Flutter之Bloc模式

叶落竹影

Flutter之Bloc模式

全称Business Logic Component，业务逻辑组件

BLoC 是独立处理业务逻辑（网络数据请求、数据处理等等的逻辑），通过流**Stream**的Sinks，streams发布监听业务处理后的数据，只关心业务处理。而Widget着重业务数据处理后的结果显示。将业务逻辑和UI分离。

***widget*做*UI*展示，*bloc*做逻辑处理，*model*做数据封装。**

1.工作流程

组件通过Sink向Bloc发送事件，BLoC接收到事件后执行内部逻辑处理，并把处理的结果通过流的方式通知给订阅事件流的组件。在BLoC的工作流程中，Sink接受输入，BLoC则对接受的内容进行处理，最后再以流的方式输出。可以发现，BLoC又是一个典型的观察者模式。

2.使用

项目采取Bloc+RxDart来开发

2.1Bloc实现原理

Bloc触发刷新的方式是使用

StreamBuilder+StreamController。

Bloc另一重要特性就是跨层级获取bloc，在bloc可以获取到model或者调用方法。

其中使用了ancestorWidgetOfExactType，因为flutter widget是树状结构的，结构层次保存在BuildContext中，可以从子节点依次往父节点找符合类型的widget，所以所有使用了BlocProvider包裹的widget都可以通过of方法找到，再返回widget中的bloc。（**bloc最终是保存在外一层的state中**）。

```
final type = _typeOf<BlocProvider<T>>();
BlocProvider<T> provider =
context.ancestorWidgetOfExactType(type);
```

2.2RxDart实现原理

StreamController实现了观察者模式，监听者不是直接被调用，而是处于观察状态，当事件加入streamController后，监听者获得异步回调。

RxDart是对StreamController的扩展，提供了更多的模式。分为两个部分Subject和Observable。

其中Observable对stream封装后提供多个处理方法，例如map、expand、merge、every、contact，可以对数据进行相应的遍历、合并等操作。

Subject是对StreamController的扩展，常用的有以下几种。

- PublishSubject：StreamController广播版，streamController只能有一个listener，PublishSubject可以多次listen。下面的其他几种Subject也都是广播版。
- BehaviorSubject: 缓存最近一次的事件。如果先发生事件，后listen，也能收到缓存的事件。
- ReplaySubject: 缓存所有的事件，之前加入的所有event，listen后，都会顺序发送过来。

3.实例

1.创建一个所有Bloc的通用基类BlocBase

```
abstract class BlocBase{
  void dispose();
}
class BlocProvider<T extends BlocBase> extends
StatefulWidget{

  final T bloc;
  final Widget child;
  BlocProvider({
    Key key,
    @required this.child,
    @required this.bloc,
  }) : super(key: key);

  @override
```

```
  _BlocProviderState<T> createState() => new
_BlocProviderState<T>();

  static Type _typeOf<T>() => T;

  static T of<T extends BlocBase>(BuildContext
context){
    final type = _typeOf<BlocProvider<T>>();
    BlocProvider<T> provider =
context.findAncestorWidgetOfExactType();
    return provider.bloc;
  }
}

class _BlocProviderState<T> extends
State<BlocProvider<BlocBase>>{
  @override
  void dispose(){
    widget.bloc.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return widget.child;
  }
}
```

2.创建相对应的TestBloc，继承BlocBase。

Bloc 的初始化可以在**任意父组件、子组件**中，获取只需要使

用

```
BlocProvider.of(context)

TestBloc  _testBloc  =  BlocProvider.of(context);
```

2.1定义Subject

```
final _subject = PublishSubject<Model>();
```

2.2定义Stream

```
Stream<Model> get stream => _subject.stream;
```

2.3在业务逻辑内将数据添加到_subject中，业务逻辑不局限与IO流数据、网络请求数据等

```
void getData(){
...数据获取方法
_suject.sink.add(data)
}
```

需要注意，在TestBloc中的dispose中需要调用subject(streamController)close.

```
@override
void dispose() {
  _subject.close();
}
```

2.4在widget中通过StreamBuilder来控制。

```
 @override
```

```
  Widget build(BuildContext context) {
    // TODO: implement build
    return StreamBuilder(
          stream: stream,
          builder: (BuildContext context,
AsyncSnapshot<Price> snapshot) {
            ...widget展示数据
        }
  }
}
```

可以在最顶层父组件构建UI时设置BlocProvider(),具体的UI显示封装在page()中。

```
@override
 Widget build(BuildContext context) {
 return SafeArea(
     child: Scaffold(
     appBar:(AppBar...),
     body:BlocProvider(
         bloc: TestBloc(),
         child: page(),//显示的UI
       ),
     ),
   );
 }
```

4.综合

使用Bloc之后，数据与widget相关联，widget状态可以随

数据的改变而改变。与 setState() 不同的是，Bloc 可以局部刷新，只刷新与数据绑定的 widget，不用全局刷新，setState() 每次会全局刷新，提升了 UI 构建速度。