P1：老师在课上曾经讲过5种设计模式。在那的23个设计模式中，还有一些其它的设计模式值得一说。今天我要说的就是代理模式，以及它的典型应用——面向切面编程（AOP）

P2：如果有自己搜过Spring AOP，面向切面编程之类的概念，那大概率可能会看到这么一坨概念。看上去十分的让人恼火，但是我们大可以先不管他。

P3：先去学习代理模式的话就能更好理解AOP了，因为AOP是对代理模式的一个应用。

P4：概括的来说，代理模式的基本思想是：我们不直接使用我们需要用到的目标类，而是使用一个代理类来作为访问目标类的中介，目标类专门写我们需要的最核心的功能，代理类中可以写一些补充性的代码。

接下来呢，我们通过一个例子来具体介绍代理模式。

P5：假设咱们正在上一门课，叫机器学习基础。我们现在需要写一个分类器，咱们选择写决策树分类。鉴于将来咱们还有可能写些别的分类器，咱们就先定义一个分类器接口（参见依赖倒置原则：依赖于抽象概念而不依赖与具体实现），然后写一个决策树分类器实现这个接口。

P6：但是现在我们多了一个需求：我们要对数据进行数据预处理，然后才能把数据扔到分类器里跑。我们一下子就可以想到：在接口和实现类里都加一个预处理方法。但是这个就有一些问题。

预处理和分类算法是两个彼此相对独立的功能，如果它们都很复杂的话，根据单一职责原则，我们最好是将这两个功能分开写，以避免未来可能遇到的种种问题。

最典型的问题是：如果将来我们需要改变数据预处理的逻辑的话，我们怎么办？如果我们直接修改决策树类，那就违背了开闭原则（不去修改、而是扩展），这很不好，而且要是数据预处理逻辑改来改去的话，那血压就直接升高了。

P7：代理模式就是一种能将预处理这样的操作分离出来的模式。我们新建一个类叫含预处理分类器的类，作为一个代理类，它也实现接口，同时引用了一个决策树分类器对象，我们称这个被代理类引用的对象为目标类。它内部的写法大概是这样的（click）。

我们可以发现：所有的训练和判断的逻辑还是写在决策树类里了，而代理类只负责写数据预处理的内容，对于训练和判断都是直接用目标类的方法完成的。

最后呈现出来的结果就是：数据预处理和分类算法这两部分被分开了。

对于编写代码的人来说，如果将来想添加新的数据预处理方法，咱们新建一个代理类，然后写一个新的数据预处理方法就可以，不需要动旧的代码。

对于使用代码的人来说，直接使用代理类就可以享受应有的功能，不需要也不能去管目标类是啥、该怎么用。这也是一个重要的特点，是它与装饰器模式最直观的区别之一。装饰器模式有兴趣大家可以自己了解一下，Java I/O库的设计就使用到了装饰器模式。

P8：像上面那样，预先写好代理类的代理模式被称为静态代理，它还有很多问题。假设现在需求变了，我们要写好几个分类器算法，并且要做统一的数据预处理。（click）如果我们用继承来解决的话，还是会遇到预处理和算法没分开的问题。

P9：如果我们用刚才介绍的那种静态代理，就会出现这种情况。每个目标类都得有一个代理类，代理类的数量跟着目标类一起爆炸了，想改预处理算法的时候还要到处改，基本上啥好处也没捞着。

这是由于代理模式的特点：代理模式要求用户只使用代理类，不能与目标类产生任何联系，（click）而如果代理类引用的是抽象的接口，就需要调用者自己传入一个目标类了（如果这么干就更像装饰类了），因此代理器中必须使用具体的目标类，而不能是抽象的接口。正是因为这个局限，即使这些目标类所使用的代理的逻辑完全一样，也得一个个写代理类。

但是这些逻辑毕竟是完全一样的，有没有可能只写一遍代理的逻辑，然后用这个逻辑造出很多代理类呢？

P10：解决这个问题的就是动态代理机制，它的基本模式长这么个德行。它用到了“反射”机制，这个机制允许你在程序运行时动态创建出一个不存在于源代码中的类，Java类库使用这一套机制实现了用于动态代理的库。

简单理解的话，你可以认为只要你告诉这个库：

你要代理的目标类、

目标类实现的所有接口、

以及一个你写的方法invoke，

你就可以调用库的方法，在运行时帮你动态创建出一个代理类并将这个类的一个对象返回给你。这个代理类实现了你输入的所有接口。

你在调用原有的方法时都会转而执行invoke方法，并将原本执行的方法对应的Method类对象作为参数传给invoke方法（这也是Java标准库中的类，是记录一个方法的所有信息的类）。你可以借助这个Method类继续执行原来的方法，或者获取原来的方法的信息，也可以添加自己想要的其他内容。

这个挺复杂，而且还有很多可以说，但是总的来说就是：动态代理可以让你用一套代码、一套代理逻辑，不断生成代理类和对象，不论目标类之间是什么样的关系。这样，如果有一批需要进行相同的额外处理的对象，我们就可以把这个处理逻辑写进之前说的invoke方法里，然后用动态代理机制获取我们想要的代理类对象了。

P11：面向切面编程的思想，就是从软件系统的对象和业务逻辑中，发现那些不同领域的对象都要使用的功能，将它们提取出来，让各个类共用。

P12：值得强调的是，面向切面编程的目的是解决面向对象程序设计所面临的问题。像这样四个业务流程，面向对象程序设计，能很好地将决策树和KNN归纳为统一接口下的对象，并对分类器、可视化工具、数据转移中介进行封装，但是不太好将数据预处理这个逻辑提取出来——毕竟它不独属于某一个具体的类。

面向切面编程把这些需要统一操作的位置的集合称为“切点”，比如决策树和KNN进行模型训练之前、可视化工具进行数据结构组织之前；把我们想要做出的操作称为“增强”，在这个例子里就是数据预处理。两者结合构成的“在哪里做什么”的完整定义就是切面。

而代理模式，正是可以提取出各个对象中相同的逻辑，编写后让各个类共用的设计模式，面向切面编程就是代理模式的一种应用，其实根本上的、最重要的，还是对代理模式的优势的理解。

而做了这么多，我们的根本目的其实还是加强代码的复用程度，尽可能符合开闭原则，让我们编程更舒服，这也是所有设计模式、编程理念的根本了。

P13：实战中，往往有更为简洁的API来帮助我们完成面向切面编程。在Spring中，Spring AOP模块就替我们封装了AOP背后的细节，让我们可以通过更简单的配置使用AOP。图中，就是通过注解配置一个切面的实例。如果对面向切面编程实战有兴趣，大家可以去学一学Spring AOP的API。