



divanikus 5 апреля 2012 в 23:24

Конечные автоматы. Пишем ДКА

Perl, Программирование

Если вы когда-нибудь пытались написать своего бота, программу-переговорщик (negotiator), интерпретатор протокола связи и тому подобн вещи, то наверняка сталкивались с конечными автоматами. Данная тема в принципе не представляет большой сложности, но если вдруг у было курса «теории автоматов», милости прошу под кат.

Сегодня мы попытаемся создать простой детерминированный конечный автомат. Мне вдруг захотелось написать его на Perl'e, но так как мы будем использовать никаких специфических трюков, перенести общую концепцию на любой другой императивный язык не составит больш труда.

Введение

Не будем сильно углубляться в теорию, для этого есть специальная литература, гугл и википедия :). Рассмотрим самую базу. Давайте разбер что такое конечный автомат.

В электронике, да и в программировании, в простейшем случае мы имеем дело с так называемыми «переключательными функциями». Это относительно примитивная абстракция не имеющая собственной памяти: мы на вход аргумент, она на выход некое значение. Выходное зн всегда зависит только от входного.

А если нам необходимо, чтобы последующее значение функции зависело от предыдущего? От нескольких предыдущих? Тут мы уже приходим к некой абстракции с собственной памятью. Это и будет автомат. Выходное значение на автомате зависит от значения на входе и текущего состояния автомата.

Конечный автомат соответственно потому так называется, что число его внутренних состояний конечно. А, наверное, самым простейшим и конечных автоматов является детерминированный: для каждого входного сигнала существует лишь одно состояние, в которое автомат мож перейти из текущего.

Таким образом, наш автомат определяется следующим:

- начальным состоянием;
- входным алфавитом (набором возможных входных сигналов);
- множеством состояний;
- и таблицей переходов.

Собственно вся суть автомата определяется последним пунктом. Таблица переходов (также изображается как диаграмма переходов) состоит из 3-х столбцов: входной сигнал (символ), текущее состояние, следующее состояние. Все станет ясно на примере.

Базовый класс

Итак, реализуем наш базовый класс. Мы уже определились, что нам нужно начальное состояние, текущее состояние и таблица переходов. Алфавит входных символов определяется задачей, поэтому также нам потребуется нормализация (упрощение) ввода. Роль выходных сигналов будут выполнять исполняемые методы класса потомка. Для упрощения себе жизни, искусственно добавим в алфавит символ "*" — любой символ.

```
package FSM;

use strict;
use Carp;
use vars qw($AUTOLOAD);

# Create new object
sub new {
    my $self = {};
    my ($proto, $initial) = @_ ;
    my $class = ref($proto) || $proto;

    # Init ourselves
```

```

$self->{INITIAL} = $initial;
$self->{CURRENT} = $initial;
$self->{STATES} = {};

bless ($self, $class);
return $self;
}

sub setInitialState {
    my ($self, $initial) = @_;
    $self->{INITIAL} = $initial;
    return $self;
}

sub setCurrentState {
    my ($self, $current) = @_;
    $self->{CURRENT} = $current;
    return $self;
}

sub getCurrentState {
    my ($self, $current) = @_;
    return $self->{CURRENT};
}

sub reset {
    my $self = shift;
    $self->{CURRENT} = $self->{INITIAL};
    return $self;
}

sub addState {
    my $self = shift;
    my %args = @_;
    $self->{STATES}->{$args{STATE}}->{$args{SYMBOL}} = {NEXT => $args{NEXT}, ACTION => $args{ACTION}};
    return $self;
}

sub removeState {
    my $self = shift;
    my %args = @_;
    if (exists $args{SYMBOL}) {
        delete $self->{STATES}->{$args{STATE}}->{$args{SYMBOL}};
    } else {
        delete $self->{STATES}->{$args{STATE}};
    }
    return $self;
}

# Be sure to override in child
sub normalize {
    my ($self, $symbol) = @_;
    my $ret = {};
    $ret->{SYMBOL} = $symbol;
    return $ret;
}

sub process {
    my ($self, $rawSymbol) = @_;
    my $state = $self->{STATES}->{$self->{CURRENT}};
    $rawSymbol = $self->normalize($rawSymbol);
    my $symbol = $rawSymbol->{SYMBOL};

    print STDERR "Current state " . $self->{CURRENT} . ", got symbol " . $symbol . "\n";
    if (!exists $state->{$symbol} && exists $state->{'*'}) {
        print STDERR "Unrecognized symbol " . $symbol . ", using *\n";
        $symbol = "*";
    }

    # Do some action!
    $state->{$symbol}->{ACTION}($self, $rawSymbol)
        if ref $state->{$symbol}->{ACTION};

    # Switch state
    if (exists $state->{$symbol}->{NEXT}) {

```

```

        $self->{CURRENT} = $state->{$symbol}->{NEXT};
    } else {
        die "Don't know how to handle symbol " . $rawSymbol->{SYMBOL};
    }

    return $self;
}

1;

```

Я полагаю что названия всех методов говорят сами за себя. Остановиться пожалуй стоит на методах **normalize** и **process**. Первый преобразовывает входную строку в хэш, содержащий поле SYMBOL с упрощенным до алфавита автомата входным символом. А **process** собственно осуществляет «тактирование» процессов перехода между состояниями, обрабатывая очередной сигнал.

Вкратце рассмотрим как это работает. Сердцем класса является таблица переходов STATES, представляющая из себя хэш хэшей хэшей :) тут проста, на первом уровне мы имеем список состояний (STATE) и связанных с ними атрибутов. Так как переход определяется только входным символом (SYMBOL), то соответственно этими атрибутами будут собственно допустимые для этого состояния сигналы. Ну а по сигналу мы можем определить следующее состояние (NEXT) и, в довесок, выполняемое действие (ACTION), которое является всего лишь ссылкой на

Т.е. по **process** мы сначала получаем символ входного алфавита из входной строки (**normalize**), затем получаем текущее состояние из списка состояний. Смотрим, определен ли для него входящий символ. Если не определен, то считаем что к нам прилетел "*" — любой другой символ. Далее, смотрим, определено ли действие для пары состояние-сигнал. Если определено, выполняем. И переходим на следующее состояние, если оно определено (меняем переменную CURRENT). Если не определено, то фактически это фатальная ошибка для нашего автомата.

В учебных целях конечно же мы выводим информацию о переключении автомата в STDERR, я думаю вам не составит труда при необходимости завернуть этот вывод в лог для дебага или т.п.

Пример

Давайте попробуем что-нибудь реализовать. Пускай это будет некое подобие чат-бота. Положим в его основу следующие правила:

- Перед выполнением команд надо сначала представиться (**login**)
- По команде **memorize** будем запоминать все что вводит пользователь, окончание по команде **exit**
- По команде **say N** выводим запомненную фразу номер N
- Завершение сеанса будет происходить по команде **exit**

Итак, составим таблицу переходов для этого примера:

Символ	Состояние	Следующее состояние	Действие
LOGIN	INIT	SESSION	Открываем сессию
*	INIT	INIT	-
*	SESSION	SESSION	-
SAY	SESSION	SESSION	Выводим строку номер N
EXIT	SESSION	INIT	-
MEMORIZE	SESSION	STORE	-
*	STORE	STORE	Сохраняем строку в буфер
EXIT	STORE	SESSION	-

Итого, алфавит автомата состоит из символов (LOGIN, MEMORIZE, SAY, EXIT, *), автомат имеет 3 состояния (INIT, SESSION и STORE).

Что ж, давайте его реализуем. В первую очередь зададим в конструкторе таблицу переходов, где необходимо — добавим ссылки на вызов методов. Никаких сложностей :)

```

package ChatBot;
use FSM;
@ISA = ("FSM");

use strict;
use Carp;
use vars qw($AUTOLOAD);

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;

    my $self = $class->SUPER::new("INIT");

    $self->addState(STATE => "INIT",    SYMBOL => "*",        NEXT => "INIT",    ACTION => \&doIntroduce);
    $self->addState(STATE => "INIT",    SYMBOL => "LOGIN", NEXT => "SESSION", ACTION => \&doLogin);
    $self->addState(STATE => "INIT",    SYMBOL => "EXIT",  NEXT => "INIT",    ACTION => \&doQuit);
    $self->addState(STATE => "SESSION", SYMBOL => "*",        NEXT => "SESSION");
    $self->addState(STATE => "SESSION", SYMBOL => "EXIT",  NEXT => "INIT");
    $self->addState(STATE => "SESSION", SYMBOL => "SAY",    NEXT => "SESSION", ACTION => \&doSay);
    $self->addState(STATE => "SESSION", SYMBOL => "MEMORIZE",NEXT => "STORE");
    $self->addState(STATE => "STORE",   SYMBOL => "*",        NEXT => "STORE",   ACTION => \&doRemember);
    $self->addState(STATE => "STORE",   SYMBOL => "EXIT",  NEXT => "SESSION");

    $self->{SESSION} = {};
    $self->{LOGIN}   = "";

    return $self;
}

sub normalize {
    my ($self, $symbol) = @_;
    my $ret = {};

    if ($symbol =~ /\^(\\S+)(.*)$/ ) {
        $ret->{SYMBOL} = uc $1;
        $ret->{DATA}   = $2;
        $ret->{RAW}     = $symbol;
    } else {
        $ret->{SYMBOL} = "*";
        $ret->{DATA}   = $symbol;
        $ret->{RAW}     = $symbol;
    }

    return $ret;
}

sub doIntroduce {
    my $self = shift;
    print "Please introduce yourself first!\n";
    return $self;
}

sub doLogin {
    my ($self, $symbol) = @_;
    print "Welcome," . $symbol->{DATA} . "\n";
    $self->{LOGIN} = $symbol->{DATA};
    $self->{SESSION}->{$self->{LOGIN}} = () unless exists $self->{SESSION}->{$self->{LOGIN}};
    return $self;
}

sub doSay {
    my ($self, $symbol) = @_;
    if (defined $self->{SESSION}->{$self->{LOGIN}}->[$symbol->{DATA}]) {
        print $self->{SESSION}->{$self->{LOGIN}}->[$symbol->{DATA}];
    } else {
        print "No record\n";
    }
    return $self;
}

sub doRemember {
    my ($self, $symbol) = @_;
```

```

push @{ $self->{SESSION}->{$self->{LOGIN}} }, $symbol->{RAW};
return $self;
}

sub doQuit {
    my ($self, $symbol) = @_;
    print "Bye bye!\n";
    exit;
    return $self;
}

1;

```

Диаграмма переходов выглядит следующим образом.

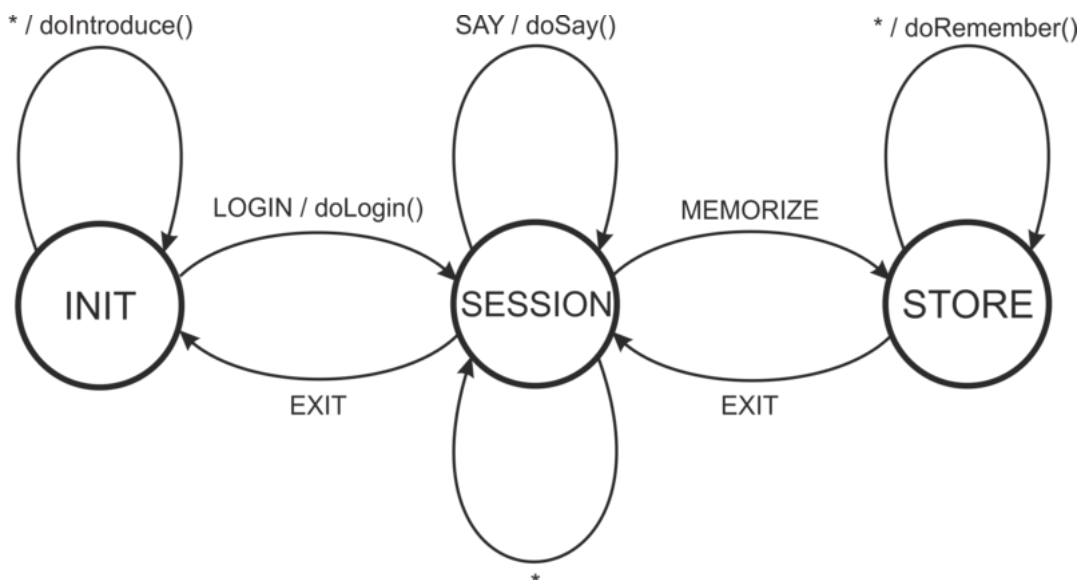


Таблица переходов в данном случае будет иметь следующий вид:

```

{
    'INIT' => {
        '*' => {
            'ACTION' => \&doIntroduce,
            'NEXT' => 'INIT'
        },
        'LOGIN' => {
            'ACTION' => \&doLogin,
            'NEXT' => 'SESSION'
        },
        'EXIT' => {
            'ACTION' => \&doQuit,
            'NEXT' => 'INIT'
        }
    },
    'STORE' => {
        '*' => {
            'ACTION' => \&doRemember,
            'NEXT' => 'STORE'
        },
        'EXIT' => {
            'NEXT' => 'SESSION'
        }
    },
    'SESSION' => {
        'SAY' => {
            'ACTION' => \&doSay,
            'NEXT' => 'SESSION'
        },
        '*' => {
            'NEXT' => 'SESSION'
        },
        'MEMORIZE' => {
            'NEXT' => 'STORE'
        }
    }
}

```

```
        },  
        'EXIT' => {  
            'NEXT' => 'INIT'  
        }  
    }  
}
```

Напишем простейшую программку с использованием этих классов.

```
use ChatBot;  
  
$bot = ChatBot->new();  
while(<>) {  
    $bot->process($_);  
}
```

Ну и простенькая проверка. На ввод дадим следующую последовательность.

```
hello world!  
login %username%  
hello world!  
say 3  
memorize  
hey, do you really remember everything i would say?  
let's check  
exit  
say 0  
exit  
hello  
login %username%  
say 1  
exit
```

Что же мы получим на выходе?

```
Please introduce yourself first!  
Welcome, %username%  
No record  
hey, do you really remember everything i would say?  
Please introduce yourself first!  
Welcome, %username%  
let's check
```

В общем, попробуйте сами.

Заключение

Таким образом мы реализовали простейший конечный автомат. Вроде бы ничего сложного? Где это может пригодиться? Ну с чат-ботами все понятно. Примерно тоже самое получается если на другой стороне будет не человек, а железка — передавая команды и слушая ответы мы можем написать бота, крутящего настройки маршрутизатора, например. Интерактивные командные интерфейсы? Да мы собственно его и реализовали! Хотите подключиться к сервису, использующему протокол с набором состояний? Нет ничего проще!

Надеюсь, моя статья была хоть кому-нибудь полезна. Да, реализация примитивная, существует великое множество уже готовых. Но ведь интересно попробовать сделать что-то с нуля, неправда ли?

Буду рад любым комментариям.

Метки: теория автоматов, конечный автомат, perl

↑ +13 ↓ 145 👁 75,8k 💬 20



↑ 76,2 ↓
Карма

0,1
Рейтинг

38
Подписчики

@divanikus

✉ Написать

✍ Подписать

Пользователь

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

30 января 2016 в 18:28

Простейшие клеточные автоматы и их практическое применение

↑ +87

👁 56,4k

🔖 339

💬 15

19 сентября 2015 в 22:24

Интересные примеры клеточных автоматов

↑ +51

👁 25,2k

🔖 138

💬 3

26 апреля 2010 в 19:06






Самостоятельное изучение схемотехники. Абстрактный автомат. Часть 2

↑ +70

👁 46,9k


🔖 233

💬 37

ВАКАНСИИ		Мой к
	Python разработчик Точка – банк для предпринимателей · Екатеринбург	до 150 (
	Ведущий программист 1С АвтоГЕРМЕС · Москва	от 130 000 до 160 (
	Системный программист Linux Кадровый центр «21 век» · Москва	от 150 000 до 250 (
	Linux администратор в техподдержку хостинг провайдера ProfitServer · Челябинск	от 40 (
	Project manager FunCorp · Москва	до 200 (
Все вакансии		

Комментарии 20

Отслеживать новые в ☐ почте ☐



Kolonist

06.04.12 в 00:14

🔖

🔖

↑

А есть ли смысл городить огород, если автомат реализуется минимумом синтаксических конструкций при помощи элементарных while-switch или да: while-if?

```
q = q0; //начальное состояние
while (q != qe) { //пока текущее состояние q не равно конечному состоянию qe
    switch(q) {
        case q0:
            //что-нибудь делаем, если надо - меняем q на какое-нибудь q1 или q2, или qe
            break;

        case q1:
            //что-нибудь делаем
            break;
```

```

    case q2:
        //что-нибудь делаем
        break;
    }
}

```

Ответить



divanikus 06.04.12 в 00:31 # 📖 📄 🔄



Чем больше состояний, тем больше будет уровней вложенности. Будет ооочень глубокая лесенка.

Ответить



NYM 06.04.12 в 12:40 # 📖 📄 🔄



Согласен. Была такая практика: писали некоторую логику «классически» и никак не выходил «каменный цветочек», переписали на конечных автоматах, тоже не легко получилось, но удалось завершить и как-то структурировать поведение и снизить кол-во странных логических багов. К логики получилось меньше, чем при классике, но вот таблица переходов стала великовата и страшновата :)

Ответить



igudym 07.04.12 в 15:39 # 📖 📄 🔄



Это почему это? Уровней вложенностей будет ровно два: внешний свитч по входному символу, и внутренние свитчи по состоянию автомата (м наоборот).

По виду это почти не отличается от приведенной в статье таблицы переходов.

Ответить



divanikus 07.04.12 в 18:43 # 📖 📄 🔄



Предлагаю не быть голословным, а написать и показать нам рабочий пример.

Ответить



igudym 07.04.12 в 19:08 # 📖 📄 🔄



Запросто, на питоне. Фактически это таже таблица переходов. Только без излишних сложностей в ее интерпретации, да и работать будет в два быстрее.

```

state = 'INIT'
while state != 'QUIT':
    symbol = read_input()
    if state == 'INIT':
        if symbol == '*':
            doIntroduce()
            state = 'INIT'
        elif symbol == 'LOGIN':
            doLogin()
            state = 'SESSION'
        elif symbol == 'EXIT':
            doQuit()
            state = 'QUIT'
    elif state == 'STORE':
        if symbol == '*':
            doRemember()
            state = 'STORE'
        elif symbol == 'EXIT':
            state = 'SESSION'
    elif state == 'SESSION':
        if symbol == 'SAY':
            doSay()
            state = 'SESSION'
        elif symbol == '*':
            state = 'SESSION'
    elif state == 'MEMORIZE':
        state = 'STORE'
    elif state == 'EXIT':
        state = 'INIT'

```

Ответить



divanikus 07.04.12 в 19:34 # 📖 📄 🔄



Ну пожалуй соглашусь, ДКА действительно можно так изобразить из-за его довольно простой природы. Только наверное не в while на засовывать, а в некий метод. Плюс таким образом мы теряем возможность менять конфигурацию автомата в рантайме, например, подгружая список состояний из файла и т.п.

Кстати, последние два `elif` должны быть над символами и внутри `SESSION`.

Ответить



dlancer 06.04.12 в 01:37 # 📌 📄 ↻



Попробуйте такой лесенкой описать простую игрушку (ходят монстры, игрок и стреляют). Через пару дней добавления фиш вы плюнете, все сотрете и напишите все уже так, как это описано в статье. Обычно для полного счастья еще добавляют возможность мониторинга переходных состояний. С их поддержкой например можно делать резкий переход в другое состояние, если монстра например убили.

Ответить



av0000 06.04.12 в 10:08 # 📌



Рискну нарваться, но «всё хорошо, вот если бы ещё не перл»!

А если серьёзно, особенно учитывая теги, хотелось бы каких-то картинок или сильно упрощённого псевдо-кода для демонстрации алгоритма. Ибо продираться сквозь, например, "\$self->{STATES}->{\$state}->{\$symbol}->{ACTION}" несколько утомительно ;) То есть, (заранее) зная, как оно должно работать, вникнуть можно, а вот разобраться «с нуля» — тяжко

Мне несколько проще — я перл просто лет 7 как забыл, а тем, кто его вообще не видел?

Ответить



HomoLuden 06.04.12 в 11:22 # 📌 📄 ↻



Мне тоже показалось сложным читать этот код. Вот было б на Ruby написали, как в *Clever Algorithms*, стало б совсем замечательно. Спасибо автору за «*normelize*» — помогло переосмыслить и понять куда дальше двигаться.

Ответить



divanikus 06.04.12 в 11:26 # 📌 📄 ↻



У меня к сожалению опыта на руби нет, но ради такого дела можно попробовать, например, в следующей статье. Или еще на каком языке :)

Ответить



1x1 06.04.12 в 13:44 # 📌 📄 ↻



Для вас будет больше пользы доработать этот код. Минимальные изменения сильно упростят его понимание, а привычка писать читабельн код поможет и с другими языками.

Ответить



divanikus 06.04.12 в 13:49 # 📌 📄 ↻



Ну я даже не знаю, приведите пример что ли.

Ответить



1x1 06.04.12 в 16:55 # 📌 📄 ↻



```
my $current_state = $self->{STATES}->{$state};
```

Ответить



divanikus 06.04.12 в 18:16 # 📌 📄 ↻



Лично мне кажется, что в процедуре на 5 строчек это уже совсем детский сад получается. Хотя если народ просит, могу и сделать.

Ответить



1x1 06.04.12 в 19:28 # 📌 📄 ↻



Не важно количество строк.

- Если что-то [неизменное] вычисляется более одного раза, создавайте переменную.
- Если что-то сложнее $2+2$ используется в разных местах — выносите в функцию/метод.
- Если что-то многократно повторяется и можно обойтись без этого — переделывайте.

Ну и, разумеется: если для чего-то уже сделан метод, то использовать его, а не повторять тот же код.

Ответить

 divanikus 07.04.12 в 15:13    

Мне известны эти правила и я ими стараюсь всегда пользоваться. Но в то же время не стоит излишне плодить сущностей.


Ответить

 rumkin 07.04.12 в 13:39    

Короче, пишите на псевдокоде с динамической типизацией, но без замыканий и лямбд)))))


А, если серьезно, когда важнее алгоритм, а не готовый код, я рисую схему а-ля UML: наглядно, абстрактно и к коду не дое**ться

Ответить

 rumkin 07.04.12 в 13:41    

P.S. Спасибо за интересную статью. Код не разобрал, но принцип понял! В общем респект и прочие почести.

Ответить

 divanikus 07.04.12 в 15:14  

А тем временем, добавил в статью более подробное описание алгоритма работы, а также чуть более глубокое освещение работы примера :)

Ответить

Написать комментарий

Предпросмотр

Отправить

☐ Markdown

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Б — Брутальность. Официальный сайт Федерации настольного тенниса Республики Башкортостан (ФНТ РБ)

 +42  14,9k  23  70

Зацените: сделал стол

 +141  17,7k  217  261

Python для ребёнка: выбор самоучителя

 +40  10,9k  242  44

Мессенджеры vs соцсети vs ... — анонс нового проекта

 +7  4k  19  37

Минтруд: тестовое задание — это трудовые отношения

 +23  11,6k  97  288

[Трекер](#)

[Хабы](#)

[Помощь](#)

[Тарифы](#)

[Диалоги](#)

[Компании](#)

[Документация](#)

[Контент](#)

[Настройки](#)

[Пользователи](#)

[Соглашение](#)

[Семинары](#)

[ППА](#)

[Песочница](#)

[Конфиденциальность](#)



© 2006 – 2018 «ТМ»

[О сайте](#)

[Служба поддержки](#)

[Мобильная версия](#)

