Optional, Objects, Preconditions, Ordering

**null**

**null代表不确定的对象：**

Java中,null是一个关键字,用来标识一个不确定的对象, 因此可以将null赋给引用类型变量.

**null本身不是对象，也不是Objcet的实例.**

* List：允许重复元素，可以加入任意多个null。
* Set：不允许重复元素，最多可以加入一个null。
* Map：Map的key最多可以加入一个null，value字段没有限制。
* 数组：基本类型数组,定义后,如果不给定初始值,则java运行时会自动给定值(整数类型的int、byte、short、long的自动赋值为0，带小数点的float、double自动赋值为0.0，boolean的自动赋值为false)。引用类型数组，不给定初始值，则所有的元素值为null。

**Optional (判断引用是否缺失)**

Optional<T>表示可能为null的T类型引用。一个Optional实例可能包含非null的引用（我们称之为引用存在），也可能什么也不包括（称之为引用缺失）。它从不说包含的是null值，而是用存在或缺失来表示。但Optional从不会包含null值引用。

**创建Optional实例（以下都是静态方法）：**

|  |  |
| --- | --- |
| Optional.of(T) | 创建指定引用的Optional实例，若引用为null则快速失败 |
| Optional.absent() | 创建引用缺失的Optional实例 |
| Optional.fromNullable(T) | 创建指定引用的Optional实例，若引用为null则表示缺失 |

**用Optional实例查询引用（以下都是非静态方法）：**

|  |  |
| --- | --- |
| Optional.isPresent() | 如果Optional包含非null的引用（引用存在），返回true |
| T  get() | 返回Optional所包含的引用，若引用缺失，则抛出java.lang.IllegalStateException |
| T  or(T) | 返回Optional所包含的引用，若引用缺失，返回指定的值 |
| T  orNull() | 返回Optional所包含的引用，若引用缺失，返回null |
| Set<T>  asSet() | 返回Optional所包含引用的单例不可变集，如果引用存在，返回一个只有单一元素的集合，如果引用缺失，返回一个空集合。 |

**实例 :**

1. **可用于方法间参数传递非null校验的快速失败**

public void dealNewsSend(HttpServletRequest request, String companyCode){

//创建不允许null值的Optional,如果companyCode传过来的是null, 则快速失败

Optional<String> possible1 = Optional.*of*(companyCode);

}

1. Integer integer = null;

Optional<Integer> possible = Optional.*fromNullable*(integer); //创建允许null值的Optional

if(possible.isPresent()){ // false

System.*out*.println(possible.get());

}

System.*out*.println(possible.or(3)); // 3; or:如果possible为null, 则执行or括号内的语句

System.*out*.println(possible.orNull()); // null

System.*out*.println(possible.asSet()); // []

1. Integer integer = 5;

Optional<Integer> possible = Optional.*fromNullable*(integer); //创建允许null值的Optional

if(possible.isPresent()){

System.*out*.println(possible.get()); // 5

}

System.*out*.println(possible.or(3)); // 5

System.*out*.println(possible.orNull()); // 5

System.*out*.println(possible.asSet()); // [5]

**Objects.equal (String,Integer等类型的比较)**

使用Objects.equal(obj, obj)避免抛出NullPointerException。例如:

[查](http://ifeve.com/google-guava-commonobjectutilities/#viewSource)Objects.equal("a", "a");  // returns true

Objects.equal(**null**, "a");  // returns false

Objects.equal("a", **null**);  // returns false

Objects.equal(**null**, **null**);  // returns true

**Preconditions (前置条件检查)**

|  |  |  |
| --- | --- | --- |
| **方法声明(不包括额外参数)** | **描述** | **检查失败时抛出的异常** |
| checkArgument(boolean) | 检查boolean是否为true，用来检查传递给方法的参数。 | IllegalArgumentException |
| checkNotNull(T) | 检查value是否为null，该方法直接返回value，因此可以内嵌使用checkNotNull。 | NullPointerException |
| checkState(boolean) | 用来检查对象的某些状态。 | IllegalStateException |
| checkElementIndex(int index, int size) | 检查index作为索引值对某个列表、字符串或数组是否有效。index>=0 && index<size \* | IndexOutOfBoundsException |
| checkPositionIndex(int index, int size) | 检查index作为位置值对某个列表、字符串或数组是否有效。index>=0 && index<=size \* | IndexOutOfBoundsException |
| checkPositionIndexes(int start, int end, int size) | 检查[start, end]表示的位置范围对某个列表、字符串或数组是否有效\* | IndexOutOfBoundsException |

*\*索引值常用来查找列表、字符串或数组中的元素，如List.get(int), String.charAt(int)*

*\*位置值和位置范围常用来截取列表、字符串或数组，如List.subList(int，int), String.substring(int)*

例如 :

1. checkArgument(boolean)

int i = -1;

Preconditions.*checkArgument*(i >= 0, "Argument was %s but expected nonnegative", i);

打印: Exception in thread "main" java.lang.IllegalArgumentException: Argument was -1 but expected nonnegative

2. checkNotNull(T)

Integer i = null;

Preconditions.*checkNotNull*(i, "i为null");

打印:Exception in thread "main" java.lang.NullPointerException: i为null

Integer i = 11;

int ii = Preconditions.*checkNotNull*(11); // ii=11

**Ordering(排序)**

|  |  |
| --- | --- |
| **创建排序器Ordering** | |
| natural() | 使用Comparable类型的自然顺序， 例如：整数从小到大，字符串是按字典顺序; |
| usingToString() | 使用toString()返回的字符串按字典顺序进行排序； |
| arbitrary() | 返回一个所有对象的任意顺序 |
| lexicographical() | 返回一个按照字典元素迭代的Ordering； |
| from(Comparator) | 把给定的Comparator转化第一元素为排序器 |
| compound(Comparator) | 把给定的Comparator转化为第二元素排序器(成绩排序先按分数(from)排,后按学号(compound)排) |
| **操作方法** | |
| sortedCopy(Iterable) | 返回指定的元素作为一个列表的排序副本。 |
| reverse() | 返回与当前Ordering相反的排序. |
| nullsFirst() | 返回一个将null放在non-null元素之前的Ordering，其他的和原始的Ordering一样； |
| nullsLast() | 返回一个将null放在non-null元素之后的Ordering，其他的和原始的Ordering一样； |
| isOrdered(Iterable) | 是否有序，Iterable不能少于2个元素。 |
| max(Iterable) | 返回Iterable的最大值 |
| min(Iterable) | 返回Iterable的最小值 |
| greatestOf(Iterable iterable, int k) | 获取可迭代对象中最大的k个元素 |
| onResultOf(Function) | 对集合中元素调用Function，再按返回值用当前排序器排序 |

例如:

Ordering<String> natural = Ordering.natural();

List<String> list = ImmutableList.of("b", "c", "a", "d", "f");

System.out.println(natural.sortedCopy(list));

//~out: [a, b, c, d, f]

System.out.println(natural.isOrdered(list));

//~out: false

System.out.println(natural.max(list));

//~out: f

System.out.println(natural.min(list));

//~out: a

System.out.println(natural.reverse().sortedCopy(list));

//~out: [f, d, c, b, a]

 // 多参数排序,先按cityByPopluation排序,再按cityByRainfall排序

// 类似于sql的 order by cityByPopluation asc, cityByRainfall asc;

List<City> cities = Lists.newArrayList(city1, city2, city3);

Ordering<City> secondaryOrdering = Ordering.from(cityByPopluation).compound(cityByRainfall);

Collections.sort(cities, secondaryOrdering);

// 单参数排序,按cityByPopluation排序

// 获取人口最多的两个城市

List<City> topTwoPop = Ordering.from(cityByPopluation).greatestOf(cities, 2);

//人口比较器

**class** CityByPopluation **implements** Comparator<City> {

@Override

**public** **int** compare(City city1, City city2) {

**return** Ints.compare(city1.getPopulation(), city2.getPopulation());

}

}

//降雨量比较器

**class** CityByRainfall **implements** Comparator<City> {

@Override

**public** **int** compare(City city1, City city2) {

**return** Doubles.compare(city1.getAverageRainfall(), city2.getAverageRainfall());

}

}

Ordering<Foo> ordering = Ordering.natural().nullsFirst().

onResultOf(**new** Function<Foo, String>() {

**public** String apply(Foo foo) {

**return** foo.sortedBy;

}

});

当阅读链式调用产生的排序器时，应该从后往前读(*compound*方法除外)。上面的例子中，排序器首先调用apply方法获取sortedBy值，并把sortedBy为null的元素都放到最前面，然后把剩下的元素按sortedBy进行自然排序。之所以要从后往前读，是因为每次链式调用都是用后面的方法包装了前面的排序器。