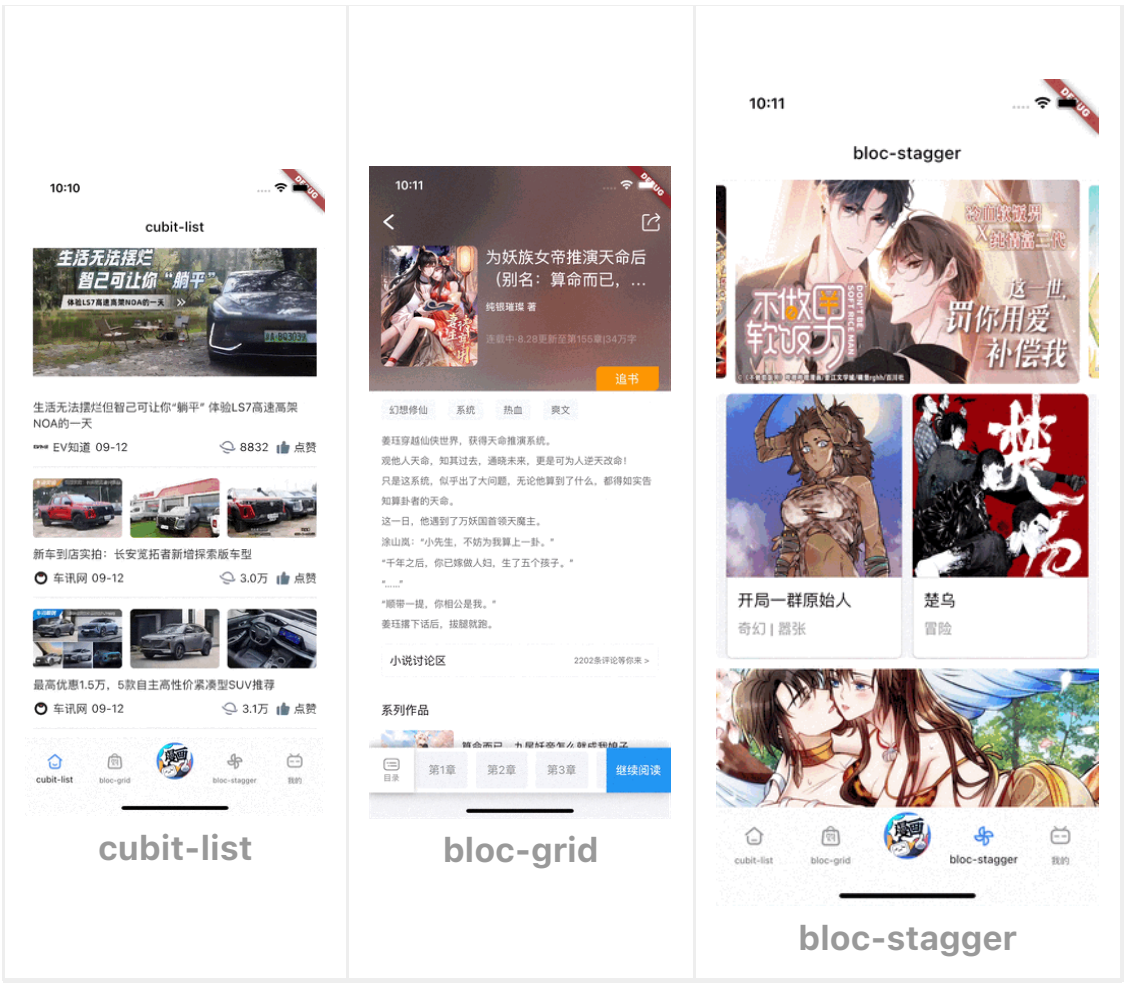
Flutter Bloc搭建通用项目架构

木子雨廷

前言：

- 最近工作较忙,利用了一些晚上下班的时间,终于写完了一个bloc Demo,之前在学习Bloc的时候看了很多文章,虽然有很多的文章在说flutter bloc模式的应用，但是百分之八九十的文章都是在说，真正写使用bloc作者开发的flutter_bloc却少之又少。没办法，只能去bloc的github上去找使用方式，最后去bloc官网翻文档。本篇文章着重讲的是bloc在项目中的使用,以及常见的场景和使用时遇到的问题。

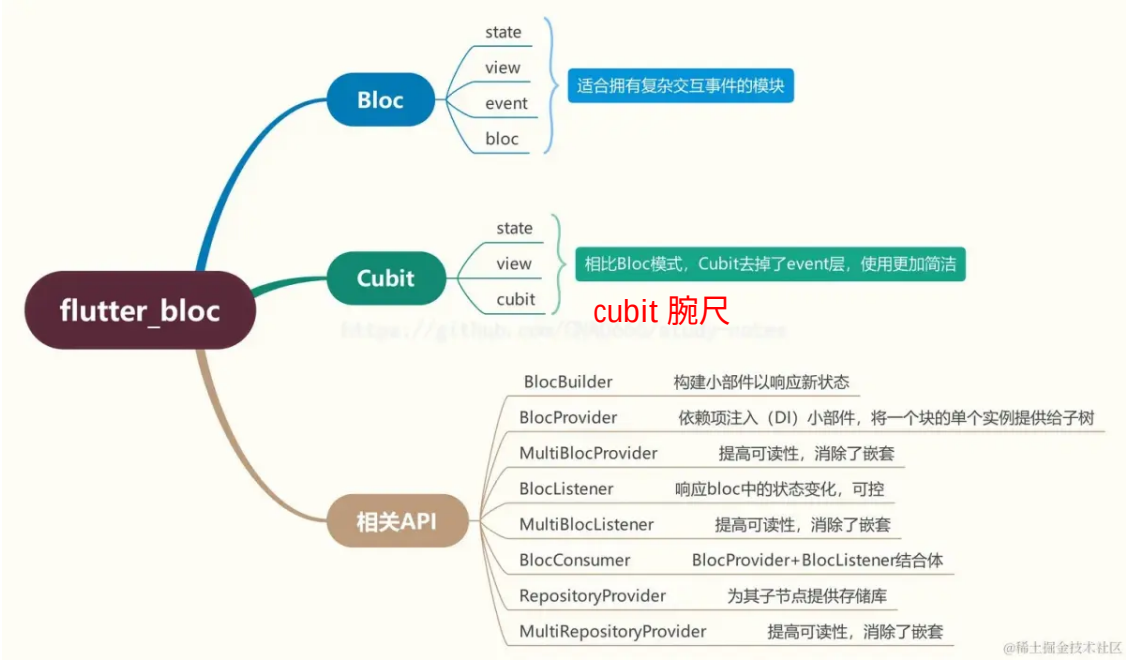
- 针对网络请求和一些常用工具也进行了封装，写了几个有针对性的页面，做项目的话可以直接拿来用。老规矩先上效果。



cubit-list　　bloc-grid　　bloc-stagger

正文：

flutter_bloc使用将从下图的三个维度说明



cubit 腕尺

image

repository 存储库

- **bloc 基本思想**

业务逻辑组件

Flutter Bloc（Business Logic Component）是一种基于流的状态管理解决方案，它将应用程序的状态与事件（也称为操作）分离开来。Bloc接收事件并根据它们来更新应用程序的状态。Bloc通常由三个主要部分组成：事件（input）、状态（output）和业务逻辑。使用Flutter Bloc，您可以将应用程序分解为不同的模块，从而使其易于维护和扩展。

- **Flutter Bloc的核心概念：**

- State：

  表示应用程序的状态。它可以是任何类型的对象，例如数字、字符串、布尔值或自定义类。是Bloc提供给外部

的数据媒介，view层通过state获取bloc里面的数据。

- Event：

  表示操作或事件，例如按钮按下、API调用或用户输入，常用场景进入页面进行网络数据请求，就定义一个网络请求的Event，当用户点击按钮就定义一个点击的Event,然后去bloc内部去处理数据然后通过state回调给view来更新状态。

- Bloc：

  通过Event获取外部操作，在内部处理逻辑接口请求或者数据处理，然后更新state,通过state把最新数据传递给view刷新状态。

  BlocProvider：是一个Flutter Bloc提供的小部件，它可以帮助我们在整个应用程序中共享和提供Bloc的实例。

- Cubit：

  相比bloc省去了Event层，view可以直接进行调用内部方法，同样也是在内部处理逻辑接口请求或者数据处理，然后更新state,通过state把最新数据传递给view刷新状态。

- BlocProvider：

  是一个Flutter Bloc提供的小部件，它可以帮助我们在

整个应用程序中共享和提供Bloc的实例。通俗来讲就是完成context和bloc对象的绑定，在我们需要用到bloc的时候，通过context就可以拿到bloc对象。BlocProvider使用的时机很重要，稍有不慎就会报错，下面会说。

context 背景，上下文

- MultiBlocProvider

  主要的使用场景就是在main方法中，绑定多个context和bloc对象，一般绑定的是在App一启动就需要展示处理逻辑的页面。

- BlocBuilder：

  是一个Flutter Bloc提供的小部件，它会在状态发生变化时自动重建，并用于构建页面。通俗来讲就是，当state对象内部的值发生变化时，BlocBuilder会自动重现构建来刷新widget。还用一个很重要的方法buildWhen:就是可以通过stateh或者view里面的其他属性来判断页面是否需要重新进行构建。

- BlocListener：

  监听bloc里面的状态，通过也是通过state进行回调，

来执行某个事件，比如说`通知刷新或界面跳转...`里面也有一个重要的方法`listenWhen:`可以有选择性的进行监听。

- BlocConsumer：
  `BlocBuilder`和`BlocListener`聚合体，既有构建功能又有监听功能。里面有`builder` `listener` `buildWhen` `listenWhen`四个方法，也很常用。

- **使用 Bloc 和 cubit 开发一个页面完整流程。**

- bloc模式：
  1.创建类，生成`bloc`类和样板代码，这里`bloc`官方提供的有插件，在`Android Studio`安装使用即可，不在多说。
  2.绑定`bloc`和`context`,使用`BlocProvider`

```
BlocProvider<NovelDetailNavBloc>(
          create: (BuildContext context) =>
NovelDetailNavBloc(),
          child: NovelDetailPage(
            imageUrl: imageUrl,
          ),
        )
```

3.定义`Event`

```
/// 获取数据
class GetNovelDetailEvent extends NovelDetailEvent {
  GetNovelDetailEvent(this.mainPath, this.seriesPath,
this.recommendPath);

  final String mainPath;
  final String seriesPath;
  final String recommendPath;
}
```

4.定义`State`

```
class NovelDetailState extends BaseState {
  CartoonModelData? mainModel;
  List<CartoonRecommendDataInfos>? recommendList;
  List<CartoonSeriesDataSeriesComics>? seriesList;

  NovelDetailState init() {
    return NovelDetailState()
      ..netState = NetState.loadingState
      ..mainModel = CartoonModelData()
      ..recommendList = []
      ..seriesList = [];
  }

  NovelDetailState clone() {
    return NovelDetailState()
      ..netState = netState
```

```
      ..mainModel = mainModel
      ..recommendList = recommendList
      ..seriesList = seriesList;
  }
}
```

5.在`Bloc`处理逻辑，并更新`state`发送更新通知

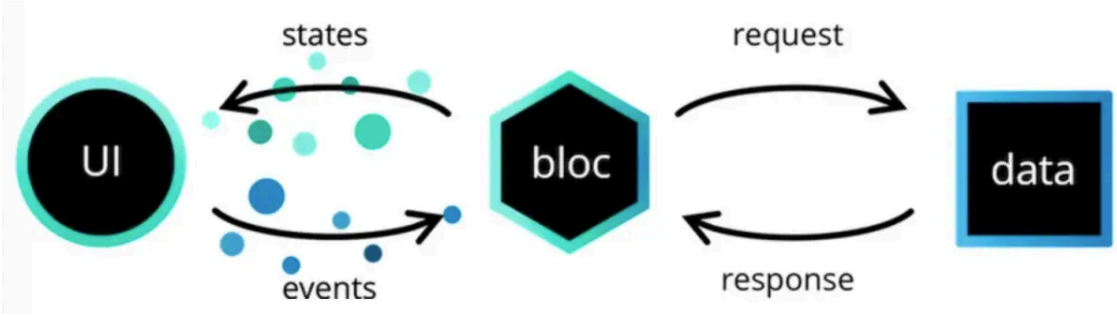
```
NovelDetailBloc() : super(NovelDetailState().init()) {
    on<GetNovelDetailEvent>(_getNovelDetailEvent);
  }

  Future<void> _getNovelDetailEvent(event, emit) async {
    XsEasyLoading.showLoading();

    /// 主数据
    ResponseModel? responseModel =
        await
LttHttp().request<CartoonModelData>(event.mainPath, method:
HttpConfig.mock);

    /// 同系列数据
    ResponseModel? responseModel2 =
        await
LttHttp().request<CartoonSeriesData>(event.seriesPath,
method: HttpConfig.mock);

    /// 推荐数据
    ResponseModel? responseModel3 =
        await
LttHttp().request<CartoonRecommendData>(event.recommendPath
, method: HttpConfig.mock);
    XsEasyLoading.dismiss();
```

```
    state.mainModel = responseModel.data;
    CartoonSeriesData cartoonSeriesData =
responseModel2.data;
    state.seriesList = cartoonSeriesData.seriesComics;
    CartoonRecommendData cartoonRecommendData =
responseModel3.data;
    state.recommendList = cartoonRecommendData.infos;
    state.netState = NetState.dataSuccessState;
    emit(state.clone());
  }
```

6.在`view`中搭建UI，通过`state`完成赋值操作。

```
Widget buildPage(BuildContext context) {
    return BlocConsumer<BlocStaggeredGridViewBloc,
StaggeredGridViewState>(
      listener: _listener,
      builder: (context, state) {
        return resultWidget(state, (baseState, context) =>
mainWidget(state), refreshMethod: () {
          _pageNum = 1;
          _getData();
        });
      },
    );
  }
```

完成上面几步，就基本玩成了一个网络列表的开发。



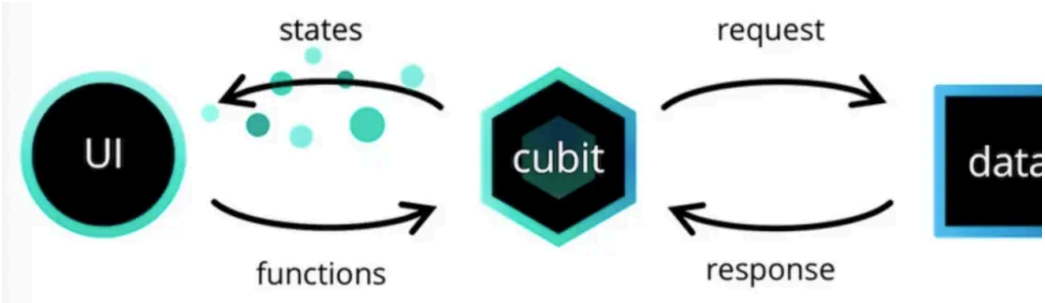
bloc模式

再结合这张官方图，有助于快速调整思路。Demo

- cubit模式：

  `cubit`模式和bloc的不同就是省去了`Event`层，其他的用法都是一样，Demo中有具体的例子。

  这就就不贴代码了。还是结合官方图，可以快速理解。



cubit模式

- **buildWhen:**

在实际的开发工作中,并不是每次`state`里面的属性发生变化都需要`build`页面,这个时候就需要`buildWhen`了.

- 使用场景

  登录注册，登录时有两个输入框，一个输入手机号码，一个输入密码，那么当输入手机号码的时候，只需要刷新手机号码的`widget`,输入密码时，只需要刷新密码的`widget`，那么这种场景就需要`buildWhen`来实现。首先来看一下`buildWhen`的内部实现

```
/// Signature for the `buildWhen` function which takes the previous `state` and
/// the current `state` and is responsible for returning a [bool] which
/// determines whether to rebuild [BlocBuilder] with the current `state`.
typedef BlocBuilderCondition<S> = bool Function(S previous, S current);
```

大致意思就是该方法返回两个`state`,根据之前的`state`和 当前的`state`来判断是否需要刷新当前的`widget`，看到这里这种场景就很好实现了。代码如下：Demo

```
buildWhen: (previous, current) {
        if (type == 1) {
          return previous.phoneNumber !=
current.phoneNumber;
        } else {
          return previous.codeNumber != current.codeNumber;
        }
      },
```

- 好处

  减少每次`build`树的范围和次数，极大的提升了性能。

  也是颗粒化刷新的一种常用方式。

- 实现原理

  底层使用`provider`的`Select`来实现的，下篇文章会着重讲一个`Select`.

- **listenWhen:**

当在`bloc`或者`cubit`中进行网络请求或者数据处理时，往往`widget`需要根据处理结果去执行某些事件，这时候就需要使用`listen`了。

- 使用场景

  在`bloc`或者`cubit`中网络请求成功后，在`widget`中，需要相应的结束下拉刷新或者上拉加载或者展示没有更多数据了，

这时候在widget中使用BlocListenr或者BlocConsumer,然后实现listen监听方法即可,但是最高效的使用listernWhen来实现，因为实际的开发当中，bloc或者cubit中会处理很多的逻辑，比如处理点赞或者收藏逻辑时，就不需要widget里面处理结束下拉刷新等事件了，只需要build页面即可。所以这种场景最好使用listernWhen了。

```
listener: _listener,
      listenWhen: (state1, state2) {
        if (state1.netLoadCount != state2.netLoadCount) {
          return true;
        }
        return false;
      },
```

在state中，定义一个属性netLoadCount,只有当前state的netLoadCount和上一个state的netLoadCount不一致时才会监听，才会去执行事件。

- **颗粒化刷新或局部刷新:**
- 例子 1
  使用buildWhen来实现，就是上面实现登录注册页面的逻辑，不在多说。

- 例子 2


2:56
DEBUG
为妖族女帝推演天命后（别名：算命...
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了？！
目录
第1章
第2章
第3章
继续阅读

颗粒化刷新例 2

以本 Demo 中的这个页面为例，首先来说，这个页面所有的数据都是网络请求而来，然后页面往上滑动时，根据滑动距离来改变导航栏的透明度和页面变化。那么就是当一开始进入页面，进行网络请求，然后`build`页面，当滑动页面时，只需要`build`导航栏`widget`就可以了，因为除了导航栏变化，别的都没有变化，没有必要从此页面的根节点进行刷新。

- 代码实现方案 1:(两个 Bloc 实现):
  当滑动`ListView`时，页面会在此`BlocBuilder`下全部都会刷新，然而，我们在滑动`ListView`时，只需要刷新`导航栏widget`,所以，可以再创建一个`NavBloc` `NavState` `NavEvent`了，`导航栏widget`用新的`导航栏的BlocBuilder`包裹，当滑动`ListView`时，更新新创建的`NavState`这样`导航栏widget`就刷新了，而根节点的`state`并没有改变，所以整体页面不会`重新build`这样就实现了局部刷新。

- 代码实现方案 2:(一个 Bloc 实现):

使用`buildWhen`来实现,`BlocBuilder`不放在`page`的根节点,滑动视图`ListView`和`NavWidget`分别用同一个`BlocBuilder`来包裹,根据不同的条件来选择重新`build`这两个`widget`.在本Demo中`有案例实现可自行查看.

- **单页面多网络请求实现思路**

- 思路1
  定义一个`bloc`或者`cubit`，使用一个`BlocBuilder`，`BlocBuilder`放在页面根节点.所有接口串行处理，等数据全部请求成功，更新`state`，调用`emit()`方法，刷新页面。loading时间会长，体验不是很好。

- 思路2
  定义一个`bloc`或者`cubit`，所有的接口并行处理，最后使用`Future.wait`来组合数据。loading时间短，体验好，注意异常逻辑处理。具体使用那种思路来实现，具体业务具体分析吧，本Demo中两种思路都有实现。

- **针对 bloc 特性 封装网络请求**

使用`bloc`多了，就会发现在`event`中如果这样请求网络会报错。代码如下：

```
https().updateData(params,
              onSuccess: (data) {
                emit();
          });
```

之前遇到过这样的问题，具体的报错信息就不贴了，`bloc`抛出的大致意思就是`event`方法是从上往下同步顺序执行的，所以当`onSuccess`异步回调时，这个`event`方法实际已经被消费掉了，所以就报错了。这是`bloc`模式下`event`的问题，在`cubit`模式下，没有此问题，可以放心大胆的写。为了在项目中使用方便,避免出错，网络统一封装成了这样，在哪种模式下都没有问题。

```
 ResponseModel? responseModel =
        await
LttHttp().request<CartoonModelData>(event.mainPath, method:
HttpConfig.get);
```

- 网络请求封装思路

返回值用通用的ResponseModel来接受，里面有`code` `message` `<T>data` 方便根据不同的`code`值进行不同处理逻辑，然后`request`方法需要传入一个泛型T,传入的这个泛型T,就是返回值`ResponseModel`的`data`,可能思路有点绕，看看代码就明白了。这样把`json`解析啥的都放在网络里面去处理了，很方便。

```
  await LttHttp().request<CartoonModelData>(event.mainPath,
methodHttpConfig.get);
state.mainModel = responseModel.data;
```

- json转model

  使用`FlutterJsonBeanFactory`插件来完成，使用方便，教程可以自行百度。使用时要注意引入别的model时，用绝对路径还是相对路径的问题。

- **BasePage设计**

常规设计吧，满足日常开发使用，属性如下。

```
/// 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
  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;
```

- **BaseState设计**

项目里面所有的 `state` 都继承于 `BaseState` 为啥要这样做？？

因为在开发一个页面需要根据网络返回的状态来判断显示`正常页面` `空数据页面` `网络报错页面`等等，也就是说页面的显示状态是由`state`来控制的，那么这些代码肯定不可能，新创建一个页面就写一堆判断，这些判断通过把`BaseState`交给`BasePage`来实现。

```
/// BaseState
/// 项目中所有需要根据网络状态显示页面的state必须继承于BaseState
enum NetState {
  /// 初始状态
  initializeState,

  /// 加载状态
  loadingState,

  /// 错误状态,显示失败界面
  error404State,

  /// 错误状态,显示刷新按钮
  errorShowRefresh,

  /// 空数据状态
  emptyDataState,

  /// 加载超时
  timeOutState,

  /// 数据获取成功状态
  dataSuccessState,
}
```

```
abstract class BaseState {
  /// 页面状态
  NetState netState = NetState.loadingState;

  /// 是否还有更多数据
  bool? isNoMoreDataState;

  /// 数据是否请求完成
  bool? isNetWorkFinish;

  /// 数据源
  List? dataList;

  /// 网络加载次数 用这个属性判断 BlocConsumer 是否需要监听刷新数据
  int netLoadCount = 0;
}
```

- 思路

  在`bloc`或者`cubit`中通过网络返回`ResponseModel`中的`code`来给`state`赋值，在`widget`中，将`state`传给`BasePage`,最终`BasePage`会根据`state`返回一个界面正确的展示效果。

处理网络层根据 ResponseModel 给 state 改变状态代码

```
class HandleState {
  static handle(ResponseModel responseModel, BaseState
state) {
    if (responseModel.code == 100200) {
      if ((state.dataList ?? []).isEmpty) {
        state.netState = NetState.emptyDataState;
      } else {
```

```
        state.netState = NetState.dataSuccessState;
      }
    } else if (responseModel.code == 404) {
      state.netState = NetState.error404State;
    } else if (responseModel.code == -100) {
      state.netState = NetState.timeOutState;
    } else {
      state.netState = NetState.errorShowRefresh;
    }
  }
}
```

widget中build代码

```
@override
  Widget buildPage(BuildContext context) {
    return BlocConsumer<MessageModuleCubit,
MessageModuleState>(
      listener: _listener,
      listenWhen: (state1, state2) {
        if (state1.netLoadCount != state2.netLoadCount) {
          return true;
        }
        return false;
      },
      builder: (context, state) {
        return resultWidget(state, (baseState, context) =>
mainWidget(state), refreshMethod: () {
          _pageNum = 1;
          _getData();
        });
      },
```

```
    );
  }
```

BasePage 中处理代码

```
Widget resultWidget(BaseState state, BodyBuilder builder,
{Function? refreshMethod}) {
    if (state.netState == NetState.loadingState) {
      return const SizedBox();
    } else if (state.netState == NetState.emptyDataState) {
      return emptyWidget('暂无数据');
    } else if (state.netState == NetState.errorShowRefresh)
{
      return errorWidget('网络错误', refreshMethod ?? () {});
    } else if (state.netState == NetState.error404State) {
      return net404Widget('页面404了');
    } else if (state.netState == NetState.initializeState)
{
      return emptyWidget('NetState 未初始化,请将状态置为
dataSuccessState');
    } else if (state.netState == NetState.timeOutState) {
      return timeOutWidget('加载超时,请重试~',
refreshMethod ?? () {});
    } else {
      return builder(state, context);
    }
  }
```

另外，所有的异常视图都支持在`widget`中重写，如果有特殊情况样式的展示，直接重写即可。

- **路由设计**

使用的是`fluro`,使用人数和点赞量很高，也比较好用，就不多说了。

- **各种base类的设计**

为了更高效的开发，Demo里面封装了常用`widget`的封装，比如`BaseListView` `BaseGridView`等等，代码写起来简直不要太爽！

结束：

就写到这里吧，针对于`Bloc`的项目架构设计已经可以了，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。