// Copyright 2013 The Flutter Authors. All rights reserved.

// Use of this source code is governed by a BSD-style license that can be

// found in the LICENSE file.

// This file contains what would normally be standard\_codec\_serializer.cc,

// standard\_message\_codec.cc, and standard\_method\_codec.cc. They are grouped

// together to simplify use of the client wrapper, since the common case is

// that any client that needs one of these files needs all three.

#include <cassert>

#include <cstring>

#include <iostream>

#include <map>

#include <string>

#include <vector>

#include "byte\_buffer\_streams.h"

#include "include/flutter/standard\_codec\_serializer.h"

#include "include/flutter/standard\_message\_codec.h"

#include "include/flutter/standard\_method\_codec.h"

namespace flutter {

// ===== standard\_codec\_serializer.h =====

namespace {

// The order/values here must match the constants in message\_codecs.dart.

enum class EncodedType {

kNull = 0,

kTrue,

kFalse,

kInt32,

kInt64,

kLargeInt, // No longer used. If encountered, treat as kString.

kFloat64,

kString,

kUInt8List,

kInt32List,

kInt64List,

kFloat64List,

kList,

kMap,

kFloat32List,

};

// Returns the encoded type that should be written when serializing |value|.

EncodedType EncodedTypeForValue(const EncodableValue& value) {

switch (value.index()) {

case 0:

return EncodedType::kNull;

case 1:

return std::get<bool>(value) ? EncodedType::kTrue : EncodedType::kFalse;

case 2:

return EncodedType::kInt32;

case 3:

return EncodedType::kInt64;

case 4:

return EncodedType::kFloat64;

case 5:

return EncodedType::kString;

case 6:

return EncodedType::kUInt8List;

case 7:

return EncodedType::kInt32List;

case 8:

return EncodedType::kInt64List;

case 9:

return EncodedType::kFloat64List;

case 10:

return EncodedType::kList;

case 11:

return EncodedType::kMap;

case 13:

return EncodedType::kFloat32List;

}

assert(false);

return EncodedType::kNull;

}

} // namespace

StandardCodecSerializer::StandardCodecSerializer() = default;

StandardCodecSerializer::~StandardCodecSerializer() = default;

const StandardCodecSerializer& StandardCodecSerializer::GetInstance() {

static StandardCodecSerializer sInstance;

return sInstance;

};

EncodableValue StandardCodecSerializer::ReadValue(

ByteStreamReader\* stream) const {

uint8\_t type = stream->ReadByte();

return ReadValueOfType(type, stream);

}

void StandardCodecSerializer::WriteValue(const EncodableValue& value,

ByteStreamWriter\* stream) const {

stream->WriteByte(static\_cast<uint8\_t>(EncodedTypeForValue(value)));

// TODO(cbracken): Consider replacing this with std::visit.

switch (value.index()) {

case 0:

case 1:

// Null and bool are encoded directly in the type.

break;

case 2:

stream->WriteInt32(std::get<int32\_t>(value));

break;

case 3:

stream->WriteInt64(std::get<int64\_t>(value));

break;

case 4:

stream->WriteAlignment(8);

stream->WriteDouble(std::get<double>(value));

break;

case 5: {

const auto& string\_value = std::get<std::string>(value);

size\_t size = string\_value.size();

WriteSize(size, stream);

if (size > 0) {

stream->WriteBytes(

reinterpret\_cast<const uint8\_t\*>(string\_value.data()), size);

}

break;

}

case 6:

WriteVector(std::get<std::vector<uint8\_t>>(value), stream);

break;

case 7:

WriteVector(std::get<std::vector<int32\_t>>(value), stream);

break;

case 8:

WriteVector(std::get<std::vector<int64\_t>>(value), stream);

break;

case 9:

WriteVector(std::get<std::vector<double>>(value), stream);

break;

case 10: {

const auto& list = std::get<EncodableList>(value);

WriteSize(list.size(), stream);

for (const auto& item : list) {

WriteValue(item, stream);

}

break;

}

case 11: {

const auto& map = std::get<EncodableMap>(value);

WriteSize(map.size(), stream);

for (const auto& pair : map) {

WriteValue(pair.first, stream);

WriteValue(pair.second, stream);

}

break;

}

case 12:

std::cerr

<< "Unhandled custom type in StandardCodecSerializer::WriteValue. "

<< "Custom types require codec extensions." << std::endl;

break;

case 13: {

WriteVector(std::get<std::vector<float>>(value), stream);

break;

}

}

}

EncodableValue StandardCodecSerializer::ReadValueOfType(

uint8\_t type,

ByteStreamReader\* stream) const {

switch (static\_cast<EncodedType>(type)) {

case EncodedType::kNull:

return EncodableValue();

case EncodedType::kTrue:

return EncodableValue(true);

case EncodedType::kFalse:

return EncodableValue(false);

case EncodedType::kInt32:

return EncodableValue(stream->ReadInt32());

case EncodedType::kInt64:

return EncodableValue(stream->ReadInt64());

case EncodedType::kFloat64:

stream->ReadAlignment(8);

return EncodableValue(stream->ReadDouble());

case EncodedType::kLargeInt:

case EncodedType::kString: {

size\_t size = ReadSize(stream);

std::string string\_value;

string\_value.resize(size);

stream->ReadBytes(reinterpret\_cast<uint8\_t\*>(&string\_value[0]), size);

return EncodableValue(string\_value);

}

case EncodedType::kUInt8List:

return ReadVector<uint8\_t>(stream);

case EncodedType::kInt32List:

return ReadVector<int32\_t>(stream);

case EncodedType::kInt64List:

return ReadVector<int64\_t>(stream);

case EncodedType::kFloat64List:

return ReadVector<double>(stream);

case EncodedType::kList: {

size\_t length = ReadSize(stream);

EncodableList list\_value;

list\_value.reserve(length);

for (size\_t i = 0; i < length; ++i) {

list\_value.push\_back(ReadValue(stream));

}

return EncodableValue(list\_value);

}

case EncodedType::kMap: {

size\_t length = ReadSize(stream);

EncodableMap map\_value;

for (size\_t i = 0; i < length; ++i) {

EncodableValue key = ReadValue(stream);

EncodableValue value = ReadValue(stream);

map\_value.emplace(std::move(key), std::move(value));

}

return EncodableValue(map\_value);

}

case EncodedType::kFloat32List: {

return ReadVector<float>(stream);

}

}

std::cerr << "Unknown type in StandardCodecSerializer::ReadValueOfType: "

<< static\_cast<int>(type) << std::endl;

return EncodableValue();

}

size\_t StandardCodecSerializer::ReadSize(ByteStreamReader\* stream) const {

uint8\_t byte = stream->ReadByte();

if (byte < 254) {

return byte;

} else if (byte == 254) {

uint16\_t value = 0;

stream->ReadBytes(reinterpret\_cast<uint8\_t\*>(&value), 2);

return value;

} else {

uint32\_t value = 0;

stream->ReadBytes(reinterpret\_cast<uint8\_t\*>(&value), 4);

return value;

}

}

void StandardCodecSerializer::WriteSize(size\_t size,

ByteStreamWriter\* stream) const {

if (size < 254) {

stream->WriteByte(static\_cast<uint8\_t>(size));

} else if (size <= 0xffff) {

stream->WriteByte(254);

uint16\_t value = static\_cast<uint16\_t>(size);

stream->WriteBytes(reinterpret\_cast<uint8\_t\*>(&value), 2);

} else {

stream->WriteByte(255);

uint32\_t value = static\_cast<uint32\_t>(size);

stream->WriteBytes(reinterpret\_cast<uint8\_t\*>(&value), 4);

}

}

template <typename T>

EncodableValue StandardCodecSerializer::ReadVector(

ByteStreamReader\* stream) const {

size\_t count = ReadSize(stream);

std::vector<T> vector;

vector.resize(count);

uint8\_t type\_size = static\_cast<uint8\_t>(sizeof(T));

if (type\_size > 1) {

stream->ReadAlignment(type\_size);

}

stream->ReadBytes(reinterpret\_cast<uint8\_t\*>(vector.data()),

count \* type\_size);

return EncodableValue(vector);

}

template <typename T>

void StandardCodecSerializer::WriteVector(const std::vector<T> vector,

ByteStreamWriter\* stream) const {

size\_t count = vector.size();

WriteSize(count, stream);

if (count == 0) {

return;

}

uint8\_t type\_size = static\_cast<uint8\_t>(sizeof(T));

if (type\_size > 1) {

stream->WriteAlignment(type\_size);

}

stream->WriteBytes(reinterpret\_cast<const uint8\_t\*>(vector.data()),

count \* type\_size);

}

// ===== standard\_message\_codec.h =====

// static

const StandardMessageCodec& StandardMessageCodec::GetInstance(

const StandardCodecSerializer\* serializer) {

if (!serializer) {

serializer = &StandardCodecSerializer::GetInstance();

}

static auto\* sInstances = new std::map<const StandardCodecSerializer\*,

std::unique\_ptr<StandardMessageCodec>>;

auto it = sInstances->find(serializer);

if (it == sInstances->end()) {

// Uses new due to private constructor (to prevent API clients from

// accidentally passing temporary codec instances to channels).

auto emplace\_result = sInstances->emplace(

serializer, std::unique\_ptr<StandardMessageCodec>(

new StandardMessageCodec(serializer)));

it = emplace\_result.first;

}

return \*(it->second);

}

StandardMessageCodec::StandardMessageCodec(

const StandardCodecSerializer\* serializer)

: serializer\_(serializer) {}

StandardMessageCodec::~StandardMessageCodec() = default;

std::unique\_ptr<EncodableValue> StandardMessageCodec::DecodeMessageInternal(

const uint8\_t\* binary\_message,

size\_t message\_size) const {

if (!binary\_message) {

return std::make\_unique<EncodableValue>();

}

ByteBufferStreamReader stream(binary\_message, message\_size);

return std::make\_unique<EncodableValue>(serializer\_->ReadValue(&stream));

}

std::unique\_ptr<std::vector<uint8\_t>>

StandardMessageCodec::EncodeMessageInternal(

const EncodableValue& message) const {

auto encoded = std::make\_unique<std::vector<uint8\_t>>();

ByteBufferStreamWriter stream(encoded.get());

serializer\_->WriteValue(message, &stream);

return encoded;

}

// ===== standard\_method\_codec.h =====

// static

const StandardMethodCodec& StandardMethodCodec::GetInstance(

const StandardCodecSerializer\* serializer) {

if (!serializer) {

serializer = &StandardCodecSerializer::GetInstance();

}

static auto\* sInstances = new std::map<const StandardCodecSerializer\*,

std::unique\_ptr<StandardMethodCodec>>;

auto it = sInstances->find(serializer);

if (it == sInstances->end()) {

// Uses new due to private constructor (to prevent API clients from

// accidentally passing temporary codec instances to channels).

auto emplace\_result = sInstances->emplace(

serializer, std::unique\_ptr<StandardMethodCodec>(

new StandardMethodCodec(serializer)));

it = emplace\_result.first;

}

return \*(it->second);

}

StandardMethodCodec::StandardMethodCodec(

const StandardCodecSerializer\* serializer)

: serializer\_(serializer) {}

StandardMethodCodec::~StandardMethodCodec() = default;

std::unique\_ptr<MethodCall<EncodableValue>>

StandardMethodCodec::DecodeMethodCallInternal(const uint8\_t\* message,

size\_t message\_size) const {

ByteBufferStreamReader stream(message, message\_size);

EncodableValue method\_name\_value = serializer\_->ReadValue(&stream);

const auto\* method\_name = std::get\_if<std::string>(&method\_name\_value);

if (!method\_name) {

std::cerr << "Invalid method call; method name is not a string."

<< std::endl;

return nullptr;

}

auto arguments =

std::make\_unique<EncodableValue>(serializer\_->ReadValue(&stream));

return std::make\_unique<MethodCall<EncodableValue>>(\*method\_name,

std::move(arguments));

}

std::unique\_ptr<std::vector<uint8\_t>>

StandardMethodCodec::EncodeMethodCallInternal(

const MethodCall<EncodableValue>& method\_call) const {

auto encoded = std::make\_unique<std::vector<uint8\_t>>();

ByteBufferStreamWriter stream(encoded.get());

serializer\_->WriteValue(EncodableValue(method\_call.method\_name()), &stream);

if (method\_call.arguments()) {

serializer\_->WriteValue(\*method\_call.arguments(), &stream);

} else {

serializer\_->WriteValue(EncodableValue(), &stream);

}

return encoded;

}

std::unique\_ptr<std::vector<uint8\_t>>

StandardMethodCodec::EncodeSuccessEnvelopeInternal(

const EncodableValue\* result) const {

auto encoded = std::make\_unique<std::vector<uint8\_t>>();

ByteBufferStreamWriter stream(encoded.get());

stream.WriteByte(0);

if (result) {

serializer\_->WriteValue(\*result, &stream);

} else {

serializer\_->WriteValue(EncodableValue(), &stream);

}

return encoded;

}

std::unique\_ptr<std::vector<uint8\_t>>

StandardMethodCodec::EncodeErrorEnvelopeInternal(

const std::string& error\_code,

const std::string& error\_message,

const EncodableValue\* error\_details) const {

auto encoded = std::make\_unique<std::vector<uint8\_t>>();

ByteBufferStreamWriter stream(encoded.get());

stream.WriteByte(1);

serializer\_->WriteValue(EncodableValue(error\_code), &stream);

if (error\_message.empty()) {

serializer\_->WriteValue(EncodableValue(), &stream);

} else {

serializer\_->WriteValue(EncodableValue(error\_message), &stream);

}

if (error\_details) {

serializer\_->WriteValue(\*error\_details, &stream);

} else {

serializer\_->WriteValue(EncodableValue(), &stream);

}

return encoded;

}

bool StandardMethodCodec::DecodeAndProcessResponseEnvelopeInternal(

const uint8\_t\* response,

size\_t response\_size,

MethodResult<EncodableValue>\* result) const {

ByteBufferStreamReader stream(response, response\_size);

uint8\_t flag = stream.ReadByte();

switch (flag) {

case 0: {

EncodableValue value = serializer\_->ReadValue(&stream);

if (value.IsNull()) {

result->Success();

} else {

result->Success(value);

}

return true;

}

case 1: {

EncodableValue code = serializer\_->ReadValue(&stream);

EncodableValue message = serializer\_->ReadValue(&stream);

EncodableValue details = serializer\_->ReadValue(&stream);

const std::string& message\_string =

message.IsNull() ? "" : std::get<std::string>(message);

if (details.IsNull()) {

result->Error(std::get<std::string>(code), message\_string);

} else {

result->Error(std::get<std::string>(code), message\_string, details);

}

return true;

}

default:

return false;

}

}

} // namespace flutter