# 第 1 章 引言

# 第 2 章 创建和销毁对象

## 1 考虑用静态工厂方法代替构造器

优势：

1. 有名称，用方法名标识获取对象的名称
2. 不必每次调用都创建新的对象
3. 可以返回原返回类型的任何子类型的对象
4. 在创建参数化类型实力的时候，它们使代码变得更加简洁

缺点：

1. 类如果不含有公有的或者受保护的构造器，就不能被子类化
2. 静态工厂方法与其它静态方法没有任何区别

常用名称：

valueOf

of

getInstance

newInstance

getType

newType

## 2 遇到多个构造器参数时考虑使用构建器

当类的属性较多时，用一般的构造方法，重载情况多。用对象的构建器来创建对象。

如：

Component c=new Component.Builder(a,b).field1(c).field2(d).build();

其中 Builder为静态内部类，field方法都返回Builder(return this)，build方法返回Component对象。

缺点：创建对象需要先创建构造器，因此它只有在有很多参数的时候才使用。

## 3 用私有构造器或者枚举类型强化Singleton属性

用私有构造方法或枚举型来强化单例模式

单例模式：

枚举型：

public enum Elvis{

INSTANCE;

public void leaveTheBuilding(){...}

}

## 4 通过私有构造器强化不可实例化的能力

对于一些只提供静态方法的工具类，我们不希望它被实例化，则可以用私有构造器来避免。如：

public class ToolUtils{

private ToolUtils(){

throw new Exception(“本类内部实例化抛异常”);

}

// other static method.

}

## 5 避免创建不必要的对象

对于在类的方法中每次调用都需要创建相同的对象，那么就可以使用静态属性及静态代码块来初始化类时创建一次,例如A类中有个方法a要比较传入的日期是否在两个固定的日期之间，一种方法是在a中构建两个日期，再比较。这样每次比较都会创建两个日期对象。简化的方法就是在类中定义两个静态成员变量。

public class Person{

private static final Date START;

private static final Date END;

//other fields

static {

//initial code can move to method isInDateArea() and initialize for the first calling.

SimpleDateFormat sdf=new SimpleDateFormat(“yyyy-MM-dd HH:mm:ss”);

START=sdf.parse(“1946-01-01 00:00:01”);

END=sdf.parse(“1965-01-01 00:00:00”);

}

public boolean isInDateArea(Date birthday){

return birthday.compareTo(START) >=0 &&birthday.compareTo(END)<0;

}

}

基本类型优先于包装类型使用。当重用对象的代价大于创建对象时不建议重用对象，例如自己的不必要的对象池，维护对象池的开销大于创建对象。除非像数据库连接池那样，创建连接对象的代价非常大，则可以用连接池来复用连接对象。

## 6 消除过期的对象引用

异常内存泄漏的原因：

1. 自定义栈中过期元素没有清除。——不用的元素要设为null
2. 过期不用的缓存。
3. 无需长久存在的监听器和回调没有显示取消。

## 7 避免使用终结方法

终结方法 super.finalize()：不一定及时执行，也不一定会执行。可用于终止非关键的本地资源。如果用终结方法作为安全网，要记得记录终结方法的非法用法。

# 第 3 章 对于所有对象都通用的方法

## 8 覆盖equals方法时遵守通用约定

约定：

1. 自反性：对于任何非null的引用值x, x.equals(x)必须返回true。
2. 对称性：对于任何非null的引用值x,y。当y.equals(x)=true时，x.equals(y)=true。
3. 传递性：非null引用值x,y,z 当x.equals(y)=true,y.equals(z)=true,则x.equals(z)=true。
4. 一致性：非null引用值x,y，不管怎样操作，只要x,y信息没修改，则多次调用x.equals(y)会一致返回true或false。
5. 非空性：所有引用不能为null。

告诫：

1. 覆盖equals时总要覆盖hashCode。
2. 不要企图让equals方法过于智能。
3. 不要将equals声明中的Object对象替换为其它类型方法参数类型要为Object。
4. 在每次覆盖equals方法都逐一审查equals 的五个约定

## 9 覆盖equals时总要覆盖hashCode

覆盖equals时如果不覆盖hashCode，会导致该类无法结合所有基于散列的集合一起正常运作，这样的集合包括HashMap,HashSet和HashTable。

覆盖hashCode约定：

1. 在应用程序执行期间，只要对象equals方法比较操作所用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法都必须始终如一地返回同一个整数。同一个应用程序多次执行过程中，每次执行所返回的整数可以不一致。
2. 如果两个对象equals比较是相等的，那么这两个对象任意一个的hashCode方法都必须产生同样的整数结果。
3. 如果两个对象equals比较是不相等的，那么这两个对象任意一个的hashCode方法都必须产生不同的整数结果。这样有可能提高散列表的性能。

利用equals比较的每个域来生成hashCode：

a 计算域f的散列值c

1. 如果该域为boolean类型，则计算(f?1:0)。
2. 如果该域为byte ,char,shot或者int类型，则计算(int)f。
3. 如果该域为long类型，则计算(int)(f^(f>>>32))。
4. 如果该域为float类型，则计算Float.floatToIntBits(f)。
5. 如果该域为double类型，则计算Double.doubleToLongBits(f)，然后按3为 得到的long类型值计算散列值。
6. 如果该域为一个对象引用，且该类的equals方法通过递归的调用equals方 法来比较这个域，则同样为这个域递归调用hashCode。
7. 如果该域为一个数组，则把每个元素当作一个单独的域来处理，每个域计 算一个散列码，然后根据b中的方法把这些散列值组合起来。

b 按下面的公式把a每个域得到的散列码c合并到result中

result=31\*reuslt+c

c 返回result。

## 10 始终要覆盖toString

注意：避免依赖于toString数据格式的细节进行编程。

## 11 谨慎地覆盖clone

## 12 考虑实现Comparable接口

对象会参与顺序排序的功能时，类可以考虑实现comparable接口，覆盖compareTo方法。参数类型与类一致。

@override

int compareTo(T t){

//比较整数型基本类型的域，可以使用关系操作符 < 和 >

//浮点型数据比较 用 Double.compare 或者 Float.compare

//如果有多个域参与比较，分清域的比较优先级，依次比较，如一个带有两个域(a,b,a比较优先级大于b)的类对象作比较时：

if(a<t.a)return -1;if(a>t.a)return 1;

if(b<t.b)return -1;if(b>b.b)return 1;

return 0;

}

# 第 4 章 类和接口

## 13 使类和成员的可访问性最小化

对于顶层的（非嵌套的）类和接口，只有两种可能的访问级别：private，public

如果用public修饰，那它就是公有的。如果是private，那它就是包级私有的，它实际上成了包的一部分，而不是导出API的一部分，在以后的发行版本中，可以对它进行修改，替换或删除，而无需担心会影响到现有的客户端程序。如果做成公有的，那你就有责任永远支持它，以保证它们的兼容性。

如果一个包级私有的顶层类或接口只是在某一个类内部被用到，就应该考虑使它成为那个类的私有嵌套类，将可访问性由某个包缩小到到某个类。

类具有公有的静态final（不希望被修改）数据域或对象域，或者返回这种域（不希望调用后的使用过程中不会改变内部静态final属性的值）的访问方法，这几乎总是错误的，客户端能够修改数组或对象中的内容。可以使用以下两种方法：

1.数组变私有，增加公有的不可变列表

private static final Thing[] PRIVATE\_VALUES={...}

public static final List<Thing> VALUES=

Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES));//add操作报错

此法对获得的VALUES不可作add,set操作，只能操作get后得到的对象。

1. 数组变私有，增加公有方法返回数组的备份

private static final Thing[] PRIVATE\_VALUES={...}

public static final Thing[] values(){return PRIVATE\_VALUES.clone();}

此法获得的values，长度不可变，但values[i]可以重新赋值。

除了公有静态final域的特殊情形之外，公有类都不应该包含公有域。并且要确保公有静态final域所引用的对象都不是可变的。

## 14 在公有类中使用访问方法而非公有域

定义私有域，提供公开getter/setter方法.

优点：保留将来改变该类的内部表示法的灵活性；可以对数据采取辅助行动或强加约束条件（校验get/set数据是否合法等）。

## 15 使可变性最小化

使用理由：不可变的类比可变类更加易于设计、实现和使用。它们不容易出错，且更加安全。

使用不可变类，需遵循以下5条规则：

1. 不要提供任何会修改对象状态的方法。
2. 保证类不会被扩展。一般定义成final的，后面还有其它方法。
3. 使所有的域都是final的。
4. 使所有的域都成为私有的。
5. 确保对于任何可变组件的互斥访问。如果类具有指向可变对象的域，则必须确保该类的客户端无法获得指向这些对象的引用。

优点：

不可变对象本质上是线程安全的，它们不要求同步；

不仅可以共享不可变对象，甚至也可以共享它们的内部信息；

不可变对象为其它对象提供了大量的构件。

缺点：对于每个不同的值都需要一个单独的对象。

除了定义final类之外，另一种不可变类的实现方法是让类的所有构造器都变成私有的或者包级私有的，并添加公有的静态工厂来代替共有的构造器。

public class Complex{

private final double re,im;

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

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

}

## 16 复合优先于继承

继承的功能非常强大，但是也存在诸多问题，因为它违背了封装原则。只有当子类和超类之间确实存在子类型关系时，使用继承才是恰当的。如果子类和超类处在不同的包中，并且超类不是为了继承而设计的，那么继承将会导致脆弱性。为了避免这种脆弱性，可以用复合和转发机制来代替继承。

## 17 要么为继承而设计并提供文档说明，要么就禁止继承

超类中的方法a调用了b方法，那么子类覆盖b方法时要注意了，覆盖b方法是否会影响a方法的功能。所以超类需要提供说明文档。

当超类的方法实现发生改变时，需要编写子类对类进行测试，以确保对子类不会产生影响。

## 18 接口优先于抽象类

现有的类可以很容易被更新，以实现新的接口。

接口是定义mixin(混合类型)的理想选择。

接口允许我们构造非层次结构的类型框架。

骨架实现类：简单实现接口的所有方法。可以直接使用，也可以被继承后用户覆盖自己想要用到的方法。

接口通常是定义允许多个实现的类型的最佳途径。这条规则有个例外，即当演变的容易性比灵活性和功能更为重要的时候。在这种情况下，应该使用抽象类来定义类型，但前提是必须理解并且可以接受这些局限性。如果你导出了一个重要的接口，就应该坚决考虑同时提供骨架实现类。最后，应该尽可能谨慎的设计所有的公有接口，并通过编写多个实现来对它们进行全面的测试。

## 19 接口只用于定义类型

当类实现接口时，接口就充当可以引用这个类的实例的类型。

常量接口模式（只有static final修饰的常量）是对接口的不良使用。

## 20 类层次优于标签类

如(反例)

class Figure{

enum Shape{RECTANGLE, CIRCLE};

final Shape shape;

Figure(double radius){shape=CIRCLE;this.radius=radius;}

Figure(double length,double width){...}

double area(){

switch(shape){

case RECTANGLE:...

case CIRCLE: ...

}

}

}

更好的方法是用继承法

abstract class Figure{ abstract double area();}

calss Circle extends Figure{...}

class Rectangle extends Figure{...}

## 21 用函数对象表示策略

函数指针的主要用途就是实现策略模式。为了在JAVA中实现这种模式，①要声明一个接口来表示该策略，并且为每个具体策略声明一个实现了该接口的类。②当一个具体策略只被使用一次时，通常使用匿名类来声明和实例化这个具体策略类。③当一个策略是设计用来重复使用的时候，他的类通常就要被实现为私有的静态成员类，并通过共有的静态final域被导出，其类型为该策略接口。

① class StringLengthComparator implements Comparator<String>{...}

② Arrays.sort(stringArray,new Comparator<String>(){

public int compare(String s1,String s2){return s1.length()-s2.length();}

});

③ class Host{

private static class StrLenCmp implements Comparator<String>,Serializable{

public int compare(String s1,String s2){return s1.length()-s2.length();}

}

public static final Comparator<String>

STRING\_LENGTH\_COMPARATOR=new StrLenCmp();

}

## 22 优先考虑静态成员类

嵌套类是指被定义在另一个类的内部的类。嵌套类有四种：

①静态成员类 ②非静态成员类 ③匿名类 ④局部类

其中②③④被称为内部类。

如果声明成员类不要求访问外围实例，就要始终把static修饰符放在它的声明中，使它成为静态成员类，否则它的实例就会包含一个额外的指向外围对象的引用。保存这份引用要消耗时间和空间，并且会导致外围实例在符合垃圾回收时却仍然得以保留。并且使用时不需要有外围类的实例，直接静态引用就可以使用该类的实例。

匿名类除了在声明时之外，是无法实例化的。且不能执行instanceof，不能实现多个接口或者扩展一个类的同时实现接口，不能调用任何成员。它们必须简短，大约10行内，否则可读性差。常见用法：如21条创建函数对象，或者创建过程对象如Runnable,Thread,TimerTask实例，或者用在静态工厂方法内部(18条)。

简而言之，共有四种不同的嵌套类，每一种都有自己的用途。如果一个嵌套类需要在单个方法之外仍然是可见的，或者它太长了，不适合于放在方法内部，就应该使用成员类。如果成员类的每个实例都需要一个指向其外围实例的引用，就要把成员类做出非静态的；否则，就做成静态的。假设这个嵌套类属于一个方法的内部，如果你只需要在一个地方创建实例，并且已经有了一个预置的类型可以说明这个类的特征，就要把它做成匿名类；否则，就做成局部类。

# 第 5 章 泛型

## 23 请不要在新代码中使用原生态类型

声明中具有一个或者多个类型参数的类或者接口，就是泛型类或者接口。统称为泛型。

List<E> （E的列表）是泛型，List是原生态类型。

使用泛型，使类型规范化，使用时不必拆箱。

List<String> 是List的子类，但不是List<Object>的子类。所以List<String> 可以作为方法m(List list)的参数，而不能作为m(List<Object> list)的参数。

原生态类型不安全，可以使用List<Object> , List<?>等。

if(o instantceof Set){ Set<?> m = Set<?> o;}

不要再新代码中使用原生态类型，但有例外：在类文字中必须使用原生态类型。如List.class ,Stirng[].class , int.class 都合法,但是List<String.class>, List<?>.class都不合法。

总之，使用原生态类型会在运行时导致异常，因此不要在新代码中使用。原生态类型只是为了与引入反省之前的遗留代码进行兼容和互用而提供的。回顾：Set<Object>是个参数化类型，表示可以包含任何对象类型的一个集合；Set<?>则是一个通配符类型，表示只能包含某种未知对象类型的一个集合；Set则是一个原生态类型，它脱离了泛型系统。前两种是安全的，最后一种不安全。

|  |  |  |
| --- | --- | --- |
| 术语 | 实例 | 所在条例 |
| 参数化的类型 | List<String> | 23 |
| 实际类型参数 | Stirng | 23 |
| 泛型 | List<E> | 23,26 |
| 形式类型参数 | E | 23 |
| 无限制通配符类型 | List<?> | 23 |
| 原生态类型 | List | 23 |
| 有限制类型参数 | <E extends Number> | 26 |
| 递归类型限制 | <T extends Comparable<T>> | 27 |
| 有限制通配符类型 | List<? extends Number> | 28 |
| 泛型方法 | static <E> List<E> asList(E[] a) | 27 |
| 类型令牌 | String.class | 29 |

## 24 消除非受检警告

@SuppressWarnings注解可以用在任何粒度的级别中，变量、方法或类上。应该在尽可能小的范围内使用，以免掩盖了重要的警告。同时，你必须先消除一些经过，如泛型安全警告，然后证实代码警告是安全的，则可以使用此注解，并注释说明能忽略此警告的原因。

**25 列表优先于数组**

Object[] o=new Long[1];o[0]=”string”;//编译不报错，运行报错

List<Object> o1=new ArrayList<Long>();//编译报错

利用数组，运行时才会报错，而使用列表，编译时就可以发现错误。数组与泛型之间的第二大区别在于，数组是具体化的，因此只有在运行时才知道并检查它们的元素类型约束。而泛型则是通过擦除来实现的，因此泛型只在编译时强化它们的类型信息，并在运行时擦除它们的元素类型信息。擦除就是使反省可以与没有使用泛型的代码随意进行互用。

## 26 优先考虑泛型类

将集合声明参数化，或者使用JDK所提供的泛型和泛型方法，这些都不困难。但怎样编写自己的泛型？

将第6条的Stack存储数组进行改造成使用泛型：

public class Stack<E>{

private Object[] elements;

private int size=0;

private static final int DEFALUT\_INITIAL\_CAPACITY=16;

public Static(){

elements=new Object[DEFALUT\_INITIAL\_CAPACITY];

}

public void push(E e){

ensureCapacity();

elements[size++] =e;

}

@SuppressWarnings(“uncheked”)

public E pop(){

if(size==0) throw new EmptyStatckException();

Object result=elements[--size];

elements[size]=null;

return (E) result;

}

}

使用泛型比使用需要在客户端代码中进行转换的类型来得更加安全，也更加容易。在设计新类型的时候，要确保它们不需要这种转换就可以使用。这通常意味着要把类做成是泛型的。只要时间允许，就把现有的类型都泛型化。这对于这些类型的新用户来说会变得更加轻松，又不会破坏现有的客户端。

## 27 优先考虑泛型方法

静态工具方法尤其适合于泛型化。编写泛型方法与编写泛型类相类似。

泛型方法的类型限制必须放在返回类型前面

public static <T extends Comparable<T>> T max(List<T> list){...}

public static <T> T max1(List list){ ...}

泛型方法就像泛型一样，使用起来比要求客户端转换输入参数并返回值的方法来得更加安全，也更加容易。就像类型一样，你应该确保新方法可以不用转换就能使用，这通常意味着要将它们泛型化。并且就像类型一样，还应该将现有的方法都泛型化，使新用户使用起来更加轻松，又不会破坏现有的客户端。

**28 利用有限制通配符来提升API的灵活性**

借助地26条的例子，添加方法：

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

但是使用时

Stack<Number> numberStack=new Stack<>();  
 Iterable<Integer> integers=new HashSet<>();  
 numberStack.pushAll(integers);

警告：pushAll要求使用Iterator<Number> 参数，不能用Iterator<Integer>类型参数。原因是Iterator<Integer> 不是Iterator<Number>的子类

解决问题：

使用有限制通配符来定义方法

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

要把元素放入，要确保元素是原本类型或子类。

同理加入我们要把所有元素取出来放到指定的集合中，则需要原本类型或其父类的泛型集合来接收。

public void popAll(Collection<? super E> dst){while (size>0){dst.add(pop()); }}

助记：PECS -- producer-extends,consumer-super ，所有的comparable和comparator都是消费者（用 <? super E>）。

第27条有个方法public static <T extends Comparable<T>> T max(List<T> list){...}

使用通配符更好public static <T extends Comparable<? super T> T max(List<T> list){...}

如果类型参数只在方法声明中出现一次，就可以用通配符取代它：如果是无限制的类型参数，就用无限制的通配符取代它；如果是有限制的类型参数，就用有限制的通配符取代它。

public static void swap(List<?> list,int i,int j){。。。}  
 //public static <E> void swap(List<E> list,int i,int j){。。。 }

## 29 优先考虑类型安全的异构容器(((φ(◎ロ◎;)φ)))

。。。总而言之，集合API说明了泛型的一般用法，限制你每个容器只能有固定数目的类型参数。你可以通过将类型参数放在键上而不是容器上来避开这一限制。对于这种类型安全的异构容器，可以用Class对象作为键。以这种方式使用的Class对象称作类型令牌。你也可以使用定制的键类型。例如，用一个DatabaseRow类型表示一个数据库行（容器），用泛型Column<T>作为它的键。

# 枚举和注解

## 30 用enum代替int常量

枚举类型：是指由一组固定的常量组成合法值的类型。java的枚举本质上是int值。

public enum Apple {FUJI,PIPPIN}

public enum Orange {NAVEL,TEMPLE,BLOOD}

对于不同枚举类型分别要覆盖枚举类型中的抽象方法，被称作特定于常量的方法实现。

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;} };

abstract double apply(double x,double y);

}

把每个类型与特定常量结合起来，并覆盖toString方法

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;//枚举类型不可变，用final修饰，一旦赋值，不可改变。

Operation(String symbol){this.symbol=symbol;}

@Override public String toString(){return symbol;}

public static Operation fromString(String symbol){  
 for(Operation o:values()){ if (o.toString().equals(symbol)){return o;}}  
 throw new ClassCastException("未找到符号对应的枚举类型："+symbol);  
 }

abstract double apply(double x,double y);

}

什么时候使用枚举呢？每当需要一组固定常量（如菜单选项，操作代码，命令标记等）的时候，包括天然的枚举类型（如行星，四季，星期几等）。

## 31 用实例域代替序数

有时候，我们需要获取枚举类型在该枚举类中的序数，即第几个：

public enum Ensemble {  
 SOLO,DUET,TRIO,QUARTET;  
 public int numberOfMusicians(){return ordinal()+1;}

}

使用：Ensumble.TRIO.numberOfMusicians() ----> 3

永远不要根据枚举的序数导出与它关联的值，而是要将它保存在一个实例域中，改进：

public enum Ensemble {  
 SOLO(1),DUET(2),TRIO(3),QUARTET(4);  
 private final int numberOfMusicians;  
 Ensemble(int size){this.numberOfMusicians=size;}  
 public int numberOfMusicians(){return numberOfMusicians;}

}

Enum规范中谈到ordinal时写道：“大多数程序员都不需要这个方法。它是设计成用于像EnumSet和EnumMap这种基于枚举的通用数据结构的。”除非你在编写的是这种数据结构，否则最好完全避免使用ordinal方法。

## 32 用EnumSet代替位域

public class Text {  
 public static final int STYLE\_BOLD =1<<0;  
 public static final int STYLE\_ITALIC =1<<1;  
 public static final int STYLE\_UNDERLINE =1<<2;  
 public static final int STYLE\_STRIKETHROUGH =1<<3;  
 public void applyStyles(int styles){。。。}

}

这种表示法让你用OR位运算将几个常量合并到一个集合中，称作位域（bit filed）：

text.applyStryles(STYLE\_BOLD | STYLE\_ITALIC);

用枚举代替位域，它更加简短、更加安全、更加清楚：

public class Text{

public enum Style{ BOLD, ITALIC ,UNDERLINE,STRIKETHROUGH }

public void applyStyles(Set<Style> styles) { ...}

}

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

## 33 用EnumMap代替序数索引

public class Herb{

pbulic enum Type{ANNUAL,PERENNIAL,BIENNIAL}

private final String name;

private final Type type;

Herb(String name,Type type){this.name=name;this.type=type;}

}

现在我们要把Herb[] herbs 的对象按type不同来分别放入不同的容器。

一种方法是Set<Herb>[] herbByTpye=(Set<Herb>[])new Set[Herb.type.values().length];

数组赋初值。。。

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

即按照Type的序数来将herb元素放到与herbByType对应的下标元素中。

用EnumMap代替：

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>());//初始化

for(Herb h:herbs)herbsByType.get(h.type).add(h);

总而言之，最好不要用序数来索引数组，而要使用EnumMap。如果你所表示的这种关系是多维的，就使用EnumMap<...,EnumMap<...>>。应用程序的程序员在一般情况下都不使用Enum.ordinal，即使要用也很少，因此这是一种特殊情况（见31条）。

## 34 用接口模拟可伸缩的枚举

扩展第30条的Operation类型

public interface Operation{

double apply(double x,double y);

}

public enum BasicOperation implements 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;//枚举类型不可变，用final修饰，一旦赋值，不可改变。

BasicOperation(String symbol){this.symbol=symbol;}

@Override public String toString(){return symbol;}

}

扩展

public enum ExtendedOperation implements Operation{

EXP(“^”) { double apply(double x,double y){return Math.pow(x,y);} },

PREMAINDER(“%”) { double apply(double x,double y){return x%y;} };//取余

private final String symbol;//枚举类型不可变，用final修饰，一旦赋值，不可改变。

ExtendedOperation (String symbol){this.symbol=symbol;}

@Override public String toString(){return symbol;}

}

使用 test(ExtendedOperation.class,2,3);

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.applay(x,y));}

或者

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.applay(x,y));}

第二种允许调用者将多个实现类型的操作合并到一起。抛弃了第32，33条建议，因此，除非需要灵活地合并多个实现类型的操作，否则可能最后使用第一种有限制的类型令牌。

不足：无法将实现从一个枚举类型继承到另一个枚举类型。

总而言之，虽然无法编写可扩展的枚举类型，却可以通过编写接口以及实现该接口的基础枚举类型，对它进行模拟。这样允许客户端编写自己的枚举来实现接口。如果API是根据接口编写的，呢吗在可以使用基础枚举类型的任何地方，也都可以使用这些枚举。

## 35 注解优先于命名模式

根据参数名称特征或者方法名称特征来告诉系统需要执行哪些功能如testUpdate名称的方法表明此方法是一个测试方法等，这就是命名模式。缺点很明显，tsetUpdate将不会执行，让人误认为执行成功等。

## 36 坚持使用Override注解

在子类或者实现类中，使用@Override来覆盖方法声明，编译器可以替你防止大量的错误。

## 37 用标记接口定义类型

标记接口是没有包含方法声明的接口，而只是指明一个类实现了具有某种属性的接口。

标记接口比标记注解有两个优点：

1. 标记注解在运行时才能捕捉错误，而标记接口在编译时就能捕捉错误。
2. 标记接口更精确，只有扩展该接口才有效，而标记注解可以应用到任何类或接口。

标记注解的优点：

1. 它可以通过默认的方式添加一个或者多个注解类型元素，给已被使用的注解类型添加更多的信息。随着时间的推移，简单的标记注解可以演变成更加丰富的注解类型。而标记接口则不能。
2. 标记注解是更大的注解机制的一部分，在那些支持注解作为编程元素之一的框架中同样具有一致性。

总而言之，标记接口和标记注解都各有用处，如果想要定义一个任何新方法都不会与之相关联的类型，标记接口就是最好的选择。如果想要标记程序元素而非类和接口，考虑未来可能要给标记添加更多的信息，或者标记要适合于已经广泛使用了注解类型的框架，那么标记注解就是正确的选择。

# 第 7 章 方法

## 38 检查参数的有效性

每当编写方法或者构造器的时候，应该考虑它的参数有哪些限制。应该把这些限制写到文档中，并在这个方法体的开头处，通过显式的检查来实施这些限制。养成这样的习惯非常重要，只要有效性检查有一次失败，你为必要的有效性检查所付出的努力便没有白费。

## 39 必要时进行保护性拷贝

如下类，用于表示一段不可变的时间周期

public final class Period{

private final Date start;

private final Date end;

/\*\*

\*@param start the beginning of the period

\*@param end..

\*@throws NullPointerException if start or end is null

\*@throws IllegalArgumentException ...

\*/

public Period(Date start,Date end){

if(start.compareTo(end)>0) throw new IllegalArgumentException(start+”after”+end);

this.start=start;this.end=end;

}

...

}

乍一看，这个类是不可变的，但实际上start,end对象在地址不可变得情况下，值是可变的。为了保护Period实例的内部信息避免受到这种攻击，对于构造器的每个可变参数进行保护性拷贝是必要的，而且使用备份对象作为Period实例的组件，而不使用原始的对象。

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) //copy之后检查参数有效性

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

}

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

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

对于参数类型可以被不可信任方子类化的参数，请不要使用clone方法进行保护性拷贝。有经验的程序员通常使用Date.getTime()返回的long基本类型作为内部的时间表示法，而不是用Date对象引用。因为date是可变的。

对于确实不可变得类，一定要仔细检查以确保不可以被外部恶意修改。

## 40 谨慎设计方法签名

谨慎地选择方法名称，不要过于追求提供便利的方法，避免过长的参数列表。

有三种方法可以缩短过长的参数列表：

1. 把方法分解成多个方法
2. 创建辅助类，用来保存参数的分组，一般为静态成员类
3. 结合1和2，从对象构建到方法调用都采用Builder模式（第2条），连续setter值，执行方法时再检验参数。

对于参数类型，要优先使用接口而不是类（见52条）。只要有适当的接口可用来定义参数，就优先使用这个接口，而不是使用实现该接口的类。

对于boolean参数，要优先使用两个元素的枚举类型。例如：

public class Thermometer{

public enum TemperatureScale{ FAHRENHEIT, CELSIUS}

private static final TemperatureScale scale;

Thermometer newInstance(TemperatureScale scale){this.scale=scale;}

}

用Thermometer.newInstance(TemperatureScale.CELSIUS)不仅比Thermometer

.newInstance(true)更有用，而且你还可以在未来的版本中将KELVIN添加到TemPeratureScale中，无需非得给Thermometer添加新的静态工厂。

## 41 慎用重载

覆盖方法的调用在运行时会基于对象的实际类型来调用实际类型的方法。而重载方法是在编译时就选定了该调用哪个重载方法，假设你的引用类型是一个超类类型，而对象是一个子类对象，那么就会选定重载方法参数为超类类型的方法，执行时也是这个方法；如果你想调用参数类型为子类类型的重载方法，那么调用方法时的参数引用类型就要为子类类型而不能用父类类型。

对于重载方法中参数数目相同又存在父子关系的重载方法改进：用一个方法代替所有重载方法，在方法里面做一个显示的 instanceof测试，做分支处理。

参数数目不同的重载方法，建议更换不同方法名称。

而对于构造方法，成员变量越多，参数组合越多，重载方法就越多，而名称又硬性规定必须与类名一致，可以选择导出静态工厂方法，采用Builder模式（第2条），连续setter选择性赋值，

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

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

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);}

预期两个集合剩下元素set为[-3,-2,-1]，list为[-3,-2,-1]，但实际上set为[-3,-2,-1]，list为[-2,0,2]。

原因：set.remove(i)选择调用重载方法set.remove(E)来按元素匹配来删除元素,这里E为Integer,会将i自动封装成Integer。list.remove(i)选择调用重载方法remove(int i),按元素下标来删除元素。改进：

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

java类库违反此条目的重载方法，String类的静态工厂方法valueOf(char[])和valueOf(Object);它们用着相同的名称，做的事情确完全不同，容易造成混淆。

简而言之，能够重载不一定要重载，一般地，对于多个具有相同参数数目的方法来说，应该尽量避免重载方法。

## 42 慎用可变参数

在重视性能的情况下，使用可变参数机制要特别小心。可变参数方法的每次调用都会导致进行一次数组分配和初始化。如果凭经验确定无法承受这一成本，但又需要可变参数的灵活性，还有一种模式可以实现，假设确定对某个方法95%的调用会用到3个或者更少的参数，就声明该方法的5个重载，每个重载方法带有0至3个普通参数，当参数的数目超过3个事，就使用一个可变参数方法。

总之，在定义参数数目不定的方法时，可变参数方法是一种很方便的方式，但是它们不应该被过度滥用。如果使用不当，会产生混乱的结果。

## 43 返回零长度的数组或集合，而不是null

返回0长度，在后续使用时，不用对null做判断处理。

return Collections.emptyList();//返回0长度的final不可变集合，且可以被共享，不会每次都创建新的对象。

## 44 为所有导出的API元素编写文档注释

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。如果没有文档注释，java doc能做的也就是重新生成该声明，作为影响的API元素的唯一文档。

方法的文档注释应该简洁地描述出它和客户端之间的约定。这个预定应该说明这个方法做了什么，而不是说明它是如何完成这项工作的。文档注释应该列举出这个方法的所有前提条件和后置条件，所谓的前提条件是指为了使客户能够调用这个方法而必须满足的条件；所谓后置条件是指在调用成功完成之后，哪些条件必须要满足。一般情况下，前提条件是由@throws标签针对未受检的异常所隐含的描述的；每个未受检的异常都对应一个前提违例。同样地，也可以在一些受影响的参数的@param标记中指定前提条件。

除了前提条件和后置条件之外，每个方法还应该在文档中描述它的副作用。所谓副作用是指系统状态中可以观察到的变化，它不是为了获得后置条件而明确要求的变化。

/\*\*  
 \* 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 emelent position</p>  
 \* @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);

为了讲过个代码示例包含在一个文档主时钟，要使用包在HTML的<pre>标签里面的Javadoc{@code}标签。换句话说，是先在多行的代码示例前使用字符<pre>{@code,然后再代码后面加上}</pre>。

不要忘记，为了产生包含HTML元字符的文档，比如小于号<,大于号>以及&，必须采取特殊的动作。让这些字符出现在文档中年的最佳办法是用{@literal}标签将它们保温起来，这样就限制了HTML标记和嵌套的Javadoc标签的处理。

\* The triangle inequality is {@literal |x+y| <|x|+|y|}.

产生文档 The triangle inequality is |x+y| <|x|+|y|.

总之，要为API编写文档，文档注释是最好、最有效的途径。对于所有可导出了API元素来说，使用文档注释应该被看作是强制性的。要采用一致的风格来遵循标准的约定。记住，在文档注释内部出现任何HTML标签都是允许的，但是HTML元字符必须要经过转义。

# 第 8 章 通用程序设计

## 45 将局部变量的作用域最小化

与第13条类似，将局部变量的作用域最小化，可以增强代码的可读性和可维护性，并降低出错的可能性。

要使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；几乎每个局部变量的声明都应该包含一个初始化表达式(try-catch除外)。

for循环比while循环更不容易出错，因为for循环参数有效范围尽在循环内有效，而while循环参数定义在while循环体外，作用范围大，容易被用错，尤其复制-粘贴时忘了更改参数名。for循环比while循环更简短，从而增强了可读性。

另一种方法是使方法小而集中，这样局部变量的作用域也就变小了。

## 46 for-each循环优先于传统的for循环

常见三种无法使用for-each循环的情况：

1. 过滤——如果需要遍历集合并删除选定的元素，就需要使用显式的迭代器，以便可以调用它的remove方法。
2. 转换——如果需要遍历列表或者数组，并取代它部分或者全部的元素值，就需要列表迭代器或者数组索引，以便设定元素的值。
3. 平行迭代——如果需要并行地遍历多个集合，就需要显式地控制迭代器或者索引变量，以便所有迭代器或者索引变量都可以得到同步前移。

以上三种情况，就要使用普通的for循环。

## 47 了解和使用类库

如：Random.nextInt(int n);

通过使用标准类库：可以充分利用这些编写标准类库的专家的知识，以及在你之前的其他人的使用经验；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，而是把时间花在应用程序上；它们的性能往往会随着时间的推移而不断提高。

## 48 如果需要精确的答案，请避免使用float和double

对于任何需要精确答案的计算任务，请不要使用float或者double，如果你想让系统来记录十进制小数点，并且不介意因为不适用基本类型而带来的不便，就请使用BigDecimal。使用BigDecimal还有一些额外的好处，它允许你完全控制舍入，每当一个操作涉及舍入的时候，它就允许你从8种舍入模式中选择其一。如果你正通过法定要求的舍入行为进行业务计算，使用BigDecimal是非常方便的。如果性能非常关键，并且你又不介意自己记录十进制小数点，而且所涉及的数值又不太大，就可以使用int或者long。如果数值范围没有超过9位十进制数字，就可以使用int；如果不超过18位数字，就可以使用long。如果数值可能超过18位数字，就必须使用BigDecimal。

## 49 基本类型优先于装箱基本类型

Long sum=0;

for(long i=0;i<Integer.MAX\_VALUE;i++){sum+=1;}

此处把sum定义成Long,性能明显会下降，

当可以选择的时候，基本类型要优先于装箱类型。两个装箱类型用==作比较，永远返回false。在以下情况下必须用装箱类型：作为集合的元素、键或值。在参数化类型中也必须用装箱类型，另外在进行反射的方法调用时，也必须使用装箱类型。

## 50 如果其它类型更适合，则尽量避免使用字符串

1. 字符串不适合代替其他的值类型
2. 字符串不适合代替枚举类型
3. 字符串不适合代替聚集类型 如 String mixKey=className+”#”+i.next();
4. 字符串也不适合代替能力表

## 51 当心字符串连接的性能

原则：不要使用字符串连接操作符（+）来合并多个字符串，除非性能无关紧要。相反，应该使用StringBuilder的apppend方法。另一种方法是，使用字符数组，或者每次只处理一个字符串，而不是将它们组合起来。

## 52 通过接口引用对象

第40条建议：应该使用接口而不是类作为参数的类型。更一般的讲，应该优先使用接口而不是类来引用对象。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，就都应该使用接口类型进行声明。只有当你利用构造器创建某个对象的时候，才真正需要引用这个对象的类。

如果你养成了用接口作为类型的习惯，你的程序将会更加灵活。当你决定更换实现时，所做的就只是改变构造器中类的名称（或者使用一个不同的静态工厂）。需要注意的是，如果被替换的实现中拥有接口定义之外的功能，而此功能又在程序中被用到，那么你更换的实现也要实现此功能。

如果没有合适的接口存在，完全可以用类而不是接口来引用对象。如值类，比如String,BigInteger等，它们通常是final的，并且很少有对应的接口。另一种情形是，对象属于一个框架，而框架的基本类型是类不是接口。如果对象属于这种基于类的框架，就应该用相关的基类(往往是抽象类)来引用这个对象，而不是它的实现类。还有一种情形，类实现了接口，也扩展了额外方法，而程序中依赖于这些额外方法，这种类就应该只被用来引用它的实例，它很少应该被用作参数类型。

## 53 接口优先于反射机制

使用反射需要付出代价：

1. 丧失了编译时类型检查的好处，包括异常检查。如果程序企图用反射方式调用不存在或者不可访问的方法，在运行时它将会失败，除非采取了特别的预防措施。
2. 执行反射访问所需要的代码非常笨拙和冗长。编写代码乏味，阅读起来也很困难。
3. 性能损失。反射方法调用比普通方法调用慢了许多。

核心反射机制最初是为了基于组件的应用创建工具而设计的，这类工具通常要根据需要装载类，并且用反射功能找出它们支持哪些方法和构造器。通常，普通应用程序在运行时不应该以反射方式访问对象。

有一些复杂的应用程序需要使用反射机制。这些实例中包括类浏览器、对象监视器、代码分析工具、解释型的内嵌式系统。在RPC(远程过程调用)系统中使用反射机制也是非常合适的，这样可以不再需要存根编译器。

反射机制是一种功能强大的机制，对于特定的复杂编程任务，它是非常必要的，但它也有一些缺点。如果你编写的程序必须要与编译时未知的类一起工作，如果有可能，就应该仅仅使用反射机制来实例化对象，而访问对象时则使用编译时已知的某个接口或者超类。

## 54 谨慎地使用本地方法

Java Native Interface(JNI) 允许java应用程序可以调用本地方法，所谓本地方法，是指用本地程序设计语言（比如C或者C++）来编写的特殊方法。

本地方法主要有三种用途：

1. 它们提供了“访问特定于平台的机制”的能力，比如访问注册表和文件锁；
2. 它们还提供了访问遗留代码库的能力，从而可以访问遗留数据。
3. 本地方法可以通过本地语言，编写应用程序中注重性能的部分，以提高系统性能。

然而随着java平台的不断成熟，jvm实现变得越来越快，而且本地方法也有不安全的特点。所以尽可能少用。（完全不知道怎么用，可以忽略了）

**54 谨慎地进行优化**

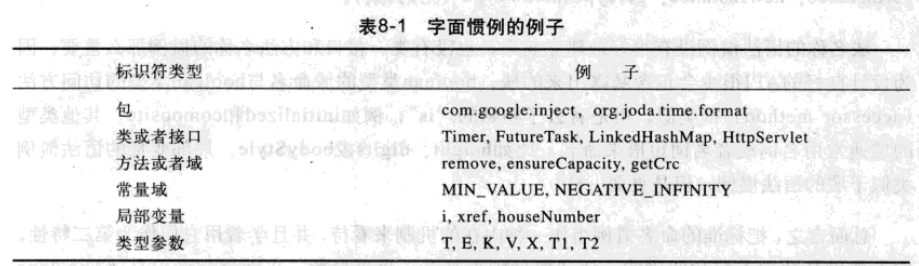
三条与优化有关的格言：

1. 很多计算上的过时都被归咎于效率，而不是任何其他原因——甚至包括盲目地做傻事。
2. 不要去计较效率上的一些小小得失，在97%的情况下，不成熟的优化才是一切问题的根源。
3. 在优化方面，因遵守两条规则：
4. 不要进行优化
5. （仅针对专家）还是不要进行优化——也就是说，在你还么有绝对i清晰的优 化方案之前，请不要进行优化。

不要因为性能而牺牲合理的结构。要努力编写好的程序而不是快的程序。努力避免那些限制性能的设计决策。当一个系统设计完成之后，其中最难以更改的组件是那些指定了模块之间交互关系以及模块与外界交互关系的组件。在这些设计组件之中，最主要的是API、线路层协议以及永久数据格式。这些设计组件不仅在事后难以甚至不可能改变，而且它们都有可能对系统本该达到的性能产生严重的限制。

总之，不要费力去编写快速的程序，应该努力编写好的程序，速度自然会快。在设计系统的时候，特别狮子设计API、线路层协议和永久数据格式的时候，一定要考虑性能的因素。当构建完系统之后，要测量它的性能。如果它足够快，你的任务就完成了。如果不够快，则可以在性能剖析器的帮助下，找到问题的根源，然后设法优化系统中相关的部分。第一个步骤是检查所选择的算法：再多的低层优化也无法弥补算法的选择不当。必要时重复这个过程，在每次改变之后都要测量性能，知道满意为止。

## 56 遵守普遍接受的命名惯例



# 第 9 章 异常

## 57 只针对异常的情况才使用异常

异常应该只用于异常的情况下；它们永远不应该用于正常的普通的控制流，也不要编写迫使它们这么做的API。一般地，应该优先使用标准的、容易理解的模式，而不是那些声称可以提供更好性能的、弄巧成拙的方法。

## 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

在决定使用受检的异常或是未受检的异常时，主要原则是：如果期望调用者能够使当地恢复，对应于这种情况就应该使用受检的异常。通过抛出受检的异常，强迫调用者在一个catch子句中处理该异常，或者将它传播出去。

如果你相信一种情况可能运行恢复，就是用受检的异常；如果不是，则使用运行时异常。如果不清楚是否有可能恢复，最好使用未受检的异常，原因见59条。

因为受检的异常往往指明了可恢复的条件，所以，对于这样的异常，提供一些辅助方法尤其重要，通过这些方法，调用者可以获得一些有助于恢复的信息。例如，假设因为用户没有储存足够数量的钱，他企图在一个收费电话上进行呼叫就会失败，于是抛出受检的异常。这个异常应该提供一个访问方法，以便允许客户查询所缺的费用金额，从而可以将这个数值传递给电话用户。

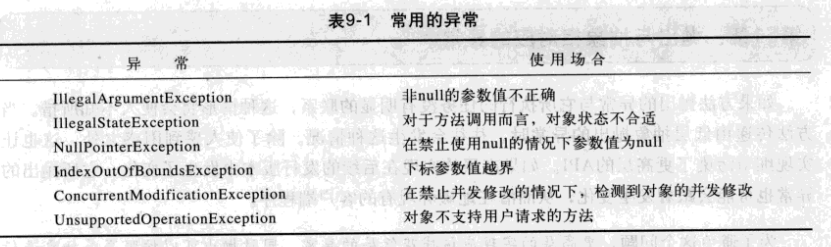
## 59 避免不必要地使用受检的异常

## 60 优先使用标准的异常

java平台类库提供了一组基本的未收检异常，它们满足了绝大多数API的异常抛出需要。

重用现有异常的好处：使你的API更加易于学习和使用，可读性好，异常类越少，装载异常类的时间开销也越少。

经常被重用的异常：



另外 ArithmeticException, NumberFormatException,

## 61 抛出与抽象相对应得异常

如果方法抛出的异常与它所执行的任务没有明显的联系，这种情形会使人不知所措，为了避免这个问题，更高层的实现应该捕获低层的异常，同时抛出可以按照高层抽象进行解释的异常。这种做法被称为异常转译。如果低层的异常原因对高层问题有帮助，那么久使用异常链，将低层异常对象传递到高层，高层提供访问方法来获得低层的异常信息。

try{...

}catch(LowerLevelException cause){

throw new HigherLevelException(cause);

}

class HigherLevelException extends Exception{

HigherLevelException(Throwable cause){

super(cause);

}

}

总之，如果不能阻止或者处理来自更低层的异常，一般的做法是使用异常转译，除非低层方法碰巧可以保证它抛出的所有异常对高层也合适才可以将异常从低层传播到高层。异常链对高层和低层异常都提供了最佳的功能：它允许抛出适当的高层异常，同时又能捕获低层的原因进行失败分析。见63条

## 62 每个方法抛出的异常都要有文档

描述一个方法所抛出的异常，是正确使用这个方法时所需文档的重要组成部分。因此，花点时间仔细为每个方法抛出的异常建立文档是特别重要的。

文档记录下未受检及受检异常是非常有必要的，而且声明throws 异常时尽量准确，要为每一个受检异常提供单独的throws字句，不要为未受检的异常提供throws字句，抛出超类异常只会让后续调用者更加难以理解究竟是何异常。

## 63 在细节消息中包含能捕获失败的信息

为了捕获失败，异常的细节信息应该包含所有“对该异常有贡献”的参数和域的值。如下标越界时，异常信息能给出是上界还是下界或者无效值，以便程序员查找错误。

## 64 努力使失败保持原子性

当对象抛出异常之后，通常我们期望这个对象仍然保持在一种定义良好的可用状态之中，对于受检查异常而言，这尤其重要，因为调用者期望能从这种异常中恢复。一般而言，失败的方法调用应该使对象保持在被调用之前的状态。具有这种属性的方法被称为具有失败原子性。

对于不可变对象，失败原子性就是显然的，因为方法执行改变不了对象的一致性。对于可变对象，最好在执行操作之前先检查参数有效性（38条），如果参数异常就先抛出异常，这样对象一致性没有遭破坏。另一种做法是编写一段恢复代码，由它来拦截操作过程中发生的失败，以及使对象回滚到操作开始之前的状态上。最后一种方法是在对象的临时备份上执行操作，当操作完成，再用临时拷贝中的结果（操作后可能存在变化）代替对象的内容。

## 65 不要忽略异常

要忽略一个异常非常容易，只需将方法调用通过try-catch语句包围起来，而不对异常作任何处理。至少，catch块也应该包含一条说明，解释为什么可以忽略这个异常。

有一种情形可以忽略异常，即关闭FileInputStream的时候。因为你还没有改变文件的状态，因此不必执行任何恢复动作，并且已经从文件中读取到所需的信息，因此不必终止正在进行的操作。即使在这种情况下，把异常记录下来还是明智的做法，因为如果这些异常经常发生，你就可以调查异常的原因。

# 第 10 章 并发

## 66同步访问共享的可变数据

关键字synchronized可以保证在同一时刻，只有一个线程可以执行某个方法或者一个代码块。正确地使用同步可以保证没有任何方法会看到对象处于不一致的状态中。

volatile修饰变量能保证每次读取变量都是最新值。但使用时要小心：

获取唯一序列码的方法

private static volatile int nextSerialNumber=0;

public static int generateerialNumber(){return nextSerialNumber++;}

如果方法没有同步，当多个线程调用generateerialNumber方法时，如果第一个线程读取了nextSerialNumber，还没来得及++，另一个线程又读到了相同的值。与设计不符。问题的原因在于，增量操作符++不是原子性的。

修正：

方法1：

private static int nextSerialNumber=0;

public static synchronized int generateerialNumber(){return nextSerialNumber++;}

方法2：

private static final AtomicLong nextSerialNum=new AtomicLong();

public static long generateSerialNumber(){

return nextSerialNum.getAndIncrement();

}

AtomicLong是java.util.concurrent.atomic下的类，可以进行原子操作。

简而言之，当多个线程共享可变数据的时候，每个读或者写数据的线程都必须执行同步。如果没有同步就无法保证一个线程所做的修改可以被另一个线程获知。如果只需要线程之间的交互通信，而不需要互斥，volatile修饰符就是一种可以接受的同步形式。

## 67 避免过度同步（需再次理解原文档）

依据情况不同，过度的同步可能会导致性能降低、死锁，甚至不确定的行为。

# 第 11章 序列化

## 74 谨慎地实现Serializable接口

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。

（如果确实需要序列化，那么就显式地定义private static final long serialVersionUID=..L，

如果不定义，系统会自动根据这个类来调用一个复杂的运算过程来生成UID，而且如果类有变化，需要重新生成UID，且与原来的UID不同，使类的兼容性遭到破坏，运行时导致InvalidClassException异常）

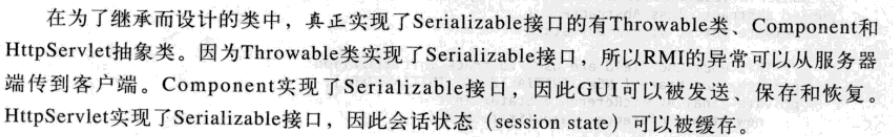
实现Serializable的第二个代价是，它增加了出现bug和安全漏洞的可能性。

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。

根据经验，

比如Date和BigInteger这样的值类应该实现Serializable，大多数的集合类也应该如此；为了继承而设计的类，应该尽可能少的去实现Serializable接口，用户的接口也应该尽可能少地继承Serializable接口。

代表活动实体的类，比如线程池，一般不应该实现Serializable。

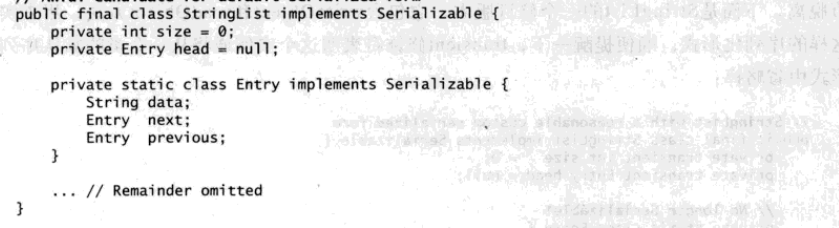


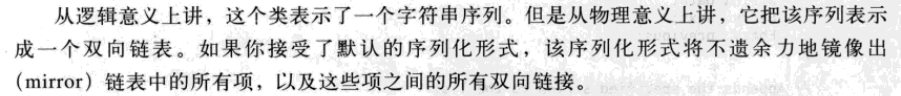
如果你实现了一个带有实例域（不赋值时会默认0，boolean，null等,没有值时序列化会赋默认值，导致错误）的类，它是可序列话和可扩展的，那么你就要给这个类添加readObjectNoData方法来约束它。

private void readObjectNoData() throws InvalidObjectException{

throw new InvalidObjectException(“Stream data required”);}

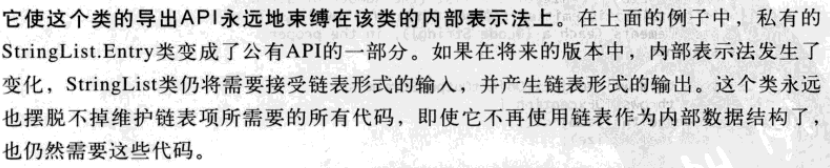
## 75 考虑使用自定义的序列化形式



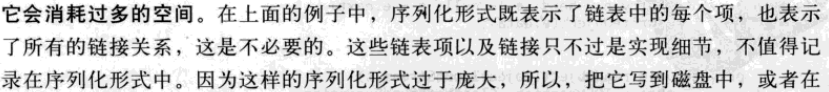


当一个对象的物理表示法与它的逻辑数据内容有实质性的区别时，使用默认序列化形式会有以下4个缺点：

1. 它使这个类的导出API永远束缚在该类的内部表示法上。

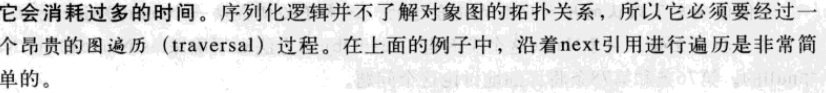


1. 它会消耗过多的空间。



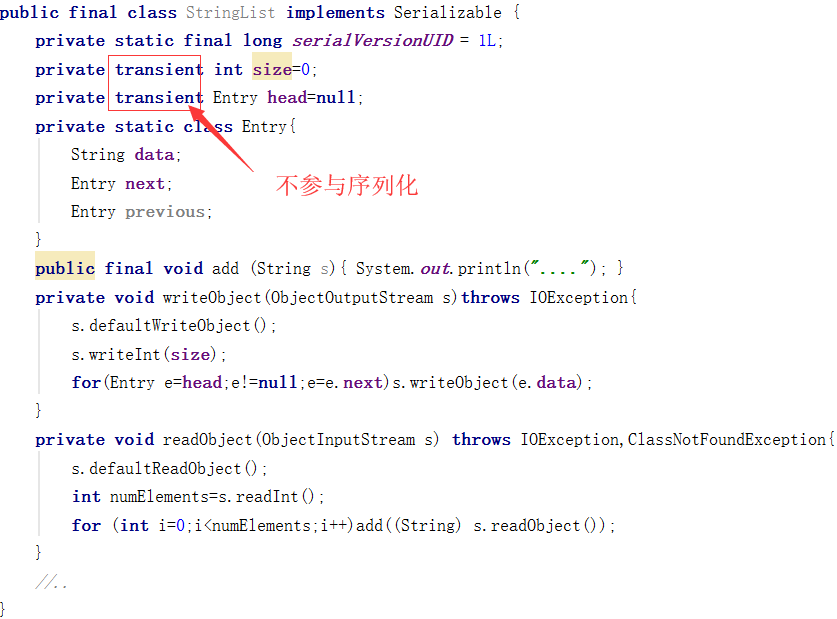


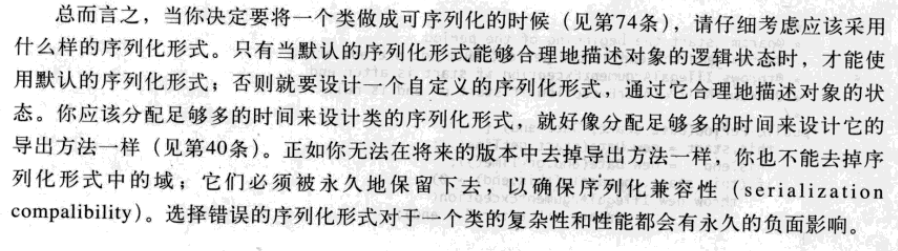
1. 它会消耗过多的时间。



1. 它会引起栈溢出。

修改上述类：自定义序列化形式





## 76 保护性地编写readObject方法