Декларативно програмиране в процедурните езици

Итератори, потоци и генератори

Владислав Ненчев

Софийски Университет “Св. Климент Охридски”
Факултет по Математика и Информатика
Катедра по Математическа Логика и Приложенията ѝ

28 Октомври 2016

Модел на итераторите

Линейно обхождане на колекция/множество/списък/структура:

- GetNext() - извлича следващия елемент;
- HasNext() - проверява дали има следващ елемент;
  понякога се реализира като (next = HasNext()) != null.

Модел на итераторите

Линейно обхождане на колекция/множество/списък/структура:

- GetNext() - извлича следващия елемент;
- HasNext() - проверява дали има следващ елемент; понякога се реализира като (next = HasNext()) != null.

Позволяват обхождане с foreach.

Позволяват последователно изброяване на елементи, без да се заделя памет за цялата колекция (особено при pipeline!!).

Позволяват ефективна работа с множества, когато търсим конкретен елемент (без да има по-бърз начин от обхождане) или се нуждаем само от част от елементите.

Итератори в C#

Interface-и: IEnumerator<T> и IEnumerable<T>.
Чрез List<T>, Enumerable.Cast<T>, .AsEnumerable<T>.

Итератори в C#

Interface-и: IEnumerator<T> и IEnumerable<T>.
Чрез List<T>, Enumerable.Cast<T>, .AsEnumerable<T>.

Полезни "декларативни" методи на IEnumerable<T>:

- First , Last, Single - специфични елементи, синглетон;
- All, Any - квантори;
- Take, TakeWhile - взимане на част от итерацията;
- Max, Min, Sum, Average, Aggregate - агрегати;
- Reverse, Skip - обръщане и пропускане на елемент;
- Concat, Zip - комбинации от итератори;
- Intersect , Union, Except - множествени операции;
- Select, Where, OrderBy, GroupBy - заявки.

Обхождане на дърво

```
public static class IEnumerableExtensions
{
  public static void ForEach<T>(
    this IEnumerable<T> iEnumerable, Action<T> action)
  {
    foreach(T item in iEnumerable)  action(item);
  }
}
```

Обхождане на дърво

```
public static class IEnumerableExtensions
{
  public static void ForEach<T>(
    this IEnumerable<T> iEnumerable, Action<T> action)
  {
    foreach(T item in iEnumerable)  action(item);
  }
}
```

```
public TreeNode(NodeValue value,
  params TreeNode[] childrenNodes)
{
  Value = value;
  children.AddRange(childrenNodes);
  Height =
    children.Select(child => child.Height + 1).
      DefaultIfEmpty().Max();
}
```

Обхождане на дърво (продължение)

```
public IEnumerable<TreeNode> PreTraverse()
{
  yield return this;
  foreach(var ownChild in children)
    foreach(var child in ownChild.PreTraverse())
      yield return child;
}
public IEnumerable<TreeNode> PostTraverse()
{
  foreach(var ownChild in children)
    foreach(var child in ownChild.PostTraverse())
      yield return child;
  yield return this;
}
```

Обхождане на дърво (продължение)

```
public IEnumerable<TreeNode> PreTraverse()
{
  yield return this;
  foreach(var ownChild in children)
    foreach(var child in ownChild.PreTraverse())
      yield return child;
}
public IEnumerable<TreeNode> PostTraverse()
{
  foreach(var ownChild in children)
    foreach(var child in ownChild.PostTraverse())
      yield return child;
  yield return this;
}
```

```
tree.PostTraverse().ForEach(
  node => Console.WriteLine(
    string.Format(node.Height + ": " + node.Value)));
```

Потоци в Java

Interface-и: Iterator <T>, Iterable <T> и Stream<T>.
Чрез List <T>.stream, Stream.generate, Stream. iterate ,
Stream.of, Pattern . splitAsStream, Stream.Builder<T>,
StreamSupport.stream+ Iterable <T>.spliterator.

Потоци в Java

Interface-и: Iterator <T>, Iterable <T> и Stream<T>.
Чрез List <T>.stream, Stream.generate, Stream. iterate ,
Stream.of, Pattern . splitAsStream, Stream.Builder<T>,
StreamSupport.stream+ Iterable <T>.spliterator.

Полезни "декларативни" методи на Stream<T>:

- allMatch, anyMatch, noneMatch - квантори;
- limit - взимане на част от потока;
- max, min, reduce, collect - агрегати;
- skip - пропускане на елемент;
- concat - конкатениране на потоци;
- forEach - изпълняване за всеки елемент;
- filter , map, sorted - филтриране, преобразуване и др.

Hill climbing

```
static class Graph
{
  HashSet<Integer> vertices;
  Hashtable<Integer, ArrayList<GraphEdge>> outgoing;
```

Hill climbing

```
static class Graph
{
  HashSet<Integer> vertices;
  Hashtable<Integer, ArrayList<GraphEdge>> outgoing;
```

```
public Stream<Integer> HillClimbing(Integer start)
{
  Iterable<Integer> HillClimbingIterable =
    new Iterable<Integer>()
    {
      public Iterator<Integer> iterator()
      {
        return new Iterator<Integer>()
        {
    ...
  return StreamSupport.stream(
    HillClimbingIterable.spliterator(), false);
}
```

Hill climbing (продължение)

```
private OptionalInt previous = OptionalInt.empty();
private OptionalInt current = OptionalInt.of(start);
public boolean hasNext()
{
  return !previous.isPresent() || current.isPresent()
    && previous.getAsInt() < current.getAsInt();
}
public Integer next()
{
  previous = current;
  current = outgoing.get(current.getAsInt()).
    stream().map(edge -> edge.To).
      mapToInt(Integer::intValue).max();
  return previous.getAsInt();
}
```

Hill climbing (продължение)

```
private OptionalInt previous = OptionalInt.empty();
private OptionalInt current = OptionalInt.of(start);
public boolean hasNext()
{
  return !previous.isPresent() || current.isPresent()
    && previous.getAsInt() < current.getAsInt();
}
public Integer next()
{
  previous = current;
  current = outgoing.get(current.getAsInt()).
    stream().map(edge -> edge.To).
      mapToInt(Integer::intValue).max();
  return previous.getAsInt();
}
```

```
graph.HillClimbing(start).forEach(
  vertex -> System.out.println(vertex));
```

Генератори в JavaScript

Interface-и: Iterable , Iterator и IteratorResult .
Чрез Array, Set, Map, String и изрично чрез yield .

Генератори в JavaScript

Interface-и: Iterable , Iterator и IteratorResult .
Чрез Array, Set, Map, String и изрично чрез yield .

Полезни "декларативни" методи на Array и Iterable (с wu.js):

- every, some - квантори;
- slice - взимане на част от списъка/генератора;
- reduce - агрегати;
- shift /drop - пропускане на елементи;
- concat/chain - конкатениране на списъци/генератори;
- forEach - изпълняване за всеки елемент;
- filter , map, sort , reverse - филтриране, преобразуване, сортиране (само за списъци) и обръщане (само списъци).

Извличане на дати

```
var exp = /(\d{2})\.(\d{2})\.(\d{4})/g;
var match = exp.exec(text);
while (match != null)
{
  if((match[1] == "25" || match[1] == "31") &&
    match[2] == "12" && parseInt(match[3]) % 4 == 0)
      console.log(match[0]);
  match = exp.exec(text);
}
```

Извличане на дати

```
var exp = /(\d{2})\.(\d{2})\.(\d{4})/g;
var match = exp.exec(text);
while (match != null)
{
  if((match[1] == "25" || match[1] == "31") &&
    match[2] == "12" && parseInt(match[3]) % 4 == 0)
      console.log(match[0]);
  match = exp.exec(text);
}
```

```
RegExp.prototype.AllMatches =
  function*(input)
  {
    var match = this.exec(input);
    while (match != null)
    {
      yield match;
      match = this.exec(input);
    }
  }
```

Извличане на дати (продължение)

```
wu(/(25|31)\.12\.\d{2}(\d{2})/g.AllMatches(text)).
  filter(match => parseInt(match[2]) % 4 == 0).
    forEach(match => console.log(match[0]));
```

Извличане на дати (продължение)

```
wu(/(25|31)\.12\.\d{2}(\d{2})/g.AllMatches(text)).
  filter(match => parseInt(match[2]) % 4 == 0).
    forEach(match => console.log(match[0]));
```

В C# същото се реализира така:

```
Regex.Matches(text, @"(25|31)\.12\.\d{2}(\d{2})").
  Cast<Match>().
    Where(match =>
      (int.Parse(match.Groups[2].Value) % 4 == 0)).
        ForEach(match =>
          Console.WriteLine(match.Value));
```

Генератори в Python

Протокол на итераторите: __iter__ и next/__next__.
iterable : списъци, множества, n-орки, низове, файлове.
Чрез iter , range, permutations, combinations, yield и конструкция ... for ... in ... if ... .

Генератори в Python

Протокол на итераторите: __iter__ и next/__next__.
iterable : списъци, множества, n-орки, низове, файлове.
Чрез iter , range, permutations, combinations, yield и конструкция ... for ... in ... if ... .

Полезни "декларативни" методи за iterable и от itertools :

- all , any - квантори;
- sclice , isclice , takewhile, dropwhile - взимане на част и пропускане на елементи;
- max, min, sum, accumulate - агрегати;
- product - декартово произведение;
- chain, zip, tee - комбиниране и разделяне на генератори;
- filter , map, sorted, reversed - филтриране, преобразуване, сортиране и обръщане.

Прости числа

```
def IsPrime(number):
  return number > 1 and all(map(lambda lesser:
    number % lesser > 0, range(2, number- 1)))
def PrimeNumbers():
  return filter(IsPrime, count())
```

Прости числа

```
def IsPrime(number):
  return number > 1 and all(map(lambda lesser:
    number % lesser > 0, range(2, number- 1)))
def PrimeNumbers():
  return filter(IsPrime, count())
```

```
def Eratosthenes(iterator = count(2)):
  current = next(iterator)
  yield current
  yield from Eratosthenes(filter(lambda number:
    number % current > 0, iterator))
```

Прости числа

```
def IsPrime(number):
  return number > 1 and all(map(lambda lesser:
    number % lesser > 0, range(2, number- 1)))
def PrimeNumbers():
  return filter(IsPrime, count())
```

```
def Eratosthenes(iterator = count(2)):
  current = next(iterator)
  yield current
  yield from Eratosthenes(filter(lambda number:
    number % current > 0, iterator))
```

```
def EratosthenesNonRecursive():
  gen = []
  for p in filter(lambda n: all(map(lambda l:
    n % l > 0, gen)), count(2)):
      gen.append(p)
      yield p
```

Упражнения

- Генериране на обхват от числа.

Упражнения

- Генериране на обхват от числа.
- Генериране на двойки, тройки и n-орки от числа.

Упражнения

- Генериране на обхват от числа.
- Генериране на двойки, тройки и n-орки от числа.
- Генериране на редицата на Фибоначи.

Упражнения

- Генериране на обхват от числа.
- Генериране на двойки, тройки и n-орки от числа.
- Генериране на редицата на Фибоначи.
- Разпознаване на принадлежност към редици.