Generatory

Michał R. Przybyłek

1 Wstęp

Generator to potencjalnie nieskończony, leniwie obliczany, ciąg wartości. W zależności od tego, która ze stron decyduje o wygenerowaniu nowej wartości wyróżniamy następujące sytuacje:

- aktywny producent, aktywny konsument (np. Python)
- aktywny producent, pasywny konsument (np. Ruby)
- pasywny producent, aktywny konsument (np. Kogut)
- pasywny producent, pasywny konsument (np. serie w Common Lispie)

Aktywność tutaj oznacza możliwość wygenerowania/odebrania nowej wartości w dogodnym dla siebie momencie (intuicyjnie — „pętla" jest po stronie aktywnej), a pasywność konieczność wygenerowania/odebrania nowej wartości na żądanie (intuicyjnie — „pętla" nie jest po naszej stronie). Generatory z aktywnym producentem zazwyczaj łatwiej się definiuje, a generatory z aktywnym konsumentem zazwyczaj wygodniej się używa. Należy jednak zwrócić uwagę, że w ogólnej sytuacji generatory z aktywnym producentem i aktywnym konsumentem wymagają od implementacji istnienia mechanizmu równoważnego jednowejściowym kontynuacjom (np. wątków).

Oczywiście z punktu widzenia konsumenta — tj. użytkownika generatora — jest zupełnie obojętne czy strona producenta — tj. implementacji generatora — jest aktywna, czy pasywna. Dlatego generatory z aktywnym konsumentem będziemy po prostu nazywać „aktywnymi", a generatory z pasywnym konsumentem „pasywnymi".

2 Generatory aktywne

Dla generatorów aktywnych przyjmiemy w Javie następującą abstrakcję:

```
public interface AGenerator<T> {
    T value() throws StopException;
    AGenerator<T> next();
}
```

Metoda `value()` daje bieżący element generatora, bądź zgłasza wyjątek `StopException` (jest to ogólny wyjątek informujący o zaprzestaniu wykonywania jakiejś czynności), jeżeli na danej pozycji generator nie ma elementu (domyślnie zakładamy, że taki generator dobiegł końca). Metoda `next()` daje generator wskazujący na kolejną pozycję w generowanym ciągu.

Przykład 2.1 (NullGenerator)**.** Generator nie produkujący żadnych wartości:

```
public class NullGenerator<T> implements AGenerator<T> {
    public NullGenerator() {}

    public AGenerator<T> next() {
        return this;
    }

    public T value() throws StopException {
        throw new StopException();
    }
}
```

Przykład 2.2 (IntGenerator)**.** Generator produkujący wszystkie liczby całkowite począwszy od ustalonej:

```
public class IntGenerator implements AGenerator<Integer> {
    private int value;

    public IntGenerator(int value) {
        this.value = value;
    }
```

```
    public AGenerator<Integer> next() {
        return new IntGenerator(value+1);
    }

    public Integer value() throws StopException {
        return value;
    }
}
```

Przykład 2.3 (ListGenerator)**.** Generator produkujący po kolei wszystkie elementy danej listy:

```
public class ListGenerator<T> implements AGenerator<T> {
    private T value = null;
    private AGenerator<T> next = null;
    private ListIterator<T> iter;

    private ListGenerator(ListIterator<T> iter) {
        this.iter = iter;
    }

    public ListGenerator(List<T> list) {
        this.iter = list.listIterator();
    }

    private void force() throws StopException {
        try {
            if(iter != null) {
                value = iter.next();
                next = new ListGenerator<T>(iter);
                iter = null;
        }
        catch(NoSuchElementException e) {
            throw new StopException();
        }
    }

    public AGenerator<T> next() {
        try {
```

```
            force();
        }
        catch(StopException e) {
            return this;
        }
        return next;
    }

    public T value() throws StopException {
        force();
        return value;
    }
}
```

Ćwiczenie 2.1. Mając dany następujący interfejs reprezentujący funkcje:

```
public interface Function<S, T> {
    public T call(S x);
}
```

- zaimplementować metodę:

  ```
  <S, T> AGenerator<T> map(Function<S, T> fun,
                           AGenerator<S> generator)
  ```

  tworzącą generator wyjściowy przez sukcesywne aplikowanie funkcji `fun` do elementów generatora `generator`

- zaimplementować metodę:

  ```
  <S1, S2, T> AGenerator<T> map2(Function<S1, Function<S2, T>> fun,
                                 AGenerator<S1> generator1,
                                 AGenerator<S2> generator2)
  ```

  tworzącą generator wyjściowy przez sukcesywne aplikowanie dwuargumentowej funkcji `fun` do pary elementów (korzystamy tutaj z faktu, że funkcje zwracające funkcje są esencjalnie tym samym co funkcje dwuargumentowe), gdzie pierwszy element pary brany jest z generatora `generator1`, a drugi z `generator2`; generator wyjściowy się kończy, jeżeli kończy się którykolwiek z generatorów wejściowych

- zaimplementować metodę:

  ```
  <T> AGenerator<T> unfold(Function<T, T> fun, T init)
  ```

  tworzącą generator, którego elementami są elementy ciągu: `init`, `fun.call(init)`, `fun.call(fun.call(init))`, ...

3 Generatory pasywne

Dla generatorów z aktywnym producentem i pasywnym konsumentem w Javie przyjmiemy następującą abstrakcję:

```
public interface PGenerator<T> {
    public void generate(Code<T> code) throws StopException;
}
```

gdzie interfejs `Code<T>` ma następującą postać:

```
public interface Code<T> {
    public void execute(T x) throws StopException;
}
```

Metoda `generate` dostaje od klienta kawałek kodu z miejscem na przekazanie wartości i wykonuje ten kod dla każdej kolejnej wygenerowanej wartości. Dla usprawnienia komunikacji pomiędzy klientem a implementacją, zezwalamy na to, aby kod klienta mógł rzucić wyjątek `StopException` informujący generator o zaprzestaniu generowania dalszych wartości (tak rzucony wyjątek powinien przelecieć przez samą metodę `generate`).

Przykład 3.1 (NullGenerator)**.** Generator nie produkujący żadnych wartości:

```
public class NullGenerator<T> implements PGenerator<T> {
    public void generate(Code<T> code) throws StopException {}
}
```

Przykład 3.2 (IntGenerator)**.** Generator produkujący wszystkie liczby całkowite począwszy od ustalonej:

```
public class IntGenerator implements PGenerator<Integer> {
    private int start;

    public IntGenerator(int start) {
        this.start = start;
    }

    public void generate(Code<Integer> code) throws StopException {
        for(int i = start ; true ; i++) {
            code.execute(i);
        }
    }
}
```

Przykład 3.3 (ListGenerator)**.** Generator produkujący po kolei wszystkie elementy danej listy:

```
public class ListGenerator<T> implements PGenerator<T> {
    private List<T> list;

    public ListGenerator(List<T> list) {
        this.list = list;
    }

    public void generate(Code<T> code) throws StopException {
        for(T elem : list) {
            code.execute(elem);
        }
    }
}
```

Ćwiczenie 3.1. Mając dany interfejs `Function<S, T>` jak w poprzednim ćwiczeniu

- zaimplementować metodę:

```
<S, T> PGenerator<T> map(Function<S, T> fun,
                         PGenerator<S> generator)
```

tworzącą generator wyjściowy przez sukcesywne aplikowanie funkcji `fun` do elementów generatora `generator`

- pomyśleć dlaczego tak trudno zaimplementować jest dla pasywnych generatorów metodę (dużo ogólniejsze rozwiązanie tego problemu znajduje się w kolejnym rozdziale):

```
<S1, S2, T> PGenerator<T> map2(Function<S1, Function<S2, T>> fun,
                               PGenerator<S1> generator1,
                               PGenerator<S2> generator2)
```

tworzącą generator wyjściowy przez sukcesywne aplikowanie dwuargumentowej funkcji `fun` do pary elementów, gdzie pierwszy element pary brany jest z generatora `generator1`, a drugi z `generator2`; generator wyjściowy się kończy jeżeli kończy się którykolwiek z generatorów wejściowych

- zaimplementować metodę:

```
<T> PGenerator<T> unfold(Function<T, T> fun, T init)
```

tworzącą generator, którego elementami są elementy ciągu: `init`, `fun.call(init)`, `fun.call(fun.call(init))`, ...

4 Generatory z aktywnym producentem i aktywnym konsumentem

Zamiast zaproponować nowy typ generatorów, pokażemy jak konwertować dwa opisane wyżej typy generatorów na siebie. Jak już powiedziano we wstępie, w ogólnej sytuacji możliwość tworzenia generatorów z aktywnym producentem i aktywnym konsumentem wymaga istnienia od języka silnych mechanizmów do kontroli przepływu wykonania programu, nic dziwnego zatem, że do konwersji generatorów pasywnych na aktywne będziemy musieli użyć wątków.

Zacznijmy jednak od prostej sytuacji — utworzenia generatora pasywnego na podstawie generatora aktywnego:

```
public class PassiveGenerator<T> implements PGenerator<T> {
    private AGenerator<T> generator;

    public PassiveGenerator(AGenerator<T> generator) {
        this.generator = generator;
    }

    public void generate(Code<T> code) {
        AGenerator<T> generator = this.generator;
        T value;
        while(true) {
            try {
                value = generator.value();
            }
            catch(StopException e) {break;}
            code.execute(value);
            generator = generator.next();
        }
   }
}
```

Aby skonwertować generator pasywny na aktywny postąpimy w następujący sposób: odpalimy generator pasywny w nowym wątku, a jako kod przekażemy mu synchronizowaną komórkę pamięci — do takiej komórki można zapisać wartość, ale ponowna próba zapisu przed jej opróżnieniem powoduje wstrzymanie procesu; na każde żądanie obliczenia nowej wartości będziemy wyciągać wartość z komórki. Zauważmy, że przy tej definicji generator pasywny zawsze będzie produkował dwie wartości z wyprzedzeniem. Należy pamiętać przy implementacji tej konwersji o właściwym zwalnianiu nieprzydatnych już zasobów (w szczególności jeżeli nie ma już referencji do generatora aktywnego, wątek obsługujący generator pasywny powinien zostać zwolniony).

```
public class Cell<T> {
    private T value;
    private boolean consumed;
    private boolean stop;

    public Cell() {
```

```
    this.value = null;
    this.consumed = true;
    this.stop = false;
}

public synchronized void put(T val) throws StopException {
    while(true) {
        if(stop == true) {
            throw new StopException();
        }
        if(consumed == true) {
            break;
        }
        try {
            this.wait();
        } catch (InterruptedException e) { }
    }
    consumed = false;
    value = val;
    this.notifyAll();
}

public synchronized T get() throws StopException  {
    while(true)  {
        if(consumed == false) {
            break;
        }
        if(stop == true) {
            throw new StopException();
        }
        try {
            this.wait();
        } catch (InterruptedException e) { }
    }
    T val = value;
    value = null;
    consumed = true;
    this.notifyAll();
```

```
        return val;
    }

    public synchronized void stop() {
        this.stop = true;
        this.notifyAll();
    }
}

public class ActiveCode<T> implements Code<T> {
    private Cell<T> cell;

    public ActiveCode(Cell<T> cell) {
        super();
        this.cell = cell;
    }

    public void execute(T x) throws StopException {
        cell.put(x);
    }
}

public class ActiveThread<T> extends Thread {
    private PGenerator<T> passive;
    private Code<T> code;
    private Cell<T> cell;

    public ActiveThread(PGenerator<T> passive, Code<T> code, Cell<T> cell) {
        super();
        this.passive = passive;
        this.code = code;
        this.cell = cell;
    }

    public void run() {
        try {
            passive.generate(code);
        }
```

```
        catch (StopException e) {}
        cell.stop();
    }
}

public class ActiveGenerator<T> implements AGenerator<T> {
    private Cell<T> cell;
    private T value = null;
    private AGenerator<T> next = null;

    private ActiveGenerator(Cell<T> cell) {
        this.cell = cell;
    }

    public ActiveGenerator(final PGenerator<T> passive) {
        this.cell = new Cell<T>();
        Code<T> code = new ActiveCode<T>(cell);
        new ActiveThread<T>(passive, code, cell).start();
    }

    private void force() throws StopException {
        if(cell != null) {
            value = cell.get();
            next = new ActiveGenerator<T>(cell);
            cell = null;
        }
    }

    public AGenerator<T> next() {
        try {
            force();
        }
        catch(StopException e) {
            return this;
        }
        return next;
    }
```

```
    public T value() throws StopException {
        force();
        return value;
    }

    protected void finalize() {
        if(cell != null) {
            cell.stop();
        }
    }
}
```

Ćwiczenie 4.1. Przekształcić tak powyższy kod, aby pasywny generator generował zawsze dokładnie tyle wartości, ile jest potrzebnych.