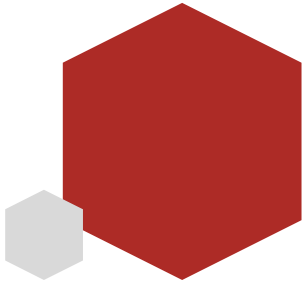
Flutter从入门到实战

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌



Flutter框架核心

安装Flutter开发环境

配置Flutter开发环境（mac/windows）

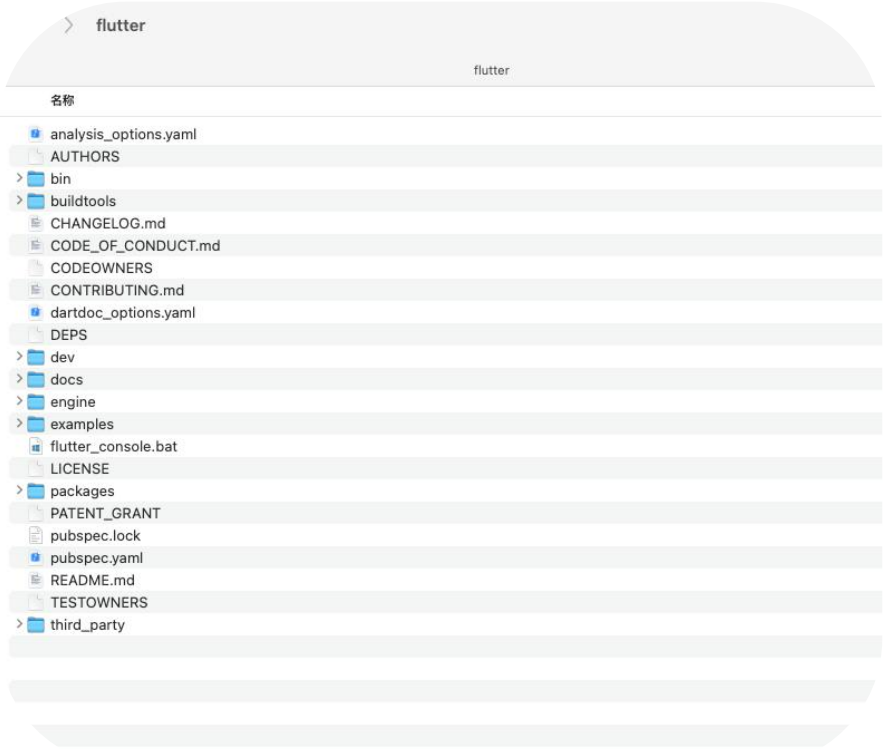
- 下载Flutter SDK
- 配置环境变量
- 诊断flutter环境

sdk下载（方式1）：

```
git clone  https://github.com/flutter/flutter.git
```

注意：务必将flutter包下载到英文目录下

sdk下载（方式2）：从课程资料的软件包中获取



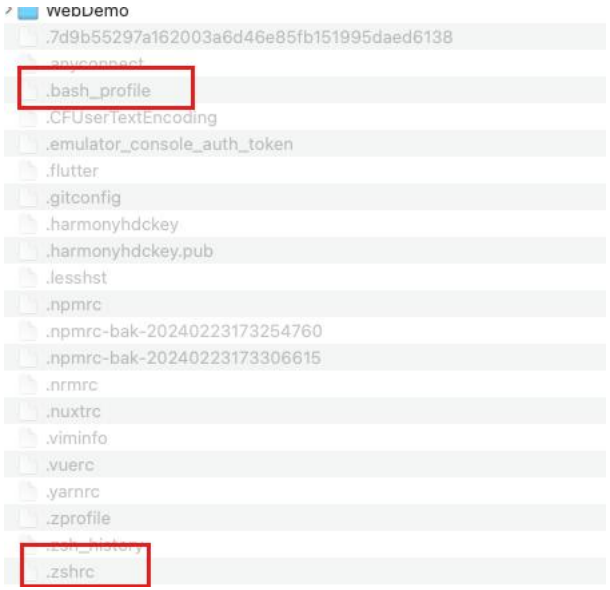
配置环境变量–mac

1.检查mac环境变量生效文件

```
echo $SHELL
```

如果显示/bin/zsh 说明.zshrc是配置文件，如果是/bin/bash 说明.bash_profile是配置文件

2.找到根目录的隐藏文件.zshrc或.bash_profile



3.打开并配置如下环境变量

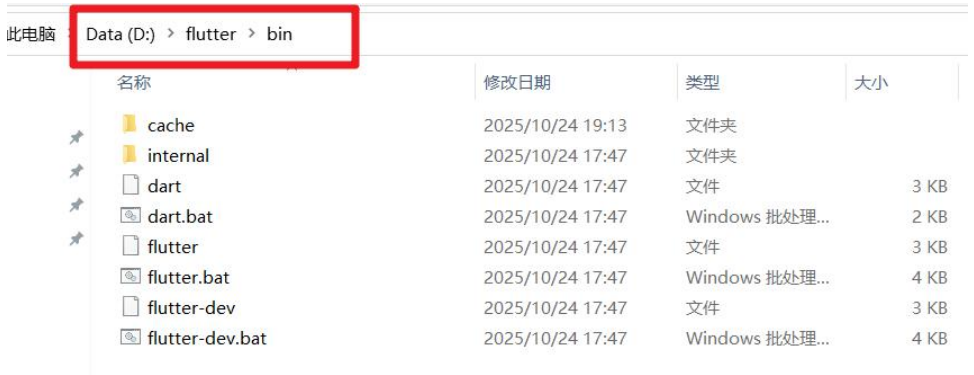
```
export PUB_HOSTED_URL="https://pub.flutter-io.cn"
export FLUTTER_STORAGE_BASE_URL="https://storage.flutter-io.cn"
export PATH="你自己电脑的路径(非中文目录)/flutter/bin:$PATH"
```

4.执行命令让配置生效

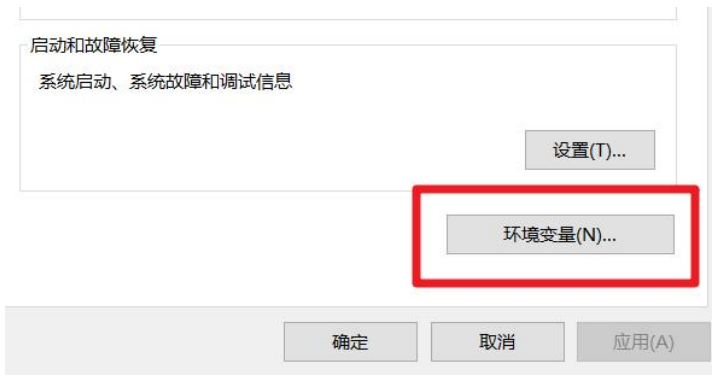
```
source ~/.zshrc 或者 source ~/.bash_profile
```

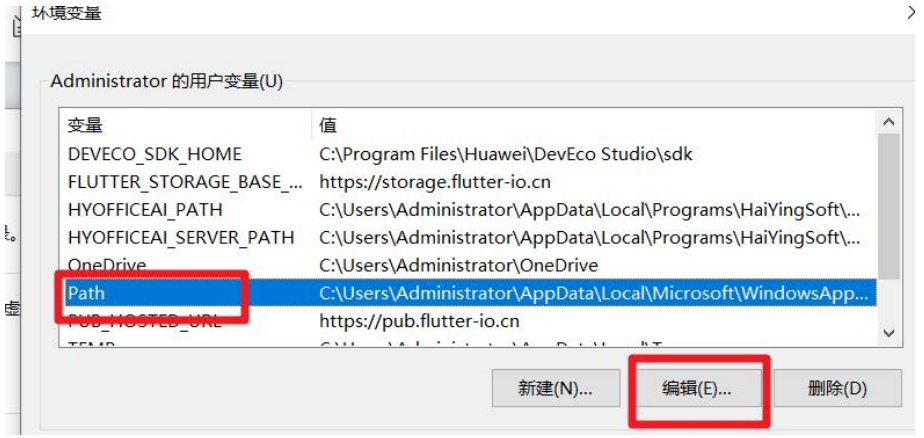
配置环境变量–windows

1.拷贝windows的flutter目录下的bin完整路径



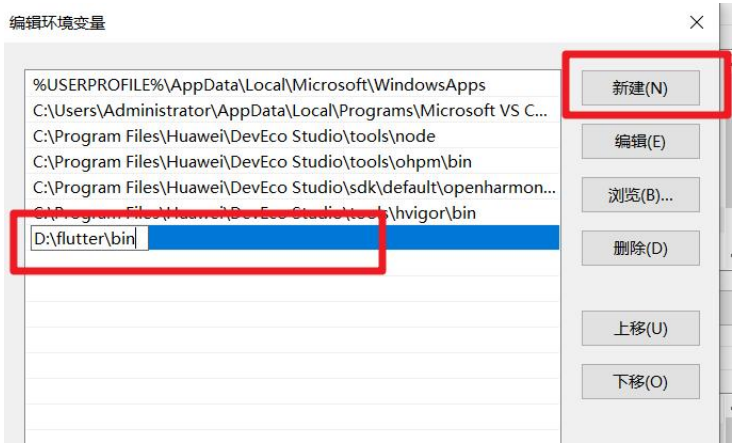
2.打开环境变量配置



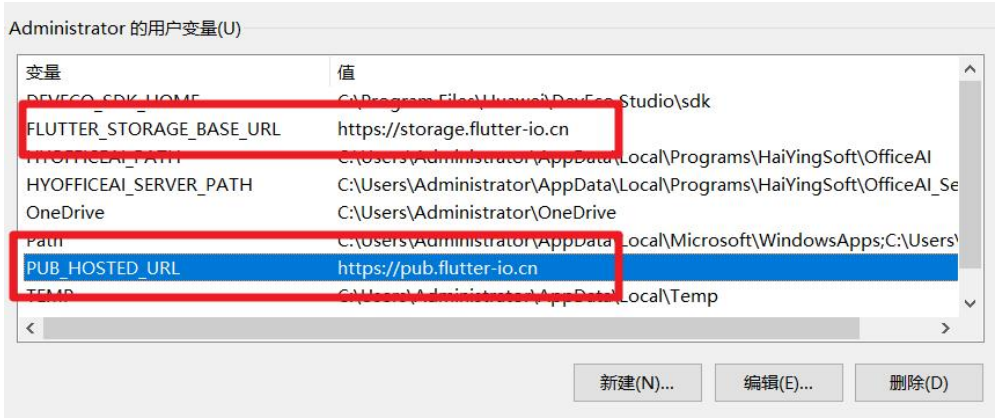


配置环境变量–windows

3.在path路径中添加之前拷贝的bin路径



4.添加两个环境变量PUB_HOSTED_URL和FLUTTER_STORAGE_BASE_URL



检查环境–windows/mac

1.检查flutter版本

```
flutter --version
```

```
gaolingyu@MacBook-Pro-4 ~ % flutter --version
Flutter 3.38.0-1.0.pre-229 • channel master •
https://github.com/flutter/flutter.git
Framework • revision 1404fe64dd (32 小时前) • 2025-10-24 03:51:27 -0400
Engine • hash e1ac576ecdc622d2c1c14bfd3d75f53b4430f34a (revision 1404fe64dd) (31
hours ago) • 2025-10-24 07:51:27.000Z
Tools • Dart 3.11.0 (build 3.11.0-56.0.dev) • DevTools 2.51.0
gaolingyu@MacBook-Pro-4 ~ %
```

2.诊断flutter环境

```
flutter doctor -v
```

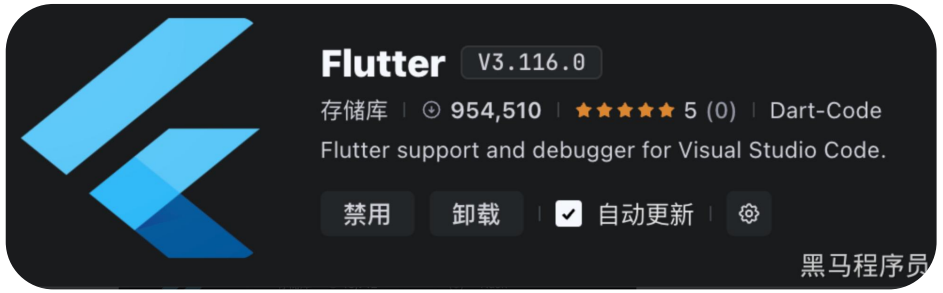
```
[✓] Flutter (Channel master, 3.38.0-1.0.pre-229, on macOS 15.6.1 24G90
darwin-arm64, locale zh-Hans-CN) [146.4s]
  • Flutter version 3.38.0-1.0.pre-229 on channel master at
    /Users/gaolingyu/flutter
  • Upstream repository https://github.com/flutter/flutter.git
  • Framework revision 1404fe64dd (32 小时前), 2025-10-24 03:51:27 -0400
  • Engine revision 1404fe64dd
  • Dart version 3.11.0 (build 3.11.0-56.0.dev)
  • DevTools version 2.51.0
  • Pub download mirror https://pub.flutter-io.cn
  • Flutter download mirror https://storage.flutter-io.cn
  • Feature flags: enable-web, enable-linux-desktop, enable-macos-desktop,
    enable-windows-desktop, enable-android, enable-ios, cli-animations,
    enable-native-assets, omit-legacy-version-file, enable-lldb-debugging
```

创建Flutter项目

- 使用命令创建Flutter工程(web)

```
flutter create --platforms web <项目名称>
```

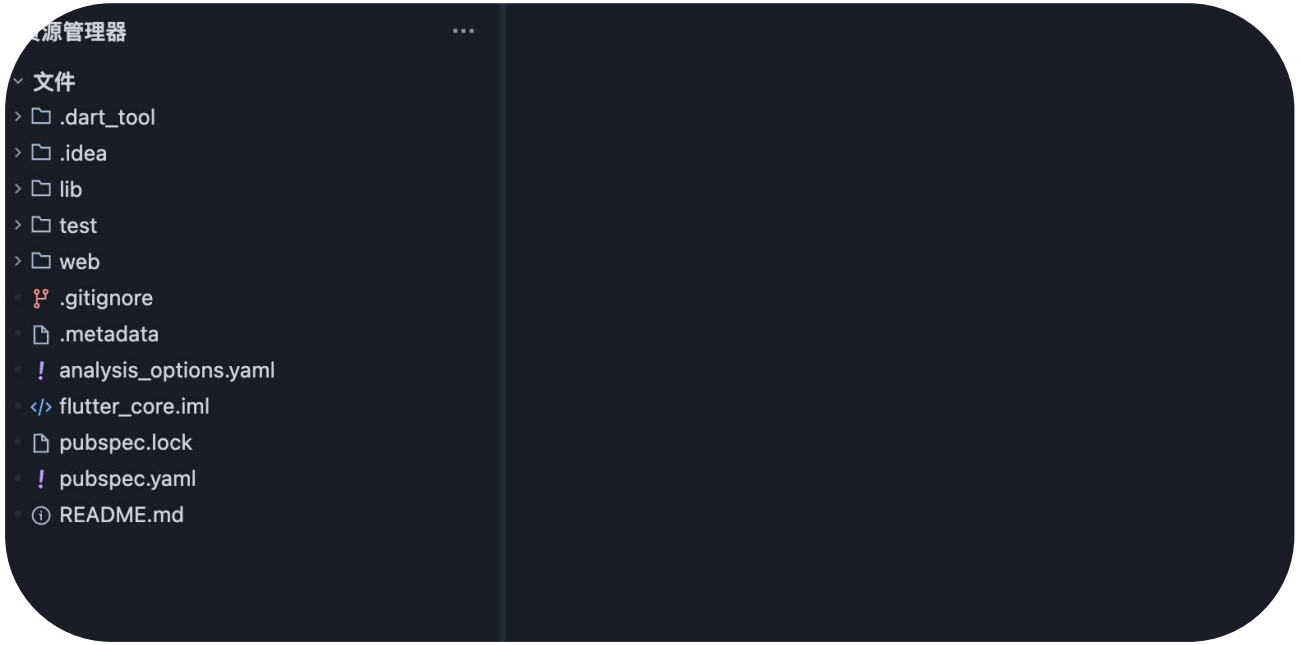
- 安装Trae插件



安装Chrome浏览器–打开项目

- 安装Chrome浏览器-软件包

- 使用Trae编辑器打开创建的flutter项目

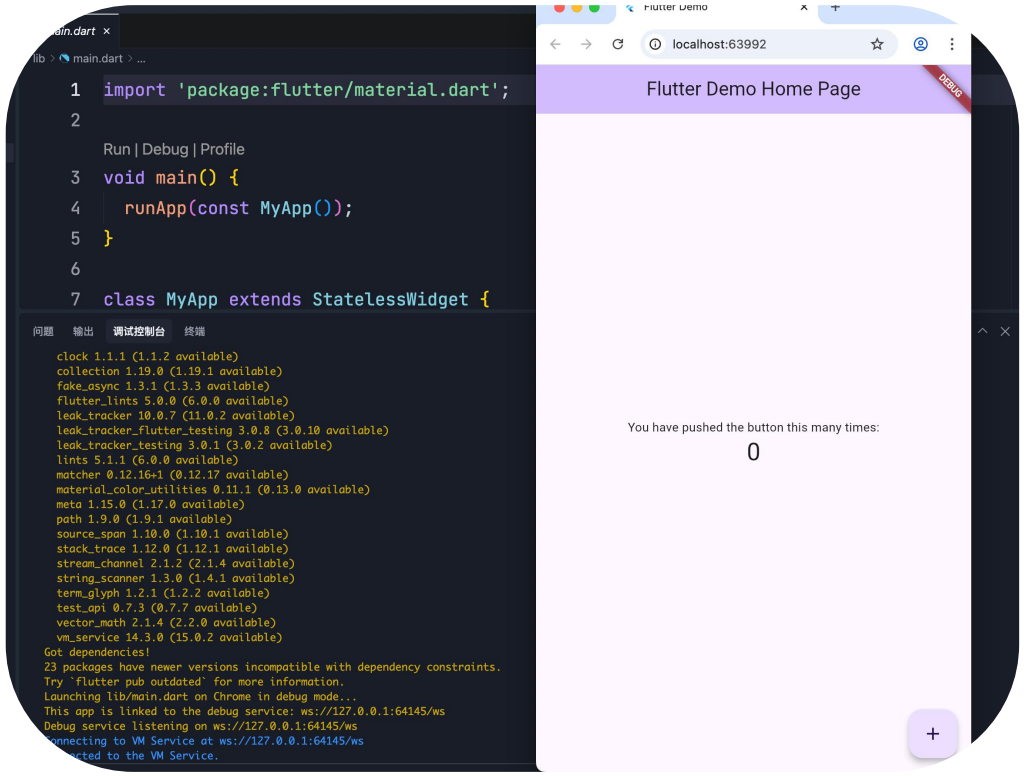


运行Flutter项目

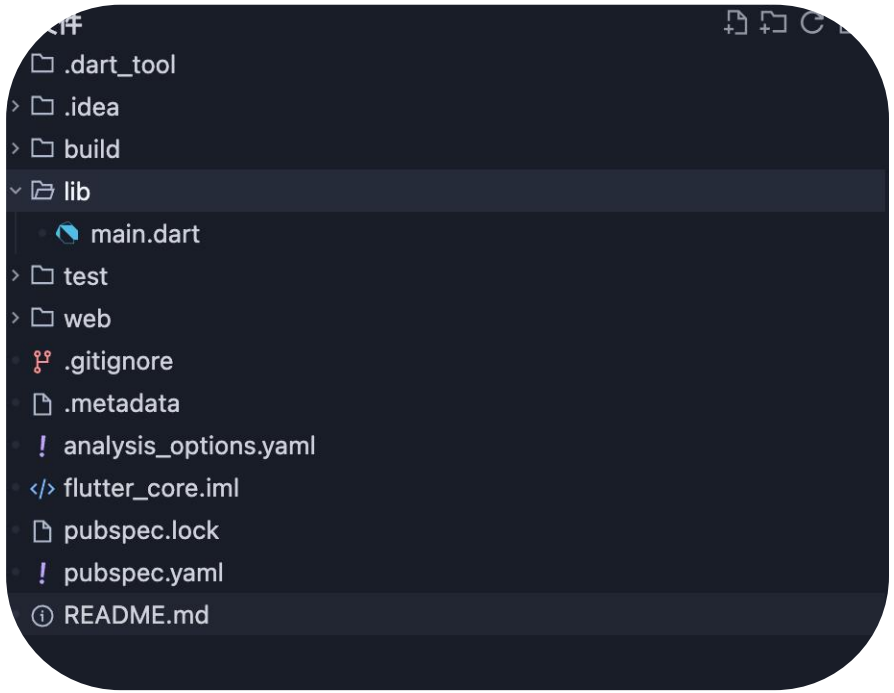
- 打开lib/main.dart文件

```dart
import 'package:flutter/material.dart';

Run | Debug | Profile
void main() {
  runApp(const MyApp());
}

```

- 点击Run运行到浏览器



解析工程目录结构



.dart_tool　# Dart工具生成的文件和缓存

.idea　# IntelliJ IDEA 配置文件

build　# 构建产物目录，包含编译生成的文件

lib　# 项目的主要源代码目录

　main.dart　# 应用程序入口点

test　# 测试文件目录

web　# Web平台特定的配置和资源文件

.gitignore　# Git版本控制忽略文件配置

metadata　# Flutter项目标识文件（自动生成）

analysis_options.yaml # 配置静态代码分析工具

flutter_core.iml　# 用于存储模块（Module）的特定设置

pubspec.lock　# 依赖锁定文件

pubspec.yaml　# 项目依赖和配置文件

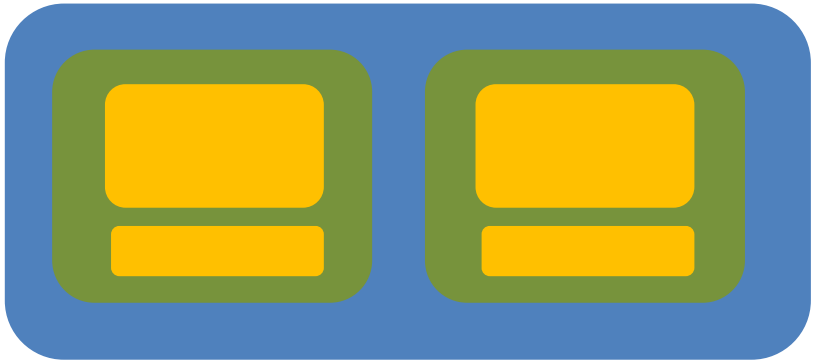
README.md　# 项目说明文档

启动文件说明-runApp和Widget

```
Run | Debug | Profile
void main() {
  runApp(const MyApp());
}
```

Widget

- runApp函数是Flutter内部提供的一个函数，启动一个Flutter应用就是从调用这个函数开始的
- Widget表示控件、组件、部件的含义， Flutter中万物皆Widget



不同的Widget组合

Flutter的默认Material库

- Material是Google公司推行的一套设计风格，有很多的设计规范，如颜色、文字排版、动画等
- 目的：Material为Android、Web、iOS、HarmonyOS多个平台提供统一的交互和视觉体验

总结

- 启动Flutter应用使用runApp方法
- runApp方法中需要传入一个Widget
- Widget是组成Flutter的重要一部分，万物皆Widget
- Material风格是Flutter内置的一套独有的设计风格，里面有很多拆箱可用的Widget

Flutter组件初体验

基础组件–MaterialApp

特性: 整个应用被MaterialApp包裹，方便我们对整个应用的属性进行整体设计

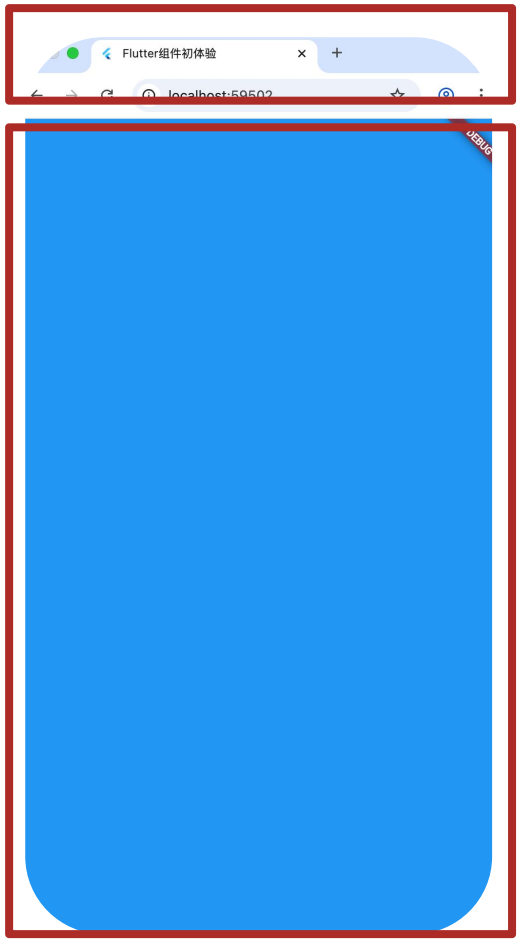
常见属性： title/theme/home

title：用来展示窗口的标题内容（可以不设置）

theme：用来设置整个应用的主题

home: 用来展示窗口的主体内容

```
Run | Debug | Profile
void main() {
  runApp(MaterialApp(
    title: "Flutter组件初体验",
    theme: ThemeData(scaffoldBackgroundColor: Colors.blue),
    home: Scaffold(),
  )); // MaterialApp
}
```

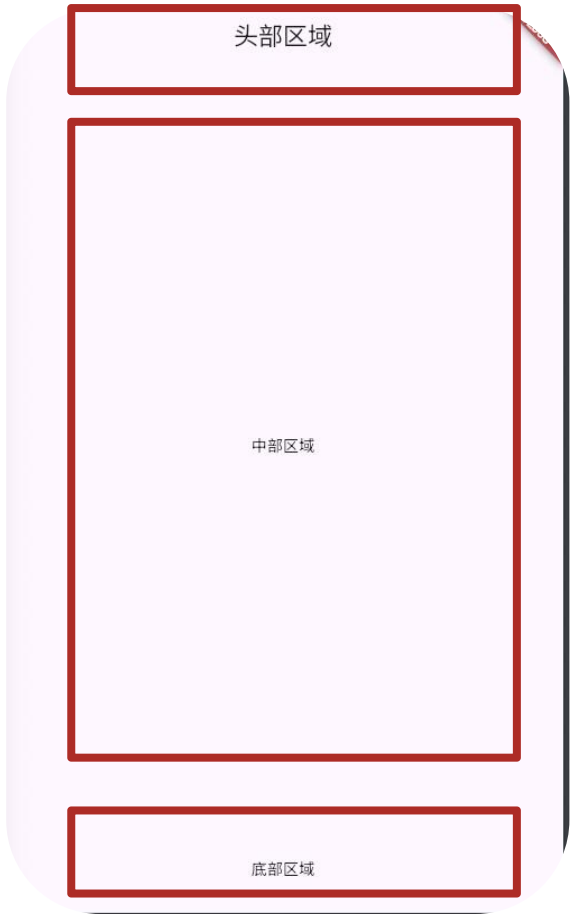


基础组件–Scaffold组件

●Scaffold：用于构建Material Design风格页面的核心布局组件，提供标准、灵活配置的页面骨架

属性	主要作用说明
appBar	页面顶部的应用栏，通常用于显示标题、导航按钮和操作菜单
body	页面的主要内容区域，可以放置任何其他组件，是页面的核心
bottomNavigationBar	底部导航栏，方便用户在不同核心功能页面间切换
backgroungColor	设置整个 Scaffold 的背景颜色
floatingActionButton	悬浮操作按钮，常用于触发页面的主要动作
...	其他

基础组件–Scaffold组件




```
| Debug | Profile
void main() {
  runApp(MaterialApp(
    title: "标题",
    home: Scaffold(
      appBar: AppBar(
        title: Text("头部区域"),
      ), // AppBar

      body: Container(
        child: Center(
          child: Text("中部区域"),
        ), // Center
      ), // Container

      bottomNavigationBar: Container(
        height: 80,
        child: Center(
          child: Text("底部区域"),
        ), // Center
      ), // Container
    ), // Scaffold
  )); // MaterialApp
```

头部

中部

底部

总结

- MaterialApp包裹整个应用形成统一的Material Design风格
- Scoffold组件可快速搭建页面骨架，如appBar、body、bottomNavigationBar等
- Container用来作为容器，设置高度(height)，child用来存放子组件
- Text是用来显示文本的组件

Flutter自定义组件–无状态组件和有状态组件

- 定义： 根据自己特定的需求创建自己的Widget
- 分类： Flutter分为无状态组件和有状态组件

特性	StatelessWidget(无状态)	StatefulWidget(有状态)
核心特征	一旦创建，内部状态不可变	持有可在其生命周期内改变的状态
使用场景	静态内容展示，外观仅由配置参数决定	交互式组件，如计数器、可切换开关、表单输入框
生命周期	相对简单，主要是构建（build）	更为复杂，包含状态创建、更新和销毁
代码结构	单个类	两个关联的类：Widget 本身和单独的 State 类

无状态组件–StatelessWidget

- 定义： 创建一个新的类，继承StatelessWidget类并实现build方法
- 要点： build返回一个Widget
- 场景： 纯展示型组件，没有用户交互操作
- 需求： 把之前案例的骨架换成无状态组件

```
Run | Debug | Profile
void main() {
  runApp(MainPage());
}
```

```
class MainPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // TODO: implement build
    return MaterialApp(
      title: "标题",
      home: Scaffold(
        appBar: AppBar(
          title: Text("头部区域"),
        ), // AppBar
        body: Container(
          child: Center(
            child: Text("中部区域"),
          ), // Center
        ), // Container
        bottomNavigationBar: Container(
          height: 80,
          child: Center(
            child: Text("底部区域"),
          ), // Center
        ), // Container
      ), // Scaffold
    ); // MaterialApp
  }
```

有状态组件–StatefulWidget

- 定义： 有状态组件是构建动态交互界面的核心，能够管理变化的内部状态，当状态改变时，组件会更新显示内容
- 实现1：创建两个类，第一个类继承StatefulWidget类，主要接收和定义最终参数，核心作用是创建State对象
- 实现2：第二个类继承State<第一个类名>，负责管理所有可变的数据和业务逻辑，并实现build构建方法
- 要点： build方法需要返回一个Widget
- 需求： 将之前骨架组件换成有状态组件

第一个类

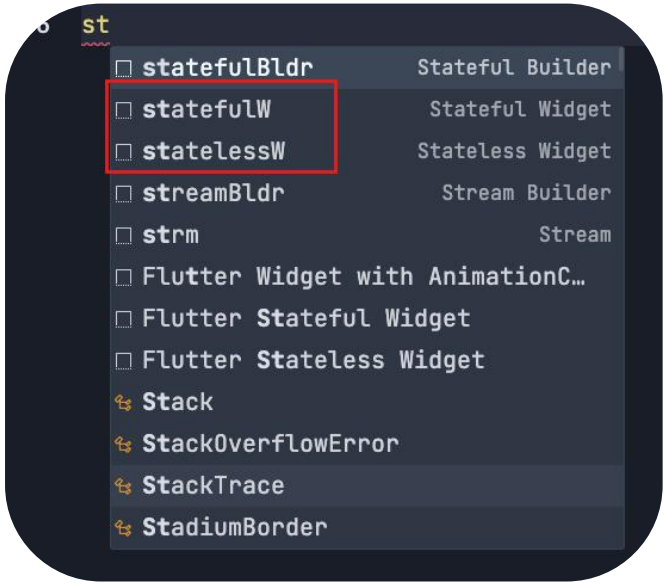
```dart
class MainPage extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    // TODO: implement createState
    return _MainPage();
  }
}
```

第二个类

```dart
class _MainPage extends State<MainPage> {
@override
Widget build(BuildContext context) {
  // TODO: implement build
  return MaterialApp(
    title: "标题",
    home: Scaffold(
      appBar: AppBar(
        title: Text("头部区域"),
      ), // AppBar
      body: Container(
        child: Center(
          child: Text("中部区域"),
        ), // Center
      ), // Container
      bottomNavigationBar: Container(
        height: 80,
        child: Center(
          child: Text("底部区域"),
        ), // Center
      ), // Container
    ), // Scaffold
  ); // MaterialApp
```

Trae–快速创建组件

- 无状态组件快捷键： statelessW
- 有状态组件快捷键： statefulW



组件生命周期–无状态组件

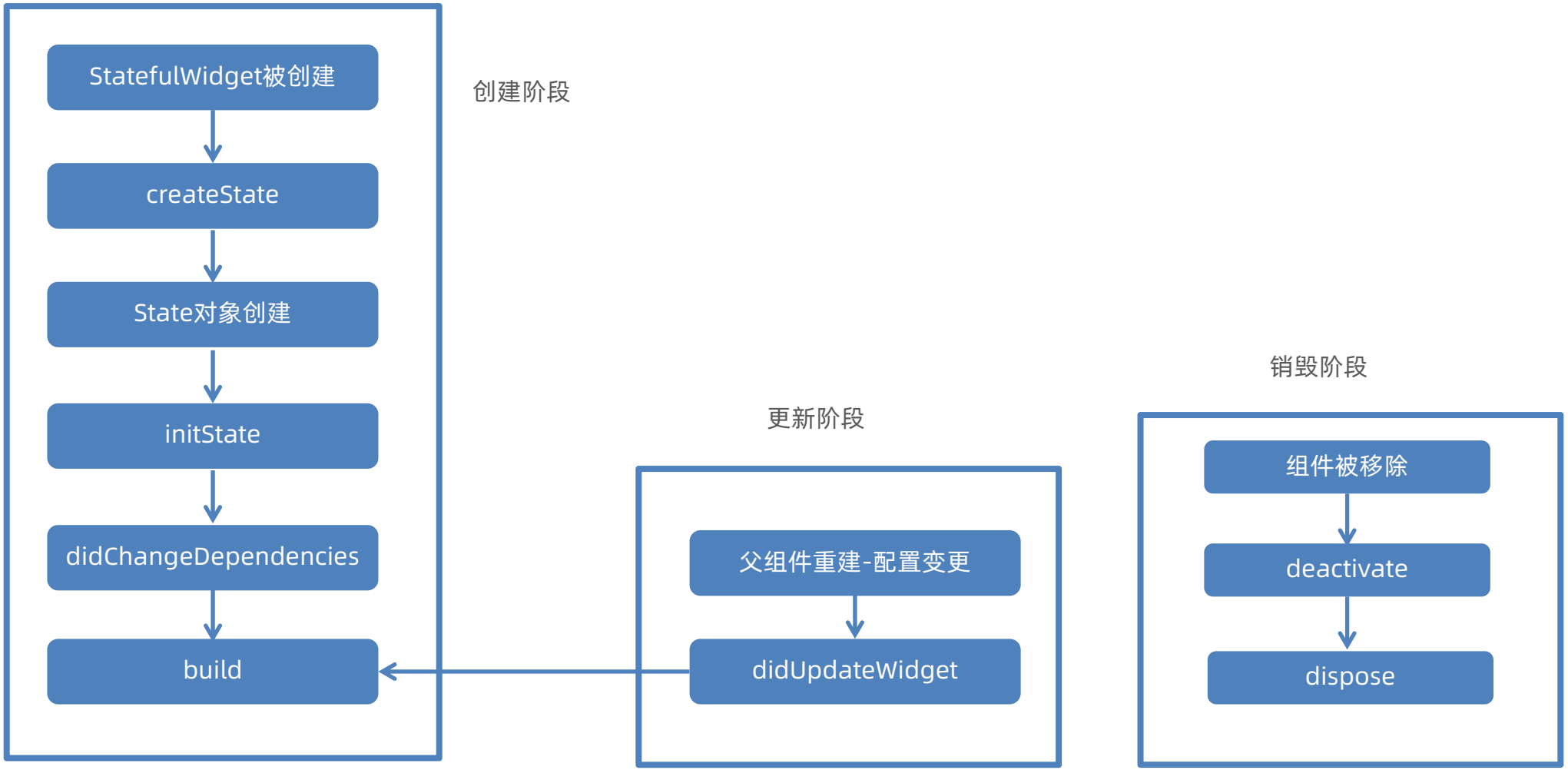
- 无状态组件-唯一阶段

build方法

- 当组件被创建或父组件状态变化导致其需要重新构建时，build方法会被调用

```
class MainPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    print("构建阶段");
    // TODO: implement build
    return MaterialApp(
      title: "标题",
      home: Scaffold(
        appBar: AppBar(
          title: Text("头部区域"),
        ), // AppBar
        body: Container(
          child: Center(
```

组件生命周期–有状态组件



组件生命周期–有状态组件

生命周期阶段	函数名	调用时机与核心任务
创建阶段	createState()	Widget初始化调用，创建State对象
	initState()	State对象插入Widget树立刻执行，仅执行一次
	didChangeDependencies()	initState后立刻执行，当所依赖的InheritedWidget更新时调用，可能多次
构建与更新阶段	build()	构建UI方法，初始化或更新后多次调用
	didUpdateWidget()	父组件传入新配置时调用，用于比较新旧配置
销毁阶段	deactiveate()	当State对象从树中暂时移除时调用
	dispose()	当State对象被永久移除时调用，释放资源，仅执行一次

组件生命周期–有状态组件

```dart
@override
_MainPageState createState() {
  print("createState阶段");
  return _MainPageState();
}
}

class _MainPageState extends State<MainPage> {
  @override
  void initState() {
    print("initState阶段");
    // TODO: implement initState
    super.initState();
  }

  @override
  void didChangeDependencies() {
    print("didChangeDependencies阶段");
    // TODO: implement didChangeDependencies
    super.didChangeDependencies();
```

创建State

initState

didChangeDependencies

build

```dart
@override
Widget build(BuildContext context) {
  print("build阶段");
  return Container(
    child: null,
  );
}
}
```

didUpdateWidget

```dart
@override
void didUpdateWidget(covariant MainPage oldWidget) {
  print("didUpdateWidget阶段");
  // TODO: implement didUpdateWidget
  super.didUpdateWidget(oldWidget);
}
```

```dart
@override
void deactivate() {
  print("deactivate阶段");
  // TODO: implement deactivate
  super.deactivate();
}

@override
void dispose() {
  print("dispose阶段");
  super.dispose();
}
```

deactivate和dispose

总结

- 无状态组件-build
- 有状态组件(创建阶段)： createState -> initState -> didChangeDependencies -> build
- 有状态组件(更新阶段)： didUpdateWidget -> build
- 有状态组件(销毁阶段)： deactivate -> dispose
- 执行一次函数：createState、initState、dispose
- InheritedWidget：专门用于在 Widget 树中自顶向下高效地共享数据，顶层组件提供数据，子孙节点直接获取

事件–点击事件GestureDetector

- 事件：用户与应用程序交互时触发的各种动作,比如触摸屏幕、滑动、点击等
- 点击事件： 当点击某个元素触发的动作
- 常规方案： GestureDetector是 Flutter 中最常用、功能最丰富的手势检测组件。
- 用法： 使用GestureDetector包裹被点击的元素，传入onTap方法

```
child: Center(
  child: GestureDetector(
    child: Text("中部区域"),
    onTap: () {
      print("点击了区域");
    },
  ), // GestureDetector
), // Center
```

```
问题 8  输出  调试控制台  终端  GitLens
Launching lib/main.dart on Chrome in debug mode...
This app is linked to the debug service: ws://127.0.0.1:59388/ws
Debug service listening on ws://127.0.0.1:59388/ws
Connecting to VM Service at ws://127.0.0.1:59388/ws
Connected to the VM Service.
点击了区域
```

事件–组件点击事件

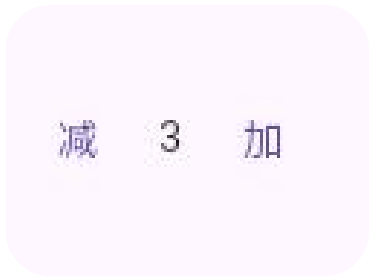
●组件：Flutter提供了多种方式为组件添加点击交互

组件类别	核心组件	主要特点/使用场景
专用按钮组件	ElevatedButton、TextButton、OutlineButton、FloatingActionButton	内置点击动画和样式，通过onPressed参数处理点击逻辑
视觉反馈组件	InkWell	提供点击事件(onTap)，有MaterialDesign风格的水纹扩散效果
其他交互组件	IconButton、Switch、Checkbox	具有特定功能的交互式控件、点击事件(onPressed)

```
child: TextButton(
    onPressed: () {
      print("点击了按钮");
    },
    child: Text("中部区域")), // TextButton
```

状态更新–setState

- 场景：计数器，点击+进行数量+1，点击-进行数量-1，UI视图需要进行相应更新。
- 语法：数据的变化要更新UI视图，需要执行setState方法，setState方法会造成build的重新执行。
- 案例：实现一个计数器案例




```
class _MainPage extends State<MainPage> {
  int count = 0; // 声明数字
  @override
  Widget build(BuildContext context) {
```

```
TextButton(
    onPressed: () {
      setState(() {
        count--;
      });
    },
    child: Text("减")), // TextButton
Text(count.toString()),
TextButton(
    onPressed: () {
      setState(() {
        count++;
      });
    },
    child: Text("加")), // TextButton
```

Flutter组件

布局组件–介绍

●Flutter 提供了丰富强大的布局组件来构建各种用户界面。下面这个表格汇总了最核心的几类布局组件

组件类别	核心组件	主要特点/使用场景
基础容器	Container、Center、Align、Padding	提供装饰、对齐、边距等基础样式和布局控制，是使用频率极高的组件
线性布局	Row、Column	在水平或垂直方向线性排列子组件，是构建界面的基础
弹性布局	Flex, Expanded, Flexible	按照比例分配剩余空间，实现自适应布局，常与 Row和 Column配合使用
层叠布局	Stack, Positioned	让子组件重叠堆叠，用于实现如图片上叠加文字、悬浮按钮等效果
流式布局	Wrap, Flow	当主轴空间不足时自动换行或换列，常用于标签、滤镜等动态宽高内容的排列
滚动布局	ListView, GridView	提供可滚动的列表或网格视图，高效展示大量数据

基础容器–Container

- 定义： Container 是功能丰富的布局组件，是一个多功能组合容器
- 尺寸控制：可通过多种方式定义大小，有明确优先级规则。
- 优先级： 明确宽高 > constraints约束 > 父组件约束 > 自适应组件大小
- 装饰系统：通过decoration属性实现视觉效果，但和color属性互斥
- 布局控制：提供内外边距和对齐方式
- 可选变化：支持绘制时进行矩阵变换，如旋转、倾斜、平移等

基础容器–Container–常见属性

属性类别	关键属性	作用说明
布局定位	alignment	控制其 child（子组件）在容器内部的对齐方式。 • 例如：Alignment.center（居中）、Alignment.topLeft（左上角）
尺寸控制	width/height/constraints	设置容器的宽度和高度/为容器设置更复杂的尺寸约束（如最小/最大宽高）
间距留白	padding/margin	容器内容与容器内边缘之间区域/设置容器外边缘与相邻组件之间区域
装饰效果	color/decoration	为容器设置一个简单的背景颜色/为容器设置复杂的背景装饰
变换效果	transform	对容器及其内容进行矩阵变换
子组件	child	容器内包含的唯一直接子组件

基础容器–Container

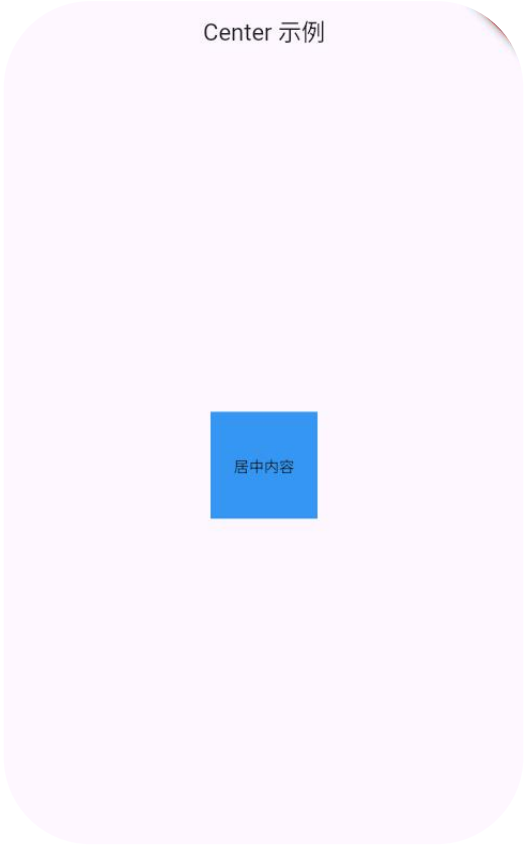
●Container： 基础布局组件，可以方便地容纳一个子组件，并对其施加各种样式、布局约束和变换


```
ainer(
// 外间距
margin: EdgeInsets.all(20.0),
// 尺寸约束
width: 200,
height: 200,
// 复杂的装饰（替代简单的color）
decoration: BoxDecoration(
  color: Colors.blue, // 在这里设置背景色
  borderRadius: BorderRadius.circular(15.0), // 圆角
  border: Border.all(color: Colors.amber, width: 3.0), // 边框
  boxShadow: [
    // 阴影
    BoxShadow(
      color: Colors.grey.withOpacity(0.5),
      blurRadius: 5,
      offset: Offset(2, 2),
    ), // BoxShadow
  ],
), // BoxDecoration
// 内间距
padding: EdgeInsets.all(30.0),
```

```
// 子组件居中对齐
alignment: Alignment.center,
// 旋转变换
transform: Matrix4.rotationZ(0.05),
child: Text(
  'Hello, Container!',
  style: TextStyle(color: Colors.white, fontSize: 16),
), // Text
), // Container
```

基础容器–Center–居中组件

●Center： 将其子组件在父容器的空间内进行水平和垂直方向上的居中排列



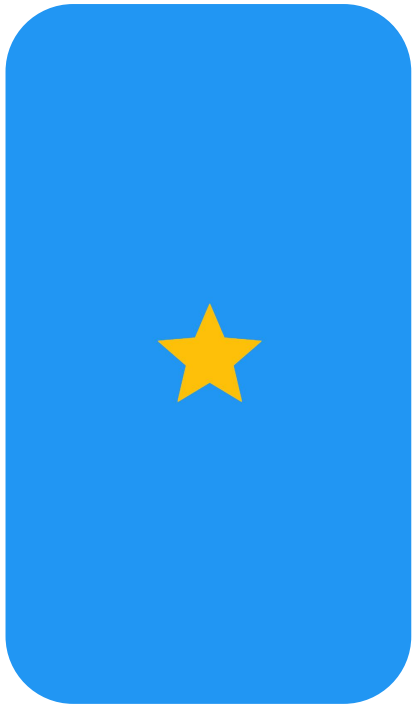

```
eturn MaterialApp(
  title: "标题",
  home: Scaffold(
    appBar: AppBar(title: Text('Center 示例')),
    body: Center(
      child: Container(
        width: 100,
        height: 100,
        color: Colors.blue,
        child: Center(
          child: Text('居中内容'),
        ), // Center
      ), // Container
    ), // Center
  ) // Scaffold
```

基础容器–Center–居中组件

- 应用场景： 页面内容整体居中，如将一个登录表单或一个加载中的提示图标在页面正中显示
- 注意事项：Center不能设置宽高，Center的最终大小取决于其父组件传递给它的约束，Center会向它的父组件申请尽可能大的空间
- 实现固定宽高且居中的组件：Center去包裹一个具有固定宽高的子组件。Container/SizeBox

基础容器–Align–对齐组件

- 作用：精确控制其子组件在父容器空间内的对齐位置
- alignment(对齐方式)：子组件在父容器内的对齐方式。
- widthFactor(宽度因子)：Align的宽度将是子组件宽度乘以该因子
- heightFactor(高度因子)：Align的高度将是子组件高度乘以该因子



```
return MaterialApp(
  title: "标题",
  home: Align(
    alignment: Alignment.center, // 将子组件对齐到父容器居中
    widthFactor: 3.0, // Align的宽度是子图标宽度的3倍
    heightFactor: 3.0, // Align的高度是子图标高度的3倍
    child: Icon(
      Icons.star,
      size: 150,
      color: Colors.amber,
    ), // Icon
  ) // Align
```

基础容器–Align–对齐组件

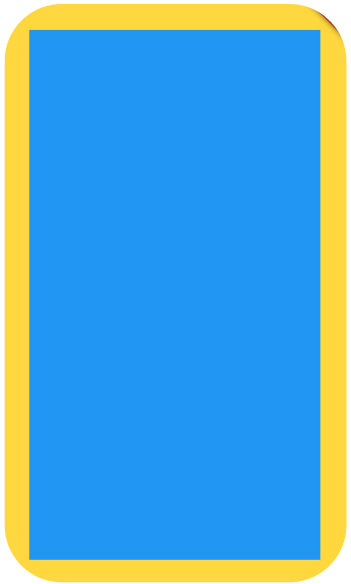
- 与Center的区别：Center是 Align的一个特例，继承自 Align，相当于一个将 alignment属性为居中的Align.center
- 使用场景：当需要将一个组件放置在父容器的特定角落，Align是理想选择。
- 动态尺寸：通过 widthFactor和 heightFactor，可以创建出与子组件大小成比例的容器，动态布局中很有用

基础容器–Padding–内边距组件

●作用：为其子组件添加内边距

属性	类型	作用说明
padding	EdgeInsetsGeometry	必需。定义内边距的大小和方向，通常使用 EdgeInsets类来设置
child	Widget	需要被添加内边距的子组件。

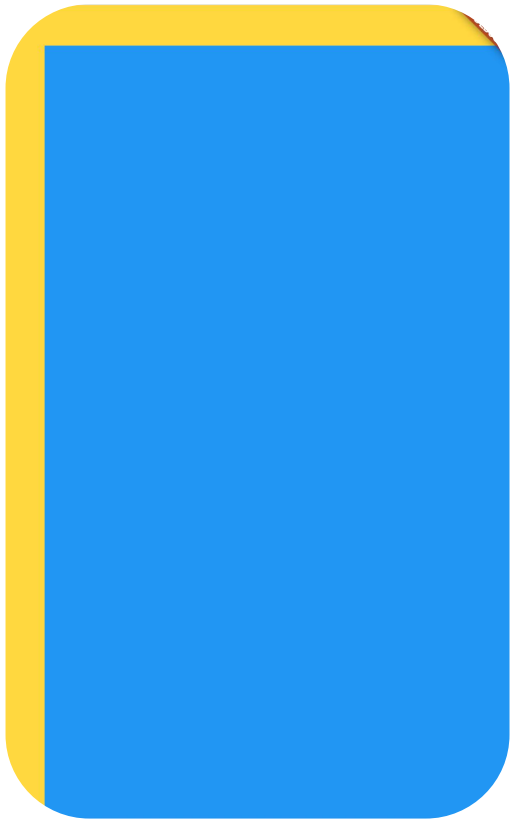
●四个方向设置相同间距-使用EdgeInsets.all进行四个方向设置



```
home: Container(
  decoration: BoxDecoration(color: Colors.amberAccent),
  child: Padding(
    padding: EdgeInsets.all(40),
    child: Container(
      decoration: BoxDecoration(color: Colors.blue),
    ), // Container
  ), // Padding
) // Container
```

基础容器–Padding–内边距组件

- 单独设置某个或某几个方向的边距

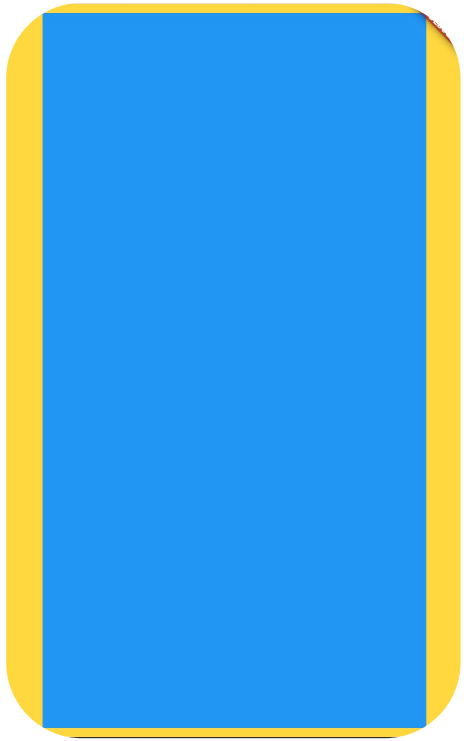


```
home: Container(
  decoration: BoxDecoration(color: Colors.amberAccent),
  child: Padding(
    padding: EdgeInsets.only(left: 40, top: 40),
    child: Container(
      decoration: BoxDecoration(color: Colors.blue),
    ), // Container
  ), // Padding
) // Container
```

- 使用EdgeInsets.only属性进行单独的设置

基础容器–Padding–内边距组件

●设置对称方向的边距。

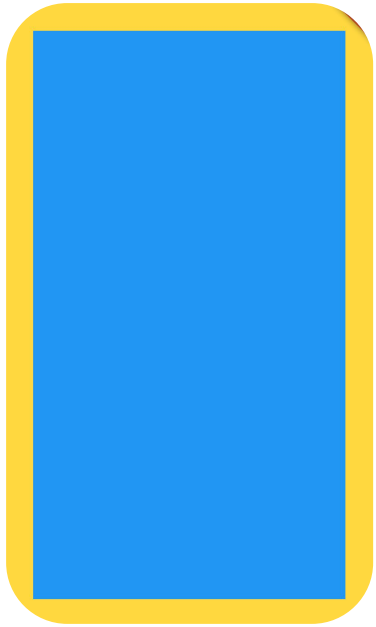


```
home: Container(
  decoration: BoxDecoration(color: Colors.amberAccent),
  child: Padding(
    padding: EdgeInsets.symmetric(vertical: 20, horizontal: 40),
    child: Container(
      decoration: BoxDecoration(color: Colors.blue),
    ), // Container
  ), // Padding
 // Container
```

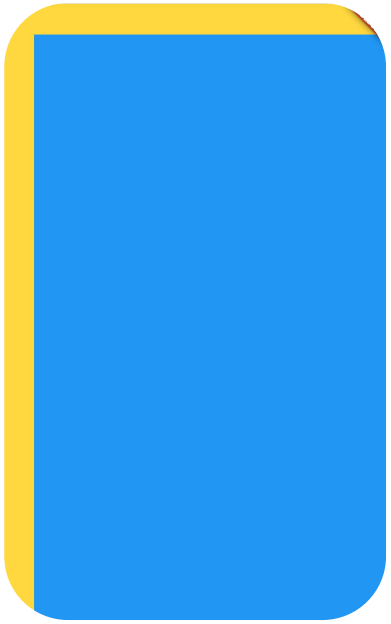
●使用EdgeInsets.symmetric属性进行对称设置，vertical(纵向)、horizontal(横向)

Padding–总结

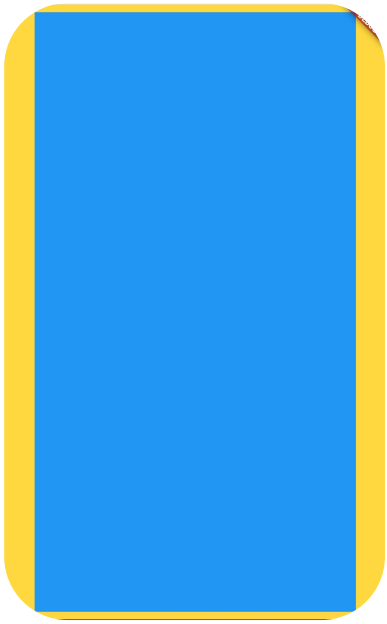
- 特点：功能单一而纯粹，就是添加内边距。如果需求仅是为组件添加间距，那么直接使用 Padding组件
- 区别：Container也有padding属性,单一需求用 Padding组件，复杂样式用 Container



四个方向



单独方向



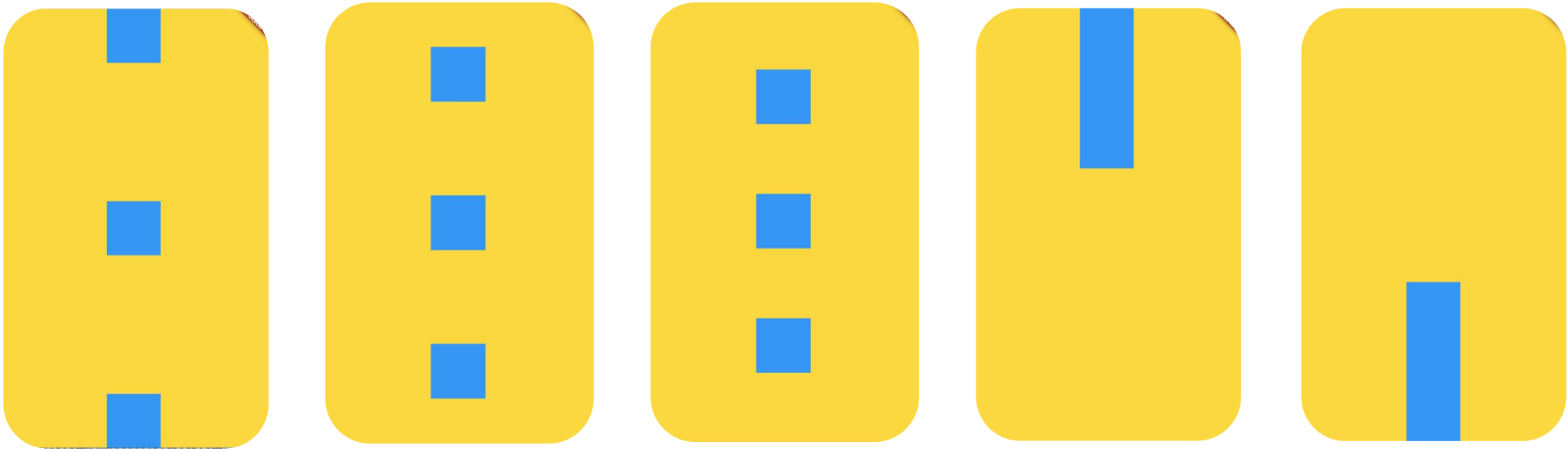
对称方向

线性布局–Column

●作用：用于垂直排列其子组件的核心布局容器

属性	类型	作用说明
mainAxisAlignment	MainAxisAlignment	控制子组件在主轴（垂直方向）上的排列方式，如顶部对齐、居中或均匀分布。
crossAxisAlignment	CrossAxisAlignment	控制子组件在交叉轴（水平方向）上的对齐方式，如左对齐、右对齐或拉伸填满。
mainAxisSize	MainAxisSize	决定 Column本身在垂直方向上的尺寸策略：是占满所有可用空间（max），还是仅仅包裹子组件内容（min）。
children	List<Widget>	需要被垂直排列的子组件列表。

线性布局–Column–主轴

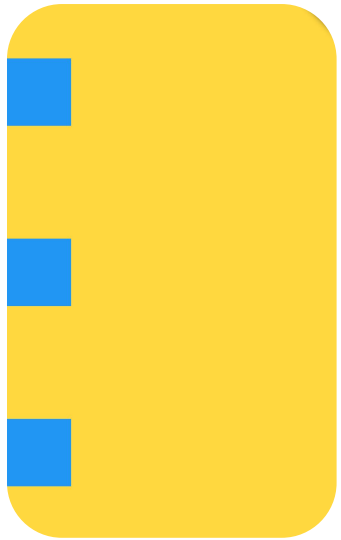


space-between　space-around　space-evently　start　end

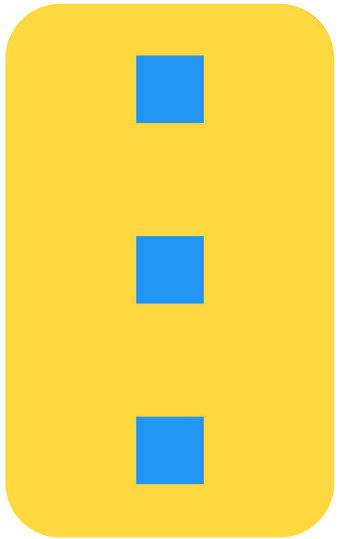
●设置主轴-mainAxisAligment

```
child: Column(
  mainAxisAlignment: MainAxisAlignment.end,
  children: [
    Container(
      width: 100,
      height: 100,
      decoration: BoxDecoration(color: Colors.blue),
    ), // Container
```

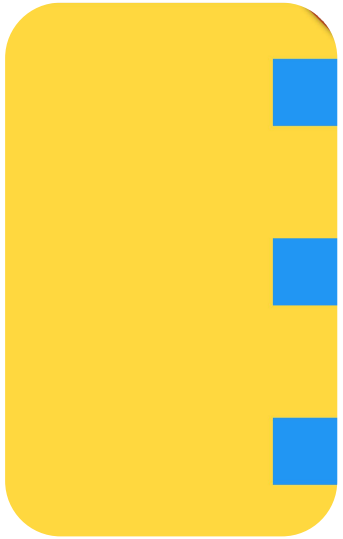
线性布局–Column–交叉轴



start



center



end

●设置交叉轴-crossAxisAlignment

```
child: Column(
  mainAxisAlignment: MainAxisAlignment.spaceAround,
  crossAxisAlignment: CrossAxisAlignment.end,
  children: [
    Container(
      decoration: BoxDecoration(color: Colors.blue),
      width: 100,
      height: 100,
    ), // Container
    Container(
      decoration: BoxDecoration(color: Colors.blue),
      width: 100,
      height: 100,
```

线性布局–Column

- 适用场景：几乎在所有需要垂直排列元素的界面中都能看到它的身影

  表单：如登录页面的用户名输入框、密码输入框和登录按钮的垂直排列。

  设置列表： 如设置页面中多个选项项的垂直堆叠。

  卡片布局： 如新闻流中多个新闻卡片的垂直排列。

  图文混排：如商品详情页的图片、标题、描述和价格等信息从上到下的展示。

- 注意事项：Column本身不支持滚动，如果内容超出，需要使用ListView或者SingleChildScrollView包裹

  明确尺寸约束，父组件的大小直接影响Column的最终大小和子组件的布局行为

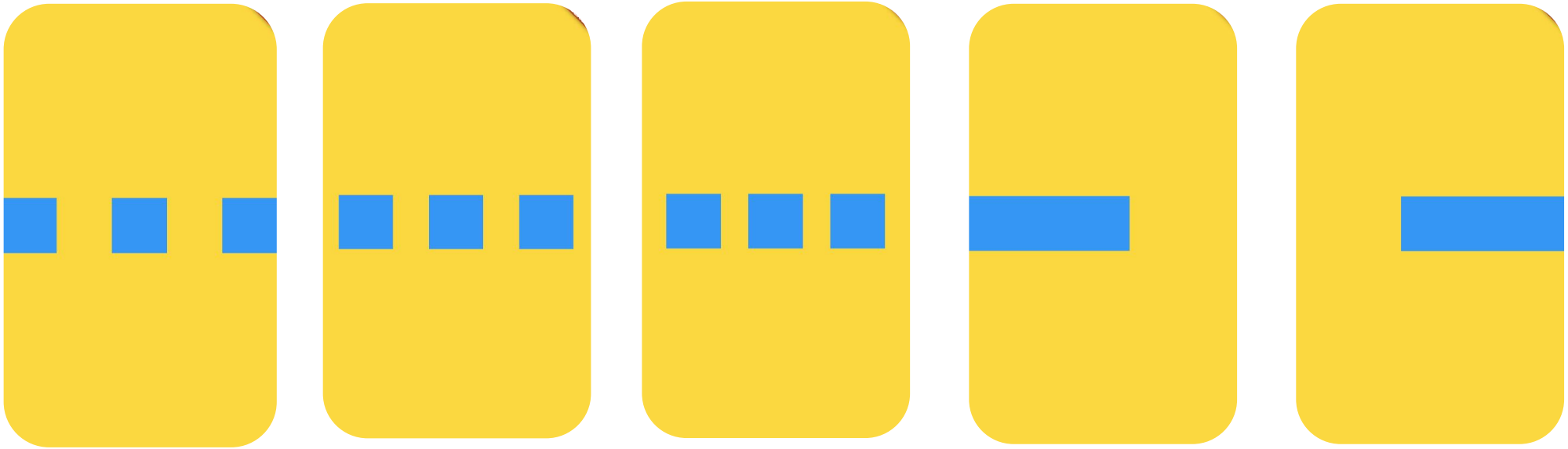
  避免过度嵌套，过深的嵌套会影响性能并增加代码维护难度

线性布局–Row

●作用：用于水平排列其子组件的核心布局容器

属性	类型	作用说明
mainAxisAlignment	MainAxisAlignment	控制子组件在主轴（水平方向）上的排列方式，如顶部对齐、居中或均匀分布。
crossAxisAlignment	CrossAxisAlignment	控制子组件在交叉轴（垂直方向）上的对齐方式，如左对齐、右对齐或拉伸填满。
mainAxisSize	MainAxisSize	决定 Row本身在水平方向上的尺寸策略：是占满所有可用空间（max），还是仅仅包裹子组件内容（min）。
children	List<Widget>	需要被水平排列的子组件列表。

线性布局–Row–主轴

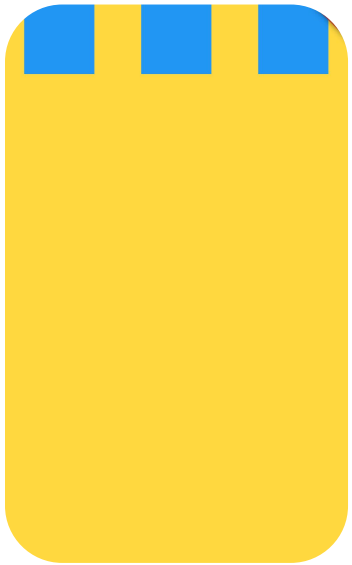


space-between　space-around　space-evenly　start　end

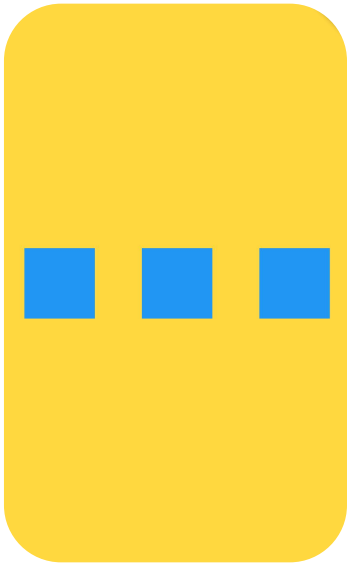
●设置主轴-mainAxisAligment

```
child: Row(
  mainAxisAlignment: MainAxisAlignment.end,
  children: [
    Container(
      width: 100,
      height: 100,
      decoration: BoxDecoration(color: Colors.blue),
    ), // Container
```

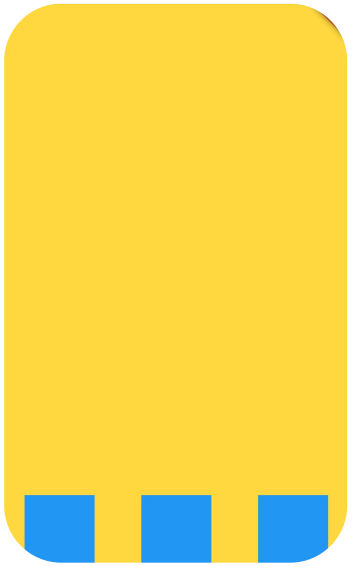
线性布局–Row–交叉轴



start



center



end

●设置交叉轴-crossAxisAlignment

```
child: Row(
  mainAxisAlignment: MainAxisAlignment.spaceAround,
  crossAxisAlignment: CrossAxisAlignment.end,
  children: [
    Container(
      decoration: BoxDecoration(color: Colors.blue),
      width: 100,
      height: 100,
    ), // Container
    Container(
```

线性布局–Row

●适用场景：几乎在需要水平排列元素的界面中都能看到它的身影

导航栏：如顶部或底部的标签栏、按钮组。

图文混排： 如列表项左侧的图标与右侧的文本描述。

表单行： 如标签和输入框的组合

●注意事项：Row本身不支持滚动，如果内容超出，需要使用ListView或者SingleChildScrollView包裹

明确尺寸约束，父组件的大小直接影响Row的最终大小和子组件的布局行为

避免过度嵌套，过深的嵌套会影响性能并增加代码维护难度

弹性布局–Flex

●作用：允许沿一个主轴（水平或垂直）排列其子组件，灵活地控制这些子组件在主轴上的尺寸比例和空间分配

属性	类型	作用说明
direction	Axis.horizontal/Axis.vertical	主轴方向，决定子组件的排列方向
mainAxisAlignment	MainAxisAlignment	子组件在主轴方向上的对齐方式。
crossAxisAlignment	CrossAxisAlignment	子组件在交叉轴方向上的对齐方式
mainAxisSize	MainAxisSize	Flex 容器自身在主轴上的尺寸策略

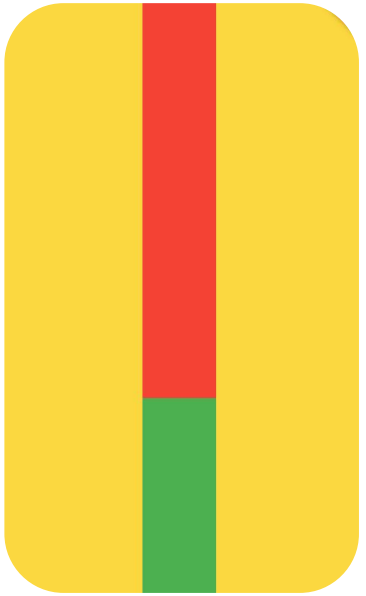
●子组件：Flex的子组件常使用Expanded或Flexible来控制空间分配

●Flex是Column和Row的结合体

弹性布局–Flex/Expanded/Flexible

- Expanded/Flexible作为Flex的子组件通过flex属性来分配Flex组件空间

```
child: Flex(
  direction: Axis.horizontal, // 水平排列
  children: <Widget>[
    Expanded(
      flex: 2, // 占据剩余空间的 2/3
      child: Container(color: Colors.red, height: 100),
    ), // Expanded
    Expanded(
      flex: 1, // 占据剩余空间的 1/3
      child: Container(color: Colors.green, height: 100),
    ), // Expanded
  ], // <Widget>[]
), // Flex
```



```
direction: Axis.vertical, // 水平排列
children: <Widget>[
  Expanded(
    flex: 2, // 占据剩余空间的 2/3
    child: Container(color: Colors.red, width: 100),
  ), // Expanded
  Expanded(
    flex: 1, // 占据剩余空间的 1/3
    child: Container(color: Colors.green, width: 100),
  ), // Expanded
], // <Widget>[]
```

- Flex 布局受其父组件传递的约束影响。确保父组件提供了适当的布局约束
- Expanded 与 Flexible 的区别: Expanded强制子组件填满所有剩余空间, Flexible根据自身大小调整,不强制占满空间

弹性布局小案例

●场景：顶部和底部高度固定，中间区域充满剩余空间

```
child: Flex(
  direction: Axis.vertical,
  children: [
    Container(
      width: double.infinity,
      height: 100,
      color: Colors.blue,
    ), // Container
    Expanded(
      child: Container(
      width: double.infinity,
      height: 100,
      color: Colors.blueGrey,
    )), // Container // Expanded
    Container(
      width: double.infinity,
      height: 100,
      color: Colors.red,
    ) // Container
  ],
), // Flex
```

流式布局–Wrap

●作用：流式布局组件，当子组件在主轴方向上排列不下时，它会自动换行（或换列）

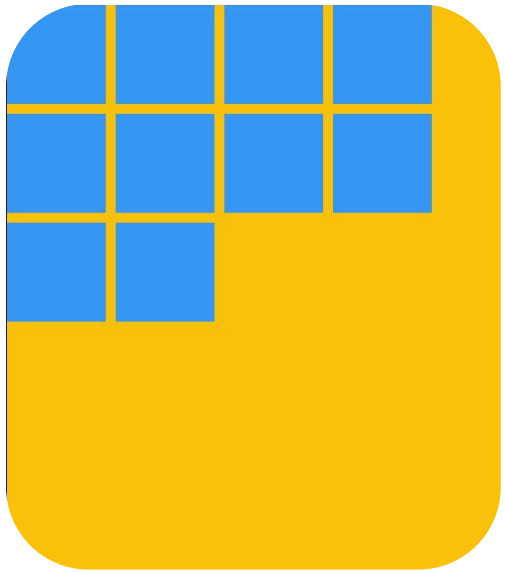
属性	常用值	作用说明
direction	Axis.horizontal(水平)/Axis.vertical（垂直）	设置主轴方向，即排列方向。
spacing	数值	主轴方向上，子组件之间的间距
runSpacing	数值	交叉轴方向上，行（或列）之间的间距
alignment	WrapAlignment	子组件在主轴方向上的对齐方式。
runAlignment	WrapAlignment	交叉轴方向上的对齐方式

●注意：Column/Row/Flex内容超出均不会换行

●Wrap组件更像是‘ Flex组件加了换行特性’

Wrap流式布局

●当子组件内容是根据数据动态生成时，使用 Wrap 可以确保布局始终适配



```
child: Wrap(
  spacing: 10,
  runSpacing: 10,
  direction: Axis.horizontal,
  children: getList(),
), // Wrap
```

```
List<Widget> getList() {
  return List.generate(10, (index) {
    return Container(
      color: Colors.blue,
      height: 100,
      width: 100,
    ); // Container
  }); // List.generate
}
```

List.generate是一个构造器，用于快速创建长度固定且每个元素可以通过索引号确定的列表

语法：List.generate(int count, E generator(int index), {bool growable: false})

层叠布局–Stack/Positioned

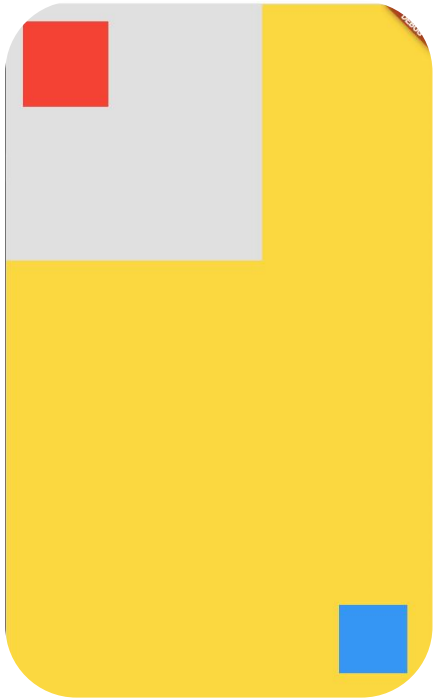
●作用：层叠布局组件，允许你将多个子组件按照 Z 轴（深度方向）进行叠加排列。

属性	类型	作用说明
alignment	AlignmentGeometry	控制非定位子组件在 Stack内的对齐方式，默认左上角
fit	StackFit	控制非定位子组件如何适应 Stack的尺寸
clipBehavior	Clip	控制子组件超出 Stack边界时的裁剪方式
children	List<Widget>	需要被层叠排列的子组件列表

●搭档：Positioned组件是 Stack的黄金搭档，对子组件进行精确定位控制。Positioned必须作为 Stack的直接子组件。

Positioned通过left、right、top、bottom 来将子组件“钉”在 Stack的某个角落或边缘

层叠布局–Stack/Positioned



```
child: Stack(
  children: [
    // 底层背景
    Container(color: Colors.grey[300], width: 300, height: 300),
    // 使用 Positioned 精确定位
    Positioned(
      left: 20,
      top: 20,
      child: Container(width: 100, height: 100, color: Colors.red),
    ), // Positioned
    Positioned(
      right: 30,
      bottom: 30,
      child: Container(width: 80, height: 80, color: Colors.blue),
    ), // Positioned
  ],
), // Stack
```

层叠布局–Stack/Positioned

●适用场景：几乎在叠加效果的界面中都能看到它的身影

叠加效果：图像上的水印、文本、徽章。

浮层交互： 如模态对话框、提示弹窗、操作菜单。

悬浮按钮： 按钮悬浮在特定内容之上

●注意事项：Stack中子组件的层叠顺序由其在 children列表中的顺序决定

明确尺寸约束，父组件的大小直接影响Stack的最终大小和子组件的布局行为

避免在 Stack中嵌套过多需要动态更新的子组件，保持渲染性能

文本组件–Text

●作用：在用户界面中显示文本的基础组件

属性	类型	作用说明
data	String	必需。要显示的文本内容。
style	TextStyle	文本样式，可设置颜色、大小、粗细等。
textAlign	TextAlign	文本在容器内的水平对齐方式，如 .left, .center。
maxLines	int	文本显示的最大行数。

文本组件–Text

Hello, Flutter!

```
home: Center(
  child: Text(
    'Hello, Flutter!',
    style: TextStyle(
      fontSize: 40.0, // 字体大小
      color: Colors.blue, // 字体颜色
      fontWeight: FontWeight.bold, // 字重（粗细）
      fontStyle: FontStyle.italic, // 字体样式（如斜体）
      decoration: TextDecoration.underline, // 装饰（如下划线）
      decorationColor: Colors.deepOrange), // TextStyle
  ), // Text
```

基本使用

这是一段非常…

```
  child: Text(
    '这是一段非常长的文本，它可能会超出容器的宽度。',
    style: TextStyle(
      fontSize: 20.0, // 字体大小
      color: Colors.blue, // 字体颜色
      fontWeight: FontWeight.bold, // 字重（粗细）
      fontStyle: FontStyle.italic, // 字体样式（如斜体）
      decoration: TextDecoration.underline, // 装饰（如下划线）
    ),   // TextStyle
    maxLines: 1, // 限制为单行
    overflow: TextOverflow.ellipsis, // 超出部分显示省略号...
    softWrap: true, // 允许换行（如果maxLines>1）
    textAlign: TextAlign.center, // 文本在容器内居中对齐
  ), // Text
), // Container
```

文本超出

文本组件–Text/TextSpan

●如果需要在同一段文本中显示不同样式，可用Text.rich构造函数配合TextSpan来实现

Hello Flutter!

```
child: Text.rich(
  TextSpan(
    text: 'Hello ', // 默认样式的文本
    style: TextStyle(fontSize: 36.0),
    children: [
      TextSpan(
        text: 'Flutter', // 特殊样式的子文本
        style: TextStyle(
          fontWeight: FontWeight.bold,
          color: Colors.green,
        ), // TextStyle
      ), // TextSpan
      TextSpan(text: '!'), // 继续默认样式
    ],
  ), // TextSpan
), // Text.rich
```

文本组件–Text

- 适用场景：所有的文本显示都需要Text组件
- 注意事项：Text组件本身和其 TextStyle中都可能有 overflow等属性,Text组件属性优先级更高

  假如文本过长请务必设置 maxLines和 overflow。

  大量重复使用的文本样式，建议统一定义， 有助于保持一致性并提升性能

图片组件–Image

●作用：在用户界面中显示图片的核心部件

●图片分类：

分类	作用说明
Image.asset()	加载项目资源目录（assets）中的图片。需要在 pubspec.yaml文件中声明资源路径
Image.network()	直接从网络地址加载图片
Image.file()	加载设备本地存储中的图片文件
Image.memory()	加载内存中的图片数据

图片组件–Image

●常用属性：

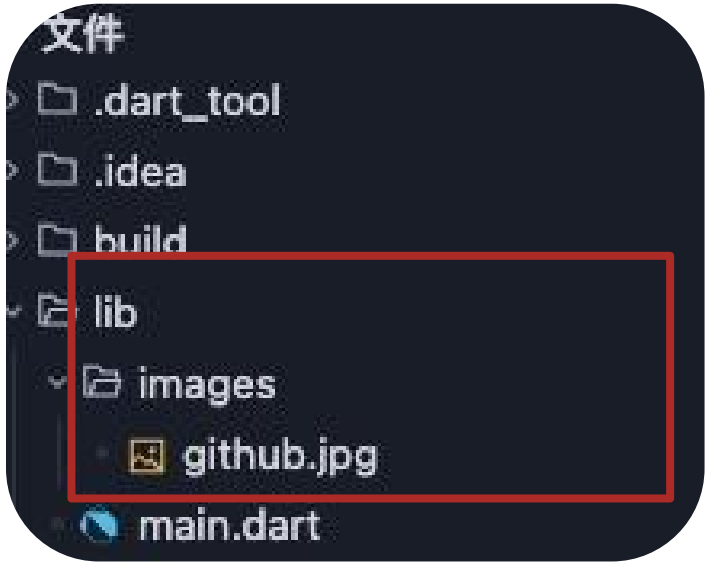
分类	类型	作用说明
width/ height	double	设置图片显示区域的宽度和高度
fit	BoxFit	控制图片如何适应其显示区域，例如是否拉伸、裁剪或保持原比例
alignment	AlignmentGeometry	图片在其显示区域内的对齐方式，如 Alignment.center
repeat	ImageRepeat	当图片小于显示区域时，设置是否以及如何重复平铺图片

图片组件–Image.asset

配置pubspec.yaml文件

```
# To add assets to your application, add
assets:
  - lib/images/
#   - images/a dot ham.jpeg
```

放置图片



使用Image.asset图片

```
home: Container(
  decoration: BoxDecoration(color: Colors.amberAccent),
  child: Center(
    child: Image.asset(
      "lib/images/github.jpg",
      fit: BoxFit.cover,
      width: 400,
      height: 400,
    ), // Image.asset
  ), // Center
) // Container
```

图片组件–Image.network(网络图片)

```
home: Container(
  decoration: BoxDecoration(color: Colors.amberAccent),
  child: Image.network(
    "https://gips2.baidu.com/it/u=752342805,1530597825&fm=3074&app=3074&f=PNG?w=2560&h=1440"
    fit: BoxFit.cover,
    width: 400,
    height: 400,
  ), // Image.network
), // Center
) // Container
```

●注意：Android/HarmonyOS/iOS使用Image.network需要配置网络权限，

后续在每个环境讲解中会进行讲解

文本输入组件–TextField

●作用：实现文本输入功能的核心组件

属性	作用说明
controller	文本编辑器控制器，用于获取、设置文档内容及监听变化
decortation	当时输入框的外观、如标签、提示文字、图标、边框等
style	定义输入文本的样式
maxLines	最大行数
onChanged	输入内容发生变化时执行的回调函数
onSubmitted	用户提交输入时的回调函数

文本输入组件–TextField

有状态组件

声明controller

```
TextEditingController _phoneController = TextEditingController();
TextEditingController _codeController = TextEditingController();
```

```
child: TextButton(
    onPressed: () {
      print(
          '${_phoneController.text}-${_codeController.text}');
    },
    child: Text(
      "登录",
      style: TextStyle(color: Colors.white),
    )), // Text // TextButton
) // Container
```

```
body: Container(
    alignment: Alignment.center,
    width: double.infinity,
    height: double.infinity,
    color: Colors.white,
    padding: EdgeInsets.all(20),
    child: Column(
      children: [
        TextField(
          controller: _phoneController,
          decoration: InputDecoration(
            contentPadding: EdgeInsets.only(left: 20),
            filled: true,
            hintText: '请输入',
            fillColor: const Color.fromARGB(245, 245, 245, 245, 221),
            border: OutlineInputBorder(
                borderSide: BorderSide.none,
                borderRadius: BorderRadius.circular(25)), // OutlineInputBorder
          ), // InputDecoration
        ), // TextField
```

文本输入组件–TextField

●使用： 使用TextField必须使用有状态组件

使用 TextEditingController管理输入内容、onChanged可以监听数据变化

decoration属性下的 InputDecoration来定制如 边框、背景、提示文字

obscureText设置为 true可隐藏输入内容，用于密码输入框

文本输入组件–TextField–完整代码

```
class MainPage extends StatefulWidget {
  MainPage({Key? key}) : super(key: key);

  @override
  _MainPageState createState() => _MainPageState();
}

class _MainPageState extends State<MainPage> {
  TextEditingController _phoneController = TextEditingController();
  TextEditingController _codeController = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("登录"),
        ),
        body: Container(
          alignment: Alignment.center,
          width: double.infinity,
          height: double.infinity,
          color: Colors.white,
          padding: EdgeInsets.all(20),
          child: Column(
            children: [
              TextField(
                controller: _phoneController,
                decoration: InputDecoration(
                  contentPadding: EdgeInsets.only(left: 20),
                  filled: true,
                  hintText: '请输入账号',
                  fillColor: const Color.fromARGB(245, 245, 245, 221),
```

常用滚动组件

组件	特点	使用场景
SingleChildScrollView	让单个子组件可以用滚动，所有内容一次性渲染	长表单、设置页、内容不固定但是总量不多的页面
ListView	线性列表，通过builder可以实现懒加载，性能优异	聊天记录、新闻、常见的单列滚动的数据列表
GridView	网格布局列表，支持懒加载，可以固定列数	图片墙、商品网格、应用图标列表
CustomScrollView	复杂布局方案，通过组合多个Sliver组件实现滚动	电商首页、社交App个人主页多个滚动紧密联动
PageView	整页滚动效果，支持横向和纵向	应用引导页、图片轮播图、书籍翻页

常用滚动组件–SingleChildScrollView

●用法：包裹一个子组件，让单个子组件具备滚动能力。


```
body: SingleChildScrollView(
  padding: EdgeInsets.all(10),
  child: Column(
    children: List.generate(100, (index) {
      return Container(
          color: Colors.blue,
          height: 100,
          margin: EdgeInsets.only(top: 10),
          child: Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Text(
                '我是第${index + 1}个',
                style: TextStyle(color: Colors.white, fontSize: 30),
              ) // Text
            ],
          )); // Row // Container
    }), // List.generate
  ), // Column
), // SingleChildScrollView
```

控制滚动–SingleChildScrollView

- controller：给组件的controller绑定ScrollController对象


```dart
class _MainPageState extends State<MainPage> {
  ScrollController _controller = ScrollController();
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("登录"),
        ), // AppBar
        body: Stack(
          children: [
            SingleChildScrollView(
                controller: _controller,
                padding: EdgeInsets.all(10),
```

声明controller并绑定

使用animateTo方法滚动

```dart
Positioned(
  right: 10,
  bottom: 10,
  child: GestureDetector(
    onTap: () {
      _controller.animateTo(0,
          duration: Duration(seconds: 10), curve: Curves.easeOut);
    },
    child: Container(
      width: 80,
      height: 80,
      decoration: BoxDecoration(
          color: Colors.red,
          borderRadius: BorderRadius.circular(40)), // BoxDecoration
      alignment: Alignment.center,
      child: Text("去顶部", style: TextStyle(color: Colors.white)),
    )), // Container // GestureDetector
) // Positioned
```

常用滚动组件–SingleChildScrollView

- 子组件：只能包含一个子组件，如果滚动多个组件，通常将其嵌套在Column或Row组件中
- 滚动方向：通过 scrollDirection属性控制，默认为垂直方向 (Axis.vertical)，也可设置为水平方向 (Axis.horizontal)
- 特点：一次性构建所有子组件，如果嵌套的 Column或 Row中包含大量子项，可能会导致性能问题，建议使用 ListView
- 控制滚动： 绑定一个ScrollController对象给controller对象，使用animateTo/jumpTo方法控制滚动
- 滚动到顶部： controller.jumpTo(0)
- 滚动到底部： controller.jumpTo(controller.position.maxScrollExtent)

常用滚动组件–ListView

- 作用：用于构建可滚动列表的核心部件,并提供流畅滚动体验
- 方式：提供多种构造函数，如默认构造函数、ListView.builder、ListView.separated
- 机制：采用按需渲染（懒加载），只构建当前可见区域的列表项，极大提升长列表性能


```
), // AppBar
body: ListView(
  padding: EdgeInsets.all(20),
  children: List.generate(100, (index) {
    return Container(
      margin: EdgeInsets.only(top: 10),
      color: Colors.blue,
      height: 100,
      alignment: Alignment.center,
      child: Text(
        "第${index + 1}个",
        style: TextStyle(color: Colors.white, fontSize: 30),
      ), // Text
    ); // Container
  }), // List.generate
))); // ListView // Scaffold // MaterialApp
```

- 特点：默认构造函数适用于静态数量有限数据一次性构建所有表项

ListView–builder模式

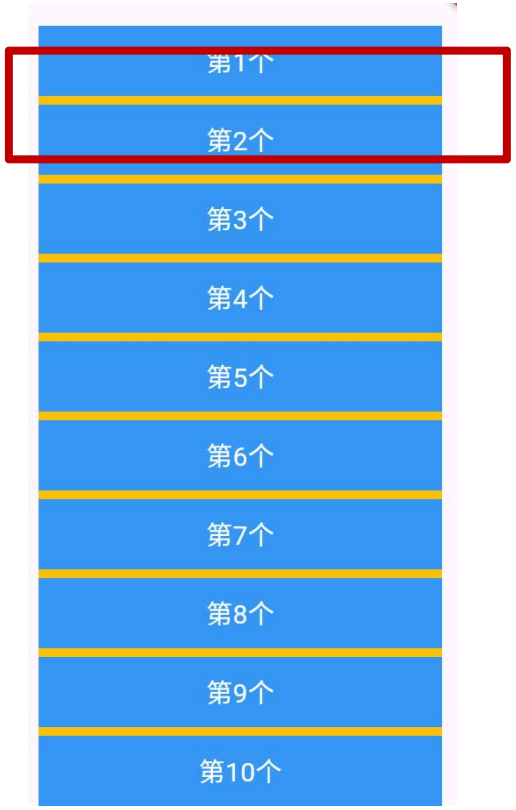
- 作用：处理长列表或动态数据的首选和推荐方式
- 方式：接受一个 itemBuilder回调函数来按需构建列表项，通过itemCount控制列表长度


```
body: ListView.builder(
  itemCount: 100,
  padding: EdgeInsets.all(20),
  itemBuilder: (BuildContext context, int index) {
    return Container(
      margin: EdgeInsets.only(top: 10),
      height: 80,
      color: Colors.blue,
      alignment: Alignment.center,
      child: Text("第${index + 1}个",
          style: TextStyle(color: Colors.white, fontSize: 30)), // Text
    ); // Container
  },
))); // ListView.builder // Scaffold // MaterialApp
```

- 优势：按需构建，不会在初始化时将所有列表项都创建，而是根据用户的滚动行为，动态地创建和销毁列表项

ListView–separated模式

- 作用：在 ListView.builder的基础上，额外提供了构建分割线的能力
- 方式：需要同时提供itemBuilder、separatorBuilder、itemCount三个属性




```
body: ListView.separated(
  itemCount: 100,
  padding: EdgeInsets.all(20),
  separatorBuilder: (BuildContext context, int index) {
    return Container(
      height: 10,
      color: Colors.amber,
      alignment: Alignment.center,
    ); // Container
  },
  itemBuilder: (BuildContext context, int index) {
    return Container(
      height: 80,
      color: Colors.blue,
      alignment: Alignment.center,
      child: Text("第${index + 1}个",
          style: TextStyle(color: Colors.white, fontSize: 30)), // Text
    ); // Container
  },
)); // ListView.separated // Scaffold // MaterialApp
```

常用滚动组件–GridView

●作用：用于创建二维可滚动网格布局的核心组件

●方式：提供多种构建方式，GridView.count、GridView.extent、GridView.builder等

GridView默认构造方式-(写起来最过繁琐，很少使用)

GridView.count-基于固定列数的网格布局(最常用之一)

GridView.extent-基于固定子项最大宽度/高度的网格布局(最常用之二)

GridView.builder用于网格项数量巨大或动态生成的情况，需要接收gridDelegate布局委托属性

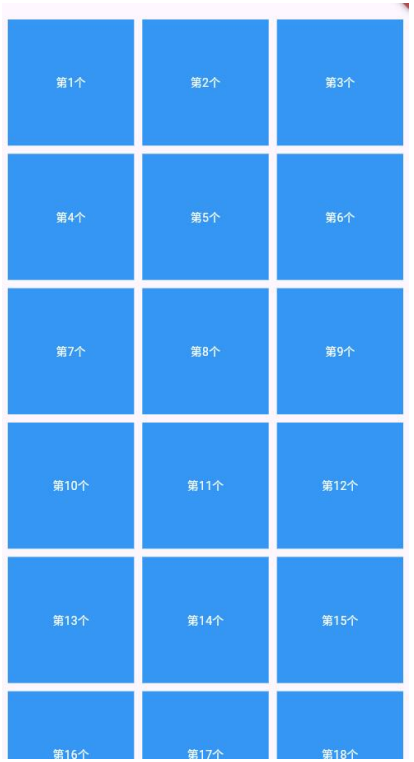
●gridDelegate：SliverGridDelegateWithFixedCrossAxisCount：固定列数 mainAxisSpacing 主轴间距

SliverGridDelegateWithMaxCrossAxisExtent：最大宽度 crossAxisSpacing 交叉轴间距

scrollDirection设置滚动方向横向/纵向(默认)

GridView–GridView.count构造

●作用：使用GridView.count创建固定列数网格

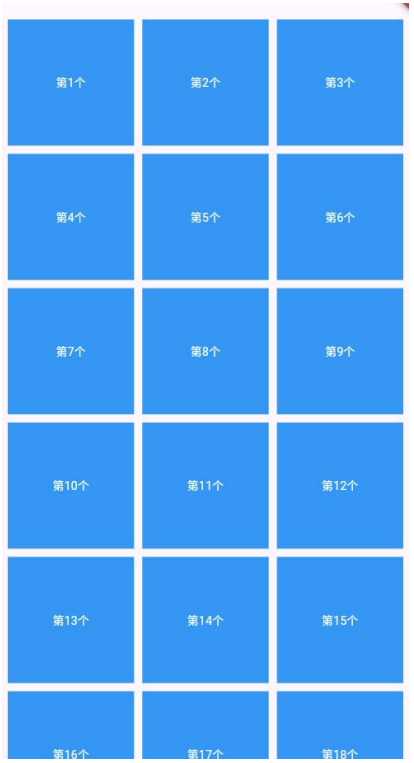



```
body: GridView.count(
  padding: EdgeInsets.all(10),
  crossAxisCount: 3,
  mainAxisSpacing: 10,
  crossAxisSpacing: 10,
  children: List.generate(100, (index) {
    return Container(
      color: Colors.blue,
      alignment: Alignment.center,
      child: Text('第${index + 1}个',
          style: TextStyle(color: Colors.white)), // Text
    ); // Container
  }), // List.generate
) // GridView.count
```

●GridView.count以列数为优先。指定网格多少列，Flutter 自动计算列的宽度，在空间内均匀排列

GridView–GridView.extent构造

●作用：使用GridView.extent指定子项最大宽度或者高度

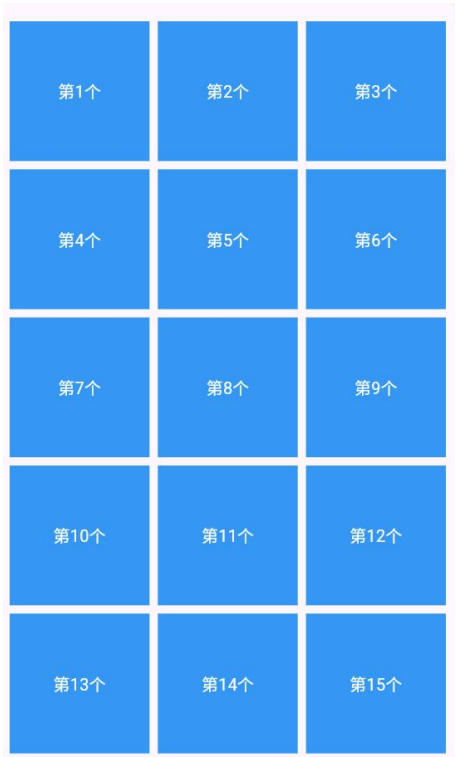



```
body: GridView.extent(
  padding: EdgeInsets.all(10),
  maxCrossAxisExtent: 200,
  mainAxisSpacing: 10,
  crossAxisSpacing: 10,
  children: List.generate(100, (index) {
    return Container(
      color: Colors.blue,
      alignment: Alignment.center,
      child: Text('第${index + 1}个',
          style: TextStyle(color: Colors.white)), // Text
    ); // Container
  }), // List.generate
) // GridView.extent
```

●GridView.extent通过maxCrossAxisExtent设置子项最大宽度/高度来计算横向或者纵向有多少列

GridView–GridView.builder构造

- 作用：使用GridView.builder实现动态长网格-(懒加载，只渲染可见区域)
- 注意：接收gridDelegate布局委托、itemBuilder构建函数、itemCount构建数量




```
body: GridView.builder(
    padding: EdgeInsets.all(20),
    gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 3,
        mainAxisSpacing: 10,
        crossAxisSpacing: 10,
        childAspectRatio: 1), // SliverGridDelegateWithFixedCrossAxisCount
    itemCount: 100,
    itemBuilder: (BuildContext context, int index) {
      return Container(
        alignment: Alignment.center,
        color: Colors.blue,
        child: Text("第${index + 1}个",
            style: TextStyle(color: Colors.white, fontSize: 20)), // Text
      ); // Container
    }), // GridView.builder
```

gridDelegates属性: SliverGridDelegateWithFixedCrossAxisCount / SliverGridDelegateWithMaxCrossAxisExtent

自定义滚动容器–CustomScrollView

- 作用：用于组合多个可滚动组件（如列表、网格），实现统一协调的滚动效果
- Sliver： Flutter 中描述可滚动视图内部一部分内容的组件，它是滚动视图的"切片"
- 用法：通过 slivers属性接收一个 Sliver 组件列表
- Siiver组件对应关系：

```
body: CustomScrollView(
  slivers: [
    // SliverToBoxAdapter
    // SliverGrid
    // SliverList
  ],
), // CustomScrollView
```

SliverList => ListView

SliverGrid => GridView

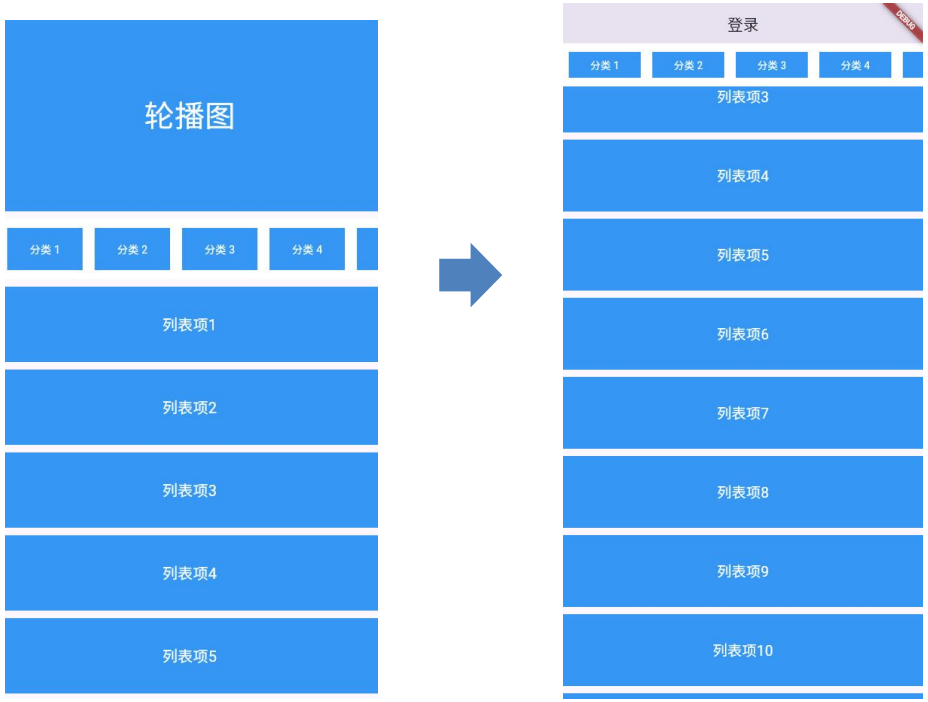
SliverAppBar => AppBar

SliverPadding => Padding

SliverToBoxAdapter => ToBoxAdapter (用于包裹普通 Widget)

SliverPersistentHeader(粘性吸顶)

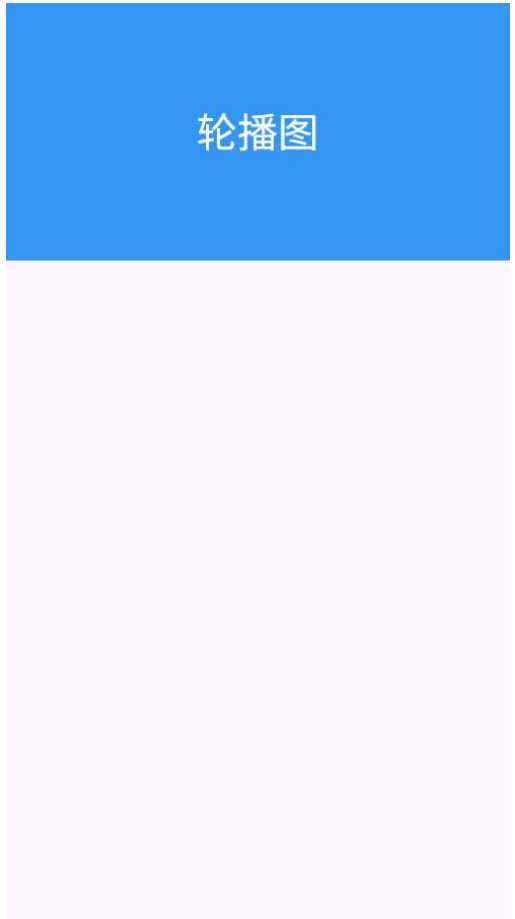
自定义滚动容器–CustomScrollView



语法：

```
CustomScrollView
    slivers:
        SliverToBoxAdapter
                轮播图
        SliverPersistentHeader(粘性吸顶)
                分类
        SliverList(列表)
```

案例代码实现–顶部轮播图




```
body: CustomScrollView(
  slivers: [
    SliverToBoxAdapter(
      child: Container(
        alignment: Alignment.center,
        color: Colors.blue,
        height: 260,
        child: Text("轮播图",
            style: TextStyle(color: Colors.white, fontSize: 40)), // Text
      ), // Container
    ), // SliverToBoxAdapter
  ],
), // CustomScrollView
```

案例代码实现–粘性吸顶分类SliverPersistentHeader

SliverPersistentHeader：给delegate属性赋值一个继承SliverPersistentHeaderDelegate的对象实例

设置固定属性pinned为true

```
SliverPersistentHeader(
  pinned: true, // 固定显示
  delegate: _StickyCategoryDelegate(),
), // SliverPersistentHeader
```

maxExtent：展开时最大高度

minExtent： 收缩时最小高度

shouldRebuild： 是否重建

build： 返回构建Widget

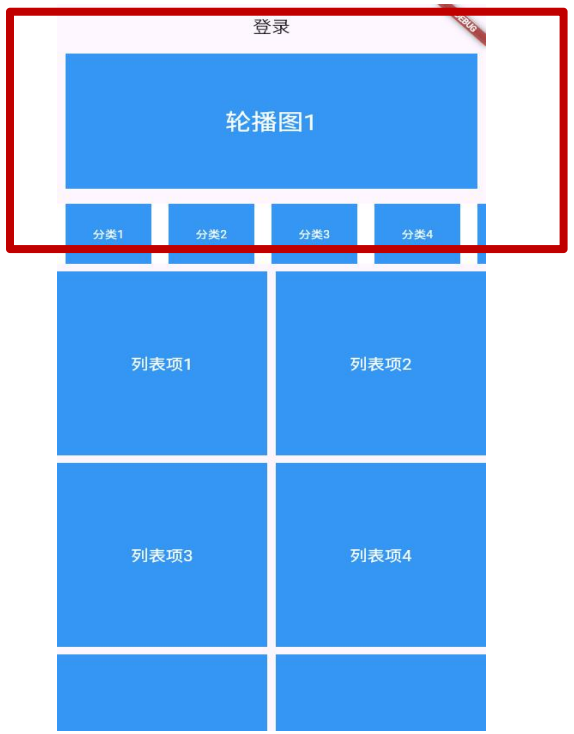
```
// 悬浮分类导航的实现
class _StickyCategoryDelegate extends SliverPersistentHeaderDelegate {
  @override
  double get maxExtent => 80;
  @override
  double get minExtent => 60;

  @override
  Widget build(
      BuildContext context, double shrinkOffset, bool overlapsContent) {
    return Container(
      color: Colors.white,
      child: ListView.builder(
        scrollDirection: Axis.horizontal,
        itemCount: 30,
        itemBuilder: (context, index) => Container(
          margin: EdgeInsets.symmetric(horizontal: 8, vertical: 12),
          width: 100,
          alignment: Alignment.center,
          decoration: BoxDecoration(
            color: Colors.blue,
          ), // BoxDecoration
          child: Text('分类 ${index + 1}', style: TextStyle(color: Colors.white)
        ), // Container
      ), // ListView.builder
    ); // Container
```

案例代码实现–列表实现–SliverList.separated


```dart
SliverList.separated(
    itemCount: 100,
    separatorBuilder: (BuildContext context, int index) {
      return SizedBox(height: 10);
    },
    itemBuilder: (BuildContext context, int index) {
      return Container(
        height: 100,
        color: Colors.blue,
        width: double.infinity,
        alignment: Alignment.center,
        child: Text("列表项${index + 1}",
            style: TextStyle(color: Colors.white, fontSize: 20)), // Text
      ); // Container
    }) // SliverList.separated
```

整页滚动容器–PageView

- 作用：用于实现分页滚动视图的核心组件
- 方式：提供多种构建方式，默认构造方式、PageView.builder等
- 优势：支持懒加载（按需渲染）

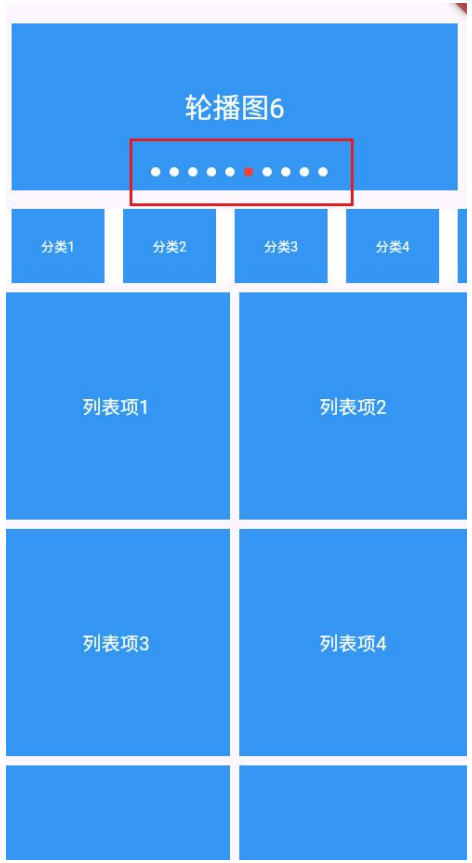



```
// 包裹普通Widget的东西
SliverToBoxAdapter(
  child: Container(
    height: 200,
    child: PageView.builder(
      itemCount: 10,
      itemBuilder: (BuildContext context, int index) {
        return Container(
          margin: EdgeInsets.all(10),
          color: Colors.blue,
          alignment: Alignment.center,
          child: Text(
            "轮播图${index + 1}",
            style: TextStyle(color: Colors.white, fontSize: 30),
          ), // Text
        ); // Container
      },
    ), // PageView.builder
  ) // Container
```

- 场景：PageView经常构建整页滚动切换场景

整页滚动容器–PageView–跳转控制

- 控制器：PageView绑定controller属性，对象类型为PageController
- 切换方法： controller.jumpPage/animateToPage




```dart
class _MainPageState extends State<MainPage> {
  int _currentIndex = 0; // 声明激活索引
  PageController _controller = PageController(); // 定义控制器
  @override
```

```dart
    height: 200,
    child: PageView.builder(
      controller: _controller, // 绑定控制器
      itemCount: 10,
```

```dart
children: List.generate(10, (index) {
  return GestureDetector(
    onTap: () {
      _controller.jumpToPage(index); // 跳转轮播图
      _currentIndex = index; // 赋值激活索引
      setState(() {}); // 状态更新
    },
    child: Container(
      margin: EdgeInsets.only(left: 10),
```

```dart
child: Container(
  margin: EdgeInsets.only(left: 10),
  width: 10,
  height: 10,
  decoration: BoxDecoration(
      // 三元运算符
    color: _currentIndex == index
        ? Colors.red
        : Colors.white,
    borderRadius: BorderRadius.circular(5)), // BoxDecoration
), // Container
```

组件通信

通信方式	方向	适用场景
构造函数传递	父 => 子	简单的数据传递
回调函数	子 => 父	子组件通知父组件
InheritedWidget	祖先 => 后代	跨层级数据共享
Provider	任意组件间	状态管理推荐方案
EventBus	任意组件间	全局事件通信
Bloc/Riverpod	任意组件间	复杂状态管理

组件通信–父传子(构造函数传参数)

●步骤： 1. 子组件定义接收属性

2. 子组件在构造函数中接收参数

3. 父组件传递属性给子组件

4. 有状态组件在‘对外的类’接收属性，‘对内的类’通过widget对象获取对应属性

5. 注意 ： 子组件定义接收属性需要使用final关键字-因为属性由父组件决定，子组件不能随意更改

●需求： 定义父子组件，父组件传递一个message变量给子组件并显示

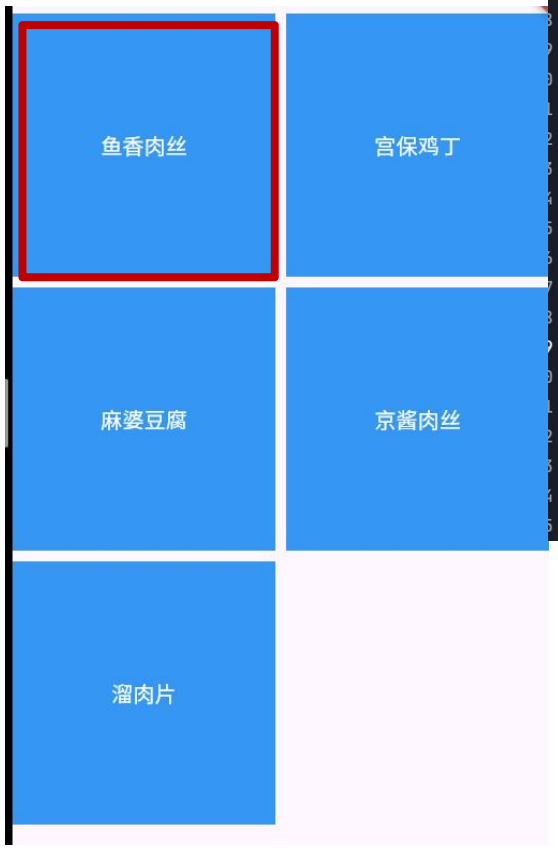

```dart
class Child extends StatelessWidget {
  final String? message;
  const Child({Key? key, this.message}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Column(
```

子组件声明接收属性

```dart
Child(
  message: "张三",
) // Child
```

父组件传递参数

组件通信–父传子案例



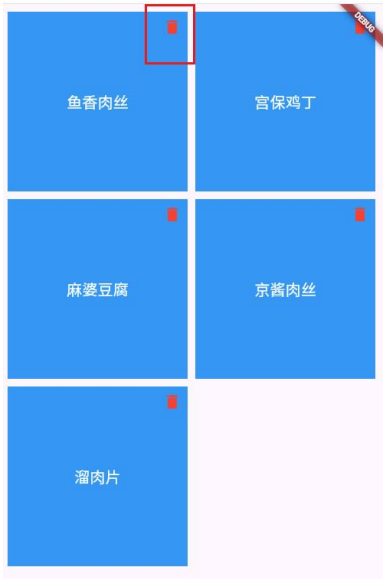

```dart
class _ParentState extends State<Parent> {
  List<String> _list = ["鱼香肉丝", "宫保鸡丁", "麻婆豆腐", "京酱肉丝", "溜肉片"];
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("父传子小案例"),
        ), // AppBar
        body: GridView.count(
          crossAxisCount: 2,
          mainAxisSpacing: 10,
          crossAxisSpacing: 10,
          children: List.generate(_list.length, (index) {
            return Child(foodName: _list[index]);
          }), // List.generate
        ), // GridView.count
      ), // Scaffold
    ); // MaterialApp
  }
}
```

```dart
class Child extends StatefulWidget {
  final String foodName;
  Child({Key? key, required this.foodName}) : super(key: key);

  @override
  _ChildState createState() => _ChildState();
}

class _ChildState extends State<Child> {
  @override
  Widget build(BuildContext context) {
    return Container(
      alignment: Alignment.center,
      color: Colors.blue,
      child: Text(widget.foodName,
          style: TextStyle(color: Colors.white, fontSize: 20)), // Text
    ); // Container
  }
}
```

子组件属性如果没有初始值，需要在构造函数中用required来接收属性

组件通信–子传父(回调函数)

- 步骤： 1. 父组件传递一个函数给子组件
  2. 子组件调用该函数
  3. 父组件通过回调函数获取参数
- 需求：点击子组件删除父组件的菜品数据并更新列表




```dart
// 子组件
class Child extends StatefulWidget {
  final String foodName;
  final int index;
  final Function(int index) delFood;
  Child(
      {Key? key,
      required this.foodName,
      required this.delFood,
      required this.index})
      : super(key: key);

  @override
  _ChildState createState() => _ChildState();
}
```

子组件接收回调函数

子组件调用回调函数

```dart
    IconButton(
        color: Colors.red,
        onPressed: () {
          widget.delFood(widget.index);
        },
        icon: Icon(Icons.delete)) // IconButton
  ],
); // Stack
```

```dart
children: List.generate(_list.length, (int index) {
  return Child(
    foodName: _list[index],
    index: index,
    delFood: (i) {
      _list.removeAt(i);
      setState(() {});
    },
  ); // 返回整个的子组件 // Child
}), // List.generate
```

父组件接收子组件参数删除列表项

网络请求–Dio插件使用

- 网络请求是Flutter移动应用开发的核心功能，最常用的网络请求工具是使用Dio插件
- 安装dio：flutter pub add dio

```
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino
  # Use with the CupertinoIcons clas
  cupertino_icons: ^1.0.8
  dio: ^5.9.0
```

- 基本使用：Dio().get(地址).then().catchError()

```
// runApp(MainPage());
Dio().get('https://geek.itheima.net/v1_0/channels').then((res) {
  print(res);
});
}
```

```
Restarted application in 67ms.
{"data":{"channels":[{"id":0,"name":"推荐"},{"id":1,"name":"html"},{"id":2,"name":"开发者资讯"},{"id":4,"name":"c++"},{"id":
6,"name":"css"},{"id":7,"name":"数据库"},{"id":8,"name":"区块链"},{"id":9,"name":"go"},{"id":10,"name":"产品"},{"id":11,"nam
e":"后端"},{"id":12,"name":"linux"},{"id":13,"name":"人工智能"},{"id":14,"name":"php"},{"id":15,"name":"javascript"},{"id":1
6,"name":"架构"},{"id":17,"name":"前端"},{"id":18,"name":"python"},{"id":19,"name":"java"},{"id":20,"name":"算法"},{"id":21,"na
me":"面试"},{"id":22,"name":"科技动态"},{"id":23,"name":"js"},{"id":24,"name":"设计"},{"id":25,"name":"数码产品"},{"id":26,"nam
e":"软件测试"}]},"message":"OK"}
```

- 一般情况下-在初始化状态initState获取页面数据

网络请求案例

●Dio封装过程

1. 创建工具类

2. 构造函数中设置基础地址和超时时间

3. 添加各类拦截器

4. 封装统一请求方法

5.请求频道数据进行循环渲染解决web端跨域问题

6. 实现UI渲染绘制

网络请求案例–1.Dio工具封装

设置基础地址和超时时间

```
class DioUtils {
  final _dio = Dio(); // 类中的属性定义一个实例对象
  // 设置一些基础的地址和超时时间
  DioUtils() {
    _dio.options
      ..baseUrl = "https://geek.itheima.net/v1_0/"
      ..connectTimeout = Duration(seconds: 10)
      ..sendTimeout = Duration(seconds: 10)
      ..receiveTimeout = Duration(seconds: 10);
    // 拦截器
    _addInterceptor(); // 添加拦截器
  }
```

封装请求方法

```
get(String url, {Map<String, dynamic>? params}) {
  return _dio.get(url, queryParameters: params);
}
```

设置拦截器

```
// 添加拦截器
_addInterceptor() {
  _dio.interceptors.add(InterceptorsWrapper(onRequest: (context, handler) {
    handler.next(context);
  }, onResponse: (context, handler) {
    if (context.statusCode! >= 200 && context.statusCode! < 300) {
      handler.next(context);
      return;
    }
    handler.reject(DioException(requestOptions: context.requestOptions));
  }, onError: (error, hanlder) {
    hanlder.reject(error);
  })); // InterceptorsWrapper
}
```

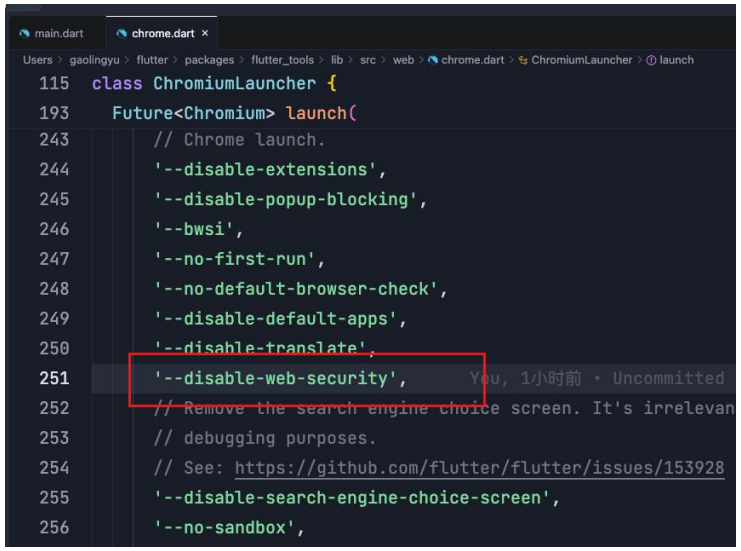
网络请求案例–2.初始化获取数据

```dart
class _MainPageState extends State<MainPage> {
  @override
  void initState() {
    // TODO: implement initState
    super.initState();
    _getChannels();
  }

  List<Map<String, dynamic>> _list = [];
  // 获取频道数据
  void _getChannels() async {
    DioUtils dioUtils = DioUtils();
    Response<dynamic> result = await dioUtils.get("channels");
    Map<String, dynamic> res = result.data as Map<String, dynamic>;
    _list = (res["data"]["channels"] as List).cast<Map<String, dynamic>>();
    setState(() {});
  }
```

网络请求案例–3.解决web端跨域问题

默认情况下, flutter 运行 web 端加载网络资源会报跨域提示错误。

1. 在flutter/packages/flutter_tools/lib/src/web/chrome.dart
如下图位置添加 '--disable-web-security',



2.删除flutter/bin/cache/下 flutter_tools.snaphot和flutter_tools.stamp

3.执行 flutter doctor -v 然后重新运行项目

网络请求案例–4.父子组件通信

```dart
@override
Widget build(BuildContext context) {
  return MaterialApp(
      home: Scaffold(
    appBar: AppBar(title: Text("频道数据")),
    body: GridView.extent(
      padding: EdgeInsets.all(10),
      maxCrossAxisExtent: 140,
      mainAxisSpacing: 10,
      crossAxisSpacing: 10,
      childAspectRatio: 2.5,
      children: List.generate(_list.length, (index) {
        return ChannelItem(
          item: _list[index],
        ); // ChannelItem
      }), // List.generate
    ), // GridView.extent
  )); // Scaffold // MaterialApp
 }
}
```

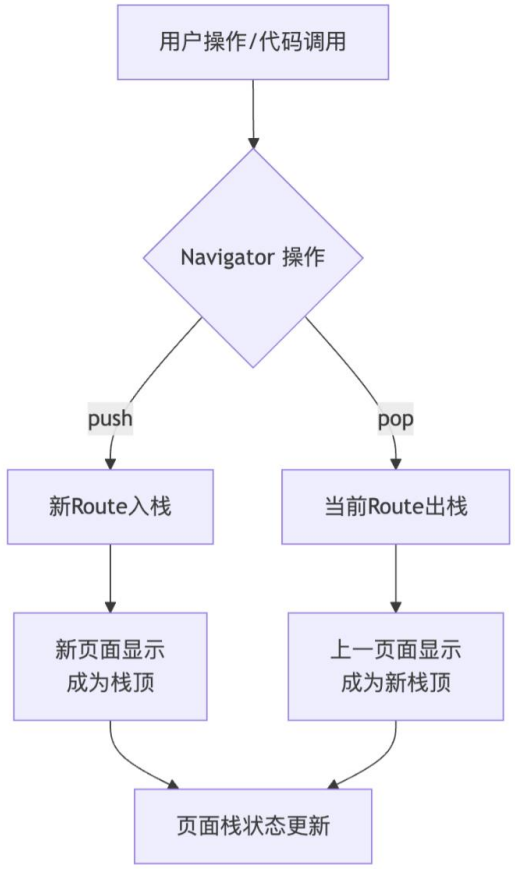
```dart
class ChannelItem extends StatelessWidget {
  final Map<String, dynamic> item;
  const ChannelItem({Key? key, required this.item}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      alignment: Alignment.center,
      padding: EdgeInsets.symmetric(horizontal: 10),
      child: Text(item["name"] ?? "空",
          style: TextStyle(color: Colors.white, fontSize: 14)), // Text
    ); // Container
  }
}
```

网络请求案例–知识点汇总

- Dio一般封装一个工具类来使用
- Dio基础配置有基础地址-超时时间
- Dio的拦截器有请求拦截器、响应拦截器、错误拦截器，通过拦截器使用handler.next() 拦截使用handler.reject
- 想要连续对一个对象赋值可以使用 ..的语法
- List的数据类型想要转化List<具体类型>可以使用cast方法

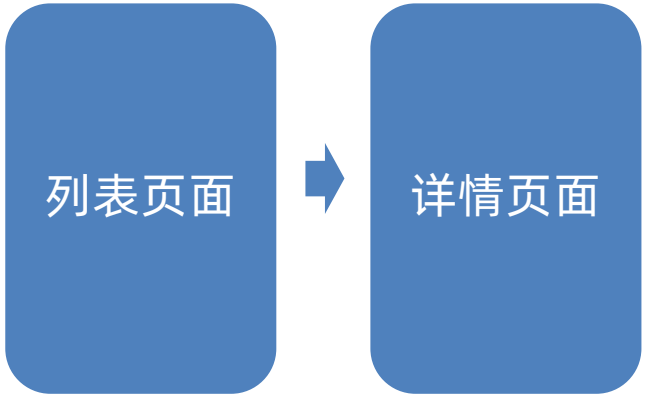
路由管理

- 定义：路由管理是构建多页面应用的核心，它通过 Navigator 和 Route 来管理页面栈，实现页面跳转和返回



路由管理–基本路由

- 场景：基本路由适合页面不多、跳转逻辑简单的场景
- 用法：无需提前注册路由，跳转时创建 MaterialPageRoute实例即可




```
turn Container(
child: Center(
  child: TextButton(
      onPressed: () {
        Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) => DetailPage()), // 创建新Route并入栈 // MaterialPageRoute
        );
      },
      child: Text("去详情")), // TextButton
), // Center
// Container
```

- 跳转新页面： Navigator.push(BuildContext context, Route route)
- 返回上一页： Navigator.pop(BuildContext context)
- 注意：MaterialApp是路由系统的组件，只能有一个MaterialApp包裹

路由管理–命名路由

- 场景：应用页面增多后，使用命名路提升代码可维护性。
- 用法：需要先在 MaterialApp中注册一个路由表（routes）并设置initialRoute(首页)

- 命名路由 vs 简单路由：

  命名路由需在MaterialApp的routes中预先注册路由表，适合中大型项目管理

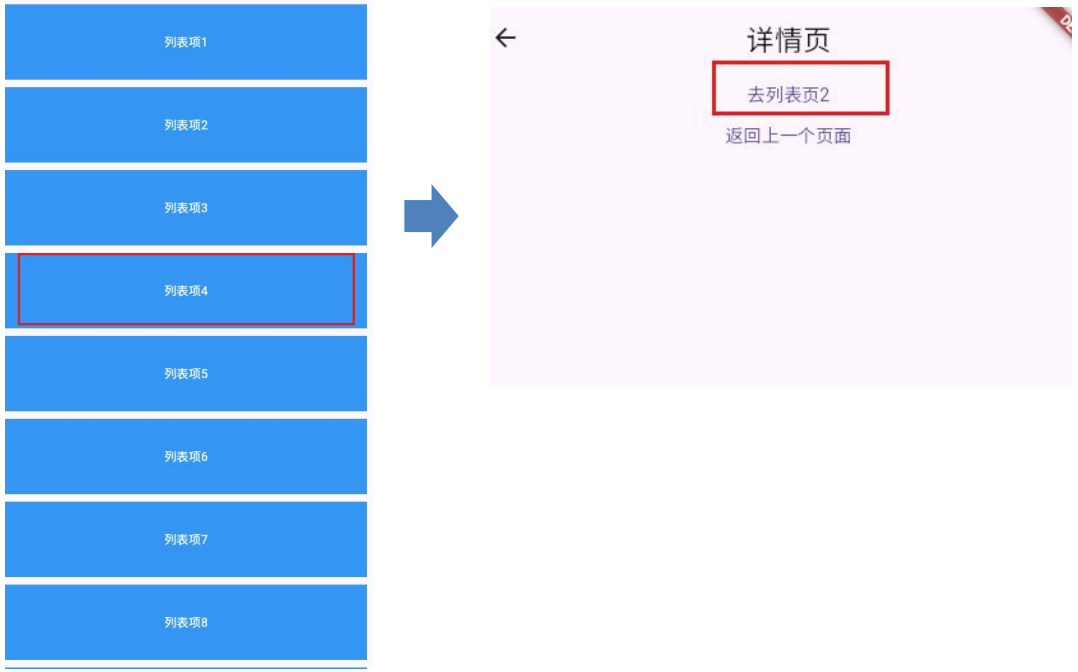
  简单路由直接构建页面，更灵活，适合简单应用或快速原型开发。

路由管理–跳转方法

方法	核心作用	使用场景	典型场景
pushNamed	进入新页面	[A, B]→ [A, B, C]	常规页面跳转，如列表页进入详情页
pushReplacementNamed	替换当前页面	[A, B]→ [A, C]	登录成功后跳转主页，并无法返回登录页
pushNamedAndRemoveUntil	跳转新页面并清理栈	[A, B, C, D]→ [A, E]	退出登录后跳转登录页，并清空所有历史页面
popAndPushNamed	返回并立即跳转新页面	[A, B, C]→ [A, B, D]	购物车页面结算后，返回商品列表并同时跳转到订单页
popUntil	连续返回直到条件满足	[A, B, C, D]→ [A, B]	从设置页的深层级，一键返回到主设置页面

路由管理–传递参数

- 作用：通过路由传递参数是实现页面间数据通信的常用方式
- 传递参数(命名路由)：Navigator.pushNamed(context, 地址，arguments: { 参数 })
- 接收参数(命名路由)：ModalRoute.of(context)?.settings.arguments
- 接收时机：initState获取不到路由参数，放置在Future.microtask(异步微任务)中

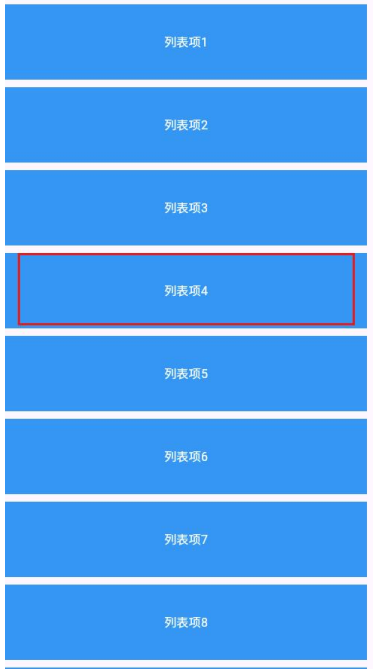



```
//      MaterialPageRoute(builder: (context) => D
Navigator.pushNamed(context, "/detail",
    arguments: {"id": index + 1});
},
```

```
class _DetailPageState extends State<DetailPage> {
  String _id = "";
  @override
  void initState() {
    // TODO: implement initState
    super.initState();
    Future.microtask(() {
      setState(() {
        if (ModalRoute.of(context) != null) {
          _id = (ModalRoute.of(context)!.settings.arguments
                  as Map<String, dynamic>)["id"]
              .toString();
        }
      });
    });
  }
```

路由管理–传递参数–基础路由

- 传递参数(基础路由)：通过组件构造函数传递参数-(父传子)
- 接收参数(基础路由)：通过组件构造函数接收参数--(父传子)
- 接收时机：initState可获取到基础路由的构造函数传参



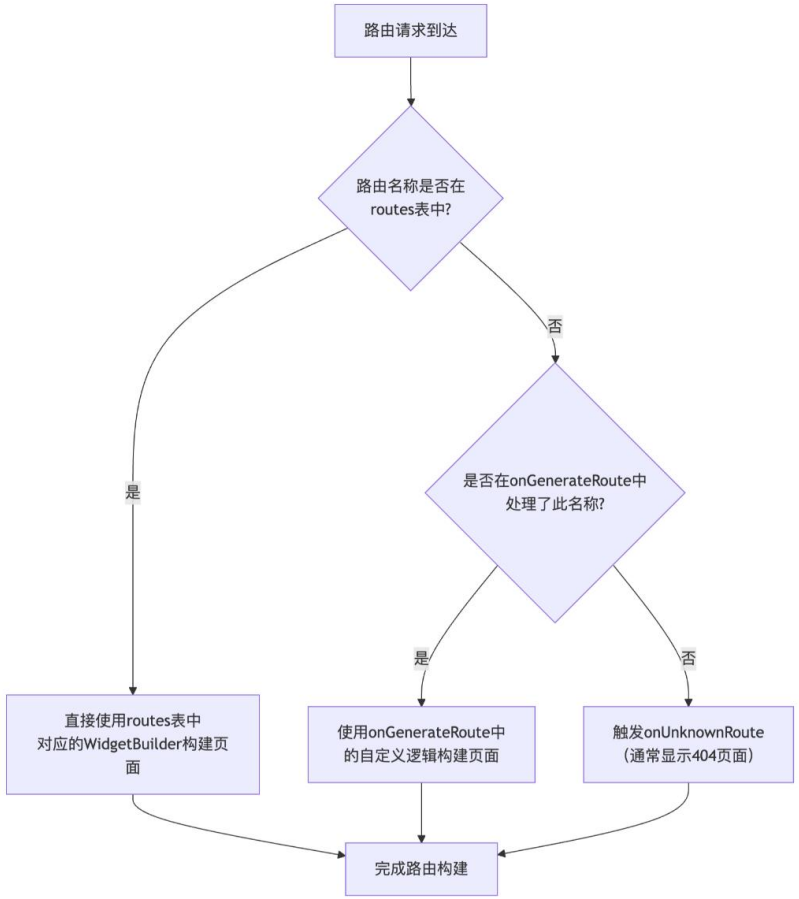

```dart
// 详情页
class DetailPage extends StatefulWidget {
  final String? id;
  DetailPage({Key? key, this.id}) : super(key: key);

  @override
  _DetailPageState createState() => _DetailPageState();
}
```

```dart
// 跳转到详情页
Navigator.push(
    context,
    MaterialPageRoute(
        builder: (context) => DetailPage(
            id: (index + 1).toString(),
        ))); // DetailPage // MaterialPageRoute
},
```

```dart
@override
void initState() {
  // TODO: implement initState
  super.initState();
  print(widget.id);
}
```

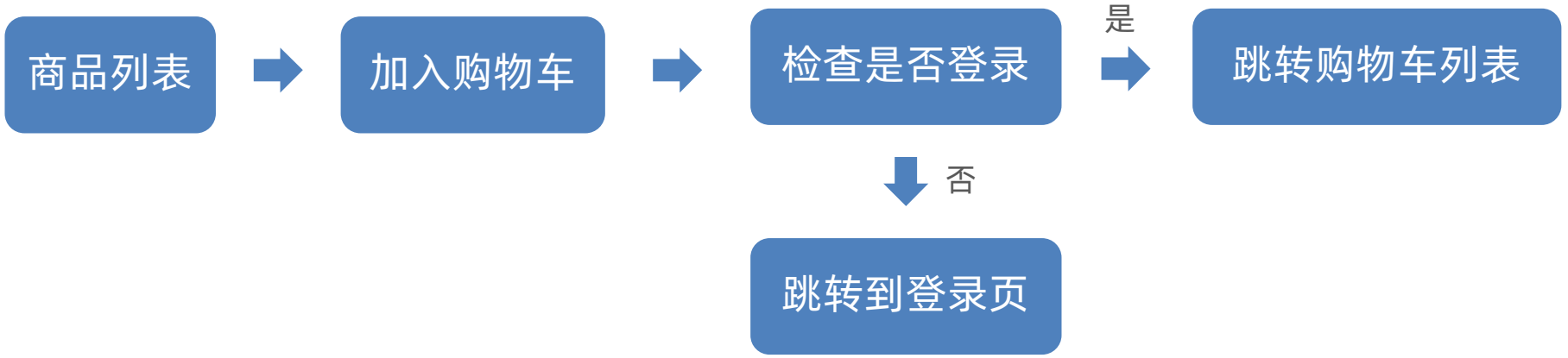
路由管理–动态路由与高级控制

●场景：更复杂的场景，如需根据参数动态生成页面，或实现路由拦截，可以使用 onGenerateRoute和 onUnknownRoute



路由管理–动态路由与高级控制

- onGenerateRoute：允许你根据 RouteSettings（包含路由名称和参数）动态创建不同的 Route



路由管理–动态路由与高级控制

商品列表组件

登录组件

购物车组件

配置onGenerateRoute

```
main() {
runApp(MaterialApp(
  title: "标题",
  initialRoute: "cart",
  routes: {"cart": (context) => CartPage()},
  onGenerateRoute: (setting) {
    bool hasLogin = false;
    if (setting.name == "cartlist") {
      // 如果要去的是购物车列表
      if (hasLogin) {
        return MaterialPageRoute(builder: (context) => CartList());
      } else {
        return MaterialPageRoute(builder: (context) => LoginPage());
      }
    }
  },
)); // MaterialApp
```

加入购物车跳转

```
return Container(
  child: Center(
    child: TextButton(
      onPressed: () {
        Navigator.pushNamed(context, "cartlist");
      },
      child: Text("加入购物车")), // TextButton
  ), // Center
); // Container
}
```

路由管理–动态路由与高级控制

●onUnknownRoute：跳转一个未在路由表中注册、也未在 onGenerateRoute中处理的路由，会调用此回调。通常显示"404"页面

注册onUnknownRoute

```
onUnknownRoute: (setting) {
  return MaterialPageRoute(builder: (context) => Notfound());
},
```

```
return Container(
  child: Center(
    child: Image.asset(
      "lib/images/404.png",
      width: 300,
      height: 300,
    ), // Image.asset
  ), // Center
); // Container
```

定义404组件

跳转一个未注册的路由

```
onPressed: () {
  Navigator.pushNamed(context, "abcde");
},
```