深度学习模型部署的工程实践

由于我知识的有限，这篇说明难免有些理论或实践不太严谨正确的地方。

前言

深度学习模型的部署需要解决至少以下问题：

- 屏蔽模型使用的框架、版本依赖
- 模型的推理
- 算法服务端的设计

模型的部署必然有一个应用系统去承载它，算法服务作为后端或者整个后端的一部分。它需要由一门工程语言去实现它。仅考虑推理的话，那么其实现的工程语言决定了算法服务端的设计与实现（主要是模型的推理策略以及分布式系统的通信），以及用什么样的方式去部署模型。

例如：假设我们的算法服务采用python编写，那么python对pytorch的友好性和支持就可以很利于模型的部署（由pytorch导出模型，python端可以很自然的调用模型），这里仍然需要去做到屏蔽掉模型的框架和版本依赖。现在pytorch对这一点支持做的比较好了，可以由以下方法导出和调用模型（更细节的参见官方文档）：

```
model.eval()
torch.jit.trace()

torch.jit.script()
torch.jit.load()
```

而算法服务端的设计，假设我们采取http做通信，那就好做了，可以考虑用Flask或Django开发服务端，对外暴露http接口。但这里仍然需要去考虑模型的调用策略，原因至少有以下几点：

- python的多线程是一种伪多线程（类比多线程模型中的N:1），同一进程中的线程并不会并行的执行在多核上，如何应对并发请求
- 如果使用torch的GPU版本，则需要去了解一点NVIDIA GPU的任务调度策略（结论，并不会像CPU的多线程那样，对每一个线程而言，其任务的执行会略微增加或者不增加，GPU上每一个任务的执行耗时会显著增加），当然这里需要看模型的大小以及数据的大小，想要的是什么，吞吐量还是响应时间

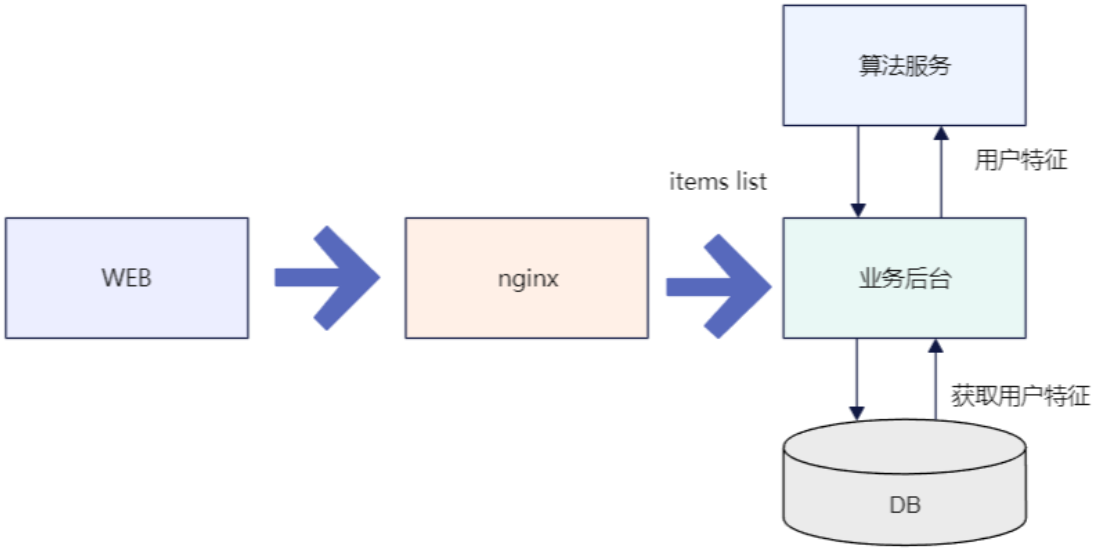
实践

这里我们模型部署的应用场景是一个简单的推荐系统。仅考虑一个召回通道（且未做排序），基于物品的协同过滤。我们让这个过程尽可能的简化，最终的业务流程为：

- 业务后台接受到用户请求，查询用户信息，将用户的特征以字符串的方式发送给算法后台
- 算法后台接受请求，解析出数据，执行推理，并将推理结果（根据得分排序的商品id）返回给业务后台

顶层设计

根据我们的业务流程，我们选择使用gRPC做通信，LibTorch（Torch C++ API）做模型部署，C++编写算法后台服务。



如上图所示，我们在业务后台对外暴露http接口，内部使用RPC做分布式系统之间的通信。

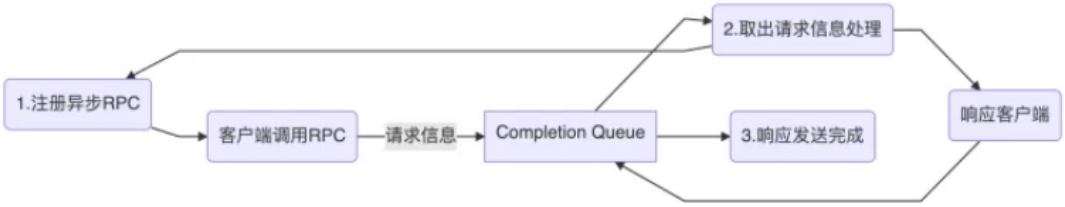
算法服务端

算法服务也是一个服务器，需要考虑并发的问题。在C10K问题中（更高级的我也不清楚了），主要使用了epoll、reactor和线程池来解决。epoll模型实现并发请求的读取，reactor模式做分发和任务调度，线程池执行逻辑处理。

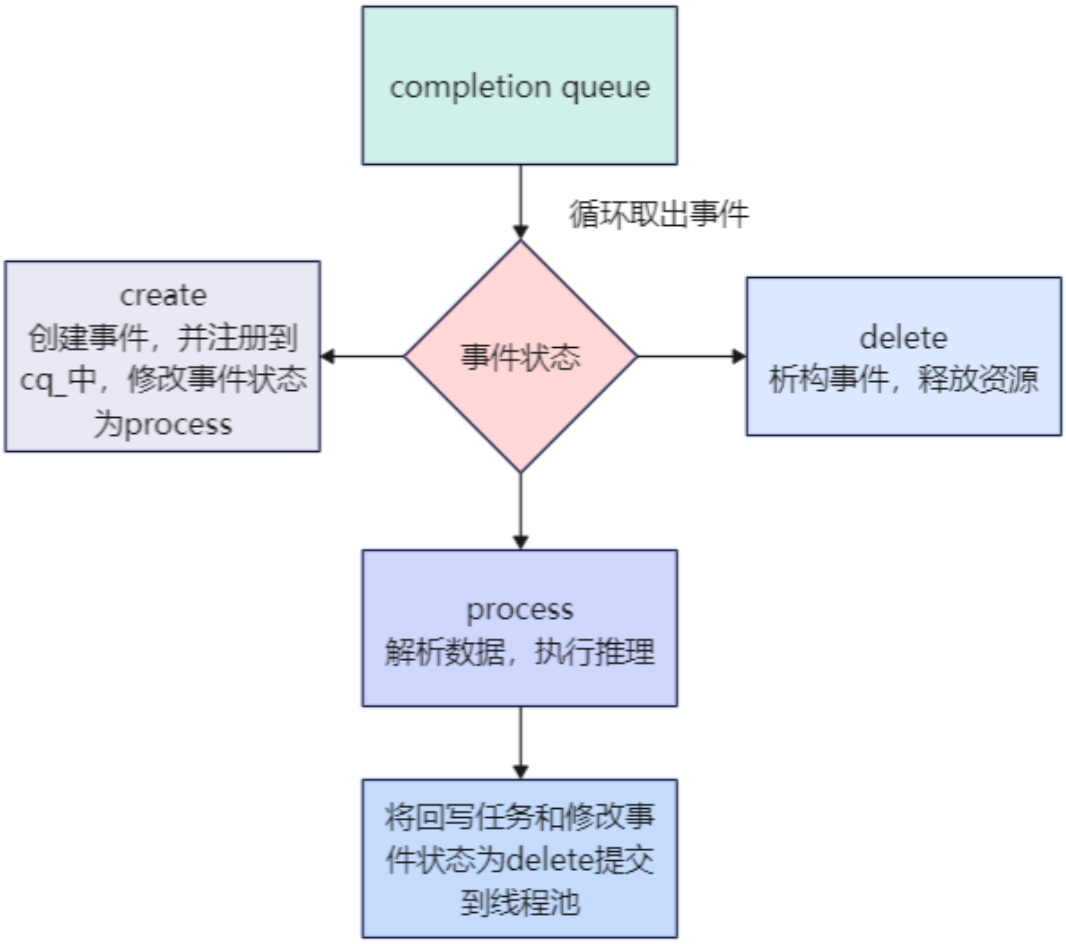
我们的服务端使用了gRPC的异步服务器模式（等价于gRPC内部实现了一个epoll + reactor），对外，暴露一个队列，我们需要从这个队列中读取事件，并进行分发和处理。

由前言所述，考虑到推荐系统的需要的商品的embedding数量是巨量的（需要装载在显存里，如果并行执行推理，那么每个推理任务需要保存的上下文信息可能也是巨大的），且需要低延迟返回，所以我们采取了串行的执行推理，推理结果的回写提交到线程池并行处理的策略。

对一个RPC请求相应的整体流程图：



算法服务内部逻辑流程图：



RPC接口：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.FinalAiServer.rs";
option java_outer_classname = "RsProto";
option objc_class_prefix = "HLW";

package rs;

// The greeting service definition.
service RSer {
  // Sends a greeting
  rpc CallRsItems (userRequest) returns (itemsString) {}
}

// The request message containing the user's name.
message userRequest {
  string vector = 1;
}

// The response message containing the greetings
message itemsString {
  string items = 1;
}
```

模型的接口

这里认为一个模型就是一个算法服务，包括数据的解析（将接受到的数据，按照一定的规则解析成可执行推理的张量），推理。将模型的加载，推理，以及数据解析封装在一个类里。

```
namespace rs {
    class RS_Model{
        // 由于这里使用了torch的C++的接口，并不清楚其构造原理，这个类本身也并未申请堆区的资源，所以这里并没有显示的编写析构函数
    public:
        // 根据用户指定的地址以及输入张量的维度信息加载模型，由于
        RS_Model(const char* model_path, at::IntArrayRef& input_tensor_shape);

        // 这里选择将数据的解析和推理封装在一起，返回推理后的结果
        std::string rs_handle(const std::string& embedding_str);

        // 对外暴露一个根据输入的张量进行推理的接口，但是需要检查输入的张量
        std::vector<float> call_inference(std::vector<float>& input_value);
    private:

        // 执行推理
        std::vector<float> inference(std::vector<float>& input_value);
        // model
        torch::jit::script::Module _model;
        //std::string _model_path;
        // 用于初始化输入到模型的维度信息
        at::IntArrayRef _input_tensor_shape;
    };
}
```