# Dagger 2 完全解析（一），Dagger 2 的基本使用与原理

JohnnyShieh JohnnyShieh 2017-07-17 19:14


> 本系列文章是基于 Google Dagger 2.11-rc2 版本

## 依赖注入

### 什么是依赖

如果在 Class A 中，有 Class B 的实例，则称 Class A 对 Class B 有一个依赖。例如 Man 中有用到一个 Car 对象，即 Man 对 Car 有一个依赖。

```
public class Man {
    Car car;
    public Man() {
        car = new Car();
    }
    ...
}
```

上面这种写法是最常见的写法，但是在下面几个场景中存在一些问题：

1. 如果要修改 Car 的构造函数，例如需要使用 `car = new Car(name)` 的方式构造时，还要修改 Man 的代码；
2. 如果想测试不同的 Car 对 Man 的影响会很困难，例如单元测试中使用 mock 的 car 测试 Man。

## 什么是依赖注入

**依赖注入**（Dependency Injection，简称 DI）是用于实现**控制反转**（Inversion of Control，缩写为 IoC）最常见的方式之一，控制反转是面向对象编程中的一种设计原则，用以降低计算机代码之间耦合度。控制反转的基本思想是：借助“第三方”实现具有依赖关系的对象之间的解耦。一开始是对象 A 对 对象 B 有个依赖，对象 A 主动地创建对象 B，对象 A 有主动控制权，实现了 Ioc 后，对象 A 依赖于 Ioc 容器，对象 A 被动地接受容器提供的对象 B 实例，由主动变为被动，因此称为控制反转。**注意，控制反转不等同于依赖注入，控制反转还有一种实现方式叫“依赖查找”（Denpendency Lookup）**。更多控制反转的信息请看控制反转的维基百科。

依赖注入就是将对象实例传入到一个对象中去（Denpendency injection means giving an object its instance variables）。依赖注入是一种设计模式，降低了依赖和被依赖对象之间的耦合，方便扩展和单元测试。

## 依赖注入的实现方式

其实在平常编码的过程中，已经不知觉地使用了依赖注入

- 基于构造函数，在构造对象时注入所依赖的对象。

```
public class Man {
    Car car;
    public Man(Car car) {
        this.car = car;
    }
    ...
}
```

- 基于 set 方法，使用 setter 方法来让外部容器调用传入所依赖的对象。

```
public class Man {
    ...
```

```
    public void setCar(Car car) {
        this.car = car;
    }
}
```

- 基于接口，使用接口来提供 setter 方法。

```
public interface CarInjector {
    void injectCar(Car car);
}

public class Man implements CarInjector {
    ...
    @Override
    public void injectCar(Car car) {
        this.car = car;
    }
}
```

- 基于注解，Dagger 2 依赖注入框架就是使用 `@Inject` 完成注入。

```
public class Man {
    @Inject
    Car car;
    ...
}
```

## Dagger 2

Dagger 2 是 Java 和 Android 下的一个完全静态、编译时生成代码的依赖注入框架，由 Google 维护，早期的版本 Dagger 是由 Square 创建的。

Dagger 2 是基于 Java Specification Request(JSR) 330标准。利用 JSR 注解在编译时生成代码，来注入实例完成依赖注入。

下面是 Dagger 2 的一些资源地址：

Github：https://github.com/google/dagger

官方文档：https://google.github.io/dagger//

API：http://google.github.io/dagger/api/latest/

# Dagger 2 的基本使用

上面介绍了依赖注入和 Dagger 2，下面由简单的示例开始一步一步地解析 Dagger 2 的基本使用与原理。

## 引入 Dagger 2

在 `build.gradle` 中添加依赖：

```
dependencies {
    ...
    compile 'com.google.dagger:dagger:2.11-rc2'
    annotationProcessor 'com.google.dagger:dagger-compiler:2.11-rc2'
}
```

如果 Android gradle plugin 的版本低于 `2.2`，还需要引入 android-apt 插件。

## 使用 @Inject 标注需要注入的依赖

继续使用上面 Man 的例子：

```
public class Man {
    @Inject
    Car car;
    ...
}
```

使用 `javax.inject.Inject` 注解来标注需要 Dagger 2 注入的依赖，build 后可以在 `build/generated/source/apt` 目录下看到 Dagger 2 编译时生成的成员属性注入类。

```
public final class Man_MembersInjector implements MembersInjector<Man> {
  private final Provider<Car> carProvider;

  public Man_MembersInjector(Provider<Car> carProvider) {
    assert carProvider != null;
    this.carProvider = carProvider;
  }

  public static MembersInjector<Man> create(Provider<Car> carProvider) {
    return new Man_MembersInjector(carProvider);
  }

  @Override
```

```
  public void injectMembers(Man instance) {
    if (instance == null) {
      throw new NullPointerException("Cannot inject members into a null i
    }
    instance.car = carProvider.get();
  }

  public static void injectCar(Man instance, Provider<Car> carProvider) {
    instance.car = carProvider.get();
  }
}
```

从上面的 `injectMembers` 方法中可以看到注入依赖的代码是 `instance.car = carProvider.get();`，所以 `@Inject` 标注的成员属性不能是 `private` 的，不然无法注入。

## 创建所依赖对象的实例

用 `@Inject` 标注构造函数时，Dagger 2 会完成实例的创建。

```
public class Car {
    @Inject
    public Car() {}
}
```

build 后可以在 `build/generated/source/apt` 目录下看到 Dagger 2 编译时生成的工厂类。

```
public final class Car_Factory implements Factory<Car> {
  private static final Car_Factory INSTANCE = new Car_Factory();

  @Override
  public Car get() {
    return new Car();
  }

  public static Factory<Car> create() {
    return INSTANCE;
  }
}
```

依赖注入是 `依赖的对象实例` → `需要注入的实例属性`，上面完成两步，通过 Dagger 2 生成的代码代码可以知道，生成了 Man 的成员属性注入类和 Car 的工厂类，接下来需要的就

是新建工厂实例并调用成员属性注入类完成 car 的实例注入。完成这个过程的桥梁就是 `dagger.Component`。

## Component 桥梁

`@Component` 可以标注接口或抽象类，Component 桥梁可以完成依赖注入过程，其中最重要的是定义注入接口，调用注入接口就可以完成 Man 所需依赖的注入。

```
@Component
public interface ManComponent {

    void injectMan(Man man);  // 注入 man 所需要的依赖

}
```

build 后会生成带有 `Dagger` 前缀的实现该接口的类：DaggerManComponent

```
public final class DaggerManComponent implements ManComponent {
  private MembersInjector<Man> ManMembersInjector;

  private DaggerManComponent(Builder builder) {
    assert builder != null;
    initialize(builder);
  }

  public static Builder builder() {
    return new Builder();
  }

  public static ManComponent create() {
    return new Builder().build();
  }

  @SuppressWarnings("unchecked")
  private void initialize(final Builder builder) {

    this.ManMembersInjector = Man_MembersInjector.create(Car_Factory.crea
  }

  @Override
  public void injectMan(Man man) {
    ManMembersInjector.injectMembers(man);  // 完成依赖注入
  }
```

```
  public static final class Builder {
    private Builder() {}

    public ManComponent build() {
      return new DaggerManComponent(this);
    }
  }
}
```

从上面生成的代码可以看出来 Component 就是连接`依赖的对象实例`和`需要注入的实例属性`之间的桥梁。Component 会查找目标类对应的成员属性注入类（即 MembersInjector），然后把依赖属性的工厂实例（即 Car_Factory.create()）传给注入类，再使用 Component 一开始定义的接口就能完成依赖注入。**注意，Component 中注入接口的参数必须为需要注入依赖的类型，不能是 Man 的父类或子类，注入接口返回值为 void，接口名可以任意。**

接下来只需要在 Man 中调用`injectMan`方法就能完成注入。

```
public class Man {
    ...
    public Man() {
        DaggerManComponent.create().injectMan(this);
    }
    ...
}
```

## Module

使用`@Inject`标注构造函数来提供依赖的对象实例的方法，不是万能的，在以下几种场景中无法使用：

- 接口没有构造函数
- 第三方库的类不能被标注
- 构造函数中的参数必须配置

这时，就可以用`@Provides`标注的方法来提供依赖实例，方法的返回值就是依赖的对象实例，`@Provides`方法必须在`Module`中，Module 即用`@Module`标注的类。所以 Module 是提供依赖的对象实例的另一种方式。

```
@Module
public class CarModule {
```

```
    @Provides
    static Car provideCar() {
        return new Car();
    }
}
```

约定俗成的是 `@Provides` 方法一般以 `provide` 为前缀，Moudle 类以 `Module` 为后缀，一个 Module 类中可以有多个 `@Provides` 方法。

接下来，需要把可以提供依赖实例的 Module 告诉 Component：

```
@Component(modules = CarModule.class)
public interface ManComponent {

    void injectMan(Man man);  // 注入 man 所需要的依赖

}
```

build之后，Module 和 Component 生成的类为：

```
public final class CarModule_ProvideCarFactory implements Factory<Car> {
  private static final CarModule_ProvideCarFactory INSTANCE = new CarModu

  @Override
  public Car get() {
    return Preconditions.checkNotNull(
        CarModule.provideCar(), "Cannot return null from a non-@Nullable
  }

  public static Factory<Car> create() {
    return INSTANCE;
  }

  /** Proxies {@link CarModule#provideCar()}. */
  public static Car proxyProvideCar() {
    return CarModule.provideCar();
  }
}
```

CarModule_ProvideCarFactory 和之前的 Car_Factory 类似，都实现 Factory接口。

生成的 DaggerManComponent 和之前相比只改变了一个方法：

```
private void initialize(final Builder builder) {
    this.manMembersInjector = Man_MembersInjector.create(CarModule_Provi
}
```

只是提供依赖实例的工厂变为了 CarModule 对应的工厂。

## 总结

现在再来看 Dagger 2 最核心的三个部分：

1. 需要注入依赖的目标类，需要注入的实例属性由 `@Inject` 标注。
2. 提供依赖对象实例的工厂，用 `@Inject` 标注构造函数或定义 `Module` 这两种方式都能提供依赖实例，Dagger 2 的注解处理器会在编译时生成相应的工厂类。**`Module` 的优先级比 `@Inject` 标注构造函数的高，意味着 Dagger 2 会先从 Module 寻找依赖实例。**
3. 把依赖实例工厂创建的实例注入到目标类中的 Component。

下面再讲述上面提到的在 Dagger 2 种几个注解的用法：

- `@Inject` 一般情况下是标注成员属性和构造函数，标注的成员属性不能是 `private`，Dagger 2 还支持方法注入，`@Inject` 还可以标注方法。
- `@Provides` 只能标注方法，必须在 Module 中。
- `@Module` 用来标注 Module 类
- `@Component` 只能标注接口或抽象类，声明的注入接口的参数类型必须和目标类一致。

**推荐阅读：**

- Dagger 2 User's Guide
- Using Dagger 2 for dependency injection in Android - Tutorial