链滴

Flutter 的一些知识点

作者：devcui

原文链接：https://ld246.com/article/1630246837383

来源网站：链滴

flutter

1.intro

flutter 本质是 widget 树

- Text: 格式文本
- Row: 水平布局
- Column: 垂直布局
- Stack: 线性布局，Positioned定位
- Container: 矩阵，BoxDecoration装饰Container
- Expanded: 填充剩余空间
- Navigator: 管理Widget栈
- Scaffold: Material 的一种布局结构
- GestureDetector: 识别用户手势,onTap回调
- StatefulWidget: 有状态控件,实现createState返回一个继承自State<WidgetType的状态型控件
- State: 可以通过给action来分层，Widget都存在State里，通过AcationWidget修改State里的值达到渲染DisplayWdiget的目的，一个负责渲染/发送Action，一个负责渲染数据

2.layout

- 数据结构为树
- 所有layout本质都是 widget
- 所有布局 widget都含有child和children属性
  - child: 包含一个子widget
  - children: 含有多个子项,布局子项，直至最后布局子项包含一个可见Widget
- 纵向用Column横向用Row 可嵌套
  - AxisAligment: 表示对齐
  - Expanded:可以自适应布局高宽
    - flex: 弹性系数
- Container: 可扩展 padding,margins,borders,background color等其他装饰元素decoration
- GridView: 可以做滚动网格
- ListView: 滚动列表
- Stack: 栈，先进后出，也就是后面的Widget盖到前一个Widget上，形成一个Widget栈
- Card是 Material中的一个组件
- EdgeInsets: 间距

自适应: 根据设备发生变化。

响应式: 根据屏幕大小发生变化。

自适应应用

- Single Child
  - Align: 子级内部对齐
  - AspectRatio: 为子级指定比例
  - ConstrainedBox: 对子级施加最大最小尺寸限制
  - CustomSingleChildLayout: 代理方法对子级进行定位
  - Expanded,Flexble: Row,Column填充剩余空间
  - FractionallySizedBox: 基于剩余控件比例限定子级大小
  - LayoutBuilder: 让子级可以基于父级的尺寸重新调整其布局
  - SingleChildScrollView: 为单一子级添加滚动
- Multi Child
  - Column,Row,Flex
  - CustomMultiChildLayout: 代理多个子级定位
  - Flow: 类似 arrow_up
  - ListView,GridView,CustomScrollView: 为所有子级添加滚动
  - Stack: 基于Stack边界对多个子级定位
  - Table: 表格布局
  - Wrap: 将子级顺序显示在多行/多列内

视觉密度

widget的疏密程度，紧凑程度等意思，其实就是一个动态的值来自动调整宽高/间距什么的

- VisualDensity

```
double densityAmt = enableTouchMode ? 0.0 : -1.0;
// 水平竖直
VisualDensity density = VisualDensity(horizontal: density, vertical: density);
return MaterialApp(
// 应用到MaterialApp主题
theme: ThemeData(visualDensity: density),
...
);
// 使用主题的疏密度
VisualDensity density = Theme.of(context).visualDensity;
return Padding(
```

padding: EdgeInsets.all(Insets.large + density.vertical * 4),
child: ...
);

基于Context的布局

- MediaQuery: 媒体查询，给了一些分界点

ScreenType getFormFactor(BuildContext context) {
// Use .shortestSide to detect device type regardless of orientation
double deviceWidth = MediaQuery.of(context).size.shortestSide;
if (deviceWidth > FormFactor.desktop) return ScreenType.Desktop;
if (deviceWidth > FormFactor.tablet) return ScreenType.Tablet;
if (deviceWidth > FormFactor.handset) return ScreenType.Handset;
return ScreenType.Watch;
}

以下是反转屏幕的一个判断，可以基于媒体查询判断是否反转，从顶传入子，使得整个UI做出屏幕翻的响应

```
bool isHandset = MediaQuery.of(context).size.width < 600;
return Flex(
  children: [...],
  direction: isHandset ?
    Axis.vertical :
    Axis.horizontal
);
```

layout builder

layout builder和之前不同的是，提供了一些对象，让你更好的做出判断和使用

```
Widget foo = LayoutBuilder(builder: (_, constraints, __){
 bool useVerticalLayout = constraints.maxWidth < 400.0;
 return Flex(
     children: [...],
     direction: useVerticalLayout ?
     Axis.vertical : Axis.horizontal
 );
});
```

设备细分

```
// Platform对象判断设备类型
bool get isMobileDevice => !kIsWeb && (Platform.isIOS || Platform.isAndroid);
// macos windows
bool get isDesktopDevice =>
    !kIsWeb && (Platform.isMacOS || Platform.isWindows || Platform.isLinux);
```

```
// browser
bool get isMobileDeviceOrWeb => kIsWeb || isMobileDevice;
bool get isDesktopDeviceOrWeb => kIsWeb || isDesktopDevice;
```

input

- scrollView,ListView: 自动支持 onPointerSingal.PointerScrollEvent滑轮输入
- 实现自定义滑动: Listener Widget的onPointerSingal.PointerScrollEvent
- FocusableActionDetector: 在元素外包裹，处理焦点，悬浮，鼠标跟随
- RawKeyboardListener: 监听键盘事件
- Shortcuts: 直接绑定快捷键
- RawKeyboard.instance.addListener(_handleKey); 全局监听键盘事件

```
// 单一快捷键是否按下了
static bool isKeyDown(Set<LogicalKeyboardKey> keys) {
return
keys.intersection(RawKeyboard.instance.keysPressed).isNotEmpty;
}
void _handleKey(event){
if (event is RawKeyDownEvent) {
// 两个shift组合键
bool isShiftDown = isKeyDown({
LogicalKeyboardKey.shiftLeft,
LogicalKeyboardKey.shiftRight,
});
if (isShiftDown && event.logicalKey == LogicalKeyboardKey.keyN) {
_createNewItem();
}
}
}
```

鼠标

```
MouseRegion
return MouseRegion(
// 进入
  onEnter: (_) => setState(() => _isMouseOver = true),
// 离开
onExit: (_) => setState(() => _isMouseOver = false),
// 悬停
onHover: (PointerHoverEvent e) => print(e.localPosition),
  child: ...,
);
```

用户期望

- ScrollBar: 根据平台切换状态

```
// 不同设备的输入是不同的
static bool get isMultiSelectModifierDown {
bool isDown = false;
// 使用DeviceOS来判断设备
if (DeviceOS.isMacOS) {
isDown = isKeyDown([LogicalKeyboardKey.metaLeft, LogicalKeyboardKey.metaRight]);
} else {
isDown = isKeyDown([LogicalKeyboardKey.controlLeft, LogicalKeyboardKey.controlRight]);
}
return isDown;
}
```

- SelectableText: 文字是否可以被选中
- SelectableText.rich(TexSpan): 富文本选中
- Tooltip: 悬停

布局约束

首先，上层 widget 向下层 widget 传递约束条件。

然后，下层 widget 向上层 widget 传递大小信息。

最后，上层 widget 决定下层 widget 的位置。

长宽计算更像是 子向父报备，父汇集各个子然后计算，在向父父汇报

- ConstrainedBox: 从父级接收约束施加给子级
- Center(child: ConstrainedBox()): 允许控制
- UnconstrainedBox: 允许子级改变为任意大小
- OverflowBox: 可溢出，需要在本级做出限制

限制 外到内，外告知内可小于外宽松约束，外告知内必须变成某个大小严格约束

添加互动

- StatefulWidget: 和React一样
- 其实就是状态管理，类似React的State，这个State可以是自己的State，也可以是自己的Props，父级State传入过来，只不过这里依靠构造函数显示声明
- 当然可以从父级传回调函数在子级改变父级状态
- GestureDetector: 添加其他互动，什么是互动啊，就是用户的行为能改变数据(切换图标也算是数变动)的地方就是有互动，理解了吧

指定资源

```
flutter:
  assets:
    - assets/my_icon.png
    - assets/background.png


flutter:
  assets:
    - directory/
    - directory/subdirectory/

// 直接读文件了
Future<String> loadAsset() async {
  return await rootBundle.loadString('assets/config.json');
}

// 不同分辨率的图片，可FLutter可以对应不同的设备自动切换
.../my_icon.png
.../2.0x/my_icon.png
.../3.0x/my_icon.png

// 使用别的包的图像，也要声明出来
flutter:
  assets:
    - packages/fancy_backgrounds/backgrounds/background1.png
```

注意: 读取android/ios图片插件和方式不同，文件夹内有android,ios两个文件夹，需要添加各自的件，图标,预加载图也是如此

路由

安卓启动深度路由AndroidManifest.xml

```
<!-- Deep linking -->
<meta-data android:name="flutter_deeplinking_enabled" android:value="true" />
<intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="http" android:host="flutterbooksample.com" />
    <data android:scheme="https" />
</intent-filter>
```

配合adb测试

```
adb shell am start -a android.intent.action.VIEW \
    -c android.intent.category.BROWSABLE \
    -d "http://flutterbooksample.com/book/1"
```

IOS Info.plist

```
<key>FlutterDeepLinkingEnabled</key>
```

```
<true/>
<key>CFBundleURLTypes</key>
<array>
   <dict>
   <key>CFBundleTypeRole</key>
   <string>Editor</string>
   <key>CFBundleURLName</key>
   <string>flutterbooksample.com</string>
   <key>CFBundleURLSchemes</key>
   <array>
   <string>customscheme</string>
   </array>
   </dict>
</array>
```

配合xsrunxshell

```
xcrun simctl openurl booted customscheme://flutterbooksample.com/book/1
```

动画

pass

http

```
import 'package:http/http.dart' as http;

var url = Uri.parse('https://example.com/whatsit/create');
var response = await http.post(url, body: {'name': 'doodle', 'color': 'blue'});
print('Response status: ${response.statusCode}');
print('Response body: ${response.body}');

print(await http.read('https://example.com/foobar.txt'));
```

安卓需要在AndroidManifest.xml中添加需要访问的网络权限

```
<manifest xmlns:android...>
 ...
 <uses-permission android:name="android.permission.INTERNET" />
 <application ...
</manifest>
```

json

```
class User {
  final String name;
  final String email;

  User(this.name, this.email);
// 解析json
  User.fromJson(Map<String, dynamic> json)
      : name = json['name'],
        email = json['email'];
```

```
// 转成json
  Map<String, dynamic> toJson() => {
        'name': name,
        'email': email,
      };
}
```

或者

```
dependencies:
  # Your other regular dependencies here
  json_annotation: <latest_version>

dev_dependencies:
  # Your other dev_dependencies here
  build_runner: <latest_version>
  json_serializable: <latest_version>
```

引入依赖，使用注解

```
@JsonSerializable()
class User {
  User(this.name, this.email);

  String name;
  String email;

  /// A necessary factory constructor for creating a new User instance
  /// from a map. Pass the map to the generated `_$UserFromJson()` constructor.
  /// The constructor is named after the source class, in this case, User.
  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);

  /// `toJson` is the convention for a class to declare support for serialization
  /// to JSON. The implementation simply calls the private, generated
  /// helper method `_$UserToJson`.
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

- @JsonKey: 指定json名称
- @JsonSerializable(fieldRename: FieldRename.snake) : json风格
- @JsonKey(defaultValue: false),@JsonKey(required: true),@JsonKey(ignore: true): json校验
- flutter pub run build_runner build 使用这个命令配合注解生成,xxx.g.dart
- flutter pub run build_runner watch 持续生成
- @JsonSerializable(explicitToJson: true): 子类一起JSON

i18n

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations: # Add this line
```

```
    sdk: flutter         # Add this line


return const MaterialApp(
  title: 'Localizations Sample App',
  localizationsDelegates: [
    GlobalMaterialLocalizations.delegate,
    GlobalWidgetsLocalizations.delegate,
    GlobalCupertinoLocalizations.delegate,
  ],
  supportedLocales: [
    Locale('en', ''), // English, no country code
    Locale('es', ''), // Spanish, no country code
  ],
  home: MyHomePage(),
);
```

添加自定义国际化文件

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
  intl: ^0.17.0 # Add this line



# The following section is specific to Flutter.
flutter:
  generate: true # Add this line
```

然后添加arb配置

```
arb-dir: lib/l10n
template-arb-file: app_en.arb
output-localization-file: app_localizations.dart
```

arb文件

```
{
    "helloWorld": "Hello World!",
    "@helloWorld": {
      "description": "The conventional newborn programmer greeting"
    }
}
```

导入本地配置

```
localizationsDelegates: [
    AppLocalizations.delegate, // Add this line
``   ],
```

代码中使用

```
Text(AppLocalizations.of(context)!.helloWorld);
```