1.基础应用加强

1.1. 泛型基础加强

1. 如何理解泛型?
   1) 参数化类型,是 JDK1.5 的新特性。(定义泛型时使用参数可以简单理解为形参)
   2) 编译时的一种类型，此类型仅仅在编译阶段有效，运行时无效。
   例如:

```
public interface List<E> extends Collection<E> {
    // Query Operations
Iterator<E> iterator();
…
}
```

2. 为何使用泛型?
   3) 提高编程时灵活性。
   4) 提高程序运行时的性能。(在编译阶段解决一些运行时需要关注的问题，例如强转)

3. 泛型的应用类型?
   5) 泛型类: class 类名<泛型>{}
   6) 泛型接口: interface 接口名<泛型>{}
   7) 泛型方法: 访问修饰符 <泛型> 方法返回值类型 方法名(形参){}

例如:

泛型类应用

```
public class PageObject<T> implements Serializable{
    private static final long serialVersionUID = -7368493786259905794L;
    /**总记录数*/
    private Integer rowCount=0;//null
    /**页面大小*/
    private Integer pageSize=3;
    /**当前页码值*/
    private Integer pageCurrent=1;
    /**总页数*/
```

```
    private Integer pageCount=0;
    /**当前页记录*/
    private List<T> records;
    …
```

泛型方法

```
public class PageUtil {
     public static <T>PageObject<T> newPageObject(
                 int rowCount,
                 List<T> records,
                 int pageCurrent,
                 int pageSize){
        PageObject<T> po=new PageObject<>();
        po.setRowCount(rowCount);
        po.setRecords(records);
        po.setPageSize(pageSize);
        po.setPageCurrent(pageCurrent);
        po.setPageCount((rowCount-1)/pageSize+1);
        return po;
      }
}
```

4. 泛型的通配符？（这里的通配符可以看成是一种不确定的类型）
   1）泛型应用时有一个特殊符号”?”,可以代表一种任意参数类型，注释是实参。
   2）通配符泛型只能应用于变量的定义。

```
    Class<?> c1=Class.forName("pkg.Point");
```

5. 泛型的上下界问题？
   1）指定泛型下界：<? super 类型>
   2）指定泛型上界：<? extends 类型>

   例如：

```
    List<? extends CharSequence> list1=new ArrayList<String>();
    List<? super Integer> list2=new ArrayList<Number>();
```

   说明：这种上下界一般会用于方法参数变量定义，方法返回值类型定义。

6. 泛型类型擦除？
   泛型是编译时的一种类型，在运行时无效，运行时候都会变成 Object 类型。

   案例：运行时向 List<String>集合中添加一个整数 100.

1.2. 序列化基础加强

1. 何为序列化&反序列化?
   1) 序列化：将对象转换为字节的过程。
   2) 反序列化：将字节转换为对象的过程。

2. 序列化的应用场景?
   1) 网络通讯
   2) 数据存储(例如文件，缓存)

3. Java 中的对象的序列化与反序列化?
   1) 对象要实现 Serializable 接口
   2) 添加序列化 id（为反序列化提供保障)
   3) 借助流对象实现序列化和反序列化?

4. Java 中的序列化存在安全问题如何解决?

   1) 添加 writeObject(ObjectOutpuStream out)方法
      对内容进行加密再执行序列化。
   1) 添加 readObject(ObjectInputStream in)方法对
      内容先进行反序列化然后在执行解密操作

5. Java 中序列化的粒度如何控制?

   1) Transient（当少量属性不需要序列化时，使用此关键字修饰)
   2) Externalizable（当只有少量属性需要序列化时实现此接口然后自己进行序列化操作，但是要序列化的对象必须时 public 修饰。）

7. Java 中序列化的性能问题及如何优化?

1.3. 注解应用基础加强

1. 如何理解注解(Annotation)?
   1) JDK1.5 推出的一种新的应用类型（特殊的 class)
   2) 元数据(Meta Data):一种描述性类型，例如@Override

2. 注解(Annotation)应用场景?

   1) 描述类及其成员(属性，方法)：例如@Overide

   2) 替换项目中 xml 方式对相关对象的描述(例如@Service,@Controller,...)

   例如 spring 框架中相关注解?

   1)@Configuration（描述配置类对象）

   2)@Service(描述业务层对象)

   3)@Controller(描述控制层对象)

   4)@Responsitory(描述数据层对象)

   5)@RestController(描述控制层对象)

   6)@RestControllerAdvice（描述控制全局异常处理）

   7)@Bean(描述 bean 对象，一般修饰方法将返回交给 spring 管理)

   8)@Autowired（实现 bean 对象的自动装配）

   9)…

3. 注解(Annotation)的定义?

   可以借助@interface 关键字进行定义，例如 Override 注解的应用

```
@Target(value=METHOD)
@Retention(value=SOURCE)
public @interface Override{}
```

   其中:

   1)@Target 用于描述定义的注解能够修饰的对象。

   2)@Retention 用于描述定义的注解何时有效。

4. 注解常用的生效范围?

   1) 编译时有效（例如@Retention(RetentionPolicy.SOURCE)）

   2) 运行时有效（例如@Retention(RetentionPolicy.RUNTIME)

   说明：我们自己定义的注解，包括框架中的很多注解基本都是运行时有效。

5. 注解应用案例分析实现?

   1) 与编译器结合实用(@Override)

   2) 与反射 API 结合使用(@RequiredLog,@Transaction,…)

   例如:

   1) 通过反射或类上的注解

   2) 通过反射获取属性或方法上的注解。

1.4. 反射应用基础加强

1．如何理解反射?

1）Java 中特有一种技术

2）JAVA 中自省特性的一种实现?(对象运行时动态发现对象成员)

3）是实现 JAVA 动态编程的基石?(例如 AOP,…)

2．反射的应用场景?

1）框架中对象的构建?（例如 mybatis 中的 resultType,resultMap,spring 中的 bean)

2）框架中方法的调用?（例如对象 set 方法，get 方法，spring mvc 控制层方法，..)

总之：反射不能预知未来，但可驾驭未来，通过反射可以更好构建一些编程框架，以实现通用性编程，从而达到简化代码编写。

3．反射的应用起点?

起点可以理解为反射应用的入口，在 java 中这个入口是字节码对象(Class 对象)。其获取方式如下：

1）类名.class

2）Class.forName(“包名.类名”)；最常用

3）实例对象.getClass()；（获取已经存在的类对象）

说明：任意的一个类在同一个 JVM 内部,类对象是唯一的,此类对象会在第一次类加载时创建。

4．反射核心 API?

1）Constructor （构造方法对象类型）

2）Field （属性对象类型）

3）Method （方法对象类型）

4）Annotation(注解对象类型)

5）…

5．反射应用案例分析及实现?

1）基于反射构建类的实例：首先要基于类对象获取构造方法对象

2）基于反射获取对象属性，并为属性赋值。

3）基于反射获取对象方法，并执行对象方法。

4）基于反射获取描述对象的注解,并基于注解的含义执行下一步操作。

5）基于反射获取类上的泛型参数？（作业）

2.设计模式加强

2.1. 创建型模式

2.1.1. 简单工厂模式

1）如何理解简单工厂模式?

a） 通过静态方法(相对比较多)或实例方法创建对象

b） 封装对象创建过程（基于条件的不同创建不同的具体产品)

2）简单工厂应用场景分析?

a）JDBC(DriverManager.getConnection(…..))

b）Druid（getConnection())

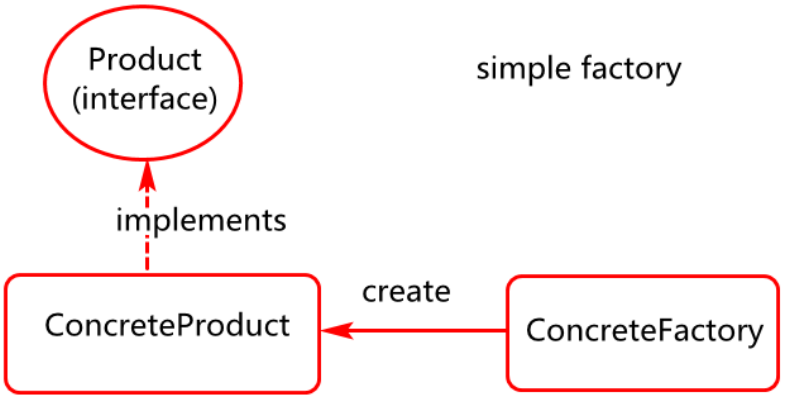
c）Mybatis(Configuration,…)

3）简单工厂中的对象角色?

a）抽象产品对象（例如 Executor,…)

b）具体产品对象（例如 SimpleExecutor,…)

c）具体工厂对象（例如 Configuration,…)



4）简单工厂应用分析?

a）优势：解耦（对象应用者与对象创建之间的耦合），简单

b）劣势：可扩展性相对较差(创建产品的对象的工厂方法不够灵活)

2.1.2. 工厂方法模式（Factory Method）？

1）如何理解工厂方法模式?

a）创建型模式（负责创建对象）

b）工厂模式(平时所说的工厂模式就是工厂方法模式)

c）此模式的特点是基于抽象工厂扩展具体工厂然后创建产品对象。

2）工厂方法模式的应用场景分析?

a）Mybatis (SqlSessionFactoryBean)
b）Shiro (ShiroFilterFactoryBean)
c）Spring(DataSourceFactory,TransactionFactory)
d）…

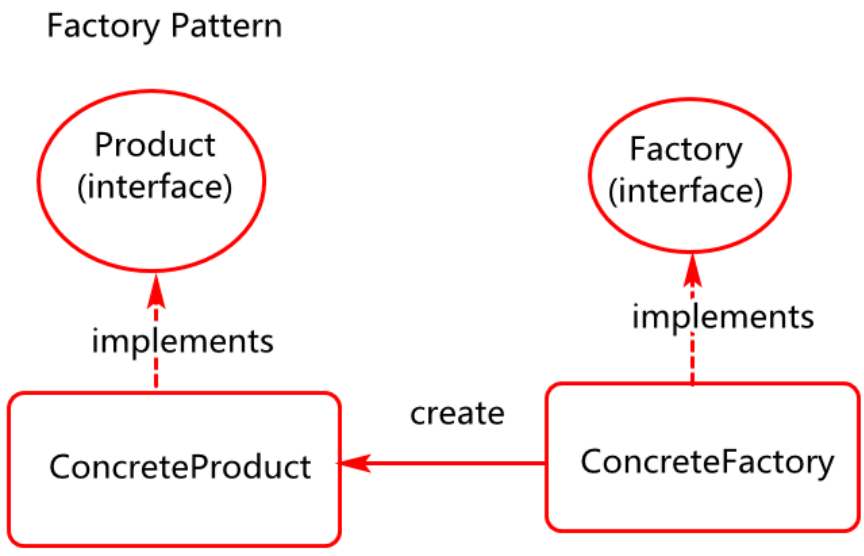
3）工厂方法模式角色分析?

a）抽象产品(Product)

b）具体产品(ConcreteProduct)

c）抽象工厂(Factory)

d）具体工厂(ConcreteFactory)



例如:

- mybatis 中创建 SqlSession 对象

✓ SqlSession(抽象产品)

✓ DefaultSqlSession(具体产品)

✓ SqlSessionFactory(抽象工厂)
✓ DefaultSqlSessionFactory(具体工厂)

- Spring 整合 mybatis 时,SqlSessionFactory 对象过程分析。

✓ SqlSessionFactory （抽象产品）
✓ DefaultSqlSessionFactory(具体产品)
✓ FactoryBean(抽象工厂)
✓ SqlSessionFactoryBean(具体工厂)

4）工厂方法模式应用分析?

a）优势：相对于简单工厂更加灵活,更加适合创建具备等级结构(继承关系)的产品。
b）劣势：假如每个抽象产品都对应一个具体工厂，那么工厂类可能会比较多。

2.1.3. 抽象工厂(Abstract Factory)

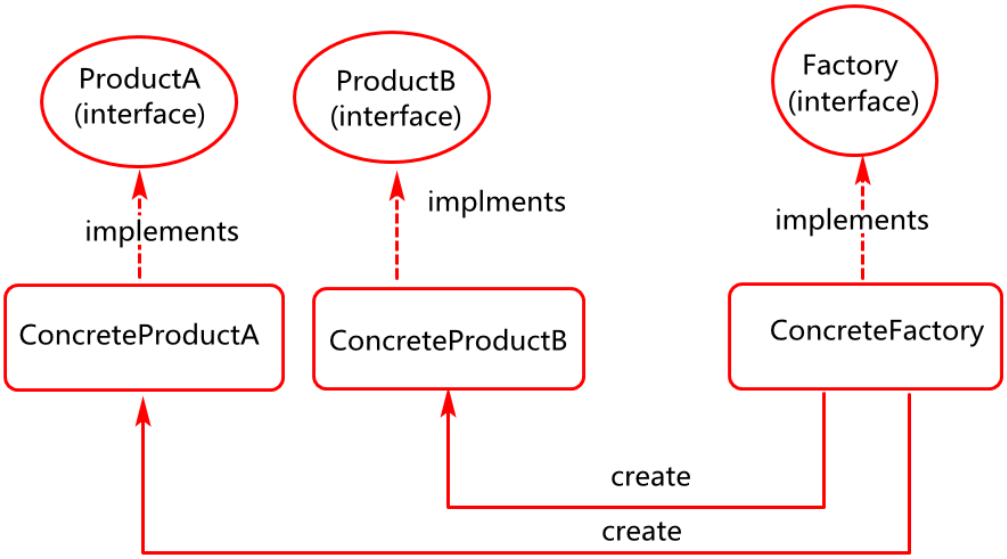
1）如何理解抽象工厂?

a）工厂方法模式用于创建具备一定等级结构的产品。
b）抽象工厂是多个工厂方法的综合应用，因为它要同时创建多个具备一定等级结构的产品，我们可以将这些产品理解产品族。

2）抽象工厂的应用场景?（产品族–>多个抽象产品）
a）Spring(ClientHttpRequestFactory)：了解消息头，消息体对象创建
b）…

3）抽象工厂对象角色分析?

a）抽象产品(Product)
b）具体产品(ConcreteProduct)
c）抽象工厂(Factory)
d）具体工厂(ConcreteFactory)



例如：ClientHttpRequestFactory 抽象工厂的应用

- ✓ 抽象工厂 （ClientHttpRequestFactory）
- ✓ 具体工厂 （SimpleClientHttpRequestFactory）
- ✓ 抽象产品 （ClientHttpRequest,HttpHeaders,…）
- ✓ 具体产品 （SimpleStreamingClientHttpRequest,…）

4）抽象工厂应用分析?

a）优势：工厂方法模式可能会产生很多个工厂类，基于此劣势借助抽象工厂创建产品族对象，可以减少工厂类对象的个数，从而更好节省资源。

b）劣势：一旦有新的产品族的诞生，这个抽象工厂扩展起来就会比较复杂。

2.1.4. 建造模式(Builder)

1）如何理解建造模式?

建造模式又称之为构建模式，通常用于构建相对比较复杂对象，例如。

a）构建过程复杂。(除了复杂很多过程又重复比较多)

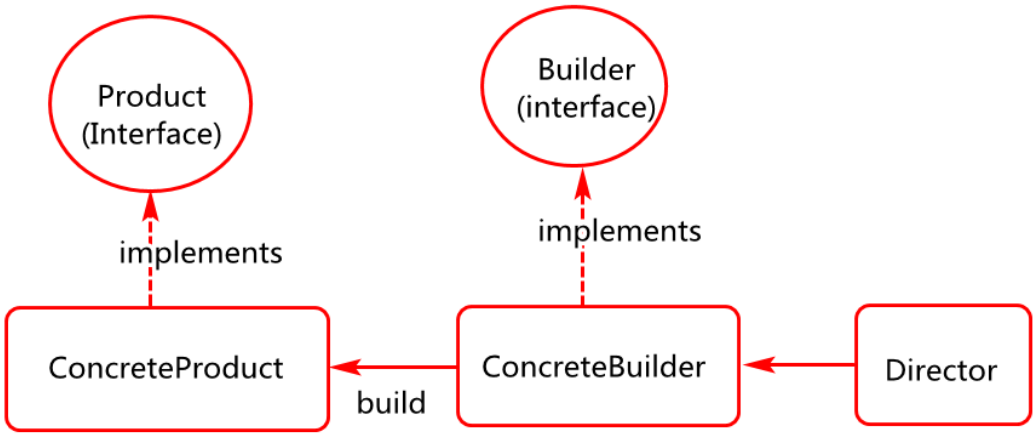
b）对象依赖关系复杂。

2）建造模式应用场景分析?

a）Mybatis(XmlConfigBuilder,XmlStatementBuilder)
b）Spring(XmlBeanDefinitionReader)
c）…

3）建造模式对象角色分析?

a）抽象建造对象角色(Builder):可能是接口，也可能是抽象类，此角色也可以省略。
b）具体建造对象角色(ConcreteBuilder)
c）导演角色(Director)：持有建造者对象，可以省略
d）抽象产品角色(Product) 抽象产品角色，可以省略
e）具体产品角色(ConcreteProduct) 具体产品角色



4）建造模式应用分析?
a）优势:解耦对象的应用以及对象创建,通过建造者创建复杂产品对象，尤其是基于配置文件创建对象的场景。
b）劣势:要构建的对象结构假如频繁变化可能导致构建者对象的设计比较复杂。

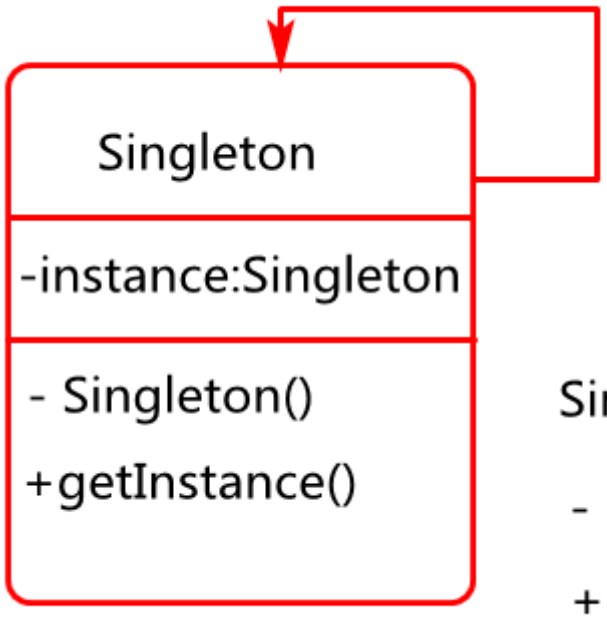
2.1.5. 单例模式(Singleton)

1）如何理解单例模式?
保证一个类的实例在”特定范围”只有一份(例如一个 JVM 内部，一个线程内部)，并且提供一个全局访问点可以访问到这份实例。

2）单例模式的应用场景?
a） Spring(Singleton 作用域的 Bean 对象)
b） MyBatis(ErrorContext 对象是每个线程一份此类实例)
c） …

3）单例模式对象角色构成

a）具体产品对象（例如 Singleton）



Singleton pattern

- private

+public

4）单例模式应用分析

a)优势：科学使用资源，避免频繁创建，销毁对象时造成的资源浪费。

b)劣势：设计不够严谨会存在线程安全问题，可扩展性相对较差。

2.2. 结构型模式

2.2.1. 适配器模式 （Adapter)

1．如何理解适配器模式?

1）适配器模式(Adapter Pattern) 将一个接口转换成客户希望的另一个接口，以解决接口不兼容问题。

2）又名包装器(Wrapper)模式

3）分为类适配器，对象适配器。

2．适配器模式场景分析?

1）生活中(一拖三充电头、HDMI 转 VGA)

2）mybatis (Log 接口)

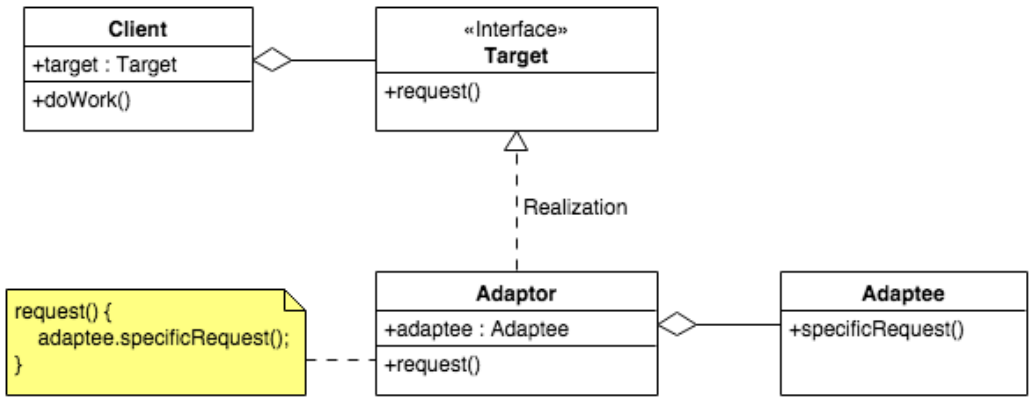
3）spring (HandlerAdapter,AdvisorAdapter)

3．适配器模式对象角色构成?

1）Target(目标抽象类)：抽象类或接口，也可以是具体类

2）Adapter(适配器类)： 负责对 Adaptee 和 Target 进行适配。

3）Adaptee(适配者类)：适配者即被适配的角色。



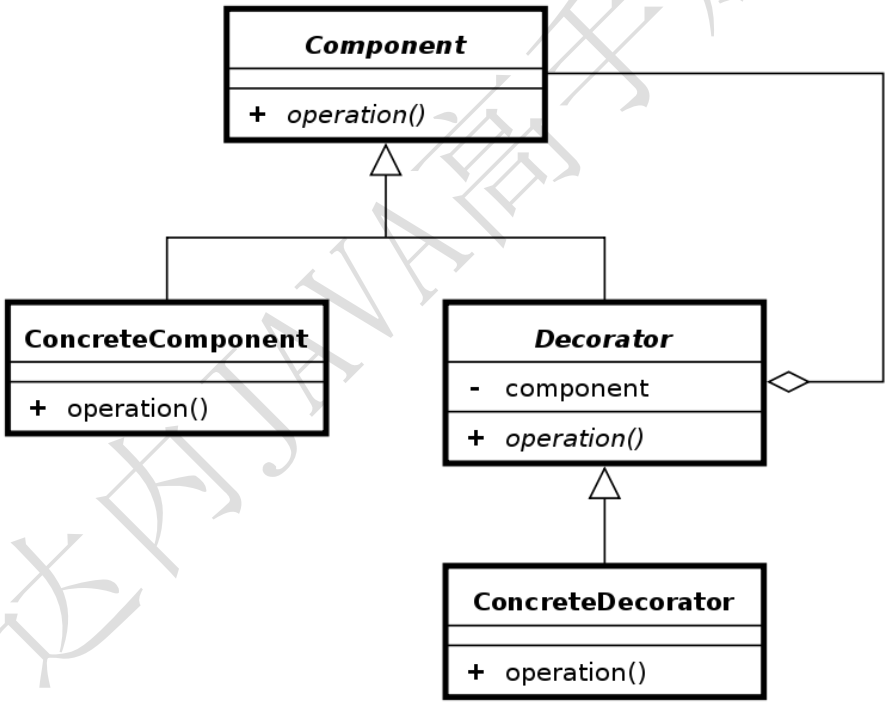
4．适配器模式应用分析?

1）优势：对客户端透明，扩展性好，复用性好

2）劣势：是一种补偿机制的实现，主要用于后期扩展。

2.2.2. 装饰模式 （Decorator）

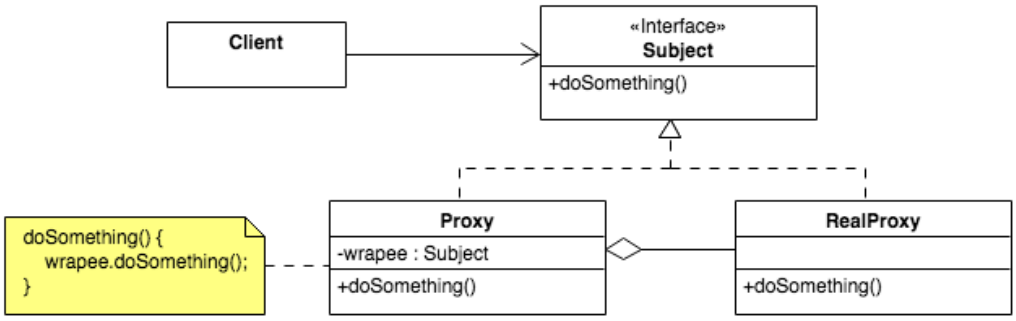
1. 如何理解装饰模式?
   1）油漆工模式
   2）基于目标对象添加额外职责（功能扩展）
2. 装饰模式场景分析?
   1）mybatis(Executor)
   2）spring()
3. 装饰模式角色构成?
   1）Component：抽象构件
   2）ConcreteComponent：具体构件
   3）Decorator：抽象装饰类
   4）ConcreteDecorator：具体装饰类



4. 装饰模式应用分析?
   1）提供了相对继承更加灵活的功能扩展方式
   2）实现相对复杂。

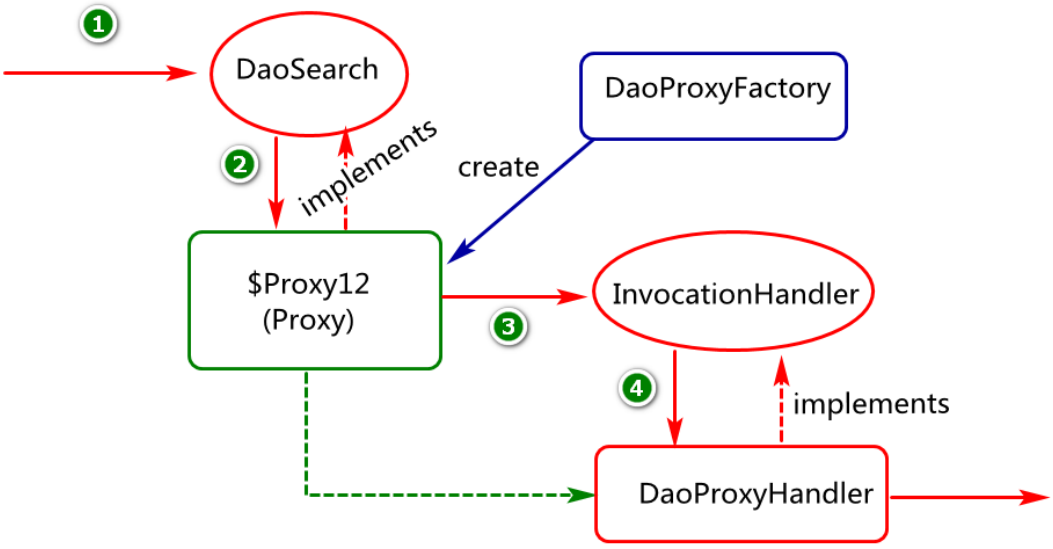
4.1.1. 代理模式 （Adapter）

1. 如何理解代理模式?
   5) 基于目标对象创建代理对象，并由代理对象控制目标对象的执行。
   6) 基于 OCP 原则扩展目标对象的功能。
2. 代理模式场景分析?
   1) mybatis（为接口创建代理对象，拦截器应用）
   2) spring（AOP）
3. 代理模式角色构成?
   1) Subject：抽象主题角色
   2) Proxy：代理主题角色
   3) RealSubject：真实主题角色

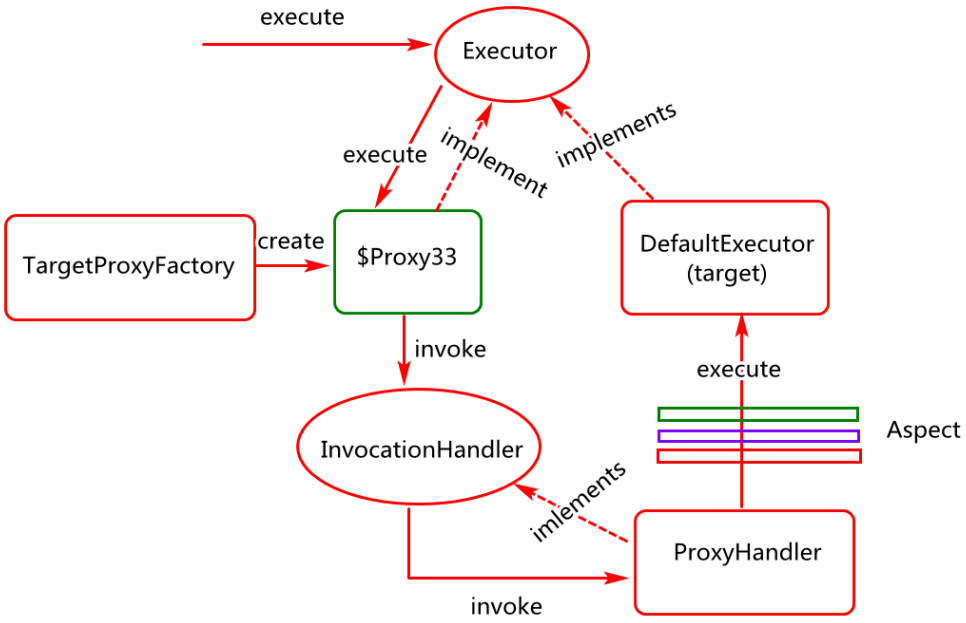


强化分析

1) MyBatis 中 mapper 接口的代理



2) JDK 动态代理（Spring AOP 思想）



4．代理模式应用分析?

1）优势：基于 OCP 进行控制，扩展目标对象

2）劣势：由于代理对象的创建可能会导致性能上的缺陷。

4.2. 行为模式

4.2.1. 策略模式(Strategy)

5． 如何理解策略模式?

6． 策略模式场景分析?

7． 策略模式角色构成?

8． 策略模式应用分析?

8.1.1. 模板方法模式(Template Method)

1． 如何理解模板方法模式?

2． 模板方法模式场景分析?

3． 模板方法模式角色构成?

4. 模板方法模式应用分析?

4.1.1. 观察者模式(Template Method)

1. 如何理解代理模式?
2. 代理模式场景分析?
3. 代理模式角色构成?
4. 代理模式应用分析?