Locust 性能测试报告

靳嘉豪

第 1 章　Locusts 压测报告

1.1　压测环境

本次压测服务端和压测段均部署在同一机器上，机器配置为 2.6 GHz 六核 Intel Core i7，16 GB 2667 MHz DDR4。操作系统为 macOS Big Sur 11.3.1。接口由 Spring Boot 编写，部署在 Tomcat 9.0.50 上。压测端用 locust 工具，版本为 1.6.0。

1.2　工具原始数据

本次压测工具选取 httprunner 对 locust 封装的 locust 压测工具。分别压测 200、500 和 1000 并发量下的接口性能，主要关注接口的 RPS（Request Per Second）、错误率和接口的响应时间。原始的压测结果如表 1.1所示，表中的取值都是在压测各项指标平稳后的取值。

表 1.1　原始测试报告数据

并发量	RPS	平均响应时间 (ms)	95% 响应时间 (ms)	错误率
200	649.7	247	460	0%
500	544.2	724	1200	0%
1000	504.6	1603	2800	0%

在压测过程中，响应时间和并发量之间的关系如图 1.1所示（完整的压测结果数据在文件中）。

1.3　原始数据分析

先分析表 1.1，可知随着并发量的上升，接口的 RPS 逐渐下降，响应时间逐渐增大。再分析图 1.1，根据图中曲线的增减关系，可以将整个压测过程分为 4 个阶段：

- **阶段.1** 并发数增加至并发数稳定后的一小段时间，响应时间迅速升高。原因是该阶段每个新增并发都会新建 tcp 连接，创建过程十分耗时。通过 netstat 调用也可以看到，此时系统中只有 ESTABLISHED 状态的连接，且不断增加。

图 1.1　并发量与响应时间关系

- **阶段.2** 并发数稳定，响应时间迅速下降。该阶段不再有新的建立连接请求，随着之前阻塞的连接请求被处理，单位时间内建立连接操作减少，对传输请求的响应频率升高，所以整体响应时间快速下降。

- **阶段.3** 并发数稳定，响应时间稳定（几乎不变）。观察接口的所有连接状态，发现该阶段只有 1 个 LISTEN 状态连接和 2 倍并发量的 ESTABLISHED 连接，意味着此时所有并发经过前期的创建请求，都有了各自的连接并开始 http 通信，因为数据传输时间基本一致，所以该阶段曲线稳定。

- **阶段.4** 并发数稳定，响应时间开始较明显浮动。此时观察端口所有连接的状态，发现开始逐渐出现 TIME_WAIT、CLOSE_WAIT、FIN_WAIT 等状态的连接，同时连接总数也开始不再是 1+2* 并发数。这表明开始有并发协程请求完毕开始断开请求，并在下次获取处理器后重新建立新的请求。由于又出现了十分耗时的创建和销毁连接操作，所以整体响应时间会开始波动，并且波动段的最小值通常大于等于平稳段的值。

1.4 总结

在合理并发量下，接口正确性上并不会出现问题，但响应时间偏长，需要进一步分析原因，进行优化。

压测结果在第四阶段更符合大多数场景，并发用户随机地创建连接、传输请求以及销毁连接，所以通常对性能指标的取值也在这个阶段统计。但不排除一些特殊场景，比如系统刚刚上线、用户扎堆新建连接，这些也会导致其他阶段的出现，所以需要看具体业务场景，选取要分析的阶段，再对关心的指标值进行统计。

第 2 章　性能瓶颈分析

首先分析系统的哪些资源造成了接口的性能瓶颈，再分析整个压测过程的哪个环节导致了这些相关资源的瓶颈，最后分析造成这一瓶颈的原因。为了简单起见，不再用 httprunner 的 pytest 文件作为 locust 中的模拟用户行为，而是直接用 python 的 requests 库发送请求作为用户行为。

2.1　系统资源分析

在压测数据趋于稳定后，通过系统的资源管理器查看各进程的资源消耗，发现压测端进程消耗 CPU 达 99%，而服务端 CPU 只有 27%，同时内存仍有超过 50% 的剩余。另一方面，接口流量也仅 317.44 kb/s。基于上述指标，确定 CPU 为造成瓶颈的主要资源。

2.2　压测环节时间消耗分析

有很多原因能够导致 CPU 对整体性能的限制，比如服务端 I/O 阻塞、请求过多频繁发生线程切换等等。为了明确原因，将压测过程分为三段：服务端、通信信道和压测端，依次分析他们的时间消耗。

首先是服务端，由于 tomcat 本身在处理请求时，会对每个已建立的连接单独创建一个线程，并放入线程池中运行，线程池默认保持 200 个核心线程。所以，在大量并发请求时，并不会存在 I/O 阻塞问题。再考虑线程池线程数量造成的影响，通过对 tomcat 参数的设置，在相同并发量（1000）的条件下，分别使线程池保持不同数量的核心线程数，观察响应时间的变化，结果如表 2.1所示。

表 2.1　核心线程数量与响应时间的关系

线程数量	平均响应时间 (ms)	95% 响应时间 (ms)
1	422	510
6	416	508
200	422	510

可知并没有明显变化，再统计每个请求的运行时间为 0.07-0.1 ms，这和压测端统计到的响应时间相差甚远，可知服务端的逻辑并不是性能瓶颈所在。另一个能够证明

服务端并不是造成性能平静的主要因素的实验是构造两个服务端，分别监听不同端口，压测时随机选取端口发送请求，实验结果见表 2.2。可知相同并发量下，两个服务端并没有比单个服务端的延迟有明显提升。

表 2.2　服务端数量和并发数量的对比结果

并发量	单服务端		双服务端	
	平均响应时间 (ms)	95% 响应时间 (ms)	平均响应时间 (ms)	95% 响应时间 (ms)
500	206	280	234	310
1000	422	510	496	600
1500	677	850	728	850

再对通信信道进行分析，考虑到可能本机间通信延时较长或者高并发下内核对多个端口的通信处理不及时导致的，甚至也有可能高并发下，本机传输也会触发 TCP 的拥塞控制、超时重传等机制。所以，深入 requests 库的源码，截取压测端的 socket 的 write 和 recive 代码块，统计该代码块的运行时间为 7-25 ms。可以证明，请求传输过程和服务端处理过程均不是时间消耗的主要环节。

现在基本确定时压测端本身造成的时间消耗，阅读 locust 源码可知，其底层记忆 gevent 的事件循环，通过协程实现模拟用户。该过程并发量越大，事件队列可能阻塞的事件越多，导致处理事件统计延迟的时间要比真正接收到事件的时间要晚。如果该猜想为真，则相同并发量下，采用多核模拟应该会比单核要快。根据这一思路，在 1500 并发下，单核 locust 统计的响应时间为 1000 ms，通过 docker 搭建 3 核模拟并发的响应时间为 680 ms，可以看出明显提升。

另一方面，locust 在计算响应时间时是以用户行为开始调用到调用结束的时间为依据的，但通过在客户行为函数中的 requests 请求的中间环节插入代码，发现执行过程是有比较多的时间协程处于等待的状态，这部分时间占用较大，且也会统计到响应时间当中。

2.3　结论

本次压测并未达到借口性能的极限，由于压测端并发能力不足，得到的压测数据项相较于接口的真实数据偏弱，可作为接口性能的下限。

第 3 章　接口压测优化方案

基于前两章的分析，可知整个压测过程的结果是不准确的，为了尽可能逼近真实的接口性能瓶颈，对压测端进行一些优化。

3.1　提升 locust 并发能力

上章已经证明 locust 的模拟并发能力不足，是导致高响应时间的从而影响压测结果准确性的主要因素，所以最简单的方法就是提升其并发能力。locust 本身提供了基于分布式的压测功能，在本机上运行分布式本质上是利用多核并行。由于服务端需要占用至少一个核，所以设置 locust 的分布式参数为 1 个 master 和 4 个 worker，在并发量为 500 的时候，接口 RPS 为 1900 以上，平均响应时间为 193 ms，95% 响应时间为 260 ms。

3.2　wrk 压测结果

通过另一个高性能压测工具 wrk（版本 4.1.0-8-ga211dd5），能够进一步逼近真实的接口性能极限。对比两个工具，在只进行 get 请求而不对相应内容进行判断的情况下，wrk 的 rps 为 4690，平均响应时间为 5.59 ms，99% 响应时间为 56.86 ms，而并发的 locust 只有 1900-2200 的 rps，平均响应时间为 180 ms，99% 响应时间为 250 ms。可见 wrk 的并发模拟能力更强，更容易逼近真实的借口性能极限。