创建型模式

单例模式

单例模式的定义:一个类只允许创建一个对象（或者叫实例），那这个类就是一个单例类，这种设计模式就叫作单例设计模式，简称单例模式

单例模式的作用:从业务概念上，有些数据在系统中只应该保存一份，就比较适合设计为单例类。比如，系统的配置信息类。除此之外，我们还可以使用单例解决资源访问冲突的问题

单例的实现:

饿汉式:饿汉式的实现方式，在类加载的期间，就已经将 instance 静态实例初始化好了，所以，instance 实例的创建是线程安全的。不过，这样的实现方式不支持延迟加载实例。

```
public class Singleton1 {
    private static  Singleton1 INSTANCE = new Singleton1();
    //私有构造器以防被其他类new
    private Singleton1(){}
    public static Singleton1 getInstance(){
        if (INSTANCE == null) {
            INSTANCE = new Singleton1();
        }
        return INSTANCE;
    }
}
```

懒汉式:懒汉式相对于饿汉式的优势是支持延迟加载

```
public class Singleton1 {
    private static  Singleton1 INSTANCE = new Singleton1();
    //私有构造器以防被其他类new
    private Singleton1(){}
    public static Singleton1 getInstance(){
        if (INSTANCE == null) {
            INSTANCE = new Singleton1();
        }
        return INSTANCE;
    }
}
```

双重加锁检测:饿汉式不支持延迟加载，懒汉式有性能问题，不支持高并发。那我们再来看一种既支持延迟加载、又支持高并发的单例实现方式，也就是双重检测实现方式。在这种实现方式中，只要 instance 被创建之后，即便再调用 getInstance() 函数也不会再进入到加锁逻辑中了。

```
public class Singleton3 {
    private volatile static Singleton3 INSTANCE;
    private Singleton3(){}
    public static Singleton3 getInstance(){
        //节约多次访问volatile变量的消耗
        Singleton3 tempInstance = INSTANCE;
        if (tempInstance == null) {
```

```
            synchronized (Singleton3.class){
                if (tempInstance == null) {
                    tempInstance = new Singleton3();
                }
            }
        }
        return tempInstance;
    }
}
```

静态内部类:INSTANCECLASS是一个静态内部类，当外部类 Singleton4被加载的时候，并不会创建INSTANCECLASS实例对象。只有当调用 getInstance() 方法时，INSTANCECLASS才会被加载，这个时候才会创建 INSTANCE。INSTANCE的唯一性、创建过程的线程安全性，都由JVM 来保证。所以，这种实现方法既保证了线程安全，又能做到延迟加载。

```
public class Singleton4 {
    private Singleton4() {
    }
    static class INSTANCECLASS {
        private static final Singleton4 INSTANCE = new Singleton4();
    }
    public static Singleton4 getInstance(){
        return INSTANCECLASS.INSTANCE;
    }
}
```

枚举法:Java 枚举类型本身的特性，只能被实例化一次,保证了实例创建的线程安全性和实例的唯一性

```
public enum Singleton5{
    INSTANCE;
}
```

单例模式的缺点:

对OOP的开发模式不友好,例如ID生成器,如果后期需要针对不同的业务类型生成不同的ID,那么修改代码的地方就会很多

单例的替代解决方案:

为了保证全局唯一，除了使用单例，我们还可以用静态方法来实现。不过，静态方法这种实现思路，并不能解决我们之前提到的问题。如果要完全解决这些问题，我们可能要从根上，寻找其他方式来实现全局唯一类了。比如，通过工厂模式、IOC 容器（比如 Spring IOC 容器）来保证，由程序员自己来保证（自己在编写代码的时候自己保证不要创建两个类对象）。

实现线程唯一,集群唯一单例

线程唯一:使用ThreadLocal存储对应单例实例即可,自己实现则是用HashMap存储线程与实例的引用关系

集群唯一:我们需要把这个单例对象序列化并存储到外部共享存储区（比如文件,Redis等）。进程在使用这个单例对象的时候，需要先从外部共享存储区中将它读取到内存，并反序列化成对象，然后再使用，使用完成之后还需要再存储回外部共享存储区。为了保证任何时刻在进程间都只有一份对象存在，一个进程在获取到对象之后，需要对对象加锁，避免其他进程再将其获取。在进程使用完这个对象之后，需要显式地将对象从内存中删除，并且释放对对象的加锁。

为什么单例唯一性的作用范围是类加载器（Class Loader）

单例唯一性的作用范围是进程，实际上，对于 Java 语言来说，单例类对象的唯一性的作用范围并非进程，而是类加载器（Class Loader）

classloader有两个作用：1. 用于将class文件加载到JVM中；2. 确认每个类应该由哪个类加载器加载，并且也用于判断JVM运行时的两个类是否相等。
双亲委派模型的原理是当一个类加载器接收到类加载请求时，首先会请求其父类加载器加载，每一层都是如此，当父类加载器无法找到这个类时（根据类的全限定名称），子类加载器才会尝试自己去加载。
所以双亲委派模型解决了类重复加载的问题， 比如可以试想没有双亲委派模型时，如果用户自己写了一个全限定名为java.lang.Object的类，并用自己的类加载器去加载，同时BootstrapClassLoader加载了rt.jar包中的JDK本身的java.lang.Object，这样内存中就存在两份Object类了，此时就会出现很多问题，例如根据全限定名无法定位到具体的类。有了双亲委派模型后，所有的类加载操作都会优先委派给父类加载器，这样一来，即使用户自定义了一个java.lang.Object，但由于BootstrapClassLoader已经检测到自己加载了这个类，用户自定义的类加载器就不会再重复加载了。所以，双亲委派模型能够保证类在内存中的唯一性。
联系到课后的问题，所以用户定义了单例类，这样JDK使用双亲委派模型加载一次之后就不会重复加载了，保证了单例类的进程内的唯一性，也可以认为是classloader内的唯一性。当然，如果没有双亲委派模型，那么多个classloader就会有多个实例，无法保证唯一性。

JDK中使用单例模式的实例

```
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() {}
}
//获取cpu数
int N = Runtime.getRuntime().availableProcessors()
```

工厂模式

简单工厂模式

定义了一个创建对象的类，由这个类来封装实例化对象的行为

假如要根据不同的类型创建不同的pizza类,原始的方式创建是在业务代码中进行if else直接创建

```
public  void factoryPizza(String type) {
        BasePizza pizza = null;
        if ("A".equals(type)) {
            pizza = new PizzaA();
        } else if ("B".equals(type)) {
            pizza = new PizzaB();
        }else{
            throw new RuntimeException("没有符合条件的pizza");
        }
        pizza.prepare();
        System.out.println("pizza 已成功准备好!");
    }
```

使用简单工厂模式,就是将创建类的这一部分工作给剥离出来,用一个单独职责的类负责创建具体实例类

```
public class SimplePizzaFactory {

    public static BasePizza createPizza(String type){
        if ("A".equals(type)) {
            return new PizzaA();
        } else if ("B".equals(type)) {
            return new PizzaB();
        }else{
            throw new RuntimeException("没有符合条件的pizza");
        }
    }
}
```

工厂方法模式

当每个对象的创建逻辑都比较复杂的时候，为了避免设计一个过于庞大的简单工厂类时，将创建逻辑拆分得更细，我们需要使用到多态,抽象出一个基本的工厂类,然后每个对象的创建逻辑独立到各自的工厂类中.

```
//声明一个工厂接口
public interface IBasePizzaFactory {
    BasePizza createPizza();
}
//具体工厂实现
public class PizzaAFactory implements IBasePizzaFactory {
    @Override
    public BasePizza createPizza() {
        PizzaA pizzaA = new PizzaA();
        pizzaA.setName("工厂方法pizzaA");
        return pizzaA;
    }
}
public class PizzBFactory implements IBasePizzaFactory {
    @Override
    public BasePizza createPizza() {
        PizzaB pizzaB = new PizzaB();
        pizzaB.setName("工厂方法pizzaA");
        return pizzaB;
    }
}
//具体创建pizza工厂类
public class MethodFacotry {
    public BasePizza createPizza(String type){
        if ("A".equals(type)) {
            return new PizzaAFactory().createPizza();
        } else if ("B".equals(type)) {
            return new PizzBFactory().createPizza();
        }else{
            throw new RuntimeException("没有符合条件的pizza工厂类");
        }
    }
}
```

抽象工厂模式

我们可以让一个工厂负责创建多个不同类型的对象,这样就可以有效地减少工厂类的个数,相当于围绕一个超级工厂创建其他工厂

```
//超级工厂
public interface IBaseAbsPizzaFactory {
    BasePizza createPizza(String type);
    BaseTackOutWay createTackOutWay(String type);
}
public class SuperFactory implements IBaseAbsPizzaFactory {
    @Override
    public BasePizza createPizza(String type) {
        if ("A".equals(type)) {
            return new PizzaA();
        } else if ("B".equals(type)) {
            return new PizzaB();
        }else{
            throw new RuntimeException("没有符合条件的pizza");
        }
    }

    @Override
    public BaseTackOutWay createTackOutWay(String type) {
        if ("elm".equals(type)) {
            return new ElmTackOut();
        } else if ("mt".equals(type)) {
            return new MtTackOut();
        }else{
            throw new RuntimeException("没有符合条件的pizza");
        }
    }
}
```

总结

使用场景:

- **简单工厂**:当每个对象的创建逻辑都比较简单的时候，将多个对象的创建逻辑放到一个工厂类中
- **工厂方法模式**:当每个对象的创建逻辑都比较复杂的时候，为了避免设计一个过于庞大的简单工厂类时，将创建逻辑拆分得更细，每个对象的创建逻辑独立到各自的工厂类中,并且需要避免if else的时候
- **抽象工厂模式**:当需要创建的对象是一系列相互关联或相互依赖的产品族时，便可以使用抽象工厂模式

JDK中应用源码

java.util.Calendar类,这里使用的就是简单工厂模式

```
    public static Calendar getInstance()
    {
        return createCalendar(TimeZone.getDefault(),
Locale.getDefault(Locale.Category.FORMAT));
    }
private static Calendar createCalendar(TimeZone zone,
                                       Locale aLocale)
```

```
    {
        CalendarProvider provider =
            LocaleProviderAdapter.getAdapter(CalendarProvider.class, aLocale)
                                 .getCalendarProvider();
        if (provider != null) {
            try {
                return provider.getInstance(zone, aLocale);
            } catch (IllegalArgumentException iae) {
                // fall back to the default instantiation
            }
        }

        Calendar cal = null;

        if (aLocale.hasExtensions()) {
            String caltype = aLocale.getUnicodeLocaleType("ca");
            if (caltype != null) {
                switch (caltype) {
                case "buddhist":
                cal = new BuddhistCalendar(zone, aLocale);
                    break;
                case "japanese":
                    cal = new JapaneseImperialCalendar(zone, aLocale);
                    break;
                case "gregory":
                    cal = new GregorianCalendar(zone, aLocale);
                    break;
                }
            }
        }
        if (cal == null) {
            // If no known calendar type is explicitly specified,
            // perform the traditional way to create a Calendar:
            // create a BuddhistCalendar for th_TH locale,
            // a JapaneseImperialCalendar for ja_JP_JP locale, or
            // a GregorianCalendar for any other locales.
            // NOTE: The language, country and variant strings are interned.
            if (aLocale.getLanguage() == "th" && aLocale.getCountry() == "TH") {
                cal = new BuddhistCalendar(zone, aLocale);
            } else if (aLocale.getVariant() == "JP" && aLocale.getLanguage() ==
"ja"
                       && aLocale.getCountry() == "JP") {
                cal = new JapaneseImperialCalendar(zone, aLocale);
            } else {
                cal = new GregorianCalendar(zone, aLocale);
            }
        }
        return cal;
    }
```

原型模式

基本介绍:

如果对象的创建成本比较大，而同一个类的不同对象之间差别不大（大部分字段都相同），在这种情况下，我们可以利用对已有对象（原型）进行复制（或者叫拷贝）的方式，来创建新对象，以达到节省创建时间的目的。这种基于原型来创建对象的方式就叫作原型设计模式，简称原型模式。

而拷贝对象又有浅拷贝与深拷贝两种区别,Object 类的 clone() 方法执行的就是浅拷贝。它只会拷贝对象中的基本数据类型的数据（比如，int、long），以及引用对象的内存地址，不会递归地拷贝引用对象本身。

原型模式的浅拷贝

直接使具体实例类继承Cloneable接口,就可以使用clone方法浅拷贝相应对象

```
public class Sheep implements Cloneable {
    private String name;
    private int age;
    private String color;
    private String address = "蒙古羊";
    public Sheep friend;
    //克隆该实例，使用默认的clone方法来完成
    @Override
    protected Object clone()  {
        Sheep sheep = null;
        try {
            sheep = (Sheep)super.clone();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        return sheep;
    }
```

原型模式的深拷贝-重写clone方法

如果需要使用深拷贝的话,针对引用对象需要特别做处理,例如

```
public class Sheep implements Cloneable {
    private String name;
    private int age;
    private String color;
    private String address = "蒙古羊";
    public Sheep friend;
    //克隆该实例，使用默认的clone方法来完成
    @Override
    protected Object clone()  {
        Sheep sheep = null;
        try {
            sheep = (Sheep)super.clone();
            //引用变量特殊处理
            sheep.friend = new Sheep();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        return sheep;
    }
```

原型模式的深拷贝-序列化与反序列化

```
public Object deepClone() {
    ByteArrayOutputStream bo = new ByteArrayOutputStream();
    ObjectOutputStream oo = new ObjectOutputStream(bo);
    oo.writeObject(object);
    ByteArrayInputStream bi = new ByteArrayInputStream(bo.toByteArray());
    ObjectInputStream oi = new ObjectInputStream(bi);
    return oi.readObject();
    }
```

总结

原型模式有两种实现方法，深拷贝和浅拷贝。浅拷贝只会复制对象中基本数据类型数据和引用对象的内存地址，不会递归地复制引用对象，以及引用对象的引用对象……而深拷贝得到的是一份完完全全独立的对象。所以，深拷贝比起浅拷贝来说，更加耗时，更加耗内存空间

原型模式在Spring中的应用

我们知道spring的scope作用域是可以根据不同的配置来指定bean的作用域,如果我们将对应bean的作用域设置为prototype,那么每次getBean的时候都会返回一个新的实例,底层就是根据原型模式来创建的

在`org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean`方法中会判断bean的作用域是什么

建造者模式

基本介绍:建造者模式又称为生成器模式,是一种对象构建模式,将复杂对象的建造过程抽象出来,建造者模式是一步一步创建一个复杂的对象

假设有一个对象,拥有四个属性,如果需要创建该对象并对对象进行一系列的校验,如果不使用建造者模式,我们只能用构造器或者set方法来设置相应属性与校验

```
public class ResourceConfig {
    private String name;
    private Integer maxTotal;
    private Integer maxIdle;
    private Integer minIdle;

    public ResourceConfig(String name ,Integer maxTotal,Integer maxIdle,Integer
minIdle){
        if (StringUtils.isEmpty(name)) {
            throw new RuntimeException("资源名称不能为空");
        }
        this.name = name;
        if (maxTotal == null || maxTotal < 0) {
            throw new RuntimeException("maxTotal参数错误,maxTotal不能 < 0");
        }
        this.maxTotal = maxTotal;
        if (maxIdle == null || maxIdle < 0 || maxIdle > maxTotal) {
            throw new RuntimeException("maxIdle参数错误,maxIdle不能 < 0 或者 >
maxTotal");
        }
        this.maxIdle = maxIdle;
        if (minIdle == null || minIdle < 0 || minIdle > maxIdle) {
            throw new RuntimeException("minIdle参数错误,minIdle不能 < 0 或者 >
maxIdle");
        }
```

```
        this.minIdle = minIdle;
    }

    public void setName(String name) {
        if (StringUtils.isEmpty(name)) {
            throw new RuntimeException("资源名称不能为空");
        }
        this.name = name;
    }

    public void setMaxTotal(Integer maxTotal) {
        if (maxTotal == null || maxTotal < 0) {
            throw new RuntimeException("maxTotal参数错误,maxTotal不能 < 0");
        }
        this.maxTotal = maxTotal;
    }

    public void setMaxIdle(Integer maxIdle) {
        if (maxIdle == null || maxIdle < 0 || maxIdle > maxTotal) {
            throw new RuntimeException("maxIdle参数错误,maxIdle不能 < 0 或者 >
maxTotal");
        }
        this.maxIdle = maxIdle;
    }

    public void setMinIdle(Integer minIdle) {
        if (minIdle == null || minIdle < 0 || minIdle > maxIdle) {
            throw new RuntimeException("minIdle参数错误,minIdle不能 < 0 或者 >
maxIdle");
        }
        this.minIdle = minIdle;
    }
}
```

使用建造者模式:

```
public class ResourceConfig {
    private String name;
    private Integer maxTotal;
    private Integer maxIdle;
    private Integer minIdle;

    private ResourceConfig(){

    }

    static class ResourceConfigBuilder {
        private String name;
        private Integer maxTotal;
        private Integer maxIdle;
        private Integer minIdle;

        public ResourceConfigBuilder setName(String name) {
            this.name = name;
            return this;
        }
```

```
        public ResourceConfigBuilder setMaxTotal(Integer maxTotal) {
            this.maxTotal = maxTotal;
            return this;
        }

        public ResourceConfigBuilder setMaxIdle(Integer maxIdle) {
            this.maxIdle = maxIdle;
            return this;
        }

        public ResourceConfigBuilder setMinIdle(Integer minIdle) {
            this.minIdle = minIdle;
            return this;
        }

        public  ResourceConfig build() {
            if (StringUtils.isEmpty(name)) {
                throw new RuntimeException("资源名称不能为空");
            }
            if (maxTotal == null || maxTotal < 0) {
                throw new RuntimeException("maxTotal参数错误,maxTotal不能 < 0");
            }
            if (maxIdle == null || maxIdle < 0 || maxIdle > maxTotal) {
                throw new RuntimeException("maxIdle参数错误,maxIdle不能 < 0 或者 >
maxTotal");
            }
            if (minIdle == null || minIdle < 0 || minIdle > maxIdle) {
                throw new RuntimeException("minIdle参数错误,minIdle不能 < 0 或者 >
maxIdle");
            }
            ResourceConfig resourceConfig = new ResourceConfig();
            resourceConfig.name = this.name;
            resourceConfig.maxIdle = this.maxIdle;
            resourceConfig.minIdle = this.minIdle;
            resourceConfig.maxTotal = this.maxTotal;
            return resourceConfig;
        }
    }
}
public static void main(String[] args) {
        ResourceConfig resourceConfig =
                new ResourceConfig.ResourceConfigBuilder().
                        setName("test").setMaxTotal(100)
                        .setMaxIdle(55).setMinIdle(50).build();
    }
```

最终复杂的对象创建过程可以被分层化,并且使用建造者模式可以避免对象的无效状态,如果使用set方法,会使对象在一段时间内处于无效状态.

```
Rectangle r = new Rectange(); // r is invalid
r.setWidth(2); // r is invalid
r.setHeight(3); // r is valid
```