# **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。

# 泛型

## 4.23 尽量不要在代码中使用原生类型

比如

List l1 = new ArrayList();

如果使用了原生类型，就失掉了泛型在安全性和表述性的优势。

为了兼容老的代码，所以还支持原生类型的使用。

List是逃避了泛型检查，和List<Object>是完全不一样的类型。

List<String>和List<Object>也是完全不一样的类型，不是他的子类型。

```java

import java.util.ArrayList;

import java.util.List;

public class A3 {

    public static void main(String[] args) {

        List l1 = new ArrayList();

        List<Object> l2 = new ArrayList<>();

        List<String> l3 = new ArrayList<>();

        f1(l1); // ok

        f1(l2); // ok

        f1(l3); // ok

        f2(l1); // ok

//        f2(l3); // 编译报错

        f3(l1); // ok

//        f3(l2); // 编译报错

        f4(l1); // ok

        f4(l2); // ok

        f4(l3); // ok

        f5(l1); // ok

        f5(l2); // ok

        f5(l3); // ok

    }

    static void f1(List l1) {

        for (Object o : l1) {

            System.out.println(o);

        }

    }

    static void f2(List<Object> l1) {

        for (Object o : l1) {

            System.out.println(o);

        }

    }

    static void f3(List<String> l1) {

        for (String s : l1) {

            System.out.println(s);

        }

    }

    static <T> void f4(List<T> l1) {

        for (T o : l1) {

            System.out.println(o);

        }

    }

    static void f5(List<?> l1) {

        for (Object o : l1) {

            System.out.println(o);

        }

    }

}

```

## 4.24 消除非受检告警

非受检告警一定要尽量清掉。

如果确定告警代码是类型安全的，可以使用@SuppressWarnings("unchecked")来禁止掉。

但要谨慎使用，可能会把坑略过。

使用时一定要尽量的缩小范围，永远不要在整个类上使用，可能会掩盖重要的告警。

```java

/\*\*

 \* @SuppressWarnings("unchecked")只能用在声明上

 \*/

@SuppressWarnings("unchecked")

public class A4 {

    @SuppressWarnings("unchecked")

    private int a1;

    @SuppressWarnings("unchecked")

    void f1() {}

    void f2() {

        @SuppressWarnings("unchecked")

        int a2;

    }

    int f3() {

//        @SuppressWarnings("unchecked") // 不能用在return语句

//        return 1;

        @SuppressWarnings("unchecked")

        int a1 = 0;

        return a1;

    }

}

```

## 4.25 列表优先于数组

数组是协变的，比如Integer[]是Number[]的子类型。但是泛型是不可变的，比如List<Integer>就跟List<Object>是两个完全不同的类型，没有sub关系。

```java

import java.util.ArrayList;

import java.util.List;

public class A5 {

    public static void main(String[] args) {

        Integer[] a1 = new Integer[10];

        Number[] a2 = a1; // ok

        List<String> l1 = new ArrayList<>();

//        List<Object> l2 = l1; // 编译报错

        /\*

        编译不报错，但是运行时

        Exception in thread "main" java.lang.ArrayStoreException: java.lang.Double

    at com.wh.p1.A5.main(A5.java:12)

         \*/

        a2[0] = 0.1;

        System.out.println(a1[0]);

    }

}

```

数组只有在运行的时候，才会检查元素的类型约束。

创建泛型数组是非法的。详情见 java-primary1.docx 泛型

数组和泛型不能很好的结合使用。最好用列表代替数组。

## 4.26 优先考虑泛型

泛型可以扩展功能。

参考集合中泛型集合的实现。很多底层还是依赖数组来实现的。

## 4.27 优先考虑泛型方法

更好的扩展性，更高的类型安全保证。

## 4.28 利用限制通配符提升API的灵活性

?

? extends Number

? super Integer

```java

void f1(List<?> l1) {}

void f2(List<? extends Number> l1) {}

void f3(List<? super Number> l1) {}

```

类型推导很复杂。在java语言规范里就有16页。

## 4.29 优先考虑类型安全的异构容器

```java

package java.lang;

public final class Class<T> implements java.io.Serializable,

                              GenericDeclaration,

                              Type,

                              AnnotatedElement {

                              }

```

类的类型是一个泛型类型。

String.class的类型是Class<String>

Integer.class的类型是Class<Integer>

List<Integer>和List<String>的类型都是List.class。

集合API的泛型使用。

# 5. 枚举和注解

注解，接口类型 @interface

枚举，引用类型， extends Annotation

## 5.30 用enum代替常量

枚举类型是final类型。提供编译时的类型安全。

枚举类中声明抽象方法，每个枚举常量对抽象方法提供不同的实现。

```java

public enum Operation {

    PLUS("+") {

        @Override

        double apply(double x, double y) {return x + y;}

    },

    MINUS("-") {

        @Override

        double apply(double x, double y) {return x - y;}

    },

    TIMES("\*") {

        @Override

        double apply(double x, double y) {return x \* y;}

    },

    DIVIDE("/") {

        @Override

        double apply(double x, double y) {return x / y;}

    };

    private final String symbol;

    private Operation(String symbol) {

        this.symbol = symbol;

    }

    @Override

    public String toString() {

        return symbol;

    }

    abstract double apply(double x, double y);

}

```

如果有多个枚举常量共享相同的行为，考虑策略模式。

利用枚举实现策略模式

```java

public class A7 {

    enum PayrollDay {

        MONDAY(PayType.WEEKDAY),

        TUESDAY(PayType.WEEKDAY),

        WEDNESDAY(PayType.WEEKDAY),

        THURSDAY(PayType.WEEKDAY),

        FRIDAY(PayType.WEEKDAY),

        SATURDAY(PayType.WEEKEND),

        SUNDAY(PayType.WEEKEND);

        private PayType payType;

        PayrollDay(PayType payType) {

            this.payType = payType;

        }

        double pay(double hoursWorked, double payRate) {

            return payType.pay(hoursWorked, payRate);

        }

        enum PayType {

            WEEKDAY {

                double overtimePay(double hrs, double rate) {

                    return hrs <= HOURS\_PRE\_SHIFT ? 0 :

                            (hrs - HOURS\_PRE\_SHIFT) \* rate / 2;

                }

            },

            WEEKEND {

                double overtimePay(double hrs, double rate) {

                    return hrs \* rate / 2;

                }

            };

            private static final int HOURS\_PRE\_SHIFT = 8;

            abstract double overtimePay(double hrs, double rate);

            double pay(double hrs, double rate) {

                double basePay = hrs \* rate;

                return basePay + overtimePay(hrs, rate);

            }

        }

    }

}

```

一般来说，优先使用枚举而不是int，但是使用枚举类有装载和初始化的成本。除了一些嵌入式，不如手机等。

## 5.31 用实例域代替序数

每个枚举常量都有一个ordinal()，返回每个常量在类型中的位置。一般不要使用。

这个方法被设计成用于实现EnumSet和EnumMap这种基于枚举的结构。

```java

public class A8 {

    enum T1 {

        A,B,C;

        // DON'T DO THIS

        int numberOfEnum() {

            return ordinal() + 1;

        }

    }

    enum T2 {

        A(1),B(2),C(3);

        private final int val;

        T2(int val) {this.val = val;}

        int numberOfEnum() {

            return val;

        }

    }

}

```

## 5.32 用EnumSet代替位域

EnumSet的实现通过位域运算。

枚举类型用在集合Set中时，优先使用EnumSet。

```java

public class A9 {

    static final int A = 1 << 0; // 1

    static final int B = 1 << 0; // 2

    static final int C = 1 << 0; // 4

    static final int D = 1 << 0; // 8

    void apply(int a) {

        System.out.println(a);

    }

    public static void main(String[] args) {

        new A9().apply(D | A); // 1

    }

}

```

## 5.33 用EnumMap代替序数索引

枚举类型用在集合map中时，优先使用EnumMap。

```java

public class A9 {

    enum Phase {

        SOLID,LIQUID,GAS;

        enum Transition {

            MELT(SOLID, LIQUID),

            FREEZE(LIQUID, SOLID);

            private final Phase src;

            private final Phase dest;

            Transition(Phase src, Phase dest) {

                this.src = src;

                this.dest = dest;

            }

            private static final Map<Phase, Map<Phase, Transition>> m =

                    new EnumMap<Phase, Map<Phase, Transition>>(Phase.class);

            static {

                for (Phase p : Phase.values()) {

                    m.put(p, new EnumMap<Phase, Transition>(Phase.class));

                }

                for (Transition t : Transition.values()) {

                    m.get(t.src).put(t.dest, t);

                }

            }

            static Transition from(Phase src, Phase dest) {

                return m.get(src).get(dest);

            }

        }

    }

}

```

## 5.34 利用接口扩展枚举

枚举是通过继承Annotatin实现的，只能单继承，但是可以通过实现接口进行扩展。

```java

package com.wh.p1;

interface Operate {

    double apply(double x, double y);

}

enum Operation11 implements Operate {

    EXP("^") {

        @Override

       public double apply(double x, double y) {return Math.pow(x, y);}

    },

    MINUS("-") {

        @Override

        public double apply(double x, double y) {return x - y;}

    };

    private final String symbol;

    private Operation11(String symbol) {

        this.symbol = symbol;

    }

}

public class A6 {

    enum Operation implements Operate {

        PLUS("+") {

            @Override

            double apply(double x, double y) {return x + y;}

        },

        MINUS("-") {

            @Override

            double apply(double x, double y) {return x - y;}

        },

        TIMES("\*") {

            @Override

            double apply(double x, double y) {return x \* y;}

        },

        DIVIDE("/") {

            @Override

            double apply(double x, double y) {return x / y;}

        };

        private final String symbol;

        private Operation(String symbol) {

            this.symbol = symbol;

        }

        @Override

        public String toString() {

            return symbol;

        }

        abstract double apply(double x, double y);

    }

}

```

## **5.35 注解优先于命名模式**

自定义注解的用处。

详细见 java-primary1.docx 注解

## 5.36 坚持使用@Override

现代IDE可以使用。

提供编译时的检查。

## 5.37 用标记接口定义类型

没有包含方法声明的接口。

```java

package java.io;

public interface Serializable {

}

```

标记接口要胜过标记注解。

标记接口定义的类型是由被标记的类的实例实现的。标记注解则没有定义类型。这个类型提供编译时的检查，而标记注解则到运行时才体现出来。

标记注解有个优点是可扩展，而接口没办法添加方法。

```java

package java.lang.annotation;

@Documented

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.ANNOTATION\_TYPE)

public @interface Inherited {

}

```

如果想要定义类型，一定要使用接口。

# 6. 方法

## 6.38 检查参数有效性

非公有的方法通常利用assert来检查参数，断言失败，抛出AssertionError。

```java

void f1(int a, List l1) {

    assert a != 0;

    assert l1 != null;

}

```

## 6.39 必要时进行保护性拷贝

如果对象内部的属性是可变的，而且提供了对外访问，这个时候就要考虑是否使用保护性拷贝来保护属性不被外部修改。

```java

import java.util.Date;

import java.util.List;

public class A10 {

    public static void main(String[] args) {

        Date start = new Date();

        Date end = new Date();

        In1 n1 = new In1(start,end);

        end.setYear(2011); // n1的值也跟着变

    }

    static class In1 {

        private final Date start;

        private final Date end;

        public In1(Date start, Date end) {

            this.start = start;

            this.end = end;

        }

        public Date start() {return start;}

        public Date end() {return end;}

    }

    static class In2 {

        private final Date start;

        private final Date end;

        public In2(Date start, Date end) {

            this.start = new Date(start.getTime()); // 保护性拷贝

            this.end = new Date(end.getTime()); // 保护性拷贝

            if (this.start.compareTo(this.end) > 0) {

                throw new IllegalArgumentException(start+" after " + end);

            }

        }

        public Date start() {

            return new Date(start.getTime()); // 保护性拷贝

        }

        public Date end() {

            return new Date(end.getTime()); // 保护性拷贝

        }

    }

}

```

## 6.40 谨慎设计方法签名

方法名称。

避免过长的方法列表。尽量少于4个参数

参数类型，优先使用接口而不是类。

## 6.41 慎用重载

调用哪个重载的方法是在编译时决定的。

重载方法的选择是静态的，被覆盖的方法选择是动态的。

覆盖的方法选择是在运行时进行的。

永远不要重载两个具有相同参数数目的重载方法。

如果方法有可变参数就一定不要重载他。

可以使用给方法起不同的名称来避免重载。

```java

import java.util.ArrayList;

import java.util.Collection;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

public class A11 {

    static String f(Set<?> s) {

        return "set";

    }

    static String f(List<?> s) {

        return "list";

    }

    static String f(Collection<?> s) {

        return "collection";

    }

    static String f2(Collection<?> s) {

        return s instanceof Set ? "set" :

                s instanceof List ? "list" : "unknown collection";

    }

    public static void main(String[] args) {

        Collection<?>[] c = {

          new HashSet<String>(),

            new ArrayList<String>(),

            new HashMap<String, String>().values()

        };

        for (Collection<?> cc : c) {

            System.out.println(f(cc));

        }

        for (Collection<?> cc : c) {

            System.out.println(f2(cc));

        }

    }

}

```

```java

import java.util.ArrayList;

import java.util.List;

import java.util.Set;

import java.util.TreeSet;

public class A12 {

    public static void main(String[] args) {

        Set<Integer> s1 = new TreeSet<>();

        List<Integer> l1 = new ArrayList<>();

        for (int i=-3;i<3;i++) {

            s1.add(i);

            l1.add(i);

        }

        // set调用的是boolean remove(Object o);

        // list调用的是E remove(int index);

        for (int i=0;i<3;i++) {

            s1.remove(i);

            l1.remove(i);

            l1.remove(Integer.valueOf(i));

        }

        System.out.println(s1 + " " + l1);

    }

}

```

## 6.42 慎用可变参数

```java

static void sum(int... arg) {

    int res = 0;

    for (int i : arg) {

        res += i;

    }

    System.out.println(res);

}

```

可变参数每次调用都会进行一次数组分配和初始化。有性能损耗。

假设确定对某个方法95%的调用会有3个或者更好的参数，就声明5个重载，比如EnumSet.java，最大限度减少创建枚举集合的成本

## **6.43 返回零长度的数组或者集合，而不是null**

Collections.emptySet();

Collections.emptyMap();

Collections.emptyList();

## 6.44 为所有导出API元素编写文档注释

javadoc工具

@param

@throws

@return

支持html标签嵌套

可以学习jdk的注释规范

# 7. 通用程序设计

## 7.45 将局部变量的作用域最小化

最好将变量的声明放在距离使用他最近的地方。（编程规范）

几乎每个变量声明都应该包含一个初始化表达式。

在循环终止之后不再使用循环变量的话，for优先于while

## 7.46 for-each优先于传统的for循环

```java

推荐使用

for (E e : elements) {

    dosomething();

}

1.5之前使用

for (Iterator i = elements.Iterator(); i.hasNext()) {

    dosomething((Element) i.next())

}

for (int i=0;i<elements.size();i++) {

    dosomething(elements[i]);

}

```

实现java.util.Iterator接口或者java.util.Collection接口就可以使用for-each了

public interface Iterator<E> {}

public interface Collection<E> extends Iterable<E> {}

以下情况不能使用for-each：

1. 过滤集合，删除指定的元素

2. 循环中转换元素

3. 平行迭代

```java

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class B3 {

    public static void main(String[] args) {

        List<String> list = new ArrayList<String>();

        list.add("a");

        list.add("b");

        list.add("c");

        for (Iterator i = list.iterator(); i.hasNext();) {

            for (Iterator i1 = list.iterator(); i1.hasNext();) {

                System.out.println((String)i.next() + (String)i1.next());

            }

        }

        for (Iterator i = list.iterator(); i.hasNext();) {

            String outer = (String)i.next();

            for (Iterator i1 = list.iterator(); i1.hasNext();) {

                System.out.println(outer + (String)i1.next());

            }

        }

        for (String s : list) {

            for (String s1 : list) {

                System.out.println(s + s1);

            }

        }

        for (String s : list) {

            if ("a".equals(s)) {

                // list.remove(s); // 运行报错，Exception in thread "main" java.util.ConcurrentModificationException

                s = "aa"; // 无法修改

            }

        }

        System.out.println(list);

    }

}

```

## 7.47 了解和使用类库

几个重要的包

java.lang

java.util

java.io

java.util.concurrent

比如产生随机数：

```java

import java.util.Random;

public class B4 {

    private static final Random random = new Random();

    static int random1(int n) {

        return Math.abs(random.nextInt()) % n;

    }

    static int random2(int n) {

        return random.nextInt(n);

    }

    public static void main(String[] args) {

        for (int i=0;i<10;i++) {

            // System.out.println(random1(10)); // 不安全，不推荐使用

            System.out.println(random2(10));

        }

    }

}

```

## 7.48 精确的计算，不要使用float/double

推荐使用BigDecimal

看下例子：

```java

import java.math.BigDecimal;

public class B5 {

    public static void main(String[] args) {

        System.out.println(1.03 - .42); // 0.6100000000000001

        System.out.println(1.00 - 9 \* .10); // 0.09999999999999998

        f1();

        f2();

    }

    static void f1() {

        double f = 1.00;

        int n = 0;

        for (double p = .10;f >= p; p+= .10) {

            f -= p;

            n++;

        }

        System.out.println("bought: " + n);

        System.out.println("change: " + f);

        /\*\*

         \* bought: 3

         \* change: 0.3999999999999999

         \*/

    }

    final static BigDecimal CNET = new BigDecimal(".10");

    static void f2() {

        BigDecimal f = new BigDecimal("1.00");

        int n = 0;

        for (BigDecimal p = CNET;f.compareTo(p) >=0; p= p.add(CNET)) {

            f = f.subtract(p);

            n++;

        }

        System.out.println("bought: " + n);

        System.out.println("change: " + f);

        /\*\*

         \* bought: 4

         \* change: 0.00

         \*/

    }

}

```

## 7.49 基本类型优先于装箱类型

装箱类型是引用类型，有null值。

对装箱类型用 == 来比较基本都是错的。

当一个操作中混用了基本类型和装箱类型，就会自动进行拆箱，如果装箱类型为null，就会发生NullPointerException

只有在集合、参数化类型、反射方法调用时，必须使用装箱类型。

装箱有额外的资源开销。

## 7.50 尽量避免使用字符串

字符串不适合代替枚举类型

## 7.51 小心字符串连接的性能

避免大量的字符串 + 操作，这个操作会复制大量的字符串。

推荐使用StringBuilder，不要使用StringBuffer（内部同步操作，性能低）。

## 7.52 通过接口引用对象

如果有合适的接口存在，那么对于参数、返回值、变量和域来说，都应该使用接口类型进行声明。

程序会更加灵活。

## 7.53 接口优先于反射机制

反射失去了编译时的类型检查。

反射的代码不好阅读。

反射的性能损失。

## 7.54 谨慎使用本地方法

Java Native Interface（JNI）

使用本地方法的方式来提高性能不值得提倡。

比如BigInteger已经用java重写了。

本地方法给程序带来了复杂性，由于本地方法和平台有关，失去了可移植性。

## 7.55 谨慎地进行优化

要努力编写好的程序而不是快的程序。

在编写程序的时候注意性能问题。努力避免那些影响性能的决策。

大部分时间都是浪费时间尝试去优化，找到性能低的代码并不是一件容易的事。

## 7.56 遵守普遍接受的命名规范

包命名规范

# **8. 异常**

## 8.57 只针对异常的情况才使用异常

异常不应该被用于正常的控制流。

## 8.58 对可恢复的情况下使用受检异常，对编程错误使用运行时异常

详细见java-primary1.docx 异常

## 8.59 避免不必要地使用受检异常

## 8.60 优先使用标准的异常

异常类越少，意味着内存印迹越少，装载类的时间开销就越小。

常见的异常

NullPointerException

IndexOutOfBoundsException

IllegalArgumentException 非null的参数值不正确

IllegalStateException（对象未被正确初始化之前被调用）

ConcurrentModificationException（如果一个对象被设计为专用于单线程或者与外部同步机制配合使用，一旦发现正在或者已经被并发修改，就抛出这个异常）

UnsupportedOperationException

## 8.61 抛出与抽象相对应的异常

更高层的异常应该捕获底层的异常，同时抛出可以按照高层抽象进行解释的异常。称为“异常转译”。

“异常链”是一种特殊的异常转译形式。

new Throwable(String msg, Throwable cause)

## 8.62 每个方法抛出的异常都要有文档

Javadoc @throws

应该用具体的异常，而不是Exception（编程规范）

【这里有异常，如何兜底？】

要为每个受检异常提供单独的throws，不要为未受检的异常提供throws。

## 8.63 在细节消息中包含能捕获失败的消息

异常应该包含不敏感的消息，比如参数的值，变量的值（编程安全规范，工号密码，个人信息，敏感信息）

## 8.64 努力使失败保持原子性

失败的方法调用应该使对象保持在被调用之前的状态。具有这种属性的方法被称为失败原子性。

最简单的方式是设计一个不可变对象。

在可变对象操作之前，最常见的操作是先进行一系列的检验。

比如：

```

java.util.Stack

public synchronized E pop() {

    E       obj;

    int     len = size();

    obj = peek();

    removeElementAt(len - 1);

    return obj;

}

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 \*/

}

```

也可以进行保护性拷贝，在拷贝之后的对象上操作，然后再把数据代替原对象。

比如

```java

java.util.Collections

public static <T> void sort(List<T> list, Comparator<? super T> c) {

    list.sort(c);

}

java.util.List

default void sort(Comparator<? super E> c) {

    Object[] a = this.toArray();

    Arrays.sort(a, (Comparator) c);

    ListIterator<E> i = this.listIterator();

    for (Object e : a) {

        i.next();

        i.set((E) e);

    }

}

```

## 8.65 不要忽略异常

禁止使用空的catch块。

# 9. 并发

## 9.66 同步访问共享的可变数据

同步不仅要保证对象在所有线程中的状态一致，还要保证进入同步方法或者同步代码块的每个线程都可以看到由同一个锁保护的对象状态修改。

也就是说，同步不仅要保证状态一致，同时也要保证状态的可见性，对状态的修改既要互斥也要保证对其它线程可见。

比如：

```java

import java.util.concurrent.TimeUnit;

public class B7 {

    private static boolean flag;

    public static void main(String[] args) throws InterruptedException {

        Thread t = new Thread(new Runnable() {

            @Override

            public void run() {

                int i = 0;

                while (!flag) {

                    i++;

                }

            }

        });

        t.start();

        TimeUnit.SECONDS.sleep(1);

        flag = true;

    }

    // 这里程序永远不会终止

    // 没有同步，状态不可见

}

虚拟机会将

while (!flag) {

    i++;

}

变为

if (!flag) {

    while (true) {

        i++;

    }

}

这种优化称为提升（hoisting）。HopSpot Server VM。

进行同步操作：

import java.util.concurrent.TimeUnit;

public class B77 {

    private static boolean flag;

    static synchronized void set() {

        flag = true;

    }

    static synchronized boolean read() {

        return flag;

    }

    public static void main(String[] args) throws InterruptedException {

        Thread t = new Thread(new Runnable() {

            @Override

            public void run() {

                int i = 0;

                while (!read()) {

                    i++;

                }

                System.out.println(i);

            }

        });

        t.start();

        TimeUnit.SECONDS.sleep(1);

        set();

    }

    // 这里程序在1s后终止

}

或者

import java.util.concurrent.TimeUnit;

public class B7 {

    // volatile没有互斥操作，单保证了变量的值是最新的

    private static volatile boolean flag;

    public static void main(String[] args) throws InterruptedException {

        Thread t = new Thread(new Runnable() {

            @Override

            public void run() {

                int i = 0;

                while (!flag) {

                    i++;

                }

            }

        });

        t.start();

        TimeUnit.SECONDS.sleep(1);

        flag = true;

    }

    // 这里程序在1s后终止

}

```

```java

// 非同步安全

static volatile int n = 0;

static void int set() {

    return n++;

}

改进：

static int n = 0;

static synchronized int set() {

    return n++;

}

static final AtomicInteger n = new AtomicInteger();

static int set() {

    return n.getAndIncrement();

}

```

## 9.67 避免过度同步

同步可能会导致性能降低、死锁和未知的错误。

在同步区域内尽量做少的工作，尽量不要调用外部方法。

## 9.68 executor和task优先于线程

java.util.concurrent.Executors

ExecutorService executor = Executors.newSingleThreadExecutor();

executor.execute(Runnable)

executor.Shutdown()

工作单元 Rannable Thread

执行机制 ExecutorService

## 9.69 并发工具优先于wait和notify

JUC的工具分为3类：

1. Executor Framework

2. 并发集合（Concurrent Collection）

3. 同步器 (Synchronized)

优先使用ConcurrentHashMap，而不是Collections.synchronizedMap或者Hashtable

package java.util.concurrent;

public interface BlockingQueue<E> extends Queue<E> {}

常用的同步器：CountDownLatch、Semaphore。不常用的有CyclicBarrier、Exchanger。

```java

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class B8 {

    static final ExecutorService service = Executors.newSingleThreadExecutor();

    public static void main(String[] args) throws InterruptedException {

        f1(5);

    }

    static void f1(int n) throws InterruptedException {

        final CountDownLatch ready = new CountDownLatch(n);

        final CountDownLatch done = new CountDownLatch(n);

        final CountDownLatch start = new CountDownLatch(1);

        for (int i=0;i<n;i++) {

            final int k = i;

            service.execute(new Runnable() {

                @Override

                public void run() {

                    ready.countDown();

                    try {

                        System.out.println("step start " + k);

//                        start.await();

                        TimeUnit.SECONDS.sleep(k);

                        System.out.println("step end " + k);

                    } catch (InterruptedException e) {

                        e.printStackTrace();

                    } finally {

                        done.countDown();

                    }

                }

            });

        }

        ready.await();

        long startNanos = System.nanoTime();

//        start.countDown();

        done.await();

        System.out.println(System.nanoTime() - startNanos);

        service.shutdown();

    }

}

```

始终使用System.nanoTime()，而不是System.currentTimeMills()。更加精确，不受系统实时时钟的影响。

wait()用来使线程等待某个条件。必须在同步区域内部被调用，这个同步区域将对象锁定在调用了wait()的对象上。

标准使用

synchronized(obj) {

    while (condition) {

        obj.wait(); // release lock, and reacquires on wakeup

    }

}

始终应该使用wait()循环模式来调用wait()。永远不要在循环之外调用wait()。循环在等待之前和等待之后测试条件。

为了唤醒等待的线程，可以使用notidy()，notifyAll()，建议使用notifyAll()，因为所有wait()都有测试条件。线程醒来之后，会检查正在等待的条件，如果条件不满足，就继续等待。

## 9.70 线程安全的文档

应该说明类的线程安全级别：

1. 不可变的（immutable）: 实例不可变。不需要同步。String、Long、BigInteger等

2. 无条件的线程安全（unconditionally thread-safe）: 实例是可变的，但是类的内部是线程安全的，实例可以被并发的使用，无需外部同步。Random、ConcurrentHashMap。

3. 有条件的线程安全（conditionally thread-safe）: 除了有些方法为了并发需要外部的同步外，其它跟无条件的线程安全一样。Collections.synchronized包装返回的集合，迭代器要求外部同步。

4. 非线程安全（not thread-safe）:  实例可变。需要进行外部同步。HashMap，ArrayList

5. 线程对立（thread-hostile）: 不能安全被多线并发使用。比如System.runFinalizersOnExit()，已废弃。基本没有类会这样来设计。

锁的使用应该说明清楚。

## 9.71 慎用延迟初始化

参考单例模式

synchronized  --> 静态内部类 --> 双重检查加锁 --> 枚举

大多数域应该进行正常的初始化，只有少数域为了性能必须延迟初始化。

对于实例域，使用双重检查模式。对于静态域，使用lazy initialization holder class idiom。

对于可以接受重复初始化的实例域，可以考虑使用单检查模式。

## 9.72 不要依赖线程调度器

过多的活跃线程不一定是性能更高的。线程不应该处于busy-wait状态。

线程优先级不具备可移植性，不同机器的优先级不一样。

Thread.yield()不可移植。不同VM有不同表现。

Thread.yield()和线程优先级仅对调度器做些暗示，实际没有真正的影响。

## 9.73 避免使用线程组

thread-group已经过时。

package java.lang;

public class ThreadGroup implements Thread.UncaughtExceptionHandler {}

安全有缺陷。使用线程池（executor）。

# 10. 序列化

对象 --- 字节流

## 10.74 谨慎地实现Serializable接口

一旦一个类实现了序列化的接口，那么这个类就失去了灵活性。因为类的字节码编码（或者类的序列化形式）就是导出API的一部分。必须保证类改变的时候这个导出API兼容旧的形式。

序列化版本UID（serial version UID）与类信息有关，最好一开始就指定好。

序列化是一种语言外的对象创建机制，增加了Bug和安全漏洞的可能性。

反序列化机制包含了隐藏的构造器，反序列的过程必须保证所有由构造器建立的约束关系，这个是很难做到和保证的。

为了继承而设计的类应该可能少的实现Serializable接口。

如果父类没有提供无参构造器，子类不可能实现可序列化。

内部类不应该实现序列化接口，内部类的默认序列化形式是定义不清楚的。

静态成员类可以实现序列化接口。

安全实现一个可序列化的子类：

```java

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import java.util.concurrent.atomic.AtomicReference;

public abstract class B9 {

    private int x, y;

    // this enum and field are used to track initialization

    private enum State {NEW, INITIALIZING, INITIALIZED}

    private final AtomicReference<State> init =

            new AtomicReference<>(State.NEW);

    public B9(int x, int y) {initialize(x, y);}

    // this constructor and the following method allow

    // subclass's readObject method to initialize our state

    protected B9() {}

    protected final void initialize(int x, int y) {

        if (!init.compareAndSet(State.NEW, State.INITIALIZING)) {

            throw new IllegalStateException("Already initialized");

        }

        this.x = x;

        this.y = y;

        // do anything else the original constructor did

        init.set(State.INITIALIZED);

    }

    // these methods provide access to internal state to it can

    // be manually serialized by subclass's writeobject method

    protected final int getX() {checkInit(); return x;}

    protected final int getY() {checkInit(); return y;}

    // must call from all public and protected instance methods

    private void checkInit() {

        if (init.get() != State.INITIALIZED) {

            throw new IllegalStateException("Uninitialized");

        }

    }

}

// 实现一个可序列化的子类

class B99 extends B9 implements Serializable {

    private static final long serialVersionUID = 1856835860954L;

    private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {

        s.defaultReadObject();

        // manually deserialize and initialize superclass state

        int x = s.readInt();

        int y = s.readInt();

        initialize(x, y);

    }

    private void writeObject(ObjectOutputStream s) throws IOException {

        s.defaultWriteObject();

        // manually serialize superclass state

        s.writeInt(getX());

        s.writeInt(getY());

    }

    // constructor does not use the fancy mechanism

    public B99(int x, int y) {super(x, y);}

}

```

## 10.75 考虑使用自定义的序列化形式

必须提供一个readObject()以保证约束关系和安全性。

Javadoc @serial 标签建立序列化的文档说明。

Heap Size -Xms -Xmx

选择错误的序列化形式对于一个类的复杂性和性能都有永久的负面影响。

## 10.76 保护性编写readObject()

readObject()实际上相当于另外一个公有的构造器。是一个“用字节流作为唯一参数”的构造器。

所以要保证字节流的正确性。

## 10.77 对于实例控制，枚举类型优先于readResolve()

readResolve()允许用readObject()创建的实例代替另外一个实例。

对于一个正在被反序列化的对象，如果类中定义了一个readResolve()，并且具备正确的声明，那么在反序列化之后，新建对象上的readResolve()就会被调用，该方法返回的对象引用将被返回，取代新建的对象。

## 10.78 考虑用序列化代理代替序列化实例

序列化代理模式（serialization proxy pattern）

首先为可序列化的类设计一个私有静态嵌套类，这个类就是序列化代理类。

从设计的角度看，序列化代理的默认序列化形式是外围类最好的序列化形式。

外围类和序列化代理必须实现序列化接口。

```java

import java.io.InvalidObjectException;

import java.io.ObjectInput;

import java.io.ObjectInputStream;

import java.io.Serializable;

import java.util.Date;

public class B10 implements Serializable {

    private static final long serialVersionUID = 1L;

    private Date start;

    private Date end;

    private B10(Date start, Date end) {

        this.start = start;

        this.end = end;

    }

    private static class B10Proxy implements Serializable {

        private static final long serialVersionUID = 2L;

        private Date start;

        private Date end;

        B10Proxy(B10 b) {

            this.start = b.start;

            this.end = b.end;

        }

        private Object readResolve() {

            return new B10(start, end);

        }

    }

    private Object writeReplace() {

        return new B10Proxy(this);

    }

    private void readObject(ObjectInputStream stream)

            throws InvalidObjectException {

        throw new InvalidObjectException("Proxy required");

    }

}

```

EnumSet内部使用了序列化代理：

```java

package java.util;

public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E>

    implements Cloneable, java.io.Serializable

{

    final Class<E> elementType;

    final Enum<?>[] universe;

    EnumSet(Class<E>elementType, Enum<?>[] universe) {

        this.elementType = elementType;

        this.universe    = universe;

    }

    private static class SerializationProxy <E extends Enum<E>>

        implements java.io.Serializable

    {

        /\*\*

         \* The element type of this enum set.

         \*

         \* @serial

         \*/

        private final Class<E> elementType;

        /\*\*

         \* The elements contained in this enum set.

         \*

         \* @serial

         \*/

        private final Enum<?>[] elements;

        SerializationProxy(EnumSet<E> set) {

            elementType = set.elementType;

            elements = set.toArray(ZERO\_LENGTH\_ENUM\_ARRAY);

        }

        // instead of cast to E, we should perhaps use elementType.cast()

        // to avoid injection of forged stream, but it will slow the implementation

        @SuppressWarnings("unchecked")

        private Object readResolve() {

            EnumSet<E> result = EnumSet.noneOf(elementType);

            for (Enum<?> e : elements)

                result.add((E)e);

            return result;

        }

        private static final long serialVersionUID = 362491234563181265L;

    }

    Object writeReplace() {

        return new SerializationProxy<>(this);

    }

    // readObject method for the serialization proxy pattern

    // See Effective Java, Second Ed., Item 78.

    private void readObject(java.io.ObjectInputStream stream)

        throws java.io.InvalidObjectException {

        throw new java.io.InvalidObjectException("Proxy required");

    }

}

```

当然序列化代理有性能损耗。

# 11.术语对照

wrapper class 包装类

strategy pattern 策略模式

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