Flutter Provider+MVVM搭建通用项目架构

木子雨廷t

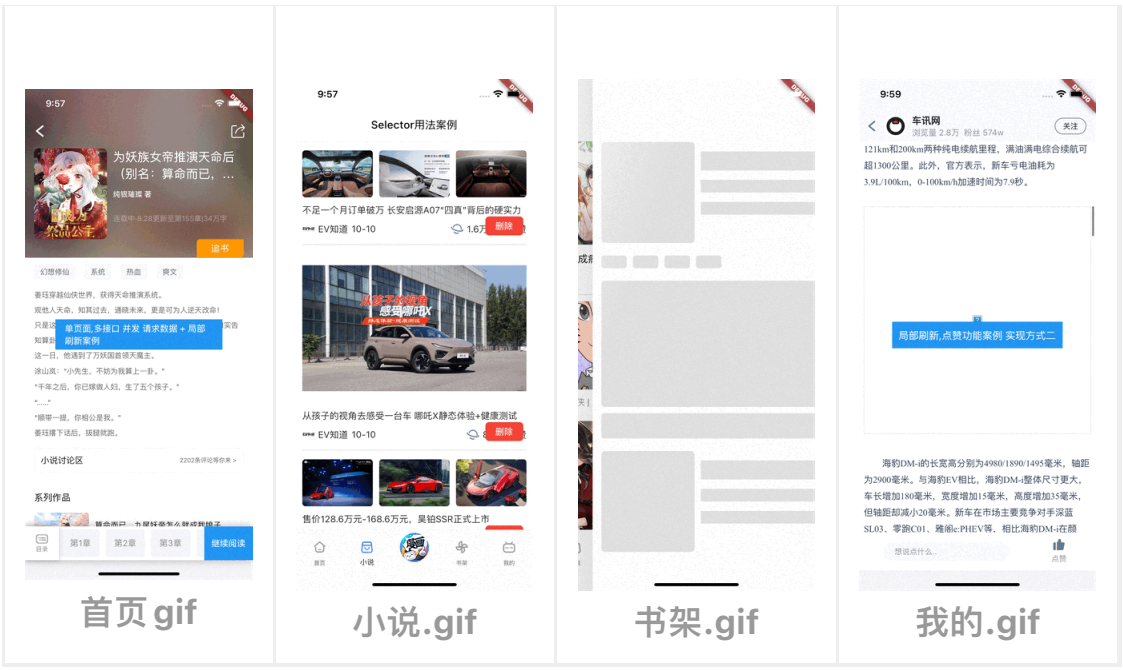
前言:

做flutter开发有些时间了,之前用过GetX和Bloc,在之前的文章中也总结过这两个框架的用法和一些常见问题,最近挤出点时间搞了一个Provider,之前在项目中也使用过Provider,但是怎么说呢,那会也是初学者用的稀里糊涂的,用的不优雅,不透彻,今天来盘一盘,MVVM+ Provider的项目写法.

Flutter 基于getX搭建通用项目架构

Flutter 基于 Bloc搭建通用项目架构

老规矩，先上效果。



一. Provider 基本用法

`Provider`有两个重要的角色。提供者：提供数据， 消费者：消费数据。他的使用也是围绕着这两个角色来展开的。

首先定义提供者， `Provider`为我们提供了非常多的提供者，总共有八种。但我们比较常用的是`ChangeNotifierProvider 和 MultiProvider 和 ChangeNotifierProxyProvider`关于其他的提供者可根据自己的实际应用场景来。可以用一个`Model`和`ViewModel`继承于或者混入`ChangeNotifier`，然后让需要使用数据的`widget`继承于`ChangeNotifierProvider`，这样当数据变化时就可以通过`ChangeNotifierProvider`来提供数据，完成页面的刷新工作。关于几个提供者的用法就不一一说了，感兴趣

的可以自己百度。

二. Provider也提供了三个消费者

Provider第三方是基于InheritedWidget封装的：

InheritedWidget

Provider也提供了三个消费者：`Provider.of`、`Consumer（会刷新不必要刷新的组件）`、`Selector（更精细化）`

1、Provider.of

InheritedWidget有个默认的约定：如果状态是希望暴露出的，应当提供一个 “of” 静态方法来获取其对象，开发者便可直接通过该方法来获取。

```
static T of<T>(BuildContext context, {bool listen = true})
```

其中 `listen`：默认true监听状态变化，false为不监听状态改变。

`Provider.of<T>(context)` 是 `Provider` 为我们提供的静态方法，当我们使用该方法去获取值的时候会返回查找到的最近的 `T` 类型的 `provider` 给我们，且也不会遍历整个组件树。

2、Consumer

Provider 中使用比较频繁的消费者，查看源码：

```
Consumer({
  Key? key,
  required this.builder,
  Widget? child,
}) : super(key: key, child: child);

...

@override
Widget buildWithChild(BuildContext context, Widget? child)
{
  return builder(
    context,
    Provider.of<T>(context),
    child,
  );
}
```

发现它就是通过 Provider.of<T>(context) 来实现的。而且实际开发中使用 Provider.of<T>(context) 比 Consumer 简单好用太多，那 Consumer有什么优势吗?

对比一下，我们发现 `Consumer` 有个 `Widget? child`，它非常重要，能够在复杂项目中，极大地缩小你的控件刷新范围。

就是在实际的开发当中只需要将需要刷新的`widget`放在

Consumer的 builder方法中，不需要刷新的方法child中，这样，大大提升了性能。

3、Selector

Selector 也是一个消费者。与Consumer类似，只是对build调用Widget方法时提供更精细的控制。 Consumer 是监听一个 Provider 中所有数据的变化， Selector 则是监听某一个/多个值的变化。

比如资讯模型 InfoModel, Selector 可以监听里面是否点赞这个属性的变化，当点赞属性变化才会刷新 点赞 widget， 其他的widget不刷新，可以做到更精细化的刷新。

但是当 Selector 监听基本数据类型时，比较的是两个值是否相同，这样是没有什么问题的，当监听的是对象时，比较的是两个对象的内存地址,所以当 Selector 监听对象时，对象进行增删操作时并不会引起 Selector 的刷新，这种就比较恶心，需要自己处理一下。

我的思路是自定义一个 Class,代码如下

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

/// select 刷新 对比的是两个对象的内存地址,用这个类来解决这个问题
class SelectorPlusData<T> {
  T? _value;
```

```
  int _version = 0;
  int _lastVersion = -1;

  T? get value => _value;

  SelectorPlusData({Key? key, T? value}) {
    _value = value;
  }

  set value(T? value) {
    _version++;
    _value = value;
  }

  bool shouldRebuild() {
    bool isUpdate = _version != _lastVersion;
    if (isUpdate) {
      _lastVersion = _version;
    }
    return isUpdate;
  }
}
```

这个对象有两个默认值 `int _version = 0; int _lastVersion = -1;` 当对象初始化时 或者 `set value`时，这两个`version`是不会相等的，所有可以用这两`version`来判断是否需要刷新。

在`Selector`中可以封装成以下代码，直接调用上面对象的`next.shouldRebuild`去决定是否进行刷新。

```
Selector<T, SelectorPlusData>(
```

```
                builder: widget.builder,
                selector: widget.plusDataSelector!,
                shouldRebuild: (pre, next) =>
next.shouldRebuild(),
                child: widget.child,
              )
```

使用代码如下：

```
ProviderSelectorWidget<NovelViewModel, List>(
        viewModel: novelViewModel,
        builder: (context, selectorPlusData, child) {
          return Container();
        },
        plusDataSelector: (context, viewModel) =>
SelectorPlusData(value: novelViewModel.dataList))
```

用`SelectorPlusData`将需要监听的数组包一下，就能完成数组变化的监听了，对于其他对象也是一样。

三. 针对 Provider+MVVM 模式设计

1、ViewModel

针对于 ViewModel的封装其实很简单，就是继承于`ChangeNotifier`监听数据变化，它用于向监听器发送通知。换言之，如果被定义为 `ChangeNotifier`，你可以订阅它的状态变

化。

另外为了方便给列表做上拉刷新和下拉加载，还增加了`ScrollController`和`RefreshController`。在列表的`viewModel`中可以直接使用。

代码如下所示：

```
class BaseViewModel extends ChangeNotifier {
  /// 列表控制器
  final ScrollController scrollController =
ScrollController();

  /// 刷新组建控制器
  final RefreshController refreshController =
RefreshController(initialRefresh: false);

}
```

2、State

状态层，主要用来定义一些属性，来进行业务上的解耦和代码上的隔离。`state`这层必须要单独分出来，因为某个页面一旦维护的状态很多，将状态变量和逻辑方法混在一起，后期维护会非常头痛。

`State`里面定义一个属性`NetState`这个属性根据网络状态来赋值，页面根据这个`NetState`来展示不同的页面，如果说展示`暂`

`无数据页面` `加载失败` `骨架屏`等等，都是根据`NetState`来决定的。

3、ProviderConsumerWidget

`Consumer`在项目中用的还是很普遍，所以直接封装了一个`ProviderConsumerWidget`。在项目中需要使用`Consumer`的地方直接使用这个类即可。

代码如下：

```
class ProviderConsumerWidget<T extends ChangeNotifier>
extends StatefulWidget {
  final Widget Function(BuildContext context, T value,
Widget? child) builder;
  final T viewModel;
  final Widget? child;
  final Function(T)? onReady;

  /// ChangeNotifierProvider 在某种场景下,程序热启动时会失效,但是
ChangeNotifierProvider<T>.value会一直保持状态
  final bool? isValue;
  const ProviderConsumerWidget(
      {super.key,
      required this.viewModel,
      this.child,
      this.onReady,
      required this.builder,
      this.isValue});

  @override
  State<ProviderConsumerWidget> createState() =>
_ProviderConsumerWidgetState<T>();
```

```dart
}

class _ProviderConsumerWidgetState<T extends
ChangeNotifier>
    extends State<ProviderConsumerWidget<T>> {
  @override
  void initState() {
    super.initState();
    if (widget.onReady != null) {
      widget.onReady!(widget.viewModel);
    }
  }

  @override
  Widget build(BuildContext context) {
    if (widget.isValue == true) {
      return ChangeNotifierProvider<T>.value(
        value: widget.viewModel,
        child: Consumer<T>(
          builder: widget.builder,
          child: widget.child,
        ),
      );
    } else {
      return ChangeNotifierProvider<T>(
        create: (_) => widget.viewModel,
        child: Consumer<T>(
          builder: widget.builder,
          child: widget.child,
        ),
      );
    }
  }
}
```

使用时需要传递的参数：

`T viewModel:` 就是viewModel对象，需要监听的数据变化的对象。
`Widget? child:`不需要刷新的`widget`，可传可不传。
`Function(T)? onReady:`数据请求或者数据变化的方法，可传可不传，不传的话在`view`中的`initState`方法中进行数据请求一样的。
`builder`函数，然后会回调`BuildContext context, T value, Widget? child`这三个参数给`view`,其中`T value`返回的就是传入的`viewModel`对象，`Widget? child`就是传入的不需要刷新的`Consumer`的`widget`
`bool? isValue:` 控制的是`ChangeNotifierProvider`两种构造方法。
当`isValue`为`true`时对应的是`ChangeNotifierProvider<T>.value`的构造方法。

```
ChangeNotifierProvider<T>.value(
        value: widget.viewModel,
        child: Consumer<T>(
          builder: widget.builder,
          child: widget.child,
        ),
      )
```

当`isValue`为`false`时对应的是`ChangeNotifierProvider<T> create`的构造方法。

```
return ChangeNotifierProvider<T>(
        create: (_) => widget.viewModel,
        child: Consumer<T>(
          builder: widget.builder,
          child: widget.child,
        ),
      );
```

问题：ChangeNotifierProvider<T>.value 和 ChangeNotifierProvider<T> create 差异之处和使用注意事项。

说实话一开始我并没有把重点放在这两个方法上面，上来直接使用的`ChangeNotifierProvider<T> create`，但是当我用`Selector`来做首页`Tabar`切换这个功能时，遇到了问题。就是每当我热重载或者热启动时，点击`Tabar`就是失效了，然后重启项目发现是好使的，只要是一热更新就失效。对此很是纳闷。百度了一下发现这方面的资料很少，就索性看了下`ChangeNotifierProvider`的源码实现。

`ChangeNotifierProvider` ->继承了 `ListenableProvider` ->继承了 `InheritedProvider` ->继承了`SingleChildStatelessWidget` ->继承了`SingleChildStatelessWidget`

结论：

`ChangeNotifierProvider`(builder模式)的父类构造器多了一个

dispose，当ChangeNotifierProvider从widget树中被移除时会自动调用dispose方法移除相应的数据，使得内存占用永远保持着一个合理的水平。

ChangeNotifierProvider.value在被移除widget树的时候不会自动调用dispose，需要手动去管理数据，比如在被移除的时候依然有其它地方想使用这个数据，并在合适的时候再去手动关闭。

那么新问题又来了，flutter热启动时ChangeNotifierProvider会被移除widget树吗?
答案肯定是不会的,我这个问题出现的原因是因为我使用的StatelessWidget,当热启动或者热重载时,StatelessWidget会失效,最后把StatelessWidget换成StatefulWidget就解决了这个问题.

但是对ChangeNotifierProvider的探索还是有用的,因为在项目开发中,经常会遇到局部刷新的场景,也就是一个view中可能会用到多个ProviderConsumerWidget或者ProviderSelectorWidget,但是他们又持有同一个viewModel对象,由于ChangeNotifierProvider create是自己释放对象的。所以这种场景就会造成当页面释放时，第一个ProviderConsumerWidget释放的时候把viewModel释放了，第二个ProviderConsumerWidget释放的时候发现自己持有的

`viewModel`已经释放了，就报错了。这种情况下就需要使用`ChangeNotifierProvider.value`初始化方法，然后在页面的`dispose`方法中自己手动释放`viewModel`。

4、ProviderSelectorWidget

```
class ProviderSelectorWidget<T extends ChangeNotifier, A>
extends StatefulWidget {
  final Widget Function(BuildContext context, dynamic
value, Widget? child) builder;
  final T viewModel;
  final Widget? child;
  final Function(T)? onReady;

  /// ChangeNotifierProvider 在某种场景下,程序热启动时会失效,但是
ChangeNotifierProvider<T>.value会一直保持状态
  final bool? isValue;

  /// 判断是否需要刷新的字段 特别需要指明的是selector的结果，必须是
不可变的对象。 如果同一个对象，只是改变对象属性，那shouldRebuild的两
个值永远是相等的。
  final SelectorPlusData Function(BuildContext, T)?
plusDataSelector;
  final A Function(BuildContext, T)? selector;

  const ProviderSelectorWidget(
      {super.key,
      required this.viewModel,
      this.child,
      this.onReady,
      required this.builder,
      this.selector,
```

```
      this.plusDataSelector,
      this.isValue});

  @override
  State<ProviderSelectorWidget> createState() =>
_ProviderSelectorWidgetState<T, A>();
}

class _ProviderSelectorWidgetState<T extends
ChangeNotifier, A>
    extends State<ProviderSelectorWidget<T, A>> {
  @override
  void initState() {
    super.initState();
    if (widget.onReady != null) {
      widget.onReady!(widget.viewModel);
    }
  }

  @override
  Widget build(BuildContext context) {
    if (widget.isValue == true) {
      return ChangeNotifierProvider<T>.value(
        value: widget.viewModel,
        child: widget.plusDataSelector != null
            ? Selector<T, SelectorPlusData>(
                builder: widget.builder,
                selector: widget.plusDataSelector!,
                shouldRebuild: (pre, next) =>
next.shouldRebuild(),
                child: widget.child,
              )
            : Selector<T, A>(
                builder: widget.builder,
```

```
                selector: widget.selector!,
                shouldRebuild: (pre, next) => pre != next,
                child: widget.child,
              ),
      );
    } else {
      return ChangeNotifierProvider<T>(
        create: (_) => widget.viewModel,
        child: widget.plusDataSelector != null
            ? Selector<T, SelectorPlusData>(
                builder: widget.builder,
                selector: widget.plusDataSelector!,
                shouldRebuild: (pre, next) =>
next.shouldRebuild(),
                child: widget.child,
              )
            : Selector<T, A>(
                builder: widget.builder,
                selector: widget.selector!,
                shouldRebuild: (pre, next) => pre != next,
                child: widget.child,
              ),
      );
    }
  }
}
```

使用时需要传递的参数：

`T viewModel:` 就是viewModel对象，需要监听的数据变化的对象。

`Widget? child:`不需要刷新的`widget`，可传可不传。
`Function(T)? onReady:`数据请求或者数据变化的方法，可传可不传，不传的话在`view`中的`initState`方法中进行数据请求一样的。
`builder`函数，然后会回调`BuildContext context, dynamic value, Widget? child`这三个参数给`view`,其中`dynamic value`返回的就是传入的`A`对象，
例子

```
ProviderSelectorWidget<TabberViewModel, int>(
      viewModel: tabberViewModel,
      selector: (context, index) =>
tabberViewModel.selectIndex,
      onReady: (viewModel) {
        viewModel.changeSelectIndex(0);
      },
      builder: (context, index, child) {
        return _buildPage(context, index);
      },
    );
```

此时`value`返回的就是传入的`int`类型的`index`。

`Widget? child`就是传入的不需要刷新的`Consumer`的`widget`
`bool? isValue:` 控制的是`ChangeNotifierProvider`两种构造方法。
当`isValue`为`true`时对应的是`ChangeNotifierProvider<T>.value`的构造方法。
`final A Function(BuildContext, T)? selector`当监听的数据是基本数据类型时使用`selector`。

`final SelectorPlusData Function(BuildContext, T)? plusDataSelector`当监听的数据是对象时使用`plusDataSelector`
例子

```
ProviderSelectorWidget<NovelViewModel, List>(
        viewModel: novelViewModel,
        onReady: (viewModel) {
          getListData();
        },
        builder: (context, selectorPlusData, child) {
          return Container();
        },
        plusDataSelector: (context, viewModel) =>
SelectorPlusData(value: novelViewModel.dataList))
```

此时`value`返回的就是传入的`SelectorPlusData`对象，获取返回值时使用`selectorPlusData.value`来获取。

5、BaseView设计

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:flutter_screenutil/
flutter_screenutil.dart';
import '../routers/navigator_utils.dart';
import '../widgets/easy_loading.dart';
import 'base_state.dart';
import 'base_will_pop.dart';
```

```dart
typedef BodyBuilder = Widget Function(BaseState baseState,
BuildContext context);

abstract class BasePage extends StatefulWidget {
  const BasePage({Key? key}) : super(key: key);

  @override
  BasePageState createState() => getState();

  ///子类实现
  BasePageState getState();
}

abstract class BasePageState<T extends BasePage> extends
State<T> {
  /// 是否渲染buildPage内容
  bool _isRenderPage = false;

  /// 是否渲染导航栏
  bool isRenderHeader = true;

  /// 导航栏颜色
  Color? navColor;

  /// 左右按钮横向padding
  final EdgeInsets _btnPaddingH =
EdgeInsets.symmetric(horizontal: 14.w, vertical: 14.h);

  /// 导航栏高度
  double navBarH = AppBar().preferredSize.height;

  /// 顶部状态栏高度
  double statusBarH = 0.0;

  /// 底部安全区域高度
```

```dart
  double bottomSafeBarH = 0.0;

  /// 页面背景色
  Color pageBgColor = const Color(0xFFF9FAFB);

  /// header显示页面title
  String pageTitle = '';

  /// 是否允许某个页iOS滑动返回，Android物理返回键返回
  bool isAllowBack = true;

  bool resizeToAvoidBottomInset = true;

  /// 是否允许点击返回上一页
  bool isBack = true;

  @override
  void initState() {
    super.initState();
    _getBarInfo();
    _addFirstFrameListener();
    print('当前类: $runtimeType');
  }

  @override
  void dispose() {
    XsEasyLoading.dismiss();
    super.dispose();
  }

  void _addFirstFrameListener() {

WidgetsBinding.instance.addPostFrameCallback((timeStamp) {
      buildComplete();
    });
```

```
  }

  void buildComplete() {}

  /// 获取屏幕状态栏和顶部导航栏的高度
  void _getBarInfo() {
    WidgetsBinding.instance.addPostFrameCallback((mag) {
      statusBarH = ScreenUtil().statusBarHeight;
      bottomSafeBarH = ScreenUtil().bottomBarHeight;
      // if (SystemUtil.isIOS() &&
ScreenUtil().bottomBarHeight > 0) {
      //   bottomSafeBarH = 14.h;
      // }
      setState(() {
        _isRenderPage = true;
      });
    });
  }

  /// 点击左边按钮
  void onTapLeft() {
    if (!isBack) return;
    NavigatorUtils.unFocus();
    NavigatorUtils.pop(context);
  }

  ///抽象header上的组件
  Widget left() {
    return Image(
      image: const AssetImage("assets/images/
back_black.png"),
      height: 20.h,
      width: 20.w,
    );
  }
```

```dart
Widget right() => SizedBox(width: 20.w);

/// 左边组件
Widget _left() {
  return InkWell(
    onTap: onTapLeft,
    child: Container(
      padding: _btnPaddingH,
      child: left(),
    ),
  );
}

/// 右边组件
Widget _right() {
  return Container(
    padding: _btnPaddingH,
    child: right(),
  );
}

/// 页面
Widget _content() {
  return Container(
    color: pageBgColor,
    height: 1.sh,
    width: 1.sw,
    child: buildPage(context),
  );
}

///子类实现，构建各自页面UI控件
Widget buildPage(BuildContext context);
```

```
  @override
  Widget build(BuildContext context) {
    return AnnotatedRegion<SystemUiOverlayStyle>(
      sized: false,
      value: SystemUiOverlayStyle.light,
      child: BaseWillPopPage(
        isAllowBack: isAllowBack,
        child: Scaffold(
          appBar: isRenderHeader == true
              ? AppBar(
                  centerTitle: true,
                  title: Text(pageTitle,
                      style: TextStyle(
                          color: Colors.black, fontSize:
17.sp, fontWeight: FontWeight.w500)),
                  leading: _left(),
                  elevation: 0.2,
                  actions: [_right()],
                  backgroundColor: navColor ??
Colors.white,
                )
              : null,
          body: _isRenderPage == false ? const SizedBox() :
_content(),
          resizeToAvoidBottomInset:
resizeToAvoidBottomInset,
        ),
      ),
    );
  }
}
```

6、MultiStateWidget设计，根据state里面的netState状态，决定页面的展示。

代码如下

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:mvvm_provider/base/empty_widget.dart';
import 'package:mvvm_provider/base/time_out_widget.dart';
import '../widgets/placeholders.dart';
import 'base_state.dart';
import 'net_error_widget.dart';

/// 空视图 builder方法 回调函数
typedef Builder = Widget Function(BuildContext context);

enum PlaceHolderType {
  /// ListView站位
  listViewPlaceHolder,

  /// GridView站位
  gridViewPlaceHolder,

  /// StaggeredGrid 站位
  staggeredGridPlaceHolder,

  /// 详情 站位
  detailPlaceHolder,

  /// 无骨架屏展示loading
  noPlaceHolder,
}
```

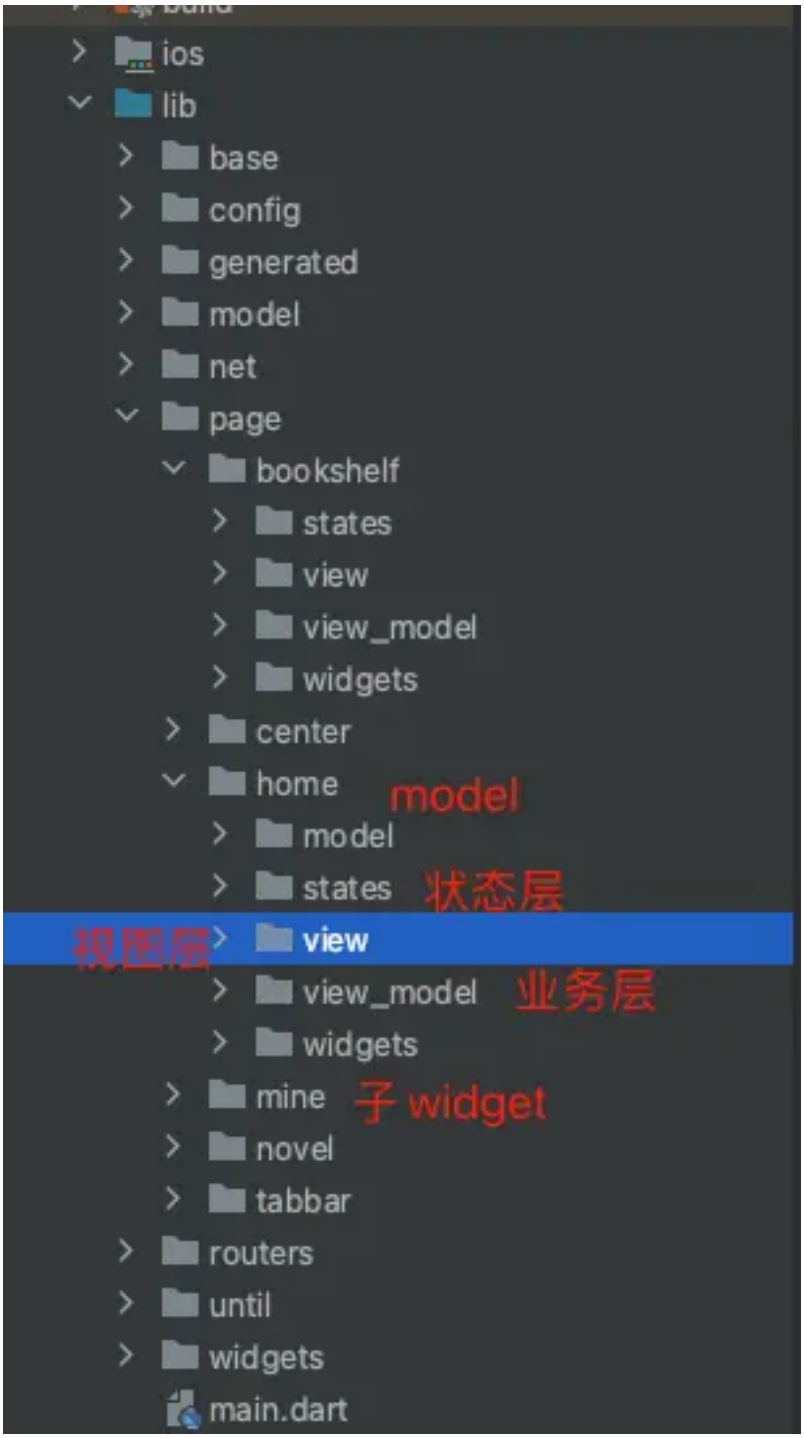
```
class MultiStateWidget extends StatelessWidget {
  final Widget? emptyWidget;
  final Widget? errorWidget;
  final String? emptyText;
  final String? errorText;
  final String? timeOutText;
  final NetState netState;
  final Builder builder;
  final Function? refreshMethod;
  final PlaceHolderType placeHolderType;
  const MultiStateWidget(
      {super.key,
      this.emptyWidget,
      this.errorWidget,
      required this.netState,
      required this.placeHolderType,
      required this.builder,
      this.refreshMethod,
      this.emptyText,
      this.errorText,
      this.timeOutText});

  @override
  Widget build(BuildContext context) {
    Widget resultWidget;
    switch (netState) {
      case NetState.error404State:
        resultWidget = NetErrorWidget(title: errorText ??
'网络404了');
        break;
      case NetState.emptyDataState:
        resultWidget = EmptyWidget(title: emptyText ?? '暂无
数据');
        break;
```

```
      case NetState.errorShowRefresh:
        resultWidget = NetErrorWidget(title: errorText ??
'网络错误', refreshMethod: refreshMethod);
        break;
      case NetState.timeOutState:
        resultWidget = TimeOutWidget(title: timeOutText ??
'加载超时请重试', refreshMethod: refreshMethod);
        break;
      case NetState.loadingState:
        if (placeHolderType ==
PlaceHolderType.gridViewPlaceHolder) {
          resultWidget = const GridViewPlaceHolder();
        } else if (placeHolderType ==
PlaceHolderType.listViewPlaceHolder) {
          resultWidget = const ListViewPlaceHolder();
        } else if (placeHolderType ==
PlaceHolderType.staggeredGridPlaceHolder) {
          resultWidget = const StaggeredGridPlaceHolder();
        } else if (placeHolderType ==
PlaceHolderType.detailPlaceHolder) {
          resultWidget = const DetailPlaceHolder();
        } else {
          resultWidget = const SizedBox();
        }
        break;
      case NetState.unknown:
        resultWidget = const EmptyWidget(title: '未知错误,请
退出重试');
        break;
      case NetState.cancelRequest:
        resultWidget = const SizedBox();
        break;
      case NetState.dataSuccessState:
        resultWidget = builder(context);
        break;
```

```
    }
    return resultWidget;
  }
}
```

7、项目截图如下



项目截图.png

四. 一个简单的列表写法案例

思路就是 定义一个`viewModel` 继承自`BaseViewModel`,在

`viewModel`中请求接口获取数据，根据接口返回数据给`state`赋值，然后一定记得给`state`里面的`netStata`完成赋值操作（这个很重要，因为页面是通过netStata的状态来展示的，如果不赋值，默认一直展示loading或者骨架屏）。

创建一个`view`继承自`BasePage`,然后在需要使用的数据的地方使用`ProviderConsumerWidget`或者`ProviderSelectorWidget`,然后`builder`里面使用`MultiStateWidget`，将`netState`传递给`MultiStateWidget`，完成页面的加载。

代码如下：

viewModel

```
import 'package:flutter/cupertino.dart';
import 'package:mvvm_provider/base/base_state.dart';
import 'package:mvvm_provider/model/banner_model.dart';
import 'package:mvvm_provider/page/home/states/
home_state.dart';
import 'package:pull_to_refresh/pull_to_refresh.dart';
import '../../../base/base_view_model.dart';
import '../../../config/handle_state.dart';
import '../../../model/response_model.dart';
import '../../../net/ltt_https.dart';
import '../../../net/http_config.dart';
import '../../../widgets/easy_loading.dart';
import '../model/cartoon_model.dart';
```

```dart
class HomeViewModel extends BaseViewModel {
  /// 创建state
  HomeState homeState = HomeState();

  /// 获取列表数据
  Future<void> getListData(String url, int number) async {
    if (url == '') {
      /// 没有更多数据了
      refreshController.refreshCompleted();
      refreshController.loadComplete();
      refreshController.loadNoData();

      homeState.netState = NetState.dataSuccessState;
      notifyListeners();
      return;
    }
    ResponseModel? responseModel =
        await LttHttp().request<CarDataModel>(url, method:
HttpConfig.get);
    homeState.netState = HandleState.handle(responseModel,
successCode: 0);
    if (homeState.netState == NetState.dataSuccessState) {
      CarDataModel carDataModel = responseModel.data;
      if (number == 1) {
        homeState.dataList = carDataModel.feeds;
        if ((homeState.dataList ?? []).isEmpty) {
          homeState.netState = NetState.emptyDataState;
        }
      } else {
        homeState.dataList?.addAll(carDataModel.feeds ??
[]);

        /// 显示没有更多数据了
        if ((carDataModel.feeds ?? []).isEmpty) {
          refreshController.loadNoData();
```

```
        }
      }
      refreshController.refreshCompleted();
      refreshController.loadComplete();
    }
    notifyListeners();
  }

  void changeIsLike() {
    homeState.isLike = !homeState.isLike;
    notifyListeners();
  }
}
```

Model

省略。。。使用 JsontoDartBeanAction 插件来完成

View

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:flutter_screenutil/
flutter_screenutil.dart';
import 'package:image_picker/image_picker.dart';
import 'package:mvvm_provider/base/
multi_state_widget.dart';
import '../../../base/base_grid_view.dart';
import '../../../base/base_stateful_page.dart';
import '../../../base/provider_consumer_widget.dart';
```

```dart
import '../../../routers/home_router.dart';
import '../../../routers/navigator_utils.dart';
import '../view_model/home_view_model.dart';
import '../widgets/car_toon_widget.dart';
import '../widgets/test.dart';

class HomePage extends BasePage {
  const HomePage({super.key});

  @override
  BasePageState<BasePage> getState() => _HomePageState();
}

class _HomePageState extends BasePageState<HomePage> {
  HomeViewModel homeViewModel = HomeViewModel();
  final ImagePicker _picker = ImagePicker();

  @override
  void initState() {
    super.initState();
    super.pageTitle = '首页';
    isBack = false;

    _onRefresh();
  }

  @override
  Widget left() {
    return const SizedBox();
  }

  /// 请求分页
  int _pageNum = 1;

  /// 上拉加载
```

```
  void _onLoading() {
    _pageNum++;
    getListData();
  }

  /// 下拉刷新
  void _onRefresh() {
    _pageNum = 1;
    getListData();
  }

  void getListData() {
    homeViewModel.getListData(getUrl(_pageNum), _pageNum);
  }

  String getUrl(int number) {
    String urlStr = '';
    if (_pageNum == 1) {
      urlStr = 'https://run.mocky.io/v3/8d98fef7-634f-4122-
a837-8c9ee892365e';
    } else if (_pageNum == 2) {
      urlStr = 'https://run.mocky.io/v3/d415f483-bdbf-445d-
ae12-703d1fd01e97';
    } else if (_pageNum == 3) {
      urlStr = 'https://run.mocky.io/v3/a9faaec6-
d70f-4365-95b2-6abdd35a6e28';
    } else {
      urlStr = '';
    }
    return urlStr;
  }

  @override
  Widget buildPage(BuildContext context) {
```

```
    // return Container(
    //   color: Colors.white,
    //   child: TextButton(
    //     child: const Text('点击拍照'),
    //     onPressed: () async {
    //       //拍照
    //       XFile? file = await _picker.pickImage(source:
ImageSource.camera,imageQuality: 100);
    //       NavigatorUtils.push(context,
HomeRouter.waterMarkPage,
    //           arguments: {"imagePath": file!.path});
    //     },
    //   ),
    // );


    return ProviderConsumerWidget<HomeViewModel>(
      viewModel: homeViewModel,
      builder: (context, viewModel, child) {
        return MultiStateWidget(
            netState: homeViewModel.homeState.netState,
            placeHolderType:
PlaceHolderType.gridViewPlaceHolder,
            builder: (BuildContext context) {
              return Container(
                color: Colors.deepOrange,
                child: BaseGridView(
                  enablePullDown: true,
                  enablePullUp: true,
                  onRefresh: _onRefresh,
                  onLoading: _onLoading,
                  refreshController:
viewModel.refreshController,
                  scrollController:
```

```
viewModel.scrollController,
                  data: viewModel.homeState.dataList ?? [],
                  padding: EdgeInsets.all(10.h),
                  childAspectRatio: 0.7,
                  crossAxisSpacing: 10.w,
                  mainAxisSpacing: 10.h,
                  crossAxisCount: 2,
                  bgColor: const Color(0xFFF3F4F8),
                  itemBuilder: (context22, index) {
                    return CarToonWidget(
                      index: index,
                      model: viewModel.homeState.dataList!
[index],
                      onTap: () async {
                        NavigatorUtils.push(context,
HomeRouter.homeDetailPage,
                            arguments: {"imageUrl":
homeViewModel.homeState.dataList?[index].image});
                      },
                    );
                  },
                ),
              );
            });
      },
    );
  }
}
```

如果感兴趣可以自行下载 Demo 观看。

8.页面多接口串行+局部刷新写法案例

2:57

为妖族女帝推演天命后（别名：算命...

连载中·8.28更新至第155章|34万字

追书

幻想修仙　系统　热血　爽文

姜珏穿越仙侠世界，获得天命推演系统。

观他人天命，知其过去，通晓未来，更是可为人逆天改命！

只是这系统，似乎出了大问题，无论他算到了什么，都得如实告知算卦者的天命。

这一日，他遇到了万妖国首领天魔主。

涂山岚："小先生，不妨为我算上一卦。"

"千年之后，你已嫁做人妇，生了五个孩子。"

"……"

"顺带一提，你相公是我。"

姜珏撂下话后，拔腿就跑。

小说讨论区　2202条评论等你来 >

系列作品

算命而已，九尾妖帝怎么就成我娘子了？！

更新至25 话

玄幻

目录　第1章　第2章　第3章　继续阅读

需求分析：
这个页面分为三个接口返回数据，分别是`小说主信息接口`，`系列作品接口`，和`更多推荐接口`。

一个页面使用三个接口，正常来说使用并发方式请求完成所有的接口再拼装数据比较好，这样用时较短对于用户用户体验较好。但是也有的情况第二个接口请求的入参，需要第一个接口的返回值，这种就必须串行了。因此，针对这个页面串行和并发两种方式都写了一下。

页面在滑动时，导航栏的透明度是随着`ListView`的滑动距离来改变的，在滑动的过程中只有`导航栏这个widget`在变化，其他的`widget`并不会发生变化，所以没有必要在根节点处刷新`整个widget`，仅仅需要刷新`导航栏widget`就可以了。完成这个`局部刷新`有三种思路吧，都是可以的。

1. 将 `导航栏widget` 抽离出去，在这个小的widget内部，使用 `setStates` 方法来完成刷新。

2. 使用 两个`ProviderConsumerWidget` 和 两个`ViewModel`来实现。
   `ViewModel A` 请求接口，完成数据组装，发送通知

notifyListeners()。
ProviderConsumerWidget A 放在页面根节点，根据数据完成整个页面的加载展示。
ViewModel B 更新ListView滑动改变距离，发送通知notifyListeners()。
ProviderConsumerWidget B 放在导航栏widget子节点，根据ListView滑动距离的改变来刷新 widget。

3. 使用 两个ProviderSelectorWidget 和 一个ViewModel来实现。
   ViewModel 请求接口，完成数据组装，更新ListView滑动改变距离,发送通知notifyListeners()。
   ProviderSelectorWidget A 放在页面根节点，根据数据完成整个页面的加载展示。根据小说的主id来决定主页面刷新还是不刷新。
   ProviderSelectorWidget B 放在导航栏widget子节点，根据ListView滑动距离的改变来刷新 widget。（最能体现 Selector 颗粒刷新 优势）

代码实现

自行下载 Demo 观看吧。

9.页面多接口并发+局部刷新写法案例


2:57
为妖族女帝推演天命后（别名：算命而已，...
纯银璀璨 著
连载中·8.28更新至第155章|34万字
追书
幻想修仙
系统
热血
爽文
姜珏穿越仙侠世界，获得天命推演系统。
观他人天命，知其过去，通晓未来，更是可为人逆天改命！
只是这系统，似乎出了大问题，无论他算到了什么，都得如实告知算卦者的天命。
这一日，他遇到了万妖国首领天魔主。
涂山岚："小先生，不妨为我算上一卦。"
"千年之后，你已嫁做人妇，生了五个孩子。"
"……"
"顺带一提，你相公是我。"
姜珏撂下话后，拔腿就跑。
小说讨论区
2202条评论等你来 >
系列作品
算命而已，九尾妖帝怎么就成我娘子
目录
第1章
第2章
第3章
继续阅读

还是这个页面，只不过是第一种方式的优化版了，接口是并发请求的，局部刷新用的是 两个`ProviderSelectorWidget` 和一个`ViewModel`来实现的。

并发请求代码

```
/// 请求全部数据
  getAllData() async {
    await Future.wait<dynamic>([getMainData(),
getSeriesData(), getRecommendData()]).then((value) {
      if (value[0] == null || value[1] == null || value[2]
== null) {
        netState = NetState.errorShowRefresh;
        notifyListeners();
        return;
      }
      mainModel = value[0];
      seriesList = value[1];
      recommendList = value[2];
      netState = NetState.dataSuccessState;
      notifyListeners();
    }).catchError((error) {
      netState = NetState.errorShowRefresh;
      notifyListeners();
    });
  }

  /// 请求主数据
  getMainData() async {
    ResponseModel? responseModel = await
LttHttp().request<CartoonModelData>(
```

```
        'https://run.mocky.io/v3/315de364-a765-40e1-8383-
f36d3ffe5bdd',
        method: HttpConfig.get);
    return responseModel.data;
  }

  /// 请求系列数据
  getSeriesData() async {
    ResponseModel? responseModel = await
LttHttp().request<CartoonSeriesData>(
        'https://run.mocky.io/v3/
c1fecbc3-296f-44c4-970c-5861970cc11b',
        method: HttpConfig.get);
    CartoonSeriesData cartoonSeriesData =
responseModel.data;
    return cartoonSeriesData.seriesComics;
  }

  /// 请求推荐数据
  getRecommendData() async {
    ResponseModel? responseModel = await
LttHttp().request<CartoonRecommendData>(
        'https://run.mocky.io/v3/7b0096eb-a1ea-4f3c-8273-
e6e700a01128',
        method: HttpConfig.get);
    CartoonRecommendData cartoonRecommendData =
responseModel.data;
    return cartoonRecommendData.infos;
  }
```

注意点：需要根据三个接口的状态来完成页面`netState`赋值

操作。

结束：

就写到这里吧，针对于`MVVM+Provider`的项目架构设计已经可以满足项目使用了，一直认为，技术就是用来沟通的，没有沟通就没有长进，在此，欢迎各种大佬吐槽沟通。Coding不易，如果感觉对您有些许的帮助，欢迎点赞评论。

声明：

仅开源供大家学习使用，禁止从事商业活动，如出现一切法律问题自行承担！！！

仅学习使用,如有侵权,造成影响,请联系本人删除,谢谢

Demo下载地址 Demo。