**依赖注入**

**1. 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。例如下面类 Human 中用到一个 Father 对象，我们就说类 Human 对类 Father 有一个依赖。

public class Human {

...

Father father;

...

public Human() {

father = new Father();

}

}

仔细看这段代码我们会发现存在一些问题：  
(1). 如果现在要改变 father 生成方式，如需要用new Father(String name)初始化 father，需要修改 Human 代码；  
(2). 如果想测试不同 Father 对象对 Human 的影响很困难，因为 father 的初始化被写死在了 Human 的构造函数中；  
(3). 如果new Father()过程非常缓慢，单测时我们希望用已经初始化好的 father 对象 Mock 掉这个过程也很困难。

**2. 依赖注入**

上面将依赖在构造函数中直接初始化是一种 Hard init 方式，弊端在于两个类不够独立，不方便测试。我们还有另外一种 Init 方式，如下：

public class Human {

...

Father father;

...

public Human(Father father) {

this.father = father;

}

}

上面代码中，我们将 father 对象作为构造函数的一个参数传入。在调用 Human 的构造方法之前外部就已经初始化好了 Father 对象。**像这种非自己主动初始化依赖，而通过外部来传入依赖的方式，我们就称为依赖注入。**  
现在我们发现上面 1 中存在的两个问题都很好解决了，简单的说依赖注入主要有两个好处：  
(1). 解耦，将依赖之间解耦。  
(2). 因为已经解耦，所以方便做单元测试，尤其是 Mock 测试。

**3. Java 中的依赖注入**

依赖注入的实现有多种途径，而在 Java 中，使用注解是最常用的。通过在字段的声明前添加 @Inject 注解进行标记，来实现依赖对象的自动注入。

public class Human {

...

@Inject Father father;

...

public Human() {

}

}

上面这段代码看起来很神奇：只是增加了一个注解，Father 对象就能自动注入了？这个注入过程是怎么完成的？

实质上，如果你只是写了一个 @Inject 注解，Father 并不会被自动注入。你还需要使用一个依赖注入框架，并进行简单的配置。现在 Java 语言中较流行的依赖注入框架有 [Google Guice](https://github.com/google/guice)、[Spring](http://projects.spring.io/spring-framework/) 等，而在 Android 上比较流行的有[RoboGuice](https://github.com/roboguice/roboguice)、[Dagger](http://square.github.io/dagger/) 等。其中 Dagger 是我现在正在项目中使用的。如果感兴趣，你可以到 [Dagger 实现原理解析](https://github.com/android-cn/android-open-project-analysis/tree/master/dagger) 了解更多依赖注入和 Dagger 实现原理相关信息。