# day28 【设计模式】

## 今日目标

- 单例设计模式
- 多例设计模式
- 工厂设计模式
- 动态代理

## 教学目标

- [ ] 能够说出单例设计模式的好处
- [ ] 能够说出多例模式的好处
- [ ] 能够说出动态代理模式的作用
- [ ] 能够使用Proxy的方法生成代理对象
- [ ] 能够使用工厂模式编写java程序

# 第一章 单例设计模式

正常情况下一个类可以创建多个对象

```
public static void main(String[] args) {
    // 正常情况下一个类可以创建多个对象
    Person p1 = new Person();
    Person p2 = new Person();
    Person p3 = new Person();
}
```

## 1.1 单例设计模式的作用

单例模式，是一种常用的软件设计模式。通过单例模式可以保证系统中，应用该模式的这个类只有一个实例。即一个类只有一个对象实例。

## 1.2 单例设计模式实现步骤

1. 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
2. 在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
3. 定义一个静态方法返回这个唯一对象。

## 1.3 单例设计模式的类型

根据实例化对象的时机单例设计模式又分为以下两种:

1. 饿汉单例设计模式

2. 懒汉单例设计模式

## 1.4 饿汉单例设计模式

饿汉单例设计模式就是使用类的时候已经将对象创建完毕，不管以后会不会使用到该实例化对象，先创建了再说。很着急的样子，故被称为“饿汉模式”。

代码如下：

```
public class Singleton {
    // 1.将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
    private Singleton() {}

    // 2.在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
    private static final Singleton instance = new Singleton();

    // 3.定义一个静态方法返回这个唯一对象。
    public static Singleton getInstance() {
        return instance;
    }
}
```

## 1.5 懒汉单例设计模式

懒汉单例设计模式就是调用getInstance()方法时实例才被创建，先不急着实例化出对象，等要用的时候才实例化出对象。不着急，故称为“懒汉模式”。

代码如下：

```
public class Singleton {

    // 2.在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
    private static Singleton instance;

    // 1.将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
    private Singleton() {}

    // 3.定义一个静态方法返回这个唯一对象。要用的时候才例化出对象
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

> 注意：懒汉单例设计模式在多线程环境下可能会实例化出多个对象，不能保证单例的状态，所以加上关键字：synchronized，保证其同步安全。

## 1.6 小结

单例模式可以保证系统中一个类只有一个对象实例。

实现单例模式的步骤：

1. 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
2. 在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
3. 定义一个静态方法返回这个唯一对象。

# 第二章 多例设计模式

一般情况下一个类可以创建多个对象

```
public static void main(String[] args) {
    // 正常情况下一个类可以创建多个对象
    Person p1 = new User();
    Person p2 = new User();
    Person p3 = new User();
}
```

## 2.1.多例设计模式的作用

多例模式，是一种常用的软件设计模式。通过多例模式可以保证系统中，应用该模式的类有固定数量的实例。多例类要自我创建并管理自己的实例，还要向外界提供获取本类实例的方法。

## 2.2.实现步骤

1.创建一个类, 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。

2.在类中定义该类被创建的总数量

3.在类中定义存放类实例的list集合

4.在类中提供静态代码块,在静态代码块中创建类的实例

5.提供获取类实例的静态方法

## 2.3.实现代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
public class Multition {
    // 定义该类被创建的总数量
    private static final int maxCount = 3;
    // 定义存放类实例的list集合
    private static List instanceList = new ArrayList();
    // 构造方法私有化,不允许外界创建本类对象
```

```
    private Multition() {
    }
    static {
        // 创建本类的多个实例,并存放到list集合中
        for (int i = 0; i < maxCount; i++) {
            Multition multition = new Multition();
            instanceList.add(multition);
        }
    }
    // 给外界提供一个获取类对象的方法
    public static Multition getMultition(){
        Random random = new Random();
        // 生成一个随机数
        int i = random.nextInt(maxCount);
        // 从list集合中随机取出一个进行使用
        return (Multition)instanceList.get(i);
    }
}
```

## 2.4.测试结果

```
public static void main(String[] args) {
    // 编写一个循环从中获取类对象
    for (int i = 0; i < 10; i++) {
        Multition multition = Multition.getMultition();
        System.out.println(multition);
    }
}
```

```
D:\software\java\jdk\jdk1.8.0_45_x64\bin\java ...
com.itheima.Multition@60e53b93
com.itheima.Multition@5e2de80c
com.itheima.Multition@60e53b93
com.itheima.Multition@5e2de80c
com.itheima.Multition@60e53b93
com.itheima.Multition@1d44bcfa
com.itheima.Multition@60e53b93
com.itheima.Multition@5e2de80c
com.itheima.Multition@5e2de80c
com.itheima.Multition@1d44bcfa
```

## 2.5.小结

多例模式可以保证系统中一个类有固定个数的实例, 在实现需求的基础上, 能够提高实例的复用性.

实现多例模式的步骤：

1. 创建一个类, 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
2. 在类中定义该类被创建的总数量
3. 在类中定义存放类实例的list集合
4. 在类中提供静态代码块,在静态代码块中创建类的实例
5. 提供获取类实例的静态方法

# 第三章 动态代理

## 3.1 代理模式【Proxy Pattern】

为什么要有“代理”？生活中就有很多例子，例如委托业务等等，**代理就是被代理者没有能力或者不愿意去完成某件事情，需要找个人代替自己去完成这件事**，这才是“代理”存在的原因。例如，我现在需要出国，但是我不愿意自己去办签证、预定机票和酒店（觉得麻烦，那么就可以找旅行社去帮我办，这时候旅行社就是代理，而我自己就是被代理了。

在我们的代码中，假如有以下业务情景：

用户登录到我们的系统后，我们的系统会为其产生一个ArrayList集合对象，内部存储了一些用户信息，而后，这个对象需要被传给后面的很多其它对象，但要求其它对象不能对这个ArrayList对象执行添加、删除、修改操作，只能get()获取元素。那么为了防止后面的对象对集合对象进行添加、修改、删除操作，我们应该怎样办呢?

要想实现这种要求，方案有很多种。"代理模式"就是其中的一种，而且是非常合适的一种。我们看一下用代理模式怎样实现这种需求：

> 1. 为ArrayList定义一个代理类：ArrayListProxy

```
import java.util.*;

public class ArrayListProxy implements List<String> {
    //被代理的List集合
    private List<String> list;

    //通过构造方法将被代理的集合传入
    public ArrayListProxy(List<String> list) {
        this.list = list;
    }
    /*
        以下方法都是List接口中的方法：
            允许被使用的方法：调用被代理对象的原方法
            不允许被使用的方法：抛出异常
     */
    @Override
    public int size() {
        return this.list.size();
    }
    @Override
    public boolean isEmpty() {
        return this.list.isEmpty();
    }
    @Override
    public boolean contains(Object o) {
```

```
        return this.list.contains(o);
    }
    @Override
    public Iterator<String> iterator() {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public Object[] toArray() {
        return this.list.toArray();
    }
    @Override
    public <T> T[] toArray(T[] a) {
        return this.list.toArray(a);
    }
    @Override
    public boolean add(String s) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public boolean remove(Object o) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public boolean containsAll(Collection<?> c) {
        return this.list.containsAll(c);
    }
    @Override
    public boolean addAll(Collection<? extends String> c) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public boolean addAll(int index, Collection<? extends String> c) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public boolean removeAll(Collection<?> c) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public boolean retainAll(Collection<?> c) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public void clear() {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public String get(int index) {
        return this.list.get(index);
    }
    @Override
    public String set(int index, String element) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public void add(int index, String element) {//不允许使用
        throw new UnsupportedOperationException();
    }
```

```
    @Override
    public String remove(int index) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public int indexOf(Object o) {
        return this.list.indexOf(o);
    }
    @Override
    public int lastIndexOf(Object o) {
        return this.list.lastIndexOf(o);
    }
    @Override
    public ListIterator<String> listIterator() {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public ListIterator<String> listIterator(int index) {//不允许使用
        throw new UnsupportedOperationException();
    }
    @Override
    public List<String> subList(int fromIndex, int toIndex) {//不允许使用
        throw new UnsupportedOperationException();
    }
}
```

1. 在测试类中定义一个方法，接收一个ArrayList对象，然后返回这个代理对象，并在main()方法中测试：

```
import java.util.ArrayList;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        //定义一个集合对象
        ArrayList<String> list = new ArrayList<>();

        list.add("张三");
        list.add("李四");
        list.add("王五");

        //调用getList()方法，根据这个集合获取一个它的"不可变"对象
        List<String> listProxy = getList(list);
        for (int i = 0; i < listProxy.size(); i++) {//OK的
            System.out.println(listProxy.get(i));//OK的

        }
        //其它方法都会抛出异常
//        listProxy.add("赵六");//UnsupportedOperationException
//        listProxy.remove(0);//UnsupportedOperationException
//        listProxy.set(0, "张老三");//UnsupportedOperationException

    }
    //此方法接收一个List<String>对象，返回一个不可变的"代理对象"
    public static List<String> getList(List<String> list) {
        ArrayListProxy proxy = new ArrayListProxy(list);
```

```
        return proxy;
    }
}
```

## 3.2 动态代理概述

动态代理简单来说是：**拦截对真实对象方法的直接访问，增强真实对象方法的功能**

动态代理详细来说是：代理类在程序运行时创建的代理对象被称为动态代理，也就是说，这种情况下，代理类并不是在Java代码中定义的，而是在运行时根据我们在Java代码中的“指示”动态生成的。也就是说你想获取哪个对象的代理，动态代理就会动态的为你生成这个对象的代理对象。**动态代理可以对被代理对象的方法进行增强，可以在不修改方法源码的情况下，增强被代理对象方法的功能，在方法执行前后做任何你想做的事情。**动态代理技术都是在框架中使用居多，例如：Struts1、Struts2、Spring和Hibernate等后期学的一些主流框架技术中都使用了动态代理技术。

## 3.3 案例引出

**演示Java已经实现的动态代理**

```
java.util.Collections:操作集合的工具类

static <T> List<T> unmodifiableList(List<? extends T> list)
        返回指定列表的不可修改视图。
        此方法允许模块为用户提供对内部列表的“只读”访问。
        在返回的列表上执行的查询操作将“读完”指定的列表。
        试图修改返回的列表（不管是直接修改还是通过其迭代器进行修改）将导致抛出
        UnsupportedOperationException:不支持操作异常
unmodifiableList作用:传递List接口,方法内部对List接口进行代理,返回一个被代理后的List接口
        对List进行代理之后,调用List接口的方法会被拦截
        如果使用的size,get方法,没有对集合进行修改,则允许执行
        如果使用的add,remove,set方法,对集合进行了修改,则抛出运行时异常
```

**代码实现:**

```
public class Demo01Proxy {
    @Test
    public void show(){
        //使用多态创建List集合,并添加元素
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");
        list.add("c");
        //调用Collections中的方法unmodifiableList对List集合进行代理
        list = Collections.unmodifiableList(list);
        //如果使用的size,get方法,没有对集合进行修改,则允许执行
        System.out.println(list.size());
        System.out.println(list.get(1));
        //如果使用的add,remove,set方法,对集合进行了修改,则抛出运行时异常
        //list.add("d");//UnsupportedOperationException
        //list.remove(0);//UnsupportedOperationException
        list.set(1, "www");//UnsupportedOperationException
```

```
    }
}
```

## 3.4 动态代理

java.lang.reflect.Proxy：这是 Java 动态代理机制的主类，它提供了一个静态方法来为一组接口的实现类动态地生成代理类及其对象。

- **newProxyInstance方法的三个参数的详解:**

该方法用于为指定类装载器、一组接口及调用处理器生成动态代理类实例

```
public static Object newProxyInstance(ClassLoader loader, Class[] interfaces,
InvocationHandler h)

1、obj.getClass().getClassLoader()目标对象通过getClass方法获取类的所有信息后，调用
getClassLoader() 方法来获取类加载器。获取类加载器后，可以通过这个类型的加载器，在程序运行时，
将生成的代理类加载到JVM即Java虚拟机中，以便运行时需要！

2、obj.getClass().getInterfaces()获取被代理类的所有接口信息，以便于生成的代理类可以具有代
理类接口中的所有方法。

3、InvocationHandler 这是调用处理器接口，它自定义了一个 invoke 方法，用于集中处理在动态代理
类对象上的方法调用，通常在该方法中实现对被代理类方法的处理以及访问.
```

- **InvocationHandler接口的invoke方法及参数详解**

```
public Object invoke(Object proxy, Method method, Object[] args)
1、Object proxy: 方法内部创建的代理对象。
2、Method method：方法内部会使用反射技术,自动被代理人使用的方法。
3、Object[] args：被代理方法中的参数。这里因为参数个数不定，所以用一个对象数组来表示。
```

## 3.5 动态代理综合案例

**需求:** 模拟unmodifiableList方法,对List接口进行代理 调用List接口的方法会被拦截 如果使用的size,get方法,没有对集合进行修改,则允许执行 如果使用的add,remove,set方法,对集合进行了修改,则抛出运行时异常

**分析:** 1.定义一个代理方法proxyList 参数:传递List集合 返回值:被代理之后的List集合 2.方法内部可以使用Proxy类中的方法实现动态代理

**代码实现:**

```
@SuppressWarnings("all")
public class Demo02Proxy {
    //1.定义一个代理方法proxyList
    public List proxyList(List<String> list){
        //2.方法内部可以使用Proxy类中的方法实现动态代理
        List<String> pList = (List<String>)
Proxy.newProxyInstance(Demo02Proxy.class.getClassLoader(),
                list.getClass().getInterfaces(),
```

```
                new InvocationHandlerImpl(list));
        return pList;
    }

    @Test
    public void show(){
        //使用多态创建List集合,并添加元素
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");
        list.add("c");
        //调用Collections中的方法unmodifiableList对List集合进行代理
        list = proxyList(list);
        //如果使用的size,get方法,没有对集合进行修改,则允许执行
        System.out.println(list.size());
        System.out.println(list.get(1));
        //如果使用的add,remove,set方法,对集合进行了修改,则抛出运行时异常
        //list.add("d");//UnsupportedOperationException
        //list.remove(0);//UnsupportedOperationException
        //list.set(1, "www");//UnsupportedOperationException
    }
}
```

```
public class InvocationHandlerImpl implements InvocationHandler{
    //定义一个List集合变量
    private List<String> list;

    //给List集合赋值
    public InvocationHandlerImpl(List<String> list) {
        super();
        this.list = list;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //invoke方法会获取List集合的方法,在invoke方法内部对List集合的方法进行拦截
        //获取List集合方法的名字
        String listMethodName = method.getName();
        //对获取的名字进行判断
        //如果使用的add,remove,set方法,对集合进行了修改,则抛出运行时异常
        if("add".equals(listMethodName)){
            throw new UnsupportedOperationException("add no run");
        }
        if("remove".equals(listMethodName)){
            throw new UnsupportedOperationException("remove no run");
        }
        if("set".equals(listMethodName)){
            throw new UnsupportedOperationException("set no run");
        }
        //如果使用的size,get方法,没有对集合进行修改,则允许执行
        Object v = method.invoke(list,args);
        return v;
    }
}
```

**动态代理案例流程图**


## 3.6 总结

动态代理非常的灵活，可以为任意的接口实现类对象做代理

动态代理可以为被代理对象的所有接口的所有方法做代理，动态代理可以在不改变方法源码的情况下，实现对方法功能的增强，

动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。 动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。

动态代理同时也提高了开发效率。

缺点：只能针对接口的实现类做代理对象，普通类是不能做代理对象的。

# 第四章 Lombok【自学扩展】

## 4.1 lombok介绍

Lombok通过增加一些“处理程序”，可以让java变得简洁、快速。

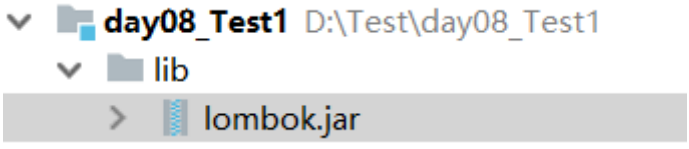
Lombok能以注解形式来简化java代码，提高开发效率。开发中经常需要写的javabean，都需要花时间去添加相应的getter/setter，也许还要去写构造器、equals等方法，而且需要维护。

Lombok能通过注解的方式，在编译时自动为属性生成构造器、getter/setter、equals、hashcode、toString方法。出现的神奇就是在源码中没有getter和setter方法，但是在编译生成的字节码文件中有getter和setter方法。这样就省去了手动重建这些代码的麻烦，使代码看起来更简洁些。

## 4.2 lombok使用

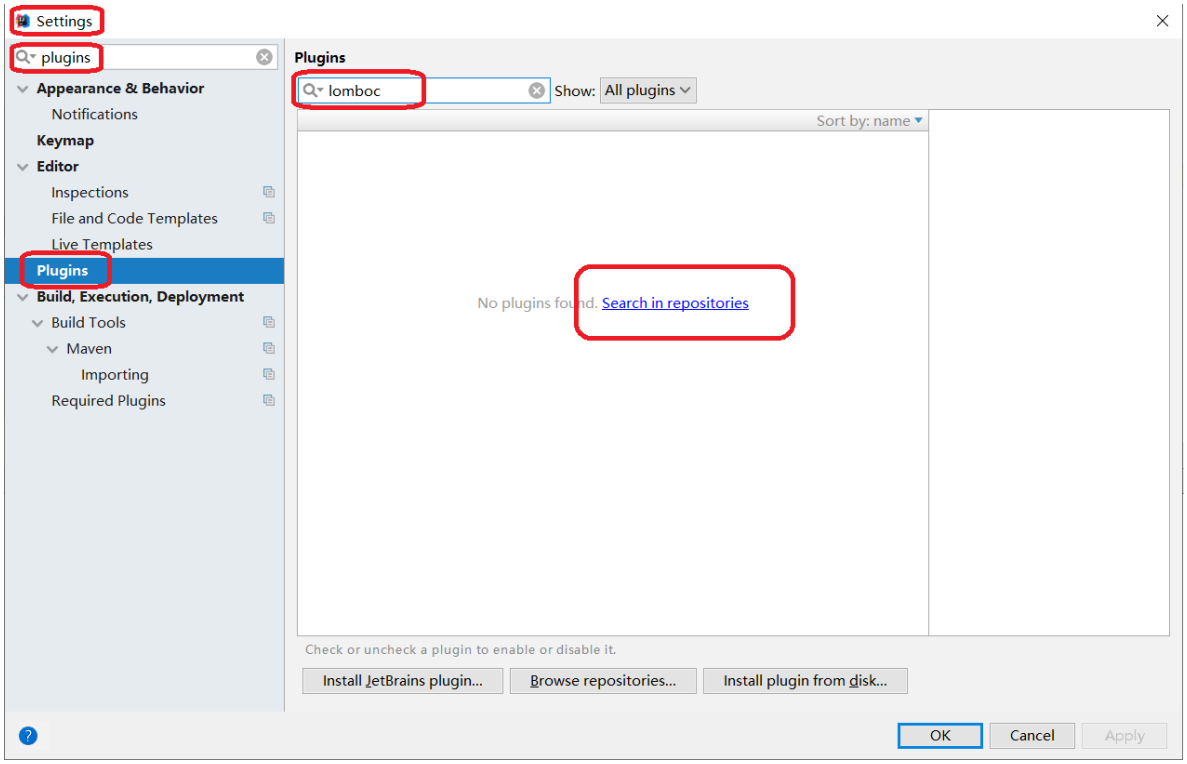
1. 添加lombox的jar包：

将lombok.jar(本例使用版本：1.18.10)，添加到模块目录下，并添加到ClassPath

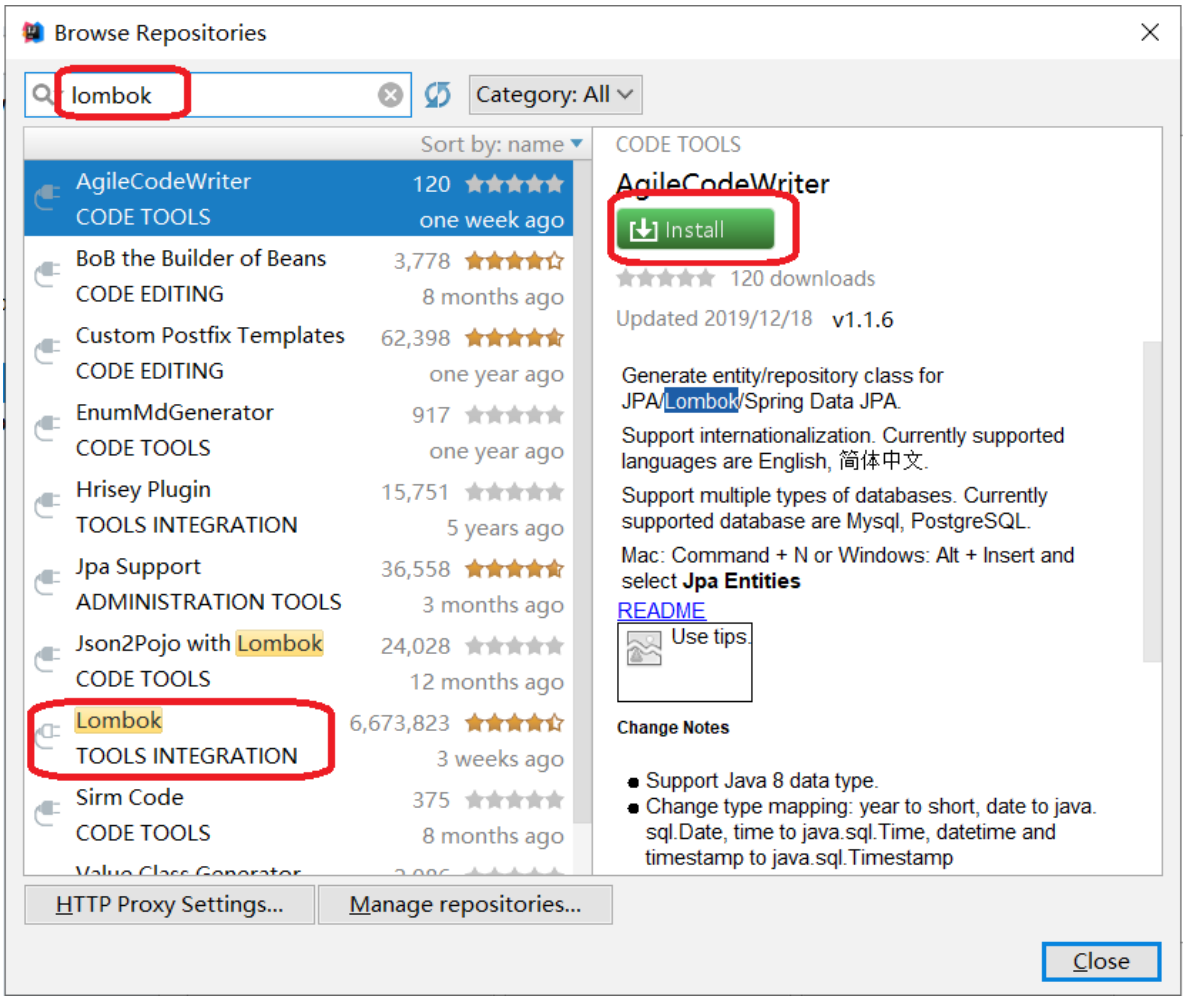



2. 为IDEA添加lombok插件（连接网络使用）

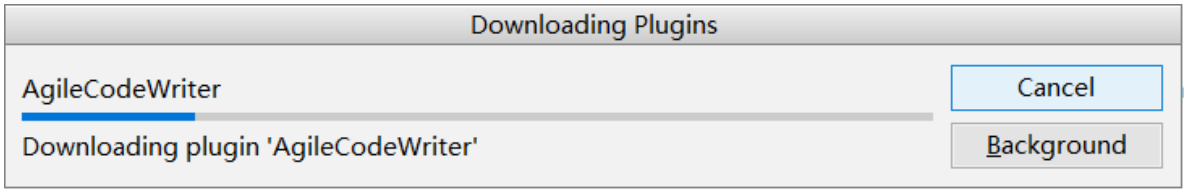
- 第一步


Settings
plugins
Appearance & Behavior
Notifications
Keymap
Editor
Inspections
File and Code Templates
Live Templates
Plugins
Build, Execution, Deployment
Build Tools
Maven
Importing
Required Plugins
Plugins
lomboc
Show: All plugins
Sort by: name
No plugins found. Search in repositories
Check or uncheck a plugin to enable or disable it.
Install JetBrains plugin...
Browse repositories...
Install plugin from disk...
OK
Cancel
Apply

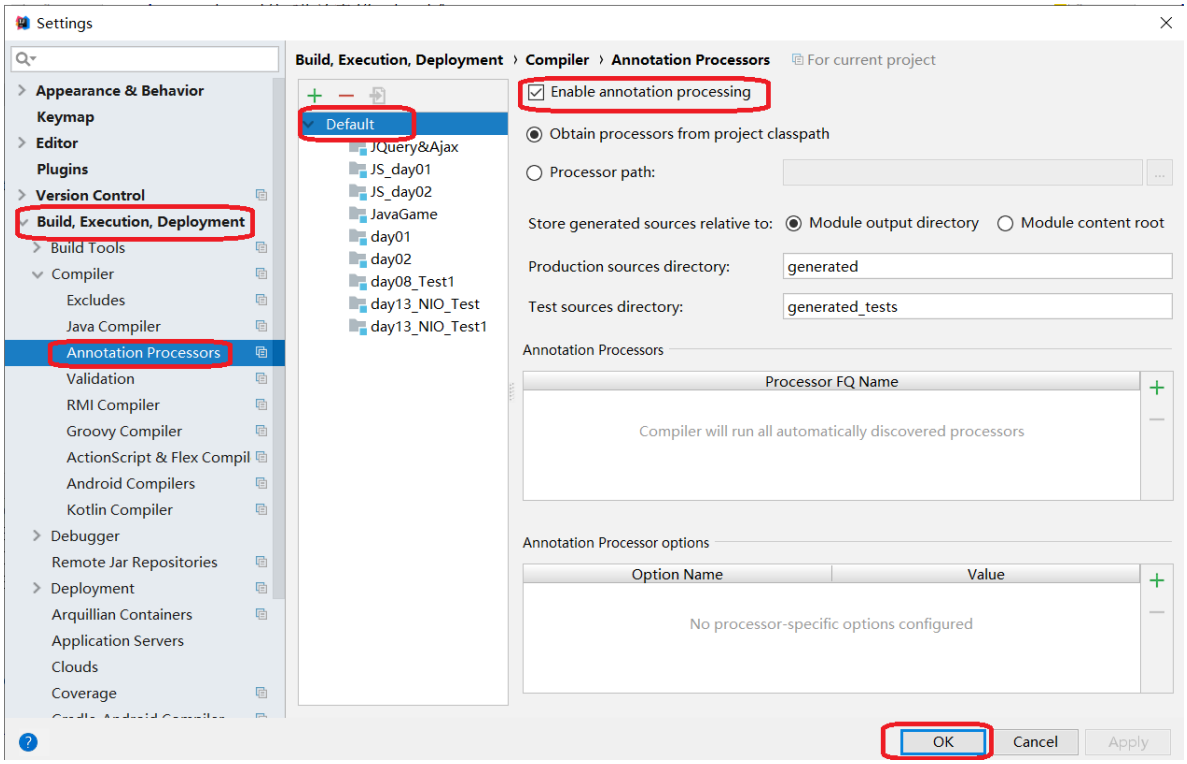
- 第二步：


Browse Repositories
lombok
Category: All
Sort by: name
AgileCodeWriter 120
CODE TOOLS one week ago
BoB the Builder of Beans 3,778
CODE EDITING 8 months ago
Custom Postfix Templates 62,398
CODE EDITING one year ago
EnumMdGenerator 917
CODE TOOLS one year ago
Hrisey Plugin 15,751
TOOLS INTEGRATION 5 years ago
Jpa Support 36,558
ADMINISTRATION TOOLS 3 months ago
Json2Pojo with Lombok 24,028
CODE TOOLS 12 months ago
Lombok 6,673,823
TOOLS INTEGRATION 3 weeks ago
Sirm Code 375
CODE TOOLS 8 months ago
CODE TOOLS
AgileCodeWriter
Install
120 downloads
Updated 2019/12/18 v1.1.6
Generate entity/repository class for JPA/Lombok/Spring Data JPA.
Support internationalization. Currently supported languages are English, 简体中文.
Support multiple types of databases. Currently supported database are Mysql, PostgreSQL.
Mac: Command + N or Windows: Alt + Insert and select Jpa Entities
README
Use tips.
Change Notes
Support Java 8 data type.
Change type mapping: year to short, date to java.sql.Date, time to java.sql.Time, datetime and timestamp to java.sql.Timestamp
HTTP Proxy Settings...
Manage repositories...
Close

- 第三步：


Downloading Plugins
AgileCodeWriter
Downloading plugin 'AgileCodeWriter'
Cancel
Background

- 第四步：




1. 安装完毕后，重启IDEA。
2. 新建一个类：Student

```java
import lombok.Data;

@Data
public class Student {
    private String name;
    private int age;
    private String sex;
}
```

```java
Student stu = new Student();
stu.setName("张三");
stu.setAge(21);
stu.setSex("男");
System.out.println(stu);
```

## 4.3 lombok常用注解

- @Getter和@Setter
    - 作用：生成成员变量的get和set方法。
    - 写在成员变量上，只对当前成员变量有效。
    - 写在类上，对所有成员变量有效。
    - 注意：静态成员变量无效。
- @ToString：
    - 作用：生成toString()方法。
    - 该注解只能写在类上。
- @NoArgsConstructor和@AllArgsConstructor
    - @NoArgsConstructor：无参数构造方法。
    - @AllArgsConstructor：满参数构造方法。
    - 该注解只能写在类上。
- @EqualsAndHashCode
    - 作用：生成hashCode()和equals()方法。
    - 该注解只能写在类上。
- @Data
    - 作用： 生成setter/getter、equals、canEqual、hashCode、toString方法，如为final属性，则不会为该属性生成setter方法。
    - 该注解只能写在类上。

# 第五章 工厂设计模式

## 5.1概述

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。之前我们创建类对象时, 都是使用new 对象的形式创建, 除new对象方式以外, 工厂模式也可以创建对象.

## 5.2作用

解决类与类之间的耦合问题

## 5.3实现步骤

1. 编写一个Car接口, 提供run方法
2. 编写一个Falali类实现Car接口,重写run方法
3. 编写一个Benchi类实现Car接口
4. 提供一个CarFactory(汽车工厂),用于生产汽车对象
5. 定义CarFactoryTest测试汽车工厂

## 5.4实现代码

1.编写一个Car接口, 提供run方法

```
public interface Car {
    public void run();
}
```

2.编写一个Falali类实现Car接口,重写run方法

```
public class Falali implements Car {
    @Override
    public void run() {
        System.out.println("法拉利以每小时500公里的速度在奔跑.....");
    }
}
```

3.编写一个Benchi类实现Car接口

```
public class Benchi implements Car {
    @Override
    public void run() {
        System.out.println("奔驰汽车以每秒1米的速度在挪动.....");
    }
}
```

4.提供一个CarFactory(汽车工厂),用于生产汽车对象

```
public class CarFactory {
    /**
     * @param id : 车的标识
     *           benchi : 代表需要创建Benchi类对象
     *           falali : 代表需要创建Falali类对象
     *           如果传入的车标识不正确,代表当前工厂生成不了当前车对象,则返回null
     * @return
     */
    public Car createCar(String id){
        if("falali".equals(id)){
            return new Falali();
        }else if("benchi".equals(id)){
            return new Benchi();
        }
        return null;
    }
}
```

5.定义CarFactoryTest测试汽车工厂

```
public class CarFactoryTest {
    public static void main(String[] args) {
        CarFactory carFactory = new CarFactory();
        Car benchi = carFactory.createCar("benchi");
        benchi.run();
        Car falali = carFactory.createCar("falali");
        falali.run();
    }
}
```

# 5.5小结

工厂模式的存在可以改变创建类的方式,解决类与类之间的耦合.

实现步骤:

1. 编写一个Car接口, 提供run方法
2. 编写一个Falali类实现Car接口,重写run方法
3. 编写一个Benchi类实现Car接口
4. 提供一个CarFactory(汽车工厂),用于生产汽车对象
5. 定义CarFactoryTest测试汽车工厂