【软件构造】第三章第二节 设计规约

第三章第二节 软件规约

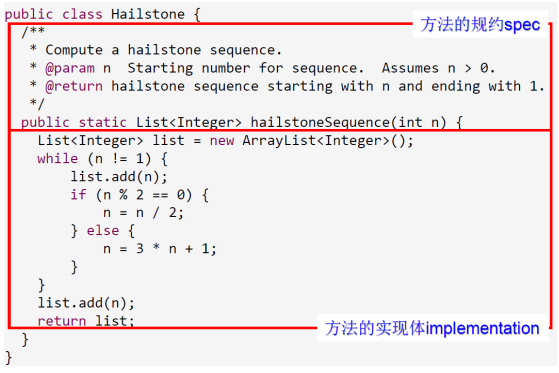
这一节我们转向关注“方法/函数/操作”是如何定义的，即讨论编程中的动词，规约。

Outline

- 一个完整的方法
- 什么是设计规约，我们为什么需要他
- 行为等价性
- 规约的结构：前置条件与后置条件
  - 规约的结构
  - 可变方法的规约
- 规约的评价
  - 规约的确定性
  - 规约的陈述性
  - 规约的强度
  - 如何设计一个好的规约
  - 是否使用前置条件

Notes

一个完整的方法

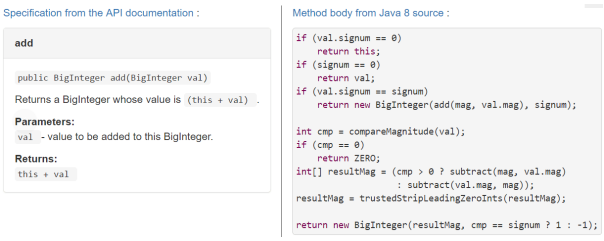



```
public class Hailstone {
  /**
   * Compute a hailstone sequence.
   * @param n  Starting number for sequence.  Assumes n > 0.
   * @return hailstone sequence starting with n and ending with 1.
   */
  public static List<Integer> hailstoneSequence(int n) {
    List<Integer> list = new ArrayList<Integer>();
    while (n != 1) {
        list.add(n);
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    list.add(n);
    return list;
  }
}
```

- 一个完整的方法包括规约spec和实现体implementation；
- "方法"是程序的积木，它可以被独立的开发、测试、复用；
- 使用“方法”的客户端，无需了解方法内部如何工作，这就是抽象的概念；
- 参数类型和返回值类型的检查都是在静态类型检查阶段完成的。
- 更多关于方法的内容，请参考 RUNBOOB Java 方法

什么是设计规约，我们为什么需要他

- 为什么要有设计规约
  - 很多bug来自于双方之间的误解；没有规约，那么不同开发者的理解就可能不同
  - 代码惯例增加了软件包的可读性，使工程师们更快、更完整的理解软件
  - 可以帮助程序员养成良好的编程习惯，提高代码质量
  - 没有规约，难以定位错误
- 使用设计规约的好处
  - 规约起到了契约的作用。代表着程序与客户端之间达成的一致；客户端无需阅读调用函数的代码，只需理解spec即可。
  - 精确的规约，有助于区分责任，给“供需双方”确定了责任，在调用的时候双方都要遵守。
- 实例



  - 规约可以隔离“变化”，无需通知客户端
  - 规约也可以提高代码效率
- 实例参考 阿里Java开发手册之编程规约

行为等价性

行为等价性就是站在客户端的角度考量两个方法是否可以互换

参考下述两个函数：

```
static int findFirst(int[] arr, int val) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == val) return i;
    }
    return arr.length;
}

static int findLast(int[] arr, int val) {
    for (int i = arr.length - 1 ; i >= 0; i--) {
        if (arr[i] == val) return i;
    }
    return -1;
}
```

- 行为等价性分析：
  - 当val不存在时，`findFirst`返回arr的长度，`findLast`返回-1；
  - 当val出现两次时，`findFirst`返回较低的索引，`findLast`返回较高的索引。
  - 但是，当val恰好出现在数组的一个索引处时，这两个方法表现相同。
  - 故，如果调用方法时，都传入一个 正好具有一个val的arr ，那么这两种方法是一样的。
- 另外，我们也可以根据规约判断是否行为等价注：规约与实现无关，规范无需讨论方法类的局部变量或方法类的私有字段。

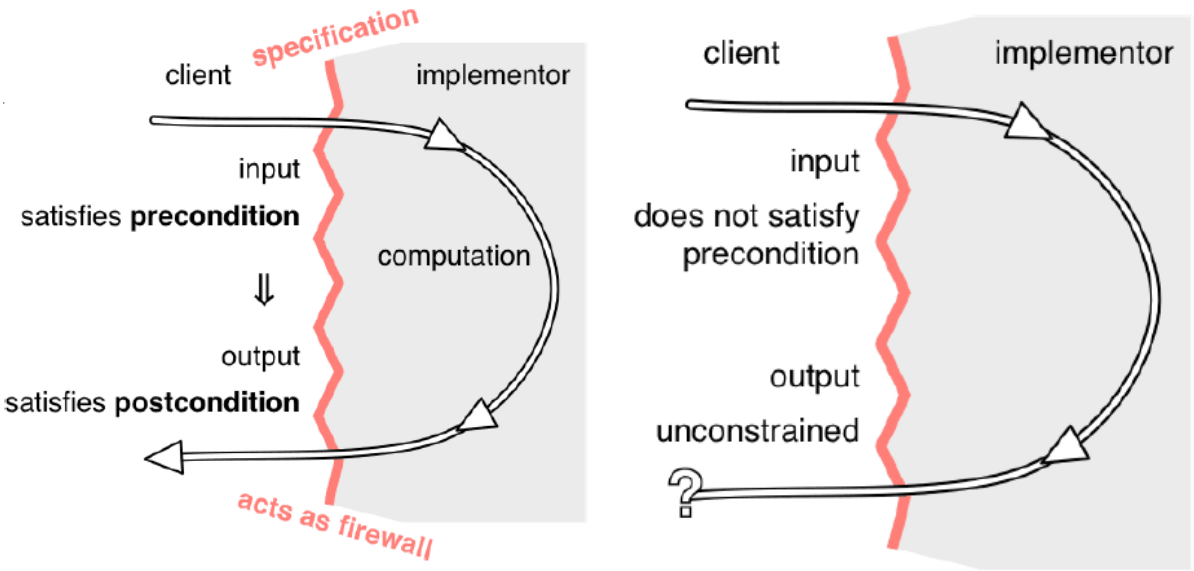
```
o static int findExactlyOne(int[] arr, int val)
  requires: val occurs exactly once in arr
  effects:  returns index i such that arr[i] = val
```

  - 两个函数附和同一个规约，故二者等价

规约的结构：前置条件与后置条件

【规约的结构】

- 一个方法的规约常由以下几个短句组成契约：如果前置条件满足了，后置条件必须满足。如果没有满足，将产生不确定的异常行为
  - 前置条件(precondition)：对客户端的约束，在使用方法时必须满足的条件。由关键字 **requires** 表示；
  - 后置条件(postcondition)：对开发者的约束，方法结束时必须满足的条件。由关键字 **effects** 表示
  - 异常行为(Exceptional behavior)：如果前置条件被违背，会发生什么
-



-

- 静态类型声明是一种规约，可据此进行静态类型检查。
- 方法前的注释也是一种规约，但需人工判定其是否满足。
  - 参数由@param 描述
  - 子句和结果用 @return 和 @ throws子句 描述
  - 尽可能的将前置条件放在 @param 中

尽可能的将后置条件放在 @return 和 @throws 中

```
/**
 * Find a value in an array.
 * @param arr array to search, requires that val occurs exactly once
 *            in arr
 * @param val value to search for
 * @return index i such that arr[i] = val
 */
static int find(int[] arr, int val)
```

【mutating methods(可变方法)的规约】

- 除非在后置条件里声明过，否则方法内部不应该改变输入参数。
- 应尽量遵循此规则，尽量不设计 mutating的spec，否则就容易引发bugs。
- 程序员之间应达成的默契：除非spec必须如此，否则不应修改输入参数 。
- 尽量避免使用可变(mutable)的对象 。
  - 对可变对象的多引用，需要程序维护一致性，此时合同不再是单纯的在用户和实现者之间维持，需要每一个引用者都有良好的习惯，这就使得简单的程序变得复杂；
  - 可变对象使得程序难以理解，也难以保证正确性；
  - 可变数据类型还会导致程序修改变得异常困难；

规约的评价

规约评价的三个标准

- 规约的确定性
- 规约的陈述性
- 规约的强度

【规约的确定性】

确定的规约：给定一个满足前置条件的输入，其输出是唯一的、明确的

```
static int find_ExactlyOne(int[] arr, int val)
  requires: val occurs exactly once in arr
  effects:  returns index i such that arr[i] = val
```

欠定的规约：同一个输入可以有多个输出

```
static int find_OneOrMore,AnyIndex(int[] arr, int val)
  requires: val occurs in arr
  effects:  returns index i such that arr[i] = val
```

未确定的规约：同一个输入，多次执行时得到的输出可能不同；但为了避免分歧，我们通常将不是确定的spec统一定义为欠定的规约。

【规约的陈述性】

- 操作式规约（Operational specs）：伪代码 。
- 声明式规约（Declarative specs）：没有内部实现的描述，只有 “初-终”状态 。
- 声明式规约更有价值； 内部实现的细节不在规约里呈现，而放在代码实现体内部注释里呈现。

举一个栗子：

```
static String join(String delimiter, String[] elements)
effects : returns the result of adding all elements to a new : StringJoiner(delimiter) // Operational specs

effects:returns the result of looping through elements and alternately appending an element and the delimiter // Operational specs

effects: returns concatenation of elements in order, with delimiter inserted between each pair of adjacent elements // Declarative specs
```

【规约的强度】

- 通过比较规约的强度来判断是否可以用一个规约替换另一个；
- 如果规约的强度 S2>=S1，就可以用S2代替S1，体现有二：一个更强的规约包括更轻松的前置条件和更严格的后置条件；越强的规约，意味着实现者(implementor)的自由度和责任越重，而客户(client)的责任越轻。
  - S2的前置条件更弱
  - S2的后置条件更强

举一个栗子：

- Original spec：

```
1 static int findExactlyOne(int[] a, int val)
2   requires: val occurs exactly once in a
3   effects:  returns index i such that a[i] = val
```

- A stronger spec:

```
1 static int findOneOrMore,AnyIndex(int[] a, int val)
2   requires: val occurs at least once in a
3   effects:  returns index i such that a[i] = val
```

- A much stronger spec:

```
1 static int findOneOrMore,FirstIndex(int[] a, int val)
2   requires: val occurs at least once in a
3   effects:  returns lowest index i such that a[i] = val
```

【如何设计一个好的规约】

- 规约应该是简洁的：整洁，具有良好的结构，易于理解。
- 规约应该是内聚的：Spec描述的功能应单一、简单、易理解。
- 规约应该是信息丰富的：不能让客户端产生理解的歧义。
- 规约应该是强度足够的：需要满足客户端基本需求，也必须考虑特殊情况。
- 规约的强度也不能太强：太强的spec，在很多特殊情况下难以达到。
- 规约应该使用抽象类型：在规约里使用抽象类型，可以给方法的实现体与客户端更大的自由度。

【是否使用前置条件】

是否使用前置条件取决于如果只在类的内部使用该方法(private)，那么可以不使用前置条件，在使用该方法的各个位置进行check——责任交给内部client。

check的代价；

方法的使用范围；

如果在其他地方使用该方法(public)，那么必须要使用前置条件，若client端不满足则方法抛出异常。