Hilt를 사용한 의존성 주입

2016010873 박정욱

목차

- 의존성 주입(Dependency Injection)이란?
- kapt를 사용하여 Kotlin에서 Java Annotation 처리하기
- 컴파일-타임 정적 종속성 주입 프레임워크 Dagger
- Hilt를 사용하여 Android에서 Dagger Component 구축
- 실제 적용 사례

의존성 주입(Dependency Injection)이란?

```
class B{ … }
```

```
class A {
    private val b: B

    init {
        b = B()
    }
}
```

- 객체 간의 결합도가 높은 코드

1. B의 생성 책임이 A에게 있음

2. A는 **구체적으로 구현된** B class에 의존적임

이런 코드는 유지보수가 **어려움**!

의존성 주입(Dependency Injection)이란?

```
interface B { … }

class BContrete :  B { … }



class A {
    private val b: B

    init {
        b = BConCrete()
    }
}
```

- 의존성 전이
  - B 객체의 구현이 변경될 시, A 객체 또한 변경이 불가피할 수 있음

- 이를 피하기 위해, **컴파일 시간 의존성**을 **실행 시간 의존성**으로 변경!

의존성 주입(Dependency Injection)이란?

```
interface B {}

class BContrete :  B{}


class A(private val b: B) {}


fun someFunction() {

    val b = BConcrete()

    val a = A(b)
}
```

- B의 생성 책임이 A에게 있으므로, 의존성 전이는 그대로 존재
- B의 생성 책임을 A의 **외부**로!
- B를 구현하는 **어떠한** 객체든지 의존성으로 제공할 수 있음!

의존성 주입(Dependency Injection)이란?

```
interface B {}

class BContrete :  B{}


class A(private val b: B) {}


fun someFunction() {

    val b = BConcrete()

    val a = A(b)
}
```

- 특정 객체의 의존성을 외부에서 **주입**(또는 **제공**)해주는 형태
- 결합도 ↓ , 재사용성 ↑
- 테스트가 용이해짐

의존성 주입(Dependency Injection)이란?

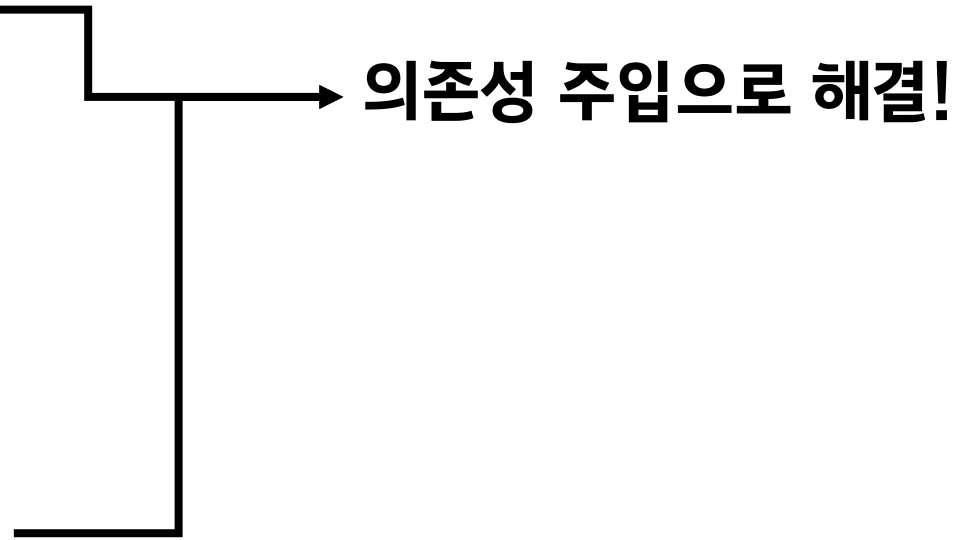
Single **R**esponsibility **P**rinciple, **SRP**

Open-**C**losed **P**rinciple, **OCP**

Liskov **S**ubstitution **P**rinciple, **LSP**

Interface **S**egragation **P**rinciple, **ISP**

Dependency **I**nversion **P**rinciple, **DIP**



kapt를 사용하여 Kotlin에서 Java Annotation 처리하기

- Dagger는 Java로 작성된 프레임워크
- Kotlin은 Java와 100% 호환되지만, Annotation의 경우 별도의 변환 작업이 필요
- 이를 위해, kapt 플러그인 적용 필요

kapt를 사용하여 Kotlin에서 Java Annotation 처리하기

build.gradle (:app)

You can use the Project Structure dialog to view and edit your project configuration

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'com.google.gms.google-services'
    id 'org.jetbrains.kotlin.kapt'
    id 'dagger.hilt.android.plugin'
}
```

```
// Hilt Dependencies
implementation "com.google.dagger:hilt-android:$hilt_version"
kapt "com.google.dagger:hilt-compiler:$hilt_version"
```

컴파일-타임 정적 종속성 주입 프레임워크 Dagger

- 의존성 주입을 위해, 외부에서 객체를 생성하여 넘겨주는 작업 필요
- 개발자가 일일이 신경쓰기엔 복잡하고, 많은 상용구 코드 필요
- **정적**, **컴파일-시간** 의존성 주입 프레임워크인 Dagger를 적용!

컴파일-타임 정적 종속성 주입 프레임워크 Dagger

- Dagger는 정적, 컴파일 시간 프레임워크
- Interface를 컴파일 시간 종속성으로 가질 경우에 어떠한 구현체를 제공받을 지 알 수 없어 의존성 주입 코드 생성 불가
- 해당 Interface에 대한 구현체 정보를 별도로 작성하여 제공!

컴파일-타임 정적 종속성 주입 프레임워크 Dagger

```kotlin
@Module
@InstallIn(SingletonComponent::class)
object RoutineDataSourceModule {

    @Singleton
    @Provides
    fun provideRoutineDataSourceImplWithRoom(routineDao: RoutineDao):
            RoutineLocalDataSource = RoutineLocalDataSourceImplWithRoom(routineDao)

    @Singleton
    @Provides
    fun provideRoutineDataSourceImplWithRealtime(
        @RoutineDatabaseRef routineDbRef: DatabaseReference,
        @TodoDatabaseRef todoDbRef: DatabaseReference
    ):
            RoutineRemoteDataSource = RoutineDataSourceImplWithRealtime(routineDbRef, todoDbRef)
}
```

Hilt를 사용하여 Android에서 Dagger Component 구축

- Android의 구성요소 대부분은 생성 책임이 프레임워크에 존재
- 개발자가 직접 생성자를 호출하여 생성할 수 없음
- AAC ViewModel의 경우, 이전에는 ViewModel을 생성하는 Factory 인터페이스를 직접 구현하여야 했음
- Google이 제공하는 Hilt를 이용하여, 밑작업을 최소화!

실제 적용 사례

```
class RoutineLocalDataSourceImplWithRoom @Inject constructor(
    private val dao: RoutineDao
    ) : RoutineLocalDataSource {
```

```
class RoutineRepositoryImpl @Inject constructor(
    private val localDataSource: RoutineLocalDataSource,
    private val remoteDataSource: RoutineRemoteDataSource,
    @IoDispatcher private val ioDispatcher: CoroutineDispatcher
    ) : RoutineRepository {
```

```
class GetRoutineListUseCase @Inject constructor(
    private val repository: RoutineRepository
    ) {
    operator fun invoke(): Flow<List<Routine>> {
        return repository.getAllRoutinesFromLocal()
    }
}
```

```
@HiltViewModel
class RoutineMainViewModel @Inject constructor(
    getUsedTodoListUseCase: GetUsedTodoListUseCase,
    private val updateUsedTodoUseCase: UpdateUsedTodoUseCase
) : ViewModel() {
```

```
@AndroidEntryPoint
class RoutineMainFragment : Fragment() {
```

정리

- 의존성과 SOLID 법칙
- 의존성 주입이 필요한 이유
- kapt를 이용하여 Java Annotation을 Kotlin으로 처리
- Dagger, Hilt