Flutter状态管理学习手册 (三)——Bloc

WinDin

一、Bloc 介绍

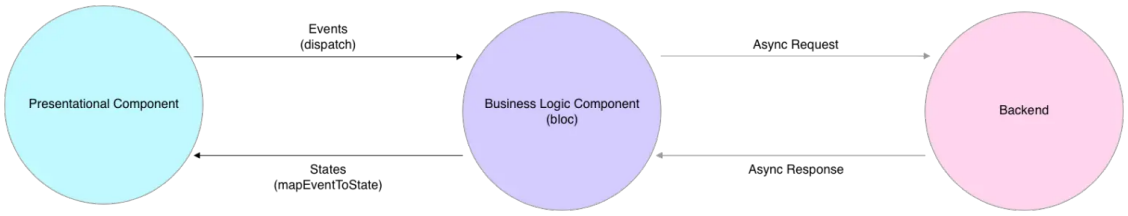
Bloc 的名字比较新颖，这个状态管理框架的目的是将 UI 层和业务逻辑进行分离。Bloc 的复杂度处于 ScopedModel 和 Redux 之间，相较于 ScopedModel，Bloc 拥有分明的架构处于业务逻辑，相较于 Redux，Bloc 着重于业务逻辑的分处理解，使得整个框架对于开发来讲简单实用。

二、Bloc 的层次结构

Bloc 分为三层：

- Data Layer(数据层)，用于提供数据。
- Bloc(Business Logic) Layer(业务层)，通过继续 Bloc 类实现，用于处理业务逻辑。
- Presentation Layer(表现层)，用于 UI 构建。

Presentation Layer 只与 Bloc Layer 交互，Data Laye 也只与 Bloc Layer 交互。Bloc Layer 作为重要一层，处于表现层和数据层之间，使得 UI 和数据通过 Bloc Layer 进行交互。



image

由此可见，Bloc 的架构和客户端主流的 MVC 和 MVP 架构比较相似，但也存在 Event 和 State 的概念一同构成响应式框架。

三、Bloc 需要知道的概念

BlocProvider，通常做为 App 的根布局。BlocProvider 可以保存 Bloc，在其它页面通过 `BlocProvider.of<Bloc>(context)` 获取 Bloc。

Event，用户操作 UI 后发出的事件，用于通知 Bloc 层事件发生。

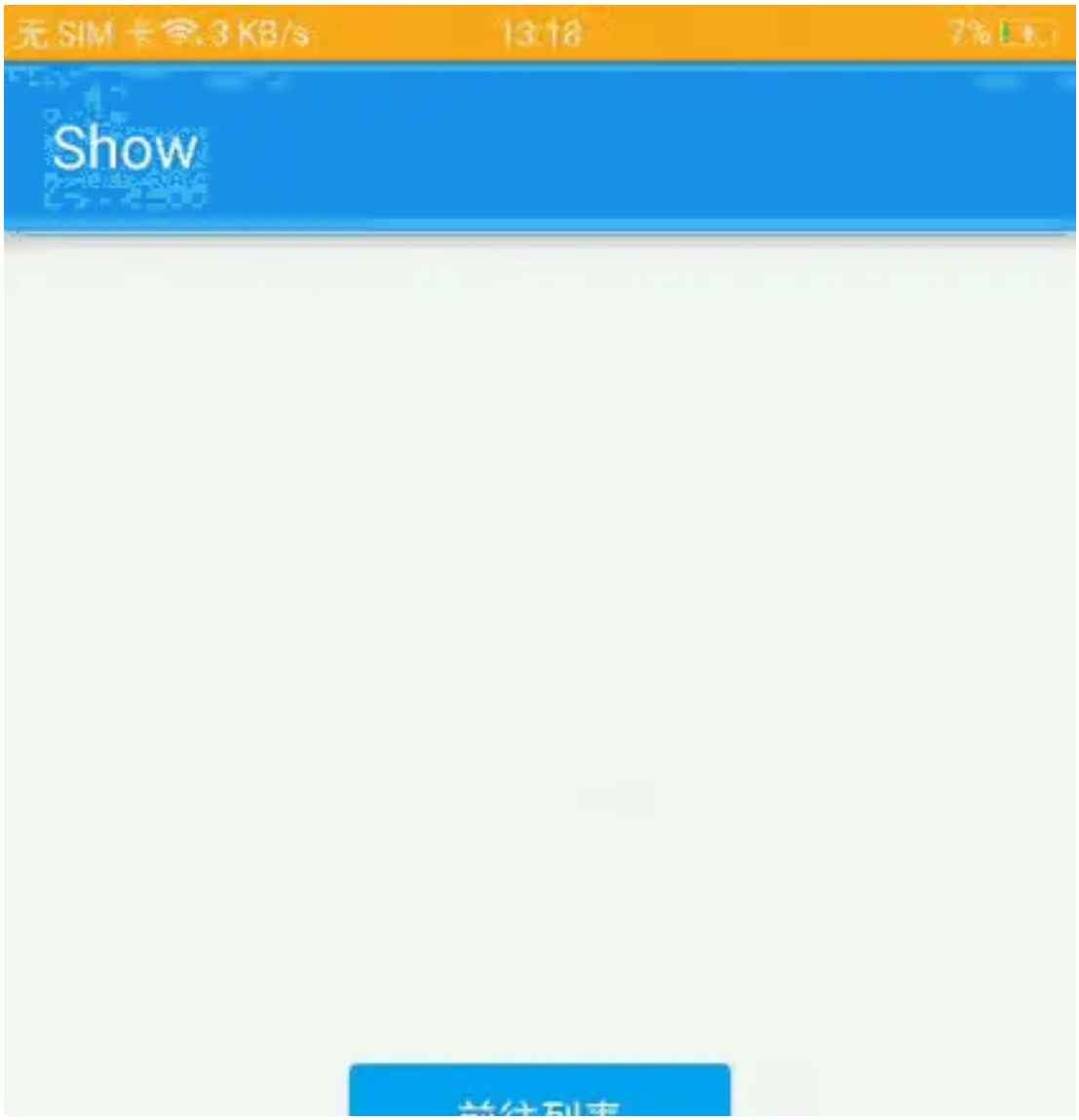
State，页面状态，可用于构建 UI。通常是 Bloc 将接收到的 Event 转化为 State。

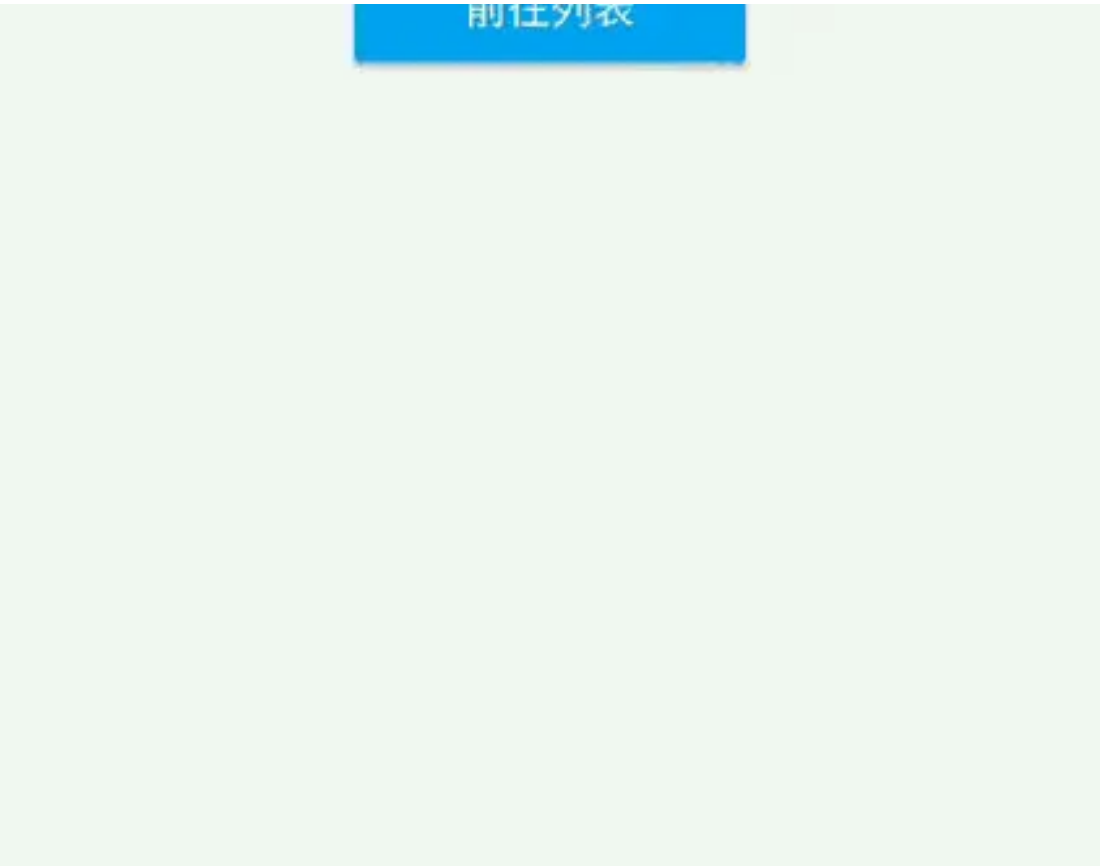
Bloc 架构的核心是 Bloc 类，Bloc 类是一个抽象类，有一个 `mapEventToState(event)` 方法需要实现。`mapEventToState(event)` 顾名思义，就是将用户点击 View 时发出的 event 转化为构建 UI 所用的 State。另外，在 StatefulWidget 中使用

bloc 的话，在 widget dispose 时，要调用 `bloc.dispose()`方法进行释放。

四、Bloc 的实践

这里以常见的获取列表选择列表为例子。一个页面用于展示选中项和跳转到列表，一个页面用于显示列表。





image

1. 引入 Redux 的第三方库

在 `pubspec.yaml` 文件中引入 `flutter_bloc` 第三方库支持 bloc 功能。

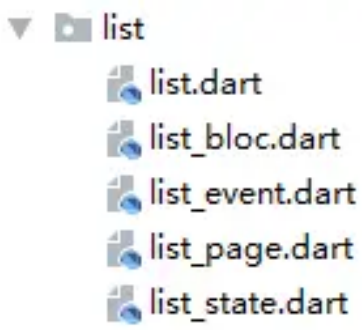
```
# 引入 bloc 第三方库
flutter_bloc: ^0.9.0
```

2. 使用 Bloc 插件

这一步可有可无，但使用插件会方便开发，不使用的话也没什么问题。

Bloc 官方提供了 VSCode 和 Android studio 的插件，方便生成 Bloc 框架用到的相关类。
下文以 Android studio 的插件为例。

比如 list 页面，该插件会生成相应的类



image

从生成的五个文件中也可以看到，`list_bloc` 负责承载业务逻辑，`list_page` 负责编写 UI 界面，`list_event` 和 `list_state` 分别是事件和状态，其中 `list.dart` 文件是用于导出前面四个文件的。

具体使用可见

Android studio 的 Bloc 插件

VSCode 的 Bloc 插件

3. 使用 BlocProvider 作为根布局

在 `main.dart` 中，使用 BlocProvider 作为父布局包裹，用于传递需要的 bloc。Demo 中包含两个页面，一个是展示页面 ShowPage，一个是列表页面 ListPage。

上面讲到，Bloc 的核心功能在于 Bloc 类，对于展示页面 ShowPage，会有一个 ShowBloc 继续自 Bloc 类。由于展示页面 ShowPage 会和列表页面 ListPage 有数据的互动，所以这里将 ShowBloc 保存在 BlocProvider 中进行传递。

```
@override
  Widget build(BuildContext context) {
    return BlocProvider(
        bloc: _showBloc,
        child: MaterialApp(
            title: 'Flutter Demo',
            theme: ThemeData(
              primarySwatch: Colors.blue,
            ),
            home: ShowPage()));
  }
```

4. 展示页面 ShowPage

① ShowEvent

列表的 item 点击后，需要发送一个 event 通知其它页面列表被选中，这里定义一个 SelectShowEvent 作为这种 event 通

知。

```
class SelectShowEvent extends ShowEvent {
  String selected;

  SelectShowEvent(this.selected);
}
```

② ShowState

State 用于表示一种界面状态，即一个 State 就对应一个界面。插件在一开始会生成一个默认状态，InitialShowState。我们可以使用 InitialShowState 来代表初始的界面。另外，我们自己定义一种状态，SelectedShowState，代表选中列表后的 State。

```
@immutable
abstract class ShowState {}

class InitialShowState extends ShowState {}

class SelectedShowState extends ShowState {
  String _selectedString = "";

  String get selected => _selectedString;

  SelectedShowState(this._selectedString);
}
```

③ ShowBloc

Bloc 的主要职责是接收 Event，然后把 Event 转化为对应的 State。这里的 ShowBloc 继续自 Bloc，需要重写实现抽象方法 `mapEventToState(event)`。在这个方法中，我们判断传过来的 event 是不是 SelectShowEvent，是则拿到 SelectShowEvent 中的 selected 变量去构建 SelectedShowState。`mapEventToState(event)`返回的是一个 Stream，我们通过 yield 关键字去返回一个 SelectedShowState。

```
class ShowBloc extends Bloc<ShowEvent, ShowState> {
  @override
  ShowState get initialState => InitialShowState();

  @override
  Stream<ShowState> mapEventToState(
    ShowEvent event,
  ) async* {
    if (event is SelectShowEvent) {
      yield SelectedShowState(event.selected);
    }
  }
}
```

④ ShowPage

在 ShowPage 的界面上，我们需要根据 showBloc 中是否有被选中的列表项目去展于页面，所以这里我们先使用使用`BlocProvider.of<ShowBloc>(context)`去拿到 showBloc，接着再用 BlocBuilder 根据 showBloc 构建界面。使用 BlocBuilder 的好处就是可以让页面自动响应 showBloc 的变化而变化。

```
var showBloc = BlocProvider.of<ShowBloc>(context);
...
BlocBuilder(
    bloc: showBloc,
    builder: (context, state) {
      if (state is SelectedShowState) {
        return Text(state.selected);
      }
      return Text("");
    }),
```

5. 列表页面 ListPage

① ListEvent

列表页面，我们一开始需要从网络中拉取列表数据，所以定义一个 FetchListEvent 事件在进入页面时通知 ListBloc 去获取列表。

```
@immutable
abstract class ListEvent extends Equatable {
```

```
  ListEvent([List props = const []]) : super(props);
}

class FetchListEvent extends ListEvent {}
```

② ListState

InitialListState 是插件默认生成的初始状态，另外定义一个 FetchListState 代表获取列表完成的状态。

```
@immutable
abstract class ListState extends Equatable {
  ListState([List props = const []]) : super(props);
}

class InitialListState extends ListState {}

class FetchListState extends ListState {

  List<String> _list = [];

  UnmodifiableListView<String> get list =>
UnmodifiableListView(_list);

  FetchListState(this._list);
}
```

③ ListBloc

在 ListBloc 中，进行从网络获取列表数据的业务。这里通过

一个延时操作摸拟网络请求，最后用 yield 返回列表数据。

```
class ListBloc extends Bloc<ListEvent, ListState> {
  @override
  ListState get initialState => InitialListState();

  @override
  Stream<ListState> mapEventToState(
    ListEvent event,
  ) async* {
    if (event is FetchListEvent) {
      // 模拟网络请求
      await Future.delayed(Duration(milliseconds: 2000));
      var list = [
        "1. Bloc artitechture",
        "2. Bloc artitechture",
        "3. Bloc artitechture",
        "4. Bloc artitechture",
        "5. Bloc artitechture",
        "6. Bloc artitechture",
        "7. Bloc artitechture",
        "8. Bloc artitechture",
        "9. Bloc artitechture",
        "10. Bloc artitechture"
      ];

      yield FetchListState(list);
    }
  }
}
```

④ ListPage

在列表页面初始化时有两个操作，一个是初始化 listBloc，一个是发出列表请求的 Event。

```
@override
void initState() {
  bloc = ListBloc(); // 初始化listBloc
  bloc.dispatch(FetchListEvent()); // 发出列表请求事件
  super.initState();
}
```

接下用，便是用 BlocBuilder 去响应状态。当 state 是 InitialListState，说明未获取列表，则显示 loading 界面，当 state 是 FetchListState 时，说明已经成功获取列表，显示列表界面。

```
body: BlocBuilder(
    bloc: bloc,
    builder: (context, state) {
      // 根据状态显示界面
      if (state is InitialListState) {
        // 显示 loading 界面
        return buildLoad();
      } else if (state is FetchListState) {
        // 显示列表界面
        var list = state.list;
        return buildList(list);
      }
```

```
    }));
```

最后，记得对 bloc 进行 `dispose()`。

```
@override
void dispose() {
  bloc.dispose();
  super.dispose();
}
```

具体代码可以到 github 查看。

总结

在 Bloc 的架构中，将一个页面和一个 Bloc 相结合，由页面产生 Event，Bloc 根据业务需要将 Event 转化为 State，再把 State 交给页面中的 BlocBuilder 构建 UI。Demo 中只是给出了简单的状态管理，实际项目中，比如网络请求，有请求中、请求成功、请求失败的多种状态，可以做适当封装使 Bloc 更加易用。相比于 Redux，Bloc 不需要将所有状态集中管理，这样对于不同模块的页面易于拆分，对于代码量比较大的客户端而言，Bloc 的架构会相对比较友好。