Rozszerzalny, szybki i bezpieczny silnik ewaluacji wyrażeń w języku Scala

Roman Janusz
opiekun: dr inż. Arkadiusz Janik

30 września 2014

Plan

1. ogólne nakreślenie zagadnienia, interesujące przypadki użycia
2. konkretyzacja wymagań
3. przegląd istniejących rozwiązań dla JVM - rozpoznanie braków
4. przegląd potencjalnych kierunków dla nowej implementacji
5. język Scala
6. makra w Scali i ich wykorzystanie do implementacji mechanizmów bezpieczeństwa
7. konfiguracja mechanizmów bezpieczeństwa
8. potencjalne kierunki rozwoju

Zagadnienie

Zagnieżdżanie środowisk dynamicznego wykonywania programów i skryptów

Wykorzystanie - ogólnie:

- Szybka metoda dostarczenia dużej i szczegółowej kontroli nad systemem macierzystym, np. w celu jego dynamicznej konfiguracji.
- Wykorzystanie systemu jako kontenera do dynamicznego wykonywania innych programów w specjalnym środowisku, często związanym z konkretną dziedziną.

Formy:

- środowiska zagnieżdżania skryptów (programów)
- silniki ewaluacji wyrażeń (funkcji)

Interesujące przypadki użycia

- Dynamiczna konfiguracja np. wiadomości powitalnej wyświetlanej użytkownikom portalu internetowego po zalogowaniu się.
  Wyrażenie jest szablonem wiadomości odwołującym się do danych konkretnego użytkownika, np.
  `Witaj, ${person.name} ${person.surname}!`
- Dynamiczne zlecanie zadań masowego przetwarzania danych w systemach zarządzających takimi danymi.
  Wyrażenia definiują aplikowane transformacje danych (np. operacje `map` i `reduce`).

Wymagania

Cel: stworzenie silnika wyrażeń dla platformy maszyny wirtualnej Javy spełniającego następujące wymagania:

- duża **wydajność ewaluacji**, nawet kosztem wydajności kompilacji
  - wyrażenia tworzone rzadko, ewaluowane masowo
- mechanizmy **bezpieczeństwa**, czyli ograniczania konstrukcji językowych i API dostępnego w wyrażeniach
  - wymaganie koniecznie w sytuacji, gdy chcemy umożliwić tworzenie wyrażeń użytkownikom interfejsu graficznego
- elastyczność i rozszerzalność języka wyrażeń konkurencyjna z obecnie istniejącymi rozwiązaniami dla JVM

Obecnie istniejące rozwiązania

Silniki wyrażeń, np.

- Java Unified Expression Language
- Spring Expression Language
- MVFLEX Expression Language

W większości interpretowane i dynamicznie typowane:

- słaba wydajność ewaluacji
- niemożność statycznej analizy kodu pod kątem bezpieczeństwa
- trudne wykrywanie błędów w kodzie

Wniosek: **potrzebny język statycznie typowany**

Język Scala

- Obiektowo - funkcyjny, statycznie typowany lecz zwięzły i elastyczny
- Kompilowany do bytecode'u, kompatybilny z Javą
- Ogromne możliwości tworzenia bogatego API - Domain Specific Languages
- Metaprogramowanie - możliwość wpływu na proces kompilacji kodu (makra)
- Kompilator napisany w Scali - działa w JVM

Nowe podejście

- Wyrażenie = kod w Scali (funkcja danych wejściowych na wyjściowe)
- Kompilowane w locie do bytecode'u - wydajność natywnego kodu!
- Walidacja wyrażenia za pomocą specjalnego makra w Scali w czasie kompilacji - nie wpływa na wydajność ewaluacji!

Walidacja wyrażeń pod kątem bezpieczeństwa

Celem jest uniemożliwienie użytkownikowi stworzenia wyrażenia zawierającego złośliwy kod.

Potrzebujemy:

- Walidacji pod kątem używanych konstrukcji językowych - np. zabraniamy używania pętli
- Walidacji pod kątem wywoływanych metod - np. zabraniamy wywołania `System.exit`

Makra

- Potężny mechanizm metaprogramowania w Scali
- Nie pozwalają na rozluźnienie składni i systemu typów Scali
- Pozwalają wstrzelić się w proces kompilacji i używać API kompilatora
- Programista makra operuje na kodzie w Scali przekazanym do makra jako AST (Abstract Syntax Tree)
- Dostęp do informacji o składni, typach, symbolach (czyli np. wywoływanych metodach)
- Wymuszanie błędów kompilacji
- Możliwość modyfikacji kodu - faktyczne metaprogramowanie
- Wykorzystywane w tej pracy do statycznej analizy kodu oraz konstrukcji języka domenowego techniką wirtualizacji języka

Makra

```
import reflect.macros.Context
import scala.language.experimental.macros

def simpleValidate[T](expr: T): T = macro simpleValidate_impl[T]

def simpleValidate_impl[T](c: Context)(expr: c.Expr[T]) = {
  import c.universe._

  // the symbol of ‘java.lang.Object.wait‘ method
  val forbidden = typeOf[Object].member(newTermName("wait"))

  // traverse through all subtrees of AST associated with ‘expr‘
  expr.tree.foreach { tree =>
    if (tree.symbol == forbidden) {
      // force compilation error
      c.error(tree.pos, "You cannot invoke ‘wait‘ method!")
    }
  }
}
```

Konfiguracja walidatora

- Administrator lub programista decyduje o tym, jakie metody mogą być używane w wyrażeniach definiując listę dozwolonych lub zabronionych metod (ACL).
- Lista jest definiowana w pliku konfiguracyjnym i jest ona poprawnym kodem w Scali, dzięki czemu jest np. rozumiana przez IDE
- Kod ten nie jest nigdy wykonywany - jest on interpretowany przez inne makro.

Konfiguracja walidatora

```
deny {

  on { any: Any =>
    any.getClass
    any.hashCode
  }

} ++ allow {

  StringContext.apply _
  String.CASE_INSENSITIVE_ORDER
  Some.apply _
  allStatic[String].membersNamed.valueOf

  on { s: String =>
    s.all.constructors
    s.length
    s.substring(_: Int)
    s.reverse
  }

}
```

Potencjalne kierunki rozwoju

- Edytor wyrażeń wykorzystujący Scala Presentation Compiler
- Kompilacja w zewnętrznym procesie
- Bardziej szczegółowe strategie walidacji
- Rozszerzenie do możliwości pisania skryptów (nie tylko wyrażeń)