Aktorový model

Róbert Novotný
UINF/KOPR,
zima 2018

Základný problém vláknovej konkurentnosti

Ako spravovať prístup k zdieľanej pamäti?

- mašinéria zámkov, monitorov…
- ťažko sa ladí
  - ťažké reprodukovať problémy
  - správanie je často nepredvídateľné
  - vývojár musí uvažovať na mnohých frontoch

Škálovateľnosť má hranice

Škálovateľnosť má hranice

- vlákna od istého momentu neškálujú
  - 10000 vlákien anyone?
- potrebujeme škálovať v oboch rovinách:
  - horizontálne: pridávame uzly / jadrá / CPU…
  - vertikálne: zvyšujeme frekvenciu, dodávame RAM…

Potreba iných modelov!

Všetkému je na vine...

- zrušme prístup k zdieľaným dátam a bude dobre!
  - kde nie sú zdieľané dáta, nie sú deadlocky, prepisy, ...
- žiadne zdieľané inštančné premenné...
- ako potom programovať?

Potreba iných modelov!

Aktor = výpočtová entita

since 1973

- vie prijímať a odosielať správy
- po prijatí správy môže:
  - odoslať konečný počet správ iným aktorom
  - vytvoriť konečný počet nových aktorov
  - zmeniť svoj stav
  - určiť správanie pre správy prijaté v budúcnosti
- možnosti sa môžu diať v ľubovoľnom poradí
  - a dokonca paralelne

Správa

- ľubovoľný **nemenný** objekt posielaný medzi aktormi
- jediný spôsob výmeny informácií / zdieľania dát medzi aktormi
- medzi aktormi neexistujú žiadne zdieľané dáta!

```
public class Správa {
   private final String data;
   public Správa(String data)
   {
      this.data = data;
   }
   public String getData()
   {
      return this.data;
   }
}
```

Asynchronicita

- správy sú odosielané **asynchrónne**
- nečaká sa na prijatie
  - **fire-and-forget**
- nezáleží na poradí prijatých správ
  - inšpirácia z packet-oriented systems (UDP?)
- v praxi: podporuje sa aj synchronicita

Buffrovanie správ: áno alebo nie?

- jeden názor: správy sa nebufferujú: všetko sa deje naraz
- kompromisný: „správy sa bufferujú v éteri"
- prakticky [Akka]: správy sa radia do buffera / frontu

Implementácie aktorového modelu

- Erlang
  - funkcionálny jazyk
  - namiesto zdieľaného stavu odovzdávanie správ
- Akka
  - open source framework pre aktorov
  - implementácie: Java / Scala

Aktory v Akke = akktory

- objekty so stavom a správaním
  - deja vu z OOP
- stav i správanie sa menia len na základe prijatej správy
- aktor v Akke: implementovaný ako vlákno
  - to nás nemá čo zaujímať ;-)

Akktor a Java

- dedí od triedy **AbstractActor**
- prekrýva metódu **createReceive**
- definuje **pravidlá** pre správy:

„ak príde správa typu X, vykonaj toto"

Akktor a Java: pravidlá pre správy

- dátový typ mapovaný na funkciu spracovávajúcu objekt správy

```
.match(String.class, m -> System.out.println(m))
```

```
import akka.actor.*; import akka.event.*;
public class ExampleActor extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder(
            .match(String.class, msg -> System.out.println(msg))
            .build();
    }
}
```

Akktory a ich vytváranie cez Props

- na vytváranie aktora sa nepoužíva **new ...()!**
- musíme deklarovať **Props** („rekvizity")

Props je mechanizmus vytvárania akktora

```
import akka.actor.*; import akka.event.*;

public class ExampleActor extends AbstractActor {
    @Override
    public Receive createReceive() {
        ...
    }

    public static Props props() {
        return Props.create(ExampleActor.class);
    }
}
```

Hello World akktor

```
public static void main(String[] args) {
    ActorSystem system = ActorSystem.create();
    ActorRef actor = system.actorOf(ExampleActor.props());

    for (int i = 0; i < 5; i++) {
        actor.tell("Hi at " + new Date(), ActorRef.noSender());
    }
}
```

povedz správu!

správu neposiela aktor

Thread-safety aktorových tried

- akktor spracuje správu pred spracovaním ďalšej správy
  - v súlade s "happens-before" zásadou z JVM
  - inštančné premenné akktora nemusia byť volatile a pod.
- implementácia cez ideu **mailboxu**

Mailboxy a buffrovanie

- akktor má mailbox v ktorom sa kopia správy
  - každý akktor má vlastný
  - nedajú sa zdieľať
- správy defaultne radené podľa času odoslania
- možno využiť prioritný mailbox:
  - radenie podľa priority
- mailbox je front!
  - aktor vidí **len** jeho začiatok

Dôležité rysy aktora

- aktor spracováva v danom čase **len jednu správu**
  - ostatné sa kopia vo fronte
  - dlhotrvajúce spracovanie správy znamená dlhšie čakanie!
- správy z aktora A do aktora B **garantujú poradie** doručenia
  - ak nemáme špeciálny typ mailboxu
  - nespoliehajme sa na poradie doručenia správ z rozličných aktorov!

Typické nasadenie

- rozsiahle transakčné operácie
  - bankové systémy
- SOAP/REST webservices
  - tony volaní, ktoré nepotrebujú stav
- dávkové spracovanie
  - map/reduce, grid..
- integrácia systémov
  - mediácia, transformácia správ

Ďalšie črty akktorov

- hierarchia akktorov
  - vhodné pri rozdeľovaní práce na podúlohy
- jednoznačná identifikácia
  - pomocou URI (hostname, port, cesta)
- a iné
  - voliteľná synchronicita posielania správ
  - podpora transakcií (model STM)

DEMO› POČÍTANIE FREKVENCIÍ SLOV

Frekvencia slov v dokumentoch

Napadlo mu: zlom dobro zlom.
V krajine Mu napadlo mnoho snehu.

napadlo:2, mu:2, zlom:2, dobro:1, v:1,
krajine:1, mnoho:1, snehu:1

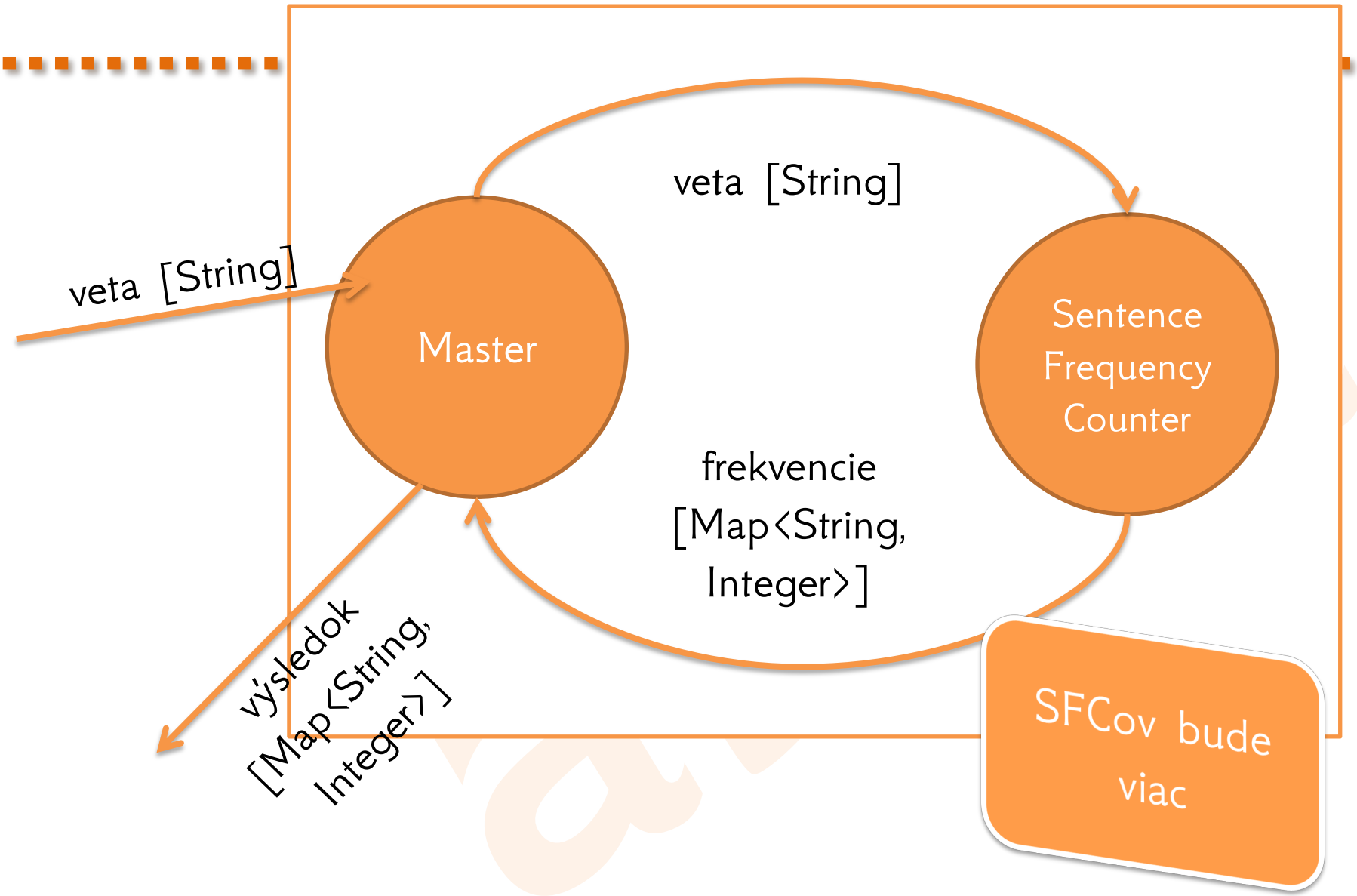
Ako na deľbu práce?

- Jednotlivé vety rozhadzujeme na **uzly**.
- **Uzol** zráta frekvencie slov v jednej vete.
- Samostatný **uzol** berie čiastkové výsledky a kumuluje ich.

uzol = aktor

Návrh v Akke

- potrebujeme navrhnúť správy tečúce medzi aktormi
- správa je identifikovaná svojim dátovým typom
  - trieda v Jave
- **odporúčanie:** z rozličných aktorov musia prichádzať správy rozličných typov


veta [String]
Master
veta [String]
Sentence Frequency Counter
frekvencie [Map<String, Integer>]
výsledok [Map<String, Integer>]
SFCov bude viac

```
private Map<String, Integer> allFreq
    = new HashMap<>();

public Receive createReceive() {
    return receiveBuilder()
        .match(String.class, sentence -> {
            sentenceCounter.tell(sentence, getSelf())
        })
        .match(Map.class, freq -> {
            allFreq = aggregate(freq, allFreq);
        })
        .build();
}
```

zaslanie vety do workerov

združenie frekvencií vety s globálnymi frekvenciami

Routing a škálovanie

- aktor môže predstavovať skupinu (pool) identických pod-aktorov
- vieme distribuovať robotu!
- najjednoduchší spôsob: RoundRobinPool
  - robotu rozdeľujeme „vypočítavankou“

```
public class Master extends UntypedActor {

   private Map<String, Integer> allFrequencies
       = new HashMap<String, Integer>();

   private ActorRef sentenceFrequencyCounters
    = getContext()
        .actorOf(new RoundRobinPool(3)
                        .props(SentenceCountActor.props())
        );

   ....
}
```

globálne frekvencie

vytvorený router s troma workermi, robota sa odovzdáva na striedačku

SentenceCountActor

```
public Receive createReceive() {
    return receiveBuilder()
        .match(String.class, sentence -> {
            Map<String, Integer> freq = calculateFrequencies(sentence);
            getSender().tell(freq, getSelf());
        })
        .build();
}
```

odpoveď odosielateľovi (masterovi)

Zlé demo!

1. kde vidím výstup?
2. kedy systém skončí?

2. Kedy systém skončí?

Mýtus: keď sa vyprázdnia mailboxy

- čo keď príde o minútu ďalšia správa?

2. Kedy systém skončí?

- master nemôže čakať na dobehnutie workerov
- blokoval by príchod ďalších správ
- **blokovanie** v aktorovom modeli je **ZLO**

Akka neháda, kedy naša aplikácia skončí, musíme jej to povedať.

Riešenie č. 1: ak viem, koľko správ pošlem do systému

- viem, koľko správ pošlem do systému
  - koľko viet, toľko správ
- mastera vytvorí s počítadlom
- s každou došlou správou od workera znížim počítadlo
- ak počítadlo klesne na nulu:
  - **vypisujem výsledok**
  - **zatváram systém**


akka

```
public Master(int numberOfSentences) {
        this.numberOfSentences = numberOfSentences;
}
...
public static Props props(int numberOfSentences) {
    return Props.create(Master.class, numberOfSentences);
}
```

konštruktor

parametrické Props

poslané do konštruktora

```
public static void main(String[] args) {
    ActorSystem system = ActorSystem.create();
    ActorRef master
        = system.actorOf(Master.props(3));
…
```

Master#createReceive()

```
.match(Map.class, freq -> {
    this.allFrequencies = aggregate(freq, this.allFrequencies);
    remainingSentences--;
    if (remainingSentences == 0) {
        logger.info(allFrequencies.toString());
        getContext().system().terminate();
    }
})
```

alternatívne: pošlem správu do tretieho aktora, vypisovača

Výsledný projekt

https://github.com/novotnyr/akka-wordfrequencies-2018/tree/message-count-tracking

Ako zavrieť aplikáciu, keď nevieme dopredu počet správ?

Ako ukončiť celý systém?

- po odoslaní všetkých viet pošleme do mastera „EOF“
- celý systém aktorov sa skončí:
  - ak príde správa EOF
  - mapovače skončia svoju robotu
  - vypíše sa globálna mapa frekvencií

Ako ukončiť celý systém?

- master vie ukončiť mapovačov
- potom vypíše globálnu mapu frekvencií
- ukončí seba
- ukončí celý systém

Ako zabiť aktora?

- pošleme mu špeciálnu správu: **PoisonPill**
  - tabletka s jedom
  - zabudovaná v Akke
- použitie:
  - pošleme aktorovi jed
  - jed sa zaradí na koniec jeho mailboxu
  - aktor dospracováva správy z mailboxu, spracuje jed a umrie

Ako zabiť router?

- pozor na aktorov, čo sú routre!
  - naše mapovače sú schované za routrom
- jed okamžite zabije router aj s deťmi
- bez šance na dokončenie spracovávania správ
- korektný spôsob: **broadcastnúť jed**
  - aktor za routrom požuje správy, zje jed, umrie
  - po umretí všetkých aktorov umrie aj router
  - upovedomí sa aktor, ktorý odosielal jed

Ako zabiť router?

- po správe „EOF“ broadcastneme jed
  - vlastná správa ResultRequest
- po prijatí správy ResultRequest:

```
router.tell(new Broadcast(PoisonPill.getInstance()),
            getSelf());
```

Reakcie na zabitie: DeathWatch

- aktor môže opatrovať iného aktora

```
@Override
public void preStart() throws Exception {
        super.preStart();
        getContext().watch(sentenceFrequencyCounters);
}
```

- o smrti opatrovníka sa opatrovateľ dozvie prijatým správy **akka.actor.Terminated**

Reakcie na zabitie: DeathWatch

```
.match(Terminated.class, message -> {
    logger.info(allFrequencies.toString());
    getContext().system().terminate();
})
```

- ak **Master**-ovi dôjde správa o úmrtí **routera**, vie vypísať výsledok a zavrieť systém

```
public static void main(String[] args) {
   ActorSystem system = ActorSystem.create();
   ActorRef master
      = system.actorOf(Master.props());
   master.tell("The quick brown fox tried to jump over the lazy dog and fell on the dog",
      ActorRef.noSender());
   master.tell("Dog is man's best friend", ActorRef.noSender());
   master.tell("Dog and Fox belong to the same family",
      ActorRef.noSender());
   master.tell(new EofMessage(), ActorRef.noSender());
}
```

Korektné uzatvorenie

- máme korektné uzatvorenie
- bez čakaní
- bez blokovania
- systém utešene dobehne a zatvorí sa

Demo!

https://github.com/novotnyr/akka-wordfrequencies-2018

Otázky?