Flutter_Bloc 使用

贾震惊

bloc 是 flutter 开发中非常优秀的状态管理库，今天我们就来浅学下 bloc 的用法。

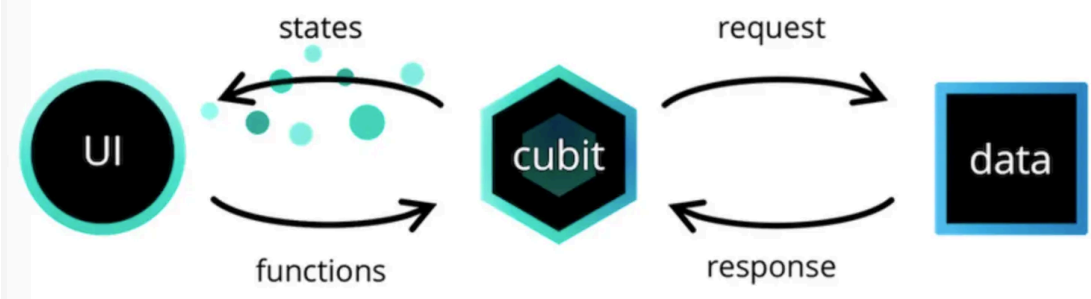
引入：

```
dependencies:
   flutter_bloc: ^8.0.0  //包含了bloc、provider库
```

bloc 可以通过 2 个类来管理任何类型的状态，Cubit 和 Bloc，它们都继承自 BlocBase 类。

Cubit

cubit 通过函数来触发 UI 状态改变



应用一张官方的图.jpg

- **创建 Cubit**

```
class CounterCubit extends Cubit<int> {
  CounterCubit(int initialState) : super(initialState);
}
```

1. 创建 cubit 需要定义管理的状态类型，这里是 int
2. 通过 super 指定初始状态，为了初始值更灵活可以通过外部值传入

- **状态变化**

```
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() => emit(state + 1);
}
```

1. Cubit 通过 emit 发出一个新状态
2. state 获取 Cubit 的当前状态
3. emit 函数是受到保护的，这也意味这它只能在 Cubit 内部使用

- **状态监听**

当 Cubit 发出新状态时，将有一个 改变发生。我们可以通过重写 onChange 方法来观察给定 Cubit 的所有变化。

```
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() => emit(state + 1);
```

```
  @override
  void onChange(Change<int> change) {
    super.onChange(change);
    print(change);
  }
}
```

1. 初始值是不会调用 onChange 方法的
2. 一个 Change 由 currentState 和 nextState 组成。

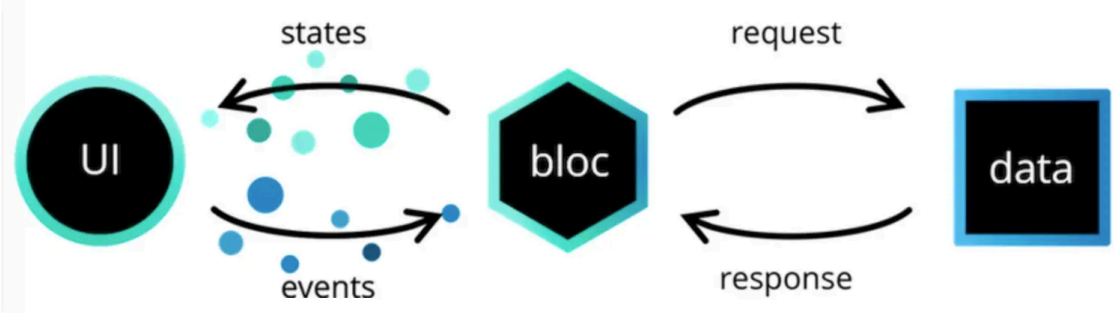
- **关闭 Cubit**

当我们不需要监听 Cubit 的状态时， 可以关闭

```
cubit.close();
```

Bloc

Bloc 和 Cubit 不同，Bloc 是通过事件来触发 UI 状态改变



来自官方的图.jpg

- **创建 Bloc**

创建一个 Bloc 类似于创建一个 Cubit，除了定义我们将要管理的状态和初始值外，我们还必须定义 Event 使其能够处理事件。

```
abstract class CounterEvent {}

class CounterIncrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0) {
    on<CounterIncrementPressed>((event, emit) {
       emit(state + 1);
    })
  }
}
```

1. 通过 on<Event> 注册事件
2. 同 Cubit 一样 emit 只能在 Bloc 内部使用

- **状态变化**

```
final bloc = CounterBloc();
bloc.add(CounterIncrementPressed());
await bloc.close();
```

通过 add 事件来触发状态改变，当需要关闭状态流时 调用 close 方法。

- **状态监听**

和 Cubit 一样，扩展了 BlocBase ,通过 onChange 方法观察 Bloc 的所以状态

```
abstract class CounterEvent {}

class CounterIncrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0) {
    on<CounterIncrementPressed>((event, emit) => emit(state
+ 1));
  }

  @override
  void onChange(Change<int> change) {
    super.onChange(change);
    print(change);
  }
}
```

- **状态转换**

Bloc 和 Cubit 之间的主要区别在于 onTransition ，由于 Bloc 是事件驱动的，因此我们也能够捕获有关触发状态更改的信息。

```
abstract class CounterEvent {}

class CounterIncrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0) {
    on<CounterIncrementPressed>((event, emit) => emit(state
+ 1));
  }

  @override
  void onChange(Change<int> change) {
    super.onChange(change);
    print(change);
  }

  @override
  void onTransition(Transition<CounterEvent, int>
transition) {
    super.onTransition(transition);
    print(transition);  // { currentState: 0, event:
CounterIncrementPressed, nextState: 1 }
  }
}
```

1. onTransition 在 onChange 之前被调用

2. 从一种状态到另一种状态的转换称为 Transition。Transition 由当前状态，事件和下一个状态组成

注意

1. 我们通过源码发现，通过 emit 发送新的状态时，只有当旧的 state 和新的 state 不等时 才会触发 onChange 。当第一次通过 emit 发送的 state 和初始 state 相等时，是会调用 onChange 方法的。
2. onTransition 和 onChange 一样

```
  @protected
  @visibleForTesting
  @override
  void emit(State state) {
    try {
      if (isClosed) {
        throw StateError('Cannot emit new states after
calling close');
      }
      if (state == _state && _emitted) return;
      onChange(Change<State>(currentState: this.state,
nextState: state));
      _state = state;
      _stateController.add(_state);
      _emitted = true;
    } catch (error, stackTrace) {
      onError(error, stackTrace);
      rethrow;
    }
```

```
  }
```

```
        if (this.state == state && _emitted) return;
        onTransition(Transition(
          currentState: this.state,
          event: event as E,
          nextState: state,
        ));
        emit(state);
      }
```

接下来我们学习 Bloc 相关的 Flutter 组件，往下看

Bloc Widgets

bloc widgets 都有集成 Cubit 和 Bloc ， 常用的组件有 BlocProvider、BlocBuilder、 BlocSelector、BlocListener、BlocConsumer

- BlocProvider

```
BlocProvider(
  lazy: false,
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

1. BlocProvider 创建 bloc ， 其子级可通过 BlocProvider.of <T>（context) 获取 bloc , 在这种情况下， 由于 BlocProvider 负责创建 bloc，它将自动处理

关闭 bloc。

2. 默认情况下 create 将在 BlocProvider.of <T>（context）查找 bloc 时执行， 通过查看源码发现，Bloc 组件 (如 BlocBuilder，其他组件类似）通过 BlocProvider 与 context 获取 bloc 时，initState 方法已经 调用了 context.read<B>() ， Provider.of<T>(context)，说明当创建 BlocBuilder 组件时，create 方法执行。如果想 create 立即执行，lazy 可以设置为 false 。

```
class _BlocBuilderBaseState<B extends StateStreamable<S>,
S>
    extends State<BlocBuilderBase<B, S>> {
  late B _bloc;
  late S _state;

  @override
  void initState() {
    super.initState();
    _bloc = widget.bloc ?? context.read<B>();
    _state = _bloc.state;
  }
```

3. 当需要将现有的bloc提供给新路线（非 BlocProvider 的子组件）时，可以通过 context.read<BlocA>() , 在这种情况下，BlocProvider 不会自动关闭该 bloc，因为它没有创建它。

MultiBlocProvider 将多个 BlocProvider 组件合并为一个。MultiBlocProvider 提高了可读性，并且消除了嵌套多个

BlocProviders 的需要。

```
MultiBlocProvider(
  providers: [
    BlocProvider<BlocA>(
      create: (BuildContext context) => BlocA(),
    ),
    BlocProvider<BlocB>(
      create: (BuildContext context) => BlocB(),
    ),
    BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
    ),
  ],
  child: ChildA(),
)
```

- BlocBuilder

  BlocBuilder 它需要 bloc 和 builder 两个方法。

  BlocBuilder 在接收到新的状态 (State) 时处理 builder 组件。如果省略了 bloc 中的参数，则 BlocBuilder 将使用 BlocProvider和当前的 BuildContext 自动执行查找。

```
BlocBuilder<BlocA, BlocAState>(
  bloc: blocA, // provide the local bloc instance
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

如果您希望对何时调用 builder 函数的时间进行十分缜密的

控制，可以通过 buildWhen返回值控制。buildWhen 获取先前的 Bloc 的 state 和当前的 state 并返回 bool 值。如果返回 true，则会调用 builder 使用当前 state 重新构建，如果返回 false，则不会调用builder，也不会进行重建。

```
BlocBuilder<BlocA, BlocAState>(
  buildWhen: (previousState, state) {
    // return true/false to determine whether or not
    // to rebuild the widget with state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

- BlocSelector

  BlocSelector 是一个和 BlocBuilder 类似的组件，但它可以允许开发者选择一个基于当前 bloc 状态的新值来过滤更新。通常通过 State 类某个特定的属性做为 selector，如果所选值不更改，则会阻止不必要的构建。选中的值必须是不可变的，以便 BlocSelector 准确地判断是否应该再次调用 builder。

```
BlocSelector<BlocA, BlocAState, SelectedState>(
  selector: (state) {
    return state.parameterA;
  },
```

```
  builder: (context, state) {
    // return widget here based on the selected state.
  },
)
```

- BlocListener

  每次状态变化都会调用一次 listener ， 不同于 BlocBuilder 中的 builder , listener 状态更改不包括 initialState，也就是初始状态不会调用 listener 方法，listener 是 void 函数。

```
BlocListener<BlocA, BlocAState>(
  bloc: blocA,  //省略了bloc参数，则BlocListener将使用BlocProvider和当前的BuildContext自动执行查找。
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container()
)
```

如果您希望对何时调用 listener 函数的时间进行十分缜密的控制，可以通过 listenWhen 返回值控制。listenWhen 获取先前的 Bloc 的 state 和当前的 state 并返回 bool 值。如果返回 true，listener 将被调用。如果条件返回 false，则不会使用状态调用 listener。

```
BlocListener<BlocA, BlocAState>(
  listenWhen: (previousState, state) {
```

```
    // return true/false to determine whether or not
    // to call listener with state
  },
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

- BlocConsumer

  BlocConsumer 公开一个 builder 和 listener 以便对新 State 做出反应。BlocConsumer与 BlocListener + BlocBuilder 类似

```
BlocConsumer<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

可以实现可选的 listenWhen 和 buildWhen ， 以更精细地控制何时调用 listener 和 builder。在每次 bloc 状态 (State) 改变时，都会调用 listenWhen 和 buildWhen 。同 BlocListener initialState 不会调用 listener。

```
BlocConsumer<BlocA, BlocAState>(
  listenWhen: (previous, current) {
```

```
    // return true/false to determine whether or not
    // to invoke listener with state
  },
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  buildWhen: (previous, current) {
    // return true/false to determine whether or not
    // to rebuild the widget with state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

介绍2个Multi组件，整个APP内 页面之间可以共享，这就有点像 Redux 了。

- MultiBlocProvider

  1、将多个 BlocProvider 部件合并为一个。

  2、MaterialApp 为 child，APP 页面之间公用 bloc ，通过 context.read<BlocA>() 或者 BlocProvider.of<BlocA>(context) 获取 bloc。

```
MultiBlocProvider(
  providers: [
    BlocProvider<BlocA>(
      create: (BuildContext context) => BlocA(),
    ),
    BlocProvider<BlocB>(
      create: (BuildContext context) => BlocB(),
```

```
    ),
    BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
    ),
  ],
  child: MaterialApp(
        title: 'Flutter Demo',
        theme: ThemeData(
        home: Child(),
      ),
)
```

- MultiRepositoryProvider

  1、将多个 RepositoryProvider （向子级提供存储库）部件合并为一个。

  2、MaterialApp 为 child，APP 页面之间共享 RepositoryProvider ， 通过 context.read<RepositoryA>() 或者 RepositoryProvider.of<RepositoryA>(context) 获取 RepositoryProvider。

```
MultiRepositoryProvider(
  providers: [
    RepositoryProvider<RepositoryA>(
      create: (context) => RepositoryA(),
    ),
    RepositoryProvider<RepositoryB>(
      create: (context) => RepositoryB(),
    ),
    RepositoryProvider<RepositoryC>(
      create: (context) => RepositoryC(),
```

```
    ),
  ],
  child: MaterialApp(
        title: 'Flutter Demo',
        theme: ThemeData(
        home: Child(),
      ),
)
```