// Autogenerated by Thrift Compiler (0.17.0)

// DO NOT EDIT UNLESS YOU ARE SURE THAT YOU KNOW WHAT YOU ARE DOING

#![allow(unused\_imports)]

#![allow(unused\_extern\_crates)]

#![allow(clippy::too\_many\_arguments, clippy::type\_complexity, clippy::vec\_box)]

#![cfg\_attr(rustfmt, rustfmt\_skip)]

use std::cell::RefCell;

use std::collections::{BTreeMap, BTreeSet};

use std::convert::{From, TryFrom};

use std::default::Default;

use std::error::Error;

use std::fmt;

use std::fmt::{Display, Formatter};

use std::rc::Rc;

use thrift::OrderedFloat;

use thrift::{ApplicationError, ApplicationErrorKind, ProtocolError, ProtocolErrorKind, TThriftClient};

use thrift::protocol::{TFieldIdentifier, TListIdentifier, TMapIdentifier, TMessageIdentifier, TMessageType, TInputProtocol, TOutputProtocol, TSerializable, TSetIdentifier, TStructIdentifier, TType};

use thrift::protocol::field\_id;

use thrift::protocol::verify\_expected\_message\_type;

use thrift::protocol::verify\_expected\_sequence\_number;

use thrift::protocol::verify\_expected\_service\_call;

use thrift::protocol::verify\_required\_field\_exists;

use thrift::server::TProcessor;

#[derive(Copy, Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct DataType(pub i32);

impl DataType {

pub const FLOAT: DataType = DataType(0);

pub const DOUBLE: DataType = DataType(1);

pub const INT32: DataType = DataType(2);

pub const INT64: DataType = DataType(3);

pub const UINT8: DataType = DataType(4);

pub const STRING: DataType = DataType(5);

pub const BYTE: DataType = DataType(6);

pub const BOOL: DataType = DataType(7);

pub const RESERVED\_1: DataType = DataType(8);

pub const RESERVED\_2: DataType = DataType(9);

pub const RESERVED\_3: DataType = DataType(10);

pub const ENUM\_VALUES: &'static [Self] = &[

Self::FLOAT,

Self::DOUBLE,

Self::INT32,

Self::INT64,

Self::UINT8,

Self::STRING,

Self::BYTE,

Self::BOOL,

Self::RESERVED\_1,

Self::RESERVED\_2,

Self::RESERVED\_3,

];

}

impl TSerializable for DataType {

#[allow(clippy::trivially\_copy\_pass\_by\_ref)]

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

o\_prot.write\_i32(self.0)

}

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<DataType> {

let enum\_value = i\_prot.read\_i32()?;

Ok(DataType::from(enum\_value))

}

}

impl From<i32> for DataType {

fn from(i: i32) -> Self {

match i {

0 => DataType::FLOAT,

1 => DataType::DOUBLE,

2 => DataType::INT32,

3 => DataType::INT64,

4 => DataType::UINT8,

5 => DataType::STRING,

6 => DataType::BYTE,

7 => DataType::BOOL,

8 => DataType::RESERVED\_1,

9 => DataType::RESERVED\_2,

10 => DataType::RESERVED\_3,

\_ => DataType(i)

}

}

}

impl From<&i32> for DataType {

fn from(i: &i32) -> Self {

DataType::from(\*i)

}

}

impl From<DataType> for i32 {

fn from(e: DataType) -> i32 {

e.0

}

}

impl From<&DataType> for i32 {

fn from(e: &DataType) -> i32 {

e.0

}

}

//

// StringTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct StringTensor {

pub strings: Vec<String>,

pub shape: Option<Vec<i64>>,

}

impl StringTensor {

pub fn new<F2>(strings: Vec<String>, shape: F2) -> StringTensor where F2: Into<Option<Vec<i64>>> {

StringTensor {

strings,

shape: shape.into(),

}

}

}

impl TSerializable for StringTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<StringTensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<String>> = None;

let mut f\_2: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<String> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_0 = i\_prot.read\_string()?;

val.push(list\_elem\_0);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_1 = i\_prot.read\_i64()?;

val.push(list\_elem\_1);

}

i\_prot.read\_list\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("StringTensor.strings", &f\_1)?;

let ret = StringTensor {

strings: f\_1.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("StringTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("strings", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::String, self.strings.len() as i32))?;

for e in &self.strings {

o\_prot.write\_string(e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 2))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// Int32Tensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct Int32Tensor {

pub ints: Vec<i32>,

pub shape: Option<Vec<i64>>,

}

impl Int32Tensor {

pub fn new<F2>(ints: Vec<i32>, shape: F2) -> Int32Tensor where F2: Into<Option<Vec<i64>>> {

Int32Tensor {

ints,

shape: shape.into(),

}

}

}

impl TSerializable for Int32Tensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<Int32Tensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<i32>> = None;

let mut f\_2: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i32> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_2 = i\_prot.read\_i32()?;

val.push(list\_elem\_2);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_3 = i\_prot.read\_i64()?;

val.push(list\_elem\_3);

}

i\_prot.read\_list\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("Int32Tensor.ints", &f\_1)?;

let ret = Int32Tensor {

ints: f\_1.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("Int32Tensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("ints", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I32, self.ints.len() as i32))?;

for e in &self.ints {

o\_prot.write\_i32(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 2))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// Int64Tensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct Int64Tensor {

pub longs: Vec<i64>,

pub shape: Option<Vec<i64>>,

}

impl Int64Tensor {

pub fn new<F2>(longs: Vec<i64>, shape: F2) -> Int64Tensor where F2: Into<Option<Vec<i64>>> {

Int64Tensor {

longs,

shape: shape.into(),

}

}

}

impl TSerializable for Int64Tensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<Int64Tensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<i64>> = None;

let mut f\_2: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_4 = i\_prot.read\_i64()?;

val.push(list\_elem\_4);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_5 = i\_prot.read\_i64()?;

val.push(list\_elem\_5);

}

i\_prot.read\_list\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("Int64Tensor.longs", &f\_1)?;

let ret = Int64Tensor {

longs: f\_1.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("Int64Tensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("longs", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, self.longs.len() as i32))?;

for e in &self.longs {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 2))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// FloatTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct FloatTensor {

pub floats: Vec<OrderedFloat<f64>>,

pub shape: Option<Vec<i64>>,

}

impl FloatTensor {

pub fn new<F2>(floats: Vec<OrderedFloat<f64>>, shape: F2) -> FloatTensor where F2: Into<Option<Vec<i64>>> {

FloatTensor {

floats,

shape: shape.into(),

}

}

}

impl TSerializable for FloatTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<FloatTensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<OrderedFloat<f64>>> = None;

let mut f\_2: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<OrderedFloat<f64>> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_6 = OrderedFloat::from(i\_prot.read\_double()?);

val.push(list\_elem\_6);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_7 = i\_prot.read\_i64()?;

val.push(list\_elem\_7);

}

i\_prot.read\_list\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("FloatTensor.floats", &f\_1)?;

let ret = FloatTensor {

floats: f\_1.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("FloatTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("floats", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::Double, self.floats.len() as i32))?;

for e in &self.floats {

o\_prot.write\_double((\*e).into())?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 2))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// DoubleTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct DoubleTensor {

pub doubles: Vec<OrderedFloat<f64>>,

pub shape: Option<Vec<i64>>,

}

impl DoubleTensor {

pub fn new<F2>(doubles: Vec<OrderedFloat<f64>>, shape: F2) -> DoubleTensor where F2: Into<Option<Vec<i64>>> {

DoubleTensor {

doubles,

shape: shape.into(),

}

}

}

impl TSerializable for DoubleTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<DoubleTensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<OrderedFloat<f64>>> = None;

let mut f\_2: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<OrderedFloat<f64>> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_8 = OrderedFloat::from(i\_prot.read\_double()?);

val.push(list\_elem\_8);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_9 = i\_prot.read\_i64()?;

val.push(list\_elem\_9);

}

i\_prot.read\_list\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("DoubleTensor.doubles", &f\_1)?;

let ret = DoubleTensor {

doubles: f\_1.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("DoubleTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("doubles", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::Double, self.doubles.len() as i32))?;

for e in &self.doubles {

o\_prot.write\_double((\*e).into())?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 2))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// BoolTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct BoolTensor {

pub booleans: Vec<bool>,

pub shape: Option<Vec<i64>>,

}

impl BoolTensor {

pub fn new<F2>(booleans: Vec<bool>, shape: F2) -> BoolTensor where F2: Into<Option<Vec<i64>>> {

BoolTensor {

booleans,

shape: shape.into(),

}

}

}

impl TSerializable for BoolTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<BoolTensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<bool>> = None;

let mut f\_2: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<bool> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_10 = i\_prot.read\_bool()?;

val.push(list\_elem\_10);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_11 = i\_prot.read\_i64()?;

val.push(list\_elem\_11);

}

i\_prot.read\_list\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("BoolTensor.booleans", &f\_1)?;

let ret = BoolTensor {

booleans: f\_1.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("BoolTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("booleans", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::Bool, self.booleans.len() as i32))?;

for e in &self.booleans {

o\_prot.write\_bool(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 2))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// RawTypedTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct RawTypedTensor {

pub data\_type: DataType,

pub content: Vec<u8>,

pub shape: Option<Vec<i64>>,

}

impl RawTypedTensor {

pub fn new<F3>(data\_type: DataType, content: Vec<u8>, shape: F3) -> RawTypedTensor where F3: Into<Option<Vec<i64>>> {

RawTypedTensor {

data\_type,

content,

shape: shape.into(),

}

}

}

impl TSerializable for RawTypedTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<RawTypedTensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<DataType> = None;

let mut f\_2: Option<Vec<u8>> = None;

let mut f\_3: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let val = DataType::read\_from\_in\_protocol(i\_prot)?;

f\_1 = Some(val);

},

2 => {

let val = i\_prot.read\_bytes()?;

f\_2 = Some(val);

},

3 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_12 = i\_prot.read\_i64()?;

val.push(list\_elem\_12);

}

i\_prot.read\_list\_end()?;

f\_3 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("RawTypedTensor.data\_type", &f\_1)?;

verify\_required\_field\_exists("RawTypedTensor.content", &f\_2)?;

let ret = RawTypedTensor {

data\_type: f\_1.expect("auto-generated code should have checked for presence of required fields"),

content: f\_2.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_3,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("RawTypedTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("dataType", TType::I32, 1))?;

self.data\_type.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("content", TType::String, 2))?;

o\_prot.write\_bytes(&self.content)?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 3))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// BinaryTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct BinaryTensor {

pub binaries: Vec<Vec<u8>>,

pub shape: Option<Vec<i64>>,

}

impl BinaryTensor {

pub fn new<F2>(binaries: Vec<Vec<u8>>, shape: F2) -> BinaryTensor where F2: Into<Option<Vec<i64>>> {

BinaryTensor {

binaries,

shape: shape.into(),

}

}

}

impl TSerializable for BinaryTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<BinaryTensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<Vec<u8>>> = None;

let mut f\_2: Option<Vec<i64>> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<Vec<u8>> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_13 = i\_prot.read\_bytes()?;

val.push(list\_elem\_13);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_14 = i\_prot.read\_i64()?;

val.push(list\_elem\_14);

}

i\_prot.read\_list\_end()?;

f\_2 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("BinaryTensor.binaries", &f\_1)?;

let ret = BinaryTensor {

binaries: f\_1.expect("auto-generated code should have checked for presence of required fields"),

shape: f\_2,

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("BinaryTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("binaries", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::String, self.binaries.len() as i32))?;

for e in &self.binaries {

o\_prot.write\_bytes(e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

if let Some(ref fld\_var) = self.shape {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("shape", TType::List, 2))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, fld\_var.len() as i32))?;

for e in fld\_var {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// GeneralTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub enum GeneralTensor {

RawTypedTensor(RawTypedTensor),

StringTensor(StringTensor),

Int32Tensor(Int32Tensor),

Int64Tensor(Int64Tensor),

FloatTensor(FloatTensor),

DoubleTensor(DoubleTensor),

BoolTensor(BoolTensor),

BinaryTensor(BinaryTensor),

}

impl TSerializable for GeneralTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<GeneralTensor> {

let mut ret: Option<GeneralTensor> = None;

let mut received\_field\_count = 0;

i\_prot.read\_struct\_begin()?;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let val = RawTypedTensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::RawTypedTensor(val));

}

received\_field\_count += 1;

},

2 => {

let val = StringTensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::StringTensor(val));

}

received\_field\_count += 1;

},

3 => {

let val = Int32Tensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::Int32Tensor(val));

}

received\_field\_count += 1;

},

4 => {

let val = Int64Tensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::Int64Tensor(val));

}

received\_field\_count += 1;

},

5 => {

let val = FloatTensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::FloatTensor(val));

}

received\_field\_count += 1;

},

6 => {

let val = DoubleTensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::DoubleTensor(val));

}

received\_field\_count += 1;

},

7 => {

let val = BoolTensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::BoolTensor(val));

}

received\_field\_count += 1;

},

8 => {

let val = BinaryTensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(GeneralTensor::BinaryTensor(val));

}

received\_field\_count += 1;

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

received\_field\_count += 1;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

if received\_field\_count == 0 {

Err(

thrift::Error::Protocol(

ProtocolError::new(

ProtocolErrorKind::InvalidData,

"received empty union from remote GeneralTensor"

)

)

)

} else if received\_field\_count > 1 {

Err(

thrift::Error::Protocol(

ProtocolError::new(

ProtocolErrorKind::InvalidData,

"received multiple fields for union from remote GeneralTensor"

)

)

)

} else {

Ok(ret.expect("return value should have been constructed"))

}

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("GeneralTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

match \*self {

GeneralTensor::RawTypedTensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("rawTypedTensor", TType::Struct, 1))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

GeneralTensor::StringTensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("stringTensor", TType::Struct, 2))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

GeneralTensor::Int32Tensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("int32Tensor", TType::Struct, 3))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

GeneralTensor::Int64Tensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("int64Tensor", TType::Struct, 4))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

GeneralTensor::FloatTensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("floatTensor", TType::Struct, 5))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

GeneralTensor::DoubleTensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("doubleTensor", TType::Struct, 6))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

GeneralTensor::BoolTensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("boolTensor", TType::Struct, 7))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

GeneralTensor::BinaryTensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("binaryTensor", TType::Struct, 8))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// COOSparseTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub struct COOSparseTensor {

pub dense\_shape: Vec<i64>,

pub indices: Int64Tensor,

pub values: GeneralTensor,

}

impl COOSparseTensor {

pub fn new(dense\_shape: Vec<i64>, indices: Int64Tensor, values: GeneralTensor) -> COOSparseTensor {

COOSparseTensor {

dense\_shape,

indices,

values,

}

}

}

impl TSerializable for COOSparseTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<COOSparseTensor> {

i\_prot.read\_struct\_begin()?;

let mut f\_1: Option<Vec<i64>> = None;

let mut f\_2: Option<Int64Tensor> = None;

let mut f\_3: Option<GeneralTensor> = None;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let list\_ident = i\_prot.read\_list\_begin()?;

let mut val: Vec<i64> = Vec::with\_capacity(list\_ident.size as usize);

for \_ in 0..list\_ident.size {

let list\_elem\_15 = i\_prot.read\_i64()?;

val.push(list\_elem\_15);

}

i\_prot.read\_list\_end()?;

f\_1 = Some(val);

},

2 => {

let val = Int64Tensor::read\_from\_in\_protocol(i\_prot)?;

f\_2 = Some(val);

},

3 => {

let val = GeneralTensor::read\_from\_in\_protocol(i\_prot)?;

f\_3 = Some(val);

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

verify\_required\_field\_exists("COOSparseTensor.dense\_shape", &f\_1)?;

verify\_required\_field\_exists("COOSparseTensor.indices", &f\_2)?;

verify\_required\_field\_exists("COOSparseTensor.values", &f\_3)?;

let ret = COOSparseTensor {

dense\_shape: f\_1.expect("auto-generated code should have checked for presence of required fields"),

indices: f\_2.expect("auto-generated code should have checked for presence of required fields"),

values: f\_3.expect("auto-generated code should have checked for presence of required fields"),

};

Ok(ret)

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("COOSparseTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("denseShape", TType::List, 1))?;

o\_prot.write\_list\_begin(&TListIdentifier::new(TType::I64, self.dense\_shape.len() as i32))?;

for e in &self.dense\_shape {

o\_prot.write\_i64(\*e)?;

}

o\_prot.write\_list\_end()?;

o\_prot.write\_field\_end()?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("indices", TType::Struct, 2))?;

self.indices.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

o\_prot.write\_field\_begin(&TFieldIdentifier::new("values", TType::Struct, 3))?;

self.values.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}

//

// SparseTensor

//

#[derive(Clone, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]

pub enum SparseTensor {

CooSparseTensor(COOSparseTensor),

}

impl TSerializable for SparseTensor {

fn read\_from\_in\_protocol(i\_prot: &mut dyn TInputProtocol) -> thrift::Result<SparseTensor> {

let mut ret: Option<SparseTensor> = None;

let mut received\_field\_count = 0;

i\_prot.read\_struct\_begin()?;

loop {

let field\_ident = i\_prot.read\_field\_begin()?;

if field\_ident.field\_type == TType::Stop {

break;

}

let field\_id = field\_id(&field\_ident)?;

match field\_id {

1 => {

let val = COOSparseTensor::read\_from\_in\_protocol(i\_prot)?;

if ret.is\_none() {

ret = Some(SparseTensor::CooSparseTensor(val));

}

received\_field\_count += 1;

},

\_ => {

i\_prot.skip(field\_ident.field\_type)?;

received\_field\_count += 1;

},

};

i\_prot.read\_field\_end()?;

}

i\_prot.read\_struct\_end()?;

if received\_field\_count == 0 {

Err(

thrift::Error::Protocol(

ProtocolError::new(

ProtocolErrorKind::InvalidData,

"received empty union from remote SparseTensor"

)

)

)

} else if received\_field\_count > 1 {

Err(

thrift::Error::Protocol(

ProtocolError::new(

ProtocolErrorKind::InvalidData,

"received multiple fields for union from remote SparseTensor"

)

)

)

} else {

Ok(ret.expect("return value should have been constructed"))

}

}

fn write\_to\_out\_protocol(&self, o\_prot: &mut dyn TOutputProtocol) -> thrift::Result<()> {

let struct\_ident = TStructIdentifier::new("SparseTensor");

o\_prot.write\_struct\_begin(&struct\_ident)?;

match \*self {

SparseTensor::CooSparseTensor(ref f) => {

o\_prot.write\_field\_begin(&TFieldIdentifier::new("cooSparseTensor", TType::Struct, 1))?;

f.write\_to\_out\_protocol(o\_prot)?;

o\_prot.write\_field\_end()?;

},

}

o\_prot.write\_field\_stop()?;

o\_prot.write\_struct\_end()

}

}