Flutter完整开发实战详解(十一、全面深入理解Stream)

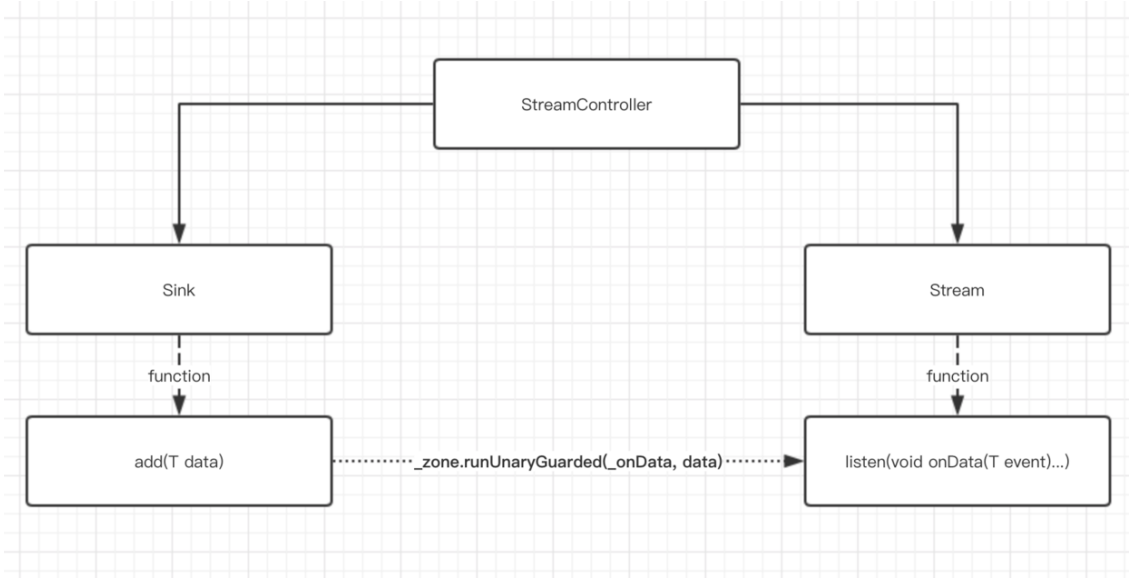
bug樱樱

一、Stream 由浅入深

`Stream` 在 Flutter 是属于非常关键的概念，在 Flutter 中，状态管理除了 `InheritedWidget` 之外，无论 `rxdart`，`Bloc` 模式，`flutter_redux` ，`fish_redux` 都离不开 `Stream` 的封装，而事实上 `Stream` 并不是 Flutter 中特有的，而是 Dart 中自带的逻辑。

通俗来说，`Stream` 就是事件流或者管道，事件流相信大家并不陌生，简单的说就是：**基于事件流驱动设计代码，然后监听订阅事件，并针对事件变换处理响应**。

而在 Flutter 中，整个 `Stream` 设计外部暴露的对象主要如下图，主要包含了 `StreamController` 、`Sink` 、`Stream` 、`StreamSubscription` 四个对象。



图片要换

1、Stream 的简单使用

如下代码所示，`Stream` 的使用并不复杂，一般我们只需要：

- 创建 `StreamController` ，
- 然后获取 `StreamSink` 用做事件入口，
- 获取 `Stream` 对象用于监听，
- 并且通过监听得到 `StreamSubscription` 管理事件订阅，最后在不需要时关闭即可，看起来是不是很简单？

```
class DataBloc {
  ///定义一个Controller
  StreamController<List<String>> _dataController =
StreamController<List<String>>();
  ///获取 StreamSink 做 add 入口
  StreamSink<List<String>> get _dataSink =>
_dataController.sink;
  ///获取 Stream 用于监听
```

```
  Stream<List<String>> get _dataStream =>
_dataController.stream;
  ///事件订阅对象
  StreamSubscription _dataSubscription;

  init() {
    ///监听事件
    _dataSubscription = _dataStream.listen((value){
      ///do change
    });
    ///改变事件
    _dataSink.add(["first", "second", "three",
"more"]);

  }

  close() {
    ///关闭
    _dataSubscription.cancel();
    _dataController.close();
  }
}
```

在设置好监听后，之后每次有事件变化时，`listen` 内的方法就会被调用，同时你还可以通过操作符对 `Stream` 进行变换处理。

如下代码所示，是不是一股 rx 风扑面而来?

```
_dataStream.where(test).map(convert).transform(stre
amTransformer).listen(onData);
```

而在 Flutter 中， 最后结合 StreamBuilder , 就可以完成 **基于事件流的异步状态控件** 了!

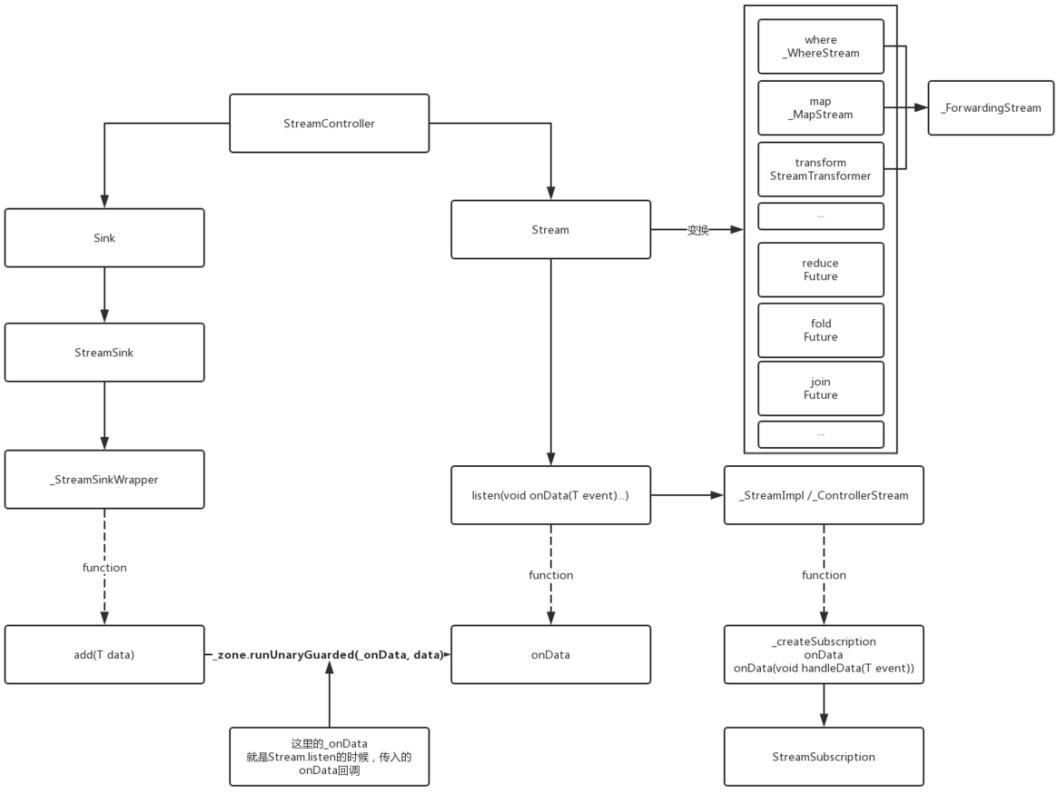
```
StreamBuilder<List<String>>(
    stream: dataStream,
    initialData: ["none"],
    ///这里的 snapshot 是数据快照的意思
    builder: (BuildContext context,
AsyncSnapshot<List<String>> snapshot) {
      ///获取到数据，为所欲为的更新 UI
      var data = snapshot.data;
      return Container();
    });
```

那么问题来了，**它们内部究竟是如果实现的呢？原理是什么？各自的作用是什么？都有哪些特性呢？后面我们将开始深入解析这个逻辑** 。

2、Stream 四天王

从上面我们知道，在 Flutter 中使用 Stream 主要有四个对象，那么这四个对象是如何“勾搭”在一起的？他们各自又担任什么责职呢?

首先如下图，我们可以从进阶版的流程图上看出 整个 Stream 的内部工作流程。



image

Flutter中 `Stream` 、`StreamController` 、`StreamSink` 和 `StreamSubscription` 都是 `abstract` 对象，他们对外抽象出接口，而内部实现对象大部分都是 `_` 开头的如 `_SyncStreamController` 、`ControllerStream` 等私有类，在这基础上整个流程概括起来就是：

有一个事件源叫 `Stream`，为了方便控制 `Stream` ，官方提供了使用 `StreamController` 作为管理；同时它对外提供了 `StreamSink` 对象作为事件输入口，可通过 `sink` 属性访问；又提供 `stream` 属性提供 `Stream` 对象的监听和变换，最后得到的 `StreamSubscription` 可以管理事件的订阅。

所以我们可以总结出：

- StreamController：如类名描述，用于整个 `Stream` 过程的控制，提供各类接口用于创建各种事件流。
- StreamSink：一般作为事件的入口，提供如 `add` ，`addStream` 等。
- Stream：事件源本身，一般可用于监听事件或者对事件进行转换，如 `listen` 、 `where` 。
- StreamSubscription：事件订阅后的对象，表面上用于管理订阅过等各类操作，如 `cacenl` 、`pause` ，同时在内部也是事件的中转关键。

回到 `Stream` 的工作流程上，在上图中我们知道， 通过 `StreamSink.add` 添加一个事件时， 事件最后会回调到 `listen` 中的 `onData` 方法，这个过程是通过 `zone.runUnaryGuarded` 执行的，这里 `zone.runUnaryGuarded` 是什么作用后面再说，我们需要知道这个 `onData` 是怎么来的?

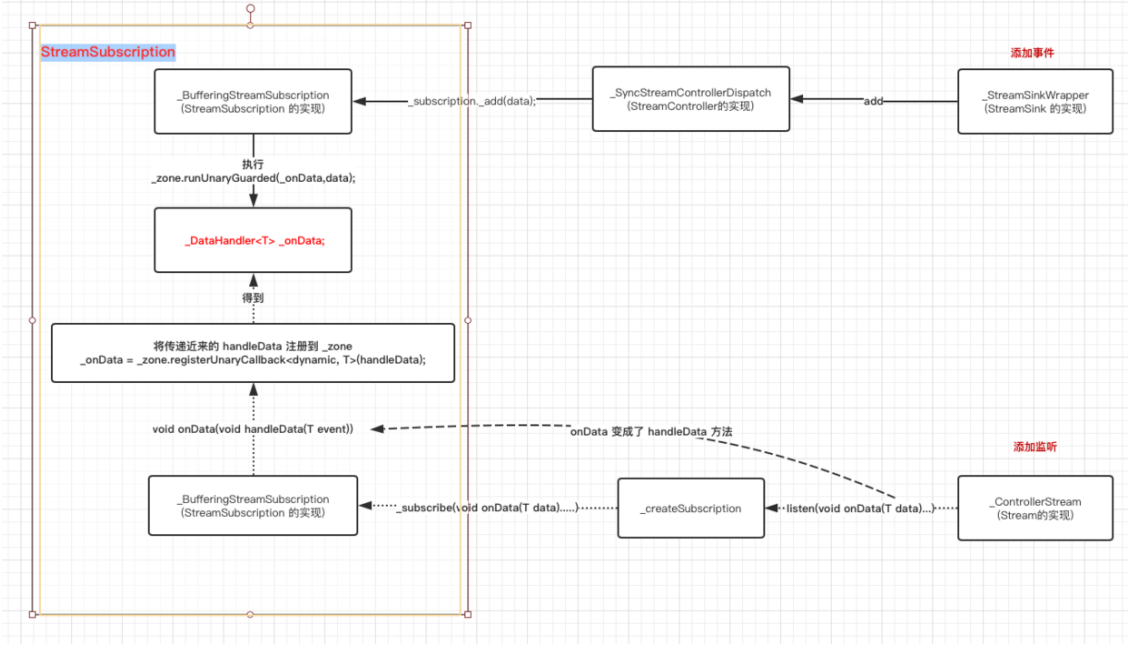


image.png
如上图，通过源码我们知道：

- 1、`Stream` 在 `listen` 的时候传入了 `onData` 回调，这个回调会传入到 `StreamSubscription` 中，之后通过 `zone.registerUnaryCallback` 注册得到 `_onData` 对象 (不是前面的 `*onData*` 回调哦)。
- 2、`StreamSink` 在添加事件是，会执行到 `StreamSubscription` 中的 `_sendData` 方法，然后通过 `_zone.runUnaryGuarded(_onData, data);` 执行 1 中得到的 `_onData` 对象，触发 `listen` 时传入的回调方法。

可以看出整个流程都是和 `StreamSubscription` 相关的，现在我们已经知道从 **事件入口到事件出口** 的整个流程时怎么运作的，那么这个过程是**怎么异步执行的呢？其中频繁出现的 `zone` 是什么？

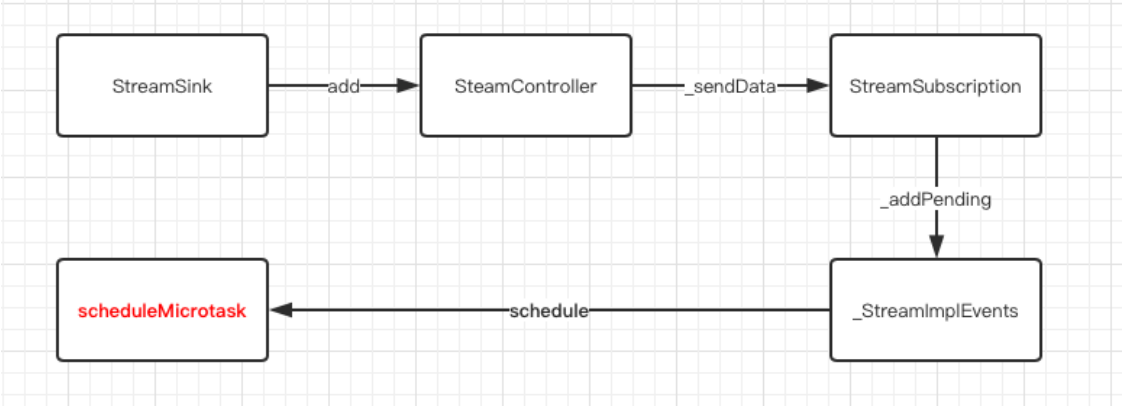
3、线程

首先我们需要知道，Stream 是怎么实现异步的?
这就需要说到 Dart 中的异步实现逻辑了，因为 Dart 是 **单线程应用** ，和大多数单线程应用一样，Dart 是以 **消息循环机制** 来运行的，而这里面主要包含两个任务队列，一个是 **microtask** 内部队列，一个是 **event** 外部队列，而 *microtask* 的优先级又高于 *event* 。
默认的在 Dart 中，如 点击、滑动、*IO*、绘制事件 等事件都属于 **event** 外部队列，**microtask** 内部队列主要是由 Dart

内部产生，而 `Stream` 中的执行异步的模式就是 `scheduleMicrotask` 了。

因为 *microtask* 的优先级又高于 *event*，所以如果 *microtask* 太多就可能会对触摸、绘制等外部事件造成阻塞卡顿哦。

如下图，就是 Stream 内部在执行异步操作过程执行流程：



image

4、Zone

那么 `Zone` 又是什么？它是哪里来的?

在上一篇章中说过，因为 Dart 中 `Future` 之类的异步操作是无法被当前代码 `try/cacth` 的，而在 Dart 中你可以给执行对象指定一个 `Zone`，类似提供一个**沙箱环境**，而在这个沙箱内，你就可以全部可以捕获、拦截或修改一些代码行为，比如所有未被处理的异常。

那么项目中默认的 `Zone` 是怎么来的？在 Flutter 中，**Dart 中的 `**Zone**` 启动是在 `**_runMainZoned**` 方法**，如下代码所示 `_runMainZoned` 的 `@pragma("vm:entry-point")` 注解表示该方式是给 Engine 调用的，到这里我们知道了 `Zone` 是

怎么来的了。

```
///Dart 中

@pragma('vm:entry-point')
// ignore: unused_element
void _runMainZoned(Function
startMainIsolateFunction, Function
userMainFunction) {
  startMainIsolateFunction((){
    runZoned<Future<void>>(····);
  }, null);
}

///C++ 中
if (tonic::LogIfError(tonic::DartInvokeField(

Dart_LookupLibrary(tonic::ToDart("dart:ui")),
"_runMainZoned",
          {start_main_isolate_function,
user_entrypoint_function}))) {
    FML_LOG(ERROR) << "Could not invoke the main
entrypoint.";
    return false;
}
```

那么 `zone.runUnaryGuarded` 的作用是什么？相较于 `scheduleMicrotask` 的异步操作，官方的解释是：**在此区域**

中使用参数执行给定操作并捕获同步错误。 类似的还有 `runUnary` 、 `runBinaryGuarded` 等，所以我们知道前面提到的 `zone.runUnaryGuarded` 就是 **Flutter 在运行的这个 zone 里执行已经注册的 `**_onData**`，并捕获异常**。

5、异步和同步

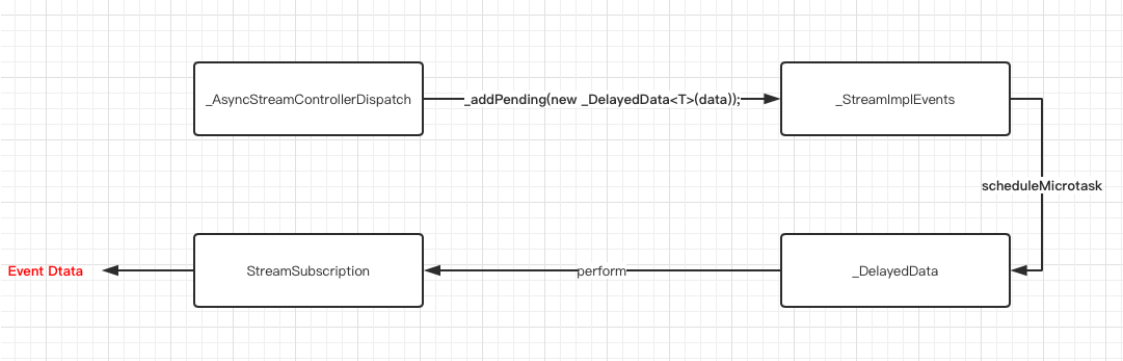
前面我们说了 `Stream` 的内部执行流程，那么同步和异步操作时又有什么区别？具体实现时怎么样的呢？

我们以默认 `Stream` 流程为例子， `StreamController` 的工厂创建可以通过 `sync` 指定同步还是异步，默认是异步模式的。 而无论异步还是同步，他们都是继承了 `_StreamController` 对象，区别还是在于 `mixins` 的是哪个 `**_EventDispatch**` 实现：

- `_AsyncStreamControllerDispatch`
- `_SyncStreamControllerDispatch`

上面这两个 `_EventDispatch` 最大的不同就是在调用 `sendData` 提交事件时，是直接调用 `StreamSubscription` 的 `_add` 方法，还是调用 `_addPending(new _DelayedData<T>(data));` 方法的区别。

如下图， **异步执行的逻辑就是上面说过的 `scheduleMicrotask`， 在 `_StreamImplEvents` 中 `scheduleMicrotask` 执行后，会调用 `_DelayedData` 的 `perform` ，最后通过 `_sendData` 触发 `StreamSubscription` 去回调数据 。**

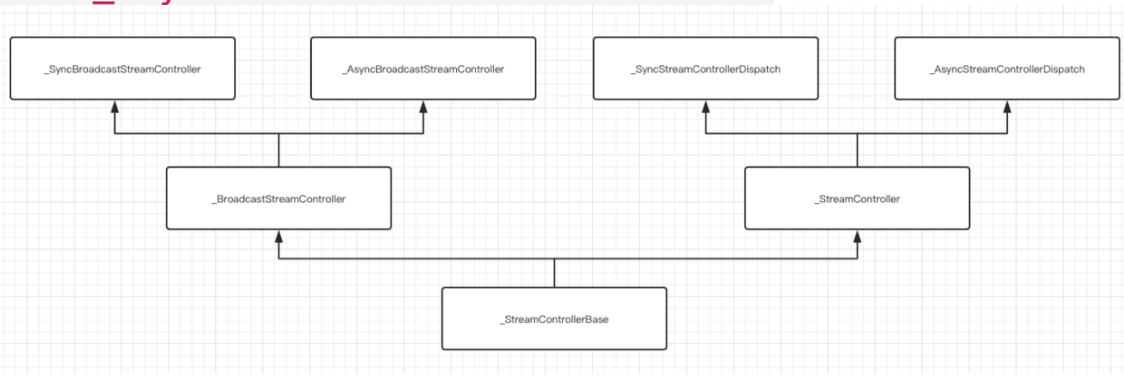


image

6、广播和非广播。

在 `Stream` 中又非为广播和非广播模式，如果是广播模式中，`StreamControlle` 的实现是由如下所示实现的，他们的基础关系如下图所示：

- `_SyncBroadcastStreamController`
- `_AsyncBroadcastStreamController`



i

广播和非广播的区别在于调用 `_createSubscription` 时，内部对接口类 `_StreamControllerLifecycle` 的实现，同时它们的差异在于：

- 在 `_StreamController` 里判断了如果 `Stream` 是 `_isInitialState` 的，也就是订阅过的，就直接报错“*Stream has already been listened to.*”，只有未订阅

的才创建 `StreamSubscription` 。

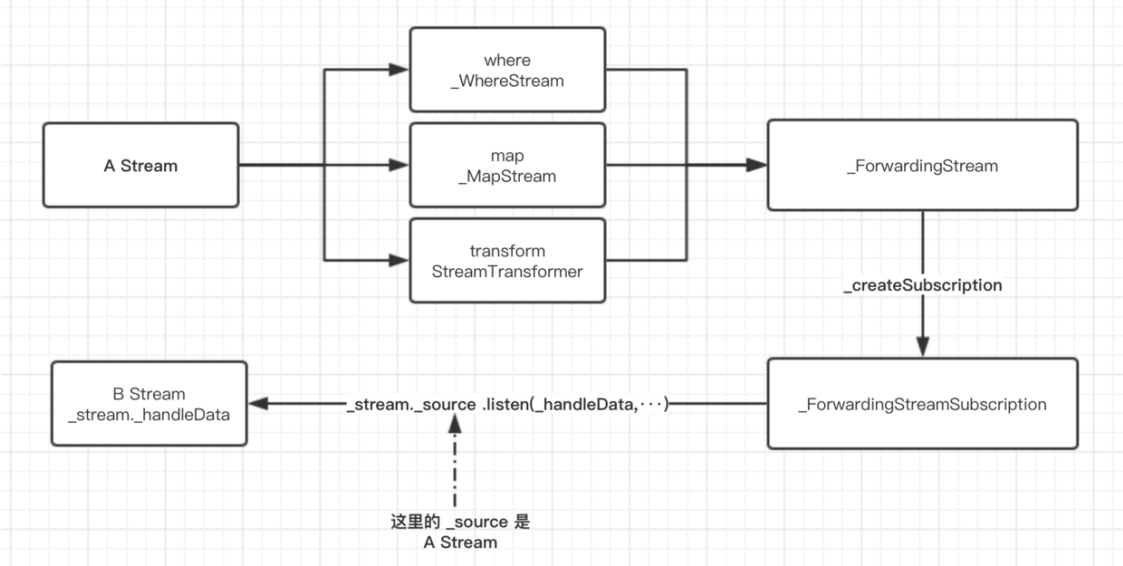
- 在 `_BroadcastStreamController` 中，`_isInitialState` 的判断被去掉了，取而代之的是 `isClosed` 判断，并且在广播中，`_sendData` 是一个 `forEach` 执行：

```
  _forEachListener((_BufferingStreamSubscription<T>
subscription) {
      subscription._add(data);
    });
```

7、Stream 变换

`Stream` 是支持变换处理的，针对 `Stream` 我们可以经过多次变化来得到我们需要的结果。那么这些变化是怎么实现的呢?

如下图所示，一般操作符变换的 `Stream` 实现类，都是继承了 `_ForwardingStream`，在它的内部的 `_ForwardingStreamSubscription` 里，会通过上一个 `Pre A Stream` 的 `listen` 添加 `_handleData` 回调，之后在回调里再次调用新的 `Current B Stream` 的 `_handleData` 。
所以事件变化的本质就是，**变换都是对 `Stream` 的 `listen` 嵌套调用组成的。**



image

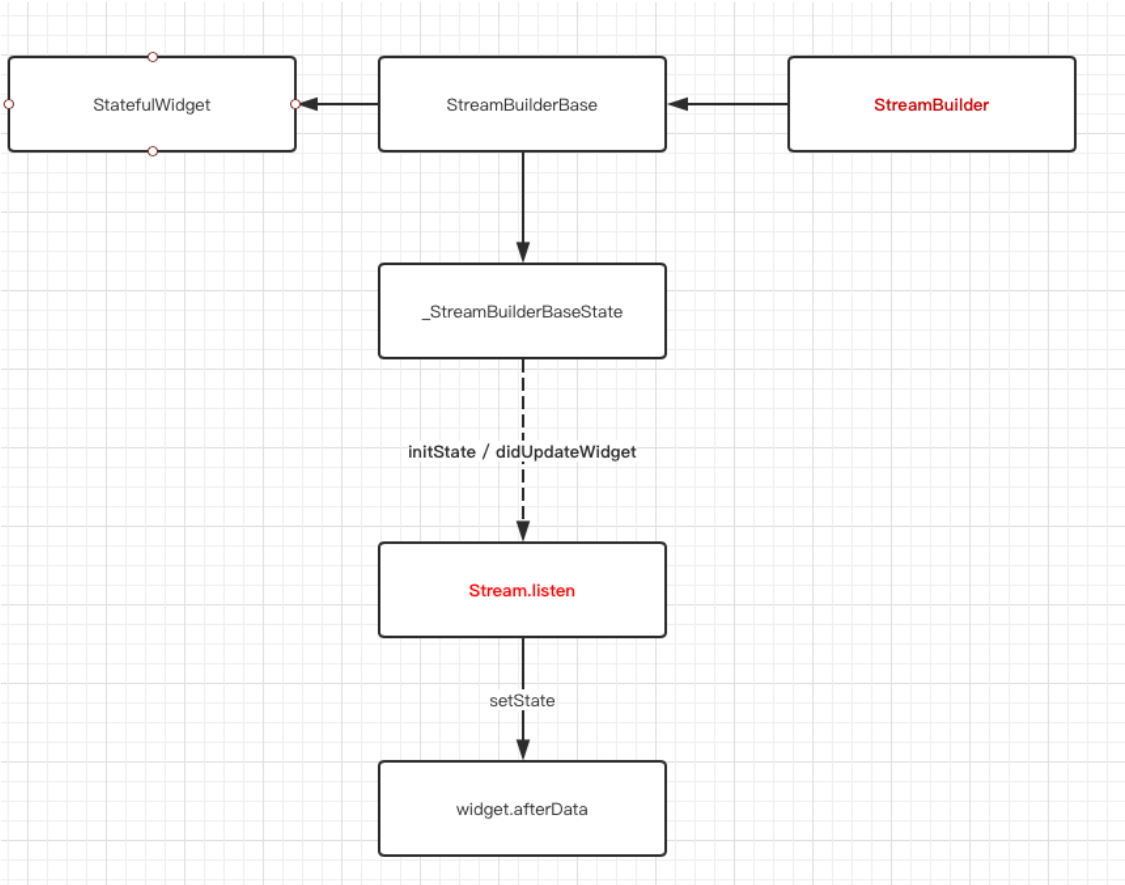
同时 `Stream` 还有转换为 `Future`，如 `firstWhere`、`elementAt`、`reduce` 等操作符方法，基本都是创建一个内部 `_Future` 实例，然后再 `listen` 的回调用调用 `Future` 方法返回。

二、StreamBuilder

如下代码所示，在 Flutter 中通过 `StreamBuilder` 构建 Widget，只需提供一个 `Stream` 实例即可，其中 `AsyncSnapshot` 对象为数据快照，通过 `data` 缓存了当前数据和状态，那 `StreamBuilder` 是如何与 `Stream` 关联起来的呢?

```
StreamBuilder<List<String>>(
    stream: dataStream,
    initialData: ["none"],
    ///这里的 snapshot 是数据快照的意思
    builder: (BuildContext context,
```

```
AsyncSnapshot<List<String>> snapshot) {
      ///获取到数据，为所欲为的更新 UI
      var data = snapshot.data;
      return Container();
    });
```



image

如上图所示， `StreamBuilder` 的调用逻辑主要在 `_StreamBuilderBaseState` 中，`_StreamBuilderBaseState` 在 `initState` 、`didUpdateWidget` 中会调用 `_subscribe` 方法，从而调用 `Stream` 的 `listen`，然后通过 `setState` 更新 UI，就是这么简单有木有?

我们常用的 `setState` 中其实是调用了 `markNeedsBuild` ，

`markNeedsBuild` 内部标记 `element` 为 `diry`，然后在下一帧 `WidgetsBinding.drawFrame` 才会被绘制，这可以看出 `setState` 并不是立即生效的哦。

三、rxdart

其实无论从订阅或者变换都可以看出， Dart 中的 `Stream` 已经自带了类似 `rx` 的效果，但是为了让 `rx` 的用户们更方便的使用，ReactiveX 就封装了 `rxdart` 来满足用户的熟悉感，如下图所示为它们的对应关系：

Dart	RxDart
StreamController	Subject
Stream	Observable

image

在 `rxdart` 中， `Observable` 是一个 `Stream`，而 `Subject` 继承了 `Observable` 也是一个 `Stream`，并且 `Subject` 实现了 `StreamController` 的接口，所以它也具有 Controller 的作用。

如下代码所示是 `rxdart` 的简单使用，可以看出它屏蔽了外界需要对 `StreamSubscription` 和 `StreamSink` 等的认知，更符合 `rx` 历史用户的理解。

```
final subject = PublishSubject<String>();

subject.stream.listen(observerA);
subject.add("AAAA1");
subject.add("AAAA2"));

subject.stream.listen(observeB);
subject.add("BBBB1");
subject.close();
```

这里我们简单分析下，以上方代码为例，

- `PublishSubject` 内部实际创建是创建了一个广播 `StreamController<T>.broadcast` 。
- 当我们调用 `add` 或者 `addStream` 时，最终会调用到的还是我们创建的 `StreamController.add`。
- 当我们调用 `onListen` 时，也是将回调设置到 `StreamController` 中。
- `rxdart` 在做变换时，我们获取到的 `Observable` 就是this，也就是 `PublishSubject` 自身这个 `Stream` ，而 `Observable` 一系列的变换，也是基于创建时传入的 `stream` 对象，比如：

```
  @override
  Observable<S> asyncMap<S>(FutureOr<S> convert(T
value)) =>
      Observable<S>(_stream.asyncMap(convert));
```

所以我们可以看出来，`rxdart` 只是对 `Stream` 进行了概念变换，变成了我们熟悉的对象和操作符，而这也是为什么

`rxdart` 可以在 `StreamBuilder` 中直接使用的原因。

所以，到这里你对 Flutter 中 Stream 有全面的理解了没?