use anyhow::Result;

use log::{info, warn};

use x509\_parser::{prelude::{parse\_x509\_pem}, parse\_x509\_certificate};

use std::collections::HashMap;

use tokio::time::Instant;

use tonic::{

Request,

Response, Status, transport::{Certificate, Identity, Server, ServerTlsConfig},

};

// protobuf related

use crate::tf\_proto::tensorflow\_serving::{

ClassificationRequest, ClassificationResponse, GetModelMetadataRequest,

GetModelMetadataResponse, MultiInferenceRequest, MultiInferenceResponse, PredictRequest,

PredictResponse, RegressionRequest, RegressionResponse,

};

use crate::{kf\_serving::{

grpc\_inference\_service\_server::GrpcInferenceService, ModelInferRequest, ModelInferResponse,

ModelMetadataRequest, ModelMetadataResponse, ModelReadyRequest, ModelReadyResponse,

ServerLiveRequest, ServerLiveResponse, ServerMetadataRequest, ServerMetadataResponse,

ServerReadyRequest, ServerReadyResponse,

}, ModelFactory, tf\_proto::tensorflow\_serving::prediction\_service\_server::{

PredictionService, PredictionServiceServer,

}, VERSION, NAME};

use crate::PredictResult;

use crate::cli\_args::{ARGS, INPUTS, OUTPUTS};

use crate::metrics::{

NAVI\_VERSION, NUM\_PREDICTIONS, NUM\_REQUESTS\_FAILED, NUM\_REQUESTS\_FAILED\_BY\_MODEL,

NUM\_REQUESTS\_RECEIVED, NUM\_REQUESTS\_RECEIVED\_BY\_MODEL, RESPONSE\_TIME\_COLLECTOR,

CERT\_EXPIRY\_EPOCH

};

use crate::predict\_service::{Model, PredictService};

use crate::tf\_proto::tensorflow\_serving::model\_spec::VersionChoice::Version;

use crate::tf\_proto::tensorflow\_serving::ModelSpec;

#[derive(Debug)]

pub enum TensorInputEnum {

String(Vec<Vec<u8>>),

Int(Vec<i32>),

Int64(Vec<i64>),

Float(Vec<f32>),

Double(Vec<f64>),

Boolean(Vec<bool>),

}

#[derive(Debug)]

pub struct TensorInput {

pub tensor\_data: TensorInputEnum,

pub name: String,

pub dims: Option<Vec<i64>>,

}

impl TensorInput {

pub fn new(tensor\_data: TensorInputEnum, name: String, dims: Option<Vec<i64>>) -> TensorInput {

TensorInput {

tensor\_data,

name,

dims,

}

}

}

impl TensorInputEnum {

#[inline(always)]

pub(crate) fn extend(&mut self, another: TensorInputEnum) {

match (self, another) {

(Self::String(input), Self::String(ex)) => input.extend(ex),

(Self::Int(input), Self::Int(ex)) => input.extend(ex),

(Self::Int64(input), Self::Int64(ex)) => input.extend(ex),

(Self::Float(input), Self::Float(ex)) => input.extend(ex),

(Self::Double(input), Self::Double(ex)) => input.extend(ex),

(Self::Boolean(input), Self::Boolean(ex)) => input.extend(ex),

x => panic!("input enum type not matched. input:{:?}, ex:{:?}", x.0, x.1),

}

}

#[inline(always)]

pub(crate) fn merge\_batch(input\_tensors: Vec<Vec<TensorInput>>) -> Vec<TensorInput> {

input\_tensors

.into\_iter()

.reduce(|mut acc, e| {

for (i, ext) in acc.iter\_mut().zip(e) {

i.tensor\_data.extend(ext.tensor\_data);

}

acc

})

.unwrap() //invariant: we expect there's always rows in input\_tensors

}

}

///entry point for tfServing gRPC

#[tonic::async\_trait]

impl<T: Model> GrpcInferenceService for PredictService<T> {

async fn server\_live(

&self,

\_request: Request<ServerLiveRequest>,

) -> Result<Response<ServerLiveResponse>, Status> {

unimplemented!()

}

async fn server\_ready(

&self,

\_request: Request<ServerReadyRequest>,

) -> Result<Response<ServerReadyResponse>, Status> {

unimplemented!()

}

async fn model\_ready(

&self,

\_request: Request<ModelReadyRequest>,

) -> Result<Response<ModelReadyResponse>, Status> {

unimplemented!()

}

async fn server\_metadata(

&self,

\_request: Request<ServerMetadataRequest>,

) -> Result<Response<ServerMetadataResponse>, Status> {

unimplemented!()

}

async fn model\_metadata(

&self,

\_request: Request<ModelMetadataRequest>,

) -> Result<Response<ModelMetadataResponse>, Status> {

unimplemented!()

}

async fn model\_infer(

&self,

\_request: Request<ModelInferRequest>,

) -> Result<Response<ModelInferResponse>, Status> {

unimplemented!()

}

}

#[tonic::async\_trait]

impl<T: Model> PredictionService for PredictService<T> {

async fn classify(

&self,

\_request: Request<ClassificationRequest>,

) -> Result<Response<ClassificationResponse>, Status> {

unimplemented!()

}

async fn regress(

&self,

\_request: Request<RegressionRequest>,

) -> Result<Response<RegressionResponse>, Status> {

unimplemented!()

}

async fn predict(

&self,

request: Request<PredictRequest>,

) -> Result<Response<PredictResponse>, Status> {

NUM\_REQUESTS\_RECEIVED.inc();

let start = Instant::now();

let mut req = request.into\_inner();

let (model\_spec, version) = req.take\_model\_spec();

NUM\_REQUESTS\_RECEIVED\_BY\_MODEL

.with\_label\_values(&[&model\_spec])

.inc();

let idx = PredictService::<T>::get\_model\_index(&model\_spec).ok\_or\_else(|| {

Status::failed\_precondition(format!("model spec not found:{}", model\_spec))

})?;

let input\_spec = match INPUTS[idx].get() {

Some(input) => input,

\_ => return Err(Status::not\_found(format!("model input spec {}", idx))),

};

let input\_val = req.take\_input\_vals(input\_spec);

self.predict(idx, version, input\_val, start)

.await

.map\_or\_else(

|e| {

NUM\_REQUESTS\_FAILED.inc();

NUM\_REQUESTS\_FAILED\_BY\_MODEL

.with\_label\_values(&[&model\_spec])

.inc();

Err(Status::internal(e.to\_string()))

},

|res| {

RESPONSE\_TIME\_COLLECTOR

.with\_label\_values(&[&model\_spec])

.observe(start.elapsed().as\_millis() as f64);

match res {

PredictResult::Ok(tensors, version) => {

let mut outputs = HashMap::new();

NUM\_PREDICTIONS.with\_label\_values(&[&model\_spec]).inc();

//FIXME: uncomment when prediction scores are normal

// PREDICTION\_SCORE\_SUM

// .with\_label\_values(&[&model\_spec])

// .inc\_by(tensors[0]as f64);

for (tp, output\_name) in tensors

.into\_iter()

.map(|tensor| tensor.create\_tensor\_proto())

.zip(OUTPUTS[idx].iter())

{

outputs.insert(output\_name.to\_owned(), tp);

}

let reply = PredictResponse {

model\_spec: Some(ModelSpec {

version\_choice: Some(Version(version)),

..Default::default()

}),

outputs,

};

Ok(Response::new(reply))

}

PredictResult::DropDueToOverload => Err(Status::resource\_exhausted("")),

PredictResult::ModelNotFound(idx) => {

Err(Status::not\_found(format!("model index {}", idx)))

},

PredictResult::ModelNotReady(idx) => {

Err(Status::unavailable(format!("model index {}", idx)))

}

PredictResult::ModelVersionNotFound(idx, version) => Err(

Status::not\_found(format!("model index:{}, version {}", idx, version)),

),

}

},

)

}

async fn multi\_inference(

&self,

\_request: Request<MultiInferenceRequest>,

) -> Result<Response<MultiInferenceResponse>, Status> {

unimplemented!()

}

async fn get\_model\_metadata(

&self,

\_request: Request<GetModelMetadataRequest>,

) -> Result<Response<GetModelMetadataResponse>, Status> {

unimplemented!()

}

}

// A function that takes a timestamp as input and returns a ticker stream

fn report\_expiry(expiry\_time: i64) {

info!("Certificate expires at epoch: {:?}", expiry\_time);

CERT\_EXPIRY\_EPOCH.set(expiry\_time as i64);

}

pub fn bootstrap<T: Model>(model\_factory: ModelFactory<T>) -> Result<()> {

info!("package: {}, version: {}, args: {:?}", NAME, VERSION, \*ARGS);

//we follow SemVer. So here we assume MAJOR.MINOR.PATCH

let parts = VERSION

.split(".")

.map(|v| v.parse::<i64>())

.collect::<std::result::Result<Vec<\_>, \_>>()?;

if let [major, minor, patch] = &parts[..] {

NAVI\_VERSION.set(major \* 1000\_000 + minor \* 1000 + patch);

} else {

warn!(

"version {} doesn't follow SemVer conversion of MAJOR.MINOR.PATCH",

VERSION

);

}

tokio::runtime::Builder::new\_multi\_thread()

.thread\_name("async worker")

.worker\_threads(ARGS.num\_worker\_threads)

.max\_blocking\_threads(ARGS.max\_blocking\_threads)

.enable\_all()

.build()

.unwrap()

.block\_on(async {

#[cfg(feature = "navi\_console")]

console\_subscriber::init();

let addr = format!("0.0.0.0:{}", ARGS.port).parse()?;

let ps = PredictService::init(model\_factory).await;

let mut builder = if ARGS.ssl\_dir.is\_empty() {

Server::builder()

} else {

// Read the pem file as a string

let pem\_str = std::fs::read\_to\_string(format!("{}/server.crt", ARGS.ssl\_dir)).unwrap();

let res = parse\_x509\_pem(&pem\_str.as\_bytes());

match res {

Ok((rem, pem\_2)) => {

assert!(rem.is\_empty());

assert\_eq!(pem\_2.label, String::from("CERTIFICATE"));

let res\_x509 = parse\_x509\_certificate(&pem\_2.contents);

info!("Certificate label: {}", pem\_2.label);

assert!(res\_x509.is\_ok());

report\_expiry(res\_x509.unwrap().1.validity().not\_after.timestamp());

},

\_ => panic!("PEM parsing failed: {:?}", res),

}

let key = tokio::fs::read(format!("{}/server.key", ARGS.ssl\_dir))

.await

.expect("can't find key file");

let crt = tokio::fs::read(format!("{}/server.crt", ARGS.ssl\_dir))

.await

.expect("can't find crt file");

let chain = tokio::fs::read(format!("{}/server.chain", ARGS.ssl\_dir))

.await

.expect("can't find chain file");

let mut pem = Vec::new();

pem.extend(crt);

pem.extend(chain);

let identity = Identity::from\_pem(pem.clone(), key);

let client\_ca\_cert = Certificate::from\_pem(pem.clone());

let tls = ServerTlsConfig::new()

.identity(identity)

.client\_ca\_root(client\_ca\_cert);

Server::builder()

.tls\_config(tls)

.expect("fail to config SSL")

};

info!(

"Prometheus server started: 0.0.0.0: {}",

ARGS.prometheus\_port

);

let ps\_server = builder

.add\_service(PredictionServiceServer::new(ps).accept\_gzip().send\_gzip())

.serve(addr);

info!("Prediction server started: {}", addr);

ps\_server.await.map\_err(anyhow::Error::msg)

})

}