《Effective Java (3rd)》读后感

《Effective Java (3rd)》是Joshua Bloch编写的一本介绍Java实战编程原则的技术书籍，具体内容分为11章，90 条最佳实践，涵盖了从对象创建、类设计、泛型、异常处理到并发等各个方面。

1. 最佳实践介绍

1.1 创建和销毁对象

1. **静态工厂方法替代构造器**
   提供命名更清晰的对象创建方式，支持缓存实例（如`Boolean.valueOf`）和灵活性。
2. **Builder模式解决多参数构造问题**
   适用于可选参数多、构造过程复杂的场景（如`StringBuilder`）。
3. **强化不可实例化**
   私有构造器阻止工具类（如`Math`）被实例化。
4. **依赖注入优于硬编码资源**
   提升灵活性和可测试性。
5. **避免创建重复对象**
   重用不可变对象（如`Pattern`编译的正则表达式）。
6. **消除过时对象引用**
   防止内存泄漏（如缓存、监听器需显式清理）。
7. **禁用finalize()**
   因执行时机不可控且性能差，替代方案为`AutoCloseable`接口。

1.2 对象通用方法

1. **覆盖`equals`、`hashCode`、`toString`**
   - `equals`需满足自反性、对称性、传递性、一致性。
   - `hashCode`必须与`equals`结果一致。
   - `toString`应返回直观信息，便于调试。
2. **谨慎覆盖`clone`**
   深拷贝需递归调用，建议使用拷贝构造器或工厂。
3. **`Comparable`接口实现排序**
   定义自然顺序，与`equals`一致避免逻辑矛盾。

1.3 类与接口设计

1. **最小化可访问性**
   使用`private`/`package-private`封装内部细节。
2. **不可变类的优势**
   线程安全且易于推理（如`String`），通过构造器或Builder创建。

3. **组合优于继承**
   避免继承破坏封装性（如`ForwardingSet`包装类）。
4. **接口定义类型**
   优先接口定义行为（如`List`），支持函数式接口与默认方法。
5. **静态成员类与匿名类**
   根据作用域选择嵌套类类型，优先静态成员类减少内存泄漏。

1.4 泛型

1. **消除原生类型警告**
   使用泛型（如`List<String>`）和`@SuppressWarnings`注解。
2. **列表优于数组**
   数组协变易导致运行时错误，泛型提供编译时类型安全。
3. **有限制通配符**
   `<? extends T>`和`<? super T>`提升API灵活性（PECS原则）。
4. **类型安全的异构容器**
   通过`Class`对象作为键实现类型安全的多类型存储。

1.5 枚举与注解

1. **枚举替代int常量**
   类型安全且支持行为关联（如`Operation.PLUS.apply(a, b)`）。
2. **EnumMap/EnumSet优化性能**
   基于枚举的键实现高效集合操作。
3. **注解定义元数据**
   替代命名模式（如JUnit 4的`@Test`），结合反射处理注解。

1.6 Lambda与Stream

1. **Lambda替代匿名类**
   简洁且延迟执行，适用于函数式接口（如`Comparator`）。
2. **方法引用提升可读性**
   `String::length`优于`str -> str.length()`。
3. **谨慎使用Stream**
   复杂数据处理时链式调用更清晰，但需避免滥用导致可读性下降。
4. **Optional替代null**
   显式处理空值，避免`NullPointerException`。

1.7 方法设计

1. **参数有效性检查**
   使用`Objects.requireNonNull`和断言提前失败。
2. **防御性拷贝**
   保护不可变对象免受外部修改（如`Date`返回克隆）。

3. **谨慎重载方法**
   避免因参数类型模糊导致调用歧义。
4. **返回空集合而非null**
   `Collections.emptyList()`避免客户端空指针检查。

1.8 通用编程

1. **最小化局部变量作用域**
   尽早声明并初始化，减少错误概率。
2. **for-each循环优于传统for**
   代码简洁且避免索引错误。
3. **基本类型优于装箱类型**
   避免不必要的性能开销和`NullPointerException`。
4. **字符串谨慎使用**
   不适合替代枚举、数值类型或聚合类型。

1.9 异常处理

1. **异常仅用于异常情况**
   避免控制流依赖异常（如循环终止条件）。
2. **受检异常与未受检异常**
   前者用于可恢复错误（如`IOException`），后者用于编程错误（如`NullPointerException`）。
3. **保持失败原子性**
   对象异常后应保持一致性状态。

1.10 并发

1. **同步访问共享数据**
   使用`synchronized`或并发工具（如`AtomicInteger`）。
2. **Executor框架管理线程**
   优于直接使用`Thread`类。
3. **并发工具库替代wait/notify**
   优先使用`java.util.concurrent`中的`CountDownLatch`、`ConcurrentHashMap`等。

1.11 序列化

1. **自定义序列化格式**
   通过`readObject`和`writeObject`控制序列化过程。
2. **防御性反序列化**
   校验反序列化对象有效性，防止非法构造。
3. **枚举实现单例序列化**
   避免反射攻击，天然保证单例。

2. 一些例子

2.1 使用静态工厂方法替代构造方法

```
public class Complex {
    private final double re;
    private final double im;

    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }

    // 其他方法...
}
```

在这个例子中，`Complex` 类使用静态工厂方法 `valueOf` 代替构造器来创建对象。这样做的好处是，可以根据不同的参数返回不同的实例，例如可以缓存一些常用的 `Complex` 对象，避免重复创建。

2.2 避免创建不必要的对象

```
String s = new String("abc"); // 错误：创建了一个不必要的 String 对象
String s = "abc"; // 正确：使用字符串字面量，避免创建不必要的对象
```

在这个例子中，第一行代码创建了一个新的 `String` 对象，而第二行代码使用了字符串字面量，它会从字符串常量池中获取已有的字符串对象，避免创建不必要的对象。

2.3 try-with-resources 优先于 try-finally

```
// 使用 try-with-resources
try (InputStream in = new FileInputStream("file.txt")) {
    // 读取文件内容...
} catch (IOException e) {
    // 处理异常...
}

// 使用 try-finally (不推荐)
InputStream in = null;
try {
    in = new FileInputStream("file.txt");
    // 读取文件内容...
} catch (IOException e) {
```

```
    // 处理异常...
} finally {
    if (in != null) {
        try {
            in.close();
        } catch (IOException e) {
            // 处理异常...
        }
    }
}
```

在这个例子中，`try-with-resources` 语句可以自动关闭 `InputStream`，避免资源泄漏。而 `try-finally` 语句需要手动关闭资源，代码比较繁琐，也容易出错。

2.4 优先考虑组合而不是继承

```
// 使用继承
public class MyList extends ArrayList<String> {
    // 添加自定义方法...
}

// 使用组合
public class MyList {
    private final ArrayList<String> list = new ArrayList<>();

    public void add(String s) {
        list.add(s);
    }

    // 添加自定义方法...
}
```

在这个例子中，`MyList` 类可以使用继承或组合来扩展 `ArrayList` 的功能。使用继承的问题是，`MyList` 类会继承 `ArrayList` 的所有方法和属性，包括一些不需要的方法和属性。而使用组合可以只使用 `ArrayList` 的部分功能，更加灵活。

2.5 接口优先于抽象类

```
// 使用接口
public interface MyInterface {
    void myMethod();
}

public class MyClass implements MyInterface {
    @Override
    public void myMethod() {
        // 实现方法...
```

```
    }
}

// 使用抽象类
public abstract class MyAbstractClass {
    public abstract void myMethod();
}

public class MyClass extends MyAbstractClass {
    @Override
    public void myMethod() {
        // 实现方法...
    }
}
```

在这个例子中，`MyClass` 类可以使用接口或抽象类来实现 `myMethod` 方法。使用接口的好处是，`MyClass` 类可以实现多个接口，从而具有多种不同的行为。而使用抽象类则只能继承一个抽象类。

2.6 优先考虑泛型方法

```
// 非泛型方法
public static Object max(Collection<?> c) {
    // ...
}

// 泛型方法
public static <T extends Comparable<? super T>> T max(Collection<? extends T> c) {
    // ...
}
```

在这个例子中，泛型方法 `max` 可以接受任何类型的集合，并返回该集合中的最大元素。而非泛型方法 `max` 则只能接受 `Collection<?>` 类型的集合，并且返回 `Object` 类型的对象，需要进行类型转换。

2.7 用枚举类型代替 int 常量

```
// 使用 int 常量
public static final int RED = 0;
public static final int GREEN = 1;
public static final int BLUE = 2;

// 使用枚举类型
public enum Color {
    RED, GREEN, BLUE
}
```