



## 使用 BLoC 对豆瓣电影App 进行重构

本节使用 BLoc 对豆瓣电影 App 进行重构。

### 重构后的工程路径

StateManager/flutter\_doubanmovie\_bloc

### BLoC 模式

BLoC 模式指的是一种开发模式，使用这种模式可以使代码的 UI 逻辑和业务逻辑完全分离，从而可以在多个平台（mobile、web、后台等）上重用代码，这里的代码重用指的是业务逻辑代码的重用。BLoC 模式第一次发表是在 2018 年的 DartConf 大会上，由 Google 的 Paolo Soares 和 Cong Hui 设计和提出。

BLoC 的全称是 **B**usiness **L**ogic **C**omponent，这个单词可以拆成两部分来看，第一部分是 Business Logic，就是业务逻辑，第二部分是 Component，就是组件，连起来就是业务逻辑组件，这里也可以看出 BLoC 其实指的就是业务逻辑组件，而且这里的业务逻辑组件是完全独立的，可以和 UI 逻辑进行完全分离。

BLoC 模式的核心思想是将 UI 逻辑和业务逻辑分开。在为了达到这一目的，BLoC 模式里使用了 [响应式编程](#)（Reactive Programming），先来介绍一下 [响应式编程](#)，看 [响应式编程](#) 是如何实现 UI 逻辑和业务逻辑分离的。

### 响应式编程

[响应式编程](#) 使用异步数据流进行编程。在响应式编程里，所有的变化，不管是被动的还是主动的，比如 UI 的点击、变量的变化、数据请求等，都会向异步数据流管道里发送消息，同时，在其他地方会监听数据流，将会收到消息并产生适当的响应。

从这段描述里就可以看到，[响应式编程](#) 里的数据不是通过参数传递来实现的，而是通过数据流管道来传送，因此数据的发送方和接收方就不需要有依赖关系，实现了解耦，这也是





## 代码所在位置

flutter\_widget\_demo/lib/reactive/CounterWidget.dart

## 使用响应式编程开发

Flutter 响应式编程的三元素是：

- StreamController：数据流管道
- StreamSink：发出消息
- Stream：收到消息

为了便于理解，这里写一个简单的例子：

```
import 'dart:async';

import 'package:flutter/material.dart';

void main() => runApp(CounterWidget());

class CounterWidget extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    // TODO: implement createState
    return CounterState();
  }
}

class CounterState extends State<CounterWidget> {
  @override
  Widget build(BuildContext context) {
    // TODO: implement build
    return MaterialApp(
      title: "Flutter Demo",
      theme: ThemeData(
        primaryColor: Colors.blue,
      ),
      home: Scaffold(
        appBar: AppBar(title: Text("Flutter 响应式编程")),
        body: Center(
          child: Text('0'),
        ),
        floatingActionButton: FloatingActionButton(
```



```
    );  
  }  
}
```

这里写了一个页面，中间是 Text，右下角还有一个 floatingActionButton，运行后的效果为：





StreamController:

```
class CounterState extends State<CounterWidget> {

  static final StreamController<int> _streamController =
    StreamController<int>();

  @override
  Widget build(BuildContext context) {
    // TODO: implement build
    ...
  }
}
```

StreamSink 通过 `_streamController.sink` 获取, Stream 通过 `_streamController.stream` 获取:

```
class CounterState extends State<CounterWidget> {

  static final StreamController<int> _streamController =
    StreamController<int>();
  static final StreamSink<int> _sink = _streamController.sink;
  static final Stream<int> _stream = _streamController.stream;

  ...
}
```

然后就可以通过 `_sink` 发送消息, 在 `_stream` 处接受消息, 这里你肯定会比较迷惑, 发送一个消息, 为什么搞的这么麻烦? 这正是响应式编程的魅力所在, 如果直接发送, 那么就是同步的, 如果要实现异步发送, 按照正常的实现, 就必须写很多监听和回调, 很容易陷入回调陷阱, 而在响应式编程里, 我们只需关心 `_sink` 和 `_stream`, 在 `_sink` 里发送消息, 在 `_stream` 处接受消息, 不需要写额外的监听和回调, StreamController 会帮我们处理, 而且在 StreamController 里也可以对接受到的数据处理后在发送。

因为要写一个自增的功能, 需要定义一个变量 `_count`, 默认值为 0:

```
class CounterState extends State<CounterWidget> {
  int _count = 0;

  static final StreamController<int> _streamController =
    StreamController<int>();
```





在 `floatingActionButton` 点击的时候发送数据：

```
floatingActionButton: FloatingActionButton(  
  child: Icon(Icons.add),  
  onPressed: () {  
    _sink.add(++_count);  
  },  
)
```

`Text` 处接受信息，因为 `Text` 是 `Widget`，所以要使用 `StreamBuilder`：

```
StreamBuilder(  
  stream: _stream,  
  initialState: 0,  
  builder: (context, snapshot) {  
    return Center(  
      child: Text('${snapshot.data}'),  
    );  
  },  
)
```

`StreamBuilder` 里的 `stream` 赋值为 `_stream`，意思是接受 `_stream` 里的数据，`initialData` 为 0，表示默认的数据为 0，`builder` 里返回 `Text`，`snapshot.data` 表示的是接受到的数据。然后一个响应式编程的自增功能就实现了，点击 `floatingActionButton`，`Text` 里的数据就不断自增。

但是在这段代码里，`floatingActionButton` 里面还涉及到了具体的业务的逻辑：

```
_sink.add(++_count);
```

正确的做法应该是，`floatingActionButton` 不应该关心到底是自增还是自减这种具体的业务逻辑，它只要发送我被点击了这个通知，具体的业务逻辑在外面进行处理，所以这里可以这么改：

```
class CounterState extends State<CounterWidget> {  
  ...  
  
  @override
```





```
...
floatingActionButton: FloatingActionButton(
  child: Icon(Icons.add),
  onPressed: () {
    _calculate();
  },
),
...
);
}

void _calculate(){
  _sink.add(++_count);
}
}
```

新增一个 `_calculate()` 方法，在 `floatingActionButton` 里调用 `_calculate()` 而不是直接使用 `_sink.add(++_count)`。这时候可能也有人有疑问，这不是闲着蛋疼吗，多写了一个方法，结果最后调用的代码都是一样的，不是变复杂了吗？

虽然确实多了几行代码，但是这几行代码对框架来说非常有意义，首先，`floatingActionButton` 就不用关心具体的业务，只负责发送事件；再者，假设功能由自增变为自减，在原来的代码里，你就得在 `floatingActionButton` 里把 `++` 改为 `--`，这里功能比较简单还好说，如果功能一旦复杂，这里修改就会比较麻烦，不仅破坏了原来的代码，使得功能不容易扩展，但是如果把 `_sink.add(++_count);` 封装在 `_calculate()` 方法里，这样如果功能有修改，只要在 `_calculate()` 方法里写就好了，`floatingActionButton` 和 `Text` 都不会受影响。就因为多了这几行代码，`floatingActionButton` 和 `Text` 就不用关心业务，只需要做好 UI 展示就行，在保证代码健壮性的同时也保证了代码的扩展性。

现在对这段代码画一个流程图：

在这个流程图里，`floatingActionButton` 被点击后，发送事件，触发 `_calculate()` 方法，`_calculate()` 方法负责业务逻辑，`_count` 自增，使用 `sink` 将数据发送出去，`stream` 收到数据 `data`，使 `Text` 用最新的 `data` 数据重建。





`_calculate()` 可以抽象成 event 事件，data 抽象成 state 状态。

接下来讲 BLoC 模式里的事件和状态流向图。

## BLoC 模式里的事件和状态流向图

上图是 BLoC 模式里的事件和状态流向图：

1. Widget 向 BLoC 发送事件
2. 事件会触发 BLoC 里的 sink
3. 然后 Stream 会把 State 通知给 Widget

这里的 Event 是为了把 Widget 和具体的业务逻辑分离抽象出来的东西，State 就是 Widget 显示需要用到的数据，也是和业务逻辑分离的。

最终，由 BLoC 实现的业务逻辑层，具有以下的特点：

- BLoC 依赖响应式编程
- 有 Event 和 State

由此，BLoC 实现了业务逻辑层和 UI 逻辑的分离，为此带来了巨大的好处：

- 可以用对 App 影响最小的方式修改业务逻辑
- 可以修改 UI，而不用担心影响业务逻辑
- 更加方便单元测试

## BLoC 模式的架构图



上图是 BLoC 模式的架构图，看到这里你觉得和某个模式很像，没错就是 MVVM：





总共有四层，从上到下分别是：

- UI Screen
- BLoC
- Repository
- Network Provider

Widget 对应的是 MVVM 里的 View，BLoC 对应的是 MVVM 里的 ViewModel，Repository 和 Network Provider 对应的是 MVVM 里的 Model。

从这里也可以看出，BLoC 其实指的是一种开发模式，BLoC 也有很多种实现，这里介绍一个第三方库 `flutter_bloc`，一个实现 BLoC 模式的 Flutter 库。

## flutter\_bloc 的使用





- Bloc
- BlocBuilder
- BlocProvider
- BlocProviderTree
- BlocListener
- BlocListenerTree

## Bloc

Bloc 类是用来实现如下模块的：

!

在这里，请大家注意一下 BLoC 和 Bloc，BLoC 是大写，Bloc 是小写，BLoC 指的是 BLoC 开发模式，Bloc 指的是 BLoC 开发模式实现里的一个类。

可以看到 Bloc 类里包含了 Event、State，也有 sink、stream 响应式编程，不过 sink、stream 响应式编程已经被 `flutter_bloc` 框架实现，我们只要关心 Event 和 State 就行，这个在 Bloc 类的定义里也能体现出来，Bloc 类的定义为：

```
abstract class Bloc<Event, State> {  
  ...  
}
```

Bloc 类里有两个泛型：Event 和 State，Event 是从外部接受到的事件，State 是输出 Widget 关心的状态。





```
enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
  @override
  int get initialState => 0;

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.decrement:
        yield currentState - 1;
        break;
      case CounterEvent.increment:
        yield currentState + 1;
        break;
    }
  }
}
```

## BlocBuilder

BlocBuilder 是一个 Widget，它的功能类似于前面讲的 StreamBuilder，但是使用起来更简单，BlocBuilder 监听 Bloc 的状态，当状态发生变化时，就重建 Widget，因此 BlocBuilder 有两个参数：Bloc 和 BlocWidgetBuilder，我们可以看一下 BlocBuilder 的构造函数：

```
const BlocBuilder({
  Key key,
  @required this.bloc,
  @required this.builder,
})
```

| 参数名字    | 参数类型              | 意义                        | 必选 or 可选 |
|---------|-------------------|---------------------------|----------|
| key     | Key               | Widget 的标识                | 可选       |
| bloc    | Bloc<E, S>        | 监听已经实现的 Bloc 类            | 必选       |
| builder | BlocWidgetBuilder | 监听 Bloc 类里状态的变化，重建 Widget | 必选       |





使用方法如下：

```
BlocBuilder(  
  bloc: BlocA(),  
  builder: (context, state) {  
    // return widget here based on BlocA's state  
    return WidgetA();  
  }  
)
```

## BlocProvider

BlocProvider 是一个 Widget，可以将 Bloc 类提供給它的子 Widget。BlocProvider 经常用来作为依赖注入的部件，以便将单个 Bloc 类的实例，在多个 Widget 里共享。

使用方法如下：

```
BlocA blocA = BlocA();
```

```
BlocProvider(  
  bloc: blocA,  
  child: ChildA(),  
)
```

```
BlocProvider(  
  bloc: blocA,  
  child: ChildB(),  
)
```

上面的使用方法，就将一个实例 blocA，分享到了 ChildA 和 ChildB 里。

然后如果想在 ChildA 或 ChildB 里拿到 BlocA 的实例，可以这么做：

```
BlocProvider.of<BlocA>(context)
```

## BlocProviderTree





假设 ChildA 需要用到 BlocA、BlocB、BlocC，如果用 BlocProvider 实现的话，就是：

```
BlocProvider<BlocA>(  
  bloc: BlocA(),  
  child: BlocProvider<BlocB>(  
    bloc: BlocB(),  
    child: BlocProvider<BlocC>(  
      value: BlocC(),  
      child: ChildA(),  
    )  
  )  
)
```

使用 BlocProviderTree 可以把多个 BlocProvider 合成一个：

```
BlocProviderTree(  
  blocProviders: [  
    BlocProvider<BlocA>(bloc: BlocA()),  
    BlocProvider<BlocB>(bloc: BlocB()),  
    BlocProvider<BlocC>(bloc: BlocC()),  
  ],  
  child: ChildA(),  
)
```

## BlocListener

BlocListener 是一个 Widget，有两个参数：Bloc 和 BlocWidgetListener，类似于 BlocBuilder，需要接收一个 Bloc 类作为参数，但也有不同的地方，BlocBuilder 里的 BlocWidgetBuilder 需要返回 Widget，而 BlocWidgetListener 不用返回 Widget，它的返回类型是 void，所以用来做一些其他操作，例如：弹对话框、弹 SnackBar、跳转到新的页面等。

使用方法如下：

```
BlocListener(  
  bloc: _bloc,  
  listener: (context, state) {  
    if (state is Success) {  
      Navigator.of(context).pushNamed('/details');  
    }  
  }  
)
```





## BlocListenerTree

BlocListenerTree 是一个 Widget，用于将多个 BlockListener 合成一个 Widget。

例如：

```
BlocListener<BlocAEvent, BlocAState>(  
  bloc: BlocA(),  
  listener: (BuildContext context, BlocAState state) {},  
  child: BlocListener<BlocBEvent, BlocBState>(  
    bloc: BlocB(),  
    listener: (BuildContext context, BlocBState state) {},  
    child: BlocListener<BlocCEvent, BlocCState>(  
      bloc: BlocC(),  
      listener: (BuildContext context, BlocCState state) {},  
      child: ChildA(),  
    ),  
  ),  
)
```

就可以用 BlocListenerTree 实现为：

```
BlocListenerTree(  
  blocListeners: [  
    BlocListener<BlocAEvent, BlocAState>(  
      bloc: BlocA(),  
      listener: (BuildContext context, BlocAState state) {},  
    ),  
    BlocListener<BlocBEvent, BlocBState>(  
      bloc: BlocB(),  
      listener: (BuildContext context, BlocBState state) {},  
    ),  
    BlocListener<BlocCEvent, BlocCState>(  
      bloc: BlocC(),  
      listener: (BuildContext context, BlocCState state) {},  
    ),  
  ],  
  child: ChildA(),  
)
```





接下来使用 `flutter_bloc` 这个库对豆瓣电影App 进行重构。

## 添加依赖

首先，在 `pubspec.yaml` 里添加 `flutter_bloc` 库的依赖：

```
dependencies:  
  ...  
  flutter_bloc: ^0.14.0
```

在 VS Code 里使用快捷键保存后，会自动下载依赖库。

## 目录调整

然后开始重构，在 `lib` 根目录下新建一个 `bloc` 的文件夹和一个 `ui` 的文件夹，然后把除了 `main.dart` 的文件都移到 `ui` 的文件夹下，如图：

这样做的目的是把 `bloc` 和 `ui` 分开。

## BLoC 的核心模块

现在开始写 BLoC 的核心模块，包括：





- Bloc

在 bloc 文件夹下，新建一个文件 CityBloc.dart，BLoC 的核心模块都写到这个文件夹下。

## 定义 State

首先，要把共享的状态定义出来。因为有了前面两节重构的经验，这次我们直接对全局状态 `_curCity` 进行重构，在 CityBloc.dart 里定义一个 CityState，代码为：

```
class CityState{
  String _curCity;

  get curCity => _curCity;

  CityState(this._curCity);
}
```

## 定义 Evnet

在 CityBloc.dart 里定义一个 CityEvent，代码为：

```
class CityState{
  ...
}

class CityEvent {
  String _city;

  get city => _city;

  CityEvent(this._city);
}
```

CityEvent 用来更新当前选中的城市。

## 写 Bloc





```
import 'package:bloc/bloc.dart';
import 'package:shared_preferences/shared_preferences.dart';

class CityState{
  ...
}

class CityEvent {
  ...
}

class CityBloc extends Bloc<CityEvent, CityState> {

  CityBloc() {
    initData();
  }

  void initData() async {
    final prefs = await SharedPreferences.getInstance(); //获取 prefs

    String city = prefs.getString('curCity'); //获取 key 为 curCity 的值

    dispatch(CityEvent(city));
  }

  @override
  // TODO: implement initialState
  get initialState => CityState(null); //默认值为空

  @override
  Stream<CityState> mapEventToState(CityEvent event) async*{
    // TODO: implement mapEventToState
    yield new CityState(event.city);
  }
}
```

注意 CityBloc 实现里的泛型：

```
class CityBloc extends Bloc<CityEvent, CityState>
```





在 CityBloc 的默认构造函数里去读取本地的数据。initialState 里 CityState 的值默认为 null。

`mapEventToState` 方法，是接受 Event，然后返回最新的状态。

## 注入 Bloc

然后在 main.dart 里，给子 Widget 注入 CityBloc：

```
class _MyHomePageState extends State<MyHomePage> {  
  ...  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: BlocProvider<CityBloc>(  
        bloc: CityBloc(),  
        child: _widgetItems[_selectedIndex], //选中不同的选项显示不同的界面  
      ),  
      ...  
    );  
  }  
  ...  
}
```

在 HotWidget 里使用 BlocBuilder 监听 CityBloc：

```
class HotWidgetState extends State<HotWidget> {  
  ...  
  
  @override  
  Widget build(BuildContext context) {  
    // TODO: implement build  
    print('HotWidgetState build');  
  
    return BlocBuilder(  
      bloc: BlocProvider.of<CityBloc>(context),  
      builder: (context, String curCity) {  
        if (curCity != null && curCity.isNotEmpty) {  
          //如果 curCity 不为空  
          ...  
        }  
      }  
    );  
  }  
}
```





```

    }
  },
);
}

void _jumpToCitysWidget() async {
  var selectCity =
    await Navigator.pushNamed(context, '/Citys', arguments: BlocProvider.of<C
  if (selectCity == null) return;

  final prefs = await SharedPreferences.getInstance();
  prefs.setString('curCity', selectCity); //存取数据

  //城市变化时, 使用 CityBloc dispatch CityEvent
  BlocProvider.of<CityBloc>(context).dispatch(CityEvent(selectCity));
}
}

```

当 CityBloc 状态发生变化时, 就会创建 HotWidgetState 里的 Widget。这里在看一下 `_jumpToCitysWidget()` 里的改动, 在子 Widget 里通过 `BlocProvider.of<CityBloc>(context)` 获取 CityBloc 实例, 从而获取当前的城市:

```
BlocProvider.of<CityBloc>(context).currentState.curCity
```

如果要改变城市, 就是用 CityBloc 的 dispatch 方法:

```
BlocProvider.of<CityBloc>(context).dispatch(CityEvent(selectCity));
```

CitysWidget 就不用重构了, 因为 CitysWidget 采用参数传递的方法就没问题。至此, 一个完整的 BLoC 模式的开发方式就展现在你的眼前。当然, 我们的重构还没有结束, 现在继续。

## HotMoviesListWidget 的 BLoC 模式重构

接下来对 HotMoviesListWidget 重构, 这个是重点, 因为 HotMoviesListWidget 里的状态是本地状态, 而且前面两种方式对这里的重构, 都不太好。

所以我们这里也要着重观察一下, BLoC 模式对本地状态的处理。





```
import 'package:bloc/bloc.dart';
import 'package:flutter_doubanmovie/bloc/HotMovieData.dart';
import 'package:flutter_doubanmovie/bloc/MoviesRepository.dart';

class HotMoviesListState{
  List<HotMovieData> _list;

  get list => _list;

  HotMoviesListState(this._list);
}

class HotMoviesEvent{
  String _curCity;

  get curCity => _curCity;

  HotMoviesEvent(this._curCity);
}

class HotMoviesListBloc extends Bloc<HotMoviesEvent,HotMoviesListState>{

  final _movieRepository = MoviesRepository();

  @override
  // TODO: implement initialState
  HotMoviesListState get initialState => HotMoviesListState(null);

  @override
  Stream<HotMoviesListState> mapEventToState(HotMoviesEvent event) async *{
    // TODO: implement mapEventToState
    List<HotMovieData> movies = await _movieRepository.fetchMoviesList(event.curC
    yield HotMoviesListState(movies);
  }
}
```

HotMoviesListBloc 和 CityBloc 类似，同样包含了：

- State
- Event
- Bloc





## Flutter 完全手册

然后实现 `MoviesRepository`，同样在 `bloc` 文件夹下新建文件 `MoviesRepository.dart`，代码为：

```
import 'package:flutter_doubanmovie/bloc/HotMovieData.dart';
import 'package:flutter_doubanmovie/bloc/MoviesApiProvider.dart';

class MoviesRepository{
  final _movieApiProvider = MoviesApiProvider();

  Future<List<HotMovieData>> fetchMoviesList(String city) async {
    return _movieApiProvider.fetchMoviesList(city);
  }
}
```

在实现 `MoviesApiProvider`，同样在 `bloc` 文件夹下新建文件 `MoviesApiProvider.dart`，代码为：

```
import 'dart:convert';

import 'package:flutter_doubanmovie/bloc/HotMovieData.dart';
import 'package:http/http.dart' as http;

class MoviesApiProvider {

  Future<List<HotMovieData>> fetchMoviesList(String city) async {
    List<HotMovieData> serverDataList = new List();
    var response = await http.get(
      'https://api.douban.com/v2/movie/in_theaters?apikey=0b2bdeda43b5688921839'
      city +
      '&start=0&count=10');
    //成功获取数据
    if (response.statusCode == 200) {
      var responseJson = json.decode(response.body);
      for (dynamic data in responseJson['subjects']) {
        HotMovieData hotMovieData = HotMovieData.fromJson(data);
        serverDataList.add(hotMovieData);
      }
    }

    return serverDataList;
  }
}
```





```
class _MyHomePageState extends State<MyHomePage> {  
  ...  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: BlocProviderTree(  
        blocProviders: [  
          BlocProvider<CityBloc>(bloc: CityBloc()),  
          BlocProvider<HotMoviesListBloc>(bloc: HotMoviesListBloc())  
        ],  
        child: _widgetItems[_selectedIndex], //选中不同的选项显示不同的界面  
      ),  
      ...  
    );  
  }  
  ...  
}
```

HotMoviesListWidget 改为:

```
import 'package:flutter/material.dart';  
import 'package:flutter_bloc/flutter_bloc.dart';  
import 'package:flutter_doubanmovie/bloc/HotMoviesListBloc.dart';  
import 'package:flutter_doubanmovie/ui/hot/hotlist/ui/item/HotMovieItemWidget.dart';  
  
class HotMoviesListWidget extends StatefulWidget {  
  
  HotMoviesListWidget() {}  
  
  @override  
  State<StatefulWidget> createState() {  
    // TODO: implement createState  
    return HotMoviesListWidgetState();  
  }  
}  
  
class HotMoviesListWidgetState extends State<HotMoviesListWidget>  
  with AutomaticKeepAliveClientMixin {  
  @override  
  void initState() {  
    // TODO: implement initState
```



## Flutter 完全手册

```

@override
Widget build(BuildContext context) {
  // TODO: implement build

  return BlocBuilder(
    bloc: BlocProvider.of<HotMoviesListBloc>(context),
    builder: (context, HotMoviesListState moviesListState) {
      if (moviesListState == null || moviesListState.list == null || moviesList
        return Center(
          child: CircularProgressIndicator(),
        );
      } else {
        return MediaQuery.removePadding(
          removeTop: true,
          context: context,
          child: ListView.separated(
            itemCount: moviesListState.list.length,
            itemBuilder: (context, index) {
              return HotMovieItemWidget(moviesListState.list[index]);
            },
            separatorBuilder: (context, index) {
              return Divider(
                height: 1,
                color: Colors.black26,
              );
            },
          ),
        );
      }
    },
  );
}

@override
// TODO: implement wantKeepAlive
bool get wantKeepAlive => true; //返回 true, 表示不会被回收
}

```

这里首先把 HotMoviesListWidget 原来有参数的构造函数删掉了，这样 HotMoviesListWidget 彻底和业务隔开，它不需要知道现在是哪个城市，只需要显示就行，然后 HotMoviesListWidgetState 里的 build 改成了 BlocBuilder：





## Flutter 完全手册

```

        builder: (context, HotMoviesListState moviesListState) {
            ...
        },
    );

```

BlocBuilder 的 bloc 为 HotMoviesListBloc，builder 里的 moviesListState 就是 HotMoviesListWidget 的本地状态，里面有要展示的电影列表数据。

还有一个很关键的点，就是 HotMoviesListBloc 也需要一个地方，来发出事件，触发 HotMoviesListWidget 的重建，这里发出事件的地方选择在 HotWidget 里：

```

class HotWidgetState extends State<HotWidget> {
    ...

    @override
    Widget build(BuildContext context) {
        // TODO: implement build
        print('HotWidgetState build');

        return BlocBuilder(
            bloc: BlocProvider.of<CityBloc>(context),
            builder: (context, String curCity) {
                if (curCity != null && curCity.isNotEmpty) {
                    //如果 curCity 不为空
                    BlocProvider.of<HotMoviesListBloc>(context)
                        .dispatch(HotMoviesEvent(curCity));
                    ...
                }
                else {
                    ...
                }
            }
        );
    }
    ...
}

```

至此，就重构完了，重构完后的目录结构为：





留言

评论将在后台进行审核，审核通过后对所有人可见

