Flutter 상태관리 개념 및 GetX 학습 리포트

1. 상태관리(State Management)

1.1 개념

상태(State)란 애플리케이션에서 변하는 데이터를 의미한다. 상태관리는 이러한 상태를 효율적으로 관리하여 UI와 데이터 간의 일관성을 유지하는 방법론이다. Flutter에서는 상태가 변경될 때 위젯을 다시 빌드하여 화면을 업데이트하는 방식으로 동작한다.

1.2 상태관리의 필요성

Flutter는 선언형 UI 프레임워크이기 때문에 상태 변화에 따라 UI가 적절히 업데이트되어야 한다. 이를 효율적으로 관리하지 않으면 다음과 같은 문제점이 발생할 수 있다:

- **불필요한 리빌드:** 전체 위젯이 불필요하게 다시 그려짐.
- **비효율적인 데이터 흐름:** 부모-자식 관계를 거쳐야 하는 데이터 전달 과정에서 복잡성이 증가.
- **예측하기 어려운 상태 변화:** 상태가 여러 곳에서 변경될 경우 유지보수가 어려움.

1.3 상태관리의 유형

Flutter의 상태관리는 크게 **로컬 상태관리**와 **글로벌 상태관리**로 나뉜다.

- **로컬 상태관리(Local State Management):** 위젯 내부에서만 사용하는 상태를 관리.
  - `setState()`
  - `InheritedWidget`
  - `Provider`
- **글로벌 상태관리(Global State Management):** 앱 전반에 걸쳐 상태를 공유하고 관리.
  - `Provider`
  - `Riverpod`
  - `GetX`
  - `Bloc`

2. 상태관리와 생명주기

2.1 위젯 생명주기 (Widget Lifecycle)

Flutter의 `StatefulWidget`은 라이프사이클을 가진다. 위젯이 생성되고 제거되는 과정에서 여러 단계가 존재하며, 상태를 관리할 때 이 생명주기를 이해하는 것이 중요하다.

StatefulWidget의 생명주기 단계

1. `createState()` - State 객체를 생성한다.
2. `initState()` - State가 처음 생성될 때 한 번 호출된다.
3. `didChangeDependencies()` - 의존성이 변경될 때 호출된다.
4. `build()` - UI를 그리는 함수, 상태 변경 시 다시 호출된다.
5. `didUpdateWidget()` - 부모 위젯이 변경될 때 호출된다.
6. `deactivate()` - 위젯이 트리에서 제거될 때 호출된다.
7. `dispose()` - State가 완전히 제거될 때 호출되며, 리소스 정리 작업을 수행한다.

```
class MyWidget extends StatefulWidget {
  @override
  _MyWidgetState createState() => _MyWidgetState();
}
```

```dart
class _MyWidgetState extends State<MyWidget> {
  @override
  void initState() {
    super.initState();
    print('initState: 위젯이 처음 생성됨');
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    print('didChangeDependencies: 의존성이 변경됨');
  }

  @override
  Widget build(BuildContext context) {
    print('build: UI가 다시 빌드됨');
    return Scaffold(
      appBar: AppBar(title: Text('생명주기 예제')),
      body: Center(child: Text('Flutter 생명주기 테스트')),
    );
  }

  @override
  void dispose() {
    print('dispose: 위젯이 완전히 제거됨');
    super.dispose();
  }
}
```

3. setState()를 활용한 상태 업데이트

3.1 setState()

`setState()`는 `StatefulWidget`에서 상태가 변경될 때 UI를 다시 빌드하도록 하는 함수이다. 이 함수는 **상태가 변경될 때마다 반드시 호출해야 UI가 갱신된다.**

```dart
class CounterPage extends StatefulWidget {
  @override
  _CounterPageState createState() => _CounterPageState();
}

class _CounterPageState extends State<CounterPage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('setState 예제')),
      body: Center(
        child: Text(
          'Counter: $_counter',
          style: TextStyle(fontSize: 24),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
```

```
        child: Icon(Icons.add),
      ),
    );
  }
}
```

4. GetX 상태관리

4.0 GetX 개요

GetX는 Flutter에서 가장 가벼우면서도 강력한 상태관리 라이브러리 중 하나로, 간단한 API와 높은 성능을 제공한다. 또한 상태관리뿐만 아니라 의존성 주입(Dependency Injection)과 라우팅 기능도 포함되어 있다.

4.1 GetX 컨트롤러(GetxController)

컨트롤러란?

GetX 컨트롤러(`GetxController`)는 **상태를 관리**하는 역할을 하며, 앱의 비즈니스 로직과 UI 상태를 분리하는 데 사용된다.

주요 기능

1. **상태 관리** - UI 변경이 필요한 데이터를 관리
2. **의존성 관리** - `Get.put()` 을 사용해 전역에서 접근 가능
3. **메모리 관리** - `onInit()` , `onClose()` 등을 활용하여 생명주기 관리 가능

기본 컨트롤러 예제

```dart
import 'package:get/get.dart';

// GetxController를 상속받아 컨트롤러 생성
class CounterController extends GetxController {
  var count = 0.obs; // 반응형 변수

  void increment() {
    count++; // 상태 업데이트 시 UI 자동 변경
  }

  @override
  void onInit() {
    super.onInit();
    print("CounterController가 생성됨");
  }

  @override
  void onClose() {
    print("CounterController가 삭제됨");
    super.onClose();
  }
}
```

UI에서 컨트롤러 사용하기

```dart
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'counter_controller.dart';

class CounterPage extends StatelessWidget {
  final CounterController controller = Get.put(CounterController());
```

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("GetX 컨트롤러 예제")),
      body: Center(
        child: Obx(() => Text(
          "Counter: ${controller.count}",
          style: TextStyle(fontSize: 24),
        )),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: controller.increment,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- `Get.put(CounterController());` → 컨트롤러를 메모리에 등록
- `Obx(() => Text(...))` → 반응형 상태 변경 시 UI 자동 업데이트

4.2 Rx와 .obs

GetX에서는 **반응형 상태관리**를 위해 `Rx<Type>` 또는 `.obs` 키워드를 사용한다.

- `Rx<Type>` : 변수를 반응형으로 만들고 자동으로 UI를 업데이트할 수 있도록 한다.
- `.obs` : `Rx<Type>` 을 간단히 표현하는 단축 표기법이다.

예제:

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';

// 컨트롤러 클래스 정의
class CounterController extends GetxController {
  // 반응형 상태 변수 선언
  var count = 0.obs;

  // count 값을 증가시키는 함수
  void increment() {
    count++; // count.value++ 와 동일
  }
}

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(
      home: CounterPage(),
    );
  }
}

class CounterPage extends StatelessWidget {
  // Get.put을 사용하여 컨트롤러를 메모리에 등록
  final CounterController controller = Get.put(CounterController());

  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('GetX Counter Example')),
      body: Center(
        // Obx를 사용하여 상태 변화를 자동으로 감지하고 UI 업데이트
        child: Obx(() => Text(
              'Counter: ${controller.count}',
              style: TextStyle(fontSize: 24),
            )),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: controller.increment,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- `count = 0.obs;` 를 사용하면 `count` 값이 변경될 때 자동으로 UI가 업데이트된다.
- `count++` 을 실행하면 `count` 의 값이 증가하며 UI가 자동으로 갱신된다.
- `Obx` 를 사용하여 UI를 자동 업데이트할 수 있다.
- `Get.put(CounterController())` 를 사용하여 컨트롤러를 전역적으로 등록하고 관리할 수 있다.

4.3 GetX 상태관리 방식

1. 단순 상태(Simple State) (`GetBuilder`)

- 상태가 변경될 때 특정 위젯만 리빌드하도록 함.
- UI 업데이트가 자주 일어나지 않는 경우 사용.
- `Obx` 보다 가벼운 방식으로, 상태 변경 시 특정 위젯만 다시 그려짐.

예제:

```
class SimpleCounterController extends GetxController {
  int count = 0;

  void increment() {
    count++;
    update(); // GetBuilder를 사용하는 UI에서 변경 사항을 반영
  }
}

class SimpleCounterPage extends StatelessWidget {
  final SimpleCounterController controller = Get.put(SimpleCounterController());

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('GetBuilder Example')),
      body: Center(
        child: GetBuilder<SimpleCounterController>(
          builder: (_) => Text(
            'Counter: ${controller.count}',
            style: TextStyle(fontSize: 24),
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: controller.increment,
        child: Icon(Icons.add),
      ),
```

```
    );
  }
}
```

- `GetBuilder` 는 `update()` 메서드를 호출해야 UI가 업데이트됨.
- `Obx` 보다 적은 리소스를 사용하지만, 반응형 상태관리보다 명시적으로 UI 갱신을 호출해야 함.

2. 반응형 상태(Reactive State) (`Rx`)

- 상태 변경을 자동으로 감지하여 UI를 업데이트.
- `Rx<Type>` 또는 `.obs` 키워드를 사용하여 반응형 변수를 생성.

3. Mixin 기반 상태(Mixin State) (`GetX`, `Obx`)

- UI에 반응형 변수를 직접 바인딩하여 업데이트.
- `Obx` 위젯을 사용하여 상태 변화를 감지하고 자동 업데이트.

예제:

```
class MixinController extends GetxController {
  var count = 0.obs;

  void increment() {
    count++;
  }
}

class MixinCounterPage extends StatelessWidget {
  final MixinController controller = Get.put(MixinController());

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('GetX Mixin Example')),
      body: Center(
        child: GetX<MixinController>(
          builder: (_) => Text(
            'Counter: ${controller.count}',
            style: TextStyle(fontSize: 24),
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: controller.increment,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- `GetX<T>` 를 사용하여 UI가 컨트롤러의 상태 변화를 감지하도록 함.
- `Obx` 와 유사하지만, 컨트롤러를 직접 바인딩하여 더 명확한 구조를 제공함.

4.4 Get.put() vs Get.lazyPut() vs Get.find()

메서드	설명
`Get.put()`	인스턴스를 즉시 메모리에 생성하고 사용할 수 있도록 등록.
`Get.lazyPut(() => Controller())`	필요한 순간에만 인스턴스를 생성하여 메모리를 절약.
`Get.find<T>()`	이미 메모리에 등록된 컨트롤러를 찾아서 사용.

예제:

```dart
class MyController extends GetxController {
  var count = 0.obs;
}

void main() {
  Get.lazyPut(() => MyController()); // 컨트롤러를 필요할 때 생성
  final MyController controller = Get.find<MyController>(); // 등록된 컨트롤러 가져오기
  runApp(MyApp());
}
```

이처럼 `Get.put()`, `Get.lazyPut()`, `Get.find()`를 적절히 사용하여 메모리 관리와 성능을 최적화할 수 있다.