Flutter EasyLoading - 让全局 Toast/Loading 更简单

jianke11

✨**flutter_easyloading**: 一个简单易用的**Flutter**插件，包含23种loading动画效果、进度条展示、Toast展示。纯Flutter端实现，支持iOS、Android。

✨**开源地址**：https://github.com/huangjianke/flutter_easyloading

前言

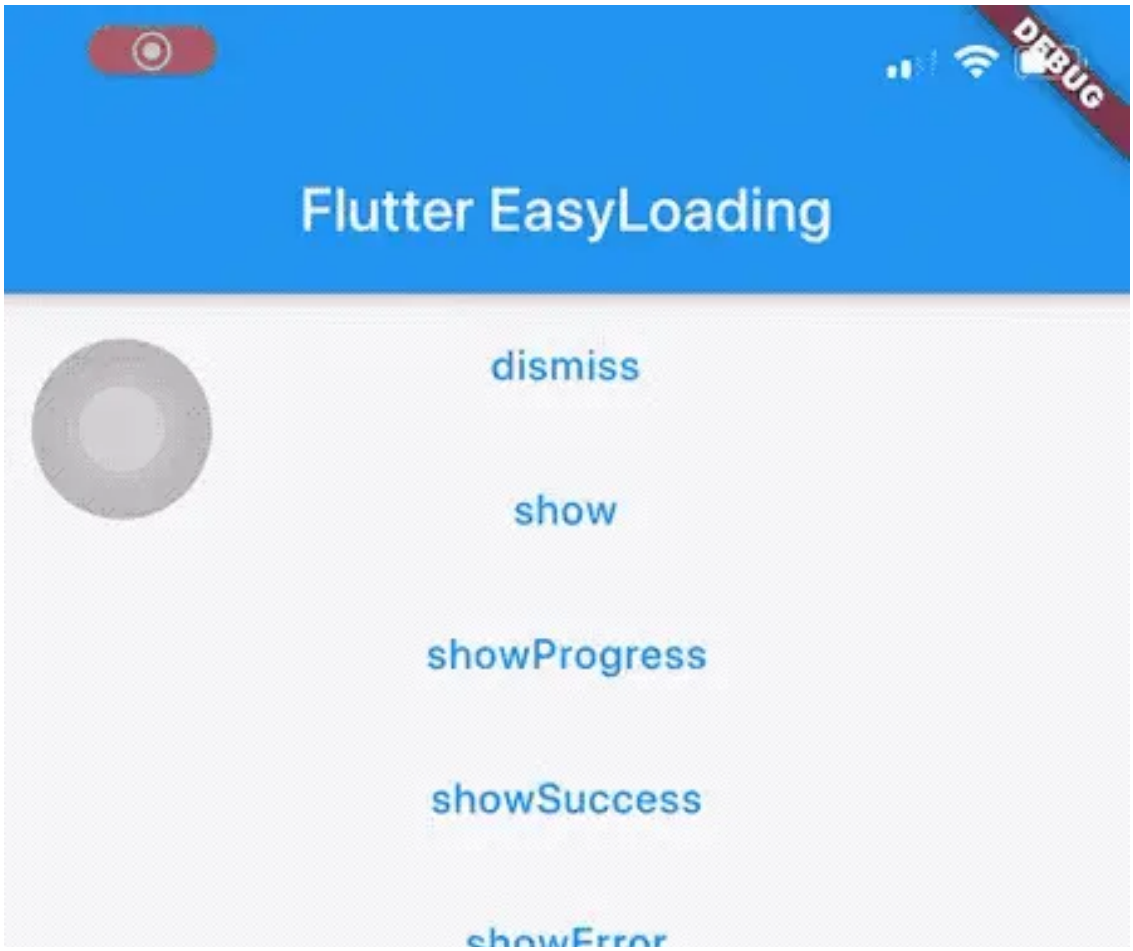
`Flutter`是`Google`在2017年推出的一套开源跨平台`UI`框架，可以快速地在`iOS`、`Android`和`Web`平台上构建高质量的原生用户界面。`Flutter`发布至今，不可谓不说是大受追捧，吸引了大批`App`原生开发者、`Web`开发者前赴后继的投入其怀抱，也正由于`Flutter`是跨平台领域的新星，总的来说，其生态目前还不是十分完善，我相信对于习惯了原生开发的同学们来说，找轮子肯定没有了那种章手就莱的感觉。比如说这篇文章即将讲到的，如何在`Flutter`应用内简单、方便的展示`Toast`或者`Loading`框呢?

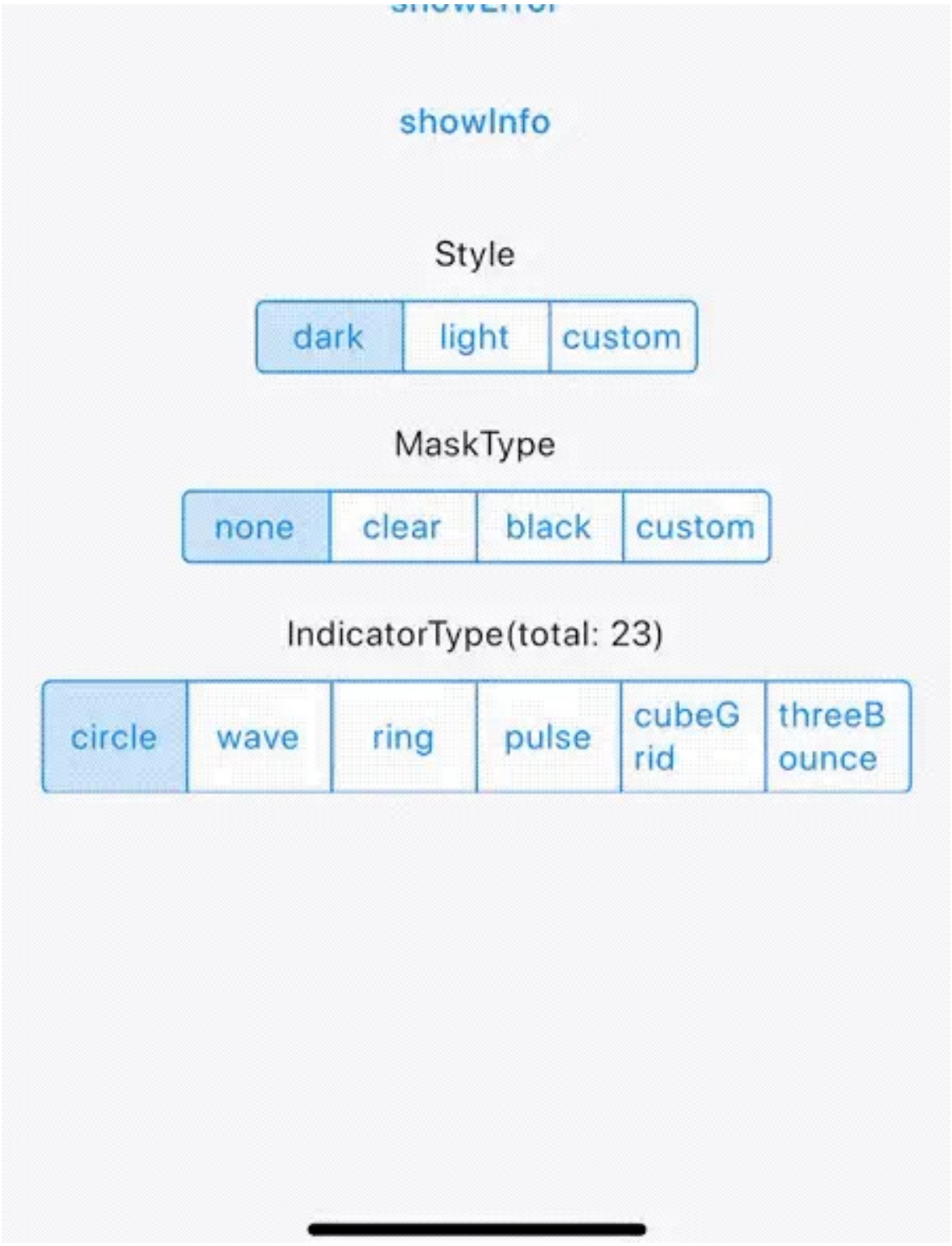
探索

起初，我也在pub上找到了几个比较优秀的插件：

- FlutterToast: 这个插件应该是很多刚入坑`Flutter`的同学们都使用过的，它依赖于原生，但对于UI层级的问题，最好在Flutter端解决，这样便于后期维护，也可以减少兼容性问题；
- flutter_oktoast: 纯`Flutter`端实现，调用方便。但缺少`loading`、进度条展示，仍可自定义实现；

试用过后，发现这些插件都或多或少不能满足我们的产品需求，于是便结合自己产品的需求来造了这么个轮子，也希望可以帮到有需要的同学们。效果预览：





flutter_easyloading

实现

showDialog 实现

先看看初期我们实现弹窗的方式`showDialog`，部分源码如下：

```
Future<T> showDialog<T>({
  @required BuildContext context,
  bool barrierDismissible = true,
  @Deprecated(
    'Instead of using the "child" argument, return the child from a closure '
    'provided to the "builder" argument. This will ensure that the BuildContext '
    'is appropriate for widgets built in the dialog. '
    'This feature was deprecated after v0.2.3.'
  )
  Widget child,
  WidgetBuilder builder,
  bool useRootNavigator = true,
})
```

这里有个必传参数`context`，想必接触过`Flutter`开发一段时间的同学，都会对`BuildContext`有所了解。简单来说`BuildContext`就是构建`Widget`中的应用上下文，是`Flutter`的重要组成部分。`BuildContext`只出现在两个地方：

- `StatelessWidget.build`方法中：创建`StatelessWidget`的`build`方法
- `State`对象中：创建`StatefulWidget`的`State`对象的`build`方

法中，另一个是State的成员变量

有关BuildContext更深入的探讨不在此文的探讨范围内，如果使用showDialog实现弹窗操作，那么我们所考虑的问题便是，如何方便快捷的在任意地方去获取BuildContext，从而实现弹窗。如果有同学恰巧也用了showDialog这种方式的话，我相信，你也会发现，在任意地方获取BuildContext并不是那么简单，而且会产生很多不必要的代码量。

那么，我们就只能使用这种体验极其不友好的方法么？

当然不是的，请继续看。

Flutter EasyLoading 介绍

Flutter EasyLoading是一个简单易用的Flutter插件，包含23种loading动画效果、进度条展示、Toast展示。纯Flutter端实现，兼容性好，支持iOS、Android。先简单看下如何使用Flutter EasyLoading。

安装

将以下代码添加到您项目中的 pubspec.yaml 文件:

```
dependencies:
  flutter_easyloading: ^1.1.0 // 请使用最新版
```

导入

```
import 'package:flutter_easyloading/
flutter_easyloading.dart';
```

如何使用

首先, 使用 `FlutterEasyLoading` 组件包裹您的App组件:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    /// 子组件通常为 [MaterialApp] 或者 [CupertinoApp].
    /// 这样做是为了确保 loading 组件能覆盖在其他组件之上.
    return FlutterEasyLoading(
      child: MaterialApp(
        title: 'Flutter EasyLoading',
        theme: ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: MyHomePage(title: 'Flutter EasyLoading'),
      ),
    );
  }
}
```

然后, 请尽情使用吧:

```
EasyLoading.show(status: 'loading...');
```

```
EasyLoading.showProgress(0.3, status: 'downloading...');

EasyLoading.showSuccess('Great Success!');

EasyLoading.showError('Failed with Error');

EasyLoading.showInfo('Useful Information.');

EasyLoading.dismiss();
```

自定义样式

首先，我们看下`Flutter EasyLoading`目前支持的自定义属性：

```
/// loading的样式, 默认[EasyLoadingStyle.dark].
EasyLoadingStyle loadingStyle;

/// loading的指示器类型, 默认
[EasyLoadingIndicatorType.fadingCircle].
EasyLoadingIndicatorType indicatorType;

/// loading的遮罩类型, 默认[EasyLoadingMaskType.none].
EasyLoadingMaskType maskType;

/// 文本的对齐方式 , 默认[TextAlign.center].
TextAlign textAlign;

/// loading内容区域的内边距.
EdgeInsets contentPadding;

/// 文本的内边距.
EdgeInsets textPadding;
```

```
/// 指示器的大小，默认40.0.
double indicatorSize;

/// loading的圆角大小，默认5.0.
double radius;

/// 文本大小，默认15.0.
double fontSize;

/// 进度条指示器的宽度，默认2.0.
double progressWidth;

/// [showSuccess] [showError] [showInfo]的展示时间，默认
2000ms.
Duration displayDuration;

/// 文本的颜色，仅对[EasyLoadingStyle.custom]有效.
Color textColor;

/// 指示器的颜色，仅对[EasyLoadingStyle.custom]有效.
Color indicatorColor;

/// 进度条指示器的颜色，仅对[EasyLoadingStyle.custom]有效.
Color progressColor;

/// loading的背景色，仅对[EasyLoadingStyle.custom]有效.
Color backgroundColor;

/// 遮罩的背景色，仅对[EasyLoadingMaskType.custom]有效.
Color maskColor;

/// 当loading展示的时候，是否允许用户操作.
bool userInteractions;
```

```
/// 展示成功状态的自定义组件
Widget successWidget;

/// 展示失败状态的自定义组件
Widget errorWidget;

/// 展示信息状态的自定义组件
Widget infoWidget;
```

因为 `EasyLoading` 是一个全局单例, 所以我们可以在任意一个地方自定义它的样式:

```
EasyLoading.instance
  ..displayDuration = const Duration(milliseconds: 2000)
  ..indicatorType = EasyLoadingIndicatorType.fadingCircle
  ..loadingStyle = EasyLoadingStyle.dark
  ..indicatorSize = 45.0
  ..radius = 10.0
  ..backgroundColor = Colors.green
  ..indicatorColor = Colors.yellow
  ..textColor = Colors.yellow
  ..maskColor = Colors.blue.withOpacity(0.5);
```

更多的指示器动画类型可查看 flutter_spinkit showcase

可以看到，`Flutter EasyLoading`的集成以及使用相当的简单，而且有丰富的自定义样式，总会有你满意的。

接下来，我们来看看`Flutter EasyLoading`的代码实现。

Flutter EasyLoading 的实现

本文将通过以下两个知识点来介绍`Flutter EasyLoading`的主要实现过程及思路：

- `Overlay`、`OverlayEntry`实现全局弹窗
- `CustomPaint`与`Canvas`实现圆形进度条绘制

Overlay、OverlayEntry 实现全局弹窗

先看看官方关于`Overlay`的描述：

```
/// A [Stack] of entries that can be managed independently.
///
/// Overlays let independent child widgets "float" visual
elements on top of
/// other widgets by inserting them into the overlay's
[Stack]. The overlay lets
/// each of these widgets manage their participation in the
overlay using
/// [OverlayEntry] objects.
///
/// Although you can create an [Overlay] directly, it's
most common to use the
/// overlay created by the [Navigator] in a [WidgetsApp] or
a [MaterialApp]. The
/// navigator uses its overlay to manage the visual
appearance of its routes.
///
/// See also:
```

```
///
///  * [OverlayEntry].
///  * [OverlayState].
///  * [WidgetsApp].
///  * [MaterialApp].
class Overlay extends StatefulWidget {}
```

也就是说，`Overlay`是一个`Stack`的`Widget`，可以将`OverlayEntry`插入到`Overlay`中，使独立的`child`窗口悬浮于其他`Widget`之上。利用这个特性，我们可以用`Overlay`将`MaterialApp`或`CupertinoApp`包裹起来，这样做的目的是为了确保`loading`组件能覆盖在其他组件之上，因为在`Flutter`中只会存在一个`MaterialApp`或`CupertinoApp`根节点组件。(注：这里的做法参考于flutter_oktoast插件，感谢)。

另外，这样做的目的还可以解决另外一个核心问题：将`context`缓存到内存中，后续所有调用均不需要提供`context`。实现如下：

```
@override
Widget build(BuildContext context) {
  return Directionality(
    child: Overlay(
      initialEntries: [
        OverlayEntry(
          builder: (BuildContext _context) {
            // 缓存 context
            EasyLoading.instance.context = _context;
```

```
            // 这里的child必须是MaterialApp或CupertinoApp
            return widget.child;
          },
        ),
      ],
    ),
    textDirection: widget.textDirection,
  );
}
```

```
// 创建OverlayEntry
OverlayEntry _overlayEntry = OverlayEntry(
  builder: (BuildContext context) => LoadingContainer(
    key: _key,
    status: status,
    indicator: w,
    animation: _animation,
  ),
);

// 将OverlayEntry插入到Overlay中
// 通过Overlay.of()我们可以获取到App根节点的Overlay
Overlay.of(_getInstance().context).insert(_overlayEntry);

// 调用OverlayEntry自身的remove()方法，从所在的Overlay中移除自己
_overlayEntry.remove();
```

`Overlay`、`OverlayEntry`的使用及理解还是很简单，我们也可以再更多的使用场景使用他们，比如说，类似`PopupWindow`的弹窗效果、全局自定义`Dialog`弹窗等等。只要灵活运用，我们可以实现很多我们想要的效果。

`CustomPaint`与`Canvas`实现圆形进度条绘制

几乎所有的UI系统都会提供一个自绘UI的接口，这个接口通常会提供一块2D画布Canvas，Canvas内部封装了一些基本绘制的API，我们可以通过Canvas绘制各种自定义图形。在Flutter中，提供了一个CustomPaint组件，它可以结合一个画笔CustomPainter来实现绘制自定义图形。接下来我将简单介绍下圆形进度条的实现。

我们先来看看CustomPaint构造函数：

```
const CustomPaint({
  Key key,
  this.painter,
  this.foregroundPainter,
  this.size = Size.zero,
  this.isComplex = false,
  this.willChange = false,
  Widget child,
})
```

- painter: 背景画笔，会显示在子节点后面;
- foregroundPainter: 前景画笔，会显示在子节点前面
- size：当child为null时，代表默认绘制区域大小，如果有child则忽略此参数，画布尺寸则为child尺寸。如果有child但是想指定画布为特定大小，可以使用SizeBox包裹CustomPaint实现。
- isComplex：是否复杂的绘制，如果是，Flutter会应用

一些缓存策略来减少重复渲染的开销。

- willChange：和`isComplex`配合使用，当启用缓存时，该属性代表在下一帧中绘制是否会改变。

可以看到，绘制时我们需要提供前景或背景画笔，两者也可以同时提供。我们的画笔需要继承`CustomPainter`类，我们在画笔类中实现真正的绘制逻辑。

接下来，我们看下怎么通过`CustomPainter`绘制圆形进度条：

```
class _CirclePainter extends CustomPainter {
  final Color color;
  final double value;
  final double width;

  _CirclePainter({
    @required this.color,
    @required this.value,
    @required this.width,
  });

  @override
  void paint(Canvas canvas, Size size) {
    final paint = Paint()
      ..color = color
      ..strokeWidth = width
      ..style = PaintingStyle.stroke
      ..strokeCap = StrokeCap.round;
    canvas.drawArc(
      Offset.zero & size,
      -math.pi / 2,
```

```
      math.pi * 2 * value,
      false,
      paint,
    );
  }

  @override
  bool shouldRepaint(_CirclePainter oldDelegate) => value !
= oldDelegate.value;
}
```

从上面我们可以看到，`CustomPainter`中定义了一个虚函数`paint`:

```
void paint(Canvas canvas, Size size);
```

这个函数是绘制的核心所在，它包含了以下两个参数：

- canvas: 画布，包括各种绘制方法, 如`drawLine(画线)`、`drawRect(画矩形)`、`drawCircle(画圆)`等
- size: 当前绘制区域大小

画布现在有了，那么接下来我们就需要一支画笔了。`Flutter`提供了`Paint`类来实现画笔。而且可以配置画笔的各种属性如粗细、颜色、样式等，比如：

```
final paint = Paint()
  ..color = color // 颜色
  ..strokeWidth = width // 宽度
```

```
  ..style = PaintingStyle.stroke
  ..strokeCap = StrokeCap.round;
```

最后，我们就是需要使用`drawArc`方法进行圆弧的绘制了：

```
canvas.drawArc(
  Offset.zero & size,
  -math.pi / 2,
  math.pi * 2 * value,
  false,
  paint,
);
```

到此，我们就完成了进度条的绘制。另外我们也需要注意下绘制性能问题。好在类中提供了重写`shouldRepaint`的方法，这个方法决定了画布什么时候会重新绘制，在复杂的绘制中对提升绘制性能是相当有成效的。

```
@override
bool shouldRepaint(_CirclePainter oldDelegate) => value != oldDelegate.value;
```

结语

毫无疑问，`Flutter`的前景是一片光明的，也许现在还存在诸多问题，但我相信更多的人会愿意陪着`Flutter`一起成长。期待着`Flutter`的生态圈的完善。后期我也会逐步完善`Flutter`

`EasyLoading`，期待您的宝贵意见。

最后，希望`Flutter EasyLoading`对您有所帮助。