Deepseek企业级Agent项目开发实战

Part 9. Ollama 服务接口压力测试

对于企业级应用来说，尤其是后台服务，考虑的因素会非常多。比如大模型问答的响应速度，系统服务的稳定性，业务请求的错误率，资源的利用率等等多个方面。不同应用场景，考虑的因素也会有所不同。像我们正在做的智能客服问答功能，更关注响应速度和稳定性，这就导致高吞吐量和高并发能力比较重要，直接影响服务承载能力和效率，往往是优化的重点。吞吐量通常指系统在单位时间内处理的请求数量，而并发量则是系统同时处理的请求数。

因为企业需要处理大量用户或设备的请求，尤其是在高峰时段。如果服务吞吐量低，可能导致延迟增加，用户体验下降，甚至服务崩溃。如果并发量不足，用户可能会遇到等待或超时；吞吐量低的话，处理速度慢，整体效率低下。

我们基于`Ollama`模型服务启动的`REST API`接口，每秒生成的`Token`数量可以被视为系统的吞吐量，因此我们需要一些方法，来根据实际的业务需求来评估当前的硬件资源是否满足需求，或者应该如何去采购硬件资源。

我们测试`Ollama`模型服务的吞吐量和并发量，需要核心关注的是以下几点：

1. 使用`REST API`接口进行测试，可以尝试使用`/api/generate`或者`/api/chat`，真实模拟用户在实际使用中的请求模式，帮助评估系统在真实场景下的表现。
2. `Ollama`原生的`REST API`接口支持多个控制`Ollama`行为的参数，可以更灵活的控制测试流程，其中：
    - `num_predict`参数来控制生成的token数量
    - `keep_alive`设置为0，使用完模型后立即卸载
    - `temperature`参数来控制生成文本的多样性，很多情况下，希望生成的文本尽可能保持一致，会将其设置为0，
3. 根据`Ollama`的`REST API`接口返回响应体中的`eval_count`和`eval_duration`来计算每秒生成的`Token`数量，即吞吐量，而不是用`resquest`发起和接收到响应的时间差值来计算，将模型服务和网络延迟解耦，更准确的评估模型服务的吞吐量。

单次调用的伪代码如下：

```
            # 调用 ollama 的 generate 接口
            async with session.post(
                f"{self.url}/api/generate",
                json={
                    "model": self.model,
                    "prompt": prompt,  # 使用随机选择的问题
                    "stream": False,
                    # "keep_alive":0,    # 使用完模型后立即卸载
                    "options": {
                        "temperature": 0.7,
                        "num_predict": 300,    # 限制生成token数量，以尽可能保证单个请求的生成时间一致
                    }
                }
            ) as response:
                result = await response.json()
                # 从响应中获取性能指标
                eval_count = result.get("eval_count", 0)  # 生成的token数
```

```
                eval_duration = result.get("eval_duration", 0)  # 生成时间(纳秒)
                total_duration = result.get("total_duration", 0)  # 总时间(纳秒)

                # 计算 tokens/second
                tokens_per_second = (eval_count / eval_duration * 1e9) if
 eval_duration > 0 else 0
```

在`01_ollama_deepseek_r1.ipynb`中，我们介绍了使用`SystemD`的启动和配置`Ollama`服务的方法，这种方式是通过创建`systemd`服务单元文件（即`ollama.service`），将`ollama serve`配置为系统服务，从而可以使用如`systemctl start ollama`等命令来启动和停止`Ollama`服务。比较适用于生产环境、需要长期稳定运行的服务以及自动化管理的场景。同时也更适合快速入门。

除此以外，`Ollama`还有另外一种启动`REST API`的方法，即直接在命令行中运行`ollama serve`命令，启动服务进程。比较适合本地开发环境，临时测试或调试，同时拥有更多的控制权限。

因此，在测试前需要先关闭通过`systemd`启动的`Ollama`服务，操作方法如下：

1. 先通过`systemctl stop ollama`停止`Ollama`服务，否则`systemd`会监听`ollama.service`文件并不断自动拉起服务；
2. 接着通过`lsof -i:11434`命令查看`ollama serve`进程的PID；
3. 然后通过`kill -9 <PID>`命令杀掉进程；
4. 再次查看就会发现进程已经被杀掉；

```
(base) root@4U:~# systemctl stop ollama.service
(base) root@4U:~# systemctl status ollama.service
● ollama.service - Ollama Service
     Loaded: loaded (/etc/systemd/system/ollama.service; enabled; vendor preset: enabled)
    Drop-In: /etc/systemd/system/ollama.service.d
             └─override.conf
     Active: inactive (dead) since Thu 2025-02-20 17:55:15 CST; 1min 22s ago
    Process: 367423 ExecStart=/usr/local/bin/ollama serve (code=exited, status=0/SUCCESS)
   Main PID: 367423 (code=exited, status=0/SUCCESS)
        CPU: 718ms

2月  20 17:50:49 4U ollama[367423]: time=2025-02-20T17:50:49.365+08:00 level=INFO source=im
2月  20 17:50:49 4U ollama[367423]: time=2025-02-20T17:50:49.365+08:00 level=INFO source=ro
2月  20 17:50:49 4U ollama[367423]: time=2025-02-20T17:50:49.366+08:00 level=INFO source=gp
2月  20 17:50:49 4U ollama[367423]: time=2025-02-20T17:50:49.982+08:00 level=INFO source=ty
2月  20 17:50:49 4U ollama[367423]: time=2025-02-20T17:50:49.982+08:00 level=INFO source=ty
(base) root@4U:~# lsof -i :11434
COMMAND     PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
ollama  366597 ollama    3u  IPv6 692933      0t0  TCP *:11434 (LISTEN)
(base) root@4U:~# kill -9 366597
(base) root@4U:~# lsof -i :11434
(base) root@4U:~#
```

使用`ollama serve`命令启动`Ollama`服务方法如下：

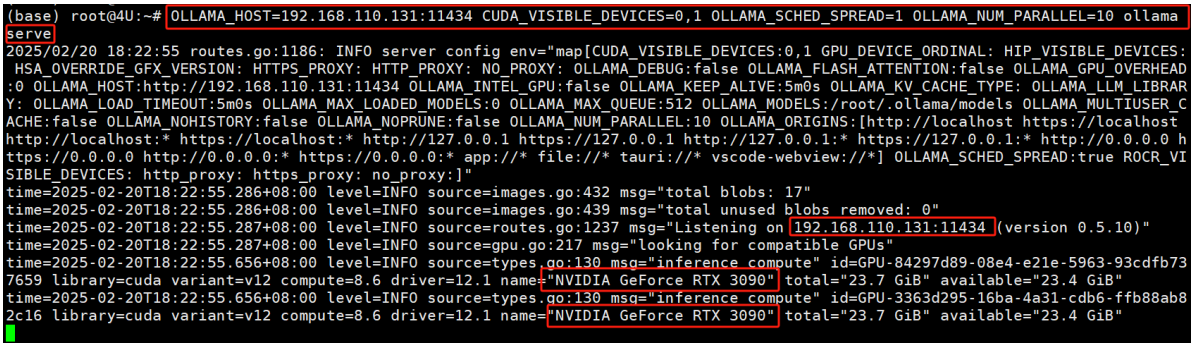
```
(base) root@4U:~# ollama serve
2025/02/20 17:58:00 routes.go:1186: INFO server config env="map[CUDA_VISIBLE_DEVICES: GPU_DEVICE_ORDINAL: HIP_VISIBLE_D
EVICES: HSA_OVERRIDE_GFX_VERSION: HTTPS_PROXY: HTTP_PROXY: NO_PROXY: OLLAMA_DEBUG:false OLLAMA_FLASH_ATTENTION:false OL
LAMA_GPU_OVERHEAD:0 OLLAMA_HOST:http://127.0.0.1:11434 OLLAMA_INTEL_GPU:false OLLAMA_KEEP_ALIVE:5m0s OLLAMA_KV_CACHE_TY
PE: OLLAMA_LLM_LIBRARY: OLLAMA_LOAD_TIMEOUT:5m0s OLLAMA_MAX_LOADED_MODELS:0 OLLAMA_MAX_QUEUE:512 OLLAMA_MODELS:/root/.o
llama/models OLLAMA_MULTIUSER_CACHE:false OLLAMA_NOHISTORY:false OLLAMA_NOPRUNE:false OLLAMA_NUM_PARALLEL:0 OLLAMA_ORIG
INS:[http://localhost https://localhost http://localhost:* https://localhost:* http://127.0.0.1 https://127.0.0.1 http:
//127.0.0.1:* https://127.0.0.1:* http://0.0.0.0 https://0.0.0.0 http://0.0.0.0:* https://0.0.0.0:* app://* file://* ta
uri://* vscode-webview://*] OLLAMA_SCHED_SPREAD:false ROCR_VISIBLE_DEVICES: http_proxy: https_proxy: no_proxy:]"
time=2025-02-20T17:58:00.996+08:00 level=INFO source=images.go:432 msg="total blobs: 17"
time=2025-02-20T17:58:00.997+08:00 level=INFO source=images.go:439 msg="total unused blobs removed: 0"
time=2025-02-20T17:58:00.997+08:00 level=INFO source=routes.go:1237 msg="Listening on 127.0.0.1:11434 (version 0.5.10)"
time=2025-02-20T17:58:00.998+08:00 level=INFO source=gpu.go:217 msg="looking for compatible GPUs"
time=2025-02-20T17:58:01.617+08:00 level=INFO source=types.go:130 msg="inference compute" id=GPU-84297d89-08e4-e21e-596
3-93cdfb737659 library=cuda variant=v12 compute=8.6 driver=12.1 name="NVIDIA GeForce RTX 3090" total="23.7 GiB" availab
le="23.4 GiB"
time=2025-02-20T17:58:01.617+08:00 level=INFO source=types.go:130 msg="inference compute" id=GPU-3363d295-16ba-4a31-cdb
6-ffb88ab82c16 library=cuda variant=v12 compute=8.6 driver=12.1 name="NVIDIA GeForce RTX 3090" total="23.7 GiB" availab
le="23.4 GiB"
time=2025-02-20T17:58:01.617+08:00 level=INFO source=types.go:130 msg="inference compute" id=GPU-2f30cfd0-af19-64b3-6b3
7-eee2eb0b531d library=cuda variant=v12 compute=8.6 driver=12.1 name="NVIDIA GeForce RTX 3090" total="23.7 GiB" availab
le="23.4 GiB"
time=2025-02-20T17:58:01.617+08:00 level=INFO source=types.go:130 msg="inference compute" id=GPU-4a3c9301-84de-7334-ca5
0-996f2b54dfa8 library=cuda variant=v12 compute=8.6 driver=12.1 name="NVIDIA GeForce RTX 3090" total="23.7 GiB" availab
```

这里有一些关键的参数，可以控制`Ollama`服务的启动行为：

- **OLLAMA_HOST**：设置`Ollama`服务的监听地址，默认是`127.0.0.1:11434`，如果需要指定其他地址，可以设置为`公网IP:11434`；
- **CUDA_VISIBLE_DEVICES**：设置`Ollama`服务使用的`GPU`设备，默认选择所有可用的`GPU`设备，如果需要指定其他`GPU`设备，可以设置为`1`，`2`等，用逗号分隔；
- **OLLAMA_SCHED_SPREAD**：设置`Ollama`所选择`GPU`资源是否均匀分布，默认是`true`，如果设置为`false`，则`Ollama`会优先选择性能最好的`GPU`设备；
- **OLLAMA_NUM_PARALLEL**：设置`Ollama`每个模型可以同时处理的最大并行请求数量。默认值会根据可用显（内）存自动选择`4`或`1`。
- **OLLAMA_MAX_QUEUE**：如果在已经加载一个或多个模型的同时，没有足够的可用内存来加载新的模型请求，则所有新请求将排队，直到可以加载新型号为止。随着先前的模型闲置，将卸载一个或多个，以腾出空间为新型号腾出空间。排队的请求将按顺序处理；
- **OLLAMA_MAX_LOADED_MODELS**：设置最大加载的模型数量，默认是 3 * GPU 的数量，如果超过这个数量，新的请求会被拒绝；

大家可以根据自己的需求，灵活设置参数组合，比如：

```
    OLLAMA_HOST=192.168.110.131:11434 CUDA_VISIBLE_DEVICES=0,1
OLLAMA_SCHED_SPREAD=1 OLLAMA_NUM_PARALLEL=10 ollama serve
```



这里有两个关键点：

1. 使用`ollama serve`命令启动`Ollama`服务后，当通过`/api/generate`接口发起请求时，会按照`Ollama Serve`选择的`GPU`去加载模型；
2. 如果`CUDA_VISIBLE_DEVICES`设置了多个`GPU`设备，则`Ollama`会按照`CUDA_VISIBLE_DEVICES`设置的顺序去加载模型；若单个`GPU`设备能够加载模型，则`Ollama`会按照`CUDA_VISIBLE_DEVICES`设置的顺序去加载模型；搭配`OLLAMA_SCHED_SPREAD`参数才会去做负载均衡（均匀分布）；

大家可以直接运行在`app/test/ollama_benchmark.py`文件，需要修改的所有参数都在`main`函数中，如下所示：

```
    benchmark = OllamaBenchmark(
        url="http://192.168.110.131:11434",  # 这里替换成实际的ollama endpoint
        model="deepseek-r1:32b"              # 这里替换成实际要进行测试的模型名称
    )

    concurrency_results = await benchmark.find_max_concurrency(
    start_concurrent=2,           # 从2开始
    max_concurrent=5,             # 最多只测到5个并发
    requests_per_test=10,          # 每轮只测10个请求
    success_rate_threshold=0.95,  # 成功率要求提高到95%
    latency_threshold=5.0         # 延迟阈值降低到5秒
)
```

根据自己的实际情况修改后即可运行，运行后会生成`logs`目录下的测试结果文件，执行方法如下所示:

1. 先激活虚拟环境
2. 进入`app/test`目录
3. 运行`python ollama_benchmark.py`文件

注意：执行压力测试程序的时候一定要注意服务器的情况，如遇到瓶颈再次增加并发数量等会直接导致服务器死机!

如下是我在本地服务器上测试`DeepSeek-R1:1.5B`模型分别在 双卡负载和四卡负载下的吞吐量和并发量，测试结果如下：

并发测试结果

测试类型	并发数	成功率	总token数	平均生成时间(秒)	平均总时间(秒)	平均每秒token数	实际总耗时(秒)	系统吞吐量(tokens/s)
两张卡	2	100%	2742	3.00	4.03	91.96	23.84	115.01
两张卡	3	100%	2686	3.61	3.71	73.56	14.78	181.75
两张卡	4	100%	2665	4.48	4.59	59.75	14.16	188.19
两张卡	5	100%	2556	4.80	4.91	53.34	12.39	206.27
单请求性能测试	-	-	300	2.44	2.52	123.00	-	-
四张卡	2	100%	2526	2.64	4.16	96.77	25.38	99.51
四张卡	3	100%	2456	3.34	3.44	72.44	14.51	169.28
四张卡	4	100%	2781	4.54	4.65	62.27	14.59	190.57
四张卡	5	100%	3000	6.53	6.65	45.93	14.42	208.07
单请求性能测试	-	-	300	2.52	2.59	119.31	-	-

从测试结果能够得出的一些关键结论是：

1. 并发会导致单个请求的处理时间变长；
2. 并发因为是并行处理，虽然单个请求时间变长，但是系统整体吞吐量会得到提升；
3. 不一定用更多的卡就可以获得更高的吞吐量，需要根据实际情况去调整。

因此，大家在实际测试的时候，要尝试在不同的硬件配置和并发级别下进行测试，以找到最佳的性能平衡点。

最后，给大家总结一下`ollama serve`和`systemd`启动`ollama`服务的区别，如下所示：

Ollama 服务启动方式对比

特性	ollama serve 启动	systemd 启动
运行方式	前台运行，依赖终端	后台运行，独立于终端
服务管理	手动管理	支持启动、停止、重启、状态查看
自动恢复	不支持	支持崩溃后自动重启
开机自启动	不支持	支持
日志管理	输出到终端，无持久化	由 journald 管理，支持持久化
资源控制	无	支持 CPU、内存等资源限制
适用场景	开发、测试、临时运行	生产环境、长期运行

大家根据实际需求选择合适的启动方式，建议大家在生产环境使用`systemd`启动`ollama`服务，在本地开发环境使用`ollama serve`启动`ollama`服务。