DI (Dagger2, Koin, Hilt)

Android Study

1110 - 김연우

안녕하세요 안드로이드 스터디 DI 주제를 하게 된 김연우 입니다.
Dependency Injection, 객체지향 프로그래밍을 하다보면 한 번 쯤 듣게 되는 용어 중 하나인데요.
이번 글에서는 의존성 주입이 무엇이고 어떤 배경으로 생겨난 개념인지, 왜 필요한지 간단히 정리해 설명하겠습니다. Dagger2 에 대해 사용법과 예제를 알려드리겠습니다.

DI?

문제점

```java
// Programmer.java
class Programmer {
    private Coffee coffee;

    public Programmer() {
        this.coffee = new Coffee();
    }

    public startProgramming() {
        this.coffee.drink(); // 일단 마시고 시작하자
        ...
    }
}
```

문제점

- 개발을 하다 보면 코드에 의존성이 생기기 마련입니다. 그럼 의존성은 무엇이고, 왜 생겨나는 걸까요?

위 코드와 같이 프로그래머 클래스에서 Start프로그래밍 함수가 호출되기 위해서는 커피 클래스를 필요로 합니다. 이것을 프로그래머 클래스는 커피 클래스의 의존성을 가진다. 라고 합니다. 이와 같이 코드를 설계하였을 때, 코드의 재활용성이 떨어지고, 위 예제에서 커피 클래스가 수정 되었을 때

즉, 결합도(coupling)가 높아지게 됩니다.

DI?

DI(의존성 주입)를 해야하는 이유

- Unit Test가 용이해진다.
- 코드의 재활용성을 높여준다.
- 객체 간의 의존성(종속성)을 줄이거나 없앨 수 있다.
- 객체 간의 결합도를 낮추면서 유연한 코드를 작성할 수 있다.

```
// Programmer.java
class Programmer {
    private Coffee coffee;

    // 그 날 마실 커피를 고를 수 있게된 개발자
    public Programmer(Coffee coffee) {
        this.coffee = coffee;
    }

    public startProgramming() {
        this.coffee.drink();
        ...
    }
}
```

```
class Coffee {...} // interface로 설계할 수도 있다

// Coffee 클래스를 상속
class Cappuccino extends Coffee {...}
class Americano extends Coffee {...}

// Programmer.java
class Programmer {
    private Coffee coffee;

    public Programmer() {
        this.coffee = new Cappuccino(); // 직접 수정
        // 또는
        this.coffee = new Americano(); // 직접 수정
    }

    ...
}
```

Unit Test란 프로그래밍을 할 때 소스코드의 특정 모듈(메서드)이 의도된 대로 정확히 작동하는지 검증하는 절차를 말합니다.
다시 말해 작성한 모든 메서드들에 대해서 테스트 케이스를 작성하는 것을 의미합니다.

만약 DI를 사용하지 않고 커피 클래스의 상속을 받은 카푸치노나 아메리카노 클래스를 사용해야 한다면 다음과 같이 직접 수정해 줘야 합니다.
극단적으로, 만약 커피 클래스를 사용하는 클래스가 100개라면 그 중 카푸치노가 필요한 클래스가 있다면 직접 수정해줘야겠죠? (바로 새벽자습)
이것은 굉장히 비효율적이라고 할 수 있습니다.
의존성 주입을 이용한다면 아래와 같이 할 수 있습니다.

위와 같이 필요한(의존하는) 클래스를 직접 생성하는 것이 아닌, 주입해줌으로써 객체간의 결합도를 줄이고 좀 더 유연한 코드를 작성할 수 있게 됩니다.
즉, 한 클래스를 수정하였을 때, 다른 클래스도 수정해야 하는 상황을 막아줄 수 있습니다.

요약
- DI, 의존성 주입은 필요한 객체를 직접 생성하는 것이 아닌 외부로 부터 필요한 객체를 받아서 사용하는 것이다.
- 이를 통해 객체간의 결합도를 줄이고 코드의 재활용성을 높여준다.

Dagger2

가장 많이 보이는 DI Framework

- 보일러 플레이트 자바 코드를 컴파일 타임에 자동으로 생성

- Reflection 사용 X

Dagger2는 자바와 안드로이드를 위해 만들어진 컴파일타임 의존성주입 프레임워크 입니다.
보일러 플레이트 자바 코드를 컴파일 타임에 자동으로 생성하고 리플렉션 사용이 없기 때문에 많은 안드로이드 개발자들이 사용하고 있습니다.
하지만 단점으로는 Dagger2 적용을 위해 환경을 세팅해야하는 과정과 원활한 적용에 필요한 러닝커브가 가파르고,
러닝커브에 사용되는 비용이 상대적으로 저렴한 (이미 알거나 금방 배울 수 있는 수준의 개발 역량을 가진)사람들이 쓰기에 알맞은 Framework이지 않을까 한다.

보일러 플레이트란
- 최소한의 변경으로 여러곳에서 재사용되며, 반복적으로 비슷한 형태를 띄는 코드를 말한다.

컴파일 타임
- 개발자에 의해 C, JAVA 등과 같은 개발 언어로 소스코드가 작성되며, 컴파일 과정을 통해 컴퓨터가 인식을할 수 있는 기계어 코드로 변환되어 실행가능한 프로그램이 되는 과정을 의미한다.

리플렉션이란
구체적인 클래스 타입을 알지 못해도 그 클래스의 메소드, 타입, 변수들에 접근할 수 있도록 해주는 자바 API

Koin

Kotlin DSL

- Koin은 코틀린을 위한 DI 라이브러리

- 아키텍처 중에 하나인 VM(ViewModel)을 이용하기 위해 별도의 라이브러리도 제공

Dagger2와 Koin의 에러시점이 다른 이유

Dagger2 동작 방식

`Build -> (DI container, DI 객체 생성 및 주입 관련 class) generate -> run`

Koin의 동작 방식

`Build -> run(DI container, DI 객체 생성 및 주입)`
‘동작 방식의 차이’때문에 에러 시점이 다르다.

러닝커브(학습곡선)가 가파른 Dagger에 비해 상대적으로 낮은 러닝커브를 가지고 있고 순수 코틀린만으로 작성이 되어있으며 어노테이션(@) 프로세싱을 및 리플렉션을 사용하지 않기 때문에 상대적으로 더 가볍습니다
아주 쉽게 MVVM 패턴을 만들 수 있습니다. Dagger2에 비해 사용법이 쉽고, DSL을 활용하여 runtime에 의존성을 주입한다.
하지만 runtime error가 발생할 수 있고 compile시점에서 오류 확인이 어려울 수 있습니다.
대신 test를 잘 작성하여 애플리케이션의 퀄리티를 보장하는 것이 개선할 수 있는 방법이 될 수 있습니다.
그러나 Framework에서 compile error를 확인하는 것과 unit Test를 통해 error를 확인하는 것. 어느 것이 배포 전 오류를 잘 잡아낼 수 있을까? 라고 한다면 Dagger2 >= Koin이 되지 않을까 생각합니다. 아무래도 unit Test 또한 사람이 하는 것이기 때문에 어쩔 수 없을 것 입니다.
계속 발전하는 신규 라이브러리라 더욱 기대가 되지만 2019 Android Dev Summit에서 구글이 Dagger2를 쓰라고 권장을 했기도 하며, 이미 수많은 업계에서 Dagger2가 기업에서는 사용하는 경우는 찾기가 힘듭니다.

런타임
- 컴파일 과정을 마친 응용 프로그램이 사용자에 의해서 실행되어 지는 때 를 의미합니다.

Hilt

Dagger 기반으로 만들어진 DI Framework

Application 에서 DI 를 사용하는 표준적인 방법을 제공
(Android Dagger, Dagger2 등을 하나로 통합)

Dagger2 기반의 라이브러리

- 표준화된 Dagger2 사용법을 제시
- 보일러플레이트 코드 감소(Google IO 앱을 Hilt 로 리팩토링 한 결과 의존성 주입 코드를 75 % 나 줄임)
- 프로젝트 설정의 간소화
- 쉬운 모듈 탐색과 통합
- 개선된 테스트 환경

의존성 주입 Framework 의 궁극적인 목표
 - 정확한 사용 방법을 제안

 - 쉬운 설정 방법 (Dagger 를 사용하지 않고 Koin 사용하는 이유중 큰 비중을 차지했다)

 - 중요한 것들에 집중할 수 있도록 한다 (컴파일 타임에 에러를 수정할 수 있는 것은 매우 큰 장점이나 너무 빈번하게 발생하면 개발자에게 피곤함을 줄 수 있다)
그리하여 Hilt가 등장하였습니다.
하지만 이렇게 좋아보이는 Hilt도 단점이 있습니다.
Hilt를 적용할 때 가장 걱정하는 부분은 안정성 보장과 현재 개발 환경과의 충돌이 될 수 있습니다.
보수적이고 안정적인 서비스를 유지하는 프로젝트에는 사용이 어렵지만, 개인 프로젝트 단계에서는 써볼만 합니다.
그래도 계속해서 업그레이드된 버전들이 나올 것이라 예상된다. 구글 문서도 빠르게 업데이트 중이고, 낮은 러닝커브와 Dagger의 장점을 모아 구글에서 만들어지고 있는 Framework인만큼 앞으로도 계속해서 발전하고 자주 사용될 것이라 예상한다.

Dagger2 사용 예제

햄버거 만들기

햄버거(Burger)를 만들어 보겠습니다.

햄버거는 밀빵(WheatBun) 과 소고기 패티(BeefPatty) 로 이루어져 있습니다.

```
public class Burger {

    public WheatBun bun;
    public BeefPatty patty;

    public Burger(WheatBun bun, BeefPatty patty) {
        this.bun = bun;
        this.patty = patty;
    }
}
```

사용하는 예제는 '김태호 - 커니의 코틀린' 에서 가져왔습니다.

Dagger2 사용 예제

밀빵

```
public class WheatBun {

    public String getBun() {
        return "밀빵";
    }
}
```

소고기 패티

```
public class BeefPatty {

    public String getPatty() {
        return "소고기 패티";
    }
}
```

햄버거를 만들기 위한 밀빵(WheatBun) 과 소고기 패티(BeefPatty) 를 준비해 봤습니다.

Dagger2 사용 예제

```
public class MainActivity extends AppCompatActivity {

    Burger burger;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        WheatBun bun = new WheatBun();
        BeefPatty patty = new BeefPatty();

        burger = new Burger(bun, patty);



    }

}
```

이렇게 new 오퍼레이터를 사용해서 밀빵과 소고기 패티를 가져와서 햄버거를 만듭니다.

이렇게 MainActivity 에서 인스턴트를 생성하는게 기존의 방식이었습니다. 여기에 의존성 주입을 사용하면 new 오퍼레이터 사용 없이 외부에서 자동으로 객체를 생성해 줍니다.

Dagger2 사용 예제

gradle

```
implementation "com.google.dagger:dagger:$daggerVersion"
implementation "com.google.dagger:dagger-android:$daggerVersion"
implementation "com.google.dagger:dagger-android-support:$daggerVersion"

kapt "com.google.dagger:dagger-android-processor:$daggerVersion"
kapt "com.google.dagger:dagger-compiler:$daggerVersion"
```

1. Module 과 Component 를 만들어야 합니다.

```
WheatBun bun = new WheatBun();
BeefPatty patty = new BeefPatty();

Burger burger = new Burger(bun, patty);
```

Module 은 필요한 객체를 제공하는 역활을 합니다.
우리가 MainActivity 에서 아래와 같이 객체를 생성했죠?
이걸 모듈이 해줍니다.

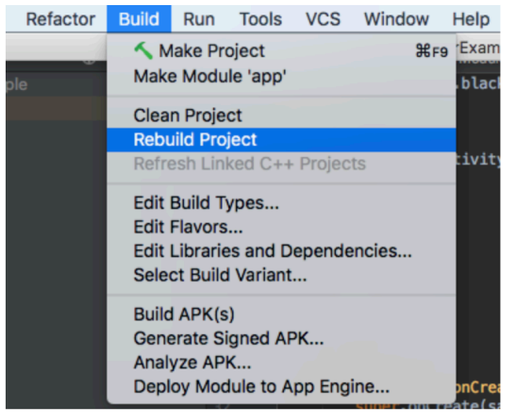
```java
@Module
public class BurgerModule {

    @Provides
    Burger provideBurger(WheatBun bun, BeefPatty patty) {
        return new Burger(bun, patty);
    }

    @Provides
    WheatBun provideBun() {
        return new WheatBun();
    }

    @Provides
    BeefPatty providePatty() {
        return new BeefPatty();
    }
}
```

생성할 객체를 @Provides 어노테이션을 사용해서 생성해 줍니다.

```java
@Component(modules = BurgerModule.class)
public interface BurgerComponent {

    void inject(MainActivity activity);
}
```

Component 는 모듈에서 제공받은 객체를 조합하여 필요한 곳에 주입하는 역활을 합니다.
@Componet 어노테이션을 통해 BurgerModule.class 를 가져왔습니다. 이를 inject() 함수를 통해 MainActivity 에 주입했습니다. (void inject() 의 함수 명은 정해진 값이 아닌 임의의 함수 명을 정해주시면 됩니다.)

2. MainActivity 에서 사용해보기



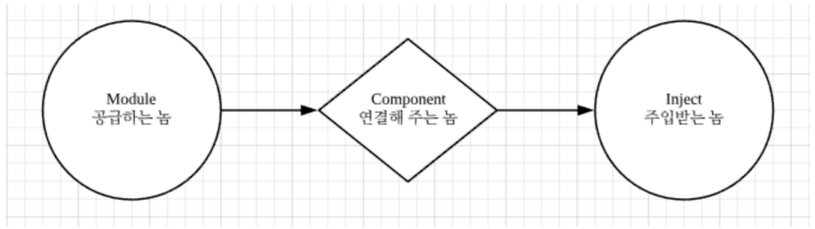
1번 작업을 한 후 프로젝트를 빌드 해 주시면 Dagger + {ComponetName} 파일이 자동 생성됩니다.

(여기서는 DaggerBurgerComponent 가 생성됩니다. 이렇게 컴파일 단계에서 의존성을 체크할 수 있어 앱의 안정성을 높여줄 수 있습니다.)

```java
public class MainActivity extends AppCompatActivity {

    @Inject
    Burger burger;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /*WheatBun bun = new WheatBun();
        BeefPatty patty = new BeefPatty();

        burger = new Burger(bun, patty);*/

        BurgerComponent component = DaggerBurgerComponent.builder()
                .burgerModule(new BurgerModule())
                .build();

        component.inject(this);

        Log.d("MyTag","burger bun : " + burger.bun.getBun() +

                    " , patty : " + burger.patty.getPatty());

    }

}
```

위와같이 new 오퍼레이터 사용 없이 위와 같은 코드로 burger 객체를 외부에서 생성하여 가져올 수 있습니다.

(burger = new Burger() 를 안했는데도 객체가 생성되었어요!! 놀라워라!!)
이름 그림으로 도식화 하여 이해하기 쉽게 표현해 보겠습니다.



이름 그림으로 도식화 하여 이해하기 쉽게 표현해 보겠습니다.

간단히 공급하는 놈(Module)과 주입받는 놈(Inject)

그리고 이 사이를 연결해주는 놈(Component)로 이해하시면 됩니다. 놈!놈!놈!

Module
Module 에서 공급할 객체를 생성합니다. (예제에서는 햄거거, 밀빵, 소고기 패티를 생성했습니다.)

Component
Component 에서는 Module을 불러오고 어디로 주입할 지를 정합니다. (MainActivity 주입할 겁니다!)

Inject
@Inject 어노테이션을 통해 주입받는 놈들 가져옵니다.
(MainActivity 에서 @inject Burger burger 변수를 선언했더니 객체가 두둥! 생성!)

결론

	Dagger2	Koin	Hilt
러닝커브	높음	낮음	낮음
Java	O	X	O
Kotlin	O	O	O
에러 검출 시점	Compile Time	Runtime	Compile Time
실제 프로젝트 검증	많은 앱에서 검증됨	많은 앱에서 검증됨	beta버전(조금씩 보임)

마치며

개발자는 완벽한 프로그램을 만들어내는 것이 목적이 아니라
비즈니스를 위한 제품을 개발하는 사람이라 생각합니다
좋은 퀄리티의 Product를 지향하지만 가장 우선 시 되어야 할 부분은 비즈니스에 맞는 기술을 선택하는 것이라고 생각하고,어떤 기술을 선택함에 있어서 현 상황과 비용을 고려하여 합리적인 선택을 하는 것도 개발자의 역량이라 생각합니다.