Flutter Gesture 학습

1. 개요

Flutter는 단일 코드베이스로 여러 플랫폼(iOS, Android, 웹, 데스크톱 등)에서 동작하는 애플리케이션을 개발할 수 있는 UI 프레임워크이다. Flutter 앱에서 사용자 입력은 보통 다음 단계를 통해 처리된다.

1. 사용자가 화면을 터치(또는 마우스 클릭 등)한다.
2. 플랫폼 운영체제에서 해당 이벤트를 수신하고, 이를 Flutter 엔진에 전달한다.
3. Flutter 엔진은 이벤트를 포인터(Pointer) 이벤트로 추상화한다.
4. UI 프레임워크 계층에서 히트 테스트(Hit Test)를 통해 이벤트가 어느 위젯에 도달했는지 확인한다.
5. 해당 위젯 혹은 해당 위젯이 내부적으로 사용하는 제스처 인식기(GestureRecognizer)들이 이벤트를 받는다.
6. 여러 제스처가 동시에 후보가 될 경우 **Gesture Arena**가 최종 승자를 결정한다.
7. 승리한 제스처 인식기의 콜백(onTap, onPan, onLongPress 등)이 실행되어, 개발자가 정의한 로직을 수행한다.

---

2. 제스처 인식 파이프라인

2.1 포인터 이벤트와 히트 테스트

Flutter에서 제스처가 인식되기 이전 단계에서는 사용자의 입력이 '포인터 이벤트'로 전달된다. 포인터 이벤트에는 손가락을 새로 화면에 댈 때 발생하는 `PointerDown`, 손가락을 떼는 `PointerUp`, 이동 시 발생하는 `PointerMove` 등이 있다. 이 포인터 이벤트는 Flutter 프레임워크의 히트 테스트 과정을 거쳐, 특정 위젯(혹은 그 위젯에 등록된 제스처 인식기)에게 전달된다.

히트 테스트는 '터치가 실제로 어느 위젯 영역과 맞닿았는가'를 판별하는 과정이며, 여러 위젯이 겹쳐 있다면 "가장 위(렌더 순서 상 최후)에서부터" 탐색한다. 각 위젯의 `HitTestBehavior` (예: `opaque`, `translucent`, `deferToChild`) 설정에 따라 이벤트가 어느 수준까지 전달될지 달라진다.

2.2 GestureDetector와 GestureRecognizer

가장 자주 사용하는 제스처 위젯은 `GestureDetector` 이라고 한다. 내부적으로 탭(Tap), 드래그(Pan), 롱프레스(LongPress) 등을 감지하기 위해 여러 종류의 `GestureRecognizer` 객체를 생성한다. 실제 로직은 다음과 같이 작동한다.

1. `GestureDetector` 가 히트 테스트를 통해 이벤트를 수신한다.
2. `TapGestureRecognizer`, `PanGestureRecognizer` 등 필요한 인식기가 이벤트를 분석한다.
3. 각 인식기는 자기 제스처로 볼 수 있다고 판단하면 **Gesture Arena**에 참여하여 '승리 의사(claim)'를 표시하거나, 적절치 않으면 '포기(reject)'한다.
4. Arena가 최종 승자를 결정하면, 그 인식기의 콜백(onTap, onPanStart 등)이 실행된다.

---

3. 내부 동작 원리: Gesture Arena

Flutter의 제스처 인식이 강력하고 유연한 이유는 **Gesture Arena**라는 충돌 처리 메커니즘 덕분이다. 여러 제스처가 동일 이벤트를 동시에 가져가려고 할 때, Arena가 중재해 최종 승자를 결정한다. 동작 원리는 크게 다음 단계로 요약된다.

1. **PointerDown 이벤트 발생**
   - 사용자가 화면을 누르는 순간, 해당 영역에 반응할 수 있는 모든 제스처 인식기가 Arena에 등록된다.
2. **포인터 이동·업 이벤트 수집**
   - 이후 발생하는 `PointerMove`, `PointerUp` 이벤트를 보고, 각 인식기는 자신이 감지할 제스처(탭, 드래그, 롱프레스 등)에 부합하는지 판별한다.
3. **Claim 또는 Reject**

- 제스처 인식기가 “이 이벤트는 내 제스처다”라고 확신하면 Arena에 claim을 선언한다.
- 반대로, 조건이 맞지 않아 포기하면 reject를 선언한다.

4. **승자 결정**
   - 여러 인식기가 동시에 claim하면, Flutter는 한 인식기가 결정적으로 “승리” 조건을 만족하는 시점에 해당 인식기를 최종 승자로 확정한다.
   - 나머지 인식기는 모두 패배 처리되고, 콜백이 호출되지 않는다.
5. **콜백 실행**
   - 승리한 인식기의 콜백이 호출되어 앱 로직을 수행한다.

동일 포인터(손가락 1개)에서 발생하는 이벤트는 하나의 Arena에서만 처리된다. 만약 두 개 손가락(두 개의 Pointer ID)을 동시에 사용할 경우, 각 Pointer ID마다 별도의 Arena가 동작하여 각기 다른 제스처가 병렬로 진행될 수 있다.

---

4. 여러 제스처 동시 사용 시 충돌 사례와 우선순위 제어

4.1 충돌 사례

1. **스크롤 vs 탭**
   - 스크롤 가능한 리스트(ListView) 영역 내부에 탭 가능한 요소(Button 등)가 있으면, 사용자가 살짝만 드래그해도 스크롤로 인식될지, 탭으로 인식될지 모호할 수 있다.
   - 이때, 일반적으로 리스트 스크롤이 우선권을 가지도록 구현한다.
2. **드래그 vs 롱프레스**
   - 길게 누르면서 살짝 움직이는 행동을 드래그로 볼지, 롱프레스로 볼지 판단이 애매할 수 있다.
   - 롱프레스 인식기의 설정 시간(대개 500ms가량)과 드래그의 최소 이동 거리(임계값)에 따라 결정된다.
3. **상위 위젯 vs 하위 위젯**
   - 부모에 `GestureDetector`가 있고, 자식에도 별도의 `GestureDetector`가 있을 경우, 부모가 이벤트를 먼저 소모하여 자식이 이벤트를 받지 못할 수 있다.
   - 이때, `behavior: HitTestBehavior.translucent` 또는 `deferToChild` 등을 적절히 설정하여 이벤트 흐름을 조절해야 한다.

4.2 우선순위 제어 기법

1. **HitTestBehavior 조정**
   - `GestureDetector(behavior: HitTestBehavior.xxx)` 형태로 이벤트가 어느 경로로 전달되는지를 명시할 수 있다.
   - 예: `HitTestBehavior.translucent`를 사용하면, 위젯 영역을 터치해도 이벤트가 하위 위젯까지 전달될 수 있다.
2. **임계값(Threshold) 조정**
   - `PanGestureRecognizer` 등은 일정 거리 이상 움직였을 때 claim을 선언한다. 이 거리를 크게 잡으면 ‘살짝 움직인 것’은 탭으로 인식될 가능성이 높아진다.
   - 반대로, 작게 잡으면 조금만 드래그해도 곧바로 onPan 이벤트가 발생한다.
3. **RawGestureDetector와 커스텀 Recognizer**
   - Flutter가 기본 제공하는 인식 로직과 달리, `RawGestureDetector`를 통해 직접 `GestureRecognizer`를 만들거나 세밀한 우선순위 로직을 설정할 수 있다.
   - 예: 특정 상황에서는 탭 인식기가 매우 빨리 claim하도록 만들고, 다른 상황에서는 반대로 드래그 인식기가 우선 claim하도록 설정할 수 있다.
4. **위젯 트리 구조의 단순화**
   - 여러 단계로 중첩된 위젯에서 제스처가 겹치면 충돌이 자주 발생한다. 필요한 곳에만 `GestureDetector`를 배치하거나, 이벤트 핸들링 로직을 분리해서 관리하면 갈등을 줄일 수 있다.

---

5. 예시: 기본 제스처 처리

다음 예시는 단일 `GestureDetector`를 사용해 탭, 롱프레스, 드래그 이벤트를 간단히 처리하는 방법을 보여준다.

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

class BasicGestureDemo extends StatefulWidget {
  @override
  _BasicGestureDemoState createState() => _BasicGestureDemoState();
}

class _BasicGestureDemoState extends State<BasicGestureDemo> {
  Color _color = Colors.blue;
  Offset _offset = Offset(100, 100);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Basic Gesture Demo')),
      body: GestureDetector(
        onTap: () {
          setState(() {
            // 탭 시 색상 변경
            _color = _color == Colors.blue ? Colors.red : Colors.blue;
          });
        },
        onLongPress: () {
          setState(() {
            // 길게 누르면 초록색으로 변경
            _color = Colors.green;
          });
        },
        onPanUpdate: (details) {
          setState(() {
            // 드래그(팬) 이동
            _offset += details.delta;
          });
        },
        child: Stack(
          children: [
            Positioned(
              left: _offset.dx,
              top: _offset.dy,
              child: Container(
                width: 100,
                height: 100,
                color: _color,
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

- `onTap` : 탭 제스처에 반응하여 박스 색상을 빨강/파랑으로 토글한다.
- `onLongPress` : 길게 누르면 박스 색상을 초록색으로 변경한다.
- `onPanUpdate` : 드래그할 때마다 현재 위치에 `details.delta`를 더해 박스를 이동시킨다.

해당 예시는 매우 간단하므로 충돌 처리 이슈가 크게 발생하지 않는다. 다만, 사용자가 누르다가 살짝만 움직이면 `onPanUpdate`가 먼저 claim하여 탭 제스처가 무시될 수 있다. 이러한 부분은 Flutter 내부의 임계값 로직에 따라 결정된다.

6. 예시: 여러 제스처 간 충돌 처리

이번에는 부모(전체 화면)가 드래그를 감지하고, 자식(버튼 위젯)이 탭을 감지하는 상황을 가정한다. 사용자가 버튼을 누르려고 했으나 약간만 손가락이 움직여도 부모 드래그로 잡힐 수 있으므로, 원하는 대로 동작하도록 `behavior` 등을 설정해 볼 수 있다.

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

class ParentChildGestureDemo extends StatefulWidget {
  @override
  _ParentChildGestureDemoState createState() => _ParentChildGestureDemoState();
}

class _ParentChildGestureDemoState extends State<ParentChildGestureDemo> {
  double _dragPosition = 0.0;
  String _message = '버튼이 눌리지 않음';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Parent-Child Gesture Demo')),
      body: GestureDetector(
        // 부모의 드래그 감지
        onPanUpdate: (details) {
          setState(() {
            _dragPosition += details.delta.dx;
          });
        },
        // hitTestBehavior를 translucent로 설정하면,
        // 자식 위젯 또한 이벤트를 받을 수 있게 된다.
        behavior: HitTestBehavior.translucent,
        child: Stack(
          children: [
            Positioned(
              left: _dragPosition,
              top: 200,
              child: GestureDetector(
                onTap: () {
                  setState(() {
                    _message = '자식 위젯(버튼)이 눌림';
                  });
                },
                child: Container(
                  width: 150,
                  height: 60,
                  color: Colors.blue,
                  alignment: Alignment.center,
                  child: Text('Tap Me', style: TextStyle(color: Colors.white)),
                ),
              ),
            ),
            Positioned(
              top: 400,
              left: 20,
              child: Text(_message, style: TextStyle(fontSize: 20)),
            ),
          ],
        ),
      ),
    );
```

```
  }
}
```

- 부모 위젯(`GestureDetector`)에서 `onPanUpdate` 를 등록하고, `behavior` 를 `HitTestBehavior.translucent` 로 설정하였다.
- 자식 위젯(파란색 버튼)에도 별도의 `GestureDetector` 가 있으며, onTap을 감지한다.
- 이 설정 덕분에 부모 위젯이 드래그를 처리하는 동시에 자식 위젯에서 탭 이벤트가 정상 처리된다.
- 만약 `behavior` 를 설정하지 않거나 `HitTestBehavior.opaque` 로 설정하면, 부모가 이벤트를 가로채서 자식의 탭 이벤트가 동작하지 않을 수 있다.

---

7. RawGestureDetector를 이용한 확장 예시

`RawGestureDetector` 를 사용하면, Flutter가 기본 제공하는 제스처 인식기가 아닌 커스텀 인식기를 등록하거나, `GestureRecognizer` 의 우선순위 제어를 직접 정의할 수 있다. 아래는 단순화된 예시로, 탭(TapGestureRecognizer)과 수평 드래그(HorizontalDragGestureRecognizer)를 동시에 등록해, 각 인식기가 claim하는 로직을 살펴보는 코드이다.

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

class RawGestureDemo extends StatefulWidget {
  @override
  _RawGestureDemoState createState() => _RawGestureDemoState();
}

class _RawGestureDemoState extends State<RawGestureDemo> {
  String _gestureText = '아직 제스처 없음';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('RawGestureDetector Demo')),
      body: RawGestureDetector(
        gestures: {
          // TapGestureRecognizer 등록
          TapGestureRecognizer: GestureRecognizerFactoryWithHandlers<
              TapGestureRecognizer>(
            () => TapGestureRecognizer(),
            (TapGestureRecognizer instance) {
              instance.onTap = () {
                setState(() {
                  _gestureText = '탭 제스처 발생';
                });
              };
            },
          ),
          // HorizontalDragGestureRecognizer 등록
          HorizontalDragGestureRecognizer:
              GestureRecognizerFactoryWithHandlers<
                  HorizontalDragGestureRecognizer>(
            () => HorizontalDragGestureRecognizer(),
            (HorizontalDragGestureRecognizer instance) {
              instance
                ..onStart = (details) {
                  setState(() {
                    _gestureText = '수평 드래그 시작';
                  });
                }
                ..onUpdate = (details) {
                  setState(() {
```

```
                    _gestureText =
                        '드래그 중 (distance: ${details.primaryDelta?.toStringAsFixed(2)})';
                  });
                }
                ..onEnd = (details) {
                  setState(() {
                    _gestureText = '수평 드래그 종료';
                  });
                };
            },
          ),
        },
        child: Center(
          child: Text(
            _gestureText,
            style: TextStyle(fontSize: 24),
          ),
        ),
      ),
    );
  }
}
```

- `RawGestureDetector`의 `gestures` 매개변수에, 여러 `GestureRecognizerFactoryWithHandlers`를 통해 인식기를 등록한다.
- 여기에서는 `TapGestureRecognizer`와 `HorizontalDragGestureRecognizer`를 동시에 등록하여, 탭과 수평 드래그를 구분한다.
- 실제 사용 시, Flutter의 내부 로직에 따라 먼저 claim하는 제스처가 최종 승자가 될 것이다(스크롤이 일정 거리 이상이면 드래그가 우선, 이동이 거의 없으면 탭으로 인식).
- 필요한 경우 `GestureRecognizer`를 상속받아 `acceptGesture`나 `rejectGesture` 메서드를 오버라이드하여 직접 우선순위를 정할 수도 있다.

---

8. 실제 구현 시 고려 사항

1. **사용자 기대 행동에 부합**
   - 스크롤 영역 내부 버튼, 길게 누르면 메뉴 열기 vs 드래그 등 사용자 경험을 먼저 고려해야 한다.
   - 필요하면 임계값(드래그 거리, 롱프레스 시간)을 조정해 의도한 시나리오에 맞춘다.
2. **위젯 트리 구조 간소화**
   - 부모와 자식, 혹은 형제 위젯들에서 중복으로 제스처를 감지하면 충돌이 잦으므로, 불필요한 `GestureDetector` 중복을 피한다.
3. **HitTestBehavior 설정 확인**
   - 이벤트가 상위, 하위 위젯 중 어디에서 처리되도록 유도할지 명확히 설정한다.
   - 기본값(`deferToChild`)으로 충분하지 않을 경우, `translucent`나 `opaque`로 바꿔서 원하는 흐름을 유도한다.
4. **성능**
   - 일반적인 앱 규모에서는 크게 문제되지 않으나, 지나치게 많은 `GestureDetector`나 복잡한 커스텀 인식기를 동시에 등록하면 성능이 저하될 수 있다.
5. **테스트 환경 다양성**
   - 시뮬레이터/에뮬레이터뿐 아니라 실제 기기, 다양한 화면 크기와 플랫폼에서 테스트하여 사용자가 실제로 어떤 제스처를 주로 사용하는지 파악한다.

---

9. 결론

Flutter의 제스처 시스템은 포인터 이벤트를 기반으로, 내부적으로 **Gesture Arena** 메커니즘을 사용하여 여러 제스처 인식기가 동일 이벤트를 claim하려 할 때 충돌을 해결한다. 이를 통해 탭, 드래그, 롱프레스, 스크롤, 스와이프 등 다양한 상호작용을 유연하게 구현할 수 있다.

- 기본적으로는 `GestureDetector`를 사용하여 `onTap`, `onPan`, `onLongPress` 등 각종 콜백을 등록하면 된다.
- 이벤트 전달 우선순위가 문제될 경우, `HitTestBehavior` 설정, 드래그 임계값 조정, `RawGestureDetector` 등을 활용하여 원하는 동작을 세밀하게 제어할 수 있다.
- **부모-자식 위젯 간의 이벤트 충돌**도 `behavior` 옵션이나 위젯 트리 설계를 통해 해결 가능하다.