use crate::{MAX\_NUM\_INPUTS, MAX\_NUM\_MODELS, MAX\_NUM\_OUTPUTS};

use arrayvec::ArrayVec;

use clap::Parser;

use log::info;

use once\_cell::sync::OnceCell;

use std::error::Error;

use time::OffsetDateTime;

use time::format\_description::well\_known::Rfc3339;

#[derive(Parser, Debug, Clone)]

///Navi is configured through CLI arguments(for now) defined below.

//TODO: use clap\_serde to make it config file driven

pub struct Args {

#[clap(short, long, help = "gRPC port Navi runs ons")]

pub port: i32,

#[clap(long, default\_value\_t = 9000, help = "prometheus metrics port")]

pub prometheus\_port: u16,

#[clap(

short,

long,

default\_value\_t = 1,

help = "number of worker threads for tokio async runtime"

)]

pub num\_worker\_threads: usize,

#[clap(

long,

default\_value\_t = 14,

help = "number of blocking threads in tokio blocking thread pool"

)]

pub max\_blocking\_threads: usize,

#[clap(long, default\_value = "16", help = "maximum batch size for a batch")]

pub max\_batch\_size: Vec<String>,

#[clap(

short,

long,

default\_value = "2",

help = "max wait time for accumulating a batch"

)]

pub batch\_time\_out\_millis: Vec<String>,

#[clap(

long,

default\_value\_t = 90,

help = "threshold to start dropping batches under stress"

)]

pub batch\_drop\_millis: u64,

#[clap(

long,

default\_value\_t = 300,

help = "polling interval for new version of a model and META.json config"

)]

pub model\_check\_interval\_secs: u64,

#[clap(

short,

long,

default\_value = "models/pvideo/",

help = "root directory for models"

)]

pub model\_dir: Vec<String>,

#[clap(

long,

help = "directory containing META.json config. separate from model\_dir to facilitate remote config management"

)]

pub meta\_json\_dir: Option<String>,

#[clap(short, long, default\_value = "", help = "directory for ssl certs")]

pub ssl\_dir: String,

#[clap(

long,

help = "call out to external process to check model updates. custom logic can be written to pull from hdfs, gcs etc"

)]

pub modelsync\_cli: Option<String>,

#[clap(

long,

default\_value\_t = 1,

help = "specify how many versions Navi retains in memory. good for cases of rolling model upgrade"

)]

pub versions\_per\_model: usize,

#[clap(

short,

long,

help = "most runtimes support loading ops custom writen. currently only implemented for TF"

)]

pub customops\_lib: Option<String>,

#[clap(

long,

default\_value = "8",

help = "number of threads to paralleling computations inside an op"

)]

pub intra\_op\_parallelism: Vec<String>,

#[clap(

long,

help = "number of threads to parallelize computations of the graph"

)]

pub inter\_op\_parallelism: Vec<String>,

#[clap(

long,

help = "signature of a serving. only TF"

)]

pub serving\_sig: Vec<String>,

#[clap(long, default\_value = "examples", help = "name of each input tensor")]

pub input: Vec<String>,

#[clap(long, default\_value = "output\_0", help = "name of each output tensor")]

pub output: Vec<String>,

#[clap(

long,

default\_value\_t = 500,

help = "max warmup records to use. warmup only implemented for TF"

)]

pub max\_warmup\_records: usize,

#[clap(long, value\_parser = Args::parse\_key\_val::<String, String>, value\_delimiter=',')]

pub onnx\_global\_thread\_pool\_options: Vec<(String, String)>,

#[clap(

long,

default\_value = "true",

help = "when to use graph parallelization. only for ONNX"

)]

pub onnx\_use\_parallel\_mode: String,

// #[clap(long, default\_value = "false")]

// pub onnx\_use\_onednn: String,

#[clap(

long,

default\_value = "true",

help = "trace internal memory allocation and generate bulk memory allocations. only for ONNX. turn if off if batch size dynamic"

)]

pub onnx\_use\_memory\_pattern: String,

#[clap(long, value\_parser = Args::parse\_key\_val::<String, String>, value\_delimiter=',')]

pub onnx\_ep\_options: Vec<(String, String)>,

#[clap(long, help = "choice of gpu EPs for ONNX: cuda or tensorrt")]

pub onnx\_gpu\_ep: Option<String>,

#[clap(

long,

default\_value = "home",

help = "converter for various input formats"

)]

pub onnx\_use\_converter: Option<String>,

#[clap(

long,

help = "whether to enable runtime profiling. only implemented for ONNX for now"

)]

pub profiling: Option<String>,

#[clap(

long,

default\_value = "",

help = "metrics reporting for discrete features. only for Home converter for now"

)]

pub onnx\_report\_discrete\_feature\_ids: Vec<String>,

#[clap(

long,

default\_value = "",

help = "metrics reporting for continuous features. only for Home converter for now"

)]

pub onnx\_report\_continuous\_feature\_ids: Vec<String>,

}

impl Args {

pub fn get\_model\_specs(model\_dir: Vec<String>) -> Vec<String> {

let model\_specs = model\_dir

.iter()

//let it panic if some model\_dir are wrong

.map(|dir| {

dir.trim\_end\_matches('/')

.rsplit\_once('/')

.unwrap()

.1

.to\_owned()

})

.collect();

info!("all model\_specs: {:?}", model\_specs);

model\_specs

}

pub fn version\_str\_to\_epoch(dt\_str: &str) -> Result<i64, anyhow::Error> {

dt\_str

.parse::<i64>()

.or\_else(|\_| {

let ts = OffsetDateTime::parse(dt\_str, &Rfc3339)

.map(|d| (d.unix\_timestamp\_nanos()/1\_000\_000) as i64);

if ts.is\_ok() {

info!("original version {} -> {}", dt\_str, ts.unwrap());

}

ts

})

.map\_err(anyhow::Error::msg)

}

/// Parse a single key-value pair

fn parse\_key\_val<T, U>(s: &str) -> Result<(T, U), Box<dyn Error + Send + Sync + 'static>>

where

T: std::str::FromStr,

T::Err: Error + Send + Sync + 'static,

U: std::str::FromStr,

U::Err: Error + Send + Sync + 'static,

{

let pos = s

.find('=')

.ok\_or\_else(|| format!("invalid KEY=value: no `=` found in `{}`", s))?;

Ok((s[..pos].parse()?, s[pos + 1..].parse()?))

}

}

lazy\_static! {

pub static ref ARGS: Args = Args::parse();

pub static ref MODEL\_SPECS: ArrayVec<String, MAX\_NUM\_MODELS> = {

let mut specs = ArrayVec::<String, MAX\_NUM\_MODELS>::new();

Args::get\_model\_specs(ARGS.model\_dir.clone())

.into\_iter()

.for\_each(|m| specs.push(m));

specs

};

pub static ref INPUTS: ArrayVec<OnceCell<ArrayVec<String, MAX\_NUM\_INPUTS>>, MAX\_NUM\_MODELS> = {

let mut inputs =

ArrayVec::<OnceCell<ArrayVec<String, MAX\_NUM\_INPUTS>>, MAX\_NUM\_MODELS>::new();

for (idx, o) in ARGS.input.iter().enumerate() {

if o.trim().is\_empty() {

info!("input spec is empty for model {}, auto detect later", idx);

inputs.push(OnceCell::new());

} else {

inputs.push(OnceCell::with\_value(

o.split(",")

.map(|s| s.to\_owned())

.collect::<ArrayVec<String, MAX\_NUM\_INPUTS>>(),

));

}

}

info!("all inputs:{:?}", inputs);

inputs

};

pub static ref OUTPUTS: ArrayVec<ArrayVec<String, MAX\_NUM\_OUTPUTS>, MAX\_NUM\_MODELS> = {

let mut outputs = ArrayVec::<ArrayVec<String, MAX\_NUM\_OUTPUTS>, MAX\_NUM\_MODELS>::new();

for o in ARGS.output.iter() {

outputs.push(

o.split(",")

.map(|s| s.to\_owned())

.collect::<ArrayVec<String, MAX\_NUM\_OUTPUTS>>(),

);

}

info!("all outputs:{:?}", outputs);

outputs

};

}