

# flutter 线上异常用 bugly 监控

江水东流

线上问题监控是程序开发必要流程,flutter 该如何做呢?

我们用 bugly 收集异常

## 1.1 异常采集

Native 的异常通常伴随着应用崩溃，而 dart 异常，提供了一种灵活的机制来捕获和处理异常，不会直接导致应用程序崩溃，常表现为页面白屏、执行操作卡死等，它们在稳定性的用户体验上并不是完全对等的。



捕获异常方式：

- `FlutterError.onError` 适用于全局异常处理
- `runZonedGuarded` 适用于在特定区域内捕获异常
- `PlatformDispatcher.instance.onError` 适用于处理底层平台异常（Flutter SDK v3.3 及以上支持）

○ try catch 适用于捕获同步代码块中的异常

○ Future.catchError 适用于捕获异步操作中的异常

pubspec.yaml 加入 bugly 插件

flutter\_bugly\_plugin: ^0.0.9

安卓需要添加几个权限,详情见下面链接

[https://pub.dev/packages/flutter\\_bugly\\_plugin](https://pub.dev/packages/flutter_bugly_plugin)

1 项目初始化时候 找个地方初始化下bugly

```
Future<void> initBugly() async {
  FlutterBuglyPlugin.init(
    appIdAndroid: "xxxxxx",
    appIdIOS: "xxxxxxx",
  );
  final onError = FlutterError.onError;

  FlutterError.onError = (FlutterErrorDetails details) {
    onError?.call(details);
    FlutterBuglyPlugin.reportException(
      exceptionName:
        'FlutterError.onError - $
{details.exceptionAsString()}',
      reason: details.stack.toString());
  };
  PlatformDispatcher.instance.onError = (error, stack) {
    FlutterBuglyPlugin.reportException(
      exceptionName:
        'PlatformDispatcher.onError - $
{error.toString()}',
      reason: stack.toString());
    return true;
  };
}
```

```

2 runApp的异常也上报
  runZonedGuarded(() {
    runApp(MyApp()));
  }, (error, stackTrace) {

    if (kReleaseMode) {
      FlutterBuglyPlugin.reportException(
        exceptionName: '${error.toString()}',
        reason: stackTrace.toString());
    }
  });

```

## 异常级别

数组越界的主动引发执行异常，可以看到同类型的错误出现的位置不一样导致的页面表现也不一致。

```
List exceptionList = [];
```

```
exceptionList[1];
```

同步初始化异常：

在执行入口 main() 主动引发异常，因为同步的初始化执行被阻塞导致后续的页面无法成功渲染，表现为页面白屏。

未捕获的异步错误：

主动在 Future.delayed 中引发异常，因为异步任务不会阻塞主线程页面渲染，所以页面表现正常。

写到 build 方法里面

组件渲染错误：

绘制主动引发异常，引发 framework 异常，表现为白屏

(dev 下红屏)。

手势事件回调异常：

Flutter 的事件处理是异步的，并且是在单独的线程中处理的，不会阻塞页面渲染，但是异常代码之后的执行逻辑将被阻塞无法响应。

### 1.3 异常细类

Flutter 异常类型主要包括以下几种（不包含全部）：

1. `FlutterError`：这是 Flutter 异常的基类，用于表示 Flutter 框架内部的错误。它提供了一个描述错误的消息和一个可选的错误详情。
2. `AssertionError`：这是一个断言异常，用于在开发过程中发现错误或无效的操作。当断言失败时，会抛出此异常。
3. `FormatException`：表示由于格式错误而导致的异常。例如，将字符串转换为数字时，如果字符串的格式不正确，则会抛出此异常。
4. `RangeError`：表示由于超出范围而导致的异常。例如，当索引超出列表的范围时，会抛出此异常。
5. `NoSuchMethodError`：表示尝试调用不存在的方法或访问不存在的属性时引发的异常。

这些是一些常见的 Flutter 异常类型和相应的案例。根据项目的具体需求和实现细节，可能会遇到其他类型的异常。

SDK 通过异常摘要的 `runtimeType` 来采集异常的异常细类，

对于 `CastError`（类型转换错误）、`RangeError`、`PlatformException`、`NoSuchMethodError`、`MissingPluginException` 等多种错误类型,我们认为其是影响业务的，所以对 Flutter 的异常进行分类的处理。

## 上报异常解析

如何你构建包时候用

安卓 `flutter build apk --release`

ios `flutter build ios --release` 后用 xcode archive 导出 ipa

这种情况没有对代码剥离调试符号,上报到 bugly 的异常直接可以看到堆栈

缺点是

1. 增加应用大小\*\*：所有的调试信息（比如堆栈信息）将会包含在你的应用程序的主二进制文件中，导致文件大小增加。
2. 包含完整的符号信息\*\*：由于调试信息没有被剥离出主二进制文件，如果有错误发生，错误报告将包含完整的符号信息，这有助于你直接在错误报告中定位到 Dart 源码的具体位置。

一般我们打线上包需要 `-split-debug-info` 剥离调试符号

// Android

`flutter build apk --release --split-debug-info=/<directory>`

// iOS

`flutter build ios --release -split-debug-info=/<directory>`

以ios为例

在Xcode中对Flutter应用执行归档操作时，实际上是使用已经构建好的Flutter框架。如果你想使用`--split-debug-info`选项（它针对的是Flutter的Dart代码部分），你需要在命令行中使用Flutter命令来构建。

在终端中，使用以下命令（确保先从Flutter项目的根目录开始）：

```
flutter build ios --release --split-debug-info=/path/to/directory/
```

这个命令会触发Flutter构建流程，并将Dart相关的调试信息存储在指定路径的目录中。构建完成后，相应的调试符号文件会生成在该目录下。(会把代码映射关系生成一个app.ios-arm64.symbols文件)

之后，你可以正常地在Xcode中进行归档操作：

1. 打开Xcode，然后打开你的Flutter项目的`Runner.xcworkspace`。
2. 选择 Product > Archive 进行归档操作。
3. 归档完成后，通过Xcode的Organizer窗口导出IPA或者分发到TestFlight。

如果以后需要使用调试信息来解析混淆过的堆栈跟踪，你需要确保保存了`--split-debug-info`命令生成的那些文件，并了

解它们与哪个版本的应用相对应。

请注意，`--split-debug-info`仅适用于 Dart 代码，Objective-C/Swift 或者其他原生端代码的调试符号需要通过 Xcode 相应的工具和流程来管理。

这样线上收集到的异常 按照#00 方式整理成下面这样 每个堆栈换一行

```
#00 abs 0000000109defa1f
_kDartIsolateSnapshotInstructions+0x1c56df
#01 abs 0000000109fee21b
_kDartIsolateSnapshotInstructions+0x3c3edb
#02 abs 000000010a18ef3b
_kDartIsolateSnapshotInstructions+0x564bfb
#03 abs 0000000109f5db67
_kDartIsolateSnapshotInstructions+0x333827
#04 abs 000000010a1277c7
_kDartIsolateSnapshotInstructions+0x4fd487
```

2. 确保你有访问生成堆栈跟踪时相应版本的 `.symbols` 文件。
3. 使用 Flutter 的 `symbolize` 命令，它是 `flutter` 工具的一部分，来解析混淆后的堆栈。

打开一个命令行或者终端窗口，并执行以下命令：

```
flutter symbolize -d /path/to/directory/ -i
input.stacktrace.txt > output.symbolicated.txt
```

参数说明：

- `-d` 后跟 `.symbols` 文件存储的目录。
- `-i` 用于指定输入堆栈跟踪文件的路径（这是你从线上问题收集到的混淆堆栈跟踪）。如果堆栈跟踪是标准输入 (STDIN) 形式的，可以使用 `-i -`。
- `>` 是将命令的输出（解析后的堆栈）重定向到一个文件，你可以自定义这个文件的名字。

### 实际例子

```
flutter symbolize -d /Users/xxxx/Desktop/symbol文件/app.ios-arm64.symbols -i /Users/xxxx/Desktop/错误文件/cuowu.txt > jiexi.txt
```

执行该命令后，`output.symbolicated.txt` 文件将会包含解析后的堆栈跟踪信息，你可以通过这些信息将混淆的堆栈跟踪映射回你的原始 Dart 源码。

请确保使用了正确版本的符号文件，因为不同的编译版本会生成不同的 `.symbols` 文件，只有匹配的版本之间才能正确解析。

这样就把异常解析好了

对于 Flutter 生成的 `.symbols` 文件，你可以使用 Flutter 的 `symbolize` 命令来解析 Flutter 的 Dart 代码部分的堆栈跟踪。如果你的堆栈跟踪同时涉及到了 iOS 原生和 Dart 代码，你需要分别用 dSYM 和 Flutter 生成的 `.symbols` 文件来分别解析这两部分的内容。



如何你还想混淆你的 flutter 代码

```
flutter build ios --release --obfuscate --split-debug-info=
```

放 symbols 文件的目录

```
安卓用 flutter build apk --release --obfuscate --split-debug-info=
```

`--obfuscate` 选项不是必须的。这个选项用于在编译你的应用时进行代码混淆，以防止逆向工程。混淆会更改类、方法和字段名的符号至难以理解的字符组合，从而提高代码的保密性。

如果你不需要混淆你的代码（比如，你的应用不包含敏感逻辑或者专有算法），你可以不加这个选项。同时，如果你不使用 `--obfuscate`，在发生异常时，堆栈跟踪中的信息会更清晰、更容易直接与你的源代码对应起来，这将直接简化调试过程。

在不使用混淆的情况下，你仍然可以使用 `--split-debug-info` 选项将符号信息拆分到一个单独的文件夹中，这样做的好处是减小了发布包的大小而没有牺牲太多的可调试性。在需要解析堆栈跟踪信息时，你仍然可以使用这些符号信息文件来还原出清晰的调用栈。

建议两个都加上,实测加上 obfuscate 解析出来堆栈有行号 对排查 bug 更有利

安卓生成的.symbols 文件有好几个

app.android-arm64.symbols

app.android-arm.symbols

异常日志必须每一行开头是# 上报上去的不分行,需要手动换行 这样才能解析出来

```
#00 abs 0000007356924e13 virt 00000000004a1e13
_kDartIsolateSnapshotInstructions+0x26b893
#01 abs 0000007356a9a257 virt 0000000000617257
_kDartIsolateSnapshotInstructions+0x3e0cd7
```

bugly 上会显示异常的 cpu 架构

CPU 架构

arm64-v8a

v8 用 app.android-arm64.symbols

v7 架构用 app.android-arm.symbols

涉及原生部分

ios 上传 dSYM 安卓上传 mapping.txt 文件到 lbugy 就好了 和原生一样,bugly 会自动解析原生异常

封装的 flutter 基础框架: [https://gitee.com/kuaipai/jd\\_flutter](https://gitee.com/kuaipai/jd_flutter), 你可以参考