

第六章 注解与枚举

30. 用 enum 常量代替 int 常量

- a) 枚举常量：有一组固定的常量组成合法值的类型
- b) int 枚举类型
 - a) 使用方便性和类型安全方面没帮助，如果将 `APPLE` 传入 `ORANGE` 也没有任何警告
 - b) 十分脆弱，如果和枚举常量相关的 `int` 类型发生变化，客户端就必须重新编译
 - c) 没法把枚举常量打印或者遍历
- c) String 枚举模式，更糟糕
 - a) 性能问题，依赖字符串比较
 - b) 导致用户把字符串常量强行编码，可变性差而且容易出错
- d) 最简单的枚举模式：通过共有的静态 `final` 域为每个枚举常量导出实例的类

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

- a) 提供了编译时类型安全，自动命名空间隔离
- b) 可以增加枚举类型的常量而无需重新编译它的客户端代码：隔离层
- c) 可以调用 `toString` 方法将枚举常量转换成可打印的字符串
- d) 可以添加任意的方法和域：例如使用枚举常量直接获得相关数据

```
// Enum type with data and behavior
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS (4.869e+24, 6.052e6),
    EARTH (5.975e+24, 6.378e6),
    MARS (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);
    private final double mass;           // In kilograms
    private final double radius;         // In meters
    private final double surfaceGravity; // In m / s^2

    // Universal gravitational constant in m^3 / kg s^2
    private static final double G = 6.67300E-11;

    // Constructor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }

    public double mass() { return mass; }
    public double radius() { return radius; }
    public double surfaceGravity() { return surfaceGravity; }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity; // F = ma
    }
}
```

为了将数据和枚举常量关联起来，得声明实例域并编写一个带有数据并将数据保存

在域中的构造器。枚举类型天生不可变，因此所有域都必须是 final。最好设置为私有并提供公有的访问方法。使用方法如下

```
public class WeightTable {  
    public static void main(String[] args) {  
        double earthWeight = Double.parseDouble(args[0]);  
        double mass = earthWeight / Planet.EARTH.surfaceGravity();  
        for (Planet p : Planet.values())  
            System.out.printf("Weight on %s is %f%n",  
                p, p.surfaceWeight(mass));  
    }  
}
```

Planet.values()返回它的值数组

- e) 除非迫不得已把枚举方法导出它的客户端，否则都应该声明成私有或者包级私有
- f) 关联本质上不同的行为：看下面例子

```
// Enum type that switches on its own value - questionable  
public enum Operation {  
    PLUS, MINUS, TIMES, DIVIDE;  
  
    // Do the arithmetic op represented by this constant  
    double apply(double x, double y) {  
        switch(this) {  
            case PLUS: return x + y;  
            case MINUS: return x - y;  
            case TIMES: return x * y;  
            case DIVIDE: return x / y;  
        }  
        throw new AssertionError("Unknown op: " + this);  
    }  
}
```

这段代码的脆弱之处：

1. 如果没有 throw 语句就不能编译
 2. 加入新的计算方法时，如果忘记在 switch 里加入条件，无法在编译时得到错误
- 一个好的解决方法：**在枚举类型中声明一个抽象的 apply 方法，并在特定于常量的类主体中，用具体的方法覆盖每个常量的抽象方法。这叫做 constant-specific method implementation**

```
// Enum type with constant-specific class bodies and data  
public enum Operation {  
    PLUS["+"] {  
        double apply(double x, double y) { return x + y; }  
    },  
    MINUS["-"] {  
        double apply(double x, double y) { return x - y; }  
    },  
    TIMES["*"] {  
        double apply(double x, double y) { return x * y; }  
    },  
    DIVIDE["/] {  
        double apply(double x, double y) { return x / y; }  
    };  
    private final String symbol;  
    Operation(String symbol) { this.symbol = symbol; }  
    @Override public String toString() { return symbol; }  
  
    abstract double apply(double x, double y);  
}
```

这样如果忘记添加，就会得到编译错误

- g) 与 toString 相对应的 fromString 方法：使用共有静态常量域+Map

```

// Implementing a fromString method on an enum type
private static final Map<String, Operation> stringToEnum
    = new HashMap<String, Operation>();
static { // Initialize map from constant name to enum constant
    for (Operation op : values())
        stringToEnum.put(op.toString(), op);
}
// Returns Operation for string, or null if string is invalid
public static Operation fromString(String symbol) {
    return stringToEnum.get(symbol);
}

```

h) 策略枚举：看下面的案例

```

// Enum that switches on its value to share code - questionable
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;
    private static final int HOURS_PER_SHIFT = 8;
    double pay(double hoursWorked, double payRate) {
        double basePay = hoursWorked * payRate;

        double overtimePay; // Calculate overtime pay
        switch(this) {
            case SATURDAY: case SUNDAY:
                overtimePay = hoursWorked * payRate / 2;
            default: // Weekdays
                overtimePay = hoursWorked <= HOURS_PER_SHIFT ? 0 :
                    (hoursWorked - HOURS_PER_SHIFT) * payRate / 2;
                break;
        }
        return basePay + overtimePay;
    }
}

```

这段代码的问题是：如果忘记在 switch 里添加计算方法，那么编译不会报错
我们真正想要的是：每当添加一个枚举常量，就要强制选择一种加班策略

策略模式：把加班工时的计算委托给一个私有嵌套枚举类，安全而灵活

```

enum PayrollDay {
    MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),
    WEDNESDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY),
    FRIDAY(PayType.WEEKDAY),
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;
    PayrollDay(PayType payType) { this.payType = payType; }

    double pay(double hoursWorked, double payRate) {
        return payType.pay(hoursWorked, payRate);
    }
    // The strategy enum type
    private enum PayType {
        WEEKDAY {
            double overtimePay(double hours, double payRate) {
                return hours <= HOURS_PER_SHIFT ? 0 :
                    (hours - HOURS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            double overtimePay(double hours, double payRate) {
                return hours * payRate / 2;
            }
        };
        private static final int HOURS_PER_SHIFT = 8;

        abstract double overtimePay(double hrs, double payRate);

        double pay(double hoursWorked, double payRate) {
            double basePay = hoursWorked * payRate;
            return basePay + overtimePay(hoursWorked, payRate);
        }
    }
}

```

- i) 枚举中的 switch 适合给外部的枚举类型增加特定于常量的方法

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
    switch(op) {
        case PLUS:   return Operation_MINUS;
        case MINUS:  return Operation_PLUS;
        case TIMES:  return Operation_DIVIDE;
        case DIVIDE: return Operation_TIMES;
        default:     throw new AssertionError("Unknown op: " + op);
    }
}
```

- j) 使用枚举的时候：需要一组固定常量，或者编译时就知道所有可能值的集合

31. 用实例域代替序数

- a) 永远不要使用 ordinal() 来获得序数。所有的序数都应该保存在实例域里

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int number_of_Musicians;
    Ensemble(int size) { this.number_of_Musicians = size; }
    public int number_of_Musicians() { return number_of_Musicians; }
}
```

- b) Ordinal() 用在 enumSet 这样的类中，如果不编写这样的类就不要使用这个函数

32. 用 EnumSet 代替位域

- a) 位域的通常做法

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
    public static final int STYLE_BOLD          = 1 << 0; // 1
    public static final int STYLE_ITALIC         = 1 << 1; // 2
    public static final int STYLE_UNDERLINE      = 1 << 2; // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8

    // Parameter is bitwise OR of zero or more STYLE_ constants
    public void applyStyles(int styles) { ... }
}
```

位域的不足之处：

- a) 使用方便性和类型安全方面没帮助
b) 没法把枚举常量打印或者遍历

- b) EnumSet 的做法：

```
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) { ... }
}
```

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

使用 EnumSet 更加灵活

33. 用 EnumMap 代替序数索引

a) 来看一个错误案例

```
// Using ordinal() to index an array - DON'T DO THIS!
Herb[] garden = ...;

Set<Herb>[] herbsByType = // Indexed by Herb.Type.ordinal()
    (Set<Herb>[]) new Set[Herb.Type.values().length];
for (int i = 0; i < herbsByType.length; i++)
    herbsByType[i] = new HashSet<Herb>();

for (Herb h : garden)
    herbsByType[h.type.ordinal()].add(h);

// Print the results
for (int i = 0; i < herbsByType.length; i++) {
    System.out.printf("%s: %s%n",
        Herb.Type.values()[i], herbsByType[i]);
}
```

这段代码的不足之处在于：使用错误的 int 值无法在编译时得到警告，难以查错

```
// Using an EnumMap to associate data with an enum
Map<Herb.Type, Set<Herb>> herbsByType =
    new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class)
for (Herb.Type t : Herb.Type.values())
    herbsByType.put(t, new HashSet<Herb>());
for (Herb h : garden)
    herbsByType.get(h.type).add(h);
System.out.println(herbsByType);
```

用 EnumMap 改写过的版本没有安全转换，不会出现索引问题

b) 一个更复杂的例子：给定两个状态，求出两个状态之间转换的行为

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase { SOLID, LIQUID, GAS;
    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;
        // Rows indexed by src-ordinal, cols by dst-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null,      MELT,      SUBLIME },
            { FREEZE,    null,      BOIL     },
            { DEPOSIT,   CONDENSE,  null     }
        };
        // Returns the phase transition from one phase to another
        public static Transition from(Phase src, Phase dst) {
            return TRANSITIONS[src.ordinal()][dst.ordinal()];
        }
    }
}
```

这段代码看似 elegant，但其实编译器并不知道序数与数组索引的关系，如果转换表出错，或者转换表加入新的 phase 就会在运行时出错。

```

// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

        private final Phase src;
        private final Phase dst;

        Transition(Phase src, Phase dst) {
            this.src = src;
            this.dst = dst;
        }

        // Initialize the phase transition map
        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 trans : Transition.values())
                m.get(trans.src).put(trans.dst, trans);
        }

        public static Transition from(Phase src, Phase dst) {
            return m.get(src).get(dst);
        }
    }
}

```

新写的版本不仅在空间或者时间上没有额外的开销，而且提升了安全性和可维护性

34. 使用接口模拟可伸缩的枚举

- a) 可伸缩性：让一个枚举类型去拓展另一个枚举类型。目前还没有很好的方法来枚举基本类型的所有元素及其扩展。
- b) 典型用例：**操作码，opcode**。枚举加接口实现。枚举类型虽然不可拓展，但是接口是可以拓展的。如果想写一个“子类”，那么重新编写一个枚举类并且实现接口即可。

```

// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    };

    private final String symbol;
    ExtendedOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}

```

```

// Emulated extensible enum using an interface
public interface Operation {
    double apply(double x, double y);
}

public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    BasicOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}

```

c) 使用测试函数遍历所有枚举

方法 1：使用有限制的类型令牌

```

public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}

private static <T extends Enum<T> & Operation> void test(
    Class<T> opSet, double x, double y) {
    for (Operation op : opSet.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
            x, op, y, op.apply(x, y));
}

```

方法 2：使用有限制的通配符

```

public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet,
    double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
            x, op, y, op.apply(x, y));
}

```

d) values()方法是编译器插入到 enum 定义中的 static 方法，所以，当你将 enum 实例向上转型为父类 Enum 时，values()就不可访问了。解决办法：在 Class 中有一个 getEnumConstants()方法，所以即便 Enum 接口中没有 values()方法，我们仍然可以通过 Class 对象取得所有的 enum 实例

35. 注解优先于命名模式

a) 命名模式的缺点

- a) 拼写错误会导致失败而且没有任何提示。例子：Junit 要求用户以 test 开始，但是拼写错误会导致 JUnit 不报错也不执行相应的测试程序
- b) 命名模式不能用在特定的程序元素上。例子：在类名前加 test 不能自动执行这个类的所有方法。
- c) 命名模式不能把参数值和程序元素关联起来。如果测试在抛出某个特定异常时才算通过，就不能把异常编码进函数名中。

b) 使用注解

```
// Marker annotation type declaration
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

第一行注解：test 注解在运行时保留

第二行注解：test 注解只有在方法中才是合法的，不能用在其他程序元素上

注意：test 注解只能用在无参数的静态方法上

```
// Program containing marker annotations
public class Sample {
    @Test public static void m1() {} // Test should pass
    public static void m2() {}
    @Test public static void m3() {} // Test Should fail
        throw new RuntimeException("Boom");
    }
    public static void m4() {}
    @Test public void m5() {} // INVALID USE: nonstatic method
    public static void m6() {}
    @Test public static void m7() {} // Test should fail
        throw new RuntimeException("Crash");
    }
    public static void m8() {}
}
```

Sample类有8个静态方法，其中4个被注解为测试。这4个中有2个抛出了异常：m3和m7，另外两个则没有：m1和m5。但是其中一个没有抛出异常的被注解方法：m5，是一个实例方法，因此不属于注解的有效使用。总之，Sample包含4项测试：一项会通过，两项会失败，另一项无效。没有用Test注解进行标注的4个方法会被测试工具忽略。

c) 添加对抛出特定异常的 testcase 的检查

```
// Annotation type with a parameter
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

```

// Program containing annotations with a parameter
public class Sample2 {
    @ExceptionTest(ArithmetcException.class)

    public static void m1() { // Test should pass
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmetcException.class)
    public static void m2() { // Should fail (wrong exception)
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmetcException.class)
    public static void m3() { } // Should fail (no exception)
}

```

注意：有可能注解参数在编译时是有效的，但是表示特定异常类型的类文件在运行时不再存在，此时会抛出 `TypeNotPresentException`

- d) 添加对抛出任意一种指定异常的 testcase 的支持

```

// Annotation type with an array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}

// Code containing an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class,
                  NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<String>();

    // The spec permits this method to throw either
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}

```

36. 坚持使用 override 注解

```
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

- a) 这个程序的错误在于：他没有覆盖 equals 而是重载了 equals
- b) 应该在你想要覆盖超类声明的每个方法声明中使用@override 注解。覆盖抽象方法或者接口时也最好加注解，这样能防止添加新方法

37. 用标记接口定义类型

- a) 标记接口是没有包含方法声明的接口。例子：Serializable 接口
- b) 标记接口可以不改进任何方法的契约，只表示整个对象的限制条件，或者表明实例能够利用其它某个类的方法进行处理。
- c) 标记接口相对于标记注解的优势
 - a) 标记接口定义的类型是由被标记类的实例实现的；标记接口没有定义这样的类型
如果参数没有实现 Serializable 接口， ObjectOutputStream.write 会在运行时失败
 - b) 标记接口可以被更加精确的锁定
- d) 标记注解的优势
 - a) 可以通过默认的方法添加一个或者多个注解类型元素，给被使用的注解类型添加更多的信息
 - b) 它们是更大注解机制的一部分，在支持注解的框架中具有一致性
- e) 适用场合
 - a) 如果标记是应用在任何程序元素而非类或者接口，那么应该使用注解。
 - b) 如果标记永远只用于限制特殊元素的接口，那么应该使用标记接口。

第七章 方法

38. 检查参数的有效性

- a) 必须在文档中指明所有的限制，并且在方法体的开头检查限制。如果传递了无效的参数，那么方法应该很快失败并且抛出适当的异常
 - a) 对于公有的方法，要用 JavaDoc 的@throws 标签在文档中说明违反参数限制会抛出的异常；对于未导出的方法，应该抛出断言

```
/*
 * Returns a BigInteger whose value is (this mod m). This method
 * differs from the remainder method in that it always returns a
 * non-negative BigInteger.
 *
 * @param m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus <= 0: " + m);
    ... // Do the computation
}

// Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // Do the computation
}
```

- b) 如果方法并没有使用到参数，而是保存了参数，那么参数检查格外重要。例子：构造器，这样可以避免构造出来的对象违反类的约束条件
- c) 例外：如果检查工作非常昂贵，或者是不切实际的，或者有效性检查已经在计算过程中完成。
 - a) 计算过程中抛出的错误的异常，应该用异常转译技术转换为正确的异常
 - b) 因地制宜最重要

39. 必要时进行保护性拷贝

- a) Java 是安全语言，自动免疫缓冲区溢出，数组越界等内存破坏错误，但是有时需要假设客户端会尽其所能来破坏这个类的约束条件，因此必须保护性的设计程序
- b) 下面这段程序虽然看上去是 final 不可变的，但是 date 本身可变，而且只有引用，所以不安全。

```

// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start the beginning of the period
     * @param end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(
                start + " after " + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }
    ...
}

// Attack the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78); // Modifies internals of p!

```

- c) 为了免受这种攻击，对于构造器的每个可变参数进行保护性拷贝是必要的，并且使用备份对象作为实例组件，而不是原始对象

```

// Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(start + " after " + end);
}

```

注意：为了防止 TOCTOU 攻击，即攻击发生在检查参数到保护性拷贝之间，**保护性拷贝应该在参数检查之前，并且应该检查拷贝得到的备份**

- d) 对于参数类型可以被不信任方子类化的参数，不要调用 `clone` 进行拷贝，因为 `clone` 可能已经被恶意代码覆盖

- e) 访问方法：应该访问私有组件的拷贝，而不是直接返回其引用

```
public Date start() {  
    return new Date(start.getTime());  
}  
  
public Date end() {  
    return new Date(end.getTime());  
}
```

- f) 每当你编写的方法需要客户提供对象进入内部数据结构中，则应该考虑是否容忍对象的改变。不能则必须进行保护性拷贝。返回的对象也应该是拷贝或者是不可变视图
- g) **真正启示：对象内部应该尽可能使用不可变组件**
- h) 如果类和客户端是同一个包的双方，那么不进行拷贝也可以，但是必须在文档中说明
- i) 如果违反类的约束条件只能伤害到客户端本身，那么也可以不进行拷贝。例子：包装类

40. 谨慎的设计方法签名

- a) **谨慎地选择方法的名称**
 - a) 应该易于理解，并且与同一个包内其他名称风格一致
 - b) 应该尽可能选择和大众认知一致的名字
- b) **不要过于追求便利的方法**
 - a) 只有一项操作经常被用到时才提供快捷方式。**如果不能确定就不要提供快捷访问**
- c) **避免过长的参数列表**
 - a) 目标是 4 个，越少越好
 - b) 相同类型的长参数列表格外有害
 - c) 缩短参数列表的方法'
 - a) **分解方法**。每个方法只需要参数的子集。可能有助于提升正交性
 - b) **创建辅助类，用来保存参数的分组。一般静态类**。如果一个频繁出现的参数序列（纸牌的点数和花色）是一个独特的实体（一张纸牌）的代表，就应该使用静态类来表示这个实体
 - c) **从对象创建到方法调用都使用 builder 模式**
- d) **参数类型优先考虑接口而不是类**。减少客户端调用函数的限制
- e) 对于 **boolean** 参数，优先使用两个元素的枚举类型

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

- f) **要考虑 API 的性能后果**。使公有类变成可变的会导致大量的保护性拷贝

41. 慎用重载

```
// Broken! - What does this program print?
public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> l) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

- a) 重载是在编译时做出的决定。在以上程序里，所有的调用都会使用 `Collection<?>`，而不会在运行时绑定到类型上
- b) 重载的方法是静态选择，而被覆盖的方法的选择则是动态的。如果子类所包含的方法声明与祖先类中的方法具有同样签名时，方法就被覆盖了。如果实例方法在子类中被覆盖了而且是在子类上调用的，那么无论子类是什么类型，子类中的覆盖方法都会执行

```
class Wine {
    String name() { return "wine"; }
}

class SparklingWine extends Wine {
    @Override String name() { return "sparkling wine"; }
}

class Champagne extends SparklingWine {
    @Override String name() { return "champagne"; }
}

public class Overriding {
    public static void main(String[] args) {
        Wine[] wines = {
            new Wine(), new SparklingWine(), new Champagne()
        };
        for (Wine wine : wines)
            System.out.println(wine.name());
    }
}
```

- c) 应该避免胡乱地使用重载机制。安全而保守的策略是：**永远不要导出两个具有相同参数数目的重载方法。如果方法使用可变参数，保守的策略是根本不要重载它。**
例子：ObjectOutputStream 类，并没有重载 write 方法，而是使用了 writeBoolean(), writeInt() 等方法签名。
- d) 构造器只能重载。所以我们可以选择静态工厂。但是不用担心覆盖和重载的相互关系，因为构造器无法被覆盖。
- e) 使用重载时，必须在列表里有至少一个参数“根本不同”。如果显然不可能把一种类型的实例转换为另一种类型，或者两个类都不是对方的后代，这两种类型就是根本不同。例如：Collection 和 int
- f) 基本装箱类型的麻烦。remove 方法有两个重载：第一个是 remove(E)，删除 E 元素；第二个是 remove(int) 删除 int 位上的元素。

```

public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }
        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }
}

```

如果想删除某个装箱元素，应该这样

```

for (int i = 0; i < 3; i++) {
    set.remove(i);
    list.remove((Integer) i); // or remove(Integer.valueOf(i))
}

```

- g) 如果重载的方法在相同的参数传入时，执行相同的功能且结果一致，重载就不会带来危害。确保这种结果的做法是：让更具体的重载方法把调用转发给更一般的重载方法

```

public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}

```

42. 慎用可变参数

- a) 可变参数：接受 0 个或者多个指定类型的参数。可变参数机制通过先创建一个数组，数组的大小为在调用为之所传递位置的参数数量，然后将参数值传到数组中，最后将数组传递给方法。
- b) 这段代码有几个问题

- a) 如果没有传递参数进去，它就会在运行时而不是编译时失败
- b) 这段代码很不美观。如果不把 min 初始化为 Integer.MAX_VALUE，否则不能使用 foreach 语句

```
// The WRONG way to use varargs to pass one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

- c) 非常优雅的代码：使用两个参数，并且完全解决了

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

- d) 目前打印数组的方法。不必改造具有 final 数组参数的每个方法：只有确实在数量不定的值上执行调用才使用可变参数

```
// The right way to print an array
System.out.println(Arrays.toString(myArray));
```

- e) 重视性能的时候，使用可变参数要格外小心。可变参数方法的每次调用都会导致一次数组初始化和分配。如果无法承受这种代价，就应该使用这种模式。假设对某个方法的 95% 的调用会使用 3 个或者更少参数，就应该声明这个方法的 5 个重载，前四个使用 0-3 个参数，最后一个使用可变参数。这个方法不太妥当，但是能派上大用场

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

这个重载模式在 EnumSet 类的静态工厂中使用过，提供了更好的性能

43. 返回 0 长度的数组或者集合，而不是 null

- a) 返回 null 而不是零长度的数组也会使返回数组或者集合的方法本身变得更加复杂
- b) 返回 0 长度的数组在性能上会有少许损失，但是有两个优点
 - a) 在这个级别上担心性能问题是不明智的，除非这个方法是造成问题的源头
 - b) 对于不返回任何元素的调用，每次都返回同一个零长度数组是可能的，因为零长度

的数组是不可变的，而不可变对象有可能被自由地共享。

44. 为所有导出的 API 元素编写文档注释

- a) Javadoc 利用特殊格式的文档注释，根据原代码自动产生 API 文档。为了正确地编写 API 文档，必须在每个被导出的类、接口、构造器、方法和域声明之前增加一个文档注释。如果类是可序列化的，也应该对它的序列化形式编写文档。
- b) 方法的文档注释应该简洁的描述它和客户端之间的约定。
 - a) 方法应该列举这个方法的前置条件和后置条件。所有的异常都要描述
 - b) 每个方法还应该文档中描述它的副作用。例如：方法启动了后台线程
 - c) 文档应该描述类和方法的线程安全性
- c) 方法的文档注释应该让每个参数都带有@param 标签，以及一个@return 标签。无论是是否受检，都要有一个@throws 标签。跟在@throws 标签后面的文字应该以 if 开始，跟一个名词短语描述这个异常什么时候被抛出。

```
/*
 * Returns the element at the specified position in this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations it may run in time proportional
 * to the element position.
 *
 * @param index index of element to return; must be
 *              non-negative and less than the size of this list
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *          {@code index < 0 || index >= this.size()})
 */
E get(int index);
```

- d) 应该使用 Javadoc{@code}标签，以避免转义 HTML 元字符
- e) 当“this”被用在实例方法的文档注释中时，它应该始终是指方法调用所在对象
- f) 为了避免混淆，同一个类或接口中的两个成员或者构造器，不应该具有同样的概要描述
- g) 为枚举类型编写文档时，要确保在文档中说明常量，类型和任何公有的方法。
- h) 包级私有的文档注释就应该放在一个称作 package-info.java 的文件中。
- i) 对于有多个互相关联的类组成的复杂 API，通常有必要有一个外部文档来描述该 API 的总体结构，对文档注释进行补充。相关的类或者包文档注释就应该包含一个对这个文档的注释

第八章 通用程序设计

45. 将局部变量的作用域最小化

- a) 要让局部变量的作用域最小化，最有力的方法就是在第一次使用它的地方声明。过早的

声明局部变量不仅会使它的作用域过早地扩展，还会让它的结束过晚。如果变量在它的目标使用域之前或者之后被意外使用，会比较糟糕

- b) 几乎每个局部变量的声明都应该包含一个初始化表达式。如果还没有足够的信息对这个变量做初始化，那就应该推迟声明，直到可以初始化位置。例外是 **try-catch** 块。
- c) 如果在循环终止之后不再需要循环变量的内容，**for** 循环就优先于 **while** 循环，因为 **while** 循环可能会用到循环外的变量。
- d) 如果循环测试涉及方法调用，就应该用下面这种方法

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {  
    doSomething(i);  
}
```

- e) 应该让方法小而集中。

46. For-each 优先于 for

- a) 正规使用方法

```
// The preferred idiom for iterating over collections and arrays  
for (Element e : elements) {  
    doSomething(e);  
}
```

该循环对边界计算只执行一次，有少许性能优势

- b) 一个很隐蔽的 bug

```
...  
Collection<Suit> suits = Arrays.asList(Suit.values());  
Collection<Rank> ranks = Arrays.asList(Rank.values());  
  
List<Card> deck = new ArrayList<Card>();  
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )  
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )  
        deck.add(new Card(i.next(), j.next()));
```

Bug 在于：`i.next()` 调用次数太多了，应该放在外层循环调用。合理方法是使用 `foreach`

```
// Preferred idiom for nested iteration on collections and arrays  
for (Suit suit : suits)  
    for (Rank rank : ranks)  
        deck.add(new Card(suit, rank));
```

- c) `foreach` 可以用来遍历任何实现了 `Iterable` 接口的对象。

```
public interface Iterable<E> {  
    // Returns an iterator over the elements in this iterable  
    Iterator<E> iterator();  
}
```

如果在编写一个类 `Collection`，那么最好让它实现 `Iterable`。这样调用者就能够使用 `foreach` 遍历整个数组

- d) 不能使用 `for-each` 循环的三个情景

- a) 过滤：如果需要遍历集合并删除元素，则需要显式迭代器
- b) 转换/取代：同上
- c) 平行迭代：需要显式迭代器控制同步前移

47. 了解并使用类库

- a) 产生 0 和某个上界之间的随机整数的错误程序

```
private static final Random rnd = new Random();

// Common but deeply flawed!
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这个程序有三个缺点

- a) 如果 `n` 是一个比较小的 2 的幂，那么它产生的随机序列将会重复
- b) 如果 `n` 不是 2 的幂，平均起来有些数比其他数更加频繁
- c) 极少数情况下，他的失败是灾难性的。比如返回 `MIN_VALUE` 会导致取余数为负
正确的做法：使用 `Rondom.nextInt(int)`。
- b) 通过使用标准类库
 - a) 可以充分利用这些编写标准类库的专家的知识，以及前人使用经验
 - b) 不必浪费时间为那些与工作不太相关的问题提供特别的解决方案
 - c) 可以让自己的代码融入主流，更易读易维护。
- c) 在每个重要的发行版本，都有很多新的特性加入类库。与新特性保持同步是很重要的。
- d) 有些时候一个类库工具并不能满足需要。如果标准类库无法满足要求，则需要自己实现这些功能

48. 如果需要精确的答案，请避免使用 float 和 double

- a) 要让一个 `float` 或者 `double` 精确到表示 `0.1` 是不可能的，因此 `float` 和 `double` 尤其不适合货币计算
- b) 这个程序的运行结果是错误的

```
// Broken - uses floating point for monetary calculation!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = .10; funds >= price; price += .10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Change: $" + funds);
}
```

- c) 精确的计算应该使用 `BigDecimal`（大于 18 位）、`int` 或者 `long`（小于 9 位或者小于 18 位）。正确做法如下

使用 `BigDecimal` 的优点：允许你完全控制四舍五入，八种模式选一
使用 `bigDecimal` 有两个缺点：

- a) 不方便
- b) 很慢

```

public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal( ".10");

    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;
         funds.compareTo(price) >= 0;
         price = price.add(TEN_CENTS)) {
        itemsBought++;
        funds = funds.subtract(price);
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: $" + funds);
}

```

49. 基本类型优先于装箱基本类型

- a) 基本类型与装箱基本类型的区别
 - a) 基本类型只有值，装箱基本类型除了值，还有同一性。两个装箱基本类型可以具有相同的值和不同的同一性
 - b) 基本类型只有功能完备的值。而装箱基本类型除了功能值以外，还有一个非功能值 `null`
 - c) 基本类型通常比装箱基本类型更节省时间和空间
- b) 看这个比较器

```

// Broken comparator - can you spot the flaw?
Comparator<Integer> naturalOrder = new Comparator<Integer>() {
    public int compare(Integer first, Integer second) {
        return first < second ? -1 : (first == second ? 0 : 1);
    }
};

```

这个比较器有一个严重的缺陷：如果传入的是两个值相同的装箱 `Integer`，那么这个式子总会返回 `1` 而不是 `0`。`First < second` 可以正常自动拆箱，但是比较 `first == second` 执行的并不是值比较，而是同一性引用比较。**对装箱基本类型运用==操作符几乎总是错误的**

- c) 修正这个问题的方法是添加两个局部变量

```

Comparator<Integer> naturalOrder = new Comparator<Integer>() {
    public int compare(Integer first, Integer second) {
        int f = first; // Auto-unboxing
        int s = second; // Auto-unboxing
        return f < s ? -1 : (f == s ? 0 : 1); // No unboxing
    }
};

```

- d) 当在一项操作中混合使用基本类型和装箱基本类型时，装箱基本类型就会自动拆箱。Null 引用被拆箱时，就会得到空指针异常，如下：

```

public class Unbelievable {
    static Integer i;

    public static void main(String[] args) {
        if (i == 42)
            System.out.println("Unbelievable");
    }
}

```

- e) 使用装箱基本类型，会导致明显性能下降。

```

// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}

```

- f) 适合使用装箱基本类型的时候

- a) 作为集合中的元素，键和值。
- b) 使用反射调用时，必须使用装箱基本类型。例子：Hibernate 和 Servlet 的反射
- c) 用作参数化类型。

50. 如果其他类型更合适，则避免使用字符串

- a) 字符串不适合代替其他的值类型。如果它是数值，就应该用数值类型；如果是“是-否”问题，就应该使用 boolean 类型，或者使用一个值类。
- b) 字符串不适合代替枚举类型。第 30 条
- c) 字符串不适合代替聚集类型。如果一个实体有多个组件，用一个字符串来表示这个实体是很不恰当的。更好的做法是：使用私有静态类类收集数据。

```

// Inappropriate use of string as aggregate type
String compoundKey = className + "#" + i.next();

```

- d) 字符串也不适合代替能力表。例子：字符串键代表了一个共享的全局命名空间，为了访问一些局部变量，客户端提供的字符串键必须是唯一的。字符串无法保证一个唯一的键。

还可以做得更好：去掉 static，然后使用泛型类型来确保线程安全。粗略地讲，这正是 ThreadLocal 提供的 API，更加快速和优雅。

```

public final class ThreadLocal<T> {
    public ThreadLocal() { }
    public void set(T value);
    public T get();
}

```

```

public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    public static class Key { // (Capability)
        Key() { }
    }

    // Generates a unique, unforgeable key
    public static Key getKey() {
        return new Key();
    }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}

```

51. 当心字符串连接的性能

- a) 为连接 n 个字符串而重复地使用字符串连接操作符，需要 $O(n^2)$ 的时间。这是由于字符串不可变，当两个字符串被连在一起时，它们的内容都要拷贝。
- b) 为了获得可以接受的性能，请使用 **StringBuilder** 代替 **String**，来储存字符串。**StringBuilder** 需要 $O(n)$ 的时间
- c) 另一种方法是，使用字符数组

```

public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
    return b.toString();
}

```

52. 通过接口引用对象

- a) 如果有合适的接口类型存在，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。如果养成了用接口作为习惯，程序将更为灵活。如下

```
// Good - uses interface as type
List<Subscriber> subscribers = new Vector<Subscriber>();
```

而不是像这样的声明：

```
// Bad - uses class as type!
Vector<Subscriber> subscribers = new Vector<Subscriber>();
```

- b) 注意：如果原来的实现提供了某种特殊的类型，而这种功能并不是这个接口的约定所要求的，并且代码又依赖于这种功能，新的功能也要提供同样的功能。

例如：如果周围代码依赖于 **Vector** 的同步策略，就不应该是 **ArrayList** 代替 **Vector**

- c) 例子: 1.4 之前 ThreadLocal 类在内部使用一个包级私有的 Map 域, 使用 HashMap 实例。
1.4 之后 Java 添加了一个新的, 被称为 IdentityHashMap 的专用 Map 实现, 只需要改动一行代码就可以让 HashMap 的速度快不少
- d) 如果没有合适的接口, 那么使用类也可以接受。
 - a) 如果类没有实现接口, 都只能使用类来引用对象
 - b) 对象属于一个框架, 而框架的基本类型是类而非接口。如果对象属于这种基于类的框架, 就应该用基类(抽象类)来引用这个框架
 - c) 类实现了接口, 但是它提供了接口中不存在的额外方法, 例如 LinkedHashMap。这种类就必须只被用来引用它的实例

53. 接口优先于反射机制

- a) 核心反射机制提供了“通过程序来访问关于已装载的类的信息”的能力。
- b) 使用 Constructor, Method 和 Filed 实例能够通过反射机制操作它们的底层对等体。反射机制允许一个类使用另一个类, 即使当前者编译时后者还根本不存在。然后这种类型:
 - a) 丧失了编译时类型检查的好处, 包括异常检查
 - b) 执行反射访问所需要的代码非常笨拙和冗长
 - c) 性能损失。反射调用比普通方法慢很多, 具体受多个因素的影响。
- c) 反射功能只是在设计时被用到。通常, 普通应用程序在运行时不应该以反射访问对象
- d) 如果只是以非常有限的形式使用反射机制, 虽然也要付出代价, 但是可以获得很多好处
 - a) 如果编译时有合适的接口或者超类, 那么就应该使用反射创建实例, 然后正常访问这些实例。如果构造器不带参, 可以直接使用 Class.newInstance()

例子: 这个例子里会生成一个第一个参数的类然后打印。如果提供的是 HashSet 那么参数会随机打印; 如果是 TreeSet 那么会按字母顺序打印。

```
// Reflective instantiation with interface access
public static void main(String[] args) {
    // Translate the class name into a Class object
    Class<?> c1 = null;
    try {
        c1 = Class.forName(args[0]);
    } catch(ClassNotFoundException e) {
        System.err.println("Class not found.");
        System.exit(1);
    }

    // Instantiate the class
    Set<String> s = null;
    try {
        s = (Set<String>) c1.newInstance();
    } catch(IllegalAccessException e) {
        System.err.println("Class not accessible.");
        System.exit(1);
    } catchInstantiationException e) {
        System.err.println("Class not instantiable.");
        System.exit(1);
    }

    // Exercise the set
    s.addAll(Arrays.asList(args).subList(1, args.length));
    System.out.println(s);
}
```

这种方法足以实现一个成熟的服务提供者。绝大多数情况下, 反射机制需要的也是这种方法。

注意这个程序实现了 `System.exit(1)`, 这个命令会直接终止 VM, 很少使用, 但是比较适合终止命令行程序。

54. 谨慎地使用本地方法

- a) 本地方法是指的使用本地程序设计语言来编写的特殊方法。本地方法在本地语言中可以执行任意的计算任务，并返回 Java 语言。本地方法的用途主要有
 - a) 提供访问特定平台的功能。例如注册表或者文件锁
 - b) 提供访问遗留代码库的能力。
 - c) 用本地语言编写应用程序中注重性能的部分，以提高系统性能
- b) 使用本地方法来提高性能的做法不值得提倡。对于大多数任务，本地方法与非本地方法性能相当
- c) 本地语言不是安全的。使用本地语言无法保证不出现内存损坏
- d) 本地语言降低了可移植性，更难调试。一个 bug 就可能破坏整个应用程序

55. 谨慎的进行优化

- a) 不要为了计较性能而牺牲了合理的结构。应该努力写好的成熟的程序，而不是快的程序
- b) 努力避免限制性能的设计决策。多用复合，少用继承。
- c) 为了获得好的性能而对 API 进行包装，是非常不好的想法：API 的性能会在后续得到修复，但是包装 API 问题会永久存在
- d) 每次做优化之前和之后，要对性能进行测量。要善用性能剖析工具，改善平方级算法
- e) 在 Java 平台上测量性能更重要，因为 Java 平台没有很强的性能模型，各种基本操作的相对开销也没有明确定义。要在不用的 JVM 或者硬件上进行测试。

56. 遵守普遍接受的命名惯例

- a) 包的层次应该用句号分割，每个部分包括小写字母和数字。用户创建的包应该用组织的 Internet 域名作为顶级域名。决不能使用 Java 和 Javax。
- b) 包的名称的其余部分应该比较简短，通常不超过 8 个字符，使用有意义的缩写。
- c) 大型工具包可以使用非正式的层次结构
- d) 类和接口的名称应该包含一个或多个单词，首字母大写。应该尽量避免缩写。
- e) 方法的第一个单词应该小心
- f) 常量的所有字母都应该大写，并且用下划线区分。枚举域也是常量
- g) 类型参数名称通常由单个字母组成。T 表示类型，E 表示元素集合，K 和 V 表示键和值，X 表示异常
- h) 返回 boolean 值的方法，其名称应该用 is 开头
- i) 如果方法返回被调用对象的一个非 boolean 的属性，应该使用名词/名词短语/带 get 的名词来做名称。JavaBean 强制使用第三种方法
- j) 转换对象的方法应该使用 toType 类型；返回视图的方法应该是 AsType
- k) 域的名称不太重要。最好不要暴露域

第九章 异常

57. 只针对异常的情况才使用异常

- a) 以下是异常的错误用法。这个方法不能提高性能

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while(true)
        range[i++].climb();
} catch(ArrayIndexOutOfBoundsException e) {}
```

这种想法有三个错误

- a) 异常机制的设计初衷适用于不正常的情形，所以很少有 JVM 尝试优化这里
 - b) 把代码放入 try-catch 块反而阻止了现代 JVM 本来可能要执行的优化
 - c) 对数组进行遍历不会导致冗余检查。现在 JVM 会进行优化
- 如果出现了真正的越界访问 bug，这个算法就会错误的结束并且不报错。
- b) 异常应该只用在异常的情形下。他们永远不应该用在正常的控制流中。
 - c) 设立良好的 API 不应该强迫他的客户端为了正常的控制流而使用异常。如果类具有状态相关的方法，那么应该有一个专门的方法来进行状态测试，或者在状态不适当的时候返回一个可识别的值，比如 null
 - a) 状态测试方法的优点
 - a) 可读性好，不会忘记检查返回值
 - b) 可识别的值的优点
 - a) 性能更好
 - b) 不会因为缺乏同步而出现问题

58. 对可恢复的情况使用受检异常，对编程错误使用运行时异常

- a) Java 语言有三种可抛出的结构：受检异常、错误和运行时异常
- b) 决定使用何种异常时主要原则是：**如果期望调用者能够从异常中适当的恢复，那么就使用受检异常强迫用户处理或者再抛出。运行时异常属于不可恢复的情形**
- c) 用运行时异常来表明编程错误。大多数运行时异常都表示前提违例，比如空指针异常。
所有未受检的抛出结构都应该是 **RuntimeException** 的子类
- d) 现实情况应该仔细判断，但原则是能否从异常中恢复。受检的异常应该提供一些辅助方法或者在文档中写明如何恢复

59. 避免不必要的使用受检的异常

- a) 过分使用受检异常会让 API 使用起来格外不方便。如果正确的使用 API 不能阻止这种异常条件的产生，并且一旦产生异常，使用 API 的程序员就可以立即采取有用的动作，才是真正使用受检异常的情形
- b) 如果程序员对异常只能这么做，那就应该使用未受检异常

```
} catch(TheCheckedException e) {  
    throw new AssertionError(); // Can't happen!  
}
```

下面这种做法如何？

```
} catch(TheCheckedException e) {  
    e.printStackTrace(); // Oh well, we lose.  
    System.exit(1);  
}
```

- c) 如果方法只抛出单个受检的异常，那么整个方法就必须被放在 try-catch 块中。
- d) 把受检异常变成未受检异常的方法是：把这个抛出异常的方法分成两个方法，其中一个方法返回一个 boolean 值表明是否应该抛出异常。

```
// Invocation with checked exception  
try {  
    obj.action(args);  
} catch(TheCheckedException e) {  
    // Handle exceptional condition  
}
```

重构为：

```
// Invocation with state-testing method and unchecked exception  
if (obj.actionPermitted(args)) {  
    obj.action(args);  
} else {  
    // Handle exceptional condition  
    ...  
}
```

这种重构会受到同步的限制。但是如果程序员知道调用一定成功，那么他可以直接使用

```
obj.action(args);
```

60. 优先使用标准的异常

- a) 重用异常有多方面好处
 - a) 让你的 API 更加易于学习和使用，因为它和程序员已经熟悉的习惯是一致的
 - b) 可读性会更好
 - c) 内存印记会更小，提高装载速度
- b) 常用异常列表及其适用情况

表9-1 常用的异常

异常	使用场合
IllegalArgumentException	非null的参数值不正确
IllegalStateException	对于方法调用而言，对象状态不合适
NullPointerException	在禁止使用null的情况下参数值为null
IndexOutOfBoundsException	下标参数值越界
ConcurrentModificationException	在禁止并发修改的情况下，检测到对象的并发修改
UnsupportedOperationException	对象不支持用户请求的方法

- c) 一定要保证抛出异常的条件与异常的文档保持一致，但是选择重用哪个异常并不精确

61. 抛出与抽象相对应的异常

- a) 更高层的实现应该捕获低级的实现，同时抛出可以按照高层抽象进行解释的异常，这种做法叫做异常转译

```
// Exception Translation
try {
    // Use lower-level abstraction to do our bidding
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

- b) 另外一种异常转译的方法叫做异常链。如果低级异常对调试导致高级异常的原因很有帮助，那就应该使用异常链，让调用者通过 `getCause` 来获得底层的异常

```
// Exception Chaining
try {
    ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}

// Exception with chaining-aware constructor
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(cause);
    }
}
```

- c) 异常转译不能被滥用。最好的做法是：调用底层方法之前，通过检查参数等手段确定调用底层方法会成功。次好的方法是让高层方法绕开这些异常，隔离高层方法的调用者和底层的问题，使用 `Log` 记录这些异常。

62. 每个抛出的异常都必须有文档

- a) 始终要单独声明受检的异常，并且利用 `JavaDoc` 的 `@throws` 标记准确的记录下抛出每个异常的条件。如果抛出多个异常，不要使用快捷方式声明抛出的超类。不要抛出 `Exception`，

不要把未受检异常放在方法声明的 `throws` 子句中

- b) 对于未受检异常，也应该仔细地为他们建立文档。这样可以有效描述方法的前置条件。对于接口中的方法，在文档中描述可能抛出的未受检异常格外重要，因为这是通用约定的一部分。
- c) 如果某个类的很多方法因为同一个原因抛出同一个异常，那么应该在这个类的文档注释中进行说明

63. 在细节信息里包含能捕获失败的信息

- a) 异常的 `toString` 方法通常应该包含类名，紧随其后的是细节信息。应该尽可能多的返回有关失败的原因。为了捕获异常，异常的细节信息里应该包含所有“对此异常有贡献”的参数和域的值。例子：数组越界异常应包含没有落在界内的下标值，数组的上界和下界。
- b) 异常的细节不应该同“用户层面的错误信息”相提并论。后者对于最终用户必须是可以理解的。而异常的细节里，信息的内容比可理解性要重要得多。
- c) 正确的做法：在异常的构造器里引入参数信息，再直接转换成消息描述

```
/*
 * Construct an IndexOutOfBoundsException.
 *
 * @param lowerBound the lowest legal index value.
 * @param upperBound the highest legal index value plus one.
 * @param index      the actual index value.
 */
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                 int index) {
    // Generate a detail message that captures the failure
    super("Lower bound: " + lowerBound +
          ", Upper bound: " + upperBound +
          ", Index: " + index);

    // Save failure information for programmatic access
    this.lowerBound = lowerBound;
    this.upperBound = upperBound;
    this.index = index;
}
```

这种做法可以有效地把代码集中在异常类中，有这些代码对异常类自身的异常产生高质量的细节信息，而不是要求类的每个用户都产生自己的细节消息

64. 努力使失败保持原子性

- a) 一般来讲，失败的方法调用应该使对象保持调用前的状态
- b) 实现这个效果的方法
 - a) 设计一个不可变的类
 - b) 在执行操作之前先检查参数有效性。这使对象状态被修改之前先抛出适当的异常。
例子：

```
public Object pop() {  
    if (size == 0)  
        throw new EmptyStackException();  
    Object result = elements[--size];  
    elements[size] = null; // Eliminate obsolete reference  
    return result;  
}
```

- c) 调整计算顺序，让任何可能会失败的计算都在改变对象之前完成。例子：TreeMap
- d) 编写一段恢复代码，由它拦截操作过程中的失败，并且让对象回滚到操作开始前的状态。
- e) 在对象的一份临时拷贝上执行操作，成功后再把结果复制回来
- c) 错误是不可恢复的，不需要保持失败原子性
- d) 如果违反这条规则，应该在文档里详细说明对象所处的状态

65. 不要忽略异常

- a) 空的 **catch** 块会让异常达不到应有的目的。至少应该在 **catch** 块里说明为什么可以忽略这个异常。
- b) 有一种情形可以忽略：关闭 `FileInputStream`。即使这样也应该记录异常
- c) 正确的异常处理能够彻底挽回失败，必须将异常传递出去。

第十章 并发

66. 同步访问共享的可变数据

- a) 同步的概念：
 - a) 当一个对象被一个线程修改的时候，可以阻止另一个线程观察到对象内部不一致的状态
 - b) 保证进入同步方法或者同步代码块的每个线程，都能看到由同一个锁保护的之前所有的修改效果。
- b) Java 保证读或写一个变量是原子性的，除非这个变量的类型为 `long` 或者 `double`。但是这并不保证一个线程写入的值对于另一个线程是可见的。**因此，为了在线程之间进行可靠通信，也为了互斥访问，同步时必须的。**
- c) Java 类库中提供了 `Thread.stop` 方法，但是这个方法是不安全的。**不要使用 `Thread.stop`。**要让一个线程终止另一个线程，应该让第一个线程轮询一个 `boolean` 域，另一个线程修改这个 `boolean` 域，终止这个线程。

以下是一个错误的方法，因为没有被同步的域会被优化成活性失败

```
if (!done)  
    while (true)  
        i++;
```

```

// Broken! - How long would you expect this program to run?
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args)
        throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested)
                    i++;
            }
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}

```

- d) 真正的同步方式必须要同时同步读操作和写操作。否则同步就不会起作用

```

// Properly synchronized cooperative thread termination
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args)
        throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested())
                    i++;
            }
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}

```

- e) 另一个方法是把 boolean 域设置成 volatile，这样就不需要同步了。Volatile 虽然不执行互斥访问，但他可以保证任何一个线程在读取该域时都能看到最近刚刚被写入的值

```

public class StopThread {
    private static volatile boolean stopRequested;
}

```

使用 volatile 务必要小心

```
// Broken - requires synchronization!
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

但是这个方法仍然不能同步，因为增量操作符++不是原子的。如果第二个线程在第一个线程读取旧值写回新值读取这个域，那两个线程就会看到同一值。这就是安全性失败。

f) 修改这个方法的原因

- a) 在它的声明中增加 synchronized 修饰符
- b) 使用 AtomicLong 类，这样就不需要手动外部同步了

```
private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

- g) 最佳方法：共享不可变的数据，或者不共享可变的数据。把可变数据限制在单个线程里。
- h) 让一个线程短时间修改一个数据对象，然后与其它线程共享，只同步共享对象引用的动作，然后其他线程没有进一步的同步也可以读取对象。这叫做事实上不可变的对象。这种传递对象引用的方法叫做安全发布。

67. 避免过度同步

- a) 在一个被同步的方法或者代码块里，永远不要放弃对客户端的控制。在一个被同步的区域内部，不要调用被设计成被覆盖的方法，或者由客户端以函数对象提供的方法。否则同步调用可能因此而异常，死锁或者数据损坏

```
private void notifyElementAdded(E element) {
    synchronized(observers) {
        for (SetObserver<E> observer : observers)
            observer.added(this, element);
    }
}
```

b)

```
set.addObserver(new SetObserver<Integer>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) s.removeObserver(this);
    }
});
```

这样调用的结果是抛出并发修改异常。原因是 Java 的锁都是可重入的，在一次迭代中列表已经被锁定了，再进入函数修改这个列表就会抛出异常。**可重入的锁简化了面向对象程序构造，但是会把活性失败变成了安全失败**

- c) 这次的调用不会抛出异常，而是会导致死锁。因为后台线程试图锁住 s，但是 s 已经被主线程锁定了。主线程一直等待后台程序完成调用，所以导致了死锁。

```
// Observer that uses a background thread needlessly
set.addObserver(new SetObserver<Integer>() {
    public void added(final ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) {
            ExecutorService executor =
                Executors.newSingleThreadExecutor();
            final SetObserver<Integer> observer = this;
            try {
                executor.submit(new Runnable() {
                    public void run() {
                        s.removeObserver(observer);
                    }
                }).get();
            } catch (ExecutionException ex) {
                throw new AssertionError(ex.getCause());
            } catch (InterruptedException ex) {
                throw new AssertionError(ex.getCause());
            } finally {
                executor.shutdown();
            }
        }
    }
});
```

- d) 解决方法：

- a) 使用快照，在无锁的情况下安全的遍历列表

```
List<SetObserver<E>> snapshot = null;
synchronized(observers) {
    snapshot = new ArrayList<SetObserver<E>>(observers);
}
for (SetObserver<E> observer : snapshot)
    observer.added(this, element);
```

- b) 使用 Java 类库提供的并发集合，称作 `CopyOnWriteArrayList`。这是 `ArrayList` 的变种，通过拷贝整个底层数组，实现所有的写操作。由于迭代不需要锁定，速度会非常快。如果需要修改，那么 `CopyOnWriteArrayList` 的性能就将大受影响

```
// Thread-safe observable set with CopyOnWriteArrayList
private final List<SetObserver<E>> observers =
    new CopyOnWriteArrayList<SetObserver<E>>();

public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
}

public boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
}

private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.added(this, element);
}
```

- e) 通常，在同步区域内应该做尽可能少的工作。多核时代的实际成本并不是只获得锁所花

费的 CPU 时间，而是失去了并行的机会，以及需要保证每个核都有一个一致的内存视图而导致的延迟。另一项潜在开销在于它会阻碍 VM 对代码的优化。

- f) 如果一个类要并发使用，应该把这个类变成线程安全的，通过内部同步，能获得比外部锁定整个对象更高的并发性。否则就不要同步。反例：`StringBuilder` 用于单个线程却被设计成内部同步，大大有损性能。不确定的时候就不要同步类，而是建立文档并说明它不是线程安全的。
- g) 如果方法修改了静态域，就必须同步对这个域的访问。

68. Executor 和 task 优先于线程

- a) `Java.util.concurrent` 包含了一个 **Executor Framework**，完成了一个简单的工作队列。这个类允许客户将后台异步处理的工作项目加入队列，当不再需要这个队列时，客户端调用一个方法可以让队列完成任务后终止自己。使用方法如下：

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

下面是为执行提交一个`Runnable`的方法：

```
executor.execute(runnable);
```

下面是告诉`executor`如何优雅地终止（如果做不到这一点，虚拟机可能将不会退出）：

```
executor.shutdown();
```

- b) **Executor service** 还可以完成更多的任务

- a) 可以等待完成一项特殊的任务
- b) 可以等待一个任务集合中的任何任务或所有任务完成(`invokeAny` 和 `invokeAll` 方法)
- c) 可以等待 `executor service` 优雅地完成 (`awaitTermination` 方法)
- d) 在任务完成时逐个获得任务的结果 (`ExecutorCompletionService` 方法)
- c) 如果想让不止一个线程来处理这个队列的请求，只要调用一个不同的静态工厂，这个工厂创建了一种不同的 `executor service`，称作**线程池** `thread pool`。你可以直接使用 `ThreadPoolExecutor` 类控制整个线程池的各个方面。
- d) 为特殊的应用程序选择 `service` 是有技巧的。
 - a) 小程序、轻载服务器：`Executor.newCachedThreadPool`，不需要配置而且能正确完成任务
 - b) 大程序，大负载服务器：`Executor.newCachedThreadPool`。缓存线程池，任务没有排成队列而是直接交给线程执行，如果没有线程可用，就新建线程，反而会更加加重服务器的负载。所以应该使用固定线程的线程池。
- e) 目前的抽象是工作单元，是 `task` 任务有两种：**Runnable** 与其近亲 **Callable**（有返回值）。
Executor Framework 的工作是执行，而 Collection Framework 的工作是聚集
- f) Executor Framework 也有可以代替 Timer 的东西叫做 **ScheduledThreadPool** 类。
 - a) Timer 虽然更容易，但是 executor 更加灵活。
 - b) Timer 只用一个线程执行任务，长期运行的任务会影响定时的准确性
 - c) Timer 唯一的线程抛出捕获异常，timer 就会停止执行。被调度的线程池 executor 支持多个线程，并且优雅的从异常中恢复

69. 并发工具优先于 wait 和 notify

- a) Java 的新版本提供了更高级的并发工具，就不要使用过去的 **wait** 和 **notify**。
- b) Java.util.concurrent 中更高级的工具分三类：**Executor Framework**、**并发集合**以及**同步器**
- c) **并发集合 Concurrent Collection**
 - a) 并发集合为标准集合接口（List、Map 或者 Queue）提供了高性能的并发实现，在内部管理同步。因此并发集合中不能排除并发活动，**不要锁定它**。
 - b) 客户端无法原子地对并发集合进行调用。例子：ConcurrentMap 类拓展了 Map 接口，添加了 putIfAbsent 方法，在没有映射时插入映射返回 null，有映射时插入新值返回旧值。ConcurrentMap 对 get 方法做了优化，因此应该这么完成

```
// Concurrent canonicalizing map atop ConcurrentHashMap - faster!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
```

- c) **应该优先使用并发集合，而不是使用外部同步的集合**
- d) 有些接口通过阻塞操作进行了拓展，会一直阻塞到成功执行。例如：**BlockingQueue** 拓展了 **Queue** 接口，使用了 **producer-consumer** 模型
- d) **同步器 Synchronizer**
 - a) 让线程等待另一个线程的对象，允许他们协调工作。常用的是 **CountDownLatch** 和 **Semaphore**，少用的是 **CyclicBarrier** 和 **Exchanger**
 - b) **倒计时锁存器：一次性的障碍，允许一个或者多个线程等待其他线程。**
CountDownLatch 的唯一构造器带有 int 参数，表示等待中的线程继续之前，必须要调用的 countDown 方法的次数
一个复杂的例子：一个框架包含单个方法，这个方法带有一个执行动作的 executor，一个并发次数以及表示该动作的 Runnable。当所有工作线程都准备好之后，timer 才开始一起执行所有线程。实现方法如下
 - c) 传递给 Timer 的 execute 方法必须**创建至少与并发次数一样多的线程**，否则会产生死锁。注意对于间歇性计时，应该使用 System.nanoTime，因为这个更加精确
- e) 调试 notify-wait 遗留代码：**应该使用 wait 循环模式来调用 wait 方法；永远不要在循环之外调用 wait 方法。**

```
// The standard idiom for using the wait method
synchronized (obj) {
    while (<condition does not hold>)

        obj.wait(); // (Releases lock, and reacquires on wakeup)
        ... // Perform action appropriate to condition
}
```

Final proposal: Semaphore – the code

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
1 void down(semaphore *s) {
2     disable_interrupt();
3     while (*s == 0) {
4         enable_interrupt();
5         special_sleep();
6         disable_interrupt();
7     }
8     *s = *s - 1;
9     enable_interrupt();
10 }
```

The calls “up()” and “down()” are implemented inside the kernel.

Also, only one process can invoke “`disable_interrupt()`”. Later processes would be blocked until “`enable_interrupt()`” is called.

Section Exit: up()

```
1 void up(semaphore *s) {
2     disable_interrupt();
3     if (*s == 0)
4         special_wakeup();
5     *s = *s + 1;
6     enable_interrupt();
7 }
```

- CSCI3150 52
- a) 先测试条件，等成立时跳过等待，这对确保活性是必要的。如果条件已经成立，并且在线程等待之前，`notify` 方法已经被调用，则无法保证该线程将会从等待中苏醒过来。
 - b) 被唤醒后再测试条件，如果条件不成立继续等待，这对确保安全性是必要的。有时线程会在一下时候苏醒过来
 - a) 另一个线程可能也得到了锁，并且在调用 `notify` 的时候先一步出了循环，改变了受保护的状态
 - b) 条件并不成立，但是另一个线程可能意外地或者恶意的调用了 `notify()`
 - c) 通知线程可能过度大方得唤醒了更多的线程
 - d) 没有通知的情况下，等待线程也有可能苏醒过来（伪唤醒）

The Wikipedia [article on spurious wakeups](#) has this tidbit:

The `pthread_cond_wait()` function in Linux is implemented using the `futex` system call. Each blocking system call on Linux returns abruptly with `EINTR` when the process receives a signal. ... `pthread_cond_wait()` can't restart the waiting because it may miss a real wakeup in the little time it was outside the `futex` system call. This race condition can only be avoided by the caller checking for an invariant. A POSIX signal will therefore generate a spurious wakeup.

Summary: If a Linux process is signaled its waiting threads will each enjoy a nice, hot *spurious wakeup*.

I buy it. That's an easier pill to swallow than the typically vague "it's for performance" reason often given.

```

// Simple framework for timing concurrent execution
public static long time(Executor executor, int concurrency,
    final Runnable action) throws InterruptedException {
    final CountDownLatch ready = new CountDownLatch(concurrency);
    final CountDownLatch start = new CountDownLatch(1);
    final CountDownLatch done = new CountDownLatch(concurrency);
    for (int i = 0; i < concurrency; i++) {
        executor.execute(new Runnable() {
            public void run() {
                ready.countDown(); // Tell timer we're ready
                try {
                    start.await(); // Wait till peers are ready
                    action.run();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    done.countDown(); // Tell timer we're done
                }
            }
        });
    }
    ready.await(); // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown(); // And they're off!
    done.await(); // Wait for all workers to finish
    return System.nanoTime() - startNanos;
}

```

- c) 何时使用 **notify** 或者 **notifyAll**
 - a) 常见做法是总是使用 **notifyAll**。这总会产生正确的结果，因为它可以保证所有需要被唤醒的线程都被唤醒。多余的线程就会检查条件并且继续等待
 - b) **notifyAll** 可以保证不受不相关线程意外或者恶意的等待。这样的线程有可能吞掉一个关键的通知，使真正的线程无线等待下去
 - c) **Notify** 在性能方面有优势

70. 线程安全性的文档化

- a) 如果没有适当的文档，程序员有可能对类的线程安全性做出错误的假设。从而导致过度同步或者缺乏同步，引发严重错误
- b) 通过查看文档中是否出现 **synchronized** 修饰符，可以确定一个方法是线程安全的。**这种说法是错误的！** 在一个方法声明里出现 **synchronized** 修饰符，这是实现细节，而不是导出 API 的一部分，不能表现这个方法是线程安全的。
- c) **一个类为了可被多个线程安全使用，必须在文档中清楚地说明它所支持的线程安全级别**
 - a) 不可变的 **immutable**。这个类是不变的，不需要同步。例子：String, Long 等
 - b) 无条件的线程安全 **unconditionally thread-safe**。这个类的实例是可变的，但是在内部做了足够的同步。它可以被并发使用，不需要外部同步。例子：ConcurrentHashMap
 - c) 有条件的线程安全 **conditionally thread-safe**。这个类的实例的某些方法需要外部同步，其他与无条件的类相同。例子：Collection.synchronized 集合，iterator 需要同步。文档在描述一个有条件的线程安全类要特别小心，必须指明哪个调用序列需要

外部同步，以及需要那些锁。如果某个对象是另外一个对象的视图，那就必须要锁定后台对象。例子：Collection.synchronizedMap

It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views:

(当遍历任何被返回Map的集合视图时，用户必须手工对它们进行同步：)

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<K, V>());
...
Set<K> s = m.keySet(); // Needn't be in synchronized block
...
synchronized(m) { // Synchronizing on m, not s!
    for (K key : s)
        key.f();
}
```

- d) 非线程安全 **not thread-safe**。这个类的实例是可变的。为了并发使用，必须用外部同步包围每个方法调用。例子：ArrayList 和 HashMap
- e) 线程对立的 **thread-hostile**。这个类即使被外部同步全部包围，也不能被并发使用。什么鬼类。。。
- d) 当一个类承诺“使用一个公共的可访问锁对象”时，意味着它允许客户端以原子方式执行方法调用序列。但是这种灵活性需要付出代价。并发集合的并发控制不能与高性能的内部并发控制兼容。客户端还可以发起拒绝服务攻击：只需要超时保留公共锁。
解决方法：使用私有锁对象。

```
// Private lock object idiom - thwarts denial-of-service attack
private final Object lock = new Object();

public void foo() {
    synchronized(lock) {
        ...
    }
}
```

这样客户端无法妨碍对象的同步。私有锁对象只能用在无条件的线程安全类上，特别适合那些专门为继承而设计的类，以防被子类干扰。

71. 慎用延迟初始化

- a) 延迟初始化是等到需要时在初始化的行为。除非绝对必要，否则不要这么做。大多数情况下，正常的初始化优先于延迟初始化。
- b) 适用延迟初始化：如果域只在类的实例部分被访问，而且初始化这个域的开销很高。应该测量性能
- c) 如果多个线程共享一个延迟初始化域，那么应该采取同步措施。例子如下

```

// Lazy initialization of instance field - synchronized accessor
private FieldType field;

synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}

```

- d) 如果处于性能考虑要对 static 域延迟初始化，那么应该使用 lazy initialization holder class 模式，如下。这个方法里没有使用同步，而且没有增加任何访问成本。一旦被初始化，现代 VM 会保证不受限制

```

// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}
static FieldType getField() { return FieldHolder.field; }

```

- e) 如果处于性能考虑要对实例域延迟初始化，那么应该使用双重检查模式。第一次检查是否初始化而不锁定，第二次才会锁定。域被声明为 volatile 很关键

```

// Double-check idiom for lazy initialization of instance fields
private volatile FieldType field;
FieldType getField() {
    FieldType result = field;
    if (result == null) { // First check (no locking)
        synchronized(this) {
            result = field;
            if (result == null) // Second check (with locking)
                field = result = computeFieldValue();
        }
    }
    return result;
}

```

使用局部变量 result：保证 volatile 变量只被访问一次，提高效率

- f) 如果处于性能考虑要对可以接受重复初始化的实例域延迟初始化，那么应该使用单重检查模式。注意 field 仍然被声明为 volatile

```

// Single-check idiom - can cause repeated initialization!
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
    if (result == null)
        field = result = computeFieldValue();
    return result;
}

```

- g) 如果域的类型是基本类型，而且不是 double 和 long，那么就可以删去 volatile 描述符。

72. 不要依赖线程调度器

- a) 任何依赖线程调度器来达到正确性或者性能要求的程序，很有可能是不可移植的。
- b) 要编写健壮的可移植的多线程程序，应该确保可运行线程的平均数量不明显多于处理器数量。注意是可运行线程，而不是线程。
- c) 减少可运行线程的方法是：如果线程没有在做有意义的事，就不应该运行。任务应该有合适的大小。
- d) 线程不应该一直处于忙等待状态：增加 CPU 负载
- e) 如果一个程序不能工作是因为某些线程无法像其他线程一样得到充足的 CPU 时间，那么不要企图通过调用 `Thread.yield` 来修正。最好的做法是重构程序，减少并发线程数量
- f) 线程优先级是最不可移植的特征。不要调整线程优先级。
- g) 不要使用 `Thread.yield` 来进行调试。应该使用 `Thread.sleep(1)` 来进行测试

73. 避免使用线程组

- a) 线程组的初衷是隔离 applet 的机制，但是他们的安全价值已经差到根本不在 Java 安全模式的标准工作中提及的底部。
- b) 从线程安全性的角度来看，`ThreadGroup API` 非常弱。线程组已经过时了，实际上根本没有必要修正。如果你正设计的类需要处理线程的逻辑组，那么就应该使用线程 `executor`。

第十一章 序列化

74. 谨慎的实现 `Serializable` 接口

- a) 实现 `Serializable` 接口而付出的最大代价是，一旦一个类被发布，就大大降低了“改变这个类的实现”的灵活性。如果一个类实现了 `Serializable` 接口，它的字节流编码便成了 API 的一部分。如果接受了默认的序列化形式，那这种序列化类型会被永久束缚在类的内部表示法上，改变可能导致序列化形式的不兼容。一定要仔细设计高质量的序列化方案
- b) 序列化会使类的演变受限，这种限制与类的序列版本 `serial version UID` 有关。如果没有显示指定这个值，系统会自动根据类名称，接口名称共有域名称等自动计算。如果增加了一个公有方法，`UID` 也会变化。如果没有声明 `UID`，兼容性会被破坏
- c) 实现 `Serializable` 的第二个代价是，它增加了出现 `bug` 和安全漏洞的可能性。反序列化机制是一种语言之外的对象创建机制，因此也必须保证所有“有真正的构造其建立起来的构造关系”，而且不允许攻击者访问对象的内部信息。
- d) 实现 `Serializable` 的第三个代价是，随着类发行新的版本，相关的测试负担也增加了。应该测试在新/旧版本实例化一个对象，在另一方版本反序列化。这些测试不可能自动构建，因为除了二进制兼容性，还要测试语义兼容性，保证结果对象是原始对象的真正复制。
- e) 实现 `Serializable` 接口的优劣
 - a) 优点：
 - a) 如果一个类将要加入某个依赖序列化来实现对象传输或持久化的框架，就应该

实现这个接口。

- b) 如果这个类要成为一个类的组件，且后者实现了 Serializable 接口，那么前者也应该实现这个接口。值类或者大多数集合类都应该实现 Serializable 接口
- b) 缺点
 - a) 为了继承而设计的类/用户接口尽可能少实现 Serializable 接口，除非是必须的情况（以上）。实现了 Serializable 接口的有 Throwable 类，component 类或者 HttpServlet 抽象类。
 - b) 如果类有一些约束条件，当类的实例域被初始化成默认值时，就会违背这些约束条件，这时候就必须 `readObjectNoData()` 函数

```
// readObjectNoData for stateful extendable serializable classes
private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Stream data required");
}
```

- c) 如果一个为继承而设计的类不可序列化，就不可能编写可序列化的子类。如果超类没有提供可访问的无参构造器，子类也不可能可序列化。因此，为继承而设计的不可序列化的类，都应该考虑一个无参构造器。
- d) 最好在所有约束关系都已经建立的情况下在创建对象。盲目为一个类增加无参构造器，而它的约束关系由其他构造器来建立，就会更加复杂易出错。
- e) 一个解决方法：给不可序列化但可扩展的类添加无参构造器，避免以上错误。

```
// Nonserializable stateful class allowing serializable subclass
public abstract class AbstractFoo {
    private int x, y; // Our state

    // This enum and field are used to track initialization
    private enum State { NEW, INITIALIZING, INITIALIZED };
    private final AtomicReference<State> init =
        new AtomicReference<State>(State.NEW);

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

    // This constructor and the following method allow
    // subclass's readObject method to initialize our state.
    protected AbstractFoo() { }
    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 so 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");
    }
    ... // Remainder omitted
}
```

初始化方法与正常的构造器一致，也建立了相同的约束关系。私有实例都有 initialize() 来设置。该抽象类的实例方法在完成任何工作之前都必须先调用 checkInit()，以确保如果有子类没初始化实例，方法就会立刻失败。注意 init 域是一个原子引用，确保对象的完整性。这个模式利用 compareAndSet() 方法来操作枚举的原子引用，这是一个很好的线程安全状态机

- f) 内部类不应该实现 **Serializable**。内部类的默认序列化形式是定义不清楚，静态成员类可以实现 **Serializable** 接口。

75. 考虑使用自定义的序列化形式

- a) 如果没有先认真考虑默认的序列化形式是否合适，则不要贸然接受。要从多个角度考虑
- b) 默认的序列化形式描述了该对象内部所包含的数据，以及每一个可以从这个对象到达其他对象的内部数据（拓扑结构）。如果一个对象的物理表示法等同于他的逻辑内容，就可以接受默认的序列化类型。如下：

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
     * Last name. Must be non-null.
     * @serial
     */
    private final String lastName;

    /**
     * First name. Must be non-null.
     * @serial
     */
    private final String firstName;

    /**
     * Middle name, or null if there is none.
     * @serial
     */
    private final String middleName;

    ... // Remainder omitted
}
```

即使确定了默认的序列化类型，通常还必须提供一个 **readObject** 方法保证约束关系

注意：即使三个实例域都是私有的，也必须在文档中用 @serial 标签标出。

- c) 以下是一个极端的例子：

```
// Awful candidate for default serialized form
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }

    ... // Remainder omitted
}
```

如果接受默认的序列化类型，JVM 将使用图的遍历方法遍历整个拓扑接口。

- a) 它使这个类的导出 API 永远束缚在该类的内部表示法。就算这个类不再用链表作为内部数据表示法，也必须接受链表形式的输入输出
 - b) 它会消耗过多的空间
 - c) 它会消耗过多的时间。图遍历非常耗时
 - d) 它会引起栈溢出。
- d) 正确的方法是这样

```
// StringList with a reasonable custom serialized form
public final class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;

    // No longer Serializable!
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    // Appends the specified string to the list
    public final void add(String s) { ... }

    /**
     * Serialize this {@code StringList} instance.
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted {@code int}, followed by all of
     * its elements (each a {@code String}), in the proper
     * sequence.
     */
    private void writeObject(ObjectOutputStream s)
        throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }

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

        s.defaultReadObject();
        int numElements = s.readInt();

        // Read in all elements and insert them in list
        for (int i = 0; i < numElements; i++)
            add((String) s.readObject());
    }
    ...
}
```

readObject 和 writeObject 的首要任务是调用 defaultReadObject 或者 defaultWriteObject。如果所有的实例域都是 transient，技术上也是可以的，但不建议这么做。**这样调用会大**

大增加灵活性和兼容性。否则会抛出 StreamCorrupted 异常

注意尽管 read/writeObject 也是私有的，但是仍需要文档注释。原因同上。

- e) 如果对象的约束方法要依赖于特定实现的细节，例如 **HashMap**，默认的序列化可能会导致严重的 bug，破坏约束关系
- f) 每一个可以标记为 **transient** 的实例域都必须标记上，包括那些冗余的域。在决定把一个域做成非 transient 的之前，请一定确定它的值是该对象逻辑状态的一部分。非 **transient** 域会被默认化成初始值。如果不能接受这些初始值，就需要在 **readObject** 方法里先调用 **defaultReadObject**，再把这些 **transient** 域恢复成正常的值。或者使用延迟初始化。
- g) 如果在读取整个对象状态的其它任何方法上强制任何同步，则也必须在对象序列化上强制同步。如下

```
// writeObject for synchronized class with default serialized form
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
    s.defaultWriteObject();
}
```

- h) 不管选择了哪种序列化方法，都要为这个序列化类声明一个显式的序列版本 **UID**。这样可以避免潜在的不兼容危险，也会提高性能。如果需要修改一个现有的没有序列 **UID** 类，就必须要旧类的 **UID**。
- i) 如果想为类生成新版本，这个类与现有的类不兼容，那么只需要修改序列版本 **UID** 的值。

76. 保护性地编写 readObject 方法

- a) 用 39 条的类做例子

```
// Immutable class that uses defensive copying
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start the beginning of the period
     * @param end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(
                start + " after " + end);
    }

    public Date start () { return new Date(start.getTime()); }
    public Date end () { return new Date(end.getTime()); }
    public String toString() { return start + " - " + end; }
    ...
}
```

如果只加上 implements Serializable，就无法保证它的关键约束。**readObject** 实际相当于一个公有的构造器，接受字节流作为唯一的输入。所以 **readObject** 也必须做保护性拷贝以及检查参数的有效性。

因为可以使用二进制流附带额外的引用，所以应该对所有的实例域做保护性拷贝，并且去掉 final 修饰符

```
// readObject method with defensive copying and validity checking
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Defensively copy our mutable components
    start = new Date(start.getTime());
    end   = new Date(end.getTime());

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}
```

- b) 增加一个公有的构造器，其参数对应该对象中每个非 transient 的域，无论参数值是什么都是不进行检查就可以保存在实例域中。如果无法接受这样的构造器，那么就应该提供一个显式的 **readObject** 并且做所有的参数有效性检查以及保护性拷贝。或者使用序列化代理模式。
- c) **readObject** 不可以调用可覆盖的方法，无论直接还是间接都不可以。否则子类的方法会在子类状态被反序列化之前被调用，程序可能失败。

77. 对于实例控制，枚举类型优先于 readResolve

- a) **readResolve** 特性允许你用 **readObject** 创建的实例代替另一个实例。应用下面这个方法

```
// readResolve for instance control - you can do better!
private Object readResolve() {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;
}
```

该方法忽略了被反序列化的对象，只返回该类初始化时创建的特殊实例。事实上，如果依赖 **readResolve** 进行实例控制，带有对象引用类型的所有实例域都必须声明为 **transient**。

- b) 一个不明觉厉的攻击方法，我没看懂，就不放上来了。
- c) 如果将一个可序列化的实例受控的类编写成枚举，就可以绝对保证除了所声明的常量外不会有别的实例。但是如果必须编写可序列化的类，它的实例在编译时还不知道，就无法把类变成一个枚举类型。

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;
    private String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
```

- d) **readResolve** 的可访问性很重要。如果把 **readResolve** 方法放在一个 **final** 类上，它就应该是私有的。否则，为了继承考虑就必须仔细考虑它的可访问性。如果 **readResolve** 方法是受保护的或者公有的，并且子类没有覆盖，那么对序列化后的子类实行进行反序列化就会得到异常。

78. 考虑用序列化代理模式

- a) 序列化代理模式相当简单。首先为可序列化的类设计一个私有的静态嵌套类，精确地表示外围类的实例的逻辑状态。它应该有一个单独的构造器，以外围类实例作为参数。这个构造器只从它的参数中复制数据，不需要一致性检查/保护性拷贝。外围类和它的序列化代理都必须实现 **Serializable** 接口。这个方法不需要外围类使用非 **final** 的域

```
// Serialization proxy for Period class
private static class SerializationProxy implements Serializable {
    private final Date start;
    private final Date end;

    SerializationProxy(Period p) {
        this.start = p.start;
        this.end = p.end;
    }

    private static final long serialVersionUID =
        234098243823485285L; // Any number will do (Item 75)
}
```

接着把这个方法加到外围类，在序列化之前将外围类的实例转变成了它的序列化代理。

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

此时外围类的 **readObject** 应该抛出异常

```
// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream)
    throws InvalidObjectException {
    throw new InvalidObjectException("Proxy required");
}
```

最后，在内部的代理类提供 **readResolve** 方法，返回一个逻辑上相当的外围类实例。

```
// readResolve method for Period.SerializationProxy
private Object readResolve() {
    return new Period(start, end); // Uses public constructor
}
```

这个**readResolve**方法仅仅利用它的公有API创建外围类的一个实例，这正是该模式的魅力之所在。它极大地消除了序列化机制中语言本身之外的特征，因为反序列化实例是利用与任何其他实例相同的构造器、静态工厂和方法而创建的。这样你就不必单独确保被反序列化的实例一定要遵守类的约束条件。如果该类的静态工厂或者构造器建立了这些约束条件，并且它的实例方法在维持着这些约束条件，你就可以确信序列化也会维持这些约束条件。

- b) 序列化代理模式有三个缺点
 - a) 不可以与可以被客户端拓展的类兼容
 - b) 不能与对象图中包含循环的某些类兼容
 - c) 开销比较大
- c) 终于整理完了偶也！