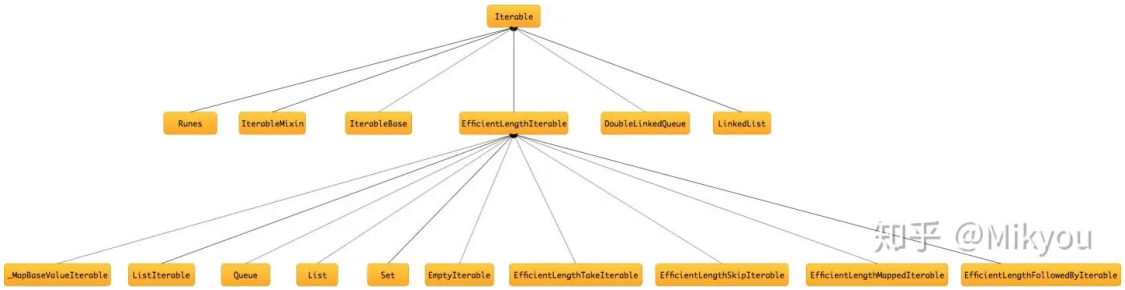
【Flutter】Dart 语法篇之集合操作符函数与源码分析 (三)

在这蓝色天空下

一、 `Iterable<E>`

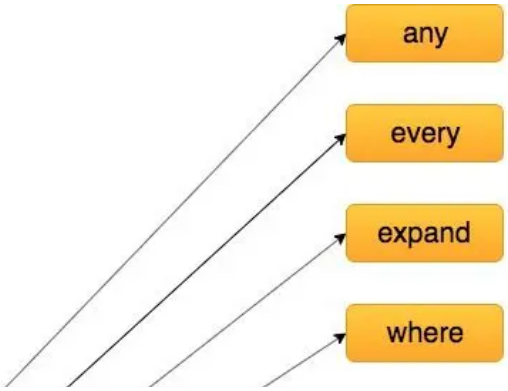
在 dart 中几乎所有集合拥有的操作符函数 (例如: map、every、where、reduce 等) 都是因为继承或者实现了 `Iterable`。 Iterable 可迭代的

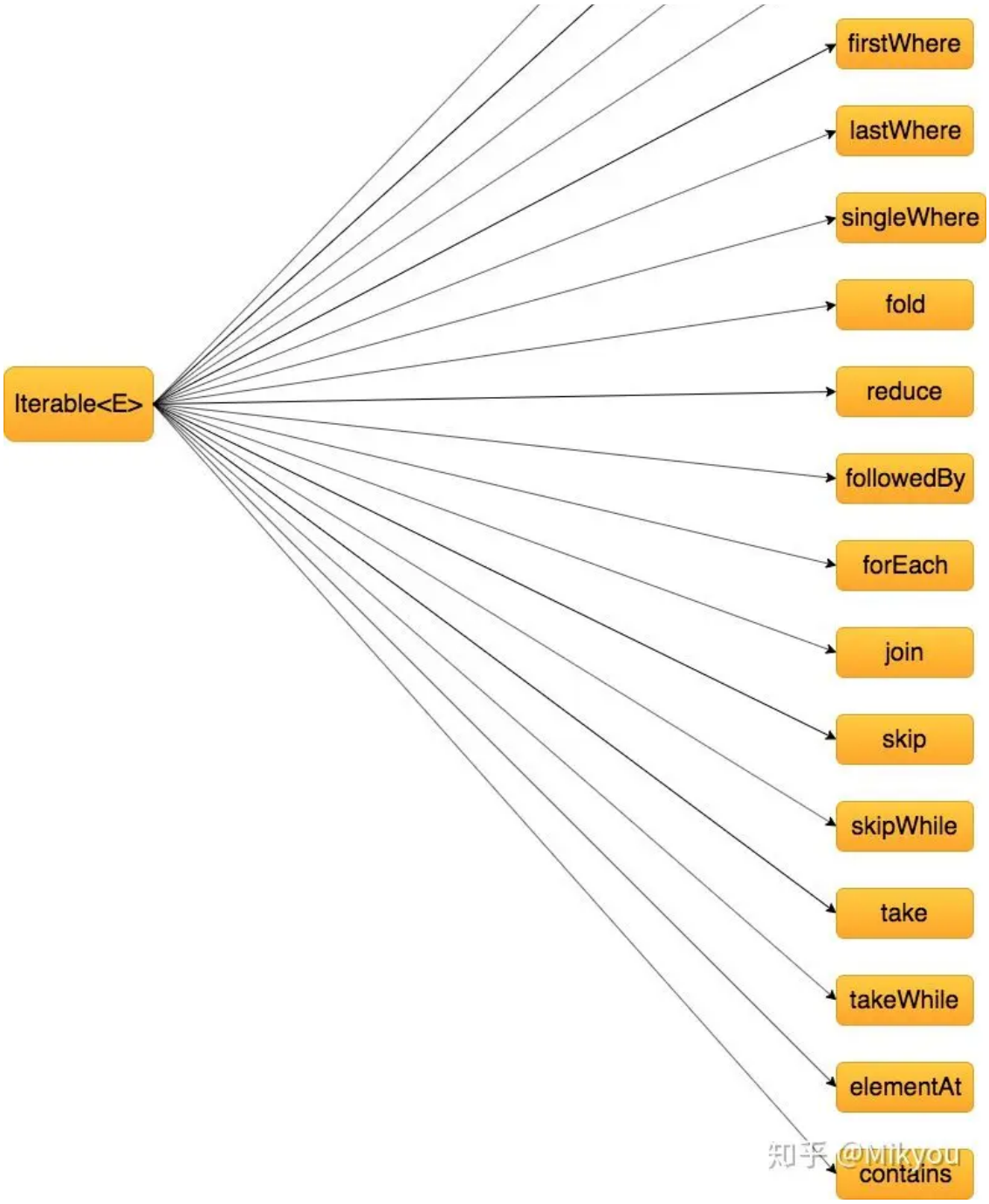
1、Iterable 类关系图



image

2、Iterable 类方法图





image

二、forEach

1、介绍

```
void forEach(void f(E element))
```

forEach在dart中用于**遍历和迭代集合**，也是dart中操作集合最常用的方法之一。接收一个`f(E element)`函数作为参数，返回值类型为空void.

2、使用方式

```
main() {
  var languages = <String>['Dart', 'Kotlin', 'Java',
'Javascript', 'Go', 'Python', 'Swift'];
  languages.forEach((language) => print('The language is
$language'));//由于只有一个表达式，所以可以直接使用箭头函数。
  languages.forEach((language){
     if(language == 'Dart' || language == 'Kotlin') {
         print('My favorite language is $language');
     }
  });
}
```

3、源码解析

```
void forEach(void f(E element)) {
    //可以看到在forEach内部实际上就是利用for-in迭代，每迭代一次就执
行一次f函数，
    //并把当前element回调出去
    for (E element in this) f(element);
  }
```

三、map

1、介绍

```
Iterable<T> map<T>(T f(E e))
```

map函数主要用于**集合中元素的映射**，也可以映射转化成其他类型的元素。可以看到map接收一个`T f(E e)`函数作为参数，最后返回一个泛型参数为`T`的`Iterable`。实际上是返回了带有元素的一个新的**惰性**`Iterable`, 然后通过迭代的时候，对每个元素都调用`f`函数。注意: `f`函数是一个接收泛型参数为`E`的元素,然后返回一个泛型参数为`T`的元素，这就是map可以将原集合中每个元素映射成其他类型元素的原因。

2、使用方式

```
main() {
  var languages = <String>['Dart', 'Kotlin', 'Java',
'Javascript', 'Go', 'Python', 'Swift'];
  print(languages.map((language) => 'develop language is $
{language}').join('---'));
}
```

3、源码解析

以上面的例子为例，

- 1、首先，需要明确一点，`languages`内部本质是一个`_GrowableList<T>`, 我们都知道`_GrowableList<T>`是继承了`ListBase<T>`,然后`ListBase<E>`又 mixin with `ListMixin<E>`. 所以`languages.map`函数调用就是调用`ListMixin<E>`中的map函数，实际上还是相当于调用了自身的成员函数map.

image

```
@pragma("vm:entry-point")
class _GrowableList<T> extends ListBase<T> {//
_GrowableList<T>是继承了ListBase<T>
    ...
}

abstract class ListBase<E> extends Object with ListMixin<E>
{//ListBase mixin with ListMixin<E>
    ...
}
```

- 2、然后可以看到`ListMixin<E>`实际上实现了`List<E>`,然后`List<E>`继承了`EfficientLengthIterable<E>`，最后`EfficientLengthIterable<E>`继承`Iterable<E>`，所以最终的map函数来自于`Iterable<E>`但是具体的实现定义在`ListMinxin<E>`中。

```
abstract class ListMixin<E> implements List<E> {
    ...
     //可以看到这里是直接返回一个MappedListIterable，它是一个惰性Iterable
     Iterable<T> map<T>(T f(E element)) => MappedListIterable<E, T>(this, f);
    ...
}
```

- 3、为什么是惰性的呢，**可以看到它并不是直接返回转化后的集合，而是返回一个带有值的MappedListIterable的，如果不执行`elementAt`方法，是不会触发执行map传入的`f`函数**，所以它是惰性的。

```
class MappedListIterable<S, T> extends ListIterable<T> {
  final Iterable<S> _source;//_source存储了所携带的原集合
  final _Transformation<S, T> _f;//_f函数存储了map函数传入的闭包，

  MappedListIterable(this._source, this._f);

  int get length => _source.length;
  //注意：只有elementAt函数执行的时候，才会触发执行_f方法，然后通过_source的elementAt函数取得原集合中的元素，
```

```
  //最后针对_source中的每个元素执行_f函数处理。
  T elementAt(int index) => _f(_source.elementAt(index));
}
```

- 4、一般不会单独使用map函数，因为单独使用map的函数时，仅仅返回的是惰性的`MappedListIterable`。由上面的源码可知，仅仅在elementAt调用的时候才会触发map中的闭包。所以我们一般使用完map后会配合`toList()、toSet()`函数或者触发`elementAt`函数的函数(例如这里的`join`)一起使用。

```
languages.map((language) => 'develop language is ${language}').toList();//toList()方法调用才会真正去执行map中的闭包。

languages.map((language) => 'develop language is ${language}').toSet();//toSet()方法调用才会真正去执行map中的闭包。

languages.map((language) => 'develop language is ${language}').join('---');//join()方法调用才会真正去执行map中的闭包。

  List<E> toList({bool growable = true}) {
    List<E> result;
    if (growable) {
      result = <E>[]..length = length;
    } else {
      result = List<E>(length);
    }
    for (int i = 0; i < length; i++) {
      result[i] = this[i];//注意: 这里的this[i]实际上是运算符重
```

```
载了[]，最终就是调用了elementAt函数，这里才会真正的触发map中的闭包，
    }
    return result;
  }
```

四、any

1、介绍

```
bool any(bool test(E element))
```

any函数主要用于**检查是否存在任意一个满足条件的元素，只要匹配到第一个就返回true，如果遍历所有元素都不符合才返回false**. any函数接收一个`bool test(E element)`函数作为参数，`test`函数回调一个`E`类型的`element`并返回一个`bool`类型的值。

2、使用方式

```
main() {
    bool isDartExisted = languages.any((language) =>
language == 'Dart');
}
```

3、源码解析

```
bool any(bool test(E element)) {
    int length = this.length;//获取到原集合的length
    //遍历原集合，只要找到符合test函数的条件，就返回true
    for (int i = 0; i < length; i++) {
      if (test(this[i])) return true;
      if (length != this.length) {
        throw ConcurrentModificationError(this);
      }
    }
    //遍历完集合后，未找到符合条件的集合就返回false
    return false;
  }
```

五、every

1、介绍

```
bool every(bool test(E element))
```

every函数主要用于**检查是否集合所有元素都满足条件，如果都满足就返回true, 只要存在一个不满足条件的就返回false.** every函数接收一个`bool test(E element)`函数作为参数，`test`函数回调一个`E`类型的`element`并返回一个`bool`类型的值。

2、使用方式

```
main() {
```

```
    bool isDartAll = languages.every((language) => language
== 'Dart');
}
```

3、源码解析

```
bool every(bool test(E element)) {
  //利用for-in遍历集合，只要找到不符合test函数的条件，就返回false.
    for (E element in this) {
      if (!test(element)) return false;
    }
//遍历完集合后，找到所有元素符合条件就返回true
    return true;
  }
```

六、where

1、介绍

```
Iterable<E> where(bool test(E element))
```

where函数主要用于**过滤符合条件的元素，类似Kotlin中的filter的作用，最后返回符合条件元素的集合。** where函数接收一个`bool test(E element)`函数作为参数，最后返回一个泛型参数为`E`的`Iterable`。类似map一样，where这里也是返回一个惰性的`Iterable<E>`, 然后对它的`iterator`进行迭代，对每

个元素都执行`test`方法。

2、使用方式

```
main() {
   List<int> numbers = [0, 3, 1, 2, 7, 12, 2, 4];
   print(numbers.where((num) => num > 6));//输出: (7,12)
   //注意: 这里是print的内容实际上输出的是Iterable的toString方法返回的内容。
}
```

3、源码解析

- 1、首先，需要明确一点`numbers`实际上是一个`_GrowableList<T>`和`map`的分析原理类似，最终还是调用了`ListMixin`中的`where`函数。

```
//可以看到这里是直接返回一个WhereIterable对象，而不是返回过滤后元素集合，所以它返回的Iterable也是惰性的。
Iterable<E> where(bool test(E element)) =>
WhereIterable<E>(this, test);
```

- 2、然后，继续深入研究下`WhereIterable`是如何实现的

```
class WhereIterable<E> extends Iterable<E> {
  final Iterable<E> _iterable;//传入的原集合
  final _ElementPredicate<E> _f;//传入的where函数中闭包参数

  WhereIterable(this._iterable, this._f);
```

```
  //注意：这里WhereIterable的迭代借助了iterator，这里是直接创建一个WhereIterator，并传入元集合_iterable中的iterator以及过滤操作函数。
  Iterator<E> get iterator => new WhereIterator<E>(_iterable.iterator, _f);

  // Specialization of [Iterable.map] to non-EfficientLengthIterable.
  Iterable<T> map<T>(T f(E element)) => new MappedIterable<E, T>._(this, f);
}
```

- 3、然后，继续深入研究下`WhereIterator`是如何实现的

```
class WhereIterator<E> extends Iterator<E> {
  final Iterator<E> _iterator;//存储集合中的iterator对象
  final _ElementPredicate<E> _f;//存储where函数传入闭包函数

  WhereIterator(this._iterator, this._f);

  //重写moveNext函数
  bool moveNext() {
  //遍历原集合的_iterator
    while (_iterator.moveNext()) {
    //注意：这里会执行_f函数，如果满足条件就会返回true，不符合条件的直接略过，迭代下一个元素；
    //那么外部迭代时候，就可以通过current获得当前元素，这样就实现了在原集合基础上过滤拿到符合条件的元素。
      if (_f(_iterator.current)) {
        return true;
      }
    }
    //迭代完_iterator所有元素后返回false,以此来终止外部迭代。
    return false;
```

```
  }
  //重写current的属性方法
  E get current => _iterator.current;
}
```

- 4、一般在使用的`WhereIterator`的时候，外部肯定还有一层`while`迭代，但是这个`WhereIterator`比较特殊，`moveNext()`的返回值由`where`中闭包函数参数返回值决定的，符合条件元素`moveNext()`就返回true,不符合就略过，迭代检查下一个元素，直至整个集合迭代完毕，`moveNext()`返回false,以此也就终止了外部的迭代循环。
- 5、上面分析，`WhereIterable`是惰性的，那它啥时候触发呢？没错就是在迭代它的`iterator`时候才会触发，以上面例子为例

```
print(numbers.where((num) => num > 6));//输出：(7,12)，最后会调用Iterable的toString方法返回的内容。

//看下Iterable的toString方法实现
String toString() =>
IterableBase.iterableToShortString(this, '(', ')');//这就是为啥输出样式是 (7,12)
//继续查看IterableBase.iterableToShortString
  static String iterableToShortString(Iterable iterable,
      [String leftDelimiter = '(', String rightDelimiter = ')']) {
    if (_isToStringVisiting(iterable)) {
      if (leftDelimiter == "(" && rightDelimiter == ")") {
        // Avoid creating a new string in the "common" case.
```

```
        return "(...)";
      }
      return "$leftDelimiter...$rightDelimiter";
    }
    List<String> parts = <String>[];
    _toStringVisiting.add(iterable);
    try {
      _iterablePartsToStrings(iterable, parts);//注意:这里实
际上就是通过将iterable转化成List，内部就是通过迭代iterator,以此会触
发WhereIterator中的_f函数。
    } finally {
      assert(identical(_toStringVisiting.last, iterable));
      _toStringVisiting.removeLast();
    }
    return (StringBuffer(leftDelimiter)
          ..writeAll(parts, ", ")
          ..write(rightDelimiter))
        .toString();
  }

  /// Convert elements of [iterable] to strings and store
them in [parts]. 这个函数代码实现比较多，这里给出部分代码
void _iterablePartsToStrings(Iterable iterable,
List<String> parts) {
  ...
  int length = 0;
  int count = 0;
  Iterator it = iterable.iterator;
  // Initial run of elements, at least headCount, and then
continue until
  // passing at most lengthLimit characters.
  //可以看到这是外部迭代while
  while (length < lengthLimit || count < headCount) {
    if (!it.moveNext()) return;//这里实际上调用了WhereIterator
中的moveNext函数，经过_f函数处理的moveNext()
```

```
    String next = "${it.current}";//获取current.
    parts.add(next);
    length += next.length + overhead;
    count++;
  }
  ...
}
```

七、firstWhere和singleWhere和lastWhere

1、介绍

```
E firstWhere(bool test(E element), {E orElse()})
E lastWhere(bool test(E element), {E orElse()})
E singleWhere(bool test(E element), {E orElse()})
```

首先从源码声明结构上来看，firstWhere、lastWhere和singleWhere是一样，它们都是接收两个参数，一个是必需参数:`test`筛选条件闭包函数，另一个是可选参数:`orElse`闭包函数。

但是它们用法却不同，firstWhere主要是**用于筛选顺序第一个符合条件的元素，可能存在多个符合条件元素**；lastWhere主要是**用于筛选顺序最后一个符合条件的元素，**

可能存在多个符合条件元素；singleWhere主要是用于筛选顺序唯一一个符合条件的元素，不可能存在多个符合条件元素，存在的话就会抛出异常IterableElementError.tooMany()， 所以使用它的使用需要谨慎注意下

2、使用方式

```
main() {
   var numbers = <int>[0, 3, 1, 2, 7, 12, 2, 4];
   //注意：如果没有找到，执行orElse代码块，可返回一个指定的默认值-1
   print(numbers.firstWhere((num) => num == 5, orElse: ()
=> -1));
   //注意：如果没有找到，执行orElse代码块，可返回一个指定的默认值-1
   print(numbers.lastWhere((num) => num == 2, orElse: () =>
-1));
   //注意：如果没有找到，执行orElse代码块，可返回一个指定的默认值，前提是集合中只有一个符合条件的元素，否则就会抛出异常
   print(numbers.singleWhere((num) => num == 4, orElse: ()
=> -1));
}
```

3、源码解析

```
//firstWhere
  E firstWhere(bool test(E element), {E orElse()}) {
    for (E element in this) {//直接遍历原集合，只要找到第一个符合条件的元素就直接返回，终止函数
      if (test(element)) return element;
```

```
    }
    if (orElse != null) return orElse();//遍历完集合后，都没找到符合条件的元素并且外部传入了orElse就会触发orElse函数
    //否则找不到元素，直接抛出异常。所以这里需要注意下，如果不想抛出异常，可能你需要处理下orElse函数。
    throw IterableElementError.noElement();
  }

  //lastWhere
  E lastWhere(bool test(E element), {E orElse()}) {
    E result;//定义result来记录每次符合条件的元素
    bool foundMatching = false;//定义一个标志位是否找到符合匹配的。
    for (E element in this) {
      if (test(element)) {//每次找到符合条件的元素，都会重置result，所以result记录了最新的符合条件元素，那么遍历到最后，它也就是最后一个符合条件的元素
        result = element;
        foundMatching = true;//找到后重置标记位
      }
    }
    if (foundMatching) return result;//如果标记位为true直接返回result即可
    if (orElse != null) return orElse();//处理orElse函数
    //同样，找不到元素，直接抛出异常。可能你需要处理下orElse函数。
    throw IterableElementError.noElement();
  }

  //singleWhere
  E singleWhere(bool test(E element), {E orElse()}) {
    E result;
    bool foundMatching = false;
    for (E element in this) {
      if (test(element)) {
        if (foundMatching) {//主要注意这里，只要foundMatching为
```

```
true,说明已经找到一个符合条件的元素，如果触发这条逻辑分支，说明不止一个元素符合条件就直接抛出IterableElementError.tooMany()异常
          throw IterableElementError.tooMany();
        }
        result = element;
        foundMatching = true;
      }
    }
    if (foundMatching) return result;
    if (orElse != null) return orElse();
     //同样，找不到元素，直接抛出异常。可能你需要处理下orElse函数。
    throw IterableElementError.noElement();
  }
```

八、join

1、介绍

```
String join([String separator = ""])
```

join函数主要是用于**将集合所有元素值转化成字符串，中间用指定的`separator`连接符连接**。 可以看到join函数比较简单，接收一个`separator`分隔符的可选参数，可选参数默认值是空字符串，最后返回一个字符串。

2、使用方式

```
main() {
```

```
  var numbers = <int>[0, 3, 1, 2, 7, 12, 2, 4];
  print(numbers.join('-'));//输出: 0-3-1-2-7-12-2-4
}
```

3、源码解析

```
//接收separator可选参数，默认值为""
  String join([String separator = ""]) {
    Iterator<E> iterator = this.iterator;
    if (!iterator.moveNext()) return "";
    //创建StringBuffer
    StringBuffer buffer = StringBuffer();
    //如果分隔符为空或空字符串
    if (separator == null || separator == "") {
      //do-while遍历iterator,然后直接拼接元素
      do {
        buffer.write("${iterator.current}");
      } while (iterator.moveNext());
    } else {
    //如果分隔符不为空
      //先加入第一个元素
      buffer.write("${iterator.current}");
      //然后while遍历iterator
      while (iterator.moveNext()) {
        buffer.write(separator);//先拼接分隔符
        buffer.write("${iterator.current}");//再拼接元素
      }
    }
    return buffer.toString();//最后返回最终的字符串。
  }
```

九、take

1、介绍

```
Iterable<E> take(int count)
```

take函数主要是用于**截取原集合前count个元素组成的集合**，take函数接收一个`count`作为函数参数，最后返回一个泛型参数为`E`的`Iterable`。类似where一样，take这里也是返回一个惰性的`Iterable<E>`, 然后对它的`iterator`进行迭代。

takeWhile函数主要用于

2、使用方式

```
main() {
   List<int> numbers = [0, 3, 1, 2, 7, 12, 2, 4];
   print(numbers.take(5));//输出(0, 3, 1, 2, 7)
}
```

3、源码解析

- 1、首先, 需要明确一点`numbers.take`调用了`ListMixin`中的`take`函数，可以看到并没有直接返回集合前`count`个元素，而是返回一个`TakeIterable<E>`惰性`Iterable`。

```
Iterable<E> take(int count) {
    return TakeIterable<E>(this, count);
  }
```

- 2、然后，继续深入研究`TakeIterable`

```
class TakeIterable<E> extends Iterable<E> {
  final Iterable<E> _iterable;//存储原集合
  final int _takeCount;//take count

  factory TakeIterable(Iterable<E> iterable, int takeCount)
{
    ArgumentError.checkNotNull(takeCount, "takeCount");
    RangeError.checkNotNegative(takeCount, "takeCount");
    if (iterable is EfficientLengthIterable) {//如果原集合是
EfficientLengthIterable，就返回创建
EfficientLengthTakeIterable
      return new EfficientLengthTakeIterable<E>(iterable,
takeCount);
    }
    //否则就返回TakeIterable
    return new TakeIterable<E>._(iterable, takeCount);
  }

  TakeIterable._(this._iterable, this._takeCount);

//注意：这里是返回了TakeIterator，并传入原集合的iterator以及
_takeCount
  Iterator<E> get iterator {
    return new TakeIterator<E>(_iterable.iterator,
_takeCount);
  }
}
```

- 3、然后，继续深入研究`TakeIterator`.

```
class TakeIterator<E> extends Iterator<E> {
  final Iterator<E> _iterator;//存储原集合中的iterator
  int _remaining;//存储需要截取的前几个元素的数量

  TakeIterator(this._iterator, this._remaining) {
    assert(_remaining >= 0);
  }

  bool moveNext() {
    _remaining--;//通过_remaining作为游标控制迭代次数
    if (_remaining >= 0) {//如果_remaining大于等于0就会继续执行
moveNext方法
      return _iterator.moveNext();
    }
    _remaining = -1;
    return false;//如果_remaining小于0就返回false,终止外部循环
  }

  E get current {
    if (_remaining < 0) return null;
    return _iterator.current;
  }
}
```

- 4、所以上述例子中最终还是调用`Iterable`的`toString`方法，方法中会进行`iterator`的迭代，最终会触发惰性`TakeIterable`中的`TakeIterator`的`moveNext`方法。

十、takeWhile

1、介绍

```
Iterable<E> takeWhile(bool test(E value))
```

takeWhile函数主要用于**依次选择满足条件的元素，直到遇到第一个不满足的元素，并停止选择**。takeWhile函数接收一个`test`条件函数作为函数参数，然后返回一个惰性的`Iterable<E>`。

2、使用方式

```
main() {
  List<int> numbers = [3, 1, 2, 7, 12, 2, 4];
  print(numbers.takeWhile((number) => number >
2).toList());//输出：[3] 遇到1的时候就不满足大于2条件就终止筛选。
}
```

3、源码解析

- 1、首先，因为`numbers`是`List<int>`所以还是调用`ListMixin`中的`takeWhile`函数

```
Iterable<E> takeWhile(bool test(E element)) {
    return TakeWhileIterable<E>(this, test);//可以看到它仅仅返
回的是TakeWhileIterable，而不是筛选后符合条件的集合，所以它是惰性。
  }
```

- 2、然后，继续看下`TakeWhileIterable<E>`的实现

```
class TakeWhileIterable<E> extends Iterable<E> {
  final Iterable<E> _iterable;
  final _ElementPredicate<E> _f;

  TakeWhileIterable(this._iterable, this._f);

  Iterator<E> get iterator {
    //重写iterator，创建一个TakeWhileIterator对象并返回。
    return new TakeWhileIterator<E>(_iterable.iterator,
_f);
  }
}

//TakeWhileIterator
class TakeWhileIterator<E> extends Iterator<E> {
  final Iterator<E> _iterator;
  final _ElementPredicate<E> _f;
  bool _isFinished = false;

  TakeWhileIterator(this._iterator, this._f);

  bool moveNext() {
    if (_isFinished) return false;
    //原集合_iterator遍历结束或者原集合中的当前元素current不满足_f
条件，就返回false以此终止外部的迭代。
    //进一步说明了只有moveNext调用，才会触发_f的执行，此时惰性的
Iterable才得以执行。
    if (!_iterator.moveNext() || !_f(_iterator.current)) {
      _isFinished = true;//迭代结束重置_isFinished为true
      return false;
    }
    return true;
```

```
  }

  E get current {
    if (_isFinished) return null;//如果迭代结束，还取current就直接返回null了
    return _iterator.current;
  }
}
```

十、skip

1、介绍

```
Iterable<E> skip(int count)
```

skip函数主要是用于**跳过原集合前count个元素后，剩下元素组成的集合**，skip函数接收一个`count`作为函数参数，最后返回一个泛型参数为`E`的`Iterable`。类似where一样，skip这里也是返回一个惰性的`Iterable<E>`, 然后对它的`iterator`进行迭代。

2、使用方式

```
main() {
  List<int> numbers = [3, 1, 2, 7, 12, 2, 4];
  print(numbers.skip(2).toList());//输出: [2, 7, 12, 2, 4] 跳过前两个元素3,1 直接从第3个元素开始
}
```

3、源码解析

- 1、首先，因为`numbers`是`List<int>`所以还是调用`ListMixin`中的`skip`函数

```
Iterable<E> skip(int count) => SubListIterable<E>(this, count, null);//返回的是一个SubListIterable惰性Iterable，传入原集合和需要跳过的count大小
```

- 2、然后，继续看下`SubListIterable<E>`的实现,这里只看下`elementAt`函数实现

```
class SubListIterable<E> extends ListIterable<E> {
  final Iterable<E> _iterable; // Has efficient length and elementAt.
  final int _start;//这是传入的需要skip的count
  final int _endOrLength;//这里传入为null
  ...
  int get _endIndex {
    int length = _iterable.length;//获取原集合长度
    if (_endOrLength == null || _endOrLength > length) return length;//_endIndex为原集合长度
    return _endOrLength;
  }

  int get _startIndex {//主要看下_startIndex的实现
    int length = _iterable.length;//获取原集合长度
    if (_start > length) return length;//如果skip的count超过集合自身长度，_startIndex为原集合长度
    return _start;//否则返回skip的count
  }
```

```
  E elementAt(int index) {
    int realIndex = _startIndex + index;//相当于把原集合中每个元素原来index,整体向后推了_startIndex,最后获取真实映射的realIndex
    if (index < 0 || realIndex >= _endIndex) {//如果realIndex越界就会抛出异常
      throw new RangeError.index(index, this, "index");
    }
    return _iterable.elementAt(realIndex);//否则就取对应realIndex在原集合中的元素。
  }
  ...
}
```

十一、skipWhile

1、介绍

```
Iterable<E> skipWhile(bool test(E element))
```

skipWhile函数主要用于**依次跳过满足条件的元素，直到遇到第一个不满足的元素，并停止筛选**。skipWhile函数接收一个`test`条件函数作为函数参数，然后返回一个惰性的`Iterable<E>`。

2、使用方式

```
main() {
  List<int> numbers = [3, 1, 2, 7, 12, 2, 4];
```

```
  print(numbers.skipWhile((number) => number <
4).toList());//输出: [7, 12, 2, 4]
  //因为3、1、2都是满足小于4的条件，所以直接skip跳过，直到遇到7不符合
条件停止筛选，剩下的就是[7, 12, 2, 4]
}
```

3、源码解析

- 1、首先，因为`numbers`是`List<int>`所以还是调用`ListMixin`中的`skipWhile`函数

```
Iterable<E> skipWhile(bool test(E element)) {
    return SkipWhileIterable<E>(this, test);//可以看到它仅仅返
回的是SkipWhileIterable，而不是筛选后符合条件的集合，所以它是惰性
的。
  }
```

- 2、然后，继续看下`SkipWhileIterable<E>`的实现

```
class SkipWhileIterable<E> extends Iterable<E> {
  final Iterable<E> _iterable;
  final _ElementPredicate<E> _f;

  SkipWhileIterable(this._iterable, this._f);
 //重写iterator，创建一个SkipWhileIterator对象并返回。
  Iterator<E> get iterator {
    return new SkipWhileIterator<E>(_iterable.iterator,
_f);
  }
}

//SkipWhileIterator
```

```
class SkipWhileIterator<E> extends Iterator<E> {
  final Iterator<E> _iterator;//存储原集合的iterator
  final _ElementPredicate<E> _f;//存储skipWhile中筛选闭包函数
  bool _hasSkipped = false;//判断是否已经跳过元素的标识，默认为
false

  SkipWhileIterator(this._iterator, this._f);

//重写moveNext函数
  bool moveNext() {
    if (!_hasSkipped) {//如果是最开始第一次没有跳过任何元素
      _hasSkipped = true;//然后重置标识为true,表示已经进行了第一
次跳过元素的操作
      while (_iterator.moveNext()) {//迭代原集合中的iterator
        if (!_f(_iterator.current)) return true;//只要找到符
合条件的元素，就略过迭代下一个元素，不符合条件就直接返回true终止当前
moveNext函数，而此时外部迭代循环正式从当前元素开始迭代，
      }
    }
    return _iterator.moveNext();//那么遇到第一个不符合条件元素之
后所有元素就会通过_iterator.moveNext()正常返回
  }

  E get current => _iterator.current;
}
```

十二、follwedBy

1、介绍

```
Iterable<E> followedBy(Iterable<E> other)
```

followedBy函数主要用于**在原集合后面追加拼接另一个`Iterable<E>`集合**，followedBy函数接收一个`Iterable<E>`参数，最后又返回一个`Iterable<E>`类型的值。

2、使用方式

```
main() {
  var languages = <String>['Kotlin', 'Java', 'Dart', 'Go',
'Python'];
  print(languages.followedBy(['Swift', 'Rust', 'Ruby', 'C+
+', 'C#']).toList());//输出: [Kotlin, Java, Dart, Go,
Python, Swift, Rust, Ruby, C++, C#]
}
```

3、源码解析

- 1、首先，还是调用`ListMixin`中的`followedBy`函数

```
Iterable<E> followedBy(Iterable<E> other) =>
      FollowedByIterable<E>.firstEfficient(this, other);//这
里实际上还是返回一个惰性的FollowedByIterable对象，这里使用命名构造器
firstEfficient创建对象
```

- 2、然后，继续看下`FollowedByIterable`中的`firstEfficient`实现

```
factory FollowedByIterable.firstEfficient(
      EfficientLengthIterable<E> first, Iterable<E> second)
```

```
{
    if (second is EfficientLengthIterable<E>) {//List肯定是一个EfficientLengthIterable，所以会创建一个EfficientLengthFollowedByIterable，传入的参数first是当前集合，second是需要在后面拼接的集合
      return new EfficientLengthFollowedByIterable<E>(first, second);
    }
    return new FollowedByIterable<E>(first, second);
  }
```

- 3、然后，继续看下`EfficientLengthFollowedByIterable`的实现,这里只具体看下`elementAt`函数的实现

```
class EfficientLengthFollowedByIterable<E> extends FollowedByIterable<E>
    implements EfficientLengthIterable<E> {
  EfficientLengthFollowedByIterable(
      EfficientLengthIterable<E> first, EfficientLengthIterable<E> second)
      : super(first, second);
 ...
  E elementAt(int index) {//elementAt在迭代过程会调用
    int firstLength = _first.length;//取原集合的长度
    if (index < firstLength) return _first.elementAt(index);//如果index小于原集合长度就从原集合中获取元素
    return _second.elementAt(index - firstLength);//否则就通过index - firstLength 计算新的下标从拼接的集合中获取元素。
  }
  ...
}
```

十三、expand

1、介绍

```
Iterable<T> expand<T>(Iterable<T> f(E element))
```

expand函数主要用于**将集合中每个元素扩展为零个或多个元素或者将多个元素组成二维数组展开成平铺一个一维数组。**expand函数接收一个`Iterable<T> f(E element)`函数作为函数参数。这个闭包函数比较特别，特别之处在于`f`函数返回的是一个`Iterable<T>`,那么就意味着可以将原集合中每个元素扩展成多个相同元素。注意expand函数最终还是返回一个惰性的`Iterable<T>`

2、使用方式

```
main() {
   var pair = [
     [1, 2],
     [3, 4]
   ];
   print('flatten list: ${pair.expand((pair) =>
pair).toList()}');//输出: flatten list: [1, 2, 3, 4]
   var inputs = [1, 2, 3];
   print('duplicated list: ${inputs.expand((number) =>
[number, number, number]).toList()}');//输出: duplicated
list: [1, 1, 1, 2, 2, 2, 3, 3, 3]
```

```
}
```

3、源码解析

- 1、首先还是调用`ListMixin`中的`expand`函数。

```
Iterable<T> expand<T>(Iterable<T> f(E element)) =>
      ExpandIterable<E, T>(this, f);//可以看到这里并没有直接返回扩展的集合，而是创建一个惰性的ExpandIterable对象返回，
```

- 2、然后继续深入`ExpandIterable`

```
typedef Iterable<T> _ExpandFunction<S, T>(S sourceElement);

class ExpandIterable<S, T> extends Iterable<T> {
  final Iterable<S> _iterable;
  final _ExpandFunction<S, T> _f;

  ExpandIterable(this._iterable, this._f);

  Iterator<T> get iterator => new ExpandIterator<S,
T>(_iterable.iterator, _f);//注意：这里iterator是一个ExpandIterator对象，传入的是原集合的iterator和expand函数中闭包函数参数_f
}

//ExpandIterator的实现
class ExpandIterator<S, T> implements Iterator<T> {
  final Iterator<S> _iterator;
  final _ExpandFunction<S, T> _f;
  //创建一个空的Iterator对象_currentExpansion
  Iterator<T> _currentExpansion = const EmptyIterator();
```

```
  T _current;

  ExpandIterator(this._iterator, this._f);

  T get current => _current;//重写current

//重写moveNext函数，只要当迭代的时候，moveNext执行才会触发闭包函数
_f执行。
  bool moveNext() {
   //如果_currentExpansion返回false终止外部迭代循环
    if (_currentExpansion == null) return false;
    //开始_currentExpansion是一个空的Iterator对象，所以
moveNext()为false
    while (!_currentExpansion.moveNext()) {
      _current = null;
      //迭代原集合中的_iterator
      if (_iterator.moveNext()) {
        //如果_f抛出异常，先重置_currentExpansion为null，遇到 if
(_currentExpansion == null) return false;就会终止外部迭代
        _currentExpansion = null;
        _currentExpansion =
_f(_iterator.current).iterator;//执行_f函数
      } else {
        return false;
      }
    }
    _current = _currentExpansion.current;
    return true;
  }
}
```

十四、reduce

1、介绍

```
E reduce(E combine(E previousValue, E element))
T fold<T>(T initialValue, T combine(T previousValue, E
element))
```

reduce函数主要**用于集合中元素依次归纳(combine)，每次归纳后的结果会和下一个元素进行归纳，它可以用来累加或累乘，具体取决于combine函数中操作，combine函数中会回调上一次归纳后的值和当前元素值，reduce提供的是获取累积迭代结果的便利条件. fold和reduce几乎相同，唯一区别是fold可以指定初始值。** 但是需要注意的是，combine函数返回值的类型必须和集合泛型类型一致。

2、使用方式

```
main() {
  List<int> numbers = [3, 1, 2, 7, 12, 2, 4];
  print(numbers.reduce((prev, curr) => prev + curr)); //累加
  print(numbers.fold(2, (prev, curr) => (prev as int) +
curr)); //累加
  print(numbers.reduce((prev, curr) => prev + curr) /
numbers.length); //求平均数
  print(numbers.fold(2, (prev, curr) => (prev as int) +
curr) / numbers.length); //求平均数
  print(numbers.reduce((prev, curr) => prev * curr)); //累乘
  print(numbers.fold(2, (prev, curr) => (prev as int) *
```

```
curr)); //累乘

  var strList = <String>['a', 'b', 'c'];
  print(strList.reduce((prev, curr) => '$prev*$curr')); //拼
接字符串
  print(strList.fold('e', (prev, curr) =>
'$prev*$curr')); //拼接字符串
}
```

3、源码解析

```
E reduce(E combine(E previousValue, E element)) {
    int length = this.length;
    if (length == 0) throw
IterableElementError.noElement();
    E value = this[0];//初始值默认取第一个
    for (int i = 1; i < length; i++) {//从第二个开始遍历
      value = combine(value, this[i]);//combine回调value值和
当前元素值，然后把combine的结果归纳到value上，依次处理。
      if (length != this.length) {
        throw ConcurrentModificationError(this);//注意：在操
作过程中不允许删除和添加元素否则就会出现
ConcurrentModificationError
      }
    }
    return value;
  }

  T fold<T>(T initialValue, T combine(T previousValue, E
element)) {
    var value = initialValue;//和reduce唯一区别在于这里value初
始值是外部指定的
    int length = this.length;
```

```
    for (int i = 0; i < length; i++) {
      value = combine(value, this[i]);
      if (length != this.length) {
        throw ConcurrentModificationError(this);
      }
    }
    return value;
  }
```

十五、elementAt

1、介绍

```
E elementAt(int index)
```

elementAt函数用于**获取对应index下标的元素**，传入一个index参数，返回对应泛型类型`E`的元素。

2、使用方式

```
main() {
  print(numbers.elementAt(3));//elementAt一般不会直接使用，更多是使用[]，运算符重载的方式间接使用。
}
```

3、源码解析

```
E elementAt(int index) {
    ArgumentError.checkNotNull(index, "index");
    RangeError.checkNotNegative(index, "index");
    int elementIndex = 0;
    //for-in遍历原集合，找到对应elementIndex元素并返回
    for (E element in this) {
      if (index == elementIndex) return element;
      elementIndex++;
    }
    //找不到抛出RangeError
    throw RangeError.index(index, this, "index", null,
elementIndex);
  }
```

总结

到这里，有关dart中集合操作符函数相关内容就结束了，关于集合操作符函数使用在Flutter中开发非常有帮助，特别在处理集合数据中，可以让你的代码实现更优雅，不要再是一上来就for循环直接开干，虽然也能实现，但是如果能适当使用操作符函数，将会使代码更加简洁。欢迎继续关注，下一篇Dart中的函数的使用...