// Licensed to the Apache Software Foundation (ASF) under one

// or more contributor license agreements. See the NOTICE file

// distributed with this work for additional information

// regarding copyright ownership. The ASF licenses this file

// to you under the Apache License, Version 2.0 (the

// "License"); you may not use this file except in compliance

// with the License. You may obtain a copy of the License at

//

// http://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing,

// software distributed under the License is distributed on an

// "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

// KIND, either express or implied. See the License for the

// specific language governing permissions and limitations

// under the License.

#include "arrow/python/common.h"

#include <cstdlib>

#include <mutex>

#include <sstream>

#include <string>

#include "arrow/memory\_pool.h"

#include "arrow/status.h"

#include "arrow/util/checked\_cast.h"

#include "arrow/util/logging.h"

#include "arrow/python/helpers.h"

namespace arrow {

using internal::checked\_cast;

namespace py {

static std::mutex memory\_pool\_mutex;

static MemoryPool\* default\_python\_pool = nullptr;

void set\_default\_memory\_pool(MemoryPool\* pool) {

std::lock\_guard<std::mutex> guard(memory\_pool\_mutex);

default\_python\_pool = pool;

}

MemoryPool\* get\_memory\_pool() {

std::lock\_guard<std::mutex> guard(memory\_pool\_mutex);

if (default\_python\_pool) {

return default\_python\_pool;

} else {

return default\_memory\_pool();

}

}

// ----------------------------------------------------------------------

// PythonErrorDetail

namespace {

const char kErrorDetailTypeId[] = "arrow::py::PythonErrorDetail";

// Try to match the Python exception type with an appropriate Status code

StatusCode MapPyError(PyObject\* exc\_type) {

StatusCode code;

if (PyErr\_GivenExceptionMatches(exc\_type, PyExc\_MemoryError)) {

code = StatusCode::OutOfMemory;

} else if (PyErr\_GivenExceptionMatches(exc\_type, PyExc\_IndexError)) {

code = StatusCode::IndexError;

} else if (PyErr\_GivenExceptionMatches(exc\_type, PyExc\_KeyError)) {

code = StatusCode::KeyError;

} else if (PyErr\_GivenExceptionMatches(exc\_type, PyExc\_TypeError)) {

code = StatusCode::TypeError;

} else if (PyErr\_GivenExceptionMatches(exc\_type, PyExc\_ValueError) ||

PyErr\_GivenExceptionMatches(exc\_type, PyExc\_OverflowError)) {

code = StatusCode::Invalid;

} else if (PyErr\_GivenExceptionMatches(exc\_type, PyExc\_EnvironmentError)) {

code = StatusCode::IOError;

} else if (PyErr\_GivenExceptionMatches(exc\_type, PyExc\_NotImplementedError)) {

code = StatusCode::NotImplemented;

} else {

code = StatusCode::UnknownError;

}

return code;

}

// PythonErrorDetail indicates a Python exception was raised.

class PythonErrorDetail : public StatusDetail {

public:

const char\* type\_id() const override { return kErrorDetailTypeId; }

std::string ToString() const override {

// This is simple enough not to need the GIL

Result<std::string> result = FormatImpl();

if (result.ok()) {

return result.ValueOrDie();

} else {

// Fallback to just the exception type

const auto ty = reinterpret\_cast<const PyTypeObject\*>(exc\_type\_.obj());

return std::string("Python exception: ") + ty->tp\_name;

}

}

void RestorePyError() const {

Py\_INCREF(exc\_type\_.obj());

Py\_INCREF(exc\_value\_.obj());

Py\_INCREF(exc\_traceback\_.obj());

PyErr\_Restore(exc\_type\_.obj(), exc\_value\_.obj(), exc\_traceback\_.obj());

}

PyObject\* exc\_type() const { return exc\_type\_.obj(); }

PyObject\* exc\_value() const { return exc\_value\_.obj(); }

static std::shared\_ptr<PythonErrorDetail> FromPyError() {

PyObject\* exc\_type = nullptr;

PyObject\* exc\_value = nullptr;

PyObject\* exc\_traceback = nullptr;

PyErr\_Fetch(&exc\_type, &exc\_value, &exc\_traceback);

PyErr\_NormalizeException(&exc\_type, &exc\_value, &exc\_traceback);

ARROW\_CHECK(exc\_type)

<< "PythonErrorDetail::FromPyError called without a Python error set";

DCHECK(PyType\_Check(exc\_type));

DCHECK(exc\_value); // Ensured by PyErr\_NormalizeException, double-check

if (exc\_traceback == nullptr) {

// Needed by PyErr\_Restore()

Py\_INCREF(Py\_None);

exc\_traceback = Py\_None;

}

std::shared\_ptr<PythonErrorDetail> detail(new PythonErrorDetail);

detail->exc\_type\_.reset(exc\_type);

detail->exc\_value\_.reset(exc\_value);

detail->exc\_traceback\_.reset(exc\_traceback);

return detail;

}

protected:

Result<std::string> FormatImpl() const {

PyAcquireGIL lock;

// Use traceback.format\_exception()

OwnedRef traceback\_module;

RETURN\_NOT\_OK(internal::ImportModule("traceback", &traceback\_module));

OwnedRef fmt\_exception;

RETURN\_NOT\_OK(internal::ImportFromModule(traceback\_module.obj(), "format\_exception",

&fmt\_exception));

OwnedRef formatted;

formatted.reset(PyObject\_CallFunctionObjArgs(fmt\_exception.obj(), exc\_type\_.obj(),

exc\_value\_.obj(), exc\_traceback\_.obj(),

NULL));

RETURN\_IF\_PYERROR();

std::stringstream ss;

ss << "Python exception: ";

Py\_ssize\_t num\_lines = PySequence\_Length(formatted.obj());

RETURN\_IF\_PYERROR();

for (Py\_ssize\_t i = 0; i < num\_lines; ++i) {

Py\_ssize\_t line\_size;

PyObject\* line = PySequence\_GetItem(formatted.obj(), i);

RETURN\_IF\_PYERROR();

const char\* data = PyUnicode\_AsUTF8AndSize(line, &line\_size);

RETURN\_IF\_PYERROR();

ss << std::string\_view(data, line\_size);

}

return ss.str();

}

PythonErrorDetail() = default;

OwnedRefNoGIL exc\_type\_, exc\_value\_, exc\_traceback\_;

};

} // namespace

// ----------------------------------------------------------------------

// Python exception <-> Status

Status ConvertPyError(StatusCode code) {

auto detail = PythonErrorDetail::FromPyError();

if (code == StatusCode::UnknownError) {

code = MapPyError(detail->exc\_type());

}

std::string message;

RETURN\_NOT\_OK(internal::PyObject\_StdStringStr(detail->exc\_value(), &message));

return Status(code, message, detail);

}

bool IsPyError(const Status& status) {

if (status.ok()) {

return false;

}

auto detail = status.detail();

bool result = detail != nullptr && detail->type\_id() == kErrorDetailTypeId;

return result;

}

void RestorePyError(const Status& status) {

ARROW\_CHECK(IsPyError(status));

const auto& detail = checked\_cast<const PythonErrorDetail&>(\*status.detail());

detail.RestorePyError();

}

// ----------------------------------------------------------------------

// PyBuffer

PyBuffer::PyBuffer() : Buffer(nullptr, 0) {}

Status PyBuffer::Init(PyObject\* obj) {

if (!PyObject\_GetBuffer(obj, &py\_buf\_, PyBUF\_ANY\_CONTIGUOUS)) {

data\_ = reinterpret\_cast<const uint8\_t\*>(py\_buf\_.buf);

ARROW\_CHECK\_NE(data\_, nullptr) << "Null pointer in Py\_buffer";

size\_ = py\_buf\_.len;

capacity\_ = py\_buf\_.len;

is\_mutable\_ = !py\_buf\_.readonly;

return Status::OK();

} else {

return ConvertPyError(StatusCode::Invalid);

}

}

Result<std::shared\_ptr<Buffer>> PyBuffer::FromPyObject(PyObject\* obj) {

PyBuffer\* buf = new PyBuffer();

std::shared\_ptr<Buffer> res(buf);

RETURN\_NOT\_OK(buf->Init(obj));

return res;

}

PyBuffer::~PyBuffer() {

if (data\_ != nullptr) {

PyAcquireGIL lock;

PyBuffer\_Release(&py\_buf\_);

}

}

} // namespace py

} // namespace arrow