Effective Java

作者：yaochengfly
原文链接：https://ld246.com/article/1479521730635
来源网站：链滴

Effective Java

---

1. 用静态工厂方法代替构造器

优势：

- 静态工厂方法有名称
- 不必每次调用的时候都创建一个新对象
- 可以返回原返回类型的任何子类型对象
- 创建参数化类型实例时，代码更加简洁

劣势：

- 如果不含有public或protect的构造器，就不能被子类化
- 他们与其他静态方法没有实际的区别，不容易被javadoc工具注意到

2. 遇到多个构造器参数是要考虑用构建器

举个栗子：

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).calories(100).sodium(35).carboh
drate(27).build();

public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int catbohydrate;

    public static class Builder {
        private final int servingSize;
        private final int servings;

        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }

        public Builder calories(int val) {
            calories = val;
            return this;
```

```
        }
        public Builder fat(int val) {
            fat = val;
            return this;
        }
        public Builder carbohydrate(int val) {
            carbohydrate = val;
            return this;
        }
        public Builder sodium(int val) {
            sodium = val;
            return this;
        }

        public NuritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

3. 用私有构造器或者枚举类型强化Singleton属性

- 公有静态final域
- 公有静态工厂方法，私有静态final域
- 包含单个元素的枚举类型

4. 通过私有构造器强化不可实例化的能力

举个栗子：

```
public lcass UtilituClass {
    private UtilityClass() {
        throw new AssertionError();
    }
    ...
}
```

5. 避免创建不必要的对象

本条内容不是暗示创建对象的代价非常昂贵！

6. 消除过期对象的引用

自己管理内存的类容易引起内存泄漏

7. 避免使用终结方法

劣势：

- 终结方法不保证被执行
- 性能损失

代替方案 try-final结构，显式终止方法，必须在一个私有域中记录“该对象已经不再有效”

可以被考虑使用的情况：

- 用作安全网，当显示的终止方法忘记被调用时，提供一定程度的安全保障
- 本地对等实体在不拥有关键资源情况下的回收

注意：

终结方法链不会被自动执行，使用方法应如下

```
try {
    ...
} finally {
    super.finalize();
}
```

或者为了防止子类使用错误方法导致忘记调用父类终结方法时，可以考虑创建一个终结方法守卫者

```
public class Foo {
    private final Object finalizerGuardian = new Object() {
        @Override protected void finalize() throws Throwable {
            ... //Finalize outer Foo Object
        }
    }
}
```

8. 覆盖equals时请遵守通用约定

- 自反性
- 对称性
- 传递性

注意：

类继承一个非抽象类容易违反这条

- 一致性
- 非空性

建议：

- 使用==操作符检测“参数是否为这个对象的引用”
- 使用instanceof操作符检查“参数是否为正确的类型”
- 把参数转换为正确的类型
- 对于该类中的每个“关键”域，检查参数中的域是否与该对象中对应的域相匹配
- 完成编写后测试是否是对称的、传递的、一致的
- 覆盖equals是总要覆盖hashCode
- 不要企图让equals方法过于智能
- 不要将equals声明中的Object对象替换为其他类型

9.覆盖equals时总要覆盖hashCode

10.始终要覆盖toString

11.谨慎的覆盖clone