64、集合的公共操作

（0）公共操作可用于只读集合与可变集合。常见操作分为以下几类：

- 参考地址：https://www.kotlincn.net/docs/reference/collection-grouping.html

（1） 集合转换

（1-1）映射转换(mapxxx)

- 从**一个类型的集合**转换为**另一个类型的集合**（前后集合中元素的类型可以相同，可以不同，可以是基本类型，包装类型，对象类型的任意一种）。map，mapIndexed，mapNotNull，mapIndexedNotNull，mapKeys，mapValues等函数。

```
#Iterator做映射时，将一个Iterator集合转换为另一个Iterator集合
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })//idx是元素索引

#为了取代map和mapIndexed函数，mapNotNull从结果集中过滤掉null值，即先map再过滤掉null值。（mapIndexedNotNull是通过索引进行map）
val numbers = setOf(1, 2, 3)
println(numbers.mapNotNull { if (it == 2) null else it * 3 })
println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })

#map函数进行集合映射时，只转换key使用mapKeys函数，只转换value使用mapValues函数。
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
println(numbersMap.mapKeys { it.key.toUpperCase() })
println(numbersMap.mapValues { it.value + it.key.length })
{KEY1=1, KEY2=2, KEY3=3, KEY11=11}
{key1=5, key2=6, key3=7, key11=16}
```

笔记目录

```
//map函数进行集合映射，对象类型的List集合映射为另一个对象类型的List集合
data class BloodPressure(
    val systolicBloodPressure: BigDecimal,
    val diastolicBloodPressure: BigDecimal,
    val measureDatetime: LocalDateTime
)
data class BloodPressureValue(
    val systolicBloodPressure: BigDecimal,
    val diastolicBloodPressure: BigDecimal
)

val bloodPressureList: List<BloodPressure>
val valueList =
bloodPressureList.map{BloodPressureValue(it.systolicBloodPressure,it.diastolicBloodPress
ure) }

//map函数进行集合映射，对象类型的集合映射为String类型的集合
val systolicBloodPressureList:  List<BloodPressure>
val valueList: List<String> =  bloodPressureList.map{it.systolicBloodPressure }
```

（1-2）合拢转换（zip和unzip）

- 将两个集合中具有相同下标位置的元素，进行构建配对，通过zip扩展函数完成的。如果两个集合大小不同，则zip函数的结果为较小集合的大小，结果中不包含较大集合的后续元素，zip函数也可以用中缀形式调用，如下：

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
println(colors zip animals)
val twoAnimals = listOf("fox", "bear")
println(colors.zip(twoAnimals))

[(red, fox), (brown, bear), (grey, wolf)]
[(red, fox), (brown, bear)]
```

- 带两个参数的zip函数，第二个参数是lamada表达式

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
println(colors.zip(animals) { color, animal -> "The ${animal.capitalize()} is $color"})

[The Fox is red, The Bear is brown, The Wolf is grey]
```

- 当拥有Pair对象的List集合时，使用反向转换unzip扩展函数完成，从这些键值对中构建两个List集合，第一个集合包含原始List集合中每个Pair的键，第二个集合包含原始List集合中每个Pair的值。

```
val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
println(numberPairs.unzip())
([one, two, three, four], [1, 2, 3, 4])
```

（1-3）***关联转换（associateXXX，List->Map）**

- **associateWith**：基本的关联函数，用来创建一个Map（**原始List集合中的元素默认作为key**，关联函数通过原始List集合来产生value，如果List集合中存在多个元素相等，则仅最后一个保留在Map中）

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
{one=3, two=3, three=5, four=4}

val numbers = listOf("one", "one", "three", "four")
println(numbers.associateWith { it.length })
{one=3, three=5, four=4}
```

- associateBy
  - 基本的关联函数，用来创建一个Map（**原始List集合中的元素默认作为value，**关联函数通过原始List集合来产生key）

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateBy { it.first().toUpperCase() })

{O=one, T=three, F=four}
```

  - Map的value也可以通过值转换函数valueTransform来产生

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateBy(keySelector = { it.first().toUpperCase() }, valueTransform = { it.length }))

{O=3, T=5, F=4}
```

- associate

- 此函数用来创建一个Map。**Map的键和值都是通过集合元素生成的**，associate会临时生成Pair对象，这可能影响性能，**性能不是关键时应使用associate**。

```
//案例1
data class FullName (val firstName: String, val lastName: String)

fun parseFullName(fullName: String): FullName {
    val nameParts = fullName.split(" ")
    if (nameParts.size == 2) {
        return FullName(nameParts[0], nameParts[1])
    } else throw Exception("Wrong name format")
}

val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
println(names.associate { name -> parseFullName(name).let { it.lastName to it.firstName } })

{Adams=Alice, Brown=Brian, Campbell=Clara}

//案例2
data class EvaluateResultQuestionsOption(
    @field:Valid
    @field:JsonProperty("questionsId", required = true) val questionsId: java.math.BigInteger,
    @field:Valid
    @field:JsonProperty("optionId", required = true) val optionId: java.math.BigInteger,
    @field:JsonProperty("optionValue", required = true) val optionValue: kotlin.String,
    @field:JsonProperty("optionCode") val optionCode: kotlin.String? = null,
    @field:JsonProperty("optionScore") val optionScore: java.math.BigDecimal? = null
)
val optionList: List<EvaluateResultQuestionsOption>
val optionMap:Map<String,String> = optionList.associate { it.optionCode to it.optionValue }
```

```
//*****常用案例（高级）
（1）、List转为Map(将optionList集合中每个对象的两个属性optionCode，optionValue，转为Map<optionCode,optionValue>)，
（2）、然后根据预先设定的optionCode（下面），去上面转好的Map中遍历取出对应value。

//List转Map
val optionMap = optionList.associate { it.optionCode to it.optionValue }

//预先设定的身高Code
const val BEHAVIOR_140001_1 = "BEHAVIOR_140001_1"

//取出的身高Value
val height = getOptionValue(BEHAVIOR_140001_1, optionMap)

private fun getOptionValue(optionCode: String, optionValueMap: Map): BigDecimal {
    if (optionValueMap.containsKey(optionCode)) {
        val value = optionValueMap[optionCode]
        return if (value.isNullOrEmpty()) BigDecimal.ZERO else BigDecimal(value)
    }
    return BigDecimal.ZERO
}
```

（1-4）嵌套转换（flatten，flatMap）

- flatten函数：返回嵌套集合中的所有元素的一个List

```
val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))
println(numberSets.flatten())

[1, 2, 3, 4, 5, 6, 1, 2]
```

- flatMap函数：更灵活的处理嵌套的集合，是map和flatten的连续调用。详情请见笔记。

（1-5）字符串转换（joinToString和joinTo）

- joinToString函数：将集合转换为String，默认逗号分割的字符串

```
val numbers = listOf("one", "two", "three", "four")
println(numbers)
println(numbers.joinToString())

[one, two, three, four]
one, two, three, four
```

- joinToString函数：想要自定义带前缀，自定义分隔符，带后缀的字符串，可以使用separator，prefix，postfix参数来指定

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))

start: one | two | three | four: end
```

- joinToString函数：针对较大的集合想要转换为字符串，使用limit参数来限制拼接的字符串中元素数量，超出的元素使用truncate参数指定的值来替换。

```
val numbers = (1..100).toList()
println(numbers.joinToString(limit = 10, truncated = "<...>"))

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, <...>
```

- joinToString函数：想要自定义元素本身的表现形式（大小写转换）,可以使用lamada表达式

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString { "Element: ${it.toUpperCase()}"})

Element: ONE, Element: TWO, Element: THREE, Element: FOUR
```

- joinTo() 函数：执行相同的操作，但将结果附加到给定的 Appendable 对象。

```
val numbers = listOf("one", "two", "three", "four")
val listString = StringBuffer("The list of numbers: ")
numbers.joinTo(listString)
println(listString)

The list of numbers: one, two, three, four
```

（2）集合过滤

（2-1）按谓词过滤：filter(谓词)

- **filter**()过滤函数，只能用来检查元素的值，返回匹配条件的集合元素，对于 List 和 Set，过滤结果都是一个 List，对 Map 来说结果还是一个 Map

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3) //[three, four]


val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
println(filteredMap) //{key11=11}

//上述map简介写法
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
val filteredValuesMap = numbersMap.filterValues { it < 10 }
println(filteredKeysMap)
println(filteredValuesMap)
{key1=1, key11=11}
{key1=1, key2=2, key3=3}
```

- 如果想在过滤中使用元素在集合中的位置，应该使用 **filterIndexed**()。它接受一个带有两个参数的谓词：元素的索引和元素的值。

```
val numbers = listOf("one", "two", "three", "four")
val filteredIdx = numbers.filterIndexed { index, number -> (index != 0) && (number.length < 5)
}
[two, four]
```

- 通过使用否定条件来过滤集合，请使用 **filterNot**()，它返回一个让谓词产生 false 的元素列表。

```
val filteredNot = numbers.filterNot { it.length <= 3 }
[three, four]
```

- 只返回给定类型的集合元素，请使用**filterIsInstance<T>()**。List<Any> 上调用filterIsInstance<T>()时，返回一个 List<T>

```
val numbers = listOf(null, 1, "two", 3.0, "four")
numbers.filterIsInstance<String>().forEach {
    println(it.toUpperCase())
}

TWO
FOUR
```

- 返回所有的非空元素，请使用**filterNotNull()**。在一个 List<T?> 上被调用时，filterNotNull() 返回一个过滤后的元素都不为空的List<T: Any>

```
val numbers = listOf(null, "one", "two", null)
numbers.filterNotNull().forEach {
    println(it)
}
one
two
```

（2-2）划分为不同的集合实现过滤：partition（谓词）

- 通过一个谓词，过滤集合并且将匹配的元素存放在一个单独的列表中，将不匹配的元素存放在另一个的列表中，请使用partition()函数

```
val numbers = listOf("one", "two", "three", "four","five")
val (match, rest) = numbers.partition { it.length > 3 }
println(match.get(1))
println(rest)

four
[one, two]
```

(2-3) 通过检验谓词实现过滤： any(谓词),none(谓词),all(谓词)

- 如果至少有一个元素匹配给定谓词，那么 any() 返回 true。
- 如果没有元素与给定谓词匹配，那么 none() 返回 true。
- 如果所有元素都匹配给定谓词，那么 all() 返回 true，注意，在一个空集合上使用任何有效的谓词去调用 all() 都会返回 true。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.any { it.endsWith("e") })
println(numbers.none { it.endsWith("a") })
println(numbers.all { it.endsWith("e") })
println(emptyList<Int>().all { it > 5 })
true
true
false
true
```

- any()和none()可以不带谓词，这种情况下它们只是用来检查集合是否为空，如果集合中有元素，any() 返回 true，否则返回 false；none() 则相反。

```
val numbers = listOf("one", "two", "three", "four")
val empty = emptyList<String>()
println(numbers.any())
println(empty.any())
println(numbers.none())
println(empty.none())

true
false
false
true
```

（3）集合加减运算（plus 与 minus 操作符）

（3-1）List和Set集合：kotlin中为其定义了plus(+)和minus(-)操作符，操作符前的操作数（第一个操作数）是一个集合，操作符后的操作数（第二个操作数）可以是一个元素或者是另一个集合，返回值是一个新的只读集合。如果第二个操作数是一个元素，那么 minus 移除其在原始集合中的 第一次 出现；如果是一个集合，那么移除其元素在原始集合中的 所有出现。

```
val numbers = listOf("one", "two", "three", "four")

val plusList = numbers + "five"
val minusList = numbers - listOf("three", "four")

println(plusList)
println(minusList)

[one, two, three, four, five]
[one, two]
```

（3-2）Map集合：plus(+)和minus(-)操作符对Map的作用与其它集合不同。

- 当使用plus(+)操作符时，操作符前的操作数（第一个操作数）是一个Map集合，操作符后的操作数（第二个操作数）可以是一个Pair元素或者是另一个Map集合，右侧操作数中有左侧 Map中已存在的键时，该条目将使用右侧的值。

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap + Pair("four", 4))
println(numbersMap + Pair("one", 10))
println(numbersMap + mapOf("five" to 5, "one" to 11))
{one=1, two=2, three=3, four=4}
{one=10, two=2, three=3}
{one=11, two=2, three=3, five=5}
```

- 当使用minus(-)操作符时，左侧操作数是一个Map集合，右侧操作数可以是单个键或键的集合： list 、set 等，是根据右侧的键来剔除

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap - "one")
println(numbersMap - listOf("two", "four"))
{two=2, three=3}
{one=1, three=3}
```

（4）集合分组（groupBy，groupingBy）

（4-1）想要对集合元素进行分组，请使用分组函数**groupBy**()，函数带一个 lambda表达式时，返回一个 **Map<String,List>**，Map的每个键都是lambda表达式的执行结果，Map对应的值集合是根据键生成的。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.groupBy { it.first().toUpperCase() })

{O=[one], T=[two, three], F=[four, five]}
```

（4-2）想要对集合元素进行分组，请使用分组函数**groupBy**()，函数带两个lambda表达式时，返回一个 **Map<String,List>**，第一个lambda表达式keySelector用来生成键，第二个lambda表达式valueTransform是接收了根据键生成的对应的值集合，而且可以接着对值集合进行转换操作，当然也可以不进行对值集合转换操作。

- 说明：键映射函数：keySelector；值转换函数：valueTransform

```
public inline fun <T, K> Iterable<T>.groupBy(keySelector: (T) -> K): Map<K, List<T>>
println(numbers.groupBy(keySelector = { it.first() })
o=[one], t=["two", "three"], f=["four", "five"]}

public inline fun <T, K> Iterable<T>.groupBy( keySelector: (T) -> K): Map<K, List<T>>
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.toUpperCase() }))
o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}
```

（4-3）**groupingBy**()函数：对集合元素先进行分组，每个组的键是groupingBy表达式的结果，每个组的值是后续的二次操作的结果。

- groupingBy()对集合进行分组，返回一个Grouping类的实例，然后操作Grouping实例可以再以一种惰性方式应用于所有分组。
- Grouping支持的操作（二次操作）
  - eachCount() 计算每个组中的元素。
  - fold() 与 reduce() 对每个组分别执行 fold 与 reduce 操作，作为一个单独的集合并返回结果。
  - aggregate() 随后将给定操作应用于每个组中的所有元素并返回结果。这是对 Grouping 执行任何操作的通用方法。当折叠或缩小不够时，可使用它来实现自定义操作。

```
//eachCount
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.groupingBy { it.first() }.eachCount())
{o=1, t=2, f=2, s=1}


val numbers = listOf(2, 11, 22, 23, 3, 4)
val groupingBy: Grouping<Int, Boolean> = numbers.groupingBy { it % 2 == 0 }
```

```
val reduce: Map<Boolean, Int> = groupingBy.reduce { key, acc, item -> acc + item }

//reduce
val reduce: Map<Boolean, Int> = numbers.groupingBy { it % 2 == 0 }
{true=28, false=37}

//fold
val numbers2 = listOf(2, 11, 22, 23, 3, 4)
println(numbers2.groupingBy { it % 2 == 0 }.fold(0) { acc, item -> acc + item })


val numbers = listOf<Int?>()
val sum = numbers.reduceOrNull { sum, element -> sum?.plus(element ?: 0) }
println(sum)
val sum1 = numbers.reduce { sum, element -> sum?.plus(element ?: 0) }
println(sum1)



//aggregate
//key : 分组组名
//acc : 聚合计算的结果
//element : 当前key组的当前位置数据
//first : 当前key组的第一项
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.groupingBy { it.first()}.aggregate{ key, acc:String?, element,first ->
    if(first) "$element" else "$acc + $element"
})
val numbers2 = listOf(2, 22, 4, 11, 23 , 3)
 println(numbers2.groupingBy { it % 2}.aggregate{ key, acc:String?, element,first ->
    if(first) "$element" else "$acc + $element"
})
```

```
}
```

（5）取集合的一部分（slice,take,drop,chunked,windowed等都是集合的扩展函数）

（5-1）**slice**扩展函数：返回具有给定索引的集合元素列表

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.slice(1..3))
println(numbers.slice(0..4 step 2))
println(numbers.slice(setOf(3, 5, 0)))
```

（5-2）**take**扩展函数：从头开始获取**指定数量**的元素，请使用take，从尾开始获取指定数量的元素，请使用takeList

- 注：如果传入数字大于集合元素个数，则两个扩展函数都返回整个集合。
- 方法返回的是一个指定数量的元素集合

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.take(3)) //[one, two, three]
println(numbers.takeLast(3)) //[four, five, six]

A(val a:Int,va b:String?)
val listOf = listOf(A(1, "a"), A(2, null))
val take: List = listOf.mapNotNull { it.b }.take(1)
```

（5-3）**drop**扩展函数：从头或从尾去除给定数量的元素，请调用 drop()或 dropLast()函数

```
println(numbers.drop(1))
println(numbers.dropLast(5))
```

（5-4）**take或drop带谓词的扩展函数**：谓词是一个用来定义要获取或去除元素的Predicate。

- **takeWhile**：从集合首部不停的获取不符合谓词的元素，直到遇见第一个符合谓词的元素，获取结束。如果首元素就符合谓词，则结果为空。
- **takeLastWhile**：从集合末尾反向不停的获取符合谓词的元素，直到遇见第一个不符合谓词的元素，获取结束，元素区间的元素是遇见第一个不符合谓词的元素的右边的元素开始。如果集合的最后一个元素就与谓词匹配，则结果为空。
- **dropWhile**：从集合首部不停的删除符合谓词的元素，直到遇见第一个不符合谓词的元素，删除结束，返回剩余元素
- **dropLastWhile**：从集合末尾反向不停的删除符合谓词的元素，直到遇见第一个不符合谓词的元素，删除结束，返回剩余元素。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")

println(numbers.takeWhile { it.startsWith('f') })
println(numbers.takeLastWhile { it != "three" })
println(numbers.dropWhile { it.length == 3 })
println(numbers.dropLastWhile { it.contains('i') })

[one, two, three]
[four, five, six]
[three, four, five, six]
[one, two, three, four]
```

- （5-5）**chunked函数**：将集合分解成给定大小的一个一个块。函数有一个参数代表将每个块中元素个数，函数返回的是一个List，包含所有块的List

```
val numbers = (0..13).toList()
println(numbers.chunked(3))//
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12, 13]]
```

- （5-6）windowed函数：查询集合中**所有可能的区间**。
  - 函数有一个参数时，代表每个区间元素的个数。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.windowed(3))

[[one, two, three], [two, three, four], [three, four, five]]
```

  - 函数的可选参数step：默认情况下，相邻两个区间的首元素也是相邻的，即step=1，如果step=2代表只会过滤出相隔一个区间的区间集合

```
val numbers = (1..10).toList()
println(numbers.windowed(3, step = 2))

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

  - partialWindows：如果最后一个区间的元素个数不满足windowed函数的第一个参数的值，但如果partialWindows=true，也会查询出此区间，如果为false，则不会查询出此区间。

```
val numbers = (1..10).toList()
println(numbers.windowed(3, step = 2, partialWindows = true))
[[1, 2, 3], [3, 4, 5], [5, 6, 7], [7, 8, 9], [9, 10]]
println(numbers.windowed(3, step = 2, partialWindows = false))
[[1, 2, 3], [3, 4, 5], [5, 6, 7], [7, 8, 9]]
```

- （5-7）windowed函数返回区间集合后，还可以调用lambda表达式，对区间集合中的每个区间进行后续处理。

```
val numbers = (1..10).toList()
println(numbers.windowed(3) { it.sum() })
[6, 9, 12, 15, 18, 21, 24, 27]
```

- （5-8）**zipEithNext**函数：构建只有两个元素的区间集合，请注意：相邻区间的首元素是相邻的。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.zipWithNext())

[(one, two), (two, three), (three, four), (four, five)]
```

- （5-9）zipEithNext函数构建好区间集合后，还可以调用lambda表达式，对区间集合中的每个区间的两个元素作为参数进行后续处理。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.zipWithNext() { s1, s2 -> s1.length > s2.length})

[false, false, true, false]
```

（6）取集合中的某个元素

（6-1）按位置获取

- **elementAt**(下标)，或者**get(下标)** 或 **[下标]**

```
val numbers = linkedSetOf("one", "two", "three", "four", "five")
println(numbers.elementAt(3))
four

val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
println(numbersSortedSet.elementAt(0)) // 元素以升序存储，four在0位置
four
```

- **elementAtOrNull**(下标)：当下标超出集合范围时，不会抛异常，而是返回null。（gerOrNull）

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.elementAtOrNull(5))
null
```

- **elementAtOrElse**(下标){下标->xxx}，此方法还接收有个lambda表达式，如果下标越界，则返回lambda表达式的结果；不越界，则获取下标位置集合元素。（getOrElse()）

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.elementAtOrElse(4) { index -> "The value for index $index is undefined"})
println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undefined"})
five
The value for index 5 is undefined
```

(6-2) 按条件获取

- **first()，last()**：获取集合中第一个和最后一个元素

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.first())
println(numbers.last())
one six
```

- **first{}，last{}**：还可以在集合中搜索与给定谓词匹配的第一个和最后一个元素

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.first( { it.length > 3 }))
println(numbers.last { it.startsWith("f") })
three
five
```

- **firstOrNull{}，lastOrNull{}**：如果集合中没有元素与谓词匹配，first{}，last{}函数会抛异常，为了避免抛异常，而是使用了firstOrNull{},lastOrNull{}，没有元素匹配时会返回null.
  - 注意：为了更方便，**find{}可以替换firstOrNull{}，findLast{}可以替换lastOrNull{}**

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.firstOrNull { it.length > 6 })
println(numbers.lastOrNull{it.startsWith("m")})
null
null
```

- **random()**，随机获取集合中的一个元素，函数可以不带参数或者带一个Random对象作为随机源。

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.random())
3
```

- **randomOrNull()**：如果集合是一个空集合，没有元素，则random()随机获取集合中一个元素时，会抛异常，而randomOrNull()不会跑异常，而是返回null

```
val numbers = listOf<Int>()
println(numbers.random())
Exception in thread "main" java.util.NoSuchElementException: Collection is empty.

val numbers = listOf<Int>()
println(numbers.randomOrNull())
null
```

- **contains(xx)**：如果集合的某个元素等于contains函数的参数，函数返回true，否则返回false

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.contains("four"))
true
```

- in：作用同contains函数，实际内部调用了contains

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.contains("four"))
println("zero" in numbers)
false
```

- isEmpty()：检查集合是空集合，是返回true，否则返回false
- isNotEmpty())：不是空集合，不是返回true，否则返回false

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.isEmpty())
println(numbers.isNotEmpty())
val empty = emptyList<String>()
println(empty.isEmpty())
println(empty.isNotEmpty())
false
true
 true
falseZ
```

（7）集合中元素排序

(7-1) Comparable

- 大多数内置类型是可比较的（已经实现了Comparable接口的compareTo函数），数值类型使用数值的自然顺序，Char和String使用字典的自然顺序。

(7-2) Comparable

- **也可以为用户自定义的类型的实例手动实现自定义顺序**，让你可以按自己喜欢的方式对自定义类型的实例进行排序，**需要让这个自定义类型继承Comparable，实现compareTo()函数**，函数参数必须是另一个具有相同类型的对象，函数返回一个整数值来表示那个对象更大
- 正值：调用者大
- 负值：参数大
- 0：二者相等
- 用户自定义类型的实例排序案例

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        } else if (this.minor != other.minor) {
            return this.minor - other.minor
        } else return 0
    }
}
fun main() {
    println(Version(1, 2) > Version(1, 3))
    println(Version(2, 0) > Version(1, 5))
}

false
true
```

（7-3）自定义顺序除了让你可以按自己喜欢的方式对任何类型的实例进行排序，特别是，**你可以为不可比较类型定义顺序，或者为可比较类型手动定义非自然顺序**，**需要实现Comparator接口的compare函数**，函数接受两个实例，返回比较的整数结果，结果解释同compareTo函数。需要搭配sortedWith使用，sortedWith正好需要一个比较器。

```
val lengthComparator = Comparator { str1: String, str2: String -> str1.length - str2.length }
//按照字符串的长度来排序，而不是按照字符串的字典顺序。
println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
[c, bb, aaa]
```

（7-4）另一种自定义排序的方式仍然是使用**Comparator接口的compare函数**，**是直接使用标准库中的 compareBy{}函数**，函数接受一个lambda表达式，表达式的入参是it，返回值就是Comparable<*>?比

较器的实例，正好sortedWith需要一个比较器。

```
//it.length是自然顺序的比较规则，compareBy是封装比较规则返回一个比较器，sortedWith拿到比较器对集合元素进行排序。
println(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))
[c, bb, aaa]
```

（7-5）自然顺序

- **内置函数sorted()，sortedDescending()**：按照自然顺序进行升序和降序。适用于已经实现了Comparable接口的类型的集合。

```
val numbers = listOf("one", "two", "three", "four")
println("Sorted ascending: ${numbers.sorted()}")
println("Sorted descending: ${numbers.sortedDescending()}")

Sorted ascending: [four, one, three, two]
Sorted descending: [two, three, one, four]
```

（7-6）自定义顺序

- **内置函数sortedBy{}，sortedByDescending{}**：按照自定义顺序或者对不可比较对象排序，所以函数可以带谓词来定义排序条件，即lambda表达式，表达式入参是it，返回值是Comparable<*>?的实例。上面两个函数适用于已经实现了Comparable接口的类型的集合
  - sortedBy{it.length}其实内部就是调用的sortedWith(compareBy { it.length }

```
val numbers = listOf("one", "two", "three", "four")

val sortedNumbers = numbers.sortedBy { it.length }
println("Sorted by length ascending: $sortedNumbers")

val sortedByLast = numbers.sortedByDescending { it.last() }
println("Sorted by the last letter descending: $sortedByLast")


Sorted by length ascending: [one, two, four, three]
Sorted by the last letter descending: [four, two, one, three]
```

（7-7）倒序排序

- reversed()：返回带有元素副本的新集合，因此，如果你之后改变了原始集合，这并不会影响先前获得的reversed的结果。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.reversed())

[four, three, two, one]
```

- asReversed()：返回同一个集合的反向视图
  - 如果原始列表不会发生变化（定义为listOf），那么它会比reversed（）更轻量，更合适。

```
val numbers = listOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()
println(reversedNumbers)

[four, three, two, one]
```

- 如果原始列表会发生变化（定义为mutableListOf），原始列表的所有更改都会立刻反映在反向视图中

```
val numbers = mutableListOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()
println(reversedNumbers)
numbers.add("five")
println(reversedNumbers)

[four, three, two, one]
[five, four, three, two, one]
```

- 因此，如果列表的可变性未，使用reversed()更合适

（7-8）随机顺序

- **shuffled**()：返回一个新集合，以随机顺序排序的新集合。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.shuffled())

[one, two, four, three]
```

(8) 集合聚合操作

- 94-4、集合的聚合操作-基于集合内容返回单个值的操作

(9)