第一章 创建与销毁

1. 静态工厂方法

- **优点**
  - 与 `构造器` 相比，有 `名称` ；
  - 与 `构造器` 相比，不必每次都创建新对象；
  - 与 `构造器` 相比，可以返回原返回类型的任何子类型；（如：`java集合框架` 的 `Collections` 类）

    发行版本1.5中引入的类java.util.EnumSet（见第32条）没有公有构造器，只有静态工厂方法。它们返回两种实现类之一，具体则取决于底层枚举类型的大小：如果它的元素有64个或者更少，就像大多数枚举类型一样，静态工厂方法就会返回一个RegalarEumSet实例，用单个long进行支持；如果枚举类型有65个或者更多元素，工厂就返回JumboEnumSet实例，用long数组进行支持。

    这两个实现类的存在对于客户端来说是不可见的。如果RegularEnumSet不能再给小的枚举类型提供性能优势，就可能从未来的发行版本中将它删除，不会造成不良的影响。同样地，如果事实证明对性能有好处，也可能在未来的发行版本中添加第三甚至第四个EnumSet实现。客户端永远不知道也不关心他们从工厂方法中得到的对象的类；他们只关心它是EnumSet的某个子类即可。
  - 与构造器相比，创建参数化类型实例时，代码更加简洁——这一点好像在jdk8中，并没有区别；
    书中**举例**：使用 `Map<String, List<String>> map = HashMap.newInstance()` 代替 `Map<String, List<String>> map = new HashMap<String, List<String>>()` ，理由是静态工厂方法可以进行类型推导（值得推荐）；

- **缺点**
  - 类如果不包含公有的或受保护的构造器，就不能被子类化；
  - 在查找方法的时候，名字不好找，不像构造器那样被特别标注；一般可以遵守默认命名规则：`valueOf/of` （一般用于类型转换）、`getInstance` 、`getType/newType` （当静态工厂方法在不同类中时）等；

2. 多参数时，考虑用构建器（ `builder` ）

- `重叠构造器` 模式：参数少时，比较好；参数太多后，果断放弃；
- `JavaBeans` 模式：缺点-使保证bean一致性变得困难(因为构造器过程被分到了几个调用中，在构造中 `JAVABean` 可能处于不一致的状态)；

```
//JAVABean模式
public class Temp2 {
  private int p1;//必要参数
  private int p2;
  private int p3;
  private int p4;

  public Temp2(int p1){this.p1 = p1; }
  public void setP1(int p1){ this.p1 = p1; }
  public void setP2(int p2){ this.p2 = p2; }
  public void setP3(int p3){ this.p3 = p3; }
  public void setP4(int p4){ this.p4 = p4; }
}

Temp2 t1 = new Temp2(1);
t1.setP2(2);
t1.setP4(4);
```

- `Builder` 模式：builder是它构建类的静态内部类，可先通过builder的 `setter` 方法设置属性，然后调用 `builder.build()` 方法，构建对象；

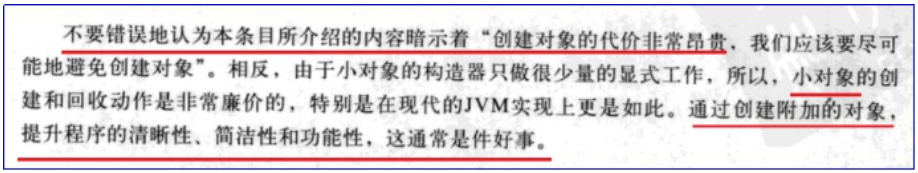
3. 枚举实现单例模式

- 抵御通过 `反射机制` 生成第二实例的方法：构建第二实例的时候抛出异常;
- 单例模式中，如果类是可序列化的（实现 `Serializable` 接口），必须重写 `readResolve` 方法，不然，每次反序列化都会产生一个实例；
- `单元素的枚举类` 是最好的实现单例模式的方法——既可防止反射攻击，也可防止反序列化产生多实例;

4. 使用私有构造器强化不可实例化能力

5. 避免创建不必要的对象

- `String a = new String("string");` 此句创建了2次实例：参数 `string` 就是一个实例；
- 优先使用 `基本类型` ，而不是 `封装类型` ；
- 有时候 `重用对象` 会导致代码很乱，逻辑糟糕，比重建对象的代价更大：


不要错误地认为本条目所介绍的内容暗示着“创建对象的代价非常昂贵，我们应该要尽可能地避免创建对象”。相反，由于小对象的构造器只做很少量的显式工作，所以，小对象的创建和回收动作是非常廉价的，特别是在现代的JVM实现上更是如此。通过创建附加的对象，提升程序的清晰性、简洁性和功能性，这通常是件好事。

6. 及时清除过期引用

- `缓存`、`监听器`/其他`回调`都比较容易发生内存泄漏

7. 避免使用终结方法

- 从一个对象变得`不可到达`开始，到它的`终结方法`执行，所花费的时间是任意长的；所以，类似在终结方法中关闭文件的做法的错误的；
- 何时执行终结方法也是`垃圾回收算法`的一个功能，而垃圾回收算法在不同的`jvm实现`中会大相径庭，如果依赖`finallizer`，那么不同`jvm`中实现会截然不同；
- 有时候`finallizer`是否执行都不能保证：程序终止，而`finallizer`方法却没执行；
- 不要被`System.runFinalizersOnExit()`和`Runtime.runFinalizersOnExit()`诱惑，它们都有致命缺陷（多线程情况）；
- `Finallizer`中的异常不会被打印，容易被忽略；
- `Finallizer`增加性能损耗；
- 建议使用`try...finally`；
- 子类如果重写了终结方法（`finallizer`），则必须再调用超类的终结方法；终结方法守卫者可以防止粗心大意而没有执行`super.finallizer`：

```
public class Parent {
  public static void main(final String[] args) throws Exception {
    doSth();
    System.gc();
    Thread.sleep(2000);
  }

  private static void doSth() {
    Child c = new Child();
    System.out.println(c);
  }

  @SuppressWarnings("unused")
  private final Object guardian = new Object() {
    @Override
```

```
    protected void finalize() {
      System.out.println("父类中匿名内部类--终结方法守卫者 重写的
finalize()执行了");
      // 在这里调用Parent重写的finalize即可在清理子类时调用父类自己的清理方
法
      parentlFinalize();
      // 注
      // Parent.this.finalize(); 这样写不对，会执行Child重写的
finalize()方法
    }
  };

  private void parentlFinalize() {
    System.out.println("父类自身的终结方法执行了");
    // 一些逻辑..
  }

  @Override
  protected void finalize() {
    parentlFinalize();
  }
}
class Child extends Parent {
  @Override
  protected void finalize() {
    System.out.println("子类finalize方法执行了，注意，子类并没有调用
super.finalize()");
    // 由于子类（忘记或者其他原因）没有调用super.finalize()
    // 使用终结方法守卫者可以保证子类执行finalize()时(没有调用
super.finalize())，父类的清理方法仍旧调用
    // "finally中显式调用super.finalize()"没被执行之后的另一种保障对象被及
时销毁的措施
  }
}
```

输出：

```
Child@131b92e6
子类finalize方法执行了，注意，子类并没有调用super.finalize()
父类中匿名内部类--终结方法守卫者 重写的finalize()执行了
父类自身的终结方法执行了
```

第二章 对于所有对象都通用的方法

8. 覆盖 `equals` 方法的通用约定

- 类的每个实例实质上都是唯一的；
- 不关心类是否提供 逻辑相等 的测试功能；
- 超类的 `equals` 方法也适合子类；
- 不明白：

  • 类是私有的或是包级私有的，可以确定它的equals方法永远不会被调用。在这种情况下，无疑是应该覆盖equals方法的，以防它被意外调用：

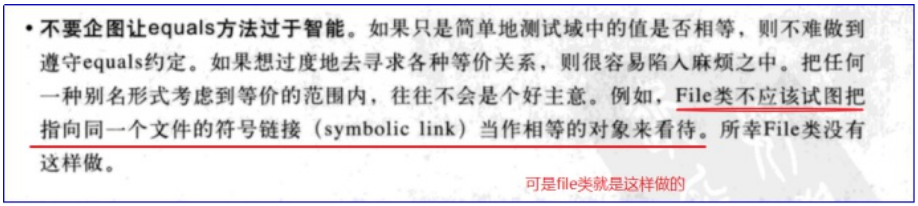
  ```
  @Override public boolean equals(Object o) {
      throw new AssertionError(); // Method is never called
  }
  ```

- 什么时候需要覆盖 `equals` 方法：

  那么，什么时候应该覆盖Object.equals呢？如果类具有自己特有的“逻辑相等”概念（不同于对象等同的概念），而且超类还没有覆盖equals以实现期望的行为，这时我们就需要覆盖equals方法。这通常属于“值类（value class）”的情形。

  对于枚举类，逻辑相等和对象相等时一个意思，所以没必要覆盖 `equals` ；
- 覆盖 `equals` 需要遵守几个特性： 自反性 、 对称性 、 传递性 、 一致性 、以及 `null` ；
- 氏替换原则 简单粗暴的理解：任何基类可以出现的地方，子类一定可以出现；
- `Timestamp` 类（ `Date` 的子类，增加了 `nanoseconds` 域）和 `Date` 类不要混合使用，混合情况下回违反 `equals` 的 自反性 ；
- `Equals` 优化：
  - 使用 `==` 检查对象引用;
  - 使用 `instanceof` 检查类型；
  - 把参数转化成正确的类型（如： `date` 转成 `long` ）；
  - 调整域的比较顺序：

    域的比较顺序可能会影响到equals方法的性能。为了获得最佳的性能，应该最先比较最有可能不一致的域，或者是开销最低的域，最理想的情况是两个条件同时满足的域。
- 重写 `equals` 的时候也要重写 `hashcode` ；
- 不要将 `equals(Object obj)` 中的 `Object` 替换为其他类型（如： `MyClass` ），这样就不是重写了，而是重载；添加 `@Override` 可以避免；
- 尴尬：


• 不要企图让equals方法过于智能。如果只是简单地测试域中的值是否相等，则不难做到遵守equals约定。如果想过度地去寻求各种等价关系，则很容易陷入麻烦之中。把任何一种别名形式考虑到等价的范围内，往往不会是个好主意。例如，File类不应该试图把指向同一个文件的符号链接（symbolic link）当作相等的对象来看待。所幸File类没有这样做。

可是file类就是这样做的

9. 重写 `equals` 的时候也要重写 `hashcode`

- `Hashcode` 也有一致性；
- 如果 `equals` 返回 `true`，那么 `hashcode` 也要相等，反之，不一定，但是不 `equals` 的对象，返回不同的 `hash` 值，有可能提高 `hash` 性能；
- 如果一个类是不可变的，并且计算 `hash` 码的开销也比较大，应该考虑将 `hash` 值缓存起来；
- 要慎重考虑计算 `hash` 值得时候舍弃某些字段的得与失；

10. 最好子类都重写 `toString` 方法

- 如果 `tostring` 方法用于持久化，那么请确定长久规范；

11. 谨慎重写 `clone`

- 数组上调用 `clone` 返回的数组类型和原类型一样；
- `clone` 结构与指向可变对象的 `final` 域的正常用法是不兼容的（文中例子是一个数组 `elemnts` 的 `clone`），除非原始对象和克隆对象之间可以安全地共享此可变对象。为了能够使一个类能够被克隆，请尽量将某个域的 `final` 修饰符去掉；
- 如果一个专为继承设计的类重写了 `clone` 方法，那么应该效仿 `object.clone`：声明为 `protected`、抛出 `CloneNotSupportedException` 异常、不能实现 `Cloneable`，留给子类选择的空间；
- 线程安全的类，也要保证 `clone` 方法和其他方法一样————— `线程安全`；

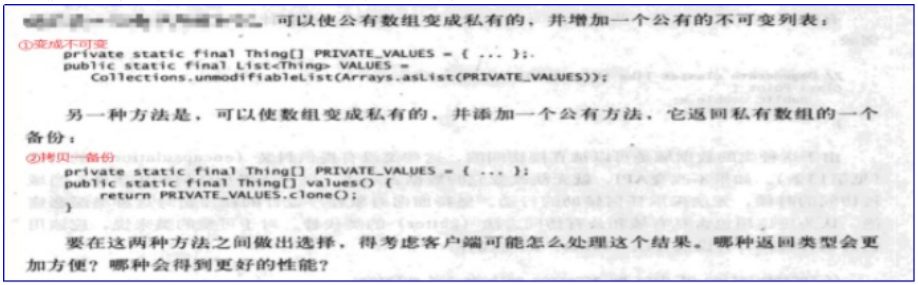
12. 考虑实现Comparable接口

- 如果创建的类是一个值类，具有明显的内排序，就应该坚定地实现 `Comparable` 接口；
- `compareTo` 和 `equals` 不需要必须等效，比如： `BigDecimal("1.0")` 和 `BigDecimal("1.00");` 但是如果使用 `treeSet` 之类的集合，则只算一个元素；

第三章 类和接口

13. 类和成员可访问性最小

- 最大透明度，称为信息隐藏或封装，软件设计原则之一；
- 好处：解耦（开发、理解、测试、维护都比较容易）；
- 实例的域决不能是公有的；
- Final域应当只包含基本类型的值或不可变对象的引用；
- final修饰的数组几乎总是错误的，解决这种矛盾的方法有2种：

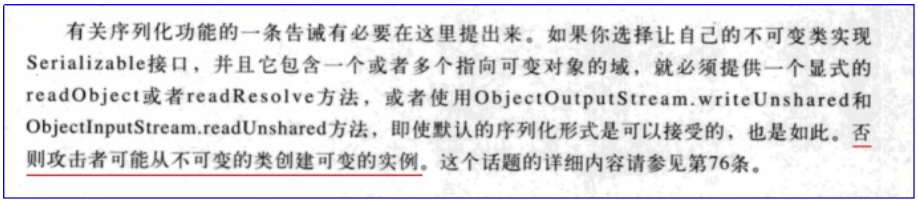


14. 避免直接访问域

- 如果公有类暴露了它的数据域，要想在将来改变其内部表示法是不可能的；

15. 使可变性最小化

- `不可变类`：实例化后不再改变（如：`string`、基本类型的包装类、`BigInteger`、`BigDecimal`等）；
- 不可变类更加容易设计、实现和使用，不易出错，更加安全；
- 设计不可变类原则：
  - 不提供修改状态的方法；
  - 保证类不被扩展（`final`）；
  - 所有域设为`private`；
  - 所有域设为`final`；
  - 避免引用可变组件（如其他可变类引用）；
- 对于不可变类，本质上就没有拷贝的必要，所以是实现`clone`是不必要的，`String`就是反面教材（jdk8中String#clone已经被移除）；
- 不可变类会造成性能的浪费（MutableBigInteger就是BigInteger的性能优化版）；
- 另外：



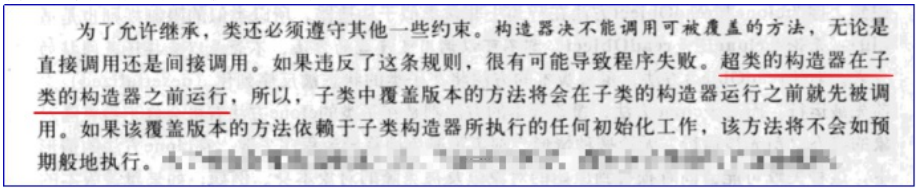
有关序列化功能的一条告诫有必要在这里提出来。如果你选择让自己的不可变类实现Serializable接口，并且它包含一个或者多个指向可变对象的域，就必须提供一个显式的readObject或者readResolve方法，或者使用ObjectOutputStream.writeUnshared和ObjectInputStream.readUnshared方法，即使默认的序列化形式是可以接受的，也是如此。否则攻击者可能从不可变的类创建可变的实例。这个话题的详细内容请参见第76条。

16. 复合（`composition`）优先于继承

- 继承比较脆弱，如果新版本中添加新的方法很可能对子类造成很大影响，导致不稳定。复合不存在这种问题，复合类似于 适配器 模式；

17. 要么为继承而设计，并提供文档说明，要么禁止继承

- 关于文档：好的 `api文档` 应该描述一个给定的方法做了什么，而不是如何做的；
- 构造器决不能调用可被重写的方法：

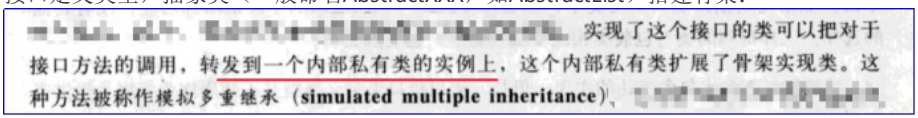


为了允许继承，类还必须遵守其他一些约束。构造器决不能调用可被覆盖的方法，无论是直接调用还是间接调用。如果违反了这条规则，很有可能导致程序失败。超类的构造器在子类的构造器之前运行，所以，子类中覆盖版本的方法将会在子类的构造器运行之前就先被调用。如果该覆盖版本的方法依赖于子类构造器所执行的任何初始化工作，该方法将不会如预期般地执行。

- 为继承而设计的类，应该慎重考虑实现 `Cloneable` 和 `Serializable` 接口；

18. 接口优于抽象类

- 现有类易被更新，以实现新的接口；比如 `jdk` 添加 `Comparable` 接口的时候；
- 接口是定义 `mixin`（混合类型）的理想选择；
- 接口定义类型，抽象类（一般命名AbstractXXX，如AbstractList）搭建骨架：



实现了这个接口的类可以把对于接口方法的调用，转发到一个内部私有类的实例上，这个内部私有类扩展了骨架实现类。这种方法被称作模拟多重继承（**simulated multiple inheritance**）。

- 公有接口的设计一定要谨慎，一旦公开发行，并被广泛实现，再想修改接口，几乎是不可能的（不过 `jdk8` 中，接口可以有默认实现）；
- 接口实现起来比抽象类灵活，但设计了接口，最好定义一个骨架（抽象类）；

19. 接口只用来定义类型

- 常量接口（只有`final`的静态域）是对接口的不良使用；

20. 类层次优于标签类

- 标签类过于冗长，易出错，且效率低下：

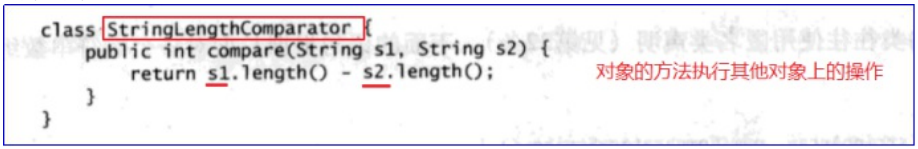
```
// Tagged class - vastly inferior to a class hierarchy
public class Figure1{
  enum Shape {
    RECTANGLE,
    CIRCLE
  }

  // Tag field - the shape of this figure
  final Shape shape;

  // These field are use only if shape if RECTANGLE
  double length;
  double width;

  // This field is use only if shape is CIRCLE
  double radius;

  // Constructor for circle
  public Figure1(double radius) {
    shape = Shape.CIRCLE;
    this.radius = radius;
  }

  // Constructor for rectangle
  public Figure1(double length, double width) {
    shape = Shape.RECTANGLE;
    this.length = length;
    this.width = width;
  }

  double area() {
    switch (shape) {
      case RECTANGLE:
      return length * width;
      case CIRCLE:
      return Math.PI * (radius * radius);
      default:
```

```
      throw new AssertionError();
    }
  }
}
```

```
/**
* 类层次优于标签类
* @author weishiyao
*
*/
// Class hierarchy replacement for a tagged class
abstract class Figure2 {
  abstract double area();
}

class Circle extends Figure2 {
  final double radius;

  Circle(double radius) {
    this.radius = radius;
  }

  double area() {
    return Math.PI * (radius * radius);
  }
}

class Rectangle extends Figure2 {
  final double length;
  final double width;

  Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }
  double area() {
    return length * width;
  }
}
```

21. 用函数对象表示策略

- 函数对象：



22. 优先考虑静态成员类

- **名词解释**
  - 定义在代码块、方法体内的类叫 局部内部类 ；
  - 函数对象 做了这么一件事，我们可以定义一个只有方法而没有数据的类，然后把这个类的对象传递给别的方法，这时传递的这个对象就是一个函数对象。jdk8的
- **访问控制修饰符**
  - `default` (即缺省，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法；
  - `private` : 在同一类内可见。使用对象：变量、方法。 注意： 不能修饰类（外部类） ；
  - `public` : 对所有类可见。使用对象：类、接口、变量、方法；
  - `protected` : 对同一包内的类和所有子类可见。使用对象：变量、方法。 注意：不能修饰类（外部类） ；
- 一个静态内部类的使用例子：

静态成员类的一种常见用法是作为公有的辅助类，仅当与它的外部类一起使用时才有意义。例如，考虑一个枚举，它描述了计算器支持的各种操作（见第30条）。Operation枚举应该是Calculator类的公有静态成员类，然后，Calculator类的客户端就可以用诸如Calculator.Operation.PLUS和Calculator.Operation.MINUS这样的名称来引用这些操作。

非静态成员类的一种常见用法是定义一个Adapter[Gamma95, p.139]，它允许外部类的实例被看作是另一个不相关的类的实例。例如，Map接口的实现往往使用非静态成员类来实现它们的集合视图（**collection view**），这些集合视图是由Map的keySet、entrySet和Values方法返回的。同样地，诸如Set和List这种集合接口的实现往往也使用非静态成员类来实现它们的迭代器（iterator）：

- 如果成员类不要求访问外围实例，就要始终添加 `static` 修饰符，因为非静态内部类总会保存一个外围实例的引用，保存这份引用会额外消耗时间和空间，并可能导致外围实例符合垃圾回收时却任然被保留。
- 当且仅当**匿名内部类**出现在 非静态环境 中时才包含外围实例的引用；（ 局部类 也是如此）
- 即使**匿名内部类**在 静态环境 中，也不可能拥有任何静态成员；（ 局部类 也是如此）
- 匿名内部类使用场景：
  - 函数对象；

- 过程对象，eg：`Runable`、`Thread`、`TimerTask`等；
- 静态工厂方法的内部；

第四章 泛型

23. 请不要在新代码中使用原生态类型

- 比如：`List<E>`对应的原生态类型是`List`；
- 泛型有子类型化规则：`List<String>`是`List`的子类，但不是`List<Object>`的子类；

泛型	术 语	示 例
	参数化的类型	List<String>
	实际类型参数	String
	泛型	List<E>
	形式类型参数	E
	无限制通配符类型	List<?>
	原生态类型	List
	有限制类型参数	<E extends Number>
	递归类型限制	<T extends Comparable<T>>
	有限制通配符类型	List<? extends Number>
	泛型方法	static <E> List<E> asList(E[] a)
	类型令牌	String.class

24. 消除非受检警告

- `SuppressWarning`可以用在任何力度的级别，应该始终在尽可能小的范围内使用`SuppressWarning`；

25. 列表优先于数组

- 数组是 协变的 （`convariant`）
  - 如果Sub是Super的子类，那么Sub[]也是Super[]的子类；
- 泛型是 不可变 的（`invariant`）
  - 对于不同类型Type1和Type2，`List<Type1>`既不是`List<Type2>`的子类，也不是的父类；

- 相较于列表（list），数组（array）是有缺陷的：

```
Object[] objcetArray = new Long[1];
objcetArray[0] = "I don't fit in"; // 抛出ArrayStoreException
```

- 为什么创建泛型数组是非法的？(如：new ArrayList[10])

> 为什么创建泛型数组是非法的？因为它不是类型安全的。要是它合法，编译器在其他正确的程序中发生的转换就会在运行时失败，并出现一个**ClassCastException**异常。这就违背了泛型系统提供的基本保证。

但是无限制通配符类型和数组可以同用，比如：List、Map ；

26. 优先考虑泛型

- 下面是一个泛型

```
public class Stack<E> {
  private E[] elements;
  private int size = 0;
  private static final int DEFAULT_INITIAL_CAPACITY = 16;
  public Stack() {
    elements = new E[DEFAULT_INITIAL_CAPACITY]; // 这样写的话会报错

  public void push(E e) {
    ensureCapacity();
    elements[size++] = e;
  }
  public E pop() {
    if (size == 0)
      throw new EmptyStackException();
    E result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
  }
  ... // no changes in isEmpty or ensureCapacity
}
```

上面的例子中`elements = new E[DEFAULT_INITIAL_CAPACITY];`会报错误或警告，是因为不能创建泛型数组（见“列表优先于数组”）；
解决`方案一`：

```
@SuppressWarnings("unchecked")  // 此处确定是安全的，可以抑制掉非受检警告
elements = (E[])new Object[DEFAULT_INITIAL_CAPACITY];
```

解决 方案二：

```
// 定义elements为Object数组
private Object[] elements;

//修改pop方法
public E pop() {
    if (size == 0)
      throw new EmptyStackException();
    @SuppressWarnings("unchecked") // 此处确定是安全的，可以抑制掉非受检警告
    E result = (E)elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
  }
```

27. 优先考虑泛型化

- 没有类型推导的jdk6中：

```
static <T> T pick(T a1, T a2) { return a2; }
// 比较特别的语法 this.<Serializable>
Serializable s = this.<Serializable>pick("d", new ArrayList<String>
());
```

28. 利用有限通配符来提升API的灵活性

- 如下代码：假如SubE是E的子类，则 `pushAll(Iterable<SubE>)` 在代码一的情况下，就会出错，因为 参数化类型 是 不可变的 （ `List<SubE>` 既不是 `List<E>` 的子类，也不是 `List<E>` 的超类）；代码二就刚好解决这问题

```
// 代码一
public void pushAll(Iterable<E> src) {
  for (E e : src)
    push(e);
}
```

```
// 代码二
public void pushAll(Iterable<? extends E> src) {
  for (E e : src)
    push(e);
}
```

29. 优先使用类型安全的异构容器

- 什么是类型安全的异构容器，我也没明白，读者自己体会，下面列出书中代码示例：

```
public class Favorites {
  private Map<Class<?>, Object> favorites = new HashMap<>();

  public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(Objects.requireNonNull(type), instance);
  }

  public <T> T getFavorite(Class<T> type) {
    // java.lang.Class<T>#cast
    return type.cast(favorites.get(type));
  }
}
```

书中提到 `Favorites` 有2个 局限性：

- 类型安全容易被破坏

Favorites类有两种局限性值得注意。首先，恶意的客户端可以很轻松地破坏Favorites实例的类型安全，只要以它的原生态形式（raw form）使用Class对象。但会造成客户端代码在编译时产生未受检的警告。这与一般的集合实现，如HashSet和HashMap并没有什么区别。你可以很容易地利用原生态类型HashSet（见第23条）将String放进HashSet<Integer>中。也就是说，如果愿意付出一点点代价，就可以拥有运行时的类型安全。确保Favorites永远不违背它的类型约束条件的方式是，让putFavorite方法检验instance是否真的是type所表示的类型的实例。我们已经知道这要如何进行了，只要使用一个动态的转换：

```
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

解决方案：添加类型转换检查

- 局限二没有很好的解决方案

Favorites类的第二种局限性在于它不能用在不可具体化的（non-reifiable）类型中（见第25条）。换句话说，你可以保存最喜爱的String或者String[]，但不能保存最喜爱的List<String>。如果试图保存最喜爱的List<String>，程序就不能进行编译。原因在于你无法为List<String>获得一个Class对象：List<String>.Class是个语法错误，这也是件好事。List<String>和List<Integer>共用一个Class对象，即List.class。如果从“字面（type literal）”上来看，List<String>.class和List<Integer>.class是合法的，并返回了相同的对象引用，就会破坏Favorites对象的内部结构。

第五章 枚举和注解

30. 使用 `Enum` 代替 `int` 常量

- `Enum` 天生不可变的，它的所有域都是 `final` 的；
- 枚举能够有效地解决 `int枚举模式` 和 `string枚举模式` 的缺点

```
// int枚举模式
// 缺点：难以通过int关联到枚举名称
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;

// string枚举模式
// 缺点：性能消耗
public static final String APPLE_FUJI = "APPLE_FUJI";
public static final String APPLE_PIPPIN = "APPLE_PIPPIN";
public static final String APPLE_GRANNY_SMITH =
"APPLE_GRANNY_SMITH";
```

这2种模式，在编译的时候即使运用错误，也难以察觉；

- 关于**getDeclaringClass**方法

> Two enum constants e1 and e2 are of the same `enum type` if and only if e1.getDeclaringClass() == e2.getDeclaringClass(), is `enum type`.

- `Enum` 在编译的时候会进行类型检查；

- `Enum`方便扩展，可以添加任意多的方法；
  - 如此，在程序中，`Enum`就可以通过扩展，从简单的常量集合，渐渐完善成为功能齐全的抽象；
- `Enum`是先了`Comparable`和`Serializable`接口；
- `Enum`有一个静态方法：values()
  - 返回所有枚举实例；
- 坏代码改进：

```
public enum Operation {
  PLUS, MINUS, TIMES, DIVIDE;
  // Do the arithmetic operation represented by this constant
  public 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);
  }
}
```

上面的代码很脆弱，当添加新的枚举常量，如果忘记修改`switch`语句，就可能抛出异常；改善方案：

```
public enum 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;}};

  public abstract double apply(double x, double y);
}
```

如此，在添加新的枚举常量的时候，就不会忘记添加`apply`方法的实现；

- `Enum`的`toString()`方法返回的是枚举常量的name值，在通过`valueOf(name)`可以得到枚举常量的实例，所以如果重写了`toString()`方法，最好提供一个与`valueOf(name)`相似功能的方法（比如：`fromString(toString())`）来得到枚举常量实例；
- 当多个枚举常量共享相同行为的时候，考虑用`策略枚举`：

```
// 缺陷：如果添加一种新的常量，而忘记修改switch语句，那么将造成巨大损失；

// 此处工资计算暂且用double，实际应该使用BigDecimal
enum PayrollDay {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

  private static final int MINS_PER_SHIFT = 8 * 60;

  int pay(int minutesWorked, int payRate) {
    int basePay = minutesWorked * payRate;
    int overtimePay;
    switch(this) {
      case SATURDAY:
      case SUNDAY: // Weekend
        overtimePay = basePay / 2;
        break;
      default: // Weekday
        overtimePay = minutesWorked <= MINS_PER_SHIFT ? 0 :
(minutesWorked - MINS_PER_SHIFT) *
        payRate / 2;
    }
    return basePay + overtimePay;
  }
}
```

改进方案：使用内部枚举类作为工资结算策略

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

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

  int pay(int minutesWorked, int payRate) {
    return payType.pay(minutesWorked, payRate);
  }

  // The strategy enum type
  private enum PayType {
    WEEKDAY {
      int overtimePay(int minsWorked, int payRate) {
```

```
        return minsWorked <= MINS_PER_SHIFT ? 0 : (minsWorked -
MINS_PER_SHIFT) * payRate / 2;
      }
    },WEEKEND {
      int overtimePay(int minsWorked, int payRate) {
        return minsWorked * payRate / 2;
      }
    };

    abstract int overtimePay(int mins, int payRate);
    private static final int MINS_PER_SHIFT = 8 * 60;

    int pay(int minsWorked, int payRate) {
      int basePay = minsWorked * payRate;
      return basePay + overtimePay(minsWorked, payRate);
    }
  }
}
```

- `Enum` 在装载和初始化时会有些许性能消耗；
  - 除了手机、烤面包机等微型设备之外，其他设备可以忽略；

31. 用实例域代替序数

- 当你添加或减少枚举常量时， `ordinal` 会跟随变动；
- 忠告： `ordinal` 是为像 `EnumSet` 、 `EnumMap` 这种基于枚举的通用数据结构而设计的；如果你不是在设计这种结构，最好尽量避免使用 `ordinal` ；
- 代码优化示例：

```
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);

  // 实例域
  private final int numberOfMusicians;

  Ensemble(int size) { this.numberOfMusicians = size; }
  public int numberOfMusicians() { return numberOfMusicians; }
}
```

32. 用 `EnumSet` 代替位域

- `EnumSet` 回顾：
  - 当对应的枚举类包含的枚举常量数量小于64个时，jdk采用 `RegularEnumSet` (一个 `long` 变量保存set的元素状态: `long` 的 `64bit` 对应的 `1` 和 `0` ，表示 `set` 是否包含对应 `ordinal` 的枚举常量)；
  - 大于64个时，采用 `JumboEnumSet` (用 `long` 数组来保存set的元素状态)；

33. 使用 `EnumMap` 代替序数索引

- `EnumMap` 回顾：
  - `EnumMap` 使用数组保存所有value，用Enum常量的 `ordinal` 来表示数组索引；
- 如果想使用 `ordinal` 作为数组的索引，最好使用 `EnumMap` ；
- 如果是表示类似的多维数组，可以使用 `EnumMap<Enum, EnumMap<...>>` ；

34. 用接口模拟可伸缩的枚举

- 书中例子，可以很好地帮助理解：

```
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;
  }
}
```

如果想要扩展计算操作功能，直接新建枚举类并实现 `Operation` 接口就好

```
// 扩展幂、取余计算；
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;
  }
}
```

35. 注解优于命名模式

36. 坚持使用 `Override` 注解

37. 用标记接口定义类型

- 比如： `Serializable` 、spring中的 `Aware` 接口；

- `Set` 就是一个 `有限制的标记接口`
  - `Set` 继承了 `Collection` 接口，但是没有添加任何其他方法，只是修改了方法的内部实现；
  - `Set` 相当于一个标记接口，实现/继承 `Set` 的类/接口会被限制在特定的功能范围内（set 的特性）而区别于其他集合类型；
- `标记接口` 相对 `标记注解` 的优点
  - 标记接口可以在编译阶段检测类型，而标记注解只能到运行时才检测出来；
  - 第二个优点（勉强算是个优点）：可以更加精确地进行锁定，比如：`Set`；
- 但是 `标记注解` 也有自己的优点
  - 相对于 `标记接口`，注解可以更好地进行扩展，而接口定义之后一般很难再进行改变；

第六章 方法

38. 检查参数的有效性

- 如果是公有方法，要用 `Javadoc` 的 `@throws` 标签在文档中说明违反参数限制时会抛出的异常；
- 非公有的方法，一般采用断言来检查参数的有效性；
  - 断言机制如果关闭（`-da`）的话，是几乎没有性能消耗的，除非使用 `-ea | -eanableassertions` 开启断言；（`断言` 也可以通过在 `-ea` 或 `-da` 后面指定包名来使一个包的断言有效或无效）；
- 但是并不要认为对参数的任何限制都是好事
  - 在参数能够合理地完成工作的情况下，限制当然是越少越好；

39. 必要的时候进行保护性拷贝

- 下面的例子声称可以表示一段不可变的时间

```
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;
  }
  ... // Remainder omitted
}
```

貌似不可变，但实际上，`Date类`是可变的，因此并不能达到预期的效果：

```
  Date start = ...;
  Date end = ...;
  Period p = new Period(start, end);
  end.setYear(78); // Modifies internals of p!
  System.out.println(p.end()); // p的end属性被改变了
```

为达到预期功能，进行保护性拷贝是必要的：

```
public Period(Date start, Date end) {
  this.start = new Date(start.getTime()); // 拷贝后达到隔离的效果
  this.end = new Date(end.getTime());

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

改进后的代码是在检测参数有效性之前进行的拷贝，并且有效性检测是针对的拷贝后的对象；如此便有效地防御了`TOCTOU(Time-Of-Check/Time-Of-Use)`攻击；同时注意，此处没有使用`Date`的`clone`方法进行拷贝，因为`Date类`不是`final`的，不能保证clone后的对象就是想要的`date`对象，也可能是被恶意修改后的对象；

- 虽然改造了`Period（前面的代码例子）`的构造器，可以避免一些被修改的问题，但是`Period`类中提供了`start()`和`end()`方法，暴露了内部可变属性对象，一样存在被修改的

风险；
  - 此时，可以分别对每个方法进行保护性拷贝；
  - 不过，此时保护性拷贝可以使用`clone`方法，因为，`clone`是改变不到内部属性对象的；

- 任何用户提供的对象进入到`内部数据结构`中时，都有必要考虑进行保护性拷贝；
- 长度非零的数组总是可变的
  - 在返回数组到客户端之前，应该总是进行`保护性拷贝`或者为客户端提供一个不可变的`数组视图`；

40. 谨慎设计方法签名

- **谨慎地选择方法名称**
  - 尽量统一、通俗易懂；
- **不要过于追求提供便利的方法**
  - 因为方法太多的话，会使得类的学习、使用、测试、维护等变得更加艰难；
  - 除非某一项操作经常被使用到，才会考虑提供便捷方法；
- **避免过长的参数列表**
  - 一般参数个数不超过4个；
  - 参数过多时
    - 通过重载，为某些参数提供默认值；
    - 创建辅助类，来传递参数；
    - 采用`builder模式`
- **参数类型优先使用接口**
- **对于`boolean`型参数，优先考虑`Boolean`的枚举类型**
  - （具体原因可参考书本P164，鄙人感觉理解有些难度）

41. 慎用重载

- 举个栗子（见书本P165）

```
public static String classify(Set<?> s) {
  return "Set";
}
public static String classify(List<?> lst) {
```

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

  @Test
  public void test() throws IOException {
    Collection<?>[] collections = {
      new HashSet<String>(), new ArrayList<BigInteger>(), new
HashMap<String, String>().values()
    };
    for (Collection<?> c : collections) {
      System.out.println(classify(c));
      System.out.println("\t->" + c.getClass().getSimpleName());
    }
  }
  //  打印结果为
  //   Unknown Collection
  //    ->HashSet
  //   Unknown Collection
  //    ->ArrayList
  //   Unknown Collection
  //    ->Values
```

解决方法就是：用于也不要编写参数数量一样的重载方法；

42. 慎用可变参数

- 可变参数举例：
  - `method(Integer ... id)`
- 可变参数是在`1.5`版本中为`printf`而设计的；`printf`和`反射机制`从中极大地受益；
- 可变参数在每次调用的时候都有数组分配和初始化，会有性能消耗；
  - 当性能和可变参数的灵活性都想兼顾的时候，可以采用重载的方法，因为95%的情况下，参数数量不超过3个：

```
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) { }
```

43. 返回零长度的数组或集合，而不是Null

44. 为所有导出的API元素编写文档注释

- 文档主要内容
  - 应该是说明这个方法做了什么，而不是如何做的；
  - 列出 `前提条件` 和 `后置条件` ；
  - 前提条件大部分是在 `@throws` 标签中添加说明，也有是在 `@param` 中进行说明；
    - `@throws` 标签后一般跟着 `if` 单词，表示在什么条件下会抛出异常，eg：`@throws NullPointerException if the specified array is null` ；
  - 应该描述方法的 `副作用` ，比如：性能消耗、启动了后台线程等；
  - 最好能描述方法的 `线程安全性` ；
  - 所有注释一般都不用句点来结束；
  - 文档中的所有HTML元素都会出现在最终的 `doc-html` 文档中；
- 为 `枚举类` 的每一个元素添加注释；
- 不要忽略 `线程安全性` 和 `可序列化性` ；
- `javadoc` 具有继承性
  - 如果 `API` 元素没有注释， `javadoc` 会搜索（搜索算法）最适合的文档注释，接口注释优于超类；
  - 可以使用 `{@inhertDoc}` 直接标记继承超类中的部分注释内容；这有利于文档的维护；

第七章 通用程序设计

45. 局部变量的作用域和最小化

- **与C语言区别**
  - `C语言` 要求变量必须再一个代码块的开始进行声明； `Java` 却比较自由；
- 在第一次使用的地方进行声明；
- 当不能对变量进行有意义的初始化时，应该推迟变量的声明，直到初始化；
- 一个性能测试

```
// 第二种方式性能更好
public void test() throws IOException {
    StopWatch w1 = new StopWatch();
    w1.start("1");
    for (int i=0; i < 100000; i++){
```

```
            List<Integer> list = new ArrayList<>();
            list.add(i);
        }
        w1.stop();
        System.out.println(w1.prettyPrint());

        StopWatch w2 = new StopWatch();
        w2.start("2");
        List<Integer> list = new ArrayList<>();
        for (int i=0; i < 100000; i++){
            list.clear();
            list.add(i);
        }
        w2.stop();
        System.out.println(w2.prettyPrint());
    }
```

46. `for-each` 循环优于传统 `for循环`

- 实现 `Iterable` 接口的类，都可以使用 `for-each`；
- 书中列举的错误示例

```
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = EnumSet.allOf(Face.class);
for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
  for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
    System.out.println(i.next() + " " + j.next());
```

47. 了解和使用类库

- 反例

```
static Random rnd = new Random();
// 生成0-n以内的随机数
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这个方法有3个缺点：

    - 如果 `n` 是比较小的 `2` 的幂值，那么一个相当短的周期后，产生的随机数列将会重复；
    - 如果 `n` 不是 `2` 的幂值，那么随机数的概率将不会平均， `n` 越大，结果（前 `1/2` 段内的概率接近 `2/3` ）越明显；
    - 因为 `Math.abs(Integer.MIN_VALUE)` 的值为负数，如果 `rnd.nextInt()` 的值刚好为 `Integer.MIN_VALUE` ，那么 `random(n)` 结果就不再 `0-n` 范围内了；
- 要解决上面问题的方法，需要了解 `伪随机数生成器` 、 `数论` 、 `2的求补算法` 等相关知识，幸运的是 `Random.nextInt(int)` 已经帮忙实现了；
- 及时了解新版本中添加的新库方法；
- 最基本的要求，至少应该了解 `java.lang.*` 、 `java.util.*` ，甚至 `java.io.*` 相关库；

48. 如果需要精确的答案，请避免使用 `float` 和 `double`

- `float` 和 `double` 至少为工程计算提供一个快速的近似计算，并不提供完全精确的结果；
- 特别是货币计算，最不能使用 `float` 和 `double` ；

49. 基本类型优先于装箱类型

- 装箱类型进行比较的时候，最好使用 `equals` 方法，比如： `Integer.equals(object)` ；

50. 如果其他类型更加合适，请避免使用字符串

- 字符串不适合代替其他类型
    - 如：数值、布尔型
- 字符串不适合代替枚举
- 字符串不适合代替聚合类型
    - 比如：object.property1 + "#" + object.property2
    - 但是json却貌似违背了这一点；

51. 当心字符串链接的性能

- 频繁进行字符串链接的地方，可以使用 `StringBuilder` ；

52. 通过接口引用变量

- 使用接口，便于更换实现，这样修改程序就更加方便，便于维护和扩展；

53. 接口优先于反射机制

- 反射机制的`失`
  - 丧失了编译时类型检查
  - 性能损失（2-50倍）
  - 代码冗长

54. 谨慎使用本地方法

- 本地方法有时是为了提高代码执行效率而编写的，但是如今jvm越来越快，大多数情况jvm已经可以满足性能要求；
- 本地方法缺乏安全性；

55. 谨慎地进行优化

- > 不要去计较效率上的小得失，很多时候，不成熟的优化才是问题的根源

- 不要因为性能而牺牲合理的结构；
- 但是也不能忽略性能问题，要努力避免那些限制性能的设计决策；
- 好的API设计一般会带来好的性能，但为了获取好的性能而对API进行包装，就是不美的想法了；

56. 遵守普遍接受的命名惯例

- 布尔型一般采用一个形容词命名，eg：`empty`；
  - 布尔型获取方法命名一般以`is`开头，eg：`isEmpty`；
  - 类型转换命名：`toType`，eg：`toString()`；
  - 返回视图类型(`view`)一般使用`asType()`，eg：`asList()`；
  - 此外，还有许多静态方法命名，eg：`valueOf`、`of`、`getInstance`、`newInstance`；

第七章 异常

57. 只针对异常的情况才使用异常

- 异常应该永远只用于异常情况，不应该用于流程控制：

```
// Horrible abuse of exceptions. Don't ever do this!
// 滥用异常。永远也别这么做！
try {
    int i = 0;
```

```
    while(true)
        range[i++].climb();
} catch (ArrayIndexOutOfBoundsException e) {
}
```

应该使用标准模式，性能会好上些许：

```
for (Mountain m : range)
        m.climb();
```

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

- 扫盲：
  - 受检异常：就是在代码中必须捕获（`catch`）或抛出（`throws`）的异常；
  - 运行时异常：不需要捕获和抛出；
  - 错误：`Throwable` 的子类，代表编译时间和系统错误，用于指示合理的应用程序不应该试图捕获的严重问题；
- 如果期待调用者能适时地恢复，可以选择受检异常；
- `Error` 一般是表示 `JVM` 资源不足、约束失败或其他程序无法执行下去的致命错误，用户不应该再自己定义 `Error` 的子类；
- 想要定义一个既不属于 `Exception`、`RuntimeException` 或 `Error` 的 抛出结构 是可能的，因为 `JLS(Java语言规范)` 并没有规定这样的机构，但是从行为上将，它们类似受检异常；
  - 不提倡自定义这种 抛出结构，因为与普通的 受检异常 相比，没有任何益处，只会困扰API用户；

59. 避免不必要的使用受检异常

- 有时候受检异常会增加调用者的负担，编写冗长的代码，改善方法：
  - 添加 `boolean` 型，针对不听情况，有选择地抛出受检异常

    ```
    if (obj.actionPermitted(args)) {
            obj.action(args);
    } else {
            ... // Handle exceptional condition
    }
    ```

    如果设计者知道将会成功，或者不介意调用失败而导致线程终止，可以直接忽略异常

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

60. 优先使用标准异常

- 专家与菜鸟的区别：
  - 代码的可重用性；
-

异常类	描述
IllegalStateException	对象状态不合适
NullPointerException	空指针
IndexOutOfBoundsException	索引越界
ConcurrentModificationException	禁止并发修改的情况下，进行了并发修改
UnsupportedOperationException	非法操作
IllegalArgumentException	非null的参数值不正确

61. 抛出与抽象相对应的异常

- 异常转译：高层的实现应该捕获低层抛出的异常，同时抛出可以按照高层抽象进行解释的异常；
- 异常链

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

异常链使用到了构造器链，最终到达异常父类的构造器`Throwable(Throwable)`，对于没有实现`XxException(Throwable)`的异常，可以调用`Throwable#initCause(Throwable)`达到同样的效果；

- 使用日志记录下异常，可以将问题与用户隔离开来；

62. 每个方法抛出的异常都要有文档

- 始终要单独地声明 `受检异常` ，并用 `@throws` 标记，准确地记录下每个异常的条件：
  - 如果有多个异常，不要使用 `快捷方式` 声明这些异常的超类，要列出并说明所有 `受检异常` ；
- 对于 `未受检异常` ，虽然没有要求一定要向 `受检异常` 进行文档标注，但是，为它们也建立文档（但是别用 `@throws` 标注）无意是明智的；
- 如果一个类中许多方法因为 `同一个原因` 抛出 `同一个异常` ，那么在该类的注释中对这个异常进行说明是必要的；

63. 在详细消息中捕获失败原因

- 为了捕获失败，异常消息的打印应该包含所有 `对该异常有共享` 的参数和域的值：
  - 但是，冗长多余的描述也是没必要的，所有异常打印消息要精练准确；
- 为了捕获精确的失败信息，应该在异常构造器中引入这些信息，而不是提供一个 `String` 参数的构造器：

```
public IndexOutOfBoundsException(int lowerBound, int upperBound, int
index){
        super(String.format( "Lower bound: %d, Upper bound: %d,
Index: %d", lowerBound, upperBound, index));
        // Save failure information for programmatic access
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
        this.index = index;
}
```

64. 努力使失败保持原子性

- 失败原子性：一般而言，失败方法的调用应该使对象保持在被调用之前的状态；可变对象保持原子性的方法：
  - 在执行 `改变状态` 的操作之前执行 `参数检查` 和 `可能发生异常的计算` 等等；
  - 编写 `回复代码` ：主要用于 `永久性` 的（ `基于磁盘` 的（ `disk-based` ））数据结构；
  - 在操作前对 `对象` 进行拷贝，如果失败，通过备份还原状态；

65. 不要忽略异常

- 应该重视注释文档中标注的异常；
- 使用空 `catch` （ `catch(XxxException e){}` ）块忽略异常是最愚蠢的做法；

第七章 并发

66. 同步访问共享的可变数据

- 除了 `long` 和 `double` ，对于其他变量， `java` 的操作都是 `原子性` 的；
- `Java` 的内存模型规定了一个线程所做的变化 `何时` 以及 `如何` 对其他线程可见；
- 关于同步的例子

```
public class StopThread {
        private static boolean stopRequested;

    public static void main(String[] args) throws
InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true; // 期望新线程执行1s后，被关闭，但，实际上，
backgroundThread并没有被关闭；
        }
}
```

`backgroundThread` 没有如期关闭的原因是没有进行同步，线程 `backgroundThread` 不能够感知到主线程中修改的数据 `stopRequested = true` ；应该做如下修改：

```
public class StopThread {
    private static boolean stopRequested;

    // 添加同步
    private static synchronized void requestStop() {
        stopRequested = true;
    }

    // 读写都要添加同步，仅仅添加写同步也是无效
```

```
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws
InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested())
                i++;
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

也可以使用`volatile`关键字修饰`stopRequested`域，这样`synchronized`就可以省略；

- 事实上不可变：

> 让一个线程在短时间内修改一个对象，然后与其他线程共享，但是只同步共享对象引用，然后其他线程没有同步也可以读取对象，只是对象不能被它修改；（P233）

- 安全发布对象引用的方法：

延伸

- Java.util.concurrent包研究

唠唠其他，开小差

- 永远不要让客户去做任何类库能够替客户完成的事

注意

- 文中的页码均表示中文版《Java高效开发（第二版）》