# 建造者模式(Builder模式)

参考：java.effect.docx 1.2

当类有多个属性时，可以考虑构造方法重载，也可以考虑java bean的方式，只提供一个无参构造方法，对必要属性提供setter方法。这时就无法保证类的不可变，因为可以在不同地方调用setter方法。

构造器重载方式

```java

public class Product {

    private int id;

    private String name;

    private int type;

    private float price;

    public Product() {

        super();

    }

    public Product(int id) {

        super();

        this.id = id;

    }

    public Product(int id, String name) {

        super();

        this.id = id;

        this.name = name;

    }

    public Product(int id, String name, int type) {

        super();

        this.id = id;

        this.name = name;

        this.type = type;

    }

    public Product(int id, String name, int type, float price) {

        super();

        this.id = id;

        this.name = name;

        this.type = type;

        this.price = price;

    }

}

```

JavaBeans方式

```java

public class Product2 {

    private int id;

    private String name;

    private int type;

    private float price;

    public int getId() {

        return id;

    }

    public void setId(int id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public int getType() {

        return type;

    }

    public void setType(int type) {

        this.type = type;

    }

    public float getPrice() {

        return price;

    }

    public void setPrice(float price) {

        this.price = price;

    }

}

Product2 p2 = new Product2();

p2.setId(10);

p2.setName("phone");

p2.setPrice(100);

p2.setType(1);

```

缺点：

构造过程被分到了几个调用中，在构造过程中JavaBeans可能处于不一致的状态，类无法仅仅通过检验构造器参数的有效性来保证一致性。

builder模式

```java

public class Product3 {

    private final int id;

    private final String name;

    private final int type;

    private final float price;

    private Product3(Builder builder) {

        this.id = builder.id;

        this.name = builder.name;

        this.type = builder.type;

        this.price = builder.price;

    }

    public static class Builder {

        private int id;

        private String name;

        private int type;

        private float price;

        public Builder id(int id) {

            this.id = id;

            return this;

        }

        public Builder name(String name) {

            this.name = name;

            return this;

        }

        public Builder type(int type) {

            this.type = type;

            return this;

        }

        public Builder price(float price) {

            this.price = price;

            return this;

        }

        public Product3 build() {

            return new Product3(this);

        }

    }

}

builder模式将属性定义为不可变的，然后定义一个内部静态类Builder来构建属性，再通过一个只有Builder参数的构造器来生成Product对象。Builder的setter方法返回builder本身，以便可以将属性连接起来。

Product3 p3 = new Product3.Builder()

                .id(10)

                .name("phone")

                .price(100)

                .type(1)

                .build();

```

当然具体使用builder的情况肯定没有这么简单，但是思路大致一样：

先通过某种方式取得构造对象需要的所有参数，再通过这些参数一次性构建这个对象。

比如MyBatis中SqlSessionFactoryBuilder就是通过读取MyBatis的xml配置文件来获取构造SqlSessionFactory所需要的参数。

参考：

<https://www.cnblogs.com/scuwangjun/p/9699895.html>

# 单例模式

饿汉式

```java

public class Singleton {

    private static Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {

        return instance;

    }

}

```

没有线程安全问题，但是没有懒加载效果。

懒汉式

```java

public class Singleton {

    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {

        if (instance == null) {

            instance = new Singleton();

        }

        return instance;

    }

}

```

实现了懒加载，但是由于编译优化进行的指令重排序，会导致线程安全问题。

改进：

```java

public class SingletonSafe {

    private static volatile SingletonSafe singleton;

    private SingletonSafe() {}

    public static SingletonSafe getSingleton() {

        if (singleton == null) {

            synchronized (SingletonSafe.class) {

                if (singleton == null) {

                    singleton = new SingletonSafe();

                }

            }

        }

        return singleton;

    }

}

```

双重检查加锁，保证只能有一个线程进入到第二个非空判断，并且volatile保证了没有指令重排。

静态内部类

```java

public class Singleton {

    private static class SingletonHolder {

        private static Singleton instance = new Singleton();

    }

    private Singleton() {}

    public static Singleton getInstance() {

        return SingletonHolder.instance;

    }

}

```

线程安全，因为类加载过程中JVM保证了类加载过程中静态域只会执行一次。

通过枚举

```java

public enum Singleton {

    INSTANCE;

    public void doSomething() {

        System.out.println("doSomething");

    }

}

调用方法：

public class Main {

    public static void main(String[] args) {

        Singleton.INSTANCE.doSomething();

    }

}

```

枚举的JDK实现保证了线程安全和单例特性，不受反射、序列化、克隆的影响单例效果。

破坏单例模式的三种方式

1. 反射

2. 序列化

3. 克隆

代码：

```java

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import java.lang.reflect.Constructor;

/\*\*

 \* 对单例的破坏

 \*/

public class SingletonTest {

    public static void main(String[] args) throws Exception{

        System.out.println("-----------序列化----------------------");

        Singleton originSingleton = Singleton.getInstance();

        ByteArrayOutputStream bos = new ByteArrayOutputStream();

        ObjectOutputStream oos = new ObjectOutputStream(bos);

        oos.writeObject(Singleton.getInstance());

        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());

        ObjectInputStream ois = new ObjectInputStream(bis);

        Singleton serializeSingleton = (Singleton) ois.readObject();

        System.out.println(originSingleton == serializeSingleton); // false

        System.out.println("-----------反射----------------------");

        Constructor<Singleton> cons = Singleton.class.getDeclaredConstructor();

        cons.setAccessible(true);

        Singleton reflextSingleton = cons.newInstance();

        System.out.println(reflextSingleton == originSingleton); // false

        System.out.println("---------------------------克隆----------------------");

        Singleton cloneSingleton = (Singleton) originSingleton.clone();

        System.out.println(cloneSingleton == originSingleton); // false

    }

    private static class Singleton implements Serializable, Cloneable {

        private Singleton() {}

        private static volatile Singleton instance;

        public static Singleton getInstance() {

            if(instance == null) {

                synchronized (Singleton.class) {

                    if(instance == null) {

                        instance = new Singleton();

                    }

                }

            }

            return instance;

        }

        @Override

        protected Object clone() throws CloneNotSupportedException {

            return super.clone();

        }

    }

}

```

解决方法

防止反射:

定义一个全局变量，当第二次创建的时候抛出异常

防止克隆破坏

重写clone(),直接返回单例对象

防止序列化破坏

添加readResolve(),返回Object对象

代码：

```java

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import java.lang.reflect.Constructor;

/\*\*

 \* 对单例的破坏

 \* 改进，防止被破坏

 \*/

public class SingletonTest1 {

    public static void main(String[] args) throws Exception{

        System.out.println("-----------序列化----------------------");

        Singleton originSingleton = Singleton.getInstance();

        ByteArrayOutputStream bos = new ByteArrayOutputStream();

        ObjectOutputStream oos = new ObjectOutputStream(bos);

        oos.writeObject(Singleton.getInstance());

        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());

        ObjectInputStream ois = new ObjectInputStream(bis);

        Singleton serializeSingleton = (Singleton) ois.readObject();

        System.out.println(originSingleton == serializeSingleton); // true

        System.out.println("-----------反射----------------------");

        try {

            Constructor<Singleton> cons = Singleton.class.getDeclaredConstructor();

            cons.setAccessible(true);

            Singleton reflextSingleton = cons.newInstance();

            System.out.println(reflextSingleton == originSingleton); // 报错

        } catch (Exception ex) {

            System.out.println(ex); // java.lang.reflect.InvocationTargetException

        }

        System.out.println("---------------------------克隆----------------------");

        Singleton cloneSingleton = (Singleton) originSingleton.clone();

        System.out.println(cloneSingleton == originSingleton); // true

    }

    private static class Singleton implements Serializable, Cloneable {

        private static volatile boolean isCreate = false;

        private Singleton() {

            if(isCreate) {

                throw new RuntimeException("已然被实例化一次，不能在实例化");

            }

            isCreate = true;

        }

        private static volatile Singleton instance;

        public static Singleton getInstance() {

            if(instance == null) {

                synchronized (Singleton.class) {

                    if(instance == null) {

                        instance = new Singleton();

                    }

                }

            }

            return instance;

        }

        @Override

        protected Object clone() throws CloneNotSupportedException {

            return instance;

        }

        private Object readResolve() {

            return instance;

        }

    }

}

```

或者用commons.lang包，提供了序列化和反序列化工具类

```java

<dependency>

    <groupId>org.apache.commons</groupId>

    <artifactId>commons-lang3</artifactId>

    <version>3.8.1</version>

</dependency>

public static void main(String[] args) {

    Singleton instance = Singleton.getInstance();

    byte[] serialize = SerializationUtils.serialize(instance);

    Singleton newInstance = SerializationUtils.deserialize(serialize);

    System.out.println(instance == newInstance);

}

```

应用

datasource, session等

java.lang.Runtime 就是经典的单例模式(饿汉式)

```java

package java.lang;

public class Runtime {

    private static Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime() {

        return currentRuntime;

    }

    /\*\* Don't let anyone else instantiate this class \*/

    private Runtime() {}

    ...

}

```

# 3. 适配器模式

是一种结构型设计模式。

适配器模式涉及3个角色：

源（Adaptee）：需要被适配的对象或类型

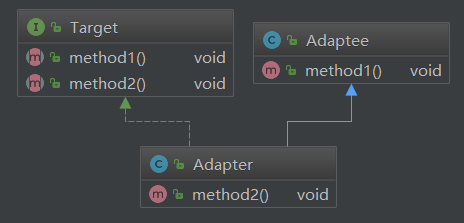
适配器（Adapter）：连接目标和源的中间对象

目标（Target）：期待得到的目标

1. 类适配器

类Targetee只有method1()，我们需要一个类同时具备method1()、method2()

通过继承原有类，并实现相应的接口，可以扩展类的的功能。



```java

需要适配的类：

public class Adaptee {

    public void method1(){

        System.out.println("method 1");

    }

}

实现的目标类：

public interface Target {

    void method1();

    void method2();

}

适配器类

public class Adapter extends Adaptee implements Target {

    @Override

    public void method2() {

        System.out.println("method 2");

    }

}

// 测试

class AdapterTest {

    public static void main(String[] args) {

        Adapter adapter = new Adapter();

        adapter.method1();

        adapter.method2();

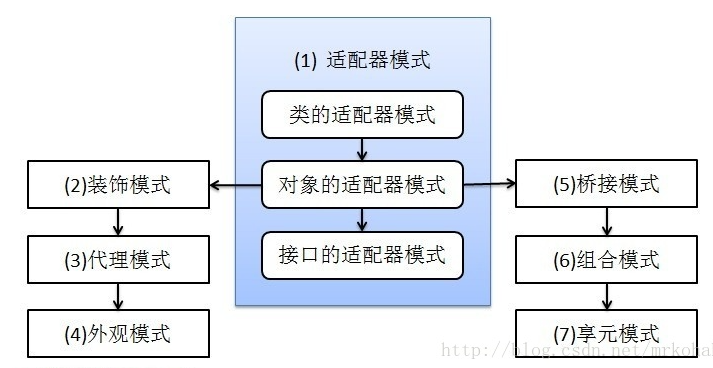
    }

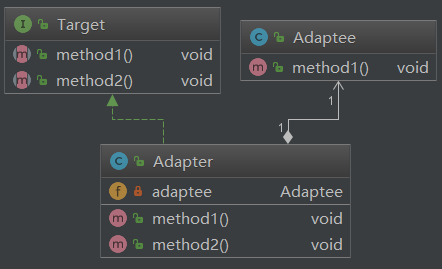
}

```

2. 对象适配器

对象适配器模式是另外6种结构型设计模式的起源。





使用组合来实现：

```java

public class Adapter implements Target {

    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {

        this.adaptee = adaptee;

    }

    @Override

    public void method1() {

        adaptee.method1();

    }

    @Override

    public void method2() {

        System.out.println("method 2");

    }

}

class AdapterTest {

    public static void main(String[] args) {

        Adapter adapter = new Adapter(new Adaptee());

        adapter.method1();

        adapter.method2();

    }

}

```

基于组合/聚合优于继承的原则，使用对象适配器是更好的选择。但是具体问题应该具体分析。凡事不是绝对的。

3. 接口适配器（缺省适配模式）

如果一个接口有很多方法，实现这个接口时需要实现所有的方法，很多方法不是必须的。

这时如果提供一个接口的默认实现类，然后通过继承这个类来进行扩展。

所以，为一个接口提供缺省实现，这样子类可以从这个缺省实现进行扩展，而不必从原有接口进行扩展。

```java

public interface Target {

    void f1();

    void f2();

    void f3();

}

public abstract class Adapter implements Target{

    public void f1() {}

    public void f2() {}

    public void f3() {}

}

public class MyClass extends Adapter{

    public void f1() {

        // do something

    }

}

class AdapterTest {

    public static void main(String[] args) {

        Adapter adapter = new MyClass();

        adapter.f1();

        adapter.f2();

        adapter.f3();

    }

}

```

适配器模式在JDK中的应用：

1. JDK源码的IO模块用到，例如

java.io.InputStreamReader(InputStream)

java.io.OutputStreamWriter(OutputStream)

2. mybatis源码日志模块用到对象适配器模式

3. awt模块中的应用

java.awt.KeyListener是一个键盘监听器接口，我们把这个接口的实现类对象注册进容器后，这个容器就会对键盘行为进行监听:

```java

package java.awt.event;

import java.util.EventListener;

public interface KeyListener extends EventListener {

    public void keyTyped(KeyEvent e);

    public void keyPressed(KeyEvent e);

    public void keyReleased(KeyEvent e);

}

public static void main(String[] args) {

    JFrame frame = new JFrame();

    frame.addKeyListener(new KeyListener() {

        @Override

        public void keyTyped(KeyEvent e) {}

        @Override

        public void keyPressed(KeyEvent e) {

            System.out.println("hey geek!");

        }

        @Override

        public void keyReleased(KeyEvent e) {

        }

    });

}

适配器：java.awt.KeyAdapter

package java.awt.event;

public abstract class KeyAdapter implements KeyListener {

    public void keyTyped(KeyEvent e) {}

    public void keyPressed(KeyEvent e) {}

    public void keyReleased(KeyEvent e) {}

}

public static void main(String[] args) {

    JFrame frame = new JFrame();

    frame.addKeyListener(new KeyAdapter() {

        @Override

        public void keyPressed(KeyEvent e) {

            System.out.println("fxcku!");

        }

    });

}

```

在任何时候，如果不准备实现一个接口里的所有方法时，就可以使用“缺省适配模式”制造一个抽象类，实现所有方法，这样，从这个抽象类再继承下去的子类就不必实现所有的方法，只要重写需要的方法就可以了。

适配器可以带来好的扩展性和复用性，但是也导致了系统的复杂性。