

Compose에서의 안정성과 성능

Stable

- 원시 타입(문자열 포함) : 예상치 못한 변화가 없으므로 안정적
- 함수(람다) : 예측 가능한 동작으로 인해 안정적 (단, 불안정한 값을 캡처하는 경우 해당 함수 또한 불안정하다고 간주)
- 클래스 : 불변 속성을 가지고 있는 data class는 안정적이며, @Stable 혹은 @Immutable 어노테이션이 추가된 클래스 또한 안정적인 것으로 간주

Stable

```
data class User(  
    val id: Int,  
    val name: String,  
)
```

Unstable

- 인터페이스 혹은 추상 클래스 : List, Map과 같은 인터페이스 혹은 Any와 같은 추상 클래스는 컴파일 타임에 구현을 보장할 수 없기 때문에 불안정
- 가변 프로퍼티를 포함한 클래스 : 최소 하나의 가변 프로퍼티 혹은 불안정한 타입을 포함한 클래스

Unstable

```
data class User(  
    val id: Int,  
    var name: String,  
)
```

Smart Recomposition

위와 같은 타입 추론을 통해 Compose Runtime이 불필요한 Recomposition을 스킵하게 되는데,

이를 Smart Recomposition이라고 한다.

- 안정성에 따른 결정 : 매개변수가 안정적이며 값이 변경되지 않은 경우 (equals()가 true) Recomposition을 건너뛰고, 매개변수가 불안정하거나 안정적이지만 값이 변경된 경우 (equals()가 false) Recomposition을 시작하여 UI를 무효화하고 다시 그린다.
- 동등성 검사 : equals()를 통한 비교는 해당 타입이 안정적일 때만 수행하며, Composable 함수의 입력값이 변경될 때마다 해당 타입의 equals()를 사용하여 이전 값과 비교한다.

Composable 함수 추론

Compose Compiler는 Composable 함수를 여러 항목으로 분류하여 실행을 최적화한다.

-> Restartable, Skippable, Moveable, Replaceable

Recomposition에 직접적인 영향을 미치는 것은 Restartable, Skippable

Restartable

- Compose Compiler에 의해 추론된 타입으로, Recomposition 과정의 기반을 제공한다.
- 입력값 혹은 상태가 변경되면 Runtime이 UI 갱신을 위해 해당 함수를 실행한다.
- 대부분의 Composable 함수는 기본적으로 Restartable로 취급된다.

-> Unit이 아닌 값을 반환하는 Composable. inline으로 호출되는 Composable은 Restartable X







```
@Composable
inline fun Box(
    modifier: Modifier = Modifier,
    contentAlignment: Alignment = Alignment.TopStart,
    propagateMinConstraints: Boolean = false,
    content: @Composable BoxScope.() -> Unit,
) {
    val measurePolicy = maybeCachedBoxMeasurePolicy(contentAlignment, propagateMinConstraints)
    Layout(
        content = { BoxScopeInstance.content() },
        measurePolicy = measurePolicy,
        modifier = modifier,
    )
}
```


Skippable

- Smart Recomposition에 의해 설정된 특정 조건 하에 Recomposition을 건너뛸 수 있다.
- 계층 구조의 최상위에 위치한 Composable 함수의 성능을 향상시키는 데 중요하다.
- Composable 함수는 동시에 Restartable과 Skippable이 될 수 있다.

Compose Metrics

```
kotlinOptions {  
    jvmTarget = "21"  
    freeCompilerArgs +=  
        listOf(  
            "-p",  
            "plugin:androidx.compose.compiler.plugins.kotlin:metricsDestination=${  
                rootProject.file(  
                    ".",  
                ).absolutePath  
            }/compose-metrics",  
        )  
    freeCompilerArgs +=  
        listOf(  
            "-p",  
            "plugin:androidx.compose.compiler.plugins.kotlin:reportsDestination=${  
                rootProject.file(  
                    ".",  
                ).absolutePath  
            }/compose-reports",  
        )  
}
```

- ✓  compose-metrics
 -  app_debug-module.json
- ✓  compose-reports
 -  app_debug-classes.txt
 -  app_debug-composables.csv
 -  app_debug-composables.txt

Compose Metrics

```
{  
  "skippableComposables": 69,  
  "restartableComposables": 81,  
  "readonlyComposables": 0,  
  "totalComposables": 82,
```

```
stable class CardUiModel {  
    stable val number: String  
    stable val expiredDate: String  
    stable val ownerName: String  
    stable val issuingBank: IssuingBank  
    <runtime stability> = Stable  
}  
  
stable class CardAdditionActivity {  
    <runtime stability> = Stable  
}  
  
stable class CardAdditionUiState {  
    stable val cardNumber: CardNumber  
    stable val expiredDate: ExpiredDate  
    stable val ownerName: String  
    stable val password: CardPassword  
    stable val issuingBank: IssuingBank  
    stable val isValidCard: Boolean  
    stable val isDateError: Boolean  
    <runtime stability> = Stable  
}
```

@Immutable

- 해당 어노테이션이 추가된 클래스는 초기 생성 후 완전한 불변임을 보장한다.
- 이는 val 키워드 혹은 다른 방법보다 더 강력한 방법
- 아래와 같은 규칙을 준수해야 함
 - 모든 프로퍼티는 val
 - 커스텀 setter를 피하고 프로퍼티가 가변성을 지원하지 않도록 함
 - 모든 프로퍼티가 불변으로 간주되어도 되는지 확인

```
@Immutable
data class User(
    val id: Int,
    // List가 Unstable 타입이기 때문에 User 클래스는 Unstable로 간주됨
    // 하지만 Immutable 어노테이션을 사용하면 된다.
    val items: List<String>,
)
```

@Stable

- 프로퍼티는 불변이지만 클래스 자체가 안정적으로 간주되지 않는 클래스에 적합하다.
- @Immutable보다는 강도가 낮음
- 동일한 입력에 대해 항상 동일한 결과를 반환한다는 의미

```
@Stable
public interface State<out T> {
    public val value: T
}
```

```
@Stable
public interface MutableState<T> : State<T> {
    override var value: T

    public operator fun component1(): T

    public operator fun component2(): (T) -> Unit
}
```

@Immutable vs @Stable

```
@Immutable
data class User(
    val id: Int,
    val nickname: String,
    val profileImages: List<String>,
)
```

- @Immutable은 주로 데이터 모델에 사용된다.
- 하지만 오용하면 원하는 곳에서 비 갱신이 제대로 이루어지지 않을 수 있으니 조심

@Immutable vs @Stable

```
@Stable
interface UiState<T> {
    val value: T
    1 Usage
    val exception: Throwable?

    val isSuccess: Boolean
    get() = exception == null
}
```

- 주로 여러 구현 가능성을 제공하며 내부적으로 변경 가능한 상태를 가질 수 있는 인터페이스에 사용

@Immutable vs @Stable

- 두 어노테이션은 목적이 다르지만, 내부적으로는 차이가 없다.
- 그래서 @Stable을 사용해야 하는 곳에 @Immutable을 사용해도 문제가 생기지 않는다.
- 왜?