// 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.

#ifndef FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_ENCODABLE\_VALUE\_H\_

#define FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_ENCODABLE\_VALUE\_H\_

#include <any>

#include <cassert>

#include <cstdint>

#include <map>

#include <string>

#include <utility>

#include <variant>

#include <vector>

// Unless overridden, attempt to detect the RTTI state from the compiler.

#ifndef FLUTTER\_ENABLE\_RTTI

#if defined(\_MSC\_VER)

#ifdef \_CPPRTTI

#define FLUTTER\_ENABLE\_RTTI 1

#endif

#elif defined(\_\_clang\_\_)

#if \_\_has\_feature(cxx\_rtti)

#define FLUTTER\_ENABLE\_RTTI 1

#endif

#elif defined(\_\_GNUC\_\_)

#ifdef \_\_GXX\_RTTI

#define FLUTTER\_ENABLE\_RTTI 1

#endif

#endif

#endif // #ifndef FLUTTER\_ENABLE\_RTTI

namespace flutter {

static\_assert(sizeof(double) == 8, "EncodableValue requires a 64-bit double");

// A container for arbitrary types in EncodableValue.

//

// This is used in conjunction with StandardCodecExtension to allow using other

// types with a StandardMethodCodec/StandardMessageCodec. It is implicitly

// convertible to EncodableValue, so constructing an EncodableValue from a

// custom type can generally be written as:

// CustomEncodableValue(MyType(...))

// rather than:

// EncodableValue(CustomEncodableValue(MyType(...)))

//

// For extracting received custom types, it is implicitly convertible to

// std::any. For example:

// const MyType& my\_type\_value =

// std::any\_cast<MyType>(std::get<CustomEncodableValue>(value));

//

// If RTTI is enabled, different extension types can be checked with type():

// if (custom\_value->type() == typeid(SomeData)) { ... }

// Clients that wish to disable RTTI would need to decide on another approach

// for distinguishing types (e.g., in StandardCodecExtension::WriteValueOfType)

// if multiple custom types are needed. For instance, wrapping all of the

// extension types in an EncodableValue-style variant, and only ever storing

// that variant in CustomEncodableValue.

class CustomEncodableValue {

public:

explicit CustomEncodableValue(const std::any& value) : value\_(value) {}

~CustomEncodableValue() = default;

// Allow implicit conversion to std::any to allow direct use of any\_cast.

// NOLINTNEXTLINE(google-explicit-constructor)

operator std::any&() { return value\_; }

// NOLINTNEXTLINE(google-explicit-constructor)

operator const std::any&() const { return value\_; }

#if defined(FLUTTER\_ENABLE\_RTTI) && FLUTTER\_ENABLE\_RTTI

// Passthrough to std::any's type().

const std::type\_info& type() const noexcept { return value\_.type(); }

#endif

// This operator exists only to provide a stable ordering for use as a

// std::map key, to satisfy the compiler requirements for EncodableValue.

// It does not attempt to provide useful ordering semantics, and using a

// custom value as a map key is not recommended.

bool operator<(const CustomEncodableValue& other) const {

return this < &other;

}

bool operator==(const CustomEncodableValue& other) const {

return this == &other;

}

private:

std::any value\_;

};

class EncodableValue;

// Convenience type aliases.

using EncodableList = std::vector<EncodableValue>;

using EncodableMap = std::map<EncodableValue, EncodableValue>;

namespace internal {

// The base class for EncodableValue. Do not use this directly; it exists only

// for EncodableValue to inherit from.

//

// Do not change the order or indexes of the items here; see the comment on

// EncodableValue

using EncodableValueVariant = std::variant<std::monostate,

bool,

int32\_t,

int64\_t,

double,

std::string,

std::vector<uint8\_t>,

std::vector<int32\_t>,

std::vector<int64\_t>,

std::vector<double>,

EncodableList,

EncodableMap,

CustomEncodableValue,

std::vector<float>>;

} // namespace internal

// An object that can contain any value or collection type supported by

// Flutter's standard method codec.

//

// For details, see:

// https://api.flutter.dev/flutter/services/StandardMessageCodec-class.html

//

// As an example, the following Dart structure:

// {

// 'flag': true,

// 'name': 'Thing',

// 'values': [1, 2.0, 4],

// }

// would correspond to:

// EncodableValue(EncodableMap{

// {EncodableValue("flag"), EncodableValue(true)},

// {EncodableValue("name"), EncodableValue("Thing")},

// {EncodableValue("values"), EncodableValue(EncodableList{

// EncodableValue(1),

// EncodableValue(2.0),

// EncodableValue(4),

// })},

// })

//

// The primary API surface for this object is std::variant. For instance,

// getting a string value from an EncodableValue, with type checking:

// if (std::holds\_alternative<std::string>(value)) {

// std::string some\_string = std::get<std::string>(value);

// }

//

// The order/indexes of the variant types is part of the API surface, and is

// guaranteed not to change.

//

// The variant types are mapped with Dart types in following ways:

// std::monostate -> null

// bool -> bool

// int32\_t -> int

// int64\_t -> int

// double -> double

// std::string -> String

// std::vector<uint8\_t> -> Uint8List

// std::vector<int32\_t> -> Int32List

// std::vector<int64\_t> -> Int64List

// std::vector<float> -> Float32List

// std::vector<double> -> Float64List

// EncodableList -> List

// EncodableMap -> Map

class EncodableValue : public internal::EncodableValueVariant {

public:

// Rely on std::variant for most of the constructors/operators.

using super = internal::EncodableValueVariant;

using super::super;

using super::operator=;

explicit EncodableValue() = default;

// Avoid the C++17 pitfall of conversion from char\* to bool. Should not be

// needed for C++20.

explicit EncodableValue(const char\* string) : super(std::string(string)) {}

EncodableValue& operator=(const char\* other) {

\*this = std::string(other);

return \*this;

}

// Allow implicit conversion from CustomEncodableValue; the only reason to

// make a CustomEncodableValue (which can only be constructed explicitly) is

// to use it with EncodableValue, so the risk of unintended conversions is

// minimal, and it avoids the need for the verbose:

// EncodableValue(CustomEncodableValue(...)).

// NOLINTNEXTLINE(google-explicit-constructor)

EncodableValue(const CustomEncodableValue& v) : super(v) {}

// Override the conversion constructors from std::variant to make them

// explicit, to avoid implicit conversion.

//

// While implicit conversion can be convenient in some cases, it can have very

// surprising effects. E.g., calling a function that takes an EncodableValue

// but accidentally passing an EncodableValue\* would, instead of failing to

// compile, go through a pointer->bool->EncodableValue(bool) chain and

// silently call the function with a temp-constructed EncodableValue(true).

template <class T>

constexpr explicit EncodableValue(T&& t) noexcept : super(t) {}

// Returns true if the value is null. Convenience wrapper since unlike the

// other types, std::monostate uses aren't self-documenting.

bool IsNull() const { return std::holds\_alternative<std::monostate>(\*this); }

// Convenience method to simplify handling objects received from Flutter

// where the values may be larger than 32-bit, since they have the same type

// on the Dart side, but will be either 32-bit or 64-bit here depending on

// the value.

//

// Calling this method if the value doesn't contain either an int32\_t or an

// int64\_t will throw an exception.

int64\_t LongValue() const {

if (std::holds\_alternative<int32\_t>(\*this)) {

return std::get<int32\_t>(\*this);

}

return std::get<int64\_t>(\*this);

}

// Explicitly provide operator<, delegating to std::variant's operator<.

// There are issues with with the way the standard library-provided

// < and <=> comparisons interact with classes derived from variant.

friend bool operator<(const EncodableValue& lhs, const EncodableValue& rhs) {

return static\_cast<const super&>(lhs) < static\_cast<const super&>(rhs);

}

};

} // namespace flutter

#endif // FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_ENCODABLE\_VALUE\_H\_