什么是依赖注入，能说说几个依赖注入的库吗？

什么是依赖（Dependency）？

依赖是类与类之间的连接，依赖关系表示一个类依赖于另一个类的定义，通俗来讲

就是一种需要，例如一个人(Person)可以买车(Car)和房子(House),Person类依赖于Car类和House类

```
public static void main(String ... args){
        //TODO:

    Person person = new Person();
    person.buy(new House());
    person.buy(new Car());

}

static class Person{

    //表示依赖House
    public void buy(House house){}
    //表示依赖Car
    public void buy(Car car){}
}

static class House{

}

static class Car{

}
```

依赖倒置 (Dependency inversion principle)

依赖倒置是面向对象设计领域的一种软件设计原则

软件设计有 6 大设计原则，合称 **SOLID**

1、单一职责原则（Single Responsibility Principle，简称SRP ）

- **核心思想:** 应该有且仅有一个原因引起类的变更
- **问题描述：** 假如有类Class1完成职责T1，T2，当职责T1或T2有变更需要修改时，有可能影响到该类的另外一个职责正常工作。

- **好处：** 类的复杂度降低、可读性提高、可维护性提高、扩展性提高、降低了变更引起的风险。
- **需注意：** 单一职责原则提出了一个编写程序的标准，用“职责”或“变化原因”来衡量接口或类设计得是否优良，但是“职责”和“变化原因”都是不可以度量的，因项目和环境而异。

2、里氏替换原则（Liskov Substitution Principle,简称LSP）

- **核心思想：** 在使用基类的的地方可以任意使用其子类，能保证子类完美替换基类。
- **通俗来讲：** 只要父类能出现的地方子类就能出现。反之，父类则未必能胜任。
- **好处：** 增强程序的健壮性，即使增加了子类，原有的子类还可以继续运行。
- **需注意：** 如果子类不能完整地实现父类的方法，或者父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系 采用依赖、聚合、组合等关系代替继承。

3、依赖倒置原则（Dependence Inversion Principle,简称DIP）

- **核心思想：**高层模块不应该依赖底层模块，二者都该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象；
- **说明：**高层模块就是调用端，低层模块就是具体实现类。抽象就是指接口或抽象类。细节就是实现类。
- **通俗来讲：** 依赖倒置原则的本质就是通过抽象（接口或抽象类）使个各类或模块的实现彼此独立，互不影响，实现模块间的松耦合。
- **问题描述：** 类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。
- **解决方案：** 将类A修改为依赖接口interface，类B和类C各自实现接口interface，类A通过接口interface间接与类B或者类C发生联系，则会大大降低修改类A的几率。
- **好处：**依赖倒置的好处在小型项目中很难体现出来。但在大中型项目中可以减少需求变化引起的工作量。使并行开发更友好。

4、接口隔离原则（Interface Segregation Principle,简称ISP）

- **核心思想：**类间的依赖关系应该建立在最小的接口上
- **通俗来讲：** 建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。
- **问题描述：** 类A通过接口interface依赖类B，类C通过接口interface依赖类D，如果接口interface对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。
- **需注意：**
- **接口尽量小，但是要有限度**。对接口进行细化可以提高程序设计灵活性，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度
- **提高内聚，减少对外交互**。使接口用最少的方法去完成最多的事情
- **为依赖接口的类定制服务**。只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。

5、迪米特法则（Law of Demeter,简称LoD）

- **核心思想：** 类间解耦。
- **通俗来讲：** 一个类对自己依赖的类知道的越少越好。自从我们接触编程开始，就知道了软件编程的总的原则：**低耦合，高内聚**。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

6、开放封闭原则（Open Close Principle,简称OCP）

- **核心思想：** 尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化
- **通俗来讲：** 一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。

依赖倒置原则的定义如下：

> *1. 上层模块不应该依赖底层模块，它们都应该依赖于抽象。*
> *2. 抽象不应该依赖于细节，细节应该依赖于抽象。*

什么是上层模块和底层模块？

不管你承认不承认，“有人的地方就有江湖”，我们都说人人平等，但是对于任何一个组织机构而言，它一定有架构的设计有职能的划分。按照职能的重要性，自然而然就有了上下之分。并且，随着模块的粒度划分不同这种上层与底层模块会进行变动，也许某一模块相对于另外一模块它是底层，但是相对于其他模块它又可能是上层

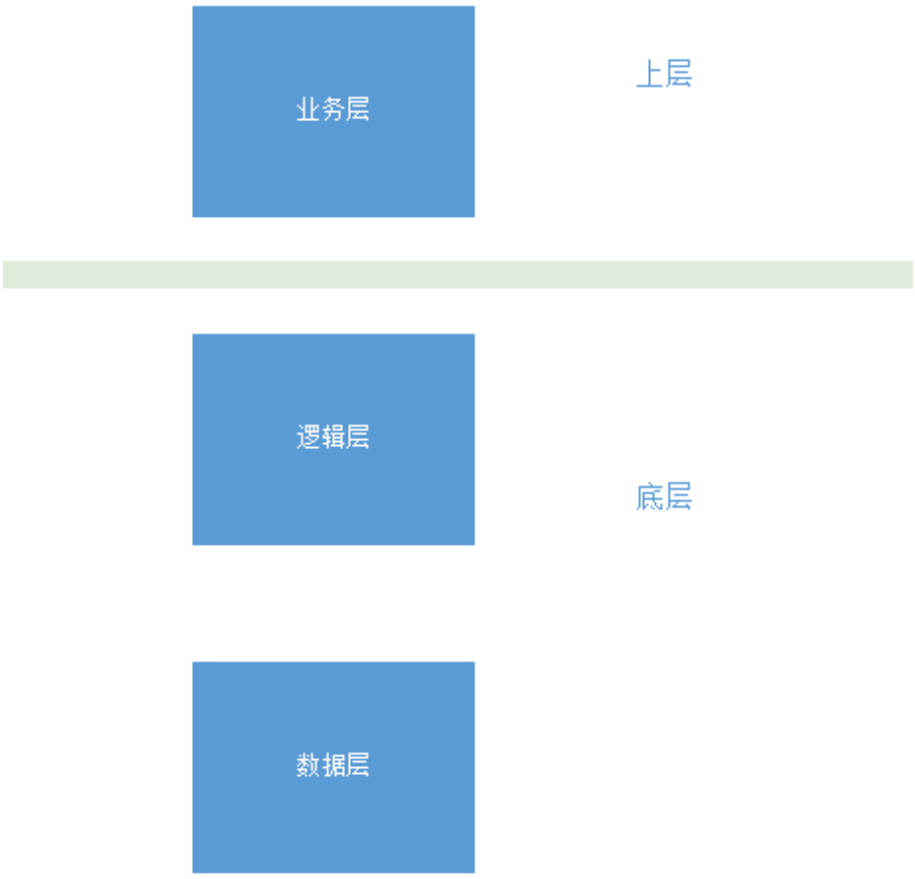
 组织架构

公司管理层就是上层，CEO 是整个事业群的上层，那么 CEO 职能之下就是底层。

然后，我们再以事业群为整个体系划分模块，各个部门经理以上部分是上层，那么之下的组织都可以称为底层。

由此，我们可以看到，在一个特定体系中，上层模块与底层模块可以按照决策能力高低为准绳进行划分。

那么，映射到我们软件实际开发中，一般我们也会将软件进行模块划分，比如业务层、逻辑层和数据层。



业务层中是软件真正要进行的操作，也就是**做什么**。 逻辑层是软件现阶段为了业务层的需求提供的实现细节，也就是**怎么做**。 数据层指业务层和逻辑层所需要的数据模型。

因此，如前面所总结，按照决策能力的高低进行模块划分。业务层自然就处于上层模块，逻辑层和数据层自然就归类为底层。

什么是抽象和细节？

象如其名字一样，是一件很抽象的事物。抽象往往是相对于具体而言的，具体也可以被称为细节，当然也被称为具象。

比如：

1. 这是一幅画。画是抽象，而油画、素描、国画而言就是具体。
2. 这是一件艺术品，艺术品是抽象，而画、照片、瓷器等等就是具体了。
3. 交通工具是抽象，而公交车、单车、火车等就是具体了。
4. 表演是抽象，而唱歌、跳舞、小品等就是具体。

上面可以知道，抽象可以是物也可以是行为。

具体映射到软件开发中，抽象可以是接口或者抽象类形式。

```java
/**
 * Driveable 是接口，所以它是抽象
 */
public interface Driveable {
    void drive();
}
```

```java
/**
 * 而 Bike 实现了接口，它们被称为具体。
 */
public class Bike implements Driveable {
    @Override
    public void drive() {
        System.out.println("Bike drive");
    }
}
```

```java
/**
 * 而 Car实现了接口，它们被称为具体。
 */
public class Car implements Driveable {
    @Override
    public void drive() {
        System.out.println("Car drive.");
    }
}
```

依赖倒置的好处

在平常的开发中，我们大概都会这样编码。

```java
public class Person {

    private Bike mBike;
    private Car mCar;
    private Train mTrain;

    public Person(){
        mBike = new Bike();
        //mCar = new Car();
//        mTrain = new Train();
    }

    public void goOut(){
        System.out.println("出门啦");
        mBike.drive();
        //mCar.drive();
//        mTrain.drive();
    }
```

```
    public static void main(String ... args){
            //TODO:
        Person person = new Person();
        person.goOut();
    }
}
```

我们创建了一个 Person 类，它拥有一台自行车，出门的时候就骑自行车。

不过，自行车适应很短的距离。如果，我要出门逛街呢？自行车就不大合适了。于是就要改成汽车。

不过，如果我要到北京去，那么汽车也不合适了。

有没有一种方法能让 Person 的变动少一点呢？因为这是最基础的演示代码，如果工程大了，代码复杂了，Person 面对需求变动时改动的地方会更多。

而依赖倒置原则正好适用于解决这类情况。

下面，我们尝试运用依赖倒置原则对代码进行改造。

我们再次回顾下它的定义。

上层模块不应该依赖底层模块，它们都应该依赖于抽象。 抽象不应该依赖于细节，细节应该依赖于抽象。 首先是上层模块和底层模块的拆分。

按照决策能力高低或者重要性划分，Person 属于上层模块，Bike、Car 和 Train 属于底层模块。

上层模块不应该依赖于底层模块。  person架构

```
public class Person {

//    private Bike mBike;
    private Car mCar;
    private Train mTrain;
    private Driveable mDriveable;

    public Person(){
//        mBike = new Bike();
        //mCar = new Car();
       mDriveable = new Train();
    }

    public void goOut(){
        System.out.println("出门啦");
        mDriveable.drive();
        //mCar.drive();
//        mTrain.drive();
    }

    public static void main(String ... args){
```

```
            //TODO:
        Person person = new Person();
        person.goOut();
    }
}
```

可以看到，依赖倒置实质上是面向接口编程的体现。

控制反转 （IoC）

控制反转 IoC 是 Inversion of Control的缩写，意思就是对于控制权的反转，对么控制权是什么控制权呢？

Person自己掌控着内部 mDriveable 的实例化。 现在，我们可以更改一种方式。将 mDriveable 的实例化移到 Person 外面。

```
public class Person2 {

    private Driveable mDriveable;

    public Person2(Driveable driveable){
        this.mDriveable = driveable;
    }

    public void goOut(){
        System.out.println("出门啦");
        mDriveable.drive();
        //mCar.drive();
//        mTrain.drive();
    }

    public static void main(String ... args){
            //TODO:
        Person2 person = new Person2(new Car());
        person.goOut();
    }
}
```

就这样无论出行方式怎么变化，Person 这个类都不需要更改代码了。

在上面代码中，Person 把内部依赖的创建权力移交给了 Person2这个类中的 main() 方法。也就是说 Person 只关心依赖提供的功能，但并不关心依赖的创建。

这种思想其实就是 IoC，IoC 是一种新的设计模式，它对上层模块与底层模块进行了更进一步的解耦。控制反转的意思是反转了上层模块对于底层模块的依赖控制。

比如上面代码，Person 不再亲自创建 Driveable 对象，它将依赖的实例化的权力交接给了 Person2。而 Person2在 IoC 中又指代了 **IoC 容器** 这个概念。

依赖注入（Dependency injection）

依赖注入，也经常被简称为 DI，其实在上一节中，我们已经见到了它的身影。它是一种实现 IoC 的手段。什么意思呢？

为了不因为依赖实现的变动而去修改 Person，也就是说以可能在 Driveable 实现类的改变下不改动 Person 这个类的代码，尽可能减少两者之间的耦合。我们需要采用上一节介绍的 IoC 模式来进行改写代码。

这个需要我们移交出对于依赖实例化的控制权，那么依赖怎么办？Person 无法实例化依赖了，它就需要在外部（IoC 容器）赋值给它，这个赋值的动作有个专门的术语叫做注入（injection），需要注意的是在 IoC 概念中，这个注入依赖的地方被称为 IoC 容器，但在依赖注入概念中，一般被称为注射器 （injector)。

表达通俗一点就是：我不想自己实例化依赖，你（injector）创建它们，然后在合适的时候注入给我

实现依赖注入有 3 种方式：

1. 构造函数中注入
2. setter 方式注入
3. 接口注入

```
/**
 * 接口方式注入
 * 接口的存在，表明了一种依赖配置的能力。
 */
public interface DepedencySetter {
    void set(Driveable driveable);
}
```

```
public class Person2  implements DepedencySetter {

    //接口方式注入
    @Override
    public void set(Driveable driveable) {
        this.mDriveable = mDriveable;
    }

    private Driveable mDriveable;

    //构造函数注入
    public Person2(Driveable driveable){
        this.mDriveable = driveable;
    }

    //setter 方式注入
    public void setDriveable(Driveable mDriveable) {
        this.mDriveable = mDriveable;
    }

    public void goOut(){
```

```
        System.out.println("出门啦");
        mDriveable.drive();
        //mCar.drive();
//        mTrain.drive();
    }

    public static void main(String ... args){
            //TODO:
        Person2 person = new Person2(new Car());
        person.goOut();
    }
}
```

Java 依赖注入标准

JSR-330 是 Java 的依赖注入标准。定义了如下的术语描述依赖注入：

- A 类型依赖 B类型（或者说 B 被 A 依赖），则 A类型 称为”依赖(物) dependency”
- 运行时查找依赖的过程，称为”解析 resolving“依赖
- 如果找不到依赖的实例，称该依赖是”不能满足的 unsatisfied”
- 在”依赖注入 dependency injection”机制中，提供依赖的工具称为”依赖注入器 dependency injector，注射器”

> *在标准中, 依赖是类型而不是实例/ 对象; 在程序中（运行时）, 需要的是依赖的实例.*

javax.inject

包 javax.inject 指定了获取对象的一种方法，该方法与构造器、工厂以及服务定位器（例如 JNDI））这些传统方法相比可以获得更好的可重用性、可测试性以及可维护性。此方法的处理过程就是大家熟知的依赖注入，它对于大多数应用是非常有价值的。

@Inject

注解 @Inject 标识了可注入的构造器、方法或字段。可以用于静态或实例成员。一个可注入的成员可以被任何访问修饰符（private、package- private、protected、public）修饰。注入顺序为构造器，字段，最后是方法。超类的字段、方法将优先于子类的字段、方法被注入。对于 同一个类的字段是不区分注入顺序的，同一个类的方法亦同

Provider

接口 Provider 用于提供类型 T 的实列。Provider 是一般情况是由注入器实现的。对于任何可注入的 T 而言，您也可以注入 Provider。与直接注入 T 相比，注入 Provider 使得：

- 可以返回多个实例。

- 实例的返回可以延迟化或可选
- 打破循环依赖。
- 可以在一个已知作用域的实例内查询一个更小作用域内的实例。

```
    class Car {
       @Inject Car(Provider<Seat> seatProvider) {
            Seat driver = seatProvider.get();
            Seat passenger = seatProvider.get();
           …
    }
}
```

- get() 用于提供一个完全构造的类型 T 的实例。 异常抛出：RuntimeException —— 当注入器在提供实例时遇到错误将抛出此异常。例如，对于一个可注入的成员 T 抛出了一个异常，注入器将包装此异常并将它抛给 get() 的调用者。调用者不应该尝试处理此类异常，因为不同注入器实现的行为不一样，即使是同一个注入器，也会因为配置不同而表现的行为不同。

@Qualifier

用于标识限定器注解。任何人都可以定义新的限定器注解。一个限定器注解：

- 是被 @Qualifier、@Retention(RUNTIME) 标注的，通常也被 @Documented 标注。
- 可以拥有属性。
- 可能是公共 API 的一部分，就像依赖类型一样，而不像类型实现那样不作为公共 API 的一部分。
- 如果标注了 @Target 可能会有一些用法限制。本规范只是指定了限定器注解可以被使用在字段和参数上，但一些注入器配置可能使用限定器注解在其他一些地方（例如方法或类）上。

@Named

- 基于 String 的[限定器]

@Scope

- 用于标识作用域注解。一个作用域注解是被标识在包含一个可注入构造器的类上的，用于控制该类型的实例如何被注入器重用。缺省情况下，如果没有标识作用域注解，注入器将为每一次注入都创建（通过注入类型的构造器）新实例，并不重用已有实例。如果多个线程都能够访问一个作用域内的实例，该实例实现应该是线程安全的。作用域实现由注入器完成。

@Singleton

- 标识了注入器只实例化一次的类型。该注解不能被继承