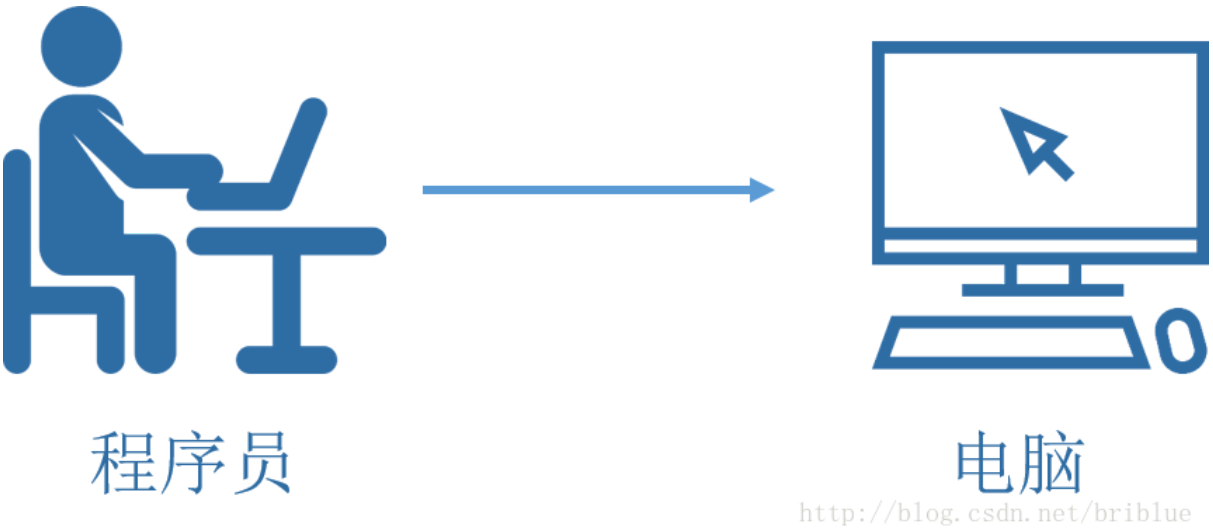
什么是依赖（Dependency）？

依赖是一种关系，通俗来讲就是一种需要。



程序员需要电脑，因为没有电脑程序员就没有办法编写代码，所以说程序员**依赖**电脑，电脑被程序员依赖。

在面向对象编程中，代码可以这样编写。

```
class Coder {

    Computer mComputer;

    public Coder () {
        mComputer = new Computer();
    }
}
```

Coder 的内部持有 Computer 的引用，这就是依赖在编程世界中的体现。

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**。

有人会有疑惑，设计原则有什么用呢?

设计原则是前辈们总结出来的经验，你可以把它们看作是内功心法。

只要你在平常开发中按照设计原则进行编码，假以时日，你编程的功力将会大增。

依赖倒置原则的定义如下：

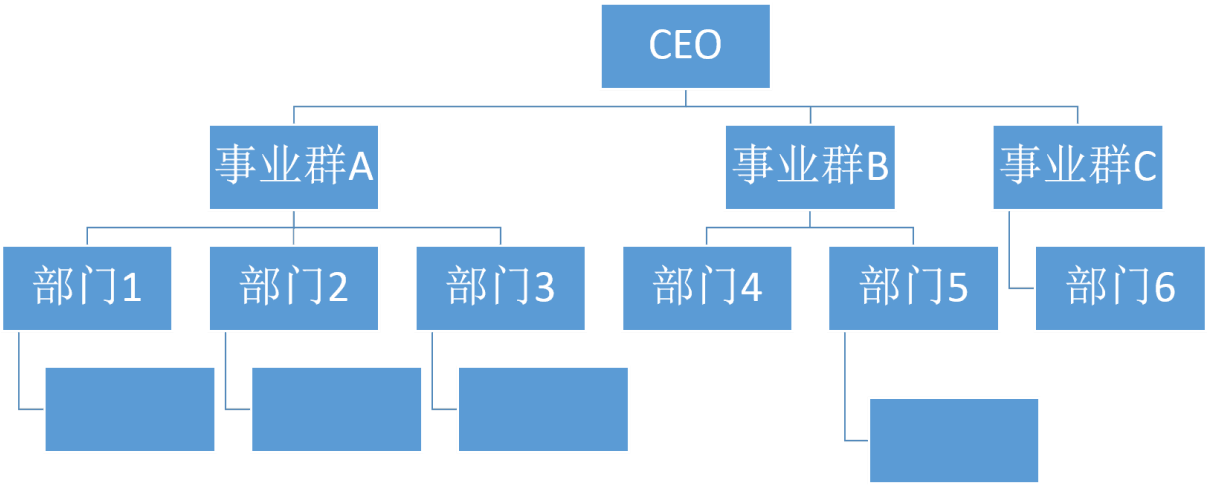
> 1. 上层模块不应该依赖底层模块，它们都应该依赖于抽象。
> 2. 抽象不应该依赖于细节，细节应该依赖于抽象。

乍一看，这会让初学者摸不清头脑。这种学术性的概括语言近乎于软件行业中的哲学。可实质上，它确实称得上是哲学，现在 SOLID 几乎等同于面向对象开发中的金科玉律，但是也正因为它的高度概括、它的晦涩难懂，对于广大初学者而言这是一件非常不友好的事物。

我们该怎么理解上面的定义呢？我们需要咬文嚼字，各个突破。

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。



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。

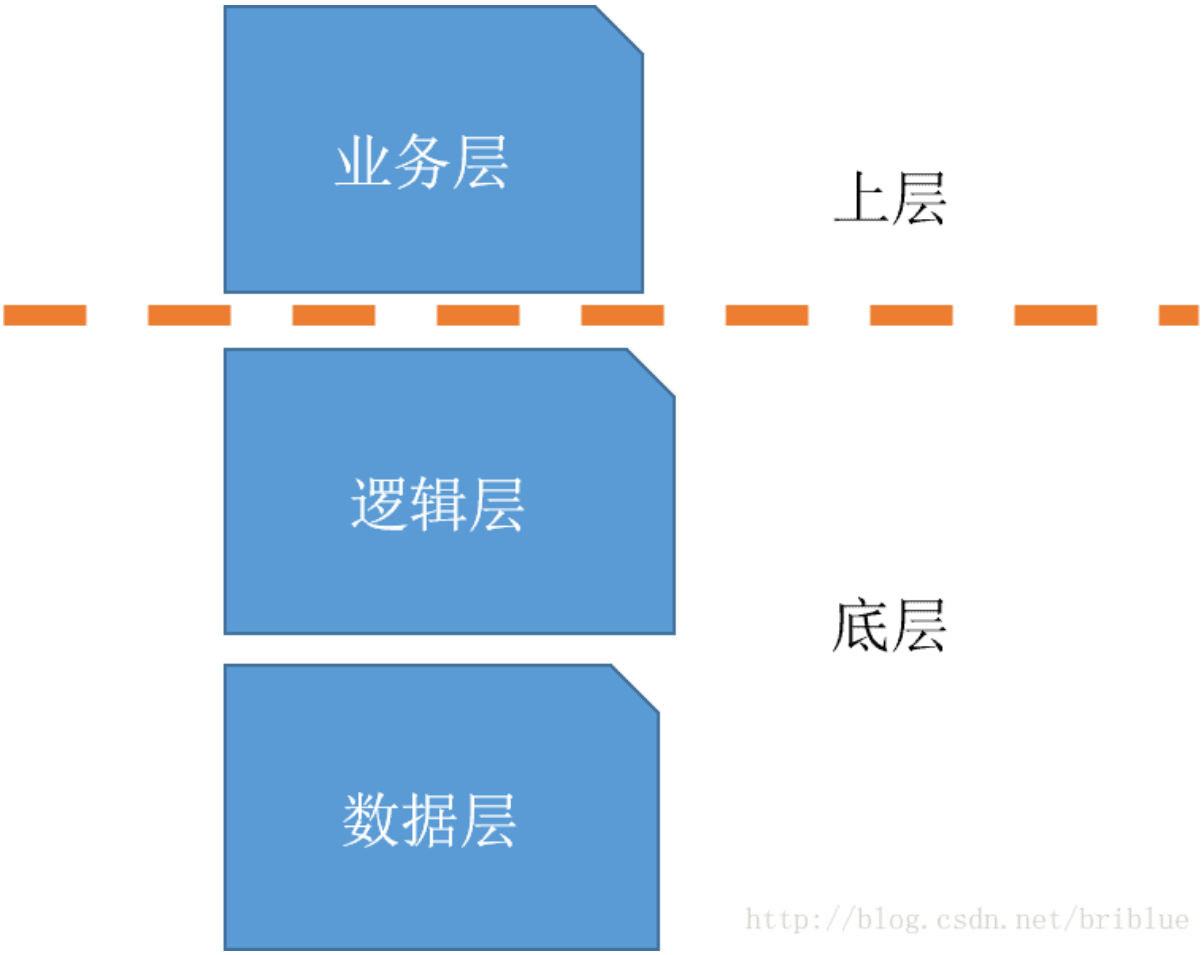
然后，我们再以事业群为整个体系划分模块，各个部门经理以上部分是上层，那么之下的组织都可以称为底层。

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？

抽象如其名字一样，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。

```
public interface Driveable {
    void drive();
}

class Bike implements Driveable{

    @Override
    public void drive() {
        // TODO Auto-generated method stub
        System.out.println("Bike drive.");
    }

}

class Car implements Driveable{

    @Override
    public void drive() {
        // TODO Auto-generated method stub
        System.out.println("Car drive.");
    }

}
```

Driveable 是接口，所以它是抽象，而 Bike 和 Car 实现了接口，它们被称为具体。

现在，我们理解了依赖、上层模块、底层模块、抽象和具体。这样我们可以正式开始学习依赖倒置原理这个概念了？

依赖倒置的好处

在平常的开发中，我们大概都会这样编码。

```
public class Person {

```

```
    private Bike mBike;

    public Person() {
        mBike = new Bike();
    }

    public void chumen() {
        System.out.println("出门了");
        mBike.drive();
    }

}
```

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。

```
public class Test1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Person person = new Person();

        person.chumen();

    }

}
```

执行结果如下：

```
出门了
Bike drive.
```

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。

```
public class Person {

    private Bike mBike;
    private Car mCar;

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

```

```
    public void chumen() {
        System.out.println("出门了");
        //mBike.drive();
        mCar.drive();
    }

}
```

我们需要修改 Person 这个类的代码。

不过，如果我要到北京去，那么汽车也不合适了。

```
class Train implements Driveable{
    @Override
    public void drive() {
        // TODO Auto-generated method stub
        System.out.println("Train drive.");
    }
}

package com.frank.test;

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 chumen() {
        System.out.println("出门了");
        //mBike.drive();
        //mCar.drive();
        mTrain.drive();
    }

}
```

我们添加了 Train 这个最新的实现类，然后再次修改了 Person 这个类。

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。

> 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。

首先是上层模块和底层模块的拆分。

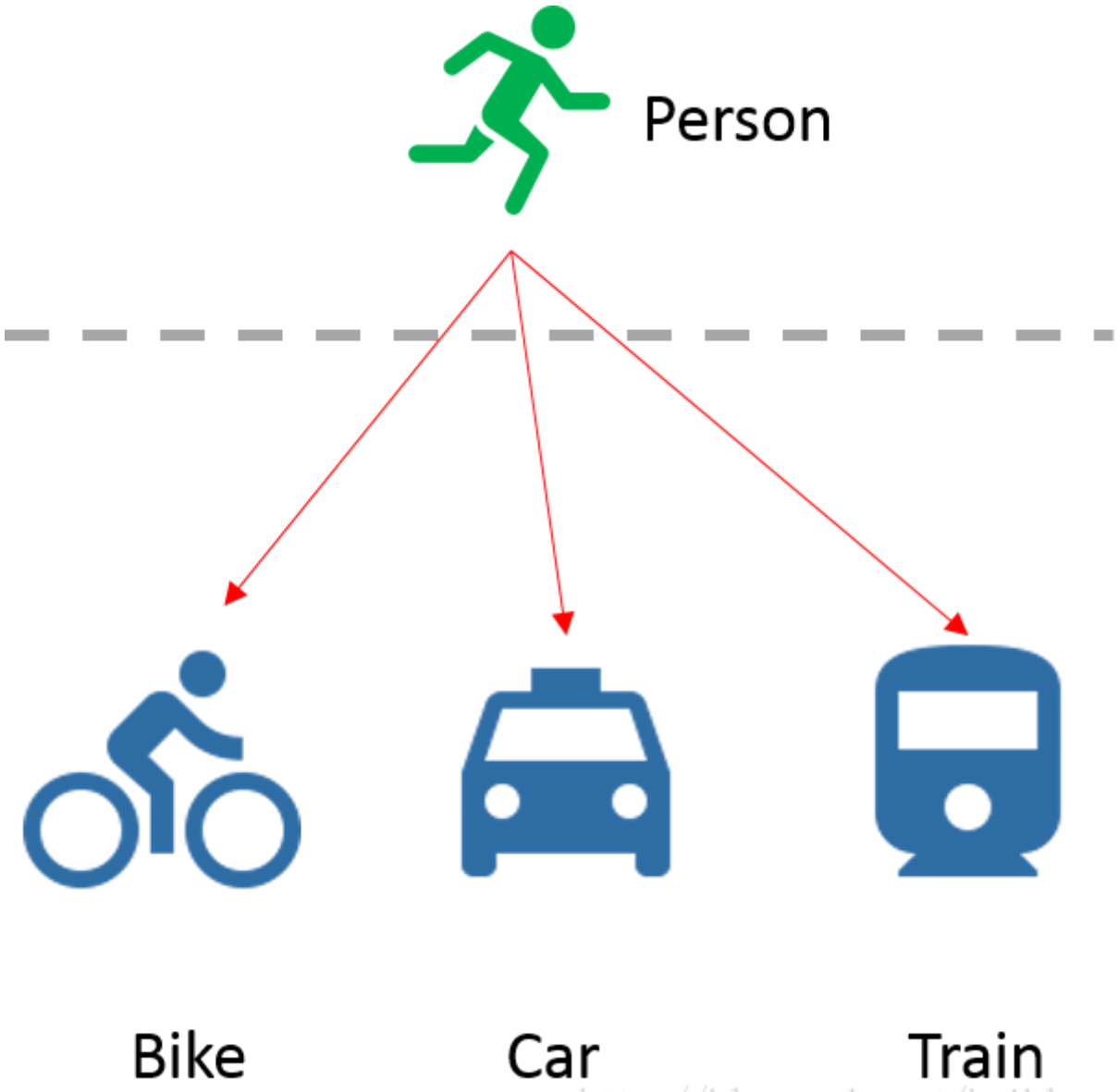
按照决策能力高低或者重要性划分，Person 属于上层模块，Bike、Car 和 Train 属于底层模块。

上层模块不应该依赖于底层模块。
但是

```
public class Person {

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

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


}
```

Person 这个类显然是依赖于 Bike 和 Car。Person 类中 chumen() 的能力完全依赖于属性 Bike 或者 Car 对象，也就是说 Person 把自己的能力依赖在 Bike 和 Car 身上。

上层和底层都应该依赖于抽象。



我们的代码中，Person 没有依赖抽象，所以我们得引进抽象。

而底层的抽象是什么，是 Driveable 这个接口。

```
public class Person {

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

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

```
    }

    public void chumen() {
        System.out.println("出门了");
        //mBike.drive();
        //mCar.drive();
        //mTrain.drive();
        mDriveable.drive();
    }

}
```

执行结果如下：

```
出门了
Train drive.
```

现在，Person 类中 chumen() 这个方法依赖于 Driveable 接口的抽象，它没有限定自己出行的可能性，任何 Car、Bike 或者是 Train 都可以的。

到这一步，我们可以说是符合了上层不依赖于底层，依赖于抽象的准则了。

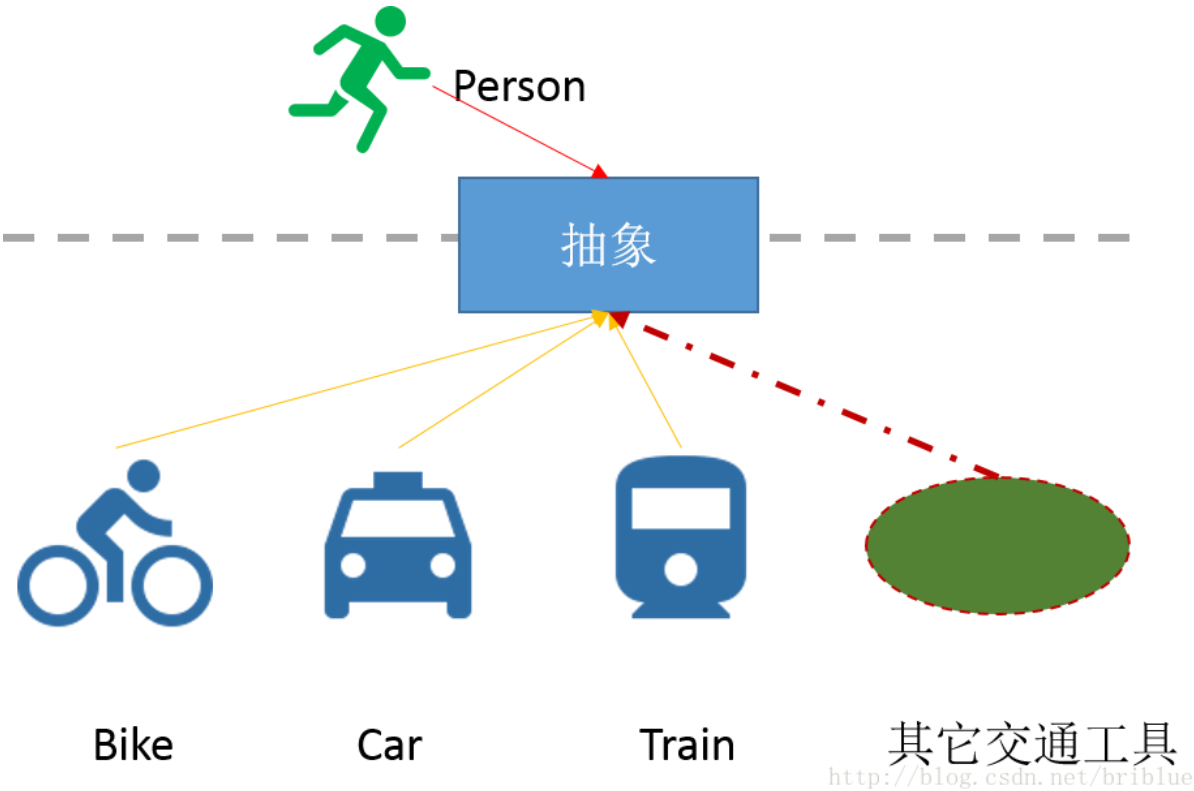
那么，抽象不应该依赖于细节，细节应该依赖于抽象又是什么意思呢？

以上面为例，Driveable 是抽象，它代表一种行为，而 Bike、Car、Train 都是实现细节。

Person 需要的是 Driveable，需要的是交通工具，但不是说交通工具一定是 Bike、Car、Train。未来也可能是 AirPlane。

```
class AirPlane implements Driveable{
    @Override
    public void drive() {
        // TODO Auto-generated method stub
        System.out.println("AirPlane fly.");
    }
}
```

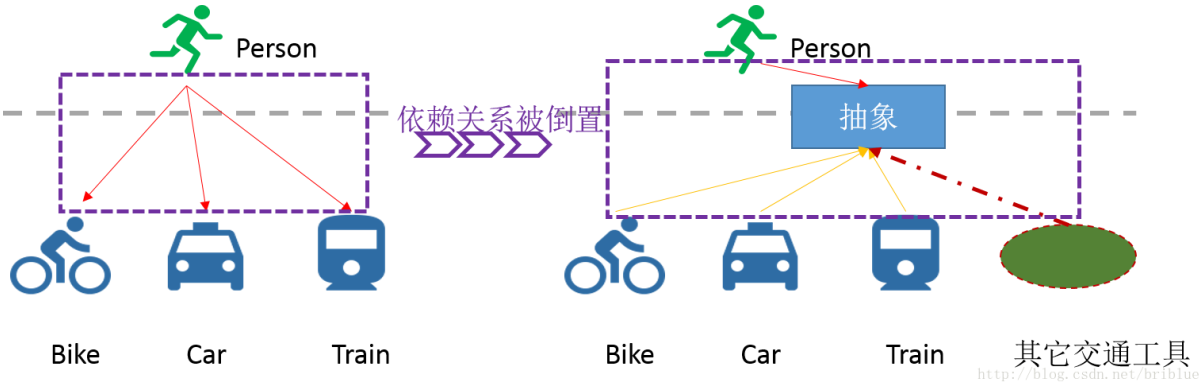
那么一个 Person，它下次出门改成飞机可以吗？当然可以的。因为依赖倒置的缘由，Person 展现出了极度的可扩展性。



上面的内容就是依赖倒置原则。

有人会考虑到倒置这个词，个人的理解是倒置是改变的意思。

本来正常编码下，肯定会出现上层依赖底层的情况，而依赖倒置原则的应用则改变了它们之间依赖的关系，它引进了抽象。上层依赖于抽象，底层的实现细节也依赖于抽象，所以依赖倒置我们可以理解为依赖关系被改变，如果非常纠结于倒置这个词，那么倒置的其实是底层细节，原本它是被上层依赖，现在它倒要依赖与抽象的接口。



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?
大家重新审视上面的代码。

```
public class Person {

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

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

    public void chumen() {
        System.out.println("出门了");
        //mBike.drive();
        //mCar.drive();
        //mTrain.drive();
        mDriveable.drive();
    }

}
```

虽然，chumen() 这个方法不再因为出行方式的改变而变动，但是每次更改出行方式的时候，Person 这个类还是要修改。

Person 类还是要实例化 mDriveable 的接口对象。

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

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 Person {

```

```
    private Driveable mDriveable;

    public Person(Driveable driveable) {

        this.mDriveable = driveable;
    }

    public void chumen() {
        System.out.println("出门了");

        mDriveable.drive();
    }

}
```

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。

```
public class Test1 {


    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Bike bike = new Bike();
        Car car = new Car();
        Train train = new Train();
//      Person person = new Person(bike);
//      Person person = new Person(car);
        Person person = new Person(train);


        person.chumen();

    }

}
```

在上面代码中，Person 把内部依赖的创建权力移交给了 Test1 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 Test1。而 Test1 在 IoC 中又指代了 **IoC 容器** 这个概念。

再举一个例子，我们到餐厅去叫外卖，餐厅有专门送外卖的外卖员，他们的使命就是及时送达外卖食品。

依照**依赖倒置原则**，我们可以创建这样一个类。

```
public abstract class WaimaiYuan {

    protected Food food;


    public WaimaiYuan(Food food) {
        this.food = food;
    }

    abstract void songWaiMai();

}

public class Xiaohuozi extends WaimaiYuan {


    public Xiaohuozi(Food food) {
        super(food);
    }

    @Override
    void songWaiMai() {
        System.out.println("我是小伙子,为您送的外卖是："+food);

    }

}

public class XiaoGuniang extends WaimaiYuan {


    public XiaoGuniang(Food food) {
        super(food);
    }

    @Override
    void songWaiMai() {
        System.out.println("我是小姑娘,为您送的外卖是："+food);
    }

}

```

```
43
44
```

WaimaiYuan 是抽象类，代表送外卖的，Xiaohuozi 和 XiaoGuniang 是它的继承者，说明他们都可以送外卖。WaimaiYuan 都依赖于 Food，但是它没有实例化 Food 的权力。

再编写食物类代码

```
1  public abstract class Food {
2      protected String name;
3
4      @Override
5      public String toString() {
6          return name;
7      }
8
9  }
10
11 public class PijiuYa extends Food {
12
13     public PijiuYa() {
14         name = "啤酒鸭";
15     }
16
17 }
18
19 public class DuojiaoYutou extends Food {
20
21     public DuojiaoYutou() {
22         name = "剁椒鱼头";
23     }
24
25 }
```

Food 是抽象类，PijiuYa 和 DuojiaoYutou 都是实现细节。

IoC 少不了 IoC 容器，也就是实例化抽象的地方。我们编写一个餐厅类。

```
1  public class Restaurant {
2
3      public static void peican(int orderid,int flowid) {
4          WaimaiYuan person;
5          Food food;
6
7          if ( orderid == 0) {
8              food = new PijiuYa();
```

```
        } else {
            food = new DuojiaoYutou();
        }

        if ( flowid % 2 == 0 ) {
            person = new Xiaohuozi(food);
        } else {
            person = new XiaoGuniang(food);
        }

        person.songWaiMai();

    }

}
```

orderid 代表菜品编号，0 是啤酒鸭，其它则是剁椒鱼头。
flowid 是订单的流水号码。 餐厅根据流水编码的不同来指派小伙子或者小姑娘来送外卖，编写测试代码。

```
public class IocTest {

    public static void main(String[] args) {

        Restaurant.peican(0, 0);
        Restaurant.peican(0, 1);
        Restaurant.peican(1, 2);
        Restaurant.peican(0, 3);
        Restaurant.peican(1, 4);
        Restaurant.peican(1, 5);
        Restaurant.peican(1, 6);
        Restaurant.peican(0, 7);
        Restaurant.peican(0, 8);
    }

}
```

餐厅一次性送了 9 份外卖。

```
我是小伙子,为您送的外卖是：啤酒鸭
我是小姑娘,为您送的外卖是：啤酒鸭
我是小伙子,为您送的外卖是：剁椒鱼头
我是小姑娘,为您送的外卖是：啤酒鸭
我是小伙子,为您送的外卖是：剁椒鱼头
我是小姑娘,为您送的外卖是：剁椒鱼头
我是小伙子,为您送的外卖是：剁椒鱼头
```

```
我是小姑娘,为您送的外卖是：啤酒鸭
我是小伙子,为您送的外卖是：啤酒鸭
```

可以看到的是，因为有 Restaurant 这个 IoC 容器存在，大大地解放了外卖员的生产力，外卖员不再依赖具体的食物，具体的食物也不再依赖于特定的外卖员。也就是说，只要是食物外卖员就可以送，任何一种食物可以被任何一位外卖员送。

大家细细体会这是怎么样一种灵活性。如果非要外卖员自己决定配送什么食物，人少则还行，人多的时候，订单多的时候肯定会乱成一锅粥。

所以，实际工作当中，基本上都是按照专业的人干专业的事这种基本规律运行。外卖员没有能力也没有义务去亲自决定该送什么订单，这种权力在于餐厅，只要餐厅配置好就 OK 了。

记住 **配置** 这个词。

在软件开发领域，类似餐厅这种调度配置然后决定依赖关系的 IOC 容器有许多框架比如 Spring。但是，由于我本身是 Android 开发的，对于 Spring 知之甚少，所以对这一块不做过多介绍。

但作为 IoC 容器，无非是针对配置然后动态生成依赖关系。有的配置是开发者按照规则编写在 xml 格式文件中，有些配置则是利用 Java 中的反射与注解。

IoC 模式最核心的地方就是在于依赖方与被依赖方之间，也就是上文中说的上层模块与底层模块之间引入了第三方，这个第三方统称为 IoC 容器，因为 IoC 容器的介入，导致上层模块对于它的依赖的实例化控制权发生变化，也就是所谓的控制反转的意思。

总之，因为 IoC 容器的存在，使得开发者编写大型系统工程的时候极大地解放了生产力。

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？

```
public class Person {


    private Driveable mDriveable;

    public Person() {

        mDriveable = new Train();

    }

    public void chumen() {
        System.out.println("出门了");
```

```
14
15          mDriveable.drive();
16      }
17
18  }
```

我们再回顾 Person 这个类。在构造 Person 的时候，Person 内部初始化了 Driveable 对象，选择了 Train() 为实现，这种编码方式太具有局限性了。下次选择其它出行方式如 Bike 或者 Car 的时候，Person 这个类需要修改。

```
1  public class Person {
2
3      private Driveable mDriveable;
4
5      public Person() {
6          mDriveable = new Bike();
7          //mDriveable = new Car();
8          //mDriveable = new Train();
9
10     }
11
12     public void chumen() {
13         System.out.println("出门了");
14
15         mDriveable.drive();
16     }
17
18 }
```

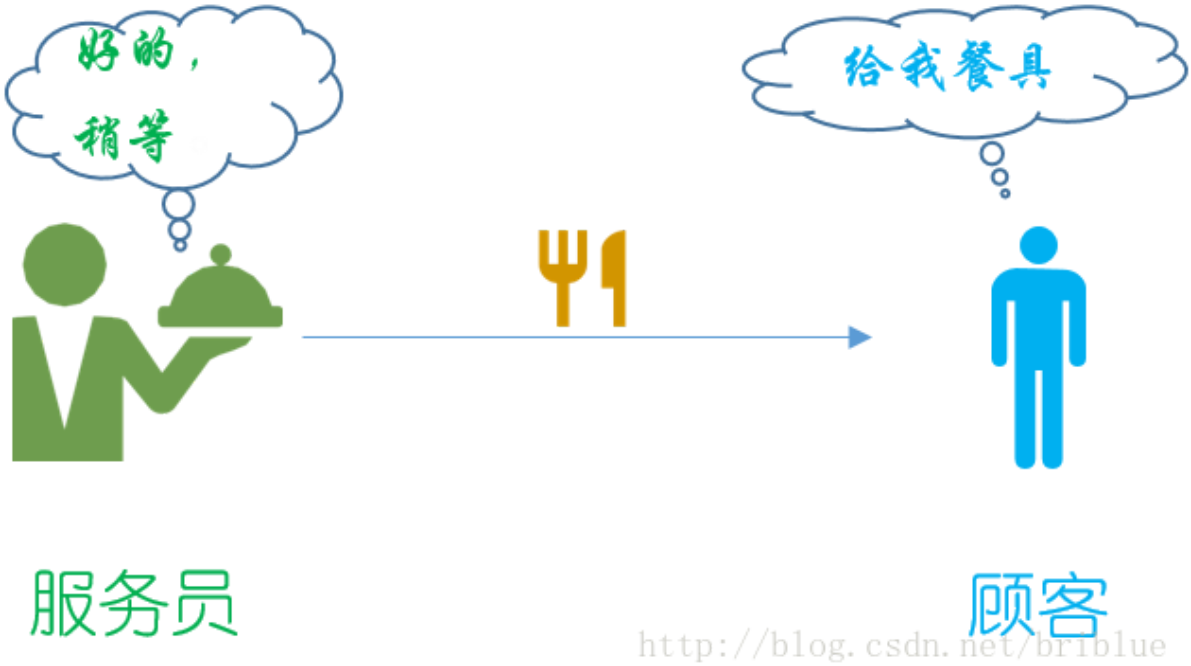
为了不因为依赖实现的变动而去修改 Person，也就是说以可能在 Driveable 实现类的改变下不改动 Person 这个类的代码，尽可能减少两者之间的耦合。我们需要采用上一节介绍的 IoC 模式来进行改写代码。

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吧。

再比如顾客去餐厅需要碗筷，但是顾客不需要自己带碗筷去，所以，在点菜的时候和服务员说，你给我一副碗筷吧。在这个场景中如果按照正常的编程方式，碗筷本身是顾客的依赖，但是应用 IoC 模式之后 ，碗筷是服务员提供（注入）给顾客的，顾客不用关心吃饭的时候用什么碗筷，因为吃不

同的菜品，可能餐具不同，吃牛排用刀叉，喝汤用调羹，虽然顾客就餐时需要餐具，但是餐具的配置应该交给餐厅的工作人员。



如果以软件角度来描述，餐具是顾客是依赖，服务员给顾客配置餐具的过程就是依赖注入。

上一节的外卖员和菜品的例子，其实也是依赖注入的例子。

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

我们现在一一观察这些方式

构造函数注入

```
public class Person {


    private Driveable mDriveable;

    public Person(Driveable driveable) {

        this.mDriveable = driveable;
    }

    public void chumen() {
        System.out.println("出门了");
```

```
13
14          mDriveable.drive();
15      }
16
17  }
```

优点：在 Person 一开始创建的时候就确定好了依赖。
缺点：后期无法更改依赖。

setter 方式注入

```
1   public class Person {
2
3       private Driveable mDriveable;
4
5       public Person() {
6
7       }
8
9       public void chumen() {
10          System.out.println("出门了");
11
12          mDriveable.drive();
13      }
14
15
16      public void setDriveable(Driveable mDriveable) {
17          this.mDriveable = mDriveable;
18      }
19
20  }
```

优点：Person 对象在运行过程中可以灵活地更改依赖。
缺点：Person 对象运行时，可能会存在依赖项为 null 的情况，所以需要检测依赖项的状态。

```
1   public void chumen() {
2
3       if ( mDriveable != null ) {
4           System.out.println("出门了");
5           mDriveable.drive();
6       }
7
8       }
```

接口方式注入

```java
public interface DepedencySetter {
    void set(Driveable driveable);
}

class Person implements DepedencySetter{
    private Driveable mDriveable;

    public void chumen() {

        if ( mDriveable != null ) {
            System.out.println("出门了");
            mDriveable.drive();
        }

    }

    @Override
    public void set(Driveable driveable) {
        this.mDriveable = mDriveable;
    }

}
```

这种方式和 Setter 方式很相似。有很多同学可能有疑问那么加入一个接口是不是多此一举呢?

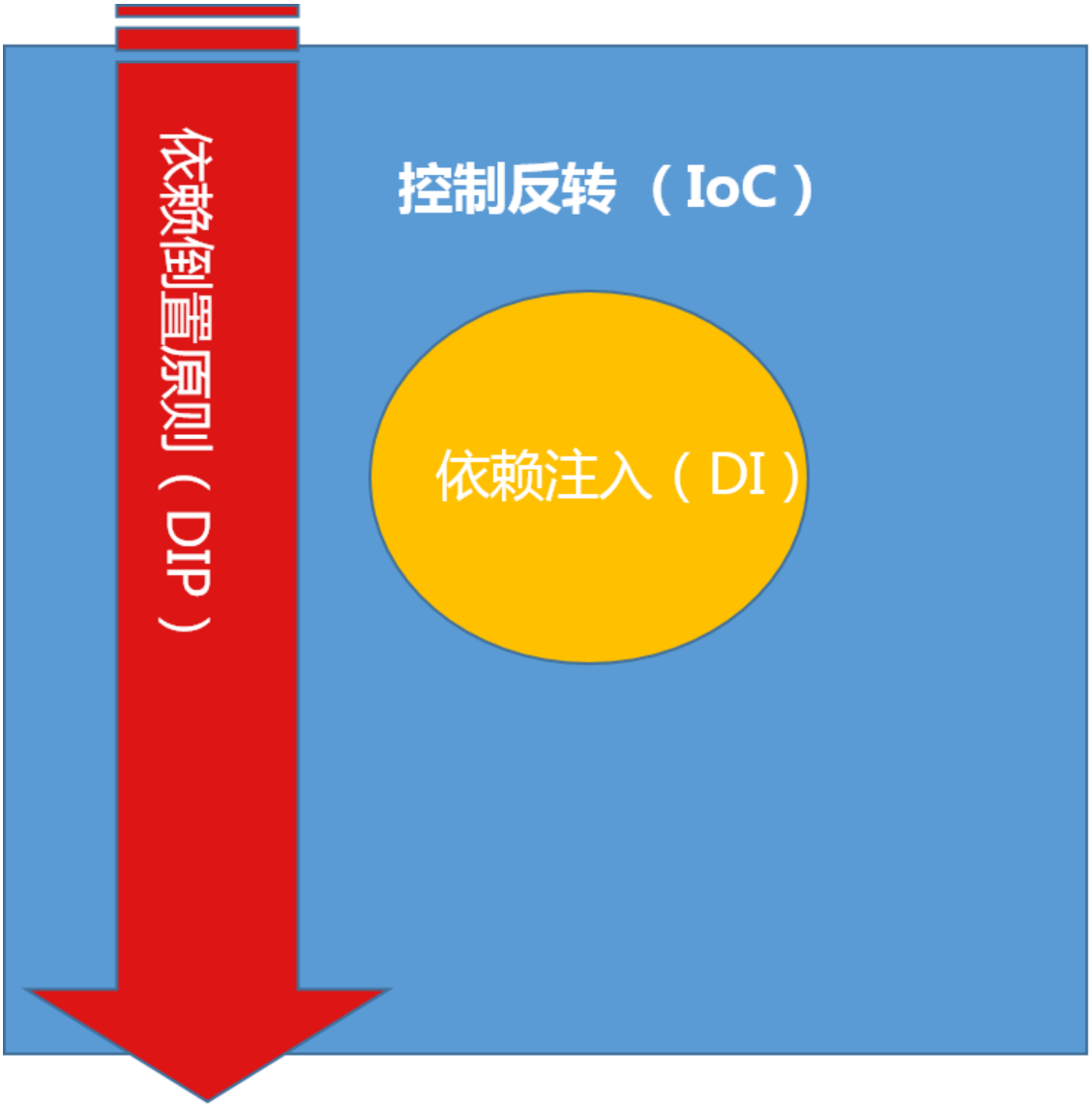
答案肯定是不是的，这涉及到一个角色的问题。还是以前面的餐厅为例，除了外卖员之外还有厨师和服务员，那么如果只有外卖员实现了一个送外卖的接口的话，那么餐厅配餐的时候就只会把外卖配置给外卖员。

接口的存在，表明了一种依赖配置的能力。

在软件框架中，读取 xml 配置文件，或者是利用反射技术读取注解，然后根据配置信息，框架动态将一些依赖配置给特定接口的类，**我们也可以说 Injector 也依赖于接口，而不是特定的实现类**，这样进一步提高了准确性与灵活性。

总结

1. 依赖倒置是面向对象开发领域中的软件设计原则，它倡导上层模块不依赖于底层模块，抽象不依赖细节。
2. 依赖反转是遵守依赖倒置这个原则而提出来的一种设计模式，它引入了 IoC 容器的概念。
3. 依赖注入是为了实现依赖反转的一种手段之一。
4. 它们的本质是为了代码更加的**“高内聚,低耦合”**。



1. 控制反转是设计模式，遵从了依赖倒置原则
2. 依赖注入是实现控制反转的手段

这篇文章我运用了大量的比喻来解释让给概念，我相信能够加深读者们对于这些概念的理解。