

-
- [为什么我们选择了Flutter？](#)
- [日常开发环境配置](#)
- [IDE配置](#)
- [Flutter代码调试](#)
- [第三方库使用注意事项](#)
- [Flutter组件速查表](#)
 - [基础组件](#)
 - [文本](#)
 - [Text](#)
 - [TextStyle](#)
 - [按钮](#)
 - [RaisedButton](#)
 - [FlatButton](#)
 - [TextButton](#)
 - [OutlineButton](#)
 - [图片和图标](#)
 - [Image](#)
 - [Icon](#)
 - [开关](#)
 - [Switch](#)
 - [复选框](#)
 - [Checkbox](#)
 - [单选组件](#)
 - [Radio](#)
 - [进度组件](#)
 - [进度条：LinearProgressIndicator](#)
 - [转圈圈：CircularProgressIndicator](#)
 -
 - [滑块组件](#)
 - [Slider](#)
 - [布局组件](#)
 - [布局流程](#)
 - [约束子组件大小](#)
 - [ConstrainedBox](#)
 - [SizedBox](#)
 - [UnconstrainedBox](#)
 - [FittedBox](#)
 - [线性布局](#)
 - [Row](#)
 - [Column](#)
 - [Flex](#)
 - [流式布局](#)
 - [Wrap](#)
 -
 - [层叠布局](#)


- Stack
- 自定义布局
 - Flow
- 单组件相对父组件定位
 - Align
 - Center
 - Padding
- 多种容器组合
 - Container
- 根据父组件约束值进行动态布局
 - LayoutBuilder
- 装饰组件
 - 背景、边框、渐变等装饰
 - DecoratedBox
 - 矩阵变换
 - Transform
 -
 - 旋转
 - RotatedBox
 - 裁剪
 - ClipOval
 - ClipRRect
 - ClipRect
 - ClipPath
 - CustomClipper
- 滚动组件
 - 单组件可滚动
 - SingleChildScrollView
 - 滚动列表
 - ListView
 - 滚动表格
 - GridView
 - 多Tab页面
 - PageView
 - TabBarView、TabBar
 - 合并多个滚动组件
 - CustomScrollView
- 功能型组件
 - 页面框架
 - Scaffold
 - AppBar
 -
 - Drawer
 - BottomNavigationBar
 - FloatingActionButton
 - SnackBar

- 对话框
 - showDialog
 - AlertDialog
 - SimpleDialog
 - Dialog
- 手势识别
 - 手机上常见操作手势
 - GestureDetector
 - 带水波效果的点击
 - InkWell
 - 拖动效果
 - Draggable
 - DragTarget
- Dart速查手册
 - 基本示例
 - 核心概念
 - 变量
 - 声明
 - final
 - const
 - 内置类型
 - Numbers（数值）
 - 两种数值类型
 - 数值与字符串转换
 - Strings（字符串）
 - Booleans（布尔值）
 - Lists（列表）
 - Maps
 - Runes
 - Symbols
 - Functions（方法）
 - 基本用法
 - 定义
 - 表达式形式
 - 可选参数
 - 返回值
 - 函数式编程
 - 方法也是对象
 - Lambda
 - 测试函数是否相等
 - 可调用的类（类模拟方法）
 - Operators（操作符）
 - 操作符优先级
 - 左操作数决定操作符
 - 分类
 - 算术操作符

- 相等相关的操作符
- 类型判定操作符
- 赋值操作符
- 条件表达式
- 级联操作符
- 流程控制
 - 分支
 - 循环
 - 断言
- 异常
 - 相关用法
 - 未捕获异常
- 类
 - 构造
 - 成员变量
 - 抽象类
 - 隐式接口
 - 枚举类型
 - mixins(混入)
- 泛型
 - 使用集合字面量
 - 在构造函数中使用泛型
 - 运行时特性（与Java不同）
 -
- 库和可见性
 - 使用库
 - 指定库前缀解决冲突
 - 导入库的一部分
 - 延迟载入库
- 异步支持
 - `async\await`
 - 生成器
 - 设计目标
 - 同步生成器：`sync*`
 - 异步生成器：`async*`
- 线程模型
 - Dart VM中的线程
 - Dart单线程流程
 - 微任务队列
 - 事件队列
 - 作者：程序员老刘

如果您觉得内容有不妥之处，或者希望添加内容，请微信联系我。



程序员老刘 



扫一扫上面的二维码图案，加我微信

为什么我们选择了Flutter？

使用过React Native、Weex这类跨平台框架，会有两个明显的感受：一是，需要花费大量的时间和精力，用于解决Android和iOS不同平台上的兼容性问题，比如同一套代码在两端UI效果不同。二是，流畅度并不尽如人意，特别是碰到JS和原生频繁通信的场景。

而Flutter通过自绘制架构设计，从根本上解决了上面两个问题。自绘制不需要借助原生控件，在Android和iOS上使用相同的绘制引擎，因此带来了极高的两端一致性体验。同时也免去了通过JSCore中转处理逻辑的通信开销。

老刘认为，在我们选择跨平台框架时，首要的问题是不同平台的一致性。这才是跨平台开发的最大意义所在。

日常开发环境配置

官网：[Install | Flutter](#)

中文：[在 Windows 操作系统上安装和配置 Flutter 开发环境](#) | [Flutter 中文文档](#) | [Flutter 中文开发者网站](#)（社区维护）

注意一

优先到官网核对环境配置问题，这里保证是最实时的。

注意二

国内网络环境要修改环境变量

```
export PUB_HOSTED_URL=https://pub.flutter-io.cn
export FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.cn
```

日常查询第三方库，也可以访问 <https://pub.flutter-io.cn/>

注意三

团队开发通常不会频繁更新Flutter版本，建议下载Flutter SDK的压缩包直接使用，使用git方式更新可能不稳定。同时，也能保证所有成员都使用相同的版本。

[Flutter SDK 版本列表](#) | [Flutter 中文文档](#) | [Flutter 中文开发者网站](#)

下载时请确认对应的系统和版本：

Windows macOS Linux

Stable channel (Windows)

请从下列列表中选择：

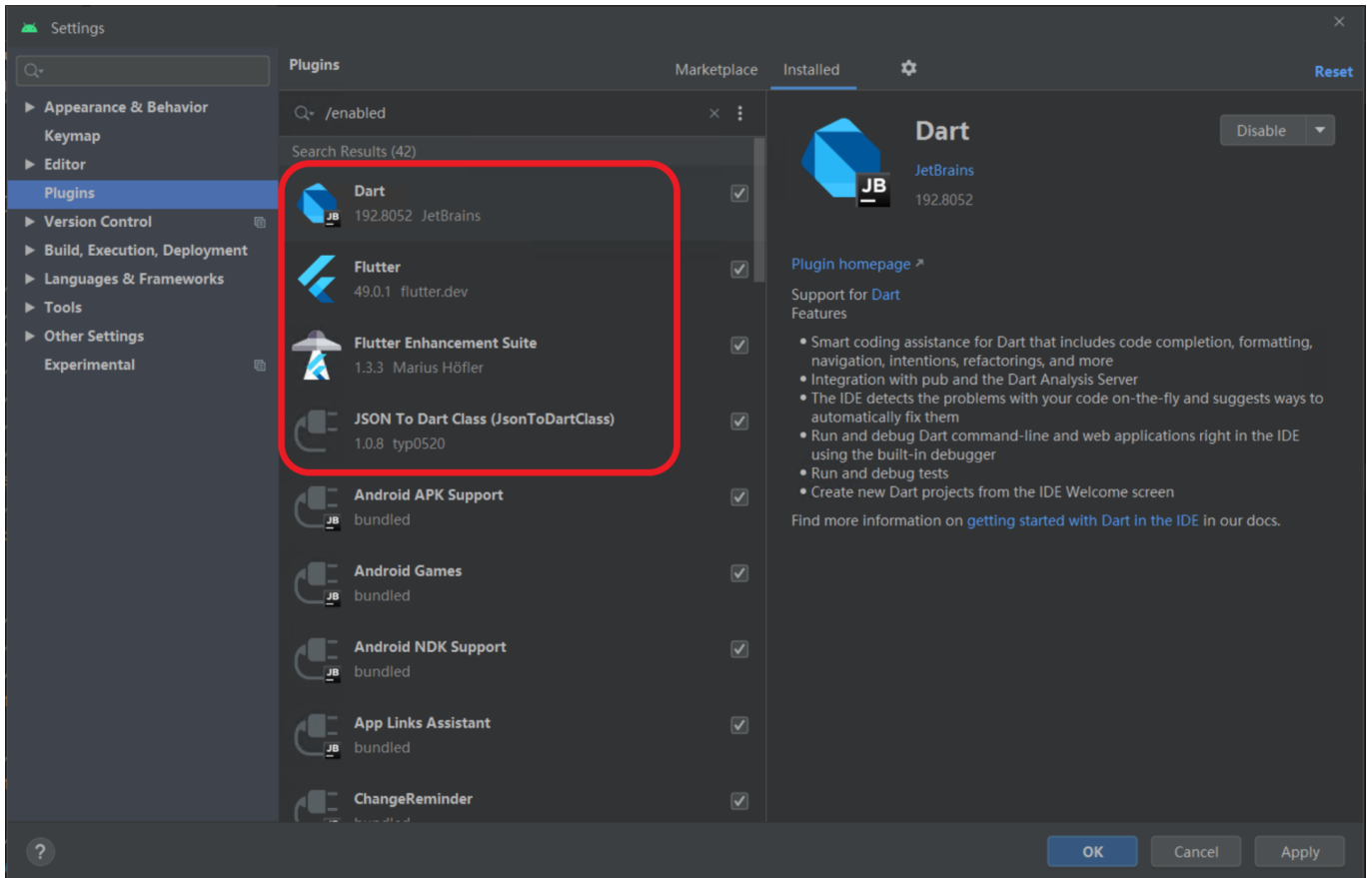
版本	Ref	发布日期
2.8.1	77d935a	2021/12/17
2.8.0	cf44000	2021/12/9
2.5.3	1811693	2021/10/16
2.5.2	3595343	2021/10/1
2.5.1	ffb2ece	2021/9/18
2.5.0	4cc385b	2021/9/9

IDE配置

目前AndroidStudio和VSCode都有很多用户，各有特点。

我们团队开发三年的经验是大规模项目开发AndroidStudio更好用。查看第三方源代码或者简单的实验，VSCode启动更快速。

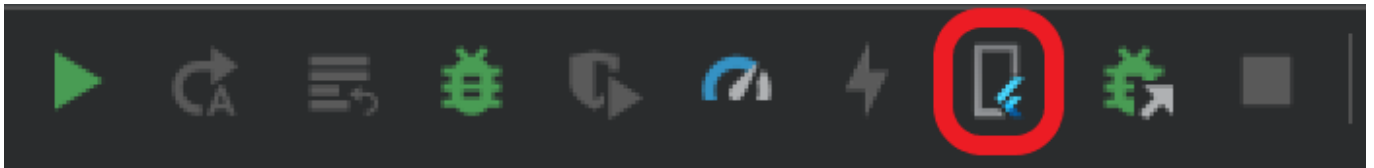
AndroidStudio上团队所有人都会安装以下几个插件，其它的看个人爱好



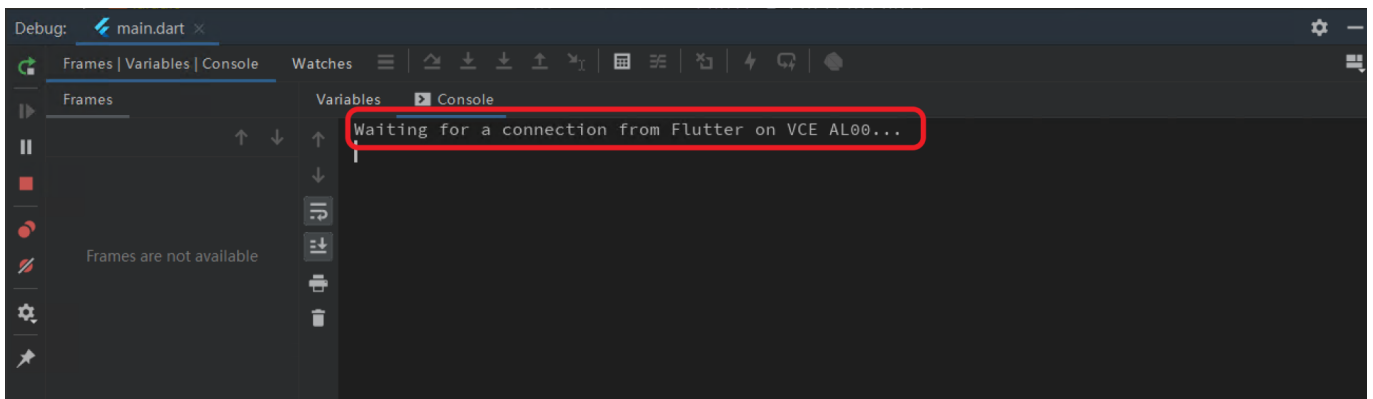
Flutter代码调试

如非调试App的启动过程，尽量不要直接用调试按钮运行，推荐的方法是正常运行，进入需要调试的页面后通过Attach的方式进入调试。

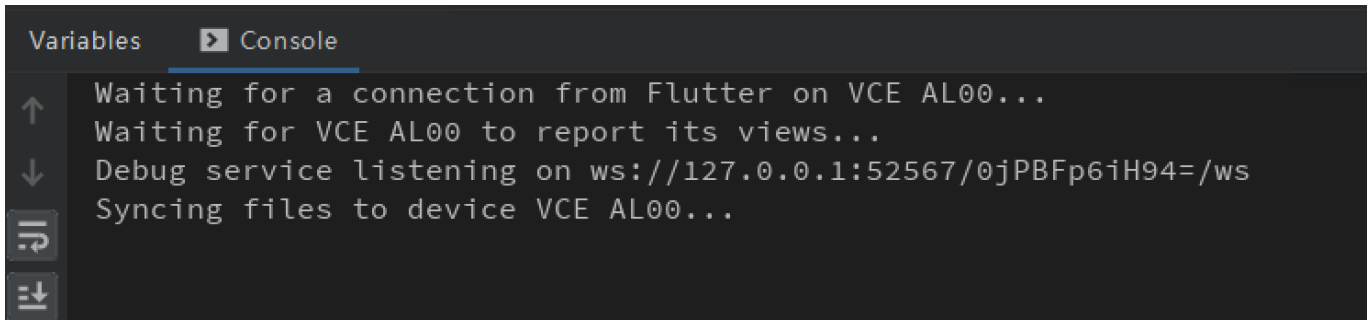
Flutter的attach按钮如下：



Android端可能出现attach后一直连接不上的问题，就是一直停留在下面这个提示。



这时可以直接在手机上杀掉App，重新进入，就可以连接成功了。成功后有如下提示：



第三方库使用注意事项

通常使用第三方库的方式如下：

```
flutter_bloc: ^6.1.1
```

这种方式会自动更新最新的小版本。但是使用中发现有一小部分第三方库，在小版本更新时会改动逻辑甚至接口，造成编译甚至运行期的错误。因此对于常规项目我们建议使用固定版本的方式，这样可以保证测试过程和发版使用的第三方库完全一致。

```
flutter_bloc: 6.1.1
```

Flutter组件速查表

本速查表不是各个组件的详细介绍。主要用于需要实现对应功能时，可以快速找到相关功能的组件。查询组件具体功能最方便的方法，在IDE中把组件代码贴进去，鼠标悬停看提示，或者直接点进去看注释。

基础组件

文本

Text

展示简单样式的文本。样式由TextStyle指定。

```
Text("Hello world",  
  textAlign: TextAlign.left,  
);
```

TextStyle

指定文本的大小、颜色、字体、粗细等样式

```
Text("Hello world",
  style: TextStyle(
    color: Colors.blue,
    fontSize: 18.0,
    fontFamily: "Courier",
  ),
);
```

按钮

Flutter提供了多种按钮组件（RaisedButton、 FlatButton、 OutlineButton、 DropdownButton、 RawMaterialButton、 PopupMenuButton、 IconButton、 BackButton、 CloseButton、 ButtonBar、 ToggleButtons）

它们都是直接或间接封装了RawMaterialButton，因此大部分的通用属性相同。

所有Material 库中的按钮都有两个相同点：

- 1、点击会有水波动画
- 2、通过onPressed属性设置点击后的回调，如果该属性为空，则按钮无法点击

RaisedButton

Material 风格“凸起”的按钮

```
RaisedButton(
  child: Text("点击"),
  onPressed: () => showToast("RaisedButton")
)
```

FlatButton

扁平风格按钮

```
FlatButton(
  child: Text("点击"),
  onPressed: () => showToast("RaisedButton")
)
```

TextButton

文本按钮，默认背景透明并不带阴影。

```
TextButton(  
  child: Text("点击"),  
  onPressed: () => showToast("RaisedButton")  
)
```

OutlineButton

带边框的按钮，不带阴影且背景透明。

```
OutlineButton(  
  child: Text("点击"),  
  onPressed: () => showToast("RaisedButton")  
)
```

图片和图标

Image

加载网络图片

```
Image.network(  
  "https://storage.googleapis.com/cms-storage-bucket/ed2e069ee37807f5975a.jpg",  
  width: 100.0,  
)
```

加载项目中的图片

```
Image.asset("images/pic1.png",  
  width: 100.0,  
)
```

Icon

通过将图标做成字体文件，指定不同的字符显示不同的图片。

相对于普通图片，图标的体积更小，可以当做文本和普通文本混排，也可以像普通文本一下设置颜色、大小等参数。

```
Icon(  
  Icons.add, // Material Design的字体图标  
  size: 40,  
  color: Colors.red,  
)
```

开关

Switch

```
Switch(  
  value: _isSelected, //存储当前状态的变量  
  onChanged: (value){  
    setState(() {  
      _isSelected=value;  
    });  
  },  
)
```

复选框

Checkbox

一个Checkbox代表一个可以选中的按钮

```
Checkbox(  
  value: _isSelected,  
  activeColor: Colors.red, //选中时的颜色  
  onChanged: (value){  
    setState(() {  
      _isSelected=value;  
    });  
  },  
)
```

单选组件

Radio

直接使用需要多个Radio，并且配合对应的Text等描述，每个Radio点击还需要更新其它Radio或者整个页面。因此建议在**项目内封装统一的单选控件**，不要直接使用。

```
Radio(  
  value: _value,  
  groupValue: _radioGroupValue,  
  onChanged: (value) {  
    setState(() {  
      _radioGroupValue = _value;  
    });  
  },  
)
```

进度组件

进度条 : LinearProgressIndicator

```
// 模糊进度条(会执行一个动画)
LinearProgressIndicator(
  backgroundColor: Colors.grey[200],
  valueColor: AlwaysStoppedAnimation(Colors.blue),
),
//进度条显示50%
LinearProgressIndicator(
  backgroundColor: Colors.grey[200],
  valueColor: AlwaysStoppedAnimation(Colors.blue),
  value: .5,
)
```

转圈圈 : CircularProgressIndicator

```
// 模糊进度条(会执行一个旋转动画)
CircularProgressIndicator(
  backgroundColor: Colors.grey[200],
  valueColor: AlwaysStoppedAnimation(Colors.blue),
)
//进度条显示50%，会显示一个半圆
CircularProgressIndicator(
  backgroundColor: Colors.grey[200],
  valueColor: AlwaysStoppedAnimation(Colors.blue),
  value: .5,
)
```

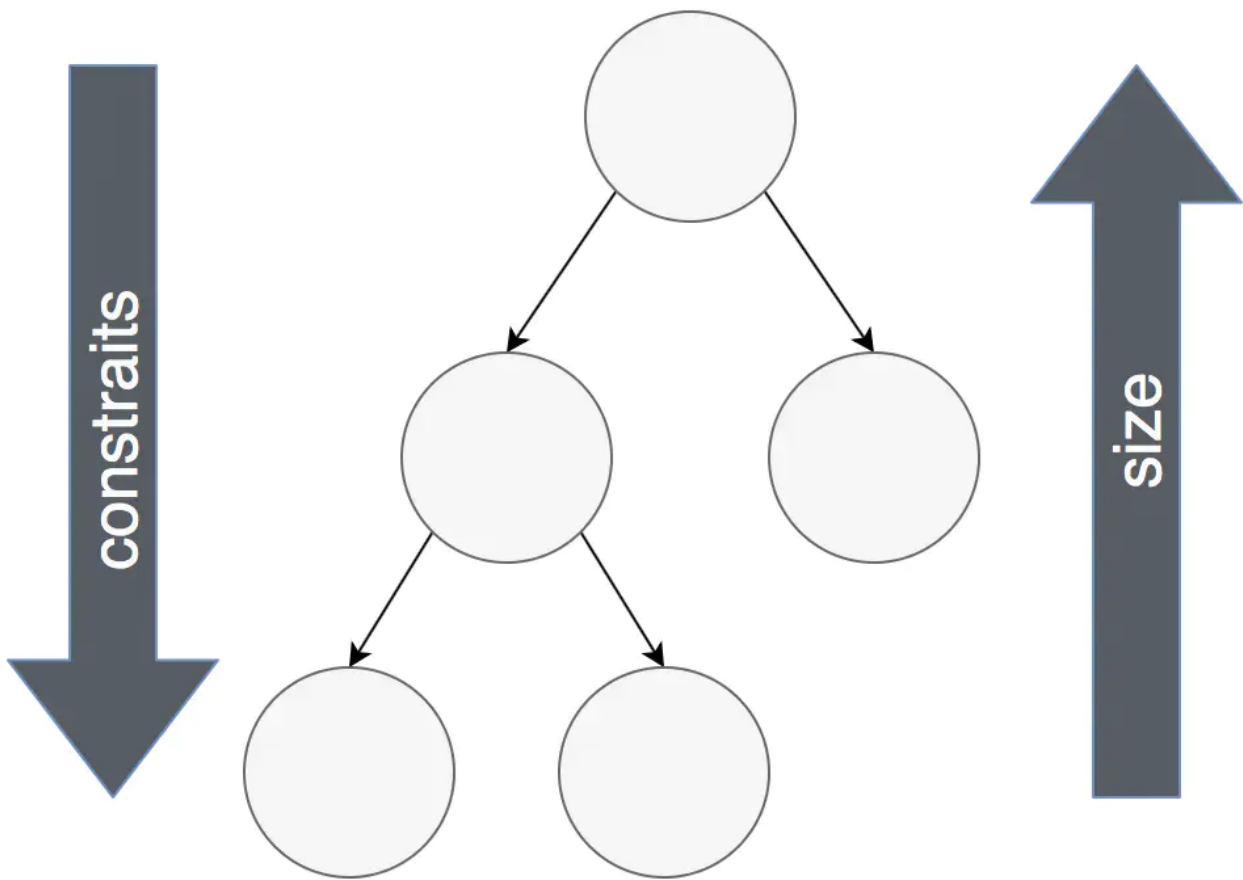
滑块组件

Slider

```
Slider(
  value: progressValue,
  min: 0.0,
  max: 100.0,
  divisions: 4, // 只能滑动到 0、25、50、75、100这几个值
  onChanged: (double){
    setState(() {
      progressValue=double.floorToDouble();//转成double
    });
  },
)
```

布局组件

布局流程



约束子组件大小

ConstrainedBox

```
ConstrainedBox(  
  constraints: BoxConstraints({  
    this.minWidth = 0.0, //最小宽度  
    this.maxWidth = double.infinity, //最大宽度  
    this.minHeight = 0.0, //最小高度  
    this.maxHeight = double.infinity //最大高度  
  }),  
  child: Container(  
    height: 5.0,  
    child: redBox ,  
  ),  
)
```

SizedBox

本质上是 ConstrainedBox的定制

```
    SizedBox(
      width: 80.0,
      height: 80.0,
      child: Container(
        height: 5.0,
        child: redBox ,
      ),
    )
```

UnconstrainedBox

让子组件不再受父组件大小约束，可以自由绘制

```
    ConstrainedBox(
      constraints: BoxConstraints(minWidth: 60.0, minHeight: 100.0), //父
      child: UnconstrainedBox( //“去除”父级限制
        child: ConstrainedBox(
          constraints: BoxConstraints(minWidth: 90.0, minHeight: 20.0), //子
          child: redBox,
        ),
      ),
    )
```

FittedBox

动态适配父组件大小。首先按照不约束进行子组件布局，获取子组件自身大小后根据指定的适配方式决定缩放还是裁剪子组件。

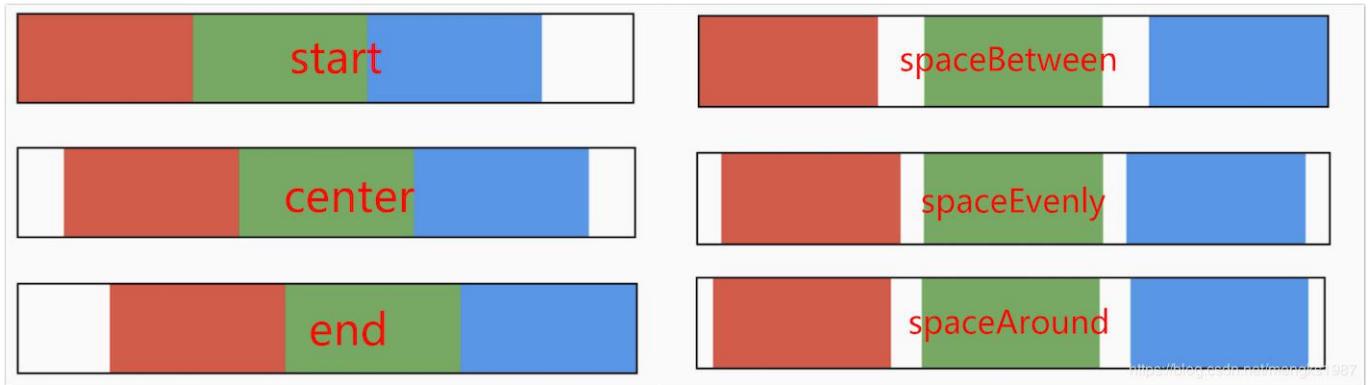
```
    Container(
      width: 50,
      height: 50,
      child: FittedBox(
        fit: BoxFit.contain, // 适配方式：按比例缩放
        child: FlutterLogo(size: 60,),
      ),
    )
```

线性布局

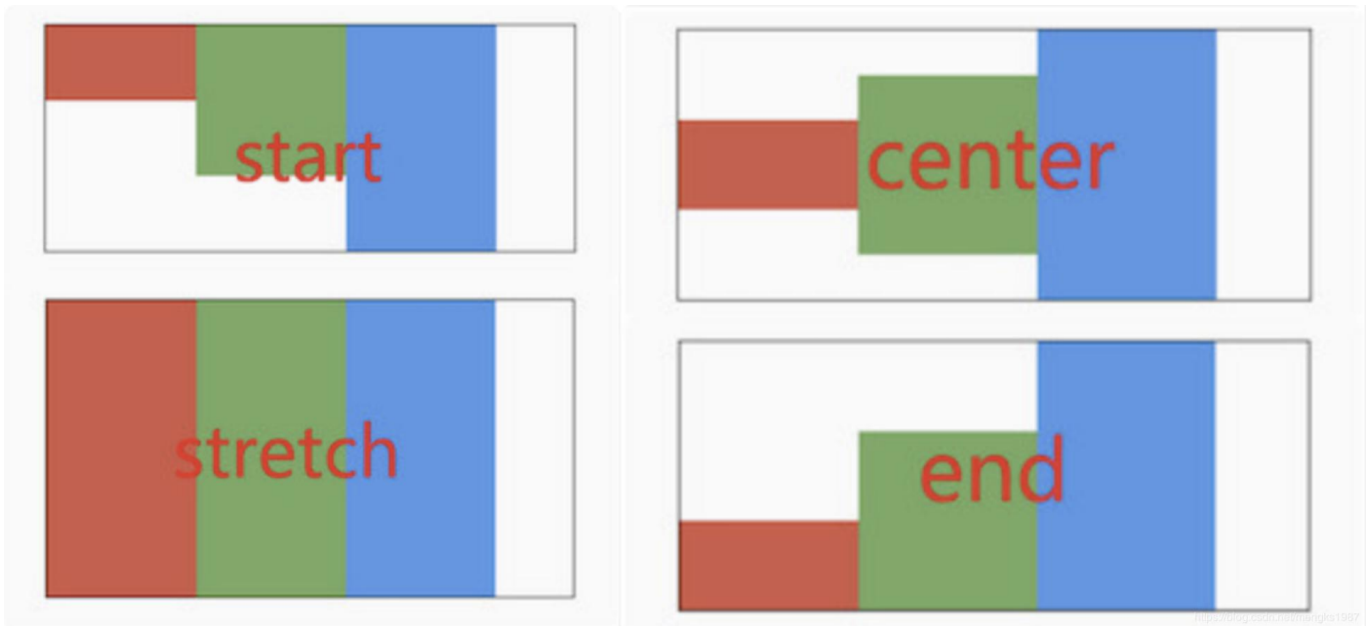
Row

横向

主轴对齐方式



交叉轴对齐方式



```
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Text(" hello world "),
    Text(" I am Jack "),
  ],
)
```

Column

纵向, 属性参考Row

```
Column(
  crossAxisAlignment: CrossAxisAlignment.center,
  children: <Widget>[
    Text("hi"),
    Text("world"),
  ],
)
```


Flex

按比例分配主轴方向空间

```
Flex( // 按照1:2分配空间
  direction: Axis.horizontal,
  children: <Widget>[
    Expanded(
      flex: 1,
      child: Container(
        height: 30.0,
        color: Colors.red,
      ),
    ),
    Expanded(
      flex: 2,
      child: Container(
        height: 30.0,
        color: Colors.green,
      ),
    ),
  ],
)
```

流式布局

Wrap

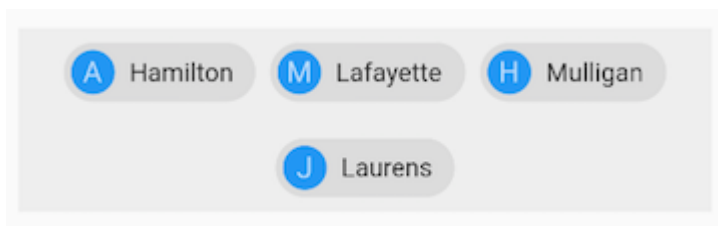
超出部分自动折行

```
Wrap(
  spacing: 8.0, // 主轴(水平)方向间距
  runSpacing: 4.0, // 纵轴(垂直)方向间距
  alignment: WrapAlignment.center, //沿主轴方向居中
  children: <Widget>[
    Chip(
      avatar: CircleAvatar(backgroundColor: Colors.blue, child: Text('A')),
      label: Text('Hamilton'),
    ),
    Chip(
      avatar: CircleAvatar(backgroundColor: Colors.blue, child: Text('M')),
      label: Text('Lafayette'),
    ),
    Chip(
      avatar: CircleAvatar(backgroundColor: Colors.blue, child: Text('H')),
      label: Text('Mulligan'),
    ),
    Chip(
```

```

        avatar: CircleAvatar(backgroundColor: Colors.blue, child: Text('J')),
        label: Text('Laurens'),
      ),
    ],
  )

```



运行结果：

层叠布局

Stack

子组件按照声明顺序覆盖上去。位置根据父组件定位。

```

Stack(
  alignment: Alignment.center, //默认对齐方式
  children: <Widget>[
    Container(
      child: Text("我是居中的"),
    ),
    Positioned( // 使用Positioned在Stack中定位
      left: 20.0,
      child: Text("我在左边"),
    ),
    Positioned(
      top: 20.0,
      child: Text("我在上边"),
    )
  ],
)

```

自定义布局

Flow

可自定义布局策略，使用复杂，建议优先考虑普通布局组件进行组合。

单组件相对父组件定位

定位单个组件在父组件中的相对位置

Align

```
Container(  
  height: 120.0,  
  width: 120.0,  
  child: Align(  
    alignment: Alignment.topRight, // 定位在父组件右上角  
    child: FlutterLogo(  
      size: 60,  
    ),  
  ),  
)
```

Center

相当于写死 alignment: Alignment.center的Align

```
Container(  
  height: 120.0,  
  width: 120.0,  
  child: Center(  
    child: FlutterLogo(size: 60,),  
  ),  
)
```

Padding

```
Padding(  
  //左边添加8像素空白  
  padding: const EdgeInsets.only(left: 8.0),  
  child: FlutterLogo(size: 60,),  
)
```

多种容器组合

Container

DecoratedBox、ConstrainedBox、Transform、Padding、Align等组件组合的一个多功能容器。同时实现多种容器的功能，方便减少代码层级。

```
Container({  
  this.alignment,  
  this.padding, //容器内补白, 属于decoration的装饰范围  
  Color color, // 背景色  
  Decoration decoration, // 背景装饰  
  Decoration foregroundDecoration, //前景装饰  
  double width, //容器的宽度
```

```
double height, //容器的高度
BoxConstraints constraints, //容器大小的限制条件
this.margin, //容器外补白, 不属于decoration的装饰范围
this.transform, //变换
this.child,
...
})
```

根据父组件约束值进行动态布局

不同设备上, 根据父组件传递的约束值, 进行不同的布局。

LayoutBuilder

```
LayoutBuilder(builder: (_, constraints) {
  return Text(""); // 这里可以根据constraints传递的约束值进行返回不同的布局
})
```

装饰组件

背景、边框、渐变等装饰

DecoratedBox

```
DecoratedBox(
  decoration: BoxDecoration(
    gradient: LinearGradient(colors:[Colors.red,Colors.orange.shade700]), //背景
    渐变
    borderRadius: BorderRadius.circular(3.0), //3像素圆角
    boxShadow: [ //阴影
      BoxShadow(
        color:Colors.black54,
        offset: Offset(2.0,2.0),
        blurRadius: 4.0
      )
    ],
  ),
  child: FlutterLogo(size: 60,),
)
```

矩阵变换

Transform

常见的实现**平移**、**旋转**、**缩放**等效果

注意：变换是在绘制阶段完成的，不会改变布局的结果。也就是说，控件的大小、位置等信息在布局阶段计算完成后不会随着变换而改变。

```
// 平移
Transform.translate(
  offset: Offset(10.0, 10.0),
  child: Text("Hello world"),
)

// 旋转
Transform.rotate(
  //旋转90度
  angle: math.pi/2 ,
  child: Text("Hello world"),
)

// 缩放
Transform.scale(
  scale: 1.5, //放大1.5倍
  child: Text("Hello world")
)
```

旋转

RotatedBox

和Transform.rotate的功能相似，但是RotatedBox是在布局阶段旋转，因此会改变控件的大小和位置。

```
RotatedBox(
  quarterTurns: 1, // 1个1/4圈，90度，顺时针
  child: Text("Hello world"),
)
```

裁剪

ClipOval

将矩形裁剪为内贴椭圆

```
ClipOval(
  child: FlutterLogo(size: 60,),
)
```

ClipRRect

裁剪为圆角矩形

```
ClipRRect(  
  borderRadius: BorderRadius.circular(5.0), // 圆角半径  
  child: FlutterLogo(size: 60,),  
)
```

ClipRect

裁减掉溢出的部分

```
ClipRect(  
  child: Align(  
    alignment: Alignment.topLeft,  
    widthFactor: .5, //宽度设为原来宽度一半，如果不裁剪，内容都能展示出来  
    child: avatar,  
  ),  
)
```

ClipPath

按照自定义的路径剪裁

```
ClipPath.shape(  
  shape: StadiumBorder(),  
  child: FlutterLogo(size: 60,),  
)
```

CustomClipper

自定义裁剪，继承CustomClipper，实现两个方法

```
class MyClipper extends CustomClipper<Rect> {  
  @override  
  // 返回裁剪区域  
  Rect getClip(Size size) => Rect.fromLTWH(10.0, 15.0, 40.0, 30.0);  
  
  @override  
  // 判断是否需要重新裁剪，如果区域始终不变，直接返回false  
  bool shouldReclip(CustomClipper<Rect> oldClipper) => false;  
}
```

滚动组件

单组件可滚动

SingleChildScrollView

```
SingleChildScrollView(  
  child: Column(  
    children: "ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMN  
              OPQRSTUVWXYZ"  
              .split("")  
              .map((c) => Text(c)) // 每行一个字母  
              .toList(),  
  ))
```

滚动列表

ListView

默认构造

```
// 默认是懒加载，但是需要将所有Widget创建好  
ListView(  
  shrinkWrap: true,  
  children: <Widget>[  
    const Text('1'),  
    const Text('2'),  
    const Text('3'),  
    const Text('4'),  
  ],  
)
```

ListView.builder

```
// 滚动到指定的item才调用其builder方法  
ListView.builder(  
  itemCount: 100,  
  itemExtent: 20.0, // 固定每个item的高度，可以提升性能  
  itemBuilder: (BuildContext context, int index) {  
    return Text("$index");  
  },  
)
```

ListView.separated

```
ListView.separated(  
  itemCount: 100,  
  itemBuilder: (BuildContext context, int index) {  
    return Text("$index");  
  },  
)
```

滚动表格

多Tab页面

24 / 57


```
    itemCount: 10000,
    itemBuilder: (context, index) {
      return pageList[index % (pageList.length)];
    },
  ),
```

TabBarView、TabBar

TabBarView封装了PageView，可以通过统一Controller和TabBar联动。

```
Widget getPage() {
  var _tabController = TabController(length: 3, vsync: this);

  return Scaffold(
    appBar: AppBar(
      title: Text("3 Tab Page"),
      bottom: TabBar( // 顶部三个Tab
        controller: _tabController,
        tabs: <Widget>[
          Text("Tab1"),
          Text("Tab2"),
          Text("Tab3"),
        ],
      ),
    ),
    body: TabBarView( // 包含三个页面的PageView，可以和TabBar联动
      controller: _tabController,
      children: <Widget>[
        Page1(),
        Page2(),
        Page3(),
      ],
    ),
  );
}
```

合并多个滚动组件

CustomScrollView

共用 Scrollable 和 Viewport 对象，将多个滚动组件对应的 Sliver 添加到这个共用的Viewport 对象中

```
Widget getWidget() {
  // 生成Sliver列表
  var listView1 = SliverFixedExtentList(
    itemExtent: 60,
    delegate: SliverChildBuilderDelegate(
      (_, index) => Text('First List $index'),
      childCount: 10,
    ),
  );
```

```

    ),
  );

  var listView2 = SliverFixedExtentList(
    itemExtent: 60,
    delegate: SliverChildBuilderDelegate(
      (_, index) => Text('Second List $index'),
      childCount: 10,
    ),
  );

  // 将两个Sliver添加到里面
  return CustomScrollView(
    slivers: [
      listView1, // 添加进来的必须是Silver
      listView2,
    ],
  );
}

```

功能型组件

页面框架

Scaffold

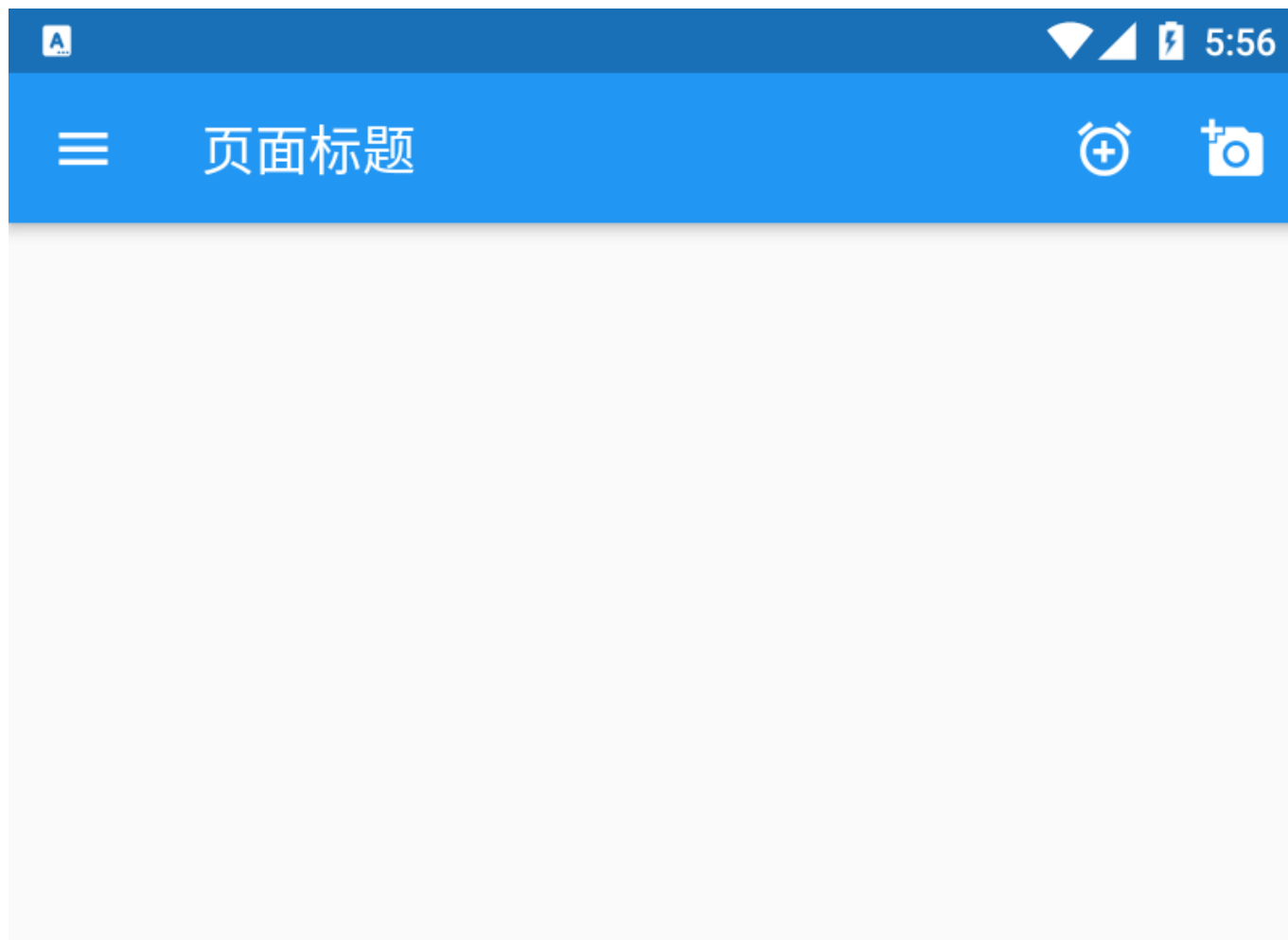
页面的基本框架，可以在里面填充导航栏、抽屉菜单、底部Tab及悬浮按钮等常见的通用页面元素。

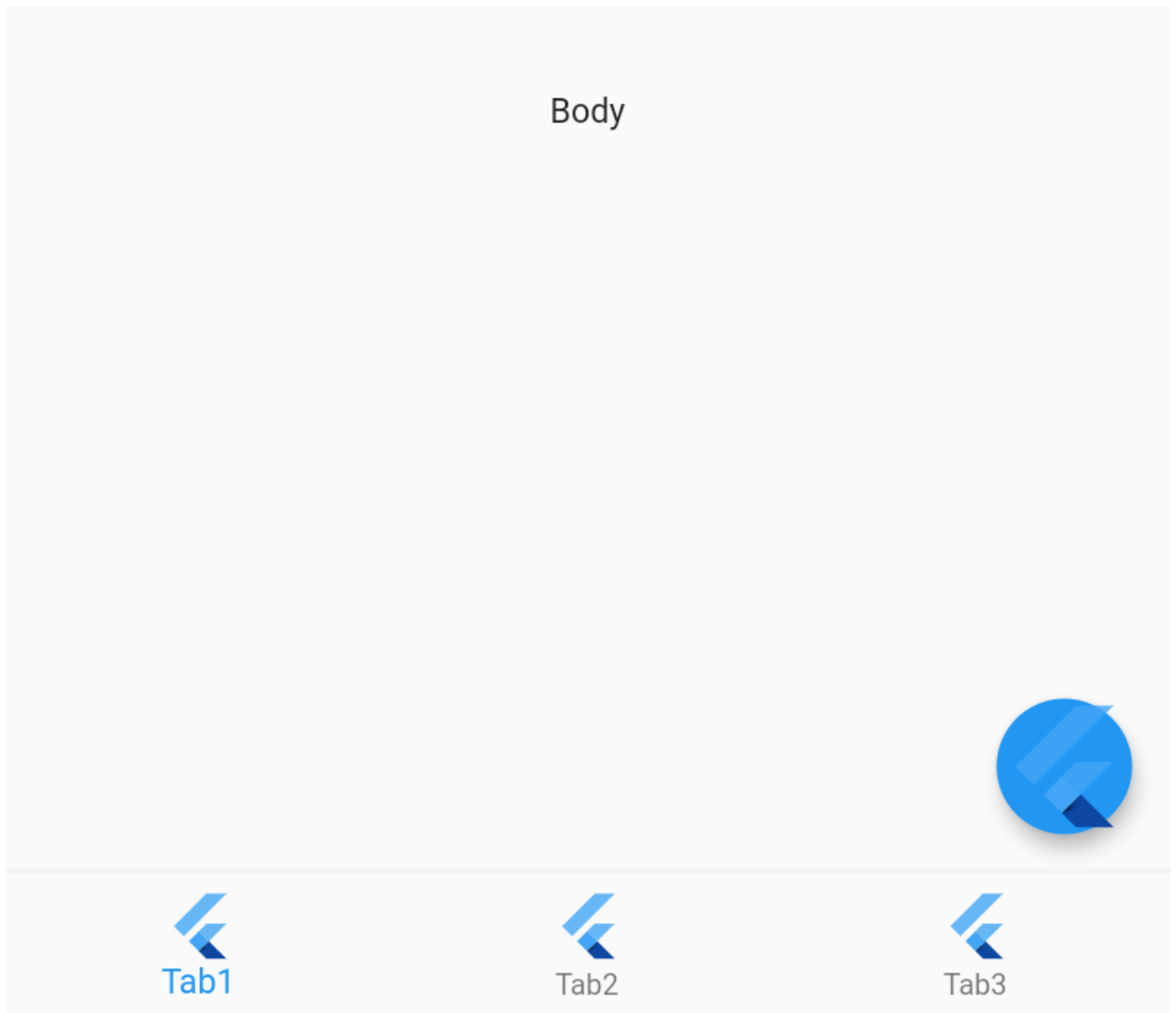
```

Scaffold(
  appBar: AppBar(
    //导航栏
    title: Text("页面标题"),
    actions: <Widget>[
      //导航栏右侧菜单
      IconButton(icon: Icon(Icons.add_alarm), onPressed: () {}),
      IconButton(icon: Icon(Icons.add_a_photo), onPressed: () {}),
    ],
  ),
  drawer: Drawer(
    child: ListView(
      children: <Widget>[
        ListTile(
          leading: const Icon(Icons.add_alarm),
          title: const Text('Add alarm'),
        ),
        ListTile(
          leading: const Icon(Icons.add_a_photo),
          title: const Text('add a photo'),
        ),
      ],
    ),
  ),
)

```

```
), //抽屉
bottomNavigationBar: BottomNavigationBar(
  // 底部导航
  items: <BottomNavigationBarItem>[
    BottomNavigationBarItem(
      icon: FlutterLogo(size: 30), title: Text('Tab1')),
    BottomNavigationBarItem(
      icon: FlutterLogo(size: 30), title: Text('Tab2')),
    BottomNavigationBarItem(
      icon: FlutterLogo(size: 30), title: Text('Tab3')),
  ],
  currentIndex: 0,
  onTap: (index) {
    print("click tab $index");
  },
),
floatingActionButton: FloatingActionButton(
  //悬浮按钮
  child: FlutterLogo(
    size: 60,
  ),
  onPressed: () {
    print("click floating button");
  },
),
body: getBody(),
)
```





AppBar

Material风格导航栏，区域包含页面顶部和底部，通常在Scaffold中使用时，我们用它来设置顶部导航栏。底部导航栏通常Scaffold组件的bottomNavigationBar属性来设置。

注意：抽屉按钮是Scaffold中设置的

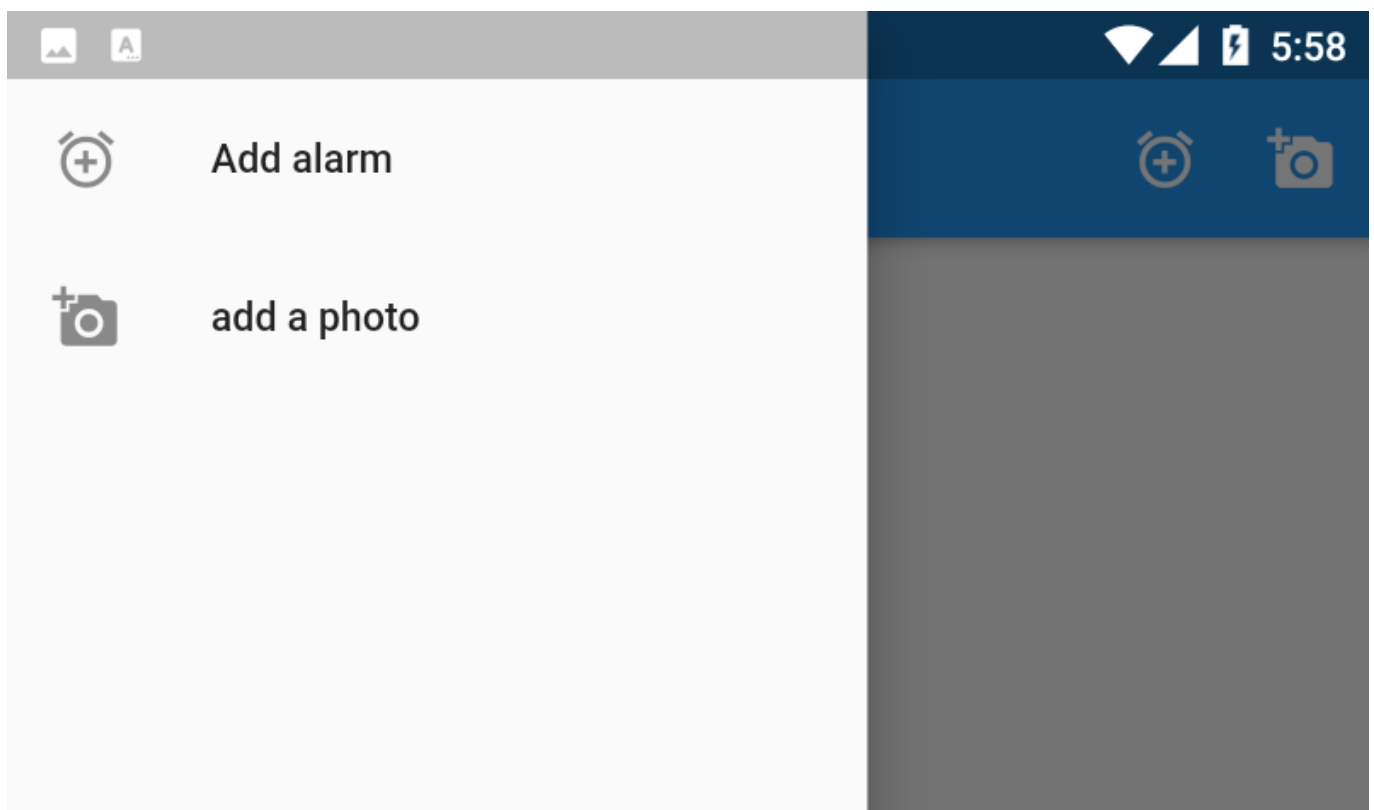
```
AppBar(  
  //导航栏  
  title: Text("页面标题"),  
  actions: <Widget>[  
    //导航栏右侧菜单  
    IconButton(icon: Icon(Icons.add_alarm), onPressed: () {}),  
    IconButton(icon: Icon(Icons.add_a_photo), onPressed: () {}),  
  ],  
)
```

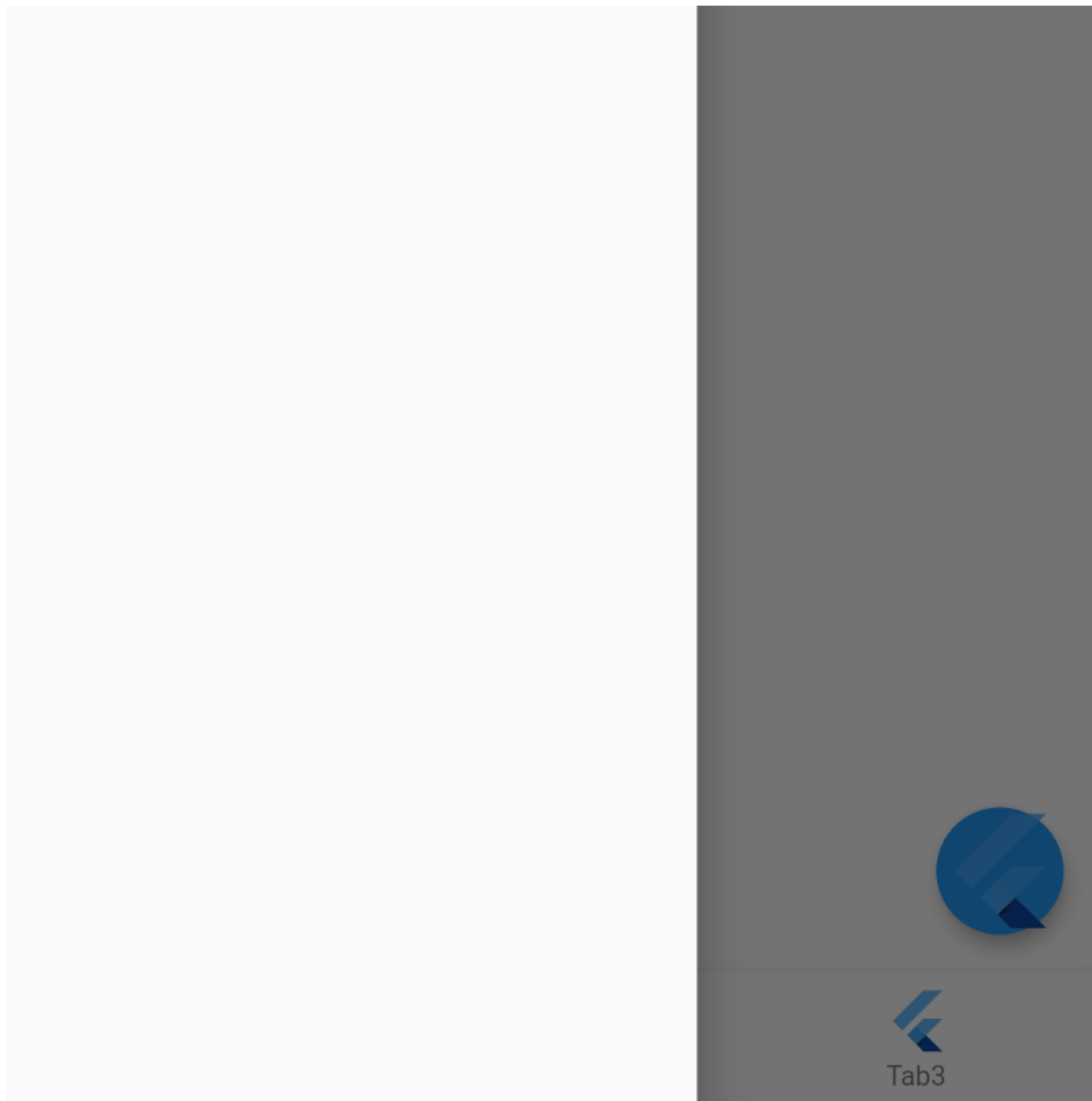


Drawer

抽屉菜单，Scaffold中设置drawer和endDrawer属性可以分别指定左右抽屉菜单，默认可以通过横滑调出。

```
Drawer(  
  child: ListView(  
    children: <Widget>[  
      ListTile(  
        leading: const Icon(Icons.add_alarm),  
        title: const Text('Add alarm'),  
      ),  
      ListTile(  
        leading: const Icon(Icons.add_a_photo),  
        title: const Text('add a photo'),  
      ),  
    ],  
  ),  
)
```





BottomNavigationBar

BottomNavigationBar底部导航栏

```
BottomNavigationBar(  
  // 底部导航  
  items: <BottomNavigationBarItem>[  
    BottomNavigationBarItem(  
      icon: FlutterLogo(size: 30), title: Text('Tab1')),  
    BottomNavigationBarItem(  
      icon: FlutterLogo(size: 30), title: Text('Tab2')),  
    BottomNavigationBarItem(  
      icon: FlutterLogo(size: 30), title: Text('Tab3')),  
  ],  
  currentIndex: 0,  
)
```

```
onTap: (index) {  
  print("click tab $index");  
},  
)
```

FloatingActionButton

页面悬浮按钮，可以指定位置

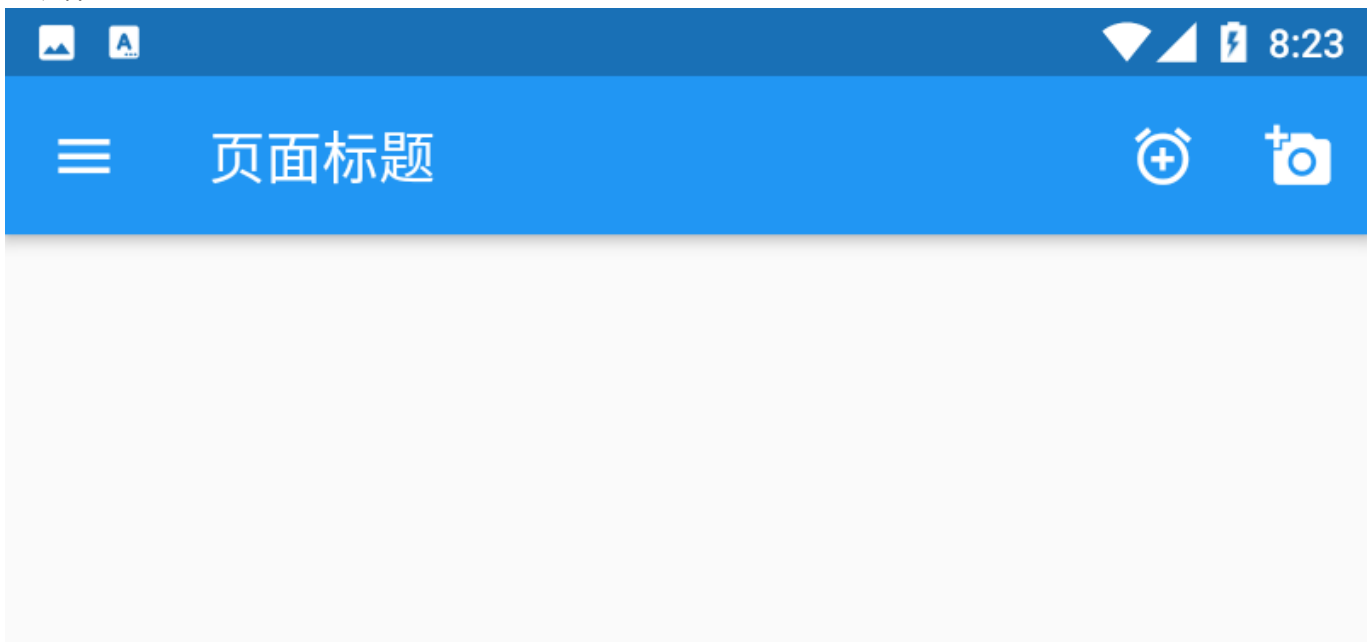
```
FloatingActionButton(  
  //悬浮按钮  
  child: FlutterLogo(  
    size: 50,  
  ),  
  onPressed: () {  
    print("click floating button");  
  })
```

SnackBar

在框架内展示通知

```
RaisedButton(  
  child: Text("点击展示通知"),  
  onPressed: () {  
    Scaffold.of(context).showSnackBar(SnackBar(  
      content: Text('我是通知内容'),  
    ));  
  })
```

默认样式如下





对话框

showDialog

对话框通过该方法启动，本质上是Navigator的push

```
int? index = await showDialog( // 异步返回结果
  context: context,
  builder: (context) {
    return AlertDialog(...);
  });
```


关闭对话框的方法

```
Navigator.of(context).pop()
```

AlertDialog

通常用于某个操作的用户确认

```
AlertDialog(  
  title: Text("提示"),  
  content: Text("提示内容 提示内容?"),  
  actions: <Widget>[  
    FlatButton(  
      child: Text("取消"),  
      onPressed: () => Navigator.of(context).pop(), //关闭对话框  
    ),  
    FlatButton(  
      child: Text("确定"),  
      onPressed: () {  
        // ... 执行删除操作  
        Navigator.of(context).pop(true); //关闭对话框  
      },  
    ),  
  ],  
)
```

提示

提示内容 提示内容?

取消

确定

SimpleDialog

通常用于提供一组选项让用户选择

```
SimpleDialog(  
  title: const Text('请选择午餐'),  
  children: <Widget>[  
    SimpleDialogOption(  
      onPressed: () {  
        Navigator.pop(context, 1); // 返回选择了列表第几项  
      },  
      child: Text('牛排'),  
    ),  
    SimpleDialogOption(  
      onPressed: () {  
        Navigator.pop(context, 2);  
      },  
      child: Text('炸鸡'),  
    ),  
    SimpleDialogOption(  
      onPressed: () {  
        Navigator.pop(context, 3);  
      },  
      child: Text('炸酱面'),  
    ),  
  ],  
)
```

请选择午餐

牛排

炸鸡

炸酱面

AlertDialog和SimpleDialog只能展示有限且固定长度的数据，如果需要在对话框中展示一个ListView，可以直接使用Dialog。

```
Dialog(  
  child: Column(  
    children: <Widget>[  
      ListTile(title: Text("请选择")),  
      Expanded(  
        child: ListView.builder(  
          itemCount: 100,  
          itemBuilder: (context, index) => Text("$index"),  
        )),  
    ],  
  ));
```





手势识别

手机上常见操作手势

GestureDetector

点击、双击、长按

```
GestureDetector(  
  child: Container(  
    color: Colors.red,  
    width: 200.0,  
    height: 100.0,  
  ),  
  onTap: () => print("点击"),  
  onDoubleTap: () => print("双击"),  
  onLongPress: () => print("长按"),  
)
```

滑动

```
GestureDetector(  
  child: Container(  
    color: Colors.red,  
    width: 400.0,
```

```
    height: 400.0,
  ),
  //手指按下
  onPanDown: (DragDownDetails e) {
    print("用户手指按下位置: ${e.globalPosition}");
  },
  //手指移动
  onPanUpdate: (DragUpdateDetails e) {
    // 此处可以通过重建child组件的位置, 实现拖动的效果
    print('滑动距离: ${e.delta}');
  },
  // 手指抬起
  onPanEnd: (DragEndDetails e){
    print('本次移动速度: ${e.velocity}');
  },
)
```

```
I/flutter: 用户手指按下位置: Offset(133.4, 316.7)
I/flutter: 滑动距离: Offset(2.4, 0.0)
I/flutter: 滑动距离: Offset(1.2, 0.0)
I/flutter: 滑动距离: Offset(3.6, 0.6)
I/flutter: 滑动距离: Offset(1.2, 0.6)
I/flutter: 滑动距离: Offset(3.0, 2.4)
I/flutter: 滑动距离: Offset(1.8, 0.6)
I/flutter: 滑动距离: Offset(2.4, 2.4)
I/flutter: 滑动距离: Offset(2.4, 1.8)
I/flutter: 滑动距离: Offset(1.2, 0.6)
I/flutter: 滑动距离: Offset(2.4, 2.4)
I/flutter: 滑动距离: Offset(1.2, 1.2)
I/flutter: 滑动距离: Offset(3.0, 2.4)
I/flutter: 滑动距离: Offset(0.6, 0.6)
I/flutter: 滑动距离: Offset(0.6, 0.6)
I/flutter: 滑动距离: Offset(0.6, 0.0)
I/flutter: 本次移动速度: Velocity(32.7, 74.3)
```

两指缩放

```
// 色块尺寸参数
ValueNotifier<double> size = ValueNotifier<double>(300);

GestureDetector(
  // ValueListenableBuilder当参数变化时重建色块
  child: ValueListenableBuilder<double>(
    builder: (BuildContext context, double value, Widget child) {
      return Container(
        width: size.value,
        height: size.value,
        color: Colors.red,
      );
    },
    valueListenable: size,
  ),
  onScaleUpdate: (ScaleUpdateDetails details) {
    // 这里根据缩放比例修改尺寸参数
    size.value = details.scale.clamp(1, 10.0) * 100;
  },
)
```

带水波效果的点击

InkWell

```
InkWell(
  onTap: () {},
  child: Container(
    width: 400,
    height: 400,
    alignment: Alignment.center,
    color: Colors.black12,
    child: Text("点击此处有水波效果"),
  ),
)
```

拖动效果

Draggable

使用 GestureDetector 配合更新组件位置也可以实现拖动效果，不过使用Draggable可以更方便的实现复杂的拖动。

```
Draggable(
  child: Container( // 拖动时留在原地
    height: 100,
    width: 100,
```

```

        alignment: Alignment.center,
        color: Colors.red,
    ),
    childWhenDragging: Container( // 拖动时原位置展示的样式, 不设置则展示child
        height: 100,
        width: 100,
        alignment: Alignment.center,
        color: Colors.green,
    ),
    feedback: Container( // 拖动时随手指移动
        height: 100,
        width: 100,
        alignment: Alignment.center,
        color: Colors.blue,
    ),
    onDragStarted: () {
        print('拖动开始');
    },
    onDragEnd: (DraggableDetails details) {
        print('拖动结束: ${details.offset}');
    },
    // 后面两个方法配合DragTarget使用, 可以实现拖动到不同位置实现不同效果
    onDragCompleted: () {
        print('拖动到了DragTarget上');
    },
    onDraggableCanceled: (Velocity velocity, Offset offset) {
        print('未拖动到DragTarget上 velocity:$velocity, offset:$offset');
    },
)

```

DragTarget

配合Draggable使用, 所有回调都会收到来自Draggable的data参数, 用以识别拖过来的控件是哪一个。

```

DragTarget<Color>(  
    // build、控件拖动到上面未松手、松手后三次调用。  
    // build和松手后两个参数列表都为空  
    builder: (BuildContext context, List<dynamic> candidateData, List<dynamic>  
rejectedData) {  
        print('candidateData: $candidateData, rejectedData: $rejectedData');  
        var color = Colors.black; // 默认展示黑色  
        if (candidateData != null && candidateData.length > 0) {  
            color = candidateData.first;  
        }  
        return Container(  
            height: 100,  
            width: 100,  
            alignment: Alignment.center,  
            color: color,  
        );  
    },  
),

```

```
// 根据数据判断是否接受，决定调用onAccept还是onLeave
onWillAccept: (Color data) {
  if (data == Colors.red) { // 只接收红色
    return true;
  } else {
    return false;
  }
},
onAccept: (Color data) {
  print('接收颜色 $data');
},
onLeave: (data) {
  print('拒绝接收颜色 $data');
},
)
```

Dart速查手册

基本示例

```
// 定义个方法。
printNumber(num aNumber) {
  print('The number is $aNumber.');// 在控制台打印内容。 $variableName (or
${expression})
}

// 这是程序执行的入口。顶层方法main
main() {
  var number = 42; // 定义并初始化一个变量。
  printNumber(number); // 调用一个方法。
}
```

核心概念

一切皆对象	numbers, functions, and null 所有对象继承Object类
强类型	默认类型推导
泛型	List List (a list of objects of any type)
函数	顶层函数（main） 类的静态方法、成员方法 嵌套方法：函数内部定义

一切皆对象	numbers, functions, and null 所有对象继承Object类
变量	顶层变量 类静态变量、成员变量
没有public, protected, and private	_ 下划线开头表示库私有
标识符	开头：字母、下划线 后面：字符、数字

变量

声明

var name = 'Bob';	类型推导
String name = 'Bob';	指定类型
int lineCount;	默认是null

final

final name = 'Bob';	运行期 只能初始化一次
----------------------------	------------------------

const

const bar = 1000000;	编译期初始化
const atm = 1.01325 * bar;	引用其他const
var foo = const [];	const构造函数 foo指向的数组不能修改（永远是空数组） foo本身可以指向别的数组

内置类型

Numbers（数值）

两种数值类型

int	默认2^-53 到 2^53 范围内的整数 超出自动使用大整数（任意精度）	var x = 1; var hex = 0xDEADBEEF; var bigInt = 34653465834652437659238476592374958739845729;
double	64-bit (双精度) 浮点数	var y = 1.1; var exponents = 1.42e5;

数值与字符串转换

```
var one = int.parse('1');           // String -> int
var onePointOne = double.parse('1.1'); // String -> double

String oneAsString = 1.toString();   // int -> String
String piAsString = 3.14159.toStringAsFixed(2); // double -> String  piAsString == '3.14'
```

Strings（字符串）

UTF-16 编码

单引号或者双引号	<pre>var s1 = 'Single quotes work well for string literals.'; var s2 = "Double quotes work just as well."; var s3 = 'It's easy to escape the string delimiter.'; var s4 = "It's even easier to use the other delimiter.";</pre>
插值	<pre>'Dart has \$s, which is very handy.' 'Dart has \${s.toUpperCase()}, which is very handy.'</pre>
连接	<pre>var s1 = 'String ' + 'concatenation' var s1 = 'String ' 'concatenation' // 可省略加号</pre>
多行字符串对象	<pre>var s1 = ''' You can create multi-line strings like this one. '''; var s2 = """"This is also a multi-line string."""";</pre>
“原始 raw” 字符串	<pre>var s = r"In a raw string, even \n isn't special.";</pre>

Booleans（布尔值）

bool 的类型	只有两个常量对象：true 和 false
只有 true 对象才被认为是 true 所有其他的值都是 false	<pre>var name = 'Bob';if (name) { // Prints in JavaScript, not in Dart. print('You have a name!'); }</pre>

Lists（列表）

字面量	<pre>var list = [1, 2, 3];</pre>
索引	<pre>0~length-1 list[1] == 2;</pre>

字面量	<code>var list = [1, 2, 3];</code>
常量列表	<code>var constantList = const [1, 2, 3];</code>

Maps

创建	<code>var gifts = { // Keys Values 'first' : 'partridge', 'second': 'turtledoves', 'fifth' : 'golden rings' };</code>	<code>var gifts = new Map(); gifts['first'] = 'partridge'; gifts['second'] = 'turtledoves'; gifts['fifth'] = 'golden rings';</code>
	<code>var nobleGases = { // Keys Values 2 : 'helium', 10: 'neon', 18: 'argon', };</code>	<code>var nobleGases = new Map(); nobleGases[2] = 'helium'; nobleGases[10] = 'neon'; nobleGases[18] = 'argon';</code>
访问	<code>gifts['first'] == 'partridge'</code>	不存在的返回null <code>gifts['fifth'] == null</code>
编译时常量	<code>final constantMap = const { 2: 'helium', 10: 'neon', 18: 'argon', };</code>	

Runes

字符串的 UTF-32 code points

Symbols

代表 Dart 程序中声明的操作符或者标识符

可以通过名字引用标识符 混淆后Symbol标识符不会改变	
字面量	<code>#radix</code> <code>#bar</code>

Functions（方法）

基本用法

定义

```
bool isNoble(int atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

表达式形式

```
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] != null;
```

可选参数

必需的参数在参数列表前面， 后面是可选参数

可选命名参数	<pre>enableFlags({bool bold, bool hidden}) { // ... }</pre>	<pre>enableFlags(bold: true, hidden: false);</pre>
可选位置参数	<pre>String say(String from, String msg, [String device]) { // ... }</pre>	<pre>say('Bob', 'Howdy') say('Bob', 'Howdy', 'smoke signal')</pre>
默认参数	<p>基于可选命名参数</p> <pre>void enableFlags({bool bold = false, bool hidden = false}) { // ... }</pre>	<pre>enableFlags(bold: true);</pre>
	<p>基于可选位置参数</p> <pre>String say(String from, String msg, [String device = 'carrier pigeon', String mood])</pre>	

返回值

所有的函数都返回一个值

默认 return null;

函数式编程

方法也是对象

	<pre>printElement(element) { print(element); } var list = [1, 2, 3]; // Pass printElement as a parameter. list.forEach(printElement);</pre>
方法作为参数	
方法赋值给变量	<pre>var loudify = (msg) => '!!! \${msg.toUpperCase()} !!!';</pre>
方法作为返回值	<pre>return loudify;</pre>

Lambda

```
list.forEach((i) {  
  print(list.indexOf(i).toString() + ': ' + i);  
});  
  
list.forEach((i) => print(list.indexOf(i).toString() + ': ' + i));
```

测试函数是否相等

方法也是对象可以用 == 判断是否相等

同一对象的同一方法相等

不同对象的同一方法不相等

可调用的类（类模拟方法）

类实现了 call() 函数则可以当做方法来调用

```
class WannabeFunction {  
  call(int a, int b) => a + b;  
}  
var wf = new WannabeFunction();  
wf(3, 4); // 7
```

Operators（操作符）

操作符优先级

按优先级顺序从左到右，从上到下

描述	操作符
unary postfix	<code>expr++</code> <code>expr--</code> <code>()</code> <code>[]</code> <code>.</code> <code>?</code>

描述	操作符
unary prefix	<i>-expr</i> <i>!expr</i> <i>~expr</i> <i>++expr</i> <i>--expr</i>
multiplicative	<i>*</i> <i>/</i> <i>%</i> <i>~/</i>
additive	<i>+</i> <i>-</i>
shift	<i><<</i> <i>>></i>
bitwise AND	<i>&</i>
bitwise XOR	<i>^</i>
bitwise OR	<i> </i>
relational and type test	<i>>=</i> <i>></i> <i><=</i> <i><</i> <i>as</i> <i>is</i> <i>is!</i>
equality	<i>==</i> <i>!=</i>
logical AND	<i>&&</i>
logical OR	<i> </i>
if null	<i>??</i>
conditional	<i>expr1 ? expr2 : expr3</i>
cascade	<i>..</i>
assignment	<i>=</i> <i>*=</i> <i>/=</i> <i>~/=</i> <i>%=</i> <i>+=</i> <i>-=</i> <i><<=</i> <i>>>=</i> <i>&=</i> <i>^=</i> <i> =</i> <i>??=</i>

左操作数决定操作符

Vector 对象和一个 Point 对象, aVector + aPoint

使用的是 Vector 对象中定义的 + 操作符

分类

算术操作符

<i>~/</i>	除号, 但是返回值为整数
其它	与java相同

相等相关的操作符

	概念上是测试内容内容是否相同 (只有null和null相同)	
<i>==</i>	Object没有直接定义该方法, 使用扩展定义, 交由子类自己实现 扩展的好处是父类中不需要提供默认实现, 子类也可以不实现?	是否是同一个对象: <i>identical()</i> 方法

	概念上是测试内容内容是否相同 (只有null和null相同)	
==	Object没有直接定义该方法，使用扩展定义，交由子类自己实现 扩展的好处是父类中不需要提供默认实现，子类也可以不实现？	是否是同一个对象：identical() 方法
其它	与java相同	

类型判定操作符

as	类型转换	(emp as Person).firstName = 'Bob';
is	如果对象是指定的类型返回 True	
is!	如果对象是指定的类型返回 False	

赋值操作符

```
a = value;    // 给 a 变量赋值
b ??= value;  // 如果 b 是 null , 则赋值给 b ;
              // 如果不是 null , 则 b 的值保持不变
```

条件表达式

<i>condition ? expr1 : expr2</i>	
<i>expr1 ?? expr2</i>	<i>expr1</i> 是 non-null, 返回其值 否则执行 <i>expr2</i> 并返回其结果
foo?.bar	foo 为 null 则返回 null 否则返回 bar 成员

级联操作符

在同一个对象上 连续调用多个函数以及访问成员变量

<code>var button = querySelector('#button');</code> <code>button.text = 'Confirm';</code> <code>button.classes.add('important');</code> <code>button.onClick.listen((e) =></code> <code>window.alert('Confirmed!'));</code>	<code>querySelector('#button') // Get an object.</code> <code> **..**text = 'Confirm' // Use its members.</code> <code> **..**classes.add('important')</code> <code> **..**onClick.listen((e) =></code> <code> window.alert('Confirmed!'));</code>
--	---

流程控制

分支

if、else	同java	注意只有 true 对象才被认为是 true
		解决方案1 <pre>switch (command) { case 'CLOSED': // Empty case falls through. case 'NOW_CLOSED': // Runs for both CLOSED and NOW_CLOSED. executeNowClosed(); break; }</pre> 解决方案2：执行两个不同case <pre>switch (command) { case 'CLOSED': executeClosed(); continue nowClosed; // Continues executing at the nowClosed label. nowClosed: case 'NOW_CLOSED': // Runs for both CLOSED and NOW_CLOSED. executeNowClosed(); break; }</pre>
switch、 case	编译错误 <pre>switch (command) { case 'OPEN': executeOpen(); // ERROR: Missing break causes an exception!! case 'CLOSED': executeClosed(); break; }</pre>	

循环

for、forEach	<pre>for (var i = 0; i < 5; i++) { ... } for (var x in collection) { print(x); }</pre> <pre>// 接受一个函数 candidates.forEach((candidate) => candidate.interview());</pre>
-------------	--

while、do-while 同java

断言

assert(text != null); 检查模式：运行有效
 生产模式：不会执行

异常

相关用法

throw	Exception 和 Error 类型 可以抛出任意的对象 throw 'Out of llamas!';	<pre>try { foo = "You can't change a final variable's value."; } catch (e) { print('misbehave() partially handled \${e.runtimeType}.'); rethrow; // 捕获的异常重新抛出 }</pre>
catch	<pre>try { breedMoreLlamas(); } on OutOfLlamasException { // 捕获某 种异常 buyMoreLlamas(); } on Exception catch (e) { // 获取具体异 常对象 print('Unknown exception: \$e'); } catch (e, s) { // e不区分类型 s为堆栈信 息 print('Exception details:\n \$e'); print('Stack trace:\n \$s'); }</pre>	
finally	同java	
未捕获异常		
程序并不会退出		
当前任务 的后续代码就不会被执行		
类		
构造		
简化构造 函数语法 糖	<pre>class Point { num x; num y; Point(this.x, this.y); //相当于后面的完 整版 }</pre>	<pre>Point(num x, num y) { // There's a better way to do this, stay tuned. this.x = x; this.y = y; }</pre>
默认构造 函数	没有定义任何构造时自动生成	调用超类的无参构造函数

简化构造函数语法糖	<pre>class Point { num x; num y; Point(this.x, this.y); //相当于后面的完整版 }</pre>	<pre>Point(num x, num y) { // There's a better way to do this, stay tuned. this.x = x; this.y = y; }</pre>
命名构造函数	<pre>Point.fromJson(Map json) { x = json['x']; y = json['y']; }</pre>	<pre>var p2 = new Point.fromJson(jsonData);</pre>
调用超类构造	<p>与Java原则相同 构造函数不继承 无显式指定自动调用超类的无参构造 显式指定超类构造</p> <pre>Employee.fromJson(Map data) : super.fromJson(data) { print('in Employee'); }</pre>	时机：当前构造函数入口处
初始化列表	<pre>Point.fromJson(Map jsonMap) : x = jsonMap['x'], y = jsonMap['y'] { print('In Point.fromJson(): (\$x, \$y)'); }</pre>	<p>先于构造函数执行（顺序） 初始化列表 超类构造 当前构造 初始化表达式等号右边的部分不能访问 this</p>
重定向构造函数	<pre>Point.alongXAxis(num x)**😬*this(x, 0);</pre>	不能有方法体
常量构造函数	<pre>class ImmutablePoint { final num x; // 所有成员变量必须为final final num y; const ImmutablePoint(this.x, this.y); // 声明const构造 static final ImmutablePoint origin = const ImmutablePoint(0, 0); }</pre>	<p>相等的编译时常量是同一个对象</p> <pre>var a = const ImmutablePoint(1, 1); var b = const ImmutablePoint(1, 1); assert(identical(a, b)); // They are the same instance!</pre> <p>创建时会根据签名存储在一个特殊的Hash查找表中</p> <pre>var foo1 = const Foo(1, 1); // #Foo#int#1#int#1 var foo2 = const Foo(1, 1); // #Foo#int#1#int#1 var foo3 = const Foo(1, 2); // \$Foo\$int\$1\$int\$2 var foo4 = const Foo(1, 3); // \$Foo\$int\$1\$int\$3 var baz1 = const Baz(const Foo(1, 1), "hello"); // \$Baz\$Foo\$int\$1\$int\$1\$String\$hello var baz2 = const Baz(const Foo(1, 1), "hello"); // \$Baz\$Foo\$int\$1\$int\$1\$String\$hello</pre>

简化构造函数语法糖	<pre>class Point { num x; num y; Point(this.x, this.y); //相当于后面的完整版 }</pre>	<pre>Point(num x, num y) { // There's a better way to do this, stay tuned. this.x = x; this.y = y; }</pre>
工厂方法构造函数	<pre>factory Logger(String name) { // 这里需要返回一个Logger // 可能不是new的, 例如从缓存取一个 }</pre> <pre>var logger = new Logger('UI'); // 这里可能没有创建新对象虽然调用了new</pre>	作用： 可以返回缓存的对象 可以返回子类对象

成员变量

自动生成getter、setter——final没有setter

通过getter和setter虚拟成员变量

```
class Rectangle {  
  num left;  
  num top;  
  num width;  
  num height;  
  Rectangle(this.left, this.top, this.width, this.height);  
  // 虚拟两个成员: right and bottom.  
  num get right      => left + width;  
  set right(num value) => left = value - width;  
  num get bottom     => top + height;  
  set bottom(num value) => top = value - height;  
}
```

抽象类

通常包含抽象方法	<pre>abstract class AbstractContainer { // ...Define constructors, fields, methods... void updateChildren(); // 抽象方法. }</pre>	不能实例化
实体类中的抽象方法	不影响实例化 编译warning 调用会运行时异常	可以实例化

隐式接口

每个类都隐式的定义了一个包含所有实例成员（属性+方法）的接口，并且这个类实现了这个接口

其它类可以实现这个接口

枚举类型

```
enum Color {
  red,          assert(Color.red.index == 0);
  green,        assert(Color.green.index == 1);
  blue          assert(Color.blue.index == 2);
}
```

List colors = Color.values;
assert(colors[2] == Color.blue);

mixins(混入)

	class T = A with S;	class T = B with A, S;
概念	T中的方法为A和S方法的集合 如果有重复, 取S中的	// 相当于伪代码 class T = (B with A) with S;

```
class Musician extends Performer with Musical {
  // ...
}
```

泛型

使用集合字面量

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

在构造函数中使用泛型

```
var views = new Map<int, View>();
var nameSet = new Set<String>.from(names);
```

运行时特性 (与Java不同)

在 Java 中你可以测试一个对象是否为 List, 但是无法测试一个对象是否为 List。

Dart中可以: print(names is List); // true

库和可见性

使用库

```
import 'dart:io'; // 内置库
import 'package:mylib/mylib.dart'; // 非内置库, package开头
import 'package:utils/utils.dart';
```

指定库前缀解决冲突

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;
// ...
Element element1 = new Element(); // Uses Element from lib1.
lib2.Element element2 = new lib2.Element(); // Uses Element from lib2.
```

导入库的一部分

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;
// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

延迟载入库

用途

- 减少 APP 的启动时间。
- 执行 A/B 测试，例如尝试各种算法的不同实现。
- 加载很少使用的功能，例如可选的屏幕和对话框。

异步支持

async\await

声明 异步 方法	<pre>checkVersion() async { // ... }</pre>	async声明代码体中有延迟执行的代码 因此，函数体不会执行，会立即返回
Future lookUpVersion() async => '1.0.0';		

await

在Dart2中有轻微的改动。async函数执行时，不是立即挂起，而是要执行到函数内部的第一个await。在多数情况下，我们不会注意到这一改动

生成器

设计目标

当我们需要创建一系列数据时，不要一次性把它们都创建出来，而是每次用的时候在进行当前需要用的这一个的计算

同步生成器：sync*

用yield标记一个表达式

通常这个表达式放在一个循环中

调用时立即返回一个Iterable

函数体不会被执行

当调用迭代器的moveNext时

开始执行函数体

每次调用moveNext时执行一次yield标记的表达式并暂停

moveNext返回true

当循环结束，yield表达式没有被执行，函数体返回时

moveNext返回false

```

1  Iterable naturalsTo(n) sync* {
2      print("Begin");
3
4      int k = 0;
5      while (k < n) yield k++;
6
7      print("End");
8  }
9
10 main() {
11     var it = naturalsTo(3).iterator;
12     while(it.moveNext()) {
13         print(it.current);
14     }
15 }
```

第一次调用
moveNext，返回
0, k==1
第二次，返回1,
k==2
Begin 第三次，返回2,
0 k==3
1 第四次，没有执
2 行yield表达式，
End 函数体返回

本质上是yield控
制
先把表达式的值
放入迭代器
然后暂停执行

异步生成器：async*

同样适用yield标记表达式放在循环中

调用时立即返回一个Stream

Stream被listen时函数体开始执行

yield表达式执行后不会暂停

一直执行，并将表达式值放入Stream

Stream被pause

函数体暂停在最后一次yield执行后 本质上是yield控制

Stream被cancel 先把表达式的值放入stream

函数体运行到下一次yield执行前 然后判断是否需要暂停或直接将函数体返回
然后函数直接返回

```

1  import 'dart:async';
2
3  Stream get asynchronousNaturals async* {
4    print("Begin");
5
6    int k = 0;
7    while (k < 3) {
8      print("Before Yield");
9      yield k++;
10   }
11
12   print("End");
13 }
14
15 main() {
16   StreamSubscription subscription = asynchronousNaturals.listen(null);
17   subscription.onData((value) {
18     print(value);
19     subscription.pause();
20   });
21 }
```

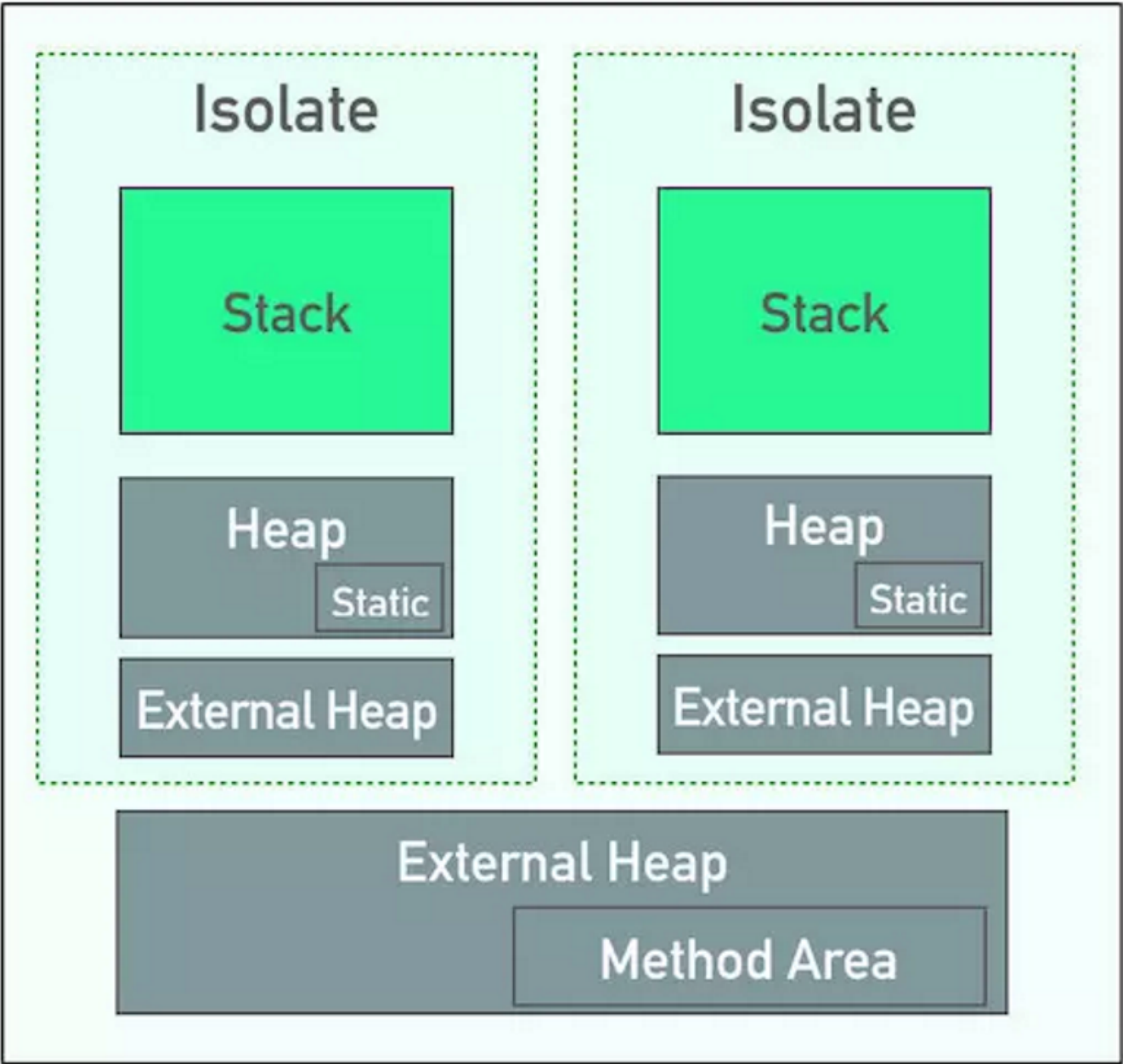
Begin
Before
Yield
0

线程模型

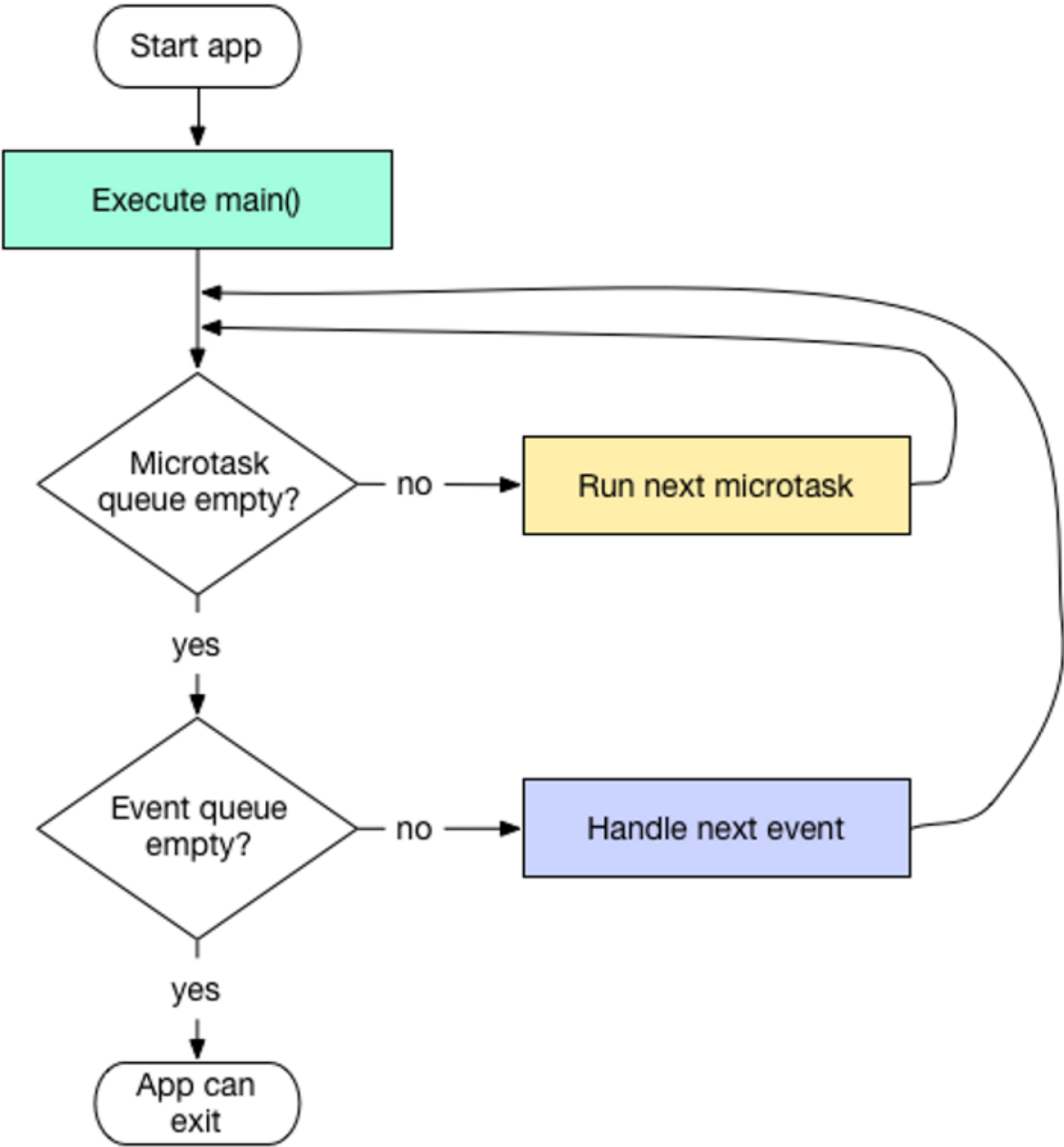
Dart VM中的线程

线程间无共享内存

Dart VM



Dart单线程流程



微任务队列

优先级高，数量少（主要源于Dart内部）

Future.microtask(...)方法向微任务队列插入一个任务

事件队列

外部事件任务：IO、计时器、点击、以及绘制事件等