Dagger 2 完全解析（二），进阶使用 Lazy、Qualifier、Scope 等

最帅的 JohnnyShieh 2017-07-31 09:40

阅读本文大概需要 9 分钟。

在上篇文章中介绍了 Dagger 2 的三个核心要素，但是在实际使用过程中，还需要 Layz 注入、Qualifier（限定符）、Scope（作用域）等作为补充，也是在 Dagger 2 进阶中必须要掌握的内容。下面由简入繁地讲解这几个概念，同时结合 Dagger 2 的编译时生成代码分析背后的原理（示例代码沿用第一篇的，建议先阅读第一篇）。

在分析 Qualifier（限定符）、Scope（作用域）之前，先介绍一些简单的概念： Lazy 和 Provider 注入。

1. Lazy （延迟注入）

有时我们想注入的依赖在使用时再完成初始化，加快加载速度，就可以使用注入 `Lazy<T>` 。只有在调用 Lazy的 get() 方法时才会初始化依赖实例注入依赖。

```
public class Man {
    @Inject
    Lazy<Car> lazyCar;

    public void goWork() {
```

```
        ...
        lazyCar.get().go(); // lazyCar.get() 返回 Car 实例
        ...
    }
}
```

2. Provider 注入

有时候不仅仅是注入单个实例，我们需要多个实例，这时可以使用注入 `Provider<T>`，每次调用它的 get() 方法都会调用到 `@Inject 构造函数` 创建新实例或者 Module 的 provide 方法返回实例。

```
public class CarFactory {
    @Inject
    Provider<Car> carProvider;

    public List<Car> makeCar(int num) {
        ...
        List<Car> carList = new ArrayList<Car>(num);
        for (int i = 0; i < num; i ++) {
            carList.add(carProvider.get());
        }
        return carList;
    }
}
```

3. Qualifier（限定符）

试想这样一种情况：沿用之前的 Man 和 Car 的例子，如果 CarModule 提供了两个生成 Car 实例的 provide 方法，Dagger 2 在注入 Car 实例到 Man 中时应该选择哪一个方法呢?

```
@Module
public class CarModule {
    @Provides
    static Car provideCar1() {
        return new Car1();
    }
    @Provides
    static Car provideCar2() {
        return new Car2();
    }
    // Car1 和 Car2 是 Car 的两个子类
}
```

这时 Dagger 2 不知道使用 `provideCar1` 还是 `provideCar2` 提供的实例，在编译时就会报错，这种情况也可以叫依赖迷失（网上看到的叫法）。而 `@Qualifier` 注解就是用来解决这个问题，使用注解来确定使用哪种 provide 方法。

下面是自定义的 `@Named` 注解，你也可以用自定义的其他 Qualifier 注解：

```
@Qualifier
@Documented
@Retention(RUNTIME)
public @interface Named {
    String value() default "";
}
```

在 provide 方法上加上 `@Named` 注解，用来区分

```
@Module
public class CarModule {
    @Provides
    @Named("car1")
    static Car provideCar1() {
        return new Car1();
    }
    @Provides
    @Named("car2")
    static Car provideCar2() {
        return new Car2();
    }
}
```

还需要在 Inject 注入的地方加上 `@Named` 注解，表明需要注入的是哪一种 Car：

```
public class Man {
    @Inject
    @Named("car1")
    Car car;
    ...
}
```

这样在依赖注入时，Dagger 2 就会使用 `provideCar1` 方法提供的实例，所以Qualifier（限定符）的作用相当于起了个区分的别名。

4. Scope（作用域）

Scope 是用来确定注入的实例的生命周期的，如果没有使用 Scope 注解，Component 每次调用 Module 中的 provide 方法或 `Inject 构造函数` 生成的工厂时都会创建一个新的实例，而使用 Scope 后可以复用之前的依赖实例。下面先介绍 Scope 的基本概念与原理，再分析 Singleton、Reusable 等作用域。

4.1 Scope 基本概念

先介绍 Scope 的用法，`@Scope` 是元注解，是用来标注自定义注解的，如下：

```
@Documented
@Retention(RUNTIME)
@Scope
public @interface MyScope {}
```

MyScope 就是一个 Scope 注解，Scope 注解只能标注目标类、@provide 方法和 Component。Scope 注解要生效的话，需要同时标注在 Component 和提供依赖实例的 Module 或目标类上。Module 中 provide 方法中的 Scope 注解必须和与之绑定的 Component 的 Scope 注解一样，否则作用域不同会导致编译时会报错。例如，CarModule 中 provide 方法的 Scope 是MyScope 的话，ManComponent 的 Scope 必须是 是 MyScope 这样作用域才会生效，而且不能是 `@Singleton` 或其他 Scope 注解，不然编译时 Dagger 2 会报错（亲自测试过）。

那么 Scope 注解又是如何产生作用的呢，怎么保证生成的依赖实例的生命周期呢?

在 Dagger 2 官方文档中我找到一句话，非常清楚地描述了 `@Scope` 的原理：

> When a binding uses a scope annotation, that means that the component object holds a reference to the bound object until the component object itself is garbage-collected.

当 Component 与 Module、目标类（需要被注入依赖）使用 Scope 注解绑定时，意味着 Component 对象持有绑定的依赖实例的一个引用直到 Component 对象本身被回收。也就是作用域的原理，其实是让生成的依赖实例的生命周期与 Component 绑定，Scope 注解并不能保证生命周期，要想保证赖实例的生命周期，需要确保 Component 的生命周期。

下面以 `@MyScope` 为例，看 Scope 注解背后的代码：

```
@Module
public class CarModule {

    @Provides
    @MyScope
```

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

```
@MyScope
@Component(modules = CarModule.class)
public interface ManComponent {
    void injectMan(Man man);
}
```

这样生成的 Car 实例就与 ManComponent 绑定了。下面看编译时生成的代码：

```
public final class DaggerManComponent implements ManComponent {
    private Provider<Car> provideCarProvider;
    private MembersInjector<Man> manMembersInjector;
    ...
    @SuppressWarnings("unchecked")
    private void initialize(final Builder builder) {

        this.provideCarProvider = DoubleCheck.provider(CarModule_ProvideC

        this.manMembersInjector = Man_MembersInjector.create(provideCarPr
    }
...
}

public final class DoubleCheck<T> implements Provider<T>, Lazy<T> {
    private static final Object UNINITIALIZED = new Object();

    private volatile Provider<T> provider;
    private volatile Object instance = UNINITIALIZED; // instance 就是依赖
...
    @SuppressWarnings("unchecked") // cast only happens when result comes
    @Override
    public T get() {
        Object result = instance;
        if (result == UNINITIALIZED) {  // 只生成一次实例，之后调用的话直接复用
            synchronized (this) {
                result = instance;
                if (result == UNINITIALIZED) {
                    result = provider.get();  // 生成实例
                    /* Get the current instance and test to see if the ca
                     * in a recursive call.  If it returns the same inst
                     * instances differ, throw. */
                    Object currentInstance = instance;
                    if (currentInstance != UNINITIALIZED && currentInstan
                        throw new IllegalStateException("Scoped provider w
                            + "different results: " + currentInstance + "
```

```
                        + "due to a circular dependency.");
                }
                instance = result;
                /* Null out the reference to the provider. We are nev
                   * can make it eligible for GC. */
                provider = null;
            }
        }
    }
    return (T) result;
  }
...
}
```

从上面 DaggerManComponent 的代码可以看出使用了 MyScope 作用域后，`provideCarProvider` 由 `CarModule_ProvideCarFactory.create()` 变为了 `DoubleCheck.provider(CarModule_ProvideCarFactory.create())`。而 DoubleCheck 包装的意义在于持有了 Car 的实例，而且只会生成一次实例，也就是说：没有用 MyScope 作用域之前，DaggerManComponent 每次注入依赖都会新建一个 Car 实例，而用 MyScope 作用之后，每次注入依赖都只会返回第一次生成的实例。DaggerManComponent 持有 provideCarProvider 引用，provideCarProvider 又持有 instance（即 Car 实例）的引用，所以生成 Car 对象实例的生命周期就和 Component 一致了，作用域就生效了。

Scope 作用域的本质：Component 间接持有依赖实例的引用，把实例的作用域与 Component 绑定，它们不是同年同月同日生，但是同年同月死。

4.2 Singleton Scope

在了解作用域的原理后，再来理解 Dagger 2 提供的自带作用域就容易了。`@Singleton` 顾名思义保证单例，那么它又是如何实现的呢，实现了单例模式那样只返回一个实例吗?

把上面例子中 `@MyScope` 换成 `@Singleton`，发现生成的 DaggerManComponent 和其他类没有变化。也只是用 `DoubleCheck` 包装了工厂而已，并没有什么特殊实现。所以 Singleton 作用域可以保证一个 Component 中的单例，但是如果产生多个 Component 实例，那么实例的单例就无法保证了。

所以在网上一些例子中，有看到 `AppComponent` 使用 Singleton 作用域，保证绑定的依赖实例的单例。它生效的原因是 `AppComponent` 只会在 Application 中创建一次，由 `AppComponent` 的单例来保证绑定的依赖实例的单例。

注意：Component 可以同时被多个 Scope 标记。即 Component 可以和多个 Scope 的 Moudle 或目标类绑定。

4.3 Reusable Scope

上文中的自定义的 `@MyScope` 和 `@Singleton` 都可以使得绑定的 Component 缓存依赖的实例，但是与之绑定 Component 必须有相同的 Scope 标记。假如我只想单纯缓存依赖的实例，可以复用之前的实例，不想关心与之绑定是什么 Component，应该怎么办呢？。

这时就可以使用 `@Reusable` 作用域，Reusable 作用域不关心绑定的 Component，Reusable 作用域只需要标记目标类或 provide 方法，不用标记 Component。下面先看看使用 Reusable 作用域后，生成的 DaggerManComponent 的变化：

```
public final class DaggerManComponent implements ManComponent {
    ...
    @SuppressWarnings("unchecked")
    private void initialize(final Builder builder) {
        this.provideCarProvider = SingleCheck.provider(CarModule_Provide(

        this.manMembersInjector = Man_MembersInjector.create(provideCarP
    }
...
}

public final class SingleCheck<T> implements Provider<T>, Lazy<T> {
    private static final Object UNINITIALIZED = new Object();

    private volatile Provider<T> provider;
    private volatile Object instance = UNINITIALIZED; // 缓存实例的引用
...
    @SuppressWarnings("unchecked") // cast only happens when result come
    @Override
    public T get() {
        // provider is volatile and might become null after the check to
        // retrieve the provider first, which should not be null if inst
        // This relies upon instance also being volatile so that the rea
        // cannot be reordered.
        Provider<T> providerReference = provider;
        if (instance == UNINITIALIZED) {
            instance = providerReference.get(); // 一般情况下只会生成一个实例
            // Null out the reference to the provider. We are never goin
            // it eligible for GC.
            provider = null;
        }
        return (T) instance;
```

```
    }
...
}
```

从上面代码可以看出使用 `@Reusable` 作用域后，利用到 Reusable 实例的 Component 会间接持有实例的引用。但是这里是用 `SingleCheck` 而不是 `DoubleCheck`，在多线程情况下可能会生成多个实例，关于这个有疑问推荐阅读「Java 设计模式之单例模式」(http://johnnyshieh.me/posts/java-singleton-pattern/)
中双重检查锁定的部分。因为 `@Reusable` 作用域目的只是可以复用之前的实例，并不需要严格地保证实例的唯一，所以使用 SingleCheck 就足够了。

4.4 Releasable references（可释放引用）

使用 Scope 注解时，Component 会间接持有绑定的依赖实例的引用，也就是说实例在 Component 还存活时无法被回收。而在 Android 中，应该尽量减少内存占用，把没有使用的对象释放，这时可以用 `@CanReleaseReferences` 标记 Scope 注解：

```
@Documented
@Retention(RUNTIME)
@CanReleaseReferences
@Scope
public @interface MyScope {}
```

然后在 Application 中注入 `ReleasableReferenceManager` 对象，在内存不足时调用 `releaseStrongReferences()` 方法把 Component 间接持有的强引用变为弱引用。

```
public class MyApplication extends Application {
    @Inject
    @ForReleasableReferences(MyScope.class)
    ReleasableReferenceManager myScopeReferences;

    @Override
    public void onLowMemory() {
        super.onLowMemory();
        myScopeReferences.releaseStrongReferences();
    }
...
}
```

这样在内存不足时，DaggerManComponent 间接持有的 Car 实例为弱引用，如果没有其他对象使用的话就可以被回收。

5. Binding Instances

通过前面作用域的讲解，可以清楚 Component 可以间接持有 Module 或 Inject 目标类构造函数提供的依赖实例，除了这两种方式，Component 还可以在创建 Component 的时候绑定依赖实例，用以注入。这就是 `@BindsInstance` 注解的作用，只能在 Component.Builder 中使用。

在 Android 中使用 Dagger 2 时，activity 实例经常也需要作为依赖实例用以注入，在之前只能使用 Module：

```
@Module
public final class HomeActivityModule {
    private final HomeActivity activity;

    public HomeActivityModule(HomeActivity activity) {
        this.activity = activity;
    }

    @Provides
    @ActivityScope  // 自定义作用域
    Activity provideActivity() {
        return activity;
    }
}
```

而使用 `@BindsInstance` 的话会更加简单：

```
@ActivityScope
@Component
public interface HomeActivityComponent {
    @Component.Builder
    interface Builder {
        @BindsInstance
        Builder activity(Activity activity);
        HomeActivityComponent build();
    }
}
```

注意在调用 `build()` 创建 Component 之前，所有 `@BindsInstance` 方法必须先调用。上面例子中 HomeActivityComponent 还可以注入 Activity 类型的依赖，但是不能注入 HomeActivity，因为 Dagger 2 是使用具体类型作为依据的（也就是只能使用 `@Inject Activity activity` 而不是 `@Inject HomeActivity activity`）。

如果 `@BindsInstance` 方法的参数可能为 null，需要再用 `@Nullable` 标记，同时标注 Inject 的地方也需要用 `@Nullable` 标记。这时 Builder 也可以不调用 `@BindsInstance` 方法，这样 Component 会默认设置 instance 为 null。

6. 总结

- Qualifier 限定符用来解决依赖迷失问题，可以依赖实例起个别名用来区分。
- Scope 作用域的本质是 Component 会持有与之绑定的依赖实例的引用，要想确保实例的生命周期，关键在于控制 Component 的生命周期。
- 优先使用 `@BindsInstance` 方法，相对于写一个带有构造函数带有参数的 Module。

下一篇文章分析 Dagger 2 种 Component 的组织方式以及 SubComponent 的概念。

END

一个白日做梦的工程师！

不只有技术，还有咖啡和彩蛋！

个人博客：http://johnnyshieh.me