

# 在 win10/11 下量化 ORLM 对话大模型+安装 Ollama+WebUI+AMD 显卡加速

笔者以 win11+WSL 环境为例

前言：

大语言模型当前日益火爆，不仅仅 ChatGPT 之类的超大模型，阿里 Qwen2:7B 之类的小模型也表现出了非常好的实用性，除了上述泛用的模型，近期杉数开发的 ORLM 运筹学大模型提供了一个不错的运筹学对话学习方式。

杉数即 copt 求解器母公司，提供了优秀的国产求解器，本次开发出来的 ORLM 模型规模为 8B，经过量化后可以运行在 8GB 显存的显卡上，而 AMD 显卡由于缺乏 CUDA 环境，在 AI 应用方面较为困难，但随着 Windows HIP SDK 的开发，Ollama 对部分 AMD 显卡的支持，笔者已经成功实现在 AMD Radeon 680m 集成显卡上以 Ollama API 形式运行 Qwen2:7B 模型。而 ORLM 模型并未在 Ollama 模型库提交量化模型，所以需要自行导入其在 hugging face 上提交的 safetensor 模型并自行量化。

此外，由于组件多数是源代码形式，所以使用 WSL 平台和 Windows Powershell 混合开发比较便利，关于 WSL 的安装见笔者上一篇教程，本开发需要全程科学上网。

1.首先在 Windows PowerShell 克隆下 hugging face 上的 ORLM 模型到 Win11 用户根目录，存入 model 文件夹：

```
git lfs install
```

```
git clone https://huggingface.co/CardinalOperations/ORLM-LLaMA-3-8B  
model
```

克隆好就可以放一边了。

## 2.安装 Ollama

如果是 Nvidia 显卡用户，拥有 8GB 以上的显存并支持 CUDA11.8，请直接前往 Ollama 官网下载安装包安装，无需自行配置。安装好后打开 Windows PowerShell 输入 `Ollama run Qwen2:7b`，等待程序执行并拉取模型，交互式对话出现后随便问问看一下回答和显卡占用。

AMD 用户请自行编译源代码加入 AMD 显卡支持，本人提供一个第三方编译的包和教程。

## 2.1 安装 AMD HIP SDK

网址：<https://www.amd.com/en/developer/resources/rocm-hub/hip-sdk.html>

选择 Windows 10 & 11 的 5.7.1 下载安装。

## 2.2 自编译 Roclab 动态链接库和 Library 的下载

地址：<https://github.com/likelovewant/ROCmLibs-for-gfx1103-AMD780M-APU>，在 release 中选择 v0.5.7 中对应你自己显卡的文件，下载并解压，笔者是 680m 对应 gfx1035。

## 2.3 下载安装第三方编译好的 Ollama

网址：<https://github.com/likelovewant/ollama-for-amd>

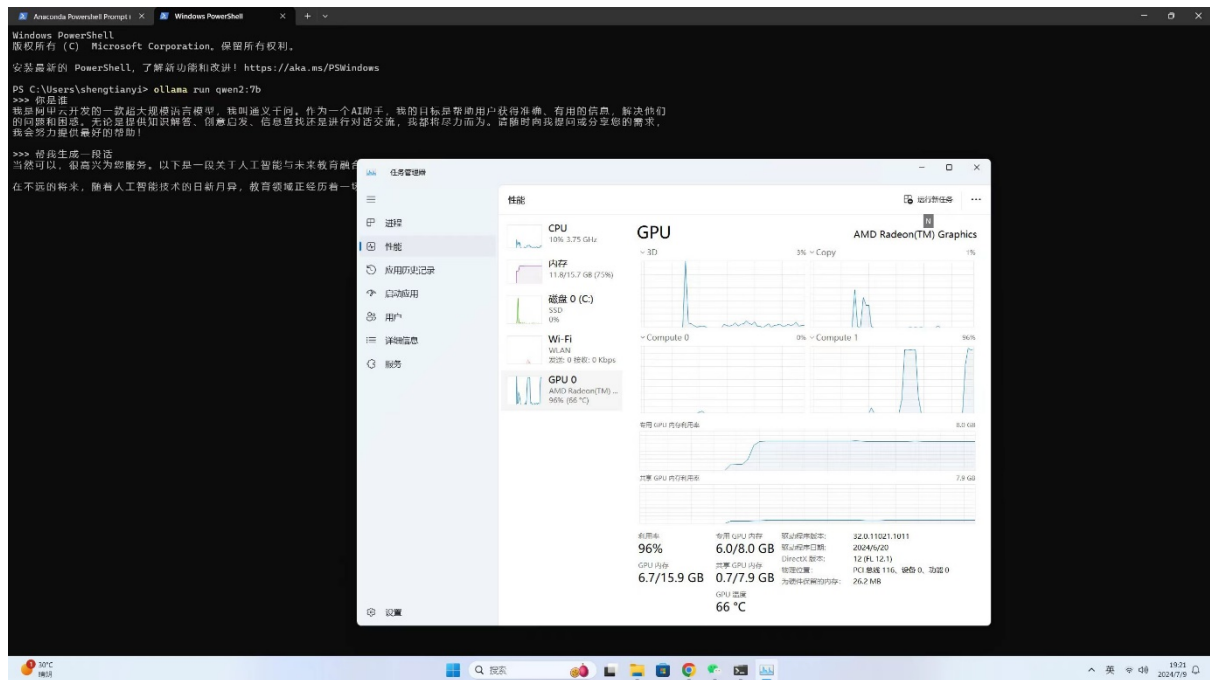
选择最新的 Release 下载 ollama-windows-amd64.7z，解压。

## 2.4 自编译 Roclab 动态链接库和 Library 的替换

这是本章重点和难点，路径容易乱，替换好了才能让显卡正确运行。

首先讲 2.2 节下载的文件夹中 rocblas.dll 复制到 library 文件夹，然后新开文件管理器窗口进入 C:\Program Files\AMD\ROCm\5.7\bin，将 2.2 节下载的 rocblas.dll 替换该目录下的文件，然后进入 C:\Program Files\AMD\ROCm\5.7\bin\rocblas，将刚才操作的 library 文件夹复制到这里来并覆盖。覆盖完毕才算是构建好特定显卡的 HIP SDK

接下来对 Ollama 文件夹构建，进入 ollama-windows-amd64 文件夹后在进入 rocm 文件夹，按照上一段的方式替换。替换完毕后将 ollama-windows-amd64 整体放入 C:\Program Files 文件夹，并在环境变量里将该文件夹加入环境变量。然后打开 Windows PowerShell 输入 Ollama run Qwen2:7b，等待程序执行并拉取模型，交互式对话出现后随便问问看一下回答和显卡占用，正常的情况应如图所示：



如图所示，回答正常，显卡的显存和计算单元有了负载，代表 Ollama 安装完毕。

## 2.5 构建 Ollama 的 WebUI

构建 WebUI 可以像使用 ChatGPT 一样对话和记录，还可以方便的条件模型的参数。首先下载 node.js 语言的解释器并安装。

在 Windows PowerShell 里使用源代码构建 Open-WebUI:

```
git clone https://github.com/open-webui/open-webui.git
```

```
cd open-webui
```

```
copy .env.example .env
```

```
npm install
```

```
npm run build
```

```
cd .\backend
```

# Optional: To install using Conda as your development environment, follow these instructions:

# Create and activate a Conda environment

```
conda create --name open-webui-env python=3.11
```

```
conda activate open-webui-env
```

```
pip install -r requirements.txt -U
```

此时 WebUI 就安装好了，在另一个 Windows PowerShell 中输入 `Ollama serve`，打开 Ollama 的 API，回到前面构建 WebUI 的窗口，输入 `start_windows.bat`，等待加载即可自动弹出网页，进入界面后就可以自行选择模型并对话了，和 Chat GPT 网页版无异。

笔者另行写了一个脚本，可以自动化开启以上服务：首先在刚才构建 WebUI 的 `backend` 文件夹下新建一个 `.bat` 文件，内容：

```
@echo off
```

```
SETLOCAL ENABLEDELAYEDEXPANSION
```

```
:: 启动 ollama serve 并等待其准备就绪
```

```
echo Starting ollama serve...
```

```
start "Ollama Serve" cmd /k "ollama serve"
```

```
:: 等待 5 秒以确保 ollama 已准备就绪
```

```
echo Waiting 5 seconds for ollama to be ready...
```

```
timeout /t 5 > nul
```

```
:: 获取当前脚本的目录
```

```
SET "SCRIPT_DIR=%~dp0"
```

```
cd /d "%SCRIPT_DIR%" || exit /b
```

```
SET "KEY_FILE=.webui_secret_key"
```

```
IF "%PORT%"==" " SET PORT=8080
```

```
IF "%HOST%"==" " SET HOST=0.0.0.0
```

```
SET "WEBUI_SECRET_KEY=%WEBUI_SECRET_KEY%"
```

```
SET "WEBUI_JWT_SECRET_KEY=%WEBUI_JWT_SECRET_KEY%"
```

```

:: 检查 WEBUI_SECRET_KEY 和 WEBUI_JWT_SECRET_KEY 是否未设置

IF "%WEBUI_SECRET_KEY%%WEBUI_JWT_SECRET_KEY%" == " " (

    echo Loading WEBUI_SECRET_KEY from file, not provided as an
    environment variable.

    IF NOT EXIST "%KEY_FILE%" (

        echo Generating WEBUI_SECRET_KEY

        :: 生成随机值作为 WEBUI_SECRET_KEY，如果用户没有提供的话

        SET /p WEBUI_SECRET_KEY=<nul

        FOR /L %%i IN (1,1,12) DO SET /p
        WEBUI_SECRET_KEY=<!random!>>%KEY_FILE%

        echo WEBUI_SECRET_KEY generated

    )

    echo Loading WEBUI_SECRET_KEY from %KEY_FILE%

    SET /p WEBUI_SECRET_KEY=<%KEY_FILE%

)

:: 启动 uvicorn

start "Uvicorn Server" cmd /k "uvicorn main:app --host %HOST% --
port %PORT% --forwarded-allow-ips '*'

:: 再次等待一段时间以确保服务器启动

timeout /t 10 > nul

:: 在默认浏览器中打开 localhost:8080

echo Launching browser to open http://localhost:8080

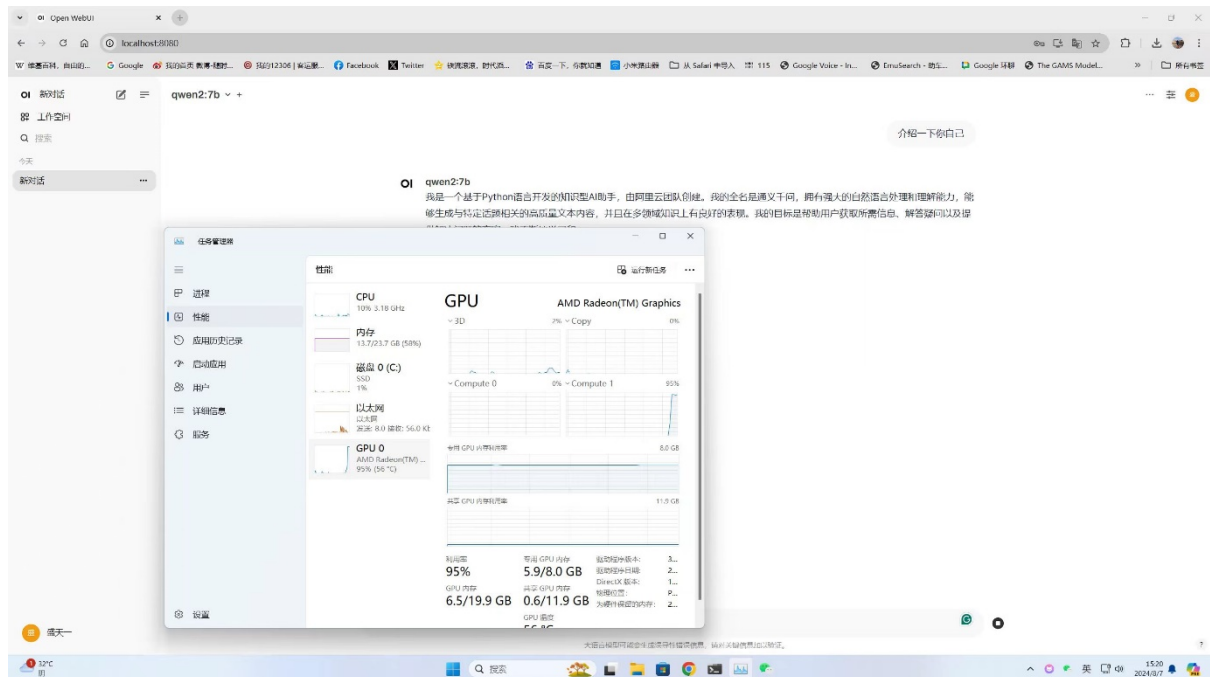
start "" "http://localhost:8080"

```

echo All processes have been started. You can close this window.

pause

保存文件并再次确实是 .bat 文件后双击该文件，即能自动打开本地的 Ollama API 和 WebUI，将其快捷方式放在任意地方即可。安装好的图如下：



### 3. 杉数 ORLM 大模型的量化与安装

#### 3.1 llama.cpp 的构建

此步骤需要在 WSL 中使用 git 下载，先打开构建必备组件：

```
sudo apt-get update
```

```
sudo apt-get install libomp-dev
```

此时关注一下 WSL 目录是否在 Windows 的用户根目录，如果是的话继续：

```
git clone https://github.com/ggerganov/llama.cpp.git
```

```
cd llama.cpp
```

```
make
```

此时将自动构建 llama.cpp

## 3.2 转换成 fp16 的 gguf 模型

先安装构建启用一个虚拟环境防止和自己的 conda 环境冲突，并安装必备的组件：

```
python3 -m venv ./venv
```

```
source .venv/bin/activate
```

```
pip install -r requirements.txt
```

一切顺利此时就可以构建好必备的组建了。

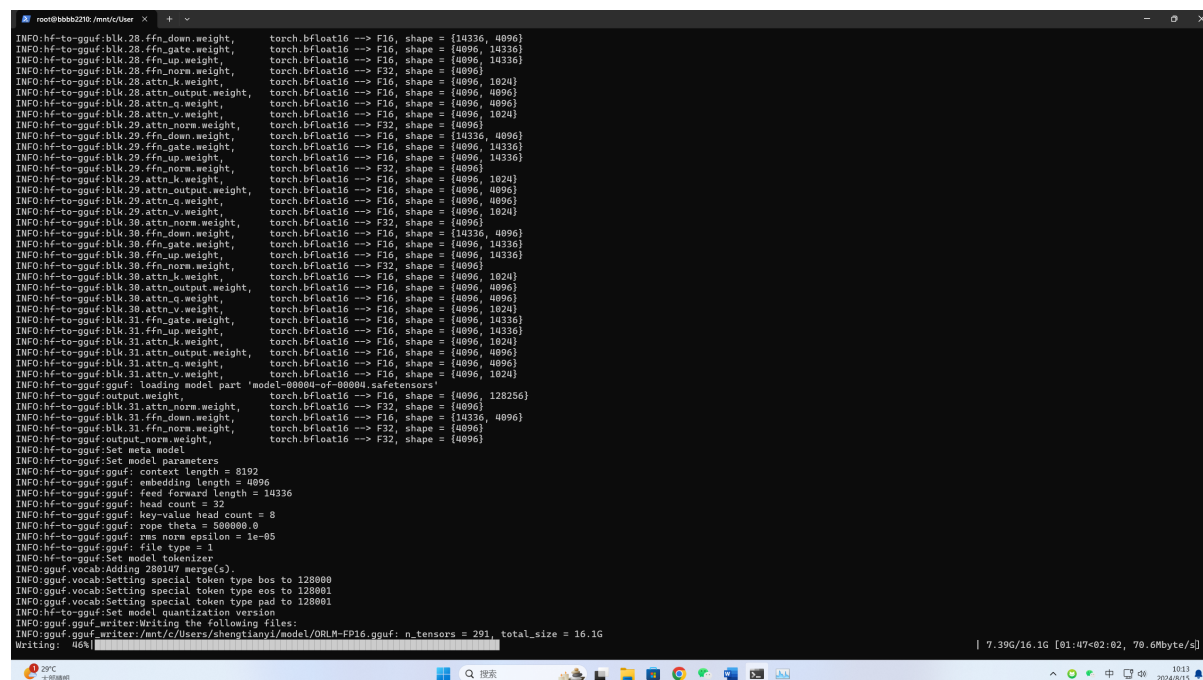
接下来找到刚才下载的 ORLM 模型，复制其路径，笔者的路径是：

/mnt/c/Users/users/model

回到 llama.cpp 文件夹下：

使用 llama.cpp 自带的 python 脚本进行 fp16 转换量化：

```
python3 convert_hf_to_gguf.py /mnt/c/Users/users/model --outtype f16 --  
outfile/mnt/c/Users/users/model/ORLM-FP16.gguf
```



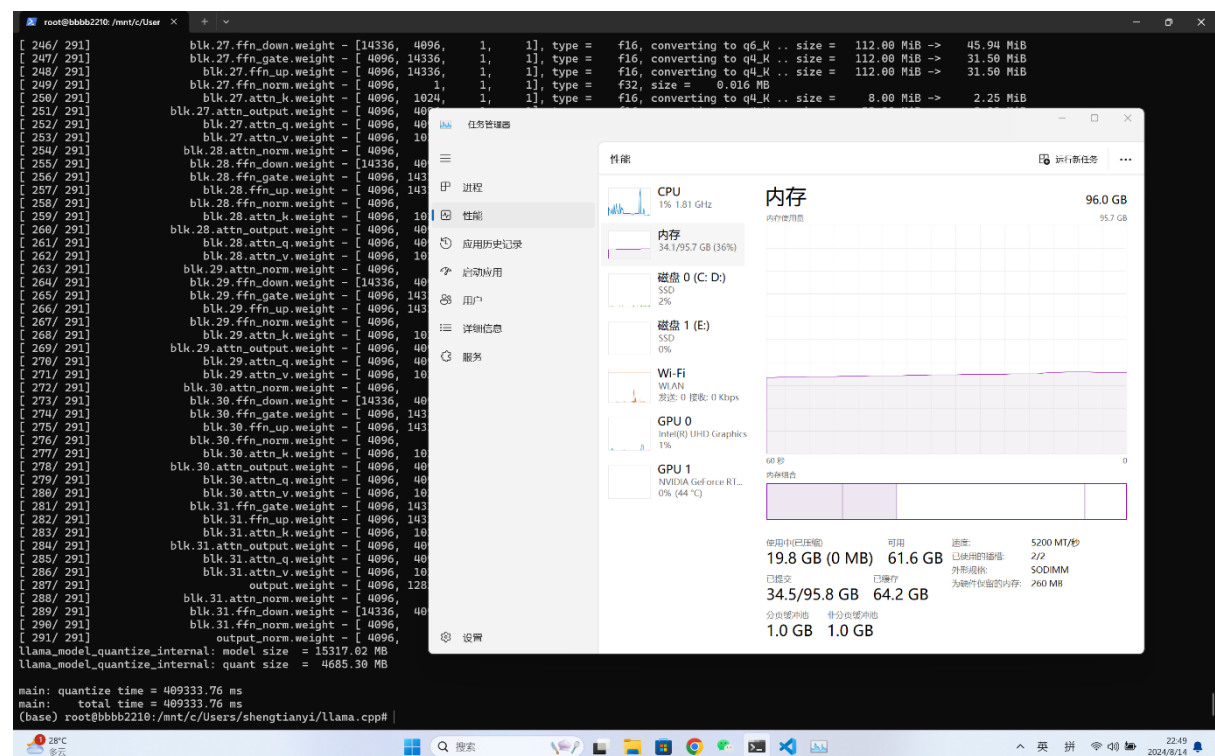
```
INFO:hf-to-gguf:blk.28.ffn_down.weight, torch.bfloat16 -> F16, shape = [14336, 4096]  
INFO:hf-to-gguf:blk.28.ffn_gate.weight, torch.bfloat16 -> F16, shape = [4096, 14336]  
INFO:hf-to-gguf:blk.28.ffn_up.weight, torch.bfloat16 -> F16, shape = [4096, 14336]  
INFO:hf-to-gguf:blk.28.ffn_norm.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.28.attn_k.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.28.attn_q.weight, torch.bfloat16 -> F16, shape = [4096, 4096]  
INFO:hf-to-gguf:blk.28.attn_v.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.28.attn_output.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.29.ffn_down.weight, torch.bfloat16 -> F16, shape = [14336, 4096]  
INFO:hf-to-gguf:blk.29.ffn_gate.weight, torch.bfloat16 -> F16, shape = [4096, 14336]  
INFO:hf-to-gguf:blk.29.ffn_up.weight, torch.bfloat16 -> F16, shape = [4096, 14336]  
INFO:hf-to-gguf:blk.29.ffn_norm.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.29.attn_k.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.29.attn_q.weight, torch.bfloat16 -> F16, shape = [4096, 4096]  
INFO:hf-to-gguf:blk.29.attn_v.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.29.attn_output.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.30.ffn_down.weight, torch.bfloat16 -> F16, shape = [14336, 4096]  
INFO:hf-to-gguf:blk.30.ffn_gate.weight, torch.bfloat16 -> F16, shape = [4096, 14336]  
INFO:hf-to-gguf:blk.30.ffn_up.weight, torch.bfloat16 -> F16, shape = [4096, 14336]  
INFO:hf-to-gguf:blk.30.ffn_norm.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.30.attn_k.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.30.attn_q.weight, torch.bfloat16 -> F16, shape = [4096, 4096]  
INFO:hf-to-gguf:blk.30.attn_v.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.30.attn_output.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.31.ffn_down.weight, torch.bfloat16 -> F16, shape = [4096, 4096]  
INFO:hf-to-gguf:blk.31.ffn_gate.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.31.ffn_up.weight, torch.bfloat16 -> F16, shape = [4096, 14336]  
INFO:hf-to-gguf:blk.31.attn_k.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:blk.31.attn_q.weight, torch.bfloat16 -> F16, shape = [4096, 4096]  
INFO:hf-to-gguf:blk.31.attn_v.weight, torch.bfloat16 -> F16, shape = [4096, 1824]  
INFO:hf-to-gguf:gguf: loading model part 'model-00000-of-00000.safetensors'  
INFO:hf-to-gguf:output.weight, torch.bfloat16 -> F16, shape = [4096, 128256]  
INFO:hf-to-gguf:blk.31.attn_norm.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.31.ffn_down.weight, torch.bfloat16 -> F16, shape = [14336, 4096]  
INFO:hf-to-gguf:blk.31.ffn_gate.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:blk.31.ffn_up.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:output_norm.weight, torch.bfloat16 -> F32, shape = [4096]  
INFO:hf-to-gguf:Set meta model  
INFO:hf-to-gguf:Set model parameters  
INFO:hf-to-gguf:gguf: context length = 8192  
INFO:hf-to-gguf:gguf: embedding length = 4096  
INFO:hf-to-gguf:gguf: feed forward length = 14336  
INFO:hf-to-gguf:gguf: head count = 32  
INFO:hf-to-gguf:gguf: key-value head count = 8  
INFO:hf-to-gguf:gguf: rope theta = 580000.0  
INFO:hf-to-gguf:gguf: rms norm epsilon = 1e-05  
INFO:hf-to-gguf:gguf: file type = 1  
INFO:hf-to-gguf:Set model tokenizer  
INFO:gguf.vocab:Adding 280147 merge(s).  
INFO:gguf.vocab:Setting special token type bos to 128000  
INFO:gguf.vocab:Setting special token type eos to 128001  
INFO:gguf.vocab:Setting special token type pad to 128001  
INFO:hf-to-gguf:Set model quantization version  
INFO:gguf:gguf:writer:Writing the following files:  
INFO:gguf:gguf:writer:/mnt/c/Users/shengtianyi/model/ORLM-FP16.gguf: n_tensors = 201, total_size = 16.1G  
Writing: 46%
```

完成之后会在 model 文件夹下获得 ORLM-FP16.gguf 模型，这是 fp16 量化模型，体积大约 16G，推理需要 20G 左右显存，对一般显卡仍然不具备可用性。如果你的显存够大，那就自便~~~

### 3.3 对 ORLM-f16 进行 Q4\_K\_M 量化：

```
./llama-quantize /mnt/c/Users/users/model/ORLM-FP16.gguf  
/mnt/c/Users/users/model/ ORLM-Q4_K_M.gguf Q4_K_M
```

此步骤需要大量内存，物理内存需要 36G 以上，不足请配足虚拟内存



此时 Q4\_K\_M 量化的 ORLM 模型量化完毕。前往 model 文件夹查看 ORLM-Q4\_K\_M.gguf，然后创建一个文件夹 Modelfile：

touch Modelfile

然后退出 WSL 即可。

### 3.4 使用 Ollama 读入 ORLM-Q4\_K\_M.gguf 并存入模型库

进入 model 文件夹使用编辑器编辑 Modelfile 文件输入：

```
FROM ORLM-Q4_K_M.gguf
```

保存退出即可。

在 model 文件夹下打开两个 Windows PowerShell 文件夹，第一个输入 `ollama serve`，第二个输入 `ollama create ORLM -f Modelfile`，将自动创建并存储 ORLM



模型到 ollama 本地模型库，然后使用之前的 WebUI 脚本，即可在网页使用 ORLM 模型。

