方法

38、检查参数的有效性

(1)每当编写方法或者构造器的时候，应该考虑他的参数有哪些限制。应该通过显示检查来实施这些限制。

(2)绝大多数方法或者构造器都有某些限制，如索引必须是非负数，对象引用不能为 null，等等。如果一个无效的参数传入，应该首先对参数进行检查，如果有问题，必须清楚地抛出一个异常并指明错误原因。

(3)检查参数有效性的方法：

a. if 语句,通过 if 语句检查错误，然后抛出异常

b.断言(assert)，如果断言失败，将会抛出 AssertionError

39、必要时进行保护性拷贝

考虑下面的类，声称表示一段不可变的时间周期：

```
import java.util.Date;

public final class Period {
    private final Date start;
    private final Date end;

    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;
    }
}
```

因为 Date 本身是可变的，因此很容易违反起始时间不能在结束时间之后的约束。

```
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78);//改变了终止时间,破坏类的约束条件
```

为保护 Period 实例的内部信息被破坏，对构造器的每个可变参数进行保护性拷贝，使用备份对象作为 Period 实例的组件，而不使用客户端传入的参数。

```
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);
}
```

进行了上面的工作后，改变 Period 实例仍有可能，因为它的访问方法提供了对其可变内部成员的访问能力：

```
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78);
```

改变访问方法，返回可变内部域的保护性拷贝：

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

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

采用新构造器和新访问方法后，Period 真正地是不可变的了，不管程序员多么恶意和不合格，都绝对不会违反周期起始时间大于结束时间的约束。

40、谨慎地设计方法签名

(1)谨慎地选择方法的名称，方法的名称应该始终遵循标准的命名习惯

a.易于理解

b.选择与大众认可的名称相一致的名称

(2)不过过于追求提供便利的方法

每个方法都应该尽其所能。方法太多会使类难以学习、使用、文档化、测试和维护。对于接口，更是这样，方法太多会使用接口实现者和接口用户的工作变得复杂起来。对于类和接口所支持的每个动作，都提供一个功能齐全的方法。只有当一项操作被经常用到时，考虑为它提供快捷的方式。

(3)避免过长的参数列表，保持参数在四个，或者更少

(4)缩短参数列表的方法

a.把方法分解成多个方法，每个方法只需要这些参数的一个子集

b.创建辅助类，用来保存参数的分组。这些辅助类一般为静态成员类。

c.从对象构建到方法都采用 Builder 模式

如果方法有多个参数，其中有些又是可选的，最好定义一个对象来表示所有参数，并允许客户端在这个对象上进行多次“setter”调用，每次调用都设置一个参数或设置一个较小的集合，一旦设置了所需要的参数，客户端就调用这个对象的“执行（execute）”方法，它对参数进行最终的有效性检查，并执行实际的计算。

41、慎用重载

总之，“能够重载方法”并不意味着就“应该重载方法”，一般情况下，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。在某些情况下，特别是涉及构造器的时候，要遵循这条建议也许是不可能的。在这种情况下，至少应该避免这样的情形：同一组参数只需经过类型转换就可以被传递给不同的重载方法，在这种情况下，我们坚决抵制这数的重载。

42、慎用可变参数

Java 1.5 增加可变参数方法，可变参数方法接受 0 个或者多个指定类型的参数。

(1)可变参数的机制通过先创建一个数组，数组的大小为在调用位置所传递的参数数量，然后将参数值传到数组中，最后将数组传递给方法。

```
static int sum(int... args) {
        int sum=0;
        for(int arg : args)
                sum += arg;
        return sum;
}
```

该方法如期望的那样，sum()=0,sum(1,2,3)=6

```
static int min(int firstArg, int... remainingArgs) {//可变参数必须放在参数列表的最后
        int min = firstArg;
        for(int arg : remainingArgs)
                if(arg < min)
                        min = arg;
        return min;
}
```

(2) 可变参数引起的问题，如下 demo

```
public static void main(String[] args)
{
    int [] arr={1,2,3,4,5,6,7,8,9,10};
    System.out.println(Arrays.asList(arr));
    System.out.println(Arrays.toString(arr));
}
```

运行结果为：

[[I@18a992f]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

这是由于可变参数引起的，Aarrys.asList()方法的源码为：

```
public static <T> List<T> asList(T... a) {
    return new ArrayList<T>(a);
}
```

它把数组 arr 装入 List 后变成 List<int[]>了，实际上我们需要的是 List<Integer>。 综上，实用可变参数确实要小心一些。所以，调用可用参数需要非常小心。

43、返回零长度的数组或者集合，而不是 null

如果返回了 null，客户端程序员就要检测返回的是不是 null，然后才能进行下一步操作，否则就会引发 NullPointException。但是如果是返回的的是空数组或者集合，就不会再后续的使用这个对象上，引发空指针异常

(1)如何返回一个零长度的数组

可以用 new int[0]之类的方法得到一个零长度的数组

(2)如何返回一个空集合

a.Collections.emptyList()

b.Collections.emptySet();

c.Collections.emptyMap();

44、为所有到处的 API 元素编写文档注释

通用规则

(1)必须在每个被导出的类、接口、构造器、方法和域声明之前加一个文档注释

(2)方法的文档注释应该简介地描述它和客户端之间的约定

a.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

所谓前提条件是指为了使客户能够调用这个方法而必须要满足的条件；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，前提条件由@throws 标签描述隐含的异常，也可以在一些受影响参数的@param 标签中指定前提条件

b.其次，文档注释中还应该描述方法的副作用，所谓副作用是指方法执行后对系统状态的影响。例如，如果方法启动了后台线程，文档中就应该说明这一点

c. 最后，文档注释也应该描述类或者方法的线程安全性