Flutter状态管理学习手册 (二)——Redux

WinDin

上一篇讲到了一个简单的状态管理架构—— ScopedModel，当然，这种简单的架构会用在商业项目中的概率比较小，本篇则讲述另一个架构： Redux ，一个优雅且实用的状态管理框架。本篇 Demo 地址：https://github.com/windinwork/flutter_redux_app

一、Redux 的准备工作

Redux 的概念源于 React，对于不是从事前端工作或者没有接触过 React 的人要理解 Redux 会比较繁复。对于不了解 Redux 的小伙伴，这里有两篇很不错的文章介绍了 Redux 的概念和相关知识：

Redux 入门教程（一）：基本用法

Redux 入门教程（二）：中间件与异步操作

二、Redux的概念

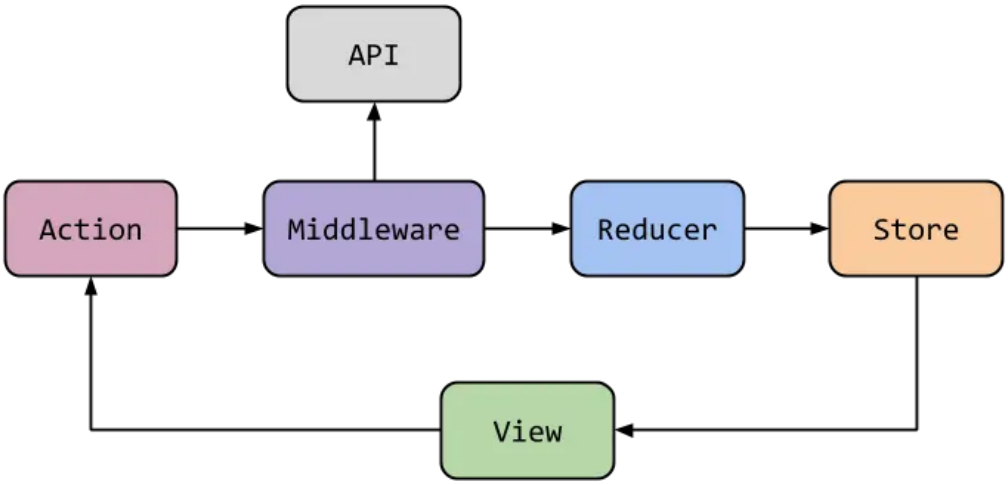
学习后 Redux 可以了解到，Redux 主要由涉及下面几种概念：

- Store，是保存数据的地方。整个应用只能有一个

Store 。Store 有十分重要的方法 dispatch(action) 来发送 Action。

- State，是某个时间点的数据快照， 一个 State 对应一个 View。只要 State 相同，View 就相同。
- Action，是 View 发出的通知，通过 Reducer 使 State 发生变化。
- Reducer，是一个纯函数，接受 Action 和当前 State 作为参数，返回一个新的 State。
- Middleware，中间件，它的操作在发出 Action 和执行 Reducer 这两步之间发生，用于增加额外功能，如处理异步操作或者打印日志功能等。

Redux 的工作流程如图所示，先用户发出 Action ， Store 自动给 Middleware 进行处理，再传递给 Reducer ， Reducer 会返回新的 State ，通过 Store 触发重新渲染 View。这里的 View 在 Flutter 中以 Widget 的形式存在。



以上这部分是 Redux 的内容，比较容易理解。

三、Flutter 中 Redux 的使用

在 Flutter 中，我们除了引入 redux 第三方库之外，还要引入 flutter_redux 第三方库 ，并且，为了对异步操作有更好的支持，还要引入 redux_thunk 库作为 Middleware ， 对 thunk action 有更好的支持。

下面来了解 flutter_redux 中的概念。

1. StoreProvider ， 是一个基础 Widget ， 一般在应用的入口处作为父布局使得，用于传递 Store 。

2. StoreConnector ， 一个可以获取 Store 的 Widget ， 作

用是响应 Store 发出的状态改变事件来重建 UI。

- `StoreConnector` 中有两个标记为@required 的参数，一个是 converter ， converter 用于将 store 中的 state 转化为 viewModel，另一个是 builder，builder 的作用是将 viewModel 进一步转化为 UI 布局。

- `StoreConnector` 有一个值得一提的函数：onInit，在这个函数中可以执行初始化操作。

3. 对于异步操作，我们引入了 redux_thunk 库，redux_thunk 库是一个中间件，它会处理 thunkAction。thunkAction 是只包含一个 Store 类型参数的函数。

通过 StoreProvider 和 StoreConnector 就可以在 Flutter 实现 Redux 的功能了，接下来是具体实践。

四、Redux 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。

1. 引入 Redux 的第三方库

```
  redux: ^3.0.0
  flutter_redux: ^0.5.3
  redux_thunk: ^0.2.1
```

2. 创建 Store

Store 全局只有一个，这里封装一个创建 Store 的方法。创建Store时，传入 Reducer ，初始化的 State 和 Middleware 。

```
Store<AppState> createStore() {
  return Store(appReducer, initialState:
AppState.initial(), middleware: [
    // 引入 thunk action 的中间件
    thunkMiddleware
  ]);
}
```

3. 创建 State

State 状态，需要创建一个 `AppState`，作为整个应用的 state，除此之外，根据不同的页面可以有不同的 State，比如有一个列表页面 `ListPage`，就可以有一个列表状态 `ListState`，并且 `ListState` 会放在 `AppState` 中作为成员变量进行管理。

```
// app state
class AppState {
  ListState listState;

  AppState(this.listState);
}
```

```
// list state
class ListState {
  bool _init = false; // 列表初始化标志
  List<String> _list = []; // 列表数据
  String _selected = '未选中'; // 选中的列表项

  ListState(this._init, this._list, this._selected);
}
```

4. 创建 Action

Action 是 Widget 在用户操作后发出的通知。对于 `ListPage` 页面，创建一个 `list_action.dart` 文件，用于存在可发出的 Action 。在示例中，需要用到两个 Action 。一个是加载完成

列表数据后发出的 Action ，用于刷新列表，一个是选中列表项时发出的 Action 。

```
/**
 * 加载完成列表数据
 */
class FetchListAction {
  List<String> list;

  FetchListAction(this.list);
}

/**
 * 选择列表
 */
class SelectItemAction {
  String selected;

  SelectItemAction(this.selected);
}
```

5. 创建 Reducer

Reducer 是一个带 State 和 Action 两个参数的纯函数，在这里，只需要关心根据传入的 Action 和旧的 State，返回一个新的 State。

和 State 一样，reducer 也分为应用的 `app_reducer` 和列表页

面的 `list_reducer`。

Reducer是纯函数。在示例中通过 `app_reducer` 返回一个新的 `AppState`，通过 `list_reducer` 分别对 `FetchListAction` 和 `SelectItemAction` 进行处理返回一个新的 `ListState`。

```
AppState appReducer(AppState state, dynamic action) {
  return AppState(listReducer(state.listState, action));
}
```

```
ListState listReducer(ListState pre, dynamic action) {
  if (action is FetchListAction) {
    return ListState(true, action.list, pre.selected);
  }
  if (action is SelectItemAction) {
    return ListState(pre.isInit, pre.list,
action.selected);
  }
  return pre;
}
```

6. 使用 Thunk Action

在创建 Store 时引入了 `thunkMiddleware`，使得项目支持 thunk action。

项目中需要一个加载列表的异步操作，可通过 thunk action 实现具体操作，注意 `ThunkAction` 是只有一个 `store` 参数的函

数。

```
ThunkAction<AppState> fetchList = (Store<AppState> store)
async {
  // 模拟网络请求
  await Future.delayed(Duration(milliseconds: 3000));
  var list = [
    "1. Redux State Management",
    "2. Redux State Management",
    "3. Redux State Management",
    "4. Redux State Management",
    "5. Redux State Management",
    "6. Redux State Management",
    "7. Redux State Management",
    "8. Redux State Management",
    "9. Redux State Management",
    "10. Redux State Management"
  ];

  store.dispatch(FetchListAction(list));
};
```

7. UI 布局

通过以上代码，我们做好了 Redux 的准备工作。接下来便可以布局页面。

① 在 `main.dart` 中，要做的工作是创建 Store，和使用 StoreProider 作为根布局。

```
    // 创建Store
    var store = createStore();
    // 使用StoreProvider作为根布局
    return StoreProvider<AppState>(
        store: store,
        child: MaterialApp(
          title: 'Flutter Demo',
          theme: ThemeData(
            primarySwatch: Colors.blue,
          ),
          home: ShowPage(),
        ));
  }
```

② `show_page.dart`，用于显示选中的列表项和提供跳转到 `ListPage` 的按钮入口。

`show_page.dart` 中，选中的列表项的数据来自于 Store，所以这里使用 StoreConnector 来根据数据构建 UI 界面。

```
StoreConnector<AppState, String>(
  converter: (store) => store.state.listState.selected,
  builder: (context, selected) {
    return Text(selected);
  },
),
```

从上面可见，StoreConnector 需要两个泛型，一个是我们创建的 AppState，另一个是 ViewModel，这里我们直接将 String 作为 ViewModel。

StoreConnector 要定义两个函数，一个是 converter，从 Store 中拿出选中的列表项数据 store.state.listState.selected，另一个是 builder，将 converter 返回的 selected 进一步转化为界面：Text(selected)。

这就是 StoreConnector 的用法。

③ list_page.dart

`list_page.dart` 中的内容比较重点，里面实现了加载列表和点击列表的功能。

`list_page.dart` 中一开始要显示加载中的界面，等待数据成功加载后显示列表界面。这里也是用 StoreConnector 构建 UI。StoreConnector 将 AppState 转换为 ListState，通过 ListState 判断当前显示 loading 界面还是列表界面。

另外，在 StoreConnector 的 onInit 函数中执行加载列表的操作。

由于加载列表是一个异步操作，所以要用到之前定义的 `fetchList` 的 thunk action，这里通过 `store.dispatch(fetchList)` 来执行。

```
StoreConnector<AppState, ListState>(
```

```
    // 初始化时加载列表
    onInit: (store) {
  if (!store.state.listState.isInit) {
    store.dispatch(fetchList);
  }
  // 将store的state转化为viewModel
}, converter: (store) {
  return store.state.listState;
  // 通过viewModel更新界面
}, builder: (contxet, state) {
  // 根据状态显示界面
  if (!state.isInit) {
    // 显示loading界面
    return buildLoad();
  } else {
    // 显示列表界面
    var list = state.list;
    return buildList(list);
  }
})
```

另外，点击列表时也要发出一个选中列表项目的 Action ，即之前定义的 `SelectItemAction` 。

```
var store = StoreProvider.of<AppState>(context);
store.dispatch(SelectItemAction(list[index]));
// 返回到上一级页面
Navigator.pop(context);
```

`store.dispatch(action)` 发出的 Action 经过 Middleware 和 Reducer 的处理后，转变成 State，StoreConnector 就会自

动地根据 State 重建 UI。

这样，一个使用 Redux 架构的应用就基本成型了。完整代码可以参考 https://github.com/windinwork/flutter_redux_app

五、总结

在 Redux 中，数据总是"单向流动"的，保证了流程的清晰。相对于 ScopedModel 的架构， Redux 无疑更加规范，更易于开发和维护。只是， Redux 会相对地增加复杂性，所以，简单小型的项目可以不需要去考虑引进 Redux 这种架构，但对于大型的 Flutter 项目来说， Redux 就十分有用的架构。

六、思考

由于 Redux 只有一个应用只有一个 Store ，所以应用需要对整个 AppState 进行维护，在大型客户端代码迭代过程中，会和组件化和单独模块编译运行这种需求相矛盾，这是否会阻碍 Redux 成为大型项目采用的主要架构的因素呢?

参考目录

Flutter Redux Thunk, an example finally.

Introduction to Redux in Flutter

官网推荐 Sample：inKino