# Java设计模式精讲

定义与类型

适用场景

优点

缺点

Coding

源码解析

## 第1节UML类图及时序图入门

### 1.1本章导航

* 定义
* 特点
* 分类
* 类图
* 时序图
* 记忆技巧

### 1.2UML定义

* 统一建模语言（Unified Modeling Language，缩写UML）
* 非专利的第三代建模和规约语言

### 1.3UML特点

* UML是一种开放的方法
* 用于说明、可视化（源代码、测试用例）、构建和编写一个正在开发的面向对象的、软件密集系统的制品的开放方法
* UML展现了一系列最佳工程实践，这些最佳实践在对大规模、复杂系统进行建模方面，特别是在软件架构层次已经被验证有效，方便程序员直接沟通交流

### 1.4UML2.2分类

UML2.2中一共定义了14中图示，分类如下：

结构式图形：强调的是系统式的建模；

行为式图形：强调系统模型中触发的事件；

交互式图形：属于行为式图形子集合，强调系统模型中资料流程；

结构式图形：

静态图（类图，对象图，包图）

实现图（组件图，部署图）

剖面图

复合图

行为式图形：

活动图

状态图

用例图

交互式图形：

通信图

交互概述图（UML2.0）

时序图（UML2.0）

时间图（UML2.0）

UML类图：

Class Diagram：用于表示类、接口、实例等之间相互的静态关系；

虽然名字叫类图，但类图中并不只有类；

### 1.5UML类图及记忆技巧

1. UML箭头方向：从子类指向父类；

提示：错误理解：可能会认为子类是以父类为基础的，箭头应从父类指向子类；

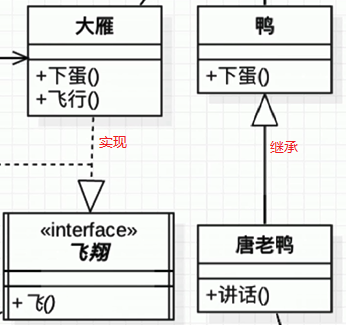
定义子类时需要通过extends关键字指定父类；

子类一定知道父类定义的，但父类并不知道子类的定义；

只有知道对方信息时才能指向对方；

所以箭头方向是子类指向父类；

1. 实线-继承 | 虚线-实现

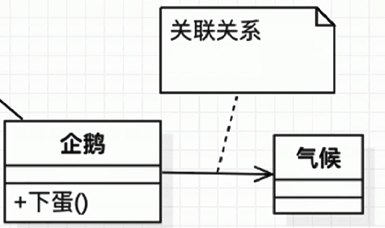


空心三角箭头：继承或实现；

实线-继承：is a关系，扩展目的，不虚，很结实；

虚线-实现：虚线代表“虚”无实体；

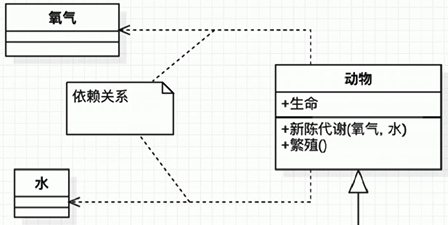
1. 实线-关联 | 虚线-依赖



实线-关联关系：关系稳定，实打实的关系，铁哥们；

表示一个类对象和另一个类对象有关联；

通常是一个类中有另一个类对象作为属性；



虚线-依赖关系：临时用一下，若即若离，虚无缥缈，若有若无；

表示一种使用关系，一个类需要借助另一个类来实现功能；

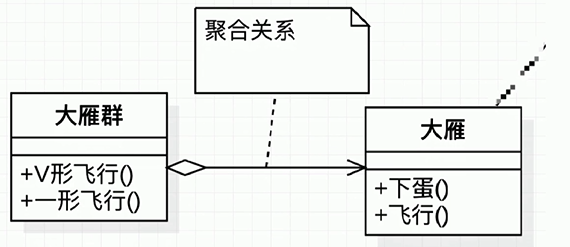
一般是一个类使用另一个类作为参数使用，或作为返回值；

1. 空心菱形-聚合 | 实心菱形-组合

菱形就是一个盛东西的器皿（例如盘子）；

聚合：代表空器皿里可以放很多相同的东西，聚在一起（箭头方向所指的类）；

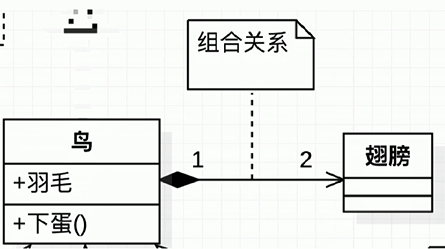
组合：代表器皿里已经有实体结构的存在，生死与共，强关系；



整体和局部的关系，两者有着独立的生命周期，是has a的关系；

弱关系

消极的词：弱-空



整体与局部的关系，和聚合的关系相比，关系更加强烈，两者有着相同的生命周期，contains a的关系

强关系

积极的词：强-满

实心菱形-组合关系中，常见数字表达及含义，假设有A类和B类，数字标记在A类侧：

0..1 ：0个或1个实例；（系统某一时刻）

0..\* ：0个或多个实例；

1..1 ：1个实例；== 1..1 表示一个B实例只能与一个A实例相关

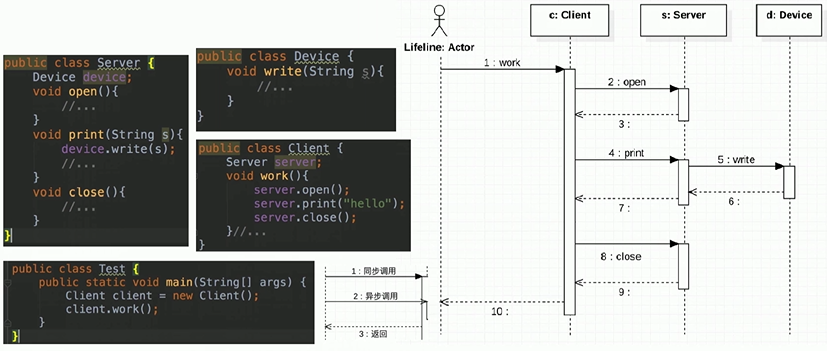
1 ：只能有一个实例；

1..\* ：至少有一个实例；表示B实例可以与多个A实例相关

### 1.6UML时序图

Sequence Diagram：是显示对象之间交互的图，这些对象是按时间顺序排列的。

时序图中包括的建模元素主要有：对象（Actor）、生命线（Lifeline）、控制焦点（Focus of control）、消息（Message）等



## 第2节UML急速入门

### 2.1why设计模式

程序员必备，面试高频考点

### 2.2学习思路

设计模式定义；

设计模式应用；

源码解析；

系统学习；

结合大量场景Coding；不枯燥、不空洞

动态递进式讲解；

采用Debug方式及内存分析；多线程Debug，人工干预线程；

更容易理解设计模式；

JDK及热门开源框架解析；JDK、Spring、Mybatis、Logback、Gruva等

真正学懂设计模式；

序列攻击，反序列化破坏...

### 2.3大纲

1）UML基础

UML类图

UML时序图

UML类关系

UML记忆技巧

2）7大设计原则

开闭原则

依赖倒置原则

单一职责原则

接口隔离原则

迪米特原则

里氏替换原则

合成复用原则

1. 设计模式-创建型模式

工厂方法模式

抽象工厂模式

建造者模式

单例模式

原型模式

1. 设计模式-结构型模式

适配器模式；

装饰者模式；

代理模式；

外观模式；

桥接模式；

组合模式；

享元模式；

1. 设计模式-行为型模式

策略模式；

观察者模式；

责任链模式；

备忘录模式；

模板方法模式；

迭代器模式；

中介者模式；

命令模式；

访问者模式；

解释器模式；

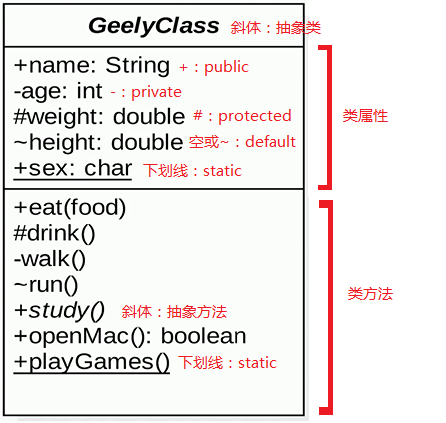
状态模式；

1. 辅助学习资料

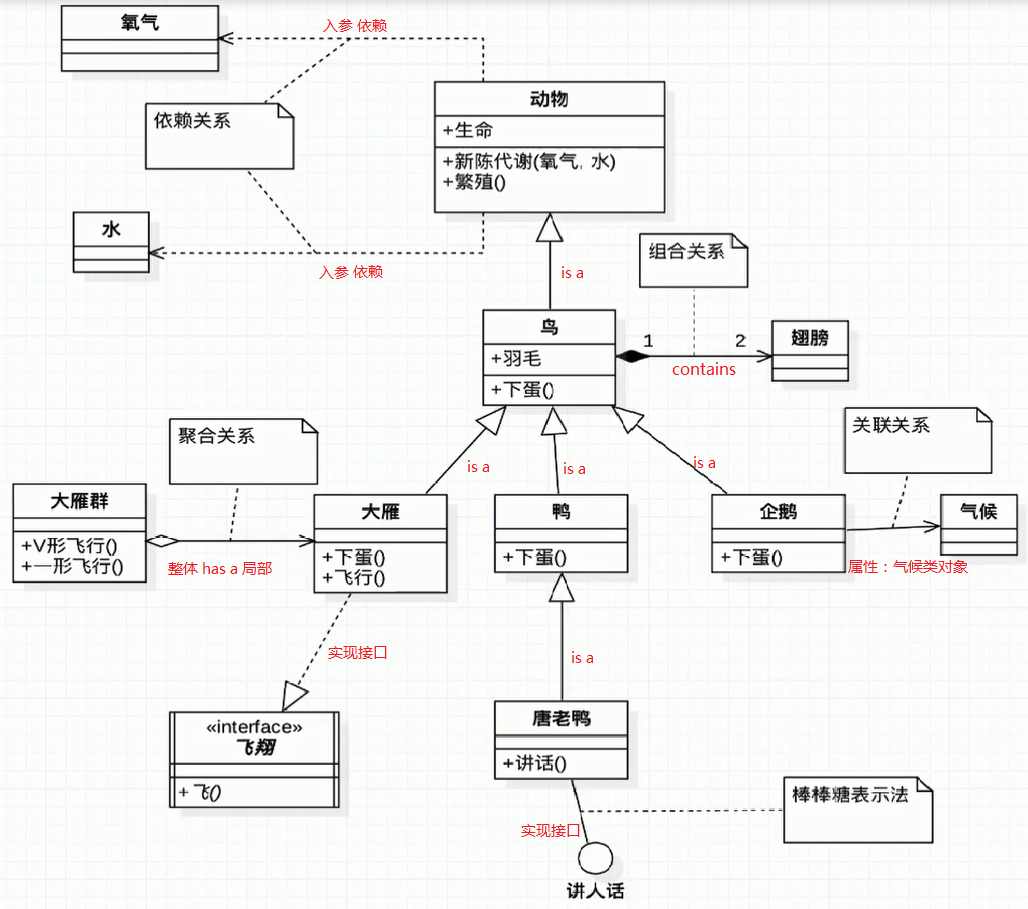
实战工厂能力；

抽象思维能力；

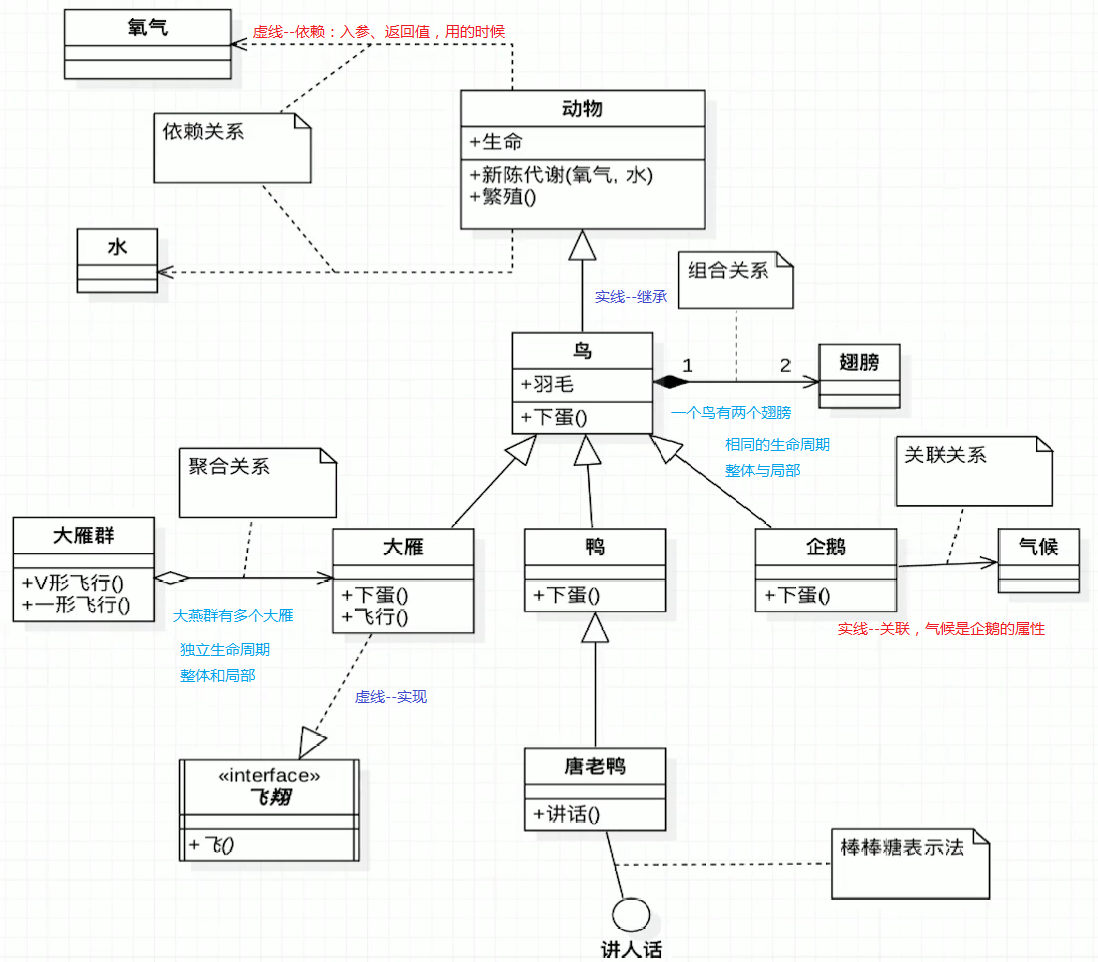
### 2.4UML类图讲解



### 2.5UML类图讲解-自上而下



### 2.6UML类图讲解-对比讲解



### 2.7总结

定义

特点

分类

类图

时序图

记忆技巧

## 软件设计七大原则

### 3.1本章导航

开闭原则

依赖倒置原则

单一职责原则

接口隔离原则

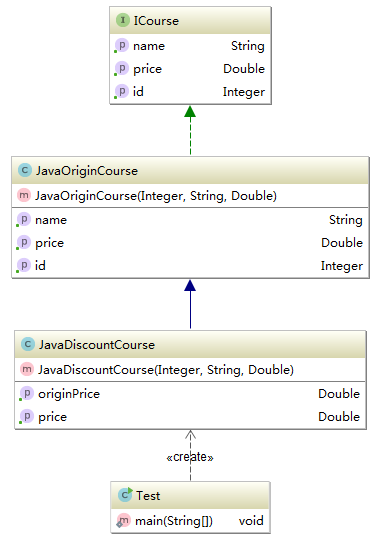
迪米特法则

里氏替换原则

合成/复用原则（组合/复用原则）

### 3.2开闭原则

1. 定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。
2. 用抽象构建框架，用实现扩展细节
3. 优点：提高软件系统的可复用性及可维护性
4. 类图



1. 代码示例

|  |
| --- |
| **public interface** ICourse {  Integer getId();  String getName();  Double getPrice(); } |

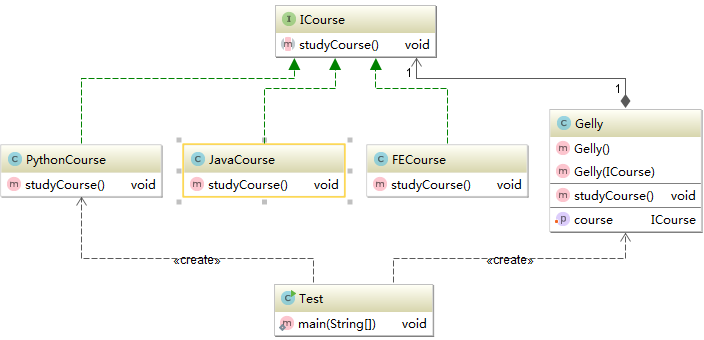
|  |
| --- |
| **public class** JavaOriginCourse **implements** ICourse {  **private** Integer **Id**;  **private** String **name**;  **private** Double **price**;   **public** JavaOriginCourse(Integer id, String name, Double price) {  **this**.**Id** = id;  **this**.**name** = name;  **this**.**price** = price;  }   @Override  **public** Integer getId() {  **return this**.**Id**;  }   @Override  **public** String getName() {  **return this**.**name**;  }   @Override  **public** Double getPrice() {  **return this**.**price**;  } } |

|  |
| --- |
| **public class** JavaDiscountCourse **extends** JavaOriginCourse{  **public** JavaDiscountCourse(Integer id, String name, Double price) {  **super**(id, name, price);  }   **public** Double getOriginPrice() {  **return super**.getPrice();  }  @Override  **public** Double getPrice() {  **return super**.getPrice()\*0.8;  } } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  ICourse javaCourse = **new** JavaDiscountCourse(96,**"Java"**,348d );  JavaDiscountCourse javaDiscountCourse = (JavaDiscountCourse)javaCourse;  System.***out***.println(**"课程ID："** + javaCourse.getId() + **"课程名称："** + javaCourse.getName() + **"课程打折价格："** + javaDiscountCourse.getPrice());  System.***out***.println(**"课程ID："** + javaCourse.getId() + **"课程名称："** + javaCourse.getName() + **"课程原来价格："** + javaDiscountCourse.getOriginPrice());  } } |

### 3.3依赖倒置原则

1. 定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；
2. 抽象不应该依赖细节；细节应该依赖抽象；
3. 针对接口编程，不要针对实现编程；
4. 优点：可以减少类间耦合性、提高系统稳定性，提高代码可读性和可维护性，可降低修改程序所造成的风险；
5. 类图



1. Coding

|  |
| --- |
| **public interface** ICourse {  **public void** studyCourse(); } |

|  |
| --- |
| **public class** FECourse **implements** ICourse {  @Override  **public void** studyCourse() {  System.***out***.println(**"学习FE课程"**);  } } |

|  |
| --- |
| **public class** JavaCourse **implements** ICourse {  @Override  **public void** studyCourse() {  System.***out***.println(**"学习Java课程"**);  } } |

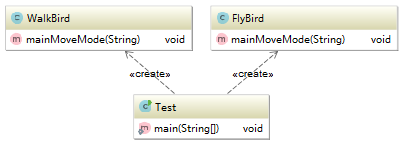
|  |
| --- |
| **public class** Gelly {  **private** ICourse **course**;  Gelly(){   }   Gelly(ICourse course){  **this**.**course** = course;  }  **public void** studyCourse() {  **course**.studyCourse();  }   **public void** setCourse(ICourse course) {  **this**.**course** = course;  }  } |

|  |
| --- |
| **public class** Test {  */\*  // 类内部实现  public static void main(String[] args) {  Gelly gelly = new Gelly();  gelly.studyJavaCourse();  gelly.studyFECourse();  }\*/    /\*// 接口方法实现  public static void main(String[] args) {  Gelly gelly = new Gelly();  gelly.studyCourse(new JavaCourse());  gelly.studyCourse(new PythonCourse());  }\*/   /\*// 构造方法实现  public static void main(String[] args) {  Gelly gelly = new Gelly(new JavaCourse());  gelly.studyCourse();  }\*/   // setter方法实现* **public static void** main(String[] args) {  Gelly gelly = **new** Gelly();  gelly.setCourse(**new** PythonCourse());  gelly.studyCourse();  } } |

### 3.4单一职责原则

1. 定义：不要存在对于一个导致类变更的原因；
2. 一个类/接口/方法只负责一项职责；
3. 优点：降低类的负责度、提高类的可读性，提高系统的可维护性、降低变更引起的风险
4. Coding

类单一职责原则



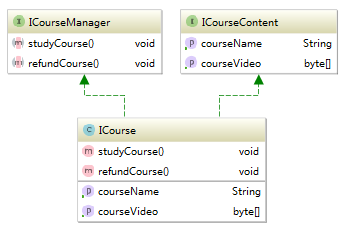
|  |
| --- |
| **public class** Bird {  **public void** mainMoveMode(String birdName) {  **if** (**"鸵鸟"**.equals(birdName)) {  System.***out***.println(birdName + **"用脚走"**);  } **else** {  System.***out***.println(birdName + **"用翅膀飞"**);  }  } } |

|  |
| --- |
| **public class** FlyBird {  **public void** mainMoveMode(String birdName) {  System.***out***.println(birdName + **"用翅膀飞"**);  } } |

|  |
| --- |
| **public class** WalkBird {  **public void** mainMoveMode(String birdName) {  System.***out***.println(birdName + **"用脚走"**);  } } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  */\*Bird bird = new Bird();  bird.mainMoveMode("鸵鸟");  bird.mainMoveMode("大雁");\*/* FlyBird flyBird = **new** FlyBird();  flyBird.mainMoveMode(**"大雁"**);   WalkBird walkBird = **new** WalkBird();  walkBird.mainMoveMode(**"鸵鸟"**);  } } |

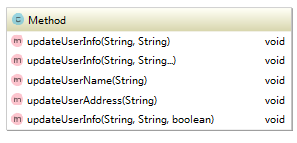
接口单一职责原则



|  |
| --- |
| **public interface** ICourseContent {  String getCourseName();  **byte**[] getCourseVideo(); } |

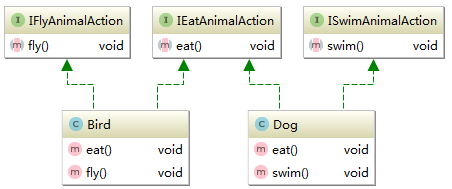
|  |
| --- |
| **public interface** ICourseManager {  **void** studyCourse();  **void** refundCourse(); } |

|  |
| --- |
| **public class** ICourse **implements** ICourseManager, ICourseContent{    @Override  **public void** studyCourse() {   }   @Override  **public void** refundCourse() {   }   @Override  **public** String getCourseName() {  **return null**;  }   @Override  **public byte**[] getCourseVideo() {  **return new byte**[0];  } } |



### 3.5接口隔离原则

1. 定义：用多个专门的接口，而不使用单一的总接口；客户端不应该依赖它不需要的接口
2. 一个类对一个类的依赖应该建立在最小的接口上；
3. 建立单一接口，不要建立庞大臃肿的接口；
4. 尽量细化接口，接口中的方法尽量少；
5. 注意适度原则，一定要适度；
6. 优点：符合我们常说的高内聚低耦合的设计思想，从而使得类具有很好的可读性、可扩展性和可维护性。
7. 类图



1. Coding

|  |
| --- |
| **public interface** IAnimalAction {  **void** eat();  **void** fly();  **void** swim(); } |

|  |
| --- |
| **public interface** IEatAnimalAction {  **void** eat(); } |

|  |
| --- |
| **public interface** IFlyAnimalAction {  **void** fly(); } |

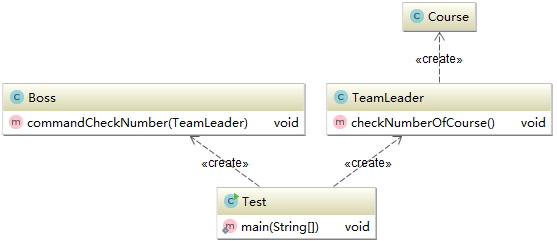
|  |
| --- |
| **public interface** ISwimAnimalAction {  **void** swim(); } |

|  |
| --- |
| **public class** Bird **implements** IEatAnimalAction, IFlyAnimalAction{   @Override  **public void** eat() {   }   @Override  **public void** fly() {   } } |

|  |
| --- |
| **public class** Dog **implements** IEatAnimalAction,ISwimAnimalAction {   @Override  **public void** eat() {   }   @Override  **public void** swim() {   } } |

### 3.6迪米特原则

1. 定义：一个对象应该对其他对象保持最少的了解。又叫最少知道原则；
2. 尽量降低类与类之间的耦合；
3. 优点：降低类之间的耦合；
4. 强调只和朋友交流，不和陌生人说话
5. 朋友：出现在成员变量、方法的输入、输出参数中的类成为成员朋友类，而出现在方法体内部的类不输入朋友类。
6. 类图



1. Coding

|  |
| --- |
| **public class** Boss {  **public void** commandCheckNumber(TeamLeader teamLeader){  teamLeader.checkNumberOfCourse();  } } |

|  |
| --- |
| **public class** TeamLeader {  **public void** checkNumberOfCourse() {  List<Course> courseList = **new** ArrayList<>();  **for** (**int** i = 0;i<20;i++) {  courseList.add(**new** Course());  }  System.***out***.println(**"在线课程的数量是："** + courseList.size());  } } |

|  |
| --- |
| **public class** Course { } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  Boss boss = **new** Boss();  TeamLeader teamLeader = **new** TeamLeader();  boss.commandCheckNumber(teamLeader);  } } |

## 简单工厂

### 4.1定义与类型

1. 定义：由一个工厂对象决定创建出哪一种产品的实例
2. 类型：创建型，但不属于GOF23种设计模式

### 4.2使用场景

工厂类负责创建的对象比较少

客户端（应用层）只知道传入工厂类的参数

对于如何创建对象（逻辑）不关心

### 4.3优点

只需要传入一个正确的参数，就可以获取所需要的对象，而无需知道其创建细节

### 4.4缺点

工厂类的职责相对过重，增加新的产品需要修改工厂类的判断逻辑，违背开闭原则

### 4.5Coding

|  |
| --- |
| **public abstract class** Video {  **public abstract void** produce(); } |

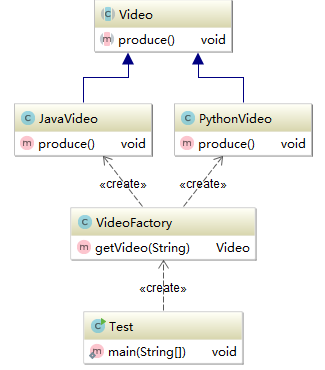
|  |
| --- |
| **public class** JavaVideo **extends** Video {  @Override  **public void** produce() {  System.***out***.println(**"录制Java课程视频"**);  } } |

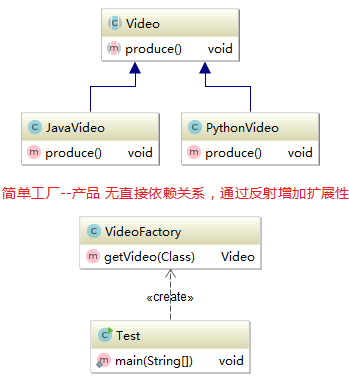
|  |
| --- |
| **public class** PythonVideo **extends** Video {  @Override  **public void** produce() {  System.***out***.println(**"录制Python的视频课程"**);  } } |

|  |
| --- |
| **public class** VideoFactory {  **public** Video getVideo(Class c) {  Video video = **null**;  **try** {  video = (Video) Class.*forName*(c.getName()).newInstance();  } **catch** (InstantiationException e) {  e.printStackTrace();  } **catch** (IllegalAccessException e) {  e.printStackTrace();  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  **return** video;  }  */\*public Video getVideo(String type) {  if ("java".equalsIgnoreCase(type)){  return new JavaVideo();  } else if ("python".equalsIgnoreCase(type)) {  return new PythonVideo();  }  return null;  }\*/* } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  VideoFactory videoFactory = **new** VideoFactory();  Video video = videoFactory.getVideo(JavaVideo.**class**);  *// Video video = new PythonVideo();* **if** (video == **null**){  **return**;  }  video.produce();  */\*VideoFactory videoFactory = new VideoFactory();  Video video = videoFactory.getVideo("java");  //Video video = new PythonVideo();  if (video == null){  return;  }  video.produce();\*/* } } |

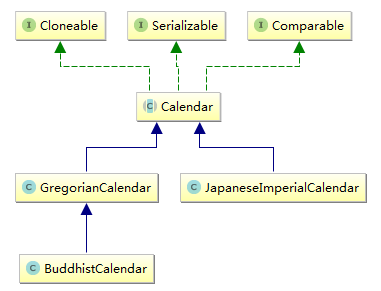
### 4.6UML类图





### 4.7源码解析（JDK、Log4j、sql驱动）

Calendar



MysqlDriver

LogFactory

## 第5节工厂方法-关注产品等级结构（一类事务：冰箱）

### 5.1定义与类型

1. 定义：定义一个创建对象的接口

但让实现这个接口的类来决定实例化哪个类；

工厂方法让类的实例化推迟到子类中进行；

1. 类型：创建型

### 5.2适用场景

1）创建对象需要大量重复的代码

2）客户端（应用层）不依赖与产品类实例如何创建、实现等细节

3）一个类通过其子类来指定创建哪个对象

面向对象多态性、里氏替换原则

### 5.3优点

1）用户只需要关心所需产品对应的工厂，无须关心创建细节

2）加入新产品符合开闭原则，提高可扩展性

### 5.4缺点：

1）类的个数容易过多，增加负责度

2）增加了系统的抽象性和理解难度

### 5.5Coding

|  |
| --- |
| **public abstract class** VideoFactory {  *// 为什么使用抽象类，而不使用接口，这个类对某些行为或某些属性都是已知的，使用抽象类比较合适，如果不知道，使用接口  // 只定义契约，不定义实现* **public abstract** Video getVideo(); } |

|  |
| --- |
| **public abstract class** Video {  **public abstract void** produce(); } |

|  |
| --- |
| **public class** JavaVideo **extends** Video {  @Override  **public void** produce() {  System.***out***.println(**"录制Java课程视频"**);  } } |

|  |
| --- |
| **public class** JavaVideoFactory **extends** VideoFactory {  @Override  **public** Video getVideo() {  **return new** JavaVideo();  } } |

|  |
| --- |
| **public class** PythonVideo **extends** Video {  @Override  **public void** produce() {  System.***out***.println(**"录制Python的视频课程"**);  } } |

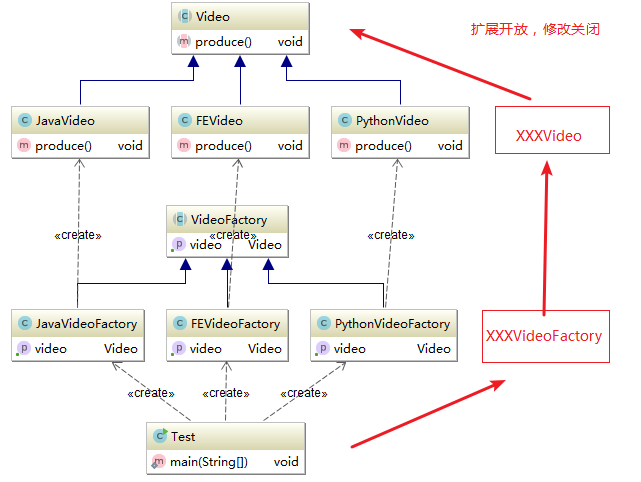
|  |
| --- |
| **public class** PythonVideoFactory **extends** VideoFactory {  @Override  **public** Video getVideo() {  **return new** PythonVideo();  } } |

|  |
| --- |
| **public class** FEVideo **extends** Video {  @Override  **public void** produce() {  System.***out***.println(**"录制FE课程视频"**);  } } |

|  |
| --- |
| **public class** FEVideoFactory **extends** VideoFactory {  @Override  **public** Video getVideo() {  **return new** FEVideo();  } } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  VideoFactory videoFactory = **new** JavaVideoFactory();  VideoFactory videoFactory2 = **new** PythonVideoFactory();  VideoFactory videoFactory3 = **new** FEVideoFactory();  Video video = videoFactory.getVideo();  video.produce();  } } |

### 5.6UML类图



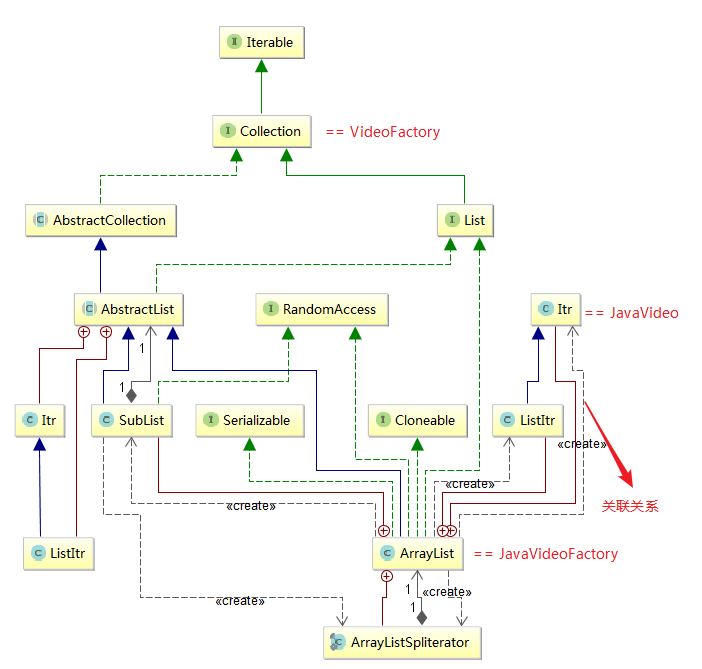
### 5.7源码解析（Collection、URLStreamHandlerFactory）

1. Collection

Collection == VideoFactory

ArrayList == JavaVideoFactory

new Itr() == JavaVideo



1. URLStreamHandlerFactory

URLStreamHandlerFactory == VideoFactory

Factory == JavaVideoFactory

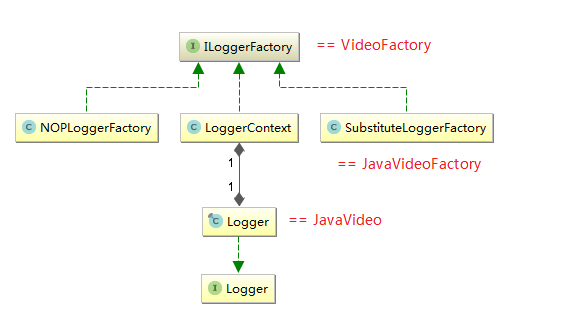
URLStreamHandler == JavaVideo

1. LoggerFactory

ILoggerFactory == VideoFactory

LoggerContext、NOPLoggerFactory、SubstituteLoggerFactory == JavaVideoFactory、PythonVideoFactory、FEVideoFactory

Logger == JavaVideo



## 第6节抽象工厂-关注产品族（同一品牌家电）

### 6.1定义与类型

1. 定义：抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口
2. 无须指定它们具体的类
3. 类型：创建型

### 6.2适用场景

1. 客户端（应用层）不依赖与产品类实例如何被创建、实现等细节
2. 强调一系列相关的产品对象（属于同一产品族）一起使用创建对象需要大量重复的代码；
3. 提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现

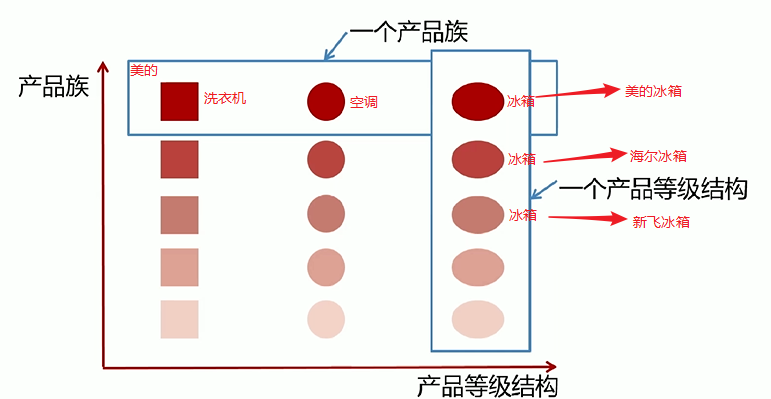
### 6.3优点

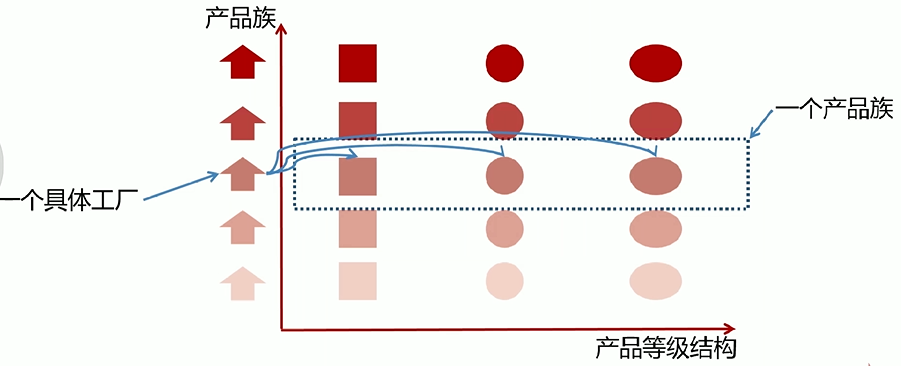
1. 具体产品在应用层代码隔离，无须关心创建细节；
2. 将一个系列的产品族统一到一起创建；

### 6.4缺点

1. 规定了所有可能被创建的产品集合，产品族中扩展新的产品困难，需要修改抽象工厂的接口；
2. 增加系统抽象性；

### 6.5产品等级结构与产品族





### 6.6 Coding

|  |
| --- |
| **public interface** CourseFactory {  Video getVideio();  Article getArticl(); } |

|  |
| --- |
| **public abstract class** Article {  **public abstract void** produce(); } |

|  |
| --- |
| **public abstract class** Video {  **public abstract void** produce(); } |

|  |
| --- |
| **public class** JavaCourseFactory **implements** CourseFactory {  **public** Video getVideio() {  **return new** JavaVideo();  }   **public** Article getArticl() {  **return new** JavaArticle();  } } |

|  |
| --- |
| **public class** PythonCourseFactory **implements** CourseFactory {  **public** Video getVideio() {  **return new** PythonVideo();  }   **public** Article getArticl() {  **return new** PythonArticle();  } } |

|  |
| --- |
| **public class** JavaArticle **extends** Article {  @Override  **public void** produce() {  System.***out***.println(**"Java文章"**);  } } |

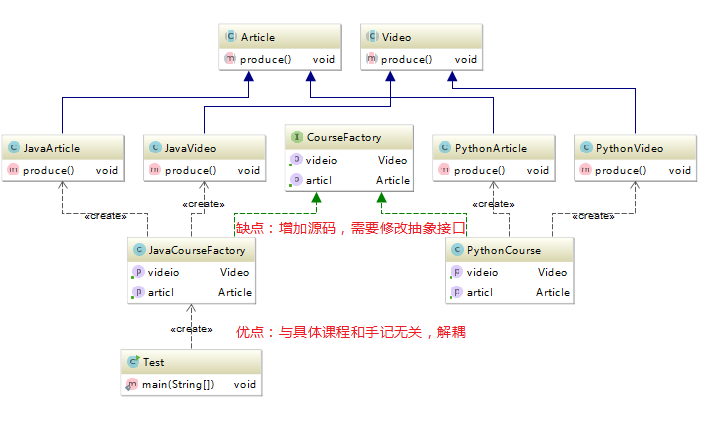
|  |
| --- |
| **public class** JavaVideo **extends** Video {  @Override  **public void** produce() {  System.***out***.println(**"录制Java课程视频"**);  } } |

|  |
| --- |
| **public class** PythonVideo **extends** Video {  @Override  **public void** produce() {  System.***out***.println(**"录制Python课程视频"**);  } } |

|  |
| --- |
| **public class** PythonArticle **extends** Article {  @Override  **public void** produce() {  System.***out***.println(**"Python文章"**);  } } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  CourseFactory courseFactory = **new** JavaCourseFactory();  Video video = courseFactory.getVideio();  Article article = courseFactory.getArticl();  video.produce();  article.produce();  } } |

### 6.7UML类图



### 6.8源码解析（mybatis）

1. java.sql.connection
2. java.sql.statement
3. Org.apache.ibatis.session
4. Org.apache.ibatis.session.SqlSessionFactory -- > DefaultSqlSessionFactory -->

Ctrl + F12打开类的方法目录

Ctrl + alt + b打开接口的实现类

Ctrl + n查找class类名

## 第7节建造者

### 7.1定义与类型

1. 定义：讲一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示

用户只需要指定需要建造的类型就可以得到它们，建造过程及细节不需要知道

1. 类型：创建型

### 7.2适用场景

1. 如果一个对象有非常复杂的内部结构（很多属性）
2. 想把复杂对象的创建和使用分离

### 7.3优点

1. 封装性好，创建和使用分离
2. 扩展性好、建造类之间独立、一定程度上解耦

### 7.4缺点

1. 产生多余的Builder对象
2. 产品内部发生变化，建造者都需要修改，成本较大

### 7.5建造者与工厂模式区别

1. 建造者模式更关注方法地调用顺序，工厂模式注重产品；
2. 创建对象的粒度不同，建造者模式可以创建复杂的产品，由各种复杂的部件组成，工厂模式创建出来都一样；
3. 关注点：工厂模式只关注把产品创建出来，而建造者模式还要关注产品由哪些部件组成；
4. 建造者模式由不同的建造顺序，根据不同产品，建造顺序不同，工厂模式不关注顺序。

### 7.6Coding

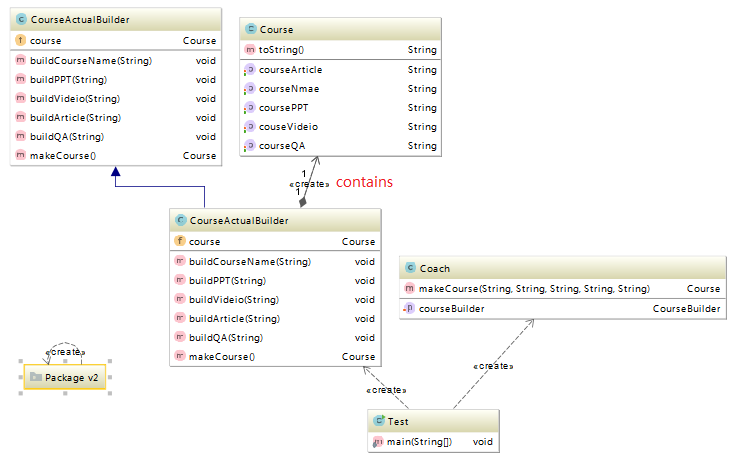
|  |
| --- |
| **public class** Course {  **private** String **courseNmae**;  **private** String **coursePPT**;  **private** String **couseVideio**;  **private** String **courseArticle**;  **private** String **courseQA**;   **public** String getCourseArticle() {  **return courseArticle**;  }   @Override  **public** String toString() {  **return "Course{"** +  **"courseNmae='"** + **courseNmae** + **'\''** +  **", coursePPT='"** + **coursePPT** + **'\''** +  **", couseVideio='"** + **couseVideio** + **'\''** +  **", courseArticle='"** + **courseArticle** + **'\''** +  **", courseQA='"** + **courseQA** + **'\''** +  **'}'**;  }   **public void** setCourseArticle(String courseArticle) {  **this**.**courseArticle** = courseArticle;  }   **public** String getCourseNmae() {  **return courseNmae**;  }   **public void** setCourseNmae(String courseNmae) {  **this**.**courseNmae** = courseNmae;  }   **public** String getCoursePPT() {  **return coursePPT**;  }   **public void** setCoursePPT(String coursePPT) {  **this**.**coursePPT** = coursePPT;  }   **public** String getCouseVideio() {  **return couseVideio**;  }   **public void** setCouseVideio(String couseVideio) {  **this**.**couseVideio** = couseVideio;  }   **public** String getCourseQA() {  **return courseQA**;  }   **public void** setCourseQA(String courseQA) {  **this**.**courseQA** = courseQA;  } } |

|  |
| --- |
| **public abstract class** CourseBuilder {  **public abstract void** buildCourseName(String courseName);  **public abstract void** buildPPT(String coursePPT);  **public abstract void** buildVideio(String courseVideo);  **public abstract void** buildArticle(String courseArticle);  **public abstract void** buildQA(String courseQA);   **public abstract** Course makeCourse(); } |

|  |
| --- |
| **public class** CourseActualBuilder **extends** CourseBuilder {   **private** Course **course** = **new** Course();    @Override  **public void** buildCourseName(String courseName) {  **course**.setCourseNmae(courseName);  }   @Override  **public void** buildPPT(String coursePPT) {  **course**.setCoursePPT(coursePPT);  }   @Override  **public void** buildVideio(String courseVideo) {  **course**.setCouseVideio(courseVideo);  }   @Override  **public void** buildArticle(String courseArticle) {  **course**.setCourseArticle(courseArticle);  }   @Override  **public void** buildQA(String courseQA) {  **course**.setCourseQA(courseQA);  }   @Override  **public** Course makeCourse() {  **return course**;  } } |

|  |
| --- |
| **public class** Coach {  **private** CourseBuilder **courseBuilder**;   **public void** setCourseBuilder(CourseBuilder courseBuilder) {  **this**.**courseBuilder** = courseBuilder;  }   **public** Course makeCourse(String courseName,  String courseVidio,  String coursePPT,  String courseArticle,  String courseQA) {  **this**.**courseBuilder**.buildCourseName(courseName);  **this**.**courseBuilder**.buildPPT(coursePPT);  **this**.**courseBuilder**.buildQA(courseArticle);  **this**.**courseBuilder**.buildQA(courseQA);  **this**.**courseBuilder**.buildVideio(courseVidio);  **return this**.**courseBuilder**.makeCourse();  } } |

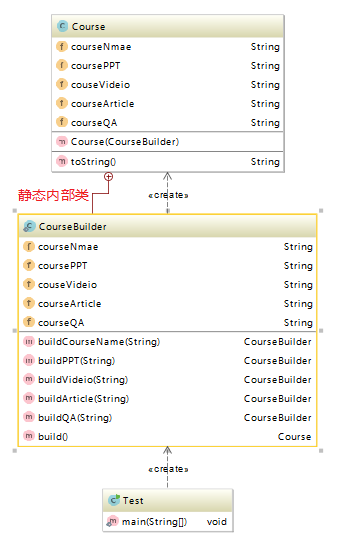
|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  Course course = **new** Course.CourseBuilder().buildCourseName(**"Java设计模式"**).buildArticle(**"Java设计模式手记"**).buildPPT(**"Java设计模式PPT"**).build();  System.***out***.println(course);  } } |



V2版本，Course有CourseBuilder静态内部类，链式build

|  |
| --- |
| **public class** Course {  **private** String **courseNmae**;  **private** String **coursePPT**;  **private** String **couseVideio**;  **private** String **courseArticle**;  **private** String **courseQA**;   **public** Course(CourseBuilder courseBuilder) {  **this**.**courseNmae** = courseBuilder.**courseNmae**;  **this**.**coursePPT** = courseBuilder.**coursePPT**;  **this**.**couseVideio** = courseBuilder.**couseVideio**;  **this**.**courseArticle** = courseBuilder.**courseArticle**;  **this**.**courseQA** = courseBuilder.**courseQA**;  }   @Override  **public** String toString() {  **return "Course{"** +  **"courseNmae='"** + **courseNmae** + **'\''** +  **", coursePPT='"** + **coursePPT** + **'\''** +  **", couseVideio='"** + **couseVideio** + **'\''** +  **", courseArticle='"** + **courseArticle** + **'\''** +  **", courseQA='"** + **courseQA** + **'\''** +  **'}'**;  }   **public static class** CourseBuilder{  **private** String **courseNmae**;  **private** String **coursePPT**;  **private** String **couseVideio**;  **private** String **courseArticle**;  **private** String **courseQA**;     **public** CourseBuilder buildCourseName(String courseNmae){  **this**.**courseNmae** = courseNmae;  **return this**;  }   **public** CourseBuilder buildPPT(String coursePPT) {  **this**.**coursePPT** = coursePPT;  **return this**;  }   **public** CourseBuilder buildVideio(String courseVideo) {  **this**.**couseVideio** = courseVideo;  **return this**;  }   **public** CourseBuilder buildArticle(String courseArticle) {  **this**.**courseArticle**=courseArticle;  **return this**;  }   **public** CourseBuilder buildQA(String courseQA) {  **this**.**courseQA** = courseQA;  **return this**;  }   **public** Course build() {  **return new** Course(**this**);  }  } } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  Course course = **new** Course.CourseBuilder().buildCourseName(**"Java设计模式"**).buildArticle(**"Java设计模式手记"**).buildPPT(**"Java设计模式PPT"**).build();  System.***out***.println(course);  } } |

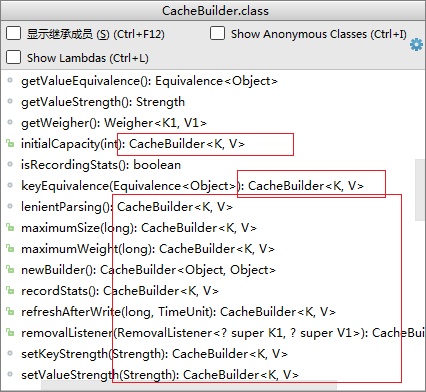


### 7.6源码解析（StringBuilder、StringBuffer、Guava：ImmutableSet、BeanDefinitionBuilder、SQLSessionFactory）

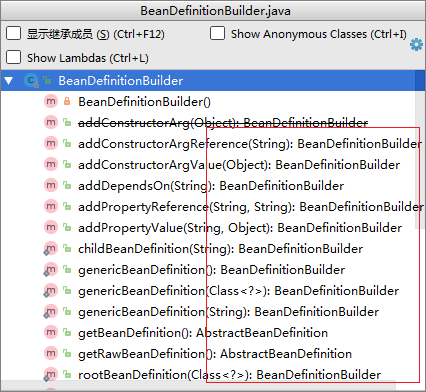
1. StringBuilder（线程不安全），StringBuffer，（安全）
2. ImmutableSet：

Set<String> set = ImmutableSet.<String>builder().add("a").add("b").add("b").build();

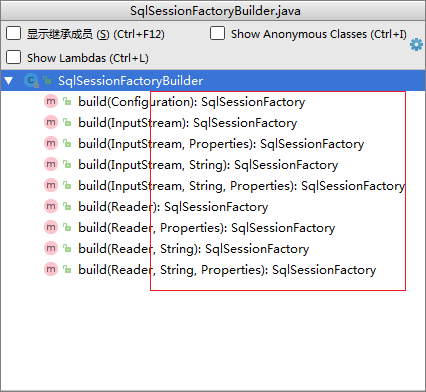
1. Cache的LRU算法，CacheBuilder，设置属性，返回值类型均为该类本身



1. BeanDefinitionBuilder



1. SQLSessionFactory



|  |
| --- |
| **public** SqlSessionFactory build(Reader reader, String environment, Properties properties) {  **try** {  XMLConfigBuilder parser = **new** XMLConfigBuilder(reader, environment, properties);  **return** build(parser.parse());  } **catch** (Exception e) {  **throw** ExceptionFactory.*wrapException*(**"Error building SqlSession."**, e);  } **finally** {  ErrorContext.*instance*().reset();  **try** {  reader.close();  } **catch** (IOException e) {  *// Intentionally ignore. Prefer previous error.* }  } } |

|  |
| --- |
| **private void** parseConfiguration(XNode root) {  **try** {  Properties settings = settingsAsPropertiess(root.evalNode(**"settings"**));  *//issue #117 read properties first* propertiesElement(root.evalNode(**"properties"**));  loadCustomVfs(settings);  typeAliasesElement(root.evalNode(**"typeAliases"**));  pluginElement(root.evalNode(**"plugins"**));  objectFactoryElement(root.evalNode(**"objectFactory"**));  objectWrapperFactoryElement(root.evalNode(**"objectWrapperFactory"**));  reflectorFactoryElement(root.evalNode(**"reflectorFactory"**));  settingsElement(settings);  *// read it after objectFactory and objectWrapperFactory issue #631* environmentsElement(root.evalNode(**"environments"**));  databaseIdProviderElement(root.evalNode(**"databaseIdProvider"**));  typeHandlerElement(root.evalNode(**"typeHandlers"**));  mapperElement(root.evalNode(**"mappers"**));  } **catch** (Exception e) {  **throw new** BuilderException(**"Error parsing SQL Mapper Configuration. Cause: "** + e, e);  } } |

## 第8节单例

### 8.1定义与类型

定义：保证一个类仅有一个实例，并提供一个全局访问点

类型：创建型

### 8.2适用场景

1. 想确保任何情况下都绝对只有一个实例；
2. 线程池、数据库连接池、网站计数器

### 8.3优点

1. 在内存里只有一个实例，减少了内存开销；
2. 可以避免对资源的多重占用；
3. 设置全局访问点，严格的控制访问；

### 8.4缺点

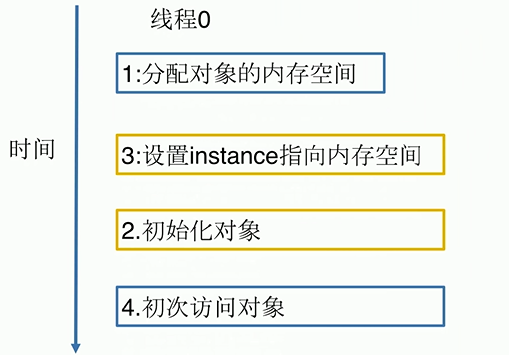
没有接口，难以扩展

### 8.5重点

1. 私有构造器；private
2. 线程安全；
3. 延迟加载；
4. 序列化和反序列化安全；
5. 反射；（反射攻击）

### 8.6单例双重检查锁

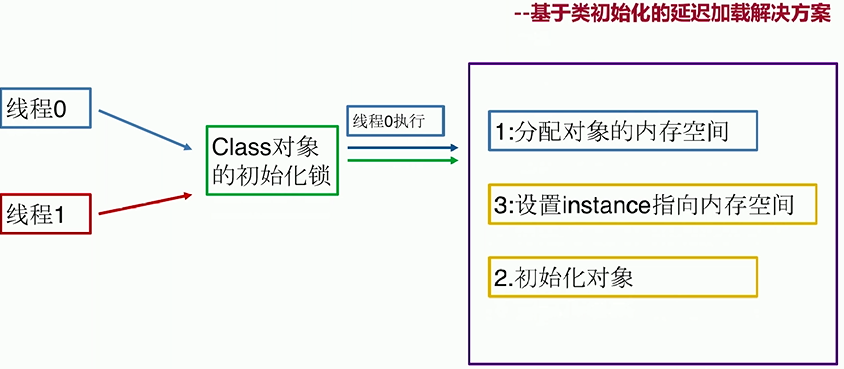
指令重排序



重排序后，instane已经有对象值，但并未初始化，另外一个线程判断instance不为null，将未初始化的instance取走使用，则会报错。



Volatile，所有线程都能看到共享内存的最新状态，保证内存的可见性，用volatile修饰的共享变量，在进行写操作的时候，会多产生一些汇编代码，起到两个作用，第一：将当前处理器缓存行的数据，写回到系统内存；该写回操作会使其他CPU缓存的该变量的地址无效，因为其他CPU缓存的数据无效，所以其他线程又从内存同步共享数据。这样就保证的数据一致性，这是缓存一致性协议的作用。



非构造线程不允许看到重排序；

初始化一个类包括这个类的静态初始化，还有初始化在这个类中声明的静态量表，五种立刻初始化的情况：首次发生，一个类（或接口）将被立刻初始化，

假设这个类是A

1）有一个A类型的实例被初始化创建；

2）A类中声明的静态方法被调用；

3）A类中声明的静态成员被赋值；

4）A类中声明的静态成员被使用，并且这个成员不是一个常量成员；

5）A类如果是一个顶级类，并且在这个类中有嵌套的断言语句；

任何一个线程getInstance，都会调用InnerClass的静态成员变量，类的初始化只有一次，任何线程过来，InnerClass对象的初始化锁就会发挥作用，那么只能有一个线程见证该静态内部类的初始化，指令重排序对其他线程而言是看不到的。不论怎么排序，其他线程一定会等该内部类初始化完毕后再使用，所以该静态成员变量一定是可用且唯一的。

实用技能

1. 反编译；
2. 内存原理；
3. 多线程Debug；（分布式锁多进程）

单例模式相关设计模式

1. 单例与工厂；
2. 单例与享元；

### 8.6Coding

#### 8.6.1懒汉式单例（线程不安全）

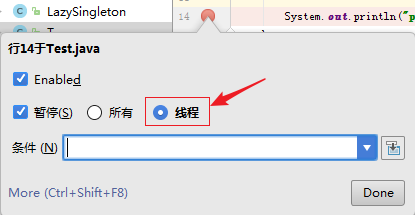
|  |
| --- |
| **public class** LazySingleton {  **private static** LazySingleton *lazySingleton* = **null**;  **private** LazySingleton() {   }   **public static** LazySingleton getInstance(){  **if**(*lazySingleton* == **null**) {  *lazySingleton* = **new** LazySingleton();  }  **return** *lazySingleton*;  } } |

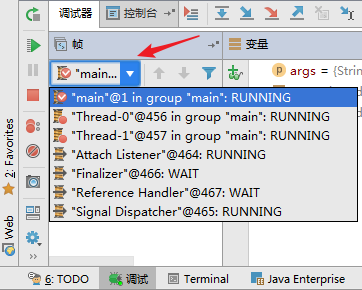
#### 8.6.2多线程Debug

|  |
| --- |
| **public class** T **implements** Runnable {  @Override  **public void** run() {  LazySingleton lazySingleton = LazySingleton.*getInstance*();System.***out***.println(Thread.*currentThread*().getName()+**" "**+lazySingleton);  } } |

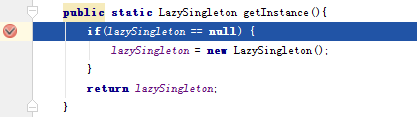
|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {Thread t1 = **new** Thread(**new** T());  Thread t2 = **new** Thread(**new** T());  t1.start();  t2.start();  System.***out***.println(**"program end"**);  } } |

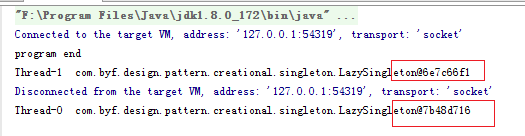
断点执行模式





Thread-0和Thread-1分别进入getInstance，判断lazySingleton都为空，同时new了两个单例

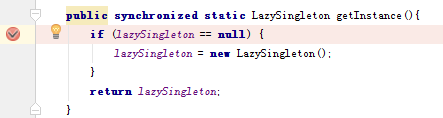




#### 8.6.3单例改进，同步

|  |
| --- |
| **public class** LazySingleton {  **private static** LazySingleton *lazySingleton* = **null**;  **private** LazySingleton() {   }   **public static** LazySingleton getInstance(){  // 等价于static前加**synchronized，锁是当前类的字节码对象，全局唯一，锁的范围较大，性能会受影响**  **synchronized** (LazySingleton.**class**) {  **if** (*lazySingleton* == **null**) {  *lazySingleton* = **new** LazySingleton();  }  }  **return** *lazySingleton*;  } } |

多线程单步调试





#### 8.6.4双重加锁机制

|  |
| --- |
| **public class** LazyDoubleCheckSingleton {  **private volatile static** LazyDoubleCheckSingleton *lazyDoubleCheckSingleton* = **null**;  **private** LazyDoubleCheckSingleton(){   }  **public static** LazyDoubleCheckSingleton getInstance(){  **if**(*lazyDoubleCheckSingleton* == **null**){  **synchronized** (LazyDoubleCheckSingleton.**class**){  **if**(*lazyDoubleCheckSingleton* == **null**){  *lazyDoubleCheckSingleton* = **new** LazyDoubleCheckSingleton();  *//1.分配内存给这个对象 // //3.设置lazyDoubleCheckSingleton 指向刚分配的内存地址  //2.初始化对象 // intra-thread semantics // ---------------//3.设置lazyDoubleCheckSingleton 指向刚分配的内存地址* }  }  }  **return** *lazyDoubleCheckSingleton*;  } } |

#### 8.6.5饿汉式

|  |
| --- |
| **public class** HungrySingleton **implements** Serializable,Cloneable{   **private final static** HungrySingleton ***hungrySingleton***;   **static**{  ***hungrySingleton*** = **new** HungrySingleton();  }  **private** HungrySingleton(){  **if**(***hungrySingleton*** != **null**){  **throw new** RuntimeException(**"单例构造器禁止反射调用"**);  }  }  **public static** HungrySingleton getInstance(){  **return *hungrySingleton***;  }   **private** Object readResolve(){  **return *hungrySingleton***;  }   @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **return** *getInstance*();  } } |

#### 8.6.6序列化破坏单例模式原理解析及解决方案

序列化再反序列化后，如果单例没有实现readResolve方法（返回实例），反序列化时将new新的实例

#### 8.6.7反射破坏单例模式原理解析几解决方案

单例类的构造函数是在反射时调用，还是在getInstance时调用，如果是前者，那么反射调用构造方法时，懒加载的实例对象是不存在的，没有走getInstance的逻辑，不符合单例的使用方法。

#### 8.6.8枚举：单例模式的最佳实践

反编译工具：jad

|  |
| --- |
| G:\design\_pattern\target\classes\com\byf\design\pattern\creational\singleton>jad  EnumInstance$1.class  Parsing EnumInstance$1.class... Generating EnumInstance$1.jad  G:\design\_pattern\target\classes\com\byf\design\pattern\creational\singleton>jad  EnumInstance.class  Parsing EnumInstance.class...Parsing inner class EnumInstance$1.class... Generat  ing EnumInstance.jad |

|  |
| --- |
| public abstract class EnumInstance extends Enum {   public static EnumInstance[] values()  {  return (EnumInstance[])$VALUES.clone();  }   public static EnumInstance valueOf(String name)  {  return (EnumInstance)Enum.valueOf(com/byf/design/pattern/creational/singleton/EnumInstance, name);  }   private EnumInstance(String s, int i)  {  super(s, i);  }   protected abstract void printTest();   public Object getData()  {  return data;  }   public void setData(Object data)  {  this.data = data;  }   public static EnumInstance getInstance()  {  return INSTANCE;  }    public static final EnumInstance INSTANCE;  private Object data;  private static final EnumInstance $VALUES[];   static   {  INSTANCE = new EnumInstance("INSTANCE", 0) {   protected void printTest()  {  System.out.println("Byf Print Test");  }   } ;  $VALUES = (new EnumInstance[] {  INSTANCE  });  } } |

#### 8.6.9单例容器类

|  |
| --- |
| **public class** ContainerSingleton {   **private** ContainerSingleton() {   }   **private static** Map<String, Object> *singletonMap* = **new** HashMap<String, Object>();   **public static void** putInstance(String key, Object instance) {  **if** (StringUtils.*isNotBlank*(key) && instance != **null**) {  **if** (!*singletonMap*.containsKey(key)) {  *singletonMap*.put(key, instance);  }  }  }   **public static** Object getInstance(String key) {  **return** *singletonMap*.get(key);  } } |

|  |
| --- |
| **public class** T **implements** Runnable {  @Override  **public void** run() {  ContainerSingleton.*putInstance*(**"object"**, **new** Object());  Object instance = ContainerSingleton.*getInstance*(**"object"**);  System.***out***.println(Thread.*currentThread*().getName() + **" "** + instance);   } } |

|  |
| --- |
| Thread t1 = **new** Thread(**new** T()); Thread t2 = **new** Thread(**new** T()); t1.start(); t2.start(); |

当两个线程都进入到*singletonMap*.put(key, instance)，那么后进入的线程将覆盖前边线程put进去的值。

#### 8.6.10ThreadLocal“单例”空间换时间

|  |
| --- |
| **public class** ThreadLocalInstance {  **private static final** ThreadLocal<ThreadLocalInstance> ***threadLocal*** = **new** ThreadLocal<ThreadLocalInstance>(){  @Override  **protected** ThreadLocalInstance initialValue() {  **return new** ThreadLocalInstance();  }  };  **private** ThreadLocalInstance(){   }  **public static** ThreadLocalInstance getInstance(){  **return *threadLocal***.get();  } } |
| ThreadLocalInstance instance = ThreadLocalInstance.*getInstance*(); System.***out***.println(Thread.*currentThread*().getName() + **" "** + instance); |
| System.***out***.println(**"main Thread : "** + ThreadLocalInstance.*getInstance*()); System.***out***.println(**"main Thread : "** + ThreadLocalInstance.*getInstance*()); System.***out***.println(**"main Thread : "** + ThreadLocalInstance.*getInstance*()); System.***out***.println(**"main Thread : "** + ThreadLocalInstance.*getInstance*()); System.***out***.println(**"main Thread : "** + ThreadLocalInstance.*getInstance*()); Thread t1 = **new** Thread(**new** T()); Thread t2 = **new** Thread(**new** T()); t1.start(); t2.start(); |
| main Thread : com.byf.design.pattern.creational.singleton.ThreadLocalInstance@3830f1c0  main Thread : com.byf.design.pattern.creational.singleton.ThreadLocalInstance@3830f1c0  main Thread : com.byf.design.pattern.creational.singleton.ThreadLocalInstance@3830f1c0  main Thread : com.byf.design.pattern.creational.singleton.ThreadLocalInstance@3830f1c0  main Thread : com.byf.design.pattern.creational.singleton.ThreadLocalInstance@3830f1c0  program end  Thread-0 com.byf.design.pattern.creational.singleton.ThreadLocalInstance@7bb94d92  Thread-1 com.byf.design.pattern.creational.singleton.ThreadLocalInstance@3750eeda |

### 8.7源码解析

懒汉式单例

|  |
| --- |
| **private static** Runtime *currentRuntime* = **new** Runtime();**public static** Runtime getRuntime() {  **return** *currentRuntime*; } |

容器单例

|  |
| --- |
| **public static synchronized** Desktop getDesktop(){  **if** (GraphicsEnvironment.*isHeadless*()) **throw new** HeadlessException();  **if** (!Desktop.*isDesktopSupported*()) {  **throw new** UnsupportedOperationException(**"Desktop API is not "** +  **"supported on the current platform"**);  }   sun.awt.AppContext context = sun.awt.AppContext.*getAppContext*();  Desktop desktop = (Desktop)context.get(Desktop.**class**);   **if** (desktop == **null**) {  desktop = **new** Desktop();  context.put(Desktop.**class**, desktop);  }   **return** desktop; } |

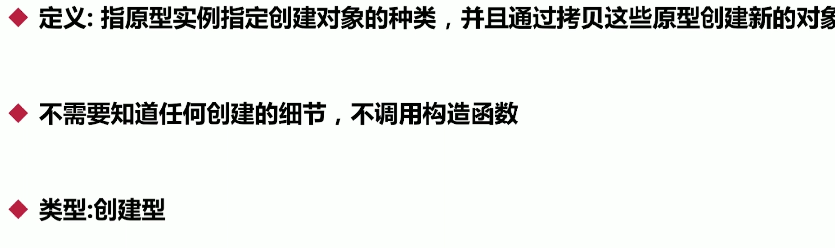
Spring容器单例与此不同，Spring将实例的数量应用于整个上下文，而当前所说的单例只应用于当前的类加载范围。

ThreadLocal“单例”

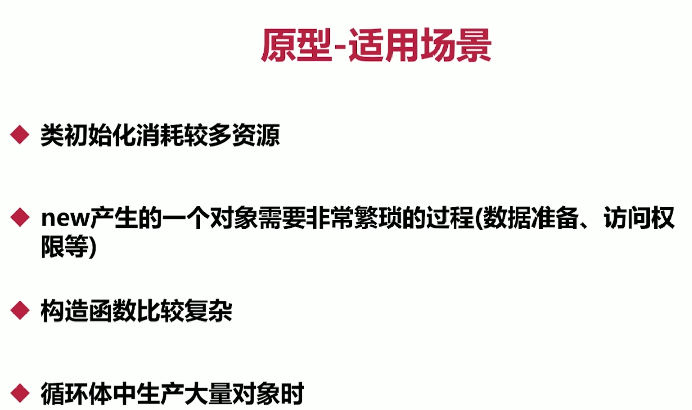
|  |
| --- |
| **private static final** ThreadLocal<ErrorContext> LOCAL = **new** ThreadLocal();  **public static** ErrorContext instance() {  ErrorContext context = (ErrorContext)LOCAL.get();  **if**(context == **null**) {  context = **new** ErrorContext();  LOCAL.set(context);  }   **return** context; } |

## 第9节 原型模式

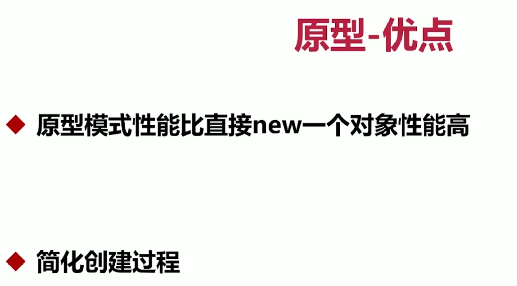
### 9.1定义与类型

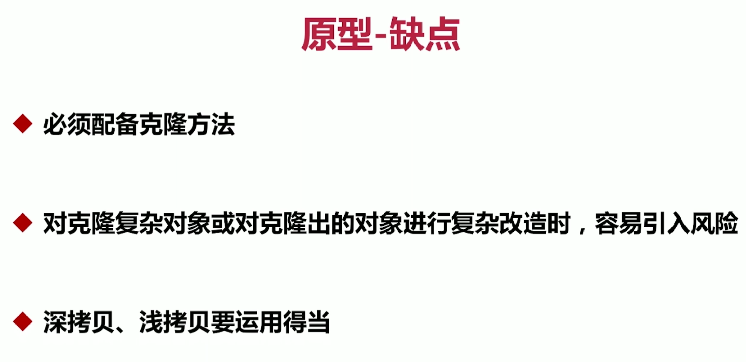


### 9.2适用场景



### 9.3优缺点







### 9.4Coding

|  |
| --- |
| **public class** Mail **implements** Cloneable{  **private** String **name**;  **private** String **emailAddress**;  **private** String **content**;  **public** Mail(){  System.***out***.println(**"Mail Class Constructor"**);  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** String getEmailAddress() {  **return emailAddress**;  }   **public void** setEmailAddress(String emailAddress) {  **this**.**emailAddress** = emailAddress;  }   **public** String getContent() {  **return content**;  }   **public void** setContent(String content) {  **this**.**content** = content;  }   @Override  **public** String toString() {  **return "Mail{"** +  **"name='"** + **name** + **'\''** +  **", emailAddress='"** + **emailAddress** + **'\''** +  **", content='"** + **content** + **'\''** +  **'}'**+**super**.toString();  }   @Override  **protected** Object clone() **throws** CloneNotSupportedException {  System.***out***.println(**"clone mail object"**);  **return super**.clone();  } } |
| **public class** MailUtil {  **public static void** sendMail(Mail mail){  String outputContent = **"向{0}同学,邮件地址:{1},邮件内容:{2}发送邮件成功"**;  System.***out***.println(MessageFormat.*format*(outputContent,mail.getName(),mail.getEmailAddress(),mail.getContent()));  }  **public static void** saveOriginMailRecord(Mail mail){  System.***out***.println(**"存储originMail记录,originMail:"**+mail.getContent());  } } |
| **public class** Test {  **public static void** main(String[] args) **throws** CloneNotSupportedException {  Mail mail = **new** Mail();  mail.setContent(**"初始化模板"**);  System.***out***.println(**"初始化mail:"**+mail);  **for**(**int** i = 0;i < 10;i++){  Mail mailTemp = (Mail) mail.clone();  mailTemp.setName(**"姓名"**+i);  mailTemp.setEmailAddress(**"姓名"**+i+**"@163.com"**);  mailTemp.setContent(**"恭喜您，此次活动中奖了"**);  MailUtil.*sendMail*(mailTemp);  System.***out***.println(**"克隆的mailTemp:"**+mailTemp);  }  MailUtil.*saveOriginMailRecord*(mail);  } } |
| **Mail Class Constructor （构造函数）**  **初始化mail:Mail{name='null', emailAddress='null', content='初始化模板'}com.geely.design.pattern.creational.prototype.Mail@34a245ab**  **clone mail object （克隆时并不调用构造函数）**  **向姓名0同学,邮件地址:姓名0@163.com,邮件内容:恭喜您，此次活动中奖了发送邮件成功**  **克隆的mailTemp:Mail{name='姓名0', emailAddress='姓名0@163.com', content='恭喜您，此次活动中奖了'}com.geely.design.pattern.creational.prototype.Mail@7cc355be**  **clone mail object**  **向姓名1同学,邮件地址:姓名1@163.com,邮件内容:恭喜您，此次活动中奖了发送邮件成功**  **克隆的mailTemp:Mail{name='姓名1', emailAddress='姓名1@163.com', content='恭喜您，此次活动中奖了'}com.geely.design.pattern.creational.prototype.Mail@6e8cf4c6**  **clone mail object**  **向姓名2同学,邮件地址:姓名2@163.com,邮件内容:恭喜您，此次活动中奖了发送邮件成功**  **克隆的mailTemp:Mail{name='姓名2', emailAddress='姓名2@163.com', content='恭喜您，此次活动中奖了'}com.geely.design.pattern.creational.prototype.Mail@12edcd21**  **存储originMail记录,originMail:初始化模板 （模板并未发生变化）** |

注意浅拷贝引入的坑：

|  |
| --- |
| **public class** Pig **implements** Cloneable{  **private** String **name**;  **private** Date **birthday**;   **public** Pig(String name, Date birthday) {  **this**.**name** = name;  **this**.**birthday** = birthday;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   @Override  **protected** Object clone() **throws** CloneNotSupportedException {  Pig pig = (Pig)**super**.clone();   *//深克隆* pig.**birthday** = (Date) pig.**birthday**.clone();  **return** pig;  }   @Override  **public** String toString() {  **return "Pig{"** +  **"name='"** + **name** + **'\''** +  **", birthday="** + **birthday** +  **'}'**+**super**.toString();  } } |
| **public class** Test {  **public static void** main(String[] args) **throws** CloneNotSupportedException, NoSuchMethodException, InvocationTargetException, IllegalAccessException {  Date birthday = **new** Date(0L);  Pig pig1 = **new** Pig(**"佩奇"**,birthday);  Pig pig2 = (Pig) pig1.clone();  System.***out***.println(pig1);  System.***out***.println(pig2);   pig1.getBirthday().setTime(666666666666L);   System.***out***.println(pig1);  System.***out***.println(pig2);  } } |
| **Pig{name='佩奇', birthday=Thu Jan 01 08:00:00 CST 1970}com.geely.design.pattern.creational.prototype.clone.Pig@7a46a697**  **Pig{name='佩奇', birthday=Thu Jan 01 08:00:00 CST 1970}com.geely.design.pattern.creational.prototype.clone.Pig@5f205aa**  **Pig{name='佩奇', birthday=Sat Feb 16 09:11:06 CST 1991}com.geely.design.pattern.creational.prototype.clone.Pig@7a46a697**  **Pig{name='佩奇', birthday=Thu Jan 01 08:00:00 CST 1970}com.geely.design.pattern.creational.prototype.clone.Pig@5f205aa （如果没有实现拷贝对象中的引用类型对象，那么修改一个值，克隆的对象也将改变，也就是浅拷贝）** |

### 9.5克隆破坏单利

|  |
| --- |
| **public class** HungrySingleton **implements** Serializable,Cloneable{   **private final static** HungrySingleton ***hungrySingleton***;   **static**{  ***hungrySingleton*** = **new** HungrySingleton();  }  **private** HungrySingleton(){  **if**(***hungrySingleton*** != **null**){  **throw new** RuntimeException(**"单例构造器禁止反射调用"**);  }  }  **public static** HungrySingleton getInstance(){  **return *hungrySingleton***;  }   **private** Object readResolve(){  **return *hungrySingleton***;  }   @Override  **protected** Object clone() **throws** CloneNotSupportedException {  // 将此处的return super.clone()修改为getInstance才能保证获取到的是单例  **return** *getInstance*();  } } |
| **public class** Test {  **public static void** main(String[] args) **throws** CloneNotSupportedException, NoSuchMethodException, InvocationTargetException, IllegalAccessException {   HungrySingleton hungrySingleton = HungrySingleton.*getInstance*();  Method method = hungrySingleton.getClass().getDeclaredMethod(**"clone"**);  method.setAccessible(**true**);  HungrySingleton cloneHungrySingleton = (HungrySingleton) method.invoke(hungrySingleton);  System.***out***.println(hungrySingleton);  System.***out***.println(cloneHungrySingleton);  } } |
| com.geely.design.pattern.creational.singleton.HungrySingleton@7cc355be  com.geely.design.pattern.creational.singleton.HungrySingleton@7cc355be |

### 9.6源码解析

ArrayList

|  |
| --- |
| @Override  **public** Object clone() {  **try** {  ArrayList<?> v = (ArrayList<?>) **super**.clone();  v.**elementData** = Arrays.*copyOf*(**elementData**, **size**);  v.**modCount** = 0;  **return** v;  } **catch** (CloneNotSupportedException e) {  *// this shouldn't happen, since we are Cloneable* **throw new** InternalError(e);  } } |

HashMap

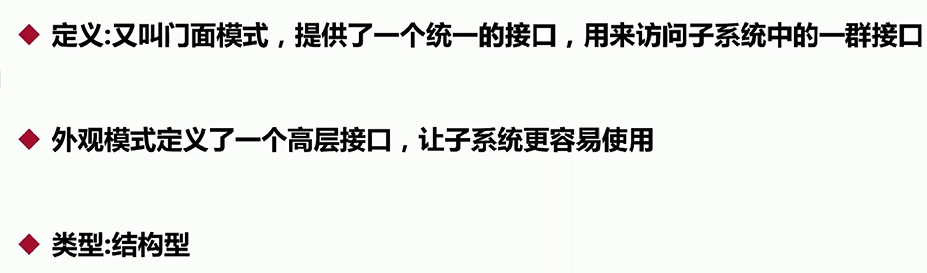
|  |
| --- |
| @Override **public** Object clone() {  HashMap<K,V> result;  **try** {  result = (HashMap<K,V>)**super**.clone();  } **catch** (CloneNotSupportedException e) {  *// this shouldn't happen, since we are Cloneable* **throw new** InternalError(e);  }  result.reinitialize();  result.putMapEntries(**this**, **false**);  **return** result; } |

Ibatis中CacheKey深拷贝

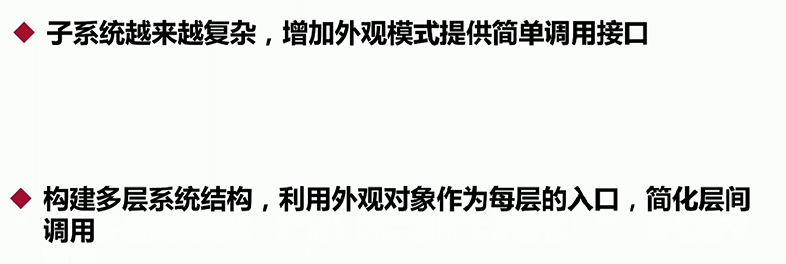
|  |
| --- |
| **public** CacheKey clone() **throws** CloneNotSupportedException {  CacheKey clonedCacheKey = (CacheKey)**super**.clone();  clonedCacheKey.updateList = **new** ArrayList(**this**.updateList);  **return** clonedCacheKey; } |

## 第10节 外观模式（门面模式）

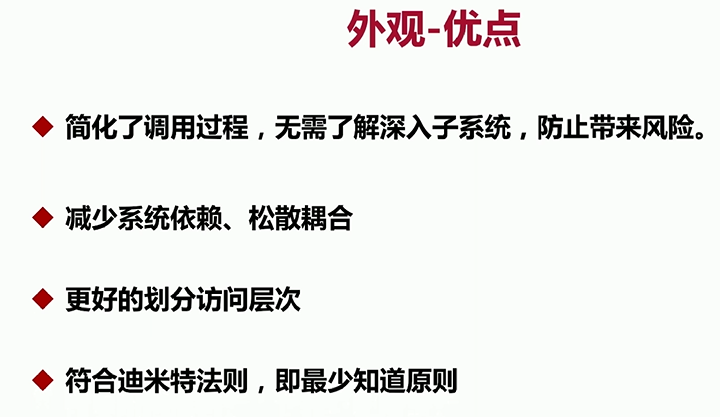
### 10.1定义与类型

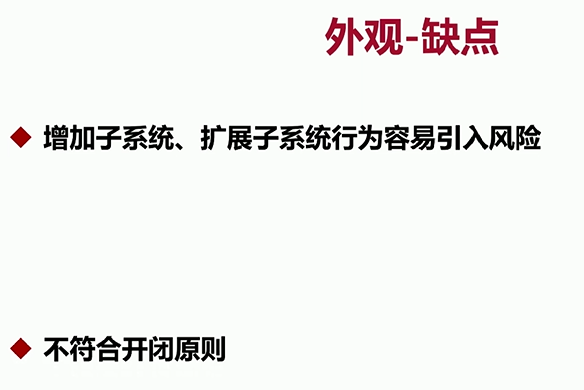


### 10.2适用场景



### 10.3优缺点







外观模式关注外界与子系统的交互；

中介者模式关注子系统内部之间的交互；



把外观模式中的外观对象做成单例模式的结合使用；



外观类可以通过外观工厂获取子系统的实例，这样子系统可以对外观类进行屏蔽；



### 10.4Coding

积分系统与子系统（积分检验、积分支付、物流系统）的交互，将子系统与客户端解耦，Client只需要与积分交易系统交互，无需知道子系统的任何接口。

Bean：礼物

|  |
| --- |
| **public class** PointGift {  **private** String **giftName**;   **public** PointGift(String giftName) {  **this**.**giftName** = giftName;  }   **public** String getGiftName() {  **return giftName**;  }   **public void** setGiftName(String giftName) {  **this**.**giftName** = giftName;  } } |

子系统1：积分校验服务

|  |
| --- |
| **public class** QualifyService {  **public boolean** checkQualified(PointGift gift){  *// 查看积分，库存是否满足* System.***out***.println(gift.getGiftName() + **" 积分足够，库存足够"**);  **return true**;  } } |

子系统2：积分支付服务

|  |
| --- |
| **public class** PayService {  **public boolean** pay(PointGift gift){  *// 支付费用* System.***out***.println(gift.getGiftName() + **" 支付成功"**);  **return true**;  } } |

子系统3：物流系统服务

|  |
| --- |
| **public class** ShipGift {  **public** String ship(PointGift gift){  *// 发送物流，产生订单号* String shipNumber = **"666"**;  System.***out***.println(gift.getGiftName() + **"物流发送成功，订单号为"** + shipNumber);  **return** shipNumber;  } } |

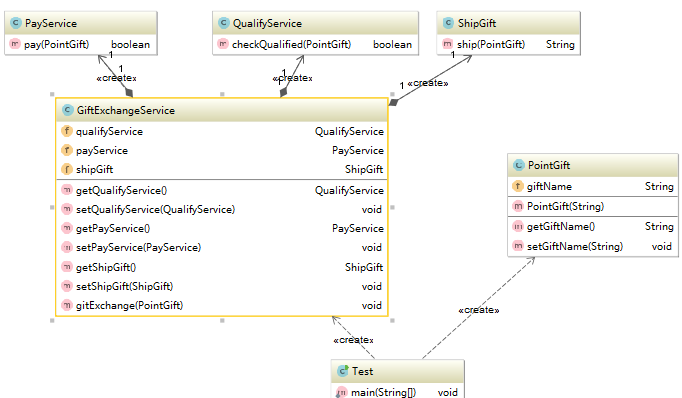
门面系统：积分兑换系统

|  |
| --- |
| **public class** GiftExchangeService {  **private** QualifyService **qualifyService** = **new** QualifyService();  **private** PayService **payService** = **new** PayService();  **private** ShipGift **shipGift** = **new** ShipGift();   **public** QualifyService getQualifyService() {  **return qualifyService**;  }   **public void** setQualifyService(QualifyService qualifyService) {  **this**.**qualifyService** = qualifyService;  }   **public** PayService getPayService() {  **return payService**;  }   **public void** setPayService(PayService payService) {  **this**.**payService** = payService;  }   **public** ShipGift getShipGift() {  **return shipGift**;  }   **public void** setShipGift(ShipGift shipGift) {  **this**.**shipGift** = shipGift;  }   **public void** gitExchange(PointGift gift){  **if** (**qualifyService**.checkQualified(gift)){  **if** (**payService**.pay(gift)){  System.***out***.println(**"订单下发成功"**);  System.***out***.println(**"订单号为："** + **shipGift**.ship(gift));  }  }  } } |

客户端

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  PointGift gift = **new** PointGift(**"T恤"**);  GiftExchangeService giftExchangeService = **new** GiftExchangeService();  giftExchangeService.gitExchange(gift);  } } |
| **T恤 积分足够，库存足够**  **T恤 支付成功**  **订单下发成功**  **T恤物流发送成功，订单号为666**  **订单号为：666** |

### 10.5UML类图



### 10.6源码解析

1. JdbcUtils对Jdbc的封装；
2. Mybatis的Configuration类封装客户端需要处理的类实例；

|  |
| --- |
| **public** MetaObject newMetaObject(Object object) {  **return** MetaObject.*forObject*(object, **objectFactory**, **objectWrapperFactory**, **reflectorFactory**); } |

逻辑修改只需要修改方法即可，调用者无需关注具体方法的实现。

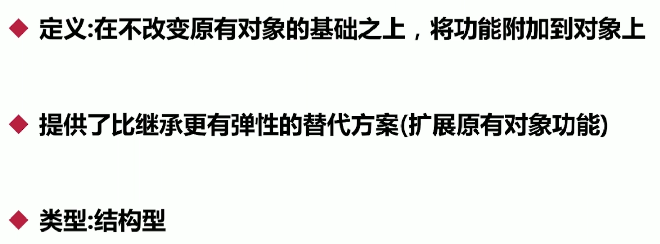
1. Tomcat中HttpServletRequest的获取采用RequestFacade进行包装HttpRequest中的方法，提供门面模式。

org.apache.catalina.connector.Request

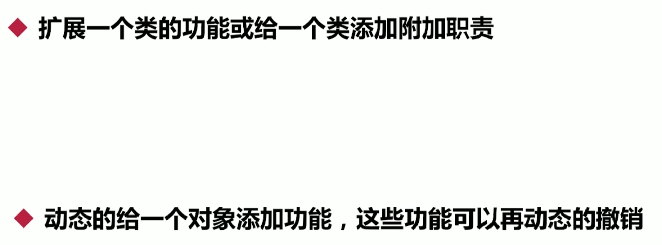
|  |
| --- |
| **public** HttpServletRequest getRequest() {  **if** (facade == **null**) {  facade = **new** RequestFacade(**this**);  }  **if** (applicationRequest == **null**) {  applicationRequest = facade;  }  **return** applicationRequest; } |

## 第11节 装饰器模式

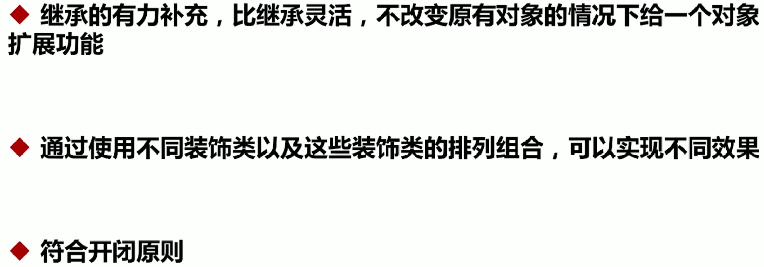
### 11.1定义与类型

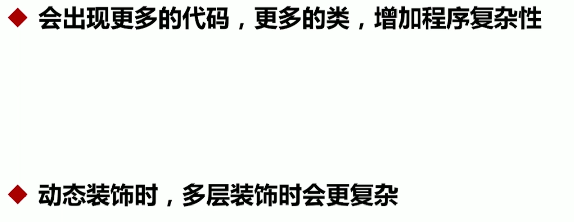


### 11.2适用场景



### 11.3优缺点





### 11.4相关设计模式



前者关注在一个对象上动态添加方法，代理模式关注对对象的访问，代理模式的代理类可以对他的客户隐藏一个对象的具体信息，常常在使用代理模式的时候，在代理类中创建一个对象的实例，而在适用装饰者模式时，通常将原始对象作为一个参数传递给装饰者的构造器。



都可以叫做包装模式：wrapper

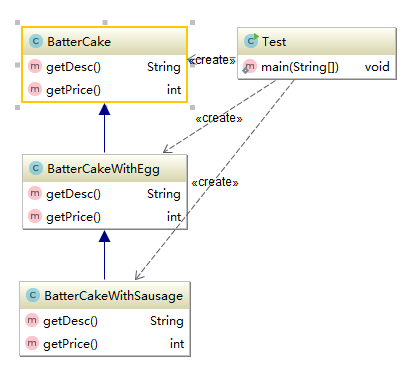
装饰者和被装饰者可以实现相同的接口，或者装饰者是被装饰者的子类；在适配器模式中，适配器和被适配的类具有不同的接口，也有可能有部分接口是重合的。

半装饰者模式：一个装饰者除了提供被装饰者接口外，还提供其他接口，变成半透明的装饰者。

### 11.5Coding

不适用装饰器模式：

|  |
| --- |
| **public class** BatterCake {  **protected** String getDesc(){  **return "煎饼"**;  }  **protected int** getPrice(){  **return** 8;  } } |
| **public class** BatterCakeWithEgg **extends** BatterCake {  @Override  **protected** String getDesc() {  **return super**.getDesc() + **" 加一个鸡蛋 "**;  }   @Override  **protected int** getPrice() {  **return super**.getPrice() + 1;  } } |
| **public class** BatterCakeWithSausage **extends** BatterCakeWithEgg {  @Override  **protected** String getDesc() {  **return super**.getDesc() + **" 加一根香肠 "**;  }   @Override  **protected int** getPrice() {  **return super**.getPrice() + 2;  } } |
| **public class** Test {  **public static void** main(String[] args) {  BatterCake batterCake = **new** BatterCake();  System.***out***.println(batterCake.getDesc() + **"售价："** + batterCake.getPrice());   BatterCake batterCake1 = **new** BatterCakeWithEgg();  System.***out***.println(batterCake1.getDesc() + **"售价："** + batterCake1.getPrice());   BatterCake batterCake2 = **new** BatterCakeWithSausage();  System.***out***.println(batterCake2.getDesc() + **"售价："** + batterCake2.getPrice());   } } |
| **煎饼售价：8**  **煎饼 加一个鸡蛋 售价：9**  **煎饼 加一个鸡蛋 加一根香肠 售价：11** |



不同场景需要实现不同的子类，例如客户需要一个假两个鸡蛋的煎饼，需要重新实现。

使用装饰器模式实现卖煎饼的场景：

抽象煎饼

|  |
| --- |
| **public abstract class** ABatterCake {  **protected abstract** String getDesc();  **protected abstract int** getPrice(); } |

具体煎饼

|  |
| --- |
| **public class** BatterCake **extends** ABatterCake {  @Override  **protected** String getDesc() {  **return "煎饼 "**;  }   @Override  **protected int** getPrice() {  **return** 8;  } } |

抽象装饰器，通过构造函数传入待包装的煎饼。

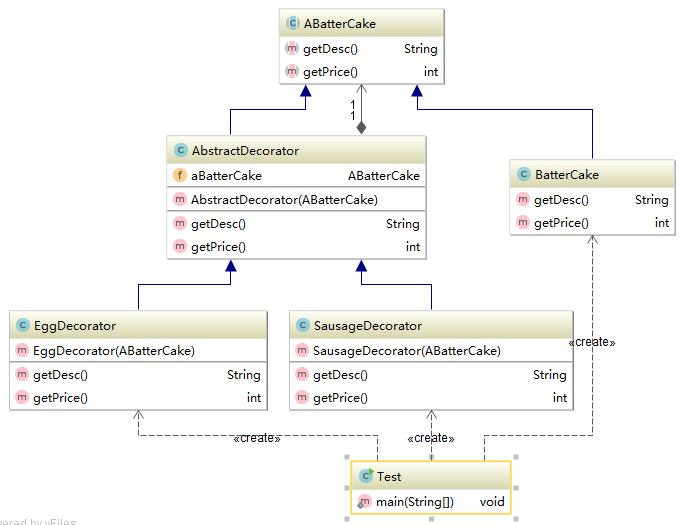
|  |
| --- |
| **public class** AbstractDecorator **extends** ABatterCake {  ABatterCake **aBatterCake**;  **public** AbstractDecorator(ABatterCake aBatterCake){  **this**.**aBatterCake** = aBatterCake;  }  @Override  **protected** String getDesc() {  **return this**.**aBatterCake**.getDesc();  }   @Override  **protected int** getPrice() {  **return this**.**aBatterCake**.getPrice();  } } |

具体装饰器，在父类的实体方法上进行包装。

|  |
| --- |
| **public class** EggDecorator **extends** AbstractDecorator {   **public** EggDecorator(ABatterCake batterCake){  **super**(batterCake);  }   @Override  **protected** String getDesc() {  **return super**.getDesc() + **" 加一个鸡蛋 "**;  }   @Override  **protected int** getPrice() {  **return super**.getPrice() + 1;  } } |
| **public class** SausageDecorator **extends** AbstractDecorator {  **public** SausageDecorator(ABatterCake aBatterCake) {  **super**(aBatterCake);  }   @Override  **protected** String getDesc() {  **return super**.getDesc() + **" 加一根香肠 "**;  }   @Override  **protected int** getPrice() {  **return super**.getPrice() + 2;  } } |

客户端

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  ABatterCake aBatterCake;  aBatterCake = **new** BatterCake();  System.***out***.println(aBatterCake.getDesc() + **" 售价： "** + aBatterCake.getPrice());  aBatterCake = **new** EggDecorator(aBatterCake);  System.***out***.println(aBatterCake.getDesc() + **" 售价： "** + aBatterCake.getPrice());  aBatterCake = **new** EggDecorator(aBatterCake);  System.***out***.println(aBatterCake.getDesc() + **" 售价： "** + aBatterCake.getPrice());  aBatterCake = **new** SausageDecorator(aBatterCake);  System.***out***.println(aBatterCake.getDesc() + **" 售价： "** + aBatterCake.getPrice());  } } |
| **煎饼 售价： 8**  **煎饼 加一个鸡蛋 售价： 9**  **煎饼 加一个鸡蛋 加一个鸡蛋 售价： 10**  **煎饼 加一个鸡蛋 加一个鸡蛋 加一根香肠 售价： 12** |



装饰器关键元素：

1. 抽象物品；
2. 实体物品；
3. 抽象装饰器（继承抽象物品，因为装饰完成后也是该类物品，在构造函数传入实体物品构造装饰后的对象）；
4. 具体装饰器（继承抽象装饰器，在抽象装饰器上变更抽象物品的方法）；

### 11.6源码解析

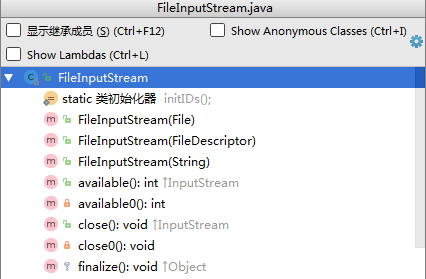
1. JDK中的装饰器模式：

BufferedReader继承自抽象类Reader，在构造函数中传入Reader实体对象，包装Reader为BufferdReader。

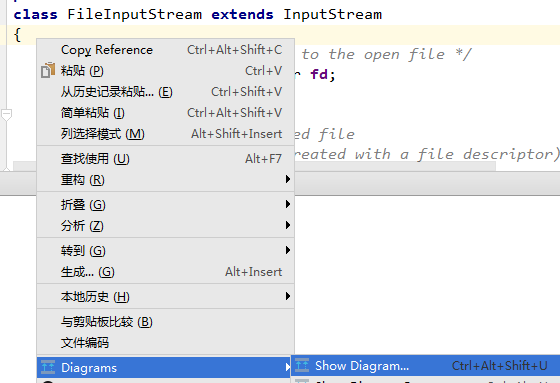
|  |
| --- |
| **public** BufferedReader(Reader in, **int** sz) {  **super**(in);  **if** (sz <= 0)  **throw new** IllegalArgumentException(**"Buffer size <= 0"**);  **this**.**in** = in;  **cb** = **new char**[sz];  **nextChar** = **nChars** = 0; } |

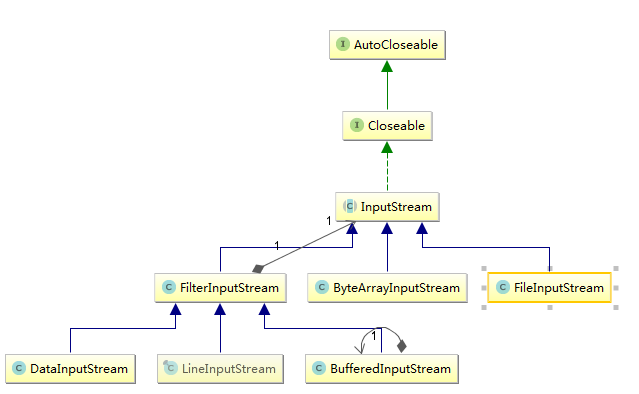
1. FilterInputStream继承自抽象类InputStream，在构造函数中传入InputStream，

Ctrl+F12查看文件结构



查看类的UML类图





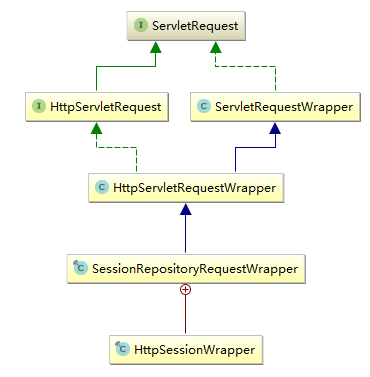
抽象被装饰者InputStream，具体被装饰者FileInputStream；抽象装饰者FilterInputStream；具体装饰者BufferedInputStream

2.Spring Session中的装饰器模式：

TransactionAwareCacheDecorator

|  |
| --- |
| **public class** TransactionAwareCacheDecorator **implements** Cache {  **private final** Cache targetCache;   **public** TransactionAwareCacheDecorator(Cache targetCache) {  Assert.notNull(targetCache, **"Target Cache must not be null"**);  **this**.targetCache = targetCache;  }  …… |

SessionRepositoryRequestWrapper

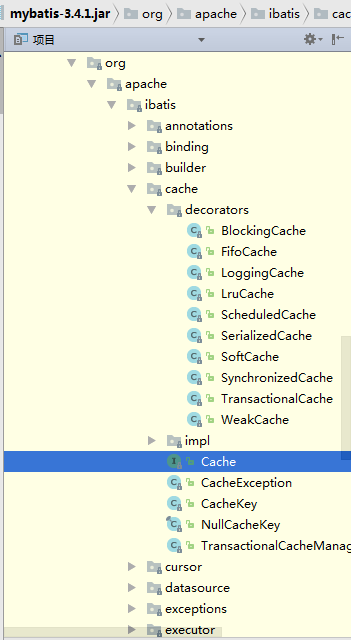


抽象被包装类ServletRequest，具体被包装类HttpServletRequest；抽象包装器ServletRequestWrapper，具体包装器HttpServletRequestWrapper。

注意：

Wrapper的实现有可能是适配器模式，也可能是装饰器模式，都是包装器Wrapper

3.Mybatis Cache中的装饰器模式



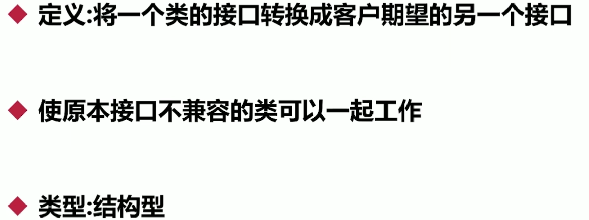
|  |
| --- |
| **public class** FifoCache **implements** Cache {   **private final** Cache **delegate**;  **private** Deque<Object> **keyList**;  **private int size**;   **public** FifoCache(Cache delegate) {  **this**.**delegate** = delegate;  **this**.**keyList** = **new** LinkedList<Object>();  **this**.**size** = 1024;  }  …… |

|  |
| --- |
| **public class** TransactionalCache **implements** Cache {   **private static final** Log ***log*** = LogFactory.*getLog*(TransactionalCache.**class**);   **private** Cache **delegate**;  **private boolean clearOnCommit**;  **private** Map<Object, Object> **entriesToAddOnCommit**;  **private** Set<Object> **entriesMissedInCache**;   **public** TransactionalCache(Cache delegate) {  **this**.**delegate** = delegate;  **this**.**clearOnCommit** = **false**;  **this**.**entriesToAddOnCommit** = **new** HashMap<Object, Object>();  **this**.**entriesMissedInCache** = **new** HashSet<Object>();  }  …… |

抽象被装饰类Cache，具体被装饰类的实现类CacheKey；抽象装饰器\*\*Decorator。

## 适配器模式

### 12.1定义与类型



### 12.2适用场景





适配器模式应用场景：

类适配器与对象适配器的使用场景一致，仅仅是实现手段稍有区别，二者主要用于如下场景：

　　（1）想要使用一个已经存在的类，但是它却不符合现有的接口规范，导致无法直接去访问，这时创建一个适配器就能间接去访问这个类中的方法。

　　（2）我们有一个类，想将其设计为可重用的类（可被多处访问），我们可以创建适配器来将这个类来适配其他没有提供合适接口的类。

　　以上两个场景其实就是从两个角度来描述一类问题，那就是要访问的方法不在合适的接口里，一个从接口出发（被访问），一个从访问出发（主动访问）。

接口适配器使用场景：

　　（1）想要使用接口中的某个或某些方法，但是接口中有太多方法，我们要使用时必须实现接口并实现其中的所有方法，可以使用抽象类来实现接口，并不对方法进行实现（仅置空），然后我们再继承这个抽象类来通过重写想用的方法的方式来实现。这个抽象类就是适配器。

### 12.3优缺点











扩展



### 12.4相关设计模式

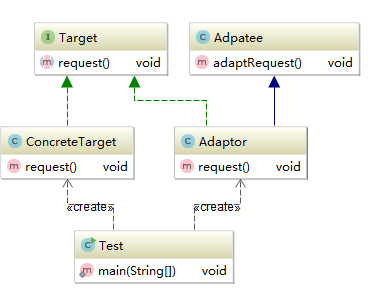


都是对现有的类或系统的封装，外观模式定义了新的接口，适配器则是复用原有的接口，适配器是使两个已有的接口能够协同工作，而外观模式是在现有的系统中提供更为方便的系统入口。

如果强行的把外观模式称之为适配的话，这两个最大的区别就在于适配的力度不同，那外观模式是用来适配整个子系统（相关的子系统），外观针对的对象力度更大。

### 12.5Coding

类适配器（适配器继承被适配的对象）：



被适配类

|  |
| --- |
| **public class** Adpatee {  **public void** adaptRequest(){  System.***out***.println(**"被适配类的方法"**);  } } |

适配目标

|  |
| --- |
| **public interface** Target {  **void** request(); } |
| **public class** ConcreteTarget **implements** Target{   @Override  **public void** request() {  System.***out***.println(**"ConcreteTarget目标对象的方法"**);  } } |

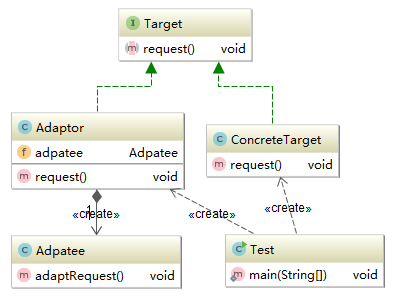
适配器

|  |
| --- |
| **public class** Adaptor **extends** Adpatee **implements** Target {   @Override  **public void** request() {  **super**.adaptRequest();  } } |

测试适配接口

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  Target target = **new** ConcreteTarget();  target.request();   Target adaptorTarget = **new** Adaptor();  adaptorTarget.request();  } } |
| **ConcreteTarget目标对象的方法**  **被适配类的方法** |

对象适配器：



只需要将Adaptor适配器修改为不要继承被适配者，而是直接实现Target接口（内部注入被适配类的实例），在实现的接口方法中适用被适配者实例的方法。

|  |
| --- |
| **public class** Adaptor **implements** Target {  Adpatee **adpatee** = **new** Adpatee();  @Override  **public void** request() {  **adpatee**.adaptRequest();  } } |

适配器的生活场景，变压器直流电220V变为5V的直流电。

|  |
| --- |
| **public class** Adaptee220V {  **public int** inputAC220V(){  **int** output = 220;  System.***out***.println(**"输入直流电"** + output +**"V"**);  **return** output;  } } |

|  |
| --- |
| **public interface** DC5V {  **int** outputDC5V(); } |

|  |
| --- |
| **public class** PowerAdaptor **implements** DC5V{  Adaptee220V **adaptee220V** = **new** Adaptee220V();   @Override  **public int** outputDC5V(){  **int** input = **adaptee220V**.inputAC220V();  **int** output = input / 44;  System.***out***.println(**"适用PowerAdaptor输入交流电"** + input + **"V"** + **" 输出直流电"** + output +**"V"**);  **return** output;  } } |

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  DC5V dc5V = **new** PowerAdaptor();  dc5V.outputDC5V();  } } |

### 12.6源码解析

1. JDK中的XmlAdaptor

|  |
| --- |
| @XmlJavaTypeAdapter(value=DateAdapter.class)  @XmlElement  private Date birthDay; |
| @XmlJavaTypeAdapter(value=MapAdapter.class)  // @XmlElementWrapper(name="role")  @XmlElement  private Map map; |
| public class MapAdapter extends XmlAdapter<MapAdapter.AdapterMap,Map<String,Object>> {  public static class AdapterMap {  public List<Entry> entry =new ArrayList<Entry>();  }  /\*\*  \* 功能:xml->Map<String,Object>  \* @param adapterMap  \* @return  \* @throws Exception  \*/  @Override  public Map<String, Object> unmarshal(AdapterMap adapterMap) throws Exception {  Map<String, Object> map = new HashMap<String, Object>();  List entry = adapterMap.entry;  for(Entry en:adapterMap.entry){  map.put(en.getKey(), en.getValue());  }  return map;  }  /\*\*  \* 功能:map<String,Object> -> xml  \*/  @Override  public AdapterMap marshal(Map<String, Object> map) throws Exception {  AdapterMap adapterMap = new AdapterMap();  for (Map.Entry<String, Object> entry : map.entrySet()) {  Entry entry2 = new Entry();  entry2.setKey(entry.getKey());  entry2.setValue(entry.getValue());  adapterMap.entry.add(entry2);  }  return adapterMap;  }  } |
| public class DateAdapter extends XmlAdapter<String, Date> {  //反序列化成日期对象Date  @Override  public Date unmarshal(String str) throws Exception {  SimpleDateFormat format = getSimpleDateFormat("yyyy-MM-dd HH:mm:ss");  return str==null ? null:format.parse(str);  }  //序列化成xmL  @Override  public String marshal(Date date) throws Exception {  SimpleDateFormat format = getSimpleDateFormat("yyyy-MM-dd HH:mm:ss");  return date==null ? "":format.format(date);  }  private SimpleDateFormat getSimpleDateFormat(String pattern){  SimpleDateFormat format = new SimpleDateFormat(pattern);  return format;  }  } |

2.Spring中的适配器AdvisorAdapter（Ctrl + Alt + B查找实现）

|  |
| --- |
| **public interface** AdvisorAdapter {**boolean** supportsAdvice(Advice advice);MethodInterceptor getInterceptor(Advisor advisor);  } |
| **class** MethodBeforeAdviceAdapter **implements** AdvisorAdapter, Serializable {  MethodBeforeAdviceAdapter() {  }   **public boolean** supportsAdvice(Advice advice) {  **return** advice **instanceof** MethodBeforeAdvice;  }   **public** MethodInterceptor getInterceptor(Advisor advisor) {  MethodBeforeAdvice advice = (MethodBeforeAdvice)advisor.getAdvice();  **return new** MethodBeforeAdviceInterceptor(advice);  } } |

3.Spring-data-jpa中的JpaVendorAdapter

|  |
| --- |
| **public interface** JpaVendorAdapter {  PersistenceProvider getPersistenceProvider();   @Nullable  **default** String getPersistenceProviderRootPackage() {  **return null**;  }   **default** Map<String, ?> getJpaPropertyMap(PersistenceUnitInfo pui) {  **return this**.getJpaPropertyMap();  }   **default** Map<String, ?> getJpaPropertyMap() {  **return** Collections.emptyMap();  }   @Nullable  **default** JpaDialect getJpaDialect() {  **return null**;  }   **default** Class<? **extends** EntityManagerFactory> getEntityManagerFactoryInterface() {  **return** EntityManagerFactory.**class**;  }   **default** Class<? **extends** EntityManager> getEntityManagerInterface() {  **return** EntityManager.**class**;  }   **default void** postProcessEntityManagerFactory(EntityManagerFactory emf) {  } } |
|  |

4.SpringMVC中的HandlerAdapter

|  |
| --- |
| **public interface** HandlerAdapter {  **boolean** supports(Object var1);   ModelAndView handle(HttpServletRequest var1, HttpServletResponse var2, Object var3) **throws** Exception;   **long** getLastModified(HttpServletRequest var1, Object var2); } |

DispatcherServlet（上帝视角/客户端/Test）获取Handler的方式，便利所有的Handler直到找到支持的Handler

|  |
| --- |
| **protected** HandlerAdapter getHandlerAdapter(Object handler) **throws** ServletException {  Iterator var2 = **this**.handlerAdapters.iterator();   HandlerAdapter ha;  **do** {  **if**(!var2.hasNext()) {  **throw new** ServletException(**"No adapter for handler ["** + handler + **"]: The DispatcherServlet configuration needs to include a HandlerAdapter that supports this handler"**);  }   ha = (HandlerAdapter)var2.next();  **if**(**this**.logger.isTraceEnabled()) {  **this**.logger.trace(**"Testing handler adapter ["** + ha + **"]"**);  }  } **while**(!ha.supports(handler));   **return** ha; } |

被适配者Controller，如需扩展Controller，则实现不同的Controller的Adaptor即可

|  |
| --- |
|  |