

GetX 教程

作者: Dstudio

联系方式: a3229785914@qq.com

B 站链接: <https://space.bilibili.com/1520286351>

一, 简介以及简单使用

1. 介绍: 是 Flutter 上的一个轻量且强大的解决方案, 提供了很多精美组件、路由管理、高性能的状态管理、依赖注入 (多个组件, 页面之间共享数据) 和主题管理等

2. 导包和演示

创建一个 Flutter 项目后导包, 点击 pubget

```
dependencies:  
  
  flutter:  
    sdk: flutter  
  
  get: ^4.6.6
```

将 main 中的内容改为:

```
import 'package:flutter/material.dart';  
import 'package:get/get.dart';  
void main() {runApp(MaterialApp(home: Home(),));}  
class Home extends StatelessWidget{  
  var times=0.obs;  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(appBar: AppBar(title: Text("这是一个标题")),  
      body: Center(child: Obx(()=>Text(times.value.toString(),style: TextStyle(fontSize: 30),)),),  
      floatingActionButton: FloatingActionButton(onPressed: (){times++;},child: Icon(Icons.add)),  
    ); // Scaffold  
  }  
}
```

显示如下:

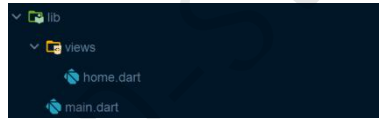


3. 其他

为方便接下来的教学, 我们将 Home 界面单独拿出来, 在 lib 目录下新建一个名为 views 的文件夹, 在 views 文件夹下新建一个名为 home.dart 的文件, 将 Home 部分的内容放入其中,

如下:

目录结构:



main.dart 中:

```
import 'package:flutter/material.dart';
import 'package:g1/views/home.dart';
void main() {runApp(MaterialApp(home: Home(),));}
```

home.dart 中:

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
class Home extends StatelessWidget{
  var times=0.obs;
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("这是一个标题")),
      body: Center(child: Obx(()=>Text(times.value.toString(),style: TextStyle(fontSize: 30))),),
      floatingActionButton: FloatingActionButton(onPressed: (){times++;},child: Icon(Icons.add)),
    ); // Scaffold
  }
}
```

二, 组件

GetX 中提供了很多美观使用的组件, 可以通过函数触发的形式来调出。这里介绍 `SnackBar`、`DefaultDialog` 和 `BottomSheet`。

在使用之前, 需要将 main.dart 中的 `MaterialApp` 改为 `GetMaterialApp` 并导入 `get` 包, 如下:

```
import 'package:flutter/material.dart';
import 'package:g1/views/home.dart';
import 'package:get/get.dart';
void main() {runApp(GetMaterialApp(home: Home(),));}
```

`GetMaterialApp` 是 GetX 在 `MaterialApp` 的基础上进行封装, 在保留原有功能的情况下可以使用 GetX 独有的功能

1. SnackBar

`SnackBar` 可以通过任意函数来调用, 现在对 home.dart 中的 `Scaffold` 部分进行修改, 如下

```
return Scaffold(appBar: AppBar(title: Text("这是一个标题")),
  floatingActionButton: FloatingActionButton(
    onPressed: (){Get.snackbar('提示', '登陆成功');},child: Icon(Icons.add)),
); // Scaffold
```

当点击按钮, 显示如下:



若想更改字体以及背景颜色，和设置停留时间（停留时间也可以传入 `microsecond` 参数，下面演示的例子是传入 `second` 参数，表示停留时间为 2 秒），如下：

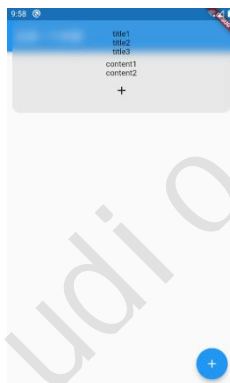
```
floatingActionButton: FloatingActionButton(  
  onPressed: () {Get.snackbar('提示', '登陆成功', backgroundColor: Colors.pinkAccent,  
    colorText: Colors.blue, duration: Duration(seconds: 2));},  
  child: Icon(Icons.add),) // FloatingActionButton
```

效果：



除此之外，我们还可以通过 `titleText` 和 `messageText` 参数传入一个其他控件来替代之前传入的字符串（此时即使传入字符串也将被覆盖），如下：

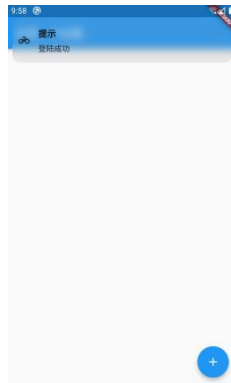
```
onPressed: () {  
  Get.snackbar('提示', '登陆成功',  
    titleText: Column(children: [Text('title1'), Text('title2'), Text('title3')]),  
    messageText: Column(children: [Text('content1'), Text('content2'),  
      IconButton(onPressed: () {}, icon: Icon(Icons.add))]) // Column  
  );  
};
```



（但是 `snackBar` 的作用通常仅仅是起一个通知的作用，简单的设置标题和内容才应该是它的主要任务，繁杂的弹窗工作应该尽量交给 `DefaultDialog` 和 `BottomSheet`）

最后还有一些用的比较多的地方就是给 `snackBar` 设置一个 `icon`、给 `snackBar` 修改一下边框、修改弹出方向（默认从顶部向下弹出），如下：

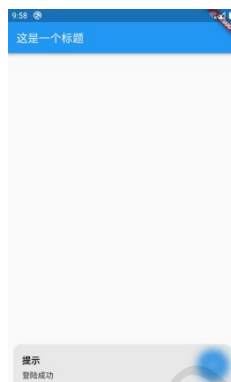
```
onPressed: () {Get.snackbar('提示', '登陆成功', icon: Icon(Icons.motorcycle));},
```



```
onPressed: () {Get.snackbar('提示', '登陆成功', borderColor: Colors.green, borderRadius: 50, borderWidth: 10);},
```



```
onPressed: () {Get.snackbar('提示', '登陆成功', snackPosition: SnackPosition.BOTTOM);},
```



2. defaultDialog

使用 `defaultDialog` 默认会从中间弹出一个消息框，点击周围暗淡区域可退出，简单案例如下：

```
- onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'))};
```



一般上面这种形式的弹窗会搭配一个确认按钮，如下：

```
- onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'),  
  — confirm: ElevatedButton(onPressed: () {}, child: Text('我已知晓'))  
);};
```



这个显示看起来怪怪的，因为一般这种情况下放的是 `TextButton`，并且点击按钮发现无法退出弹窗，此时我们可以在按钮点击事件中调用 `Get.back()` 来退出这个弹窗，现在我们来解决这两个问题，如下：

```
- onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'),  
  — confirm: TextButton(onPressed: () {Get.back();}, child: Text('我已知晓'))  
);};
```



当然，我们也可以给 `cancel` 也传入一个按钮，如下：

```
onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'),
  confirm: TextButton(onPressed: () {Get.back();}, child: Text('我已知晓')),
  cancel: TextButton(onPressed: () {Get.back();}, child: Text('跳过通知'))
);},
```



除了这种直接将按钮传入 `confirm` 和 `cancel` 参数的方式，它还提供了另外一种形式，通过传入 `textCancel` 和 `textConfirm` 文字以及设置 `onCancel` 和 `onConfirm` 来设置点击事件，如下：

```
onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'),
  textCancel: '取消', textConfirm: '确定',
  onCancel: () {print("点击了取消");}, onConfirm: () {print("点击了确定");}
);},
```



此时你会发现一件有意思的事，点击取消按钮，弹窗会自动退出，但是点击取消却貌似没有任何反应，此时需要在 `onConfirm` 的事件中再加上 `Get.back()` 即可

```
onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'),
  textCancel: '取消', textConfirm: '确定',
  onCancel: () {print("点击了取消");}, onConfirm: () {print("点击了确定");Get.back();}
);},
```

这种情况下如果想要改变样式就需要通过传入 `buttonColor` 参数或者 `cancelTextColor` 参数来调整了，如下：

```
onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'),
  textCancel: '取消', cancelTextColor: Colors.red, textConfirm: '确定', confirmTextColor: Colors.red,
  onCancel: () {print("点击了取消");}, onConfirm: () {print("点击了确定");Get.back();}, buttonColor: Colors.brown
);},
```




假如此时我们有这样一个需求时会出现这样一个问题,需求: 点击确认按钮,弹出 `snackBar`,但是实际上却没有弹出,类似代码如下。运行,你会发现,本应该出现 `snackBar` 弹窗以及退出 `defaultDialog` 的行为并没有发生

```
onPressed: () {Get.defaultDialog(title: '群通知',content: Text('XXXXXXXXXXXXXXXXXX'),
  textConfirm: '确定', onConfirm: () {Get.snackbar('标题', '内容');Get.back();}
);},
```

实际上这是因为 `Get` 在这种情况下没办法同时出现两种弹窗,需要先执行退出,再执行弹窗,正确代码如下:

```
onPressed: () {Get.defaultDialog(title: '群通知',content: Text('XXXXXXXXXXXXXXXXXX'),
  textConfirm: '确定', onConfirm: () {Get.back();Get.snackbar('标题', '内容');}
);},
```

除此之外,我们还可以对 `defaultDialog` 中的 `content` 传入自定义的控件,如下:

```
var tec1=TextEditingController();
@override
Widget build(BuildContext context) {
  return Scaffold(appBar: AppBar(title: Text("这是一个标题")),
    floatingActionButton: FloatingActionButton(
      onPressed: () {Get.defaultDialog(title: '请登录',content: Center(child: Column(children: [
        TextField(controller: tec1,decoration: InputDecoration(icon: Icon(Icons.people_alt_outlined))),
        ElevatedButton(onPressed: () {print(tec1.text);Get.back();}, child: Text("登录"))
      ]),), // Column, Center
    );},
    child: Icon(Icons.add), // FloatingActionButton
  ); // Scaffold
}
```



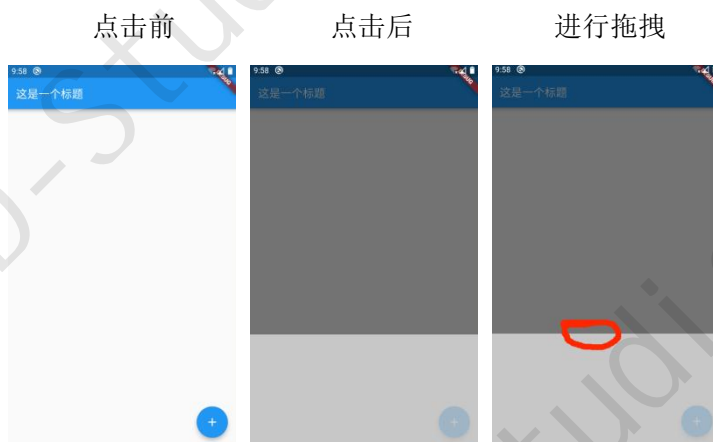
一般来说,当弹出 DefaultDialog 之后,如果点击 DefaultDialog 周围暗掉的部分,那么也会将 dialog 退出,GetX 提供了一个参数使得无效这个动作

```
onPressed: () {Get.defaultDialog(title: '群通知', content: Text('XXXXXXXXXXXXXXXXX'),
  textConfirm: '确定', onConfirm: () {Get.back(); Get.snackbar('标题', '内容');}, barrierDismissible: false
);},
```

3. BottomSheet

调用这个组件将从底部弹出一个窗口,使用它时,需要设定其高度(默认高度为 0)和颜色(默认透明色,虽然看不到,但是确实存在)简单使用如下(一般在 BottomSheet 中放入一个 Contriner 组件):

```
onPressed: () {Get.bottomSheet(Container(height: 200, color: Colors.white60,));},
```



关闭拖拽功能:

```
onPressed: () {Get.bottomSheet(Container(height: 200, color: Colors.white60, enableDrag: false));},
```

点击暗掉的区域,可以退出 bottomsheets, 我们可以通过设置 isDismissible 来关闭这个功能(如果你同时设置了 enableDrag 和 isDismissible 为 false, 同时没有设置 Get.back(), 那么将无法关闭这个底部弹窗):

```
onPressed: () {Get.bottomSheet(Container(height: 200, color: Colors.white60, isDismissible: false));},
```

通过点击按钮退出底部弹窗:

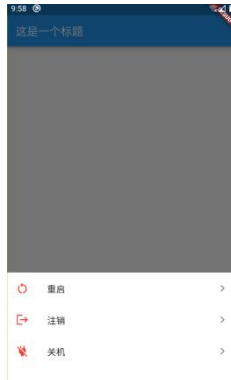
```
onPressed: () {Get.bottomSheet(
  Container(height: 200, color: Colors.white60,
    child: IconButton(onPressed: () {Get.back();}, icon: Icon(Icons.arrow_back),)),);},
```



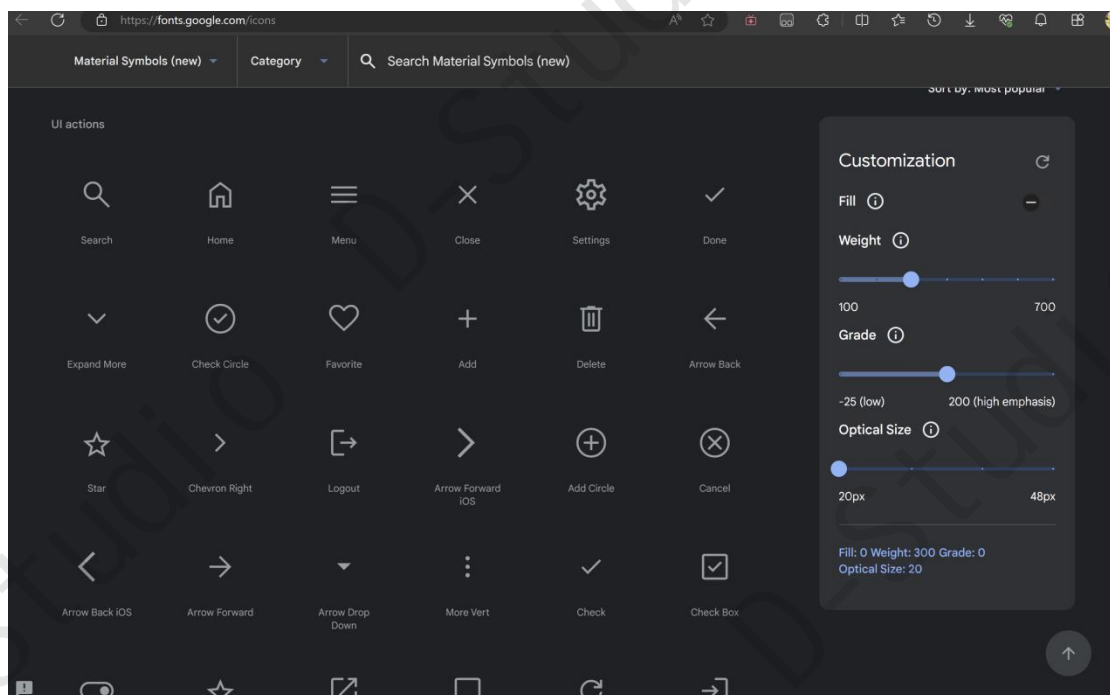
自定义显示 BottomSheet:

```
onPressed: () {Get.bottomSheet(Container(height: 200, color: Colors.white,
  child: const Column(children: [
    ListTile(leading: Icon(Icons.restart_alt, color: Colors.red), title: Text("重启"),
      trailing: Icon(Icons.keyboard_arrow_right), // ListTile
    ListTile(leading: Icon(Icons.logout, color: Colors.red), title: Text("注销"),
      trailing: Icon(Icons.keyboard_arrow_right), // ListTile
    ListTile(leading: Icon(Icons.power_off, color: Colors.red), title: Text("关机"),
      trailing: Icon(Icons.keyboard_arrow_right), // ListTile
  ]));}, // Column, Container
```

显示:



【提示】：如果想要的 icon 找不到，不妨打开如下网页，查看想要的图标（此时应该需要科学上网）



三，主题管理

由关于 GetX 的主题管理非常简单，基本使用如下：

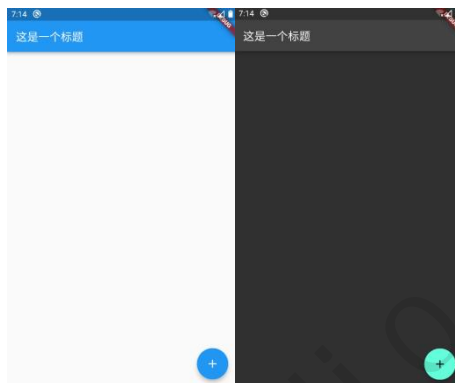
```
onPressed: () {Get.changeTheme(ThemeData.dark());}
```

这里展示一个小案例

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
class Home extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("这是一个标题")),
      floatingActionButton: FloatingActionButton(
        onPressed: (){Get.changeTheme(Get.isDarkMode?ThemeData.light():ThemeData.dark());}, child: Icon(Icons.add),),
    ); // Scaffold
  }
}
```

点击按钮前

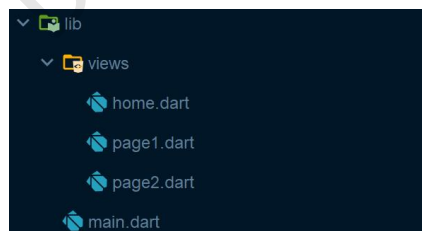
点击后



其原理是通过点击按钮来调用 Get 提供的改变主题的功能，同时使用 Get 来检测当前模式是否是 DarkMode，如果是 ThemeData.light()生效，否则 ThemeData.dark()生效

四，路由

GetX 总共为我们提供了两种跳转方式，一种是普通路由跳转，一种是命名路由跳转。使用 GetX 的路由功能也需要将 main.dart 中的 MaterialApp 变为 GetMaterialApp。同时，因为本章节涉及到路由跳转，因此我们需要在 views 目录下创建几个页面文件，此时 lib 目录如下：



这几个文件内容分别如下：

```
main.dart x home.dart page1.dart page2.dart
1 import 'package:flutter/material.dart';
2 import 'package:g1/views/home.dart';
3 import 'package:get/get.dart';
4 void main() {runApp(GetMaterialApp(theme: ThemeData.dark(),home: Home(),));}
```

在这里设置深色模式是为了在手机模拟器界面在文档中的显示效果，方便查看（否则留白的部分可能会和文档的白色背景重合），读者可以省略掉上图中的第二个红色框框中的内容，

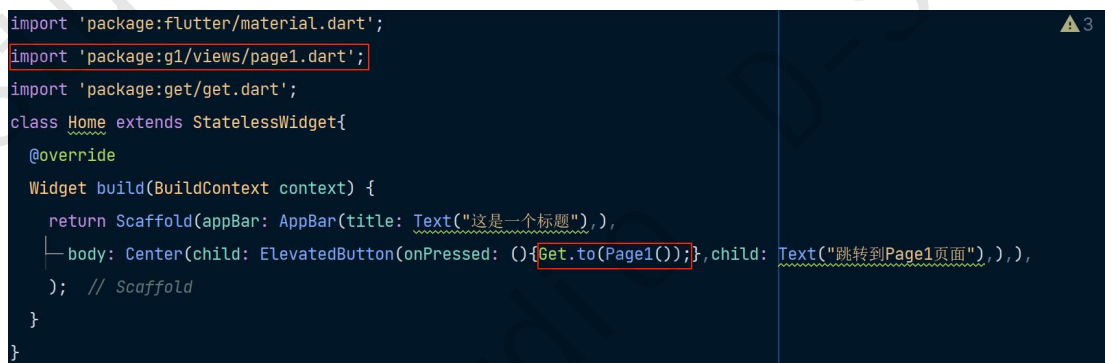


1. 普通路由

对于普通路由的使用，我们一般使用 `Get.to`, `Get.back`, `Get.off`, `Get.offAll`, 使用如下：

Get.to:

需要传入一个跳转的目标，在 `home.dart` 中，通过设置按钮点击事件调用 `Get.to` 来跳转到 `Page1`，如下：

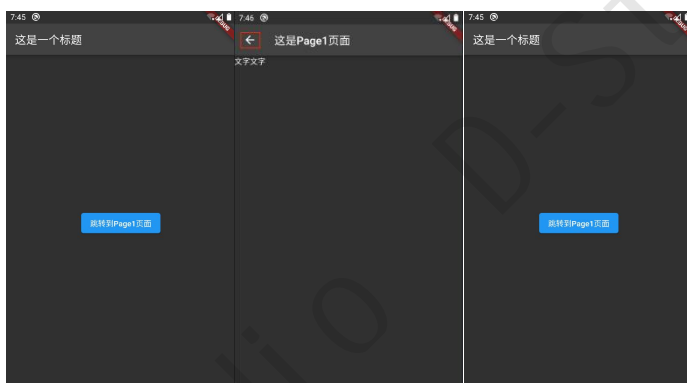


显示如下：

初始界面

点击按钮后

点击返回后



Get.back

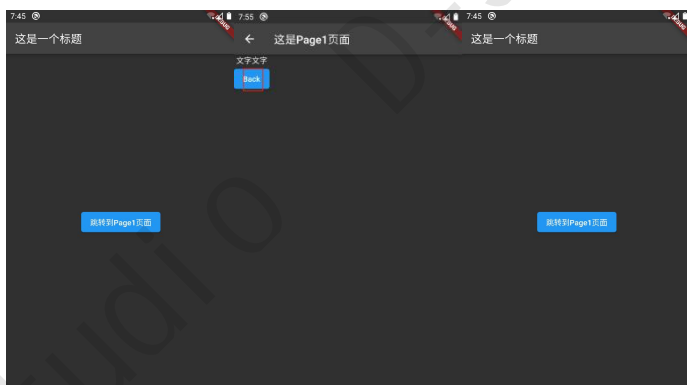
不需要传入参数。实际上，上面过程中的点击返回按钮（红色框框中的按钮）就已经触发了 `Get.back` 函数了，当然我们也可以将其写在函数中，我们在 `page1.dart` 中添加一个按钮，按钮的点击事件中放入 `Get.back`，如下：



初始界面

点击按钮后

点击返回后



Get.off

需要传入下一个页面。有时候，我们跳转到了另一个页面后就不想再回到最初的页面，例如第一次打开 app 出现的情况。使用方法非常简单，可直接将 `home.dart` 中的 `Get.to` 改为 `Get.off`，如下：

```
import 'package:flutter/material.dart';
import 'package:g1/views/page1.dart';
import 'package:get/get.dart';

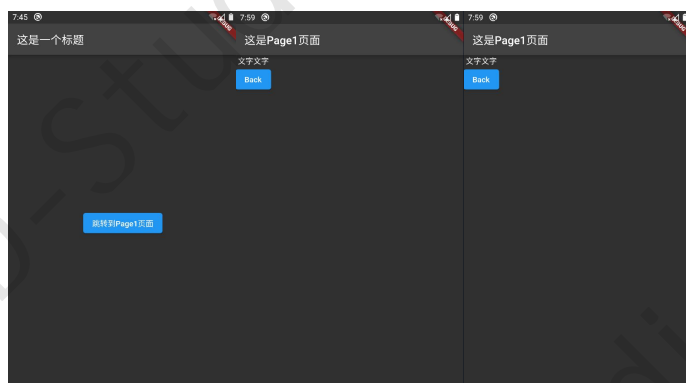
class Home extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("这是一个标题")),
      body: Center(child: ElevatedButton(onPressed: () {Get.off(Page1());}, child: Text("跳转到Page1页面")),),
    ); // Scaffold
  }
}
```

运行后:

初始界面

点击按钮后

点击返回后



此时你会发现两件事，一个是跳转到 Page1 后， appBar 上面没有了返回按钮，另一件事是点击 Page1 的返回按钮，尽管会执行 Get.back 函数，但是也无法跳转回去了

Get.offAll

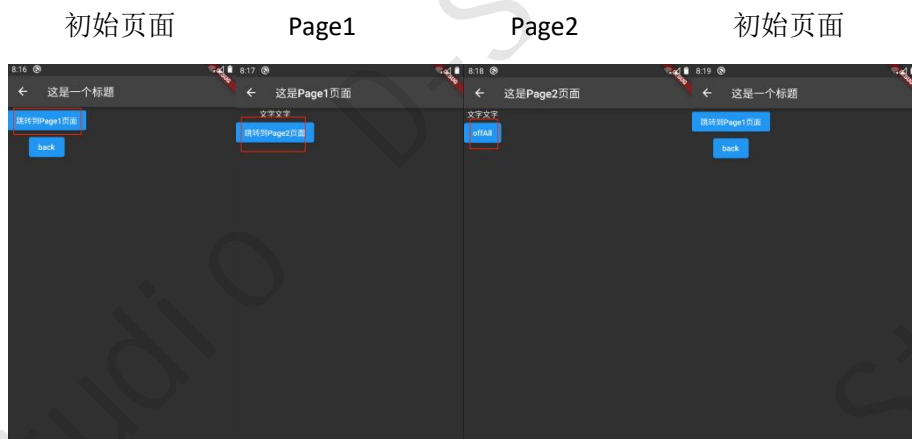
需要传入下一个页面。Get.offAll 将会清除所有路由记录,用于形如跳转多个界面之后，最后回到初始页面。，为了测试这个函数，我们对这三个页面进行修改，结果如下：

```
home.dart x page1.dart page2.dart
1 import 'package:flutter/material.dart';
2 import 'package:g1/views/page1.dart';
3 import 'package:get/get.dart';
4 class Home extends StatelessWidget{
5   @override
6   Widget build(BuildContext context) {
7     return Scaffold(appBar: AppBar(title: Text("这是一个标题")),
8       body: Column(children: [
9         ElevatedButton(onPressed: () {Get.to(Page1());}, child: Text("跳转到Page1页面")),
10        ElevatedButton(onPressed: () {Get.back();}, child: Text("back")),
11      ],) // Column
12    ); // Scaffold
13  }
14 }
```

```
home.dart  page1.dart x  page2.dart
1 import 'package:flutter/material.dart';
2 import 'package:g1/views/page2.dart';
3 import 'package:get/get.dart';
4 class Page1 extends StatelessWidget{
5   @override
6   Widget build(BuildContext context) {
7     return Scaffold(appBar: AppBar(title: Text("这是Page1页面")),
8     body: Column(children: [
9       Text('文字文字'),
10      ElevatedButton(onPressed: () {Get.to(Page2());}, child: Text('跳转到Page2页面')),
11    ],), // Column
12  ); // Scaffold
13  }
14 }
```

```
home.dart  page1.dart  page2.dart x
1 import 'package:flutter/material.dart';
2 import 'package:g1/views/home.dart';
3 import 'package:get/get.dart';
4 class Page2 extends StatelessWidget{
5   @override
6   Widget build(BuildContext context) {
7     return Scaffold(appBar: AppBar(title: Text("这是Page2页面")),
8     body: Column(children: [
9       Text('文字文字'), ElevatedButton(onPressed: () {Get.offAll(Home());}, child: Text('offAll')),
10    ],), // Column
11  ); // Scaffold
12  }
13 }
```

运行结果如下：



乍一看，貌似 Page2 中的 Get.offAll 函数只是返回到了初始页面 Home，使用 Get.to 也能够达到和上面一样的展示结果，确实如此，但是两者的区别是，使用后者在最后回到初始页面后，点击 back 能够回到 Page2 页面；前者的话点击 back 无效。也就是说，Get.to 是一种会留下路由跳转记录的跳转方式，而 Get.offAll 会删除所有的路由跳转记录。

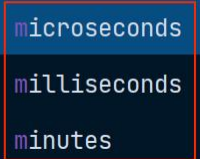
2. 转场效果

从一个页面跳转到第二个页面涉及到一些跳转动画，转场效果的需要通过 transition 参数传入，有些非常快速，可以使用 duration 来传入时间，基本使用：

```
Page1(),transition:Transition.fadeIn,duration: Duration(seconds: 2));},
```


当然也可以传入毫秒或者其他单位

```
Get.to(Page1(), transition: Transition.fadeIn, duration: Duration(m: 2));}, child:
Get.back();}, child: Text("back"),),
```



除了用 transition 这个参数来定义动画，使用 curve 参数亦可，如下：

```
Get.to(Page1(), curve: Curves.easeOutQuint, duration: Duration(seconds: 2));
```

除此之外，GetX 也为我们提供了全局定义路由跳转的方式，需要在 main.dart 中的 GetMaterialApp 中进行设置，

```
import 'package:flutter/material.dart';
import 'package:g1/views/home.dart';
import 'package:get/get.dart';
void main() {runApp(GetMaterialApp(defaultTransition: Transition.upToDown,
  theme: ThemeData.dark(), home: Home(),));} // GetMaterialApp
```

但是在 GetMaterialApp 中无法设置动画持续时间，可以放在 Get.to 中定义持续时间（并不是全局的）

3. 普通路由传值

普通路由传值的方式只有一种，使用 arguments 参数进行传入，如下：

Home 中传入：

```
import 'package:flutter/material.dart';
import 'package:g1/views/page1.dart';
import 'package:get/get.dart';
class Home extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("这是一个标题"),),
      body: Column(children: [
        ElevatedButton(onPressed: () {Get.to(Page1(), arguments: {'name': '小明', 'age': 18});}, child: Text("跳转到Page1页面")),
        ElevatedButton(onPressed: () {Get.back();}, child: Text("back")),
      ],) // Column
    ); // Scaffold
  }
}
```

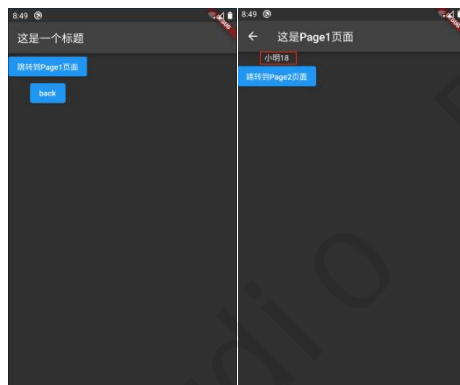
Page1 接受：

```
import 'package:flutter/material.dart';
import 'package:g1/views/page2.dart';
import 'package:get/get.dart';
class Page1 extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("这是Page1页面")),
      body: Column(children: [
        Text(Get.arguments['name']+Get.arguments['age'].toString()),
        ElevatedButton(onPressed: () {Get.to(Page2());}, child: Text("跳转到Page2页面")),
      ],) // Column
    ); // Scaffold
  }
}
```

运行后显示:

初始界面

跳转后



4. 命名路由

通过提前给每一个页面进行命名后使用的路由模式
简单使用:

我们对 `mian.dart` 进行修改, 改后如下:

```
import 'package:flutter/material.dart';
import 'package:g1/views/home.dart';
import 'package:g1/views/page1.dart';
import 'package:g1/views/page2.dart';
import 'package:get/get.dart';

void main() {runApp(GetMaterialApp(theme: ThemeData.dark(),
  getPages: [
    GetPage(name: '/', page: ()⇒Home()),
    GetPage(name: '/p1', page: ()⇒Page1()),
    GetPage(name: '/p2', page: ()⇒Page2()),
  ],
));} // GetMaterialApp
```

这里 '/' 表示根目录所在的路由名称, 如果没有指定首页, 那么它就是首页, 现在我们来指定:

```
import 'package:flutter/material.dart';
import 'package:g1/views/home.dart';
import 'package:g1/views/page1.dart';
import 'package:g1/views/page2.dart';
import 'package:get/get.dart';

void main() {runApp(GetMaterialApp(theme: ThemeData.dark(),
  initialRoute: '/p1',
  getPages: [
    GetPage(name: '/', page: ()⇒Home()),
    GetPage(name: '/p1', page: ()⇒Page1()),
    GetPage(name: '/p2', page: ()⇒Page2()),
  ],
));} // GetMaterialApp
```

此时使用 `Get.to` 仍旧可行, 但是此时你多了一种选择, 例如, 你可以将如下的

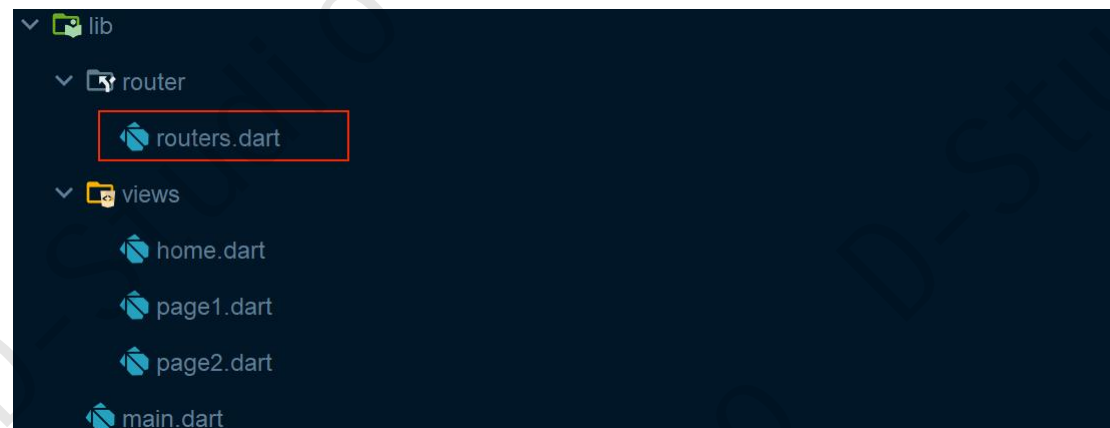
```
Get.to(Page1());
```

修改成:

```
Get.toNamed('/p1');
```

同理, `Get.off(Page2())` 可以改为 `Get.offNamed('/p2')` 以及 `Get.offAll(Page2())` 可以改为 `Get.offAllNamed('/p2')`,

通常情况下我们会将路由配置文件单独放在一个名为 `routers.dart` 中, 现在我们在 `lib` 下新建一个名为 `router` 的文件夹, 并在 `router` 文件夹中添加一个名为 `routers.dart` 的文件, 此时目录如下:



在该文件中写入如下:

```
import 'package:g1/views/page1.dart';
import 'package:g1/views/page2.dart';
import 'package:get/get.dart';
import '../views/home.dart';

class Router1{
  static var routes=[
    GetPage(name: '/', page: ()⇒Home()),
    GetPage(name: '/p1', page: ()⇒Page1()),
    GetPage(name: '/p2', page: ()⇒Page2()),
  ];
}
```

接着在 `mian.dart` 中进行引入:

```
import 'package:flutter/material.dart';
import 'package:g1/router/routers.dart';
import 'package:get/get.dart';

void main() {runApp(GetMaterialApp(theme: ThemeData.dark(),
  getPages: Router1.routes,
));} // GetMaterialApp
```

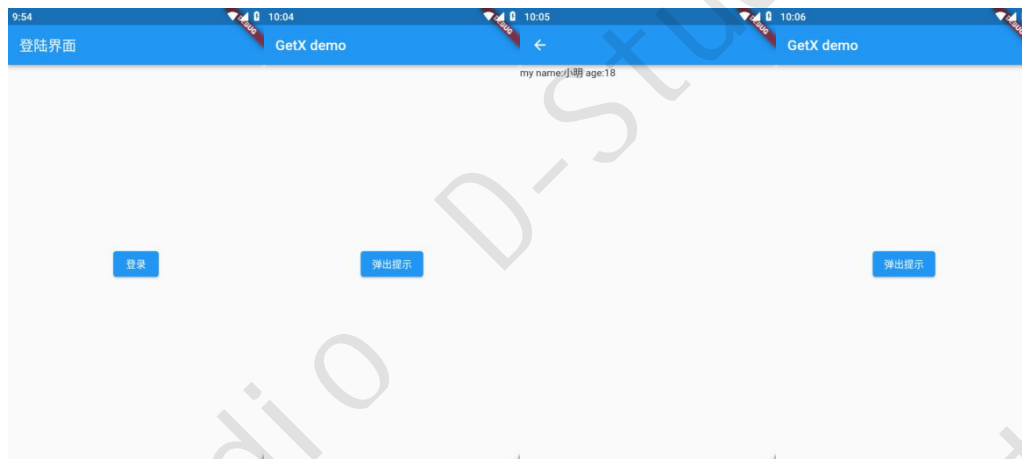
效果:

初始界面

点击按钮后

点击按钮后

点击返回后



5. 命名路由跳转动画和传值

命名路由跳转动画的使用和传值都可以参考普通路由的那一套,但是命名路由的在跳转动画和传值方法更对

首先是跳转动画:

在命名路由的用法中,可以在定义路由处选择跳转动画:

```
class Router1{
  static var routes=[
    GetPage(name: '/', page: ()⇒Home()),
    GetPage(name: '/p1', page: ()⇒Page1(),transition: Transition.fadeIn),
    GetPage(name: '/p2', page: ()⇒Page2()),
  ];
}
```

这样就能实现无论哪个页面跳转到 p1 页面都能实现 Transition.zoom 的跳转动画效果。

关于命名路由的传值,还有一种类似于前端使用的 url 链接的形式,对比两种形式如下:

```
Get.toNamed('/p1',arguments: {'name':'小明','age':18});
```

```
Text(Get.arguments['name']+"-----"+Get.arguments['age']),
```

```
Get.toNamed("/p1?name=小明&age=18");
```

```
Text(Get.parameters['name']!+"----"+Get.parameters['age']!),
```

6. 中间件

当发生路由跳转时调用,可用于检查是否满足能够跳转的条件(如是否登录)。类似的,路由中间件也经常被抽离出来单独作为一个文件,于是我们在 lib 目录下新建一个名为 middleware 的文件夹,文件夹中新建一个名为 login_check.dart,写入如下:

```
class LoginCheck extends GetMiddleware{
  @override
  RouteSettings? redirect(String? route) {
    print('这是路由中间件打印的文本。。。。。');
    return null;
  }
}
```

这里 `return null`，表示不做任何处理，放行；还可以设置为 `return RouteSettings(name: '/p2')`，此时带代表会跳转到 **Page2** 页面。

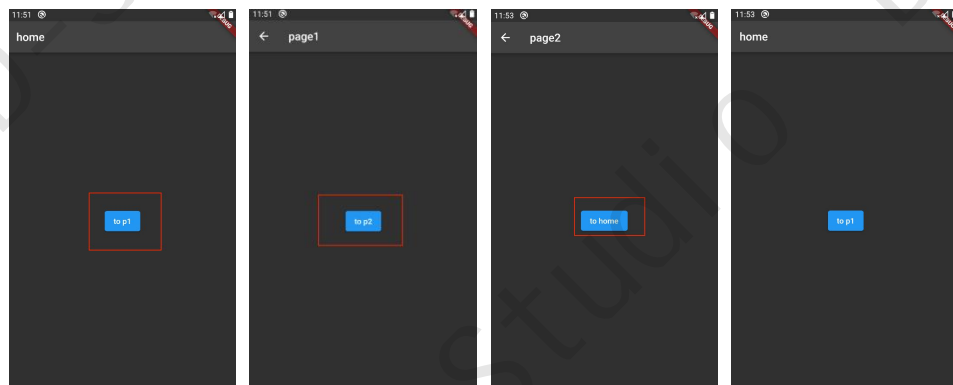
接着我们让 **Page1** 使用这个中间件（需要在路由中进行配置）

```
class Router1{
  static var routes=[
    GetPage(name: '/', page: ()⇒Home()),
    GetPage(name: '/p1', page: ()⇒Page1(), middleware: [LoginCheck(),]),
    GetPage(name: '/p2', page: ()⇒Page2()),
  ];
}
```

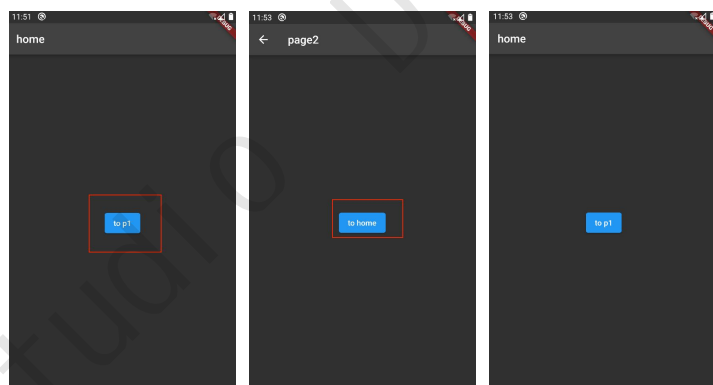
这里我们自定义的中间件被放置在一个列表中，不难想到，既然是列表，那么可以传入多个中间件，我们这边只做这一个作为案例。现在，这个中间件就被 **Page1** 界面引用了，当有任何页面想要跳转到 **Page1**，都必须先经过这个中间件。

在这里我们仅介绍 `redirect`，因为其他的函数很少用

运行结果：



但是当我们把 `return null` 换成 `return RouteSettings(name: '/p2')`，运行结果如下：



五，状态管理

状态管理指的就是让数据被多组件可共享的行为。

`getx` 为我们提供两种状态管理器，一个是响应式状态管理（被动方式，如 `Obx` 和 `GetX`），一个是简单状态管理（主动方式 `GetBuilder`）。

1. 响应式状态管理器 `Obx`

本课程的第一节的案例使用的就是 `Obx`

在使用这种状态管理器之前我们需要去定义一些数据（`int`，`String`，`List`，类等等）并使其成

为一个响应式数据，案例 1 如下：

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

void main() {runApp(MaterialApp(home: Home(),));}

class Home extends StatelessWidget{
  var times=0.obs;
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("这是一个标题")),
      body: Center(child: Obx(()=>Text(times.value.toString(),style: TextStyle(fontSize: 30),)),),
      floatingActionButton: FloatingActionButton(onPressed: (){times++;},child: Icon(Icons.add),),
    ); // Scaffold
  }
}
```

在这个案例中，我们通过将 0 后面加上.obs 使其成为一个响应式变量，实际上将普通变量变为响应式变量的方式还有两种，举例来说，以下三种写法是等价的：

```
var time1=0.obs;
var time2=Rx<int>(0);
var time3=RxInt(0);
```

当然，类似的还有 Rx<String>, RxDouble，但是实际开发中第一种做法是最常见的。

并且实际上，当我们使用状态管理的时候，通常不会将变量直接写在一个页面中，而是另起一个文件用来管理多个页面的数据，做法是在 lib 下新建一个名为 controller 的文件夹，在该文件夹中再新建一个名为 text_controller.dart 的文件，写入如下：

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

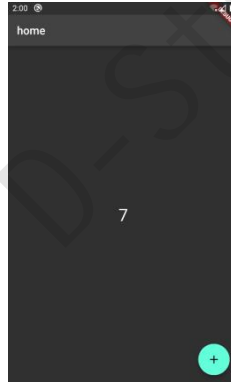
class TextController extends GetxController{
  var times=0.obs;
  void increase(){
    times.value++;
  }
}
```

这里要注意，当我们需要拿到响应式变量中的值的时候，通常需要使用.value 将其中的值取出，home.dart 中：

```
import 'package:flutter/material.dart';
import 'package:g1/controller/test_controller.dart';
import 'package:get/get.dart';

class Home extends StatelessWidget{
  var tc1=Get.put(TextController());
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("home")),
      body: Center(child: Obx(()=>Text(tc1.times.value.toString(),style: TextStyle(fontSize: 30),)),),
      floatingActionButton: FloatingActionButton(onPressed: (){tc1.increase();},child: Icon(Icons.add),),
    ); // Scaffold
  }
}
```

需要注意的是，只有 Obx 包含的部分会刷新数据，所以实际上，用 Obx 包着整个 Scaffold 也是可以的，但是这将使得更新数据的量很大，我们只需要更新一个 Text 控件，那么为了避免没必要的性能浪费就只嵌套 Text 就可以了，显示：



假如我们希望每次不是加 1 而是加其他的数字，我们可以通过向 `test_controller.dart` 的 `increase` 函数进行传参，`test_controller.dart`:

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

class TextController extends GetxController{
  var times=0.obs;
  void increase(int i){
    times.value+=i;
  }
}
```

Home.dart 中:

```
class Home extends StatelessWidget{
  var tc1=Get.put(TextController());
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("home")),
      body: Center(child: Obx(()=>Text(tc1.times.value.toString(),style: TextStyle(fontSize: 30),)),),
      floatingActionButton: FloatingActionButton(onPressed: (){tc1.increase(5);}, child: Icon(Icons.add),),
    ); // Scaffold
  }
}
```

自定义的类使用的响应式变量使用

首先在 `lib` 下新建一个名为 `entity` 的文件夹，在里面放咱们的实体类文件，在该文件夹下创建一个名为 `user.dart` 的文件并写入如下:

```
class User{
  String name;
  String age;
  User( this.name, this.age);
}
```

在 `test_controller.dart` 中:

```
class TextController extends GetxController{
  var u1=User('小明', '18').obs;
  void change(){
    u1.update((u1) {u1?.age='20'; u1?.name='小刚';});
  }
}
```

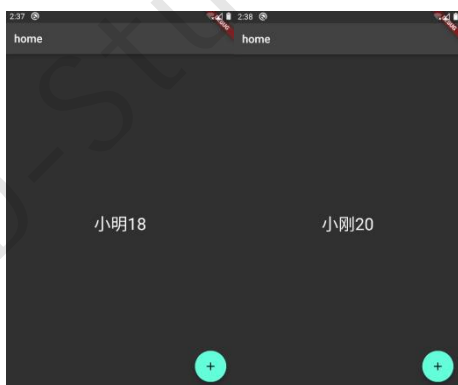
home.dart 中:

```
body: Center(child: Obx(()⇒Text(tc1.u1.value.name+tc1.u1.value.age, style: TextStyle(fontSize: 30),)),),
floatingActionButton: FloatingActionButton(onPressed: (){tc1.change();}, child: Icon(Icons.add),),
```

显示:

点击前

点击后



实际上, 关于自定义类使用状态管理还有一种方法, 那就是在定义类的成员变量的时候就将其设定为响应式变量, 如下:

user.dart 中:

```
class User{
  var name=''.obs;
  var age=''.obs;
  User( this.name, this.age);
}
```

test_controller.dart 中(此时并不是直接对自定义类进行更新, 因此不需要使用 update 函数):

```
class TextController extends GetxController{
  var u1=User('小明'.obs, '18'.obs);
  void change(){
    u1.age.value='20';
    u1.name.value='小刚';
  }
}
```

home.dart 中:

```
body: Center(child: Obx(()⇒Text(tc1.u1.name.value+tc1.u1.age.value, style: TextStyle(fontSize: 30),)),),
floatingActionButton: FloatingActionButton(onPressed: (){tc1.change();}, child: Icon(Icons.add),),
```

显示效果和之前的一致

列表使用的响应式变量使用

列表的响应式变量使用可能比你想的还要简单，如下：

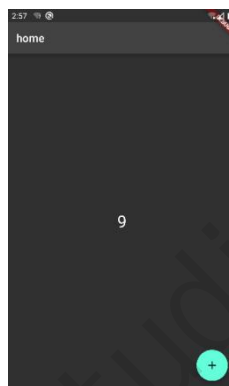
test_controller.dart 中：

```
class TextController extends GetxController{
  var l1=[0,1,2].obs;
  void add(){l1.add(3);}
}
```

home.dart 中：

```
body: Center(child: Obx(()⇒Text(tc1.l1.length.toString(), style: TextStyle(fontSize: 30),)),),
floatingActionButton: FloatingActionButton(onPressed: (){tc1.add();}, child: Icon(Icons.add),),
```

显示：



2. 响应式状态管理器 GetX

使用方法基本上和 Obx 一样，明显不同的地方就在于，Obx 使用 controller 只需要在类中获取一次，但是 GetX 的形式就需要多次取出，都是只需要将组件进行包裹即可，我们来对比一下：

Obx 包裹：

```
Obx(()⇒Text(tc1.l1.length.toString(), style: TextStyle(fontSize: 30),))
```

GetX 包裹：

```
GetX<TextController>(builder: (TextController)⇒Text(tc1.l1.length.toString(), style: TextStyle(fontSize: 30),))
```

这两种方法达到的效果一致

你可能会看到另外一种写法：

Obx 包裹：

```
Center(child: Obx(){return Text(tc1.l1.length.toString(), style: TextStyle(fontSize: 30),);})),
```

GetX 包裹：

```
body: Center(child: GetX<TextController>(
  — builder: (TextController){return Text(tc1.l1.length.toString(), style: TextStyle(fontSize: 30),);}
),), // GetX, Center
```

这两种写法区别在于有无箭头函数和 return，实际上箭头函数只是一种更加便捷的写法，两者并没有区别

3. 简单的状态管理

简单的控制器虽然使用上没有响应式控制器好用（其实也不算难用），但是从原理上简单的控制器是通过通知的形式去让监听数据的区域进行更新，而响应式控制器的实现是基于流（每时每刻都在监听控件，一旦属性改变，就刷新）的，因此性能并没简单的状态管理好；基本使用如下：

test_controller.dart 中：

```
class TextController extends GetxController{
  var times=0;
  void increase(){times++;update();}
}
```

home.dart 中：

```
class Home extends StatelessWidget{
  var tc1=Get.put(TextController());
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("home")),
      body: Center(child:
        GetBuilder<TextController>(builder: (_)⇒Text(_!.times.toString()),), // Center
        FloatingActionButton(onPressed: (){tc1.increase();}, child: Icon(Icons.add)),
      ); // Scaffold
  }
}
```

六，依赖管理

依赖管理实际上指的就是 controller 的管理，通过对依赖的管理可以实现多个页面之间共享一套数据

test_controller.dart 中：

```
class TextController extends GetxController{
  var times=0.obs;
  void increase(){times.value++;}
}
```

home.dart 中：

```
class Home extends StatelessWidget{
  var tc1=Get.put(TextController());
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("home")),
      body: Center(child: Column(children: [
        Obx(()⇒Text(tc1.times.value.toString(), style: TextStyle(fontSize: 40))),
        ElevatedButton(onPressed: (){Get.toNamed('/p1')}, child: Text('go p1')),
      ]),), // Column, Center
      floatingActionButton: FloatingActionButton(onPressed: (){tc1.increase();}, child: Icon(Icons.add)),
    ); // Scaffold
  }
}
```

page1.dart 中:

```
class Page1 extends StatelessWidget{
  var tc1=Get.find<TextController>();
  @override
  Widget build(BuildContext context) {
    return Scaffold(appBar: AppBar(title: Text("page1")),
      body: Center(child: Obx(()⇒Text(tc1.times.value.toString(),style: TextStyle(fontSize: 40),)),)
    ); // Scaffold
  }
}
```

运行效果:



从上面的实例中我们可以看出,使用 `getx` 中的依赖管理,首先需要去在一个页面中通过 `put` 来创建一个 `controller`,之后其他页面才能够被使用。这就需要你规定好一个页面去专门的创建 `put`,这时候,应用打开的第一个页面通常会作为创建 `controller` 的最佳地点,但是实际有很多情况会要求第一个页面进行变更,比如登录等场景。为了应对这种情况,`getx` 为我们提供了一种解决途径,Getx Binding。使用方法如下:

在 `lib` 文件夹下新建一个 `binding` 文件夹,在该文件夹下新建一个名为 `binding.dart` 的文件,写入如下:

```
class MyBinding implements Bindings{
  @override
  void dependencies() {Get.lazyPut(() ⇒ TextController());}
}
```

接着就可以在 `GetMaterialApp` 中使用它了:

```
void main() {runApp(GetMaterialApp(theme: ThemeData.dark(),
  initialBinding: MyBinding(),
  initialRoute: '/',
  getPages: Router1.routes
));} // GetMaterialApp
```

之后各个页面如果要使用到状态管理就可以都使用 `find` 了

七, Get Cli

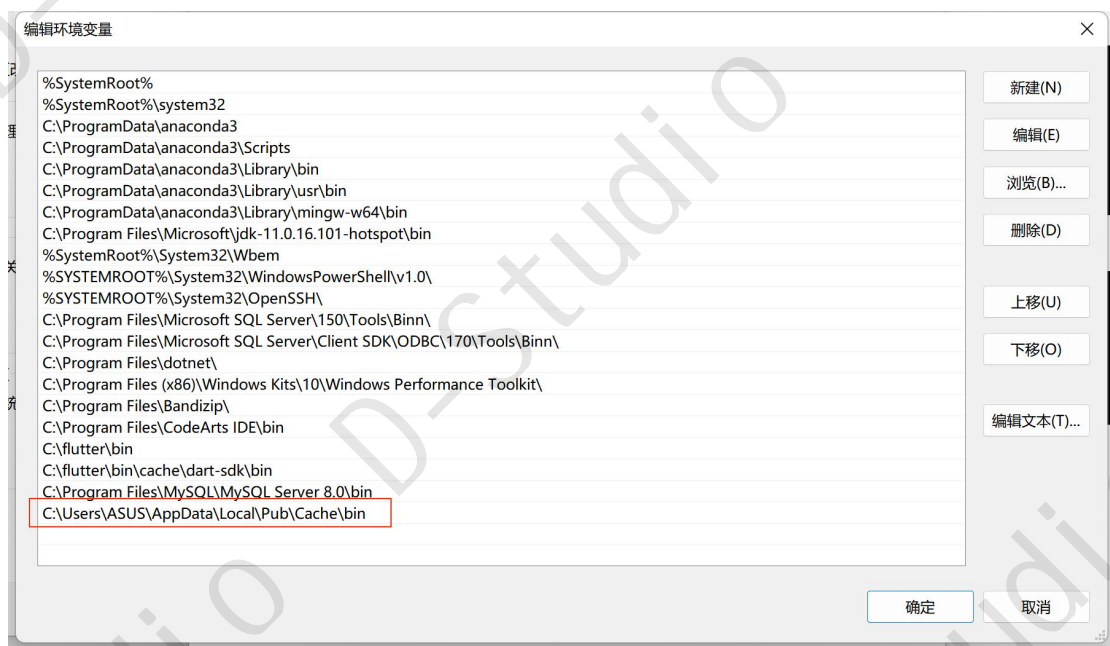
直至目前我们已经学了关于 GetX 的非常多的知识点, 包括路由, 中间件, 依赖管理, 状态管理等, 如果我们在咱们的项目中使用到这些 GetX 体系的应用, 你那么你需要去进行导入新建各种文件夹和文件, 这将是一个非常麻烦的过程, GetX 就为提供了能够快速生成一套包含基本 GetX 应用的工具, GetCli, 使用如下:

首先需要进行工具的下载, 在终端中输入: flutter pub global activate get_cli

```
C:\Users\ASUS\Desktop>flutter pub global activate get_cli
Flutter assets will be downloaded from https://storage.flutter-io.cn. Make sure you trust this source!
+ _fe_analyzer_shared 61.0.0 (64.0.0 available)
+ analyzer 5.13.0 (6.2.0 available)
+ ansicolor 2.0.1
+ archive 3.3.8
+ args 2.4.2
+ async 2.11.0
```

```
.....
+ yaml 3.11.2
Building package executables... (16.1s)
Built get_cli:get.
Installed executables get and getx.
Warning: Pub installs executables into C:\Users\ASUS\AppData\Local\Pub\Cache\bin, which is not on your path.
You can fix that by adding that directory to your system's "Path" environment variable.
A web search for "configure windows path" will show you how.
Activated get_cli 1.8.4.
```

按照提示设置环境变量:








设置完成环境变量就可以开始创建 GetX 项目了, 我们需要在桌面创建一个名为 getXtest 的文件夹, 在这个文件夹中打开终端, 输入 get create project:g1,g1 就是我们的项目名称了, 但是你可能会碰到找不到 git 的情况, 如下:

```
C:\Users\ASUS\Desktop\getXtest>get create project:g1
Error: Unable to find git in your PATH.
```

这里提供一个下载 git 的地址:

<https://registry.npmmirror.com/binary.html?path=git-for-windows/v2.42.0.windows.1/>

Index of /git-for-windows/v2.42.0.windows.1/

	Name	Last modified	Size
	Parent Directory		-
	Git-2.42.0-32-bit.exe	2023-08-21T20:05:50Z	59.05MB
	Git-2.42.0-32-bit.tar.bz2	2023-08-21T20:06:00Z	105.08MB
	Git-2.42.0-64-bit.exe	2023-08-21T20:05:37Z	58.35MB
	Git-2.42.0-64-bit.tar.bz2	2023-08-21T20:05:48Z	104.80MB

选择上图中的 Git-2.42.0-64-bit.exe（如果是其他系统或 win32 系统，请选择其他版本）进行下载并安装,安装过程一路默认，安装完成新建终端，输入 git，如果发现无任何反应请重启电脑

安装完成的标志是终端输入 git，能出现类似如下结果：

```
C:\Users\ASUS\Desktop>git
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
          [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
```

现在就可以继续创建 GetX 工程了，
终端输入 get create project:g1，如下：

```
C:\Users\ASUS\Desktop\getXtest>get create project:g1
```

- 1) Flutter Project
- 2) Get Server

Select which type of project you want to create ? [1] 1
你公司的域名是? 例: com.yourcompany com.d

包名

- 1) Swift
- 2) Objective-C

你想在 iOS 端使用什么语言? [1] 1

这里选择1或者2都行，
但是推荐都选1

- 1) Kotlin
- 2) Java

你想在 Android 端使用什么语言? [1] 1

这一处如果使用高版本flutter必
须选2否则报错

- 1) 是的!
- 2) 不

要使用 null safe 吗? [1] 2

未经过测试，应该不影响

- 1) yes
- 2) no

你想用代码检查器吗? [1] 1

```
✓ 文件: analysis_options.yaml 创建成功，路径: analysis_options.yaml
```

- 1) GetX Pattern (by Kauê)
- 2) CLEAN (by Arktekko)

Which architecture do you want to use? [1] 1

- 1) 是的!
- 2) 不

你的 lib 文件夹不是空的。你确定要覆盖你的应用吗?

警告:操作不可逆 [1] 1