深入理解 Flutter 中的 Stream （一）

Stream 是 Flutter 处理数据响应的一个重要手段，它提供了一种处理数据流的方式，其作用类似于 Kotlin 中的 Flow，基于发布订阅模式的设计，通过监听Stream，订阅者能不断接收到数据流的最新变化。

Stream 的基本用法

Stream能通过async*和StreamController产生，也能通过其它Stream转换而来。相较于async*，StreamController因为灵活度更高，因此更为常见，两者在使用场景上也有一定差别。

async*

相信大家一定认识async，但async*就未必，同样作为 Flutter 里异步处理的一环，async主要跟Future打交道，而async*处理的对象是Stream，async*在使用上需要搭配yield。下面这段代码演示了如何使用async*进行 1 到 10 的相加。

```
Stream<int> countStream(int to) async* {
  print("stream 被监听");
  for (int i = 1; i <= to; i++) {
    yield i;  yield 收益，产量
```

```
  }
}

Future<int> sumStream(Stream<int> stream)
async {
  var sum = 0;
  await for (final value in stream) {
    sum += value;
  }
  return sum;
}

void main() async {
  var stream = countStream(10);
  // 当注释掉下面这行，控制台不会打印出 "stream 被
监听"，也就表示 async* 方法体没被执行
  var sum = await sumStream(stream);
  print(sum); // 55
}
```

在上面的示例中，`async*`方法体里`yield`在每次的遍历中，都往`Stream`返回一个数据，通过 `await for`的监听拿到每次返回的值，接着执行`sum`操作。**值得注意的是，`async*`这种方式产**

生的stream，当stream没有被监听时，async* 方法体是不会被执行的。

如果你看着async*还有点别扭，请记住：async返回的是一个Future，而async*返回的是一个Stream。

StreamController

日常开发中，通常会通过StreamController创建Stream。只需要构造出StreamController对象，通过这个对象的.stream就可以得到Stream。

```
Stream<int> countStream(int to) {
  // 先创建 StreamController
  late StreamController<int> controller;
  controller =
StreamController<int>(onListen: () {
    // 当 Stream 被监听时会触发 onListen 回调
    for (var i = 0; i < to; i++) {
      controller.add(i);
    }
    controller.close();
  });

  return controller.stream;
```

```
}

Future<int> listenOn(Stream<int> stream)
async {
  var completer = Completer<int>();
  var sum = 0;

  // 监听 stream
  stream.listen(
    (event) { 流里的数据
      sum += event;
    },
    onDone: () => completer.complete(sum),
  );

  return completer.future;
}

void main() async {
  var stream = countStream(10);
  // 当注释掉下面这行，控制台也不会打印出 "stream
被监听"
```

```
  var sum = await listenOn(stream);
  print(sum); // 55
}
```

在创建StreamController的时候传入了一个onListen回调，当流第一次被监听的时候，会触发这个回调，此时会往流里面依次添加多个数据，listenOn方法里拿到这些数据执行相加操作。这里使用了stream的listen的方法进行监听。

Stream 左右护法

Flutter 中的 Stream 处理，涉及到三类对象，以发布订阅模式的角度去看的话，可以分为发布者 StreamController、数据通道 Stream、订阅者 StreamSubscription。

```
class Example {

  var controller = StreamController<int>();
  Stream<int> get stream =>
controller.stream;
  StreamSubscription<int>? _subscription;

  void initState() {
    _subscription = stream.listen((event) {
      print(event);
```

```
    });
    for(var i = 0; i <= 10; i++) {
      controller.add(i);
    }
  }

  void dispose() {
    _subscription?.cancel();
    _subscription = null;
  }

}
```

每一个StreamController都对应着一个Stream，当Stream被订阅时，会得到一个StreamSubscription对象。

上面的例子中，接口使用是简单的，但是他们内部的工作原理是如何？一个事件从发布到消费中间经过了哪些流程？

数据流向图

在事件处理上：Stream在被订阅时，会创建StreamSubscription，并将其中的onData等事件处理的回调传给StreamSubscription。

在事件输入输出上：StreamController通过add方法输入事件

后，先判断此时是否存在订阅者StreamSubscription，如果存在则调用StreamSubscription的onData处理，不存在就先存到_pendingEvents里，等到下次StreamSubscription出现了再向它输出事件。

可以看到，StreamController 在整个事件流向的处理中肩负着最重要的使命，它控制着事件如何输入和输出，StreamSubscription负责处理输出到这里的事件，Stream在得到StreamSubscription后就完成了它的使命选择“退隐山林”。

这么讲可能还有点“干”，为了更直观的介绍他们各自的职责，接下来我们从他们定义的接口出发，去思考他们都能做哪些事件。为了方便呈现，我只取其中最关键的部分。

StreamController

```
abstract interface class StreamController<T>
implements StreamSink<T> {

  // stream 流
  Stream<T> get stream;

  // 流状态的回调
```

```
  abstract void Function()? onListen; // 被监听
  abstract void Function()? onPause; // 流暂停
  abstract void Function()? onResume; // 流恢复
  abstract void Function()? onCancel; // 流取消/关闭

  // 流状态
  bool get isClosed;
  bool get isPaused;
  bool get hasLitener; // 当前流是否有订阅者

  // 监听 source, 转发给 stream
  Future addStream(Stream<T> source, {bool? cancelOnError});

  // 往流里面添加事件
  void add(T event);
  void addError(Object error, [StackTrace?
```

```
stackTrace]);
  Future close(); // 关闭流

  // 输出事件
  // 以下这三个接口在 _StreamControllerBase 中
  void _sendData(T data);
  void _sendError(Object error, StackTrace
stackTrace);
  void _sendDone();
}
```

1. StreamController负责管理事件流的状态，当状态变化时，会触发到相应的回调(onListen/onPause等)。
2. StreamController负责事件的输入，输入的方式有两种，一种是事件接口add、addError；另外一种是通过监听其它的Stream；同时事件也分为两种，一种是正常事件，一种是错误事件。
3. StreamController能关闭这个事件流通道，会产生一个onDone事件。
4. StreamController负责事件的输出，不同的输入对应不同的输出。

Stream

```
abstract mixin class Stream<T> {

  // 是否地广播流，广播流允许多订阅
  bool get isBroadcast => false;

  // 监听流变化，返回订阅者
  StreamSubscription<T> listen(void onData(T
event)?,
    {Function? onError, void onDone()?,
bool? cancelOnError});

  // 一系列 Stream 处理和变换操作

  Stream<T> where(bool test(T event))
{ ... }
  Stream<S> map<S>(S convert(T event))
{ ... }
  Stream<E> asyncMap<E>(FutureOr<E>
convert(T event)) { ...}
  Stream<E> asyncExpand<E>(Stream<E>?
convert(T event)) { ... }
```

```
  Stream<T> handleError(Function onError,
{bool test(error)?}) { ... }
  ...
}
```

1. Stream暴露事件流的订阅方法listen，返回当前订阅者，并把listen方法中的onData等参数注册到当前订阅者里面。
2. Stream有很多过滤转换等语法糖方法。

StreamSubscription

```
abstract interface class
StreamSubscription<T> {

  // 监听数据变化
  void onData(void handleData(T data)?);
  void onError(Function? handleError);
  void onDone(void handleDone()?);

  // 暂停/恢复 监听
  void pause([Future<void>? resumeSignal]);
  void resume();
  bool get isPaused;
```

```
  // 取消监听
  Future<void> cancel();
  // 转成 Future 对象，监听流结束事件
  Future<E> asFuture<E>([E? futureValue]);
}
```

1. StreamSubscription作为事件输出端，负责事件的输出处理。
2. StreamSubscription也能对自己的订阅行为进行暂停、恢复或取消等动作。

Stream 的分类

Stream有很多子类，对应不同场景的实现。比如对于输入端而言，可以分为同步流和异步流；在输出端上，又可分为广播流和非广播流。

同步和异步

StreamController的工厂方法中，通过sync可以指定同步或者异步。**同步和异步的区别是：事件输入后是否会立即输出。**同步流在事件输入后会立刻执行onData，异步流在事件输入后会注册一个异步事件，等到当前EventLoop中的同步事件处

理后触发onData。

```
factory StreamController(
    {void onListen()?,
    void onPause()?,
    void onResume()?,
    FutureOr<void> onCancel()?,
    bool sync = false}) {
  return sync
      ? _SyncStreamController<T>(onListen,
onPause, onResume, onCancel)
      : _AsyncStreamController<T>(onListen,
onPause, onResume, onCancel);
}
```

在实现上看，_SyncStreamController最终输出时使用的是_SyncStreamControllerDispatch，_AsyncStreamController使用的是_AsyncStreamControllerDispatch。

两者在输出处理不同，_SyncStreamControllerDispatch调用的是subscription的_add方法，_AsyncStreamControllerDispatch调用的是subscription的_addPending方法。_addPending会先将事件存到队列里，同

时如果队列没有在跑就开启队列的处理，通过scheduleMicrotask对事件进行异步处理，处理完当前事件继续处理队列时的其它事件，直到队列清空。

广播和非广播

上述代码中生产的是非广播流，广播流通过StreamController.broadcast方法创建。**广播和非广播的区别是是否允许多次订阅。**

```
factory StreamController.broadcast(
    {void onListen()?, void onCancel()?,
bool sync = false}) {
  return sync
      ?
_SyncBroadcastStreamController<T>(onListen,
onCancel)
      :
_AsyncBroadcastStreamController<T>(onListen,
onCancel);
}
```

非广播StreamController继承自_StreamController，广播StreamController继承自_BroadcastStreamController，两者的区别可以通过_subscribe的实现体现。

_StreamController的实现如下，当重复订阅后会直接抛出StateError 异常。

```
StreamSubscription<T> _subscribe(void
onData(T data)?, Function? onError,
    void onDone()?, bool cancelOnError) {
  if (!_isInitialState) {
    throw StateError("Stream has already
been listened to.");
  }
  _ControllerSubscription<T> subscription =
_ControllerSubscription<T>(
      this, onData, onError, onDone,
cancelOnError);
  // ...
  return subscription;
}
```

_BroadcastStreamController里面有两个对象_firstSubscription、_lastSubscription，_BroadcastSubscription是双向链表结构。当需要输出事件时，通过整个链表，通知所有的订阅进行消息的处理。

```
_BroadcastSubscription<T>?
_firstSubscription;
```

```
_BroadcastSubscription<T>?
_lastSubscription;
```

开发实战

通过前面接口和分类的分析，我们对这 Stream 有了更深刻的认识。刀已磨好，接下来便通过两个例子来试试这把刀到底锋不锋利。

利用 Stream 实现事件的广播

事件的广播，在开发时总会遇到，尤其是在跨组件或跨页面的场景，相信大部分开发者的项目里也都会引入类似EventBus的三方或自研框架。比如：当我在编辑个人资料时，Save之后需要通知其它页面进行刷新以展示最新的个人信息。

```
// user entity
class UserInfo {
  int uid;
  String name;

  UserInfo(this.uid, this.name);
}

// userinfo update
```

```
class UserInfoChangeEvent {

  static final _controller =
StreamController<UserInfo>.broadcast();

  static StreamSubscription<UserInfo>
subscribe(Function(UserInfo) callback) {
    return
_controller.stream.listen(callback);
  }

  static void broadcast(UserInfo userInfo) {
    _controller.add(userInfo);
  }
}

// 用户编辑页面
class UserProfileViewModel {
  ...
  // 点击 save 时，会调用到 broadcast 方法向外发
送事件
  void onSave(int uid, String name) {
```

```
UserInfoChangeEvent.broadcast(UserInfo(uid,
name));
  }

}

// 其它页面状态刷新
class ViewState extends State<ViewWidget> {

  StreamSubscription<UserInfo>?
_subscription;
  UserInfo? _curUserInfo;

  @override
  void initState() {
    super.initState();
    // 初始化时，监听 UserInfoChangeEvent
    _subscription =
UserInfoChangeEvent.subscribe((userInfo) {
      setState(() {
        _curUserInfo = userInfo;
```

```
      })
    });
  }

  @override
  void dispose() {
    super.dispose();
    // 退出时，要取消监听。否则会有内存泄漏
    _subscription?.cancel();
    _subscription = null;
  }

}
```

这里，UserInfoChangeEvent 定义了广播类型的StreamController，并且向外暴露了subscribe和boradcast接口，用户编辑页面在点击save时走到onSave方法，这个方法里调用了UserInfoChangeEvent的broadcast方法向外发送了一个更改信息的事件；

接着ViewState这里在initState时通过UserInfoChangeEvent的subscribe方法注册了监听，接收到了事件赋值到当前_curUserInfo，调用setState刷新页面。

StreamBuilder 实现 Widget 自动刷新

Flutter 提供了一个组件`StreamBuilder`，能帮助我们方便的监听`Stream`并刷新 Widget。例如进入一个页面时，通常会有一个数据加载的过程，此时页面会经历 Loading -> Loaded/LoadError 的状态变更，不同的状态会呈现不同的页面 UI，这时我们就需要定义一个 LoadingState 的枚举类型，在数据加载后时通过 StreamController 发布 LoadState 状态，`StreamBuilder`监听到更新然后会自动触发 Widget 的刷新。`AsyncSnapShot`是快照的意思，保存着此刻最新的事件。

```
StreamBuilder<LoadingState>(
  stream: viewModel.loadingStateStream,
  initialData: LoadingState.loading,
  builder: (BuildContext context,
AsyncSnapshot snapshot) {
    // 根据 snapshot 的数据处理返回
    var data = snapshot.data;
    if (data == LoadingState.Loaded) {
        reutrn Container(child: Text("Loaded
Success"));
    } else if (data ==
LoadingStat.LoadError) {
```

```
        reutrn Container(child:
Text("Loading Error"));
    } else {
        return LoadingView()
    }
  },
)
```

不过，StreamBuilder也有坑。我们知道，对于Stream来说，事件被消费了就会丢掉，无论是StreamController还是Stream都不会保存上次的值，以页面加载为例，页面进来后ViewModel执行数据加载完成后，向loadingStateStream里发布了Loaded的状态，如果此时页面还没有布局StreamBuilder，StreamBuilder就无法收到这次监听，等到后面StreamBuilder真正添加到界面上时已经错过了上次的事件，AsyncSnapshot拿到的还是initialData时设置的数据，也就是 loading 态，这样状态就会展示异常。

你可能会有疑问，为什么StreamBuilder不能一开始就添加到页面的build方面里？当然可以，但即便如此也无法保证StreamBuilder的监听一定会比viewModel的状态更新早，因为如果页面的内容较长，一开始StreamBuilder还不在可视区内，它的initState方法就不会执行，也就不会监听

`loadingStateStream`。

`StreamBuilder`会面临这种囧境，归根到底是因为`Stream`的设计。

Stream 的变换和处理

前面在介绍`Stream`的接口时，我们提到过`Stream`里面有很多操作方法。在这part，着重挑几个从名字上不太好理解的展开讲讲。

Future<E> drain<E>([E? futureValue]);

`drain`意为“排出、消耗“。这里指”排掉”这条流中间所有的事件，只响应结束信号，当流关闭时返回 futureValue。

```
final result = await Stream.fromIterable([1, 2, 3]).drain(100);
print(result); // Outputs: 100.
```

Future<S> fold<S>(S initialValue, S Function(S previous, T element) combine)

事件迭代。根据`combine`合并流里面的事件，该方法可以指定返回的类型`S`，同时可以指定初始值 `initialValue`。

```
final result = await Stream<int>.fromIterable([2, 4, 6, 8, 10])
```

```
    .fold<String>("0", (previous, element)
=> "$previous - $element");
print(result); // 0 - 2 - 4 - 6 - 8 - 10
```

Future<T> reduce(T Function(T previous, T element) combine);

也是事件迭代。与`fold`不同的是，`reduce`无法指定初始值且它只能返回与原流相同的类型`T`。

```
final result = await
Stream.fromIterable<int>([2, 6, 10, 8, 2])
    .fold<int>(10, (previous, element) =>
previous + element);
print(result); // 38
```

Future pipe(StreamConsumer<T> streamConsumer);

流管道拼接。将当前流的事件流向`streamConsumer`中，`streamConsumer`的子类实现通常是一个`StreamController`，拿到事件后通知给它的订阅者。

```
var controller = StreamController<int>();
var stream = controller.stream;
```

```
stream.listen((event) {
  print(event); // 2 4 6 8 10
});

var result = await
Stream<int>.fromIterable([2, 4, 6, 8,
10]).pipe(controller);
print("result: $result"); // null
```

Stream<E> asyncExpand<E>(Stream<E>? Function(T event) convert);

异步展开。将原流中的事件做一次展开操作，得到一个E类型的新流。

```
var stream = Stream<int>.fromIterable([2, 4,
6, 8, 10]);

var newStream = stream.asyncExpand((event) {
  return Stream<int>.fromIterable([event,
event + 1]);
});

newStream.listen((event){
  print(event);  // 2 3 4 5 6 7 8 9 10
```

```
});
```

Stream<E> asyncMap<E>(FutureOr<E> Function(T event) convert);

异步映射。跟`asyncExpand`类似，只是转换操作返回的是`FutureOr`对象，为那些转换过程中涉及到异步处理的场景提供便利。

```
var newStream = stream.asyncMap((event)
async {
  await Future.delayed(const
Duration(seconds: 1));
  return event + 1;
});

newStream.listen((event){
  print(event); // 3 5 7 9 11
});
```

实在忍不住想吐槽一下，有些方法的名字起的真心不昨滴，其中部分都有点“挂羊头卖狗肉”的感觉了。。。

你真的懂了吗?

讲了很多，现在来检验一下。假设有一段逻辑，`controller`

会增加三个事件，分别是`add(1) add(2) add()3`，`subscription`会在每次收到事件时打印`output: $event`，中间会有一次暂停，3 秒后恢复，猜一下在以下几种场景下最后输出的顺序是什么?

1. 同步流

```
void main() async {

  // 同步流：sync 为 true
  var controller =
StreamController<int>(sync: true);
  var subscription =
controller.stream.listen((event) {
    print('output: $event');
  });

  controller.add(1);
  controller.add(2);
  controller.add(3);

  print('暂停');
  subscription.pause();
```

```
  Future.delayed(const Duration(seconds: 3),
() {
    print('3秒后 -> 恢复');
    subscription.resume();
  });
}

// will print:
// output: 1
// output: 2
// output: 3
// 暂停
// 3秒后 -> 恢复
```

2. 异步流

保持 1 中其它代码不变，将`sync`的值设置成`false`。

```
// will print:
// 暂停
// 3秒后 -> 恢复
// output: 1
// output: 2
```

```
// output: 3
```

3. 异步流：使用 Future.delayed 延迟暂停

保持 2 中其它代码不变，将暂停恢复延迟`Duraiton.zero`。

```
  Future.delayed(Duration.zero, (){
    print('暂停');
    subscription.pause();

    Future.delayed(const Duration(seconds:
3), () {
      print('3秒后 -> 恢复');
      subscription.resume();
    });
  });

// will print
// output: 1
// output: 2
// output: 3
// 暂停
// 3秒后 -> 恢复
```

4. 异步流：使用 scheduleMicrotask 延迟暂停

保持 3 中其它代码不变，用scheduleMicrotask代替Future.delayed。

```
  scheduleMicrotask(() {
    print('暂停');
    subscription.pause();

    Future.delayed(const Duration(seconds:
3), () {
      print('3秒后 -> 恢复');
      subscription.resume();
    });
  });

// will print
// output: 1
// 暂停
// 3秒后 -> 恢复
// output: 2
// output: 3
```

上面的输出是否如你所料？相信如果你理解了之前的介绍，对1 2 3点的输出结果是没有问题的。但是对于第 4 点：虽然同

样为延迟暂停，3 和 4 中的输出完全不一样，4 中的输出在输出`output: 1`后才会触发暂停。这又是为什么呢？要解释这个输出，就要从源码出发了。

结语

所以，我们首先要从概念上理解他们，其次我们还要从代码上知道具体的实现。当程度的执行不及预期，缺乏代码实现层面的理解，我们便会显得手忙脚乱。像前面出现的**StreamBuilder处理中的坑**和**输出顺序的问题**，只有阅读底层源码，才能发现原因并准确修复。下一篇文章，将从源码实现上深入分析`Stream`。