Flutter 异常捕获

KB_MORE

Flutter 异常指的是，Flutter 程序中 Dart 代码运行时意外发生的错误事件。我们可以通过与 Swift 类似的 try-catch 机制来捕获它。但与 Swift 不同的是，Dart 程序不强制要求我们必须处理异常。

这是因为，Dart 采用事件循环的机制来运行任务，所以各个任务的运行状态是互相独立的。也就是说，即便某个任务出现了异常我们没有捕获它，Dart 程序也不会退出，只会导致当前任务后续的代码不会被执行，用户仍可以继续使用其他功能。

Dart 异常，根据来源又可以细分为 App 异常和 Framework 异常。Flutter 为这两种异常提供了不同的捕获方式。

APP异常捕获

App 异常，就是应用代码的异常，通常由未处理应用层其他模块所抛出的异常引起。根据异常代码的执行时序，App 异常可以分为两类，即同步异常和异步异常：同步异常可以通过 try-catch 机制捕获，异步异常则需要采用Future 提供的catchError 语句捕获。

这两种异常的捕获方式，如下代码所示：

```
// 使用 try-catch 捕获同步异常
try {
    throw SYReportException('发生一个dart 同步异常');
}
catch(e) {
```

```
  print(e);
}

// 使用 catchError 捕获异步异常
Future.delayed(Duration(seconds: 1)).then((e) {
  if (sendFlag) {
    print('异步异常发生之前 >>>>>>>>>>>');
    throw SYReportException('发生一个dart 异步异常');
  }
  print('异步异常后执行的代码 <<<<<<<<<<<');
}).catchError((error){
print('error);
});

// 注意，以下代码无法捕获异步异常

try {
    Future.delayed(Duration(seconds: 1)).then((e) {
      if (sendFlag) {
        print('异步异常发生之前 >>>>>>>>>>>');
        throw SYReportException('发生一个dart 异步异常');
      }
      print('异步异常后执行的代码 <<<<<<<<<<<');
    });
} catch (e) {
    print("这是不会执行的. ");
}
```

需要注意的是，这两种方式是不能混用的。可以看到，在上面的代码中，我们是`无法使用 try-catch 去捕获一个异步调用所抛出的异常`的。

`同步的 try-catch` 和`异步的 catchError`，为我们提供了直接捕获特定异常的能力，而如果我们想`集中管理`代码中的所有异常，

Flutter 也提供了 `Zone.runZoned`方法。

我们可以给代码执行对象指定一个 `Zone`，在 Dart 中，`Zone`表示一个代码`执行的环境范围`，其概念类似`沙盒`，不同`沙盒`之间是互相隔离的。如果我们想要观察沙盒中代码执行出现的异常，沙盒提供了 `onError 回调函数`，拦截那些在代码执行对象中的未捕获异常。

在下面的代码中，我们将可能抛出异常的语句放置在了 Zone 里。可以看到，在没有使用`try-catch` 和 `catchError` 的情况下，无论是同步异常还是异步异常，都可以通过`Zone` 直接捕获到：

```
runZoned(() {
  // 同步抛出异常
  throw SYReportException('发生一个dart 同步异常');
}, onError: (dynamic e, StackTrace stack) {
  print('zone捕获到了同步异常');
});

runZoned(() {
  // 异步抛出异常
  Future.delayed(Duration(seconds: 1))
      .then((e) => throw SYReportException('发生一个dart 异步异常'));
}, onError: (dynamic e, StackTrace stack) {
  print('zone捕获到了异步异常');
});
```

因此，如果我们想要集中捕获 Flutter 应用中的未处理异常，可以把 main 函数中的 `runApp` 语句也放置在 `Zone` 中。这样在

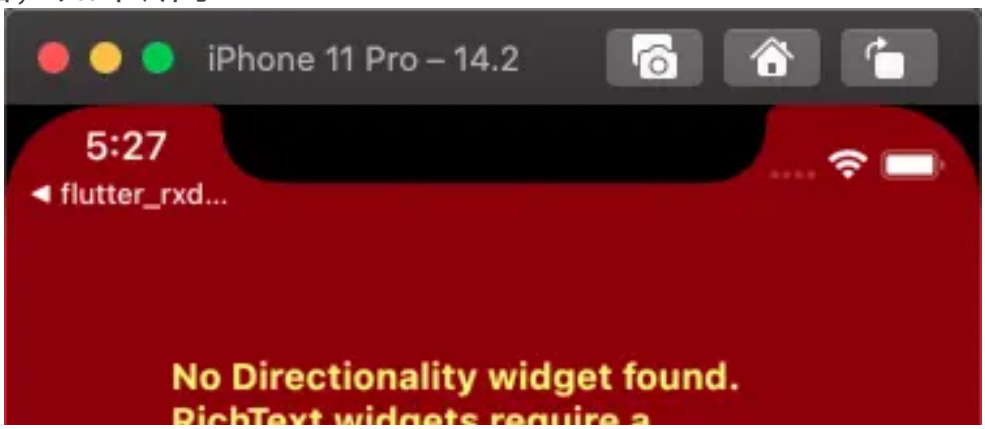
检测到代码中运行异常时，我们就能根据获取到的异常上下文信息，进行统一处理了：

```
runZonedGuarded(() {
    runApp(MyApp());
}, (error, stackTrace) {
    // 这个闭包中发生的Exception是捕获不到的 @山竹
    SYExceptionReportChannel.reportException(error,
stackTrace);
}, zoneSpecification: ZoneSpecification(
    print: (Zone self, ZoneDelegate parent, Zone zone,
String line) {
  // 记录所有的打印日志
  parent.print(zone, "line是啥: $line");
},
));
```

接下来，我们再看看 Framework 异常应该如何捕获吧。

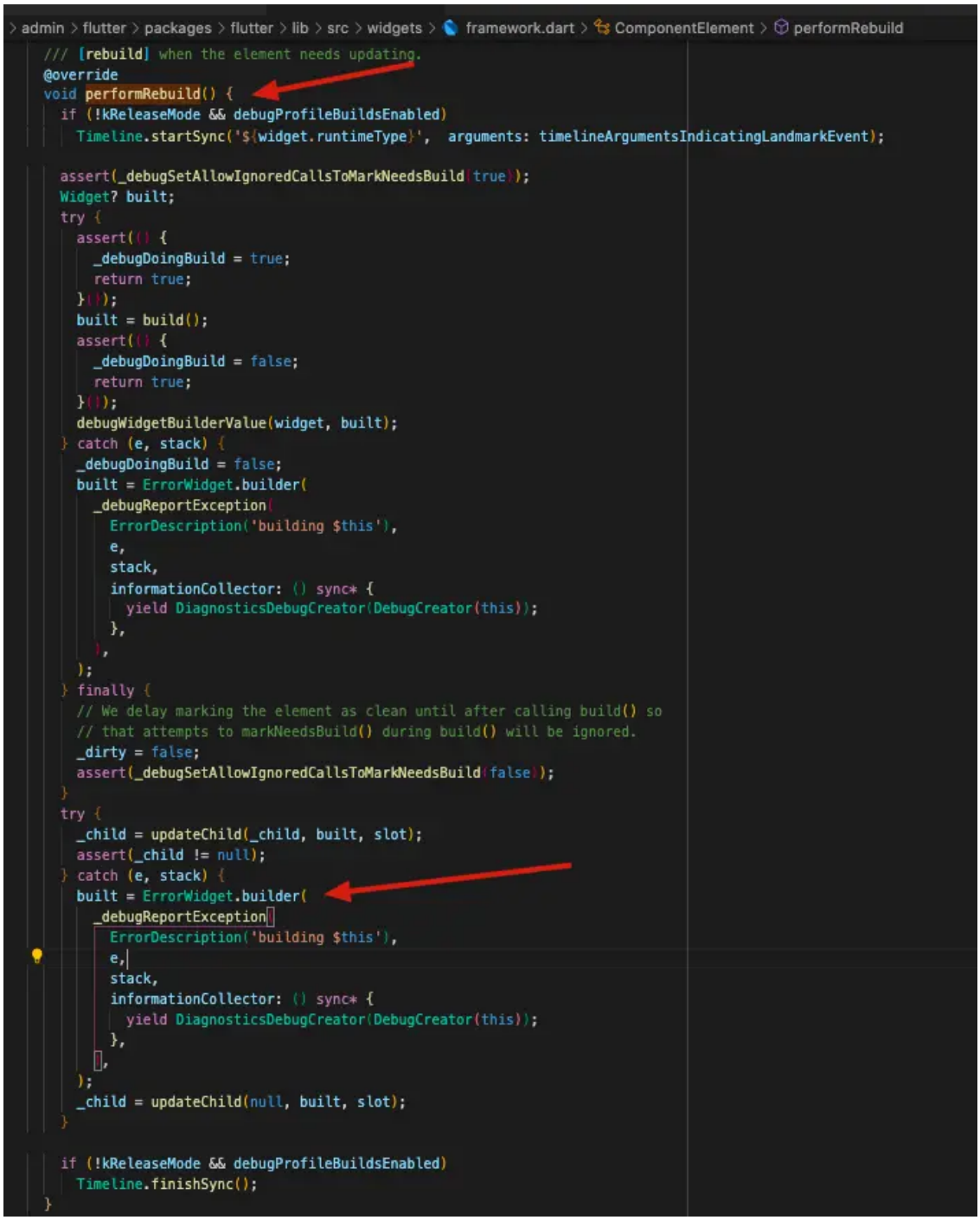
Framework 异常的捕获

`Framework 异常`，就是`Flutter 框架`引发的`异常`，通常是由应用代码触发了 Flutter 框架底层的`异常判断`引起的。比如，当布局不合规范时，Flutter 就会自动弹出一个触目惊心的红色错误界面，如下所示：



RichText widgets require a Directionality widget ancestor.
The specific widget that could not find a Directionality ancestor was:
  RichText
The ownership chain for the affected widget is: "RichText ← Text ← MyApp ← [root]"
Typically, the Directionality widget is introduced by the MaterialApp or WidgetsApp widget at the top of your application widget tree. It determines the ambient reading direction and is used, for example, to determine how to lay out text, how to interpret "start" and "end" values, and to resolve EdgeInsetsDirectional, AlignmentDirectional, and other *Directional objects.
See also:
https://flutter.dev/docs/testing/errors

这其实是因为，`Flutter` 框架在调用 `build` 方法构建页面时进行了 `try-catch` 的处理，并提供了一个 `ErrorWidget`，用于在出现异常时进行信息提示：



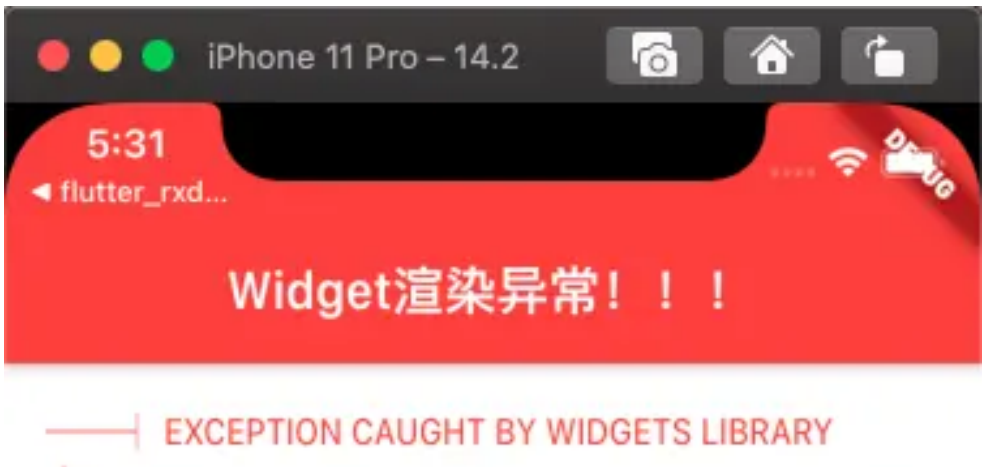
这个页面反馈的信息比较丰富，适合开发期定位问题。但如果让用户看到这样一个页面，就很糟糕了。因此，我们通常会重写 `ErrorWidget.builder` 方法，将这样的错误提示页面替换成一个更加友好的页面。

下面的代码演示了自定义错误页面的具体方法。在这个例子中，我们自定义了错误页面，显示导航栏和可滚动的错误信息：

```
// 重写 ErrorWidget 的builder，显示地优雅一些
ErrorWidget.builder = (FlutterErrorDetails details) {
  print('错误widget详细的错误信息为：' + details.toString());

  return MaterialApp(
    title: 'Error Widget',
    theme: ThemeData(
      primarySwatch: Colors.red,
    ),
    home: Scaffold(
      appBar: AppBar(
        title: Text('Widget渲染异常！！！'),
      ),
      body: _createBody(details),
    ),
  );
};
```

运行效果如下所示：



The following assertion was thrown building Text("test"):
No Directionality widget found.
RichText widgets require a Directionality widget ancestor.
The specific widget that could not find a Directionality ancestor
was:
  RichText
The ownership chain for the affected widget is: "RichText ← Text
  ← MyApp ← [root]"
Typically, the Directionality widget is introduced by the MaterialApp or WidgetsApp widget at the top of your application
widget tree. It determines the ambient reading direction and is
used, for example, to determine how to lay out text, how to
interpret "start" and "end" values, and to resolve EdgeInsetsDirectional, AlignmentDirectional, and other *Directional objects.

The relevant error-causing widget was:
  Text

file:///Users/yanlang/CodeBySelf/Flutter/Learns/flutter_catch_exception/lib/app/app.dart:27:12

When the exception was thrown, this was the stack:
#0   debugCheckHasDirectionality.<anonymous closure> (package:flutter/src/widgets/debug.dart:247:7)
#1   debugCheckHasDirectionality (package:flutter/src/widgets/debug.dart:263:4)
#2   RichText.createRenderObject (package:flutter/src/widgets/basic.dart:5161:37)
#3   RenderObjectElement.mount (package:flutter/src/widgets/framework.dart:5366:28)

比起之前触目惊心的红色错误页面，自定义的看起来优雅一些，当然也可以找UI帮忙设计更友好的界面。需要注意的是，`ErrorWidget.builder`方法提供了一个参数`details`用于表示当前的`错误上下文`，为避免用户直接看到错误信息，这里我们并没有将它展示到界面上。但是，我们不能丢弃掉这样的异常信息，需要提供统一的异常处理机制，用于后续分析异常原因。

为了`集中处理框架异常`，Flutter 提供了`FlutterError`类，这个类的`onError`属性会在接收到框架异常时执行相应的回调。因此，要实现自定义捕获逻辑，我们只要为它提供一个自定义的错误处理回调即可。

在下面的代码中，我们使用 Zone 提供的`handleUncaughtError`语句，将 Flutter 框架的异常统一转发到当前的 Zone 中，这样我们就可以统一使用 Zone 去处理应用内的所有异常了：

```
// framework异常捕获，转发到当前的 Zone
FlutterError.onError = (FlutterErrorDetails details) async
{
    Zone.current.handleUncaughtError(details.exception,
details.stack);
};
```

异常上报

到目前为止，我们已经捕获到了应用中所有的未处理异常。但如果只是把这些异常在控制台中打印出来还是没办法解决问题，我们还需要把它们上报到开发者能看到的地方，用于后续分析定位并解决问题。

三方，我们一般都是用`bugly`。如果公司有自研的bug系统，那就更好了。

这些异常上报，我们将使用`MethodChannel`推送给`Native`，由`Native`上报到bugly或自研的异常系统。

这里只展示Dart的代码实现，至于Native怎么实现Channel，自行Google即可

Dart实现

代码如下:

```
/// flutter exception channel
class SYExceptionReportChannel {
  static const MethodChannel _channel =
      const MethodChannel('sy_exception_channel');

  // 上报异常
  static reportException(dynamic error, dynamic stack) {
    print('捕获的异常类型 >>> : ${error.runtimeType}');
    print('捕获的异常信息 >>> : $error');
    print('捕获的异常堆栈 >>> : $stack');

    Map reportMap = {
      'type': "${error.runtimeType}",
      'title': error.toString(),
```

```
      'description': stack.toString()
    };

    // 得使用这个
    print('这是通过convert转的json');
    print(jsonEncode(reportMap));

    _channel.invokeListMethod('reportException',
reportMap);
  }
}
```

我们捕获到的异常后，由 channel 推送给 Native，包含三个信息：

异常的类型信息

异常的简要说明信息（即 error 的 toString 的值）

异常的堆栈信息

优化、封装及问题点

综合上述的阐述，我们将代码做一些封装和优化。

优化：异常捕获后，在 debug 和 release 的模式下是不一样的处理，debug 模式，直接打印到控制台是最直观的，release 模式下，无法感知哪里出了问题，所以我们需要上报，然后分析问题。

区分当前是 debug 还是 release，有一个比较巧妙的方式，代码及注释如下：

```
// 比较巧妙的一种方式判定是否是debug模式
static bool get isInDebugMode {
    bool inDebugMode = false;
    // 如果debug模式下会触发赋值，只有在debug模式下才会执行assert
```

```
    assert(inDebugMode = true);
    return inDebugMode;
}
```

基于上述的思路，我们将未捕获的异常转发到zone做一个判断：

```
// framework异常捕获，转发到当前的 Zone
    FlutterError.onError = (FlutterErrorDetails details)
async {
      // debug模式
      if (ExceptionReportUtil.isInDebugMode) {
        // 打印到控制台
        FlutterError.dumpErrorToConsole(details);

        // release模式
      } else {
        // 转发到zone
        Zone.current.handleUncaughtError(details.exception,
details.stack);
      }
    };
```

封装：main函数中的代码，自然是越简练越好，但将未捕获的异常转发到zone及错误Widget重写必须放在main中，所以抽取一个工具类

`ExceptionReportUtil`

```
/// 工具类
class ExceptionReportUtil {
  // 比较巧妙的一种方式判定是否是debug模式
  static bool get isInDebugMode {
    bool inDebugMode = false;
    // 如果debug模式下会触发赋值，只有在debug模式下才会执行assert
```

```
    assert(inDebugMode = true);
    return inDebugMode;
  }

  // 初始化异常捕获配置
  static void initExceptionCatchConfig() {
    // framework异常捕获，转发到当前的 Zone
    FlutterError.onError = (FlutterErrorDetails details)
async {
      // debug模式
      if (ExceptionReportUtil.isInDebugMode) {
        // 打印到控制台
        FlutterError.dumpErrorToConsole(details);

        // release模式
      } else {
        // 转发到zone
        Zone.current.handleUncaughtError(details.exception,
details.stack);
      }
    };

    // 重写 ErrorWidget 的builder，显示地优雅一些
    ErrorWidget.builder = (FlutterErrorDetails details) {
      print('错误widget详细的错误信息为：' +
details.toString());

      return MaterialApp(
        title: 'Error Widget',
        theme: ThemeData(
          primarySwatch: Colors.red,
        ),
        home: Scaffold(
          appBar: AppBar(
            title: Text('Widget渲染异常！！！'),
```

```
          ),
          body: _createBody(details),
        ),
      );
    };
  }

  // 创建错误widget body
  static Widget _createBody(dynamic details) {
    // 正确代码
    return Container(
      color: Colors.white,
      child: SingleChildScrollView(
        child: Padding(
          padding: const EdgeInsets.all(16.0),
          child: Text(
            details.toString(),
            style: TextStyle(color: Colors.red),
          ),
        ),
      ),
    );
  }
}
```

问题点： 在runZonedGuarded函数的闭包中接收未捕获的异常，然后上报，如果执行该闭包中的代码发生异常，是无法捕获的：

```
代码及注释如下：
main(List<String> args) {
  // 初始化Exception 捕获配置
  ExceptionReportUtil.initExceptionCatchConfig();
```

```
  runZonedGuarded(() {
    runApp(MyApp());
  }, (error, stackTrace) {
    // 这个闭包中发生的Exception是捕获不到的 @山竹
    SYExceptionReportChannel.reportException(error,
stackTrace);
  }, zoneSpecification: ZoneSpecification(
    print: (Zone self, ZoneDelegate parent, Zone zone,
String line) {
      // 记录所有的打印日志
      parent.print(zone, "line是啥: $line");
    },
  ));
}
```

我们通过`SYExceptionReportChannel.reportException(error, stackTrace)`将错误上报给 Native，但在 Native 如果没有实现 channel 的链接，那么必然会报`MissingPluginException`，这个异常是不在当前的 zone 中的，所以无法捕获。