# 23种设计模式

## UML类图

### 1UML类图关系准备知识：

uml图包含很多种图示类型，uml类图只是其中一种，还有时序图等等待整理。

类图/接口图的结构——矩形表示法：类图：一个矩形分成三块（类名、变量、方法）、接口图：一个接口分成两块（无变量）。

+-#~：public、private、protected、default（或不写）

接口图中，类名上要加<interface>。接口还有棒棒糖表示法。

### 2UML类图关系记忆：菱三三：

菱形。菱空实。聚合组合

空三。线虚实。实现泛化

实三。线虚实。依赖关联

含义分别如下。

含有菱形。按菱形空心实心。得到聚合组合。

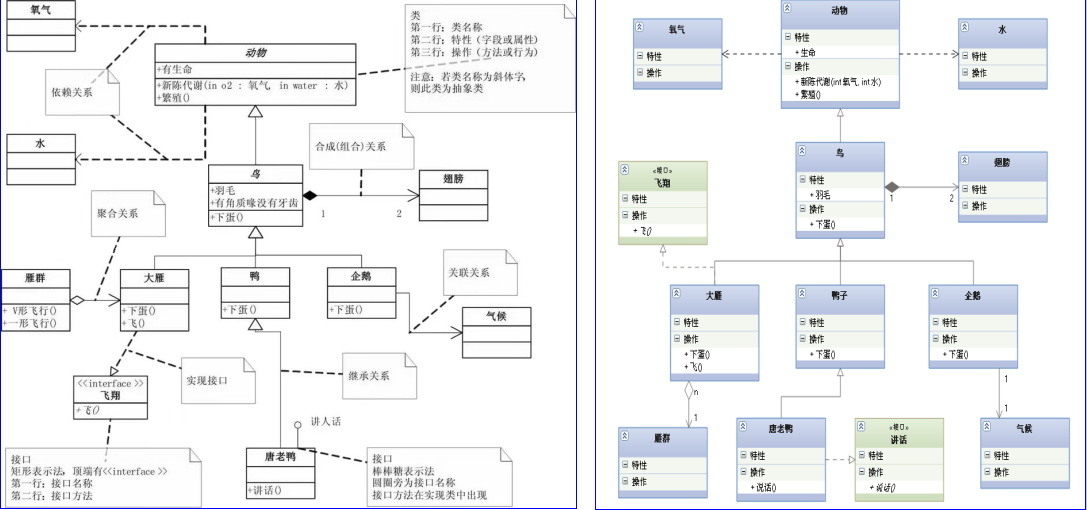
含有空心三角箭头。按连线虚线实线。得到实现泛化（泛化即继承）。

含有实心三角箭头(不含菱形和空心三角箭头)。按连线虚线实线。得到依赖关联。

### 3UML类图6大关系

以下参考：https://www.cnblogs.com/liuxin13070013/p/5452193.html

 各种关系的强弱顺序：        泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖



#### 依赖（广义）：

依赖关系，只要在那中用到了对方，那么这他们之间就存在依赖关系。如果没有了，对方，连编译都通过不了。用到的情形。具体分类如下。

依赖（广义）

1依赖（狭义）（局部变量（或只要是在方法体中的）、参数列表、返回值类型、静态方法调用）

2实现（依赖的特例）

3泛化（依赖的特例）

4关联（广义）（依赖的特例，位于成员变量）

关联（狭义）（一方需要另一方，仅此而已，无整体部分的逻辑关系，老师学生）

聚合（关联的特例，有整体部分的逻辑关系，学校与老师，部分可独立）

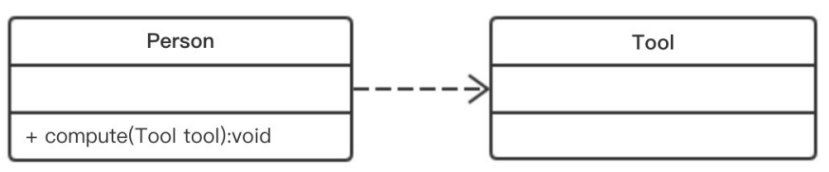
组合（关联的特例，有整体部分的逻辑关系，电视与屏幕，部分不可独立）

##### 1依赖（狭义）：

是一种使用的关系，即一个类的实现需要另一个类的协助，一般是单向的

【代码表现】：局部变量（或只要是在方法体中的）、参数列表、返回值类型、静态方法调用。因此，不属于关联关系，关联关系的标志是成员变量

【箭头及指向】：带箭头的虚线，指向被使用者



##### 2泛化：

是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。例如：老虎是动物的一种

【箭头指向】：带三角箭头的实线，箭头指向父类



##### 3实现：

是一种类与接口的关系，表示类是接口所有特征和行为的实现

【箭头指向】：带三角箭头的虚线，箭头指向接口



##### 关联（广义）：

此处的关联指的是狭广义的关联，包括：关联、聚合、组合

聚合、组合，属于关联的特殊情形，它们在类中的位置相同，都是成员位置，但可以从逻辑上区分三者：

聚合是逻辑上的整体与部分，且部分可以独立存在（学校-老师，雁群-大雁）；

组合也是逻辑上的整体与部分，但部分不能脱离整体（人-手）；

关联是逻辑上一个类需要另一个类，仅此而已，没有整体和部分这样特殊的逻辑关系（老师与学生，丈夫与妻子）。

###### 4关联（狭义）：

此处的关联指的是狭义的关联，即没有的整体与部分的逻辑关系，只有基本的需要与被需要关系，故与聚合和组合这两种特殊情形并列

是一种拥有的关系，它使一个类知道另一个类的属性和方法；如：老师与学生，丈夫与妻子关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。

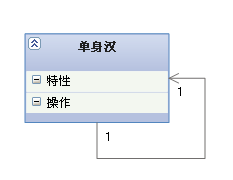
【代码体现】：成员变量

【箭头及指向】：带普通箭头的实心线，指向被拥有者

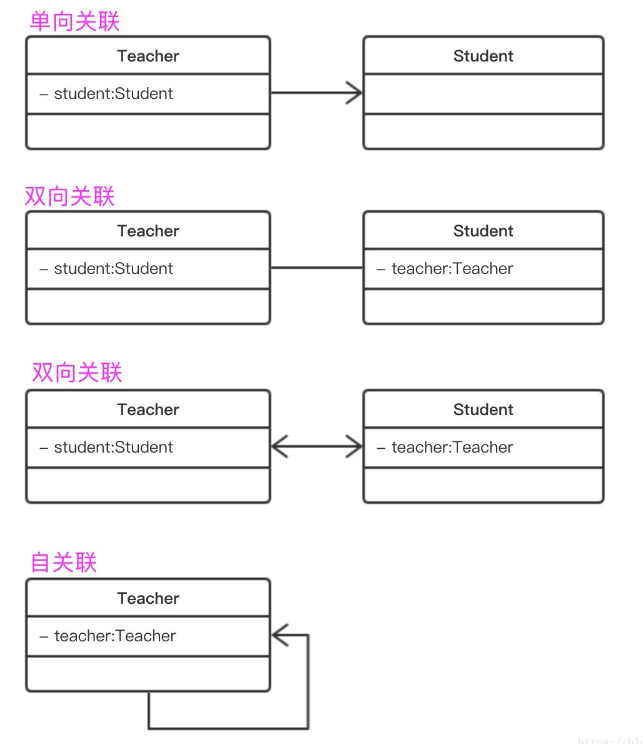


上图中，老师与学生是双向关联，老师有多名学生，学生也可能有多名老师。但学生与某课程间的关系为单向关联，一名学生可能要上多门课程，课程是个抽象的东西他不拥有学生。

        下图为自身关联：



关联关系汇总：



###### 5聚合：

是一种特殊的关联关系，是整体与部分的关系，且部分可以离开整体而单独存在。关联和聚合在语法上无法区分，必须考察具体的逻辑关系。如大雁和雁群、汽车和轮胎或发动机

【代码体现】：成员变量

【箭头及指向】：带空心菱形的实心线，菱形指向整体



###### 6组合：

也是一种特殊的关联关系，是整体与部分的关系，但部分不能离开整体而单独存在。关联和组合在语法上也无法区分，也必须考察具体的逻辑关系。如公司和部门是整体和部分的关系，没有公司就不存在部门，如人和手脚。组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

【代码体现】：成员变量

【箭头及指向】：带实心菱形的实线，菱形指向整体



联 > 依赖

## UML时序图

待整理

## 软件设计7大原则

记忆：喝醉接单已离开。含义如下。

合成复用原则：尽量先使用组合（contains a ）或者聚合等关联关系 来实现（has a），其次才考虑使用继承关系来实现（is a关系）

最少知道原则：一个对象应该对其他对象保持最少的了解，只跟朋友交流。什么是朋友呢？就是只出现在参数列表的类。而不是出现在方法体中的类。 如果在方法体中引入了跟参数列表不相关的类，则违背了最少知道原则。

接口隔离原则：不要大而全的接口。要为各个类建立它们需要的专用接口。

单一职责原则：不要大而全的类。控制类的粒度大小、将对象解耦、提高其内聚性。

依赖倒置原则：要面向接口编程，不要面向实现编程。。即接口多态和父类多态。

里氏替换原则：继承必须确保超类所拥有的性质在子类中仍然成立。同时子类可以扩展自己新的功能。包括父类多态。

开闭原则：对扩展开放，对修改关闭。编码关键：找到变化，封装变化。

注意，设计原则虽然是设计模式的基础。但在实际过程中，并不能完全实现设计原则。要考虑到时间成本，人力成本，资源成本。不能说某个设计原则就绝对是好的。不要追求完美。很多时候，完美并没有什么卵用。大家要去都是不能生搬硬套，一切以实事求是。

### 7合成复用原则

尽量首先使用关联/组合/聚合的方式，其次使用继承。

### 6最少知道原则

一个类应该对其他类保持最少的了解，只和朋友交流，不和陌生人说话。类与类关系越密切，耦合度越大。我们只需要拿到这个类对象，并且调用他的public修饰的方法即可。又叫迪米特法则。

那什么是朋友呢？

每个对象都会与其他对象有各种耦合关系。耦合的方式有很多，比如依赖关联聚合组合。其中我们称出现在成员变量，参数列表，方法返回值中的类朋友类。而出现在方法体中的类（如局部变量位置的类），称为陌生人。因此，陌生的类最好不要以局部变量的形式出现在类的内部。对局部位置耦合了不是直接的朋友，重构方法就是将违反迪米特法则的代码部分。放到这个类的内部。用public方法封装起来。对外提供。

### 5接口隔离原则

接口要小而专，方法尽量少。不要建立庞大臃肿的大而全接口。举例如下：

有一个接口a，里边有五个方法。

如果一个类b1只需要用到接口a里的12方法，另一个类b2，需要用到12和345方法。

则应该把该接口a拆分为接口a1和接口a2。分别包含12方法。和345方法。

每个类只需要与他们自己需要的那个接口建立联系。即类b2分别实现接口a1和接口a2。类b1则只需要实现接口a1。

**接口隔离原则与单一职责异同：**  
接口隔离原则约束的是接口依赖的隔离。针对的是抽象。  
单一职责原则约束的是类/接口/方法的职责是单一的，强调的是职责。针对的是程序中的实现和细节。

### 4单一职责原则

就一个类而言,应该仅有一个引起它变化的原因。一个类/接口/方法只负责一项职责

### 3.依赖倒置原则

抽象不应该依赖细节，细节应该依赖抽象。相对于细节的多变性，抽象的东西要稳定的多。在java中抽象指的是接口或抽象类，细节就是具体的实现类。这说的其实就是：针对接口编程，即多态，不要针对实现编程。

依赖关系传递的三种方式。具体就是：

类b中，

1接口a（包含方法do）作为方法形参

2有参构造器的参数列表是a

3set方法的参数列表是a

然后b中有个方法c，c调用a的实现类实现的具体的方法do：c(){a对象.do()}即可。

### 2里氏替换原则

父类对象被替换成子类对象后，程序行为不应当发生变化。具体说明**：**  
**1：**子类可以实现父类的抽象方法，但不应当覆盖父类的非抽象方法。  
**2：**子类中可以增加自己特有的方法。  
**3：**当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的入参更宽松（变宽松方向，原来要求传入鸟，现在不能限制传入飞鸟，但可以把入参条件扩大为动物。这样，原来父类方法中要求传入的鸟，与子类重写的方法中的动物相比，依然成立。即子类方法的参数列表为（Animal animal），传入的鸟参数一定是animal，但不能是（FlyBird flyBird），这就是上面说的：子类重写的方法，入参限制要变宽松）  
**4：**当子类的方法实现父类的方法时(重写/重载或实现抽象方法)，方法的后置条件（即方法的返回值）要比父类更严格或相等（变窄方向，如原来返回鸟，你不能返回动物，但可以返回大雁，即变窄了，因为大雁一定是鸟，所以一定符合条件）。即Bird bird = return 大雁。

### 1开闭原则

一个软件实体如类、模块和函数，应该对扩展开放（**针对服务提供方而言**），对修改关闭（**针对调用方而言**）。是程序设计中最基础，最重要的设计原则。其他六大原则都是为了更好地实现开闭原则。

具体来说就是：当软件需求变化时，尽量通过扩展新实体类来实现变化（这些实体类一般实现同一个接口或者抽象类，同时把具体功能实现，放在这些实体中，而不是放在服务调用方），而不是通过修改已有的代码（服务调用方）来实现变化。服务调用方一般不动。

比如，调用方有个方法excute(){a.do()}。do这个方法。只需要在不同的对象a中具体实现do即可。而且每个a都实现同一个接口或抽象类及其中的do方法。这样，**调用方**的excute**就很稳定（即**修改关闭**）**，只需要增加新的对象a并实现新的do行为即可（即对扩展开放）

## 软件设计模式：GoF23

设计模式最重要的作用是解耦。

目的。1写好代码。2更好地重构

// 待做：这个要追加到下面的记忆模板中-----------------------------------------------------------------------------------

应用场景：如在什么情况下使用

应用举例：具体应用例子，如应用在数据库连接对象，如线程池

uml类图

结构角色

扩展维度：哪几个角色属于可扩展的维度，实现解耦的

运行流程

使用场景：一对多依赖关系下，一个对象状态改变，引起与之相关的n个对象状态的改变。

应用案例或源码案例

入门案例

优点特点

缺点

备注

对比

记忆：

单工抽建原

适享组装代桥外

备访观策中，模迭状命职解

图场入优缺应对备

怎样使用设计模式？分为三类共23种。

**创建型模式∶**

单例模式、工厂模式、抽象工厂模式、建造者模式、原型模式。

**结构型模式∶**

适配器模式，桥接模式，装饰模式，组合模式，外观模式，享元模式，代理模式

**行为型模式∶**

模板方法模式，命令模式，迭代器模式，观察者模式，中介者模式，备忘录模式，解释器模式，状态模式，策略模式，职责链模式，访问者模式。

使用设计模式之前的注意点：

入门案例的代码里，在采用了抽象类和接口的地方，实际抽象类和接口二者都可以。但是由于类无法多继承，以及无法在接口中持有变量这两个局限，故有一种很实用的方案是：一般可以先采用接口，然后创建一个抽象类去空实现这个接口里的方法，同时抽象类里可以持有变量。

设计模式知识，已经入门，故下一步，需要在源码中加深对原理流程的理解

程序员的抽象能力特别重要。而学习设计模式就是使抽象能力进行升级。

你在开发过程中遇到过了并解决解决了哪些问题

你的开发过程中使用过或遇到过哪些是几模式？

什么是设计模式？

设计模式（Design Pattern）是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。

1995年，GoF（Gang of Four，四人组/四人帮）合作出版了《设计模式∶可复用面向对象软件的基础》一书，共收录了23 种设计模式，从此树立了软件设计模式领域的里程碑，人称GoF设计模式。

学习设计模式的意义？

设计模式的本质是面向对象设计原则的实际运用，是对类的封装性、继承性和多态性以及类的关联关系和组合关系的充分理解。

正确使用设计模式具有以下优点∶

可以提高程序员的思维能力、编程能力和设计能力。

使程序设计更加标准化、代码编制更加工程化，使软件开发效率大大提高，从而缩短软件的开发周期。

使设计的代码可重用性高、可读性强、可靠性高、灵活性好、可维护性强。

熟悉每个设计模式的特点。熟悉每个设计模式可能有多种实现方式，各有优劣

各个设计模式并非完全独立，而是经常一起联合作战。比如组合模式几乎被所有其他模式用到。又比如工厂模式和备忘录模式中，想要创建对象，可以叠加克隆模式。

每种设计模式的典型结构，在实际使用中会根据具体需要而适度变化或进行简省，产生各种分身。总之，只要熟悉每种模式的核心UML类图关系，就能以不变应万变。

### 单例模式-以下创建型-单工（抽）建原

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

记忆：单例饿懒二五分

1什么是单例模式？

采取一定的方法使得整个软件系统中某个类只存在一个实例。故单例模式中，构造器必须私有，以保证一个类只有一个实例

2为什么要使用单例模式？

单例模式的使用场景：

在需要频繁创建和销毁对象 或者耗时较长，耗费资源较大的重量级对象。的场景下，推荐使用单例模式。可以提高性能。使用获取实例的方法即可，而不是用new的方式。

3怎么样实现单例？

两大类。饿汉式和懒汉式。

3.1饿汉式单例有两种方式。静态变量初始化。。静态代码块初始化。

饿汉式单例的优点是，在类加载的时候就初始化实例。均不会发生线程安全问题。

饿汉式单例的缺点是，由于有多种方式导致类被加载。导致初始化实例。但该实例未必被使用。无法实现懒加载。

3.2懒汉式单立

3.2.1原生方式，线程不安全，不可用

3.2.2同步方法，实现线程安全。缺点是每次都要同步获取实例的方法，效率低，可以用，但不推荐

3.2.3同步代码块，解决上面的低效率问题，但是仍然导致了线程不安全，原因跟懒汉式单立的原生方式一样，不可用。有人这么写要能识别出来。

3.2.3双重检查。既可以实现懒加载。也可以避免上面同步方法的低效率，又可以线程安全，推荐。。另外还要在属性上加valitile关键字，防止指令重排（待探究）

3.2.4静态内部类方式。静态内部类的特点是外部类加载的时候不会加载静态内部类。因此可以利用这点实现懒加载的效果，同时在获取实例时，JVM加载类的特点是单线程，包括加载静态内部类，所以也不会有现成安全问题。推荐

3.2.5枚举。自带单例

使用枚举：防止反射破坏单例（idea的编译工具和jdk的javap -s 反编译后，结果欺骗了我们：给出了一个无参构造。使用jad.exe工具反编译后，发现只有一个有参构造，故反射时使用该有参构造，并传值进去就，就可以防止反射破坏单例）



4应用：

JDK中的runtime。使用的就是饿汉式单例。由于runtime总会用到的，所以不会造成内存浪费。

5饿汉式单例的缺点是，由于各个能有多种方式导致类被加载。导致初始化实例。但该实例未必被使用。无法实现懒加载。

懒汉单例缺点：反射会破坏单例

Spring中的类默认也是单例。哪种？

### 工厂模式

记忆：工厂两类分三种

简单工厂模式、工厂方法模式、抽象工厂模式

|  |  |
| --- | --- |
|  |  |
|  |  |
| 场景 | 实例化对象不使用new，而用工厂方法代替，将new动作封装到工厂方法里。  当用到大量的创建某种、某类对象时，常用到工厂模式 |
| 入门案例 |  |
| 应用 |  |
| 优点 |  |
| 缺点 |  |
| 对比 |  |
| 备注 |  |
| UML类图 |  |
|  |  |
|  |  |

3.1简单工厂模式（也叫静态工厂模式）

3.1.1是什么？

只有一个车工厂类，类中包含一个或多个接收参数并创建实例的静态方法。

消费者不再自己new对象，改为调用工厂类里的静态方法来获取实例。静态方法根据接收的不同参数去实例化不同的对象后返回给调用者。

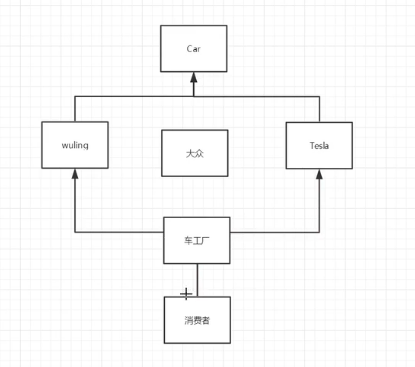
3.1.2为什么？

优点：虽然不符合开闭原则，但是复杂度低，故在源码中大量使用存在。易开发，易维护

3.1.3怎么样？

所有的车使用同一个车工厂（下面的工厂方法：不同的车使用自己专门的车工厂）。消费者调用工厂类里的静态方法时，传入不同参数，来获取不同实例。静态方法根据接收的不同参数去实例化不同的对象后返回给调用者。

类图：



3.1.4应用：ddd

3.1.5缺点：某种程度上不符合开闭原则，即扩展性不好，增加新的类时，需要修改原来的工厂类的代码（虽然有这个缺点，但依然不妨碍静态工厂模式受到热烈欢迎）

方案：见下面：工厂方法模式

3.2工厂方法模式：

3.2.1是什么？

依然由车工厂类去创建对象，只不过存在多个车工厂类（每个品牌的车，都有自己的车工厂类），每个车工厂类实现同一个车工厂类接口及其方法

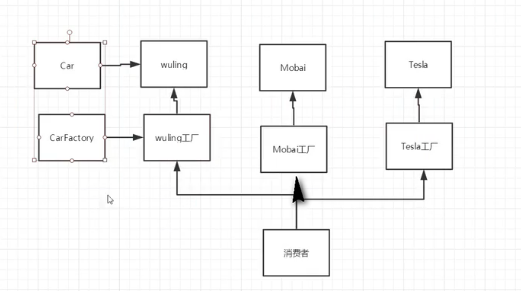
3.2.2为什么？

优点：符合开闭原则，即不用修改原有代码，扩展性好

3.2.3怎么样？

消费者想要得到某个品牌车A时，调用A的车工厂B，由B实例化一个对象返回。存在多个车工厂，每个车工厂都实现同一个工厂接口及其接口方法。而上面的静态工厂模式，只有一个车工厂。

类图：



3.2.4应用：ddd

3.2.5缺点：虽然扩展性好，但这也导致了新增文件越来越多（虽然符合设计原则，但是不符合业务实际需求，也是不可取的）

3.3抽象工厂模式

3.3.1是什么？

3.3.2为什么？

产品等级：一类产品就是一个产品等级，如小米手机、华为手机、苹果手机...都是手机，属于手机这个产品等级

产品族：小米手机、小米音箱、小米电脑...都是小米旗下产品

使用场景：

静态工厂模式：**适用于产品等级维度场景下**的类实例化。特点：扩展时需要修改原有工厂类代码，虽然不符合开闭原则，但扩展比较灵活，用得最多

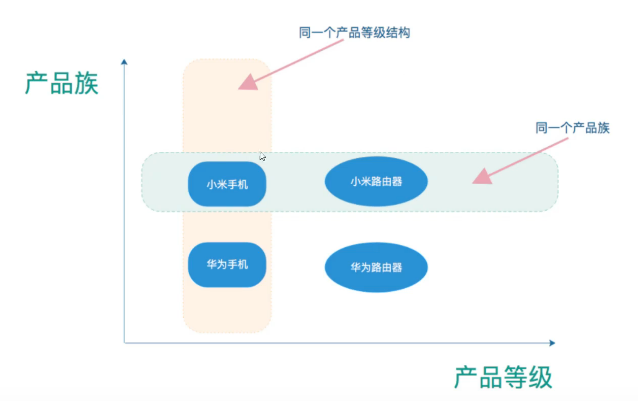
工厂方法模式：**适用于产品等级维度场景下**的类实例化。特点：虽然符合开闭原则，即扩展时无需修改原有工厂类代码，创建新的工厂类即可，但新的文件也越来越多

抽象工厂模式：**适用于只增加产品族维度场景下**的类实例化。特点：不可新增产品等级，即不可新增产品类（如不可新增一个新产品，如风扇。这种模式下，产品等级即产品种类已经固定。只能增加产品族，即增加新的品牌商（的工厂类）。这样的话，只需要先分别实现各个产品C们的接口如手机电脑路由器等，然后增加一个工厂类B实现统一的工厂接口，然后B生产C们即可）。比如新增一个苹果工厂A，A提供苹果手机、苹果电脑、苹果手表等类的实例化静态方法

3.3.3怎么样？

抽象工厂模式，实际是在工厂方法模式基础上进行了扩展：将工厂方法模式下的单个产品工厂接口扩展成了综合产品工厂接口A，同时在A中增加了新产品的获取方法，如在原有车的基础上新增了电脑，手机的获取方法。。。这样，一个工厂就能创建更多的产品（即一个产品族）

名词解释：

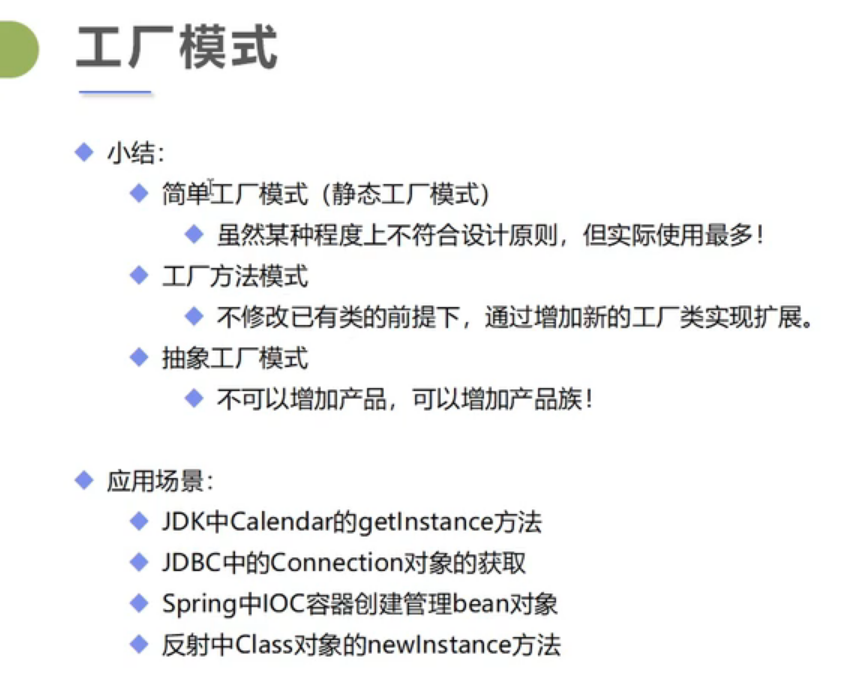


3.3.4应用：ddd

3.3.5缺点

不可新增产品等级，即不可新增产品类。只适合增加新的产品族，比如新增一个苹果工厂A，A提供苹果手机、苹果电脑、苹果手表等类的实例化静态方法

三种工厂小结：



### 抽象工厂模式

见上面工厂模式下的抽象工厂模式

### 建造者模式

记忆：建造一理三实现

1是什么？

抽象工厂模式VS建造者模式：

与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。

抽象工厂模式实现对产品家族的创建，一个产品家族是这样的一系列产品∶ 具有不同分类维度的产品组合，采用抽象工厂模式不需要关心构建过程，只关心什么产品由什么工厂生产即可。**而建造者模式则是要求按照指定的蓝图（即抽象Builder）建造产品，它的主要目的是通过组装零配件而产生一个新产品**

2为什么？

使用场景：

建造者模式所创建的产品一般具有较多的共同点，其组成部分相似。

即一些基本部件不会变，而其组合经常变化的时候。

优点：

1）客户端（使用程序）不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦

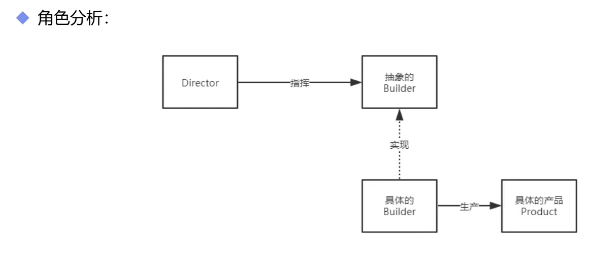
2）每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者， 用户使用不同的具体建造者即可得到不同的产品对象。增加新的具体建造者无须修改原有类库的代码。故相同的创建过程可以创建不同的产品对象

3）可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程

3怎么样？

实现方式1（此方式是其他两种方式的基础，必须会）：

典型的建造者模式，包含角色：消费者、Director(指挥者)、抽象Builder、具体Builder1、具体Builder2、产品类



消费者调用指挥者D，D调用具体建造者A（A继承了抽象Builder），A完成产品的实例化、组装、返回。要想获取不同的产品，只需传入新的具体建造者对象即可。

实现方式2：

有时为了更加简便，指挥者可以直接由消费者代替，这样组装实例更加灵活

实现方式3：静态内部类：待整理

https://blog.csdn.net/Apple\_wolf/article/details/113341242

4应用：

1、去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。 2、JAVA 中的 StringBuilder。

5缺点：

产品必须有共同点，范围有限制。如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制

### 原型模式

用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

当对象比较复杂时，可以使用原型模式，当对象比较简单时，就没必要使用该模式了

有几种方式可以实现原型模式：序列化反序列化（涉及到io，会较低效率）、实现克隆接口（简单常用），这里介绍克隆方式，包括浅克隆（克隆时不克隆对象的对象属性）、深克隆（连同对象的对象属性一起克隆）

### 适配器模式-以下结构型-适桥组装外享代

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。

UML类图关系：

适配器类，通过继承/实现/继承目标抽象类/接口/具体类Target，并组合/继承一个Adaptee对象，使二者产生联系

使用场景：想要使两个不相干的类在一起工作。

入门案例：\*\*\*

优点：\*\*\*

缺点：\*\*\*

应用：\*\*\*

对比：\*\*\*

备注：

适配器模式分为两类三种

第一类包含两种：类适配器模式、对象适配器模式。它们具有如下相同的角色和运行流程

角色和结构：用户、适配器类Adapter、被适配类Adaptee、被适配器类继承/实现/继承的目标抽象类/接口/具体类Target

运行流程：1用户实例化适配器类对象，并调用其方法A（继承/实现/重写自Target），2A中必须调用被适配类的方法B，并进行二次加工后返回

区别：运行流程中第二步，如何才能调用到原方法？有组合和继承两种方式。组合——对象适配器，继承——类适配器。这就是区别。

第二类包含一种：缺省适配器模式，是适配器模式的一种变体，应用广泛

详解：当不需要实现一个接口所有方法时，可先设计一个(抽象)类实现该接口，类中为每个方法提供默认实现（空方法），则子类可选择性地覆盖父类某些方法来实现需求

角色和结构：接口C、空实现类A，A的子类B（B只需重写A的部分方法即可，B可以是匿名内部类或有名，B中可以组合其他对象）

运行流程：用户创建子类B对象（匿名或有名均可），并调用其方法

由于继承——类适配器具有单继承缺陷，TODO 故推荐使用对象适配器模式。另外缺省适配器模式也很常用

### 桥接模式

UML类图关系：客户端、桥接类(可抽象)、桥接类子类、行为接口、行为接口实现类

使用场景：桥接模式适用于一个系统具有两个独立变化维度，且一个维度需要调用另外一个维度的方法这样的场景

入门案例：电脑两个维度：类型和品牌。。另外，虚拟机使用的是桥接模式，不同的Java虚拟机组合了不同的操作系统。

优点：两个维度可以各自独立进行扩展，有利于系统的分层设计。代替多层继承，可减少子类个数

缺点：桥接模式适用于一个系统具有两个独立变化维度，适用范围有一定限制

应用：\*\*\*

对比：

桥接模式和适配器模式的异同：

相似：代码结构上有些相似性，都是在两个类之间建立联系，在一个类方法中调用另外一个类方法

不同：

适配器模式下被连接的两个类并不属于某个产品的两个维度，因而不会各自扩展。

而桥接模式下，两个类属于同一个产品的两个维度，存在朝各自维度进行扩展的需要。

备注：\*\*\*

### 组合模式

UML类图关系：\*\*\*

使用场景：一个类把另一个类时或其集合作为自己的属性时

入门案例：\*\*\*

优点：避免了类的单继承缺点。应用广发，其他设计模式几乎离不开组合模式

缺点：\*\*\*

应用：\*\*\*

对比：\*\*\*

备注：

### 装饰者模式

// 待整理：不仅是这里的装饰者，还有其他模式：一定要在对比中发现各自鲜明的特点。比如下面的与桥接模式的对比，就很有效果。很容易展示二者不同。容易理解。

UML类图关系：接口c。c的实现类a及a的子类f（相当于单品咖啡）。c的新实现类b(即装饰者类)及b子类d（相当于调料）。流程：将代表单品咖啡（或加了调料的单品咖啡）的c组合到代表不同调料的b中。

使用场景：\*\*\*

入门案例：单品咖啡➕调料

优点：\*\*\*

缺点：\*\*\*

应用：

Io流里边的字节缓冲流和字符转换流。

对比：

装饰者模式跟桥接模式的比较：

都有两个维度。

但不同在于扩展的特点上：

桥接模式：台式类型的华为品牌电脑。

装饰者模式：大碗类型拉面加配菜三块锅巴，两个鸡蛋。单品咖啡加各种调料，同样调料加三分。

看出不同了没有？虽然都是两个维度。但桥接模式同一个维度不可以叠加。同一款电脑，不能既是华为又是小米。。而装饰者模式却可以套娃，一个大碗拉面可以同时配n块锅巴，n个鸡蛋。

桥接模式是类型持有品牌。

而装饰者模式是类型作为参数传入装饰者类，生成的新对象又可以作为参数传入装饰者类，如此套娃。

还有一点不同：虽然他们都互相持有对方，但装饰者模式下，两个类实现共同的接口。而桥接模式下的两个接口/超类是平级的。

备注：

A和b实现相同的接口c。

装饰者类的某些属性如价格计算包含两部分。调料的价格+传进来的单品或加过调料的单品的价格

### 外观模式

UML类图关系：客户端、外观类(是客户端与子系统交互的代理人)、子系统abc

使用场景：当系统需要进行分层设计，或调用子系统比较麻烦的时候

入门案例：开公司需要与工商局、银行、质检局等机构交互

优点：通过外观类这个代理类，为客户端提供调用子系统的统一接口，从而实现了客户端与子系统之间的解耦。典型体现了迪米特原则(最少知道原则)

缺点：\*\*\*

应用：DBUtils封装了jdbc相关操作

对比：\*\*\*

备注：\*\*\*

### 享元模式

UML类图关系：客户端、享元工厂、享元接口/抽象类、内部状态类（即享元类，存储内部状态，如棋子颜色）、外部状态类（存储外部状态，如棋子坐标）

使用场景：系统状态分为两部分，且无需创建多个实例的

入门案例：1不同位置上（外部状态）的不同颜色（内部状态）的棋子。2不同用户（外部状态）访问不同类型（内部状态）的网站

优点：实现相同对象的复用，极大节省内存

缺点：为了读取分离出的外部状态，运行时间变长，以时间换了空间

应用：String常量池、Integer源码、各种池技术，如数据库连接池、线程池

对比：单例模式

备注：

掌握享元模式的关键，是能够抽象出事物的内外两种状态

内部状态：可以共享，不随环境变化，即系统中享元类的属性，如棋子颜色等属性，所有内部状态单独成类

外部状态：不可以共享，不随环境变化，即系统中非享元类的属性，如棋子的坐标，所有外部状态单独成类

创建享元类对象后，不可再次进行修改

外部状态是善变的，故必须独立于内部状态，单独设置

### 代理模式

UML类图关系：\*\*\*

使用场景：\*\*\*

入门案例：\*\*\*

优点：\*\*\*

缺点：\*\*\*

应用：\*\*\*

对比：\*\*\*

备注：分为静态代理、JDK代理、CGlib代理

### 观察者模式-以下行为型-观模迭状命备中职访解

UML类图关系：

抽象主题：组合了一个观察者的list,并提供add、remove方法，还必须提供一个抽象的通知方法。类似一个抽象的公众号接口

具体主题ab：抽象主题的实现类，必须实现具体的通知方法。类似众多具体的公众号

抽象观察者：即用户或订阅者，用来接收主题类推送的信息

具体观察者abc：抽象观察者的实现类，提供update方法

使用场景：一对多依赖关系下，一个对象状态改变，引起与之相关的n个对象状态的改变

入门案例：公众号消息的发布-订阅

优点：\*\*\*

缺点：\*\*\*

应用：JDK的Observable类（相当于Subject的实现类或抽象类的子类），Observer相当于案例中的Observer。使用时，直接继承Observable即可

对比：\*\*\*

备注：又叫【发布-订阅模式】

### 策略模式

UML类图关系：

客户端

环境类及其子类：如Bird和ABird、BBird。。名称可以不叫context

策略接口/抽象类：可存在多个，如bird有FlyStrategy策略接口（fly();等待具体实现）、SpeakStrategy策略接口（speak();等待具体实现）

具体策略类：

如上面的FlyStrategy策略接口的2个具体策略实现类：FlyStrategy1（fly()：优雅地飞）、FlyStrategy2（fly()：不会飞）

如上面的FlyStrategy策略接口的2个具体策略实现类：FlyStrategy1（speak()：嘤嘤嘤）、FlyStrategy2（speak()：嘎嘎嘎）

使用场景：取代多个if-else，提高扩展性

入门案例：鸟类的不同子类在fly()和speak()上拥有不同表现

优点：\*\*\*

缺点：\*\*\*

应用：\*\*\*

对比：1与简单工厂模式结构相似，但侧重点不同 2与装饰者模式对比

备注：

1环境类这个角色并非总是会存在,名字也时常不叫context,若无,则其实是抽象策略接口承担了环境类的角色功能

2环境类可以有一个或多个子类(完整版的策略模式见下面 strategy2)

3环境类可以持有一个或多个策略类接口对象(完整版的策略模式见下面 strategy2)

### 模板(方法)模式

UML类图关系：

客户端、

抽象模板：包含四类方法：模板方法(规定执行顺序)、抽象方法(交由子类视情具体实现)、具体通用方法、钩子方法(默认不做任何事情，但用于补偿或给流程传入条件)

具体模板

使用场景：行为是按照固定的顺序执行的时候

入门案例：银行办理业务取号排队流程、炒菜流程

优点：\*\*\*

缺点：\*\*\*

应用：\*\*\*

对比：\*\*\*

备注：

上述四类方法，每类方法都可以有不止一个

### 迭代器模式

UML类图关系：

迭代器接口。定义了next。hasNext。Remove等方法（容器和迭代器都可以有remove方法）

迭代器实现类

容器接口。定义了add、remove、获取迭代器方法iterater。

容器实现类a：iterater()需要一个容器（数组或list）作为参数给迭代器的构造方法（数据容器就是这样传递给迭代器的）。因此需要在a中组合一个容器对象

使用场景：

迭代器模式是与集合共生共死的，一般来说，我们只要实现一个集合，就需要同时提供这个集合的迭代器，就像java中的Collection，List、Set、Map等，这些集合都有自己的迭代器。假如我们要自定义一个这样的新的容器a，为了遍历a里的数据，就需要给我们的容器a准备一个迭代器。这就会用到迭代器模式。

入门案例：学院和部门

优点：1隐藏内部遍历细节 2 \*\*\*

缺点：\*\*\*

应用：\*\*\*

对比：\*\*\*

备注：

由于集合和迭代是如此重要和如此紧密。以至于所有的语言都天然实现了迭代器模式。我们只需用它们提供的api即可。但为了弄清楚它的原理。下面我们演示自己实现一下迭代器模式。非常简单。注意下jdk实现和自己实现这两种方式，有一个很小的不同。就是jdk中的容器接口角色(即collection接口)的实现类(即arraylist.hashset等)使用的底层容器是数组(其实集合容器建立在数组容器为基础上。即集合的底层就是数组)。而我们自定义演示中的容器接口的实现类使用的底层容器既可以是已经被jdk实现的现成的list等集合，也可以像jdk那样也使用数组。这里我们为了让大家看清楚，故写了两个容器接口的实现类。一个底层用数组，另一个底层用jdk集合。

核心理解。数据存放在容器。那怎么把数据从容器中传送到迭代器里面去遍历呢？需要在容器里组合一个迭代器。

由于容器与迭代器的关系太密切了，所以大多数语言在实现容器的时候都给提供了迭代器，并且这些语言提供的容器和迭代器在绝大多数情况下就可以满足我们的需要，所以现在需要我们自己去实践迭代器模式的场景还是比较少见的，我们只需要使用每种语言中已有的容器和迭代器就可以了。

### 状态模式

UML类图关系：

1客户端。实例化环境类对象d，调用其处理方法并传入参数c(比如分数)，以触发状态切换(比如分数决定等级优良中差)。

2环境类(组合了状态接口/抽象类)。环境类的处理方法b中需要接收客户端传来的参数c，并连同环境类对象d，一起传递给具体状态实现类的同名方法。

3状态接口/抽象类a。a定义了抽象方法e。e接收c和d。

4a的实现类bc...。重写/实现抽象方法e，进行本状态下的具体业务处理。

使用场景：

应用程序的部分对象可能会根据不同的情况做出不同的行为。我们把这种对象称为有状态的对象。而把影响对象行为的一个或多过动态变化的属性称为状态。当有状态的对象与外部事件产生互动时，其内部状态就会发生改变，从而使其行为也发生改变。

状态模式解题思路，适当控制一个对象状态转换的条件表达式过于复杂时，把相关判断逻辑提取出来，用各个不同的类进行表示系统处于那种情况，直接使用相映的状态类对象进行处理。目的是消除if else switch case等冗余语句，代码更有层次性，具备良好的扩展能力。

入门案例：传入分数，切换决定优良中差四种状态

优点：1取代了if-else、switch-case 2符合开闭原则

缺点：状态类可能会比较多

应用：线程状态

对比：\*\*\*

备注：

状态模式跟其他模式一样。在实际使用中千变万化。会有各种各样的变种。但原理不变。比如状态模式核心思想是根据不同状态来切换行为，其实这里的状态可以是任意属性，如投票次数或分数。根据该属性具体属性值的不同，同一功能会走不同的具体实现方法。

处理业务逻辑时，handle方法可能需要传入环境类对象a和客户端传递的参数b，其中a可用于获取和设置环境类里的属性或方法(这里暂略)。

### 命令模式

UML类图关系：客户端、命令调用者、抽象命令类/接口、具体命令类A和B、命令接收者

使用场景：\*\*\*

入门案例：顾客下单，服务员（调用者）送单（命令类）给厨师（接收者），厨师做饭。

优点：通过在命令调用者和接受者之间增加命令类，实现命令的调用者和接收者的解耦。如服务员把单子放下即可，无需等着厨师，从而解耦。

缺点：\*\*\*

应用：\*\*\*

对比：\*\*\*

备注：最终实际干活的是战士，即接收者

### 备忘录模式

UML类图关系：备忘录模式的四大角色（对应四个类），注意其中管理者有三种状态保存场景（见CareTaker）

使用场景：提供了可恢复状态的机制：存档、撤销操作、后退、后悔药等

入门案例：游戏角色状态恢复

优点：\*\*\*

缺点：如果类的属性过多，势必占用较大资源，且每次保存都会消耗一定内存

应用：\*\*\*

对比：\*\*\*

备注：

所谓状态即原对象的各个属性值

备忘录角色里的属性一般与发起者属性相同

备忘录模式可以叠加原型模式，以节省内存

备忘录模式的简省用法（对某些角色类进行省略或简化）。如常用的：新建一个现有对象a的同类的对象b，并将a的状态（即属性值）set赋值给b

### 中介者模式

UML类图关系：

客户端

抽象中介者a

具体中介者b

抽象同事

具体同事c

具体同事d

使用场景：\*\*\*

入门案例：婚介公司

优点：通过将网状结构改造成星型结构，实现调用者(一个colleague)和被调用者(其他符合条件的colleague)解耦

缺点：\*\*\*

应用：springmvc中c是m和v的中介者

对比：\*\*\*

备注：要实现怎样的通知效果，只需修改中介者类即可。关键也在于维护这个中介者类

### 职责链模式

UML类图关系：

客户端

请求类

抽象处理者

具体处理者abcd。。。

使用场景：\*\*\*

入门案例：请假审批

优点：\*\*\*

缺点：\*\*\*

应用：\*\*\*

对比：\*\*\*

备注：

1必须将第一个处理者set到最后一个处理者里，形成闭环，防止不从第一个开始处理请求这种情形

2实际中，一般用一个类组合一个list来存放所有的处理者，这样似乎关系到了另一种模式，各个模式不是绝对独立的

### 访问者模式

<http://c.biancheng.net/view/1397.html>

<https://www.runoob.com/design-pattern/visitor-pattern.html>

大概了解一下使用场景：

在现实生活中，有些集合对象存在多种不同的元素，且每种元素也存在多种不同的访问者和处理方式。例如，

公园中存在多个景点，也存在多个游客，不同的游客对同一个景点的评价可能不同；

医院医生开的处方单中包含多种药元素，査看它的划价员和药房工作人员对它的处理方式也不同，划价员根据处方单上面的药品名和数量进行划价，药房工作人员根据处方单的内容进行抓药。

电影或电视剧中的人物角色，不同的观众对他们的评价操作也不同；

还有顾客在商场购物时放在“购物车”中的商品，顾客主要关心所选商品的性价比，而收银员关心的是商品的价格和数量。  
  
这些被处理的数据元素相对稳定（一个公园分为确定的n个部分，一张处方上药品种类不变、电影角色不会随意变动），而访问方式多种多样（即每个访问者对某个元素的操作多种多样）的[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)，用“访问者模式”来处理比较方便。访问者模式能把处理方法从数据结构（容纳多个元素的容器）中分离出来，并可以根据需要增加新的处理方法，且不用修改原来的程序代码与数据结构，这提高了程序的扩展性和灵活性。

/\*\*

\*

\* 模式角色有5个，ClassTest不算：

\* ClassTest：1-init objectStructure(add element) 2-create visitor and objectStructure.receive(visitor)：receive 与 Element 的accept一个意思，便于区分，receive是总接收入口

\*

\* ObjectStructure：1-add/remove element to list 2-receive(visitor){iterate element list and element.accept(visitor)}

\* Element(被访问者接口)：accept(visitor){visitor.visit(this)}：(维度一，可扩展新类，如ElementB)

\* ElementA(具体元素)

\* Visitor(访问者接口)：visitor(element){element.do}：(维度二，可扩展新类，如VisitorB)

\* VisitorA(具体访问者)

\*

\* 核心：是要将element传给visitor的visit(element)方法,从而实现访问者visitor对元素element的访问其他操作

\*

\* 流程表述方式1：

\* ObjectStructure 每次接收一个 visitor，并将该 visitor 传给自己内部维护的每个 element，

\* element 再将自己反传给该visitor，最终由该visitor操作element

\*

\* 流程表述方式2：

\* 先在测试类ClassTest里初始化一个 ObjectStructure ，接着ObjectStructure 接收（receive）一个 visitor，并在内部遍历自己作为容器所存放的所有 element 时，将visitor通过element.accept(visitor)传给每个 element。

\* 然后每个element调用入参visitor的visit方法——即visitor.visit(this)——将自己(this)反传给该visitor，最后由该visitor对element进行操作（包括调用element自己的方法：element.operation()）

\*

\* visitor.visit(this); // todo 这里用到了双分派，即参数变成了动作执行者，动作执行者变成了参数

\*/

### 解释器模式

解释器模式不常用，故不作深入整理

代码已整理，但只是一个案例就，还需要结合其他案例，达到整体的理解

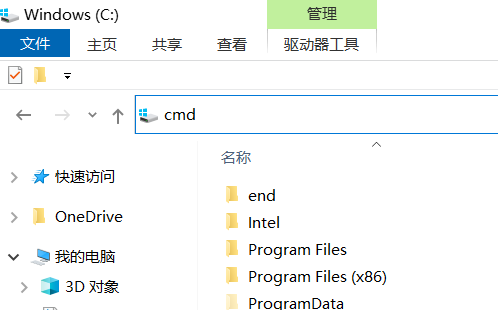
# 其他

问题：

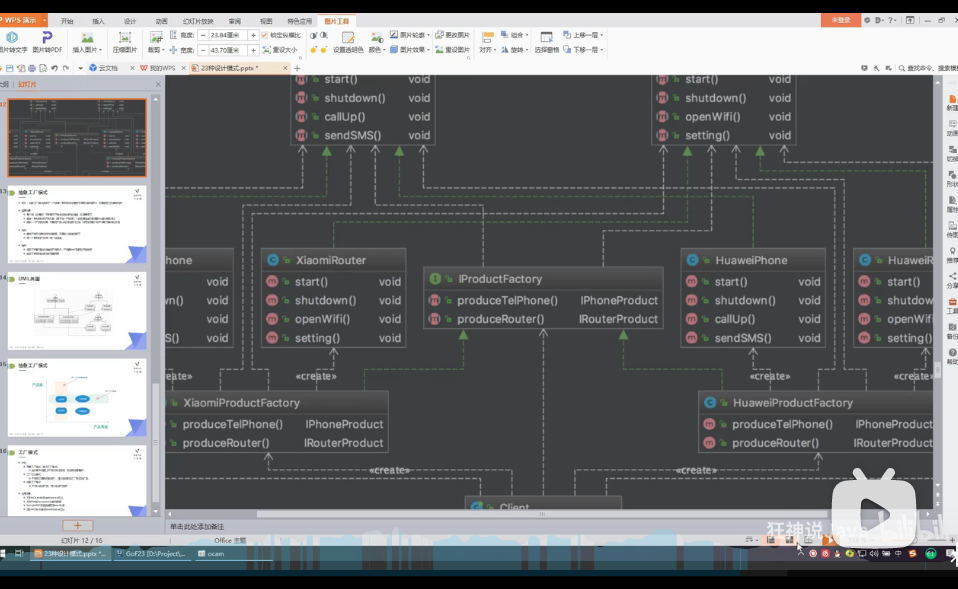
*指令重排*

浏览器快捷键：A+空格：打开浏览器设置

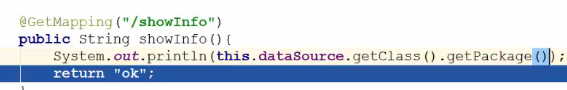
直接在导航栏打开cmd



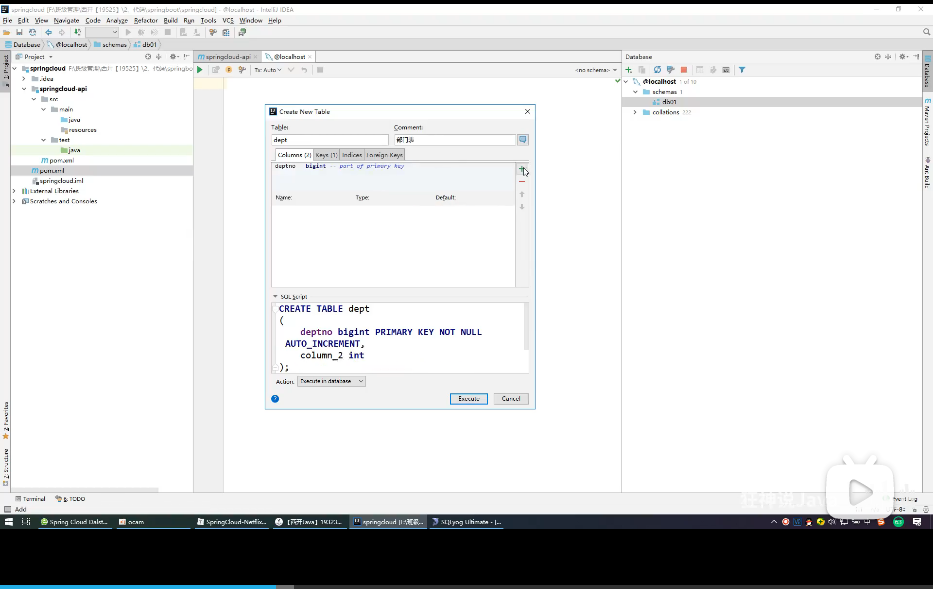
生成类图



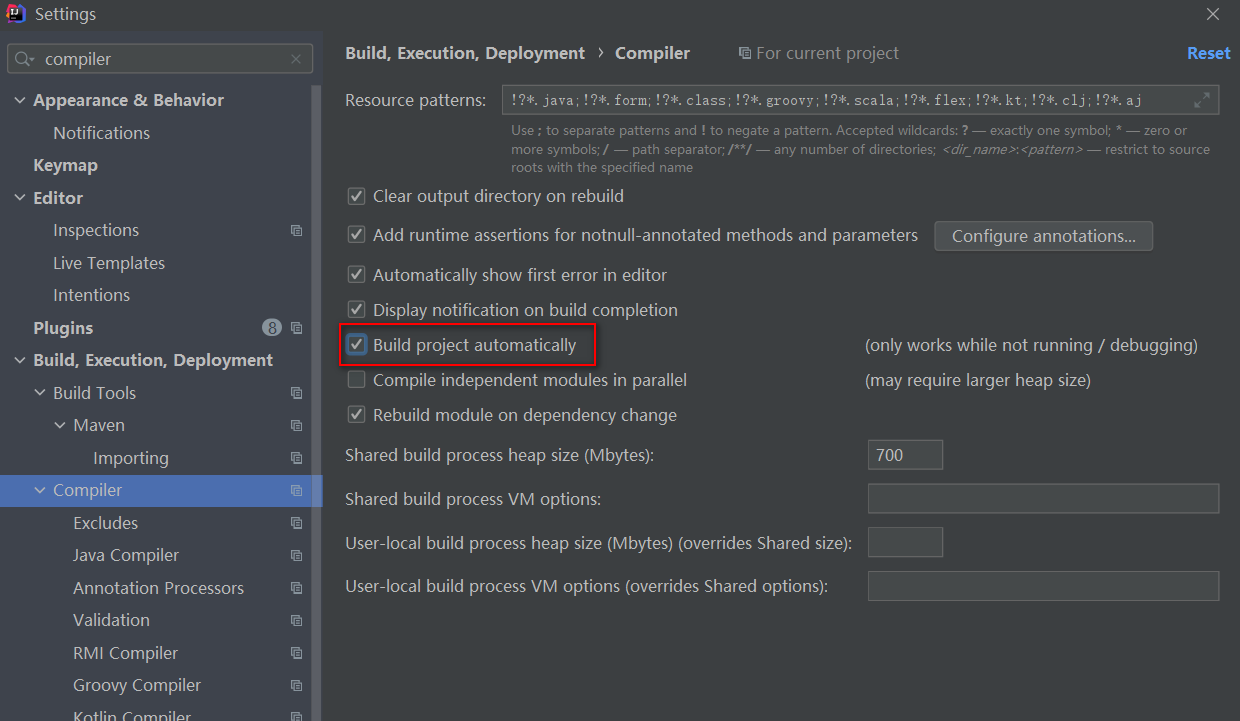
反射可以获取包名



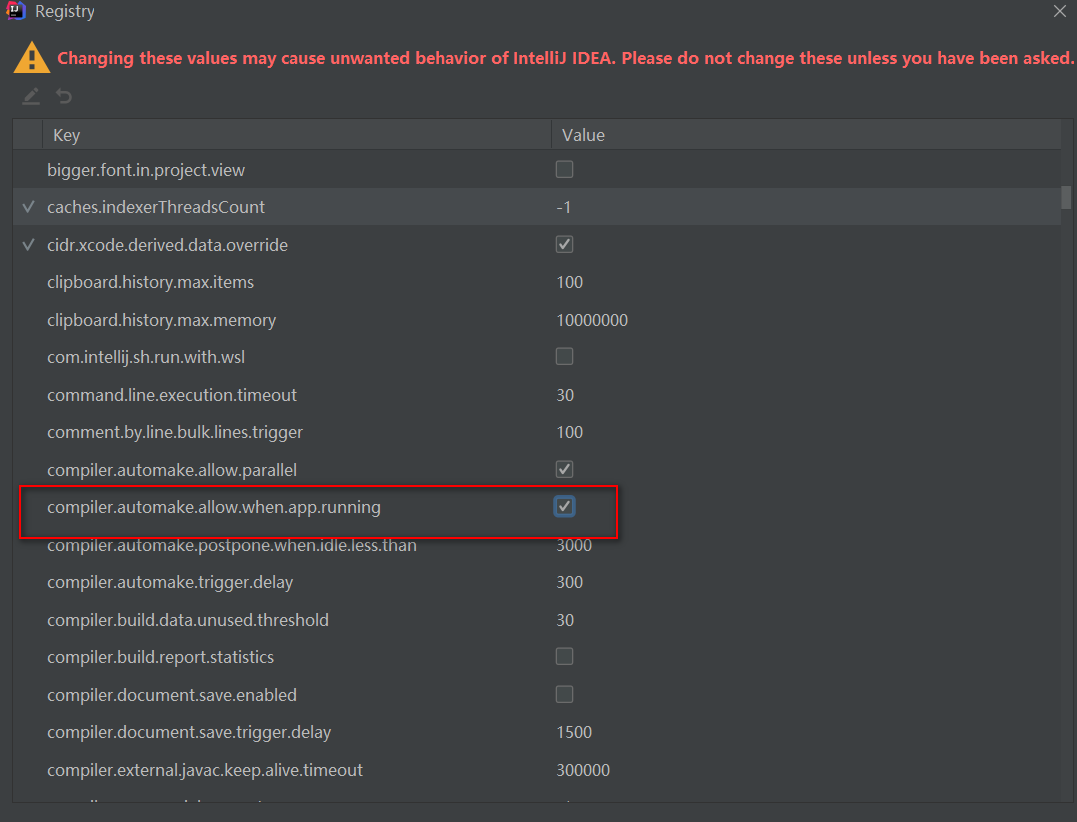
Idea创建table



自动编译：



C+S+A+/：



# ======底=====