Dagger, Hilt 비교

[Overview]

네트워크를 통해 이미지를 가져와서 화면에 표시해 주는 예제를 통해 의존성 주입 라이브러리인 Dagger2와 Hilt의 차이점에 대해서 간략하게 살펴보겠습니다.

[Dagger2]

먼저 Component 인터페이스를 생성해야 합니다. Component는 객체를 생성하고 제공하는 역할을 합니다.

이후에 설명할 @Module에서 생성한 객체를 @Inject를 통해 주입할 수 있도록 연결해 줍니다.

AndroidSupportInjectionModule를 모듈에 추가함으로써, dagger.android에서 제공하는 클래스를 사용할 수 있습니다.

ApplicationComponent

```
@Singleton
@Component(modules = [AndroidSupportInjectionModule::class, NetworkModule::class, ActivityBuilder::
class])
interface ApplicationComponent {
    @Component.Builder
    interface Builder {
        @BindsInstance
        fun application(application: Application): Builder

        fun build(): ApplicationComponent
    }

    fun inject(app: RootApplication)
}
```

네트워크 통신을 위한 모듈 생성 과정입니다. @Provides로 객체를 생성하며 @Singleton으로 어디서나 동일한 객체를 제공합니다.

NetworkModule

```
@Module
class NetworkModule {

    @Provides
    @Singleton
    fun provideApiService(okHttpClient: OkHttpClient): ApiService {
        ...
    }

    @Provides
    @Singleton
    fun provideRepository(apiService: ApiService): Repository = ImageRepository(apiService)
}
```

아래 Activity 설정 부분은 의존성 주입을 사용할 Activity를 Dagger에게 알려주기 위해 필요합니다.

각각의 Activity에 대해 별도의 모듈을 만든다면 클래스가 너무 많아지기 때문에, Activity와 관련된 모듈을 하나로 모으기 위하여 다음과 같은 ActivityBuilder모듈을 만들었습니다.

이 ActivityBuilder는 위 ApplicationComponent의 @Component에 추가했습니다.

Activity의 경우 AndroidInjector를 상속한 Subcomponent와 Subcomponent.Factory를 만들어야 하는데 이를 자동으로 해주는것이 @ContributesAndroidInjector입니다.

ActivityModule

```
@Module
abstract class ActivityBuilder {

    @ContributesAndroidInjector(modules = [MainModule::class])
    abstract fun bindMainActivity(): MainActivity

    @ContributesAndroidInjector(modules = [ImageModule::class])
    abstract fun bindImageActivity(): ImageActivity
    ...
}
```

Application 클래스에서 DaggerApplicationComponent를 초기화 해줍니다.

프로젝트를 빌드 하면 위에서 설명한 ApplicationComponent는 DaggerApplicationComponent로 자동 생성 됩니다.

HasActivityInjector를 implement 해주면 AndroidInjector를 리턴하는 메서드를 작성할 수 있는데 DispatchingAndroidInjector변수에 @Inject를 붙여 리턴하고 있습니다.

이렇게 하면 모듈로 등록해 둔 Activity들이 연결됩니다.

Application

```
class RootApplication: Application(), HasActivityInjector {
    @Inject
    lateinit var activityDispatchingAndroidInjector: DispatchingAndroidInjector<Activity>

    override fun activityInjector(): AndroidInjector<Activity> = activityDispatchingAndroidInjector

    override fun onCreate() {
        super.onCreate()
        DaggerApplicationComponent.builder()
                        .application(this)
                        .build()
                        .inject(this)
    }
}
```

다음은 ViewModel 작성입니다.

ViewModel의 경우 기본 생성자 외 매개변수가 있는 생성자로 객체를 생성하려면, 다음에 설명할 ViewModelFactory를 통해서 생성해야 합니다.

ViewModel

```
class MainViewModel(val repository: Repository) : ViewModel() {

    @SuppressLint("CheckResult")
    fun fetchImages(query: String, page: Int, size: Int) {
        repository.fetchImages(query, page, size)
        ...
    }
}
```

이 예제에서 ViewModelFactory는 매개변수로 Repository를 가진 ViewModel을 생성합니다.

@Inject constructor를 사용하면 Repository는 외부에서 주입 받을 수 있고 ViewModelFactory 역시 외부에 주입이 가능합니다.

ViewModelFactory

```
class MainViewModelFactory @Inject constructor(private val repository: Repository) : ViewModelProvider.
Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return MainViewModel(repository) as T
    }
}
```

이제 기본설정은 끝났습니다.

실제 사용하는 곳에서는 AndroidInjection.inject(this)를 추가하고 @Inject 어노테이션을 사용하여 주입 받으면 됩니다.

Activity

```
class MainActivity : AppCompatActivity() {

    @Inject
    lateinit var viewModelFactory: MainViewModelFactory

    private val mainViewModel: MainViewModel by lazy{
        ViewModelProviders.of(this, viewModelFactory)[MainViewModel::class.java]
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        AndroidInjection.inject(this)
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        mainViewModel.fetchImages("", 1, 30)
    }
}
```

[Hilt]

Hilt를 사용하고자 하는 앱은 @HiltAndroidApp으로 주석이 지정된 Application 클래스를 추가해야 합니다.

@HiltAndroidApp이 기존 Dagger2에서 ApplicationComponent를 정의하고 Application 내 코드로 초기화하던 과정에 해당합니다.

Application 구현

```
@HiltAndroidApp
class RootApplication: Application() {
    ...
}
```

Module은 @Provides나 @Binds로 생성합니다.

기존 Dagger2에서는 Module을 생성한 후 개발자가 정의한 Component에 직접 include 해주어야 했습니다.

Hilt에서는 @InstallIn을 사용하여 표준 Component를 설정해 주면 됩니다.

표준 Component에는 ApplicationComponent, ActivityComponent, FragmentComponent, ViewComponent, ServiceComponent 등이 있습니다.

기본적으로 Hilt는 @Inject를 요청할 때마다 새 인스턴스를 생성하므로, 동일한 인스턴스를 원하는 경우는 Component별로 정의된 Scope을 지정해 주면 됩니다.

ApiModule

```
@Module
@InstallIn(ApplicationComponent::class)
object ApiModule {

    @Singleton
    @Provides
    fun provideApiService(okHttpClient: OkHttpClient): ApiService {
        ...
    }

    @Singleton
    @Provides
    fun provideRepository(apiService: ApiService): Repository = ApiRepository(apiService)
}
```

Hilt는 ViewModel과 WorkManager같은 Jetpack 라이브러리를 지원합니다.

ViewModel의 경우 Dagger2에서는 별도 ViewModelFactory를 생성해야 했지만 Hilt에서는 이미 내부에 정의 되어 있어 @HiltViewModel을 붙여주는 걸로 주입이 가능합니다.

ViewModel

```
@HiltViewModel
class MainViewModel @Inject constructor(
        val repository: Repository
): ViewModel() {

    fun fetchImages(searchText: String) {
        repository.fetchImages(searchText)
        ...
    }
}
```

Hilt는 @AndroidEntryPoint 주석이 있는 클래스에 종속 항목을 제공할 수 있습니다.

Android 클래스에 @AndroidEntryPoint로 주석을 지정하면 이 클래스에 종속된 Android 클래스에도 주석을 지정해야 합니다.

예를 들어 프래그먼트에 주석을 지정하면 이 프래그먼트를 사용하는 Activity에도 주석을 지정해야 합니다.

Activity

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    private val viewModel by viewModels<MainViewModel>()
    ...
}
```

[Conclusion]

Hilt는 기존 Dagger2에서 의존성 주입을 위해 필요했던 많은 보일러 플레이트 코드를 어노테이션 추가로 간결하게 처리할 수 있게 하였습니다.

물론 학습할 양이 적진 않지만 Dagger2에 비해 러닝커브도 상당히 낮아진 걸로 보입니다.

현재 저희 파트에서는 의존성 주입 방법으로 Dagger2를 사용하고 있는데 신규 프로젝트 진행 시에는 Hilt를 사용해보는 것도 좋을 듯 합니다.