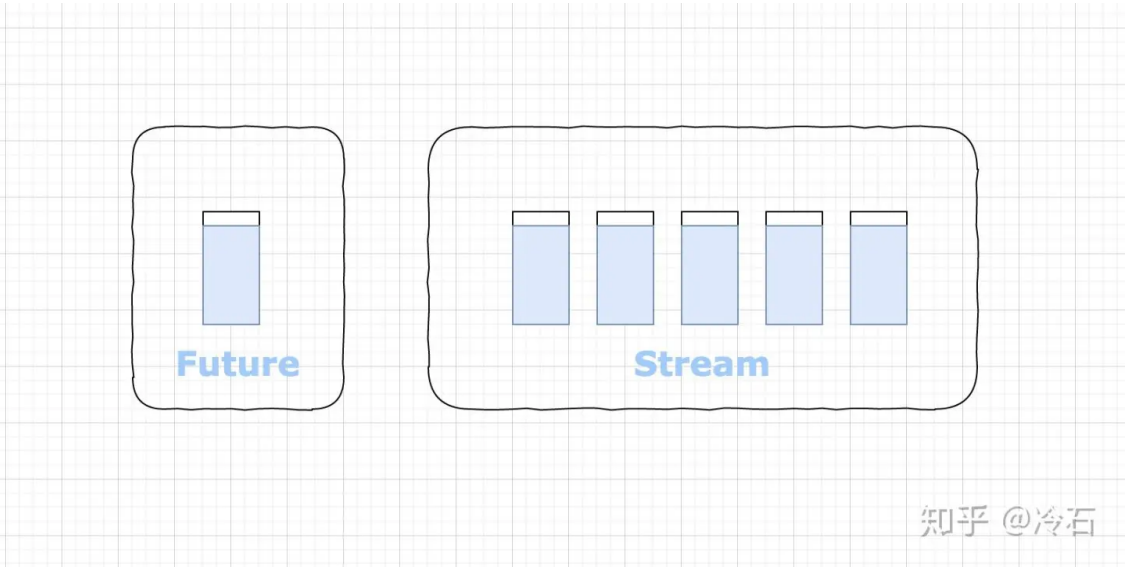
Flutter 之 Stream

SnoopPanda

Future和Stream类是Dart异步编程的核心。

Future 表示一个不会立即完成的计算过程。与普通函数直接返回结果不同的是异步函数返回一个将会包含结果的 Future。该 Future 会在结果准备好时通知调用者。

Stream 是一系列异步事件的序列。其类似于一个异步的 Iterable，不同的是当你向 Iterable 获取下一个事件时它会立即给你，但是 Stream 则不会立即给你而是在它准备好时告诉你。 iterable 可迭代的



Future 用于表示单个运算的结果，而 Stream 则表示多个结果的序列。

Stream的两种类型

1、Single-Subscription类型的Stream

最常见的类型是一个Stream只包含了某个众多事件序列的一个。而这些事件需要按顺序提供并且不能丢失。当你读取一个文件或接收一个网页请求时就需要使用这种类型的Stream。

这种Stream只能设置一次监听。重复设置则会丢失原来的事件，而导致你所监听到的剩余其他事件毫无意义。当你开始监听时，数据将以块的形式提供和获取。

2、Broadcast 类型的 Stream

另一种流是针对单个消息的，这种流可以一次处理一个消息。例如可以将其用于浏览器的鼠标事件。

你可以在任何时候监听这种 Stream，且在此之后你可以获取到任何触发的事件。这种流可以在同一时间设置多个不同的监听器同时监听，同时你也可以在取消上一个订阅后再次对其发起监听。

处理Stream的方法

```
Future<T> get first;
Future<bool> get isEmpty;
Future<T> get last;
```

```
Future<int> get length;
Future<T> get single;
Future<bool> any(bool Function(T element) test);
Future<bool> contains(Object needle);
Future<E> drain<E>([E futureValue]);
Future<T> elementAt(int index);
Future<bool> every(bool Function(T element) test);
Future<T> firstWhere(bool Function(T element) test, {T
Function() orElse});
Future<S> fold<S>(S initialValue, S Function(S previous, T
element) combine);
Future forEach(void Function(T element) action);
Future<String> join([String separator = ""]);
Future<T> lastWhere(bool Function(T element) test, {T
Function() orElse});
Future pipe(StreamConsumer<T> streamConsumer);
Future<T> reduce(T Function(T previous, T element)
combine);
Future<T> singleWhere(bool Function(T element) test, {T
Function() orElse});
Future<List<T>> toList();
Future<Set<T>> toSet();
```

上述所有的方法，除了 `drain()` and `pipe()` 方法外，都在 Iterable 类中有对应的相似方法。如果你在异步函数中使用了 **await for 循环**（或者只是在另一个方法中使用），那么使用上述的这些方法将会更加容易。例如，一些代码实现大概是这样的：

```
Future<bool> contains(Object needle) async {
  await for (var event in this) {
```

```
    if (event == needle) return true;
  }
  return false;
}

Future forEach(void Function(T element) action) async {
  await for (var event in this) {
    action(event);
  }
}

Future<List<T>> toList() async {
  final result = <T>[];
  await this.forEach(result.add);
  return result;
}

Future<String> join([String separator = ""]) async =>
    (await this.toList()).join(separator);
```

修改 Stream 的方法

下面的方法可以对原始的 Stream 进行处理并返回新的 Stream。当调用了这些方法后，设置在原始 Stream 上的监听器会先监听被转换后的新 Stream，待新的 Stream 处理完成后才会转而回去监听原始的 Stream。

```
Stream<R> cast<R>();
Stream<S> expand<S>(Iterable<S> Function(T element)
convert);
Stream<S> map<S>(S Function(T event) convert);
Stream<T> skip(int count);
```

```
Stream<T> skipWhile(bool Function(T element) test);
Stream<T> take(int count);
Stream<T> takeWhile(bool Function(T element) test);
Stream<T> where(bool Function(T event) test);
```

在 Iterable 类中也有一些将一个 iterable 转换为另一个 iterable 的方法，上述的这些方法与 Iterable 类中的这些方法相似。如果你在异步函数中使用了 **await for** 循环，那么使用上述的这些方法将会更加容易。

```
Stream<E> asyncExpand<E>(Stream<E> Function(T event)
convert);
Stream<E> asyncMap<E>(FutureOr<E> Function(T event)
convert);
Stream<T> distinct([bool Function(T previous, T next)
equals]);
```

asyncExpand() 和 asyncMap() 方法与 expand() 和 map() 方法类似，不同的是前两者允许将一个异步函数作为函数参数。Iterable 中没有与 distinct() 类似的方法，但是在不久的将来可能会加上。

```
Stream<T> handleError(Function onError, {bool
test(error)});
Stream<T> timeout(Duration timeLimit,
    {void Function(EventSink<T> sink) onTimeout});
Stream<S> transform<S>(StreamTransformer<T, S>
```

```
streamTransformer);
```

最后这三个方法比较特殊。它们用于处理 await for 循环不能处理的错误：当循环执行过程中出现错误时，该循环会结束同时取消 Stream 上的订阅且不能恢复。你可以使用 handleError() 方法在 await for 循环中使用 Stream 前将相关错误移除。

transform() 方法

`transform()` 方法并不只是用于处理错误；它更是一个通用的 Stream “map 映射”。通常而言，一个 “map 映射”会为每一个输入事件设置一个值。但是对于 I/O Stream 而言，它可能会使用多个输入事件来生成一个输出事件。这时候使用 StreamTransformer 就可以做到这一点。例如像 Utf8Decoder 这样的解码器就是一个变换器。一个变换器只需要实现一个 bind() 方法，其可通过异步函数轻松实现。

```
Stream<S> mapLogErrors<S, T>(
  Stream<T> stream,
  S Function(T event) convert,
) async* {
  var streamWithoutErrors = stream.handleError((e) =>
log(e));
  await for (var event in streamWithoutErrors) {
    yield convert(event);
  }
```

```
}
```

读取和解码文件

下面的代码示例读取一个文件并在其 Stream 上执行了两次变换。第一次转换是将文件数据转换成 UTF-8 编码格式，然后将转换后的数据变换成一个 LineSplitter 执行。文件中除了 # 开头的行外其它的行都会被打印出来。

```
import 'dart:convert';
import 'dart:io';

Future<void> main(List<String> args) async {
  var file = File(args[0]);
  var lines = utf8.decoder
      .bind(file.openRead())
      .transform(LineSplitter());
  await for (var line in lines) {
    if (!line.startsWith('#')) print(line);
  }
}
```

listen() 方法

最后一个重要的方法是 listen()。这是一个“底层”方法，其它所有的 Stream 方法都根据 listen() 方法定义。

```
StreamSubscription<T> listen(void Function(T event) onData,
    {Function onError, void Function() onDone, bool
```

```
cancelOnError});
```

你只需继承 `Stream` 类并实现 `listen()` 方法来创建一个 `Stream` 类型的子类。`Stream` 类中所有其它的方法都依赖于对 `listen()` 方法的调用。

`listen()` 方法可以让你对一个 Stream 进行监听。在你对一个 Stream 进行监听前，它只不过是个惰性对象，该对象描述了你想查看的事件。当你对其进行监听后，其会返回一个 StreamSubscription 对象，该对象用以表示一个生产事件的活跃的 Stream。这与 `Iterable` 对象的实现方式类似，不同的是 `Iterable` 对象可返回迭代器并可以进行真实的迭代操作。

Stream 允许你暂停、继续甚至完全取消一个订阅。你也可以为其设置一个回调，该回调会在每一个数据事件、错误事件以及 Stream 自身关闭时通知调用者。