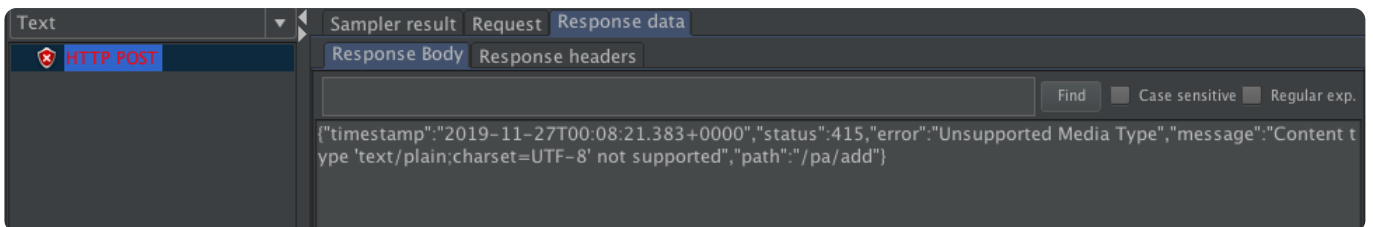
下面我将 Method 改为 POST，POST 接口与 GET 接口的区别有这么几处：

1. 要把 Path 改为 /pa/add;

2. 输入 JSON 格式的 Body Data。



执行起来，查看下结果。



你会发现上来就错了，提示如下：

复制代码

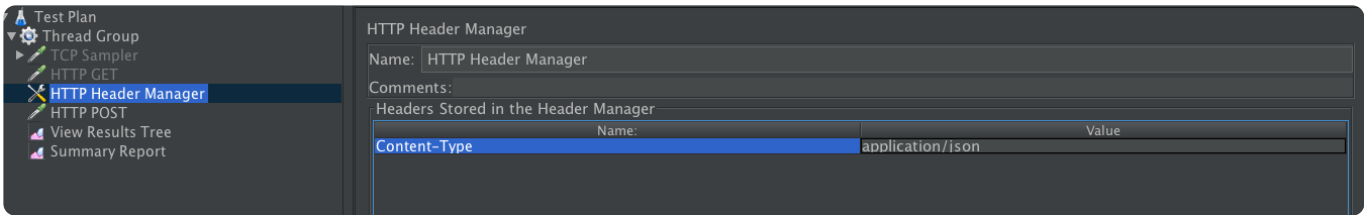
```
1  "status":415,"error":"Unsupported Media Type","message":"Content type 'text/pli
```

这里你需要注意，无论遇到什么问题，都要针对问题来处理。当看不懂问题信息时，先查资料，想办法看懂。这是处理问题的关键，我发现很多做性能测试的新同学，一旦碰到问题就懵了，晕头转向地瞎尝试。

我经常对我的团队成员说，先看懂问题，再处理问题，别瞎蒙！

上面这个问题其实提示得很清楚：“不支持的媒体类型”。这里就两个信息，一个是 Content type，一个是 charset。它们是 JMeter 中 HTTP Header 里默认自带的。我们要发送的是 JSON 数据，而 JMeter 默认是把它当成 text 发出去的，这就出现了问题。所以我们要加一个 Header，将 Content type 指定为 JSON。

加一个 HTTP Header，如下所示：

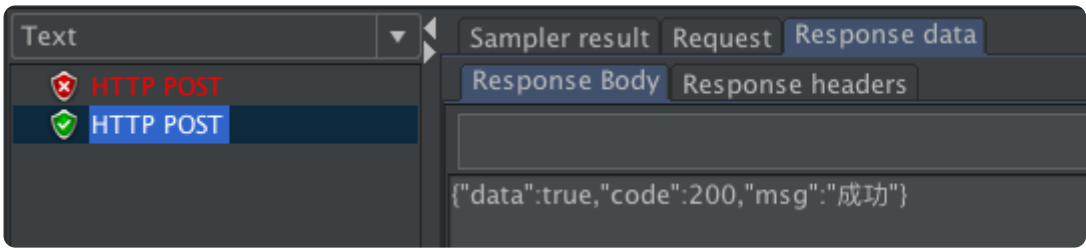


如果你不知道加什么样的 Header，建议你用 HTTP 抓包工具抓一个看一看，比如说用 Charles，抓到如下信息：

Status	Complete
Response Code	200
Protocol	HTTP/1.1
► TLS	-
Method	POST
Kept Alive	No
Content-Type	application/json; charset=UTF-8
Client Address	127.0.0.1:57324

这时你就会知道头里的 Content-Type 原来是 `application/json; charset=UTF-8`。这里的 `charset=UTF-8` 可以不用写，因为它和默认的一样。

这时再回放，你就会看到如下结果：



到此，一个 POST 脚本就完成了。是不是很简单。

在这里，我需要跟你强调的是，手工编写 HTTP 脚本时，要注意以下几点：

1. 要知道请求的类型，我们选择的类型和后端接口的实现类型要是一致的。
2. 业务的成功要有明确的业务判断（在下面的 TCP 中，我们再加断言来判断）。
3. 判断问题时，请求的逻辑路径要清晰。

编写完 HTTP 脚本时，我们再来看一下如何编写 TCP 脚本。

手工编写 TCP 脚本

服务端代码逻辑说明

我在这里写一个非常简单的服务端接收线程（如果你是开发，不要笑话，我只是为了说明脚本怎么写）。

 复制代码

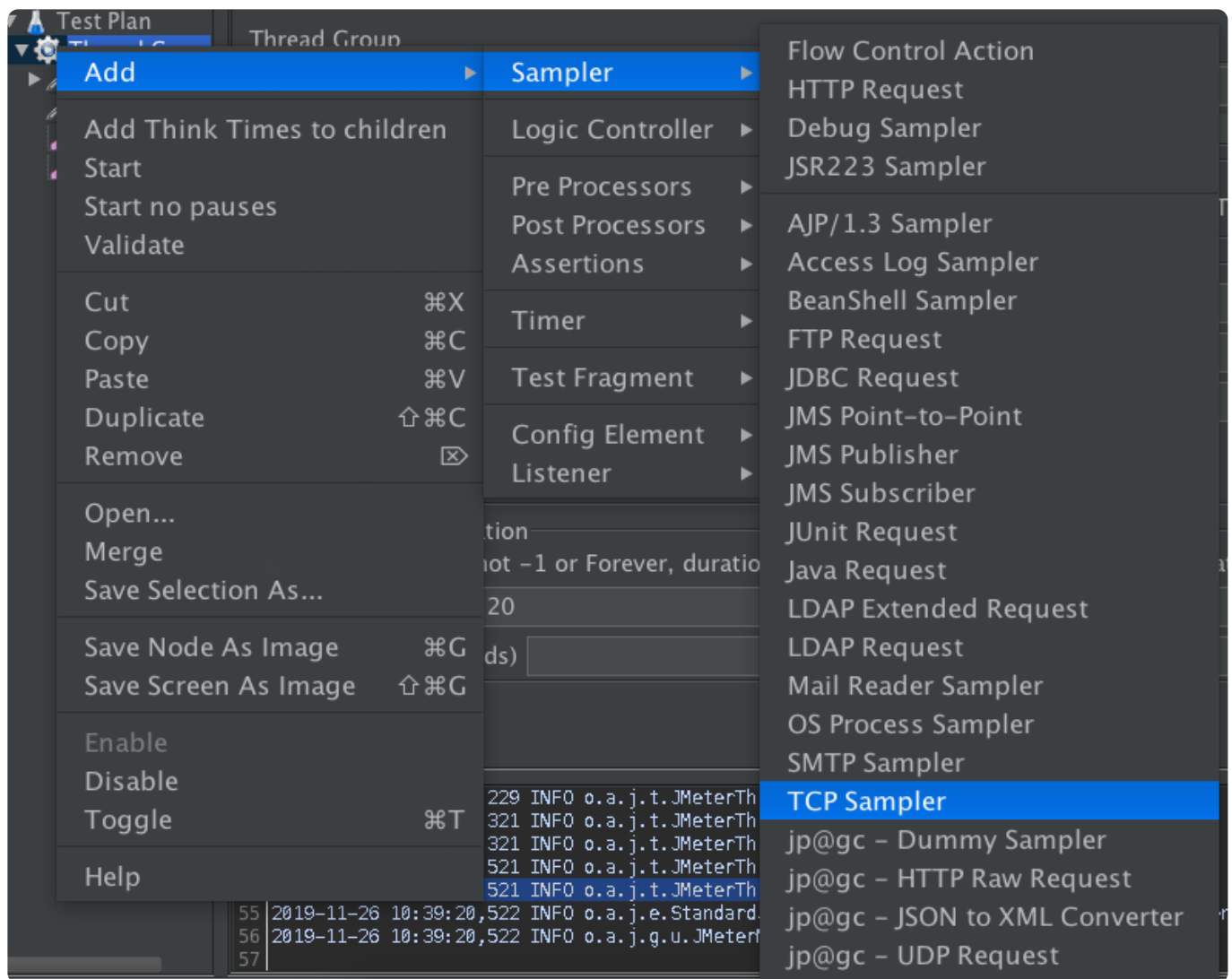
```
1 package demo.socket;
2
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.io.OutputStream;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8 import java.util.concurrent.ArrayBlockingQueue;
9 import java.util.concurrent.ThreadPoolExecutor;
10 import java.util.concurrent.TimeUnit;
11
12 public class SocketReceiver {
13     // 定义初始
14     public static final int corePoolSize = 5;
15     // 定义最大线程池
16     public static final int maximumPoolSize = 5;
17     // 定义 socket 队列长度
18     public static final int blockingQueue = 50;
19
20
21     /**
22      * 初始化并启动服务
23      */
24     public void init() {
25         // 定义线程池
26         ThreadPoolExecutor executor = new ThreadPoolExecutor(corePoolSize, maximumPoolSize,
27             TimeUnit.MILLISECONDS, new ArrayBlockingQueue(blockingQueue));
28         // 定义 serverSocket
29         ServerSocket serverSocket = null;
30         try {
31             // 启动 serverSocket
32             serverSocket = new ServerSocket(Constants.PORT);
33             // 输出服务启动地址
34             System.out.println(" 服务已启动:" + serverSocket.getLocalSocketAddress().toString());
35             // 接收信息并传递给线程池
36             while (true) {
37                 Socket socket = serverSocket.accept();
38                 executor.submit(new Handler(socket));
39             }
40         } catch (IOException e) {
41             e.printStackTrace();
42         }
43     }
44 }
```

```
39     }
40 } catch (IOException e) {
41     e.printStackTrace();
42 } finally {
43     if (serverSocket != null) {
44         try {
45             serverSocket.close(); // 释放 serverSocket
46         } catch (IOException e) {
47             e.printStackTrace();
48         }
49     }
50 }
51 }
52
53 // 处理请求类
54 class Handler implements Runnable {
55
56     private Socket socket;
57
58     public Handler(Socket socket) {
59         this.socket = socket;
60     }
61
62     public void run() {
63         try {
64             // 接收客户端的信息
65             InputStream in = socket.getInputStream();
66             int count = 0;
67             while (count == 0) {
68                 count = in.available();
69             }
70             byte[] b = new byte[count];
71             in.read(b);
72             String message = new String(b);
73             System.out.println(" receive request: " + socket.getInetAddress() + "
74
75             // 睡 2 秒模拟思考时间, 这里是为了模拟服务器端的业务处理时间
76             try {
77                 Thread.sleep(2000);
78             } catch (InterruptedException e) {
79                 e.printStackTrace();
80             }
81
82             // 向客户端发送确认消息
83             // 定义输出流 outer
84             OutputStream outer = socket.getOutputStream();
85             // 将客户端发送的信息加上确认信息 ok
86             String response = message + " is OK";
87             // 将输入信息保存到 b_out 中
88             byte[] b_out = response.getBytes();
89             // 写入输入流
90             outer.write(b_out);
```

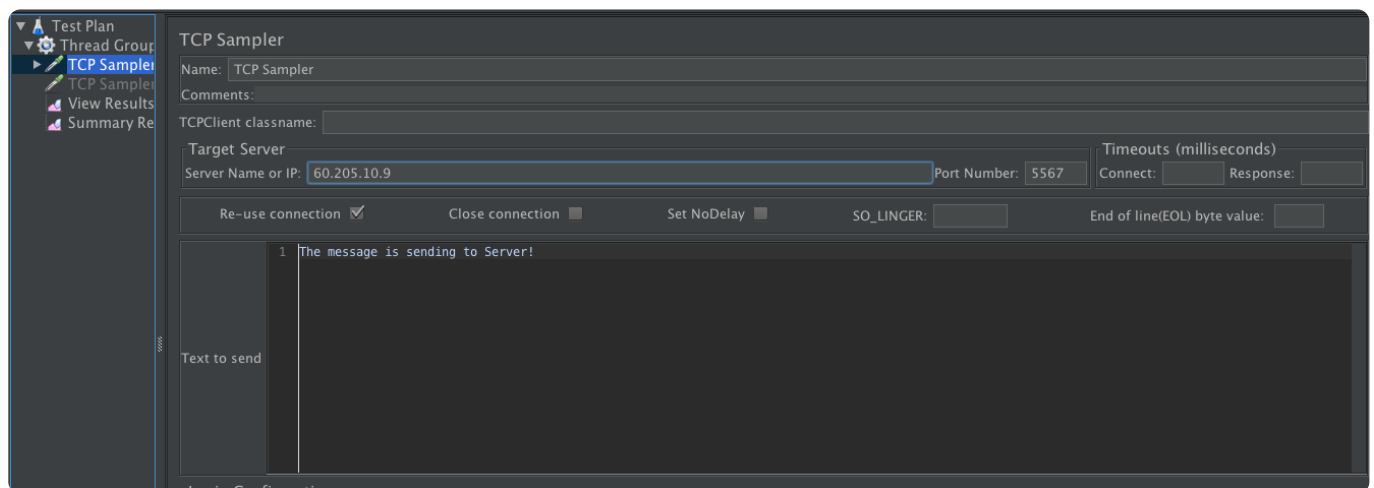
```
91         // 推送输入流到客户端
92         outer.flush();
93
94     } catch (IOException e) {
95         e.printStackTrace();
96     } finally {
97         // 关闭 socket
98         try {
99             socket.close();
100         } catch (IOException e) {
101             e.printStackTrace();
102         }
103     }
104 }
105 }
106
107 // 程序入口
108 public static void main(String[] args) {
109     // 定义服务端
110     SocketReceiver receiver = new SocketReceiver();
111     // 启动服务端
112     receiver.init();
113 }
114 }
```

编写 JMeter 脚本

首先创建 TCP Sampler。右键点击 Thread Group - Add - Sampler - TCP Sampler 即可创建。



输入配置和要发送的信息。



IP 地址和端口是必须要输入的。对于创建一个 TCP 协议的 JMeter 脚本来说，简单地说，过程就是这样的：创建连接 - 发数据 - 关闭连接。

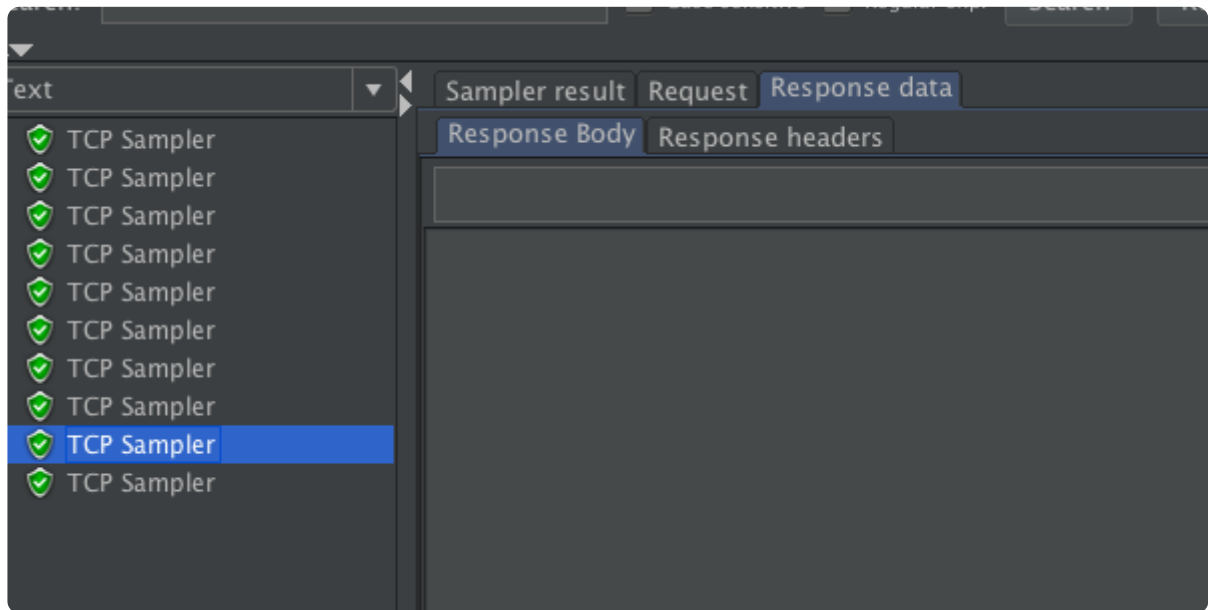
就这样，这个手工的脚本就完成了。

你可能会问，就这么简单吗？是的，手工编写就是这么简单。

但是（对嘛，但是才是重点），通常我们在创建 TCP 协议的脚本时，都是根据业务接口规范来说的，**复杂点其实不在脚本本身上，而是在接口的规则上。**

添加断言

我回放了一下脚本，发现如下情况：



都执行对了呀，为什么下面的没有返回信息呢？这种情况下只有第一个请求有返回信息，但是下面也没有报错。这里就需要注意了。

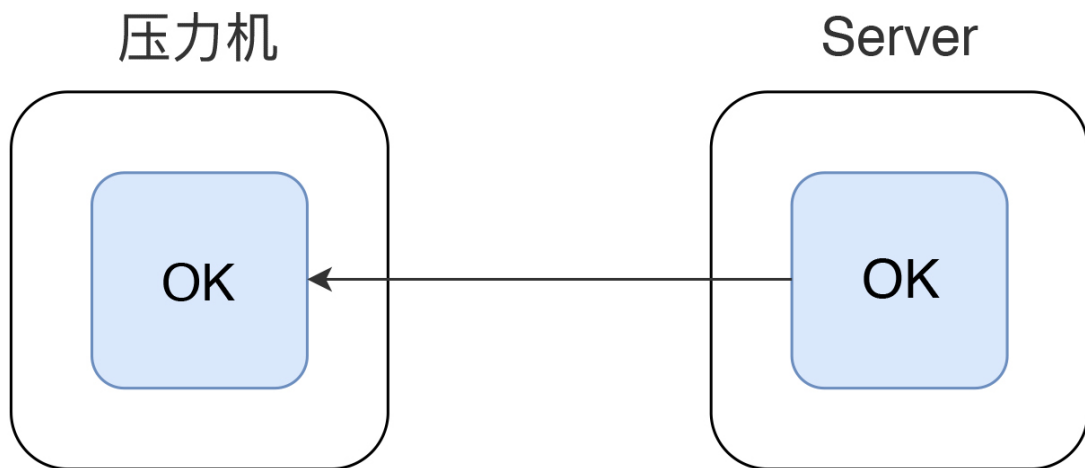
测试工具的成功，并不等于业务的成功。

所以我们必须要做的就是响应断言，也就是返回值的判断。在 JMeter 中，断言有以下这些：

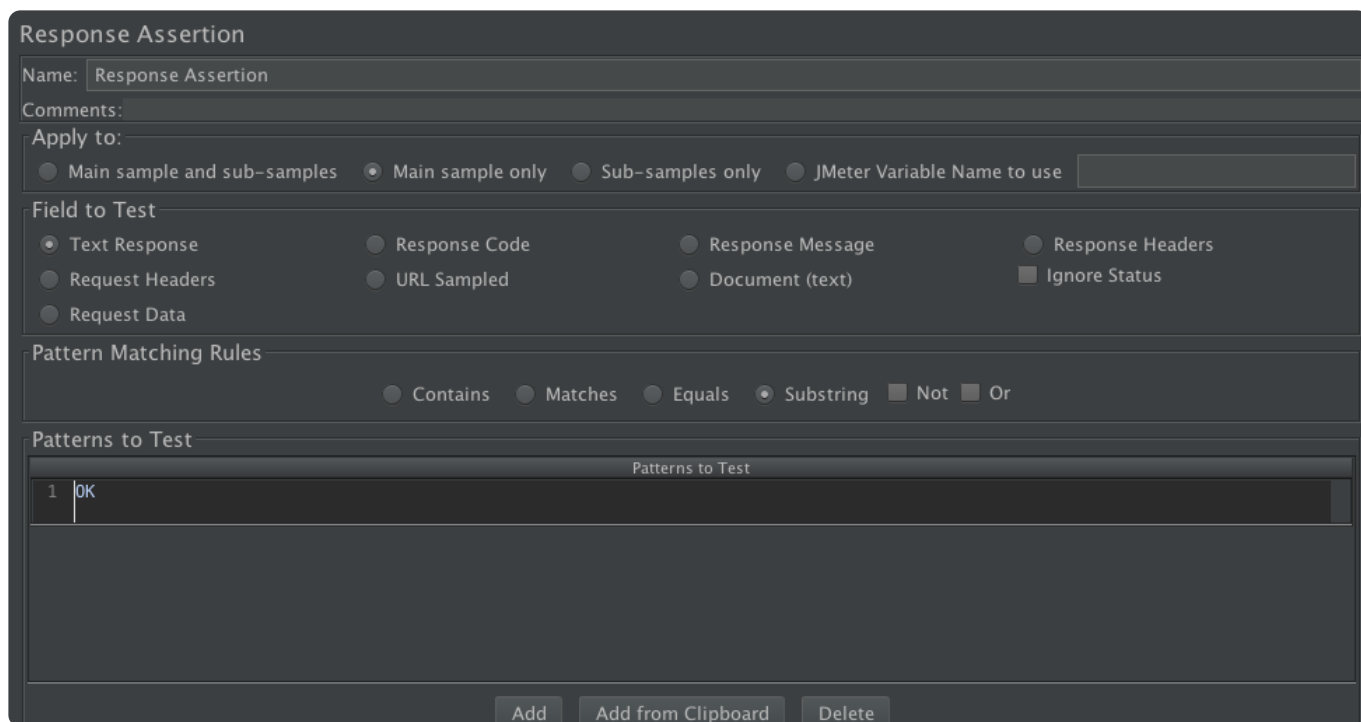

```
Response Assertion
JSON Assertion
Size Assertion
JSR223 Assertion
XPath Assertion

Compare Assertion
Duration Assertion
HTML Assertion
MD5Hex Assertion
SMIME Assertion
XML Assertion
XML Schema Assertion
jp@gc - JSON/YAML Path Assertion
BeanShell Assertion
```

因为今天的文章不是工具的教程，所以我不打算全讲一遍。这里我只用最基础的响应断言。
什么是断言呢？



断言指的就是服务器端有一个业务成功的标识，会传递给客户端，客户端判断是否正常接收到了这个标识的过程。



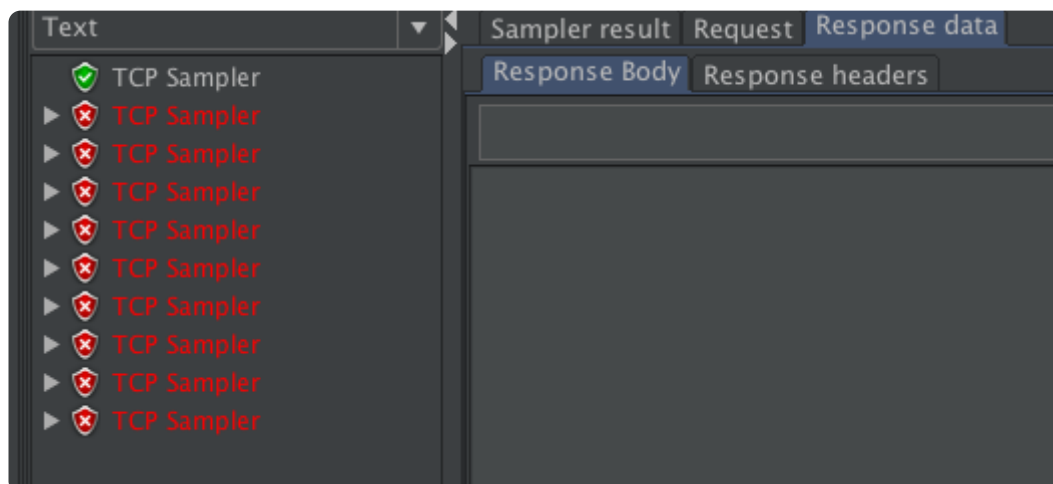
在这里我添加了一个断言，用以判断服务器是否返回了 OK。你要注意这个“OK”是从哪来的哦，它是从服务端的这一行代码中来的。

 复制代码

```
1 String response = message + " is OK";
```

请注意，这个断言的信息，一是可以判断出业务的正确性。我在工作中发现有些人用页面中一些并不必要的文字来判断，这样就不对了，我们应该用有业务含义的判断标识。


如果我们再次回放脚本，你会发现除了第一个请求，后面 9 个请求都错了。



所以，在做脚本时，请你一定要注意，断言是必须要加的。

长短连接的问题

既然有错，肯定是要处理。我们查看一下 JMeter 的控制台错误信息：

 复制代码

```
1 2019-11-26 09:51:51,587 ERROR o.a.j.p.t.s.TCPSampler:
2 java.net.SocketException: Broken pipe (Write failed)
3   at java.net.SocketOutputStream.socketWrite0(Native Method) ~[?:1.8.0_111]
4   at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:109) ~[?:1.8.0_111]
5   at java.net.SocketOutputStream.write(SocketOutputStream.java:141) ~[?:1.8.0_111]
6   at org.apache.jmeter.protocol.tcp.sampler.TCPClientImpl.write(TCPClientImpl.java:109)
7   at org.apache.jmeter.protocol.tcp.sampler.TCPSampler.sample(TCPSampler.java:109)
8   at org.apache.jmeter.threads.JMeterThread.doSampling(JMeterThread.java:622)
9   at org.apache.jmeter.threads.JMeterThread.executeSamplePackage(JMeterThread.java:446)
10  at org.apache.jmeter.threads.JMeterThread.processSampler(JMeterThread.java:437)
11  at org.apache.jmeter.threads.JMeterThread.run(JMeterThread.java:253) [Apache/2.4.18]
12  at java.lang.Thread.run(Thread.java:745) [?:1.8.0_111]
```

从字面上来看，就是通道瓦塔（被破坏）了，Broken pipe。这个提示表明客户端上没有这个连接了，而 JMeter 还以为有这个链接，于是接着用这个链接来发，显然是找不到这个通道，于是就报错了。

这是一个典型的压力工具这边的问题。

而服务端，只收到了一条请求。

```
receive request: /124.64.16.240 1234aeqwfadf
```

为什么会报这个错呢？因为我们代码是短链接的，服务端处理完之后，就把这个链接给断掉了。


这里是压力机上的抓包信息：

 复制代码

```
1 // 从这里开始，上面已经看到了有 Fin (结束) 包了，后面还在发 Push (发送数据) 包。显然是通不了
2 11:58:07.042915 IP localhost.57677 > 60.205.107.9.m-oap: Flags [P.], seq 34:67
3 11:58:07.046075 IP localhost.57677 > 60.205.107.9.m-oap: Flags [FP.], seq 67:34
4 11:58:07.076393 IP 60.205.107.9.m-oap > localhost.57677: Flags [R], seq 398676:
```

```
5 11:58:07.079156 IP 60.205.107.9.m-oap > localhost.57677: Flags [R], seq 398676:
```

服务端的抓包信息：

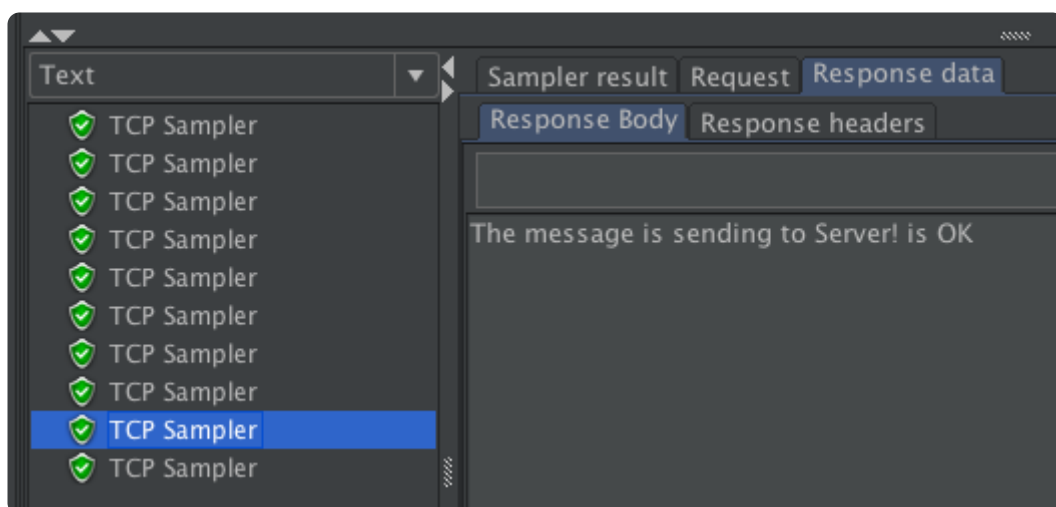
 复制代码

```
1 // 服务端也是没有办法，只能在看到了 Push 包之后，给回了个 Reset 包。
2 11:58:07.047001 IP 124.64.16.240.bones > 7dgroup1.enc-eps-mc-sec: Flags [P.], :
3 11:58:07.047077 IP 7dgroup1.enc-eps-mc-sec > 124.64.16.240.bones: Flags [R], s
4 11:58:07.054757 IP 124.64.16.240.bones > 7dgroup1.enc-eps-mc-sec: Flags [FP.],
5 11:58:07.054844 IP 7dgroup1.enc-eps-mc-sec > 124.64.16.240.bones: Flags [R], s
```

这是为什么呢？因为在 JMeter 中，默认是复用 TCP 连接的，但是在我们这个示例中，服务端并没有保存这个连接。所以，我们应该在脚本中，把下图中的 Re-use connection 给去掉。



这时再回放脚本，你就会发现 10 次迭代全都对了。如下图所示：



但是，这里还有一个知识点，希望你注意。短连接的时候，必然会产生更多的 TCP 连接的创建和销毁，对性能来说，这会让系统变得缓慢。

所以你可以看到上面 10 条迭代全都对了的同时，响应时间也增加了。

可能会有人问，那这怎么办呢？长短连接的选择取决于业务的需要，如果必须用短链接，那可能就需要更多的 CPU 来支撑；要是长连接，就需要更多的内存来支撑（用以保存 TCP 连接）。

根据业务需要，我们选择一个合适的就好。

TCP 连接超时

这个问题，应该说非常常见，我们这里只做问题的现象说明和解决，不做原理的探讨。原理的部分，我会在监控和分析部分加一说明。

下面这个错误，属于典型的主机连不上。

 复制代码

```
1 java.net.ConnectException: Operation timed out (Connection timed out)
2   at java.net.PlainSocketImpl.socketConnect(Native Method) ~[?:1.8.0_111]
3   at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:33)
4   at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:188)
5   at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)
6   at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392) ~[?:1.8.0_111]
7   at java.net.Socket.connect(Socket.java:589) ~[?:1.8.0_111]
8   at org.apache.jmeter.protocol.tcp.sampler.TCPSampler.getSocket(TCPSampler.java:111)
9   at org.apache.jmeter.protocol.tcp.sampler.TCPSampler.sample(TCPSampler.java:111)
10  at org.apache.jmeter.threads.JMeterThread.doSampling(JMeterThread.java:622)
11  at org.apache.jmeter.threads.JMeterThread.executeSamplePackage(JMeterThread.java:447)
12  at org.apache.jmeter.threads.JMeterThread.processSampler(JMeterThread.java:447)
13  at org.apache.jmeter.threads.JMeterThread.run(JMeterThread.java:253) [Apache:1.8.0_111]
14  at java.lang.Thread.run(Thread.java:745) [?:1.8.0_111]
```

time out 是个如果你理解了逻辑，就觉得很简单，如果没理解逻辑，就觉得非常复杂的问题。

要想解决这个问题，就要先确定服务端是可以正常连通的。

如果不能正常连通，那么通常都是 IP 不正确、端口不正确、防火墙阻止之类的问题。解决了网络连通性的问题，就可以解决 connection timed out 的问题。

编写 LoadRunner 脚本

针对上面这个示例，如果你要想编写一个 LoadRunner 的示例脚本，也是简单到不行。

首先创建一个空的 winsock 脚本，复制下面代码到 action 里面。

 复制代码

```
1 // 创建 socket1
2 lrs_create_socket("socket1", "TCP", "RemoteHost=60.205.10.9:5567", LrsLastArg)
3 // 走 socket1, 发送 buf1 中定义的数据
4 lrs_send ("socket1", "buf1", LrsLastArg );
5 // 走 socket1, 接收数据保存在 buf2 中
6 lrs_receive("socket1", "buf2", LrsLastArg);
7 // 关掉 socket1
8 lrs_close_socket("socket1");
```

从上面的信息就可以看到，socket1 这个标识是我们操作的基础。如果你在一个脚本中想处理两个 socket，也是可以的，只要控制好你的标识不会乱就行。

接着再将下面的内容复制到 data.ws 里面。

 复制代码

```
1 send buf1 5
2     "12345"
3
4 recv buf2 10
```

你可能会问，这个 recv 怎么不写返回的值是什么？

当你手写 socket 脚本的时候，都还没有运行，你怎么知道返回值是什么呢？所以这里，可以不用写。

而 recv 后面的 10 是指接收 10 个字节。如果多了怎么办？截掉？！不会的，LoadRunner 还是会把所有信息全部接收并保存下来，除非你提前定义了截取字符长度的函数。

最后看下我们回放的结果：

```
1 Action.c(6): lrs_create_socket(socket1, TCP, ...)
2 Action.c(7): lrs_send(socket1, buf1)
3 Action.c(8): lrs_receive(socket1, buf2)
4 Action.c(8): Mismatch in buffer's length (expected 10 bytes, 11 bytes actually
5 =====EXPECTED BUFFER=====
6 =====
7 =====RECEIVED BUFFER=====
8 "12345 is OK"
9 =====
10 Action.c(8): callRecv:11 bytes were received
11 Action.c(9): lrs_close_socket(socket1)
```

看，脚本正常执行了，只是报了一个 Mismatch，这是因为我们定义了 buf2 是 10 字节，而我们实际上接收了 11 字节，所以这里给出了 Mismatch。

到此，一个 LoadRunner 的手工 TCP 脚本就完成了。后面我们就可以根据需要，增强脚本了，加个参数化、关联、检查点等等。

总结

其实这篇文章只想告诉你一件事情，手工编写脚本，从基础上说，是非常简单的，只是有三点需要特别强调：

1. 涉及到业务规则和逻辑判断之后，编写脚本就复杂了起来。但是了解业务规则是做脚本的前提条件，也是性能测试工程师的第一步。
2. 编写脚本的时候，要知道后端的逻辑。这里的意思不是说，你一开始写脚本的时候，就要去读后端的代码，而是说你在遇到问题的时候，要分析整个链路上每个环节使用到了什么技术，以便快速地分析判断。
3. 写脚本是以**最简为最佳**，用不着故意复杂。

脚本的细节功能有很多，而现在我们可以看到市场上的书籍也好，文档也好，基本上是在教人如何用工具，很少会从前到后地说明一个数据从哪发到哪，谁来处理这样的逻辑。

希望学习性能测试工具的你，不仅知其然，更知其所以然。

思考题

学习完今天的内容，你不妨思考一下，HTTP 的 GET 和 POST 请求，在后端处理中有什么不同？断言的作用是什么？如何使用断言呢？

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

点击查看 

打卡学习，成为真正的性能测试高手




PC端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | 性能测试工具：如何录制脚本？

下一篇 09 |  关联和断言：一动一静，核心都是在取数据

精选留言 (12)

 写留言



善行通

2020-01-01

感谢高老师无私分享，在刚开始学习性能测试的时候，一直不理解做脚本为什么要这样做，也报名参加过培训机构，也许培训机构的老师都不会，或者自己没有写过后端代码，更不会讲解后端怎么实现，还有调用关系，或者根本不想让学员知道，担心教会徒弟饿死师父。...

展开 ▾

作者回复: 理解的很深刻哦。

◀ ▶



👍 3



@zzw

2020-01-02

前面几个同学回答的蛮好，我在此补充下幂等性相关的区别吧。

什么是幂等性

一次和多次请求某一个资源应该具有同样的副作用（对资源变更带来连锁反应或影响）： $f(x) = f(f(x))$ 。...

展开 ▾



👍 2



律飛

2020-01-01

1.HTTP 的 GET 和 POST 请求，在后端处理中有什么不同？

由于spring的RequestParam注解接收的参数是来自于requestHeader中，即请求头，也就是在url中，格式为xxx?username=123&password=456，而RequestBody注解接收的参数则是来自于requestBody中，即请求体中。

因此如果为get请求时，后台接收参数的注解应该为RequestParam，如果为post请求时...

展开 ▾

作者回复: 你也说的很好。

◀ ▶



👍 1



许童童

2020-01-10

如果我们配置了 100 线程，这里配置为 10 秒，那么就是 $100 / (10s * 1000ms) = 1$ 线程 / 10 ms 这个公式没看懂，老师可以讲一下吗？谢谢

展开 ▾

作者回复: 这个???

10秒启动100个线程。平均下来不就是100ms启动一个线程吗？

◀ ▶





Geek_5860ac

2020-01-10

“如果我们配置了 100 线程，这里配置为 10 秒，那么就是 $100/(10s*1000ms)=1$ 线程 / 10ms” 这个地方不明白，怎么算都是1线程/ 100ms啊，请老师指正

展开 ∨

作者回复: 少了一个零。哈，马上让编辑小美女加上。



小老鼠

2020-01-08

断言会不会影响压测试机发虚拟用户数的速度？

展开 ∨



吴小喵

2020-01-07

老师，tcp抓包用的什么工具呀

展开 ∨

作者回复: linux下面用Tcpdump就可以。windows用wireshark。



悦霖

2020-01-03

抓包信息看不懂怎么办😓

展开 ∨

作者回复: 说明需要补充网络知识。



月亮和六便士

2020-01-02

老师，我还有问题想问：前面的文章中提到过，1. 什么叫压力补偿，压力补偿的作用是什么？ 2. 还有为什么要动态扩展？ 比如内存不够了，我们不应该找到谁占用了内存吗？ 3.

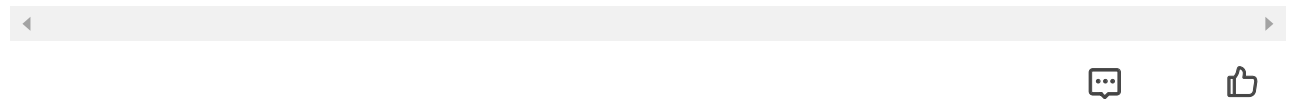
每次测试前需要清理缓存吗？比如我跑一轮脚本 就需要把redis 缓存清一下吗？

展开 ∨

作者回复: 1. 就是在场景执行的过程中，发现场景产生的压力和生产上比例不一致，或者有中间需要的增加的第三方压力。就需要在场景执行过程中再增加新的线程或者压力机。

2. 动态扩展是验证线上的能力。如果业务流量突增了。就需要动态扩展哇。

3. 看机制。如果不是预热类型的。可以在每次跑之前清一下。



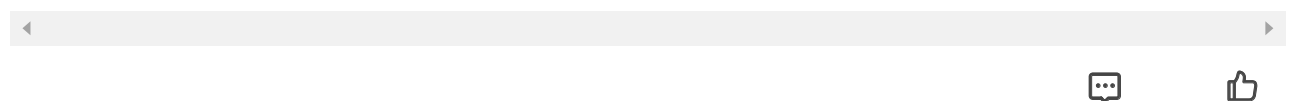
月亮和六便士

2020-01-02

老师 ” 要想解决这个问题，就要先确定服务端是可以正常连通的 ” 确定了服务端可以联通，然后怎么解决timeout？

展开 ∨

作者回复: 拆时间看哪里长。



新思维

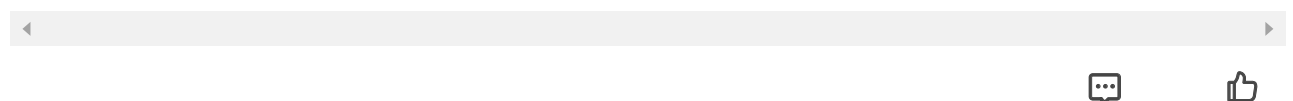
2020-01-02

get请求，一般后端服务只是通过传过来的参数查询数据库，返回结果；post请求，一般后端服务会将请求所包含的内容更新到数据库，返回更新结果。

断言判断后端服务返回的请求是否为所期望的请求结果。涉及到业务逻辑的断言需要对响应内容进行检查，包括关键字检查、或者数据处理逻辑结果检查等。

展开 ∨

作者回复: 理解的非常对。



土耳其小土豆

2020-01-01

post的请求体放到body里、get的请求的消息体放到请求头里面、所以post请求相对较安全。

作者回复: 话说的没毛病。不过在性能的专栏中就得说性能的话哦。

