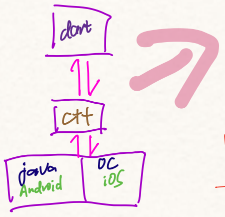


communication through PlatformChannel

based on

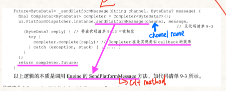
1) BasicMessageChannel

2) MethodChannel

3) EventChannel

Codec → message 序列化，方便 dart，c++，java 对象互相转化

Messenger → 通信

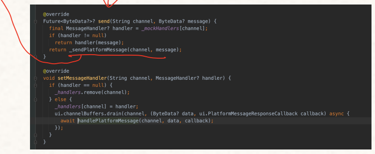


```
Future<ByteData?> _sendPlatformMessage(String channel, ByteData? message) {
  final Completer<ByteData?> completer = Completer<ByteData?>();
  ui.PlatformDispatcher.instance.sendPlatformMessage(channel, message,
                                                                    // 见代码清单 9-3
    (ByteData? reply) { // 将在代码清单 9-13 中被触发
      try {
        completer.complete(reply); // Completer 在此实现类似 callback 的效果
      } catch (exception, stack) { ...... }
    }
  );
  return completer.future;
}
```

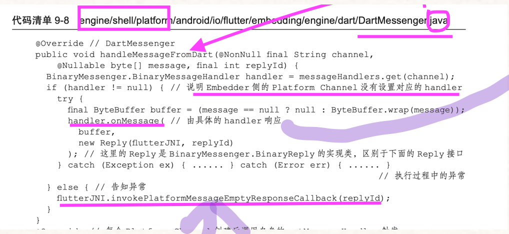
channel name

以上逻辑的本质是调用 Engine 的 SendPlatformMessage 方法，如代码清单 9-3 所示。

↳c++ method

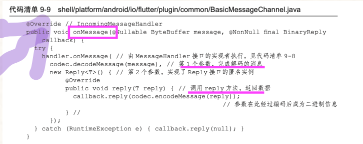


```
@override
Future<ByteData?>? send(String channel, ByteData? message) {
  final MessageHandler? handler = _mockHandlers[channel];
  if (handler != null)
    return handler(message);
  return _sendPlatformMessage(channel, message);
}

@override
void setMessageHandler(String channel, MessageHandler? handler) {
  if (handler == null) {
    _handlers.remove(channel);
  } else {
    _handlers[channel] = handler;
    ui.channelBuffers.drain(channel, (ByteData? data, ui.PlatformMessageResponseCallback callback) async {
      await handlePlatformMessage(channel, data, callback);
    });
  }
}
```



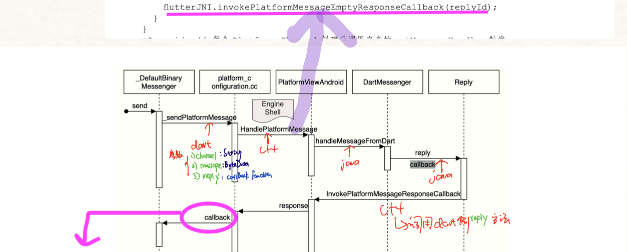
代码清单 9-8　engine/shell/platform/android/io/flutter/embedding/engine/dart/DartMessenger.java

```
@Override // DartMessenger
public void handleMessageFromDart(@NonNull final String channel,
    @Nullable byte[] message, final int replyId) {
  BinaryMessenger.BinaryMessageHandler handler = messageHandlers.get(channel);
  if (handler != null) { // 说明 Embedder 侧的 Platform Channel 没有设置对应的 handler
    try {
      final ByteBuffer buffer = (message == null ? null : ByteBuffer.wrap(message));
      handler.onMessage( // 由具体的 handler 响应
        buffer,
        new Reply(flutterJNI, replyId)
      ); // 这里的 Reply 是 BinaryMessenger.BinaryReply 的实现类，区别于下面的 Reply 接口
    } catch (Exception ex) { ...... } catch (Error err) { ...... }
                                                          // 执行过程中的异常
  } else { // 告知异常
    flutterJNI.invokePlatformMessageEmptyResponseCallback(replyId);
  }
}
```



代码清单 9-9　shell/platform/android/io/flutter/plugin/common/BasicMessageChannel.java

```
@Override // IncomingMessageHandler
public void onMessage(@Nullable ByteBuffer message, @NonNull final BinaryReply
    callback) {
  try {
    handler.onMessage( // 由 MessageHandler 接口的实现者执行，见代码清单 9-8
      codec.decodeMessage(message), // 第 1 个参数，完成解码的消息
      new Reply<T>() { // 第 2 个参数，实现了 Reply 接口的匿名实例
        @Override
        public void reply(T reply) { // 调用 reply 方法，返回数据
          callback.reply(codec.encodeMessage(reply));
                                              // 参数在此经过编码后成为二进制信息
        } //
      });
  } catch (RuntimeException e) { callback.reply(null); }
}
```

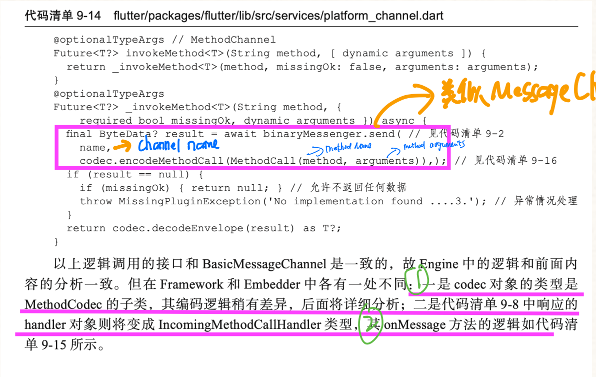


dart侧有一个completer， callback会call completer.complete(reply)，如此一来，await completer.future便会得到结果

图 9-2　BasicMessageChannel 流程

② Method Channel

类似 MessageChannel 的实现，但参数不同



代码清单 9-14　flutter/packages/flutter/lib/src/services/platform_channel.dart

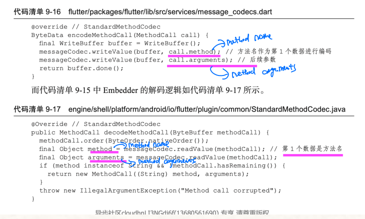
```
@optionalTypeArgs // MethodChannel
Future<T?> invokeMethod<T>(String method, [ dynamic arguments ]) {
  return _invokeMethod<T>(method, missingOk: false, arguments: arguments);
}
@optionalTypeArgs
Future<T?> _invokeMethod<T>(String method, {
    required bool missingOk, dynamic arguments }) async {
  final ByteData? result = await binaryMessenger.send( // 见代码清单 9-2
    name,
    codec.encodeMethodCall(MethodCall(method, arguments)),); // 见代码清单 9-16
  if (result == null) {
    if (missingOk) { return null; } // 允许不返回任何数据
    throw MissingPluginException('No implementation found ....3.'); // 异常情况处理
  }
  return codec.decodeEnvelope(result) as T?;
}
```

channel name

method name

method arguments

以上逻辑调用的接口和 BasicMessageChannel 是一致的，故 Engine 中的逻辑和前面内容的分析一致。但在 Framework 和 Embedder 中各有一处不同：①一是 codec 对象的类型是 MethodCodec 的子类，其编码逻辑稍有差异，后面将详细分析；二是代码清单 9-8 中响应的 handler 对象则将变成 IncomingMethodCallHandler 类型，②其 onMessage 方法的逻辑如代码清单 9-15 所示。



代码清单 9-16　flutter/packages/flutter/lib/src/services/message_codecs.dart

```
@override // StandardMethodCodec
ByteData encodeMethodCall(MethodCall call) {
  final WriteBuffer buffer = WriteBuffer();
  messageCodec.writeValue(buffer, call.method); // 方法名作为第 1 个数据进行编码
  messageCodec.writeValue(buffer, call.arguments); // 后续参数
  return buffer.done();
}
```

method name

method arguments

而代码清单 9-15 中 Embedder 的解码逻辑如代码清单 9-17 所示。

代码清单 9-17　engine/shell/platform/android/io/flutter/plugin/common/StandardMethodCodec.java

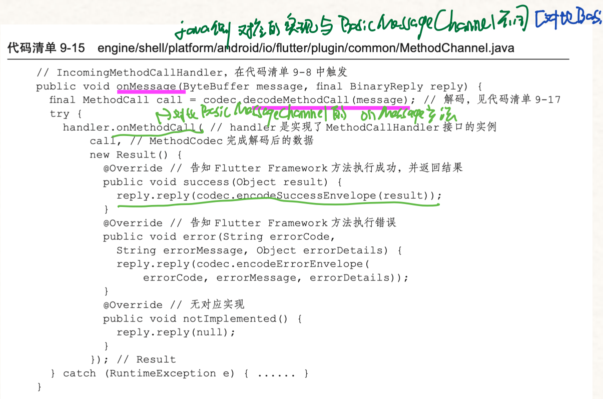
```
@Override // StandardMethodCodec
public MethodCall decodeMethodCall(ByteBuffer methodCall) {
  methodCall.order(ByteOrder.nativeOrder());
  final Object method = messageCodec.readValue(methodCall); // 第 1 个数据是方法名
  final Object arguments = messageCodec.readValue(methodCall);
  if (method instanceof String && !methodCall.hasRemaining()) {
    return new MethodCall((String) method, arguments);
  }
  throw new IllegalArgumentException("Method call corrupted");
}
```

method name

method arguments

异步社区cloudbnLI3NGdi6f(13680561690) 专享 请尊重版权

java侧 对接的实现与 BasicMessageChannel不同 [对比BasicMessageChannel的 handleMessageFromDart]

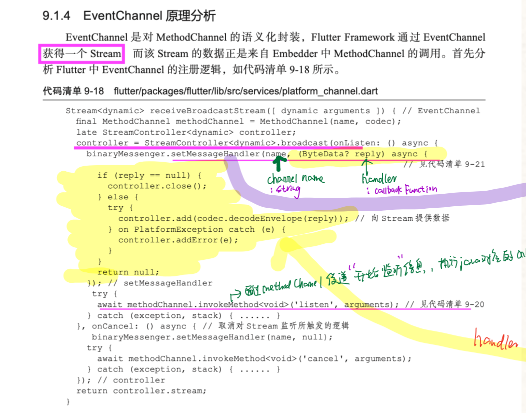


代码清单 9-15　engine/shell/platform/android/io/flutter/plugin/common/MethodChannel.java

```
// IncomingMethodCallHandler，在代码清单 9-8 中触发
public void onMessage(ByteBuffer message, final BinaryReply reply) {
  final MethodCall call = codec.decodeMethodCall(message); // 解码，见代码清单 9-17
  try {
    handler.onMethodCall( // handler 是实现了 MethodCallHandler 接口的实例
        call, // MethodCodec 完成解码后的数据
        new Result() {
          @Override // 告知 Flutter Framework 方法执行成功，并返回结果
          public void success(Object result) {
            reply.reply(codec.encodeSuccessEnvelope(result));
          }
          @Override // 告知 Flutter Framework 方法执行错误
          public void error(String errorCode,
            String errorMessage, Object errorDetails) {
            reply.reply(codec.encodeErrorEnvelope(
                errorCode, errorMessage, errorDetails));
          }
          @Override // 无对应实现
          public void notImplemented() {
            reply.reply(null);
          }
        }); // Result
  } catch (RuntimeException e) { ...... }
}
```

对比 BasicMessageChannel的 onMessage方法

③ Event Channel



9.1.4 EventChannel 原理分析

EventChannel 是对 MethodChannel 的语义化封装，Flutter Framework 通过 EventChannel 获得一个 Stream，而该 Stream 的数据正是来自 Embedder 中 MethodChannel 的调用。首先分析 Flutter 中 EventChannel 的注册逻辑，如代码清单 9-18 所示。

代码清单 9-18 flutter/packages/flutter/lib/src/services/platform_channel.dart

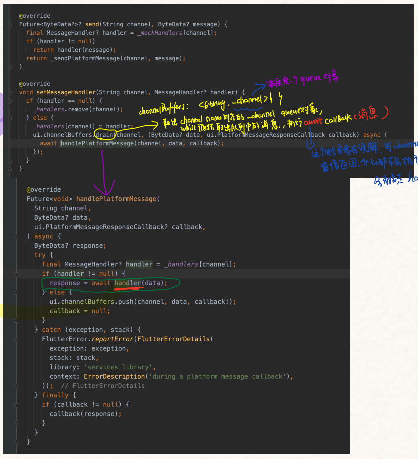
```
Stream<dynamic> receiveBroadcastStream([ dynamic arguments ]) { // EventChannel
  final MethodChannel methodChannel = MethodChannel(name, codec);
  late StreamController<dynamic> controller;
  controller = StreamController<dynamic>.broadcast(onListen: () async {
    binaryMessenger.setMessageHandler(name, (ByteData? reply) async {
                                                          // 见代码清单 9-21
      if (reply == null) {
        controller.close();
      } else {
        try {
          controller.add(codec.decodeEnvelope(reply)); // 向 Stream 提供数据
        } on PlatformException catch (e) {
          controller.addError(e);
        }
      }
      return null;
    }); // setMessageHandler
    try {
      await methodChannel.invokeMethod<void>('listen', arguments); // 见代码清单 9-20
    } catch (exception, stack) { ...... }
  }, onCancel: () async { // 取消对 Stream 监听所触发的逻辑
    binaryMessenger.setMessageHandler(name, null);
    try {
      await methodChannel.invokeMethod<void>('cancel', arguments);
    } catch (exception, stack) { ...... }
  }); // controller
  return controller.stream;
}
```

channel name : string

handler : callback function

→ 通过 method Channel 传递"开始监听信息"，执行java对应的 onMessage方法

handler



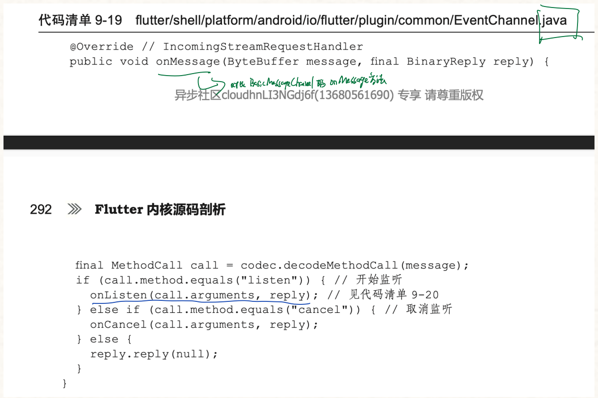
```
@override
Future<ByteData?>? send(String channel, ByteData? message) {
  final MessageHandler? handler = _mockHandlers[channel];
  if (handler != null)
    return handler(message);
  return _sendPlatformMessage(channel, message);
}

@override
void setMessageHandler(String channel, MessageHandler? handler) {
  if (handler == null) {
    _handlers.remove(channel);
  } else {
    _handlers[channel] = handler;
    ui.channelBuffers.drain(channel, (ByteData? data, ui.PlatformMessageResponseCallback callback) async {
      await handlePlatformMessage(channel, data, callback);
    });
  }
}

@override
Future<void> handlePlatformMessage(
  String channel,
  ByteData? data,
  ui.PlatformMessageResponseCallback? callback,
) async {
  ByteData? response;
  try {
    final MessageHandler? handler = _handlers[channel];
    if (handler != null) {
      response = await handler(data);
    } else {
      ui.channelBuffers.push(channel, data, callback!);
      callback = null;
    }
  } catch (exception, stack) {
    FlutterError.reportError(FlutterErrorDetails(
      exception: exception,
      stack: stack,
      library: 'services library',
      context: ErrorDescription('during a platform message callback'),
    )); // FlutterErrorDetails
  } finally {
    if (callback != null) {
      callback(response);
    }
  }
}
```

本质是一个queue对象

channelBuffers: <String, _channel>

取出 channel name 对应的 _channel queue对象，while循环取出队列中的消息，执行 await callBack(消息)

这个地方容易有误解，当channelBuffer中无数据时，while循环直接返回，什么都不会执行，当engine侧有数据抵达时，会触发 handlePlatformMessage方法



代码清单 9-19 flutter/shell/platform/android/io/flutter/plugin/common/EventChannel.java

```
@Override // IncomingStreamRequestHandler
public void onMessage(ByteBuffer message, final BinaryReply reply) {
```

对比 BasicMessageChannel 的 onMessage方法

异步社区cloudhnLI3NGdj6f(13680561690) 专享 请尊重版权

```
  final MethodCall call = codec.decodeMethodCall(message);
  if (call.method.equals("listen")) { // 开始监听
    onListen(call.arguments, reply); // 见代码清单 9-20
  } else if (call.method.equals("cancel")) { // 取消监听
    onCancel(call.arguments, reply);
  } else {
    reply.reply(null);
  }
}
```

④ Platform View阶段

Native侧 iOS

① 实现 FlutterPlugin 协议的
- +(void) registerWithRegistrar:(NSObject *) registar // 注册通道
- handleMethodCall方法 // 对具体的 method call 处理

② 添加注册创建的 iosView (由 viewtype 参数唯一确认)

③ iOS View — define

↓ 点击事件的传递: method channel
↓ invokeMethod "touch"

flutter
- ↙ UikitView
- ↘ Android View

_textureId = SystemChannels.platformView.invokeMethod('create', args)

↓

参数
- width
- height
- direction
- viewtype
- id

To Sumnarize

flutter在创建 UiKitView时，会通过 method channel 通知 native 创建相应的 view，flutter 获得 textureID，并画到 surface 上，这样就完成了 nativeView

↓ 在GPU中的纹理数据