Eastern Academic Journal

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

ШАБЛОН ЗА ДВУСТЪПКОВ МОДЕЛ НА ИЗЧИСЛИТЕЛЕН ПРОЦЕС

Мария Р. Армянова*

Икономически университет - Варна
България

РЕЗЮМЕ — *В съвременния конкурентен и динамичен свят бизнесът изисква качествени софтуерни системи, които отразяват текущите му нужди и се адаптират лесно към изискванията на световния пазар. Шаблоните обобщават най-добрата практика в областта на софтуерните системи. Те предлагат гъвкави решения на често срещани софтуерни проблеми. Целта на шаблона е веднъж намерено, доказалото се като успешно, решение да послужи за основа на нова разработка. Развитието на информационните технологии непрекъснато изправя разработчиците пред нови предизвикателства. Едно от тях е обработката на големи по обем данни. В статията е описан формално шаблон за проектиране на двустъпков изчислителен процес, който представя възможно решение на този проблем. Тъй като той е предназначен за облачни приложения, би трябвало да се възползва от предимствата на облачната среда, а именно възможността за реална паралелна обработка, която е в основата на микросървис архитектурата. Той предлага паралелно извършване на времеемките и ресурсоемките манипулации с данните, като първа стъпка от алгоритъма си и обобщаване на резултатите във втора стъпка.*

Ключови думи: архитектура на микроуслугите, облачна среда, паралелени изчисления, шаблони за проектиране

DESIGN PATTERN FOR A TWO-STEP COMPUTATIONAL PROCESS

Mariya R. Armyanova*

University of Economics, Varna, Bulgaria

ABSTRACT— *In today's competitive and dynamic world, business requires quality software systems that reflect current needs and adapt easily to world market requirements. The design patterns generalize the best practice in software systems. They offer flexible solutions to common software issues. The design pattern's purpose is to use the proven successful solution, as a new development's base. The growth of information technology has constantly leaded developers to new challenges. One of them is the processing of large data. The paper describes a formal design pattern for designing a two-step computational process that presents the problem's possible solution. Since it is designed for cloud applications, it should take advantage of the cloud environment - the possibility of real parallel processing that is the microsurvey architecture's basis. It offers the simultaneous execution of time-consuming and resource-intensive data manipulations as a its algorithm's first step and a results' summary in a second step.*

Keywords: big data, cloud computing, design patterns, microservices, parallel processing

*Corresponding author at: University of Economics, Varna, Bulgaria, *e-mail:* **armianova@abv.bg**

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

1. ВЪВЕДЕНИЕ

В световен мащаб тенденцията е към непрекъснатото увеличаване на количеството съхранявани данни. За различни цели (статистически обработки, проучвания, анализи, запазване на изображения и др.) постоянно се генерират и събират данни, които поради законови, нормативни или други изисквания се съхраняват и извличат през дълги периоди от време. Проблемът се усложнява и от голямото разнообразие и разнородност на данните. Цената на паметта спада и системите съхраняват всички данни, нужни или не. Според фирмите, изследващи сектора на информационните технологии, обемът на данните в световен мащаб нараства с около 50-55% (Saga Tehnology) годишно.

За целите на анализа и проучването на клиенти се събира много и разнородна информация, за да могат да се изготвят различни прогнози, например от телекомуникационните компании. Маркетинговите проучвания изискват интегрирането на данните от проучванията за настроенията на клиентите към даден продукт или услуга с данните от потребителския им профил. Откриването и предотвратяването на множество видове измами и рискове в различни области, като отпускане на заеми, застрахователни измами и др, също изисква анализа на огромни и разнородни данни. Все повече навлизат и различни системи за обработки на изображения, например за разпознаване на лица, които работят с големи количества данни. Големите данни (Big Data) са предизвикателство както за инфраструктурата, така и за анализа и обработката. Търсят се нови, надеждни, евтини и ефективни решения за съхранение на данни, и наемането на облачно пространство е едно от тях.

За да са полезни натрупаните данни е важно е да нарастне скоростта на обработка им. Като решение на този проблем се използват технологии и архитектури, които позволяват паралелна им обработка. В световен мащаб тенденцията е към преминаване към тази технология, тъй като въпреки техническия прогрес не може достатъчно бързо да се увеличава скоростта на работа на отделния процесор и обема на регистрите и кеша му, за да се справи с нарастналото количество данни. Затова и този начин на работа е много по-ефикасен. Той увеличава скоростта на решаване на продължителните задачи и позволява равномерно натоварване на всички ресурси. Разбира се той се изправя пред някои предизвикателства, като трудности при реализацията на такъв тип архитектура, заради нуждата от синхронизацията на процесите, обединяването на резултата от паралелните обработки и нуждата от приспособяване на архитектурата към различните цели на обработките. Но въпреки тези трудности, обработката чрез паралелни процеси се налага като популярна технология. Тя намира особено голямо приложение в облачните системи, които дават възможност за използване на голям брой отделни изчислителни ресурси. И именно това е най-ефективния начин за ускоряване на изчисленията в такава среда. Като начин за реализация на тази технология се използват микросървис архитектурите, които позволяват паралелна обработка, т.е. облакът чрез микросървиса върви към паралелизъм.

Терминът микросървис (Microservices микроуслуги) описва новия стил на разработка на софтуерната архитектура. Все повече проекти използват този стил през последните няколко години, а резултатите досега са положителни. Това увеличава популярността му до такава степен, че много от разработчиците го използват по подразбиране за изграждане на корпоративни приложения. За съжаление, понеже все още е сравнително нов начин на разработка, няма много публикувана информация, която строго да дефинира микросървиса и начина му на прилагане.

Терминът "Микросървис архитектура" (Fowler, 2014) се налага през последните няколко години и дефинира конкретен начин за проектиране на софтуерни приложения като група

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

от независимо реализиращи се услуги. Макар че няма точно определение на архитектурния стил микросървис, има някои общи характеристики, които позволяват откриването му. Неговите особености са свързани с начина на организацията на разработката, бизнес приложимостта му, автоматизираното внедряване, интелигентността в крайните точки и децентрализирания контрол на езиците и данните. За първи път терминът микросървис или микроуслуги се появява на семинар край Венеция през май 2011 г., като наименование на архитектурен стил, въведен е от Левис през март 2013г. (Lewis, 2012) в доклад на конференцията в Краков.

Според Фаулер (Fowler, 2014) архитектурният стил на микросървиса е подход към разработването на едно единствено софтуерно приложение като набор от малки услуги, всяка от които работи със свои собствен процес. Тези процеси се обвързват помежду си със слабо обвързващи механизми за комуникация, например използващи съобщения, за да формират своите приложно програмни интерфейси (API) през протокола HTTP. Тези услуги са изградени на базата на бизнес изискванията и могат да бъдат развивани. Все пак се изисква минимално централизирано управление за тези услуги. Те могат да бъдат написани на различни програмни езици и да използват различни технологии за съхранение на данни.

Архитектурата на микроуслугите е представена, като архитектурен шаблон, който е независим от използваната програмна парадигма. Предназначен е за разработването на корпоративни приложения от страна на сървъра. Софтуерното приложение трябва да осигурява достъп до услугите на голямо разнообразие от клиенти, включително десктоп браузъри, мобилни браузъри и мобилни приложения. То може да реализира връзка с други приложения чрез API. Освен това може да се интегрира и с други приложения чрез уеб услуги или трансфер на съобщения. Приложението обработва заявки (HTTP заявки и съобщения), като изпълнява бизнес логика; достъпва бази от данни; обменя съобщения с други системи; и връща отговор в формат HTML, JSON, XML или др. Има логически компоненти, съответстващи на различни функционални области на приложението.

Предпоставките за нарасналия интерес към разработване на приложения с микросървис шаблона са няколко. Част от тях са организационни. Този тип архитектура поддържа гъвкавите подходи и итеративния процес на разработка, като позволява към приложението непрекъснато да се добавят нови услуги. Всяка микроуслуга е достатъчно малка, което я прави лесна за разбиране и разработване. Съкращава се времето за началните етапи на разработката, което увеличава продуктивността на разработчиците и ускорява разработката. Също позволява на новите членове на екипа ефективно да се включат в работата, тъй като могат бързо да навлязат в предметната област и да се справят с разработката на малки, относително самостоятелни услуги.

Самото приложението става по-лесно за промяна и адаптиране. Всяка услуга се създава независимо от останалите, което позволява лесно и често да се включват в системата новите им версии. Освен това всяка от тях работи независимо, което подобрява се изолацията на проблемите. Ако се получи проблем в някоя услуга, другите не се засягат и могат да продължат да работят. Докато при монолитните архитектури един неправилен компонент може да спре работата на цялата система.

Друго предимство е, че не се обвързва софтуерна система с микросървис архитектура с конкретна технологична парадигма. Всяка услуга може да се разработи с различна технология, което позволява лесно въвеждане на иновациите.

Подходът на микросървиса позволява приложението да се разработи съгласно изискванията за мащабируемост и наличност на облачната среда, като няколко екземпляра

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

на услуга да работят на множество машини.

Микросървис архитектурата е шаблон от най-висок мащаб – за архитектурата на цялата система. Тя определя кои са и как се организират и взаимодействат компонентите на системата. Но и на средното ниво, това на механизмите, също могат да се приложат принципите на паралелизма. Идеите на микросървис архитектурата може да се приложат и в по-малък мащаб за реализацията на компоненти или изчислителни процеси чрез шаблоните за проектиране.

Целта на изследването е да се открие шаблон, представящ решение на проблемите, свързани с анализа и обработката на Големите данни, който да се базира на облачните технологии и микросървис архитектърата.

Важен критерий при търсене на решение е до този момент да няма предложен шаблон, който да решава сходен проблем. Тъй като някои от шаблоните могат да се използват и в облачна среда е направена проверка дали сред тях няма подобен шаблон.

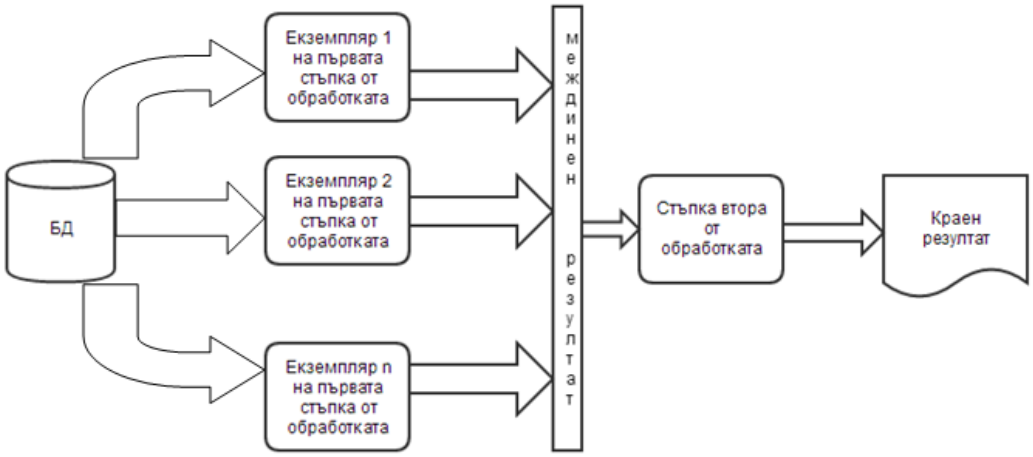
Има разработени редица шаблони за реализация на паралелни и псевдопаралелни процеси, например за многонишкова обработка, които реализират архитектура, известна като entity-life modeling (ELM) (Sandén, 1989, 1996, 1997). Но предложените от Сандън шаблони за идентифициране на отделни нишки са насочени към реализацията на взаимодействието с потребителя в реално време, а не към организацията на изчислителните процеси.

Други шаблони, чиито идеи се използват и при паралелизма са тези за системите, работещи в реално време. Такива системи се състоят от софтуерни и хардуерни компоненти, които трябва да синхронизират и координират различни процеси (Douglass, 2002). Те често трябва да съчетават множество задачи, които са с твърдо определен краен срок (Glossary, FAQ and Troubleshooting). Такива задачи могат да използват различни механизми, за да осигурят отговор в реално време – например буфери, когато закъснението може да предизвика нежелани последици, като при преглед на музикални или видео файлове. Друг механизъм, които съответства на идеята за паралелизъм е подреждането на процеси в нишки, според приоритета им. Процесите, при които е недопустимо закъснение се организират в нишки с голям приоритет, а тези, при които е допустимо известно закъснение, – в нишки с малък приоритет. Този тип шаблони са предназначени за архитектурата на подобни системи, а не за реализацията на конкретен процес. Освен това многонишковата обработка не е задължително да се извършва от няколко процесора, а може да се реализира от един, което е т.нар. псевдопаралелизъм. Но при облачните приложения има възможност за организация на реално паралелно изпълнение на процесите. От тази гледна точка се налага да се потърси нов шаблон, който да организира паралелно продължителен и ресурсоемък изчислителен процес. Той трябва да е на ниво процес, т.е. това е т.н. шаблон за проектиране.

Проблемът, които шаблонът се опитва да реши, е оптимизацията на сложен или бавен изчислителен процес. Тъй като той е предназначен за облачни приложения, би трябвало да се възползва от предимствата на облачната среда, а именно възможността за реална паралелна обработка, която е в основата на микросървис архитектурата. Следвайки нейния принцип, възниква идеята за паралена реализация на изчислителен процес. Както споменахме когато един процес обработва големи данни възникват редица проблеми – забавяне на обработката, трудности при зареждане на подлежащите за обработка данни и др. Една възможност за решаването им е използването на шаблон за облачно приложение за реализация на двустъпков изчислителен процес. Първата стъпка предполага, извършване на паралелни изчисления върху отделни парчета от данните. Задачата, реализирана на

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

първа стъпка, предполага извличане на отделните парчета от данните, подлежащи на обработка. Те могат да се селектират по различни признаци – сървър, БД, период или др. Целта е да се извлече парче от данните (data slice), според заложения в първа стъпка алгоритъм. То трябва да е достатъчно малко, за да може да се зареди в паметта и да се обработи. Целият изчислителе процес се ускорява, ако получените парчета от данни могат да се извличат и обработват едновременно, затова се създават екземпляри на отговорния за първата стъпка модул (клас, услуга или др.), които работят паралелно. Получените отделни резултати от работата на тези екземпляри имат нужда от синхронизиране и обединяване. За тази цел се налага да се въведе един междинен модул, които да натрупва резултатите и да изчаква приключването на всички стартирани задачи от първа стъпка. Накрая се прилага втората стъпка от двустъпковото изчисление. Това е допълнителна обработка на данните, която може да е свързана с подготвянето на данните за представяне във вид подходящ за крайния потребител или за съхранение, например подготвяне на отчет, изпращане на писмо и др.. Схема на последователността на обработките върху данните е представена на фиг. 1.



Фиг. 1. Схема на шаблон за двустъпков изчислителен процес.

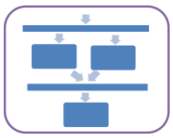
2. МАТЕРИАЛИ И МЕТОДИ

За да се създаде представения шаблон са направени проучвания върху съществуващите в момента шаблони. Използвани са методи на системен, сравнителен и исторически анализ. За разработката на шаблона са ползвани методи за моделиране. А при апробацията на шаблона са използвани техники за апробация и прототипиране. Използвани са кодове на езика за програмиране JAVA.

3. РЕЗУЛТАТИ

Резултатът от направените проучвания е откриване на шаблон. Той е представен формално, диаграмите са разработени в съответсвие с UML подхода.

Име: Двустъпков модел на изчислителен процес

Икона: 

Контекст:

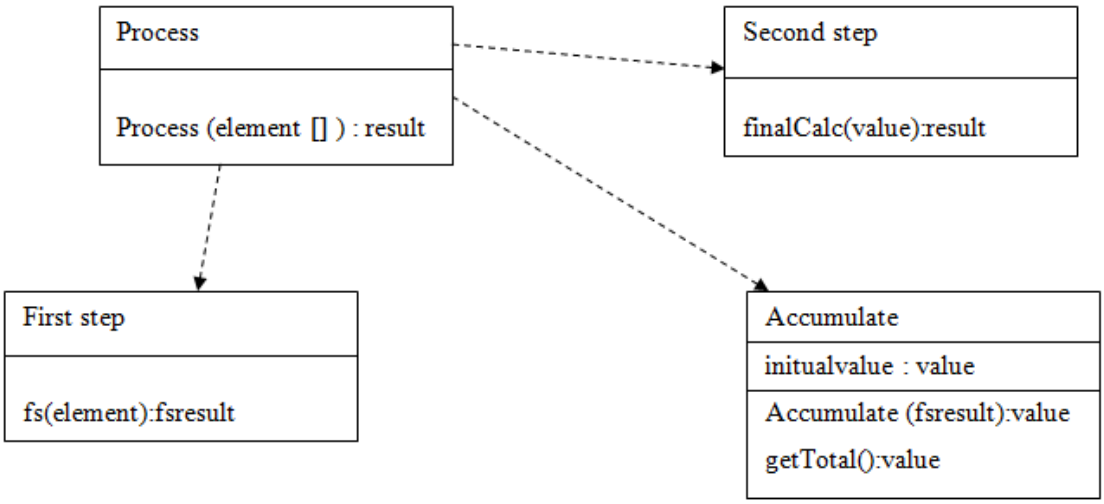
ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

Предназначен е за извършване на продължителни изчислителни задачи, които могат да се разделят на две стъпки. Такива изчислителни процеси изискват много памет и процесорно време. За облачните архитектури подобни процеси са неподходящи. Този шаблон е ефективен, когато обработката на първата стъпка може да се раздели на независими части, всяка от които остава достатъчно времеемка.

Описание:

Когато се обработва и обобщава голямо количество еднотипни данни, процесът на обработка е дълъг. Облачната среда изисква разпределение на ресурсите между конкуриращи се процеси, а не обсебване от един дълъг процес. Така се намалява натоварването на един ресурс в системата, като натоварването се разпределя между ресурсите на облачната система. Тя е ефективна при паралелизъм, а не при строго последователни обработки. Така паралелните обработки се поемат от много независими машини. Затова и микросървис архитектурата, която се основава на декомпозицията, е подходяща за облачна среда. В поддръжка на тези изисквания към приложенията и извършваните от тях обработки се явява и шаблонът за двустъпковия модел на изчислителния процес, който декомпозира крупната обработка в множество паралелно извършващи се процеси на едно ниво. Той извършва обработката на две стъпки. Първата е извличането и обработката на отделни парчета от данните чрез паралелно протичащи процеси. Втората стъпка извършва обработката на събраните обобщени данни и привеждането им в подходящ за използване вид, например за целите на бизнес анализа или разполагането им в отчетна форма. Обработката на първата стъпка се извършва върху цялата колекция от данни. Тя може да се реализира чрез множество паралелни процеси, които извършват обработката върху част от данните. Шаблонът не предполага използването на определен модел на извличане на данните, за да се осигури по-голям обхват на прилагането му. Информацията, необходима за извършването на всяка от паралелните задачи се подава през еднотипни елементи, събрани в колекция, като отново се абстрахираме от конкретните детайли по организация на тази информация. Всеки такъв елемент се описва от клас Element. Стъпките на обработката са представени в отделни класове First step и Second step. Те са характеристики на класа Process. Класът Process управлява извършването на цялата обработка. Той инициализира обекта на класа Accumulate и стартира паралелните процеси. След приключване на първата стъпка от обработката данните резултатите се обобщават чрез метод на клас Accumulate. Той обработва резултатите от паралелно протичащите процеси за обработката, реализирана на първата стъпка. Този клас отговаря и за синхронизацията при работа с резултатите от паралелно протичащите процеси. Втората стъпка получава резултата от клас Accumulate, след приключване на всички паралелни процеси и извършва последваща обработката. Първоначален модел на диаграмата на класовете е показан на фиг. 2.

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017



Фиг 2. Диаграма на класовете за Двустъпков модел на изчислителен процес.

Примерен код:

Използва се код на Java за илюстрация на двустъпковия шаблон. Целият код на примера е даден в приложение 1.3а тази имплементация на шаблона се създава клас Element, който има за цел да извлече парче от данните. Конкретният метод и размер на данните е строго специфичен за всяка реализация и не е част от шаблона. В случая на конструктора на класа се подава дължина на парчето от данни, което се извлича чрез length и в startIndex начална позиция, от която да започне извличането.

```
public class Element {
   int startIndex, length;
   int[] vaules;
   float multiplier;
   public Element(int[] values, int startIndex, int length){
     // специфичен метод за извличане на отделен елемент (парче от данни)
   }
}
```

Методът Service подава парчето от данни подлежащо на обработка чрез Element и къде се натрупва резултата чрез Accumulator. В разгледания примерен код управляващият клас е именуван Processor, той създава паралелно протичащите процеси. Има и други възможности за разлизация на паралелните изчисления. Функционалността по създаването на паралелните процеси може да се прехвърли на класа First step, който да реши как да планира задачата, дали да я изпълни синхронно или да определи колко процеса са му нужни. В предложената имплементация на шаблона в приложение 4, управлението е предоставено на Processor, който създава нужния брой нишки. Той може да изчака завършването на всички нишки или не според нуждите на конкретното приложение, като ако не изчака няма как да разбере какъв е крайният резултат от работата им.

```
class Service implements Runnable {
   Element e;
   Accumulator<Float, Float> acc;
   Service(Element e, Accumulator<Float, Float> acc) {
```

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

```
        this.e = e;
        this.acc = acc;
    }
    @Override
    public void run() {
        new FirstStepMultiplySum(acc).process(e);
    }
}
public class Processor {
    public void process(Element[] elements, boolean waitFinish) {
        SecondStep<Float, Void> ss = new SecStepPrint();
        Accumulator<Float, Float> acc = new AccumulatorSum(ss, elements.length);
//…..
    }
}
```

Абстрактният клас за първа стъпка е FirstStep. В примерния код той отговаря за извършването на изчисленията от първа стъпка и връщането на резултата на Акумулатора.

```
public abstract class FirstStep<ElementType, OutputType> {
    public FirstStep(Accumulator<OutputType, ?> acc){
        accumulator = acc;
    }
    public abstract void process(ElementType input);
    protected Accumulator<OutputType, ?> accumulator;
}
public class FirstStepMultiplySum extends FirstStep<Element, Float> {
    public FirstStepMultiplySum(Accumulator<Float, Float> acc) {
        super(acc);
    }
    @Override
    public void process(Element input) {
        //Обработва данните според заложения на първа стъпка метод
        accumulator.accumulate(fsresult);
    }
}
```

Класът, синхронизиращ независимите процеси е Accumulator. За целите на илюстрацията е избран метод на синхронизация с флаг, които е броят на процесите. За да приключи обработката от първата стъпка трябва да се изпратят същия брой резултати. Кодът е:

```
public abstract class Accumulator<InputType, AccType> {
    protected AccType value;
```

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

```
    protected SecondStep<AccType, ?> secStep;
    public Accumulator(AccType initialVal, SecondStep<AccType, ?> ss) {
      value = initialVal;
      secStep = ss;
    }
    public abstract void accumulate(InputType input);
}
```

Абстрактният клас се имплеметира с клас AccumulatorSum, който при създаването си получава броя на стартираните паралелни процеси чрез cntProcess.

```
public class AccumulatorSum extends Accumulator<Float, Float>{
    int totalProcess;
    public AccumulatorSum(SecondStep<Float, Void> sec, int cntProcess){
    }
    @Override
    public void accumulate(Float input) {
      synchronized(this){
        --totalProcess;
        // метод за натрупване на подадените от обектите на паралелните процеси стойности в value
        if (totalProcess == 0){
          secStep.process(value);
        }
      }
    }
}
```

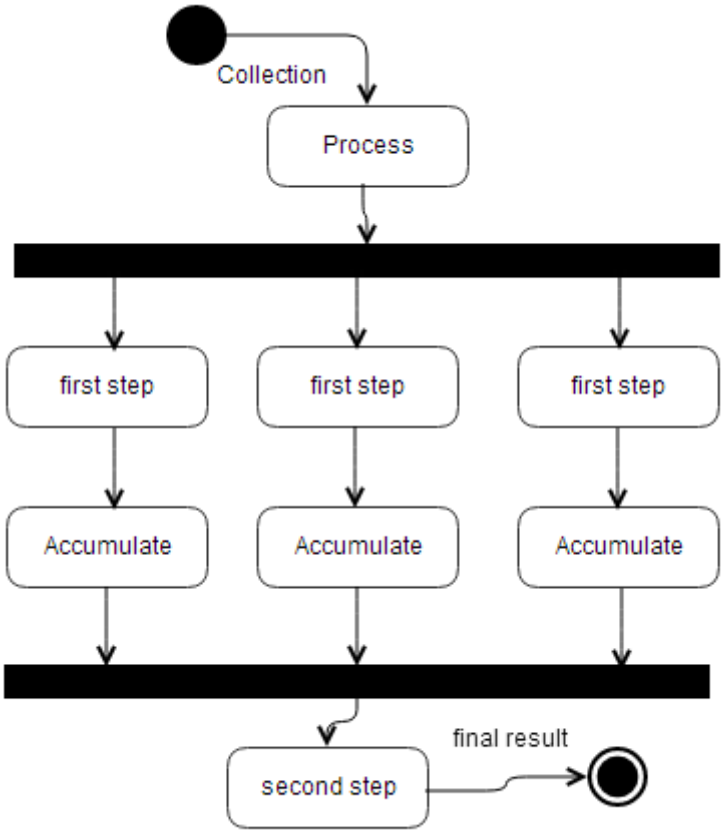
Представен е кодът на интерфейса на класа SecondStep за реализацията на втора стъпка. Той обработва резултата на класа Accumulator и връща желания краен резултат. От конкретната реализация зависи какъв тип е резултатът и как се връща. Може да се използва технологията на callback, т.е. да се подаде на SecondStep адрес на интерфейс или указател към функция, на когото да изпрати резултата. Друга възможност е резултатите да се запишат във файл или БД.

```
public interface SecondStep<InputType, OutputType> {
    OutputType process(InputType input);
}
```

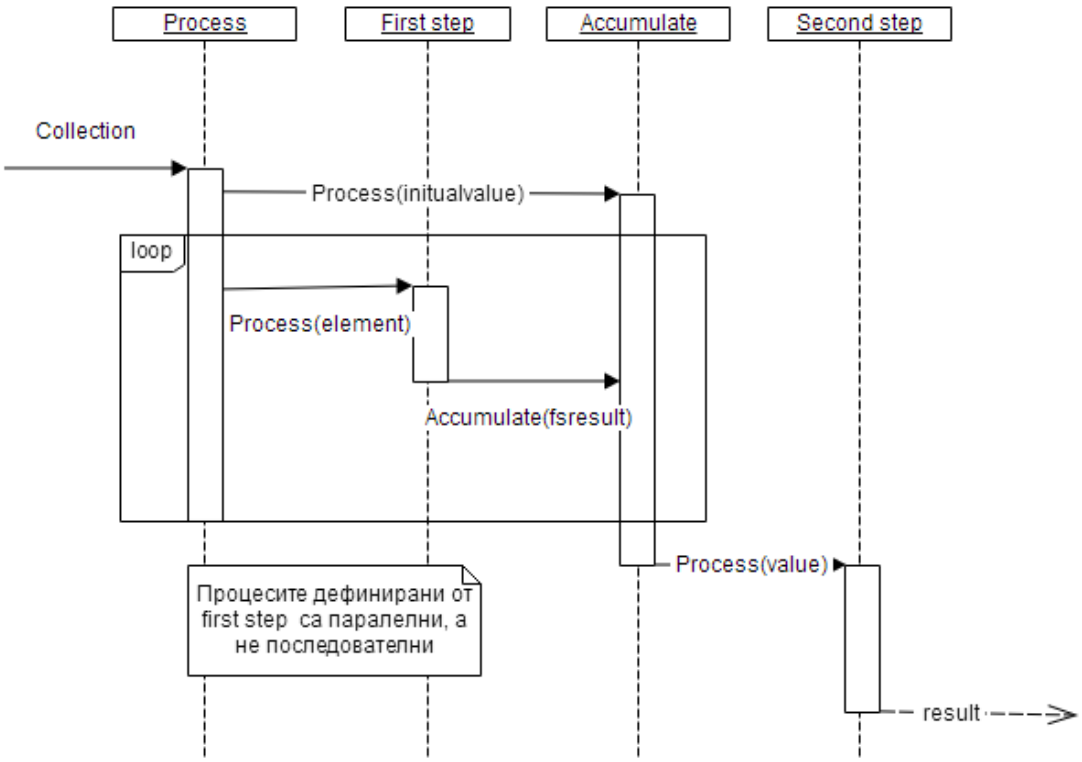
Начин на работа:

Обикновено по време на работа под контрола на Process се създават няколко екземпляра на First step, за първоначалното извличане и обработка на данните. Process извършва и инициализирането на Accumulate, за да може той да получи резултатите от изпълнението на всички екземпляри на First step. След като събере всички резултати единственият екземпляр на Accumulate изпраща събраната информация на Second step. Последователността на обработките и паралелното протичане на част от процесите се вижда на диаграмата на дейностите на фиг. 3.

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017



Фиг. 3. Диаграма на дейностите на шаблона за двустъпковия модел.



Фиг. 4. Интерактивна диаграма на шаблона за двустъпковия модел.

Сценарият на взаимодействие между обектите на шаблона е показан на фиг. 4, както и потоците управляваща информация, която се разменя между процесите.

Предизвикателства:

- Как да се декомпозира процесът на стъпки;
- Какъв метод да се избере за синхронизиране на независимо изпълняваните процеси;
- По какъв начин да се достъпват данните от независимите процеси, дали за всеки процес да се създава копие на обработваното парче от тях, дали да се организира специална услуга, която да ги извлича и препраща и т.н.;
- Важно е да се реши на кое ниво ще се управляват паралелните процеси. Те могат да се създават от Process или тази функционалността да се прехвърли на класа First step, който да реши как да планира задачата, дали да я изпълни синхронно или да пусне нишки за десктоп реализацията. Ако се използва за облачно приложение First step може да реши да се обърне към услуга или ако самият той да е услуга.

Приложения: За анализ на големи данни. Особено подходящ е за системи, които са разпределени и данните се извличат от няколко източника.

Данните могат да се извличат в паралелните процеси по различни критерии, например всеки процес да натрупва данните от определен сървър, БД, архивен файл, период и др.

Може да се разработи и вариация на шаблона специално за реализации на различни анализи. В такава вариация всеки процес от първата стъпка извършва различен тип анализ върху едни и същи данни, след което на втората стъпка от обработката получените резултати се обобщават в единна прогноза. Типичен пример е за анализ на стоките и прогнозиране на продажбите и печалбите при пускане на определени промоции. Тогава данните за стоките се подлагат на различни анализи, например откриване на залежалите или стоките с изтичащ срок на годност, откриване на типа търсени стоки, откриване на свързаните стоки (frequently bought together), като например ел. книга и калъф за нея, книги от една серия и др.. Като резултат от обобщаването във втората стъпка на всички анализи може да се окаже примерно, че пускането на остарелия модел ел. книга на по-ниска цена, съчетано с продажбата на по-скъпи калъфи може да реализира същата печалба.

Друго приложение на шаблона е за системи за обработка на изображения. Те работят с огромни по обем данни. Например едно приложение е за цветова корекция на видеоклип. Особено, ако клипът е с професионално качество той е с голям обем и обработката на всеки кадър е времеемка. Работата с всеки кадър е независима и подходяща за паралелна реализация. Шаблонът се прилага, като за обработката на всеки кадър се създава процес от първа стъпка, който да го обработи. Класът Акумулатор ги събира в клип, процесът от втора стъпка генерира финалния клип – компресира данните, изработва индекси и т.н.

Шаблонът може да се използва за разпознаване на образи. Тогава чрез независимо изпълняваните процеси на първата стъпка се претърсват различни части от изображението за нужните елементи. Получените резултати се събират в Акумулатора, а втората стъпка ги обобщава и обработва намерените елементи – може да ги загради, изсветли, изтрие и т.н.

Една от вариациите на шаблона го доближава до микросървис архитектурата, ако на отделните процеси съответстват услуги. На първа стъпка независимите паралелно протичащи услуги извличат данни, а на втора ги обработват. Възможно е дори процесите от първа стъпка да реализират data mining и да извличат знания от различни източници.

Например подобен шаблон може да е част от система за маркетингови проучвания.

Шаблонът може да се използва и за многостъпков изчислителен процес, като в зависимост от нуждите се повтори някоя от стъпките. Например, ако се налага няколкократно обобщение на първоначалните данни, първата стъпка може да се повтори.

Подходящи за реализацията на шаблона са облачните приложения, локално разпределени системи и др. многопроцесорни архитектури, паралелни системи основани на Message Passing Interface (MPI).

Приложимост:

Шаблонът е предназначен за приложения, работещи в облачна среда. Но може да се използва и за приложения, работещи на десктоп системи. Тогава той трябва да се представи като шаблон за многонишкови обработки. При такива условия обектите на класовете Process, Accumulate и Second step са в една нишка, защото работят последователно и са синхронизирани, а всички обекти на класа First step са асинхронни. За микросървис архитектурата на облачната платформа не е така. Между изпълнението на процесите, дефинирани от класовете Process, Accumulate и Second step има момент на изчакване, а принципът на организация на работа на облачните платформи не препоръчва да се държат пасивни процеси. Така, че те ще са в отделни нишки.

Въпроси:

Използването на този шаблон за реализацията на обработка върху малки по обем данни не е ефективно. Например използването му за изчисляване на стойността на фактура. В случая на паралелните процеси може да се присвои функционалност за обработката на отделен ред от фактурата, като изчисляване на стойността на продадената стока. При някои данъчни системи процесите могат да изчисляват и дължимия данък, особено ако това зависи и от типа на стоките. Получените стойности се сумират в обекта на класа Accumulator, след което се подлагат на последваща обработка, която може да е изчисляване на отстъпката според типа клиент или общ дължим данък и се попълват в необходимата за визуализиране форма. Също както и при архитектурния шаблон микросървис и тук паралелизмът не е ефективен. Вместо да имаме едно обръщение към БД, с което да се извлекат всички данни за фактурата при подобен начин на организация се правят множество обръщения. Обръщението към БД, както и самото стартиране на паралелна задача са бавни операции и дори изчислителният процес да се ускори, това не би могло да компенсира това забавяне. Освен това ако процесът е монолитен той се реализира с един цикъл и условен оператор за изчисление на данъка, а ако се реализира чрез паралелни процеси е нужно да се създават екземпляри на процеса на обработка и да се синхронизира връщането на резултатите им, което покачва броя на обработките. От гледна точка на използването на памет този вариант също е не ефективен. За всеки екземпляр на изчислителния процес се запазва памет, която общо е повече от нужната за работата на монолитен процес. Извършването на изчисленията с подобна архитектура прави обработката трудна и бавна на фона на останалите варианти за решение.

Свързани шаблони:

Тъй като шаблонът е предназначен за облачни приложения, той най-добре си взаимодейства с други облачни шаблони. Например за MS Azure[1] има разработен шаблон Pipes and Filters Patterns (Homer, 2014). Той позволява да се извличат данни от различни източници за целите на бизнес приложението. Той би могъл да се съчетае с шаблона за двустъпкова обработка, за да позволи на процесите от първа стъпка да извличат данни от

[1] Облачна платформа на MicroSoft

различни източници.

За достъп до данните може да се използва и шаблонът Data Access Component на Филинг (Fehling, 2014). Той позволява да се скрие от процесите използващи данните сложния за реализация начин за достъп до тях.

За синхронизиране на независимите паралелно протичащи процеси от първа стъпка могат да се използват различни методи. Например може да се използва шаблон Queue-Based Load Leveling Pattern на MS Azure. Той описва използването на опашка, която действа като буфер между задачите и виканата от тях услуга. В някои случаи този шаблон замества дейността на класа Accumulate, но тогава на втората стъпка от процеса трябва да се извърши и първоначалното обединяване на данните.

Ако процесите от първа стъпка извличат различна информация от едни и същи данни шаблонът би могъл да се съчетае с предложения от Филинг шаблон Compliant Data Replication. Този шаблон позволява да се създаде копие от данните за всеки екземпляр на услугата, т.е. за всеки процес от първа стъпка. Тъй като всека услуга в облака се реализира независимо, със собствени ресурси и не може да споделя обща памет с останалите услуги този вариант може и да не е удачен. Да се прави копие на парче от данните е по-оптимално. Тъй като обикновено общото количество памет, което може да използва облачното приложение е ограничено (например колкото е платено), не е желателно да се копират всички данни.

Шаблонът обаче може да се ползва и за десктоп системи, тогава обаче трябва да се съчетае с шаблони, реализиращи многонишковата работа. Например с шаблона на Сандън Resource User, които подрежда процесите в нишки, за достъп до процесора.

4. ЗАКЛЮЧЕНИЕ

Представеният шаблон за проектиране на двустъпков изчислителен процес е разработен самостоятелно от автора. Той е резултат от проучване на проблемите в областта на облачните системи, паралелните обработки и шаблоните. Проучени са съвременните технологии за разработка, реализация и поддържане на софтуерните системи. Шаблонът подпомага реализирането на продължителните обработки за приложения, работещи в облачна среда или базиращи се на многонишковото програмиране.

Използването на шаблоните намалява разходите, времето и трудността при разработката на системите. Създадените с тях приложения са с гарантирано ниво на качество, по-дълъг жизнен цикъл, гъвкави, преносими, могат динамично да се мащабират и персонализират според нуждите на потребителите.

4. СЪТРУДНИЧЕСТВО

Диаграмите са разработени с безплатна среда Gliffy. Шаблонът е апробиран на облачната платформа на Google, в рамките на безплатно предоставеното облачно пространство и софтуер.

ISSN: 2367–7384
Issue 3, pp. 25-38, September, 2017

5. ДОПЪЛНИТЕЛНИ МАТЕРИАЛИ

Шаблонът е апробиран и за многонишково приложение и кодът на отделните класове е изпратен в приложение1.docs.

6. ЛИТЕРАТУРА

Статия
Sandén, B. I.(1997). Modeling concurrent software. IEEE Software.

E-journal статия
Fowler, M., Microservices Resource Guide. https://martinfowler.com/microservices/. Accessed on [2017-08-17].
Lewis, J., Micro services - Java, the Unix Way, paper in conference 33rd Degree conference for Java masters, March 2012, Krakow. http://2012.33degree.org/talk/show/67. Accessed on [2017-08-17].
Saga Tehnology, http://business.sagabg.net/whoswho/koj-koj-e-v-storidzh-sistemite/. Accessed on [2017-08-17].
Glossary, FAQ and Troubleshooting: www.opcdatahub.com/Docs/gl-defs.html/. Accessed on [2017-08-17].

Книга
Douglass, (2002). Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley, Boston, MA, USA.
Fehling, C., Leymann, F. Retter, R. Schupeck, W., Arbitter, P. (2014). Cloud Computing Patterns. Springer.
Homer, A., Sharp, J., Brader, L., Narumoto, M., Swanson, Tr. (2014). Cloud Design Patterns, Microsoft. 978-1-62114-036-8.
Sandén, B. I. (1996). A course in real-time software design based on Ada 95. Available through the ASSET repository as ASSET_A_825.
Sandén, B. I. (1989). Entity-life modeling and structured analysis in real-time software design - a comparison. Communications of the ACM.

7. СЪКРАЩЕНИЯ

1. БД - бази от данни