Flutter Stream的使用

1.首先，来了解一下stream是什么

- 异步数据事件的来源。
- 流提供了一种接收一系列事件的方法。每个事件要么是一个数据事件，也称为流的元素，要么是一个错误
  事件，即某事已失败的通知。当流发出所有事件时，单个“完成”事件将通知监听器已达到结束。
- 您通过调用异步函数生成流，然后返回流。消耗该流将导致函数发出事件，直到它结束，流关闭。您使用*async*或*async*函数中可用的await for循环来消耗流，或者在async函数中使用*b yield*直接转发其事件

2.Stream的四大要素

- StreamController:作为整个流的控制器
- StreamSink:流事件入口
- Stream:事件源
- StreamSubscription:订阅管理

3.订阅模式

- Stream分为单订阅和多订阅模式
- 一般创建的Stream都是单订阅模式，只能被监听一次，在创建StreamController时添加broadcast使其变为多订阅模式

```
StreamController controller = StreamController.broadcast();
```

4.创建流的方式

StreamController

```
StreamController controller =
StreamController.broadcast(onCancel: () {
    print('onCancel');
  }, onListen: () {
    print('onListen');
  });
late Stream stream;
late StreamSink sink;
```

```
late StreamSubscription sub;
int count = 0;

@override
  void initState() {
    super.initState();
    stream = controller.stream;
    sink = controller.sink;
    sub = stream.listen((event) {
      print(event);
    });
  }

  @override
  void dispose() {
    super.dispose();
    controller.close();
    sub.cancel();
  }

  add() {
    count++;
    sink.add(count);
  }
```

- 通过StreamBuilder来接收数据

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('StreamPage'),
      ),
```

```
      body: StreamBuilder(
          stream: stream,
          initialData: count,
          builder: (_, snapshot) {
            return Text('$count');
          }),
      floatingActionButton: IconButton(
        icon: Icon(Icons.add),
        onPressed: () => add(),
      ),
    );
  }
```

- 点击 floatingActionButton，控制台打印

```
flutter: onListen
flutter: 1
flutter: 2
flutter: 3
flutter: 4
flutter: 5
flutter: 6
flutter: 7
flutter: 8
flutter: 9
flutter: 10
```

FromFuture

首先创建一个Future函数

```
Future<String> fromFuture() async {
    await Future.delayed(Duration(seconds: 1), () {});
    return 'fromFuture';
  }
```

通过FromFuture工厂函数创建流

```
streamFromFuture() {
    Stream stream = Stream.fromFuture(fromFuture());
    stream.listen((event) {
      print('Stream.fromFuture $event');
    }).onDone(() {
      print('Stream.fromFuture onDone');
    });
    return stream;
  }
```

FromFutures

通过FromFutures工厂函数创建流,顾名思义，就是通过一个包含多个Future的Iterable创建流

```
streamFromFutures() {
    Stream stream =
        Stream.fromFutures([fromFuture(), fromFuture(),
fromFuture()]);
    stream.listen((event) {
      print('Stream.fromFutures $event');
    }).onDone(() {
      print('Stream.fromFutures onDone');
    });
    return stream;
  }
```

FromIterable

通过集合迭代对象创建流

```
streamFromIterable() {
    final List<String> iterable = ['1', '2', '3', '4', '5'];
```

```
    Stream stream = Stream.fromIterable(iterable);
    stream.listen((event) {
      print('Stream.fromIterable $event');
    }).onDone(() {
      print('Stream.fromIterable onDone');
    });
    return stream;
  }
```

periodic

创建以 [周期] 间隔重复发出事件的流。

```
streamFromPeriodic() {
    Stream stream = Stream.periodic(Duration(seconds: 1), (i)
=> i + 1);
    stream.listen((event) {
      print('Stream.periodic $event');
    }).onDone(() {
      print('Stream.periodic onDone');
    });
    return stream;
  }
```

async*

通过async*标记函数直接创建

```
Stream<int> countStream() async* {
    for (var i = 0; i < 100; i++) {
      count++;
      await Future.delayed(Duration(seconds: 1), () {});
      yield count;
    }
  }
```

5.部分操作方法

skip

- 跳过此流中的第一个 [2] 数据事件。
- 返回一个流，该流发出的事件与此流在同时侦听时发出的事件相同，只是不会发出第一个 [2] 数据事件。当此流完成时，返回的流即告完成。

```
stream.skip(2);
```

skipWhile

- 跳过此流中的数据事件，同时它们与 [i>2] 匹配。

```
stream.skipWhile((i)=>i>2);
```

take

- 最多提供此流的第一个 [2] 数据事件。
- 返回一个流，该流发出的事件与此流同时侦听时将发出的事件相同，直到此流结束或发出 [count] 数据事件，此时返回的流完成。

```
stream.take(2);
```

takeWhile

- 在 [i > 2] 条件满足时转发数据事件。
- 返回一个流，该流提供与此流相同的事件，直到数据事件 [i > 2] 不满足。当此流完成时，或者当此流首次发出未通过 [test] 的数据事件时，将完成返回的流。

```
stream.takeWhile((i) => i > 2);
```

where

- 从此流创建一个丢弃某些元素（ i > 2）的新流。
- 新流发送与此流相同的错误和完成事件，但它仅发送满足 [test] 的数据事件。

```
stream.where((i) => i > 2);
```

map

- 将此流的每个元素转换为新的流事件。
- 创建一个新流，该流使用 [i > 2] 函数将此流的每个元素转换为新值，并发出结果。

```
stream.map((i) => i > 2);
```