Flutter 状态管理

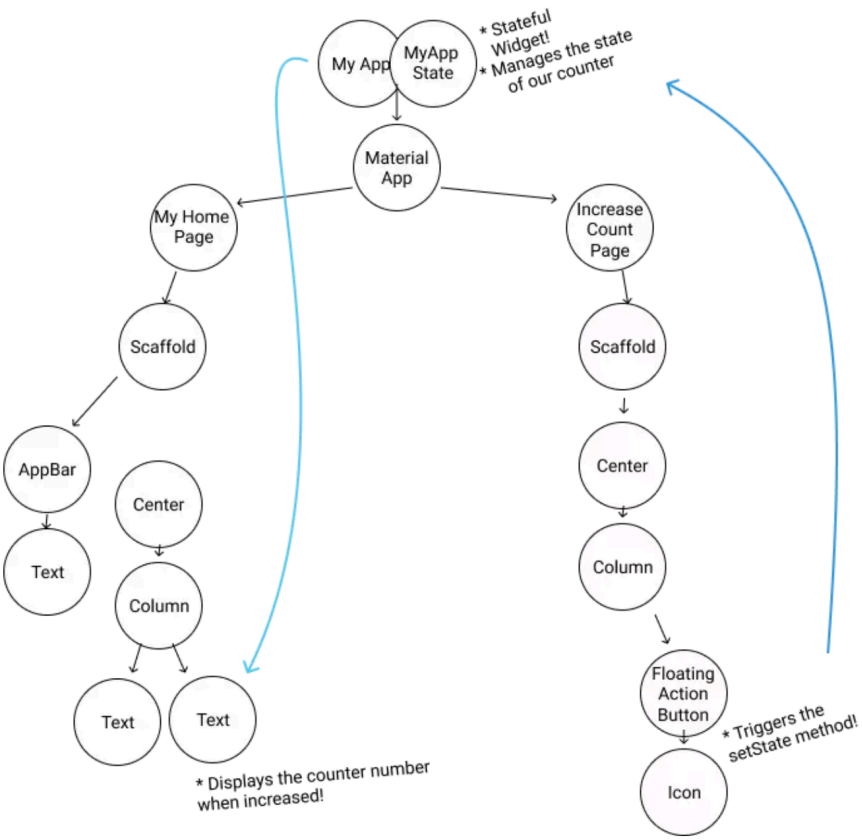
1. 为什么需要状态管理

1.1 StatefulWidget与StatelessWidget区别

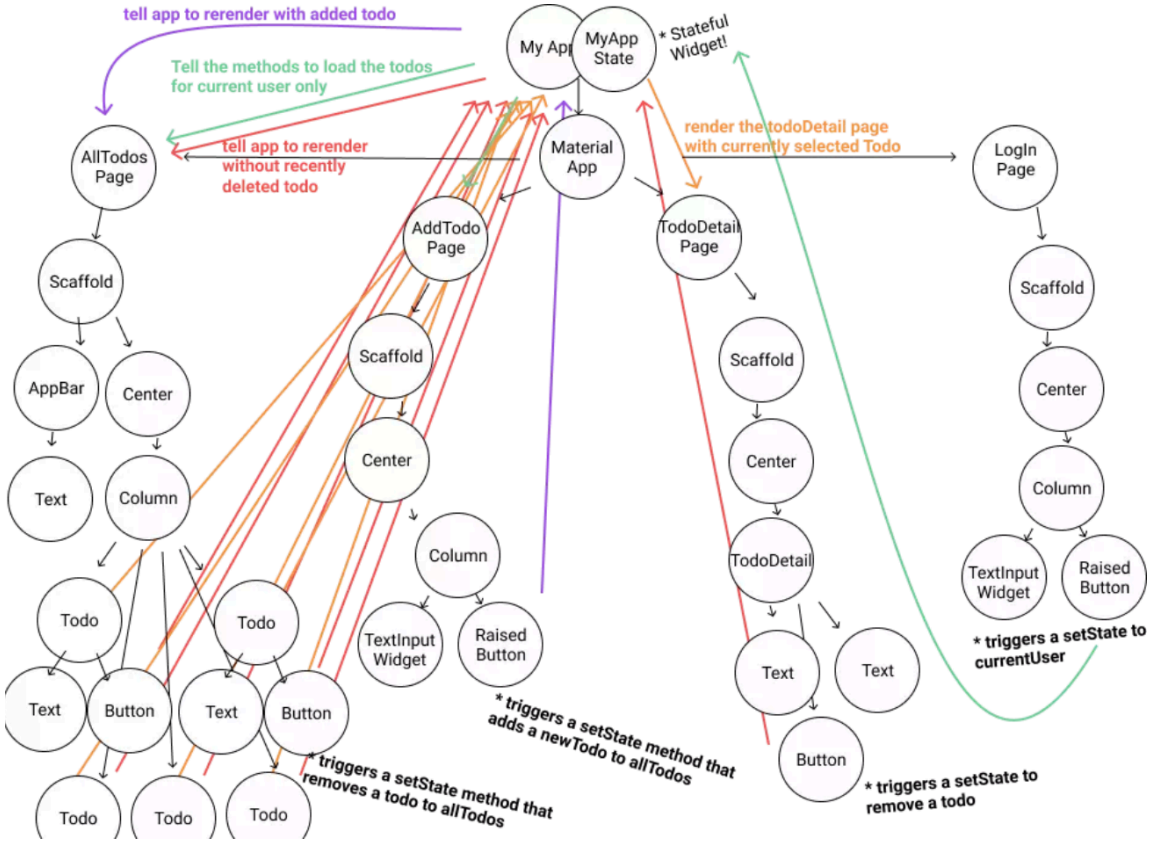
StatelessWidget初始化之后就无法改变，如果想改变，那便需要重新创建，new另一个StatelessWidget进行替换。但StatelessWidget因为是静态的，他没有办法重新创建自己。所以StatefulWidget便提供了这样的机制，通过调用setState((){})标记自身为dirty状态，以等待下一次系统的重绘检查。

1.2 StatefulWidget 动态化代价

我们改变状态的时候setState一下就可以了。 在我们一开始构建应用的时候，也许很简单，我们这时候可能并不需要状态管理。



但是随着功能的增加，你的应用程序将会有几十个甚至上百个状态。这个时候你的应用应该会是这样。



在State类中的调用setState((){})更新视图，将会触发State.build！，也将间接的触发其每个子Widget的构造方法以及build方法。 这意味这什么呢？ 如果你的根布局是一个StatefulWidget，那么每在根State中调用

一次setState((){})，都将是一次整页所有Widget的rebuild

1.3 如何选择StatefulWidget与StatelessWidget

- 优先使用 StatelessWidget
- 含有大量子 Widget（如根布局、次根布局）慎用 StatefulWidget
- 尽量在叶子节点使用 StatefulWidget
- 将会调用到setState((){}) 的代码尽可能的和要更新的视图封装在一个尽可能小的模块里。

2.如何进行状态管理

2.1 状态管理的几种方式

- setState
- InheritedWidget & Scoped Model
- StreamBuilder, RxDart
- Provider
- Redux
- MobX

2.2 状态管理的使用

2.2.1 InheritedWidget

`InheritedWidget` 是Flutter中非常重要的一个功能型 `Widget`，它可以高效的将数据在Widget树中向下传递、共享。我们可以在 `InheritedWidget` 为根节点的树下任一Widget中调用。

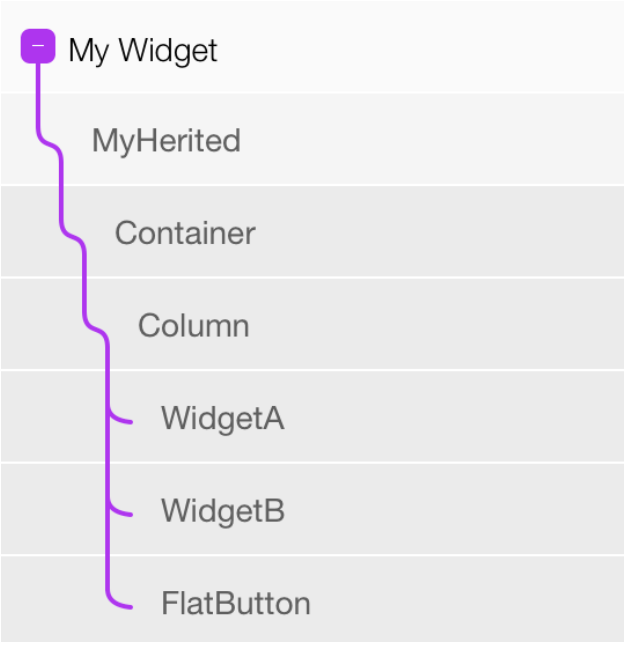
模板

```
class MyInherited extends InheritedWidget {
  final int data;

  MyInherited({this.data, Widget child}) : super(child : child) {
    print('MyInherited construct');
  }

  @override
  bool updateShouldNotify(MyInherited oldWidget) {
    bool result = oldWidget.data != this.data;
    print('MyInherited updateShouldNotify result = $result');
    return result;
  }

  static MyInherited of(BuildContext context) {
    return context.inheritFromWidgetOfExactType(MyInherited);
  }
}
```

BuildContext.inheritFromWidgetOfExactType 来获取离其最近的 InheritedWidget 实例。这在一些需要在Widget树中共享数据的场景中非常方便，如Flutter中，正是通过 InheritedWidget来共享应用主题(Theme)和Locale(当前语言环境)信息的。

```
Localizations.of<l10n>(context, l10n).string_name 获取字符串
MediaQuery.of(context).padding.top 获取状态栏高度
```

> Demo组件层级示例

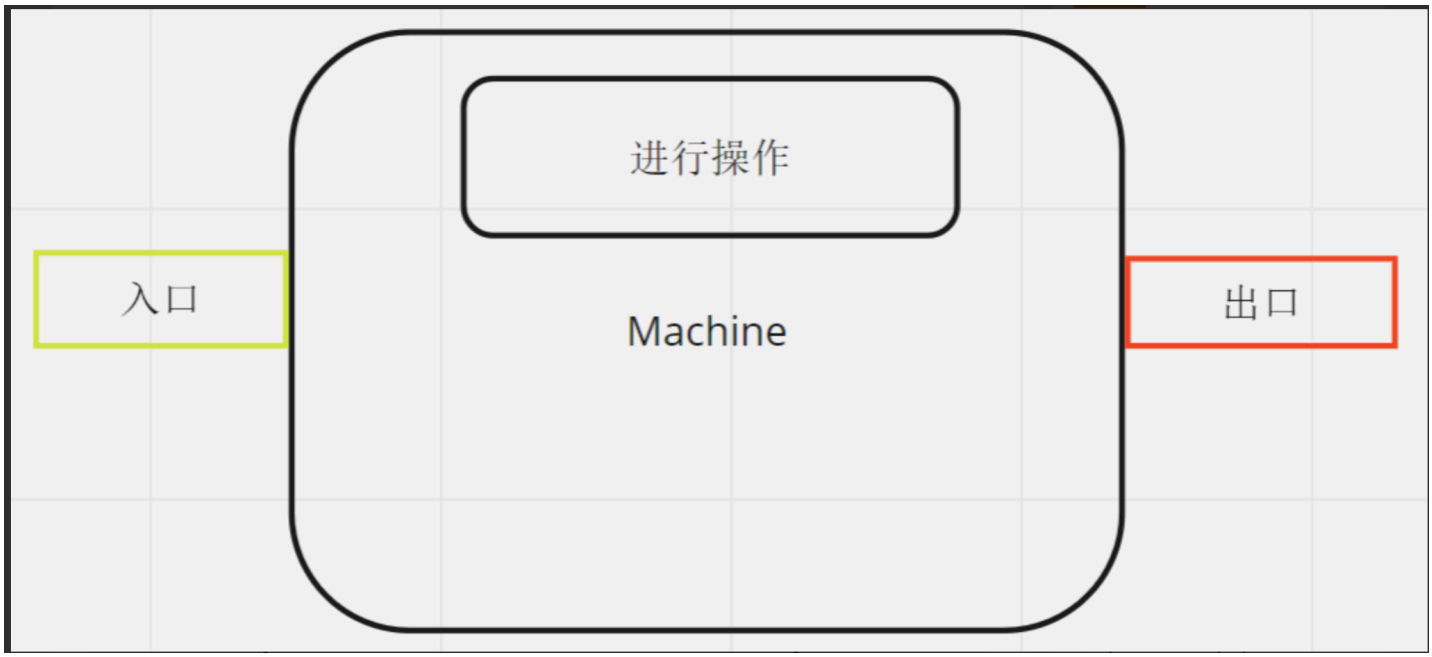


2.2.2 Scoped Model

Dart 的一个第三方库scoped_model。Scoped model使用了观察者模式，将数据模型放在父代，后代通过找到父代的model进行数据渲染，最后数据改变时将数据传回，父代再通知所有用到了该model的子代去更新状态。

2.2.3 Stream & RxDart

1. 概念



1. 我们可以简单把Stream想象为一个有两个端口的管道，只有其中的一个允许插入一些东西。 当您将某物插入管道时，它会在管道内流动并从另一端流出。
2. 所有类型以及任何类型。 从值，事件，对象，集合，映射，错误或甚至另一个流，任何类型的数据都可以由Stream传递 。

在整个过程中，时间都是一个不确定因素，我们随时都可以向这个机器的入口放东西进去，放进去了以后机器进行处理，但是我们并不知道它多久处理完。所以出口是需要专门派人盯着的，等待机器流出东西来。整个过程都是以异步的眼光来看的。

2. StreamController

1. `StreamController`，它是创建流的方式之一。
2. `StreamController` 有一个入口，叫做 `sink`
3. `sink` 可以使用 `add` 方法放东西进来，放进去以后就不再关心了。
4. `StreamController` 有一个出口，叫做 `stream`
5. 多个物品被放进来了之后，它不会打乱顺序，而是先入先出。

入口

```
//任意类型的流
StreamController controller = StreamController();
controller.sink.add(123);
controller.sink.add("xyz");
controller.sink.add(Anything);

//创建一条处理int类型的流
StreamController<int> numController = StreamController();
numController.sink.add(123);
```

出口

```
StreamController controller = StreamController();

//监听这个流的出口，当有data流出时，打印这个data
StreamSubscription subscription =
controller.stream.listen((data)=>print("$data"));

controller.sink.add(123);
```

> Is a Stream only a simple pipe?
>
> No, a Stream also allows to process the data that flows inside it before it goes out.
>
> StreamTransformer

Stream不止是一个简单的pipe，他有很多操作符可以在数据输出之前对数据通过进行变换`(StreamTransformer)`，这里不做过多介绍

知道了流的概念，那么它是如何结合`Flutter`进行状态管理的呢？ 答案就是`StreamBuilder`

3. StreamBuilder

StreamBuilder其实是一个StatefulWidget，它通过监听Stream，发现有数据输出时，自动重建，调用builder方法。

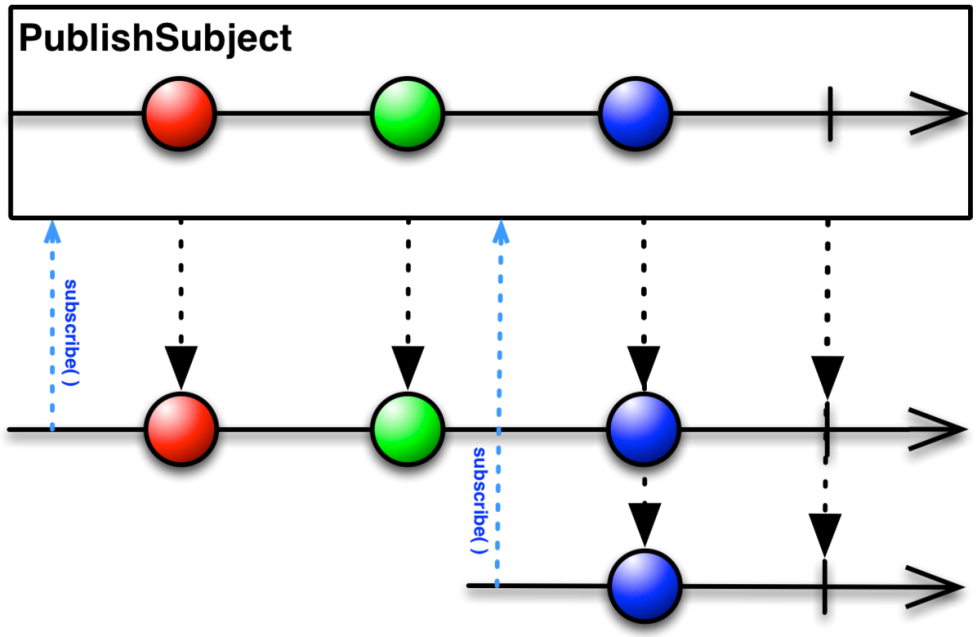
```
StreamBuilder<T>(
    stream: ...需要监听的stream...
    initialData: ...初始数据，否则为空...
    builder: (BuildContext context, AsyncSnapshot<T> snapshot){
        if (snapshot.hasData){
            return ...基于snapshot.hasData返回的控件
        }
        return ...没有数据的时候返回的控件
    },
)
```

StreamController需要在控件dispose()的时候被释放。

4. RxDart

`RxDart` 扩展了原始的 `Stream` 并提供了3种 `StreamController` 主要的变体

- **`PublishSubject`**



- **`BehaviorSubject`**
- **`ReplaySubject`**

2.2.4 Provider

开源社区创建，Google 2019推荐

1. **`ChangeNotifier`**

```
class Counter with ChangeNotifier {
  int _counter;

  Counter(this._counter);

  getCounter() => _counter;
  setCounter(int counter) => _counter = counter;

  void increment() {
    _counter++;
    notifyListeners();
  }

  void decrement() {
    _counter--;
    notifyListeners();
  }
}
```

> notifyListeners(); **通知更新UI**

2. `ChangeNotifierProvider`

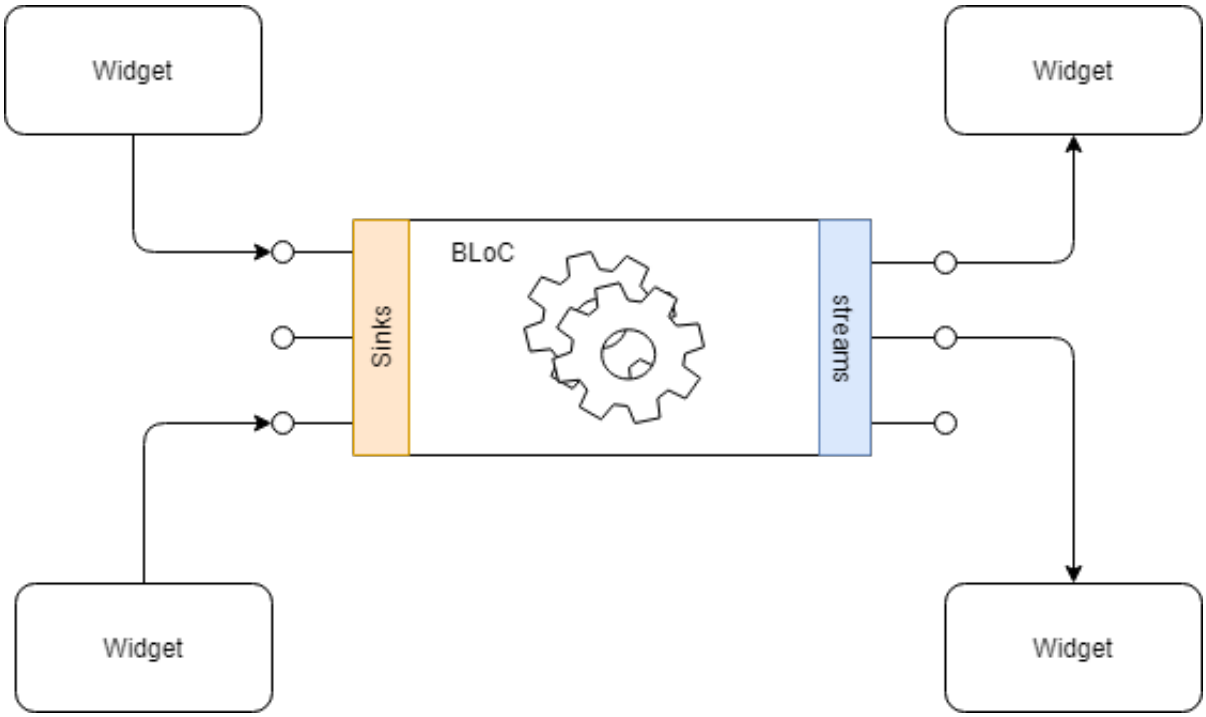
```
ChangeNotifierProvider<Counter>(
    builder: (_) => Counter(0),
    child: HomePage(),
)
```

3. 两种方式访问数据

- `Provider.of<Object>(context)`
- `Consumer Widget`

3.Bloc

BLoC(Business Logic Component)代表业务逻辑组件。它是一种模式(patten)或者说是一种架构(Architecture)



- Widgets通过Sinks向BLoC发送事件，
- BLoC通过Stream通知Widgets，
- 由BLoC实现的业务逻辑不是他们关注的问题。

1. BloC实现了责任分离，将整个业务集中在单独的Bloc类中
2. 可测试性
3. 自由组织布局，页面独立于业务逻辑，只要监听 `stream` 即可
4. 减少 `build` 次数

4.其他

1. 代码模板 https://github.com/AweiLoveAndroid/Flutter-learning