06

Stream

네트워킹과 비동기

스트림과 스트림 빌더

스트림 — Stream

- Stream은 반복해서 발생하는 데이터를 다룰 때 사용
- Future와 비슷한 목적으로 미래에 반복해서 발생하는 데이터를 다룰 때 주로 사용하지만 비동기가 아닌 곳에서도 사용

• 데이터 한 번 반환

```
Future<int> futureFun() async {
  return await Future.delayed(Duration(seconds: 1),() {
    return 10;
  });
}

void onPress() async {
  await futureFun().then((value) => print("result: $value"));
}
```

• 데이터 5번 반환

```
Stream<int> streamFun() async* {
  for (int i = 1; i <= 5; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}

void onPress() async {
  await for (int value in streamFun()) {
    print("value : $value");
  }
}
```

스트림과 스트림 빌더

스트림 — Stream

- 데이터를 반복해서 반환하는 비동기 함수는 yield를 사용하며 async*로 선언하고 반환 타입이 Stream
- 여러 개의 데이터를 받고자 for 문을 이용
- listen() 함수로 여러 번 반환하는 데이터를 받을 수도 있습니다.

• listen() 함수로 반환값 여러 번 받기

```
void onPress() {
  streamFun().listen((value) {
    print('value : $value');
  });
}
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- Iterable 타입 데이터 만들기 — fromIterable()
  - fromIterable()은 Stream의 생성자
  - Stream 객체를 만들면서 매개변수로 List 같은 Iterable 타입의 데이터를 전달

• Iterable 타입 데이터 만들기

```
var stream = Stream.fromIterable([1, 2, 3]);
stream.listen((value) {
  print("value : $value");
});
```

▶ 실행 결과

```
value : 1
value : 2
value : 3
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- Future 타입 데이터 만들기 — fromFuture()
  - fromFuture()는 Future 타입의 데이터를 Stream 객체로 만들어 주는 생성자

• Future 타입 데이터 만들기

```
Future<int> futureFun() {
  return Future.delayed(Duration(seconds: 3), () {
    return 10;
  });
}

test4() {
  var stream = Stream.fromFuture(futureFun());
  stream.listen((value) {
    print("value : $value");
  });
}
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- 주기 지정하기 — periodic()
  - periodic()은 주기적으로 어떤 작업을 실행하는 Stream 객체를 만드는 생성자

• 주기 지정하기

```
int calFun(int x) {
  return x * x;
}

test1() async {
  Duration duration = Duration(seconds: 2);
  Stream<int> stream = Stream<int>.periodic(duration, calFun);
  await for (int value in stream) {
    print('value : $value');
  }
}
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- 횟수 지정하기 — take()
  - take() 함수는 데이터 발생 횟수를 지정할 때 사용

• 횟수 지정하기

```
int calFun(int x) {
  return x * x;
}

test1() async {
  Duration duration = Duration(seconds: 2);
  Stream<int> stream = Stream<int>.periodic(duration, calFun);   // 2초에 한 번씩 데이터 발생
  stream = stream.take(3);
  await for (int value in stream) {
    print('value : $value');
  }
}
```

▶ 실행 결과

```
value : 0
value : 1
value : 4
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- 조건 지정하기 — takeWhile()
  - takeWhile() 함수는 발생 조건을 설정할 때 사용
  - 조건 함수에서 true를 반환할 때마다 데이터를 만들고 false를 반환하면 멈춥니다.

• 조건 지정하기

```
int calFun(int x) {
  return x * x;
}

test1() async {
  Duration duration = Duration(seconds: 2);
  Stream<int> stream = Stream<int>.periodic(duration, calFun);
  stream = stream.takeWhile((value) {
    return value < 20;
  });
  await for (int value in stream) {
    print('value : $value');
  }
}
```

▶ 실행 결과

```
value : 0
value : 1
value : 4
value : 9
value : 16
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- 생략 지정하기 — skip()
  - skip() 함수에 지정한 횟수만큼만 생략하고 그 이후부터 데이터를 만듭니다.

• 생략 지정하기

```
int calFun(int x) {
  return x * x;
}

test1() async {
  Duration duration = Duration(seconds: 2);
  Stream<int> stream = Stream<int>.periodic(duration, calFun);
  // stream = stream.take(3);
  stream = stream.takeWhile((value) {
    return value < 20;
  });
  stream = stream.skip(2);
  await for (int value in stream) {
    print('value : $value');
  }
}
```

▶ 실행 결과

```
value : 4
value : 9
value : 16
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- 생략 조건 지정하기 — skipWhile()
  - skipWhile()은 매개변수에 지정한 함수에서 true가 반환될 때 데이터 발생을 생략하는 함수
  - skipWhile()의 매개변수에 지정한 함수에서 false가 반환될 때까지 데이터 발생

• 생략 조건 지정하기

```
int calFun(int x) {
  return x * x;
}

test1() async {
  Duration duration = Duration(seconds: 2);
  Stream<int> stream = Stream<int>.periodic(duration, calFun);
  stream = stream.take(10);
  stream = stream.skipWhile((value) {
    return value < 50;
  });
  await for (int value in stream) {
    print('value : $value');
  }
}
```

▶ 실행 결과

```
value : 64
value : 81
```

스트림과 스트림 빌더

스트림을 만드는 여러 가지 방법

- List 타입으로 만들기 — toList()
  - Stream으로 발생한 여러 데이터를 모아서 한 번에 List 타입으로 반환

• List 타입으로 만들기

```
int calFun(int x) {
  return x * x;
}

test2() async {
  Duration duration = Duration(seconds: 2);
  Stream<int> stream = Stream<int>.periodic(duration, calFun);
  stream = stream.take(3);
  Future<List<int>> future = stream.toList();
  future.then((list) {
    list.forEach((value) {
      print('value : $value');
    });
  });
}
```

▶ 실행 결과

```
value : 0
value : 1
value : 4
```

스트림과 스트림 빌더

스트림 빌더 — StreamBuilder

- 여러 번 발생하는 데이터를 앱의 화면에 출력할 때는 StreamBuilder 위젯을 이용
- stream 매개변수에 반복해서 데이터를 발생시키는 Stream을 지정
- 데이터가 발생할 때마다 builder 매개변수에 지정한 함수가 호출
- AsyncSnapshot 객체이며, 이 객체의 hasData 속성으로 발생한 데이터가 있는지를 판단
- data 속성으로 발생한 데이터를 받을 수 있습니다.

• 스트림 빌더 사용하기

```
body: Center(
    child: StreamBuilder(
        stream: test(),
        builder: (BuildContext context, AsyncSnapshot<int> snapshot) {
          if (snapshot.hasData) {
            return Text('data : ${snapshot.data}');
          }
          return CircularProgressIndicator();
        }),
    )
),
```

스트림과 스트림 빌더

스트림 빌더 — StreamBuilder

- AsyncSnapshot의 connectionState 속성을 이용, Stream의 연결 상태를 얻을 수 있습니다.
  - •ConnectionState.waiting: 데이터 발생을 기다리는 상태
  - •ConnectionState.active: 데이터가 발생하고 있으며 아직 끝나지 않은 상태
  - •ConnectionState.done: 데이터 발생이 끝난 상태

• 연결 상태 파악하기

```
Center(
    child: StreamBuilder(
        stream: test(),
        builder: (BuildContext context, AsyncSnapshot<int> snapshot) {
          if (snapshot.connectionState == ConnectionState.done) {
            return Text(
              'Completed',
              style: TextStyle(
                fontSize: 30.0,
              ),
            );
          } else if (snapshot.connectionState == ConnectionState.waiting) {
            return Text(
              'Waiting For Stream',
              style: TextStyle(
                fontSize: 30.0,
              ),
            );
          }
          return Text(
            'data :${snapshot.data}',
            style: TextStyle(
              fontSize: 30.0,
            ),
          );
        }),
    )
```

스트림 구독, 제어기, 반환기

StreamSubscription

- 스트림 구독자 — StreamSubscription
- 스트림에서 반복해서 발생하는 데이터를 별도의 구독자로도 이용
- listen() 함수의 반환 타입이 StreamSubscription

• 스트림 데이터 얻기

```
var stream = Stream.fromIterable([1, 2, 3]);
stream.listen((value) {
  print("value : $value");
});
```

스트림 구독, 제어기, 반환기

StreamSubscription

- listen() 함수의 매개변수에는 데이터를 받는 기능 외에 오류나 데이터 발생이 끝났을 때 실행할 함수 등을 등록할 수 있습니다.

• onError와 onDone 함수

```
var stream = Stream.fromIterable([1, 2, 3]);
stream.listen((value) {
  print("value : $value");
},
onError: (error) {
  print('error : $error');
},
onDone: () {
  print('stream done...');
});
```

▶ 실행 결과

```
value : 1
value : 2
value : 3
stream done...
```

스트림 구독, 제어기, 반환기

StreamSubscription

- listen() 함수에 등록할 게 많고 복잡하다면 StreamSubscription을 이용

• onError와 onDone 따로 정의하기

```
var stream = Stream.fromIterable([1, 2, 3]);
StreamSubscription subscription = stream.listen(null);

subscription.onData((data) {
  print('value : $data');
});
subscription.onError((error) {
  print('error : $error');
});
subscription.onDone(() {
  print('stream done...');
});
```

스트림 구독, 제어기, 반환기

스트림 제어기 — StreamController

- 스트림 제어기는 하나의 내부 스트림을 가지고 있으며 이 스트림을 이용해 데이터 발행 가능

• 스트림 제어기

```
var controller = StreamController();

var stream1 = Stream.fromIterable([1, 2, 3]);
var stream2 = Stream.fromIterable(['A','B','C']);

stream1.listen((value) {
  controller.add(value);
});
stream2.listen((value) {
  controller.add(value);
});

controller.stream.listen((value) {
  print('$value');
});
```

스트림 구독, 제어기, 반환기

스트림 제어기 — StreamController

- 스트림 제어기에 데이터를 추가하는 것은 꼭 스트림으로 발생하는 데이터뿐만 아니라 다른 데이터도 담을 수 있습니다.

• 스트림에 다른 데이터 추가

```
controller.stream.listen((value) {
  print('$value');
});

controller.add(100);
controller.add(200);
```

스트림 구독, 제어기, 반환기

스트림 제어기 — StreamController

- 같은 스트림을 2번 이상 listen()으로 가져오면 두 번째 listen()부터 오류가 발생
- 스트림 제어기를 이용하면 listen() 함수를 여러 번 호출할 수 있습니다

• 두 번째 가져오기 실패

```
var stream1 = Stream.fromIterable([1, 2, 3]);
stream1.listen((value) {print('listen1 : $value');});   // 정상
stream1.listen((value) {print('listen2 : $value');});   // 오류
```

• 방송용 스트림 제어기 만들기

```
var controller = StreamController.broadcast();
controller.stream.listen((value) {print('listen1 : $value');});
controller.stream.listen((value) {print('listen2 : $value');});
controller.add(100);
controller.add(200);
```

스트림 구독, 제어기, 반환기

스트림 변환기 — StreamTransformer

- StreamTransformer는 스트림으로 발생한 데이터를 변환하는 역할
- 스트림으로 발생한 데이터를 이용하기 전에 스트림 변환기로 데이터를 변환하고 그 결과를 listen()에서 이용할 때 사용

• 스트림 변환기

```
var stream = Stream.fromIterable([1, 2, 3]);

StreamTransformer<int, dynamic> transformer =
    StreamTransformer.fromHandlers(handleData: (value, sink) {
  print('in transformer... $value');
});

stream.transform(transformer).listen((value) {
  print('in listen... $value');
});
```

▶ 실행 결과

```
in transformer... 1
in transformer... 2
in transformer... 3
```

스트림 구독, 제어기, 반환기

스트림 변환기 — StreamTransformer

- 스트림 변환기를 이용하면 스트림 데이터의 로그를 출력하거나 필터링 적용, 데이터 변환 작업 등을 할 수 있습니다.

• sink 매개변수 이용하기

```
var stream = Stream.fromIterable([1, 2, 3]);

StreamTransformer<int, dynamic> transformer = StreamTransformer.fromHandlers(handleData:
(value, sink) {
  print('in transformer... $value');
  sink.add(value * value);
});

stream.transform(transformer).listen((value) {
  print('in listen... $value');
});
```

▶실행 결과

```
in transformer... 1
in listen... 1
in transformer... 2
in listen... 4
in transformer... 3
in listen... 9
```

감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.

윌리엄 셰익스피어
William Shakespeare