# **1. 对象的创建和销毁**

## 1.1 有多个构造器的时候，优先考虑用静态工厂方法来创建对象

1. 方法有名称，可以确切描述返回对象信息

2. 可以实现每次调用都返回同一个对象

Boolean.valueOf

枚举类的实现就提供了静态工厂方法来创建对象，而不是构造方法，保证了 if Enum.A == Enum.A Enum.A.equals(Enum.A)

所以利用枚举可以实现单例。

3. 可以实现返回类型的任何子类型对象

借鉴了接口编程，提高了类的扩展性。例如JDBC的实现，不同数据库类型提供自己的驱动和相关方法的实现。

可以返回私有类的对象。例如java.util.Collections类中方法返回的对象的类都是非公有的。

静态工厂方法一些惯用名称：

valueOf

of

getInstance

newInstance

getType

newType

类型推导

Map<String, List<String>> map = new HashMap<String, List<String>>();

Map<String, List<String>> map = new HashMap<String, List<>>();

public static Boolean valueOf(boolean f) {

    return b ? Boolean.TRUE : Boolean.FALSE;

}

## **1.2 类的构造器或者静态工厂方法里有多个参数时，考虑Builder模式实现。**

下面代码演示了当有多个参数，有些参数可选，有些参数必选的情况下，提供多种构造方法：

```java

/\*\*

 \* 提供多个构造方法的实现

 \*/

public class Test {

    private int a1; // required

    private int a2; // required

    private int a3; // non-required

    private int a4; // non-required

    // 第一种方式

    public Test(int a1) {

        this(a1,0,0,0);

    }

    public Test(int a1, int a2) {

        this(a1,a2,0,0);

    }

    public Test(int a1, int a2, int a3) {

        this(a1,a2,a3,0);

    }

    public Test(int a1, int a2, int a3, int a4) {

        this.a1 = a1;

        this.a2 = a2;

        this.a3 = a3;

        this.a4 = a4;

    }

}

```

毫无疑问，没有任何扩展性，可读性极差。

利用Java Bean的方式来实现

```java

/\*\*

 \* Java Bean模式

 \*/

public class Test1 {

    private int a1; // required

    private int a2; // required

    private int a3 = 0; // non-required

    private int a4 = 1; // non-required

    public Test1() {}

    public void setA1(int a1) {

        this.a1 = a1;

    }

    public void setA2(int a2) {

        this.a2 = a2;

    }

    public static void main(String[] args) {

        Test1 t = new Test1();

        t.setA1(1);

        t.setA2(2);

    }

}

```

这种调用方式无法实现不可变对象。

Builder模式：

```java

/\*\*

 \* Builder模式

 \*/

 public class Test2 {

     public static void main(String[] args) {

         Test2 t = new Test2.Builder(1, 2).setA1(3).setA2(4).build();

     }

     private final int a1; // required

     private final int a2; // required

     private final int a3; // non-required

     private final int a4; // non-required

     static class Builder {

         private int a1; // required

         private int a2; // required

         private int a3 = 0; // non-required

         private int a4 = 1; // non-required

         public Builder(int a1, int a2) {

             this.a1 = a1;

             this.a2 = a2;

         }

         public Builder setA1(int a1) {

             this.a1 = a1;

             return this;

         }

         public Builder setA2(int a2) {

             this.a2 = a2;

             return this;

         }

         public Test2 build() {

             return new Test2(this);

         }

         public <T> T builder(T t) {

             return t;

         }

     }

     public Test2(Builder builder) {

         this.a1 = builder.a1;

         this.a2 = builder.a2;

         this.a3 = builder.a3;

         this.a4 = builder.a4;

     }

}

```

builder是构建类的静态成员。

1.5开始，提供了一个接口来标识builder类型

```java

package javafx.util;

/\*\*

 \* Interface representing a builder. Builders are objects that are used to

 \* construct other objects.

 \*

 \* @since JavaFX 2.0

 \*/

@FunctionalInterface

public interface Builder<T> {

    /\*\*

     \* Builds and returns the object.

     \*/

    public T build();

}

```

Builder模式也叫建造者模式。

## 1.3 用私有构造器或枚举类型强化singleton属性

结论：

单元素的枚举类型是实现单例的最佳方法。

枚举无偿提供了序列化机制，绝对防止多次实例化，即使是面对复杂的序列化或者反射攻击。

现状：

光有私有构造器无法保证绝对的单例，利用反射setAccessible可以访问到私有方法的。

另外，在多线程模式下，要注意线程安全问题。

还有一点，序列化过程中，无法保证对象的唯一性。

为了保证序列化之后的唯一性，所有实例域都是transient，并且提供readResolve方法

// readResolve method to preserve singleton property

private Object readResolve() {

    return INSTANCE;

}

详情见 java-pattern.docx 单例模式

## 1.4 通过私有构造器强化不可实例化的能力

对于一些不需要实例化的类，例如java.util.Arrays，java.util.Collections，如果没有一个私有构造方法，由于jvm默认提供一个无参构造器，所以为了避免类在任何情况下不被实例化，提供一个私有构造器来进行强化

```java

// Suppresses default constructor, ensuring non-instantiability.

private Arrays() {}

// Suppresses default constructor, ensuring non-instantiability.

private Collections() {}

public class UtilityClass {

    // Suppresses default constructor, ensuring non-instantiability.

    private UtilityClass() {

        throw new AssertionError();

    }

}

```

坏处：

这个类就不能被子类化，因为子类是调用构造方法时，所有的构造方法都会显式或者隐式调用父类的构造方法。

## **1.5 避免创建不必要的对象**

String s = new String("ss"); // don't do this!!

String s = "ss";

慎用对象池，除非对象非常重量级。比如数据库连接池。因为创建连接代价昂贵。

一般对象的创建和gc是非常廉价的，为了代码清晰，创建对象是必要的。

优先使用基本数据类型，避免无意识的自动装箱。

## 1.6 消除过期的对象引用

jdk中Stack的代码：

```java

package java.util.Stack;

public synchronized E pop() {

    E       obj;

    int     len = size();

    obj = peek();

    removeElementAt(len - 1);

    return obj;

}

package java.util.Vector;

public synchronized void removeElementAt(int index) {

    modCount++;

    if (index >= elementCount) {

        throw new ArrayIndexOutOfBoundsException(index + " >= " +

                                                 elementCount);

    }

    else if (index < 0) {

        throw new ArrayIndexOutOfBoundsException(index);

    }

    int j = elementCount - index - 1;

    if (j > 0) {

        System.arraycopy(elementData, index + 1, elementData, index, j);

    }

    elementCount--;

    elementData[elementCount] = null; /\* to let gc do its work \*/

}

```

过期的引用比较隐蔽。会导致内存泄漏（Memory Leak），最后可能引起OOM。

只能通过检查代码或者借助Heap工具（Heap Profiler）才能发现内存泄漏问题。

比较典型的场景就是栈先增长，然后再收缩。

消除过期引用的最好方式是让包含该引用的变量结束掉生命周期。

内存泄漏的场景还包括缓存。

## 1.7 避免使用finalizer()

对象在被回收前可能会执行的动作，jvm规范没有规定该方法一定会执行，也没有保证这个方法执行的时机。

对象被回收前，最后一个能拯救的动作，简单的做法是在这个方法里重新引用该对象。

该方法会导致性能问题。对象的创建销毁耗时大大增加。

不应该依赖这个方法去做一些关键性的动作，比如文件的关闭，锁的释放等。

慎用System.gc、System.runFinalization()，并不能保证finalizer()一定会被执行。

System.runFinalizersOnExit()、Runtime.runFinalizersOnExit()。废弃，不能用。

```java

package java.lang.Object

protected void finalize() throws Throwable { }

```

gc线程的优先级是比较低的，无法确切知道何时gc。无法控制。

# **所有对象通用的方法**

3. 类和接口

4. 泛型

5. 枚举和注解

6. 方法

7. 通用的程序设计

8. 异常处理

9. 并发

10. 序列化