scalaz-stream

Radek Miček

Spino Co:

Obsah

- 1. Jednoduché zpracování proudů
  - C# + standardní knihovna vs Scala + scalaz-stream
- 2. Použití knihovny scalaz-stream v naší aplikaci
  - O naší aplikaci
  - Architektura naší aplikace
  - Příklad: Služba pro ukládání historie hovorů
- 3. Srovnání
  - S aktory (známými z Erlangu)
  - S líným vstupem (známým z Haskellu)

1. Jednoduché zpracování proudů

scalaz-stream

- Knihovna pro zpracování proudů dat
  - Proud dat = posloupnost dat, kde data přicházejí postupně a může jich být i nekonečně mnoho.
- Společnost Spinoco knihovnu scalaz-stream používá a spoluvyvíjí
- https://github.com/scalaz/scalaz-stream

Př.: Čtení souboru (C# – TextReader)

- Zadání: Načíst prvních 100 řádek souboru.

```
var result = new List<string>();
using (var r = File.OpenText("a.txt"))
{
    int i = 0;
    var line = "";
    while ((line = r.ReadLine()) != null && i < 100)
    {
        result.Add(line);
        i++;
    }
}
```

Př.: Čtení souboru (C# – TextReader)

- Zadání: Načíst prvních 100 řádek souboru, jenž začínají slovem 'Spinoco'.

```
var result = new List<string>();
using (var r = File.OpenText("a.txt"))
{
    int i = 0;
    var line = "";
    while ((line = r.ReadLine()) != null && i < 100)
    {
        if (line.StartsWith("Spinoco"))
        {
            result.Add(line);
            i++;
        }
    }
}
```

C# – TextReader

- Nepřehledný kód – měli bychom oddělit filtrování a omezování počtu řádek
  - Filtrování dle predikátu (např. řádka začíná slovem 'Spinoco') nebo omezování počtu vrácených řádek (např. 100 prvních řádek) jsou často požadované funkcionality.
  - Mohly by tedy existovat funkce, jenž transformují daný TextReader – např.:

    `TextReader FilterLines(TextReader r, Predicate<string> p)` nebo

    `TextReader LimitLines(TextReader r, int n).`
  - Standardní knihovna však funkce pro transformaci TextReaderu neobsahuje (a počet metod ve třídě TextReader nijak neusnadňuje psaní vlastních).
- TextReader je pouze pro čtení textových proudů dat
  - Proč se ale omezovat na znaky a řetězce?

C# – IEnumerable / Java – Iterable

- Rozhraní pro procházení posloupností prvků.

Př.: Čtení souboru (C# – IEnumerable)

- Zadání: Načíst prvních 100 řádek souboru, jenž mají prefix 'Spinoco'.

```
var result = File.ReadLines("a.txt")
    .Where(line => line.StartsWith("Spinoco"))
    .Take(100)
    .ToList();
```

- Obyčejné volání metod

```
var result =
    (from line in File.ReadLines("a.txt")
     where line.StartsWith("Spinoco")
     select line
    ).Take(100).ToList();
```

- Speciální syntax připomínající SQL

Rozbor: Čtení souboru (C# – IEnumerable)

- Zadání: Načíst prvních 100 řádek souboru, jenž mají prefix 'Spinoco'.

```
var result = File.ReadLines("a.txt")
    .Where(line => line.StartsWith("Spinoco"))
    .Take(100)
    .ToList();
```

- Kód je přehledný
- IEnumerable je obvykle „líné“
  - Ale v tomto případě, se soubor otevře ihned
  - Metoda `ReadLines` není líná (.NET Framework 4.5.1) viz http://referencesource.microsoft.com/#mscorlib/system/io/ReadLinesIterator.cs
- Metody `Where` a `Take` transformují IEnumerable

Př.: Zápis souborů (C# – IEnumerable)

- Zadání: Buď `lines` instance `IEnumerable<string>`. Do prvního souboru chceme zapsat všechny řádky z `lines` a do druhého souboru jen ty, které začínají slovem 'Spinoco'.

```
File.WriteAllLines("a.txt", lines);
File.WriteAllLines(
    "b.txt",
    lines.Where(line => line.StartsWith("Spinoco")));
```

  - `lines` čteme dvakrát

Př.: Zápis souborů (C# – IEnumerable)

- Zadání: Buď `lines` instance `IEnumerable<string>`. Do prvního souboru chceme zapsat všechny řádky z `lines` a do druhého souboru jen ty, které začínají slovem 'Spinoco'.

```
using (var w1 = File.CreateText("a.txt"))
using (var w2 = File.CreateText("b.txt"))
{
    foreach (var line in lines)
    {
        w1.WriteLine(line);
        if (line.StartsWith("Spinoco"))
        {
            w2.WriteLine(line);
        }
    }
}
```

  - `lines` čteme jen jednou, ale řešení už není přehledné

Př.: Uvolňování zdrojů (C# – IEnumerable)

- Zadání: Buď `lines` instance `IEnumerable<string>`. Chceme vypsat každý druhý prvek `lines`.

```
using (var e = lines.GetEnumerator())
{
    for (e.MoveNext(); e.MoveNext(); e.MoveNext())
    {
        Console.WriteLine(e.Current);
    }
}
```

  - Bez `using` by hrozilo, že zdroje alokované enumerátorem nebudou uvolněny.

C# – IEnumerable

- Přehledný kód pro čtení a filtrování
- Speciální podpora v kompilátoru (`yield return` a `yield break`)
- Nedostatečně řeší zápis
  - Lze zlepšit
- Uvolňování zdrojů závisí na kázni programátora
  - Lepší by bylo, kdyby knihovna byla navržena tak, aby nemohlo dojít k úniku zdrojů
  - Typové systémy některých jazyků (ATS, Rust, ParaSail) umí pomoci
  - Scala ani C# takový typový systém nemají

Jak scalaz-stream řeší uvolňování zdrojů?

Jak scalaz-stream řeší uvolňování zdrojů?

- Sama je alokuje i dealokuje

Př.: Čtení souboru (scalaz-stream)

- Zadání: Načíst prvních 100 řádek souboru, jenž mají prefix 'Spinoco'.

```
val result = io.linesR("a.txt")
    .filter(_.startsWith("Spinoco"))
    .take(100)
    .runLog.run
```

- Výrazy

```
io.linesR("a.txt")
io.linesR("a.txt").filter(_.startsWith("Spinoco"))
io.linesR("a.txt").filter(_.startsWith("Spinoco")).take(100)
```

  mají typ `Process[Task, String]`, a proto jim budeme říkat procesy

- Proces je „líný“ – na rozdíl od IEnumerable to platí vždy
  - Soubor se neotevře, dokud to není třeba
  - Otevření souboru a načtení řádek je vynuceno voláním `runLog.run`
  - Metody `filter` a `take` transformují proces

Př.: Čtení souboru (scalaz-stream)

- Zadání: Načíst prvních 100 řádek souboru, jenž mají prefix 'Spinoco'.

```
val result = io.linesR("a.txt")
    .filter(_.startsWith("Spinoco"))
    .take(100)
    .runLog.run
```

- scalaz-stream

```
var result = File.ReadLines("a.txt")
    .Where(line => line.StartsWith("Spinoco"))
    .Take(100)
    .ToList();
```

- C# - IEnumerable

Spino co:

Př.: Transformace řádků (scalaz-stream)

- Zadání: Načíst délky řádek v souboru.

```
val result = io.linesR("a.txt")
    .map(_.length)
    .runLog.run
```

- `map` transformuje řádky pomocí dané funkce
  - V našem příkladu je funkce: řetězec ↦délka řetězce
- Typy výrazů

```
io.linesR("a.txt")              : Process[Task, String]
io.linesR("a.txt").map(_.length): Process[Task, Int]
```

Př.: Čtení dvou souborů

- Zadání: Soubor mesta.txt obsahuje hlavní města zemí ze souboru zeme.txt (ve stejném pořadí). Cílem je spárovat země s jejich hlavními městy a výsledek zapsat do jiného souboru.

```
val result = io.linesR("zeme.txt")
    .zip(io.linesR("mesta.txt"))
    .runLog.run
```

- `result` je posloupnost dvojic země – město
- `zip` kombinuje dva procesy dohromady
  - Opakovaně: Přečte jeden prvek z prvního vstupu (např. zeme.txt), jeden prvek ze druhého vstupu (např. mesta.txt), vytvoří dvojici a pošle ji dál
- Oba soubory jsou automaticky otevřeny i uzavřeny

Př.: Výpis souboru (scalaz-stream)

- Zadání: Vypsat řádky souboru.

```
io.linesR("a.txt")
    .to(io.stdOutLines)
    .run.run
```

- `io.stdOutLines` je sink, který vypisuje příchozí řetězce jako řádky na standardní výstup
- Řádky do sinku posíláme metodou `to`
  - Použitím metody `to` řádky „zmizí" v sinku, místo nich zůstane hodnota `()` typu `Unit` (typ `Unit` je něco jako typ `void`, ale na rozdíl od něj má jedinou hodnotu)
- Metoda `run` (první) ignoruje výstup procesu (na rozdíl od metody `runLog`, jenž výstup procesu ukládá a vrací)
  - Proč používáme `run` místo `runLog`?

Př.: Zápis souborů (scalaz-stream)

- Zadání: Buď `lines` instance `Process[Task, String]`. Do prvního souboru chceme zapsat všechny řádky z `lines` a do druhého souboru jen ty, které začínají slovem 'Spinoco'.

```
def linesW(filename: String): Sink[Task, String] =
    io.fileChunkW(filename).pipeIn(text.utf8Encode)

lines
    .observe(linesW("a.txt"))
    .filter(_.startsWith("Spinoco"))
    .to(linesW("b.txt"))
    .run.run
```

- Metoda `observe` pošle řádky do sinku
  - Na rozdíl od metody `to` řádky nezmizí (dále s nimi můžeme pracovat)
- `linesW` je sink, který zapisuje řádky do souboru (bohužel není součástí knihovny)

Spino co:

Co knihovna scalaz-stream umí?

- Transfromace prvků
  - `map`, `collect`, ...
- Filtrování
  - `filter`, `drop`, `take`, `dropWhile`, `takeWhile`, `dropLast` ...
- Agregování
  - `reduce`, `fold`, `exists`, `forall`, ...
- Kombinování procesů
  - `append`, `repeat`
  - deterministicky: `pipe`, `zip`, `zipWith`, `tee`, ...
  - nedeterministicky: `merge`, `wye`, `mergeN`, ...

Spino co:

Co knihovna scalaz-stream umí?

- Bezpečná práce se zdroji
  - `linesR`, `fileChunkR`, `fileChunkW`, `resource`, ...
- Odlišení procesů s vedlejšími efekty (pomocí Task)
  - `Process[Task, A]`, `Sink[Task, A]`, `Channel[Task, A, B]`, ...
- A další
  - `flatMap`, `intersperse`, `awakeEvery`, ...
  - ...

scalaz-stream vs C# – IEnumerable

- scalaz-stream
  - automatická alokace i dealokace zdrojů
  - nedeterministické čtení z více vstupů
  - lepší podpora zápisu
  - odlišení procesů s vedlejšími efekty pomocí typových signatur
  - větší množství funkcí pro skládání a transformace procesů
- C# – IEnumerable
  - speciální podpora v kompilátoru

2. Použití knihovny scalaz-stream v naší aplikaci

Naše vize

- Jedna aplikace pro různé komunikační kanály
  - Telefonní hovory
  - Chat
  - E-mail
  - SMS
  - Sociální sítě
- Funkce
  - Historie komunikace
  - On-line monitoring
  - Statistiky
  - Rozdělování úkolů a plánování času

Architektura naší aplikace

- Front end
  - JavaScript, AngularJS, HTML5
- Back end
  - Scala, scalaz-stream, Play
  - Apache Kafka, Apache Cassandra

Hledáme nové kolegy

- 1. Vývojáře ve Scale
- 2. Vývojáře v JavaScriptu

- U nás se skutečně programuje funkcionálně
- Je možné pracovat na zkrácený úvazek

- Pro více informací pište na

create@spinoco.com

Back end

- Webový server + supervizor + služby
  - Propojeno pomocí procesů z knihovny scalaz-stream
- Webový server autentizuje uživatele a jejich požadavky předává konkrétním službám
- Supervizor startuje služby, dohlíží na ně a v případě potřeby je restartuje
- Služba
  - Umí obnovit stav po restartu
  - Vstupem může být výstup jiných služeb

Př.: Služba pro ukládání historie hovorů

- K událostem o telefonních hovorech doplňuje uživatele, ukládá je do databáze a poskytuje je dalším službám (např. pro výpočet statistik)
- Vstupy
  - Události o telefonních hovorech
  - Stav telefonů – kdo je právě přihlášen na každém telefonu a od kdy (spojitý proces – prvek načtený z procesu obsahuje aktuální stav telefonů, vždy je možné načíst prvek)

Př.: Služba pro ukládání historie hovorů

- Párování událostí o hovorech s uživateli:

```
val callEventsWithUser = callEvents
    .zip(phonesWithUser.continuous)
    .map(findUserFromPhoneMap)
    .through(pairFromHistory(findUser).onlyIf(_.user.isEmpty))
```

- `callEvents` je proces událostí o hovorech
- `phonesWithUser.continuous` je spojitý proces s aktuálním stavem telefonů
- `zip` vytvoří proces dvojic (událost o hovoru, aktuální stav telefonů)
- `findUserFromPhoneMap` bere dvojici (událost o hovoru, aktuální stav telefonů) a pokud může, přiřadí události o hovoru uživatele
- `through` je jako `map`, ale navíc může provést vedlejší efekt (např. hledání v DB)
- `findUser` najde v databázi uživatele, který byl přihlášen na telefonu v daný čas
- `pairFromHistory` přiřadí uživatele z DB události o hovoru

Př.: Služba pro ukládání historie hovorů

- Ukládání událostí o hovorech do databáze:

```
callEventsWithUser
    .observe(storeEvent)
```

- `storeEvent` je sink, jenž uloží událost o hovoru do databáze
  - Pokud událost projde za `observe`, je garantováno, že je uložena v databázi

Obnovení stavu služby po restartu

- Komunikace mezi službami je posílána přes Apache Kafku
  - Apache Kafka je systém pro posílání zpráv
  - Podobá se např. RabbitMQ, na rozdíl od něj však garantuje trvanlivost zpráv (při vhodném nastavení) – tj. zprávy se neztratí, když zápis do Kafky uspěje
  - Zprávy se publikují (zapisují) do tzv. topiců
  - Publikováním je zprávě automaticky přiřazeno pořadové číslo (offset)
  - Více služeb může publikovat do jednoho topicu
  - Více služeb může číst z jednoho topicu (i zprávy s různými offsety)
- Stav služby se jednou za čas ukládá do databáze společně s offsety naposledy zpracovaných zpráv z Kafky

Obnovení stavu služby po restartu

- Po restartu služby se načte poslední uložený stav služby z databáze společně s offsetem naposledy zpracované zprávy z Kafky
- Služba se připojí ke Kafce a začne číst z následujícího offsetu

Spino co:

Výhody knihovny scalaz-stream

- Požadovaná funkcionalita se poskládá použitím jednoduchých a znovupoužitelných funkcí
  - Kód je lépe čitelný
- Služby nezávisí na implementaci databáze nebo Kafky
  - Služby ani neví, že používají Apache Cassandru nebo Apache Kafku – při konstrukci dostávají obyčejné procesy
  - Výrazně usnadňuje testování (např. lze snadno nasimulovat selhání procesu, ale je těžké nasimulovat selhání Kafky nebo Cassandry)
- Typy popisují kód

3. Srovnání

Spino co:

Aktory

- Aktor je objekt, jenž zprávy zpracovává sekvenčně
  - Aktor přijímá zprávy, na základě přijatých zpráv mění svůj vnitřní stav, provádí vedlejší efekty a posílá zprávy jiným aktorům
- Služba by byla aktor, kdybychom aktory používali

Spino co:

scalaz-stream vs aktory

- scalaz-stream
  - Pull – z procesu se čte, jen když je to třeba
  - Procesy se lehce kombinují pomocí jednoduchých funkcí – existující proces můžeme využít jako součást procesů, jenž vytváříme
  - Procesy se automaticky starají o alokaci a dealokaci zdrojů
  - Z typu procesu jde poznat jeho vstup, výstup a i to, zda provádí vedlejší efekty
- Aktory
  - Push – aktor běží a posílá zprávy dalším aktorům, i když třeba na žádné zprávy nečekají
  - Některé implementace aktorů (Erlang, Akka, Orleans) mají zabudovanou podporu pro dohled nad aktory (supervizor)
  - Většina implementací aktorů nevyužívá typový systém – z typu aktoru typicky nejde poznat, zda aktor dělá vedlejší efekty nebo jaké zprávy posílá jiným aktorům

Líný vstup v Haskellu

- Funkce `readFile` načte obsah souboru jako řetězec (tj. seznam znaků)
  - Ve skutečnosti čtení souboru probíhá až při žádosti o vyhodnocení načteného řetězce
- Potíž je v tom, že soubor není možné zavřít
  - Soubor se zavře po dočtení nakonec nebo ho uzavře GC
- Výhoda líného vstupu spočívá v tom, že se souborem je možné pracovat jako s obyčejným seznamem (což je velice pohodlné)
  - Některým programům navíc nevadí, že soubor není uzavřen ihned

Spinoco:

Děkuji za pozornost!