

www.cnblogs.com

CarpenterLee

技术只是工具，重要的是人才！

--更多内容欢迎访问--

[博主github主页](#)昵称：[CarpenterLee](#)

园龄：3年

粉丝：403

关注：2

[+加关注](#)

< 2019年5月 >						
日	一	二	三	四	五	六
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

最新随笔



1. [Java线程池详解](#)
2. [Java Proxy和CGLIB动态代理原理](#)
3. [Nginx限速模块初探](#)
4. [200行Java代码搞定计算器程序](#)
5. [深入理解Java内置锁和显式锁](#)
6. [使用linux perf工具生成java程序火焰图](#)
7. [《深入理解Java函数式编程》系列文章](#)
8. [Java原子变量](#)
9. [Java Stream API性能测试](#)
10. [深入理解Java Stream流水线](#)

随笔分类

[博客园](#) [首页](#) [新随笔](#) [管理](#)

随笔-28 评论-186 文章-0

Lambda表达式和Java集合框架

[本文github地址](#)

Java8为容器新增一些有用的方法，这些方法有些是为完善原有功能，有些是为引入函数式编程（Lambda表达式），学习和使用这些方法有助于我们写出更加简洁有效的代码。本文分别以ArrayList和HashMap为例，讲解Java8集合框架（Java Collections Framework）中新加入方法的使用。

前言

我们先从最熟悉的Java集合框架(Java Collections Framework, JCF)开始说起。

为引入Lambda表达式，Java8新增了 `java.util.function` 包，里面包含常用的函数接口，这是Lambda表达式的基础，Java集合框架也新增部分接口，以便与Lambda表达式对接。

首先回顾一下Java集合框架的接口继承结构：

[深入理解Java函数式编程\(8\)](#)
[深入理解Java集合框架\(11\)](#)

最新评论



1. [Re:深入理解Java Stream流水线学习了](#)

--竹马今安在
2. [Re:史上最清晰的红黑树讲解 \(上\)](#)

43个赞, 已顶
一步一步解释的真清晰
--陈...大...海
3. [Re:史上最清晰的红黑树讲解 \(下\)](#)

图1明显不是红黑树, 明显违反了性质: 从任一节点到叶子节点(null节点)的所有路径都包含相同数目的BLACK节点。15→12→null路径上, 只有2个BLACK节点。其他路径: 15→12→5→null.....
--M了个J
4. [Re:史上最清晰的红黑树讲解 \(下\)](#)

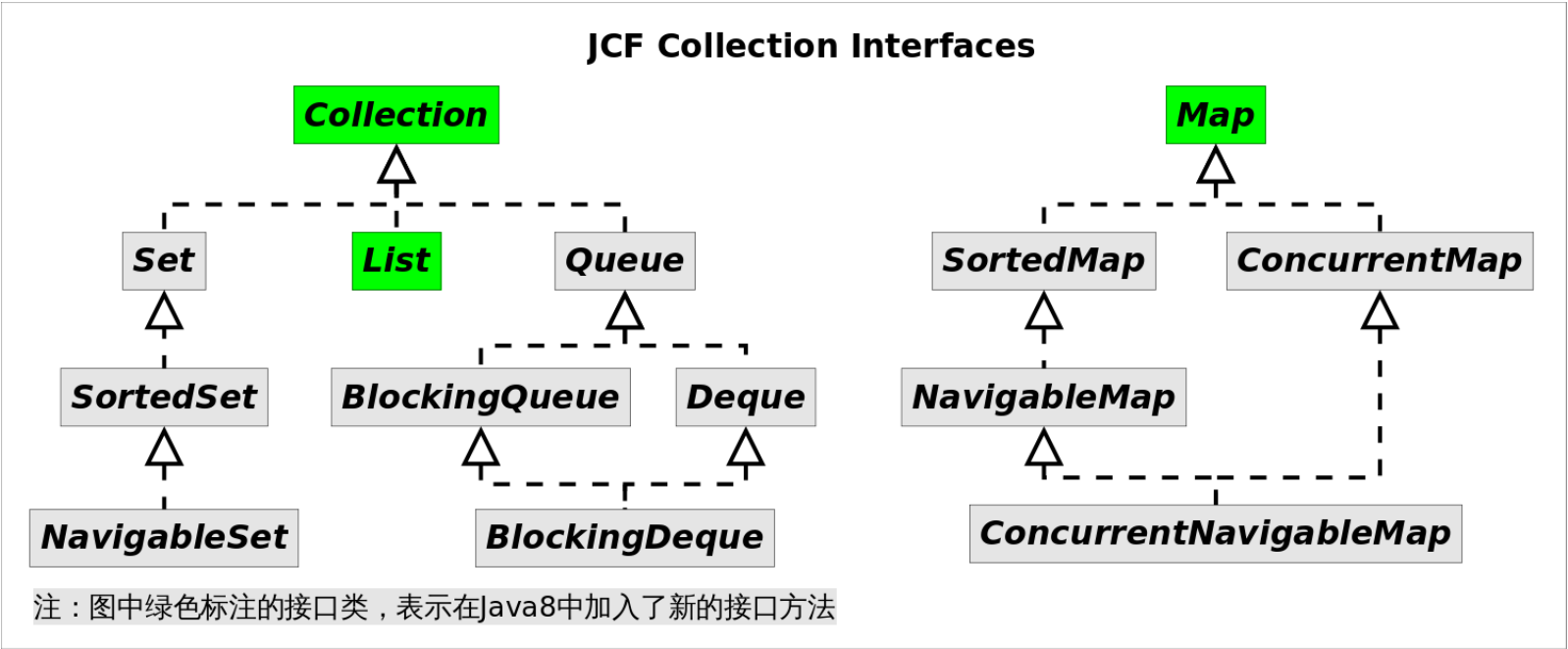
@蜗牛大师我也觉得第一个图不是红黑树。。。红黑树的叶子结点好像指的是空结点吧, 值为12的那个结点右子树为空那它的左子树只能为一个红色结点吧???...
--hthhpeng
5. [Re:史上最清晰的红黑树讲解 \(上\)](#)

有点不理解的是, 在步骤(b)的时候, 直接把节点p着色成黑色, 不就满足红黑树条件了吗? 为什么还要继续下面的更多操作?
--Stack Over Flow

阅读排行榜



1. [史上最清晰的红黑树讲解 \(上\)](#)
(117698)
2. [Java Stream API进阶篇\(34799\)](#)
3. [深入理解Java PriorityQueue\(33283\)](#)



上图中绿色标注的接口类，表示在Java8中加入了新的接口方法，当然由于继承关系，他们相应的子类也都会继承这些新方法。下表详细列举了这些方法。

接口名	Java8新加入的方法
Collection	removeIf() spliterator() stream() parallelStream() forEach()
List	replaceAll() sort()
Map	getOrDefault() forEach() replaceAll() putIfAbsent() remove() replace() computeIfAbsent() computeIfPresent() compute() merge()

这些新加入的方法大部分都要用到 `java.util.function` 包下的接口，这意味着这些方法大部分都跟Lambda表达式相关。我们将逐一学习这些方法。

[4. C语言编译过程详解\(30722\)](#)[5. Java Stream API入门篇\(24226\)](#)

Collection中的新方法

如上所示，接口 `Collection` 和 `List` 新加入了一些方法，我们以是 `List` 的子类 `ArrayList` 为例来说明。了解[Java7](#) `ArrayList` [实现原理](#)，将有助于理解下文。

forEach()

该方法的签名为 `void forEach(Consumer<? super E> action)`，作用是对容器中的每个元素执行 `action` 指定的动作，其中 `Consumer` 是个函数接口，里面只有一个待实现方法 `void accept(T t)`（后面我们会看到，这个方法叫什么根本不重要，你甚至不需要记忆它的名字）。

需求：假设有一个字符串列表，需要打印出其中所有长度大于3的字符串。

Java7及以前我们可以用增强的for循环实现：

```
// 使用增强for循环迭代
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
for(String str : list){
    if(str.length()>3)
        System.out.println(str);
}
```

现在使用 `forEach()` 方法结合匿名内部类，可以这样实现：

```
// 使用forEach()结合匿名内部类迭代
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.forEach(new Consumer<String>(){
    @Override
    public void accept(String str){
        if(str.length()>3)
            System.out.println(str);
    }
});
```

上述代码调用 `forEach()` 方法，并使用匿名内部类实现 `Consumer` 接口。到目前为止我们没看到这种设计有什么好处，但是不要忘记Lambda表达式，使用Lambda表达式实现如下：

```
// 使用forEach()结合Lambda表达式迭代
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.forEach( str -> {
    if(str.length()>3)
        System.out.println(str);
});
```

上述代码给 `forEach()` 方法传入一个Lambda表达式，我们不需要知道 `accept()` 方法，也不需要知道 `Consumer` 接口，类型推导帮我们做了一切。

removeIf()

该方法签名为 `boolean removeIf(Predicate<? super E> filter)`，作用是删除容器中所有满足 `filter` 指定条件的元素，其中 `Predicate` 是一个函数接口，里面只有一个待实现方法 `boolean test(T t)`，同样的这个方法的名字根本不重要，因为用的时候不需要书写这个名字。

需求：假设有一个字符串列表，需要删除其中所有长度大于3的字符串。

我们知道如果需要在迭代过程中对容器进行删除操作必须使用迭代器，否则会抛出 `ConcurrentModificationException`，所以上述任务传统的写法是：

```
// 使用迭代器删除列表元素
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
Iterator<String> it = list.iterator();
while(it.hasNext()){
    if(it.next().length()>3) // 删除长度大于3的元素
        it.remove();
}
```

现在使用 `removeIf()` 方法结合匿名内部类，我们可是这样实现：

```
// 使用removeIf()结合匿名内部类实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.removeIf(new Predicate<String>(){ // 删除长度大于3的元素
    @Override
    public boolean test(String str){
        return str.length()>3;
    }
});
```

上述代码使用 `removeIf()` 方法，并使用匿名内部类实现 `Predicate` 接口。相信你已经想到用Lambda表达式该怎么写了：

```
// 使用removeIf()结合Lambda表达式实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.removeIf(str -> str.length()>3); // 删除长度大于3的元素
```

使用Lambda表达式不需要记忆 `Predicate` 接口名，也不需要记忆 `test()` 方法名，只需要知道此处需要一个返回布尔类型的Lambda表达式就行了。

replaceAll()

该方法签名为 `void replaceAll(UnaryOperator<E> operator)`，作用是**对每个元素执行** `operator` 指定的操作，并用操作结果来替换原来的元素。其中 `UnaryOperator` 是一个函数接口，里面只有一个待实现函数 `T apply(T t)`。

需求：假设有一个字符串列表，将其中所有长度大于3的元素转换成大写，其余元素不变。

Java7及之前似乎没有优雅的办法：

```
// 使用下标实现元素替换
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
for(int i=0; i<list.size(); i++){
    String str = list.get(i);
    if(str.length()>3)
        list.set(i, str.toUpperCase());
}
```

使用 `replaceAll()` 方法结合匿名内部类可以实现如下：

```
// 使用匿名内部类实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.replaceAll(new UnaryOperator<String>(){
    @Override
    public String apply(String str){
        if(str.length()>3)
            return str.toUpperCase();
        return str;
    }
});
```

上述代码调用 `replaceAll()` 方法，并使用匿名内部类实现 `UnaryOperator` 接口。我们知道可以用更为简洁的Lambda表达式实现：

```
// 使用Lambda表达式实现
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.replaceAll(str -> {
    if(str.length()>3)
        return str.toUpperCase();
    return str;
});
```

sort()

该方法定义在 `List` 接口中，方法签名为 `void sort(Comparator<? super E> c)`，该方法根据 `c` 指定的比较规则对容器元素进行排序。

`Comparator` 接口我们并不陌生，其中有一个方法 `int compare(T o1, T o2)` 需要实现，显然该接口是个函数接口。

需求：假设有一个字符串列表，按照字符串长度增序对元素排序。

由于Java7以及之前 `sort()` 方法在 `Collections` 工具类中，所以代码要这样写：

```
// Collections.sort()方法
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
Collections.sort(list, new Comparator<String>() {
    @Override
    public int compare(String str1, String str2) {
        return str1.length()-str2.length();
    }
});
```

现在可以直接使用 `List.sort()` 方法，结合Lambda表达式，可以这样写：

```
// List.sort()方法结合Lambda表达式
ArrayList<String> list = new ArrayList<>(Arrays.asList("I", "love", "you", "too"));
list.sort((str1, str2) -> str1.length()-str2.length());
```

splitator()

方法签名为 `Splititerator<E> splititerator()`，该方法返回容器的可拆分迭代器。从名字来看该方法跟 `iterator()` 方法有点像，我们知道 `Iterator` 是用来迭代容器的，`Splititerator` 也有类似作用，但二者有如下不同：

1. `Splititerator` 既可以像 `Iterator` 那样逐个迭代，也可以批量迭代。批量迭代可以降低迭代的开销。
2. `Splititerator` 是可拆分的，一个 `Splititerator` 可以通过调用 `Splititerator<T> trySplit()` 方法来尝试分成两个。一个是 `this`，另一个是新返回的那个，这两个迭代器代表的元素没有重叠。

可通过（多次）调用 `Splititerator.trySplit()` 方法来分解负载，以便多线程处理。

stream()和parallelStream()

`stream()` 和 `parallelStream()` 分别返回该容器的 `Stream` 视图表示，不同之处在于 `parallelStream()` 返回并行的 `Stream`。
`Stream` 是Java函数式编程的核心类，我们会在后面章节中学习。

Map中的新方法

相比 `Collection`，`Map` 中加入了更多的方法，我们以 `HashMap` 为例来逐一探秘。了解Java7 `HashMap` 实现原理，将有助于理解下文。

forEach()

该方法签名为 `void forEach(BiConsumer<? super K, ? super V> action)`，作用是对 `Map` 中的每个映射执行 `action` 指定的操作，其中 `BiConsumer` 是一个函数接口，里面有一个待实现方法 `void accept(T t, U u)`。`BiConsumer` 接口名字和 `accept()` 方法名字都不重要，请不要记忆他们。

需求：假设有一个数字到对应英文单词的Map，请输出Map中的所有映射关系。

Java7以及之前经典的代码如下：

```
// Java7以及之前迭代Map
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + "=" + entry.getValue());
}
```

使用 `Map.forEach()` 方法，结合匿名内部类，代码如下：

```
// 使用forEach()结合匿名内部类迭代Map
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.forEach(new BiConsumer<Integer, String>() {
    @Override
    public void accept(Integer k, String v) {
        System.out.println(k + "=" + v);
    }
});
```

上述代码调用 `forEach()` 方法，并使用匿名内部类实现 `BiConsumer` 接口。当然，实际场景中没人使用匿名内部类写法，因为有Lambda表达式：

```
// 使用forEach()结合Lambda表达式迭代Map
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.forEach((k, v) -> System.out.println(k + "=" + v));
}
```

getOrDefault()

该方法跟Lambda表达式没关系，但是很有用。方法签名为 `V getOrDefault(Object key, V defaultValue)`，作用是按照给定的 `key` 查询

`Map` 中对应的 `value`，如果没有找到则返回 `defaultValue`。使用该方法的程序员可以省去查询指定键值是否存在的麻烦。

需求：假设有一个数字到对应英文单词的Map，输出4对应的英文单词，如果不存在则输出NoValue

```
// 查询Map中指定的值，不存在时使用默认值
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
// Java7以及之前做法
if (map.containsKey(4)) { // 1
    System.out.println(map.get(4));
} else {
    System.out.println("NoValue");
}
// Java8使用Map.getOrDefault()
System.out.println(map.getOrDefault(4, "NoValue")); // 2
```

putIfAbsent()

该方法跟Lambda表达式没关系，但是很有用。方法签名为 `V putIfAbsent(K key, V value)`，作用是只有在不存在 `key` 值的映射或映射值为 `null` 时，才将 `value` 指定的值放入到 `Map` 中，否则不对 `Map` 做更改。该方法将条件判断和赋值合二为一，使用起来更加方便。

remove()

我们都知道 `Map` 中有一个 `remove(Object key)` 方法，来根据指定 `key` 值删除 `Map` 中的映射关系；Java8新增了 `remove(Object key, Object value)` 方法，只有在当前 `Map` 中 `key` 正好映射到 `value` 时才删除该映射，否则什么也不做。

replace()

在Java7及以前，要想替换 `Map` 中的映射关系可通过 `put(K key, V value)` 方法实现，该方法总是会用新值替换原来的值。为了更精确的控制替换行为，Java8在 `Map` 中加入了两个 `replace()` 方法，分别如下：

- `replace(K key, V value)`，只有在当前 `Map` 中 `key` 的映射存在时才用 `value` 去替换原来的值，否则什么也不做。
- `replace(K key, V oldValue, V newValue)`，只有在当前 `Map` 中 `key` 的映射存在且等于 `oldValue` 时才用 `newValue` 去替换原来的值，否则什么也不做。

replaceAll()

该方法签名为 `replaceAll(BiFunction<? super K, ? super V, ? extends V> function)`，作用是对 `Map` 中的每个映射执行 `function` 指定的操作，并用 `function` 的执行结果替换原来的 `value`，其中 `BiFunction` 是一个函数接口，里面有一个待实现方法 `R apply(T t, U u)`。不要被如此多的函数接口吓到，因为使用的时候根本不需要知道他们的名字。

需求：假设有一个数字到对应英文单词的Map，请将原来映射关系中的单词都转换成大写。

Java7以及之前经典的代码如下：


```
// Java7以及之前替换所有Map中所有映射关系
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    entry.setValue(entry.getValue().toUpperCase());
}
```

使用 `replaceAll()` 方法结合匿名内部类，实现如下：

```
// 使用replaceAll()结合匿名内部类实现
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.replaceAll(new BiFunction<Integer, String, String>() {
    @Override
    public String apply(Integer k, String v) {
        return v.toUpperCase();
    }
});
```

上述代码调用 `replaceAll()` 方法，并使用匿名内部类实现 `BiFunction` 接口。更进一步的，使用Lambda表达式实现如下：

```
// 使用replaceAll()结合Lambda表达式实现
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.replaceAll((k, v) -> v.toUpperCase());
```

简洁到让人难以置信。

merge()

该方法签名为 `merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)`，作用是：

1. 如果 `Map` 中 `key` 对应的映射不存在或者为 `null`，则将 `value`（不能是 `null`）关联到 `key` 上；
2. 否则执行 `remappingFunction`，如果执行结果非 `null` 则用该结果跟 `key` 关联，否则在 `Map` 中删除 `key` 的映射。

参数中 `BiFunction` 函数接口前面已经介绍过，里面有一个待实现方法 `R apply(T t, U u)`。

`merge()` 方法虽然语义有些复杂，但该方法的使用方式很明确，一个比较常见的场景是将新的错误信息拼接到原来的信息上，比如：

```
map.merge(key, newMsg, (v1, v2) -> v1+v2);
```

compute()

该方法签名为 `compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`，作用是把 `remappingFunction` 的计算结果关联到 `key` 上，如果计算结果为 `null`，则在 `Map` 中删除 `key` 的映射。

要实现上述 `merge()` 方法中错误信息拼接的例子，使用 `compute()` 代码如下：

```
map.compute(key, (k,v) -> v==null ? newMsg : v.concat(newMsg));
```

computeIfAbsent()

该方法签名为 `V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)`，作用是：只有在当前 `Map` 中不存在 `key` 值的映射或映射值为 `null` 时，才调用 `mappingFunction`，并在 `mappingFunction` 执行结果非 `null` 时，将结果跟 `key` 关联。

`Function` 是一个函数接口，里面有一个待实现方法 `R apply(T t)`。

`computeIfAbsent()` 常用来对 `Map` 的某个 `key` 值建立初始化映射。比如我们要实现一个多值映射，`Map` 的定义可能是 `Map<K, Set<V>>`，要向 `Map` 中放入新值，可通过如下代码实现：

```
Map<Integer, Set<String>> map = new HashMap<>();
// Java7及以前的实现方式
if (map.containsKey(1)) {
    map.get(1).add("one");
} else {
    Set<String> valueSet = new HashSet<String>();
    valueSet.add("one");
    map.put(1, valueSet);
}
// Java8的实现方式
map.computeIfAbsent(1, v -> new HashSet<String>()).add("yi");
```

使用 `computeIfAbsent()` 将条件判断和添加操作合二为一，使代码更加简洁。

computeIfPresent()

该方法签名为 `V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`，作用跟 `computeIfAbsent()` 相反，即，只有在当前 `Map` 中存在 `key` 值的映射且非 `null` 时，才调用 `remappingFunction`，如果 `remappingFunction` 执行结果为 `null`，则删除 `key` 的映射，否则使用该结果替换 `key` 原来的映射。

这个函数的功能跟如下代码是等效的：

```
// Java7及以前跟computeIfPresent()等效的代码
if (map.get(key) != null) {
    V oldValue = map.get(key);
    V newValue = remappingFunction.apply(key, oldValue);
    if (newValue != null)
        map.put(key, newValue);
    else
        map.remove(key);
    return newValue;
}
return null;
```

总结

1. Java8为容器新增一些有用的方法，这些方法有些是为**完善原有功能**，有些是为**引入函数式编程**，学习和使用这些方法有助于我们写出更加简洁有效的代码。
2. **函数接口**虽然很多，但绝大多数时候我们根本不需要知道它们的名字，书写Lambda表达式时类型推断帮我们做了一切。

[本文github地址](#)

分类: [深入理解Java函数式编程](#)

标签: [Lambda](#), [functional programming](#), [Java8](#), [JCF](#)

好文要顶

关注我

收藏该文



CarpenterLee

关注 - 2

粉丝 - 403

[+加关注](#)

3

推荐

0

反对

« 上一篇: [C语言编译过程详解](#)


» 下一篇: [Java Stream API入门篇](#)

posted on 2017-03-06 07:03 [CarpenterLee](#) 阅读(14405) 评论(4) [编辑](#) [收藏](#)


评论:

#1楼 2017-03-07 09:49 | [selfImpr689](#)


赞一个。

[支持\(0\)](#) [反对\(0\)](#)#2楼 2018-04-03 14:30 | [乔胖胖](#) 


有收获

[支持\(0\)](#) [反对\(0\)](#)#3楼 2018-05-31 10:28 | [Hance](#) 

有帮助！谢谢！！

[支持\(0\)](#) [反对\(0\)](#)#4楼 2018-08-16 11:43 | [moon0521](#) 

好理解，用起来方便

[支持\(0\)](#) [反对\(0\)](#)[刷新评论](#) [刷新页面](#) [返回顶部](#) 注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。[【推荐】超50万C++/C#源码：大型实时仿真组态图形源码](#)[【活动】看雪2019安全开发者峰会，共话安全领域焦点](#)[【培训】Java程序员年薪40W，他1年走了别人5年的路](#)

百度智能云

云生态狂欢季

云服务器

148元/首年

立即购买

相关博文：

- [Lambda表达式和Java集合框架](#)
- [Java Lambda表达式初探](#)
- [Java Lambda表达式初探](#)
- [JavaLambda表达式初探](#)
- [Java Lambda表达式初探](#)



最新新闻：

- [微软是一家养老公司？微软副总裁洪小文回应：员工能待20年我们很自豪](#)
 - [瓜子二手车成立物流科技公司 注册资本5000万元](#)
 - [曝光超级通讯员：癌细胞可以在体内进行远距离交流](#)
 - [B站财报电话会议实录：虚拟直播营收占直播及增值服务收入40%](#)
 - [“跨界明星”DNA 实力“圈粉”材料科学家](#)
- » [更多新闻...](#)

Powered by: [博客园](#) 模板提供: [沪江博客](#) Copyright ©2019 CarpenterLee