Flutter异常捕捉原理和异常上报

Flutter线程模型/事件机制

在介绍Flutter异常捕捉原理之前，先说明一下Dart的模型。方便我们了解Dart代码的执行流程和获取一个合适的异常捕捉切入点。

我们知道在Java中，如果程序运行发生异常并且没有做捕获处理，程序会直接终止运行发生Crash。但这种情况在Dart中会有所不同。Dart不同于Java的多线程模型，Dart和JavaScript类似属于单线程模型。Dart的单线程模型是以消息循环机制来运行的，其中包含两个任务队列，一个是“微任务队列”microtask queue，另外一个叫“事件队列” event queue。其中微任务队列优先级高于事件队列。

下图是对Dart运行原理和事件机制的一个简单说明：

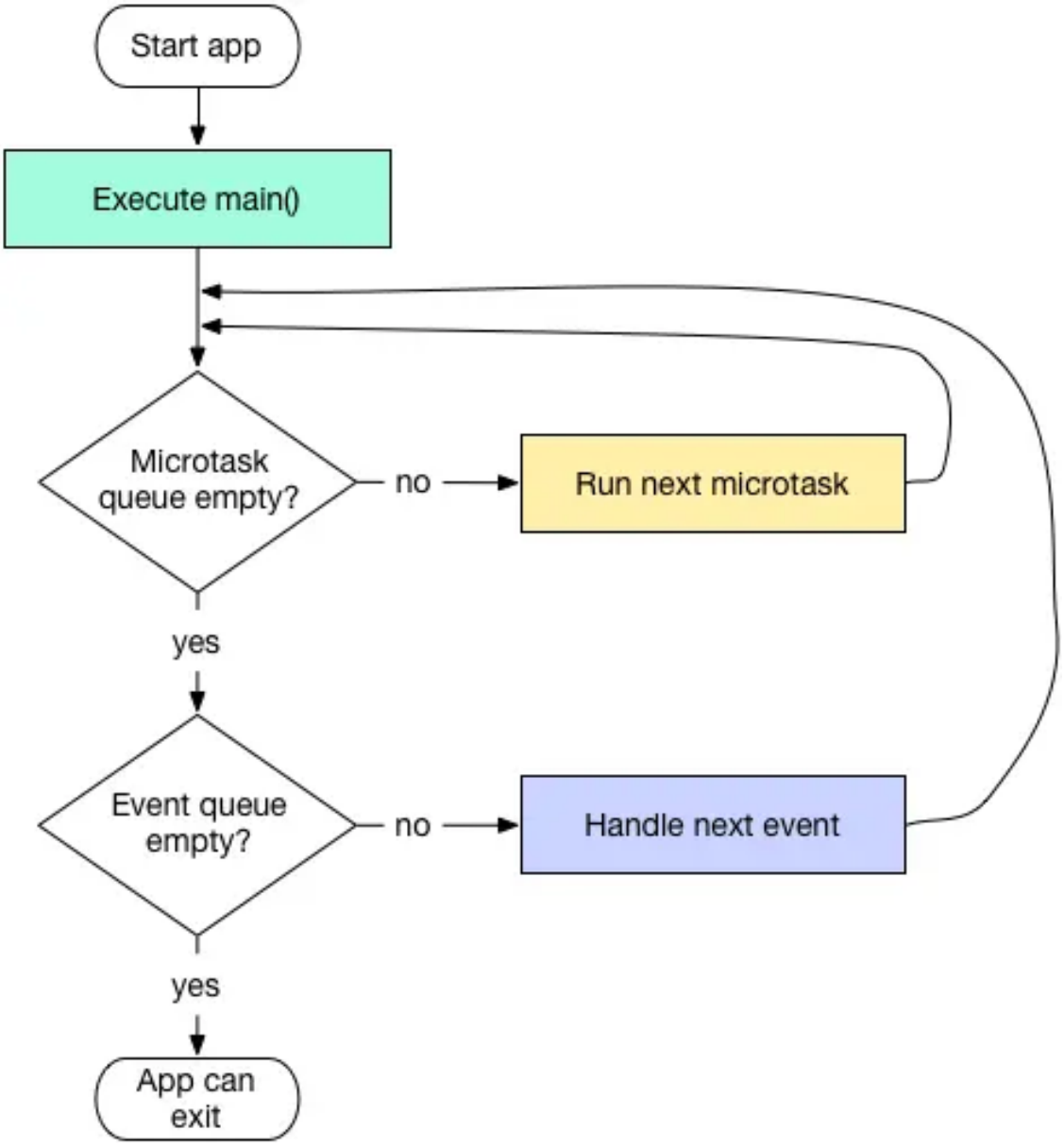

image.png

如图，main() 函数（Dart 程序的入口函数）执行完后，消息循环机制就会启动。main 中的代码将最先执行，然后再执行微任务队列中的任务（按 FIFO 先进先出顺序），再次是事件队列中的任务。执行顺序：Main > MicroTask > EventQueue。在事件任务执行过程中又可以插入新的微任务和事件任务，所有任务执行完毕程序就会退出。

有意思的是在事件循环中，当某个任务发生异常并没有被捕获时，程序并不会退出。而直接导致的结果是当前任务的后续代码不会被执行。也就是说一个任务中的异常是不会影响其它任务执行的。

Flutter 异常分类

1. Flutter dart代码异常（包含app 代码异常，和framework部分异常，和未处理的异步异常）

2. Flutter Engine 异常

Flutter Dart 代码异常

1. Dart 代码异常捕捉

Dart中也有try/catch/finally的存在，用于捕捉代码同步异常。

```
try {

  result = await _methodChannel

      .invokeMethod(METHOD_ADD_WALLET_START, {"cardNo":
cardNo});

} catch(ignored) {
}
```

```
return result;
```

2. Framework异常捕捉

Flutter的框架本身也做了很多异常捕捉措施，例如著名的Flutter红屏异常（布局发生越界或者不符合规范的写法就会导致红屏），就是因为Flutter在框架中已经帮我们预埋了异常的捕获处理逻辑。这个我们可以通过Flutter的源码来了解其实现细节。

Flutter中万物皆Widget，无论是从StatefulWidget还是StatelessWidget中都能发现Widget在创建时都创建了对应的Element的。这里以StatefulWidget源码为例，发现其中会创建StatefulElement。

```
abstract class StatefulWidget extends Widget {

    /// Initializes [key] for subclasses.

    const StatefulWidget({ Key key }) : super(key: key);

    @override

    StatefulElement createElement() =>
StatefulElement(this);

  ...
```

```
}
```

进入StatefulElement，发现StatefulElement继承于ComponentElement

```
class StatefulElement extends ComponentElement {
```

再次追踪到ComponentElement中，会在其**performRebuild()**方法中发现熟悉的try/catch结构。

```
void performRebuild() {

...

  Widget built;

  try {

    built = build();

    debugWidgetBuilderValue(widget, built);

  } catch (e, stack) {

    built = ErrorWidget.builder(
```

```
        _debugReportException(
          ErrorDescription('building $this'),
          e,
          stack,
          informationCollector: () sync* {
            yield
DiagnosticsDebugCreator(DebugCreator(this));
          },
        ),
      );
    }
```

在这里我们发现发生异常时，flutter框架对异常进行了捕捉，并且创建了一个ErrorWidget来做进一步的处理。Flutter的红屏也就是从这里产生的。这里就是我们实现Flutter框架异常捕捉的切入点。 查看_debugReportException的源码来确定异常捕捉的具体实现方式。_debugReportException的源码如下：

```
 FlutterErrorDetails _debugReportException(
    DiagnosticsNode context,
    dynamic exception,
    StackTrace stack, {
    InformationCollector informationCollector,
  }) {
    final FlutterErrorDetails details =
FlutterErrorDetails(
      exception: exception,
      stack: stack,
      library: 'widgets library',
      context: context,
      informationCollector: informationCollector,
    );
    FlutterError.reportError(details);
    return details;
  }
```

核心的是 FlutterError.reportError(details);这一句。进入后发现：

```
 /// Calls [onError] with the given details, unless it is
null.
    static void reportError(FlutterErrorDetails details) {
      assert(details != null);
      assert(details.exception != null);
      if (onError != null)
```

```
        onError(details);
    }
  }
```

继续跟踪这里的onError就会发现它是FlutterError的一个静态属性，有个名为dumpErrorToConsole的默认处理方法。

```
static FlutterExceptionHandler onError =
dumpErrorToConsole;
```

所以我们只需要实现一个自定义的FlutterError.onError来处理异常就能实现Flutter框架异常的捕捉和上报。最终实现如下：

```
void main() {
    FlutterError.onError = (FlutterErrorDetails details) {
      reportError(details);
    };

   ...
}
```

3. dart的异步异常捕捉

刚才介绍的两种异常捕捉方式并不足以应对所有的flutter异常。例如下面的这种异常：

```
  try {
     Future.delayed(Duration(seconds: 1)).then((e) =>
Future.error("xxx"));
   } catch (e) {
     print(e)
  }
```

Dart有个Zone的概念，可以简单的理解为沙箱。不同的Zone相处独立，互不影响。借助于Zone就可以指定代码的执行环境，捕获、拦截或者修改代码行为。Flutter中有一个Zone.runZoned方法。

```
R runZoned<R>(R body(),
    {Map zoneValues, ZoneSpecification zoneSpecification, Function onError}) {
  if (onError == null) {
    return _runZoned<R>(body, zoneValues, zoneSpecification);
  }
  void Function(Object) unaryOnError;
  void Function(Object, StackTrace) binaryOnError;
  if (onError is void Function(Object, StackTrace)) {
    binaryOnError = onError;
  } else if (onError is void Function(Object)) {
    unaryOnError = onError;
  } else {
    throw new ArgumentError("onError callback must take either an Object "
        "(the error), or both an Object (the error) and a StackTrace.");
  }
  HandleUncaughtErrorHandler errorHandler = (Zone self, ZoneDelegate parent,
      Zone zone, error, StackTrace stackTrace) {
    try {
      if (binaryOnError != null) {
        self.parent.runBinary(binaryOnError, error, stackTrace);
```

image.png

在runZoned方法的源码中我们又看到了熟悉的onError。在这里我们注入自定义的onError回调方法，用于捕获方法1、方法2中未能捕获处理的异常。 最终的实现如下：

```
runZoned(
    () => MyApp(),
    onError: (dynamic ex, StackTrace stack) {
      reportError(ex, stack);
    },
);
```

4. 最终实现

```
///flutter 应用入口

  void main() {
  //  Flutter framework 异常捕获
    FlutterError.onError = (FlutterErrorDetails details) {
      bool isDebugMode = false;
      assert(() {
        isDebugMode = true;
        return true;
      }());
      if (isDebugMode) {
        FlutterError.dumpErrorToConsole(details);
      } else {
        //profile,release两个模式下下捕捉异常信息
        reportFrameworkError(details);
      }
   };
```

```
    //其他类型异常

    runZoned(
      () => runAutoSizeApp(MyApp(), width: 375, height:
667),
      onError: (dynamic ex, StackTrace stack) {
        reportError(ex, stack);
      },
    );
  }
```

注：针对*debug*环境下的异常不需要捕捉上报，我们任然让它走旧有的异常处理逻辑。

异常信息的上报

在处理异常信息上报之前我们先来看看捕捉到的Flutter日志格式。

```
MissingPluginException(No implementation found for method
getCache on channel com.zhongan.beeline/ZACache)
#0      MethodChannel.invokeMethod (package:flutter/src/
services/platform_channel.dart:319:7)
<asynchronous suspension>
#1      CachePlugin.getCache (package:ZABank/plugin/
CachePlugin.dart:34:37)
#2      new HomePageViewModel (package:ZABank/home/bloc/
HomePageViewModel.dart:158:17)
#3      HomePageViewModel.create (package:ZABank/home/bloc/
HomePageViewModel.dart:138:23)
```

```
#4      _NewHomeTabState.build.<anonymous closure>
(package:ZABank/home/NewHomeTab.dart:36:47)
#5      BuilderStateDelegate.initDelegate
(package:provider/src/delegate_widget.dart:249:14)
#6      _DelegateWidgetState._initDelegate
(package:provider/src/delegate_widget.dart:118:21)
#7      _DelegateWidgetState.initState (package:provider/
src/delegate_widget.dart:110:5)
#8      StatefulElement._firstBuild (package:flutter/src/
widgets/framework.dart:4355:58)
#9      ComponentElement.mount (package:flutter/src/
widgets/framework.dart:4201:5)
#10     Element.inflateWidget (package:flutter/src/widgets/
framework.dart:3194:14)
#11     MultiChildRender
```

从上面的 flutter crash 日志来看，我们可以将其简单的分为两部分：**异常标题**（异常类型和对应异常详情说明）；**异常堆栈信息**。将这些信息上报到不同的异常监控平台需要做不同的处理

1. 以 Android 端为例，如果上报平台为 bugly。我们只需要将上面的异常日志分为异常标题和异常堆栈，通过 methodchannel 传递到 native 层，然后通过 bugly 接口直接上报即可。以下为 bugly 上报接口调用示例：

```
if (!CrashModule.hasInitialized()) return;
```

```
    CrashReport.postException(4, excpetionType,
excpetionMessage, stack, null);
```

2. 如果我们的上报目标平台为 firebase，那需要做更进一步的处理。因为 firebase 平台目前给出的接口没有 bugly 的灵活，上报时接口只认 Java 的 Exception 对象。因此为了将 flutter 异常上报到 firebase 平台我们需要对其进行解析转换，将其格式转换为 Java 的异常类型然后再上报到 firebase。

   flutter 异常的标题没什么好说的，可以直接作为 Java Exception 的 detailMessage 字段。

   flutter 异常堆栈部分按行切割后，每一行日志还需要按规则进行拆解。

   以下为拆解规则：

```
#1   CachePlugin.getCache  (package:ZABank/plugin/
CachePlugin.dart:34:37)
```

丢弃	class	method	file	line	丢弃
#1	CachePlugin	getCache	package:ZABank/plugin/CachePlugin.dart	34	37

这些字段分别对应了 Java 中 StackTraceElement 类的几个字

段。

```
*public final class StackTraceElement implements [java.io](http://java.io).Serializable {*

*// Normally initialized by VM (public constructor added in 1.5)*

*private String declaringClass;*

*private String methodName;*

*private String fileName;*

*private int    lineNumber;*
```

最终我们就能在firebase中看见上报的flutter异常。