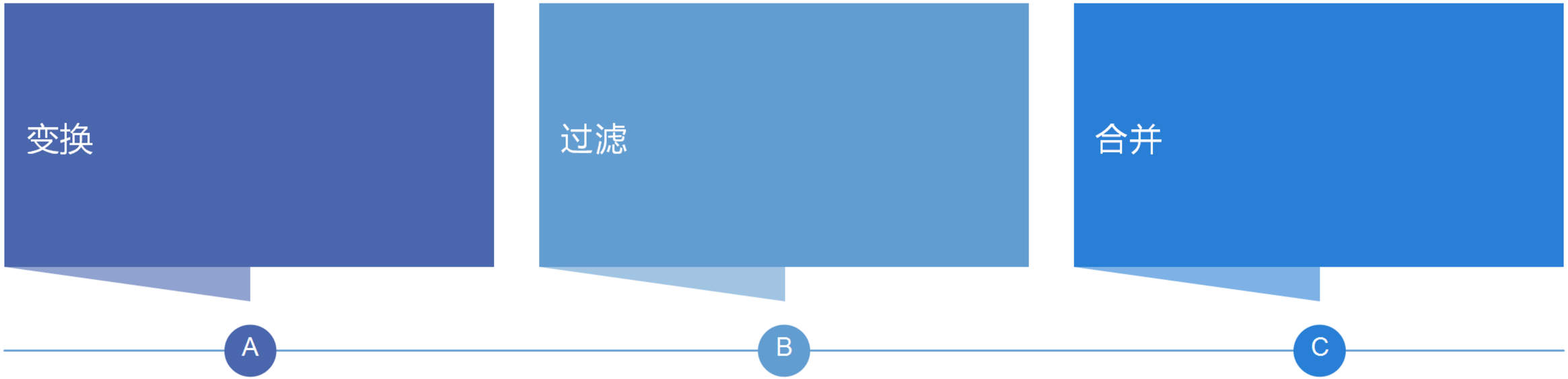
彻底掌握Kotlin



变换

函数式编程

- 我们一直在学习面向对象编程范式，另一个较知名的编程范式是诞生于20世纪50年，基于抽象数学的λ演算发展而来的函数式编程，尽管函数式编程语言更常用在学术而非商业软件领域，但它的一些原则适用于任何编程语言。函数式编程范式主要依赖于高阶函数（以函数为参数或返回函数）返回的数据，这些高阶函数专用于处理各种集合，可方便地联合多个同类函数构建链式操作以创建复杂的计算行为。Kotlin支持多种编程范式，所以你可以混用面向对象编程和函数式编程范式来解决手头的问题。

函数类别

- 一个函数式应用通常由三大类函数构成：变换transform、过滤filter、合并combine。每类函数都针对集合数据类型设计，目标是产生一个最终结果。函数式编程用到的函数生来都是可组合的，也就是说，你可以组合多个简单函数来构建复杂的计算行为。

变换

- 变换是函数式编程的第一大类函数，变换函数会遍历集合内容，用一个以值参形式传入的变换器函数，变换每一个元素，然后返回包含已修改元素的集合给链上的其他函数。
- 最常用的两个变换函数是map和flatMap。

map

- map变换函数会遍历接收者集合，让变换器函数作用于集合里的各个元素，返回结果是包含已修改元素的集合，会作为链上下一个函数的输入。

```
1  fun main() {
2      val animals: List<String> = listOf("zebra","giraffe","elephant","rat")
3      val babies: List<String> = animals
4          .map {animal -> "A baby $animal"}
5          .map { baby -> "$baby, with the cutest little tail ever!" }
6      println(babies)
7      println(animals)
8  }
```

map

- 可以看到，原始集合没有被修改，map变换函数和你定义的变换器函数做完事情后，返回的是一个新集合，这样，变量就不用变来变去了。
- 事实上，函数式编程范式支持的设计理念就是不可变数据的副本在链上的函数间传递。

map

- map返回的集合中的元素个数和输入集合必须一样，不过，返回的新集合里的元素可以是不同类型的。

```
1  fun main() {
2      val animals: List<String> = listOf("zebra","giraffe","elephant","rat")
3      val animalsLength: List<Int> = animals.map { it.length }
4      println(animalsLength)
5      //看下map函数的定义
6      //public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>
7  }
```

flatMap

- flatMap函数操作一个集合的集合，将其中多个集合中的元素合并后返回一个包含所有元素的单一集合。

```
1  fun main() {
2      val result:List<Int> = listOf(
3          listOf(1, 2, 3),
4          listOf(4, 5, 6)).flatMap { it }
5      println(result)
6  }
```

过滤

过滤

- 过滤是函数式编程的第二大类函数，过滤函数接受一个predicate函数，用它按给定条件检查接收者集合里的元素并给出true或false的判定。如果predicate函数返回true，受检元素就会添加到过滤函数返回的新集合里。如果predicate函数返回false，那么受检元素就被移出新集合。

filter

- 过滤集合中元素含有“J”字母的元素。

```
1  fun main() {
2      val result:List<String> = listOf("Jack","Jimmy","Rose","Tom")
3                                .filter { it.contains( other: "J") }
4      println(result)
5  }
```

filter

- filter过滤函数接受一个predicate函数，在flatMap遍历它的输入集合中的所有元素时，filter函数会让predicate函数按过滤条件，将符合条件的元素都放入它返回的新集合里。最后，flatMap会把变换器函数返回的子集合合并在一个新集合里。

```
fun main() {
    val items : List<List<String>> = listOf(
        listOf("red apple", "green apple", "blue apple"),
        listOf("red fish", "blue fish"),
        listOf("yellow banana", "teal banana")
    )
    val redItems : List<String> = items.flatMap { it.filter { it.contains( other: "red") } }
    println(redItems)
}
```

filter

- 找素数，除了1和它本身，不能被任何数整除的数。仅使用了几个简单函数，我们就解决了找素数这个比较复杂的问题，这就是函数式编程的独特魅力：每个函数做一点，组合起来就能干大事。

```
1  fun main() {
2      val numbers: List<Int> = listOf(7, 4, 8, 4, 3, 22, 18, 11)
3      val primes: List<Int> = numbers.filter { number ->
4          (2 until number).map { number % it }
5              .none { it == 0 }
6      }
7      println(primes)
8  }
```

合并

合并

- 合并是函数式编程的第三大类函数，合并函数能将不同的集合合并成一个新集合，这和接收者是包含集合的集合的flatMap函数不同。

zip

- zip合并函数来合并两个集合，返回一个包含键值对的新集合。

```
fun main() {
    val employees: List<String> = listOf("Jack", "Jason", "Tommy")
    val shirtSize: List<String> = listOf("large", "x-large", "medium")
    val employeeShirtSizes: Map<String, String> = employees.zip(shirtSize).toMap()
    println(employeeShirtSizes["Jack"])
}
```

fold

- 另一个可以用来合并值的合并类函数是fold，这个合并函数接受一个初始累加器值，随后会根据匿名函数的结果更新。

```
fun main() {
    val foldedValue: Int = listOf(1, 2, 3, 4).fold( initial: 0) { accumulator, number ->
        println("Accumulated value: $accumulator")
        accumulator + (number * 3) ^fold
    }
    println("Final value: $foldedValue")
}
```

为什么要使用函数式编程

- 想象一下用面向对象编程范式来实现同样的任务。

```
List<String> employees = Arrays.asList("Jack", "Jason", "Tommy");
List<String> shirtSize = Arrays.asList("large", "x-large", "medium");
Map<String, String> employeeShirtSizes = new HashMap<>();
for (int i = 0; i < employees.size(); i++) {
    employeeShirtSizes.put(employees.get(i), shirtSize.get(i));
}
```

为什么要使用函数式编程

- 乍看之下，实现同样的任务，Java版本和函数式版本的代码量差不多，但仔细分析一下，就能看出函数式版本的诸多优势。
  - 累加变量（employeeShirtSizes）都是隐式定义的。
  - 函数运算结果会自动赋值给累加变量，降低了代码出错的机会。
  - 执行新任务的函数很容易添加到函数调用链上，因为他们都兼容Iterable类型。

为什么要使用函数式编程

- 假设一个employeeShirtSizes集合，你需要格式化它。

```
List<String> formattedList = new ArrayList<>();
for (Map.Entry<String, String> entry : employeeShirtSizes.entrySet()) {
    formattedList.add(String.format("%s, shirt size:%s",
            entry.getKey(),entry.getValue()));
}
System.out.println(formattedList);
```

```
//函数式编程
val list:List<String> = employeeShirtSizes.map { "${it.key}, shirt size: ${it.value}" }
println(list)
```

序列

- List、Set、Map集合类型，这几个集合类型统称为及早集合（eager collection)

  这些集合的任何一个实例在创建后，它要包含的元素都会被加入并允许你访问。

  对应及早集合，Kotlin还有另外一类集合：惰性集合（lazy collection）类似于类的惰性初始化，惰性集合类型的性能表现优异，尤其是用于包含大量元素的集合时，因为集合元素是按需产生的。

序列

- Kotlin有个内置惰性集合类型叫序列（Sequence），序列不会索引排序它的内容，也不记录元素数目，事实上，在使用一个序列时，序列里的值可能有无限多，因为某个数据源能产生无限多个元素。

generateSequence

- 针对某个序列，你可能会定义一个只要序列有新值产生就被调用一下的函数，这样的函数叫迭代器函数，要定义一个序列和它的迭代器，你可以使用Kotlin的序列构造函数generateSequence，generateSequence函数接受一个初始种子值作为序列的起步值，在用generateSequence定义的序列上调用一个函数时，generateSequence函数会调用你指定的迭代器函数，决定下一个要产生的值。

generateSequence

- 惰性集合究竟有什么用呢？为什么要用它而不是List集合呢？假设你想产生头1000个素数。

```
fun main() {
    fun Int.isPrime(): Boolean {
        (2 until this).map { it: Int
            if (this % it == 0) {
                return false
            }
        }
        return true
    }
    val toList : List<Int> = (1..5000).toList().filter { it.isPrime() }.take( n: 1000)
    println(toList.size)
}
```

generateSequence

- 这样的代码实现表明，你不知道该检查多少个数才能得到整1000个素数，所有你用了5000这个预估数。但事实上5000个数远远不够，只能找出669个素数。

```
fun main() {
    fun Int.isPrime(): Boolean {
        (2 until this).map { it: Int
            if (this % it == 0) {
                return false
            }
        }
        return true
    }
    val oneTousandPrimes : Sequence<Int> = generateSequence( seed: 3) { value ->
        value + 1
    }.filter { it.isPrime() }.take( n: 1000)
    println(oneTousandPrimes.toList().size)
}
```