如果碰到服务器无法访问外网情况，可设置环境变量

- 在文件开始写入这两行代码，代表访问HG的镜像社区

```
import os
os.environ['HF_ENDPOINT'] = 'https://hf-mirror.com'
```

- 或者在终端输入 `export USE_MODELSCOPE_HUB=1` 或者 `export HF_ENDPOINT=https://hf-mirror.com`

peft是一个微调开源库，导入 `peft` 名下的 `LoraConfig` 与 `get_peft_model` ，可以快速完成微调

- LoraConfig

```
 self.loraconfig =  LoraConfig(
            r=config["lora"]["r"],
            target_modules=config["lora"]["target_modules"],
            bias=config["lora"]["bias"],
            lora_alpha=config["lora"]["lora_alpha"],
            lora_dropout=config["lora"]["lora_dropout"],
        )
```

  - `r` ：秩，直接决定微调的参数量
  - `lora_alpha` ：一般为2r或者r
  - `lora_dropout` ：随机将低秩矩阵参数置0
  - `target_modules` ：对哪几个层进行微调

- get_peft_model

  - 传入加载好的模型和配置好的lora参数，可以得到一个配置好的模型

  - `.print_trainable_parameters()` 可以查看能微调多少参数

```
self.peft_model = get_peft_model(self.model,self.loraconfig)
        self.peft_model.print_trainable_parameters()
```

导入 `transformers` ，可以快速加载模型，提词器，以及量化配置

- `AutoModelForCausalLM`

  - 加载预训练模型的工具

    - `device_map` ：使用什么设备加载模型参数
    - `quantization_config` ：量化配置
    - `trust_remote_code` ：是否信任远程代码

```
self.model = AutoModelForCausalLM.from_pretrained(
            config["model"]["model_path"],
            device_map=config["model"]["device_map"],
            trust_remote_code=config["model"]["trust_remote_code"],
```

```
            quantization_config=quan_configs,
        )
```

- `AutoTokenizer`
  - □动化分词器加载工具，将文本转换为模型可理解的数字形式，指定了模型路径会自动下载匹配的分词器

    ```
    self.tokenizer = AutoTokenizer.from_pretrained(
                config["model"]["model_path"],
                trust_remote_code=config["tokenizer"]["trust_remote_code"],
            )
    ```

  - 用分词器处理文本
    - `max_length`：能接受的文本最大长度，超过这个长度则截断
    - `return_tensors`：可为 `pt`，最后返回的就是张量
    - ``truncation`：采取何种截断方式

      ```
      tokenized = self.tokenizer(
                 combined_texts,
                 truncation=True,
              max_length=config["tokenizer"]["max_length"],
                 padding=False,
                 return_tensors=None   # □回Python列表而不是张量
             )
      ```

  - `.tokenizer.encode`：将原始文本（字符串）转换为模型可理解的 Token ID 序列
    - `text`：输入文本
    - `add_special_tokens`：是否添加特殊token
    - `max_length`：截断
    - `truncation`：是否启用截断
    - `rerturn_tensors`：返回格式（□ `pt`）

      ```
      turn_tokens = self.tokenizer.encode(
                          self.tokenizer.apply_chat_template([turn],
      tokenize=False),
                          add_special_tokens=False
                      )
      ```

  - `.apply_chat_template`：将多轮对话历史转换为模型所需的标准化格式
    - `tokenize`：是否返回token IDs，为true则返回模型能理解的数字
    - `return_tensors`：返回张量格式，`"pt"` (PyTorch)，`"np"` (NumPy)
    - `max_length`：最大token长度

        - `truncation`：超长是否截断
        - `padding`：填充方式，`"longest"`、`True`、`False`
        - `return_dict`：是否返回字典格式

```
tokenized = self.tokenizer.apply_chat_template(
                conversation,
                tokenize=True,
                max_length=config["tokenizer"]["max_length"],
                truncation=config["tokenizer"]["truncation"],
                return_tensors=None,
                return_dict=True
            )
```

- `BitsAndBytesConfig`
    - 模型量化的工具，最后要作为参数传入给`load_model`
        - `load_in_8bit`：最后以8bit整数保存模型参数
        - `load_in_4bit`：最后以4bit整数保存模型
        - `torch_dtype`：模型训练的过程的中使用的参数类型

```
quan_configs = BitsAndBytesConfig(
            load_in_8bit=config["model"]["load_in_8bit"],
            torch_dtype=config["model"]["torch_dtype"]
        )
```

- `Trainer`
    - 训练执行引擎
        - `model`：待训练的模型，微调的话则需要传入`peft_model`
        - `tokenizer`：分词器
        - `args`：`TrainingArguments`实例
        - `train_dataset`：训练集
        - `eval_dataset`：测试集
        - `data_collator`：数据整理器
        - `callbacks`：回调函数，用于早停

```
trainer = Trainer(
            model=model,
            tokenizer=self.tokenizer,
            args=training_args,
            train_dataset=train_dataset,
            eval_dataset=eval_dataset,
            data_collator=data_collator,
```

```
            callbacks=[EarlyStoppingCallback(config["training"]
["patience"])]
        )
```

- `TrainingArguments`
  - 控制训练过程的超参数和基础设施配置
    - `output_dir`：输出目录
    - `learning_rate`：初始学习率
    - `num_train_epochs`：训练轮次
    - `weight_decay`：权重衰减，L2正则化
    - `per_device_train_batch_size`：训练的时候每个设备处理的批次大小，占用显存
    - `per_device_eval_batch_size`：推理的时候每个设备处理的批次大小，不占用显存
    - `eval_strategy`：模型评估策略
    - `save_strategy`：模型保存策略
    - `eval_steps`：每N步评估一次
    - `save_steps`：每N步保存一次
    - `logging_steps`：每N步记录日志
    - `logging_dir`：TensorBoard日志目录
    - `report_to`：日志上报平台
    - `fp16`：是否启用混合精度训练
    - `gradient_accumulation_steps`：梯度累积步数，用时间换空间策略
    - `gradient_checkpointing`：是否开启梯度检查点，开启的话可以节省显存，因为正向传播的时候系统不会暂存中间值，而是重新计算
    - `max_grad_norm`：梯度裁剪，防止梯度爆炸
    - `load_best_model_at_end`：最佳模型自动保存
      - 启用自动保存后，保存模型的步数要是评估步数的倍数，要保证每次评估之后都能保存，而不是评估到一半保存
    - `lr_scheduler_type`：学习率调度器
      - `cosine`：余弦退火
      - `linear`：线性衰减
    - `metric_for_best_model`：

```
        training_args = TrainingArguments(
            output_dir=config["model"]["output_path"],
            learning_rate=config["training"]["learning_rate"],
            num_train_epochs=config["training"]["num_epochs"],
```

```
            weight_decay=config["training"]["weight_decay"],
            per_device_train_batch_size=config["training"]
["per_device_train_batch_size"],
            per_device_eval_batch_size=config["training"]
["per_device_eval_batch_size"],
            eval_strategy=config["training"]["eval_strategy"],
            save_strategy=config["training"]["save_strategy"],
            eval_steps=config["training"]["eval_steps"],
            save_steps=config["training"]["save_steps"],
            logging_steps=config["training"]["logging_steps"],
            logging_dir=config["training"]["logging_dir"],
            report_to=config["training"]["report_to"],
            load_best_model_at_end=config["training"]
["load_best_model_at_end"],
            fp16=config["training"]["fp16"],
            gradient_accumulation_steps=config["training"]
["gradient_accumulation_steps"],
            dataloader_num_workers=4,  # 增加数据加载器的工作进程
        )
```

- `datasets.Dataset` 是 Hugging Face `datasets` 库中的核心类，用于高效处理大规模数据集
    - `.from_dict()`：用于快速构建数据集，返回的是字典
    - `.map()`：对数据集中的每一行或批次应用自定义函数
        - `function`：函数名
        - `batched`：是否批次处理
        - `batch_size`：每批次的样本数
        - `num_proc`：使用的进程数
    - `.save_to_disk`：存储Dataset格式的数据集
- `load_dataset`：是 `datasets` 中一个加载数据集的方法
    - `split`：用于指定加载数据集的哪个子集，可以指定加载百分之多少，使用切片语法
    - `streaming`：是否启用流式加载，避免OOM，不会一次性把数据集读入内存，仅受磁盘空间限制

```
 self.dataset = load_dataset("BelleGroup/multiturn_chat_0.8M",
split="train[:70%]",streaming=true)
```

- `DataCollatorForSeq2Seq`：用于动态批处理输入数据，并自动处理填充、注意力掩码和标签对齐
    - `tokenizer`：分词器
    - `model`：模型，可用于自动推断解码器结构
    - `padding`：填充策略
    - `max_length`：文本最长的长度
    - `pad_to_multiple_of`：将长度填充到该值的倍数
    - `label_pad_token_id`：标签序列的填充 Token ID
    - `return_tensors`：返回的张量格式

```
data_collator = DataCollatorForSeq2Seq(
            tokenizer=self.tokenizer,
            pad_to_multiple_of=config["dataloader"]
["pad_to_multiple_of"],
            return_tensors=config["dataloader"]["return_tensors"],
            padding=config["dataloader"]["padding"],
            label_pad_token_id=config["dataloader"]
["label_pad_token_id"]
        )
```