一、设计模式六大原则

1.单一原则（Single Responsibility Principle）：一个类或者一个方法只负责一项职责。

2.里氏替换原则（LSP liskov substitution principle）：子类可以扩展父类的功能，但不能改变原有父类的功能

3.依赖倒置原则（dependence inversion principle）：面向接口编程，（通过接口作为参数实现应用场景）

抽象就是接口或者抽象类，细节就是实现类

上层模块不应该依赖下层模块，两者应依赖其抽象；

抽象不应该依赖细节，细节应该依赖抽象；

通俗点就是说变量或者传参数，尽量使用抽象类，或者接口；

4.接口隔离（interface segregation principle）：建立单一接口；（扩展为类也是一种接口，一切皆接口）

定义：

a.客户端不应该依赖它不需要的接口；

b.类之间依赖关系应该建立在最小的接口上；

简单理解：复杂的接口，根据业务拆分成多个简单接口；（对于有些业务的拆分多看看适配器的应用）

【接口的设计粒度越小，系统越灵活，但是灵活的同时结构复杂性提高，开发难度也会变大，维护性降低】

5.迪米特原则（law of demeter LOD）：最少知道原则，尽量降低类与类之间的耦合，一个对象应该对其他对象有最少的了解

6.开闭原则（open closed principle）：对扩展开放，对修改闭合

二、工厂设计模式

工厂模式分为简单工厂模式，工厂方法模式和抽象工厂模式，它们都属于设计模式中的创建型模式。其主要功能都是帮助我们把对象的实例化部分抽取了出来，目的是降低系统中代码耦合度，并且增强了系统的扩展性。

（1）简单工厂设计模式

简单工厂模式最大的优点在于实现对象的创建和对象的使用分离，将对象的创建交给专门的工厂类负责，但是其最大的缺点在于工厂类不够灵活，增加新的具体产品需要修改工厂类的判断逻辑代码，而且产品较多时，工厂方法代码将会非常复杂。

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public interface Car {
    /**
     * 汽车运行的接口方法
     */
    void run();
}
```

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Benz implements Car {
    public void run() {
        System.out.println("奔驰 is running! ");
    }
}

public class Bike implements Car {
    public void run() {
        System.out.println("我只有自行车! ");
    }
}

public class Bmw implements Car {
    public void run() {
        System.out.println("宝马 is running! ");
    }
}
```

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class CarFactory {

    public static Car getCar(String type){
        if("benz".equalsIgnoreCase(type)){
            //其中可能有很复杂的操作
            return new Benz();
        }else if("bmw".equalsIgnoreCase(type)){
            //其中可能有很复杂的操作
            return new Bmw();
        }else {
            return new Bike();
        }
    }
}
```

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {
        Car bmw = CarFactory.getCar("bmw");
        bmw.run();
        Car benz = CarFactory.getCar("benz");
        benz.run();
    }
}
```

（2）工厂方法模式

我们说过java开发中要遵循开闭原则，如果将来有一天我想增加一个新的车，那么必须修改CarFactory，就不太灵活。解决方案是使用工厂方法模式。

我们为每一个车都构建成一个工厂：

> 先抽象一个工厂接口

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public interface Factory {
    /**
     * 统一的创建方法
     * @return
     */
    Car create();
}
```

> 然后针对每一个产品构建一个工厂方法

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class BenzFactory implements Factory {
    public Car create() {
        //中间省略一万行代码
        return new Benz();
    }
}

public class BmwFactory implements Factory {
    public Car create() {
        //中间省略一万行代码
        return new Bmw();
    }
```

```
}

public class BikeFactory implements Factory {
    public Car create() {
        //中间省略一万行代码
        return new Bike();
    }
}
```

应用场景

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {
        Factory benzFactory = new BenzFactory();
        Car benz = benzFactory.create();
        benz.run();
        Factory bmwFactory = new BmwFactory();
        Car bmw = bmwFactory.create();
        bmw.run();
    }
}
```

好处

此模式中，通过定义一个抽象的核心工厂类，并定义创建产品对象的接口，创建具体产品实例的工作延迟到其工厂子类去完成。这样做的好处是核心类只关注工厂类的接口定义，而具体的产品实例交给具体的工厂子类去创建。当系统需要新增一个产品是，无需修改现有系统代码，只需要添加一个具体产品类和其对应的工厂子类，使系统的扩展性变得很好，符合面向对象编程的开闭原则。

缺点

工厂方法模式虽然扩展性好，但是增加了编码难度，大量增加了类的数量，所以怎么选择还是看实际的需求。

三、代理设计模式

代理模式分为静态代理和动态代理。代理的核心功能是方法增强。

1、静态代理

静态代理角色分析

- 抽象角色 : 一般使用接口或者抽象类来实现
- 真实角色 : 被代理的角色
- 代理角色 : 代理真实角色 ; 代理真实角色后 , 一般会做一些附属的操作 .
- 客户 ： 使用代理角色来进行一些操作 .

代码实现

写一个接口

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public interface Singer {
    /**
     * 歌星都能唱歌
     */
    void sing();
}
```

定义男歌手

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public class MaleSinger implements Singer{

    private String name;

    public MaleSinger(String name) {
        this.name = name;
    }

    @Override
    public void sing() {
        System.out.println(this.name + "开始唱歌了！");
    }
}
```

定义经纪人

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public class Agent implements Singer {

    private Singer singer;

    public Agent(Singer singer) {
        this.singer = singer;
    }

    @Override
    public void sing() {
        System.out.println("节目组找过来！需要演出，谈好演出费用。。。。。");
        singer.sing();
        System.out.println("结算费用，下一次合作预约。。。。。。");
    }
}
```

Client . java 即客户

```java
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public class Client {

    public static void main(String[] args) {
        Singer singer = new MaleSinger("鹿晗");
        Singer agent = new Agent(singer);
        agent.sing();
    }
}
```

分析：在这个过程中，你直接接触的就是鹿晗的经济人，经纪人在鹿晗演出的前后跑前跑后发挥了巨大的作用。

除了实现共同的接口，我们还能使用继承类的方式

```java
/**
 * @author itnanls
 * @date 2021/1/25
 **/
public class Agent extends MaleSinger {

    private MaleSinger maleSinger;

    public void setMaleSinger(MaleSinger maleSinger) {
        this.maleSinger = maleSinger;
    }

    @Override
    public void sing() {
        System.out.println("开始唱歌了-----------");
        maleSinger.sing();
        System.out.println("结束唱歌了-----------");
    }
}
```

```java
public static void main(String[] args) {
    MaleSinger maleSinger = new MaleSinger("鹿晗");
    Agent agent = new Agent();
    agent.setMaleSinger(maleSinger);
    agent.sing();
}
```

优点

- 鹿晗还是鹿晗，没有必要为了一下前置后置工作改变鹿晗这个类
- 公共的统一问题交给代理处理
- 公共业务进行扩展或变更时，可以更加方便

- 这不就是更加符合开闭原则，单一原则吗?

> 缺点：

- 每个类都写个代理，麻烦死了。

2、动态代理

- 动态代理的角色和静态代理的一样 .
- 动态代理的代理类是动态生成的 . 静态代理的代理类是我们写的
- 动态代理分为两类 : 一类是基于接口动态代理 , 一类是基于类的动态代理
- - 基于接口的动态代理----JDK动态代理
  - 基于类的动态代理--cglib

动态代理就是当有大量的类需要执行一些共同代码时，我们自己写太麻烦，那能不能直接使用java代码，自动生成一个类帮助我们批量的增强某些方法。

（1）JDK原生的动态代理

JDK的动态代理需要了解两个类

核心 : InvocationHandler 和 Proxy ， 打开JDK帮助文档看看

【InvocationHandler：调用处理程序】

```
Object invoke(Object proxy, 方法 method, Object[] args);
//参数
//proxy - 调用该方法的代理实例
//method -所述方法对应于调用代理实例上的接口方法的实例。方法对象的声明类将是该方法声明的接口，它可以是代理类继承该方法的代理接口的超级接口。
//args -包含的方法调用传递代理实例的参数值的对象的阵列，或null如果接口方法没有参数。原始类型的参数包含在适当的原始包装器类的实例中，例如java.lang.Integer或java.lang.Boolean 。
```

【Proxy ：代理】

```
Proxy.newProxyInstance(ClassLoader loader,
                       Class<?>[] interfaces,
                       InvocationHandler h)
```

代码实现

抽象角色和真实角色和之前的一样!

还是歌星和男歌星

Agent. java 即经纪人

```
/**
```

```
 * @author IT楠老师
 * @date 2020/5/21
 * 经纪人
 */
public class Agent implements InvocationHandler {

    private Singer singer;

    /**
     * 设置代理的经济人
     * @param singer
     */
    public void setSinger(Singer singer) {
        this.singer = singer;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("---------------经纪人把把关--------------");
        Object returnObj = method.invoke(singer, args);
        System.out.println("---------------唱完了收收钱-----------------------");
        return returnObj;
    }

    /**
     * 获取一个代理对象
     * @return
     */
    public Object getProxy(){
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
                new Class[]{Singer.class},this);
    }
}

```

Client . java

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {

        MaleSinger luhan = new MaleSinger();

        Agent agent = new Agent();
        agent.setSinger(luhan);
        Singer singer = (Singer)agent.getProxy();

        singer.sing();
    }
}
```

核心：**一个动态代理，一般代理某一类业务，一个动态代理可以代理多个类，代理的是接口！、**

```
//该设置用于输出jdk动态代理产生的类
System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles",
"true");
```

(2) 基于cglib

```
<dependencies>
    <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>3.3.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>utf-8</encoding>
            </configuration>
        </plugin>
    </plugins>
</build>
```

```
public static void main(String[] args) {
    Enhancer enhancer=new Enhancer();
    enhancer.setSuperclass(MaleSinger.class);
    enhancer.setCallback(new MethodInterceptor() {
        public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
            System.out.println("---------");
            Object invoke = methodProxy.invokeSuper(o,objects);
            System.out.println("++++++++++");
            return invoke;
        }
    });
    MaleSinger maleSinger = (MaleSinger)enhancer.create();
    maleSinger.sing();
}

public static void main(String[] args) {
    Enhancer enhancer=new Enhancer();
    enhancer.setSuperclass(MaleSinger.class);
    enhancer.setCallback(new MethodInterceptor() {
```

```
        public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
            System.out.println("---------");
            Object invoke = methodProxy.invokeSuper(o,objects);
            System.out.println("++++++++++");
            return invoke;
        }
    });
    MaleSinger maleSinger = (MaleSinger)enhancer.create(new Class[]
{String.class},new Object[]{"小李"});
    maleSinger.sing();
}
```

```
 /**
     * All generated proxied methods call this method instead of the
original method.
     * The original method may either be invoked by normal reflection using
the Method object,
     * or by using the MethodProxy (faster).
     * @param obj "this", the enhanced object
     * @param method intercepted Method
     * @param args argument array; primitive types are wrapped
     * @param proxy used to invoke super (non-intercepted method); may be
called
     * as many times as needed
     * @throws Throwable any exception may be thrown; if so, super method
will not be invoked
     * @return any value compatible with the signature of the proxied
method. Method returning void will ignore this value.
     * @see MethodProxy
     */
    public Object intercept(Object obj, java.lang.reflect.Method method,
Object[] args,
                            MethodProxy proxy) throws Throwable;
```

```
//该设置用于输出cglib动态代理产生的类
System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY,
"D:\\class");
```

1. Java动态代理只能够对接口进行代理，不能对普通的类进行代理（因为所有生成的代理类的父类为Proxy，Java类继承机制不允许多重继承）；
2. CGLIB能够代理普通类；
3. Java动态代理使用Java原生的反射API进行操作，在生成类上比较高效；CGLIB使用ASM框架直接对字节码进行操作，在类的执行过程中比较高效

四、实战

1、hikari.properties

```
username=root
password=root
jdbcUrl=jdbc:mysql://127.0.0.1:3306/ssm?useUnicode=true&characterEncoding=utf8&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true&useSSL=false
driverClassName=com.mysql.cj.jdbc.Driver
```

druid.properties

```
url=jdbc:mysql://127.0.0.1:3306/ssm?useUnicode=true&characterEncoding=utf8&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true&useSSL=false
username=root
password=root
driverClassName=com.mysql.cj.jdbc.Driver
```

2、mybatis-config.properties

```
<mybatis>
    <dataSource>hikari</dataSource>
    <mapper>mapper/userMapper.xml</mapper>
    <mapper>mapper/studentMapper.xml</mapper>
</mybatis>
```

3、UserMapper.xml

```
<mapper namespace="cn.itnanls.UserDao">
    <insert id="saveUser" resultType="cn.itnanls.User" paramType="cn.itnanls.User">
        insert into user values(?,?,?)
    </insert>
    <select id="findUser" resultType="cn.itnanls.User" paramType="cn.itnanls.User">
        select * from user where id = ?
    </select>
</mapper>
```

4、User

```java
/**
 * @author zn
 * @date 2021/1/25
 */


public class User implements Serializable {
    private int id;
    private String username;
    private String password;

    public User() {
    }

    public User(int id, String username, String password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
}
```

5、UserDao

```
/**
 * @author zn
 * @date 2021/1/25
 */
public interface UserDao {

    Integer saveUser(User user);

    List<User> findUser(Integer id);
}
```

6、DaoWrapper

```
/**
 * 用于描述一个Dao的方法的必要条件
 * @author zn
 * @date 2021/1/25
 */
public class DaoWrapper {

    /**
     * 类型，insert|update|delete
     */
    private String type;
    /**
     * 返回值的类型
     */
    private String resultType;
    /**
     * 参数的类型
     */
    private String paramType;
    /**
     * sql语句
     */
    private String sql;

    public DaoWrapper(String type, String resultType, String paramType,
String sql) {
        this.type = type;
        this.resultType = resultType;
        this.paramType = paramType;
        this.sql = sql;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
```

```
        this.type = type;
    }

    public String getResultType() {
        return resultType;
    }

    public void setResultType(String resultType) {
        this.resultType = resultType;
    }

    public String getParamType() {
        return paramType;
    }

    public void setParamType(String paramType) {
        this.paramType = paramType;
    }

    public String getSql() {
        return sql;
    }

    public void setSql(String sql) {
        this.sql = sql;
    }

    @Override
    public String toString() {
        return "DaoWrapper{" +
                "type='" + type + '\'' +
                ", resultType='" + resultType + '\'' +
                ", paramType='" + paramType + '\'' +
                ", sql='" + sql + '\'' +
                '}';
    }
}
```

7、DataSourceFactory

```
/**
 * 数据源工厂
 * 简单工厂的应用
 * @author zn
 * @date 2021/1/25
 */
public class DataSourceFactory {

    public static DataSource createDataSource(String type){

        DataSource dataSource = null;
        Properties properties = new Properties();
        if("hikari".equals(type)){
            try {
```

```
 properties.load(DataSourceFactory.class.getClassLoader().getResourceAsStream("hikari.properties"));
            } catch (IOException e) {
                e.printStackTrace();
            }
            HikariConfig hikariConfig = new HikariConfig(properties);
            dataSource = new HikariDataSource(hikariConfig);

        } else if ("druid".equals(type)){
            try {

 properties.load(DataSourceFactory.class.getClassLoader().getResourceAsStream("druid.properties"));
            } catch (IOException e) {
                e.printStackTrace();
            }
            DruidDataSource druidDataSource = new DruidDataSource();
            druidDataSource.configFromPropety(properties);
            dataSource = druidDataSource;
        }

        return dataSource;
    }

}
```

8、Session

```
/**
 * 会话对象
 * @author zn
 * @date 2021/1/25
 */
public class Session {

    /**
     * 每个会话持有一个链接
     */
    private Connection connection;
    /**
     * 当前会话的上下文
     */
    private Map<String, Map<String, DaoWrapper>> env = new HashMap<>(8);

    public Session(Connection connection,Map<String, Map<String, DaoWrapper>> env) {
        this.connection = connection;
        this.env = env;
    }

    /**
     * 拿到一个包装类
```

```
     * @param clazz
     * @param <T>
     * @return
     */
    public <T> T getMapper(Class<T> clazz) {
        T t = (T) Proxy.newProxyInstance(this.getClass().getClassLoader(),
                new Class[]{clazz},
                new SQLHandler(connection, clazz,env.get(clazz.getName())));
        return t;
    }

    // 开始会话
    public void begin() {
        try {
            connection.setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // 提交
    public void commit() {
        try {
            connection.commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }


    // 回滚
    public void rollback() {
        try {
            connection.rollback();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}
```

9、SessionFactory

```
/**
 * @author zn
 * @date 2021/1/25
 */
public class SessionFactory {

    private DataSource dataSource;

    private Map<String, Map<String, DaoWrapper>> env = new HashMap<>(8);

    public SessionFactory(String config) {
        loadXml(config);
```

```
    }

    // 打开一个会话
    public Session openSession() {
        Connection connection = null;
        try {
            connection = dataSource.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return new Session(connection, env);
    }

    // 加载资源环境
    public void loadXml(String config) {
        try {
            SAXReader reader = new SAXReader();
            Document configDom =
reader.read(Session.class.getClassLoader().getResourceAsStream(config));
            Element configRoot = configDom.getRootElement();
            String dataSourceType =
configRoot.element("dataSource").getTextTrim();
            dataSource = DataSourceFactory.createDataSource(dataSourceType);

            List mapperElements = configRoot.elements("mapper");
            List<String> mapperPaths = new ArrayList<>();
            for (Object element : mapperElements) {
                Element mapper = (Element) element;
                mapperPaths.add(mapper.getTextTrim());
            }

            for (String mapperPath : mapperPaths) {
                Map<String, DaoWrapper> wrapper = new HashMap<>(2);
                Document document =
reader.read(Session.class.getClassLoader().getResourceAsStream(mapperPath));
                Element root = document.getRootElement();
                String namespace = root.attribute("namespace").getValue();
                Iterator iterator = root.elementIterator();
                while (iterator.hasNext()) {
                    Element el = (Element) iterator.next();
                    String type = el.getName();
                    String id = el.attribute("id").getValue();
                    String resultType =
el.attribute("resultType").getValue();
                    String paramType = el.attribute("paramType").getValue();
                    String sqlStr = el.getTextTrim();

                    wrapper.put(id, new DaoWrapper(type, resultType,
paramType, sqlStr));
                }
                env.put(namespace, wrapper);
            }
        } catch (DocumentException e) {
            e.printStackTrace();
        }

    }
}
```

```
66
```

10、SQLHandler

```
/**
 * @author zn
 * @date 2021/1/25
 */
public class SQLHandler implements InvocationHandler {

    /**
     * 需传入一个链接
     */
    private Connection connection;
    /**
     * 需传入一个dao的类型
     */
    private Class clazz;
    /**
     * 需传入一个独立的环境
     */
    private Map<String, DaoWrapper> env;


    public SQLHandler(Connection connection, Class clazz,Map<String,
DaoWrapper> env) {
        this.connection = connection;
        this.clazz = clazz;
        this.env = env;
    }

    /**
     * 生成代理对象
     * @param proxy
     * @param method
     * @param args
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

        // 拿到包装
        DaoWrapper wrapper = env.get(method.getName());

        PreparedStatement statement =
connection.prepareStatement(wrapper.getSql());

        // 对每一种sql语句进行独立的操作
        if ("insert".equals(wrapper.getType())) {
            String paramType = wrapper.getParamType();
            // 暂定传入一个对象
            Class<?> clazz = args[0].getClass();
            Field[] fields = clazz.getDeclaredFields();
```

```
            for (int i = 0; i < fields.length; i++) {
                fields[i].setAccessible(true);
                statement.setObject(i + 1, fields[i].get(args[0]));
            }
            return statement.executeUpdate();

        } else if ("delete".equals(wrapper.getType())) {
            for (int i = 0; i < args.length; i++) {
                statement.setObject(i + 1, args[i]);
            }
            return statement.executeUpdate();

        } else if ("select".equals(wrapper.getType())) {
            for (int i = 0; i < args.length; i++) {
                statement.setObject(i + 1, args[i]);
            }
            ResultSet result = statement.executeQuery();
            List list = new ArrayList();
            while (result.next()) {
                Class<?> clazz = Class.forName(wrapper.getResultType());
                Object object = clazz.newInstance();
                Field[] fields = clazz.getDeclaredFields();
                for (int i = 0; i < fields.length; i++) {
                    fields[i].setAccessible(true);
                    fields[i].set(object,
result.getObject(fields[i].getName()));

                }
                list.add(object);
            }
            return list;
        }
        return null;
    }
}

```