JML Level 0 简化手册

【以下为王浩羽大佬的整理，此文为引用】

在学习课程组下发的《JML手册》（简写，下文同）过程中，随手记录了一些我认为重要或值得注意的内容，在此和大家分享。

本文中涉及的全部内容均可以在《JML手册》中找到，可以当作简化版的手册使用。

一、注释和基本关键字

1. 两种注释格式

`JML`中以`javadoc`注释的方式表示规格，每一行都以`@`开头，注释方式有两种：行注释和块注释。两种注释方式分别可以表示为：

- 行注释：**//@ 注释内容**（注意中间的空格）

```
//@ annotation
```

- 块注释：**/*@ 注释内容 @*/**

```
/*@ annotation1
  @    annotation2
  @    annotation3
  @    annotation4
  @    annotation5
  @    annotation6
  @    annotation7
  @    annotation8
  @*/
```

按照`javadoc`的习惯，`JML`注释一般放在被注释成分的近邻上部。

2. 一些其他的关键字

- 方法中间的`pure`：表示该方法是纯粹查询方法，不会造成任何其他影响（即**没有副作用**）,如`Level 0`手册中的两个方法：

```
public abstract /*@ pure @*/ int largest();

public abstract /*@ pure @*/ int size();
```

- `model`：表示规格描述，仅仅用作**规格变量的声明**，无需在类中声明和定义。
- `non_null`：数组的引用对象不能为空。

*`normal_behavior`：正常的行为（下面会提到），其下文往往是前置条件、副作用范围限定和后置条件，仍旧拿手册中的类举例：

```
public abstract class IntHeap { // line 3
```

```
    //@ public model non_null int [] elements;

    /*@ public normal_behavior
    @ requires elements.length >= 1;
    @ assignable \nothing;
    @ ensures \result
    @         == (\max int j;
    @             0 <= j && j < elements.length;
    @             elements[j]);
    @*/
    public abstract /*@ pure @*/ int largest();
    //@ ensures \result == elements.length;
    public abstract /*@ pure @*/ int size();
};
```

- `requires`：定义方法的前置条件，在本例中即

  ```
  elements.length >= 1;//elements中至少有一个元素
  ```

- `assignable`：副作用范围限定，列出方法能够修改的类成员属性
- `\nothing`：表示方法不对任何成员属性进行修改，是`pure`方法（详见上文）
- `ensures`：定义方法的后置条件，在本例中即

  ```
  ensures \result == (\max int j; 0 <= j && j < elements.length; elements[j]);
  ```

  作用为获取`elements`中的最大元素。（其中的`\max`是取最大数的关键词）

  注意：规格中的每个子句都要以分号结尾！

3. 规格变量

`JML`中规格变量分为两种：静态和实例。如果在接口中声明规格变量，需要明确变量的类别。

- 静态：增加`static`关键字，访问规则与`JAVA`中`static`类型变量相似。

```
//@public static model non_null int []elements
```

- 实例：增加`instance`关键字，不声明时默认为实例型规格变量。

```
//@public instance model non_null int []elements
```

二、JML表达式

1. 原子表达式

①\result

表示一个返回值类型非`void`的方法的执行结果，也就是函数返回值，**类型与方法定义时的声明相同。**

②\old(expr)

表示表达式`expr`在相应方法执行前的取值，遵循`JAVA`引用规则，即：针对一个对象引用，只判断**引用本身**是否发生变化，并不判断**引用指向的对象实体的内容**是否发生变化。故只有当引用本身发生了变化（指向了另外一个对象）时，取值才会发生变化。

在使用时，我们应该把`\old()`关心的表达式取值**整体括起来**。

③\not_assigned(x, y, ...)

表示括号中的变量是否在方法执行过程中被赋值，没有的话返回`true`，否则返回`false`。主要用于后置条件的约束上，用以限制方法的实现**不能对列表中的变量赋值**。

④\not_modified(x, y, ...)

限制括号中的变量在方法执行期间取值未发生变化。

⑤\nonnullelements(container)

表示`container`对象中存储的对象不会有`null`，等价于如下断言：

```
container != null && (\forall int i; 0 <= i && i < container.length;container[i] != null)
```

⑥\type(type)

返回类型`type`对应的类型（`Class`），如`\type(boolean)`为`Boolean.TYPE`。`TYPE`是`JML`中的缩略表示，等同于`Java`中的`java.lang.Class`。

⑦\typeof(expr)

返回`expr`对应的准确类型，如`\typeof(false)`为`Boolean.TYPE`。

2. 量化表达式

①\forall

全称量词修饰的表达式，表示对应给定范围内的元素，**每个元素都满足对应的约束。**

举例来说，`(\forall int i,j; 0 <= i && i <= j&& j < 10;a[i] < a[j])`的意思为，对于任意`0<=i<j<=10`，`a[i]<a[j]`。这个表达式如果为真，表明数组`a`事实上是升序排列。

②\exists

存在量词修饰的表达式，表示对于给定范围内的元素，**存在某个元素满足对应的约束。**

举例来说，`(\exists int i; 0 <= i && i < 10;a[i] < 0)`的意思为，对于`0 <= i < 10`，存在一个`a[i] < 0`。

③\sum

求和表达式，返回给定范围内表达式的和。

`(\sum int i; 0<= i && i < 5;i)`，意为计算`[0,5)`范围内的整数`i`的和，即：0+1+2+3+4=10。值得注意的是，`0 <= i && i < 5`是对`i`的范围的限制，而**求和表达式**是最后的那个`i`。

进一步说，如果表达式为`(\sum int i; 0 <= i && i < 5 ; i * i)`，则等价于表达式：$\sum_{4}^{i=0} i^2$

④\product

返回给定范围内表达式连乘的结果，如`(\product int i;0 < i && i < 5; i)`等价于：$\prod_{n=1}^{4} i$

⑤\max

返回给定范围内表达式的最大值，如`(\max int i; 0 <= i && i < 5;i)`，这个表达式返回`[0,5)`中最大的整数，即4 。

⑥\min

返回最小值

⑦\num_of()

返回指定变量中满足相应条件的取值个数。如：`(\num_of int x; 0 < x && x <= 20;x % 2 == 0)`，这个表达式给出(0,20]中可以被2整除的整数的个数，即`10`。

一般而言，对于`(\num_of T x; R(x); P(x))`，`T`为变量`x`的类型，`R(x)`为`x`的取值范围，`P(x)`定义`x`需要满足的约束条件。从逻辑等价角度看，相当于`(\num_of T x; R(x) && P(x);1)`

3. 集合表达式

可以在JML规格中构造一个局部的集合（容器），明确集合中包含的元素。例如下面这句声明：

```
new JMLObjectSet {Integer i | s.contains(i) && 0 < i.intValue()}
```

其含义为：构造一个`JMLObjectSet`对象，其包含的元素类型为`Integer`，都在容器`s`中出现，并且整数值大于0。

PS：这里的`s`代表**容器集合**，也就是我们平时在作业中使用的`ArrayList`、`HashSet`、`HashMap`等容器！

更加通俗地讲，集合构造表达式的一般形式可以表述为：

```
new ST{T x | R(x) && P(x)}
```

其中`R(X)`对应集合中`x`的范围，通常是来自于**某个既有集合中的元素**，如`s.has(x)`，`P(x)`对应`x`取值的约束。

4. 操作符

①子类型关系操作符——`<:`

`E1<:E2`，如果`E1`是类型`E2`的**子类型**(sub type)或`E1`和`E2`是**相同的类型**，那么表达式的**结果为真**，否则为假。

举例如下：

```
Integer.TYPE <: Integer.TYPE//为真
Integer.TYPE <: ArrayList.TYPE//为假
```

PS：对于任何类`X`，都满足`X.TYPE <: Object.TYPE`，因为任何类都是`Object`的子类。

②等价关系操作符——`<==>`和`<=!=>`

`b_expr1<==>b_expr2`或者`b_expr1<=!=>b_expr2`，其中`b_expr1`和`b_expr2`是布尔表达式，意为`b_expr1 == b_expr2`或者`b_expr1 != b_expr2`。可以看出这两个操作符和Java中的`==`与`!=`效果相同，但是按照JML的定义，`<==>`的优先级低于`==`，`<=!=>`低于`!=`。

③推理操作符——`==>`和`<==`

对于表达式`b_expr1==>b_expr2`或者`b_expr2<==b_expr1`，当`b_expr1==false`或者`b_expr1==true`且`b_expr2==true`时，整个表达式的值为`true`。

PS：和离散课上学的有些类似。若前式为假，结果一定为真；若前式为真，需要后式同时为真，结果才能为真。

④变量引用操作符

- `\nothing`表示空集。
- `\everything`表示全集，包括当前作用域下能访问到的所有变量。

该操作符经常在`assignable`句中使用，如`assignable \nothing`表示当前作用域下的每个变量**都不可以**在方法执行过程中被赋值。

三、方法规格

方法规格的核心内容包括三个方面：**前置条件、后置条件和副作用约定。**

1. 前置条件：对方法**输入参数**的限制，若不满足，则方法执行结果**不可预测**（无法保证方法结果的正确性）。
2. 后置条件：对方法**执行结果**的限制，如果执行结果满足后置条件，则表示方法执行正确，否则执行错误。
3. 副作用约定：方法在执行过程中对输入对象或`this`对象进行了**修改**（对其成员变量进行了赋值，或者调用其修改方法）。

两类方法：**全部过程和局部过程。**

1. 全部过程：前置条件**恒为真**，即可以适应于**任意调用场景。**
2. 局部过程：前置条件**不恒为真**，要求调用者必须确保调用时**满足相应的前置条件。**

从设计角度看，我们的软件应当能够处理用户得到所有可能输入，因此，需要对不符合前置条件的输入进行处理，这一般意味着**异常处理。**

从规格角度，JML区分两种场景，对应**正常行为规格**和**异常行为规格。**

1. 前置条件

通过`requires`子句表示：`requires P;`。其中`requires`是JML的关键词，表示“要求调用者确保P为真”。

值得注意的是，方法规格中可以有**多个**`requires`子句，为**并列关系**，即，调用者必须**同时满足所有的并列子句要求。**

如果设计者想表达或的逻辑，则应当使用一个`requires`子句，在其中的谓词`P`中使用逻辑或操作符表达相应的约束，如`requires P1 || P2;`。

2. 后置条件

通过`ensures`子句表示：`ensures Q;`。其中`ensures`是JML的关键词，表示"方法实现者确保方法执行返回的结果一定满足谓词Q的要求，即，确保Q为真"。

多个`ensures`子句是被允许的且为**并列关系即须同时满足**。类似地，**或**的表达应当在一个`ensures`子句中用逻辑或`||`约束。

3. 副作用约定

副作用：方法在执行过程中会**修改对象的属性数据或者类的静态成员数据**，从而给后续方法的执行造成影响。

从方法规格的角度，必须明确给出副作用的范围。JML提供了副作用范围子句，使用关键词`assignable`或`modifiable`。

从语法上看，副作用约束子句有两种形态：

- 不指明具体的变量，而是用JML关键词来概括。
- 指明具体的变量列表。

以手册中的栗子为模板解释：

①变量可否被修改

```
public class IntegerSet{
    private /*@spec_public@*/ ArrayList<Integer> elements;
    private /*@spec_public@*/ Integer max;
    private /*@spec_public@*/ Integer min;
    /*@
      @ assignable \nothing;
      @ assignable \everything;
      @ modifiable \nothing;
      @ modifiable \everthing;
      @ assignable elements;
      @ modifiable elements;
      @ assignable elements, max, min;
      @ modifiable elements, max, min;
      @*/
}
```

- `assingable`表示可赋值，`modifiable`表示可修改，两者在大部分情况下可以交换使用。
- `\nothing`和`\everything`：前者表示当前作用域中可见的所有类成员变量和方法输入对象**都不可以被赋值或修改**；后者表示示当前作用域内可见的所有类成员变量和方法输入对象**都可以被赋值或者修改。**
- 也可以指明具体的**可以修改的变量列表**，为一个或多个变量，多个的时候需要用逗号分隔，如`@assignable elements, max, min`。

②纯粹访问性

纯粹访问性方法：不会对对象的状态进行任何改变，也无需提供输入参数，无需描述前置条件，也不会有任何副作用，且一定可以正常执行结束。

可以使用`pure`关键词来描述：

```
public /*@ pure @ */ String getName();//不需做任何限定，是一种极简的场景

//@ ensures \result == bachelor || \result == master;
public /*@ pure @*/ int getStatus();//限定返回值等于bachelor或master

//@ ensures \result >= 0;
public /*@ pure @*/ int getCredits();//限定返回值大于等于0
```

在方法规格中，可以引用`pure`方法返回的结果：

```
/*@ requires c >= 0;
    @ ensures getCredits() == \old(getCredits()) + c;
    @*/
public void addCredits(int c);
```

有时前置条件和后置条件要对**不止一个变量**进行约束，这时需要使用`\forall`或`\exists`表达式，这块在手册中已经说的很清楚了，在此不多赘述。

③方法正常功能和异常行为的区分

仍旧援引手册中的栗子：

```
/*@ public normal_behavior //对方法的正常功能给出规格
  @ requires z <= 99;//
  @ assignable \nothing;//
  @ ensures \result > z;//
  @ also //连接正常规格和异常规格
  @ public exceptional_behavior //对方法的异常功能给出规格
  @ requires z < 0;
  @ assignable \nothing;
  @ signals (IllegalArgumentException e) true;
  @*/
public int cantBeSatisfied(int z) throws IllegalArgumentException;
```

部分语句的解释见代码中注释，下面列出一些值得注意的点：

- 正常功能：输入方法关联`this`对象的状态在**正常范围**内是时所指向的功能，与之相对的是异常功能。
- `public`关键词表示：相应的规格在**所在包范围内**的所有其他规格处都可见。
- 若一个方法**无异常处理行为**，就**无需区分**正常功能规格和异常功能规格，也**无需使用**这两个关键词了。
- 使用`also`关键词的两个场景：
    - 父类中对相应方法定义了规格，子类重写了该方法，需要**补充规格**，这时应该在补充的规格之前使用`also`。
    - 一个方法规格中设计了**多个功能规格描述**，正常功能规格或者异常功能规格，需要使用`also`来**分隔**。
- 细心的小朋友还可以看出，这段规格描述存在`bug`，可以停下来好好想想在哪里哦。（**手册里有解释**）
- 不论是正常功能规格还是异常功能规格，都包括了**前置条件、后置条件和副作用声明**。但不同的是，异常功能规格中，后置条件常表示为**抛出异常**，使用`signals`子句表示。

4. signals子句

用来表示抛出异常，一般的结构为：

```
signals (***Exception e)b_expr//当b_expr为true时，抛出括号中的异常e
```

- 抛出的异常既可以是`Java`预定义的异常类型，也可以是用户自定义的异常类型。
- 如果一个方法在运行时抛出异常，一定要在**方法声明**中明确指出（使用`Java`的`throws`表达式），并且保证`signals`子句中给出的异常类型一定等同于**方法声明中给出的异常类型**，或者是**后者的子类型。**
- `signals_only`子句：不强调对象状态条件，强调**满足前置条件**时抛出相应的异常。

有时，为了更明显地区分异常，会针对输入参数的取值范围抛出不同的异常，从而提醒调用者进行不同的处理。这时可以使用多个`exceptional_behavior`：

```
public class Student {
    //@ public model non_null int[] credits;//从规格角度定义规格数据int[] credits
     /*@ normal_behavior //正常功能规格
      @ requires z >=0 && z <= 100;
      @ assignable \nothing;
      @ ensures \result == credits.length;
      @ also //进行分隔
      @ exceptional_behavior //异常功能规格
      @ requires z < 0;
      @ assignable \nothing;
      @ signals_only IllegalArgumentException;
      @ also
      @ exceptional_behavior //异常功能规格
      @ requires z > 100;
      @ assignable \nothing;
      @ signals_only OverFlowException;
      @*/
public int recordCredit(int z) throws IllegalArgumentException,
OverFlowException;
}
```

- 三个功能的`requires`子句合在一起覆盖了方法输入参数的所有取值范围，并且没有交叉，这是功能规格设计的基本要求。
- 两个异常功能规格使用`signals_only`子句分别抛出相应的异常。
- 在异常功能规格中，除了抛出异常，也一样可以正常使用`ensures`子句来描述方法执行的其他结果。

四、类型规格

针对`JAVA`程序中**定义的数据类型**所设计的限制规则，一般来说是针对**类或接口**设计的**约束规则。**

从面向对象角度来看，类或接口包含**数据成员和方法成员**的声明及（或）实现。不失一般性，一个类型的成员要么是**静态成员**（static member），要么是**实例成员**（instance member）。一个类的静态方法不可以访问这个类的非静态成员变量（即实例变量）。静态成员可以直接**通过类型引用**，而实例成员只能**通过实例化对象来引用。**因此，在设计和表示类型规格时需要加以区分。

JML针对类型规格定义了多种限制规则，课程主要涉及两类：**不变式限制和约束限制。**无论哪一种，类型规格都是针对类型中定义的数据成员所定义的显示规则，一旦违反限制规则，就称相应的状态有错。

1. 不变式限制——invariant

要求在所有**可见状态**下都必须满足的特性，语法上定义为`invariant P`。其中`invariant`为关键词，`P`为谓词。

可见状态：特定时刻下对一个对象状态的观察。下面是几个实例：

- 对象的有状态构造方法（用来初始化对象成员变量初值）的执行**结束**时刻
- 在调用一个对象回收方法（`finilize`方法）来释放相关资源**开始**的时刻
- 在调用对象`o`的非静态、有状态方法（non-helper）的开始和结束时刻
- 在调用对象`o`对应的类或父类的静态、有状态方法的开始和结束时刻
- 在未处于对象`o`对应类或者父类的静态方法被调用过程中的任意时刻

在以上几种时刻对象`o`的状态都是**可见状态**。

光是看这几个说法很可能一脸懵（我就是），省流版本直接看下面：

凡是会修改成员变量（包括静态成员变量和非静态成员变量）的方法执行期间，对象的状态都是不可见状态（其本质原因是对象的状态修改未完成，此时观察到的状态可能不完整）。

注意：此处的"可见"指的是**完整可见**，而非一般意义上的可以看到！

类型规格强调在任意可见状态下都要满足不变式，见下例：

```
public class Path{
    private /*@spec_public@*/ ArrayList <Integer> seq_nodes;
    private /*@spec_public@*/ Integer start_node;
    private /*@spec_public@*/ Integer end_node;
    /*@ invariant seq_nodes != null &&
      @ seq_nodes[0] == start_node &&
      @ seq_nodes[seq_nodes.legnth-1] == end_node &&
      @ seq_nodes.length >=2;
      @*/
}
```

上面的例子中，`Path`类的不变式定义了`seq_nodes`不能为`null`，且任何一个`Path`对象至少包含两个节点，一个起始节点（`start_node`）和一个终止节点（`end_node`）。

几个值得注意的点：

- 一个类可以**包括多个不变式**，相互独立。
- 如果一个对象的可见状态不满足不变式，那么称**该对象的状态有错。**
- 如果一个类有两个产生逻辑矛盾的不变式（不可能同时为真），则出现了**规格设计缺陷。**
- 不变式中可以直接引用`pure`状态方法。

由于类成员变量可分为**静态和非静态**，JML也区分两类不变式：**静态不变式和实例不变式**（非静态）。

1. 静态不变式：只针对类中的静态成员变量的取值进行约束。
2. 实例不变式：针对静态成员变量和非静态成员变量的取值进行约束。
3. 可以在不变式定义中使用`instance invariant`或者`static invariant`表示两种不变式。

2. 约束限制——constraint

对象的状态发生变化时也需要满足一定的约束，这种约束本质上也是一种不变式（我的理解为动态约束，有别于不变式限制的静态约束）。JML规定：`invariant`只针对**当下可见状态**的取值进行约束，使用`constraint`对**前序可见状态和当前可见状态的关系**进行约束。

```
public class ServiceCounter{
    private /*@spec_public@*/ long counter;//成员变量counter
    //@ invariant counter >= 0;//不变式
    //@ constraint counter == \old(counter)+1;//状态约束限制
}
```

- 本例中，状态约束限制每一次修改`counter`的值只能加1。
- 加入状态约束限制可以保证所有可能修改`counter`的方法都满足该“后置条件”，大大简便了书写。

此外，`invariant`和`constraint`还可以直接被子类继承获得。

和`invariant`类似，根据类的静态成员变量，可以将约束限制分为两种，详见手册。

3. 方法与类型规格的关系

这块和我们的要求关系不大，故略去。（向助教证实过）