// 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\_METHOD\_CHANNEL\_H\_

#define FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_METHOD\_CHANNEL\_H\_

#include <iostream>

#include <string>

#include "basic\_message\_channel.h"

#include "binary\_messenger.h"

#include "engine\_method\_result.h"

#include "method\_call.h"

#include "method\_codec.h"

#include "method\_result.h"

namespace flutter {

class EncodableValue;

// A handler for receiving a method call from the Flutter engine.

//

// Implementations must asynchronously call exactly one of the methods on

// |result| to indicate the result of the method call.

template <typename T>

using MethodCallHandler =

std::function<void(const MethodCall<T>& call,

std::unique\_ptr<MethodResult<T>> result)>;

// A channel for communicating with the Flutter engine using invocation of

// asynchronous methods.

template <typename T = EncodableValue>

class MethodChannel {

public:

// Creates an instance that sends and receives method calls on the channel

// named |name|, encoded with |codec| and dispatched via |messenger|.

MethodChannel(BinaryMessenger\* messenger,

const std::string& name,

const MethodCodec<T>\* codec)

: messenger\_(messenger), name\_(name), codec\_(codec) {}

~MethodChannel() = default;

// Prevent copying.

MethodChannel(MethodChannel const&) = delete;

MethodChannel& operator=(MethodChannel const&) = delete;

// Sends a message to the Flutter engine on this channel.

//

// If |result| is provided, one of its methods will be invoked with the

// response from the engine.

void InvokeMethod(const std::string& method,

std::unique\_ptr<T> arguments,

std::unique\_ptr<MethodResult<T>> result = nullptr) {

MethodCall<T> method\_call(method, std::move(arguments));

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

codec\_->EncodeMethodCall(method\_call);

if (!result) {

messenger\_->Send(name\_, message->data(), message->size(), nullptr);

return;

}

// std::function requires a copyable lambda, so convert to a shared pointer.

// This is safe since only one copy of the shared\_pointer will ever be

// accessed.

std::shared\_ptr<MethodResult<T>> shared\_result(result.release());

const auto\* codec = codec\_;

std::string channel\_name = name\_;

BinaryReply reply\_handler = [shared\_result, codec, channel\_name](

const uint8\_t\* reply, size\_t reply\_size) {

if (reply\_size == 0) {

shared\_result->NotImplemented();

return;

}

// Use this channel's codec to decode and handle the

// reply.

bool decoded = codec->DecodeAndProcessResponseEnvelope(

reply, reply\_size, shared\_result.get());

if (!decoded) {

std::cerr << "Unable to decode reply to method "

"invocation on channel "

<< channel\_name << std::endl;

shared\_result->NotImplemented();

}

};

messenger\_->Send(name\_, message->data(), message->size(),

std::move(reply\_handler));

}

// Registers a handler that should be called any time a method call is

// received on this channel. A null handler will remove any previous handler.

//

// The handler will be owned by the underlying BinaryMessageHandler.

// Destroying the MethodChannel will not unregister the handler, so

// the caller is responsible for unregistering explicitly if the handler

// stops being valid before the engine is destroyed.

void SetMethodCallHandler(MethodCallHandler<T> handler) const {

if (!handler) {

messenger\_->SetMessageHandler(name\_, nullptr);

return;

}

const auto\* codec = codec\_;

std::string channel\_name = name\_;

BinaryMessageHandler binary\_handler = [handler, codec, channel\_name](

const uint8\_t\* message,

size\_t message\_size,

BinaryReply reply) {

// Use this channel's codec to decode the call and build a result handler.

auto result =

std::make\_unique<EngineMethodResult<T>>(std::move(reply), codec);

std::unique\_ptr<MethodCall<T>> method\_call =

codec->DecodeMethodCall(message, message\_size);

if (!method\_call) {

std::cerr << "Unable to construct method call from message on channel "

<< channel\_name << std::endl;

result->NotImplemented();

return;

}

handler(\*method\_call, std::move(result));

};

messenger\_->SetMessageHandler(name\_, std::move(binary\_handler));

}

// Adjusts the number of messages that will get buffered when sending messages

// to channels that aren't fully set up yet. For example, the engine isn't

// running yet or the channel's message handler isn't set up on the Dart side

// yet.

void Resize(int new\_size) {

internal::ResizeChannel(messenger\_, name\_, new\_size);

}

// Defines whether the channel should show warning messages when discarding

// messages due to overflow.

//

// When |warns| is false, the channel is expected to overflow and warning

// messages will not be shown.

void SetWarnsOnOverflow(bool warns) {

internal::SetChannelWarnsOnOverflow(messenger\_, name\_, warns);

}

private:

BinaryMessenger\* messenger\_;

std::string name\_;

const MethodCodec<T>\* codec\_;

};

} // namespace flutter

#endif // FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_METHOD\_CHANNEL\_H\_