Lập trình đa nền tảng

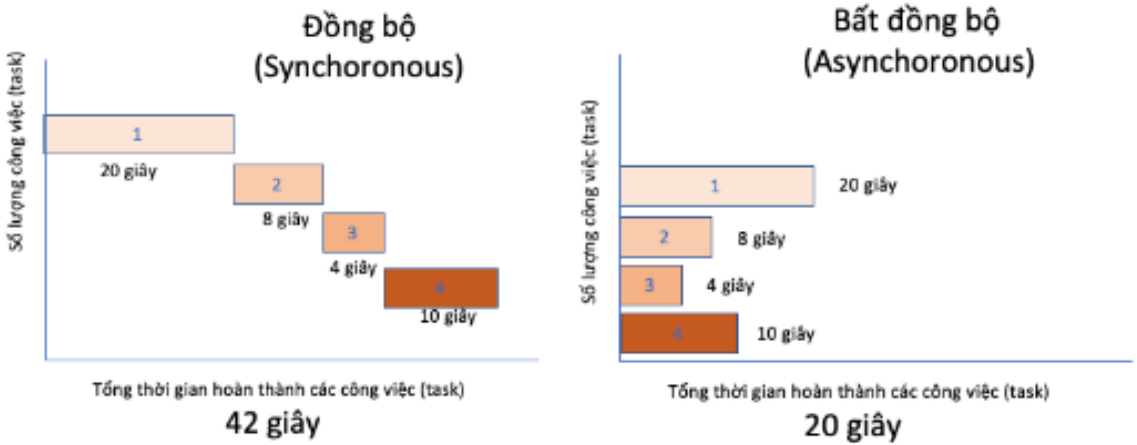
Nguyễn Duy Nhật Viễn

Chương 4

Lập trình bất đồng bộ trong Dart

Giới thiệu

- Lập trình bất đồng bộ là hoạt động cho phép chương trình thực hiện tiếp các công việc mà trong khi chờ các công việc khác vẫn đang trong qúa trình hoàn thành.



Giới thiệu

- Lập trình bất đồng bộ dùng khi :
  - Lấy dữ liệu từ thiết bị ngoại vi, Server, Firestore
  - Ghi, update vào database
  - Đọc nội dung từ file

    …
- Nếu không dùng bất đồng bộ thì sảy ra trường hợp : dữ liệu nhận về bị lỗi.

Future, async, await

Future

Future là kết quả trả về của hoạt động bất đồng bộ.

Future có hai trạng thái:

- **Future hoàn thành** : Khi hoạt động bất đồng bộ thực hiện xong, lúc đó future ở trạng thái hoàn thành và trả về một giá trị hoặc một lỗi. Ví dụ:
  - Future <T> // trả về giá trị kiểu T, trả về lỗi khi hoạt động thất bại
  - Future <String> // trả về một chuỗi String , trả về lỗi khi hoạt động thất bại
- **Future chưa hoàn thành**: Khi hoạt động đồng bộ được gọi nó sẽ trả về một Future chưa hoàn thành.

Sử dụng Future, async, await

- Future : dùng cho kiểu dữ liệu trả về.
- async : được đặt trước khối chứa await, đánh dấu khối đó là bất đồng bộ.
- await : được đặt bên trong khối async và đặt ở trước các phương thức thực hiện việc load dữ liệu, hay ghi, update vào database..

```
Future<int> processData(int param1, double param2) async {
  var value = 0;
  for (var i = 0; i < param1; ++i) {
    for (var j = 0; j < param1 * param2; j++) {
      // a lot of work here...
    }
  }
  final res = httpGetRequest(value);
  return Future<int>.value(res);
}

Run | Debug
void main() {
  Future<int> val = processData(1, 2.5);
  val.then((result) => print(result));
}
```

Sử dụng .then

- Khi dùng .then(), có thể thêm onError hoặc catchError(error) để xử lý lỗi
- Để trả về lỗi thì bạn có thể return Future.error hoặc throw một exception.

```
processData(1, 2.5).then((value) {
      // run completed code here
  })
  .catchError( (error) {
    print(error);
      // handle error
  });
```

```
Future<bool> myTypedFuture() async {
    await Future.delayed(Duration(seconds: 1));
    throw Exception('Error from Exception');
  }
```

Quản lý nhiều future

- Khi cần tải nhiều dữ liệu trước khi làm tiếp một công việc => cần phải chờ tất cả future đó hoàn thành.
- Future có hàm .wait() nhận vào list future và trả về một future duy nhất khi tất cả future trong list hoàn thành.
- Future.wait() trả về kiểu Future<List<T>> là một future có kiểu dữ liệu là list các giá trị T của các future mà nó nhận vào.

```dart
// ui to call futures
FlatButton(
  child: Text('Run Future'),
  onPressed: () async {
    await runMultipleFutures();
  },
)
// Future to run
Future<bool> myTypedFuture(int id, int duration) async {
  await Future.delayed(Duration(seconds: duration));
  print('Delay complete for Future $id');
  return true;
}
// Running multiple futures
Future runMultipleFutures() async {
  // Create list of multiple futures
  var futures = List<Future>();
  for(int i = 0; i < 10; i++) {
    futures.add(myTypedFuture(i, Random(i).nextInt(10)));
  }
  // Waif for all futures to complete
  await Future.wait(futures);
  // We're done with all futures execution
  print('All the futures has completed');
}
```

Timeouts

- Khi hoạt động bất đồng bộ quá lâu và muốn xác định thời gian tối đa cho hoạt động đó, ta có thể sử dụng hàm .timeout()

```
Future<bool> myTypedFuture(int id, int duration) async {
  await Future.delayed(Duration(seconds: duration));
  print('Delay complete for Future $id');
  return true;
}

Future runTimeout() async {
  await myTypedFuture(0, 10)
      .timeout(Duration(seconds: 2), onTimeout: (){
        print('0 timed out');
        return false;
      });
}
```

whenComplete

- Future còn một hàm khác đó là .whenComplete(), bạn có thể dùng hàm này để thực hiện code khi mà future hoàn thành.

```
var server = connectToServer();
server.post(myUrl, fields: {"name": "john", "profession": "juggler"})
      .then(handleResponse)
      .catchError(handleError)
      .whenComplete(server.close);
```

```
void main() {
  funcThatThrows()
    .then((_) => ...)          // Future completes with an error.
    .catchError((e) {
      handleError(e);
      printErrorMessage();
      return someObject;
    })                                      // Future completes with someObject.
    .whenComplete(() => print("Done!")); // Future completes with someObject.
}
```

Sự khác nhau giữa then và whenComplete

- .whenComplete: Hàm bên trong .whenComplete được gọi khi Future hoàn thành, kết quả có thể một giá trị hay một lỗi.
- .then: Trả về Future hoàn thành với kết quả của lệnh gọi onValue (nếu Future này hoàn thành với một giá trị) hoặc onError (nếu Future này hoàn thành với lỗi)

```
myTypedFuture().then((value) {
  print("called with value = $value");
}).catchError(
  (onError) {
    print("called when there is an error catches error $onError");
  },
).whenComplete(() {
  print("called when future completes");
});
```

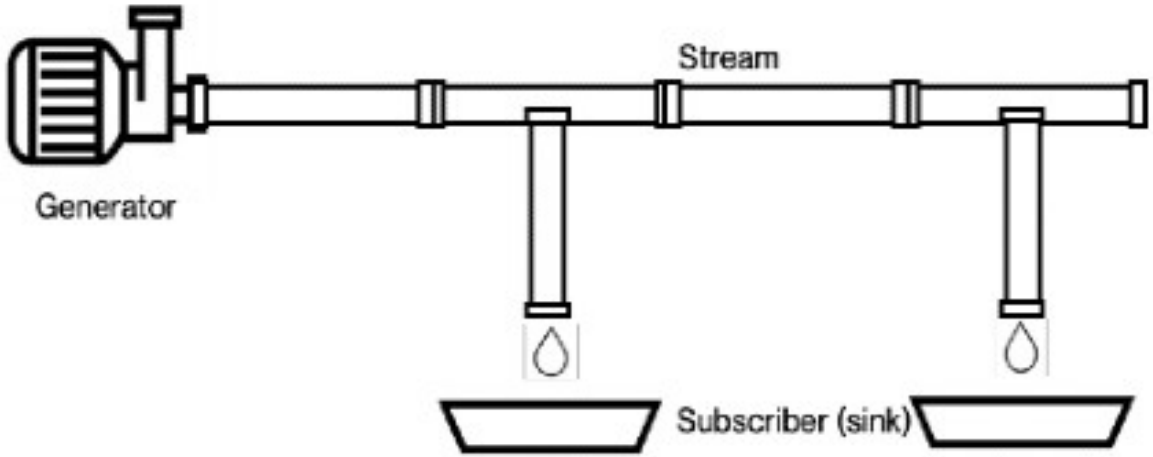
Streams

Khái niệm

- Stream là một chuỗi dữ liệu bất đồng bộ, ví dụ như : các event từ người dùng hay chuỗi dữ liệu được đọc từ file.
- Stream được sử dụng trong lập trình bất đồng bộ khi đọc, ghi lấy dữ liệu từ Server, database và các file.
- Để xử lý một Stream, ta có thể dùng các phương thức : await for hoặc listen().
- Stream có hai loại:
  - Single Subscription : Chỉ có tối đa một listener dùng cho một Stream.
  - Broadcast: Có nhiều listener để xử lý một Stream

Khái niệm

- Generator: Tạo dữ liệu mới và gửi chúng qua Stream.
- Stream: Luồng dữ liệu.
- Subscriber: Nơi quan tâm đến dữ liệu được truyền trong Stream. Nếu dữ liệu mới được gửi qua Stream bởi Generator, Subscriber đang listen sẽ nhận được thông báo.



Generator

Generator phát dữ liệu và chuyển lên Stream, vì Stream là nơi duy nhất để người nghe đăng ký.

Có 2 loại Generator:

- Không đồng bộ: trả về một đối tượng Stream <T>. Luồng dữ liệu không đồng bộ được xử lý bởi người đăng ký với await.
- Đồng bộ: trả về một đối tượng Iterable <T>. Luồng dữ liệu đồng bộ có thể được xử lý trong vòng lặp vì dữ liệu này được gửi đi một cách tuần tự.

Streams và generators

- Ví dụ: tạo một Generator bất đồng bộ tạo ra 100 số ngẫu nhiên, một số mỗi giây.
- Để cho trình biên dịch biết rằng hàm này là một Generator, sử dụng modifier: async *.

1. Generator bất đồng bộ, kiểu trả về Stream <T>. Dùng async * để phát dữ liệu.

2. Vòng lặp tạo ra 100 số ngẫu nhiên.

```dart
import 'dart:math';

Stream<int> randomNumbers() async* { // 1.
  final random = Random();
  for(var i = 0; i < 100; ++i)
  { // 2.
    await Future.delayed(Duration(seconds: 1)); // 3.
    yield random.nextInt(50) + 1; // 4.
}// 5.
```

3. Dùng phương thức khởi tạo Future.delayed (...) trả về Future sau một thời gian xác định (do Duration quyết định). Được sử dụng để "ngủ" Stream đang thực thi trong một thời gian nhất định mà không bị nghẽn.

4. Từ khóa yield đẩy dữ liệu lên Stream luồng. Yield chịu trách nhiệm gửi các sự kiện mới lên Stream

5. Khi một hàm có modifier async * thì không có lệnh return. Vì dữ liệu đã được gửi lên Stream nhờ yield nên không có gì để return khi trình Generator "tắt".

Streams và generators

- Cũng với ví dụ trên nhưng sử dụng một Iterable Generator:

```
// contains the 'sleep' function
import 'dart:io';

Iterable<int> randomNumbers() sync* {
        final random = Random();
        for(var i = 0; i < 100; ++i) {
          sleep(Duration(seconds:1));
          yield random.nextInt(50) + 1;
} }
```

Generator đồng bộ, kiểu trả về Iterable <T>. Dùng sync * để phát dữ liệu.

Sử dụng hàm sleep để ngủ

Khi một hàm có modifier sync * thì không có lệnh return.

Lưu ý

- Có thể sử dụng nhiều yield trong 1 khối lệnh để đẩy nhiều giá trị / tham số lên Stream

```
Stream<int> randomNumbers() async* {
  final random = Random();

  for(var i = 0; i < 100; ++i) {
    await Future.delayed(Duration(seconds: 1));
    yield random.nextInt(50) + 1;
    yield random.nextInt(50) + 1;
    yield random.nextInt(50) + 1;
  }
}
```

Một số phương thức khởi tạo của Stream

- Stream<T>.periodic()
  - Tạo một Stream phát các sự kiện lặp lại theo chu kỳ Duration.
  - Đối số của chạy từ 0 và tuần tự tăng lên 1 sau mỗi sự kiện được phát (là một "bộ đếm sự kiện").

```
final random = Random();

final stream = Stream<int>.periodic(
    const Duration(seconds: 2),
    (count) => random.nextInt(10)
);
```

Một số phương thức khởi tạo của Stream

- Stream<T>.value()
  - Tạo một Stream phát ra một sự kiện trước khi hoàn tất.
- Stream<T>.error()
  - Tạo một Stream phát ra một lỗi trước khi hoàn tất.

```
final stream = Stream<String>.value("Hello");
Future<void> something(Stream<int> source) async {
    try {
        await for (final event in source) { ... }
    } on SomeException catch (e) {
        print("An error occurred: $e");
    }
}

// Pass the error object
something(Stream<int>.error("Whoops"));
```

Một số phương thức khởi tạo của Stream

```
final stream = Stream<double>.fromIterable(const <double>[
    1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9
]);

final stream = Stream<double>.fromFuture(
    Future<double>.value(15.10)
);

final stream = Stream<double>.empty();
```

- Stream<T>.fromIterable()
  - Tạo một Stream đăng ký một lần (single-subscription), phát các giá trị trong danh sách.
- Stream<T>.fromFuture()
  - Tạo một Stream đăng ký một lần từ đối tượng Future <T> đã cho.
  - Khi Future <T> hoàn thành, 2 sự kiện được phát ra gồm: sự kiện dữ liệu (hoặc lỗi) và sự kiên “hoàn thành” báo hiệu luồng đã kết thúc
- Stream<T>.empty()
  - Gửi một sự kiện ”hoàn thành" để báo hiệu kết thúc.

Subscribers

- Khi một Generator đã sẵn sàng để định kỳ phát ra các số ngẫu nhiên, ta có thể đăng ký để nhận thông báo về các giá trị mới.

```dart
Stream<int> randomNumbers() async* {
    // see code above...
}

void main() async {                    // 1.
  final stream = randomNumbers();      // 2.

  await for (var value in stream) {    // 3.
    print(value);
  }

  print("Async stream!");              // 4.

}
```

1. Vì đang xử lý một Stream bất đồng bộ, nên cần phải đánh dấu hàm là async và trong hàm sử dụng await.
2. Đăng ký Stream. Đây là thời điểm đầu tiên Generator bắt đầu phát ra dữ liệu có một listener (khởi tạo theo yêu cầu).
3. Khi xử lý các Stream, vòng lặp await for có thể "bắt" các giá trị được gửi qua Stream bởi một yield trong Generator. await for hoạt động như một vòng lặp for thông thường.
4. Chuỗi được in ra khi kết thúc vòng lặp.

Subscribers

- Stream cũng có thể gửi các sự kiện lỗi, các sự kiện có thể xảy ra do một ngoại lệ đã được đưa vào trong Generator.
- Ví dụ: Mỗi giây, một số mới được xuất ra cho đến khi đạt đến maxCount;

```
Stream<int> counterStream([int maxCount = 10000]) async* {
    final delay = const Duration(seconds: 1);
    var count = 0;

    while (true) {
        if (count == maxCount) {
            break;
        }
        await Future.delayed(delay);
        yield ++count;
    }
}

void main() async {
    await for(var c in counterStream) {
        print(c);
    }
}
```

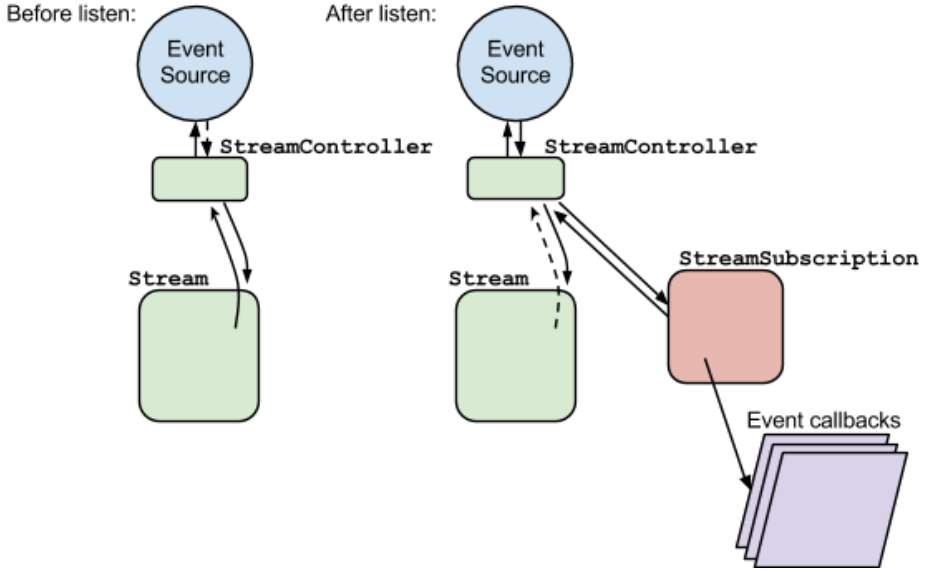
Đồng bộ và bất đồng bộ

- Trong cả hai trường hợp, yield được sử dụng để gửi dữ liệu lên Stream và không sử dụng return trong hàm.

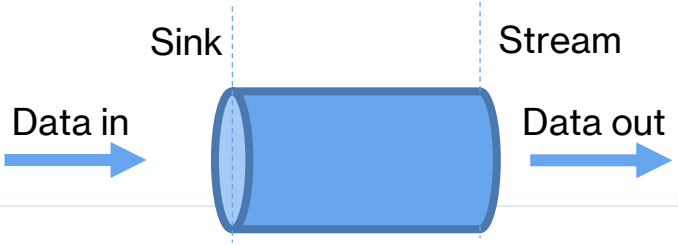
Đồng bộ	Bất đồng bộ
Trả về một Stream<T>	Trả về một Iterable<T>
Đánh dấu hàm với async*	Đánh dấu hàm với sync*
Có thể sử dụng await	Không thể sử dụng await
Các Subscribers phải dùng await for	Các Subscribers phải dùng for

StreamController

- Đối tượng StreamController điều khiển Dart Streams.
- StreamController được sử dụng để tạo Stream và gửi các dữ liệu, lỗi và các sự kiện đã thực hiện lên Stream.
- StreamController giúp kiểm tra các thuộc tính của Stream



StreamController



Để handle một Stream, sử dụng StreamController.

Dữ liệu được đẩy vào Stream: dùng thuộc tính Sink.

Để publish dữ liệu ra ngoài: dùng thuộc tính Stream.

Để transform, modify hay bất kỳ một thao tác để chỉnh sửa như xoá, cập nhật thì chúng ta dùng StreamTransformer.

StreamController

- Có 2 loại Stream:
  - Single - Subscription Stream
  - Broadcast Streams

Single - Subscription Stream

- Ví dụ một Subscriber nghe các sự kiện được gửi lên Stream thông qua StreamController
- Code này hoạt động tốt với một Subscriber nhưng không hoạt động nếu Stream có nhiều listeners.

```dart
import 'dart:async';

// Initializing a stream controller
StreamController<String> controller = StreamController<String>();

// Creating a new stream through the controller
Stream<String> stream = controller.stream;

Run | Debug
void main() {
  // Setting up a subscriber to listen for any events sent on the stream
  StreamSubscription<String> subscriber = stream.listen((String data) {
    print(data);
  }, onError: (error) {
    print(error);
  }, onDone: () {
    print('Stream closed!');
  });

  // Adding a data event to the stream with the controller
  controller.sink.add('Hello!');

  // Adding an error event to the stream with the controller
  controller.addError('Error!');

  // Closing the stream with the controller
  controller.close();
}
```

Output 1.56s

```
Hello!
Error!
Stream closed!
```

Broadcast Streams

- Để giải quyết vấn đề này, StreamController có thể thiết lập broacast Streams bằng phương thức broacast.
- Một số phương thức hữu dụng:
  - hasListener → bool: xác định Stream có listener không.
  - isClosed → bool: kiểm tra Stream đóng chưa.
  - isPaused → bool: kiểm tra Stream có dừng không.

```dart
import 'dart:async';
// Initializing a stream controller for a broadcast stream
StreamController<String> controller = StreamController<String>.broadcast();
// Creating a new broadcast stream through the controller
Stream<String> stream = controller.stream;
Run | Debug
void main() {
    // Setting up a subscriber to listen for any events sent on the stream
    StreamSubscription<String> subscriber1 = stream.listen((String data) {
        print('Subscriber1: ${data}');
    },
    onError: (error) {
        print('Subscriber1: ${error}');
    },
    onDone: () {
        print('Subscriber1: Stream closed!');
    });
    // Setting up another subscriber to listen for any events sent on the stream
    StreamSubscription<String> subscriber2 = stream.listen((String data) {
        print('Subscriber2: ${data}');
    },
    onError: (error) {
        print('Subscriber2: ${error}');
    },
    onDone: () {
        print('Subscriber2: Stream closed!');
    });
    // Adding a data event to the stream with the controller
    controller.sink.add('Hello!');
    // Adding an error event to the stream with the controller
    controller.addError('Error!');
    // Closing the stream with the controller
    controller.close();
}
```