# **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**

## 2.8 覆盖equals()方法时遵守通用约定

能不覆盖equals()就不覆盖，覆盖错了是要出大问题的。

以下情况就不需要覆盖了：

1. 类的每个实例都保证了唯一，比如Thread

2. 不关心类的对象是否相等，比如java.util.Random

3. 超类已经覆盖了，子类一般不需要再覆盖

4. 类是私有或者包级私有，可以确定equals()永远不会被调用

```java

package java.lang;

public class Object {

    public boolean equals(Object obj) {

        return (this == obj);

    }

    public native int hashCode();

}

```

遵守约定：

1. 自反性，x.equals(x) == true

2. 对称性, x.equals(y) == y.equals(x)

3. 传递性

4. 一致性, 多次调用结果一致

5. 对于非null, x.equals(null) == false

违反这些约定，会发生不可预知的错误。

里式替换原则认为：一个类型的任何重要属性也将适用于他的子类型，为该类型编写的任何方法，在子类型上也应该运行良好。

覆盖equals()和hashCode()、toString()时，一定要注明@Override。

IDEA中生成的equals()和hashCode()、toString()

```java

import java.util.Arrays;

import java.util.Objects;

public class A1 {

    private int a1;

    private int a2;

    private String s1;

    private boolean b1;

    private int[] arr1;

    private float f1;

    @Override

    public boolean equals(Object o) {

        if (this == o) return true;

        if (!(o instanceof A1)) return false;

        A1 a11 = (A1) o;

        return a1 == a11.a1 &&

                a2 == a11.a2 &&

                b1 == a11.b1 &&

                Float.compare(a11.f1, f1) == 0 &&

                Objects.equals(s1, a11.s1) &&

                Arrays.equals(arr1, a11.arr1);

    }

    @Override

    public int hashCode() {

        int result = Objects.hash(a1, a2, s1, b1, f1);

        result = 31 \* result + Arrays.hashCode(arr1);

        return result;

    }

    @Override

    public String toString() {

        return "A1{" +

                "a1=" + a1 +

                ", a2=" + a2 +

                ", s1='" + s1 + '\'' +

                ", b1=" + b1 +

                ", arr1=" + Arrays.toString(arr1) +

                ", f1=" + f1 +

                '}';

    }

}

```

## 2.9 覆盖equals()时总要覆盖hashCode()

没有覆盖hashCode()可能会在使用集合时有意想不到的错误。

正确覆盖hashCode()，才能得到很高的散列性能。

假如这样写

```java

@Override

public int hashCode() {

    return 31;

}

这时所有对象都有相同的散列码，每个对象都被映射到一个相同的散列桶中，这时，散列表退化成链表。性能严重下降。

好的散列函数应该把不同对象均匀分布到不同的散列桶里。

```

习惯用素数来计算散列值，一般用31，因为31有个很好的特性，可以用移位和减法来代替乘法，可以得到很好的性能，31 \* i == (i << 5) - i。现代的VM可以自动完成这种优化。

## 2.10 永远覆盖toString()

不是那么重要。

当对象传递给println、printf、字符串操作符+、assert或者各种被打印时，会自动调用toString()。

好的覆盖，可以在调试时得到更多有用的信息。

## 2.11 谨慎覆盖clone()

覆盖之后，意味着多了一个途径来得到类的实例。稍有错误会导致原有实例的改变，这个就涉及到浅拷贝和深拷贝。

拷贝不彻底的话会引起意想不到的错误。

拷贝的过程中，没有调用构造器。

如果有复杂的继承关系的类，更要慎用。或者不要用。

更多见 java-primary1.docx 深拷贝和浅拷贝

```java

public class A1 implements Cloneable {

    @Override

    protected Object clone() throws CloneNotSupportedException {

        return super.clone();

    }

}

```

## 2.12 考虑实现Comparable接口

使用了这个接口后，类的比较规则就定下来了，后续这个类的比较规则就很难再去扩展。

实现了Comparable接口的类，可以用在Arrays.sort和Collections.sort中了。

推荐使用Comparator，实现比较器来定义比较规则。

更多见 java-primary1.docx Comparable和Comparator

Arrays.sort(list, Comparator)

Collections.sort(list, Comparator)

# 类和接口

## 3.13 使类和成员的访问最小化

更好的封装，意味着隐藏更多的细节。

对外暴露的越多，问题越不好定位，不好维护。每一次升级都头疼。

更强的模块性。降低了大型系统的风险，毕竟不给外部暴露，外部就很难就使用到。一切问题就集中到内部了。

访问控制权限：

public

protected

default

private

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 本类 | 同包 | 子类（同包/不同包） | 不同包 |
| public | 1 | 1 | 1 | 1 |
| protected | 1 | 1 | 1 |  |
| default | 1 | 1 |  |  |
| private | 1 |  |  |  |

接口的默认修饰符：public abstract

除了公有的静态final域之外，公有类都不应该包含公有的域。并且要确保公有静态final域引用的对象是不可变的。

## 3.14 在公有类中使用访问方法，而不是公有域

封装。

公有类永远都不应该暴露可变的域。

不管是为了以后类的域的作用域改变来考虑，都应该最小化作用域的范围。

```java

public class A1 {

    private int a1;

    private int a2;

    public int getA1() {

        return a1;

    }

    public void setA1(int a1) {

        this.a1 = a1;

    }

    public int getA2() {

        return a2;

    }

    public void setA2(int a2) {

        this.a2 = a2;

    }

}

```

## 3.15 使可变性最小化

使类成为不可变的类，可以：

1. 不提供任何修改对象状态的方法

2. 保证类不被扩展，一般声明为final，可以有其它做法

3. 使所有域都是final的

4. 使所有域都是私有的

5. 确保对任何可变组件的互斥访问，比如对象调用一个方法修改了对象的属性，这时可以返回一个新的对象，确保原有对象不变。

坚决不用对每一个get()都编写一个set()。除非有好的理由让类成为可变的类，否则就应该是不可变的。

## 3.16 复合优先于继承

继承可以很好实现重用。但是继承打破了封装性。

使用复合可以带来很好的封装性和扩展性。

只有当A 真正 is B时，才考虑用继承。否则复合是优选。

## 3.17 要么设计继承并提供文档说明，要么就禁用继承

继承中的覆盖动作一定要谨慎，特别是方法之间的调用关系一定要有完善的文档说明。负责对于使用者来说很容易掉坑里。

好的文档API说明应该描述一个给定的方法做了什么工作，而不是怎么实现的。

## 3.18 接口优先于抽象类

接口是定义混合(Mixin)类型的最优选择。

接口能构造非层次结构的类型框架。

## 3.19 接口只用于定义类型

常量接口是对接口的不良使用。不应该被用来导出常量。

## 3.20 类层次优于标签

类的嵌套过多就要考虑继承\接口\组合来改造。重构成一个层级结构中。

## 3.21 用函数对象表示策略

遗憾的是java中没有函数对象。函数不能作为一等公民。

## 3.22 优先考虑内部静态成员类

嵌套类有4种：

1. 静态成员类

2. 非静态成员类

3. 匿名类

4. 局部类

非静态成员类的实例都要与外部类的实例进行关联，要不然实例不出来，这样就多出来额外的开销。有关联就有了耦合。

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