|  |
| --- |
| **Страница 1** |

*Част 1*

*Основи от рубин*

T той цел на тази част на книгата е да ви даде широк но практически фондации

ния слой, върху който да се изгради и към който да се закрепи по-нататъшното проучване на

Ruby, които следват в части 2 и 3. Ще започнем с глава за начално зареждане

вашата грамотност на Ruby; след като преминете през първата глава, ще можете да стартирате

Програмите на Ruby са удобни и имат добра представа за оформлението на типичен Ruby

инсталация. Започвайки с глава 2, ще влезем в детайлите на Руби

гейдж. Ruby е обектно-ориентиран език и колкото по-скоро се потопите в това как

Руби се справя с предмети, толкова по-добре. Съответно обектите ще служат и като начин за

начална дискусия на езика (и вашите познания за него) и като a

златна нишка, водеща ни към по-нататъшни теми и техники.

Обектите се създават от класове и в глава 3 ще научите как работят класовете.

Обсъждането на класовете е последвано от разглеждане на модулите в глава 4. Модули

ви позволяват да прецизирате класове и обекти, като разделите част от обектния дизайн

в отделни, многократно използвани единици код. За да разберете Ruby програмите - и двете си

собствени и чужди “- трябва да знаете за представата на Руби за текущ *обект по подразбиране* ,

известен с ключовата дума self. Глава 5 ще ви отведе дълбоко в концепцията за себе си,

заедно с третирането на работата на Ruby с променлива видимост и обхват.

В глава 6, последната в тази част на книгата, ще научите за контролния поток

в програмите Ruby - т.е. как да насочвате интерпретатора на Ruby чрез условия

национална (ако) логика, как да се циклира многократно през кода и дори как да се прекъсва

далеч от нормалното изпълнение на програмата, когато възникне грешка. До края на глава

ter 6, ще мислите заедно с Руби, докато пишете и разработвате кода си.

Заглавието на тази част е „Основи от рубин“, което очевидно предполага това

това, което е тук, ще бъде надградено по-късно. И това е вярно. Но това не означава, че

|  |
| --- |
| **Страница 2** |

материалът в част 1 не е важен сам по себе си. Както ще видите, след като ги прочетете, тези шест

главите ви представят реални техники на Ruby, реален код и информация, която ще използвате

всеки път, когато пишете или изпълнявате програма Ruby. Не е „основите“, защото

ще го научите веднъж и след това ще го игнорирате, но тъй като има толкова много *повече* за Руби

тепърва да последва!

|  |
| --- |
| **Страница 3** |

**3**

*Буутстрапинг*

*вашата грамотност на Ruby*

Тази книга ще ви даде основа в Ruby, а тази глава ще ви даде основите-

ция фондация. Целта на главата е да ви задейства в изучаването на Руби

с достатъчно знания и умения, за да преминете удобно към онова, което се крие отвъд.

Ще разгледаме основните синтаксис и техники на Ruby и как работи Ruby: какво

правите, когато пишете програма, как карате Ruby да стартира вашата програма и

как разделяте програма на повече от един файл. Ще научите няколко от

превключватели, които променят начина, по който интерпретаторът на Ruby (програмата с името ruby,

към които подавате вашите програмни файлове за изпълнение), както и как да използвате

някои важни помощни инструменти, предназначени да улеснят живота ви като рубист

по-продуктивен.

***Тази глава обхваща***

Комплект за оцеляване на синтаксис Ruby

Основни инструкции за програмиране на Ruby: писане,

програми за запазване, стартиране и проверка на грешки

Обиколка на инсталацията Ruby

Механиката на разширенията Ruby

Готовите инструменти на Ruby за командния ред,

включително интерактивния интерпретатор на Ruby (irb)

|  |
| --- |
| **Страница 4** |

**4**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

Главата се основава на изглед на целия рубинен пейзаж, на който е разделен

три основни нива:

Основен език: принципи на проектиране, синтаксис и семантика

Разширения и библиотеки, които се доставят с Ruby, и съоръженията за добавяне на разширения

вашите собствени

Инструменти за команден ред, които се доставят с Ruby, с които стартирате интерпретатора

и някои други важни помощни програми

Не винаги е възможно да се говори за тези три нива изолирано - в крайна сметка те са

блокиране на части от една система, но ние ще ги обсъдим отделно, колкото

в тази глава. Във всеки случай можете да използвате описанията на три нива като фикси

закачете подтемите, където и да са въведени.

Тази първа глава също не съществува единствено в услуга на по-късните глави. Той има съдържание в

сам по себе си: ще научите истински техники на Ruby и важни моменти относно дизайна

на езика. Целта е да ви зареди или да стартирате, но дори и този процес ще го направи

включват внимателно проучване на някои ключови аспекти на езика Ruby.

***1.1***

Целта на този раздел е да ви накара да отидете с Ruby. Необходим е подход, основан на ширина:

ще преминем през целия цикъл на изучаване на синтаксис, писане на код и

стартиране на някои програми.

В този момент трябва да имате инсталиран Ruby на вашия компютър. 1 Примерите в

тази книга използва Ruby 2.1.0. Имате нужда и от текстов редактор (можете да използвате всеки редактор, който харесвате,

стига да е редактор на обикновен текст, а не текстов процесор) и директория (известна още като

папка), в която да съхранявате вашите програмни файлове Ruby. Можете да назовете тази директория

rubycode или rubysamples - всяко име е добре, стига да е отделно от друга работа

области, така че да можете да следите вашите програмни файлове за практика.

Рубин, **рубин** и ... РУБИН ?!

Ruby е език за програмиране. Говорим за неща като „да научим Руби“ и ние

задавайте въпроси като: „Познавате ли Руби?“ Версията с малки букви, рубин, е ком-

путер програма; по-специално, това е интерпретаторът Ruby, програмата, която чете вашия

програми и ги изпълнява. Ще видите това име, използвано в изречения като „Тичах рубин

в моя файл, но нищо не се е случило “или„ Какъв е пълният път към вашия ruby ​​execut-

способен?" И накрая, има RUBY - или, по-точно, няма. Руби не е акро-

nym и никога не е правилно да се пише с всички главни букви. Хората правят това, както правят

правят (също погрешно) с Perl, може би защото са свикнали да виждат език

имена като BASIC и COBOL. Руби не е такъв език. Това е Ruby

гюдж, рубин за преводача.

1

Можете да намерите пълни актуални инструкции за инсталиране на Ruby на [http://ruby-lang.org.](https://translate.google.com/translate?hl=en&prev=_t&sl=auto&tl=bg&u=http://ruby-lang.org)

|  |
| --- |
| **Страница 5** |

**5**

С инсталиран Ruby и създадена вашата работна област, нека продължим да зареждаме вашия Ruby

грамотност, така че имаме споделена основа, върху която да продължим да изграждаме и изследваме.

Едно нещо, от което се нуждаете, е достатъчно излагане на основния синтаксис на Ruby, за да започнете.

***1.1.1***

***Комплект за оцеляване на синтаксис Ruby***

Следващите три таблици обобщават някои функции на Ruby, в които ще намерите полезни

разбиране на примерите в тази глава и започване на експерименти с Руби.

Не е нужно да ги запомняте, но ги преглеждайте и се обръщайте към тях по-късно

колкото е необходимо.

Таблица 1.1 съдържа някои от основните операции на Ruby. Таблица 1.2 обхваща извличането

основен вход от клавиатурата, изпращане на изход към екрана и основен условен

изявления. Таблица 1.3 описва накратко специалните обекти и синтаксиса на Ruby за коментари.

Няколко основни аспекта на Ruby и Ruby синтаксиса са твърде ангажирани за обобщение

в таблица. Трябва да можете да разпознаете шепа различни идентификатори на Ruby

и най-вече трябва да усетите какво представлява обектът в Ruby и какво извиква метод

изглежда като. След това ще разгледаме и двата аспекта на езика.

Интерактивната конзолна програма Ruby (irb), вашият нов най-добър приятел

Помощната програма irb се доставя с Ruby и е най-често използваният инструмент за команден ред Ruby

различни от самия преводач. След като стартирате irb, въвеждате Ruby код в него и той

изпълнява кода и отпечатва получената стойност.

Въведете irb в командния ред и въведете примерен код, когато го срещнете в текста.

Например:

>> 100 + 32

=> 132

Наличието на отворена irb сесия означава, че можете да тествате фрагменти на Ruby по всяко време и по всяко време

количество. Повечето разработчици на Ruby намират irb за незаменим и ще видите няколко примера

за използването му, докато продължаваме през тази глава.

Примерите за irb, които ще видите в тази книга, ще използват опция за команден ред, която прави

irb изход по-лесен за четене:

irb --прост-бърз

Ако искате да видите ефекта на опцията --simple-prompt, опитайте да стартирате irb с

и без него. Както ще видите, простото подсказване поддържа екрана ви много по-ясен. The

подканата по подразбиране (непроста) показва повече информация, като например броя на редовите номера

за вашата интерактивна сесия; но за примерите, които ще разгледаме, простият ред

е достатъчно.

Тъй като irb е един от инструментите на командния ред, който се доставя с Ruby, той не се обсъжда

подробно до раздел 1.4.2. Чувствайте се свободни да преминете към този раздел и да погледнете; това е

доста директно.

|  |
| --- |
| **Страница 6** |

**6**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

Таблица 1.1 Основни операции в Ruby

Операция

Примери)

Коментари

Аритметика

2 + 3 (добавяне)

2 - 3 (изваждане)

2 \* 3 (умножение)

2/3 (разделяне)

10,3 + 20,25

103 - 202,5

32,9 \* 10

100,0 / 0,23

Всички тези операции работят върху цели числа или плаващи

номера на точки ( *плува* ). Смесване на цели числа и плувки

заедно, както правят някои от примерите, се получава a

резултат с плаваща запетая.

Имайте предвид, че трябва да пишете 0.23, а не .23 .

Възлагане

x = 1

низ = "Здравей"

Тази операция обвързва локална променлива (вляво) с

обект (отдясно). Засега можете да се сетите

обект като стойност, представена от променливата.

Сравнете две стойности

x == y

Обърнете внимание на двата знака за равенство (не само един, както в

възлагане).

Преобразуване на числово

низ към число

x = "100" .до\_i

s = "100"

x = s.to\_i

За да извършите аритметика, трябва да сте сигурни, че сте

имат числа, а не низове от символи.

to\_i извършва преобразуване от низ към цяло число.

Таблица 1.2 Основни методи за вход / изход и контрол на потока в Ruby

Операция

Примери)

Коментари

Отпечатайте нещо на

на екрана

печат "Здравей"

поставя "Здравей"

x = "Здравей"

поставя х

x = "Здравей"

отпечатайте x

x = "Здравей"

px

put добавя нов ред към низа, който извежда, ако има

вече не е накрая; печат не.

print отпечатва точно това, на което е казано и оставя

курсора в края. (Забележка: На някои платформи допълнително

ред се извежда автоматично в края на програмата.)

p извежда инспектиращ низ, който може да съдържа допълнително

информация за това какво отпечатва.

Вземете ред ключ-

вход на борда

получава

низ = получава

Можете да присвоите реда за въвеждане директно на променлива (

променлив низ във втория пример).

Условна

екзекуция

ако x == y

поставя "Да!"

друго

поставя "Не!"

край

Условните твърдения винаги завършват с думата край .

Повече за тях в глава 6.

Таблица 1.3 Специални обекти и коментари на Ruby

Операция

Примери)

Коментари

Специална стойност

обекти

вярно

невярно

нула

Обектите true и false често служат като възвратна стойност

ues за условни изрази. Обектът nil е един вид

на „nonobject“, което показва липса на стойност или

резултат. false и nil предизвикват условен израз

да се провали; всички други обекти (включително истина , разбира се, но

включително 0 и празни низове) го карат да успее.

Повече за тях в глава 7.

|  |
| --- |
| **Страница 7** |

**7**

***1.1.2***

***Разнообразието от идентификатори на Ruby***

Ruby има малък брой типове идентификатори, които бихте искали да можете да откриете и различите

се отклоняват един от друг с един поглед. Родословното дърво на идентификатора изглежда така:

Променливи:

- Местен

- Инстанция

- Клас

- Глобално

Константи

Ключови думи

Имена на методите

Това е малко семейство и лесно се учи. Ще ги проучим тук. Имайте предвид, че това

Целта на раздела е да ви научи да разпознавате различните идентификатори. Също така ще научите a

много повече в цялата книга за това кога и как да ги използваме. Това е само първото

урок по грамотност на идентификатора.

V АРИАБИ

*Локалните променливи* започват с малка буква или долна черта и се състоят от букви,

долни черти и / или цифри. x, string, abc, start\_value и firstName са валидни

локални имена на променливи. Имайте предвид обаче, че конвенцията на Ruby е да се използват подчертавания

вместо камила, когато съставя имена на местни променливи от множество думи -

например first\_name, а не firstName.

*Екземплярни променливи* , които служат за съхраняване на информация за отделни лица

обекти, винаги започнете с единичен знак (@) и след това се състоят от едни и същи символи

ter, зададени като локални променливи - например @age и @last\_name. Въпреки че местна вариа-

не може да започне с главна буква, променливата на екземпляра може да има такава в първата

позиция след знака at (макар че може да няма цифра в тази позиция). Но обикновено

знакът след знака at е малка буква.

*Променливи на класа* , които съхраняват информация за йерархия на класа (отново, не се притеснявайте

относно семантиката на този етап), следвайте същите правила като променливите на екземпляра, с изключение на

че започват с две при знаци - например @@ running\_total.

*Глобалните променливи* се разпознават по техния водещ знак за долар ($) - например,

$ население. Сегментът след знака за долар не следва именуване на местни променливи

Обект по подразбиране

себе си

Ключовата дума self се отнася до обекта по подразбиране. Азът е a

роля, която играят различните обекти, в зависимост от изпълнението

ционния контекст. Извиквания на методи, които не посочват повикване

обект се извикват на себе си . Повече за това в глава 5.

Поставете коментари

кодови файлове

# Коментар

x = 1 # Коментар

Коментарите се пренебрегват от преводача.

Таблица 1.3 Специални обекти и коментари на Ruby ***(продължение)***

Операция

Примери)

Коментари

|  |
| --- |
| **Страница 8** |

**8**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

конвенции; има глобални променливи, наречени $ :, $ 1 и $ /, както и $ stdin и

$ LOAD\_PATH. Докато започва със знак за долар, това е глобална променлива. Колкото до

не-буквени, единствените такива идентификатори, които вероятно ще видите, са предварително дефинирани,

така че не е нужно да се притеснявате кои пунктуационни знаци са законни и кои не.

Таблица 1.4 обобщава правилата за именуване на променливи на Ruby.

C НЕСТАНТИ

Константите започват с главна буква. A, String, FirstName и STDIN са всички

валидни имена на константи. Конвенцията на Ruby е да се използва или калъф за камила (FirstName), или

подчертани разделени главни думи (FIRST\_NAME) при съставяне на константни имена

от множество думи.

K EYWORDS

Ruby има множество ключови думи: предварително дефинирани, запазени термини, свързани с конкретни

граматични задачи и контексти. Ключовите думи включват def (за дефиниции на метод), клас

(за дефиниции на класове), ако (условно изпълнение) и \_\_FILE\_\_ (името на файла

в момента се изпълнява). Има около 40 от тях и обикновено са ниски,

идентификатори с една дума (за разлика от съставените с подчертаване).

М ИМЕНА НА ЕТОД

Имената на методите в Ruby следват същите правила и конвенции като локалните променливи

(с изключение на това, че те могат да завършат с?,! или =, със значение, което ще видите по-късно). Това е от

дизайн: методите не привличат вниманието към себе си като методи, а по-скоро се смесват

текстурата на програма като просто изрази, които осигуряват стойност. В някои контексти

не можете да разберете само като погледнете израз дали виждате локална променлива или

име на метод - и това е умишлено.

Говорейки за методи, сега, след като имате пътна карта за идентификаторите на Ruby, нека вземем

да се върнем към някои езикови семантики - по-специално, най-важната роля на обекта

и нейните методи.

***1.1.3***

***Извиквания на методи, съобщения и обекти на Ruby***

Руби вижда всички структури от данни и стойности - от прости скаларни (атомни) стойности като инте-

gers и низове, до сложни структури от данни като масиви - като *обекти* . Всеки обект е способен

за разбиране на определен набор от *съобщения* . Всяко съобщение, което обектът разбира

Таблица 1.4 Валидни имена на променливи в Ruby по тип променлива

Тип

Конвенция за руби

Неконвенционален

Местен

първо име

firstName, \_firstName, \_\_firstName, name1

Инстанция

@първо име

@ Първо име, @ първо име, @ име1

Клас

@@първо име

@@ Първо име, @@ първо име, @@ име1

Глобален

$ FIRST\_NAME

$ first\_name, $ firstName, $ name1

|  |
| --- |
| **Страница 9** |

**9**

съответства директно на *метод* - име, изпълнима програма, чието изпълнение на

обектът има способността да задейства.

Обектите са представени или от литерални конструктори - като кавички за

низове - или от променливи, към които са били обвързани. Изпращането на съобщение се постига чрез

специалният точков оператор: съобщението отдясно на точката се изпраща до обекта до

вляво от точката. (Има и други, по-специализирани начини за изпращане на съобщения до обекти,

но точката е най-често срещаният и основен начин.) Помислете за този пример от

таблица 1.1:

x = "100" .до\_i

Точката означава, че съобщението to\_i се изпраща до низа "100". Низът

"100" се нарича *получател* на съобщението. Можем също да кажем, че методът to\_i е

се *извиква* на низа "100". Резултатът от извикването на метода - цялото число 100—

служи като дясната страна на присвояването на променливата x.

Методите могат да приемат *аргументи* , които също са обекти. (Почти всичко в Ruby е

обект, въпреки че някои синтактични структури, които ви помагат да създавате и манипулирате

обектите сами по себе си не са обекти.) Ето извикване на метод с аргумент:

x = "100" .to\_i (9)

Извикването на to\_i на 100 с аргумент 9 генерира десетично цяло число, еквивалентно на

базовата деветка 100: x вече е равна на 81 десетична запетая.

Този пример също показва използването на скоби около аргументите на метода. Тези

скобите обикновено не са задължителни, но в по-сложни случаи може да се наложи да

изясни какво в противен случай може да бъде неясноти в синтаксиса. Много програмисти използват

скоби в повечето или всички извиквания на методи, само за да бъде в безопасност.

Цялата вселена на Ruby програма се състои от обекти и съобщения, които

им се изпращат. Като програмист на Ruby, прекарвате по-голямата част от времето си или

извличане на нещата, които искате обектите да могат да правят (чрез дефиниране на методи) или задаване на

обекти, които да правят тези неща (като им изпращат съобщения).

Защо двойната терминология?

Защо да си правим труда да казваме едновременно „изпращане на съобщението до\_i“ и „извикване на метода до\_i“?

Защо има два начина за описване на една и съща операция? Защото те не са съвсем

един и същ. По-голямата част от времето изпращате съобщение до получател и обекта

изпълнява съответния метод. Но понякога няма съответно

метод. Можете да поставите всичко отдясно на точката и няма гаранция за това

приемникът ще има метод, който съответства на съобщението, което изпращате.

Ако това звучи като хаос, не е, защото обектите могат да прихващат неизвестни съобщения

и се опитайте да ги осмислите. Рамката за уеб разработка Ruby on Rails, за

например, използва широко техниката за изпращане на неизвестни съобщения до обекти,

прихващането на тези съобщения и осмислянето им в движение въз основа на динамика

условия като имената на колоните в таблиците на текущата база данни.

|  |
| --- |
| **Страница 10** |

**10**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

Ще проучим всичко това в много по-голяма дълбочина по-късно в книгата. Отново, това кратко

скица е само част от процеса на зареждане на вашата грамотност Ruby. Когато видите a

точка в това, което иначе би било необяснима позиция, трябва да го тълкувате като

съобщение (отдясно), изпратено до обект (отляво). Имайте предвид и това

някои извикване на метод под формата на *bareword* -стил извиквания, като поканата да поставя в

този пример:

поставя „Здравей“.

Тук, въпреки липсата на точка за изпращане на съобщения и изричен приемник за

съобщение, изпращаме съобщенията с аргумент „Здравей“. към обект:

обекта по подразбиране самостоятелно. Винаги има самодефинирано, когато програмата ви работи,

въпреки че кой обект се самопроменя, в съответствие с конкретни правила. Ще научите много

повече за себе си в глава 5. Засега вземете под внимание факта, че една гола дума като поставя

може да бъде извикване на метод.

Най-важната концепция в Ruby е концепцията на обекта. Тясно свързани,

и да играе важна поддържаща роля, е концепцията на *класа* .

T HE ПРОИЗХОД НА ОБЕКТИ в класове

*Класовете* дефинират клъстери на поведение или функционалност и всеки обект е екземпляр на

точно един клас. Ruby предоставя голям брой вградени класове, представляващи

важни основни типове данни (класове като String, Array и Fixnum). Всеки

когато създавате низ обект, вие сте създали екземпляр на класа String.

Можете също да напишете свои собствени класове. Можете дори да модифицирате съществуващите класове Ruby; ако

не ви харесва поведението на низове или масиви, можете да го промените. Почти винаги е a

лоша идея да го направя, но Руби го позволява. (Ще разгледаме плюсовете и минусите на правенето

промени във вградените класове в глава 13.)

Въпреки че всеки обект на Ruby е екземпляр на клас, концепцията за клас е по-малка

важно от понятието обект. Това е така, защото обектите могат да се променят, придобивайки

методи и поведения, които не са били дефинирани в техния клас. Класът е отговорен за

стартиране на обекта в съществуване, процес, известен като *инстанциране* ; но обектът,

след това има собствен живот.

Способността на обектите да възприемат поведение, което техният клас не им е дал, е една от тях

най-централните определящи принципи на дизайна на Ruby като език. Както можеш

предположение, ще се връщаме често към него в различни контексти. Засега просто бъдете

съзнавайки, че въпреки че всеки обект има клас, класът на обект не е единственото определящо

минтант на това, което обектът може да направи.

Въоръжен с известна грамотност на Ruby (и някои материали, на които да се обърнете, когато сте в

съмнение), нека да преминем през стъпките, включени в стартирането на програма.

***1.1.4***

***Писане и запазване на проста програма***

На този етап можете да започнете да създавате програмни файлове в примерната кодова директория на Ruby

създадохте малко назад. Вашата първа програма ще бъде темп по Целзий до Фаренхайт

преобразувател на температура.

|  |
| --- |
| **Страница 11** |

**11.**

ЗАБЕЛЕЖКА Преобразувател на температура в реалния свят, разбира се, би използвал плаващ

точки точки. Ние се придържаме към цели числа във входа и изхода, за да запазим

съсредоточете се върху въпросите на структурата и изпълнението на програмата.

Ще обработим този пример няколко пъти, като го добавяме и модифицираме, докато вървим.

Последващите повторения ще

Подредете изхода на програмата

Приемете въвеждането от потребителя чрез клавиатурата

Прочетете стойност от файл

Запишете резултата от програмата във файл

Първата версия е проста; фокусът е върху създаването на файлове и изпълняваната програма

цеси, а не всяка сложна програмна логика.

C ПОВТОРЯНЕ НА ПЪРВИ ПРОГРАМЕН ФАЙЛ

С помощта на редактор на обикновен текст въведете кода от следния списък в текстов файл и

запазете го под името на файла c2f.rb във вашата примерна кодова директория.

Целзий = 100

по Фаренхайт = (по Целзий \* 9/5) + 32

поставя "Резултатът е:"

поставя фаренхайт

поставя "."

ЗАБЕЛЕЖКА В зависимост от вашата операционна система, може да можете да стартирате Ruby

програмни файлове самостоятелни - т.е. само с името на файла или с кратко име

(като c2f) и без разширение на файл. Имайте предвид обаче, че .rb името на файла

удължаването е задължително в някои случаи, основно включващи програми, които заемат

повече от един файл (за който ще научите подробно по-късно) и който се нуждае от

механизъм за файлове за взаимно намиране. В тази книга цялата програма Ruby

имената на файлове завършват на .rb, за да гарантират, че примерите работят на толкова много платформи,

и с възможно най-малко административни отклонения.

Вече имате пълна (макар и малка) програма Ruby на диска си и можете да я стартирате.

***1.1.5***

***Хранене на програмата на Руби***

Стартирането на програма Ruby включва предаване на изходния файл (или файлове) на програмата на

Интерпретатор на рубин, който се нарича рубин. Ще направиш това сега ... нещо като. Ще нахраните про-

грам в рубин, но вместо да помолите Руби да стартира програмата, ще поискате да провери

програмен код за синтаксични грешки.

C ПРОВЕРКА ЗА ГРЕШКИ В СИНТАКС

Ако добавите 31 вместо 32 във вашата формула за преобразуване, това е грешка в програмирането.

Руби все още ще изпълнява с радост вашата програма и ще ви даде недостатъчния резултат. Но ако вие

Листинг 1.1 Прост конвертор Целзий-Фаренхайт с ограничена цел (c2f.rb)

|  |
| --- |
| **Страница 12** |

**12**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

случайно оставете затварящата скоба във втория ред на програмата, това е

синтаксична грешка и Ruby няма да стартира програмата:

$ рубин разбит\_c2f.rb

broken\_c2f.rb: 5: синтаксична грешка, неочакван край на въвеждане, очаква се ')'

(Грешката се отчита на ред 5 - последният ред на програмата - защото Руби чака

търпеливо, за да видите дали някога ще затворите скобата, преди да завършите

че не сте.)

Удобно е, че интерпретаторът Ruby може да проверява програмите за синтаксични грешки без

стартиране на програмите. Той чете файла и ви казва дали синтаксисът е

добре. За да изпълните проверка на синтаксиса във вашия файл, направете следното:

$ рубин -cw c2f.rb

Флагът на командния ред на -cw е съкращение за два флага: -c и -w. Флагът -c означава

*проверете за синтаксични грешки* . Флагът -w активира по-високо ниво на предупреждение: Руби ще се суети

вие, ако сте правили неща, които са законни Руби, но са под въпрос на други основания

отколкото синтаксис.

Ако приемем, че сте написали файла правилно, трябва да видите съобщението

Синтаксис ОК

отпечатани на вашия екран.

R ОТКЛЮЧВАНЕ НА ПРОГРАМАТА

За да стартирате програмата, предайте файла още веднъж на интерпретатора, но този път без

комбинираните флагове -c и -w:

$ рубин c2f.rb

Ако всичко върви добре, ще видите резултата от изчислението:

Резултатът е

212

.

Резултатът от изчислението е правилен, но изходът, разпределен в три реда, изглежда зле.

S ТОКИ CONVERTER итерация

Проблемът може да бъде проследен до разликата между командата put и

команда за печат. put добавя нов ред в края на низа, който отпечатва, ако низът

вече не свършва с един. print, от друга страна, отпечатва низа, който го поискате

до и след това спира; той не преминава автоматично към следващия ред.

За да разрешите проблема, променете първите две команди за отпечатване:

print "Резултатът е"

печат по Фаренхайт

поставя "."

(Обърнете внимание на празното пространство след is, което гарантира, че се появява интервал между is и

номер.) Сега изходът е

Резултатът е 212.

|  |
| --- |
| **Страница 13** |

**13**

*put* е съкращение от *put* (т.е. print) *низ* . Въпреки че *пут* може да не се позовава интуитивно на

идеята за прескачане към следващия ред, това е, което прави: като печат , той печата

на какво му казвате, но след това автоматично преминава към следващия ред. Ако питате поставя

за да отпечатате ред, който вече завършва с нов ред, не се притеснява да добавите такъв.

Ако сте свикнали да печатате съоръжения на езици, които не добавят автоматично нов

ред, като функцията за печат на Perl, може да откриете, че пишете код като този в Ruby

когато искате да отпечатате стойност, последвана от нов ред:

отпечатване по Фаренхайт, "\ n"

Почти никога не се налага да правите това, защото пут добавя нов ред за вас.

Ще вземете навика за путове, заедно с други идиоми и конвенции на Ruby, като вас

върви заедно.

ПРЕДУПРЕЖДЕНИЕ На някои платформи (в частност Windows), допълнителна нова линия

символ се отпечатва в края на изпълнението на програма. Това означава a

печат, който наистина трябва да бъде путове, ще бъде труден за откриване, защото ще действа

като путове. Да сте наясно с разликата между двете и да използвате едната

което искате въз основа на обичайното поведение трябва да е достатъчно, за да сте сигурни, че получавате

желаните резултати.

След като разгледахме малко изхода на екрана, нека разширим малко полето за вход / изход, за да включим ключ-

операции с въвеждане на файлове и файлове.

***1.1.6***

***Клавиатура и файл I / O***

Ruby предлага много техники за четене на данни по време на изпълнението на програмата

както от клавиатурата, така и от дисковите файлове. Ще намерите приложения за тях - ако не в

курса на писане на всяко приложение, след това почти сигурно докато пишете кода на Ruby

за поддържане, преобразуване, домакинство или манипулиране по друг начин на средата, в която

ти работиш. Тук ще разгледаме някои от тези техники за въвеждане; разширен поглед към I / O

операции можете да намерите в глава 12.

K ВХОД ЗА КЛАВИАТУРА

Програма, която ви казва отново и отново, че 100 ° по Целзий се равнява на 212 ° по Фаренхайт

има ограничена стойност. По-ценна програма ви позволява да зададете температура на Целзий и

ви казва еквивалент по Фаренхайт.

Модифицирането на програмата, за да позволи тази функционалност, включва добавяне на няколко

стъпки и използвайки по един метод от таблици 1.1 и 1.2: получава (вземете ред от клавиатурата

вход) и to\_i (преобразуване в цяло число), с едно от които вече сте запознати.

Тъй като това е нова програма, а не само корекция, поставете версията от следното

вписване в нов файл: c2fi.rb ( *i* означава интерактивен).

print "Здравейте. Моля, въведете стойност по Целзий:"

celsius = получава

по Фаренхайт = (celsius.to\_i \* 9/5) + 32

Листинг 1.2 Интерактивен температурен преобразувател (c2fi.rb)

|  |
| --- |
| **Страница 14** |

**14.**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

print "Фаренгейтският еквивалент е"

печат по Фаренхайт

поставя "."

Няколко примерни изпълнения демонстрират новата програма в действие:

$ рубин c2fi.rb

Здравейте. Моля, въведете стойност по Целзий: 100

Еквивалентът по Фаренхайт е 212.

$ рубин c2fi.rb

Здравейте. Моля, въведете стойност по Целзий: 23

Еквивалентът по Фаренхайт е 73.

Сега имаме обобщено, ако не и ужасно нюансирано решение на проблема с

вертинг от Целзий към Фаренхайт. Нека разширим кръга, за да включим въвеждане на файл.

R ЧЕТЕНЕ ОТ ФАЙЛ

Четенето на файл от програма Ruby не е много по-трудно, поне в много случаи,

отколкото четене на ред за въвеждане от клавиатурата. Следващата версия на нашия температурен преобразувател

ще прочете едно число от файл и ще го преобразува от Целзий във Фаренхайт.

Първо, създайте нов файл, наречен temp.dat (температурни данни), съдържащ един ред

с едно число върху него:

100

Сега се създаде една трета файл програма, наречена c2fin.rb ( *в* продължение на [файл] вход), както е показано на

следващ списък.

поставя „Четене на температурна стойност по Целзий от файл с данни ...“

num = File.read ("temp.dat")

Целзий = num.to\_i

по Фаренхайт = (по Целзий \* 9/5) + 32

поставя "Числото е" + число

print "Резултат:"

поставя фаренхайт

Съкращаване на кода

Можете да съкратите кода в списък 1.2 значително, като консолидирате операциите

на входа, изчислението и изхода. Компресирано пренаписване изглежда така:

print "Здравейте. Моля, въведете стойност по Целзий:"

print "Фаренгейтският еквивалент е", gets.to\_i \* 9/5 + 32, ". \ n"

Тази версия пести от променливи - няма такива, но изисква някой да чете

да го следвате по-плътен (но по-кратък!) набор от изрази. Всяка дадена про-

gram обикновено има няколко или много места, където трябва да избирате между по-дълго

(но може би по-ясно?) и по-кратко (но може би малко загадъчно). А понякога и по-кратко

може да бъде по-ясна. Всичко това е част от разработването на стил на кодиране на Ruby.

Листинг 1.3 Преобразувател на температура, използващ въвеждане на файл (c2fin.rb)

|  |
| --- |
| **Страница 15** |

**15**

***Анатомия на инсталацията Ruby***

Този път пробното изпълнение и изходът му изглеждат така:

$ рубин c2fin.rb

Четене на температурна стойност по Целзий от файл с данни ...

Числото е 100

Резултат: 212

Естествено, ако промените номера във файла, резултатът ще бъде различен.

Ами записването на резултата от изчислението във файл?

W ПОДХОДЯЩЕ ДО ФАЙЛ

Най-простата операция за писане на файлове е малко по-сложна от най-простата

операция за четене. Както можете да видите от следващия списък, основната допълнителна стъпка кога

пишете във файл е спецификацията на файлов *режим* - в този случай w (за *запис* ). Запази

версия на програмата от този списък на c2fout.rb и го стартирайте.

print "Здравейте. Моля, въведете стойност по Целзий:"

Целзий = gets.to\_i

по Фаренхайт = (по Целзий \* 9/5) + 32

поставя "Запазване на резултата в изходния файл 'temp.out'"

fh = File.new ("temp.out", "w")

fh.puts фаренхайт

fh.close

Извикването на метода fh.puts fahrenheit има ефект на отпечатване на стойността на fahrenheit

към файла, за който fh е манипулатор за запис. Ако проверите файла temp.out, трябва да видите

че съдържа еквивалент по Фаренхайт на каквото и число да сте въвели.

Като упражнение можете да опитате да комбинирате предишните примери в Ruby про-

грам, който чете число от файл и записва преобразуването по Фаренхайт в различен

входящ файл. Междувременно, с някои основни синтаксиси на Ruby, следващата ни спирка ще бъде

изследване на инсталацията Ruby. Това от своя страна ще ви подготви да погледнете как

Ruby управлява разширения и библиотеки.

***1.2***

***Анатомия на инсталацията Ruby***

Инсталирането на Ruby на вашата система означава наличието на няколко дискови директории

Рубиезични библиотеки и файлове за поддръжка. През повечето време Руби знае как да намери

от какво се нуждае, без да бъдете подканени. Но знаейки как се справяте с инсталирането на Ruby

лацията е част от доброто заземяване на Ruby.

Листинг 1.4 Температурен преобразувател с изход на файл (c2fout.rb)

Разглеждайки изходния код на Ruby

В допълнение към дървото на директориите за инсталиране на Ruby, може да имате и източника на Ruby

кодово дърво на вашата машина; ако не, можете да го изтеглите от началната страница на Ruby. The

Дървото на изходния код съдържа много Ruby файлове, които се озовават в евентуалната инсталация

и много C-езикови файлове, които се компилират в обектни файлове, които след това се инсталират.

Освен това дървото на източника съдържа информационни файлове като ChangeLog и soft-

лицензи за стоки.

|  |
| --- |
| **Страница 16** |

**16.**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

Ruby може да ви каже къде се намират инсталационните му файлове. За да получите информацията докато

в irb сесия трябва да заредите предварително Ruby библиотечен пакет, наречен rbconfig

вашата irb сесия. rbconfig е интерфейс към много компилирана информация за конфигурация

за вашата Ruby инсталация и можете да получите irb, за да го заредите, като използвате irb -r

флаг на командния ред и името на пакета:

$ irb --simple-prompt -rrbconfig

Сега можете да поискате информация. Например можете да разберете къде е Ruby exe-

инсталирани са файлове за готвене (включително ruby ​​и irb):

>> RbConfig :: CONFIG ["bindir"]

RbConfig :: CONFIG е *константа,* отнасяща се до *хеш* (вид структура от данни), където

Ruby запазва своите знания за конфигурацията. Низът "bindir" е хеш *ключ* . Запитване

хешът с клавиша "bindir" ви дава съответната хеш *стойност* , която е

име на директорията за инсталиране на двоичен файл.

Останалата информация за конфигурацията се предоставя по същия начин: като вал

ues в структурата на конфигурационните данни, до която можете да получите достъп със специфичен хеш

ключове. За да получите допълнителна информация за инсталиране, трябва да замените bindir в

команда irb с други термини. Но всеки път, когато използвате една и съща основна формула:

RbConfig :: CONFIG ["*термин*"]. Таблица 1.5 описва термините и директориите, които те имат

Препоръчай на.

Ето кратко описание на основните директории за инсталиране и какво съдържат те. Вие

не трябва да ги запомняте, но трябва да сте наясно как да ги намерите, ако сте

имат нужда от тях (или ако ви е интересно да ги разгледате и да разгледате някои примери

на кода на Ruby!).

Таблица 1.5 Ключови директории на Ruby и техните условия **RbConfig**

Срок

Съдържание на директорията

рубилибдир

Стандартна библиотека на Ruby

биндир

Инструменти за команден ред Ruby

архидър

Архитектурно специфични разширения и библиотеки (компилирани, двоични файлове)

sitedir

Ваши собствени разширения и библиотеки на трети страни (написани на Ruby)

вендордир

Разширения и библиотеки на трети страни (написани на Ruby)

sitelibdir

Вашите собствени разширения за език на Ruby (написани на Ruby)

sitearchdir

Вашите собствени разширения за език Ruby (написани на C)

|  |
| --- |
| **Страница 17** |

**17**

***Анатомия на инсталацията Ruby***

***1.2.1***

***Поддиректорията на стандартната библиотека на Ruby***

***(RbConfig :: CONFIG [rubylibdir])***

В rubylibdir ще намерите програмни файлове, написани на Ruby. Тези файлове предоставят стандарт

библиотечни съоръжения, които можете да изискате от собствените си програми, ако имате нужда от функцията

националност, която осигуряват.

Ето извадка от файловете, които ще намерите в тази директория:

*cgi.rb - Инструменти* за улесняване на програмирането на CGI

*fileutils.rb -* Помощни програми за лесно манипулиране на файлове от програми Ruby

*tempfile.rb* — Механизъм за автоматизиране на създаването на временни файлове

*drb.rb —Съоръжение* за разпределено програмиране с Ruby

Някои от стандартните библиотеки, като например библиотеката drb (последният елемент от предишния

списък), обхваща повече от един файл; ще видите както файл drb.rb, така и цял поддирекция на drb-

тори, съдържащи компоненти на библиотеката drb.

Преглеждането на вашата директория rubylibdir ще ви даде добро (ако може би първоначално

усещане за многото задачи, за които Ruby предоставя съоръжения за програмиране.

Повечето програмисти използват само подмножество от тези възможности, но дори и подмножество на такива

голямата колекция от библиотеки за програмиране ви дава много работа.

***1.2.2***

***Директорията с разширения C (RbConfig :: CONFIG [archdir])***

Обикновено разположен на едно ниво надолу от rubylibdir, archdir съдържа специфична за архитектурата

разширения и библиотеки. Файловете в тази директория обикновено имат имена, завършващи на .so,

.dll или .bundle (в зависимост от вашия хардуер и операционна система). Тези файлове са

C разширения: двоични файлове с възможност за зареждане, генерирани от разширения на C-езика на Ruby

sion код, компилиран в двоична форма като част от процеса на инсталиране на Ruby.

Подобно на програмните файлове на езика Ruby в rubylibdir, файловете в archdir съдържат

стандартни библиотечни компоненти, които можете да заредите в собствените си програми. (Между

други, ще видите файла за разширението rbconfig - разширението, с което използвате

irb, за да разкрие имената на директориите.) Тези файлове не се четат от човека, но Ruby

преводач знае как да ги зареди, когато бъде помолен да го направи. От гледна точка на

Ruby програмист, всички стандартни библиотеки са еднакво използваеми, независимо дали са написани на Ruby

или написани на C и компилирани в двоичен формат.

Файловете, инсталирани в archdir, варират в зависимост от инсталацията

кои разширения са компилирани - което от своя страна зависи от комбинация от това, което

човек, който прави компилацията, поиска и кои разширения Ruby успя да компилира.

***1.2.3***

***Site\_ruby (RbConfig :: CONFIG [sitedir]) и vendor\_ruby***

***(RbConfig :: CONFIG [vendordir]) директории***

Вашата Ruby инсталация включва поддиректория, наречена site\_ruby, където вие и / или

вашият системен администратор съхранява разширения и библиотеки на трети страни. Някои от тези

може да е код, който пишете, докато други са инструменти, които изтегляте от сайтовете на други хора

и архиви на Ruby библиотеки.

|  |
| --- |
| **Страница 18** |

**18.**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

Директорията site\_ruby е успоредна на основната инсталационна директория на Ruby в

усещането, че има свои собствени поддиректории за разширения на Ruby-език и C-език

(sitelibdir и sitearchdir, съответно, в условията RbConfig :: CONFIG). Когато имате нужда

разширение, интерпретаторът на Ruby го проверява в тези поддиректории на site\_ruby, като

както и в основния рубилибдир, така и в главния архдир.

Заедно с site\_ruby ще намерите директорията vendor\_ruby. Разширяване на някои трети страни

sions се инсталират тук. Директорията vendor\_ruby е нова от Ruby 1.9 и

стандартната практика за това коя от двете области получава кои пакети все още се развива.

***1.2.4***

***Директорията на скъпоценните камъни***

Помощната програма RubyGems е стандартният начин за пакетиране и разпространение на Ruby библиотеки.

Когато инсталирате *скъпоценни камъни* (както се наричат ​​пакетите), отделените библиотечни файлове попадат в

директорията на скъпоценните камъни. Тази директория не е изброена в структурата на конфигурационните данни, но е използвана

съюзник на същото ниво като site\_ruby; ако сте намерили site\_ruby, вижте какво друго е

инсталиран до него. Ще научите повече за използването на скъпоценни камъни в раздел 1.4.5.

Нека да разгледаме сега механиката и семантиката на това как Ruby използва собственото си разширение

както и тези, които можете да напишете или инсталирате.

***1.3***

***Ruby разширения и библиотеки за програмиране***

Първият основен момент, който трябва да вземете под внимание, докато четете този раздел, е, че това не е Ruby

стандартна библиотечна справка. Както е обяснено във въведението, тази книга не цели

документирайте езика Ruby; тя има за цел да ви научи езика и да предостави на Руби

гражданство, за да можете да продължите да разширявате хоризонтите си.

Целта на този раздел, съответно, е да ви покаже как работят разширенията: как

получавате Ruby да стартира своите разширения, разликата между техниките за това и

архитектурата на разширенията, която ви позволява да пишете свои собствени разширения и библиотеки.

Разширенията, които се доставят с Ruby, обикновено се наричат ​​общо като *стандарт*

*dard библиотека* . Стандартната библиотека включва разширения за голямо разнообразие от проекти и

задачи: управление на бази данни, работа в мрежа, специализирана математика, XML обработка,

и много други. Точният състав на стандартната библиотека обикновено се променя, поне a

малко, с всяко ново издание на Ruby. Но повечето от по-широко използваните библиотеки са склонни

да останат, след като са доказали своята стойност.

Ключът към използването на разширения и библиотеки е методът require, заедно с неговия

натоварване в близост до релацията. Тези методи ви позволяват да зареждате разширения по време на изпълнение, включително

разширения, които пишете сами. Ще ги разгледаме сами по себе си и след това ще разширим

нашият обхват да използваме тяхното зареждане на вградени разширения.

***1.3.1***

***Зареждане на външни файлове и разширения***

Съхранението на програма в един файл може да бъде удобно, но започва да е по-скоро отговорност, отколкото

актив, когато имаш стотици или хиляди или стотици *от* хиляди-на линии

на кода. Някъде по линията започва разбиването на вашата програма на отделни файлове

имат много смисъл. Ruby улеснява този процес с методите за изискване и зареждане.

Ще започнем с натоварване, което е по-простото проектиране от двете.

|  |
| --- |
| **Страница 19** |

**19.**

***Ruby разширения и библиотеки за програмиране***

За да изпробвате примерите, които следват, ще ви е необходима програма, разделена на два файла. The

първият файл, loaddemo.rb, трябва да съдържа следния Ruby код:

поставя "Това е първият (главен) програмен файл."

зареди "loadee.rb"

поставя "И отново към първия файл."

Когато срещне извикване на метода за зареждане, Ruby чете във втория файл. Този файл,

loadee.rb, трябва да изглежда така:

поставя "> Това е вторият файл."

Двата файла трябва да са в една и съща директория (вероятно вашият примерен код дирек-

тори). Когато стартирате loaddemo.rb от командния ред, ще видите този изход:

Това е първият (главен) програмен файл.

> Това е вторият файл.

И обратно към първия файл.

Изходът ви дава следа от кои редове от кои файлове се изпълняват и

в какъв ред.

Извикването за зареждане в loaddemo.rb предоставя име на файл, loadee.rb, като свой аргумент:

зареди "loadee.rb"

Ако файлът, който зареждате, е в текущата ви работна директория, Ruby ще може да го намери

то по име. Ако не е, Ruby ще го потърси в *пътя* на *зареждане* .

***1.3.2***

„***Зареждане“ -ing файл в пътя на зареждане по подразбиране***

Пътят за зареждане на интерпретатора на Ruby е списък с директории, в които той търси файловете ви

помолете го да се зареди. Можете да видите имената на тези директории, като разгледате съдържанието на

специалната глобална променлива $: (долар-двоеточие). Това, което виждате, зависи от каква платформа

Вие сте на. Типична проверка на пътя на натоварване в Mac OS X изглежда по следния начин (an

пример, който включва директорията .rvm, където Ruby Version Manager поддържа

избор на версии на Ruby):

$ ruby ​​-e 'поставя $:'

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/site\_ruby/2.1.0

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/site\_ruby/2.1.0/x86\_64-

darwin12.0

Функция, разширение или библиотека?

Нещата, които зареждате в програмата си по време на изпълнение, се извикват с няколко различни имена.

*Функцията* е най-абстрактна и рядко се чува извън специализираната употреба

„Изискване на функция“ (т.е. с изискване). *Библиотеката* е по-конкретна и по-удобна

мон. Той означава действителния код, както и основния факт, че набор от програмиране

съоръжения съществува и може да се зарежда. *Разширението* може да се отнася до всяка зареждаща се библиотека с добавки

но често се използва за означаване на библиотека за Ruby, която е написана в програма C

език, а не в Ruby. Ако кажете на хората, че сте написали разширение Ruby,

те вероятно ще приемат, че имате предвид, че е в C.

**-е сигнали**

**че сте**

**осигуряване на**

**вграден скрипт**

**да тълкува**

|  |
| --- |
| **Страница 20** |

**20.**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/site\_ruby

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/vendor\_ruby/2.1.0

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/vendor\_ruby/2.1.0/x86\_64-

darwin12.0

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/vendor\_ruby

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/2.1.0

/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/2.1.0/x86\_64-darwin12.0

На вашата машина частта отляво на “ruby-2.1.0” може да казва нещо различно, например

“/ Usr / local / lib /”, но основният модел на поддиректориите ще бъде същият. Кога

зареждате файл, Ruby го търси във всяка от изброените директории, по ред отгоре

до долу.

ЗАБЕЛЕЖКА : Текущата работна директория, обикновено представена с една точка (.),

всъщност не е включен в пътя на натоварване. Командата за зареждане действа така, сякаш е, но

това е специално проектиран случай.

Можете да навигирате в относителни директории в командите за зареждане с конвенционалния

символ с двойна точка „директория нагоре“:

заредете "../extras.rb"

Имайте предвид, че ако промените текущата директория по време на изпълнение на програма, относителна директория

препратките също ще се променят.

ЗАБЕЛЕЖКА Имайте предвид, че натоварването е метод и се изпълнява в точката

където Ruby го среща във вашия файл. Руби не търси целия файл

търси директиви за натоварване; намира ги, когато ги намира. Това означава вие

може да зарежда файлове, чиито имена се определят динамично по време на изпълнението на

програма. Можете дори да обгърнете повикване за зареждане в условен израз, в който

в случай, че повикването ще бъде изпълнено само ако условието е вярно.

Можете също така да принудите зареждането да намери файл, независимо от съдържанието на пътя за зареждане, чрез

давайки му напълно квалифициран път до файла:

заредете "/home/users/dblack/book/code/loadee.rb"

Това, разбира се, е по-малко преносимо от използването на пътя на натоварване или относителните директории,

но може да бъде полезно, особено ако имате абсолютен път, съхраняван като низ в

променлива и искате да заредите файла, който представлява.

Обаждането за зареждане винаги зарежда файла, който искате, независимо дали вече сте го заредили или

не. Ако даден файл се промени между зарежданията, значи всичко в новата версия на файла

че пренаписва или заменя нищо в оригиналната версия, има приоритет. Това може да бъде

полезно, особено ако сте в irb сесия, докато модифицирате файл в редактор в

по едно и също време и искате незабавно да проучите ефекта от промените.

Другият метод за зареждане на файлове, също, търси в директориите, които се намират

пътят на зареждане по подразбиране. Но изискването има някои функции, които товарът няма.

|  |
| --- |
| **Страница 21** |

**21.**

***Ruby разширения и библиотеки за програмиране***

***1.3.3***

„***Изискване“ - функция***

Една основна разлика между натоварването и изискването е, че изискването, ако се извика повече от

веднъж със същите аргументи, не презарежда файлове, които вече са заредени. Руби държи

проследявайте кои файлове сте искали и не дублира усилията.

изискването е по-абстрактно от натоварването. *Строго погледнато, не ви е необходим файл; имате нужда*

*функция.* И обикновено го правите, без дори да посочите разширението на файла-

име. За да видите как работи това, променете този ред в loaddemo.rb,

зареди "loadee.rb"

до това:

изискват "./loadee.rb"

Когато стартирате loaddemo.rb, получавате същия резултат като преди, въпреки че сте

не сте предоставили пълното име на файла, който искате да заредите.

Разглеждайки loadee като „функция“, а не като файл, изискването ви позволява да третирате exten-

sions, написани в Ruby, по същия начин, по който се отнасяте към разширенията, написани на C - или, за да го кажем

друг начин - да третирате файлове, завършващи на .rb, по същия начин като файловете, завършващи на .so, .dll или .bundle

Можете също така да подадете напълно квалифицирана пътека, която да изисквате, както можете да заредите, и тя ще изтегли

във файла / характеристиката. И можете да смесвате и съчетавате; следният синтаксис работи, за изпит-

ple, въпреки че смесва спецификацията на статичния път с по-абстрактния синтаксис на

характеристиката в края на пътя:

изисква "/home/users/dblack/book/code/loadee.rb"

Въпреки че зареждането е полезно, особено когато искате да заредите файл повече от веднъж,

изисква е ежедневната техника, която ще използвате за зареждане на разширения и библиотеки на Ruby -

стандартно и по друг начин. Зареждането на стандартни функции на библиотеката не е по-трудно от това на load-

ing loadee. Просто се нуждаете от това, което искате. След като го направите, и разбира се в зависимост от това

какво е разширението, ще имате на разположение нови класове и методи. Ето a

пример преди и след в irb сесия:

>>"Дейвид Блек" .scanf ("% s% s")

NoMethodError: недефиниран метод `scanf 'за" Дейвид Блек ": String

>> изисква "scanf"

Посочване на работната директория

require не знае за текущата работна директория (.). Можете да го посочите изрично

изискват "./loadee.rb"

или можете да го добавите към пътя за зареждане с помощта на оператора за добавяне на масив,

$: <<"."

така че не е необходимо да го посочвате в повиквания, за да изисквате:

изискват "loadee.rb"

б

° С

|  |
| --- |
| **Страница 22** |

**22.**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

=> вярно

>>"Дейвид Блек" .scanf ("% s% s")

=> [„Дейвид“, „Черен“]

Първата покана за scanf се провали с грешка B . Но след изискването се обадете c , и с no

допълнителна намеса на програмист, низови обекти като "Дейвид Блек" отговарят на

съобщение scanf. (В този пример d искаме да бъдат два последователни низа

извлечен от оригиналния низ, с интервали като подразбиращ се разделител.)

***1.3.4***

***require\_relative***

Има трети начин за зареждане на файлове: require\_relative. Тази команда зарежда функции чрез

търсене по отношение на директорията, в която се намира файлът, от който се нарича. Поради това

в предишния пример можете да направите това

require\_relative "loadee"

без да се манипулира пътя на зареждане, за да се включи текущата директория. require\_relative

е удобно, когато искате да навигирате в йерархия на локална директория - например:

require\_relative "lib / music / sonata"

Ще завършим тази глава с преглед на инструментите на командния ред, които се доставят

с Руби.

***1.4***

***Готови инструменти и приложения на Ruby***

Когато инсталирате Ruby, получавате шепа важни инструменти на командния ред, които

са инсталирани във всяка директория, конфигурирана като bindir - обикновено / usr / local / bin,

/ usr / bin или еквивалентите на / opt. (Можете да изискате "rbconfig" и да разгледате

RbConfig :: CONFIG ["bindir"] за проверка.) Тези инструменти са

рубин - преводачът

irb - Интерактивният интерпретатор на Ruby

rdoc и ri - инструменти за документация на Ruby

rake - Ruby make, помощна програма за управление на задачи

скъпоценен камък - помощна програма за управление на библиотека Ruby и приложения

erb - система за шаблониране

testrb - инструмент на високо ниво за използване с тестовата рамка на Ruby

В този раздел ще разгледаме всички тези инструменти с изключение на erb и testrb. И двамата използват-

пълно в определени ситуации, но не непосредствен приоритет, докато се ориентирате и

заземяване в Ruby.

Не е нужно да запомняте на място всички техники в този раздел. По-скоро,

прочетете го и разберете какво има тук. Скоро ще използвате част от материала

и често (особено някои от превключвателите на командния ред и помощната програма ri) и някои

от него все повече, докато навлизате по-дълбоко в Руби.

д

|  |
| --- |
| **Страница 23** |

**23.**

***Готови инструменти и приложения на Ruby***

***1.4.1***

***Превключватели на командния ред на интерпретатора***

Когато стартирате интерпретатора на Ruby от командния ред, можете да предоставите не

само името на програмен файл, но също така и един или повече превключватели на командния ред, като

вече сте виждали в главата. Превключвателите, които сте избрали, инструктират преводача

да се държи по конкретни начини и / или да предприема конкретни действия.

Ruby има повече от 20 превключватели на командния ред. Някои от тях се използват рядко,

докато други се използват всеки ден от много програмисти на Ruby. Таблица 1.6 обобщава

най-често използваните.

Нека разгледаме всеки от тези превключватели по-подробно.

C ПРОВЕРКА СИНТАКС (- C )

Превключвателят -c казва на Ruby да проверява кода в един или повече файлове за синтактична точност

без да изпълнява кода. Обикновено се използва заедно с флага -w.

Т УРН ЗА ПРЕДУПРЕЖДЕНИЯ (- З )

Стартирането на вашата програма с -w кара интерпретатора да работи в режим на предупреждение. Това

означава, че виждате повече предупреждения, отпечатани на екрана, отколкото в противен случай,

привличане на вниманието ви към места в програмата, които, макар и да не са синтаксични грешки, са

стилистично или логично подозрително. Това е начинът на Руби да каже: „Това, което сте направили, е синтак-

типично правилно, но е странно. Сигурни ли сте, че сте искали да направите това? " Дори и без това

превключвател, Ruby издава определени предупреждения, но по-малко, отколкото в режим на пълно предупреждение.

E ЕКСУКТИРАН ЛИТЕРАЛЕН СЦЕНАРИЙ (- E )

Ключът -e казва на интерпретатора, че командният ред включва кода на Ruby в quo-

и че трябва да изпълнява този действителен код, вместо да изпълнява кода

Таблица 1.6 Обобщение на често използваните превключватели на команден ред Ruby

Превключване

Описание

Пример за употреба

-° С

Проверете синтаксиса на програмен файл с-

изпълнение на програмата

рубин -c c2f.rb

-w

Давайте предупредителни съобщения по време на програмата

екзекуция

рубин -w c2f.rb

-е

Изпълнете кода, предоставен в оферта

марки в командния ред

ruby -e 'поставя „Код демо!“

-л

Режим на линия: отпечатва нов ред след всеки

линия на изхода

ruby -le 'print "+ newline!"'

- *rname*

Изисквайте посочената функция

ruby –rprofile

-v

Показване на информация за версията на Ruby и

изпълнете програмата в подробен режим

рубин –v

--версия

Показване на информация за версията на Ruby

рубин –версия

-х

Показване на информация за всички команди-

линейни превключватели за преводача

рубин –ч

|  |
| --- |
| **Страница 24** |

**24**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

съдържащи се във файл. Това може да бъде удобно за бързи задачи за скриптове, когато влизате в

код във файл и пускането на рубин във файла може да не си струва труда.

Да предположим например, че искате да видите името си отпечатано назад. Ето как

можете да направите това бързо с една команда от командния ред, като използвате ключа за изпълнение:

$ ruby ​​-e 'поставя "Дейвид А. Блек" .reverse'

kcalB .ДиваD

Това, което се крие вътре в единичните кавички, е цял (макар и кратък) Ruby про-

грам. Ако искате да подадете програма с повече от един ред към превключвателя -e, можете

използвайте буквални прекъсвания на редове (натиснете Enter) в мини-програмата:

$ ruby ​​-e 'print "Въведете име:"

поставя gets.reverse '

Въведете име: Дейвид А. Блек

kcalB .ДиваD

Или можете да разделите линиите с точка и запетая:

$ ruby ​​-e 'print "Въведете име:"; print gets.reverse '

ЗАБЕЛЕЖКА Защо има празен ред между програмния код и изхода в

двуредов обратен пример? Защото редът, който въвеждате на клавиатурата

завършва с символ на нов ред, така че когато обърнете въвеждането, новият низ

започва с нов ред! Руби ви приема много буквално, когато го помолите да манипулира-

късно и отпечатайте данни.

R UN В РЕЖИМ НА ЛИНИЯ (- L )

Превключвателят -l създава ефекта, върху който се поставя всеки изведен от програмата низ

собствена линия, дори ако обикновено не би била. Обикновено това означава, че линии, които са

изход, използвайки print, вместо да поставя, и това следователно не завършва автоматично с

знак за нов ред, сега завършете с нов ред.

Използвахме отпечатъка спрямо разликата, за да гарантираме, че температурата

конверторните програми не вмъкнаха допълнителни нови редове в средата на изхода си (вижте раздел

1.1.5). Можете да използвате превключвателя -l, за да обърнете ефекта; причинява дори отпечатани редове

да се появят на собствена линия. Ето разликата:

$ рубин c2f-2.rb

Резултатът е 212.

$ рубин -l c2f-2.rb

Резултатът е

212

.

Резултатът с -l е в този случай точно това, което не искате. Но примерният илюзор-

отчита ефекта на превключвателя.

*Ако ред вече завършва с символ на нов ред, прекарването му през -l добавя друг нов ред* .

Като цяло ключът -l не се използва често или не се вижда, най-вече поради наличността

ity на путове, за да постигнете поведението „добавяне на нов ред само ако е необходимо“, но е добре да го направите

знам -l е там и да мога да го разпозная.

|  |
| --- |
| **Страница 25** |

**25**

***Готови инструменти и приложения на Ruby***

R EQUIRE ИМЕНЕН ФАЙЛ ИЛИ РАЗШИРЕНИЕ (- RNAME )

Извикванията -r switch изискват аргумента му; ruby -rscanf ще изисква scanf, когато

стартира преводач. Можете да поставите повече от един -r превключвател на един команден ред.

R ООН В РЕЖИМ НА ГЛЕБОКА (- V , - ВЕРБОЗА )

Изпълнението с -v прави две неща: отпечатва информация за версията на Ruby

използвате и след това се включва същият предупредителен механизъм като флага -w. The

най-честата употреба на -v е да откриете номера на версията на Ruby:

$ рубин -v

рубин 2.1.0p0 (2013-12-25 ревизия 44422) [x86\_64-darwin12.0]

В този случай използваме Ruby 2.1.0 (patchlevel 0), издаден на 25 декември 2013 г.,

и компилиран за i686 базирани машина работи на Mac OS X . Защото няма про-

грам или код за изпълнение, Ruby излиза веднага след като е отпечатала информацията за версията.

P RINT R UBY ВЕРСИЯ (- ВЕРСИЯ )

Този флаг кара Ruby да отпечата низ с информация за версията и след това да излезе. Не става

изпълнете всеки код, дори ако предоставите код или име на файл. Ще си спомните, че отпечатва -v

информация за версията и след това стартира вашия код (ако има такъв) в подробен режим. Може да се каже

че -v хитро се застъпва както за *версия, така* и за *многословен* , докато --version е просто *версия* .

P ечат помощ ИНФОРМАЦИЯ (- Н , - HELP )

Тези суичове ви дават таблица, в която са изброени всички комутатори на командния ред, налични за вас,

и обобщаване на това, което правят.

В допълнение към използването на единични превключватели, можете също да комбинирате два или повече в един

призоваване на Руби.

C КОМБИНИРАТЕЛНИ КЛЮЧОВЕ (- CW )

Вече сте виждали комбинацията -cw, която проверява синтаксиса на файла без

изпълнявайки го, като същевременно ви дава предупреждения:

$ ruby ​​-cw име на файл

Друга комбинация от превключватели, които често ще виждате, е -v и -e, която ви показва

версия на Ruby, която използвате, и след това изпълнява кода, предоставен в кавички.

Ще видите тази комбинация много в дискусиите на Ruby, в пощенските списъци и други-

където; хората го използват, за да демонстрират как един и същ код може да работи по различен начин

предстоящи версии на Ruby. Например, ако искате да покажете ясно, че низ метод

наречен start\_with? не присъства в Ruby 1.8.6, но присъства в Ruby 2.1.0, можете

стартирайте примерна програма, използвайки първо едната версия на Ruby, а след това другата:

$ ruby-1.8.6-p399 -ve "поставя 'abc'.start\_with? (' a ')"

ruby 1.8.6 (2010-02-05 patchlevel 399) [x86\_64-linux]

-e: 1: недефиниран метод `start\_with? ' за "abc": String (NoMethodError)

$ ruby-2.1.0p0 -ve "поставя 'abc'.start\_with? (' a ')"

ruby 2.1.0p0 (2013-12-25 ревизия 44422) [x86\_64-linux]

вярно

(Разбира се, трябва да имате и двете версии на Ruby инсталирани на вашата система.) Недефинираното

метод 'start\_with?' означава съобщение Б при първото изпълнение (това, което използва версия 1.8.6)

б

° С

|  |
| --- |
| **Страница 26** |

**26**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

че сте се опитали да извършите несъществуваща операция с име. Но когато стартирате

същия Ruby фрагмент, използващ Ruby 2.1.0, работи c : Ruby отпечатва вярно. Това е удобно

начин за споделяне на информация и формулиране на въпроси относно промените в поведението на Руби

от едно издание до друго.

На този етап ще се върнем назад и ще разгледаме по-отблизо интерактивния интерфейс на Ruby

preter, irb. Може би вече сте разглеждали този раздел, когато той беше споменат близо

началото на главата. Ако не, можете да се възползвате от тази възможност, за да научите повече

за този изключително полезен инструмент Ruby.

***1.4.2***

***По-подробен поглед към интерактивната интерпретация на Ruby с irb***

Както видяхте, irb е интерактивен интерпретатор на Ruby, което означава, че вместо да

прекратявайки файл, той обработва това, което въвеждате по време на сесия. irb е чудесно средство за тест-

ing Ruby код и чудесен инструмент за изучаване на Ruby.

За да започнете сесия на irb, използвате командата irb. irb отпечатва своя ред:

$ irb

2.1.0: 001>

Както видяхте, можете също да използвате опцията --simple-prompt, за да запазите изхода на irb

по-късо:

$ irb --simple-prompt

>>

След като irb стартира, можете да въведете команди на Ruby. Можете дори да стартирате версия с един изстрел

на програмата за преобразуване Целзий във Фаренхайт. Както ще видите в този пример, irb

се държи като джобен калкулатор: оценява каквото въведете и отпечатва резултата.

Не е нужно да използвате команда за печат или поставяне:

>> 100 \* 9/5 + 32

=> 212

За да разберете колко минути са там за една година (ако не разполагате с CD на вни-

vant хитова песен от мюзикъла *Rent* handy), въведете съответното умножение

израз:

>> 365 \* 24 \* 60

=> 525600

Посочване на превключватели

Можете да захранвате превключвателите Ruby отделно, по този начин

$ рубин -c -w

или

$ ruby ​​-v -e "поставя 'abc'.start\_with? (' a ')"

Но е обичайно да ги пишете заедно, както в примерите в основния текст.

|  |
| --- |
| **Страница 27** |

**27**

***Готови инструменти и приложения на Ruby***

irb също, разбира се, ще обработва всички въведени от вас инструкции за Ruby. Например, ако вие

искате да присвоите броя на деня, часа и минутите на променливите и след това да ги умножите

променливи, можете да направите това в irb:

>> дни = 365

=> 365

>> часа = 24

=> 24

>> минути = 60

=> 60

>> дни \* часове \* минути

=> 525600

Последното изчисление е това, което бихте очаквали. Но вижте първите три реда на влизане.

Когато пишете дни = 365, irb реагира, като отпечата 365. Защо?

Изразът days = 365 е израз на присвояване: вие присвоявате стойността

365 до променлива, наречена дни. Основният бизнес на израза за присвояване е

задайте стойност на променлива, за да можете да използвате променливата по-късно. Но заданието

израз (целият ред дни = 365) има стойност. Стойността на заданието е изразена

sion е дясната му страна. Когато irb види някакъв израз, той отпечатва стойността на това

израз. Така че, когато irb вижда дни = 365, той отпечатва 365. Това може да изглежда като прекалено

ревностен печат, но идва с територията; това е същото поведение, което ви позволява

въведете 2 + 2 в irb и вижте резултата, без да се налага да използвате изрично изявление за печат.

По същия начин дори извикването на метода puts има върната стойност - а именно nil. Ако ти

напишете оператор put в irb, irb послушно ще го изпълни и ще отпечата

върната стойност на путове:

$ irb --simple-prompt

>> поставя "Здравей"

Здравейте

=> нула

Има начин да получите irb, за да не бъдете толкова приказлив: флагът --noecho. Ето как

работи:

$ irb --simple-prompt --noecho

>> 2 + 2

>> поставя "Здравей"

Здравей

Благодарение на --noecho изразът за добавяне не отчита резултата си. Путовете

командата се изпълнява (така че виждате "Здравей"), но връщаната стойност на put (nil)

се потиска.

Прекъсване на irb

Възможно е да се забиете в цикъл в irb или сесията да се почувства сякаш не е така

отговаря (което често означава, че сте въвели начална кавичка, но не и

затваряне на едно или нещо в този ред). Начинът на връщане на контрола е донякъде

зависим от системата. В повечето системи Ctrl-C ще свърши работа. На други може да се наложи

|  |
| --- |
| **Страница 28** |

**28**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

След като разберете подхода на irb за отпечатване на стойността на всичко, и

как да го затворите, ако искате, ще го намерите изключително полезен инструмент (и играчка).

Изходният код на Ruby е маркиран по такъв начин, че да осигурява автоматично генерация

на документация; и инструментите, необходими за интерпретиране и показване на този документ-

тацията са ri и RD oc, които ще разгледаме сега.

***1.4.3***

***ri и RDoc***

ri (Ruby Index) и RD oc (Ruby Documentation), първоначално написано от Дейв Томас,

са тясно свързана двойка инструменти за предоставяне на документация за програмите Ruby.

ri е инструмент за команден ред; системата RD oc включва инструмента за команден ред rdoc. ri

и rdoc са самостоятелни програми; стартирате ги от командния ред. (Можеш

използвайте и съоръженията, които те предоставят във вашите Ruby програми, въпреки че сме

няма да разглеждам този аспект от тях тук.)

RD oc е система за документиране. Ако поставите коментари в програмните си файлове (Ruby

или В) в предписания формат RD oc, rdoc сканира вашите файлове, извлича коментарите,

организира ги интелигентно (индексирани според това, което коментират) и създава

събира добре форматирана документация от тях. Можете да видите RD oc маркиране в много

на изходните файлове, както в Ruby, така и в C, в дървото на източника на Ruby и в много от

Ruby файлове в инсталацията Ruby.

ri свързва с RD oc: дава ви начин да видите информацията, която RD oc има

извлечени и организирани. По-конкретно (макар и не изключително, ако го персонализирате),

ri е конфигуриран да показва RD oc информация от изходните файлове на Ruby. Така нататък

всяка система, която има напълно инсталиран Ruby, можете да получите подробна информация за Ruby

с просто извикване на команден ред на ri.

Например, ето как заявявате информация за метода upcase на

низ обекти:

$ ri String # upcase

И ето какво получавате обратно:

= String # upcase

(от рубинено ядро)

-------------------------------------------------- ----------------------------

str.upcase -> new\_str

***(продължение)***

за да използвате Ctrl-Z. Най-добре е да приложите каквато и да е обща информация за прекъсване на програмата

има за вашата система директно към irb. Разбира се, ако нещата наистина замръзнат, можете

отидете на вашия процес или инструменти за управление на задачи и убийте процеса на irb.

За да излезете от irb нормално, можете да напишете exit. На много системи работи и Ctrl-D.

Понякога irb може да ви взриви (т.е. да удари фатална грешка и да се прекрати).

По-голямата част от времето обаче улавя собствените си грешки и ви позволява да продължите.

|  |
| --- |
| **Страница 29** |

**29**

***Готови инструменти и приложения на Ruby***

-------------------------------------------------- ----------------------------

Връща копие на str с всички малки букви, заменени с техните

главни аналози. Операцията е нечувствителна към локала --- само символи

„a“ до „z“ са засегнати. Забележка: заместването на калъфа е ефективно само в

Регион ASCII.

"hEllO" .upcase # =>"ЗДРАВЕ"

Хеш-знакът (#) между String и upcase в командата ri показва това

търсите метод на екземпляр, различен от метод на клас. В случай че

метод на клас, бихте използвали separator :: вместо #. Ще стигнем до метода на класа /

разграничение на метода на екземпляр в глава 3. Основното за момента е, че вие

имате много документация на ваше разположение от командния ред.

СЪВЕТ По подразбиране ri изпълнява изхода си чрез пейджър (като например повече за Unix). То

може да направи пауза в края на изхода, чакайки да натиснете интервала или някоя друга

друг клавиш за показване на следващия екран с информация или за изход изцяло, ако всички

е показана информация. Точно това, което трябва да натиснете в този случай, варира

от една операционна система и един пейджър на друг. Интервал, Enter,

Escape, Ctrl-C, Ctrl-D и Ctrl-Z са добри залози. Ако искате ri да напишете

изход, без да го филтрирате през пейджър, можете да използвате командния ред –T

превключвател (ri –T тема).

Следващото сред инструментите на командния ред на Ruby е рейк.

***1.4.4***

***Помощната програма за управление на задачи на рейк***

Както подсказва името му (идва от „Ruby make“), рейкът е задача, вдъхновена от марката -

помощна програма за управление. Написана е от покойния Джим Уейрих. Като марка, рейк чете и

изпълнява задачи, дефинирани във файл - Rakefile. За разлика от make обаче, рейкът използва Ruby syn-

данък, за да определи своите задачи.

Листинг 1.5 показва Rakefile. Ако запишете списъка като файл, наречен Rakefile, можете

след това издайте тази команда в командния ред:

$ rake admin: clean\_tmp

rake изпълнява задачата clean\_tmp, дефинирана в администраторското пространство на имената.

пространство от имена: администратор прави

desc "Интерактивно изтриване на всички файлове в / tmp"

задача: clean\_tmp do

Dir ["/ tmp / \*"]. Всеки прави | f |

следваща освен File.file? (f)

print "Изтриване # {f}?"

отговор = $ stdin.gets

отговор на случая

когато / ^ y /

File.unlink (f)

когато / ^ q /

почивка

Листинг 1.5 Rakefile, дефиниращ задачи **clean\_tmp** в **администраторското** пространство на имената

**Декларира**

**задача clean\_tmp**

б

° С

д

д

е

|  |
| --- |
| **Страница 30** |

**30**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

край

край

край

край

Задачата за рейк, дефинирана тук, използва няколко техники на Ruby, които все още не сте виждали,

но основният алгоритъм е доста прост:

1

Loop през всяко влизане директория в / TMP директория на B .

2

Пропуснете текущата итерация на цикъла, освен ако този запис не е файл. Имайте предвид, че скритите файлове

също не се изтриват, тъй като операцията с изброяване на директории не включва

тях c .

3

Подкана за изтриване на файла d .

4

Ако потребителят напише y (или нещо, което започва с y), изтрийте файла e .

5

Ако потребителят напише q, прекъснете цикъла; задачата спира f .

Основната логика на програмиране идва от цикъла в списъка с директории

записи (вижте страничната лента „Използване на всеки, за да прегледате колекция“) и от случая

израз, условна структура за изпълнение. (Ще видите и двете техники в

подробности по-нататък в глава 6.)

Командата desc над дефиницията на задачата предоставя описание на задачата. Това

е полезен не само когато преглеждате файла, но и ако искате да видите всички

задачите, които рейкът може да изпълни в даден момент. Ако сте в директорията, съдържаща

Rakefile в списък 1.5 и вие давате командата

$ рейк --задачи

ще видите списък на всички дефинирани задачи:

$ рейк --задачи

(в / Потребители / рубин / хакерство)

rake admin: clean\_tmp # Интерактивно изтрийте всички файлове в / tmp

Можете да използвате всякакви имена, които искате, за вашите пространства и задачи за рейк. Дори не го правите

нужда от пространство от имена; можете да дефинирате задача в пространството от имена на най-високо ниво,

задача: clean\_tmp do

# и т.н.

край

Използването на **всеки, за** да прегледате колекция

Изразът Dir ["/ tmp / \*"]. Всеки do | f | е извикване на всеки метод на

масив от всички имена на записи в директорията. Целият блок код, започващ с do и

завършващ с end (краят, който се подрежда с Dir в отстъп) се изпълнява веднъж

за всеки елемент в масива. Всеки път текущият елемент е обвързан с параметъра

етер f; това е значението на | f | част. Ще видите всеки няколко пъти в

следващите глави и ще го разгледаме подробно, когато разгледаме *итератори* (методи

които автоматично обхождат колекции) в глава 9.

|  |
| --- |
| **Страница 31** |

**31**

***Готови инструменти и приложения на Ruby***

и след това го извикайте с простото име:

$ рейк clean\_tmp

Но задаването на имена на вашите задачи е добра идея, особено ако и кога броят на

задачите, които определяте, нараства значително. Можете да разполагате с имена на всякаква дълбочина; тази структура,

например, е легитимно:

пространство от имена: администратор прави

пространство от имена: clean do

задача: tmp do

# и т.н.

край

край

край

Дефинираната тук задача се извиква по следния начин:

$ rake admin: clean: tmp

Както показва примерът за почистване на директории, задачите за рейк не трябва да се ограничават

действия, свързани с програмирането на Ruby. С рейк получавате целия език на Ruby

на ваше разположение, за да напишете каквито и да е задачи, от които се нуждаете.

Следващият инструмент в обиколката е командата за скъпоценен камък, която прави инсталиране на трето-

парти руби пакети много лесно.

***1.4.5***

***Инсталиране на пакети с командата gem***

Библиотеката и помощната колекция на RubyGems включва съоръжения за опаковане и

инсталиране на Ruby библиотеки и приложения. Тук няма да отразяваме създаването на скъпоценни камъни,

но ще разгледаме инсталирането и използването на скъпоценни камъни.

Инсталирането на Ruby скъпоценен камък може да бъде и обикновено е толкова лесно, колкото издаването на проста инсталация

команда:

$ gem инсталирате скарида

Такава команда ви дава изход нещо като следното (в зависимост от това кое

скъпоценни камъни, които вече сте инсталирали и кои зависимости трябва да бъдат изпълнени чрез инсталиране

нови скъпоценни камъни):

Извличане: Ascii85-1.0.2.gem (100%)

Извличане: ruby-rc4-0.1.5.gem (100%)

Извличане: hashery-2.1.0.gem (100%)

Извличане: ttfunk-1.0.3.gem (100%)

Извличане: afm-0.2.0.gem (100%)

Извличане: pdf-reader-1.3.3.gem (100%)

Извличане: скариди-0.12.0.gem (100%)

Успешно инсталиран Ascii85-1.0.2

Успешно инсталиран ruby-rc4-0.1.5

Успешно инсталиран hashery-2.1.0

Успешно инсталиран ttfunk-1.0.3

Успешно инсталиран afm-0.2.0

Успешно инсталиран pdf-четец-1.3.3

Успешно инсталирана скарида-0.12.0

Инсталирани са 7 скъпоценни камъка

|  |
| --- |
| **Страница 32** |

**32**

C ГЛАВА 1 ***Стартирайте вашата грабитна грамотност***

Тези доклади за състоянието са последвани от няколко реда, указващи, че ri и RD oc документират

инсталирането на различни скъпоценни камъни. (Инсталирането на документа-

тацията включва обработка на изходните файлове на скъпоценния камък чрез RD oc, така че бъдете търпеливи; това е

често най-дългата фаза на инсталиране на скъпоценни камъни.)

По време на процеса на инсталиране на скъпоценен камък, скъпоценният камък изтегля скъпоценни файлове при нужда от ruby-

gems.org ( [www.rubygems.org).](https://translate.google.com/translate?hl=en&prev=_t&sl=auto&tl=bg&u=http://www.rubygems.org) Тези файлове, които са във формат .gem, се записват в

кеш поддиректория на вашата директория за скъпоценни камъни. Можете също да инсталирате скъпоценен камък от скъпоценен файл

пребиваващи локално на вашия твърд диск или друг носител. Посочете името на файла

към инсталатора:

$ gem install /home/me/mygems/ruport-1.4.0.gem

Ако назовете скъпоценен камък без цялото име на файл (например ruport), скъпоценен камък го търси

в текущата директория и в локалния кеш, поддържан от системата RubyGems.

Локалните инсталации все още търсят дистанционно зависимости, освен ако не предоставите -l

(локален) флаг на командния ред към командата на скъпоценен камък; този флаг ограничава всички операции до

локален домейн. Ако искате да са инсталирани само отдалечени скъпоценни камъни, включително зависимости, тогава

можете да използвате флага -r (отдалечен). В повечето случаи обикновеният скъпоценен камък инсталира gemname

командата ще ви даде това, от което се нуждаете. (За да деинсталирате скъпоценен камък, използвайте деинсталирането на скъпоценен камък

команда gemname.)

След като сте инсталирали скъпоценен камък, можете да го използвате чрез метода require.

L ИЗПОЛЗВАНЕ И ИЗПОЛЗВАНЕ НА СКЪЦИ

Въпреки че няма да видите скъпоценни камъни в първоначалния път на зареждане ($ :), все пак можете да ги „изисквате“

и те ще се заредят. Ето как бихте искали "мотика" (помощна програма, която ви помага да пакетирате вашия

собствени скъпоценни камъни), ако приемем, че сте инсталирали скъпоценния камък:

>> изисква "мотика"

=> вярно

В този момент съответната директория за мотика ще се появи в пътя за зареждане, както можете да видите дали

отпечатвате стойността на $: и grep (изберете по съвпадение на модела) за модела "мотика":

>> поставя $ :. grep (/ hoe /)

/Users/dblack/.rvm/gems/ruby-2.1.0/gems/hoe-3.8.1/lib

Ако имате инсталиран повече от един скъпоценен камък за определена библиотека и искате да принудите

използване на скъпоценен камък, различен от най-новия, можете да го направите, като използвате метода на скъпоценен камък. (Забележка

че този метод не е същият като инструмента за команден ред, наречен gem.) Тук, за изпит-

ple, как бихте принудили да използвате не съвсем актуална версия на Hoe:

>> скъпоценен камък "мотика", "3.8.0"

=> вярно

>> поставя $ :. grep (/ hoe /)

/Users/dblack/.rvm/gems/ruby-2.1.0/gems/hoe-3.8.0/lib

В повечето случаи, разбира се, ще искате да използвате най-новите скъпоценни камъни. Но скъпоценният камък

tem ви дава инструменти за фина настройка на използването на вашия скъпоценен камък, ако трябва да го направите.

С темата за RubyGems на картата, сега приключихме с текущия ни бизнес

bess / директорията - и ще преминем към внимателно изучаване на основния език.

**Няма нужда от изискване, ако**

**използвате метод за скъпоценни камъни**

|  |
| --- |
| **Страница 33** |

**33**

***Обобщение***

***1.5***

***Обобщение***

В тази глава разгледахме редица важни основополагащи теми за Ruby,

включително

Разликата между Ruby (езикът) и ruby ​​(интерпретаторът на Ruby)

Типографията на променливите Ruby (всички, които ще срещнете отново и ще изучавате в

повече дълбочина)

Основни Ruby оператори и вградени конструкции

Писане, съхраняване и стартиране на програмен файл на Ruby

Вход за клавиатура и изход на екрана

Манипулиране на Ruby библиотеки с изискване и зареждане

Анатомията на инсталацията Ruby

Инструментите на командния ред, доставени с Ruby

Вече имате добър план за това как работи Ruby и какви инструменти използва програмата Ruby-

ming среда, а вие сте виждали и практикували някои важни Ruby

техники. Вече сте готови да започнете да изследвате Ruby систематично.

|  |
| --- |
| **Страница 34** |

**34**

*Обекти, методи,*

*и локални променливи*

В тази глава ще започнем да проучваме детайлите на програмната верига на Ruby

гейдж. Ще разгледаме преди всичко концепцията за обекта, около която

почти всеки ред от Ruby кода, който ще напишете, ще се върти. Какво правите с обекти,

най-общо казано, е да им изпращате съобщения, повечето от които съответстват на имена на

методи, които искате от обекта да изпълни. Ще разгледаме доста подробно

при комбинираните процеси на изпращане на съобщения и извикване на метод.

Руби обектите често (може би най-често) се обработват чрез променливи, които представляват

ги изпрати и в тази глава технически ще навлезем по-дълбоко, отколкото досега

същността и поведението на променливите в Ruby. След като преработите това

глава, ще имате твърда опора в пейзажа на обектите от Ruby и техните

манипулация.

***Тази глава обхваща***

Обекти и обектна ориентация

Вродени спрямо възможностите на научен обект

Параметър на метода, аргумент и

извикващ синтаксис

Присвояване и използване на локална променлива

Препратки към обекти

|  |
| --- |
| **Страница 35** |

**35**

***Говорене с предмети***

Както винаги, можете да въведете примерните кодове в irb и / или да ги съхраните във файл, който

след това стартирате с помощта на интерпретатора Ruby. Към който и подход да се наклоните, това е така

не е лоша идея да държите irb сесията отворена, докато продължавате - за всеки случай.

***2.1***

***Говорене с предмети***

Във всяка програма Ruby по-голямата част от дизайна, логиката и действията се въртят наоколо

обекти. Когато пишете Ruby програми, основните ви дейности са създаването на обекти,

да ги дарите със способности и да ги помолите да извършват действия. Обектите са ваши

се справят с вселената на вашата програма. Когато искате нещо да се направи - калкулация-

ция, изходна операция, сравнение на данни - вие поискате от обект да го направи. Отколкото

питам абстрактно дали *едно* е равно на *б* , питаш *за* това дали тя счита за равен с *б* . Ако

искате да знаете дали даден ученик взема клас от даден учител, вие

попитайте ученика: „Вие сте ученик на този учител?“ Как точно този вид заявки

по отношение на структурите на данни и синтаксиса, зависи от спецификата на вашата

грам дизайн. Но през цялото време писането на програма Ruby е до голяма степен въпрос на инженер-

да обектите ви така, че всеки обект да играе ясна роля и да може да изпълнява свързани действия

към тази роля.

***2.1.1***

***Рубин и обектна ориентация***

Руби стига до идеята за манипулиране на данни чрез обекти чрез програмата-

езиков дизайн принцип *обектна ориентация* . Много изключително популярна програмна мрежа

графите са обектно-ориентирани (като Java, C ++ и Python, както и Ruby) и някои

езиците, които не са напълно обектно-ориентирани, разполагат с възможности за писане на обектно-ориентирани

код (например Perl, както е описано в *Обектно-ориентиран Perl* от Damian Conway [Manning,

1999]). В обектно-ориентираното програмиране ( OOP ) извършвате изчисления, манипулиране на данни

ulation и операции за въвеждане / извеждане чрез създаване на обекти и искане от тях да изпълняват

действия и ви предоставя информация.

В повечето обектно-ориентирани езици, включително Ruby, всеки обект е пример или

екземпляр на определен клас и поведението на отделните обекти се определя на

поне до известна степен чрез дефинициите на метода, присъстващи в класа на обекта. Добре

изследвайте класовете в дълбочина в глава 3. Тук ще се фокусираме директно върху обектите.

Реалният свят

Понятието *реален свят* се хвърля много в дискусиите за програмиране. Има

стая за дебат (и там *е* разискване) за това дали този или онзи програмиране стан-

guage, или дори този или онзи вид език за програмиране, съответства по-тясно

отколкото други към формата на реалния свят. Много зависи от това как възприемате

света. Възприемате ли го като запълнено с неща, всяко от които има задачи за вършене и

чака някой да поиска задачата? Ако е така, можете да заключите, че е обектно-ориентиран

езиците моделират най-добре света. Виждате ли живота като поредица от задачи за проверка при проверка

списък, за да преминете по ред? Ако е така, може да видите строго процедурно програмиране

езикът като имащ по-тесни връзки с характеристиките на реалния свят.

|  |
| --- |
| **Страница 36** |

**36**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

Проектирането на обектно-ориентиран софтуер е до голяма степен въпрос да разберете какво искате

вашите обекти да бъдат: какво трябва да правят, как ще си взаимодействат помежду си, как

много от всеки трябва да има (например много студенти, един регистратор) и други

такива въпроси. Както ще видите, Ruby предоставя пълен набор от инструменти за именуване, създаване на

обръщане, адресиране и манипулиране на обекти - и чрез манипулирането на тези

обекти, данните, върху които работят.

***2.1.2***

***Създаване на родов обект***

Отначало концепцията за ООП има тенденция да се среща едновременно като проста (вие пишете програми

които имат книги и бутилки, коли и къщи, а вие организирате един вид

версия между тези неща) и абстрактни ( *Обект? Какво означава това? Какво въвеждам*

*в моя програмен файл, за да създам обект „къща“* ?). OOP има компонент на простота;

тя ви позволява да черпите от обекти, обекти, роли и поведения като източник за начина, по който проектирате

вашите програми и това може да ви помогне. В същото време да създавате и използвате обекти в

вашите програми, трябва да научите как се прави на даден език.

ЗАБЕЛЕЖКА В зависимост от вашия произход и очаквания, може да се чудите

да разберем защо не започваме нашето изследване на обекти отблизо

класове, а не обекти. Класовете са важни в Ruby; те са начин да

поведение на пакети и етикети (можете да имате клас Person, клас Task и т.н.

on) и лесно да създавате множество обекти с подобно поведение. Но - и в

в това отношение Ruby се различава от някои други обектно-ориентирани езици -

реалното действие е с отделните обекти: всеки обект има потенциал да

„Научете“ поведение (методи), на което класът не го е научил. Концепцията на класа

пасва върху концепцията на обекта, а не обратното. Всъщност, клас в

Ruby сама по себе си е обект! По-късно ... но това е основата на това защо започваме

с предмети.

Виждането на специфично за езика обяснение на ООП може да улесни абстрактните части

хванете. Следователно ще преминем към някакъв Ruby код. Ще създадем нов обект. Няма да стане

представляват или моделират нещо конкретно, като къща, книга или учител; ще бъде a

родов обект:

obj = Object.new

Сега имате обект и променлива, чрез които можете да го адресирате.

Всички обекти на Ruby са създадени с определени вродени способности - определени методи, които

те знаят как да изпълняват, защото са обекти на Ruby. Тези способности обаче е

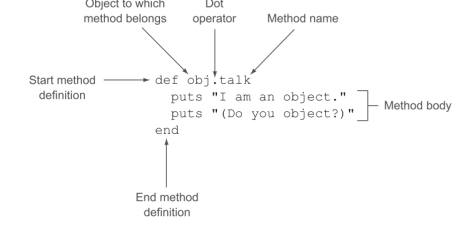


Figure 2.1 Anatomy of a method definition

|  |
| --- |
| **Страница 37** |

**37**

***Говорене с предмети***

важно, не са ли толкова вълнуващи, така че за момента ще ги държим отстрани. | Повече ▼

вълнуващо е това, което се случва, когато научите обекта си как да прави нещата, които искате

това да направя.

D EFINING обект " S ПОВЕДЕНИЕ

Да приемем, че сте създали обект и искате той да направи нещо интересно: вие

искам да говори. За да го накарате да говори, трябва да го помолите да говори. Но преди да го помолите да говори,

трябва да го научите как да говори.

По-конкретно и по-технически, трябва да дефинирате *метод* за вашия обект. Вие

направете това, като използвате специален термин - *ключова дума* - име, ключовата дума def.

Ето как да дефинирате метода talk за обекта obj:

def obj.talk

поставя „Аз съм обект“.

поставя "(възразявате ли?)"

край

Фигура 2.1 показва анализ на тази част от кода.

Сега obj знае как да говори и можете да го помолите да го направи.

S КРАЙНИ СЪОБЩЕНИЯ ЗА ОБЕКТИ

За да помолите obj да говори, използвайте синтаксиса за изпращане на съобщения или извикване на метод, който срещнахте

в глава 1:

obj.talk

И той говори:

Аз съм обект.

(Възразявате ли?)

Обектът obj разбира или *отговаря* на съобщението. Казва се обект

отговори на съобщение, ако обектът има дефиниран метод, чието име съответства

съобщението.

Няколко неща, които трябва да имате предвид относно синтаксиса за изпращане на съобщения, базиран на точки:

Точката (.) Е операторът за изпращане на съобщения. Съобщението отдясно е изпратено

към обекта (или *приемника* , както често се нарича в тази роля) вляво.

Обект на който

метод принадлежи

Метод за стартиране

определение

Краен метод

определение

Тяло на метода

Точка

оператор

Име на метода

def obj.talk

поставя „Аз съм обект“.

поставя "(възразявате ли?)"

край

Фигура 2.1 Анатомия на

дефиниция на метод

|  |
| --- |
| **Стр. 38** |

**38**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

Приемникът може да бъде, и често е, представен от променлива, която заменя

обект. Но приемникът може да бъде и буквална обектна конструкция - например a

низ в кавички.

На практика изпращаното съобщение почти винаги е името на метод (като

беседа, методът, дефиниран по-рано). Обектът винаги се опитва да действа въз основа на

че съобщението е името на метод. Ако няма метод по този начин

име, се вземат мерки за обработка на грешки.

Семантиката на извикванията на методи ви позволява да отидете по-далеч от сравнително едномерната

Случай „talk“, особено когато започнете да извиквате методи с аргументи.

***2.1.3***

***Методи, които вземат аргументи***

Методите в Ruby много приличат на математически функции: въвеждането влиза, колелата

завой и излиза резултат. За да подадете вход към метод Ruby, извиквате метода

с един или повече аргументи.

В дефиниция на метод посочвате аргументите посредством списък с променливи

в скоби след името на метода. (Аргументите могат да бъдат изискани или незадължителни. Ще го направим

погледнете задължителните аргументи тук и незадължителните аргументи малко по-късно.) Когато се обадите

метода, вие предоставяте стойности, съответстващи на тези променливи във вашето извикване на метод.

По-точно променливите, изброени в дефиницията на метода, са *формални* за метода

*параметри* и стойностите, които предоставяте на метода, когато го извикате, са корекцията

спонгиращи *аргументи* . (Често се използва думата *аргументи* , неофициално, за да се позове на a

параметри на метода, както и аргументите на извикване на метод, но е полезно да знаете

техническо разграничение.)

Да предположим, че искате вашият обект да функционира като преобразувател Целзий-Фаренхайт. Вие

може да го научи как да извършва преобразуването, като дефинира метод:

def obj.c2f (c)

c \* 9,0 / 5 + 32

край

(Този път 9 стана 9.0 във формулата за преобразуване. Това ще принуди резултата да бъде

float, което е по-точно от цяло число.) Методът obj.c2f има един формален

параметър, което означава, че отнема един аргумент. Когато извикате метода, вие

вижте аргумент:

поставя obj.c2f (100)

Резултатът е

212,0

Както можете да видите, има пряко съответствие между синтаксиса на параметъра

list в дефиниция на метод и синтаксиса на списъка с аргументи, когато методът е

Наречен. В двата случая скобите не са задължителни; можете да направите това

def obj.c2f c

|  |
| --- |
| **Страница 39** |

**39**

***Говорене с предмети***

и този:

obj.c2f 100

Те не винаги са по избор, особено когато нанизвате множество

методът извиква заедно, така че е добре да се наклоните към използването им, вместо да ги оставяте

навън. Можете да направите изключение за често срещани или конвенционални случаи, когато

ses обикновено се изключват, като повикванията към путове. Но когато се съмнявате, използвайте скобите.

В другия край на процеса всяко извикване на метод *връща* обратно - *връща* стойност.

***2.1.4***

***Връщаната стойност на метод***

Кодът Ruby се състои от изрази, всеки от които оценява до определена стойност.

Таблица 2.1 показва някои примери за изрази и техните стойности (заедно с обясненията

тори коментари).

Погледнете последния запис в таблица 2.1: това е извикване на obj.c2f. Всяко извикване на метод е израз

сион. Когато извикате метод, извикването на метод оценява нещо. Този резултат от

извикването на метод е *връщаната стойност* на метода .

Връщаната стойност на всеки метод е същата като стойността на последния израз eval-

използвани по време на изпълнение на метода. В случай на преобразуване на температурата

метод, последният оценен израз е единственият ред на тялото на метода:

c \* 9,0 / 5 + 32

По този начин резултатът от това изчисление осигурява възвръщаемата стойност на метода.

Ruby ви дава ключова дума за изрично връщане на стойности за връщане: return. Използването на това

ключовата дума обикновено не е задължителна, но много програмисти обичат да я използват, защото я прави

изрично това, което иначе се подразбира:

def obj.c2f (c)

връщане c \* 9.0 / 5 + 32

край

Таблица 2.1 Примери за изрази на Ruby и стойностите, на които те оценяват

Израз

Стойност

Коментари

2 + 2

4

Аритметичните изрази оценяват резултатите си.

"Здравейте"

"Здравейте"

Прост, буквален низ (в кавички) оценява

към себе си.

"Здравей" + "там"

"Здравей"

Низовете могат да се „добавят“ един към друг (обединени)

със знака плюс.

c = 100

100

Когато присвоявате на променлива, цялото присвояване

оценява на стойността, която сте задали.

c \* 9/5 + 32

212

Прилагат се обичайните правила за приоритет: умножение

и разделението се свързват по-плътно от добавянето и са

изпълнени първи.

obj.c2f (100)

212

Извикването на метод е израз.

|  |
| --- |
| **Страница 40** |

**40**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

Това е еквивалентно на по-ранната версия на метода, но е по-изразително за

какво прави От друга страна е по-трескаво. Трябва да решите като въпрос

вашия собствен стил, независимо дали искате да използвате return. Трябва да го използвате, ако върнете мулти-

ple стойности, които автоматично ще бъдат опаковани в масив: върнете по-скоро a, b, c

отколкото просто a, b, c (въпреки че можете да върнете множество стойности в изричен масив, като

[a, b, c], без връщане). Също така трябва да използвате return, ако искате да се върнете от

някъде по средата на метод. Независимо дали използвате return или не, *нещо*

ще се връща от всяко извикване на метод. Дори извикване на празно тяло на метода, състоящо се от

извличане само на операторите def и end, връща нула.

В този момент обектът прави това, от което се нуждаем: слушане на съобщения и

действайки върху тях. Това е добра илюстрация на това как работи Ruby, но е оскъдна.

Започнахме с общ обект и го научихме да говори и да преобразува температурите. Че

ви показва механиката на дефиниране и извикване на методи, но това води до по-скоро

странен обект. Нека разгледаме обект, който моделира нещо малко по-структурирано.

Ще изработим общ обект, така че той да разбира поведението и бизнес логиката

на билет за събитие.

***2.2***

***Изработване на обект: Поведението на билета***

Билетът е познат обект с известен набор от свойства и поведения. Да вземем a

изглед на високо ниво на това, което очакваме да направи подобен на билет обект Ruby и да знае за себе си.

***2.2.1***

***Билетният обект, първо поведението***

Обектът на билета трябва да може да предоставя данни за себе си. Трябва да подава заявки за

информация за събитието, за което е предназначено: кога, къде, име на събитието, изпълнител, кой

място и колко струва.

Когато бъдете помолени, билетът ще предостави следната информация въз основа на изображение

публично четене от Марк Твен през 1903 г .:

01/02/03

Кметство

Авторско четиво

Марк Твен

Втори балкон, ред J, седалка 12

5,50 долара

Целта е да създадем обект, от който лесно да получим цялата тази информация.

C ПОВТОРЕНО ОБЕКТ НА БИЛЕТ

Общ обект ще служи като основа за билета:

билет = Object.new

След като съществува, можем да започнем да даряваме обектния билет със свойства и данни от

дефиниране на методи, като всеки връща съответната стойност:

def ticket.date

"01/02/03"

край

|  |
| --- |
| **Страница 41** |

**41**

***Изработване на обект: Поведението на билета***

def Tick.venue

"Кметство"

край

def билет.event

"Авторско четене"

край

def билет.изпълнител

"Марк Твен"

край

def билет.седалка

"Втори балкон, ред J, седалка 12"

край

def билет.цена

5.50

край

По-голямата част от дефинираните тук методи връщат низови стойности. Можете да видите това в a

поглед: връщат стойност в кавички. Методът на цената се връща

на число с плаваща точка Б . Сега, когато обектът на билета знае малко за себе си, нека

помолете го да сподели информацията.

***2.2.2***

***Заявка за билетния обект***

Вместо да създаваме суров списък с елементи, нека генерираме удобно за читателите обобщение на

подробности за билета. Използването на печат и путове може да помогне за получаване на информация

повече или по-малко разказвателна форма:

print "Този билет е за:"

печат на билет.event + ", в"

печат на ticket.venue + ", on"

поставя ticket.date + "."

печат "Изпълнителят е"

поставя ticket.performer + "."

print "Седалката е"

печат на билет.seat + ","

print "и струва $"

поставя "% .2f." % цена на билет

Запазете целия код, като започнете с ticket = Object.new, във файл, наречен ticket.rb, и изпълнете

то. Ще видите следното:

Този билет е за: Авторско четене, в кметството, на 01/02/03.

Изпълнител е Марк Твен.

Седалката е втори балкон, ред J, седалка 12 и струва $ 5,50.

Кодът за този пример се състои от поредица от извиквания към методите, дефинирани по-рано:

ticket.event, ticket.venue и т.н. Кодът за печат вгражда тези повиквания - в

с други думи, той вгражда възвръщаемите стойности на тези методи ("Прочитане на автора",

"Кметство" и т.н.) - в последователност от изходни команди и добавя конектори

(", at", ", on" и т.н.), за да накарате текста да се чете добре и да изглежда добре.

Билетът за Твен е прост пример, но включва някои жизненоважни процедури на Ruby

дюри и принципи. Най-важният урок е, *че знанията, необходими за*

б

**Събитие за печат**

**информация**

**Печатен изпълнител**

**информация**

**Седалка за печат**

**информация**

**Отпечатайте число с плаваща запетая**

**до два знака след десетичната запетая**

|  |
| --- |
| **Страница 42** |

**42**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

*програма, която да прави каквото и да е полезно, се намира в обекта* . Билетният обект притежава знанията; Вие

използвайте това знание, като поискате билета за него чрез извиквания на методи. Нищо не е повече

от основно значение за програмирането на Ruby от това. Всичко е в това да се иска от обекти да правят неща и

да ти кажа неща.

Кодът на билета работи и въплъщава полезни уроци; но е многословно. Руби има

репутацията на мощен език на високо ниво. Трябва да можете да получите много

направено с относително малко код. Но примерът за билет отнема десет реда печат и

поставя инструкции за генериране на три реда изход.

Нека подобрим малко това съотношение.

***2.2.3***

***Съкращаване на кода на билета чрез интерполация на низове***

Една от най-полезните техники за програмиране, достъпни в Ruby, е *интерполация на низове*

*ция* . Операторът за интерполация на низове ви дава начин да пуснете нещо в низ:

променлива например или връщаната стойност на метод. Това може да ви спести много

назад и напред между отпечатването и поставянето.

Освен това низовете могат да бъдат обединени със знака плюс (+). Ето как

изглежда, като се използва интерполация на низове за вмъкване на стойностите на изразите в

низ и използването на добавяне на низ за консолидиране на множество поставя повиквания в едно:

поставя „Този ​​билет е за: # {ticket.event}, на номер {{ticket.venue}.“ +

„Изпълнителят е # {ticket.performer}.“ +

„Седалката е # {ticket.seat},“ +

"и струва $ # {"%. 2f. "% ticket.price}"

Каквото и да е вътре в оператора за интерполация # {...} се изчислява отделно и

резултатите от изчислението се вмъкват в низа. Когато изпълните тези редове,

няма да видите оператора # {...} на екрана си; вместо това ще видите резултатите от кал

кулиране или оценка на това, което е било между къдравите скоби. Интерполацията помогна на elimi-

nate 6 от 10 реда код и също така направи кода да прилича много повече на евентуалния

формат на изхода, а не нещо, което работи, но не предава много

визуална информация.

Досега искахме билета за информация под формата на низове и цифри

bers. Билетите също така съдържат вярна / невярна - логическа информация за себе си.

***2.2.4***

***Наличност на билета: Изразяване на булево състояние в метод***

Като логическа информация, помислете за въпроса дали билетът е продаден

или все още е на разположение. Един от начините да дарите билет със знания за собствената си наличност

tus е това:

def Tick.availability\_status

"продаден"

край

Друг начин е да попитате билета дали е наличен и да го уведомите за вярно или невярно:

def ticket.available?

невярно

край

|  |
| --- |
| **Страница 43** |

**43**

***Изработване на обект: Поведението на билета***

false е специален термин в Ruby, както и терминът true. true и false са обекти. Руби

използва ги, за да представи резултатите от, наред с други неща, операции за сравнение (като

x> y) и можете да ги използвате, за да представите истината и лъжата във вашите собствени методи.

Може би сте забелязали, че името на метода е налично? завършва с въпросителен знак.

Ruby ви позволява да направите това, за да можете да пишете методи, които оценяват на true или false и правят

извикванията на метода изглеждат като въпроси:

ако билетът е наличен?

поставя "Имате късмет!"

друго

поставя „Извинете - тази седалка е продадена.“

край

Но в истината и лъжа има нещо повече от истинските и фалшивите обекти. Всеки експрес-

sion в Ruby оценява на обект и всеки обект в Ruby има стойност на истината. Истината

стойност на почти всеки обект в Ruby е вярна. Единствените обекти, чиято истинска стойност (или Bool-

ean value) е false са обектът false и специалният обект nonentity nil. Ще видите

Булеви стойности и нула по-подробно в глава 7. За момента можете да се сетите

както фалшиви, така и нулеви като функционално еквивалентни показатели за отрицателен резултат от теста.

Играта с изразите if в irb е добър начин да усетите колко условни

националната логика се играе в Ruby. Опитайте няколко примера като тези:

>> ако "abc"

>> поставя "Струнните са" верни "в Ruby!"

>> край

Струните са „истински“ в Ruby!

=> нула

>> ако 123

>> поставя "Номерата също!"

>> край

Номерата също!

=> нула

>> ако 0

>> поставя "Дори 0 е вярно, което не е на някои езици."

>> край

Дори 0 е вярно, което не е на някои езици.

=> нула

>> ако 1 == 2

>> поставя "Един не е равен на два, така че това няма да се появи."

>> край

=> нула

(Първият от тези примери, ако е "abc", ще генерира предупреждение за низови литерали в

условия. Можете да игнорирате предупреждението за настоящите ни цели.)

Забележете как irb не само се подчинява на извикванията на метода put B, но и по своя инициатива

tive, извежда стойността на целия израз c . В случаите, когато путове

писалки, целият израз се оценява на нула - защото възвръщаемата стойност на путове е

винаги нула. В последния случай, когато низът не се отпечатва (тъй като условието е неуспешно),

стойността на израза също е нула - защото оператор if, който се проваля (и няма

else клон да го спаси) също оценява на нула d .

б

° С

д

|  |
| --- |
| **Страница 44** |

**44**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

Като си спомните, че nil има булева стойност false, можете, ако желаете, да влезете

акробатика с irb. Извикването към put връща нула и следователно е невярно, въпреки че

низ се отпечатва. Ако поставите путове в клауза if, клаузата ще бъде невярна. Но ще стане

все още да бъдат оценени. Така,

>> ако постави "Ще видите това"

>> поставя "но не това"

>> край

Ще видите това

=> нула

Първите путове се изпълняват, но стойността, която връща, а именно nil, не е вярна в Bool-

смисъл - така че второто путване не се изпълнява.

Това е измислен пример, но е добре да свикнете с факта, че *всеки*

*нещо в Ruby има булева стойност* и понякога не е това, което може да очаквате. Както е

често случаят, irb може да бъде от голяма помощ при овладяването на тази концепция.

Сега, когато билетният обект има някои ръчно изработени поведения, нека да се върнем обратно

и разгледайте въпроса с какво поведение е надарен всеки обект в Ruby

неговото създаване.

***2.3***

***Вроденото поведение на обект***

Дори новосъздаденият обект не е празен лист. Веднага щом обектът съществува

то отговаря на редица съобщения. Всеки обект се „ражда“ със сигурност

вродени способности.

За да видите списък с вродени методи, можете да извикате метода на методите (и да хвърлите

операция за сортиране, за да улесните визуалното сърфиране):

p Object.new.methods.sort

Резултатът е списък на всички съобщения (методи), които този новоизсечен обект се предлага

болен с. (Предупреждение: Изходът изглежда претрупан. Ето как Ruby показва масиви -

а методът методи ви дава масив от имена на методи. Ако искате списък на

методи по един на ред, използвайте поставя вместо p в командата.)

[:!,:! =,:! ~,: <=>,: ==,: ===,: = ~,: \_\_ id\_\_,: \_\_ send\_\_,: class,: clone,

: define\_singleton\_method,: display,: dup,: enum\_for,: eql ?,: jednaк ?,: удължи,

: freeze,: frozen ?,: hash,: inspect,: instance\_eval,: instance\_exec,

: instance\_of ?,: instance\_variable\_defined ?,: instance\_variable\_get,

: instance\_variable\_set,: instance\_variables,: is\_a ?,: kind\_of ?,: метод,

: методи,: нула ?,: object\_id,: private\_methods,: protected\_methods,

: public\_method,: public\_methods,: public\_send,: remove\_instance\_variable,

: response\_to ?,: send,: singleton\_class,: singleton\_methods,: taint,: tainted ?,

: tap,: to\_enum,: to\_s,: trust,: untaint,: untrust,: untrusted?]

Не се притеснявайте, ако повечето от тези методи в момента нямат смисъл за вас. Можете да опитате

ги в irb, ако ви е интересно да видите какво правят (и ако не се страхувате да получите

някои съобщения за грешка).

|  |
| --- |
| **Страница 45** |

**45**

***Вроденото поведение на обект***

Но някои от тези вродени методи са достатъчно често срещани и достатъчно полезни,

дори в ранните фази на запознаването с Руби - че ще ги разгледаме подробно

тук. Следните методи отговарят на това описание:

object\_id

отговарям на?

изпрати (синоним: \_\_send\_\_)

Добавянето им към вашата кутия с инструменти на Ruby няма да е лошо поради това, което правят и

защото те служат като примери за вродени методи.

***2.3.1***

***Идентифициране на обекти по уникален начин с метода object\_id***

Всеки обект в Ruby има уникален идентификационен номер, свързан с него. Можете да видите

ID на обекта, като помолите обекта да ви покаже неговия object\_id, използвайки този или подобен код:

obj = Object.new

поставя "Идентификаторът на obj е # {obj.object\_id}."

str = "Низовете също са обекти и това е низ!"

поставя "Идентификаторът на низовия обект str е # {str.object\_id}."

поставя "А идентификаторът на цялото число 100 е # {100.object\_id}."

Наличието на уникален идентификационен номер за всеки обект може да ви бъде полезен, когато се опитвате

определят дали два обекта са еднакви един с друг. Как могат да бъдат два обекта

един и същ? Е, целият обект 100 е същият като ... целият обект 100. (Попитайте 100

за неговия идентификатор на обект два пъти и резултатът ще бъде същият.) И ето още един случай:

a = Object.new

b = a

поставя „идентификаторът на a е # {a.object\_id}, а идентификаторът на b е # {b.object\_id}.“

Въпреки че променливите a и b са различни, обектът, към който и двамата се отнасят, е

един и същ. (Вижте раздел 2.5.1 за повече информация относно концепцията за препратки към обекти.) Обратното

може да се случи и сценарий: понякога два обекта изглеждат еднакви, но те са

не. Това се случва много със струни. Помислете за следния пример:

string\_1 = "Здравейте"

string\_2 = "Здравейте"

поставя "идентификаторът на string\_1 е # {string\_1.object\_id}."

поставя „id на string\_2 е # {string\_2.object\_id}.“

Общи обекти спрямо основни обекти

Помолете Ruby да създаде нов обект за вас с командата Object.new произвежда

това, което наричаме тук, неофициално, *родов* обект. Ruby също има *основни* обекти -

и това е по-официално име. Ако се обадите на BasicObject.new, получавате вид прото-

обект, който може да направи много малко. Дори не можете да помолите основен обект да ви покаже метода му

ods, защото няма метод метод! Всъщност има само седем метода - достатъчно

за да може обектът да съществува и да бъде разпознаваем, и не много повече. Ще научите повече за

тези основни обекти в глави 3 и 13.

**string\_1 id:**

**287090**

**string\_2 id:**

**279110**

|  |
| --- |
| **Страница 46** |

**46**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

Въпреки че тези два низа съдържат един и същ текст, технически те не са еднакви

обект. Ако ги разпечатате, и два пъти ще видите един и същ резултат („Здравейте“). Но

самите низови обекти са различни. Все едно да имаш две копия на една и съща

книга: те съдържат един и същ текст, но не са едно и също нещо. Вие

може да унищожи единия, а другият няма да бъде засегнат.

ИДЕНТИФИКАЦИОННИ НОМЕРА И РАВНОСТ НА ОБЕКТИТЕ

Както в случая с човешките институции, един от пунктовете за даване на идентификационни номера на обекти в

Руби трябва да може да прави уникални идентификации и по-специално да може

определят кога два обекта са един и същ обект.

Ruby предоставя разнообразни начини за сравняване на обекти за различни видове равенство. Ако

имате два низа, можете да тествате, за да видите дали съдържат едни и същи символи. Вие

може също така да тества дали един и същ обект (което, както току-що видяхте, не е

задължително случаят, дори ако съдържат едни и същи знаци). Същото важи,

с леки вариации, за други обекти и други видове обекти.

Сравняването на идентификационни номера за равенство е само един от начините за измерване на равенството на обекта.

Ще влезем в повече подробности за тези сравнения малко по-късно. Точно сега ще се обърнем

към следващия вроден метод от нашия списък: response\_to ?.

***2.3.2***

***Запитване за способностите на обекта с response\_to? метод***

Обектите Ruby отговарят на съобщения. По различно време по време на изпълнение на програма,

За обекта и какви видове методи са дефинирани за него, обектът може

или може да не отговори на дадено съобщение. Например следният код води до

грешка:

obj = Object.new

obj.talk

Ruby се радва само да ви уведоми за проблема:

NoMethodError: недефиниран метод `talk 'за # <Обект: 0x00000102836550>

Можете предварително да определите (преди да помолите обекта да направи нещо) дали

обектът знае как да обработва съобщението, което искате да му изпратите, като използва

отговарям на? метод. Този метод съществува за всички обекти; можете да попитате всеки предмет

дали отговаря на някакво съобщение. отговарям на? обикновено се появява във връзка с

условна (ако) логика:

obj = Object.new

ако obj.respond\_to? ("разговор")

obj.talk

друго

поставя „За съжаление обектът не разбира съобщението„ talk “.“

край

отговарям на? е пример за *самоанализ* или *размисъл* , два термина, които се отнасят до

изследване на състоянието на програмата, докато тя работи. Ruby предлага редица удобства за

самоанализ. Изследване на методите на обекта с метода методите, както направихме

|  |
| --- |
| **Страница 47** |

**47**

***Вроденото поведение на обект***

по-рано, е друга интроспективна или отразяваща техника. (Ще видите много повече такива

техники в част 3 на книгата.)

Досега използвахме точковия оператор (.) За изпращане на съобщения до обекти. Нищо

не е наред с това. Но какво, ако не знаете кое съобщение искате да изпратите?

***2.3.3***

***Изпращане на съобщения до обекти с метода на изпращане***

Да предположим, че искате да позволите на потребителя да получава информация от обекта на билета, като въведете

подходящ термин за заявка (място, изпълнител и т.н.) на клавиатурата. Ето какво

бихте добавили към съществуващата програма:

print "Желана информация:"

заявка = gets.chomp

Вторият ред на кода получава ред на въвеждане от клавиатурата, "chomps" от последващия new-

ред символ и запазва получения низ в заявката за променлива.

В този момент можете да тествате входа за една стойност след друга, като използвате

ble оператор за сравнение на равен знак (==), който сравнява низове въз основа на тяхната

палатка и извикване на метода, чиято стойност осигурява съвпадение:

ако заявка == "място"

поставя билет.приход

elsif request == "изпълнител"

поставя билет.изпълнител

...

За да бъдете задълбочени обаче, ще трябва да продължите през целия списък с предложения за билети

erties. Това ще стане дълго.

Има алтернатива: можете да изпратите думата директно към обекта на билета. Вместо

от предишния код, бихте направили следното:

ако билет.отговорете\_ към? (заявка)

поставя ticket.send (заявка)

друго

поставя „Няма такава информация“

край

Тази версия използва метода за изпращане като универсален начин за получаване на съобщение до

билетен обект. Освобождава ви от необходимостта да преминете през целия списък с възможни

заявки. Вместо това, след като сте проверили дали обектът на билета знае какво да прави Б , вие

предайте съобщението на билета и го оставете да свърши своето.

Използване на **\_\_send\_\_** или **public\_send** вместо **изпращане**

Изпращането е широко понятие: изпраща се имейл, данните се изпращат към I / O сокети и т.н.

напред. Не е необичайно програмите да дефинират метод, наречен изпращане, който противоречи

с вградения в Ruby метод за изпращане. Следователно Ruby ви дава алтернативен начин за обаждане

изпрати: \_\_send\_\_. По споразумение никой никога не пише метод с това име, така че

вградената версия на Ruby е винаги достъпна и никога не влиза в конфликт с новозаписани

десет метода. Изглежда странно, но е по-безопасно от обикновената версия за изпращане от

гледна точка на сблъсъци с име на метод.

б

|  |
| --- |
| **Страница 48** |

**48**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

През повечето време ще използвате оператора на точки, за да изпращате съобщения до обекти. Но изпращането

алтернатива може да бъде полезна и мощна - достатъчно мощна и достатъчно податлива на грешки,

че почти винаги заслужава поне нивото на мрежово обезпечаване, представено чрез призив към

отговарям на?. В някои случаи отговаряйте\_ на? може дори да е твърде широк, за да бъде в безопасност; Вие

може да изпрати съобщение до обект само ако съобщението е включено в предварително определен

съобщение „бял ​​списък“. Водещият принцип е грижата: бъдете внимателни при изпращане на произволни

съобщения към обекти, особено ако тези съобщения се основават на потребителски избор или въвеждане.

След това ще поставим синтаксиса и семантиката на аргумента на метода под микроскоп.

***2.4***

***Поглед отблизо на аргументите на метода***

Методите, които пишете в Ruby, могат да вземат нула или повече аргументи. Те също могат да позволят a

променлив брой аргументи. Ще разгледаме семантиката на аргументите в няколко различни

начини в този раздел:

Разликата между задължителни и незадължителни аргументи

Как да присвоите стойности по подразбиране на аргументи

Правилата, уреждащи реда, в който трябва да подредите параметрите

подписът на метода, така че Ruby да може да осмисли списъците с аргументи в метода

извиква и обвързва параметрите правилно

Какво не можете да правите с аргументи в Ruby

Таблица 2.2 ще ги обобщи в края на този раздел.

ЗАБЕЛЕЖКА Семантиката на списъка с аргументи има повече, отколкото ще разгледаме тук. Специфи-

в Руби има нещо като аргументи на ключови думи (или с име parame-

ters). Тази функция е силно свързана с използването на хеш като метод

аргументи - поради което няма да видите пълно обяснение за това, докато не го видим

говори за хешове в дълбочина в глава 9.

***2.4.1***

***Задължителни и незадължителни аргументи***

Когато извиквате метод на Ruby, трябва да предоставите точния брой аргументи.

Ако не го направите, Руби ви казва, че има проблем. Например извикване на един аргумент

метод с три аргумента

obj = Object.new

def obj.one\_arg (x)

поставя "Изисквам един и само един аргумент!"

край

obj.one\_arg (1,2,3)

***(продължение)***

Освен това има безопасна, но по различен начин версия на изпращането (или \_\_send\_\_)

наречен public\_send. Разликата между обикновения send и public\_send е тази

send може да извика частни методи на обект, а public\_send не. Покриваме частно

методи по-късно в книгата, но в случай, че ви е интересно какво прави public\_send

в списъка с методи, това е същността.

|  |
| --- |
| **Страница 49** |

**49**

***Поглед отблизо на аргументите на метода***

резултати в

ArgumentError: грешен брой аргументи (3 за 1)

Възможно е да се напише метод, който позволява произволен брой аргументи. За да направите това, поставете a

звезда (звездичка, \*) пред име на единичен аргумент:

def obj.multi\_args (\* x)

поставя "Мога да взема нула или повече аргументи!"

край

Нотацията \* x означава, че когато извикате метода, можете да предоставите произволен брой

аргументи (или няма). В този случай на променливата x се присвоява масив от стойности, съответстващи на

спондиране на каквито и аргументи да са били изпратени. След това можете да разгледате стойностите една в а

време чрез обхождане на масива. (Ще разгледаме по-отблизо масивите в глава 9.)

Можете да прецизирате броя на аргументите чрез смесване на задължителни и незадължителни

аргументи:

def two\_or\_more (a, b, \* c)

поставя "Изисквам два или повече аргумента!"

поставя "И разбира се, разбрах:"

pa, b, c

край

В този пример a и b са задължителни аргументи. Крайният \* c ще разпръсне всеки

други аргументи, които можете да изпратите и да ги поставите в масив в променливата c. Ако ти

обадете се на two\_or\_more (1,2,3,4,5), ще получите следния отчет за това, на което е присвоено

a, b и c:

Изисквам два или повече аргумента!

И със сигурност имам:

1

2

[3, 4, 5]

(Използване на p вместо печат или поставяне на резултати в масива, който се отпечатва в масив

нотация. В противен случай всеки елемент на масив ще се появи на отделен ред, което го прави

по-трудно е да се види, че въобще е включен масив.)

Можете също така да направите аргумент по избор, като му дадете стойност по подразбиране.

***2.4.2***

***Стойности по подразбиране за аргументи***

Когато предоставяте стойност по подразбиране за аргумент, резултатът е, че ако този аргумент

не е предоставена, променливата, съответстваща на аргумента, получава стойността по подразбиране.

Аргументите по подразбиране са посочени със знак за равенство и стойност. Ето пример:

def default\_args (a, b, c = 1)

поставя "Стойности на променливи:", a, b, c

край

Ако се обадите по този начин

default\_args (3,2)

|  |
| --- |
| **Страница 50** |

**50**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

ще видите този резултат:

Стойности на променливи:

3

2

1

Не беше предоставена стойност в извикването на метода за c, така че c беше зададена на стойността по подразбиране

vided за него в списъка с параметри: 1. Ако предоставите трети аргумент, тази стойност е над-

управлява заданието по подразбиране на 1. Следващото повикване

default\_args (4,5,6)

дава този резултат:

Стойности на променливи:

4

5

6

Истинското забавление започва, когато смесите и съчетаете различните елементи на синхронизиране на аргументи

данък и трябва да разбера в какъв ред да сложа всичко.

***2.4.3***

***Ред на параметрите и аргументите***

Какъв изход бихте очаквали от следния кодов фрагмент?

def mixed\_args (a, b, \* c, d)

поставя "Аргументи:"

pa, b, c, d

край

смесени\_арги (1,2,3,4,5)

Видяхте, че параметър със звезда, като \* c, запълва останалите аргументи -

поне го направи в метода two\_or\_more, където \* c се появи последно в параметъра

списък. Какво се случва, когато следва друг аргумент?

По принцип, Ruby се опитва да присвоява стойности на възможно най-много променливи. И

параметрите на гъбата получават най-ниския приоритет: ако методът остане без аргументи след

той изпълнява заданията на необходимите аргументи, след това общ параметър като

\* c завършва като празен масив. Получават се необходимите аргументи както преди \* c, така и след \* c

се погрижи преди \* c да.

Резултатът от предишния фрагмент е следният:

Аргументи:

1

2

[3, 4]

5

Параметрите a и b получават първите два аргумента, 1 и 2. Тъй като параметърът при

краят на списъка, d, представлява задължителен аргумент, той грабва първата налична стойност

от десния край на списъка с аргументи - а именно, 5. Каквото е останало в средата на

dle (3, 4) се забърква от c.

|  |
| --- |
| **Страница 51** |

**51**

***Поглед отблизо на аргументите на метода***

Ако дадете само достатъчно аргументи, за да съответстват на необходимите аргументи на

метод, тогава гъбеният масив ще бъде празен. Извикването на метода

смесени\_арги (1,2,3)

резултати в този изход:

1

2

[]

3

В този пример c няма късмет; няма нищо останало.

Можете да придобиете разумна фантазия със синтаксиса на параметрите. Ето метод, който взема a

необходим аргумент; незадължителен аргумент, който по подразбиране е 1; още два задължителни спора-

моменти, взети отдясно; и някъде по средата, всичко останало:

def args\_unleashed (a, b = 1, \* c, d, e)

поставя "Аргументи:"

pa, b, c, d, e

край

И ето една irb сесия, която поставя този метод през неговите стъпки. Имайте предвид, че връщането

Стойността на извикването на метода във всеки случай е масив, състоящ се от всички стойности. Това е

връщаната стойност на обаждането до p. Това е представяне на масив на същите стойности като

виждате отпечатани като отделни стойности на отделни редове:

>> args\_unleashed (1,2,3,4,5)

1

2

[3]

4

5

=> [1, 2, [3], 4, 5]

>> args\_unleashed (1,2,3,4)

1

2

[]

3

4

=> [1, 2, [], 3, 4]

>> args\_unleashed (1,2,3)

1

1

[]

2

3

=> [1, 1, [], 2, 3]

>> args\_unleashed (1,2,3,4,5,6,7,8)

1

2

[3, 4, 5, 6]

7

8

б

° С

д

д

|  |
| --- |
| **Страница 52** |

**52**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

=> [1, 2, [3, 4, 5, 6], 7, 8]

>> args\_unleashed (1,2)

ArgumentError: грешен брой аргументи (2 за 3+)

Първата покана за args\_unleashed има пет аргументи B . Това означава, че има

достатъчно, за да обиколи: b получава замяна по подразбиране и масивът c получава един елемент.

Второто обаждане е по-скъпо c , а c губи: b получава, за да замени подразбирането си, оставяйки c

празно (тъй като последните два аргумента се говорят от необходимите аргументи d

и д).

Третото обаждане затяга колана още повече d . Този път има само достатъчно

аргументи за задоволяване на основните изисквания - тоест нещо, което да се присвои на a, d и

д. Параметърът b се връща по подразбиране и c е празен.

Четвъртото обаждане преминава в друга посока e : този път има повече аргументи от

методът изисква и повече от достатъчно за попълване на незадължителните и по подразбиране-

стойностни параметри. Това е бонанса за c, която си върши работата, като разпръсква всички спорове -

моменти, които не са необходими другаде и в крайна сметка съдържат четири елемента.

Петото обаждане, от друга страна, не изпраща достатъчно аргументи, за да задоволи основното

изисквания f . Обвързването на променливите не може да бъде направено, така че получавате грешка в аргумента.

Заедно с гайките на спора

синтаксис, най-важното нещо за отнемане

от тези примери е може би очевидното

посочете, че каквото и да правите, всеки

параметър завършва обвързан с някаква стойност.

Няма такова нещо като параметър, който

просто отплава в нищото. Ако е

в списъка той завършва като локална променлива вътре

методът - дори и само да е обвързан с

празен масив, като c понякога е. Можете или

може да не използва всяка такава променлива, но bind-

винаги се правят ингове.

Фигура 2.2 предлага графично представяне на

основната логика на присвояване на аргументи. Списъкът със задания в полето показва

ред на приоритет: първо се обработват задължителни аргументи, след това стойността по подразбиране по избор

аргумент и след това гъбата.

Ако имате сложни нужди от аргументи, трябва внимателно да следвате правилата - и

имайте предвид и това, което не можете да направите.

***2.4.4***

***Какво не можете да направите в списъците с аргументи***

Параметрите са подредени. Необходимите получават приоритет, независимо дали се случват в

отляво или отдясно на списъка. Всички незадължителни трябва да се появят в средата.

Средата може да е средата на нищо:

def all\_optional (\* аргументи)

е

arg\_demo (1,2,3,4,5,6,7,8)

def arg\_demo (a, b, c = 1, \* d, e, f)

1,2

7,8

3

4,5,6

Необходими аргументи a, b

Необходими аргументи e, f

Незадължителен аргумент c

Аргументен масив d

Фигура 2.2 Логика за присвояване на аргументи

в действие

**Нула отляво или отдясно**

**необходими аргументи**

|  |
| --- |
| **Страница 53** |

**53**

***Поглед отблизо на аргументите на метода***

И можете да имате необходими аргументи само отляво или само отдясно - или и двете.

Това, което не можете да направите, е да поставите гъбата за аргументи вляво от стойността по подразбиране

аргументи. Ако направите това,

def broken\_args (x, \* y, z = 1)

край

това е синтаксична грешка, защото няма начин тя да бъде правилна. След като дадете x на него

аргумент и разпръсна всички останали аргументи в масива y, нищо не може

някога да бъде оставен за z. И ако z получи десния аргумент, оставяйки останалото за y, прави

няма смисъл да се описва z като „по избор“ или „по подразбиране“. Ситуацията става дори трън-

ier, ако се опитате да направите нещо като също толкова незаконно (x, \* y, z = 1, a, b). За щастие,

Ruby не позволява повече от един аргумент на гъба в списък с параметри. Уверете се

подреждате аргументите си разумно и, когато е възможно, поддържате вашите списъци с аргументи отново

звуково просто!

Таблица 2.2 обобщава наученото досега за синтаксиса на аргументи и

семантика. Можете да третирате тази таблица повече като справка, отколкото като нещо, от което се нуждаете

ангажирайте се с паметта и изпитвайте себе си - стига да следвате основните разсъждения на

защо всеки пример работи така, както работи.

Както можете да видите от таблица 2.2, аргументите, които изпращате на методи, се присвояват

към променливи - по-специално локални променливи, видими и използваеми по време на

метод. Присвояването на локални променливи чрез свързване на аргументи на метода е справедливо

Таблица 2.2 Примерни подписи на методите с задължителни, незадължителни и стойности по подразбиране

Тип (и) на аргумента

Подпис на метода

Примерни разговори

Променливи задания

Задължително (R)

def m (a, b, c)

м (1,2,3)

a = 1, b = 2, c = 3

По избор (O)

def m (\* a)

м (1,2,3)

a = [1,2,3]

Стойност по подразбиране (D)

def m (a = 1)

м

м (2)

a = 1

a = 2

R / O

def m (a, \* b)

м (1)

a = 1, b = []

R / D

def m (a, b = 1)

м (2)

м (2,3)

a = 2, b = 1

a = 2, b = 3

D / O

def m (a = 1, \* b)

м

м (2)

a = 1, b = []

a = 2, b = []

R / D / O

def m (a, b = 2, \* c)

м (1)

м (1,3)

м (1,3,5,7)

a = 1, b = 2, c = []

a = 1, b = 3, c = []

a = 1, b = 3, c = [5,7]

R / D / O / R

def m (a, b = 2, \* c, dm (1,3)

м (1,3,5)

м (1,3,5,7)

м (1,3,5,7,9)

a = 1, b = 2, c = [], d = 3

a = 1, b = 3, c = [], d = 5

a = 1, b = 3, c = [5], d = 7

a = 1, b = 3, c = [5,7], d = 9

|  |
| --- |
| **Страница 54** |

**54**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

един случай на общия процес на присвояване на локална променлива, процес, който ще разгледаме

по-нататък в детайли.

***2.5***

***Локални променливи и присвояване на променливи***

Местните имена на променливи започват с малка буква или долна черта и са измислени

на буквено-цифрови знаци и долни черти. Всички те са валидна локална променлива

имена, включително самотното подчертаване:

х

\_х

име

първо име

план9

user\_ID

\_

На *местните* в *местните променливи* се отнася до факта, че те имат ограничен *обхват* : местен общес-

able се вижда само в ограничена част от програма, като дефиниция на метод. Местен

имената на променливите могат да бъдат използвани повторно в различни обхвати. Можете да използвате, да речем, името на променливата x

на повече от едно място и докато тези места имат различен обхват, двете x варират

ables се третират като напълно отделни. (Не забравяйте, че конвенционалният стил на Ruby

fers under\_score имена над camelCase имена за локални променливи.)

Обхватът е важна тема сама по себе си и ние ще вникнем дълбоко в нея в главата

ter 5. Можете да започнете да се запознавате с някои ключови аспекти от него сега, както и вие

изследвайте как местните променливи идват и си отиват. Класическият случай на локален обхват е метод

определение. Вижте какво се случва с x в този пример:

def say\_goodbye

x = "Сбогом"

поставя х

край

def start\_here

x = "Здравей"

поставя х

кажи довиждане

поставя "Нека проверим дали x е останал същият:"

поставя х

край

Започни тук

Резултатът от тази програма е както следва:

Здравейте

Довиждане

Нека проверим дали x остана същият:

Здравейте

Когато извикате start\_here f , методът start\_here се изпълнява. Вътре в този метод,

низът Hello е присвоен на x c - т.е. на *този* x, x в обхвата вътре в метода.

start\_here отпечатва своя x (Hello) и след това извиква метода say\_goodbye d . В

say\_goodbye, нещо подобно се случва: низ (Goodbye) се определя на х B . Но

б

° С

д

д

е

|  |
| --- |
| **Страница 55** |

**55**

***Локални променливи и присвояване на променливи***

това е различен х - както виждате, когато повикването към say\_goodbye приключи и контролира

връща се към start\_here: Ruby отпечатва този x и стойността все още е Hello e . Използване на x

като име на локална променлива в обхвата на един метод не е повлияло на стойността му в обхвата

на другия.

Локалните променливи в този последен пример се създават чрез изрично присвояване.

(Локалните променливи могат да се появят, както видяхте, чрез свързването на метода

аргументи към параметрите на метода.) Но какво се случва, когато заданието или

нация се извършва? Каква точно е връзката между променлива и обекта, който

представлява ли?

***2.5.1***

***Променливи, обекти и препратки***

Променливите задания придават вид и имат очевиден ефект на причиняване

променливата отляво да бъде зададена равна на обекта отдясно. След тази задача,

например,

str = "Здравей"

изявления като put str ще доставят низа "Hello" за печат и обработка.

Сега вижте този пример:

str = "Здравей"

abc = str

поставя abc

Това също отпечатва "Здравей". Очевидно променливата abc също съдържа "Hello", благодарение на

след като му беше назначена str.

Но има и още нещо. Следващият пример включва метод, наречен replace, който

прави заместване на съдържанието на низа на място с нов текст:

str = "Здравей"

abc = str

str.replace ("Сбогом")

поставя ул

поставя abc

def say\_goodbye

str = "Здравей"

abc = str

str.replace ("Сбогом")

поставя ул

поставя abc

край

кажи довиждане

Погледнете внимателно резултата:

Довиждане

Довиждане

Първото "Сбогом" е str; второто е abc. Но ние заменихме само str. Как се получи

низ в abc да бъде заменен?

|  |
| --- |
| **Страница 56** |

**56**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

Е НТЕР РЕФЕРЕНЦИИ

Отговорът е, че променливите в Ruby (с няколко изключения, най-вече променливите

обвързани с цели числа) не съдържат обектни стойности. str не съдържа "Hello". По-скоро ул

съдържа препратка към низ обект. Низовият обект има характеристиката

на съдържащи буквите, които съставляват "Здравей".

При задание с име на променлива отляво и обект отдясно,

променлива получава препратка към обекта. При задание от една променлива към

друга (abc = str), променливата вляво получава копие на препратката, съхранена в

променливата вдясно, в резултат на което и двете променливи вече съдържат препратки към

един и същ обект.

Фактът, че променливите съдържат препратки към обекти, има значение за операциите

които променят обектите. Операцията за замяна на низ

str.replace ("Сбогом")

замества символите на низа, към който str е препратка, с текста "Сбогом".

Променливата abc съдържа друга препратка към същия низ обект. Въпреки че

съобщението за замяна отива към str, то причинява промяна на обекта, към който се отнася

ence в abc се отнася. Когато разпечатвате abc, виждате резултата: съдържанието на

низ се промени.

Нереференцията: Незабавни стойности

Някои обекти в Ruby се съхраняват в променливи като непосредствени стойности. Те включват

цели числа, символи (които приличат на: this) и специалните обекти true, false и

нула. Когато присвоите една от тези стойности на променлива (x = 1), променливата се задържа

самата стойност, а не препратка към нея.

На практика това няма значение (и често ще бъде оставено по подразбиране, а не

изписани многократно, в дискусии на препратки и свързани теми в тази книга).

Ruby се справя автоматично с пренасочването на препратки към обекти; не е нужно

направете допълнителна работа, за да изпратите съобщение до обект, който съдържа, да речем, препратка към

низ, за ​​разлика от обект, който съдържа непосредствена целочислена стойност.

Но правилото за незабавно представяне на стойност има няколко интересни разклонения -

особено когато става въпрос за цели числа. От една страна, всеки обект, който представлява

възприема като непосредствена стойност винаги е абсолютно един и същ обект, без значение колко

променливи, на които е присвоено. Има само един обект 100, само един обект е фалшив и

скоро.

Непосредствената, уникална природа на целочислените променливи стои зад липсата на Ruby

оператори преди и след увеличаване - което означава, че не можете да направите това в Ruby:

x = 1

x ++ # Няма такъв оператор

Причината е, че поради непосредственото присъствие на 1 в x, x ++ ще бъде като 1 ++,

което означава, че ще промените числото 1 на число 2 - и това прави

няма смисъл.

|  |
| --- |
| **Страница 57** |

**57**

***Локални променливи и присвояване на променливи***

За всеки обект в Ruby може и трябва да има една или повече препратки към този обект.

Ако няма препратки, обектът се счита за несъществуващ и неговото пространство в паметта е

освободен и използван повторно.

Ако имате две или повече променливи, съдържащи препратки към един обект, можете

използвайте някоя от тях, на равна основа, за изпращане на съобщения до обекта. Референциите имат a

много-към-едно отношение към техните обекти. Но ако присвоите изцяло нов обект на

променлива, която вече се отнася до обект, нещата се променят.

***2.5.2***

***Препратки при присвояване и преназначаване на променлива***

Всеки път, когато присвоите променлива - всеки път, когато поставите име на променлива вляво от

знак за равенство и нещо друго вдясно - започвате от нулата: променливата е

избърсва се и се прави ново задание.

Ето нова версия на нашия по-ранен пример, илюстрираща тази точка:

str = "Здравей"

abc = str

str = "Сбогом"

поставя ул

поставя abc

Този път изходът е

Довиждане

Здравейте

Второто присвояване на str дава на str препратка към различен низ обект. ул

и ABC част компания в този момент. abc все още се отнася до стария низ (този, чийто

съдържанието е "Hello"), но str сега се отнася до различен низ (низ, чийто кон-

палатките са "Сбогом").

Първата версия на програмата промени един низ, но втората версия

има два отделни низа. След повторното й използване променливата str няма какво повече да прави

с обекта, за който се е позовал по-рано. Но повторното използване на str няма ефект върху abc, което

все още съдържа препратка към оригиналния низ.

ЗАБЕЛЕЖКА Примерите използват локални променливи, за да демонстрират какво прави и

не се случва, когато присвоите променлива, на която вече е присвоена.

Но правилата и поведението, които виждате тук, не са само за локални променливи.

Променливите за клас, глобални и екземпляри следват същите правила. (Така правят и т.нар

константи, които можете да присвоите повече от веднъж, колкото и да е странно!) Всички

тези категории идентификатори са *l-стойности* : те могат да служат като лява страна,

или цел, на задание. (Сравнете с, да речем, 100 = 10, което не успява, защото

100 не е стойност l.) И всички те се държат еднакво по отношение на това как

обвържете се с дясната им страна и какво се случва, когато използвате дадена

повече от веднъж.

Ruby променливите често се описват като етикети или имена на обекти. Това е полезно сравнение

ison. Кажете, че имате две имена за вашето куче. „Водя Фидо на ветеринар“ и „Взимам

|  |
| --- |
| **Страница 58** |

**58**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

Роувър до ветеринар ”се отнасят до същото животно. Но ако вземете ново куче и прехвърлите

наречете му Фидо, а след това обвързването от име към куче се е променило. Фидо и Роувър не

вече се отнасят до едно и също животно и името Fido няма повече връзка с

първото куче.

А новият Фидо дори не трябва да е куче; можеш да спреш да се обаждаш на кучето си

Фидо и вместо това започнете да използвате името на колата си. Същото е, когато правите x = 1 след-

намалено с x = "низ". Повторно използвате идентификатора x за изцяло нов клас

обект (по-скоро низ, отколкото Fixnum). За разлика от някои езици, Ruby не пише

променливи. Всяка променлива може да бъде свързана с всеки обект от всеки клас по всяко време.

Семантиката на препратките и (пре) заданието имат важни последици за

как се развиват нещата, когато извикате метод с аргументи. Какво означава методът

получавате? И какво може да направи методът с него?

***2.5.3***

***Препратки и аргументи на метода***

Нека се придържаме към пример, базиран на низове, защото низовете са лесни за промяна и проследяване.

Ето метод, който взема един аргумент:

def change\_string (str)

str.replace ("Ново съдържание на низ!")

край

След това създайте низ и го изпратете на change\_string:

s = "Оригинално съдържание на низ!"

низ на промяна

Сега разгледайте s:

поставя s

Изследването разкрива, че съдържанието на низа, за който се отнася s, се е променило:

Съдържание на нов низ!

Това ви казва, че вътре в метода change\_string на променливата str се присвоява ref-

отнесеност към низа, посочен също от s. Когато извикате метод с аргументи,

наистина търгувате с препратки към обекти. И след като методът задържа препратка

оттам, всички промени, които той прави в обекта чрез референцията, са видими, когато вие

изследвайте обекта чрез някоя от неговите препратки.

Ruby ви дава някои техники за защита на обектите от промяна,

ако желаете или трябва да го направите.

D ВЪЗСТАНОВЯВАНЕ И ЗАМРАЗЯВАНЕ НА ОБЕКТИ

Ако искате да защитите обектите от промяна в методите, към които изпращате

тях, можете да използвате метода dup, който дублира обект:

s = "Оригинално съдържание на низ!"

низ на промяна (s.dup)

поставя s

**Отпечатва „Оригинал**

**низ съдържание! "**

|  |
| --- |
| **Страница 59** |

**59**

***Локални променливи и присвояване на променливи***

Можете също да замразите обект, което му пречи да бъде подложена на по-нататъшна промяна:

s = "Оригинално съдържание на низ!"

s.freeze

низ на промяна

Имайте предвид, че няма подходящ метод за размразяване. Замразяването е завинаги.

За да завършите картината, има и метод, наречен клонинг. Много прилича на дуп. The

разликата е, че ако клонирате замразен обект, клонингът също е замразен - докато ако вие

дублиране на замразен обект, дубликатът не е замразен.

С тези инструменти в ръка - дублиране, клониране и замразяване - можете да защитите обектите си

срещу повечето операции за промяна на измами. Все пак някои опасности все още дебнат. Дори ако ти

замразяване на масив, все още е възможно да промените обектите *в* масива (ако приемем

те не са замразени):

>> числа = ["едно", "две", "три"]

=> ["едно", "две", "три"]

>> числа. замръзване

=> ["едно", "две", "три"]

>> числа [2] = "четири"

RuntimeError: не може да променя замразения масив

>> числа [2] .replace ("четири")

=>"четири"

>> числа

=> ["едно", "две", "четири"]

В този пример фактът, че масивът от числа е замразен, означава, че не можете да промените

масив B . Но низовете в масива не са замразени. Ако извършите операция по подмяна

на низа "три", като го превръща палаво в "четири"в , новото съдържание на

низът се разкрива, когато преразгледате (все още замразен!) масив d .

Внимавайте с препратките и не забравяйте, че препратката към обект вътре в

collection не е същото като препратка към колекцията. (Ще получите силно усещане за

колекции като обекти сами по себе си, когато ги разгледаме подробно в глава 9.)

Последният момент относно променливите - по-специално локалните променливи - включва тяхното физическо

прилика с извикванията на методи и как Ruby разбра какво имаш предвид, когато имаш

хвърлете обикновен, без украса идентификатор върху него.

***2.5.4***

***Локални променливи и нещата, които приличат на тях***

Когато Руби види обикновена дума, която седи там - идентификатор на гола дума, като s, билет,

put или потребителско\_име - то го интерпретира като едно от трите неща:

Локална променлива

Ключова дума

Извикване на метод

*Ключовите думи* са специални запазени думи, които не можете да използвате като имена на променливи. def е ключ-

дума; единственото нещо, за което можете да го използвате, е да стартирате дефиниция на метод. (Строго говорете-

ing, можете да подведете Ruby да назове метод def. Но ... добре ... не.) Ако също е a

ключова дума; много Ruby код включва условни клаузи, които започват с if, така че би

**RuntimeError: не може**

**промяна на замразения низ**

б

° С

д

|  |
| --- |
| **Страница 60** |

**60**

C ГЛАВА 2 ***Обекти, методи и локални променливи***

да бъде объркващо, за да позволи също използването на if като име на променлива. Последователност като if = 3

ще бъде трудно за Ruby да анализира.

Подобно на локалните променливи, извикванията на методи могат да бъдат обикновени думи. Видяхте няколко примера,

включително путове и печат. Ако извикването на метода включва аргументи в скоби - или

дори празни скоби - тогава е ясно, че това не е локална променлива. В други случаи

може да има някаква неяснота и Руби трябва да го разбере.

Ето как Ruby решава какво вижда, когато срещне обикновен идентификатор:

1

Ако идентификаторът е ключова дума, това е ключова дума (Ruby има вътрешен списък с тях

и ги разпознава).

2

Ако има знак за равенство (=) вдясно от идентификатора, това е локална променлива

претърпява задача.

3

В противен случай се приема, че идентификаторът е извикване на метод.

Ако използвате идентификатор, който не е нито едно от тези три неща, тогава Руби ще се оплаче

и спиране на изпълнението с фатална грешка. Съобщението за грешка, което получавате, когато това се случи

е поучително:

$ ruby ​​-e "x"

-e: 1: в `<main>': недефинирана локална променлива или метод' x 'за main: Object

(NameError)

Обърнете внимание, че Ruby не може да разбере дали сте мислили, че x е променлива или метод. То знае

че x не е ключова дума, но може да е някоя от другите две. Така че съобщението за грешка

включва и двете.

В този момент имате голям, нарастващ запас от знания за обектите и различните

ables и как са свързани. Ще обърнем следващата глава 3 към темата как да създадем

яде обекти по структуриран, мащабируем начин с класове.

***2.6***

***Обобщение***

Ние разгледахме много основания в глава 2. В тази глава видяхте

Как да създадете нов обект и да дефинирате методи за него

Основите на механизма за изпращане на съобщения, чрез който изпращате заявки

обекти за информация или действие

Няколко от важните вградени методи, с които се предлага всеки обект на Ruby:

object\_id, response\_to ?, и изпрати

Подробности за синтаксиса на списъците с аргументи на метода, включително използването на задължителни,

незадължителни и по подразбиране аргументи

Как работят локалните променливи и присвояването на променливи

Използването на Ruby на препратки към обекти и как се възпроизвеждат препратки, когато са множество

променливите се отнасят до един и същ обект

Написването на програма Ruby може да включва мислене за това как бихте могли да картографирате елементи на a

домейн (дори скромен едноличен домейн като „билет за събитие“) върху система от

обекти, така че тези обекти да могат да съхраняват информация и да изпълняват задачи. В същото

|  |
| --- |
| **Страница 61** |

**61**

***Обобщение***

времето е важно да не мислите твърде строго за връзката между обектите и

реалния свят. Обектно-ориентираните езици със сигурност предлагат силен компонент на реалния

световно моделиране; но Руби, в същото време, е изключително еластична в своите моделиращи съоръжения-

връзки - както можете да видите от това колко лесно е да се подобри поведението на даден обект. Началникът

Целта при проектирането на програма и обектите в нея е да се измисли система, която

работи и има вътрешна последователност.

И, разбира се, езикът предлага много възможности за разработване на програмни структури

тура. Създаването на обекти един по един, както направихме в тази глава, е малко повече от

върхът на айсберга. След това ще разширим дискусията експоненциално, като разгледаме как

за създаване на обекти на множество, по-автоматизирани основи, използвайки класове Ruby.

|  |
| --- |
| **Стр. 62** |

**62**

*Организиране на обекти*

*с класове*

Създаване на нов обект с Object.new - и оборудване на този обект със собствена meth-

os, един метод в даден момент - е чудесен начин да почувствате центрирането на обекта

на програмирането на Ruby. Но този подход не е точно мащабен; ако използвате

онлайн касата и вашата база данни трябва да обработва записи за билети от логото

dreds, трябва да намерите друг начин за създаване и манипулиране на подобни на билети обекти в

вашите Ruby програми.

Със сигурност Ruby ви предлага пълен набор от техники за програмиране за създаване

извличане на обекти на база партида. Не е нужно да дефинирате отделен ценови метод за

всеки билет. Вместо това можете да определите *клас* билет , проектиран по такъв начин, че

всеки отделен обект на билет автоматично има метод на цената.

***Тази глава обхваща***

Създаване на множество обекти с класове

Задаване и четене на състояние на обекта

Автоматизиране на създаването на четене на атрибут и

методи за писане

Механика на наследяване на клас

Синтаксис и семантика на константите на Ruby

|  |
| --- |
| **Страница 63** |

**63**

***Класове и екземпляри***

Дефинирането на клас ви позволява да групирате поведението (методите) в удобни пакети, така че

че можете бързо да създадете много обекти, които се държат по същество по същия начин. Можеш

добавете методи и към отделни обекти, ако това е подходящо за това, което се опитвате

направете във вашата програма. Но не е нужно да правите това с всеки обект, ако моделирате своя

домейн в класове.

Всичко, с което се справяте в Ruby, е или обект, или конструкция, която оценява

обект и всеки обект е екземпляр на някакъв клас. Този факт е валиден дори

където в началото може да изглежда малко странно. Целите числа са екземпляри на клас и класове

самите те са обекти. В тази глава ще научите как този широко разпространен аспект на

дизайнът на Ruby работи.

Говоренето за класове не означава, че не говорите за обекти; ето защо това

глава има заглавието, което има, вместо да кажем „Руби класове“. Голяма част от това, което ще разгледаме

тук се отнася до обекти и методи - но това е така, защото класовете са по същество начин да се

организира обекти и методи. Ще разгледаме видовете неща, които можете и ще направите

вътре в класовете, както и какви са самите класове.

***3.1***

***Класове и екземпляри***

Типичният клас се състои от колекция от дефиниции на методи. Класове обикновено съществуват за

целта да бъде *създаден екземпляр* - тоест да има създадени обекти, които са екземпляри

на класа.

Вече сте виждали екземпляр в действие. Това е нашата стара мелодия:

obj = Object.new

Обектът е вграден клас Ruby. Когато използвате точковото обозначение в клас, изпращате

съобщение до класа. Класовете могат да отговарят на съобщения, точно като обекти; всъщност, както и вие

имат основание да са наясно във всеки брой ситуации, класовете са обекти. Новото

method е *конструктор* : метод, чиято цел е да произвежда и връща към вас

нов екземпляр на класа, новоотсечен обект.

Вие дефинирате клас с ключовата дума class. Класовете се именуват с *константи* , спе-

тип идентификатор, разпознаваем по факта, че започва с главна буква. Con-

stants се използват за съхраняване на информация и стойности, които не се променят в течение на a

изпълнение на програма.

ПРЕДУПРЕЖДЕНИЕ Константите могат да се променят - те не са толкова постоянни, колкото името им

предполага. Но ако присвоите нова стойност на константа, Ruby отпечатва предупреждение.

Най-добрата практика е да избягвате да присвоявате нови стойности на константите, които сте имали

вече присвои стойност на. (Вижте раздел 3.7.2 за повече информация относно

преназначаване на константи.)

Нека дефинираме клас Ticket. В дефиницията на класа дефинираме един-единствен прост метод:

клас Билет

събитие def

„Все още не може да бъде посочено ...“

край

край

**Определете метода на събитието**

**за клас Билет**

|  |
| --- |
| **Страница 64** |

**64**

C ГЛАВА 3 ***Организиране на обекти с класове***

Сега можем да създадем нов билетен обект и да го попитаме (безсмислено, но за да видим процеса)

за да опише неговото събитие:

билет = Ticket.new

поставя билет.event

Извикването на метод ticket.event води до изпълнението на нашия метод на събитие и,

следователно отпечатването на (по-скоро неинформативен) низ, посочен вътре

този метод:

Все още не може да бъде посочено ...

Информацията е неясна, но процесът е напълно оперативен: написахме и изпълнихме

изрязан метод на екземпляр.

Имайки предвид какво, точно?

***3.1.1***

***Инстанционни методи***

Примерите за дефиниции на методи в глава 2 включват директно дефиниране на методи

върху отделни обекти:

def билет.event

Методът на събитията в предишния пример обаче е дефиниран по общ начин,

в класа на Ticket:

събитие def

Това е така, защото този метод на събитие ще бъде споделен от всички билети - тоест от всички инстанции

на Билет. Методи от този вид, определени в клас и предназначени за използване от всички

екземпляри на класа, се наричат *методи на екземпляра* . Те не принадлежат само на един обект.

Вместо това всеки екземпляр на класа може да ги извика.

ЗАБЕЛЕЖКА Методи, които дефинирате за един конкретен обект - като в def Tick

.price - наричат ​​се *единични методи* . Вече сте виждали примери и ние ще го направим

разгледайте по-задълбочено как работят сингъл методите в глава 13. Обект

който има метод за цена, не го интересува дали извиква сингълтън метод

или метод на екземпляр от своя клас. Но разграничението е важно от

перспектива на програмиста.

След като дефинирате метод на екземпляр в клас, нищо не ви спира да го дефинирате

отново - тоест, заменяне на първата дефиниция с нова.

***3.1.2***

***Отменящи методи***

Ето пример за дефиниране на един и същ метод два пъти в един клас:

клас С

def m

поставя "Първо определение на метод m"

край

|  |
| --- |
| **Страница 65** |

**65**

***Класове и екземпляри***

def m

поставя "Второ определение на метод m"

край

край

Като се имат предвид тези две дефиниции, какво се случва, когато извикаме m на екземпляр на C? Нека да

попитайте обекта:

C. new.m

Отпечатаният резултат е Второ определение на метод m. Второто определение има предварително

vailed: виждаме изхода от тази дефиниция, а не от първата. Когато замените a

метод, новата версия има предимство.

(Предишният пример е умишлено минималистичен, защото илюстрира някои

нещо, което обикновено не бихте направили точно в тази форма. Когато замените a

метод, обикновено защото сте написали клас, който наследява от първоначалния клас,

и искате да се държи по различен начин. Скоро ще разгледаме наследството.)

Можете също да добавите към методите на класа или да ги замените, като отворите отново класа

определение.

***3.1.3***

***Подновяване на класовете***

В повечето случаи, когато дефинирате клас, създавате единичен блок за дефиниция на клас:

клас С

# код на клас тук

край

Но е възможно да отворите отново клас и да направите допълнения или промени. Ето пример:

клас С

def x

край

край

клас С

def y

край

край

Отваряме тялото на дефиницията на класа, добавяме един метод (x) и затваряме тялото на дефиницията.

След това отваряме отново тялото на дефиницията, добавяме втори метод (y) и затваряме дефиницията

ционно тяло. Резултатът е същият, както ако бяхме направили това:

клас С

def x

край

def y

край

край

Тук отваряме класа само веднъж и добавяме и двата метода. Разбира се, няма да отидете

за да разбиете дефинициите на класа си на отделни блокове само за забавление. Трябва да има

синко - и това би трябвало да е основателна причина, защото разделящите дефиниции на клас могат да го направят

по-трудно за хората, които четат или използват кода ви, за да следят какво се случва.

|  |
| --- |
| **Страница 66** |

**66**

C ГЛАВА 3 ***Организиране на обекти с класове***

Една от причините да разбиете дефинициите на класове е да ги разпространявате в множество файлове. Ако

имате нужда от файл, който съдържа дефиниция на клас (може би го зареждате от диска на

време на изпълнение от друг файл и имате също частична дефиниция на същия клас в

файлът, от който се изисква вторият файл), двете дефиниции се обединяват. Това

не е нещо, което бихте направили произволно: това трябва да е случай, когато дизайнът на програмата

изисква класът да бъде дефиниран частично на едно място и частично на друго място.

Ето пример от реалния живот. Руби има клас „Време“. Позволява ви да манипулирате пъти, за-

ги матирайте за целите на клеймото за време и т.н. Можете да използвате формат за дата в стил UNIX

низове, за да получите желания формат. Например командата

поставя Time.new.strftime ("% m-% d-% y")

отпечатва низа "02-09-14", представляващ датата, на която е направено извикването на метода.

В допълнение към вградения клас Time, Ruby има и програмен файл, наречен time.rb,

вътре в които са различни подобрения и допълнения към класа Time. време.rb

постига целта си да подобри класа Time чрез повторно отваряне на този клас. Ако търсите

файла time.rb или в поддиректорията lib на изходното дърво на Ruby, или във вашия Ruby

инсталация, ще видите това на или близо до линия 87:

клас Време

Това е повторно отваряне на класа Time, направено с цел добавяне на нови методи.

Можете да видите ефекта най-добре, като го изпробвате в irb. irb ви позволява да извикате несъществуващ метод

без да карате сесията да се прекратява, за да можете да видите ефектите от изискването

команда всичко в една сесия:

>> t = Time.new

=> 2014-02-09 09:41:29 -0500

>> t.xmlschema

NoMethodError: недефиниран метод 'xmlschema' за 2014-02-09 09:41:29 -

0500: Време

от (irb): 2

от /Users/dblack/.rvm/rubies/ruby-2.1.0/bin/irb:11:in `<main>'

>> изисква 'време'

=> вярно

>> t.xmlschema

=>"2014-02-09T09: 41: 29-05: 00"

Тук изпращаме неразпознатото съобщение xmlschema до нашия обект Time, но това не става

работа B . След това зареждаме файла time.rb c и, разбира се, обектът Time вече има

метод xmlschema. (Този метод, съгласно неговата документация, „връща a

низ, който представлява времето като dateTime, дефинирано от XML схема. ")

Можете да разпространявате код за един клас върху множество файлове или върху множество местоположения

в същия файл. Но имайте предвид, че се счита за по-добра практика да не го правите кога

възможен. В случай на удължаване на времето, хората често предполагат възможността за унифициране

фикция: даване на обектите Time на всички методи за разширение на първо място, а не отделяне

класирайте тези методи в отделна библиотека. Възможно е такова обединение да отнеме

място в по-късно издание на Ruby.

б

° С

|  |
| --- |
| **Страница 67** |

**67**

***Променливи на инстанцията и състояние на обекта***

Ruby е за обекти, а обектите са екземпляри на класове. Ще разгледаме следващото

instance променливи, специална езикова функция, предназначена да позволява всеки екземпляр на всеки

клас в Ruby, за да зададете и поддържате собствена частна информация.

***3.2***

***Променливи на инстанцията и състояние на обекта***

Когато създавахме отделни обекти и пишехме методи за всяко действие или стойност ние

необходимо, ние кодирахме стойността в обекта чрез методите. С тази технология

nique, ако билетът струва $ 117,50, тогава той има метод, наречен цена, който връща предварително

чиста тази сума:

билет = Object.new

def билет.цена

117.50

край

Но сега се отдалечаваме от еднократното създаване на обекти с Object.new и

насочвайки вниманието си към практиката на проектиране на класове и създаване на много обекти

от тях.

Това означава, че променяме правилата на играта, когато става въпрос за информация

като цената на билета. Ако създадете клас на билет, не можете да му дадете ценови метод

което връща $ 117,50, по простата причина, че не всеки билет струва $ 117,50. Simi-

като цяло, не можете да дадете на всеки билет името на събитието Benefit Concert, нито на всеки билет

мисля, че е за ред G, седалка 33.

Вместо твърдо кодирани стойности във всеки обект, ние се нуждаем от начин да различаваме

обекти, които имат различни стойности. Трябва да можем да създадем нов билет

обект и съхранявайте с този обект информацията за събитието, цената и други

Имоти. Когато създаваме друг обект на билет, трябва да съхраняваме различна информация-

ция с този обект. И ние искаме да можем да правим това, без да се налага да правим a

метод със свойството, кодирано твърдо в него.

Информацията и данните, свързани с определен обект, олицетворяват *състоянието* на

обект. Трябва да можем да направим следното:

Задайте или нулирайте състоянието на даден обект (кажете на билет „Вие струвате $ 11,99.“).

Прочетете държавата (попитайте билет „Колко струвате?“).

Удобно е, че обектите Ruby се предлагат със собствен механизъм за съхранение и извличане на

стойности: *променливи на екземпляра* .

Променливата на екземпляра позволява на отделните обекти да запомнят състоянието. Вариант на инстанция

ables работят подобно на други променливи: присвоявате им стойности и ги четете

стойности обратно; можете да ги добавяте заедно, да ги разпечатвате и т.н. Но примерна вариация

ables имат няколко разлики:

Имената на променливите на инстанцията винаги започват с един знак @ (при знак). Това ви позволява да

разпознаване на променлива на екземпляр с един поглед.

Променливите на инстанцията са видими само за обекта, към който принадлежат. (Същество

„Видим за обект“ има техническа дефиниция, свързана със стойността по подразбиране

обект self, за което ще видите повече в глава 5.)

|  |
| --- |
| **Страница 68** |

**68**

C ГЛАВА 3 ***Организиране на обекти с класове***

Променлива на екземпляр, инициализирана в един метод в клас, може да се използва от всеки

метод, дефиниран в рамките на този клас.

Следващият списък показва прост пример, илюстриращ начина, по който присвоената стойност на

променлива на екземпляра остава жива от едно извикване на метод към друг.

клас Личност

def set\_name (низ)

поставя „Задаване на името на човек ...“

@ име = низ

край

def get\_name

поставя „Връщане на името на човека ...“

@ име

край

край

joe = Person.new

joe.set\_name ("Джо")

поставя joe.get\_name

Благодарение на заданието B, което се случва в резултат на повикването към set\_name c , когато

попитате за името на човека d , получавате обратно това, което сте сложили: "Джо". За разлика от местния

променлива, променливата на екземпляра @name запазва стойността, присвоена й дори след

методът, в който е инициализиран, е прекратен. Това свойство на варианта на

ables - оцеляването им при извиквания на методи - ги прави подходящи за поддържане на състояние

в обект.

Ще видите по-добри, по-идиоматични начини за съхраняване и извличане на стойности в обекти скоро.

Но всички те се основават на задаване и извличане на стойностите на променливите на екземпляра, така че

плаща, за да получите добро усещане за това как се държат променливите.

Сцената е настроена да направи нещо близко до полезно с нашия клас за билети. Липсващият

стъпка, която сега ще попълним, е процесът на инициализация на обекта.

***3.2.1***

***Инициализиране на обект със състояние***

Когато пишете клас (като Ticket), можете, ако желаете, да дефинирате специален метод

наречен инициализиране. Ако го направите, този метод ще се изпълнява всеки път, когато създавате

нов екземпляр на класа.

Например, даден метод за инициализиране, който отпечатва съобщение

клас Билет

def инициализира

поставя "Създаване на нов билет!"

край

край

ще видите съобщението "Създаване на нов билет!" всеки път, когато създавате нов билет

обект, като се обадите на Ticket.new.

Листинг 3.1 Променлива на екземпляр, поддържаща стойността си между извикванията на методите

б

° С

д

|  |
| --- |
| **Страница 69** |

**69**

***Променливи на инстанцията и състояние на обекта***

Можете да използвате този процес на автоматична инициализация, за да зададете състоянието на обекта в

време на създаване на обекта. Да кажем, че искаме да дадем на всеки обект на билет място и

дата, когато е създадена. Можем да изпратим правилните стойности като аргументи на Ticket.new,

и същите тези аргументи ще бъдат изпратени за инициализиране автоматично. Вътре се инициализира,

ще имаме достъп до информацията за мястото и датата и можем да запазим тази информация от

средства на променливи на екземпляра:

клас Билет

def инициализиране (място, дата)

@venue = място

@ дата = дата

край

Преди да затворим дефиницията на класа с end, трябва да добавим нещо друго: начин за

прочетете мястото и датата. Нека пуснем формулата get\_, с която използвахме

get\_name (в списък 3.1) и вместо това назовете методите get след варианта на

ables, чиито стойности връщат. Добавете този код (който включва крайната директива за

определение на клас) към предишните редове:

def място

@venue

край

дата на деф

@дата

край

край

Всеки от тези методи връща стойността на променлива на екземпляр. Във всеки случай това

променливата е последният (и единствен) израз в метода и следователно служи и като

връщаната стойност на метода.

ЗАБЕЛЕЖКА Имената на променливите, методите и аргументите на

инициализирайте не трябва да съвпадат. Можете да използвате @v вместо @venue, за

например да съхранява стойността, предадена в мястото на аргумента. Можете да се обадите на

втори метод event\_date и все още използвайте @date вътре в него. И все пак обикновено е добре

практикувайте да съчетаете имената, за да стане ясно какво се съчетава с какво.

Сега сме готови да създадем няколко билета с динамично зададени стойности за място и дата,

вместо твърдо кодираните стойности на нашите по-ранни примери:

th = Ticket.new ("Town Hall", "11/12/13")

cc = Ticket.new ("Convention Center", "12/13/14")

поставя „Създадохме два билета.“

поставя „Първият е за събитие # {th.venue} на # {th.date}.“

поставя „Второто е за събитие на # {cc.date} на # {cc.venue}.“

Изпълнете този код, заедно с предишната дефиниция на класа на Ticket, и ще видите

следното:

Създадохме два билета.

Първият е за събитие в кметството на 12.11.13.

Втората е за събитие на 13.12.14 г. в Convention Center.

|  |
| --- |
| **Страница 70** |

**70**

C ГЛАВА 3 ***Организиране на обекти с класове***

Фразата в Convention Center е малко стилна, но процесът на запазване и извличане

информацията за отделни обекти благодарение на променливите на екземпляра работи перфектно.

Всеки билет има свое собствено състояние (запазена информация), благодарение на това, което инициализираме

метод прави; и всеки билет ни позволява да го запитваме за мястото и датата, благодарение на двамата

методи с тези имена.

Това отваря изключително много нашите перспективи. Можем да създаваме, манипулираме, сравняваме и

изследвайте произволен брой билети едновременно, без да се налага да пишете отделни meth-

оди за всеки от тях. Всички билети споделят ресурсите на класа Ticket. В същото

време, всеки билет има свой собствен набор от променливи на екземпляра за съхраняване на информация за състоянието.

Досега сме подреждали нещата по такъв начин, че да задаваме стойностите на екземпляра

променливи в точката, в която е създаден обектът и след това може да извлече тези стойности в

всяка точка през живота на обекта. Това споразумение често е адекватно, но е така

не симетрични. Ами ако искате да зададете стойности за променливите на екземпляра в даден момент

различно от времето за създаване на обект? Ами ако искате да промените състоянието на обекта след него

вече е зададен веднъж?

***3.3***

***Методи на сетер***

Когато трябва да зададете или промените състоянието на даден обект в даден момент от вашата програма друго

от метода за инициализиране, сърцевината на въпроса е присвояване (или преназначаване) на вал

ues към променливите на екземпляра. Можете, разбира се, да промените стойността на всяка променлива в

всеки метод. Например, ако искахме билетите да имат възможност да ги отстъпят-

себе си, бихме могли да напишем метод на екземпляр като този в дефиницията на класа Ticket:

деф отстъпка (проценти)

@price = @price \* (100 - процента) / 100.0

край

Но най-често срещаният случай е най-простият: извикване на метод за задаване с аргумент

и задаване на подходящата променлива на екземпляра на аргумента. Това е, което set\_name

прави в примера за клас Person.

Има обаче още нещо. Ruby има някои специализирани конвенции за именуване на методи-

тиони, които ви позволяват да пишете методи за настройка по начин, който е по-елегантен от залепването на set\_

пред описателна дума като *име* . Този път ще направим още един пропуск на Ticket

с оглед на методите на сетер и наличните техники за тяхното рационализиране.

***3.3.1***

***Знакът за равенство (=) в имената на методи***

Да кажем, че искаме начин да определим цената на билета. Като отправна точка може да се определи цената

заедно с всичко останало по време на създаване на обект:

клас Билет

def инициализиране (място, дата, цена)

@venue = място

@ дата = дата

@ цена = цена

край

# и т.н.

|  |
| --- |
| **Страница 71** |

**71**

***Методи на сетер***

def цена

@цена

край

# и т.н.

край

th = Ticket.new ("Кметство", "12.11.13", 63.00)

Командата за инициализация обаче става ужасно дълга и изисква от нас

запомнете в какъв ред да сложите многото аргументи, за да не се получи a

билет, чиято цена е "Кметство". И все още нямаме начин да сменим билета

цена по-късно.

Нека решим проблема първоначално с метод set\_price, който ни позволява да зададем, или

нулиране, цената на съществуващ билет. Също така ще пренапишем метода за инициализиране, така че

не очаква фигура на цената:

клас Билет

def инициализиране (място, дата)

@venue = място

@ дата = дата

край

def set\_price (сума)

@ цена = сума

край

def цена

@цена

край

край

Ето някои манипулации на цените в действие:

билет = Ticket.new ("Кметство", "11/12/13")

ticket.set\_price (63.00)

поставя "Билетът струва $ # {"%. 2f "% ticket.price}."

ticket.set\_price (72.50)

поставя „Уфс - току-що се покачи. Вече струва $ # {"%. 2f "% ticket.price}."

Изходът е

Билетът струва $ 63,00.

Упс - току-що се покачи. Сега струва $ 72,50.

Тази техника работи: можете да напишете всички необходими методи set\_property и

екземпляр базирани на променливи методи за извличане, за да отидете с тях. Но има по-хубав начин.

СЪВЕТ Техниката за знак за проценти, която видяхте в последния пример, ви позволява

форматиране на низове, използвайки синтаксис, подобен на sprintf. Руби също има метод спринтф

(предлага се и с формата на името); бихме могли да пренапишем цената на билета

пример като sprintf ("%. 2f", ticket.price). Възможни спецификатори на формата (

% неща вътре в шаблона) включват% d за десетични числа,% s за

низове,% f за плувки и% x за шестнадесетични числа. Run ri sprintf

за пълна документация.

**Форматирайте цената в**

**два знака след десетичната запетая**

|  |
| --- |
| **Страница 72** |

**72**

C ГЛАВА 3 ***Организиране на обекти с класове***

Ruby ви позволява да дефинирате методи, които завършват със знак за равенство (=). Нека заменим

set\_price с метод, наречен цена = („цена“ плюс знак за равенство):

def цена = (сума)

@ цена = сума

край

price = прави точно това, което направи set\_price, и въпреки леко странния метод

име, можете да го извикате като всеки друг метод:

ticket.price = (63.00)

Знакът за равенство ви дава това познато усещане за „присвояване на стойност на нещо“, така че вие

знам, че си имаш работа със сетер метод. Все пак изглежда странно; но Руби взема

грижа и за това.

***3.3.2***

***Синтактична захар за методи, подобни на възлагане***

Програмистите използват термина *синтактична захар, за* да се отнасят до специални правила, които ви позволяват да пишете

кода си по начин, който не отговаря на нормалните правила, но това е по-лесно

не забравяйте как да направите и изглежда по-добре.

Ruby ви дава синтактична захар за извикване на методи за настройка. Вместо

ticket.price = (63.00)

можете да направите това:

билет.цена = 63,00

Когато интерпретаторът види тази последователност от кодове, той автоматично игнорира

интервал пред знака за равенство и чете цена = като единично съобщение цена = (повикване

към метода, чието име е цена =, което дефинирахме). Що се отнася до дясната ръка

страна, скобите не са задължителни за аргументите на метода, стига да няма

guity. Така че можете да поставите 63,00 там и то ще бъде взето като аргумент на

цена = метод.

Целта на включването на този специален синтаксис е да ви позволи да пишете

извиквания на методи, които приличат на задания. Ако току-що видях ticket.price = 63,00 в про-

грам, може да предположите, че ticket.price е някакъв вид l-стойност, към която стойността

63.00 се възлага. Но не е така. Цялото нещо е извикване на метод. Приемникът е

билет, методът е цена =, а единичният аргумент е 63,00.

Колкото повече използвате този метод на сетер, толкова повече ще оценявате как

много по-добре изглежда захарната версия. Този вид внимание към външния вид е типичен

на Руби.

Имайте предвид също, че методите за задаване могат да направят повече от просто присвояване на променливи.

***3.3.3***

***Развързани методи на сетер***

Възможността да пишете свои собствени = прекратени методи и фактът, че Ruby предоставя

синтактичният захарен начин на извикване на тези методи отваря някои интересни възможности.

|  |
| --- |
| **Страница 73** |

**73**

***Методи на сетер***

Една от възможностите е злоупотреба. Възможно е да се напишат = прекратени методи, които изглеждат така

те ще направят нещо, включващо задание, но не:

клас Глупаво

def цена = (x)

поставя „Текущото време е # {Time.now}“

край

край

s = Глупаво.ново

с. цена = 111,22

Този пример отхвърля аргумента, който получава (111.22), и отпечатва

закъсняло съобщение:

Текущото време е 2014-02-09 09:53:31 -0500

Този пример е умишлена карикатура. Но важното е: Руби проверява вашето

синтаксис, но не контролира вашата семантика. Имате право да пишете методи с имена

които завършват с = и винаги ще получавате синтаксис за присвояване на захар. Дали

Името на метода има смисъл във връзка с това, което методът прави, е във вашите ръце.

Методите за равен знак могат също да служат като филтри или пазачи. Да кажем, че искаме да зададем

цената на билета само ако цената има смисъл като сума в долари и центове. Ние можем

добавете интелигентност към метода цена =, за да осигурите коректността на данните. Тук,

ще умножим числото по 100, ще извадим всички останали числа с десетични знаци с

операцията to\_i (конвертиране в цяло число) и сравнете резултата с оригинала

число, умножено по 100. Това трябва да разкрие всички допълнителни десетични цифри извън

стотна колона:

клас Билет

def цена = (сума)

ако (сума \* 100) .to\_i == сума \* 100

@ цена = сума

друго

поставя "Цената изглежда неправилно"

край

край

def цена

@цена

край

край

Можете също да използвате този вид техника за филтриране, за да *нормализирате* данните - тоест да направите

уверете се, че определени данни винаги приемат определена форма. Да приемем например, че пътувате

уебсайт на агент, където потребителят трябва да въведе желаната дата на заминаване. Ти искаш

за да позволите както mm / dd / yy, така и mm / dd / yyyy.

Ако имате, да речем, Ruby CGI скрипт, който обработва входящите данни, бихте могли

нормализирайте годината, като напишете метод за задаване като този:

клас TravelAgentSession

def година = (y)

@ година = y.to\_i

|  |
| --- |
| **Страница 74** |

**74**

C ГЛАВА 3 ***Организиране на обекти с класове***

ако @ година <100

@ година = @ година + 2000

край

край

край

След това, ако приемем, че имате променлива, наречена дата, в която сте съхранили полето за дата

от формуляра (с помощта на библиотеката CGI на Ruby ) можете да получите компонентите на датата

като този:

месец, ден, година = date.split ('/')

self.year = година

Идеята е низът на датата да се раздели на три низа, като се използва наклонената черта (/) като

делител, с любезното съдействие на вградения метод на разделяне и след това за съхраняване на годишната стойност в

TravelAgentSession обект, използващ метода year = този обект.

ПРЕДУПРЕЖДЕНИЕ Методите за настройка не връщат това, което може да си мислите. Когато използвате

синтактичната захар, която ви позволява да се обаждате на = методи, които изглеждат като присвояване

Рубин приема сериозно семантиката на заданието. Задания (като x = 1)

оценявайте каквото е от дясната им страна. Методите обикновено връщат

стойност на последния израз, оценен по време на изпълнение. Но = извиквания на метод

дръжте се като задания: стойността на израза ticket.price = 63.00 е

63,00, дори ако методът ticket = връща низа „Ха ха!“. Идеята е да

поддържайте последователността на семантиката. Под капака е извикване на метод; но изглежда

като задание и се държи като задание по отношение на стойността му като

израз.

Понякога ще пишете сложни методи за получаване и задаване, но простият метод get и set

операциите, увити около променливи на екземпляра, са най-често срещаните - толкова често срещани,

всъщност, че Ruby ви дава някои преки пътища за писането им.

***3.4***

***Атрибути и семейството на методите attr\_ \****

Един *атрибут* е собственост на един обект, чиято стойност може да се прочете и / или писмено чрез

предметът. В случай на обекти на билети, бихме казали, че всеки билет има атрибут цена

както и атрибут за дата и атрибут на място. Нашата цена = метод може да бъде описан

като метод за *писане* на *атрибути* . дата, място и цена (без знака за равенство) са *атрибути*

методи за *четене на бутове* . (Терминологията за запис / четене е еквивалентна на термина

нология, използвана по-рано, но писането / четенето е по-често в дискусиите на Ruby.)

Атрибутите на обектите на Ruby са внедрени като методи за четене и / или запис

обгърнати около променливи на екземпляра - или, ако предпочитате, променливи на екземпляра, увити

в методите за четене и / или писане. Няма отделна конструкция „атрибут“ във фасада

ниво на измерване. *Атрибутът* е термин на високо ниво за определена конфигурация на методи и

инстанция променливи. Но това е полезен термин и Руби вгражда концепцията за

бутове на езика, под формата на преки пътища, които ви помагат да напишете методите, които

изпълнете ги.

**Работете с едно- или двуцифрена цифра**

**число чрез добавяне**

**век към него**

|  |
| --- |
| **Страница 75** |

**75**

***Атрибути и семейството на методите attr\_ \****

***3.4.1***

***Автоматизиране на създаването на атрибути***

Помислете за пълната картина на следващия списък на това, което имаме, като четец на атрибути

и / или методи за писане в нашия клас за билети. (Тук няма нищо ново; кодът е

просто да бъдат събрани на едно място.)

клас Билет

def инициализиране (място, дата)

@venue = място

@ дата = дата

край

def цена = (цена)

@ цена = цена

край

def място

@venue

край

дата на деф

@дата

край

def цена

@цена

край

край

Има един атрибут за четене / запис (цена) и два атрибута за четене (място и дата). То

работи, но кодът се повтаря. Три метода изглеждат така:

дефинирайте нещо

@ нещо

край

И има повторение на върха на повторението: не само че има три такива метода,

но всеки от тези три метода повтаря името си в името на променливата на екземпляра

то използва.

Всеки път, когато видите повторение в тази скала, трябва да се опитате да го отрежете - не чрез намаляване

какво прави вашата програма, но като намерите начин да изразите едно и също нещо по-кратко.

В преследването на тази лаконичност, Ruby е една крачка напред: предоставя вграден пряк път, който

автоматично създава метод, който чете и връща стойността на променливата на екземпляра

със същото име като метода (дайте или вземете @). Работи по следния начин:

клас Билет

attr\_reader: място,: дата,: цена

край

Елементите, които започват с двоеточие (: място и т.н.), са *символи* . Символите са a

вид съоръжение за именуване или етикетиране. Те са братовчед на струни, макар и не съвсем

същото нещо. Ще разгледаме символите по-задълбочено в глава 8. За настоящата ни цел

пози, можете да ги мислите като функционално еквивалентни на низовете.

Листинг 3.2 Клас на **билета** , с изписани методи за четене / запис на атрибути

|  |
| --- |
| **Страница 76** |

**76**

C ГЛАВА 3 ***Организиране на обекти с класове***

Методът attr\_reader (четец на атрибути) автоматично записва за вас вида

метод, който току-що разглеждахме. Има и метод attr\_writer:

клас Билет

attr\_writer: цена

край

С този единствен ред написахме (или по-точно Руби написа за нас) нашата цена = сетер

метод. Един ред заменя мястото на три. В случай на методите за четене, един ред

зае мястото на девет!

Цялата програма сега изглежда като следния списък.

клас Билет

attr\_reader: място,: дата,: цена

attr\_writer: цена

def инициализиране (място, дата)

@venue = място

@ дата = дата

край

край

Кодът в списъка 3.3 не само е по-кратък, но и по-информативен - самодокументиране,

дори. С един поглед можете да видите, че всеки обект на билет има място, дата и цена. The

първите два са четими атрибути и цената може да се чете или записва.

Можете дори да създавате методи за четене и запис с една команда.

C ъздаване ЧЕТЕЦ / WRITER ATTRIBUTES ATTR С \_ Accessor

В сферата на атрибутите на обекта атрибутите на комбинация четец / запис като цена са

често срещани. Ruby предоставя един-единствен метод attr\_accessor за създаване и на двата четеца

и метод на запис за атрибут. attr\_accessor е еквивалентът на attr\_reader

плюс attr\_writer. Можем да използваме тази комбинирана техника за цена, защото искаме

и двете операции:

клас Билет

attr\_reader: място,: дата

attr\_accessor: цена

# ... и т.н.

край

**самостоятелно** като приемник по подразбиране

Виждате повече извиквания на методи без изричен приемник; няма лява ръка

обект и без точка в attr\_reader, например. При липса на изричен приемник,

съобщенията отиват в себе си, обектът по подразбиране. В най-горното ниво на дефиниция на клас

тяло, себе си е самият обект на класа. Така че обектът, който получава съобщението attr\_reader

sage е действителният обект на клас Ticket. Ще навлезем по-задълбочено за класовете като

обекти и по този начин като получатели на съобщения по-късно в тази глава и по-задълбочено

за себе си в глава 5.

Листинг 3.3 Клас на **билета** , с методи за получаване и **задаване,** определени чрез повиквания **attr\_ \***

|  |
| --- |
| **Страница 77** |

**77**

***Наследяването и йерархията на класа Ruby***

Като алтернатива можете да постигнете функцията attr\_accessor с обикновения метод attr,

както следва:

attr: цена, вярно

Извикването на attr с true като втори аргумент задейства създаването и на двата четеца

и атрибути на писател, като attr\_accessor. Но attr\_accessor е по-ясен в своите намерения

- думата „достъп“ ви казва какво се случва - и тя също има предимството

че можете да му давате повече от едно име на достъп едновременно (докато attr отнема само

едно, плюс незадължителния верен аргумент). Без втория аргумент на true, attr

просто предоставя атрибут за четец.

***3.4.2***

***Резюме на методите attr\_ \****

Семейството от attr\_ \* методи е обобщено в таблица 3.1.

Във всички случаи техниките attr\_ имат ефект на писане на един или повече get и / или

задайте методи за вас. Те са мощен набор от преки пътища за кодиране.

Нека да намалим мащаба към по-широк изглед на класовете - по-конкретно към въпроса

*наследяване на* класа .

***3.5***

***Наследяването и йерархията на класа Ruby***

*Наследяването* е вид връзката надолу, свързана между два класа (супер-

клас и подклас), при което един клас „наследява“ от друг и екземплярите на

подкласът придобива поведението - методите - дефинирани в суперкласа.

Таблица 3.1 Резюме на семейството **attr\_ \*** на методите за създаване на getter / setter

Име на метода

Ефект

Пример

Еквивалентен код

attr\_reader

Създава четец

метод

attr\_reader: място

def място

@venue

край

attr\_writer

Създава писател

метод

attr\_writer: цена

def цена = (цена)

@ цена = цена

край

attr\_accessor

Създава четец и

писателски методи

attr\_accessor

:цена

def цена = (цена)

@ цена = цена

край

def цена

@цена

край

attr

Създава четец

и по желание a

метод на писател (ако

вторият спор-

мента е вярно )

1. attr: място

2. attr: цена, вярно

1. Вижте attr\_reader

2. Вижте attr\_accessor

|  |
| --- |
| **Страница 78** |

**78**

C ГЛАВА 3 ***Организиране на обекти с класове***

В този пример Magazine наследява от Publication. Обърнете внимание на синтаксиса в Magazine's

определение на класа:

клас Публикация

attr\_accessor: издател

край

клас Списание <Публикация

attr\_accessor: редактор

край

Символът <обозначава Списание като подклас на Публикация. Защото всеки

Катионният обект има методи publisher и publisher = (благодарение на attr\_accessor

: publisher), всеки обект на списание също има тези методи. Освен това списание

обектите имат редактор и редактор = методи:

mag = Magazine.new

mag.publisher = "Дейвид А. Блек"

mag.editor = "Джо Смит"

поставя „Mag е публикувано от # {mag.publisher} и редактирано от # {mag.editor}.“

Можем да продължим каскадата надолу:

клас Ezine <Списание

край

Екземплярите на Ezine имат атрибути на издател и редактор, както са дефинирани в

суперклас и супер-клас на Ezine. Имайте предвид, че не е задължително да добавяте нови

методи към всеки подклас. Може да искате да създадете клас Ezine само заради

да можете да се обадите на Ezine.new, а не на Magazine.new, за да направите кода си повече

изразителен.

Разбира се, не всичко е в методите за достъп на атрибути. Всеки метод на екземпляр вие

дефиниране в даден клас може да бъде извикан от екземпляри на този клас, а също и от екземпляри на

всякакви подкласове от този клас:

клас Личност

def видове

"Homo sapiens"

край

край

клас Рубист <Личност

край

david = Rubyist.new

поставя david.species

В този пример рубистката класа произлиза от Person c . Това означава даденост

Рубистки екземпляр, като например David, може да извика метода на видовете, дефиниран в

Лице за клас Б . Както винаги в Ruby, става въпрос за обекти: какво може и какво не може даден обект

направете в даден момент от програмата. Обектите получават поведението си от своите класове,

от техните индивидуални или единични методи, а също и от предците (суперклас,

супер-суперклас и т.н.) на техните класове (и от едно или две места, които не сме

б

° С

**Изход: Хомо**

**sapiens**

|  |
| --- |
| **Страница 79** |

**79**

***Наследяването и йерархията на класа Ruby***

погледна още). Като цяло обектите от Ruby водят интересен и динамичен живот. Наследяване

е част от тази картина.

Наследяването обаче има важно ограничение.

***3.5.1***

***Единично наследство: Едно на клиент***

В някои обектно-ориентирани езици е възможно даден клас да наследи от повече

отколкото един клас. Може например да имате клас на учител, който наследява от човек

клас и също наследява от клас Employee или клас Car, който наследява от Machine,

Задвижван и управляем. Ruby не позволява многократно наследяване; всеки клас Ruby може

имат само един суперклас, в съответствие с принципа на *единичното наследяване* .

Въпреки това, което може да е първото ви впечатление, единичното наследство на Руби не го прави

ограничават ви: Ruby предоставя *модули* , които представляват пакети от функционалност за програмиране

подобно на класовете (с изключение на това, че нямат екземпляри), върху които лесно можете да присадите

родословното дърво на вашия клас, за да предостави толкова методи за вашите обекти, колкото ви е необходимо.

(Глава 4 ще се съсредоточи върху модулите.) Няма ограничение за това колко богато можете да моделирате вашия

обекти - просто не може да се направи стриктно с класове и наследяване.

Принципът на единично наследяване означава, че не можете просто да нарисувате голямо дърво на

връзки и след това превежда дървото директно в йерархия на класа. Наследството често функционира-

по-скоро като удобен начин да получите два или повече класа за споделяне на дефиниции на методи

отколкото като окончателно изявление за това как обектите от реалния свят са свързани помежду си по отношение на

общност и конкретност. Има част от това, което участва; всеки клас в Ruby, за изпит-

ple, в крайна сметка се спуска (като подклас или подклас и т.н.) от класа Object,

и очевидно Object е по-общ клас от, да речем, String или Ticket. Но сингълът-

ограничението на наследството означава, че не можете да се ангажирате с проектирането на йерархия на класовете

тази каскада надолу по строг дървовиден график.

Отново модулите играят ключова роля тук и те ще получат дължимото си в глава 4. За

сега обаче ще следваме нишката на наследството нагоре, така да се каже, и ще разгледаме

класовете, които се появяват в горната част на дървото за наследяване на всеки обект Ruby:

Класове Object и BasicObject.

***3.5.2***

***Предмет на обекта и не толкова липсващата връзка: Класът Обект***

Виждали сте стандартната техника за създаване на общ обект:

obj = Object.new

Вече сте в състояние да разберете по-задълбочено какво се случва в този фрагмент.

Класът Обект е почти в горната част на диаграмата за наследяване. Всеки клас е или

подклас на обект, подклас на обект или на известно разстояние пряк потомък

на обект:

клас С

край

клас D <C

край

поставя D.superclass

поставя D.superclass.superclass

|  |
| --- |
| **Страница 80** |

**80**

C ГЛАВА 3 ***Организиране на обекти с класове***

Изходът е

° С

Обект

защото C е суперкласът на D (това правим ние), а Object е суперкласът на C (това е

Руби прави).

Ако се изкачите по веригата достатъчно далеч от който и да е клас, натиснете Object. Всеки метод

достъпна за гола инстанция на Обект е достъпна за всеки обект; тоест, ако можете

obj = Object.new

obj.some\_method

тогава можете да извикате some\_method за всеки обект.

Има обаче това „почти“. Както се оказва, на върха има още едно поколение.

***3.5.3***

***По-големият брат на El Viejo: BasicObject***

По-малкият брат на баща ми, сега 85-годишен прадядо, е известен на неговия

потомци като El Viejo: The Old Man. Това представи на моя братовчед загадка:

а именно как да обясни на малката си дъщеря - внучката на Ел Виехо - кой точно

баща ми беше, за първи път го срещна. В крайна сметка той хвана бика за рогата

и представи баща ми на неговата племенница като „по-големия брат на El Viejo“.

Класът BasicObject, както покойният ми баща по негово време, е по-стар от стария: идва

преди Object в родословното дърво на класа Ruby. Идеята на BasicObject е да предложи a

вид обект с празен лист - обект с почти никакви методи. (Всъщност прецедентът

за BasicObject беше библиотека от Джим Уейрих, наречена BlankSlate.) BasicObjects имат така

няколко метода, с които ще срещнете проблеми, ако създадете екземпляр BasicObject в irb:

>> BasicObject.new

(Обектът не поддържа #inspect)

Обектът се създава, но irb не може да покаже обичайното низово представяне на него

защото няма метод за проверка!

Новосъздаден екземпляр BasicObject има само 8 метода на екземпляр - докато a

нов екземпляр на Object има 55. (Тези числа могат да се променят малко при различните

версии или издания на Ruby, но те са достатъчно точни, за да се разбере

BasicObject има малко методи.) Не е вероятно да се наложи да създадете екземпляр или да подклас

BasicObject редовно, ако някога. Удобен е предимно за ситуации, в които сте

моделиране на обекти в близост до даден домейн, почти до точката на писане на a

вид рубински диалект и не искате никакви фалшиви положителни резултати, когато изпращате съобщения до

тези обекти. 55-те метода могат да ви попречат, ако имате собствени идеи за

дали вашите обекти трябва да играят тъпо, когато им изпращате съобщения като дисплей,

разширяване или клониране. (Ще има какво повече да кажем за това, когато се заемем с темата

нишка на BasicObject в глава 13.)

След като включихме наследството в комбинацията и разгледахме някои от ключовите компоненти на

родословието на обектите на Ruby, нека се върнем към темата на класовете - по-конкретно, към една

от най-поразителните аспекти на класовете: фактът, че те са обекти и следователно могат

служат като приемници на съобщения, точно както другите обекти.

|  |
| --- |
| **Страница 81** |

**81**

***Класове като обекти и получатели на съобщения***

***3.6***

***Класове като обекти и получатели на съобщения***

Класовете са специални обекти: те са единственият вид обект, който има силата да се появява

нови обекти (екземпляри). Въпреки това те са обекти. Когато създавате клас, като

Билет, можете да му изпращате съобщения, да добавяте методи към него, да го предавате на други обекти

като аргумент на метод и обикновено правите всичко, което бихте направили с друг обект.

Подобно на други обекти, класовете могат да бъдат създадени - наистина, по повече от един начин.

***3.6.1***

***Създаване на обекти на класа***

Всеки клас - обект, човек, билет - е екземпляр на клас, наречен клас. Както и вие

вече видян, можете да създадете обект на клас със специалната формула за ключова дума за клас:

клас Билет

# Вашият код тук

край

Тази формула е специална разпоредба на Ruby - начин да се направи лесно изглеждащ добре

достъпен блок с дефиниция на клас. Но можете също да създадете клас по същия начин, по който създавате

изяде повечето други обекти, като изпрати съобщението ново за обекта на класа Class:

my\_class = Class.new

В този случай на променливата my\_class се присвоява нов обект на клас.

Class.new съответства точно на други извиквания на конструктори като Object.new и

Ticket.new. Когато създадете екземпляр на класа Class, вие създавате клас. Този клас, в

от своя страна, може да създава свои собствени екземпляри:

instance\_of\_my\_class = my\_class.new

В раздел 3.1 видяхте, че обектите на класа обикновено са представени от константи (като

Билет или обект). В предходния сценарий обектът на клас е обвързан с обикновен

локална променлива (my\_class). Извикването на новия метод изпраща съобщението ново към класа

чрез тази променлива.

Дефиниране на методи на екземпляр във връзка с **Class.new**

Ако искате да създадете анонимен клас с помощта на Class.new и искате да добавите

екземплярни методи по време на създаването му, можете да го направите, като добавите кодов блок

след обаждането до нов. А *блок код* е фрагмент от код, който ви предоставят като част от

извикване на метод, което може да се изпълни от метода. Ще видите много повече за

кодови блокове, когато разгледаме итератори в глава 6. Междувременно, ето един малък изпит-

ple на Class.new с блок:

c = Class.new do

def say\_hello

поставя "Здравейте!"

край

край

Ако сега създадете екземпляр на класа (с c.new), ще можете да извикате

метод say\_hello за този екземпляр.

|  |
| --- |
| **Страница 82** |

**82**

C ГЛАВА 3 ***Организиране на обекти с класове***

И да, тук има парадокс ...

T HE КЛАС / ПРЕДМЕТ ПИЛЕ - ИЛИ - ЯЙЧЕН PARADOX

Класът Class е екземпляр от себе си - тоест той е обект на Class. И има още.

Помните ли класа Object? Е, Object е клас, но класовете са обекти. Така,

Обектът е обект. А Класът е клас. И Object е клас, и Class е обект.

Кое дойде първо? Как може да се създаде класът Class освен класът Object

вече съществува? Но как може да има обект на клас (или който и да е друг клас), докато няма

class Клас, на който може да има екземпляри?

Най-добрият начин да се справите с този парадокс, поне засега, е да го игнорирате. Руби трябва

направете някои от тези пилешки или яйчни неща, за да стартирате и стартирате системата от класове и обекти-

ning - и тогава кръгообразността и парадоксите нямат значение. В хода на програмата-

ming, просто трябва да знаете, че класовете са обекти, екземпляри на извикания клас

Клас. (Ако искате да разберете накратко как работи, това е така: всеки обект има

вътрешен запис за какъв клас е екземпляр и вътрешен запис вътре в

обект Клас сочи обратно към самия Клас.)

Класовете са обекти, а обектите получават съобщения и изпълняват методи. Как

точно ли се играе процесът на извикване на метод в случай на обекти от клас?

***3.6.2***

***Как обектите на класа извикват методи***

Когато изпращате съобщение до обект на клас, то изглежда така:

Ticket.some\_message

Или, ако сте в тяло с дефиниция на клас и класът играе ролята на

обект по подразбиране самостоятелно, изглежда така:

клас Билет

някакво\_съобщение

Така обектът на класа получава съобщения. Но откъде идват методите до

кои съобщения отговарят?

За да разберете къде класовете получават своите методи, помислете къде обектите в

eral получават своите методи (минус модули, които все още не сме изследвали):

От техния клас

От суперкласа и по-ранните предци от техния клас

От техния собствен магазин на единични методи („talk“ в def obj.talk)

По принцип ситуацията е същата за класовете. Има някои, но много малко, специални

калъфи или камбани за обекти от клас. Предимно те се държат като други предмети.

Нека разгледаме трите сценария за току-що изброено извикване на метод, в случай на

обекти от клас.

Екземплярите на Class могат да извикват методи, които са дефинирани като екземплярни методи в техните

клас. Билетът например е екземпляр на Class и Class определя екземпляр

метод, наречен нов. Ето защо можем да пишем

Ticket.new

**Като**

**"attr\_accessor"!**

|  |
| --- |
| **Страница 83** |

**83**

***Класове като обекти и получатели на съобщения***

Това се грижи за сценарий 1. Сега, сценарий 2.

Суперкласът на Class е Module. Следователно копията на Class имат достъп до

екземпляри методи, дефинирани в модул сред тях са семейството attr\_accessor на

методи. Ето защо можем да пишем

клас Билет

attr\_reader: място,: дата

attr\_accessor: цена

Тези извиквания на методи отиват директно към обекта на класа Ticket, който е в ролята на

обект по подразбиране самостоятелно в момента, в който се извършват повикванията.

Това оставя само сценарий 3: извикване на единичен метод на обект от клас.

***3.6.3***

***Единичен метод с друго име ...***

Ето един пример. Да приемем, че сме създали нашия клас за билети. Към този момент Ticket не е така

само клас, от който могат да възникнат обекти (екземпляри на билети). Билетът (класът) също е

обект сам по себе си. Както направихме с други обекти, нека добавим сингълтон

метод към него. Нашият метод ще ни каже кой билет от списък с обекти на билети е

Най-скъпият. Тук има код за черна кутия. Не се притеснявайте за подробностите;

основната идея е, че операцията max\_by ще намери билета, чиято цена е най-висока:

def Ticket.most\_expensive (\* билети)

tickets.max\_by (&: цена)

край

Сега можем да използваме метода Ticket.most\_expensive, за да разберем кой от няколко билета е

най-скъпото. (Ще избягваме да имаме два билета с една и съща цена, тъй като нашите

методът не се справя грациозно с тази ситуация.)

th = Ticket.new ("Town Hall", "11/12/13")

cc = Ticket.new ("Convention Center", "12/13/14 /")

fg = Ticket.new ("Fairgrounds", "13/14/15 /")

th.цена = 12.55

cc.цена = 10.00

fg.price = 18.00

най-висока = Ticket.most\_скъп (th, cc, fg)

поставя „Билетът с най-висока цена е този за # {най-висока.венуация}.“

Изходът е

Най-скъпият билет е този за Fairgrounds.

Методът most\_expensive се дефинира директно върху обекта на класа Ticket, в sin-

стил на gleton-метод. Обикновено се използва единичен метод, дефиниран за обект на клас

наричан *метод* на клас на класа, за който е дефиниран. Идеята за клас

Методът е, че изпращате съобщение на обекта, който е класът, а не на един от

екземплярите на класа. Съобщението most\_expensive отива в клас Ticket, а не в a

конкретен билет.

|  |
| --- |
| **Страница 84** |

**84**

C ГЛАВА 3 ***Организиране на обекти с класове***

Защо бихте искали да направите това? Не разваля ли основния ред - т.е.

създаване на билетни обекти и изпращане на съобщения до тези обекти?

***3.6.4***

***Кога и защо да се напише метод на клас***

Класовите методи служат за определена цел. Някои операции, отнасящи се до клас, не могат да бъдат изпълнени

образуван от отделни екземпляри от този клас. Новият метод е отличен изпит-

пле Обаждаме се на Ticket.new, защото докато не създадем индивидуален билет, не можем

изпращайте му всякакви съобщения! Освен това логично принадлежи работата на хайвера на нов обект

класа. Няма смисъл случаи на билети да се раждат помежду си. Но

има смисъл процесът на създаване на екземпляри да бъде централизиран като дейност на

клас Билет.

Друг подобен случай е вграденият Ruby метод File.open - метод, който, както

видяхте в глава 1, отваря файл за четене и / или писане. Отворената операция е

малко като нов; той инициира въвеждане и / или извеждане на файл и връща обект File. Това прави

смисъл за open да бъде метод на клас на File: искате създаването на

индивидуален обект от класа. Класът действа като отправна точка за

обекти, които създава.

Ticket.most\_expensive е различен случай, тъй като не създава нов обект—

но все пак това е метод, който логически принадлежи към класа. Намиране на най-скъпото

билет в списък с билети може да се разглежда като операция отгоре, нещо, което е

Терминът ***метод на класа*** : Повече проблеми, отколкото си струва?

Ruby позволява на обектите да имат единични методи, а класовете са обекти. Така че, когато го направите

def Ticket.most\_expensive, вие всъщност създавате сингълтон метод за

Билет. От страна на повикването, когато видите метод, извикан върху обект на клас - като

Ticket.new - не можете да разберете само като погледнете дали си имате работа с единичен

метод, дефиниран директно в този клас (def Ticket.new) или метод на екземпляр на

клас Клас.

Само за да бъде още по-забавно, класът Class има както версия на метода на класа на

нова и версия на метода на екземпляр; първият се извиква, когато пишете Class.new

и последното, когато пишете Ticket.new. Освен ако, разбира се, не го замените с определено

въвеждане на нови за себе си билет ...

Разбира се, ново е особено трънлив случай. Но като цяло терминът *клас метод*

не е непременно подходящ за Руби. Това е концепция, споделена с други обектно-ориентирани

езици, но в тези езици има по-голяма разлика между метода на класа

методи и екземпляри. В Ruby, когато изпращате съобщение до обект на клас, вие

не мога да кажа къде и как е дефиниран съответният метод.

Така че *методът на класа* има размито и рязко значение. Размито, всеки метод, който

се извиква директно върху обект на клас е метод на клас. Ясно е, че методът на класа е

дефиниран, а не само извикан, директно върху обект на Class. Ще чуете, че се използва и по двата начина, и

стига да сте наясно с основното инженерство и можете да направите рязкото

тинкции, когато трябва, ще се оправите.

|  |
| --- |
| **Страница 85** |

**85**

***Класове като обекти и получатели на съобщения***

направено колективно по отношение на билетите, вместо нещо, което се прави от индивид

обект на индивидуален билет. Писането на most\_expensive като метод на класа на Ticket ни позволява да запазим

методът в семейството на билетите, така да се каже, докато го приписва на абстрактния, надзор

сори ниво, представено от класа.

Не е нечувано да се създаде клас само с цел да му се даде метод

отс. Нашите предишни упражнения за преобразуване на температурата предлагат възможност за използване

този подход.

C ОБРЪЩАНЕ НА КОНВЕРТОРА

Нека конвертираме конвертора в конвертор клас, като добавим методи за преобразуване на класа

в двете посоки:

клас Температура

def Temperature.c2f (по Целзий)

Целзий \* 9,0 / 5 + 32

край

def Temperature.f2c (по Фаренхайт)

(по Фаренхайт - 32) \* 5 / 9,0

край

край

И нека го изпробваме:

поставя Temperature.c2f (100)

Идеята е, че имаме свързани с температурата полезни методи - съответни методи

към температурата като понятие, но не и до определена температура. Температурата

class е добър избор на обект, който да притежава тези методи. Бихме могли да получим по-красив и да имаме

Температурни случаи, които са знаели дали са Целзий или Фаренхайт и могат

да се преобразуват; но на практика, имайки Температурен клас с клас

методите за извършване на преобразуванията е адекватен и е приемлив дизайн. (Дори

по-добре, защото изобщо не ни трябват екземпляри за температура, би било да използваме

ule - един вид „без екземпляри“ клас, за който ще научите подробно в глава 4.)

Класовите методи и методите на екземпляра не се различават коренно един от друг;

всички те са методи и тяхното изпълнение винаги се задейства чрез изпращане на съобщение до

предмет. Просто обектът, който получава съобщението, може да е обект от клас. Все още там

са разлики и важни моменти, които трябва да имате предвид, когато започнете да пишете методи в

различни нива.

***3.6.5***

***Методи на класа срещу методи на екземпляр***

Като дефинирахме Ticket.most\_expensive, дефинирахме метод, до който можем да осъществим достъп

чрез обекта на класа Ticket, но не и чрез неговите екземпляри. Индивидуални обекти на билети

(екземпляри от билета за класа) нямат този метод. Можете лесно да тествате това. Опитвам

добавяйки това към кода от раздел 3.6.3, където променливата fg се отнася до билет

обект (за събитие на панаирната площадка):

поставя „Тестване на отговора на екземпляр на билет ....“

грешно = fg.most\_скъп

**Изходи 212.0**

|  |
| --- |
| **Страница 86** |

**86**

C ГЛАВА 3 ***Организиране на обекти с класове***

Получавате съобщение за грешка, тъй като fg няма метод, наречен most\_expensive. Класа

на fg, а именно Ticket, има такъв метод. Но fg, който е екземпляр на Ticket, не го прави.

Помня:

Класовете са обекти.

Екземплярите на класове също са обекти.

Обектът на класа (като Ticket) има свои собствени методи, собствено състояние и собствена идентичност.

*Тя не споделя тези неща със случаи на себе си.* Изпращането на съобщение до Ticket не е

същото като изпращането на съобщение до fg или cc или друг екземпляр на Ticket.

Ако някога се объркате какво е метод на клас и какво е метод на екземпляр,

обикновено можете да разрешите объркването, като се върнете към тези три принципа.

Обсъждането на класове винаги предполага използването на много константи - както и

предстоящо обсъждане на модули в глава 4. Така че нека да разгледаме по-задълбочено, отколкото имаме

досега какви са константите и как работят.

***3.7***

***Константи отблизо***

Много класове се състоят главно от методи на екземпляри и / или методи на класове. Но кон-

стойките са важна и често срещана трета съставка в много класове. Вече сте го направили

вижда константи, използвани като имена на класове. Константите също могат да се използват за задаване и предварително

обслужват важни стойности на данни в класове.

По-късно ще разгледаме обхвата на константите и техниките за влагането им вътре

многостепенни класове и модули. Засега ще се съсредоточим върху основите на начина на използване

тях - и въпроса колко постоянни са всъщност тези константи.

Бележка за нотация на метода

Когато пишете за и се позовавате на методите на Ruby (извън кода, т.е.), това е по поръчка-

ary да се позовава на методите на екземпляра, като назовава класа (или модула, според случая

be), в който са дефинирани, последвано от хеш знак (#) и името на метода;

и да се позовава на методи на клас с подобна конструкция, но използвайки точка вместо

знакът за хеш. Понякога ще видите двойно двоеточие (: :) вместо точка в

клас-метод случай.

Ето няколко примера за тази нотация и за какво се отнасят:

Цена на билет # се отнася до цената на метода на екземпляра в класа Ticket.

Ticket.most\_expensive се отнася до метода на класа най-скъпо в

клас Билет.

Ticket :: most\_expensive се отнася и до метода на класа most\_expensive in

клас Билет.

Отсега нататък, когато видите тази нотация (в тази книга или другаде), ще знаете какво

това означава. (Вторият пример - справка за метод на клас с точка) изглежда

същото като извикване на метода, но от контекста ще разберете дали е метод

обаждане или препратка към метода в дискусия.)

|  |
| --- |
| **Страница 87** |

**87**

***Константи отблизо***

***3.7.1***

***Основно използване на константи***

Името на всяка константа започва с главна буква. Много присвоявате на константите

както правите с променливите.

Да приемем, че решаваме да създадем списък с предварително дефинирани места за класа на билетите - а

списък, към който всеки обект на билет може да се позовава и да избира от него. Можем да присвоим списъка на

постоянство. Постоянните дефиниции обикновено отиват в или близо до върха на дефиниция на клас:

клас Билет

МЕСТА = ["Конгресен център", "Панаирни площадки", "Кметство"]

Константа, дефинирана в клас, може да бъде препратена от инстанцията или класа на класа

методи. Да предположим, че сте искали да сте сигурни, че всеки билет е за легитимен

място на провеждане. Можете да пренапишете метода за инициализация по следния начин:

def инициализиране (място, дата)

ако VENUES.include? (място)

@venue = място

друго

повишаване на ArgumentError, "Неизвестно място # {място на провеждане}"

край

@ дата = дата

край

Също така е възможно да се отнесе изцяло към константа извън определението на класа, използвайки

специална константна справочна нотация: двойно двоеточие (: :). Ето пример за настройка

константа вътре в клас и след това препратка към тази константа извън класа:

клас Билет

МЕСТА = ["Конгресен център", "Панаирни площадки", "Кметство"]

край

поставя "Затворихме дефиницията на класа."

поставя "Така че трябва да използваме нотация на пътя, за да достигнем константата."

поставя "Местата са:"

поставя Ticket :: VENUES

Нотацията с двойно дебело черво посочва постоянните ВЕНА в рамките на класа, познат от

постоянният билет и списъкът на местата се отпечатва.

Ruby се предлага с някои предварително дефинирани константи, до които можете да получите достъп по този начин

може да ви бъде полезно.

R UBY " S Предварително-дефинирани константи

Опитайте да напишете това в irb:

Математика :: PI

Математиката е модул (предмет на глава 4), но принципът е същият като в случая

на константа, дефинирана вътре в клас: използвате :: конектора, за да направите справка в

константа PI, дефинирана в математическия модул. Можете да търсите Е по същия начин.

Много от предварително дефинираните константи можете да разгледате, когато стартирате Ruby (или

irb) са имената на вградените класове: String, Array, Symbol и т.н. Някои

**Това ли е един от**

**известни места?**

**Повишете изключение**

**(фатална грешка - вижте**

**глава 6)**

**Съхранява местата като**

**масив от низове**

|  |
| --- |
| **Страница 88** |

**88**

C ГЛАВА 3 ***Организиране на обекти с класове***

са информационни; дори без зареждане на пакета rbconfig (който видяхте в

глава 1), можете да накарате тълкувателя да ви каже справедлива сума за неговите настройки.

Ето няколко примера:

>> RUBY\_VERSION

=>"2.1.0"

>> RUBY\_PATCHLEVEL

=> 0

>> RUBY\_RELEASE\_DATE

=>"25.12.2013"

>> RUBY\_REVISION

=> 44422

>> RUBY\_COPYRIGHT

=>"рубин - Авторско право (C) 1993-2013 Юкихиро Мацумото"

Както можете да видите, информацията, съхранявана в тези константи, съответства на информацията

получавате с превключвателя -v:

$ рубин -v

рубин 2.1.0p0 (2013-12-25 ревизия 44422) [x86\_64-darwin12.0]

Една особеност на константите на Ruby е, че те не са постоянни. Можете да ги промените,

в два смисъла на думата *промяна* - и в това се крие поучителен урок.

***3.7.2***

***Преназначаване срещу модифициране на константи***

Възможно е да извършите задание върху константа, към която вече сте задали

нещо - тоест да се пренасочи към константата. Но получавате предупреждение, ако направите това

(дори ако не работите с превключвателя на командния ред). Опитайте това в irb:

A = 1

A = 2

Ще получите следното съобщение:

(irb): 2: предупреждение: вече инициализирана константа A

(irb): 1: предупреждение: предишното определение на A беше тук

Фактът, че постоянните имена могат да се използват многократно, докато практиката на повторното им използване е предупреждение

евентуално нарушение представлява компромис. От една страна, това е полезно за езика

да има отделна категория за константи, като начин за съхраняване на данни, които остават видими

по-дълъг участък от програмата, отколкото обикновена променлива. От друга страна, Руби

е динамичен език, в смисъл, че всичко може да се промени по време на изпълнение. Engi-

търсенето на константи да бъдат изключение от това би било теоретично възможно, но

би въвела аномалия в езика.

Освен това, тъй като можете да презаредите вече заредени програмни файлове и да

gram файловете могат да включват постоянни задания, забраняващи преназначаване на константи

би означавало, че много операции за презареждане на файлове биха се провалили с фатална грешка.

Така че можете да преназначите на константа, но това не се счита за добра практика. Ако

ако искате идентификатор за многократна употреба, трябва да използвате променлива.

|  |
| --- |
| **Страница 89** |

**89**

***Природата срещу възпитанието в обекти от Рубин***

Другият смисъл, в който е възможно да се „промени“ константа, е чрез извършване на промени

към обекта, за който се отнася константата. Например, добавяне на място към билета

списъкът с местата на класа е лесен:

места = Билет :: МЕСТА

места <<"Гимназиален салон"

Няма предупреждение, защото няма предефиниране на константа. По-скоро сме

ifying на масив - и този масив няма конкретни познания, че е бил присвоен

до константа. Той просто прави това, което поискате.

Разликата между преназначаването на име на константа и модифицирането на ref

erenced от константата е важно и предоставя полезен урок от два вида

промяна в Ruby: промяна на картографирането на идентификатори на обекти (присвояване) и

промяна на състоянието или съдържанието на обект. С обикновени имена на променливи не сте

предупреждаван, когато правите преназначаване; но преназначаването все пак се различава от правенето

промени в обект, за всяка категория идентификатор.

Ако съберете темите в тази глава с някои от примерите, които сте

виждани по-рано, започвате да получавате добра цялостна картина за това как са обектите на Ruby

neered: те извличат своята функционалност от методите на екземпляра, дефинирани в техните

класове и предците на тези класове, но те също така са способни да „учат“ спе-

конкретни, индивидуализирани поведения под формата на сингълтон методи. Това е, което прави

Руби толкова очарователна. Животът на Ruby обект е поне потенциално смес от

обстоятелствата на неговото „раждане“ и чертите, които придобива през целия си живот. Ще приключим

тази глава с по-нататъшно изследване по тези важни линии.

***3.8***

***Природата срещу възпитанието в обекти от Рубин***

Връзката между класовете и техните екземпляри е по същество връзка между

общо и специфично - познат модел от света като цяло. Свикнали сме

виждайки животинското царство като цяло / специфични термини, а също и всичко от

музикални инструменти към университетските катедри към библиотечните стелажни системи за панте-

богове.

Доколкото езикът за програмиране ви помага да моделирате реалния свят (или,

обратно, че реалният свят ви предоставя начини за организиране на вашите програми), вие

може да направи по-лошо, отколкото да разчита в голяма степен на връзката общо към конкретно. Както можеш

вижте, наследяването - връзката суперклас към подклас - отразява общото / специфичното

съотношение отблизо. Освен това, ако се мотаете в обектно-ориентирани кръгове, ще вземете някои

стенография за тази връзка: фразата *е a* . Ако, да речем, Ezine наследява от Magazine,

казваме, че „едно списание е списание.“ По същия начин обект на Списание е Публикация, ако

Списанието наследява от Публикация.

Ruby ви позволява да моделирате по този начин. Можете да изкарате много километри, като премислите

вашия домейн като каскадна, базирана на наследство диаграма на обектите. Руби дори предлага

е? метод, който ви казва дали даден обект има даден клас или като свой клас, или като

един от родовите класове на своя клас:

**Използва << за добавяне на нов елемент**

**към съществуващ масив**

|  |
| --- |
| **Страница 90** |

**90**

C ГЛАВА 3 ***Организиране на обекти с класове***

>> mag = Magazine.new

=> # <Списание: 0x36289c>

>> mag.is\_a? (Списание)

=> вярно

>> mag.is\_a? (Публикация)

=> вярно

Организиране на класове в родословни дървета на свързани лица, с всяко поколение по малко

по-специфични от последните, могат да придадат приятно чувство за ред и детерминизъм

пейзажа на вашата програма.

Но Ruby обекти (за разлика от обектите в някои други обектно-ориентирани езици) могат да бъдат

индивидуално модифициран. Екземпляр на даден клас не е заседнал само с поведението

и черти, които класът му е придал. Винаги можете да добавяте методи към

обектна основа, както сте виждали в многобройни примери. Освен това класовете могат да се променят.

Възможно е обектът да придобие способности - методи - по време на живота си, ако е неговият клас

или родов клас придобива нови методи на екземпляр.

В езици, където не можете да добавяте методи към отделни обекти или към класове, които

вече са написани, клас на обект (и суперклас на този клас и т.н.

напред) ви казва всичко, което трябва да знаете за обекта. Ако обектът е

екземпляр на Magazine и сте запознати с методите, предоставени от класа

Списание за използването на неговите екземпляри, вие знаете точно как се държи обектът.

Но в Ruby поведението или възможностите на обекта могат да се отклоняват от тези,

пласиран от своя клас. Можем да направим списание покълнали крила:

mag = Magazine.new

def mag.wings

поставя "Вижте! Мога да летя!"

край

маг. крила

Това показва, че възможностите, с които е роден обектът, не са непременно възможностите

цялата история.

По този начин дървото за наследяване - възходящата каскада на класа към суперклас и супер-

суперклас - не е единственият определящ фактор за поведението на обекта. Ако искате да знаете

какво прави чисто нов предмет на списанието, погледнете методите в класа Magazine

и нейните предци. Ако искате да знаете какво може да направи обект от списание по-късно, трябва

знам какво се е случило с обекта от създаването му. (И да отговорите\_ на? —На

метод, който ви позволява предварително да определите дали даден обект знае как да борави с

конкретен метод - може да бъде полезен.)

Обектите Ruby са изключително гъвкави и динамични. Тази гъвкавост се превръща в

мощност на програмиста: можете да накарате списанията да летят, да накарате кравите да ви кажат кой е публикувал

тях, и всички останали. Както ясно показват тези глупави примери, силата предполага

отговорност. Когато правите промени в отделен обект - когато добавяте meth-

шансове за този обект и само този обект - трябва да имате основателна причина.

Повечето програмисти на Ruby са консервативни в тази област. Ще видите по-малко добавяне на

методи за отделни обекти, отколкото бихте очаквали. Най-често използваният случай за

**Резултат: Вижте!**

**Мога да летя!**

|  |
| --- |
| **Страница 91** |

**91**

***Обобщение***

добавянето на методи директно към обекти е добавянето на методи на клас към обекти на класа. The

по-голямата част от дефинициите на метода в стил singleton, които ще видите (def some\_object.some

\_method) ще бъдат дефиниции на метода на класа. Добавяне на методи към други обекти (

zines, билети, крави и т.н.) също е възможно, но трябва да го направите внимателно и

избирателно и с оглед на дизайна на програмата.

В повечето случаи, индивидуализация на обект (предмет на цялата глава 13, от

way) е свързано с динамично определени условия по време на изпълнение; например вие

може да добави методи за достъп към обекти, за да съвпадат имената на колоните на базата данни, които

не знаете, докато програмата не се стартира и не сте поискали базата данни. Или ти

може да има библиотека от специални методи, които сте написали за низови обекти, и това

искате да имат достъп само определени низове. Руби ви освобождава да правите тези неща,

защото класът на обекта е само част от историята - нейната същност, може да се каже, за разлика

за неговото възпитание.

И има още една част от пъзела: модули, конструкция на Ruby, която сте виждали

споменати тук няколко пъти мимоходом, които ще срещнете отблизо и задълбочено

следващата глава.

***3.9***

***Обобщение***

В тази глава сте научили основите на класовете Ruby:

Как писането на клас и след това създаването на екземпляри от този клас ви позволява да споделяте

поведения сред многобройни обекти.

Как да използваме методите на setter и getter, изписани или създадени автоматично

свързан със семейството методи attr\_ \*, за да се създадат атрибути на обекти, които се съхраняват

състояние на обект в променливи на екземпляра.

Като обекти, класовете могат да имат добавени методи към тях за всеки обект - такива

методи, които са широко известни като класови методи и осигуряват обща полезност

ity функционалност, свързана с класа.

Константите на Ruby са специален вид идентификатори, които обикновено се намират в класа (или

модул) дефиниции.

Наследяването е връзка клас-клас между суперклас и един или повече

подкласове и всички обекти на Ruby имат общ произход в Обекта и

Класове BasicObject.

Структурата на суперклас / подклас може строго да се поддаде на моделиращи обекти

йерархичен, таксономичен начин, но динамичните качества на обектите Ruby (включително

обекти от клас!) може да предложи по-малко строго определени начини на мислене за обектите

и как поведението им може да се развие в течение на живота им.

Този поглед върху класовете ви дава стабилна основа за разбиране как идват предметите

да съществуват и да се свързват помежду си в Ruby. След това ще надградим върху тази основа до

разглеждайки модулите, другият важен градивен елемент на обектната система.

|  |
| --- |
| **Страница 92** |

**92**

*Модули и*

*организация на програмата*

Тази глава ще ви запознае с конструкцията на Ruby, която е тясно свързана с класовете:

модули. Както подсказва името им, модулите насърчават модулния дизайн: програма

дизайн, който разделя големи компоненти на по-малки и ви позволява да смесвате и съчетавате

обектно поведение.

Подобно на класовете, модулите са пакети от методи и константи. За разлика от класовете,

модулите нямат екземпляри; вместо това посочвате, че искате да добавите функцията-

националност на определен модул спрямо този на клас или на конкретен обект.

Не случайно модулите приличат в много отношения на класовете: Класът

class е подклас на класа Module, така че всеки обект на клас е и модулен обект.

Първо обсъдихме класовете, защото Ruby е обектно-ориентиран, а обектите са екземпляри

класове. Но може да се каже, че модулите са по-основната структура, а класовете са

***Тази глава обхваща***

Капсулиране на поведението в модули

Модулно разширение на класовете

Пътят за търсене на метода на обекта

Обработка на грешка при търсене на метод

Създаване на пространства от имена с модули

и гнездене

|  |
| --- |
| **Страница 93** |

**93**

***Основи на създаването и използването на модула***

просто специализация. Изводът е, че и двамата са част от Ruby, и двете са

достъпни за вас, докато проектирате вашите програми и моделирате вашите данни.

Разглеждането на модули ни отвежда по-нататък по някои пътеки, по които частично сме минавали

предишна глава:

Видяхте, че всички обекти слизат от Object; тук ще срещнете ядрото

модул, който съдържа по-голямата част от методите, общи за всички обекти.

Научихте, че обектите търсят своите методи както в клас, така и в суперклас

пътя нагоре по наследственото дърво; в тази глава ще разгледаме значително

подробно как работи този процес на търсене на метод, когато и класове, и mod-

участват ules.

***4.1***

***Основи на създаването и използването на модула***

Писането на модул е ​​подобно на писането на клас, освен ако не започнете дефиницията си с

модулна ключова дума вместо ключова дума:

модул MyFirstModule

def say\_hello

поставя "Здравей"

край

край

Когато пишете клас, тогава създавате екземпляри на класа. Тези случаи могат да

сладки методи на екземпляра на класа. За разлика от тях, модулите нямат екземпляри. Вместо,

модулите се *смесват* с класове, използвайки или метода на включване, или метода на добавяне.

ЗАБЕЛЕЖКА prepend е ново в Ruby 2.0, докато include е част от Ruby

от началото.

Модулът, „смесен“ по този начин, понякога се нарича „смесване“. Резултатът

на смесване в модул е, че екземплярите от класа имат достъп до екземпляра meth-

дефинирани в модула.

Например, използвайки малкия модул от предишния пример, можете да продължите към

направите това:

клас ModuleTester

включва MyFirstModule

край

mt = ModuleTester.new

mt.say\_hello

Обектът ModuleTester извиква подходящия метод (say\_hello) и извежда

Здравейте. Забележете, че say\_hello не е дефиниран в класа, на който обектът е

инстанция. Вместо това се определя в модул, в който се смесва класът.

Операцията на смесване в този пример се постига с повикването за включване. Смесване

в модул силно прилича на наследяване от суперклас. Ако, да речем, клас Б

наследява от клас A, екземплярите от клас B могат да извикват методи на екземпляр от клас A. И ако,

да речем, клас С се смесва в модул М, екземплярите от клас С могат да извикват методи на екземпляр на модул М.

|  |
| --- |
| **Страница 94** |

**94**

C ГЛАВА 4 ***Модули и организация на програмата***

И в двата случая екземплярите на класа в долната част на списъка се възползват от предимствата: те

да извикват не само методите на екземплярите на собствения си клас, но и тези на (в един случай)

суперклас или (в другия случай) смесен модул.

Основната разлика между наследяването от клас и смесването в модул е

че можете да смесвате в повече от един модул. Никой клас не може да наследява от повече от един

клас. В случаите, когато искате множество допълнителни поведения за екземпляри на даден клас - и

не искате да ги скриете всички в суперкласа на класа и неговите родови класове - вие

може да използва модули, за да организира кода ви по по-подробен начин. Всеки модул може да добавя

нещо различно от методите, налични в класа. (Ще проучим

смесване срещу избор на наследяване по-нататък в раздел 4.4.1.)

Модулите отварят много възможности - особено за споделяне на код между повече

от един клас, тъй като произволен брой класове могат да се смесват в един и същ модул. Ще погледнем

следващите примери и ще разберете възможностите.

***4.1.1***

***Модул, капсулиращ „подобие на стека“***

Модулите ви дават начин за събиране и капсулиране на поведения. Типичен модулен кон-

поддържа методи, свързани с определена подмножина от това, което в крайна сметка ще бъде пълното

възможности на обект.

Като уточняваме това твърдение, ще напишем модул, който капсулира

характерно за това да бъде като стек или *подобност* на стека . След това ще използваме този модул за

придават на стека подобно поведение на обекти, чрез процеса на смесване на модула, подобен на стека

в един или повече класове.

Както може би знаете от предишни проучвания, *стекът* е структура от данни, която работи

последният принцип , първи излязъл ( LIFO ). Класическият пример е (физически) стек от

плочи. Първата използвана плоча е последната, поставена върху стека. Обикновено стековете са

обсъжда се в съчетание с опашки, които проявяват поведение първо в, първо в ( FIFO ). Мисля за

кафене: чиниите са в купчина; клиентите са на опашка.

Многобройни елементи се държат по начин, подобен на стека, LIFO . Последният лист принтер

хартията, която поставяте в тавата, е първата, отпечатана. Двойно паркираните коли трябва

оставете в ред, който е противоположен на реда на тяхното пристигане. Качеството на

като стек може да се прояви в голямо разнообразие от колекции и агрегации

на субектите.

Тук влизат модулите. Когато проектирате програма и се идентифицирате

поведение или набор от поведения, които могат да бъдат демонстрирани от повече от един вид субекти

или обект, намерихте добър кандидат за модул. Подобността на стека отговаря на сметката:

повече от един обект, и следователно въображаемо повече от един клас, показват стека

поведение. Чрез създаване на модул, който дефинира методи, в които се намират всички обекти, подобни на стека

обикновено, вие си давате начин да призовете подобност на стека във всички и всички класове, които

трябва ми.

Следващият списък показва проста реализация на stacklikeness в Ruby

модулна форма. Запазете този списък във файл, наречен stacklike.rb; ще заредите този файл по-късно

примери.

|  |
| --- |
| **Страница 95** |

**95**

***Основи на създаването и използването на модула***

модул Stacklike

деф стек

@stack || = []

край

def add\_to\_stack (obj)

stack.push (obj)

край

def take\_from\_stack

stack.pop

край

край

Модулът Stacklike в този списък използва *масив* (подредена колекция от обекти)

да представлява стека. Масивът се запазва в променливата на екземпляра @stack и

достъпни в метод комин на B . Този метод използва обща технология

nique за условно задаване на променлива: операторът || = (или-равно). Ефектът от

този оператор е да зададе променливата на определената стойност - която в този случай е нова,

празен масив - ако и само ако променливата не е настроена на нещо различно от нула

или невярно. На практика това означава, че първият стек за време се извиква, той ще се зададе

@stack до празен масив, докато при следващите повиквания ще види, че @stack вече

има стойност и просто ще върне тази стойност (масива).

Когато обект се добави към стека c , операцията се обработва чрез натискане на

обект върху масива @stack - тоест, добавяйки го в края. (достъп до @stack

чрез извикване на метода на стека, което гарантира, че той ще бъде инициализиран в

празен масив при първото добавяне на обект.) Премахване на обект от стека d

включва изскачане на елемент от масива - тоест премахването му от края. (натиснете

Списък 4.1 Модул, подобен на **стека** , капсулиращ структура и поведение на стека

Операторите за бърз достъп на Ruby

В допълнение към или-равно, Ruby има и друго семейство оператори за бърз достъп, подобно на

външен вид || = но проектиран малко по-различно. Тези оператори се разширяват до разговори

към основен метод. Общ е операторът + =; изразът a + = 1 е

еквивалентно на a = a + 1. Други членове на това семейство преки пътища включват - =, \* =, / =, \*\* =

(повишаване до степен), & = (побитово И), | = (побитово ИЛИ), ^ = (побитово ИЗКЛЮЧИТЕЛНО ИЛИ),% =

(по модул) и (рядко използван) и-равно оператор (&& =), който работи подобно на or-

се равнява. По този начин a - = 1 означава a = a - 1, a \* = 10 означава a = a \* 10 и т.н.

Всеки от тези оператори за опаковане на методи работи с всеки обект, който има съответния

основен метод, включително екземпляри от вашите собствени класове. Ако дефинирате метод +,

например можете да използвате синтаксиса x + = y за екземпляр на вашия клас (x) и

изразът ще бъде автоматично разширен до x = x + y. А това от своя страна е просто

тактическа захар за x = x. + (y), повикване към метода +.

Ще срещнете тези техники за пряк път „официално“ в глава 7. Междувременно от всички

означава да ги изпробвате в irb.

б

° С

д

|  |
| --- |
| **Страница 96** |

**96**

C ГЛАВА 4 ***Модули и организация на програмата***

и pop са методи на екземпляр на класа Array. Ще ги видите отново, когато погледнем

в контейнерни обекти, включително масиви, в глава 10.)

По този начин модулът Stacklike реализира подобно на стека чрез селективно разгръщане

поведения, които вече съществуват за Array обекти: добавете елемент в края на масива,

вземете елемент от края. Масивите са по-гъвкави от стековете; стек не може да направи

всичко, което може масив. Например можете да премахнете елементи от масив във всеки

ред, докато по дефиниция единственият елемент, който можете да премахнете от стека, е този

това беше добавено наскоро. Но масивът може да направи всичко, което може стека. Стига ние

не го молете да прави нещо неустановено, като използвате масив като вид агент или прокси за

специфичните за стека действия за добавяне / премахване имат смисъл.

Сега имаме модул, който реализира подобно на стека поведение: поддържане на списък с

елементи, така че да могат да се добавят нови до края и най-скоро добавения

могат да бъдат премахнати. Следващият въпрос е следният: Какво можем да направим с този модул?

***4.1.2***

***Смесване на модул в клас***

Както видяхте, модулите нямат екземпляри, така че не можете да направите това:

s = Stacklike.new

За да създадете екземпляри (обекти), имате нужда от клас и за да направите тези обекти подобни на стек,

ще трябва да смесите модула, подобен на стека, в този клас. Но какъв клас? Повечето

очевидно нещо подобно на стека вероятно е стек. Запазете кода в следния списък в

stack.rb, в същата директория като stacklike.rb.

require\_relative "stacklike"

клас Stack

включват Stacklike

край

Финансовият край на класа на стека в този списък е изявлението за включване B , което

има ефект на смесване в модула, подобен на стека. Той гарантира, че копията на Stack

показват поведението, дефинирано в Stacklike.

Листинг 4.2 Смесване на модула, **подобен** на **стека, в** класа на **стека**

Синтаксис на **require** / **load** спрямо синтаксис на **include**

Може да сте забелязали, че когато използвате изискване или зареждане, поставяте името на

артикул, който изисквате или зареждате в кавички; но с включване (и добавяне),

ти не го правиш. Това е така, защото изискването и зареждането вземат низове като техни аргументи,

като има предвид, че включва името на модул под формата на константа. Повече забавления-

по същество това е, защото изискването и зареждането са локализиране и зареждане на дискови файлове,

като има предвид, че включва и подготвя извършване на програмно пространство, операция в паметта, която

няма нищо общо с файловете. Често срещана е последователността да се изисква функция и след това

включва модул, който функцията дефинира. Поради това двете операции често вървят

заедно, но те са напълно различни един от друг.

**Грешно! Няма такъв метод.**

б

|  |
| --- |
| **Страница 97** |

**97**

***Основи на създаването и използването на модула***

Забележете, че името на нашия клас е съществително, докато името на модула е прилагателно.

Нито една от тези практики не е задължителна, но и двете са често срещани. Какво свършваме

с, изразено в ежедневния език, е вид предикат на класа: *Стек обекти*

*са подобни на стека.* Това е английски за

клас Stack

включват Stacklike

край

За да видим цялото нещо в действие, нека създадем обект на стека и го поставим през неговия

крачки. Кодът в следващия списък създава обект на стека и изпълнява някои операции

ции върху него; можете да въведете този код в края на вашия файл stack.rb.

s = Stack.new

s.add\_to\_stack ("елемент първи")

s.add\_to\_stack ("елемент втора")

s.add\_to\_stack ("елемент три")

поставя "Обекти в момента в стека:"

поставя s.stack

взето = s.take\_from\_stack

поставя „Премахна този обект:“

поставя взети

поставя „Сега в стека:“

поставя s.stack

Този списък започва с невинно изглеждащия (но мощен) екземпляр B на нов

Обект на стека, който се присвоява на променливата s. Този обект на стека се ражда с

знания за това какво да правим, когато го помолим да извършва действия, свързани със стека, благодарение на

фактът, че класът му се смесва в модула на Stacklike. Останалата част от кода включва

молете го да прескочи през някои обръчи, подобни на стека: добавяне на елементи (низове) към себе си c и

изскачане на последния от себе си d . По пътя ние молим обекта да докладва за състоянието му.

Сега, нека стартираме програмата. Ето извикване на stack.rb, заедно с out-

поставете от бягането:

$ ruby ​​stack.rb

Обекти в момента в стека:

т. първа

т. втора

т. трета

Премахна този обект:

т. трета

Сега в стека:

т. първа

т. втора

Разбира се, нашият малък обект на Stack знае какво да прави. Както е рекламирано, той е подобен на стека.

Класът Stack е добре, доколкото стига. Но може да ви накара да се чудите: защо го направихме

притеснявате се да пишете модул? В крайна сметка би било възможно да се опаковат всички функционалности на

модула, подобен на стека, директно в класа на стека, без да пише модул. Следващите

ниският списък показва как би изглеждал класът.

Листинг 4.3 Създаване и използване на екземпляр на клас **Stack**

б

° С

**Обаждането поставя масив**

**обаждания поставя върху всеки масив**

**елемент от своя страна**

д

|  |
| --- |
| **Страница 98** |

**98**

C ГЛАВА 4 ***Модули и организация на програмата***

клас Stack

attr\_reader: стек

def инициализира

@stack = []

край

def add\_to\_stack (obj)

@ stack.push (obj)

край

def take\_from\_stack

@ stack.pop

край

край

Както ще видите, ако добавите кода в списък 4.3 към списък 4.4 и го стартирате през Ruby,

той дава същите резултати като изпълнението, което използва модул.

Преди да заключите, че модулите са безсмислени, не забравяйте какво модуларизира-

купува ви: позволява ви да приложите обща концепция като подобност на стека към няколко случая, а не

само един.

И така, какво друго е подобно на стека?

***4.1.3***

***По-нататъшно използване на модула***

Няколко примера излязоха по-рано: чинии, хартия за принтер и т.н. Нека използваме нов

един, заимстван от света на градската легенда.

Много хора вярват, че ако сте първият пътник, който се регистрира за полет, вашият

багажът ще бъде последният извън самолета. Опитът от реалния свят предполага, че не е така

работи по този начин. И все пак, за практиката на стека, нека видим какъв Ruby модел на градско-легендарно-

правилното задържане на товара би изглеждало.

За да го моделираме разумно отблизо, ще дефинираме следното:

Клас на куфара без кости: резервоар (или мъниче), който ни позволява да създадем куфар

предмети, които да хвърлят в товарния отсек.

Клас CargoHold с два метода: load\_and\_report и разтоварване.

- load\_and\_report отпечатва съобщение, което съобщава, че добавя куфар към

товарен багаж и той ни дава идентификационния номер на куфара , който ще ни помогне

проследете какво се случва с всеки куфар.

Разтоварване на повиквания take\_from\_stack -. Бихме могли да извикаме take\_from\_stack директно, но

*разтоварването* звучи по-скоро като термин, който може да използвате, за да опишете премахването на костюм

кутия от товарен отсек.

Поставете кода в следващия списък в cargohold.rb и го стартирайте.

require\_relative "stacklike"

клас Куфар

край

Листинг 4.4 Немодулно пренаписване на класа **Stack**

Листинг 4.5 Използване на модула **Stacklike** за втори път, за различен клас

**Метод на конструктора:**

**@stack все още не може да има**

**е инициализиран, така че е зададен**

**като се използва =, а не || =**

|  |
| --- |
| **Страница 99** |

**99**

***Модули, класове и търсене на методи***

клас CargoHold

включват Stacklike

def load\_and\_report (obj)

печат "Зареждане на обект"

поставя obj.object\_id

add\_to\_stack (obj)

край

def разтоварване

take\_from\_stack

край

край

ch = CargoHold.new

sc1 = Куфар.нови

sc2 = Куфар.нови

sc3 = Куфар.нови

ch.load\_and\_report (sc1)

ch.load\_and\_report (sc2)

ch.load\_and\_report (sc3)

first\_unloaded = ch.unload

print "Първият куфар от самолета е ...."

поставя first\_unloaded.object\_id

В основата си програмата в този списък не е толкова различна от тази в списъците 4.2

и 4.3 (които сте запазили постепенно в stack.rb). Следва почти същата процедура

dure: смесване на Stacklike в клас B , създаване на екземпляр на този клас f и добавяне на

поставяне на елементи в d и премахването им от e този екземпляр (нещо подобно на стека -

багаж, в този случай). Той също така докладва за текущото състояние на стека c ,

както направи другата програма.

Резултатът от програмата за задържане на товари изглежда така (не забравяйте, че куфарите

са посочени от техните идентификационни номера на обекти , които могат да се различават във вашата система):

Зареждане на обект 1001880

Зареждане на обект 1001860

Зареждащ обект 1001850

Първият куфар извън самолета е .... 1001850

Примерът за задържане на товар показва как можете да използвате съществуващ модул за нов клас.

Понякога си струва да обгърнете методите с нови методи с по-добри имена за новите

домейн (като разтоварване вместо take\_from\_stack), макар че ако се окажете

променяйки се твърде много, това може да е знак, че модулът не е подходящ.

В следващия раздел ще съберем няколко от парчетата, които разгледахме повече или

по-малко отделно: извиквания на методи (изпращане на съобщения), обекти и състоянието им като екземпляри на

класове и смесването на модули в класове. Всички тези концепции се обединяват

процесът, при който обект, след изпращане на съобщение, търси и намира (или

не успява да намери) метод за изпълнение, чието име съвпада със съобщението.

***4.2***

***Модули, класове и търсене на методи***

Вече знаете, че когато даден обект получи съобщение, предвидената (и обичайна)

резултатът е изпълнението на метод със същото име като съобщението в обекта

б

° С

д

д

е

|  |
| --- |
| **Страница 100** |

**100**

C ГЛАВА 4 ***Модули и организация на програмата***

клас или суперклас на този клас - и нататък, до Object или дори BasicObject

клас - или в модул, който е бил смесен с някой от тези класове. Но как става това

ами И какво се случва в двусмислени случаи - например, ако клас и a

смесеният модул и двете дефинират метод с дадено име? Кой прави

обект да изберете да се изпълни?

Струва си да отговорите точно на тези въпроси. Неточни сметки за случилото се са

лесно достъпен. Понякога дори са адекватни: ако кажете: „Този ​​обект има тласък

метод “, може да успеете да предадете това, което се опитвате да комуникирате,

въпреки че обектите не разполагат с методи, а по-скоро ги намират чрез търсене на класове

и модули.

Но неточната сметка няма да се мащабира. Това няма да ви помогне да разберете какво става

в по-сложни случаи и няма да ви подкрепи, когато проектирате сами

код. Най-добрият начин на действие е да научите какво наистина се случва, когато изпращате съобщение

мъдреци към предмети.

За щастие начинът, по който работи, се оказва ясен.

***4.2.1***

***Илюстрира основите на търсенето на методи***

В интерес на работата за ясно разбиране на това как обектите намират методи,

нека направим обратна връзка с реалните справки и вместо това напишем някои класове и

ules с прости имена като C и M. Това ще ви помогне да се концентрирате върху логиката

и механика на търсене на методи, без да се налага да мислим едновременно за

Eling домейн от реалния свят. Ще напишем и някои методи, които не правят нищо

с изключение на отпечатайте съобщение, което съобщава, че са били извикани Това ще помогне за проследяването на

ред на търсене на метод.

Погледнете програмата в следващия списък.

модул М

def доклад

поставя метод "'report' в модул M"

край

край

клас С

включват М

край

клас D <C

край

obj = D.нови

obj.report

Отчетът за метода на екземпляра е дефиниран в модул М. Модул М се смесва в клас C.

Клас D е подклас на C, а obj е екземпляр на D. Чрез тази каскада обектът

(obj) получава достъп до метода на отчета.

И все пак, *получаването на достъп до* метод, както *има* метод, е неясен начин да се изрази. Нека се опитаме да

получите повече корекция на процеса, като разгледате погледа на обекта от него.

Листинг 4.6 Демонстрация на включване и наследяване на модул

|  |
| --- |
| **Страница 101** |

**101**

***Модули, класове и търсене на методи***

A N ОБЕКТ " S - EYE ОГЛЕД НАЧИН LOOKUP

Вие сте обектът и някой ви изпраща съобщение. Трябва да разберете как да

отговорете на него - или дали изобщо *можете да* отговорите на него. Ето малко поток от обекти-

на съзнание:

Аз съм обект на Ruby и ми беше изпратено съобщението „report“. Трябва да се опитам да намеря

метод, наречен отчет в моя път за търсене на метод. доклад, ако съществува, се намира в a

клас или модул.

Аз съм екземпляр на клас, наречен D. Клас D дефинира ли доклад за метод на инстанция?

*Не.*

D смесва ли се в някакви модули?

*Не.*

Суперкласът на D, C, дефинира ли метод на екземпляр на отчет?

*Не.*

Смесва ли С в някакви модули?

*Да,* М.

М дефинира ли метод на отчет?

*Да.*

Добре! Ще изпълня този метод.

Търсенето завършва, когато методът, който се търси, бъде намерен или с грешка

dition, ако не е намерен. Условието за грешка се задейства от специален метод, наречен

method\_missing, който се извиква като последна мярка за иначе несравними съобщения.

Можете да замените method\_missing (т.е. дефинирайте го наново в един от вашите собствени класове или

модули), за да дефинирате персонализирано поведение за такива съобщения, както ще видите подробно в сек.

4.3.

Нека да преминем сега от обектния поток на съзнание към спецификите за метода-

търсене, и по-специално въпроса докъде може да стигне.

H OW FAR ли метод на търсене GO ?

В крайна сметка всеки обект в Ruby е екземпляр на някакъв клас, произлязъл от големия

клас в небето: BasicObject. Колкото и класове и модули да пресича

между другото, търсенето на метод винаги може да стигне до BasicObject. Но припомнете

че целият смисъл на BasicObject е, че има няколко метода на екземпляра. Достигане до

know BasicObject не ви казва много за по-голямата част от методите, които всички Ruby

обекти споделят.

Ако искате да разберете общото поведение и функционалност на всички Ruby

обекти, трябва да слезете от облаците и да погледнете Object, а не Basic-

Обект. По-точно трябва да погледнете Kernel, модул, в който се смесва Object.

Именно в Kernel (както подсказва името му) повечето от основните методи на Ruby обектират

са определени. И тъй като Object се смесва в ядрото, всички екземпляри на Object и всички

потомците на Object имат достъп до методите на екземпляра в Kernel.

Да предположим, че сте обект и се опитвате да намерите метод, на който да се изпълни

съобщение, което сте получили. Ако сте търсили в Kernel и BasicObject и вие

|  |
| --- |
| **102** |

**102**

C ГЛАВА 4 ***Модули и организация на програмата***

не сте го намерили, няма да го направите. (Възможно е да се смесват модули в BasicObject,

като по този начин предоставя на всички обекти допълнителен потенциален източник на методи. Трудно е да се мисли

в случай, в който бихте направили това.)

Фигура 4.1 илюстрира пътя за търсене на метод от нашия по-ранен пример (клас D

обект) чак нагоре по стълбата. В примера търсенето на метода е успешно

в модул М; фигурата показва докъде би изглеждал обектът, ако не намери

метод там. Когато съобщението x е изпратено до обекта, търсенето на метода започва,

удряне на различните класове и смеси (модули), както е показано със стрелките.

Вътрешните дефиниции на BasicObject, Object и Kernel са написани в C

език. Но можете да получите разумна справка за това как си взаимодействат, като разгледате

Руби макет на техните отношения:

клас BasicObject

# тук има оскъдни седем дефиниции на методи

край

модул Ядро

# над 100 дефиниции на методи отиват тук!

край

клас Object <BasicObject

# един или два частни метода отиват тук,

# но основната точка е да се смесва в модула Kernel

включва ядрото

край

клас Обект

(

)

*вграден*

включва ядрото

модул Ядро

(

)

*вграден*

клас D <C

включват М

край

клас BasicObject

(

)

*вграден*

клас С

край

обект = D.нови

**обект.x**

модул М

def x

край

край

Фигура 4.1 Екземпляр на клас **D**

търси метод **x** в неговия метод

път за търсене.

|  |
| --- |
| **Страница 103** |

**103**

***Модули, класове и търсене на методи***

Обектът е подклас на BasicObject. Всеки клас, който няма изрично супер-

class е подклас на Object. Можете да видите доказателства за това по подразбиране в irb:

>> клас С

>> край

=> нула

>> C. суперклас

=> Обект

Всеки клас има Object - и следователно Kernel и BasicObject - сред своите предци -

торс. Разбира се, все още има парадоксът, че BasicObject е обект, а Object е

клас, а класът е обект. Но както видяхте по-рано, малко циркулярност в класа

модел служи за стартиране на йерархията; и веднъж пуснат в действие, той работи логично

и чисто.

***4.2.2***

***Дефиниране на един и същ метод повече от веднъж***

В глава 3 научихте, че ако дефинирате метод два пъти в един и същ клас,

второто определение има предимство пред първото. Същото важи и за модулите. The

правило се свежда до това: може да има само един метод от дадено име за клас или

модул по всяко време. Ако имате метод, наречен calcu\_interest във вашия

Клас BankAccount и вие пишете втори метод, наречен Calcu\_interest в

същия клас, класът забравя всичко за първата версия на метода.

Ето как се поддържат класовете и модулите. Но когато се обърнем към окото на обекта

изглед, въпросът за достъп до два или повече метода с едно и също име

става по-ангажиран.

Методите на обекта могат да идват от произволен брой класове и модули. Вярно, всякакви

един клас или модул може да има само един метод calcu\_interest (за да се използва това

име като пример). Но обектът може да има множество методи за изчисляване на лихвата

в своя път за търсене на метод, тъй като пътят за търсене на метод преминава през множество

класове или модули.

И все пак правилото за обекти е аналогично на правилото за класове и модули: обект

може да види само една версия на метод с дадено име по всяко време. Ако

методът за търсене на метод на обекта включва два или повече едноименни метода, първият

срещнат е „победителят“ и се изпълнява.

Следващият списък показва случай, когато две версии на метод лежат върху обект

път за търсене на метод: един в класа на обекта и един в модул, смесен от

този клас.

модул InterestBearing

def изчисли\_лихва

поставя "Placeholder! Ние сме в модул InterestBearing."

край

край

клас BankAccount

включват InterestBearing

Листинг 4.7 Два едноименни метода в един път за търсене

|  |
| --- |
| **104** |

**104**

C ГЛАВА 4 ***Модули и организация на програмата***

def изчисли\_лихва

поставя "Placeholder! Ние сме в клас BankAccount."

поставя „И ние отменяме метода на Calcu\_\_interest ...“

поставя "което беше дефинирано в модула InterestBearing."

край

край

акаунт = BankAccount.new

account.calculate\_interest

Когато стартирате този списък, получавате следния изход:

Placeholder! Ние сме в клас BankAccount.

И ние заместваме метода calcu\_interest ...

което беше дефинирано в модула InterestBearing.

Два метода Calcu\_\_interest лежат на пътя за търсене на метод на обект c. Но

търсенето удря класа BankAccount (клас на акаунта), преди да удари модула Лихва-

Лагер (смесване от клас BankAccount). Следователно методът на отчета, който изпълнява, е

дефинираната в BankAccount.

Обектът може да има два метода с едно и също име в пътя си за търсене на методи

при друго обстоятелство: когато клас смесва два или повече модула, повече от един

реализира търсения метод. В такъв случай модулите се търсят в

обратен ред на включване - т.е. първо се търси най-скоро смесеният модул.

Ако последният смесен модул случайно съдържа метод със същото име

като метод в модул, който е бил смесен по-рано, версията на метода в

ново смесеният модул има предимство, защото по-новият модул е ​​по-близо до

метод за търсене на метод на обекта.

Например, помислете за случай, когато два модула, M и N (ще запазим този пример

относително схематично), и двете дефинират метод на отчет и и двете са смесени в клас, C,

както в следващия списък.

модул М

def доклад

поставя метод "'report' в модул M"

край

край

модул N

def доклад

поставя "'report' метод в модул N"

край

край

клас С

включват М

включват N

край

Какво прави екземпляр от този клас, когато му изпратите съобщението „report“ и то

върви по пътя за търсене, търсейки метод за съвпадение? Нека го попитаме:

c = C. нов

в. доклад

Листинг 4.8 Смесване в два модула с дефиниран метод със същото име

|  |
| --- |
| **Страница 105** |

**105**

***Модули, класове и търсене на методи***

Отговорът е "метод на отчета" в модул N ". Първият срещнат метод на доклад

в метода на c търсенето е пътят в *последния смесен модул* . В такъв случай,

това означава N - така че методът на отчета на N печели над метода на M със същото име.

Към това трябва да се добави наблюдението, че включването на модул повече от веднъж

няма ефект.

Аз питомен модул повече от веднъж

Погледнете този пример, който се основава на предишния пример - но този път ние

включете М втори път, след N:

клас С

включват М

включват N

включват М

край

Може да очаквате, че когато стартирате метода на отчета, ще получите версията на M,

защото M беше най-скоро включеният модул. Но повторното включване на модул не го прави

Направи нещо. Тъй като М вече лежи на пътя за търсене, вторият включва M инструкции

ция няма ефект. N все още се счита за последно включен модул:

c = C. нов

в. доклад

Накратко, можете да манипулирате пътищата за търсене на методи на вашите обекти, но само до

точка.

Във всички примери досега използвахме включване за смесване на модули. Време е за

да се върнете в дискусията.

***4.2.3***

***Как работи prepend***

Всеки път, когато включите модул в клас, вие влияете какво се случва кога

екземплярите от този клас трябва да разрешават съобщенията в имена на методи. Същото е вярно

на prepend. Разликата е, че ако добавите модул към клас, обектът изглежда

в този модул първо, преди да изглежда в класа.

Ето пример:

модул MeFirst

def доклад

поставя "Здравей от модул!"

край

край

клас Личност

добавете MeFirst

def доклад

поставя "Здравей от клас!"

край

край

p = Person.new

стр. доклад

**Резултат: 'доклад'**

**метод в модул N**

|  |
| --- |
| **106 стр** |

**106**

C ГЛАВА 4 ***Модули и организация на програмата***

Резултатът е "Здравейте от модул!" Защо? Защото ние добавихме

MeFirst модул към класа. Това означава, че екземплярът на класа ще погледне

модула първо, когато се опитва да намери метод, наречен отчет. Ако бяхме използвали

включва, класът ще бъде търсен преди модула и версията на класа на

докладът ще „спечели“.

Можете да видите разликата между включване и добавяне, отразена както на фигура 4.2

и в списъка с предците на даден клас - което означава всички класове и модули къде

екземпляр на класа ще търси методи, изброени по ред. Ето ги предците

на класа Person от последния пример, в irb:

> Person.ancersors

=> [MeFirst, Person, Object, Readline, Kernel, BasicObject]

Сега модифицирайте примера, за да използвате включване вместо добавяне. Случват се две неща. Първо,

изходът се променя:

Здравейте от класа!

Второ, редът на предците се променя:

> Person.ancersors

=> [Person, MeFirst, Object, Readline, Kernel, BasicObject]

(Разбира се, името MeFirst престава да има смисъл, но вие разбирате общата идея.)

Можете да използвате prepend, когато искате версия на модула на един или повече методи

имат предимство пред версиите, дефинирани в даден клас. Както е споменато по-рано,

prepend е ново в Ruby 2.0. Няма да го видите да се използва много, поне не още. Но е полезно

да знаете, че е там, както за да можете да го използвате, ако имате нужда, така и за да знаете

какво означава, ако го срещнете в чужд код.

***4.2.4***

***Обобщени са правилата за търсене на методи***

Основните правила, регулиращи търсенето на методите и подреждането на метода за търсене

пътеката в Ruby 2 са илюстрирани на фигура 4.2.

За да разреши съобщение в метод, обект търси метода в

1

Модули, добавени към своя клас, в обратен ред на предварителните

2

Класът му

3

Модули, включени в неговия клас, в обратен ред на включване

4

Модули, добавени към своя суперклас

5

Суперкласът на своя клас

6

Модули, включени в неговия суперклас

7

По същия начин, до Object (и неговото смесено ядро) и BasicObject

По-специално обърнете внимание на точката, че модулите се търсят за методи в обратен ред

на предшестващо или включване. Това гарантира предсказуемо поведение в случай, че даден клас

смесва в два модула, които дефинират един и същ метод.

|  |
| --- |
| **107** |

**107**

***Модули, класове и търсене на методи***

За навигация се предлага малко специализирана, но полезна и често срещана техника

пътеката за търсене изрично: ключовата дума super.

***4.2.5***

***Изкачване на пътя за търсене на метода със супер***

В тялото на дефиницията на метод можете да използвате ключовата дума super, за да скочите нагоре

до следващата най-висока дефиниция в метода за търсене на пътя на метода, който

отдаване под наем.

Следващият списък показва основен пример (след което ще стигнем до „Защо

би ли направил това? " аспект).

Какво ще кажете за единичните методи?

От глава 3 сте запознати с *единичния метод* - метод, дефиниран директно

върху обект (def obj.talk) - и може да се чудите къде в пътя за търсене на метода

единични методи лъжат. Отговорът е, че те лежат в специален клас, създаден за

единствената им цел да ги съдържа: единичният клас на обекта. Ще разгледаме сингълтън

класове в детайли по-късно в книгата, след което ще ги включим в метода-

модел за търсене.

клас Обект

(

)

*вграден*

включва ядрото

модул Ядро

(

)

*вграден*

клас С

добавете P

включват М

край

клас BasicObject

(

)

*вграден*

клас D <C

край

обект = D.нови

**обект.x**

модул P

край

модул М

край

Фигура 4.2 Екземпляр на клас **D** търси метод **x** в неговия път за търсене на метод и в двата

включени и добавени модули.

|  |
| --- |
| **108 серия** |

**108**

C ГЛАВА 4 ***Модули и организация на програмата***

модул М

def доклад

поставя метод "'report' в модул M"

край

край

клас С

включват М

def доклад

поставя метод "'report' в клас C"

поставя „На път да задейства следващия по-горе метод на отчета ...“

супер

поставя „Обратно от„ супер “обаждането.“

край

край

c = C. нов

в. доклад

Резултатът от стартирането на списък 4.9 е както следва:

метод „доклад“ в клас С

На път да задейства следващия по-горе метод на доклад ...

метод „доклад“ в модул М

Обратно от „супер“ обаждането.

Екземпляр на C (а именно, c) получава съобщението „доклад“ e . Търсене на метод

процесът започва с класа на c (C) - и със сигурност има метод на доклад c . Че

метод се изпълнява.

Вътре в метода е извикване на супер d . Това означава, въпреки че обектът е намерен

метод, съответстващ на съобщението („доклад“), той трябва да продължи да търси и да намира

следващия мач. Следващото съвпадение за отчета, в този случай, е методът на отчета

дефинирано в модул М B .

Имайте предвид, че отчетът M # би бил първото съвпадение при търсене на отчет

метод, ако C # отчет не съществува. Супер ключовата дума ви дава начин да се обадите на това, което би

са били приложимата версия на метод в случаите, когато този метод е бил

заменени по-късно в пътя за търсене. Защо бихте искали да направите това?

Понякога, особено когато пишете подклас, метод в съществуващ клас

прави почти, но не съвсем това, което искате. Със супер можете да имате най-доброто и от двете

светове чрез свързване или опаковане на оригиналния метод, както илюстрира следващият списък.

клас Велосипед

attr\_reader: предавки,: колела,: седалки

def инициализиране (предавки = 1)

@ колела = 2

@seats = 1

@gears = предавки

край

край

Листинг 4.9 Използване на **супер** ключовата дума за достигане на едно ниво в търсещата пътека

Листинг 4.10 Използване на **супер** за увиване на метод в подклас

б

° С

д

д

б

|  |
| --- |
| **109** |

**109**

***Методът на липсващия метод***

клас Тандем <Велосипед

def инициализиране (зъбни колела)

супер

@seats = 2

край

край

super предоставя чист начин да направите тандем почти като велосипед. Ние се променяме само

какво трябва да се промени (броят на местата c ) и супер задейства по-ранното

инициализира метод B , който задава стойности по подразбиране, подобни на велосипед, за останалите свойства

на тандема.

Когато се обадим супер, не препращаме изрично аргумента gears, който е предаден

за инициализиране. И все пак, когато се извика оригиналният метод за инициализиране в Bicycle, всеки

аргументите, предоставени на тандемната версия са видими. Това е специално поведение на

супер. Начинът, по който супер се справя с аргументите, е както следва:

Извиква се без списък с аргументи (празен или по друг начин), супер автоматично за

съхранява аргументите, които са били предадени на метода, от който е извикан.

Извикан с празен списък с аргументи - super () - super не изпраща аргументи

методът нагоре, дори ако аргументите са били предадени на текущия метод.

Извикан с конкретни аргументи - супер (a, b, c) - super изпраща точно тези

аргументи.

Това необичайно третиране на аргументи съществува, защото най-често срещаният случай е първият

един, където искате да се приближите до следващия по-висок метод със същите аргументи

като тези, получени по метода, от който се извиква супер. Този случай е даден

най-простият синтаксис - просто пишете супер. (И тъй като super е ключова дума, а не

метод, той може да бъде проектиран, за да осигури това специално поведение.)

След като видяхте как работи търсенето на методи, нека помислим какво се случва

когато търсенето на метод се провали.

***4.3***

***Методът на липсващия метод***

Модулът Kernel осигурява метод на екземпляр, наречен method\_missing. Този метод

се изпълнява всеки път, когато обект получи съобщение, че не знае как да отговори

до - т.е. съобщение, което не съвпада с метод никъде в метода на обекта -

пътека за търсене:

>> o = Object.new

=> # <Обект: 0x0000010141bbb0>

>> o.blah

NoMethodError: недефиниран метод `blah # # Обект: 0x0000010141bbb0>

Лесно е да прихващате повиквания към липсващи методи. Заменяте method\_missing, или на

единична основа за обекта, на който извиквате метода, или в класа на обекта или

един от предците на този клас:

>> def o.method\_missing (m, \* аргументи)

>> поставя „Не можете да се обадите на номер {m} за този обект; моля, опитайте отново.“

>> край

° С

б

|  |
| --- |
| **110. стр** |

**110**

C ГЛАВА 4 ***Модули и организация на програмата***

=> нула

>> o.blah

Не можете да се обадите на бла на този обект; Моля, опитайте отново.

Когато замените method\_missing, трябва да имитирате подписа на метода на

оригиналния B . Първият аргумент е името на липсващия метод - съобщението, което

изпратили сте обекта и той не е разбрал. Параметърът \* args се задушава

всички останали аргументи. (Можете също така да добавите специален аргумент за свързване с код

блок, но нека не се тревожим за това, докато не разгледаме кодовите блокове по-подробно.)

Първият аргумент идва при вас под формата на символен обект. Ако искате да изпитате-

или да го анализирате, трябва да го преобразувате в низ.

Дори ако замените method\_missing, предишната дефиниция все още е достъпна за

вие чрез супер.

***4.3.1***

***Комбиниране на method\_missing и super***

Обичайно е да искате да прихванете неразпознато съобщение и да решите на място,

дали да се справя с него или да го предаде на оригиналния method\_missing (или евентуално

междинна версия, ако е дефинирана друга). Можете да направите това лесно, като използвате супер.

Ето пример за типичния модел:

клас Студент

def method\_missing (m, \* аргументи)

ако m.to\_s.start\_with? ("grade\_for\_")

# връща подходящата оценка въз основа на анализирането на името на метода

друго

супер

край

край

край

Като се има предвид този код, обаждането до, да речем, grade\_for\_english на екземпляр на студент води до

истинският клон на теста if. Ако липсващото име на метод не започва с

grade\_for, фалшивият клон се взема, което води до извикване на super. Това обаждане ще отнеме

вие към каквото и да е следващото изпълнение на method\_missing по протежение на обекта

път за търсене на метод. Ако не сте заменили method\_missing никъде другаде

линията, супер ще намери метода на Kernel\_missing и ще изпълни това.

Нека разгледаме по-обширен пример за тези техники. Ще напишем човек

клас. Нека започнем отгоре с някакъв код, който илюстрира как искаме да бъде класът

използвани. След това ще приложим класа по такъв начин, че кодът да работи.

Следващият списък показва някакъв код за използване за класа Person.

j = Person.new ("Джон")

p = Person.new ("Павел")

g = Person.new ("Джордж")

r = Person.new ("Ринго")

j.has\_friend (p)

j.has\_friend (g)

Листинг 4.11 Примерно използване на класа **Person**

**Преобразуване на символ в низ,**

**с to\_s, преди тестване**

|  |
| --- |
| **111** |

**111**

***Методът на липсващия метод***

g.has\_friend (p)

r.has\_hobby ("звъни")

Person.all\_with\_friends (p) .each do | person |

поставя „# {person.name} е приятел с # {p.name}“

край

Person.all\_with\_hobbies ("звъни"). Всеки прави | човек |

поставя "# {person.name} е в пръстени"

край

Бихме искали изходът на този код да бъде

Джон е приятел с Пол

Джордж е приятел с Пол

Ринго е в пръстени

Общата идея е, че човек може да има приятели и / или хобита. Освен това

Класът на човека ни позволява да търсим всички хора, които имат даден приятел или всички хора, които го имат

дадено хоби. Търсенията се извършват с all\_with\_friends и

all\_with\_hobbies клас методи.

Формулата all\_with\_ \* метод-име изглежда като добър кандидат за работа

чрез method\_missing. Въпреки че използваме само два варианта от него (приятели и

bies), това е видът модел, който може да се простира до произволен брой имена на методи. Нека да

прехващане на метод\_пропускане в класа Person.

В този случай методът\_missing, с който имаме работа, е методът на класа: трябва

прихваща липсващи методи, извикани Person. Следователно някъде по линията, ние

нужда от дефиниция като тази:

клас Личност

def self.method\_missing (m, \* аргументи)

# код тук

край

край

Името на метода, m, може или не може да започва с подниза all\_with\_. Ако го направи,

ние го искаме; ако не стане, ние го хвърляме обратно - или нагоре - с любезното съдействие и оставяме Kernel

#method\_missing се справя. (Не забравяйте: класовете са обекти, така че класът обект Person

има достъп до всички методи на екземпляра на ядрото, включително method\_missing.)

Ето малко по-сложен (но все пак схематичен) изглед на method\_missing:

клас Личност

def self.method\_missing (m, \* аргументи)

метод = m.to\_s

ако method.start\_with? ("all\_with\_")

# Обработете заявката тук

друго

супер

край

край

край

Причината за извикването на to\_s B е, че името на метода (съобщението) се връчва

off to method\_missing под формата на символ. Символите нямат start\_with?

метод, така че трябва да преобразуваме символа в низ, преди да тестваме съдържанието му.

**Дефинирайте метода директно върху себе си,**

**който е обектът на клас Person**

б

° С

д

|  |
| --- |
| **Страница 112** |

**112**

C ГЛАВА 4 ***Модули и организация на програмата***

Условната логика c се разклонява дали обработваме all\_with\_ \* mes-

мъдрец. Ако сме, ние се справяме. Ако не, пунтваме със супер d .

С поне един проект на method\_missing на място, нека разработим останалата част от

Клас на човек. Няколко изисквания са ясни от изброения по-рано код за повикване на най-високо ниво:

Предметите на хората следят своите приятели и хобита.

Класът Person проследява всички съществуващи хора.

Всеки човек има име.

Втората точка се подразбира от факта, че вече питаме Човека

клас за списъци с хора, които имат определени хобита и / или определени приятели.

Следващият списък съдържа изпълнение на частите от класа Person

които се отнасят до тези изисквания.

клас Личност

ХОРА = []

attr\_reader: име,: хобита,: приятели

def инициализиране (име)

@ име = име

@hobbies = []

@ приятели = []

ХОРА << себе си

край

def has\_hobby (хоби)

@hobbies << хоби

край

def has\_friend (приятел)

@friends << приятел

край

Ние скривалище всички съществуващи хора в масив, който се проведе в константа ХОРА B . Когато нов

човек е инстанциран, този човек се добавя към масива от хора, с любезното съдействие на масива

метод на добавяне <<д . Междувременно се нуждаем от някои читателски атрибути: име, хоби,

и приятели c . Предоставянето на тези атрибути позволява на външния свят да вижда важно

аспекти на обектите Личност; хобитата и приятелите също ще ви бъдат от полза в пълен размер

внедряване на method\_missing.

Методът за инициализация приема име като единствен аргумент и го записва в @name. То

също инициализира масивите на хобитата и приятелите d . Тези масиви отново влизат в игра

методите has\_hobby и has\_friend f , които всъщност са само удобни за потребителя обвивки-

pers около тези масиви.

Сега имаме достатъчно код, за да завършим изпълнението на Person.method\_missing.

Листинг 4.13 показва как изглежда (включително крайния краен разделител за цялото

клас). Използваме удобен вграден метод за заявка, public\_method\_defined ?, който

ни казва дали Person (представен в метода с ключовата дума self) има a

метод със същото име като този в края на низа all\_with\_.

Листинг 4.12 Внедряване на основната логика на класа **Person**

б

° С

д

д

е

|  |
| --- |
| **113** |

**113**

***Методът на липсващия метод***

def self.method\_missing (m, \* аргументи)

метод = m.to\_s

ако method.start\_with? ("all\_with\_")

attr = метод [9 ..- 1]

ако self.public\_method\_defined? (attr)

PEOPLE.find\_all do | person |

person.send (attr) .include? (args [0])

край

друго

повиши ArgumentError, "Не мога да намеря # {attr}"

край

друго

супер

край

край

край

Ако имаме съобщение all\_with\_ B , искаме да игнорираме тази част и да уловим останалото

на низа, което можем да направим, като вземем подниза, който се намира в деветия

последни позиции на символи; това е индексирането на низа с 9 ..- 1 постига c . (Това

означава, започвайки от десетия знак, защото индексирането на низове започва от нула.) Сега ние

искам да знам дали полученият подниз съответства на един от екземпляра на Person

методи - по-специално, хобита или приятели. Вместо да кодирате тези две имена,

поддържаме нещата гъвкави и мащабируеми, като проверяваме дали класът Person определя a

метод с нашето поднизово като име d .

Какво ще се случи по-нататък зависи от това дали търсенето на символа е успешно. Да се

започнете първо с втория клон, ако исканият атрибут не съществува, ние издигаме

грешка с подходящо съобщение f . Ако успее - което ще стане, ако съобщението

е приятели или хобита или друг атрибут, който сме добавили по-късно - стигаме до същността на

въпроса.

В допълнение към името на метода all\_with\_ \*, извикването на метода включва аргумент

мента, съдържащ нещото, което търсим (името на приятел или хоби, за

пример). Този аргумент се намира в args [0], първият елемент на аргумента

Масив „гъба“, обозначен като \* args в списъка с аргументи; бизнес края на цялото

method\_missing метод е да се намерят всички хора, чиито attr включва аргументи [0] e . Че

формулата се превръща в, да речем, всички хора, чиито хобита включват музика, или всички хора

чиито приятели включват някой конкретен приятел.

Имайте предвид, че тази версия на method\_missing включва две условни структури.

Това е така, защото две неща могат да се объркат: първо, може да обработваме съобщение, което

не отговаря на модела all\_with\_ \* (например "бла"); и второ, ние

може да има заявка all\_with\_ \*, когато \* частта не отговаря на нищо, което

класът Person знае за (например всички\_с\_деца). Ние лекуваме втория

като фатално състояние и повишаване на грешка F . Ако първото условие се провали, това означава това

конкретното съобщение не е това, което търси този конкретен method\_missing. Ние връчваме

ТРОЛ нагоре към следващата най-висока дефиниция на method\_missing като се обадите на супер G .

Листинг 4.13 Пълно изпълнение на **Person.method\_missing**

б

° С

д

д

е

ж

|  |
| --- |
| **114** |

**114**

C ГЛАВА 4 ***Модули и организация на програмата***

Извикан без аргументи, супер автоматично получава всички аргументи, дошли до

текущ метод; по този начин голото повикване към super е, в този случай, еквивалентно на super (m,

\* аргументи) (но по-кратки и по-удобни).

ЗАБЕЛЕЖКА Ще разгледаме отново method\_missing в глава 15, като част от по-широко

вижте кукичките и обратните обаждания на Ruby по време на изпълнение, от които е само method\_missing

един. (Има и едно, наречено response\_to\_missing ?, което като негово име

предполага е нещо като хибрид; ще го срещнете в глава 15.) Струва си да го имате

въведена тук method\_missing, защото това е може би най-много

често използван член на семейството за обратно извикване и такъв, за който е вероятно

вижте и чуйте дискутираните по-скоро, отколкото по-късно във вашите изследвания на Ruby.

Вече имате добра представа както за класовете, така и за модулите, както и колко индивидуални

обекти, при получаване на съобщения, търсете метод за съвпадение, като обхождате техния клас /

модулно родословно дърво и как се справят с неуспешно търсене. След това ще разгледаме какво сте

можете да правите с тази система - по-конкретно видовете решения, които можете и трябва да вземете

по отношение на дизайна и именуването на вашите класове и модули, в интерес на писането

ясни и разбираеми програми.

***4.4***

***Проектиране и именуване на клас / модул***

Фактът, че Ruby има класове и модули - заедно с факта, че от обект

перспектива, всичко, което има значение, е дали даден метод съществува, а не какъв клас или модификация

ule дефиницията на метода е в - означава, че имате много избор, когато става въпрос

дизайна и структурата на вашите програми. Това богатство на избора на дизайн поражда известни

странични действия, за които трябва да сте наясно.

Вече разгледахме един случай (класът Stack), където би било възможно

да поставя всички необходими дефиниции на метода в един клас, но беше изгодно

за да издърпате някои от тях, поставете ги в модул (подобен на стека) и след това смесете модула

ule в класа. Няма правило за решаване кога да се направи кое. Зависи от вашите

настоящи и - доколкото можете да ги предскажете - бъдещи нужди. Понякога е така

изкушава да раздели всичко на отделни модули, защото модулите, които пишете

за една програма може да бъде полезна в друга („Просто знам, че ще ми трябва

Модул ThreePronged отново някой ден! “ казва гласът на пакрат в главата ви). Но

има такова нещо като свръхмодуларизация. Зависи от ситуацията. Имате

няколко мощни инструмента, налични за вас - миксове и наследяване - и вие трябва

обмислете във всеки случай как да ги балансирате.

***4.4.1***

***Смесвания и / или наследяване***

Комбинираните модули са тясно свързани с наследяването на класа. И в двата случая един обект (клас

или модул) установява тясна връзка с друг, като става съсед на

път за търсене на метод. В някои случаи може да откриете, че можете да проектирате част от вашия

програма или с модули, или с наследяване.

Нашият клас CargoHold е пример. Ние го внедрихме, като го смесихме в

Модул, подобен на стека. Но бяхме ли тръгнали по пътя на писане на Stack клас вместо a

|  |
| --- |
| **115** |

**115**

***Проектиране и именуване на клас / модул***

Подобен на стек модул, все още можехме да имаме CargoHold. Това би било подклас

на Stack, както е показано в следващия списък.

клас Stack

attr\_reader: стек

def инициализира

@stack = []

край

def add\_to\_stack (obj)

@ stack.push (obj)

край

def take\_from\_stack

@ stack.pop

край

край

клас Куфар

край

клас CargoHold <Stack

def load\_and\_report (obj)

печат "Зареждане на обект"

поставя obj.object\_id

add\_to\_stack (obj)

край

def разтоварване

take\_from\_stack

край

край

От гледна точка на отделен обект CargoHold, процесът работи в това

изброявайки точно както е работило в по-ранното внедряване, където CargoHold се смеси

модулът, подобен на стека. Обектът се занимава с намирането и изпълнението на методи

които съответстват на съобщенията, които получава. Той или намира такива методи на своя

път за търсене на метод или не. Не се интересува дали методите са дефинирани

в модул или клас. Това е като да търсиш къща за отвертка: не ти пука

в коя стая го намирате и в коя стая го намирате няма значение за какво

се случва, когато впоследствие използвате отвертка за дадена задача.

Няма нищо лошо в този подход, основан на наследяване на изпълнението

CargoHold, с изключение на това, че изяжда възможността за наследяване, която CargoHold има. Ако

друг клас може да е по-подходящ от Stack, за да служи като суперклас на CargoHold

(като хипотетично StorageSpace или AirplaneSection), в крайна сметка може да се наложи

гъвкавостта, която бихме спечелили, като превърнем поне един от тези класове в модул.

Нито едно правило или формула не винаги води до правилния дизайн. Но е полезно да запазите a

няколко съображения, които трябва да имате предвид, когато вземате решения клас-срещу-модул:

*Модулите нямат екземпляри.* От това следва, че субектите или нещата обикновено са най-добри

моделирани в класове, а характеристиките или свойствата на обектите или нещата са

най-добре капсулирани в модули. Съответно, както е отбелязано в раздел 4.1.1, клас

Листинг 4.14 **CargoHold** , наследяване от **Stack** вместо смесване в **Stacklike**

|  |
| --- |
| **116 стр** |

**116**

C ГЛАВА 4 ***Модули и организация на програмата***

имената обикновено са съществителни, докато имената на модулите често са прилагателни (Stack

срещу Stacklike).

*Класът може да има само един суперклас, но може да смесва толкова модули, колкото иска.* Ако

използвате наследяване, дайте приоритет на създаването на разумен суперклас / подклас

връзка. Не използвайте единствената връзка на класа, която да дарите

класът с това, което може да се окаже само един от няколкото набора характеристики.

Обобщавайки тези правила в един пример, ето какво не трябва *да* правите:

модул Превозно средство

...

клас SelfPropelling

...

class Truck <SelfPropelling

включват превозно средство

...

По-скоро трябва да направите това:

модул SelfPropelling

...

клас Превозно средство

включват SelfPropelling

...

клас Камион <Превозно средство

...

Втората версия моделира обектите и свойствата много по-добре. Камион

слиза от Vehicle (което има смисъл), докато SelfPropelling е характер-

характер на превозните средства (поне всички, които ни интересуват в този модел на света) - характер

теристика, която се предава на камионите по силата на това, че Камионът е потомък или е специализиран

форма, на превозно средство.

Друго важно съображение при проектирането на клас / модул е ​​влагането на модул

ules и / или класове един в друг.

***4.4.2***

***Влагане на модули и класове***

Можете да вложете дефиниция на клас в дефиниция на модул по следния начин:

модул Инструменти

клас Hammer

край

край

За да създадете екземпляр на класа Hammer, дефиниран вътре в модула Tools, използвате

маркер за постоянно търсене с двоеточие (: :), за да посочи пътя към името на класа:

h = Инструменти :: Hammer.new

Вложени вериги модул / клас като Tools :: Hammer понякога се използват за създаване на отделни

оценете пространства от имена за класове, модули и методи. Тази техника може да помогне, ако две

класовете имат подобно име, но не са един и същи клас. Например, ако имате

|  |
| --- |
| **117** |

**117**

***Обобщение***

Инструменти :: Клас Hammer, можете също да имате Piano :: Class Hammer и двата Hammer

класовете няма да влизат в конфликт помежду си, защото всеки е вложен в собственото си пространство от имена

(Инструменти в единия случай, Пиано в другия).

(Алтернативен начин за постигане на това разделяне би бил класът ToolsHammer

и клас PianoHammer, без да си прави труда да ги влага в модули. Но нанизване

Такива имена заедно могат бързо да доведат до визуална бъркотия, особено когато елементи

са вложени по-дълбоко от две нива.)

Ще разгледаме допълнително вложени класове, модули и други константи в следващата глава,

когато говорим по-подробно за предмета на обхвата. Междувременно имайте предвид, че тази способност

това ви позволява да влагате модули и класове един в друг (до всякаква дълбочина, в произволен ред)

още една ос, по която можете да планирате дизайна и структурата на вашата програма.

***4.5***

***Обобщение***

Глава 4 е едновременно спътник и продължение на предишната глава

на класове. В тази глава сте виждали

Модули, отблизо и в детайли

Прилики и разлики между модулите и класовете (и двете могат да се обединят

методи и константи заедно, но модулите не могат да бъдат създадени)

Примери за това как можете да използвате модули, за да изразите дизайна на програма

Поглед на обекта от процеса на намиране и изпълнение на метод в

отговор на съобщение или обработка на грешка с method\_missing в случаите, когато

съобщението не съответства на метод

Как да влагате класове и модули един в друг, с предимството да ги запазите

пространства от имена отделни и ясни

Особено важно е да се вземе предвид начинът, по който обектите разрешават съобщенията

методи: те търсят в поредица от класове и модули. Обекти

Клас или модул?

Когато видите конструкция като Tools :: Hammer, не можете да разберете само от тази кон-

структурирайте какво е клас и какво е модул - нито за това дали Hammer е

обикновена, стара константа. (Инструментите трябва да са клас или модул, защото има Hammer

в много случаи, фактът, че не можете да различавате класове от модули

в този вид контекст няма значение; това, което има значение, е гнезденето или веригата на

имена по начин, който има смисъл. Това е също толкова добре, защото не можете да разберете

какво е какво, без да гледате изходния код или документацията. Това е кон-

последователност на факта, че класовете са модули - класът Class е подклас на

Модулът на класа - и в много отношения (с най-забележителното изключение, че

класове могат да бъдат инстанцирани), тяхното поведение е подобно. Разбира се, обикновено бихте

знайте какво представлява Tools :: Hammer , било защото сте написали кода или

защото сте виждали документация. И все пак си струва да осъзнаем, че самата нотация

не ти казва всичко.

|  |
| --- |
| **118** |

**118**

C ГЛАВА 4 ***Модули и организация на програмата***

нямат самите методи, въпреки че понякога това да се формулира така е

удобен пряк път. Класовете и модулите имат методи; обектите имат способността да преминават

класове и модули в търсене на методи.

Сега, когато влагаме елементи един в друг, следващата тема трябва и

ще разгледа подробно обхвата: какво се случва с данните и променливите, когато вашата програма

преминава от един контекст на код в друг. Ще разгледаме обхвата във връзка с

свързана, често преплетена тема за себе си, обектът по подразбиране.

|  |
| --- |
| **119** |

**119**

*Обектът по подразбиране (самостоятелно),*

*обхват и видимост*

Когато описваме и обсъждаме компютърни програми, ние често използваме пространствени и, някои-

времена, човешки метафори. Говорим за това, че сме „в“ определение на клас или се връщаме

„От“ извикване на метод. Ние се обръщаме към обекти от второ лице, както в obj.respont

\_to? ("x") (т.е., "Хей, обещавате ли, отговаряте ли на 'x'?"). С изпълнението на програмата,

въпрос към кои обекти се обръща и къде във въображаемото пространство на

програмата, която стоят, непрекъснато се измества.

И разместванията не са просто метафорични. Значенията на идентификаторите също се променят. A

малко елементи означават едно и също нещо навсякъде. Целите числа например означават какво

те означават където и да ги видите. Същото важи и за ключовите думи: не можете да използвате

думи като def и class като имена на променливи, така че когато ги видите, можете лесно

съберете какво правят. Но повечето елементи зависят от контекста за тяхното значение.

***Тази глава обхваща***

Ролята на текущия или обекта по подразбиране, самостоятелно

Правила за обхват за локален, глобален и клас

променливи

Постоянно търсене и видимост

Правила за достъп до метод

|  |
| --- |
| **120. стр** |

**120**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

Повечето думи и символи - повечето идентификатори - могат да означават различни неща по различен начин

места и времена.

Тази глава е за ориентиране в Ruby кода: знаейки как

вашите, които използвате, ще разрешат, следвайки промените в контекста и правят

чувство за използване и повторна употреба на идентификатори и термини. Ако разбирате какво може

промяна от един контекст в друг, както и това, което предизвиква промяна в контекста (за

например, въвеждайки блок с дефиниция на метод), винаги можете да се ориентирате в a

Програма Ruby. И това не е въпрос само на пасивна грамотност на Ruby: вие също трябва

да знаете за контекста и как те влияят върху значението на това, което правите кога

пишеш Руби.

Тази глава се фокусира първоначално върху две теми: *себе си* и *обхвата* . *Себето* е „токът“ или

Обект по подразбиране, роля, която обикновено се възлага на много обекти в последователност (макар и само

един по един), докато се изпълнява програма. Аз-обектът в Ruby е като първото лице или *аз* на

програмата. Както в книга с множество разказвачи от първо лице, ролята *I* може да получи

минаха наоколо. Винаги има един Аз, но какъв обект е той ще варира. Правилата на

обхватът управлява видимостта на променливите (и други елементи, но до голяма степен променливи). Това е

важно е да знаете в какъв обхват се намирате, за да можете да разберете за какво се отнасят променливите

и не ги бъркайте с променливи от различни области, които имат едно и също име,

нито с подобни назовани методи.

Между тях себе си и обхватът са основните ключове за ориентиране в Ruby

програма. Ако знаете в какъв обхват се намирате и знаете какъв обект е себе си, ще можете

за да разберете какво се случва и ще можете бързо да анализирате грешките.

Третата основна тема на тази глава е *достъпът до метода* . Ruby осигурява механизми за

правене на разграничение между нивата на достъп на методите. По принцип това означава, че ограниченията на правилата

извикване на методите в зависимост от това какво е аз. Следователно достъпът до метод е a

мета-тема, основана на изследването на себе си и обхвата.

Накрая ще обсъдим и тема, която обединява няколко от тези нишки: *най-високо ниво*

*методи* , които са написани извън определения клас или модул.

Нека започнем със себе си.

***5.1***

***Разбиране на self, текущия / обект по подразбиране***

Един от крайъгълните камъни на програмирането на Ruby - гръбнакът, в някои отношения - е

обекта по подразбиране или текущия обект, достъпен за вас във вашата програма чрез

ключова дума самостоятелно. Във всеки момент, когато програмата ви работи, има един и само един

себе си. Да бъдеш себе си има определени привилегии, както ще видиш. В този раздел ще разгледаме как

Ruby определя кой обект е самостоятелен в даден момент и какви привилегии се предоставят

към обекта, който е себе си.

***5.1.1***

***Кой може да бъде себе си и къде***

Винаги има един (и само един) текущ обект или себе си. Можете да разберете кой обект е

като следвате малкия набор от правила, обобщени в таблица 5.1. Съдържанието на таблицата ще бъде

обяснено и илюстрирано, докато вървим напред.

|  |
| --- |
| **Страница 121** |

**121**

***Разбиране на self, текущия / обект по подразбиране***

За да знаете кой обект е себе си, трябва да знаете в какъв контекст се намирате. На практика,

няма много контексти, за които да се притеснявате. Там е най-горното ниво (преди да сте

въведени във всеки друг контекст, като дефиниция на клас). Има дефиниция на класа

блокове, блокове с дефиниция на модули и блокове с дефиниция на метод. Освен няколко под-

нещата в начина, по който тези контексти си взаимодействат, това е всичко. Както е показано в таблица 5.1, себе си е

определя се в кой от тези контексти се намирате (дефинициите на клас и модул са

подобни и тясно свързани).

Фигура 5.1 ви дава схематичен изглед на повечето случаи в таблица 5.1. И двете

да ви покаже, че някой обект винаги е себе си и че кой обект е аз зависи от това къде

вие сте в програмата.

Най-основният програмен контекст и в някои отношения уникален е върхът

ниво: контекстът на програмата преди дадено определение на клас или модул

отворени или след като всички са затворени. След това ще разгледаме идеите на най-високото ниво

за себе си.

Таблица 5.1 Как се определя текущият обект (самостоятелно)

Контекст

Пример

Кой обект е себе си?

Най-високо ниво на програмата

Всеки код извън други блокове

main (вграден по подразбиране от най-високо ниво

обект)

Определение на класа

клас С

себе си

Обектът на класа C

Определение на модула

модул М

себе си

Модулният обект M

Определения на методите

1 Най-високо ниво (извън който и да е блок за дефиниция):

def method\_name

себе си

Какъвто и да е обект е себе си, когато

метод се извиква; методи от най-високо ниво

са достъпни като частни методи за

всички обекти

2 Дефиниция на метод на екземпляр в клас:

клас С

def method\_name

себе си

Екземпляр на C , отговарящ на

име\_на метода

3 Определение на метода на инстанция

в модул:

модул М

def method\_name

себе си

Индивидуален обект, разширен от M

Екземпляр на клас, който се смесва в M

4 Сингълтън метод за конкретен обект:

def obj.method\_name

себе си

Obj

|  |
| --- |
| **Страница 122** |

**122**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

***5.1.2***

***Самообект от най-високо ниво***

Терминът *най-високо ниво се* отнася до програмен код, написан извън който и да е клас или модул-

дефиниционен блок. Ако отворите чисто нов текстов файл и въведете

x = 1

създали сте локална променлива от най-високо ниво x. Ако пишете

def m

край

създали сте метод от най-високо ниво. (Ще разгледаме методите от най-високо ниво в много повече

подробности в раздел 5.4; те са уместни тук точно като указатели за съществуването на топ-

ниво самостоятелно.) Редица наши примери, особено в ранните глави (например,

тези в глава 2, демонстриращи семантиката на аргументите) включваха код от най-високо ниво. Веднъж

започнахме да пишем дефиниции на класове и модули, започна да се появява повече от нашия код

вътре в тези определения. Начинът, по който се променя в дефинициите на клас, модул и метод е

униформа: ключовата дума (клас, модул или деф) маркира превключване към ново аз. Но какво

self, когато все още не сте въвели никакъв блок за дефиниция?

Отговорът е, че Ruby ви осигурява стартираща самостоятелност на най-високо ниво. Ако питате

то, за да се идентифицира

ruby -e "поставя себе си"

ще ви каже, че се нарича main.

поставя "Най-високо ниво"

поставя „self is # {self}“

клас С

край

поставя "Блок за дефиниция на клас:"

поставя „self is # {self}“

def self.x

край

def m

край

поставя "Метод на класа Cx:"

поставя „self is # {self}“

поставя "Метод на екземпляр C # m:"

поставя „self is # {self}“

Себе си във всеки метод

е обектът, към който

съобщение (методът

обаждане) е изпратено.

За метод на клас,

това означава

обект на клас.

За метод на екземпляр,

това означава екземпляр

на класа, чийто

екземпляр метод е.

Аз в a

определение на класа

е класът

самия обект.

Аз на най-високо ниво

е „по подразбиране

обект по подразбиране, ”

основен.

Фигура 5.1 Определянето на себе си в различен контекст

|  |
| --- |
| **Страница 123** |

**123**

***Разбиране на self, текущия / обект по подразбиране***

main е специален термин, който самостоятелният обект по подразбиране използва, за да се отнася към себе си. Не можеш

посочете го като основен; Ruby ще тълкува използването на main като обикновена променлива или метод

име. Ако искате да вземете main по някаква причина, трябва да го присвоите на променлива в

Най-високо ниво:

m = себе си

Не е вероятно да се наложи да направите това, но така се прави. По-често,

ще почувствате необходимостта от доста фино усещане за това какво е аз във вашия клас, модул,

и дефиниции на методи, където ще се проведе по-голямата част от вашето програмиране.

***5.1.3***

***Самостоятелни дефиниции на клас, модул и метод***

Струва си да следите внимателно себе си, докато пишете класове, модули и методи. Там

не са толкова много правила за учене и те се прилагат последователно. Но си струват

да се учите добре отпред, така че сте наясно защо различните техники, които използвате това

зависят от стойността на самоиграването по начина, по който го правят.

Всичко е свързано със самостоятелно превключване от един обект на друг, което се прави, когато вие

въведете дефиниция на клас или модул, дефиниция на метод на екземпляр или сингълтон-

дефиниция на метод (включително метод на клас).

S ELF В ОПРЕДЕЛЕНИЯТА В КЛАС И МОДУЛ

В дефиниция на клас или модул self е класът или модулният обект. Този невинен

правилото за звучене е важно. Ако го овладеете, ще се спасите от няколко от

най-често срещаните грешки, които хората правят, когато учат Ruby.

Можете да видите какво е self на различни нива на дефиниция на клас и / или модул, като използвате

поставя изрично, както е показано в следващия списък.

клас С

поставя "Току-що започнал клас C:"

поставя себе си

модул М

поставя "Вложен модул C :: M:"

поставя себе си

край

поставя „Обратно във външното ниво на C:“

поставя себе си

край

Веднага след като преминете граница на ключова дума за клас или модул, класът или модулът, чийто

дефиниционният блок, който сте въвели - обектът Клас или Модул - става самостоятелен. Списък-

извод 5.1 показва два случая: въвеждане на C и след това въвеждане на C :: M. Когато напуснете C :: M, но

все още са в C, самостоятелно е отново C.

Разбира се, блоковете за дефиниране на класове и модули правят нещо повече от само начало и край.

Те също така съдържат дефиниции на методи, които както за методите на екземпляра, така и за класа

методи, има правила за определяне на себе си.

Листинг 5.1 Изследване на себе си чрез повиквания към **поставя** в дефиниции на клас и модул

**Изход: C**

**Изход: C :: M**

**Изход: C**

|  |
| --- |
| **Страница 124** |

**124**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

S ЕЛФ НА ИНСТАНЦИЯ - ОПРЕДЕЛЕНИЯ НА МЕТОДА

Идеята за себе си в дефиницията на метод на екземпляр е фина, за следващата реакция

син: когато интерпретаторът срещне def / end блок, той дефинира метода незабавно

наскоро. Но кодът в дефиницията на метода не се изпълнява по-късно, когато

обект, способен да задейства изпълнението му, получава съответното съобщение.

Когато разглеждате дефиниция на метод на хартия или на екрана, можете

знам само по принцип, че когато методът бъде извикан, self ще бъде обектът, който

го нарече (получателят на съобщението). По времето, когато методът се дефинира,

най-много можете да кажете, че самостоятелно в този метод ще бъде някакъв бъдещ обект, който извиква

Методът.

Можете да изготвите метод, който да ви покаже себе си, докато работи:

клас С

def x

поставя "Клас C, метод x:"

поставя себе си

край

край

c = C. нов

cx

поставя „Това беше повикване към x от: # {c}“

Този фрагмент извежда

Клас C, метод x:

# <C: 0x00000101b381a0>

Това беше обаждане до x от: # <C: 0x00000101b381a0>

Странно изглеждащият елемент в изхода (# <C: 0x00000101b381a0>) е начинът на Руби за

казвайки „екземпляр на C.“ (Шестнадесетичното число след дебелото черво е памет-

референция за местоположение. Когато стартирате кода на вашата система, вероятно ще получите различно

както се вижда, приемникът на съобщението “x”, а именно c, поема

ролята на себе си по време на изпълнение на x.

S елф SINGLETON - МЕТОДИ И КЛАС - НАЧИН ОПРЕДЕЛЕНИЯ

Както можете да очаквате, когато се изпълнява сингълтон метод, self е обектът, който притежава

метода, тъй като обектът лесно ще ви каже:

obj = Object.new

def obj.show\_me

поставя "Вътре в единичен метод show\_me на # {self}"

край

obj.show\_me

поставя „Обратно от обаждане до show\_me от # {obj}“

Резултатът от този пример е както следва:

Вътре в единичен метод show\_me на # <Обект: 0x00000101b19840>

Обратно от обаждането до show\_me от # <Обект: 0x00000101b19840>

Има смисъл, че ако методът е написан, за да бъде извикан само от един обект, този обект

става себе си. Освен това, сега е подходящ момент да запомните методите на класа - които са,

|  |
| --- |
| **125. стр** |

**125**

***Разбиране на self, текущия / обект по подразбиране***

по същество, единични методи, прикрепени към обекти от клас. Следните примерни отчети

на самостоятелно от вътре клас метод на C:

клас С

def Cx

поставя "Метод на клас от клас С"

поставя "self: # {self}"

край

край

Cx

Ето какво съобщава:

Метод на клас от клас С

аз: C

Разбира се, обектът е самостоятелен метод в единичен метод (в този случай метод на клас)

чийто единичен метод е.

Да бъдеш себе си в даден момент от програмата идва с някои привилегии. Началникът

привилегията, която се ползва от себе си, е да служи като приемник на съобщения по подразбиране, като

ще видите следващото.

Използване на self вместо твърдо кодирани имена на класове

Като малък съвет за програмиране, ето вариант на последния пример:

клас С

def self.x

поставя "Метод на клас от клас С"

поставя "self: # {self}"

край

край

Обърнете внимание на използването на self.x B, а не Cx Този начин на писане на метод на клас е необходим

предимството на факта, че в дефиницията на класа self е C. Така че def self.x е

същото като def Cx

Версията self.x предлага леко предимство: ако някога решите да преименувате

class, self.x ще се приспособи автоматично към новото име. Ако твърдо кодирате Cx,

ще трябва да смените C с новото име на вашия клас. Но трябва да внимавате.

Не забравяйте, че самостоятелно в метода винаги е обектът, върху който е бил методът

Наречен. Можете да влезете в ситуация, в която се чувствате като себе си трябва да бъде един клас

обект, но всъщност е друг:

клас D <C

край

Dx

D трябва да извика x, тъй като подкласовете трябва да извикат методите на класа на техните супер-

класове. Както ще видите дали стартирате кода, методът Cx отчита себе си - правилно -

като D, защото е D, на който се извиква методът.

б

|  |
| --- |
| **Страница 126** |

**126**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

***5.1.4***

***Self като приемник на съобщения по подразбиране***

Методите за извикване (т.е. изпращането на съобщения до обекти) обикновено включват точковото обозначение:

obj.talk

билет.приход

"abc" .капитализиране

Това е нормалната, пълна форма на синтаксиса за извикване на методи в Ruby. Но специално правило

управлява извикванията на методите: ако получателят на съобщението е самостоятелен, можете да пропуснете приемника

и точката. Ruby ще използва self като приемник по подразбиране, което означава съобщението, което изпращате

ще бъдат изпратени на себе си, както показват следните еквивалентности:

говоря

място на провеждане

капитализиране

ПРЕДУПРЕЖДЕНИЕ Можете да дадете на метод и локална променлива едно и също име, но е така

рядко, ако някога е добра идея. Ако едновременно метод и променлива от дадено име

съществуват и използвате голия идентификатор (като talk), променливата отнема

денс. За да принудите Ruby да вижда идентификатора като име на метод, ще трябва

използвайте self.talk или извикайте метода с празен списък с аргументи: talk ().

Тъй като променливите не вземат аргументи, скобите установяват, че вие

означава метод, а не променлива. Отново, най-добре е да ги избягвате

име сблъсъци, ако можете.

Нека да видим тази концепция в действие, като предизвикам ситуация, в която знаем какво е аз и

след това тестване на безпроблемната форма на извикване на метод. В най-горното ниво на дефиниция на клас

блок, self е обектът на класа. И ние знаем как да добавяме методи директно към класа

обекти. Така че имаме съставките, за да направим демонстрация на приемник по подразбиране:

клас С

def C.no\_dot

поставя "Докато self е C, можете да извикате този метод без точка"

край

no\_dot

край

C.no\_dot

Първото повикване към no\_dot B няма изричен приемник; това е просто слово. Кога

Руби вижда това (и определя, че това е извикване на метод, а не променлива или ключ-

дума), това означава, че го разбирате като стенография за

self.no\_dot

и съобщението се отпечатва. В случая с нашия пример self.no\_dot ще бъде

същото като C.no\_dot, защото сме вътре в блока за дефиниция на C и следователно self е C.

Резултатът е, че се извиква методът C.no\_dot и ние виждаме резултата.

Вторият път, когато извикаме метода c , се връщаме извън дефиницията на класа

блок, така че C вече не е самостоятелно. Следователно, за да извикаме no\_dot, трябва да посочим получателя: C.

**Същото като self.talk**

**Същото като self.venue**

**Същото като self.capitalize**

б

° С

|  |
| --- |
| **Страница 127** |

**127**

***Разбиране на self, текущия / обект по подразбиране***

Резултатът е второ извикване на no\_dot (макар и с точка) и ново отпечатване на

изход от този метод.

Най-честото използване на извикване на безпроблемен метод се случва, когато се обаждате на такъв

екземпляр метод от друг. Ето пример:

клас С

def x

поставя "Това е метод 'x'"

край

def y

поставя "Това е метод 'y', който ще извика x без точка."

х

край

край

c = C. нов

cy

Изходът е

Това е метод 'y', който ще извика x без точка.

Това е метод 'x'.

При извикване на cy, методът y се изпълнява, със самостоятелно задаване на c (което е екземпляр на

° С). Вътре y референтната дума на x се интерпретира като съобщение, което трябва да бъде изпратено до себе си.

Това от своя страна означава, че методът x е изпълнен.

Има една ситуация, при която не можете да пропуснете обекта плюс точка част от метод

повикване: когато името на метода завършва със знак за равенство - с други думи метод на *задаване* .

Трябва да направите self.venue = "Town Hall", а не venue = "Town Hall", ако искате

за да извикате метода venue = on self. Причината е, че Руби винаги интерпретира

*идентификатор на* последователност *= стойност* като присвояване на локална променлива. За да извикате метода

venue = на текущия обект, трябва да включите изричното себе си. В противен случай вие

завършват с променлива, наречена място и без извикване на метода на задаване.

По подразбиране самостоятелно като приемник за безпроблемно извикване на метод ви позволява да стриймвате

подредете добре кода си в случаите, когато един метод използва друг. Често срещан

case съставя цяло име от неговите компоненти: първо, незадължително средно и последно.

Следващият списък показва техника за това, като се използват атрибути за трите

стойности на имената и условна логика, за да се включи средното име, плюс интервал след това, ако

и само ако има бащино име.

клас Личност

attr\_accessor: first\_name,: middle\_name,: last\_name

def цялото\_име

n = първо име + ""

n <<"# {средно\_име}", ако средно\_име

n << фамилно име

край

край

david = Person.new

david.first\_name = "Дейвид"

Листинг 5.2 Съставяне на цялото име от стойности, като се използват извиквания на метод за имплицитно самостоятелно

б

|  |
| --- |
| **Страница 128** |

**128**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

david.last\_name = "Черен"

поставя „Цялото име на Дейвид: # {david.whole\_name}“

david.middle\_name = "Алън"

поставя „новото цяло име на Дейвид: # {david.whole\_name}“

Резултатът от извикващия код в този списък е както следва:

Цялото име на Дейвид: Дейвид Блек

Цялото ново име на Дейвид: Дейвид Алън Блек

Дефиницията на whole\_name зависи от извикванията на метода bareword към first\_name,

средно и фамилно име се изпращат на себе си - себе си като екземпляр на Личност

(Дейвид в примера). Променливата n служи като акумулатор на низове, като компонентът

имена на името, добавено към него едно по едно. Връщаната стойност на целия метод е n,

тъй като изразът n << last\_name B има ефекта на добавяне на last\_name към n

и връщане на резултата от тази операция.

В допълнение към това, че автоматично служи като приемник за съобщения без говор, себе си също

се радва на привилегията да бъде собственик на променливи на екземпляра.

***5.1.5***

***Разрешаване на променливи на екземпляра чрез самостоятелно***

Просто правило управлява променливите на инстанцията и тяхната резолюция: всяка променлива на екземпляра

някога ще видите, че в Ruby програма принадлежи на какъвто и да е обект в момента

(самостоятелно) в този момент от програмата.

Ето един класически случай, когато това знание е полезно. Вижте дали можете да разберете

разберете какво ще отпечата този код, преди да го стартирате:

клас С

def show\_var

@v = "Аз съм променлива на екземпляр, инициализирана в низ."

поставя @v

край

@v = "Променливите на инстанцията могат да се появят навсякъде ...."

край

C.new.show\_var

Кодът отпечатва следното:

Аз съм променлива на екземпляр, инициализирана в низ.

Капанът е, че може да си мислите, че ще отпечата „Променливите на инстанцията могат да се появят всякакви-

къде .... "Кодът отпечатва това, което прави, защото @v в дефиницията на метода B

и @v извън него c са напълно несвързани помежду си. И двамата са инстанция

променливи и двете са наречени @v, но те не са една и съща променлива. Те принадлежат на

различни обекти.

Чии са те?

Първият @v B се намира вътре в дефиниционния блок на екземплярния метод на C. Този факт

има значение не за един обект, а за екземпляри на C като цяло: всеки екземпляр

на C, който извиква този метод, ще има своя собствена променлива @v.

Вторият @v c принадлежи на обекта клас C. Това е един от многото

места, където си струва да се помни, че класовете са обекти. Всеки обект може да има свой собствен

б

° С

|  |
| --- |
| **Страница 129** |

**129**

***Определяне на обхвата***

променливи на екземпляра - собствена скрита информация и състояние на обекта. Обекти на класа

наслаждавайте се на тази привилегия колкото всеки друг обект.

Отново логиката, необходима, за да разберем какъв обект притежава дадена променлива на екземпляра е

просто и последователно: всяка променлива на екземпляра принадлежи на какъвто и да е обект, който играе

роля на self в момента, в който се изпълнява кодът, съдържащ променливата на инстанцията.

Нека направим бързо пренаписване на примера, като този път го направим малко по-бърборещ

какво става. Следният списък показва пренаписването.

клас С

поставя "Точно вътре в блока за дефиниране на клас. Ето аз:"

р себе си

@v = "Аз съм променлива на екземпляр в най-горното ниво на тялото на класа."

поставя "И ето променливата на екземпляра @v, принадлежаща на # {self}:"

p @v

def show\_var

поставя "Вътре в блок за дефиниране на метод на екземпляр. Ето себе си:"

р себе си

поставя "И ето променливата на екземпляра @v, принадлежаща на # {self}:"

p @v

край

край

c = C. нов

c.show\_var

Резултатът от тази версия е както следва:

Точно вътре в блока за дефиниция на клас. Ето себе си:

° С

И тук е променливата на екземпляра @v, принадлежаща на C:

„Аз съм променлива на екземпляр в най-горното ниво на тяло на клас.“

Вътре в блок за дефиниция на метод на екземпляр. Ето себе си:

# <C: 0x00000101a77338>

И тук е променливата на екземпляра @v, принадлежаща на # <C: 0x00000101a77338>:

нула

Разбира се, всеки от тези два различни обекта (обектът на класа C и инстанцията

на C, c) има своя собствена променлива @v. Фактът, че @v на инстанцията е нула, демонстрира

показва, че присвояването на @v на класа няма нищо общо с @v на екземпляра.

Разбиране на себе си - както основният факт, че такава роля играят някои

обект във всяка точка на програма и знаейки как да разберете кой обект е себе си - е едно

от най-важните аспекти на разбирането на Руби. Друг също толкова важен аспект е

постоянен обхват, към който ще се обърнем сега.

***5.2***

***Определяне на обхвата***

*Обхватът се* отнася до обхвата или видимостта на идентификаторите, по-специално променливите и константите.

Различните видове идентификатори имат различни правила за обхват; използвайки, да речем, идентификатора x

за локална променлива във всяка от двете дефиниции на метод има различен ефект от използването

глобалната променлива $ x на същите две места, защото локалните и глобалните променливи се различават

Листинг 5.3 Демонстриране на връзката между променливите на екземпляра и self

|  |
| --- |
| **Страница 130** |

**130**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

по отношение на обхвата. В този раздел ще разгледаме три типа променливи: глобални, локални и

променливи на класа. (Както току-що видяхте, променливите на екземпляра са по-скоро самообвързани, отколкото

ще обсъдим и правилата за разрешаване на константи.

Аз и обхват са сходни по това, че и двете се променят в хода на програмата,

и по това можете да разберете какво се случва с тях, като прочетете и програмата

като го изпълнявате. Но обхватът и себе си не са едно и също нещо. Можете да започнете нов локален обхват

без да се променя, но понякога обхватът и себе си се променят заедно. Те са в

честият факт е, че и двамата са необходими, за да разберат какво върви вашият код

да направя. Като да знаете кой обект е аз, знаейки в какъв обхват сте, ви казва

значение на кода.

Нека започнем с глобални променливи - не най-често използваната конструкция, но

важно за схващане.

***5.2.1***

***Глобален обхват и глобални променливи***

*Глобален обхват* е обхват, който обхваща цялата програма. Глобалният обхват се радва на глобалния

променливи, които се разпознават по първоначалния им знак за долар ($). Те са

налични навсякъде. Те минават през стени: дори ако започнете нов клас или метод

дефиниция, дори ако самоличността на себе си се промени, глобалните променливи, които сте инициализирали

все още са достъпни за вас.

С други думи, глобалните променливи никога не излизат извън обхвата. (Изключение от това е

„Локални глобални нишки“, които ще срещнете в глава 14.) В този пример метод

дефиниран вътре в тяло с дефиниция на клас (два обхвата, отстранени от външната или горната

обхват на нивото на програмата) има достъп до глобална променлива, инициализирана отгоре:

$ gvar = "Аз съм глобален!"

клас С

def exam\_global

поставя $ gvar

край

край

c = C. нов

c.examine\_global

$ Gvar ще ви каже, без никакви несигурни думи, „Аз съм глобален!“ Ако промените всички

появата на $ gvar на неглобална променлива, като local\_var, ще видите, че горната

ниво local\_var не е в обхвата в блока за дефиниране на метод.

B UILT - В ГЛОБАЛНИ ПРОМЕНЛИВИ

Интерпретаторът на Ruby стартира с доста голям брой глобални променливи, които вече са започнали

тиализиран. Тези променливи съхраняват информация, която е потенциално полезна навсякъде и всеки

където във вашата програма. Например глобалната променлива $ 0 съдържа името на

стартиращ файл за текущо изпълняваната програма. Глобалният $: (знак за долар, последван от

двоеточие) съдържа директориите, които съставляват пътя, който Ruby търси, когато зареждате

външен файл. $$ съдържа идентификатора на процеса на процеса Ruby. И има още.

**Изход:**

**"Аз съм глобален!"**

|  |
| --- |
| **Страница 131** |

**131**

***Определяне на обхвата***

СЪВЕТ Добро място да видите описания на всички вградени глобални променливи, които сте

вероятно ще се нуждае - и след това малко - е файлът English.rb във вашата инсталация на Ruby-

ция. Този файл предоставя по-малко загадъчни имена на известния загадъчен глобален

набор от променливи. (Не обвинявайте Руби за имената - повечето от тях идват от

shell езици и / или Perl и awk.) Ако искате да използвате малко по-приятелски

имена във вашите програми, можете да изисквате "английски", след което можете

вижте $ IGNORECASE вместо $ =, $ PID вместо $$ и т.н. Няколко

глобалните имена имат предварително заредени имена на английски език; можете да кажете $ LOAD\_PATH

вместо $: дори без зареждане на English.rb.

Създаването на ваши собствени глобални променливи може да бъде примамливо, особено за начинаещи

граматисти и хора, които учат нов език (не само Ruby, също). Но това е

рядко добър или подходящ избор.

П РОС И ПРОТИВИ НА ГЛОБАЛНИ ПРОМЕНЛИВИ

Глобалите изглежда решават много дизайнерски проблеми: не е нужно да се притеснявате за обхвата,

и множество класове могат да споделят информация, като я скриват в глобални, а не в дизайнерски

обекти, които трябва да бъдат заявени с извиквания на методи. Без съмнение глобални променливи

имат известна привлекателност.

Но те се използват много малко от опитни програмисти. Причините за избягване

те са подобни на причините, поради които са изкушаващи. Използването на глобални променливи има тенденция да свършва

като заместител на солиден, гъвкав дизайн на програмата, вместо да допринася за него.

Една от основните точки на обектно-ориентираното програмиране е, че данните и действията са такива

капсулирани в обекти. Предполага се, *че* трябва да потърсите обекти за информация и

да поискат те да извършват действия.

И обектите трябва да имат известна поверителност. Когато поискате даден обект да направи

нещо, не трябва да ви интересува какво прави обектът вътрешно, за да получите работата

Свършен. Дори ако сами сте написали кода за методите на обекта, когато изпращате

обект на съобщение, третирате обекта като черна кутия, която работи зад кулисите и

дава отговор.

Глобалните променливи изкривяват пейзажа, като предоставят слой от споделена информация

от всеки обект във всеки контекст. Резултатът е, че обектите спират да говорят помежду си

и вместо това споделяйте информация, като задавате глобални променливи.

Ето един малък пример - пренаписване на по-ранния ни клас Person (този с

първо, незадължително средно и фамилно име). Този път, вместо атрибути на обекта,

ще генерираме цялото име от глобални:

клас Личност

def цялото\_име

n = $ first\_name + ""

n <<"# {$ middle\_name}", ако $ middle\_name

n << $ last\_name

край

край

|  |
| --- |
| **132 стр** |

**132**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

За да използвате този клас и за да получите цяло име от негов екземпляр, трябва да направите това:

david = Person.new

$ first\_name = "Дейвид"

$ middle\_name = "Алън"

$ last\_name = "Черен"

поставя david.whole\_name

Тази версия все още извлича цялото име, отвън, чрез заявка за обекта. Но

компонентите на името се връчват над главите на предметите, така че да

говори в отделна мрежа от глобални променливи. Това е кратко и лесно, но също така е и

тически ограничен. Какво би се случило, ако имате много обекти на Личност или ако искате

да запишете обект Person, включително различните му имена, в база данни? Вашият код би

бързо се заплетете, най-малкото.

Данните с глобален обхват в основата си противоречат на обектно-ориентираната философия

phy да дарите предмети със способности и след това да свършите нещата чрез изпращане

заявки към тези обекти. Някои програмисти на Ruby работят години наред и никога не използват греховен

глобална променлива gle (с изключение на някои от вградените). Това може или не

в крайна сметка това е вашият опит, но не е лоша цел, към която да се стремите.

След като приключихме с частта „опитайте се да не правите това“, нека преминем към a

подробно разглеждане на местния обхват.

***5.2.2***

***Местен обхват***

*Локалният обхват* е основен слой от тъканта на всяка програма Ruby. Във всеки един момент,

вашата програма е в определен локален обхват. Основното нещо, което се променя от едно

локален обхват на друг е доставката ви на локални променливи. Когато напуснете местен

обхват - чрез връщане от извикване на метод или чрез извършване на нещо, което задейства ново

местен обхват - получавате нова доставка. Дори ако сте задали на локална променлива x в

един обхват, можете да присвоите нов х в нов обхват и двата х няма да пречат

един с друг.

Можете да разберете, като разгледате програма Ruby, където местните области започват и завършват,

въз основа на няколко правила:

Най-горното ниво (извън всички дефиниционни блокове) има свой собствен локален обхват.

Всеки блок или блок с дефиниция на модул (клас, модул) има свой собствен локален обхват,

дори вложени блокове за дефиниция на клас / модул.

Всяка дефиниция на метод (def) има свой собствен локален обхват; по-точно всяко обаждане

към метод генерира нов локален обхват, като всички локални променливи се нулират на

недефинирано състояние.

Изключения и допълнения към тези правила съществуват, но те са сравнително малко и не касаят

сега.

Фигура 5.2 илюстрира създаването на редица локални обхвати.

Всеки път, когато преминете в блок с клас, модул или метод за дефиниране - всеки

когато преминете през клас, модул или ключова дума def - стартирате нов локален обхват. Местен

**Резултат: Дейвид**

**Алън Блек**

|  |
| --- |
| **133** |

**133**

***Определяне на обхвата***

променливите, които са физически много близо една до друга, всъщност не могат да имат нищо

някога да се правят един с друг, както показва този пример:

клас С

a = 1

def local\_a

a = 2

поставя а

край

поставя а

край

a = 0

def t

край

край

a = 1

def self.x

a = 2

поставя "Cx; a = # {a}"

край

def M

a = 3

поставя "C # m; a = # {a}"

край

поставя "Обхват на класа: a = # {a}"

def n

a = 4

поставя "C # n; a = # {a}"

поставя "метод от най-високо ниво t"

Клас-дефиниция

обхват

Най-високо ниво

(външен обхват)

Метод-дефиниция

обхват

клас С

край

Cx

c = C. нов

см

cn

поставя "Най-високо ниво: a = # {a}"

Фигура 5.2 Схематичен изглед на локални обхвати на най-високо ниво, ниво на дефиниция на класа и метод-

ниво на дефиниция

б

° С

д

|  |
| --- |
| **Страница 134** |

**134**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

c = C. нов

c.local\_a

Този код извежда следния изход:

1

2

Променливата a, която се инициализира в локалния обхват на дефиницията на класа B, е в a

различен обхват от променливата a вътре в дефиницията на метода c . Когато получиш

към оператора put поставя след дефиницията на метода d , вие сте отново в класа-

определение локален обхват; a, което се отпечатва, е a, което инициализирате обратно отгоре B ,

не а, което е в обхвата в дефиницията на метода c . Междувременно второто а не е

отпечатан до по-късно, когато сте създали екземпляра c и сте изпратили съобщението local\_a

към него д .

Когато влагате класове и модули, всяко преминаване в блок с нова дефиниция се създава

дава нов локален обхват. Следващият списък показва някои дълбоки вложения на класове и

модули, с редица променливи, наречени инициализирани и отпечатани заедно

начинът.

клас С

a = 5

модул М

a = 4

модул N

a = 3

клас D

a = 2

def show\_a

a = 1

поставя а

край

поставя а

край

поставя а

край

поставя а

край

поставя а

край

d = C :: M :: N :: D.нови

d.show\_a

Всеки блок за дефиниция - независимо дали е за клас, модул или метод - стартира нов локален

scope - нова табла за писане с локална променлива - и получава собствена променлива a. Този пример

също илюстрира факта, че целият код в блоковете за дефиниране на клас и модул получава

изпълнява се при първото му срещане, докато методите не се изпълняват до обект

се изпраща съответното съобщение. Ето защо стойността на a, която е зададена вътре в show\_a

метод се показва последен сред петте стойности, които програмата отпечатва; останалите четири

Листинг 5.4 Повторно използване на име на променлива в вложени локални обхвати

д

**Изход: 2**

**Изход: 3**

**Изход: 4**

**Изход: 5**

**Изход: 1**

|  |
| --- |
| **Страница 135** |

**135**

***Определяне на обхвата***

се изпълняват, тъй като се срещат в дефинициите на класа или модула, но последните

един не се изпълнява, докато show\_a не бъде изпълнен от обекта d.

Локалният обхват се променя често, както можете да видите. Също и идентичността на себе си. Понякога,

но само понякога те варират заедно. Нека да разгледаме малко по-отблизо връзката

между обхвата и себе си.

***5.2.3***

***Взаимодействието между местния обхват и себе си***

Когато стартирате блок за дефиниция (метод, клас, модул), стартирате нов локален

обхват, а също така създавате блок с код с конкретно себе си. Но местен обхват и

самостоятелно не работят изцяло паралелно, не само защото не са едно и също нещо, но

също и защото не са едно *и* също нещо.

Обмислете следния списък. Тази програма използва *рекурсия* : методът на екземпляра

x се извиква. Въпросът е да се демонстрира, че всеки път, когато даден метод е извикан - дори

ако предишно извикване на метода все още е в процес на стартиране - нов локален обхват

се генерира.

клас С

def x (value\_for\_a, repeatse = false)

a = стойност\_за\_а

print "Ето контролния низ за" self ":"

р себе си

поставя "И ето:"

поставя а

ако се повтори

поставя "Обаждам се (рекурсия) ..."

x ("Втора стойност за a")

поставя "Обратно след рекурсия; ето:"

поставя а

край

край

край

c = C. нов

cx ("Първа стойност за a", вярно)

Методът на екземпляра C # x взема два аргумента: стойност, която се присвоява на променливата a

и знаме казва методът дали да се нарича B . (Използването на флага осигурява

начин за предотвратяване на безкрайна рекурсия.) Първият ред на метода инициализира c и

следващите няколко реда от метода отпечатват низовото представяне на self и

стойност на d .

Сега идва решението: да се повтори или да не се повтори. Зависи от стойността на

рекурсивната променлива e . Ако се случи рекурсията, тя извиква x, без да посочва стойност

за рекурсивния параметър f ; този параметър по подразбиране ще бъде false и рекурсия

няма да се случи втори път.

Рекурсивното извикване използва различна стойност за аргумента value\_for\_a; Следователно,

по време на разговора ще бъде разпечатана различна информация. Но при връщане от

Листинг 5.5 Демонстриране на генерирането на нов локален обхват за извикване на метод

б

° С

д

д

е

ж

з

|  |
| --- |
| **136** |

**136**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

рекурсивното извикване, откриваме, че стойността на a в това изпълнение на x не се е променила g . В

накратко, всяко повикване към x генерира нов локален обхват, въпреки че самостоятелността не се променя.

Резултатът от извикването на x на екземпляр на C и задаване на флага за повторение на true h

изглежда така:

Ето инспекционния низ за „самостоятелно“: # <C: 0x00000101b25be0>

И ето:

Първа стойност за a

Обаждам се (рекурсия) ...

Ето инспекционния низ за „самостоятелно“: # <C: 0x00000101b25be0>

И ето:

Втора стойност за a

Обратно след рекурсия; ето:

Първа стойност за a

Няма промяна в self, но локалните променливи се нулират.

СЪВЕТ Вместо да разпечатвате низовото представяне по подразбиране на обект, вие

може също да използва метода object\_id, за да идентифицира обекта по уникален начин. Опитайте

ing p self поставя self.object\_id и поставя a to поставя a.object\_id в

предишен пример.

Ако този списък изглежда като дълъг път, за да се каже, че всеки метод

повикване има свой собствен локален обхват, мислете за него като за шаблон или модел за видовете демонстрации

страции, които може да опитате сами, докато развивате все по-фино усещане за

как обхватът и самостоятелността работят, поотделно и заедно.

ЗАБЕЛЕЖКА Също така е възможно да се направи обратното на това, което показва списък 5.5 -

а именно да сменя себе си, без да влизам в нов локален обхват. Това е придружено

plish с методите instance\_eval и instance\_exec, които ще направим

погледнете по-късно.

Подобно на променливите, константите се управляват от правила за обхват. След това ще разгледаме как тези

правилата работят.

***5.2.4***

***Обхват и резолюция на константите***

Както видяхте, константите могат да бъдат дефинирани в блокове за дефиниране на клас и метод. Ако

знаете веригата от вложени дефиниции, можете да получите достъп до константа отвсякъде.

Помислете за това гнездо:

модул М

клас С

клас D

модул N

X = 1

край

край

край

край

|  |
| --- |
| **137** |

**137**

***Определяне на обхвата***

Можете да се обърнете към модул M, клас M :: C и т.н., до простия кон-

stant M :: C :: D :: N :: X (което е равно на 1).

Константите имат вид глобална видимост или достижимост: стига да познавате

път към константа през класовете и / или модулите, в които е вложена, можете

стигнете до тази константа. Лишени от гнезденето си, обаче, константите определено не са

глобални. Константата X в един обхват не е константа X в друг:

модул М

клас С

X = 2

клас D

модул N

X = 1

край

край

край

край

поставя M :: C :: D :: N :: X

поставя M :: C :: X

Според гнезденето, първото поставяне B ви дава 1; второто c ви дава 2. Определено

постоянният идентификатор (като X) няма абсолютно значение по начина, по който глобалната променлива

(като $ x) прави.

*Постоянно търсене* - процес на разрешаване на постоянен идентификатор или намиране на правилния

съвпадение за него - наподобява почти търсенето на файлова система за файл в част-

ular директория. От една страна, константите се идентифицират спрямо точката на изпълнение

ция. Друг вариант на нашия пример илюстрира това:

модул М

клас С

клас D

модул N

X = 1

край

край

поставя D :: N :: X

край

край

Тук идентификаторът D :: N :: X се интерпретира спрямо мястото, където се среща: вътре в деф-

начален блок на класа M :: C. От гледна точка на M :: C, D е само на едно ниво.

Няма нужда да правите M :: C :: D :: N :: X, когато само D :: N :: X сочи пътя надолу по

път към дясната константа. Със сигурност получаваме това, което искаме: разпечатка на номера

ber 1.

F ORCING абсолютно ПОСТОЯННА PATH

Понякога не искате относителен път. Понякога наистина искате да започнете

процес на постоянно търсене на най-високо ниво - точно както понякога трябва да използвате абсолютно

лютня път за файл.

б

° С

**Изход: 1**

|  |
| --- |
| **138 серия** |

**138**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

Това може да се случи, ако създадете клас или модул с име, подобно на

име на вграден клас или модул в Ruby. Например, Ruby идва със String

клас. Но ако създадете клас по цигулка, може да имате и Strings:

клас Цигулка

клас String

attr\_accessor: стъпка

def инициализиране (стъпка)

@pitch = стъпка

край

край

def инициализира

@e = String.new ("E")

@a = String.new ("A")

... и т.н ....

Константата String в този контекст B се разрешава до Violin :: String, както е дефинирана. Сега

да кажем, че другаде в цялостната дефиниция за клас Цигулка трябва да се обърнете към

Вграденият в Ruby клас String. Ако имате обикновена препратка към String, той разрешава

Цигулка :: Струна. За да сте сигурни, че се позовавате на вградения, оригинален клас String,

трябва да поставите разделителя на константния път :: (двойно двоеточие) в началото на

име на класа:

деф история

:: String.new (maker + "," + дата)

край

По този начин получавате обект Ruby String вместо обект Violin :: String. Като наклонената черта

в началото на име на път, :: пред константа означава „започнете търсенето на

това на най-високо ниво. " (Да, можете просто да сглобите низа заедно в двойни кавички,

използвайки интерполация и заобикаляйте String.new. Но тогава нямаше да имаме толкова ярки

пример за сблъсък на име!)

В допълнение към константите и локалните, екземпляри и глобални променливи, Ruby също

tures *променливи на класа* , категория идентификатор с някои идиосинкратични правила за обхват.

***5.2.5***

***Синтаксис, обхват и видимост на променливата на класа***

Променливите на класа започват с две по знаци - например @@ var. Въпреки името си, клас

променливите не са обхват на класа. По-скоро те са обхванати от клас-йерархия, с изключение на ... някои-

пъти. Не се притеснявайте; ще разгледаме подробностите. След поглед към това как променливите на класа

работа, ще оценим колко добре те изпълняват ролята на поддържане на състояние за клас.

С ЛАСОВИ ПРОМЕНЛИВИ ПРЕЗ КЛАСОВЕ И ИНСТАНЦИИ

Най-простото, идеята зад променливата на класа е, че тя осигурява механизъм за съхранение

nism, който се споделя между клас и екземпляри от този клас и това не се вижда

към всякакви други обекти. Никой друг обект не може да изпълни тази роля. Локалните променливи не оцеляват

промяна на обхвата между дефинициите на класове и техните вътрешни дефиниции на методи.

Глобалите го правят, но те също са видими и променливи навсякъде другаде в програмата, не

само в един клас. Константи по същия начин: методите на екземпляра могат да виждат дефинираните константи

б

|  |
| --- |
| **139 серия** |

**139**

***Определяне на обхвата***

в класа, в който са дефинирани, но останалата част от програмата може да ги види

постоянства също. Променливите на инстанцията, разбира се, се виждат строго за всеки обект. Клас не е

един и същ обект като всеки от неговите екземпляри и нито един от неговите екземпляри не е същият като

взаимно. Следователно е невъзможно по дефиниция клас да споделя варианта на екземпляр

ables с неговите екземпляри.

Така че променливите на класа имат ниша, която трябва да запълнят: видимост за клас и неговите екземпляри и за не

един друг. Обикновено това означава да бъде видим в дефинициите на метода на класа и

дефиниции на методи, а понякога и на най-високото ниво на дефиницията на класа.

Ето пример: малък тракер за автомобили. Нека започнем с пробно изпълнение и изхода;

след това ще разгледаме как работи програмата. Да кажем, че искаме да регистрираме марките (man-

ufacturer names) на автомобили, което ще направим, използвайки метода на класа Car.add\_make (make).

След като марката е регистрирана, можем да създадем автомобили от тази марка, като използваме Car.new (марка).

Ще регистрираме Honda и Ford и ще създадем две Hondas и една Ford:

Car.add\_make ("Honda")

Car.add\_make ("Форд")

h = Car.new ("Honda")

f = Car.new ("Ford")

h2 = Car.new ("Honda")

Програмата ни казва кои автомобили се създават:

Създаване на нова Honda!

Създаване на нов Форд!

Създаване на нова Honda!

На този етап можем да получим обратно някаква информация. Колко коли има същите

направи като h2? Ще използваме метода на инстанцията make\_mates, за да разберем, интерполирайки

резултат в низ:

поставя "Брой коли със същата марка като h2 ..."

поставя "Има # {h2.make\_mates}."

Както се очакваше, има две коли със същата марка като h2 (а именно Honda).

Колко автомобила има общо? Знанията от този вид се намират в класа,

не в отделните автомобили, затова питаме класа:

поставя "Брой общо коли ..."

поставя "Има # {Car.total\_count}."

Изходът е

Преброяване на общите коли ...

Има 3.

И накрая, ние се опитваме да създадем кола с несъществуваща марка:

x = Car.new ("Марка X")

Програмата не го харесва и получаваме фатална грешка:

car.rb: 21: в `инициализиране ': Няма такава марка: Марка X. (RuntimeError)

|  |
| --- |
| **140** |

**140**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

Основното действие тук е в създаването на автомобили и способността на двата отделни автомобила

и класът Car за съхраняване и връщане на статистика за създадените автомобили.

Следващият списък показва програмата. Ако запазите този списък и след това добавите предишния

примерен код до края на файла, можете да стартирате целия файл и да видите изхода на

кодът.

клас Автомобил

@@ прави = []

@@ автомобили = {}

@@ общо\_броене = 0

attr\_reader: направи

def self.total\_count

@@ общ\_ брой

край

def self.add\_make (make)

освен ако @@ make.include? (make)

@@ прави << направи

@@ автомобили [марка] = 0

край

край

def инициализира (направи)

ако @@ make.include? (make)

поставя "Създаване на нов # {make}!"

@make = направи

@@ автомобили [марка] + = 1

@@ общ\_ брой + = 1

else рейз "Няма такава марка: # {make}."

край

край

def make\_mates

@@ автомобили [самостоятелно направете]

край

край

Ключът към програмата е наличието на трите променливи на класа, дефинирани в горната част

на определение клас на B . @@ make е масив и съхранява имената на марки. @@ автомобили е

на *хеш* : шпонков структура, чиито ключове са марки автомобили, и отговарящите на Val-

ues са броя на броя от всяка марка. И накрая, @@ total\_count е run-

изчисляване на броя на автомобилите, които са създадени като цяло.

Класът Car също има атрибут make четец c , който ни позволява да питаме всяка кола

каква е неговата марка. Стойността на атрибута make трябва да бъде зададена, когато автомобилът е създаден.

Няма атрибут писател за марки автомобили, защото не искаме код извън

клас, променящ марките автомобили, които вече съществуват.

За да осигури достъп до променливата на класа @@ total\_count, класът Car определя a

метод total\_count d , който връща текущата стойност на променливата на класа. Има

също метод на клас, наречен add\_make e ; този метод взема един аргумент и добавя

тя към масива от известни марки автомобили, използвайки оператора << добавяне на масив. Първо

взема предпазните мерки, като се увери, че масивът от марки вече не включва това

Листинг 5.6 Проследяване на статистиката за производството на автомобили с променливи на класа

б

° С

д

д

е

ж

з

i

|  |
| --- |
| **141** |

**141**

***Определяне на обхвата***

конкретна марка. Ако приемем, че всичко е наред, той добавя марката и също така задава брояча

това прави колата равна на нула. По този начин, когато регистрираме марката Honda, ние също установяваме

фактът, че съществуват нула Хонди.

Сега стигаме до метода за инициализиране, където се създават нови автомобили. Всяка нова кола

се нуждае от марка. Ако марката не съществува (тоест, ако не е в масива @@ make), тогава

повдигаме фатална грешка h . Ако марката съществува, тогава задаваме атрибут make на тази кола

подходящата стойност f , увеличете с една броя на колите от тази марка, които са

записан в @@ hash g , и също така се увеличава с един общият брой на съществуващите-

ing автомобили, съхранявани в @@ total\_count. (Може да предположите, че @@ total\_count представлява

изпраща общата стойност на всички стойности в @@ автомобили. Съхраняването на общото количество ни спестява

проблем със събирането на всички стойности всеки път, когато искаме да видим общата сума.) Има и

внедряване на метода на екземпляра make\_mates i , който връща списък на всички автомобили

на дадена марка автомобил.

Методът за инициализиране използва интензивно променливите на класа, дефинирани отгоре,

външно ниво на дефиницията на класа - напълно различен локален обхват от вътрешността на

инициализиране, но не различно за целите на видимостта на променливи клас. Тези вариации на класа

ables бяха използвани и в методите на класа Car.total\_count и Car.add\_make - всеки

от които също има свой собствен локален обхват. Можете да видите, че променливите на класа следват своите

правила: тяхната видимост и обхват не се изравняват с тези на локалните променливи и те намаляват

в множество ценности на себе си. (Не забравяйте, че на външното ниво на определение на клас

а вътре в методите на класа self е обектът на класа - Car - докато в екземпляра

методи, self е екземпляр на Car, който извиква метода.)

Досега сте виждали най-простите аспекти на променливите на класа. Дори на това ниво,

йони се различават по въпроса дали или поне колко често е добре да създадете променливи, които

изрежете този път през множество самостоятелни обекти. Фактът, че една кола е пример за кола

наистина означава, че обектът на автомобила и обектът от клас Car трябва да споделят данни? Или трябва

да бъдат третирани навсякъде като двата отделни обекта, които са?

Няма еднозначен (или прост) отговор. Но има малко повече за това как класът варира-

ables работят; и най-малкото, вероятно ще заключите, че те трябва да бъдат

преследван с внимание.

С ЛАСОВИ ПРОМЕНИ И КЛАСОВАТА ИЕРАРХИЯ

Както беше отбелязано по-рано, променливите на класа не са променливи с обхват на класа. Те са клас-йерархия-

обхватни променливи.

Ето един пример. Какво бихте очаквали да отпечата следният код?

клас Родител

@@ стойност = 100

край

клас Дете <Родител

@@ стойност = 200

край

клас Родител

поставя @@ стойност

край

**Задава променлива на класа**

**в клас Родител**

**Задава променлива на класа в клас**

**Дете, подклас на родител**

**Обратно в родителски клас:**

**какъв е изходът?**

|  |
| --- |
| **142** |

**142**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

Това, което се отпечатва, е 200. Класът Child е подклас на Parent и това означава

Родител и дете споделят едни и същи променливи на класа - не различни променливи на класа с

същите имена, но същите действителни променливи. Когато присвоявате на @@ стойност в Child,

задавате единствената променлива @@ value, която се споделя в целия йерархичен

chy - тоест от родител и дете и други класове потомци на всеки от тях.

Терминът *променлива клас* става малко трудно да се примири с факта, че две (и

потенциално много повече) класовете споделят абсолютно същите.

Както обещахме, ще завършим този раздел с разглеждане на плюсовете и минусите на

използване на променливи на класа като начин за поддържане на състояние в клас.

E ОЦЕНЯВАНЕ НА Плюсовете и минусите на променливите от класа

Начинът за поддържане на състояние в обект е променливата на екземпляра. Клас

променливите са полезни, защото разбиват язовира между обект на клас и

екземпляри от този клас. Но по този начин и особено поради тяхната йерархична основа

обхват, те придобиват вид квазиглобално качество: променливата на класа не е глобална, но е

сигурно се вижда от много обекти, след като добавите всички подкласове и всички

копия на тези подкласове.

Въпросът е, че е полезно да имате начин да поддържате състояние в клас. Вие

видях това дори в простия пример за клас Car. Искахме някъде да скрием клас-

подходяща информация, като марки автомобили и общия брой произведени автомобили

туриран. Ние също искахме да получим тази информация, както от методите на класа, така и от

екземплярни методи. Променливите на класа са популярни, защото те са най-лесният начин да

данни за данък в тази конфигурация.

Но те също са спукани. Твърде много обекти могат да се доберат до тях. Да кажем, че искахме

да се създаде подклас на автомобила, наречен Hybrid, за да се отчита броят на произведените (частично)

електрически превозни средства. Не можахме да направим това:

клас Хибрид <Лек автомобил

край

hy = Hybrid.new ("Honda")

поставя "Съществуват # {Hybrid.total\_count} хибриди!"

защото Hybrid.total\_count е същият метод като Car.total\_count и се увива

същата променлива. Променливите на класа не се преиздават прясно за всеки подклас, по начина

примерни променливи са за всеки обект.

За да проследяваме хибридите поотделно, ще трябва да направим нещо подобно:

клас Хибрид <Лек автомобил

@@ total\_hybrid\_count = 0

# и т.н.

край

Въпреки че има начини за абстрактно и полуавтоматизиране на този вид разделяне на кода

по пространство на имена на класове, това не е най-лесната или най-прозрачна техника в света.

Каква е алтернативата?

|  |
| --- |
| **143** |

**143**

***Определяне на обхвата***

М AINTAINING PER - КЛАС състояние с съд ПРОМЕНЛИВИ от клас ОБЕКТИ

Алтернативата е да се върнете към основите. Нуждаем се от слот, където можем да поставим стойност (общата

count) и това трябва да е различен слот за всеки клас. С други думи, ние трябва да поддържаме

tain състояние за всеки клас; и тъй като класовете са обекти, това означава за всеки обект

основа (за определена група обекти, а именно обекти от клас). И по състояние на обект, независимо дали

въпросният обект е клас или нещо друго, предполага променливи на екземпляра.

Следният списък показва пренаписване на класа Car в списък 5.6. Две от варианта на класа

ables все още са там, но @@ total\_count е трансформиран в променлива на екземпляр.

клас Автомобил

@@ прави = []

@@ автомобили = {}

attr\_reader: направи

def self.total\_count

@total\_count || = 0

край

def self.total\_count = (n)

@total\_count = n

край

def self.add\_make (make)

освен ако @@ make.include? (make)

@@ прави << направи

@@ автомобили [марка] = 0

край

край

def инициализира (направи)

ако @@ make.include? (make)

поставя "Създаване на нов # {make}!"

@make = направи

@@ автомобили [марка] + = 1

self.class.total\_count + = 1

друго

рейз "Няма такава марка: # {марка}."

край

край

def make\_mates

@@ автомобили [самостоятелно направете]

край

край

Ключът тук е съхраняване на брояча в променлива на екземпляр, принадлежаща на обекта на класа

Car и опаковане на тази променлива на екземпляра в методи за достъп - ръчно написани,

но методи за достъп въпреки това. Методите за достъп са Car.total\_count и

Car.total\_count =. Първият от тях изпълнява задачата за инициализиране на @total\_count

до нула B . Той прави инициализацията условно, използвайки оператора or-equals, така че

при второто и следващите извиквания на total\_count, стойността на променливата на екземпляра

просто се връща.

Методът total\_count = е метод за писане на атрибути, също написан като клас

метод, така че обектът, чиято променлива на екземпляр се използва, е обектът на класа c . С

Листинг 5.7 **Автомобил** с **@@ total\_count** заменен от променлива на **екземпляр @total\_count**

б

° С

д

|  |
| --- |
| **144** |

**144**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

тези методи на място, вече можем да увеличим общия брой от

екземпляр метод инициализира чрез извикване на self.class.total\_count = d .

Изплащането идва, когато подклас Car. Нека да разгледаме още един хибрид и

някакъв примерен код, който го използва:

клас Хибрид <Лек автомобил

край

h3 = Hybrid.new ("Honda")

f2 = Hybrid.new ("Ford")

поставя "На път има # {Hybrid.total\_count} хибриди!"

Хибридът е нов обект от клас. Това не е същият обект като Car. Следователно тя има своя собствена

инстанция променливи. Когато създаваме нов хибриден екземпляр, методът за инициализиране

от Car се изпълнява. Но този път, изразът

self.class.total\_count + = 1

има различно значение. Получателят на съобщението “total\_count =” е хибриден, а не

Кола. Това означава, че когато се изпълни методът total\_count = class, инстанцията варира

able @total\_count принадлежи на Hybrid. (Променливите на инстанцията винаги принадлежат на себе си.) Add-

следователно общият брой на хибридите няма да повлияе на общия брой на автомобила.

Направихме така, че подклас Car да може да поддържа собственото си състояние, защото сме го направили

изместен от променлива на клас към променлива на екземпляр. Всеки път total\_count или

total\_count = се извиква, @total\_count, за който се отнася, е този, който принадлежи

себе си в този момент на изпълнение. За пореден път се върнахме в бизнеса, използвайки вариант на

може да поддържа състояние на обект (в този случай обекти от клас).

Най-голямата пречка за разбирането на тези примери е разбирането на факта

че класовете са обекти - и че всеки обект, независимо дали е кола, човек *или клас* ,

получава своя собствена скривалище от променливи на екземпляра. Автомобилът и хибридът могат да следят човешкия

ufacturing номера поотделно, благодарение на начина, по който променливите на екземпляра са поставени под карантина

на обект.

Достигнахме лимита на пътуването си с обхвата на идентификатора. Виждали сте много от

какви променливи и константи могат да правят (и какво не могат) и как тези способности

са обвързани с правилата, регулиращи обхвата и себе си. В интерес на изпълнението на

Целта на главата е да ви покаже как да се ориентирате по отношение на това кой какво да прави,

и където в кода на Ruby ще разгледаме още една основна подтема: системата на Ruby на

правила за достъп до метод.

***5.3***

***Внедряване на правила за достъп до метод***

Както видяхте, основният бизнес на Ruby програма е да изпраща съобщения до обекти.

И основният бизнес на обекта е да отговаря на съобщения. Понякога обект

иска да може да си изпраща съобщения, които не иска никой друг да може

изпрати го. За този сценарий Ruby предоставя възможността да направи метод частен.

Има две нива на достъп, различни от частно: защитено, което е малка вариация

на частни и публични. Public е нивото на достъп по подразбиране; ако не посочите, че a

методът е защитен или частен, той е публичен. Общите методи на публична инстанция са често срещани

**Изход: Има**

**са 2 хибрида**

**на пътя!**

|  |
| --- |
| **Страница 145** |

**145**

***Внедряване на правила за достъп до метод***

валута на програмирането на Ruby. Повечето съобщения, които изпращате на обекти, се обаждат

публични методи.

Тук ще се съсредоточим върху методи, които не са публични, като започнем с частни методи.

***5.3.1***

***Частни методи***

Мислете за обект като за човек, когото поискате да изпълни задача вместо вас. Да кажем, че питате някои-

един да ви изпече торта. По време на печенето на торта, пекарят вероятно ще го направи

изпълнявайте много малки задачи: измервайте захар, напуквайте яйце, разбърквайте тесто и т.н.

Пекарят прави всички тези неща, но не всички имат равен статут, когато го правят

стига до това, което пекарят е готов да направи в отговор на искания от други хора. То

би било странно, ако се обадите на пекар и кажете: „Моля, разбъркайте малко тесто“ или „Моля

напукайте яйце. " Това, което казвате, е „Моля, изпечете ми торта“ и оставете пекаря да се справи

подробностите.

Нека моделираме сценария на печене. За някои ще използваме минимални, запазени класове

на обектите в нашия домейн, но ще разработим по-подробно класа Baker.

Запазете кода в следващия списък във файл, наречен baker.rb.

клас Торта

def инициализиране (тесто)

@batter = тесто

@baked = вярно

край

край

клас Яйце

край

клас Брашно

край

клас Бейкър

def bake\_cake

@batter = []

налейте\_брашно

add\_egg

разбъркайте\_батер

return Cake.new (@batter)

край

def pour\_flour

@ batter.push (Flour.new)

край

def add\_egg

@ batter.push (Egg.new)

край

def stir\_batter

край

private: pour\_flour,: add\_egg,: stir\_batter

край

В този код има нещо ново: частният метод B , който приема като аргументи

списък с методите, които искате да направите частни. (Ако не предоставите никакви аргументи,

Листинг 5.8 **Бейкър** и други класове на домейн за печене

**Прилага @batter като масив**

**на обекти (съставки)**

**Връща ново**

**Обект за торта**

**Добавя елемент (съставка)**

**към @batter**

б

|  |
| --- |
| **146** |

**146**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

извикване на частни действия като превключвател за включване: всички методи на екземпляра, които дефинирате под него,

докато не обърнете ефекта, като се обадите на публичен или защитен, ще бъде частен.)

Private означава, че методът не може да бъде извикан с изричен приемник. Не можеш да кажеш

b = Baker.new

b.add\_egg

Както ще видите, ако опитате, извикването на add\_egg по този начин води до фатална грешка:

`<main>': частен метод` add\_egg', извикан за # <Baker: 0x00000002aeae50>

(NoMethodError)

add\_egg е частен метод, но изрично сте посочили получаващият обект b.

Това не е позволено.

Добре; да се примирим с правилата. Няма да посочим приемник. Само ще кажем

add\_egg

Но почакай. Можем ли да извикаме add\_egg изолирано? Къде ще отиде съобщението? Как може a

метод да бъде извикан, ако няма обект, обработващ съобщението?

Малко детективска работа ще отговори на този въпрос.

Ако не използвате изричен приемник за извикване на метод, Руби приема, че искате

за да изпратите съобщението до текущия обект, самостоятелно. Мислейки логично, можете да заключите

че съобщението add\_egg има обект, към който да отиде, само ако self е обект, който отговаря

към add\_egg. С други думи, можете да извикате само метода add\_egg на Baker

когато self е екземпляр на Baker.

А кога самостоятелно е екземпляр на Бейкър?

Когато се изпълнява който и да е метод на Baker. Вътре в дефиницията на

bake\_cake, например, можете да се обадите на add\_egg и Ruby ще знае какво да прави. Кога-

винаги, когато Ruby удари това извикване на add\_egg в дефиницията на този метод, той изпраща съобщението

add\_egg към себе си и self е обект на Baker.

Това се свежда до това: като маркирате add\_egg като частен, вие казвате обекта Baker

получава да си изпрати това съобщение (пекарят може да си каже да добави яйце

до тестото), но никой друг не може да изпрати съобщението до пекаря (вие, като външен човек,

не може да каже на пекаря да добави яйце към тестото). Ruby налага тази поверителност чрез

Private и singleton са различни

Важно е да се отбележи разликата между частен метод и сингълтон

метод. Единичният метод е „частен“ в свободен, неформален смисъл, че принадлежи

само за един обект, но той не е частен в технически смисъл. (Можете да направите едно-

ton метод private, но по подразбиране не е.) Частен, не-единичен метод на екземпляр,

от друга страна, може да се споделя от произволен брой обекти, но може да се извиква само

при правилните обстоятелства. Какво определя дали можете да се обадите на частен

метод не е обектът, на който изпращате съобщението, а кой обект е самостоятелен в

когато изпратите съобщението.

|  |
| --- |
| **Страница 147** |

**147**

***Внедряване на правила за достъп до метод***

механизмът за забрана на изричен приемник. И единствените обстоятелства при

които можете да пропуснете приемника са точно обстоятелствата, при които е добре да

извикайте частен метод.

Всичко това е елегантно проектирано. В мехлема обаче има една малка муха.

P RIVATE SETTER (=) МЕТОДИ

Изпълнението на частния достъп чрез правилото „без изричен получател“ се сблъсква

затруднение, когато става въпрос за методи, които завършват с равни знаци. Както ще си спомните, когато сте

извикайте метод за настройка, трябва да посочите приемника. Не можете да направите това

dog\_years = възраст \* 7

защото Ruby ще си помисли, че dog\_years е локална променлива. Трябва да направите това:

self.dog\_years = възраст \* 7

Но необходимостта от изричен приемник затруднява декларирането на метода dog\_years =

частен, поне по логиката на изискването „без изричен приемник“ за извикване на

вате методи.

Изходът от тази загадка е, че Руби не прилага правилото за определяне на метода

отс. Ако декларирате dog\_years = private, можете да го извикате с приемник - стига, че

приемникът е самостоятелен. Не може да бъде поредната препратка към себе си; трябва да е ключовата дума самостоятелно.

Ето изпълнение на куче, познаващо кучетата:

клас Куче

attr\_reader: възраст,: dog\_years

def dog\_years = (години)

@dog\_years = години

край

def age = (години)

@age = години

self.dog\_years = години \* 7

край

private: dog\_years =

край

Посочвате на колко години е кучето и кучето автоматично знае възрастта си в кучешки години:

rover = Dog.new

rover.age = 10

поставя „Rover е # {rover.dog\_years} в кучешки години.“

Методът за установяване на възраст = извършва услугата за определяне на годините на кучето, което прави до

извикване на частния метод dog\_years =. По този начин той използва явния приемник самостоятелно.

Ако го направите по друг начин, няма да работи. Без приемник бихте задали локална променлива

способен. И ако използвате един и същ обект, но под друго име, като този

def age = (години)

@age = години

куче = себе си

dog.dog\_years = години \* 7

край

**Изход: Rover е**

**70 в кучешки години**

|  |
| --- |
| **148** |

**148**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

изпълнението е спряно от фатална грешка:

NoMethodError: изисква се частен метод 'dog\_years ='

# <Куче: 0x00000101b0d1a8 @ age = 10>

Политиката на Руби е, че е добре да се използва изричен приемник за методите на частния сетер, но

трябва да завиете иглата, като се уверите, че приемникът е точно самостоятелен.

Третото ниво на достъп до метод, заедно с публичен и частен, е защитено.

***5.3.2***

***Защитени методи***

Защитеният метод е като малко по-добър, по-нежен частен метод. Правилото за про-

тектирани методи е както следва: можете да извикате защитен метод на обект x, стига да

обектът по подразбиране (self) е екземпляр от същия клас като x или на предшественик или

низходящ клас на класа на x.

Това правило звучи объркано. Но обикновено се използва по определена причина: вие

искат един екземпляр на определен клас да направи нещо с друг екземпляр на своя клас.

Следващият списък показва такъв случай.

клас С

def инициализира (n)

@n = n

край

def n

@н

край

def сравнение (c)

ако cn> n

поставя "n на другия обект е по-голям."

друго

поставя "n на другия обект е същият или по-малък."

край

край

защитен: n

край

c1 = C. нов (100)

c2 = C. нов (101)

c1.compare (c2)

Целта в този списък е да се сравни един C екземпляр с друг C екземпляр. Ком-

parison зависи от резултата от извикване на метода n. Обектът, който прави сравнението

ing (c1, в примера) трябва да поиска от другия обект (c2) да изпълни своя n метод.

Следователно n не може да бъде частно.

Тук влиза защитеното ниво. С n защитен, а не частен, c1

може да поиска от c2 да изпълни n, тъй като и двата c1 и c2 са екземпляри от един и същи клас. Но ако

се опитвате да извикате метода n на C обект, когато self е нещо различно от екземпляр

на C (или на един от предците или потомците на C), методът се проваля.

Листинг 5.9 Пример за защитен метод и неговото използване

**Резултат: Другият**

**n е по-голям.**

|  |
| --- |
| **Страница 149** |

**149**

***Писане и използване на методи от най-високо ниво***

Следователно защитеният метод е като частен метод, но с изключение за случаите

където класът на себе си (c1) и класът на обекта, за който е извикан методът

(c2) са еднакви или свързани по наследство.

Последната тема, която ще разгледаме в тази глава, са методите от най-високо ниво. Както ще видите, най-високо ниво

методи се ползват със специален случай. Но дори това състояние логично се свързва с

аспекти от дизайна на Руби, с които сте се сблъсквали в тази глава.

***5.4***

***Писане и използване на методи от най-високо ниво***

Най-естественото нещо, което трябва да направите с Ruby, е да проектирате класове и модули и

изядоха вашите класове. Но понякога просто искате да напишете бърз скрипт - няколко ком-

мандове, натъпкани във файл и изпълнени. Понякога е по-удобно да пишете

дефиниции на методите на най-горното ниво на вашия скрипт и след това ги извикайте на най-високо ниво

обекти, отколкото да обгръща всичко в дефиниции на класове. Когато правите това, вие кодирате

в контекста на обекта по подразбиране от първо ниво, main, който е екземпляр на Object

създадени автоматично по единствената причина, че *нещо* трябва да бъде себе си, дори

на най-високо ниво.

Но вие не сте в дефиницията на клас или модул, така че какво точно се случва кога

дефинирате метод?

***5.4.1***

***Определяне на метод от най-високо ниво***

Да предположим, че дефинирате метод на най-високо ниво:

def talk

поставя "Здравей"

край

Той не е вътре в блок за дефиниция на клас или модул, така че изглежда не е

екземпляр метод на клас или модул. И така, какво е това?

Метод, който дефинирате на най-високо ниво, се съхранява като метод на частен екземпляр

класът Обект. Предишният код е еквивалентен на този:

клас Обект

частни

def talk

поставя "Здравей"

край

край

Наследяване и достъп до метод

Подкласовете наследяват правилата за достъп до методите на техните суперкласове. Като се има предвид клас С

с набор от правила за достъп и клас D, който е подклас на C, екземпляри от D показват

същото поведение на достъп като екземплярите на C. Но можете да зададете нови правила в

дефиниция на клас на D, като в този случай новите правила имат предимство за случаи на D

над правилата, наследени от C.

|  |
| --- |
| **150. стр** |

**150**

C ГЛАВА 5 ***Обект по подразбиране (самостоятелно), обхват и видимост***

Определянето на методите на частния екземпляр на Object има някои интересни последици.

Първо, тези методи не само могат, но и *трябва да* бъдат извикани в стил гола дума. Защо?

Защото те са частни. Можете да ги извикате само самостоятелно и то без изрично

приемник (с обичайното освобождаване от методите на частния сетер, които трябва да бъдат извикани

със самостоятелно като приемник).

Второ, методите на частния екземпляр на Object могат да се извикват от всяко място във вашия

код, защото Object се намира в пътя за търсене на метода на всеки клас (с изключение на Basic-

Обект, но това е твърде специален случай, за който да се притеснявате). Така че методът от най-високо ниво е винаги

на разположение. Без значение какво е аз, той ще може да разпознае съобщението, което сте му изпратили

това съобщение се разрешава до метод на частен екземпляр на Object.

За илюстрация нека разширим примера за беседа. Ето го отново, с някакъв код, който

упражнява го:

def talk

поставя "Здравей"

край

поставя „Опит за„ разговор “без приемник ...“

говоря

поставя „Опит за„ разговор “с явен приемник ...“

obj = Object.new

obj.talk

Първият разговор за разговор успява B ; вторият се проваля с фатална грешка c , защото се опитва

да извика частен метод с явен приемник.

Хубавото на начина, по който работят методите от най-високо ниво е, че те предоставят полезна информация

функционалност (прости, лесни за използване на скриптове командни думи в процедурен стил), но те

направете го в пълно съответствие с правилата на Ruby: частните методи трябва да са по подразбиране

to self като приемник, а методите, дефинирани в Object, са видими за всички обекти. Не

включени са допълнителни конструкции на езиково ниво, просто елегантна и мощна комбинация

ция на тези, които вече съществуват.

Правилата относно определянето и използването на методите от най-високо ниво ни довеждат докрай

да се върнем към някои от основните методи, които използваме още от глава 1.

Вече сте в състояние да разберете как работят тези методи.

***5.4.2***

***Предварително дефинирани (вградени) методи от най-високо ниво***

От нашите най-ранни примери нататък, ние отправяме разговори в стила на речта към путове

и отпечатайте, като този:

поставя "Здравей"

put и print са вградени методи за частен екземпляр на ядрото - не като тези

пишете, на Object, но на Kernel. Резултатът е подобен, обаче (защото Object

миксове в ядрото): можете да извикате такива методи по всяко време и трябва да ги извикате с-

от приемник. Модулът на ядрото по този начин предоставя значителен набор от наложителни инструменти

методи, като путове и печат, което увеличава силата на Ruby като скриптова мрежа

гейдж. Можете да свършите много неща с Ruby скриптове, които нямат клас, модул или

б

° С

|  |
| --- |
| **Страница 151** |

**151**

***Обобщение***

дефиниции на метода, защото можете да направите толкова много (четете и пишете, стартирайте системна ком-

Mands, излезте от програмата си и т.н.) с методите на Ruby от най-високо ниво.

Ако искате да видите всички методи на частния екземпляр, които Kernel предоставя, опитайте това:

$ ruby ​​-e 'p Kernel.private\_instance\_methods.sort'

Методът private\_instance\_methods ви дава масив от всички съответни методи

ods и sort сортира масива от имена на методи за по-лесно четене. Както виждаш,

тези методи, макар и често полезни при императивно програмиране в стил скрипт, не са

ограничени по своята полезност до този стил; те включват команди като require, load,

рейз (повишаване на изключение) и други, които са сред най-често срещаните техники

във всички програми на Ruby, независимо от стила на дизайн на програмата, който показват.

***5.5***

***Обобщение***

Тази глава обхваща

Въртящата се роля на себе си (текущия или обекта по подразбиране)

Self като приемник за извиквания на методи без изричен приемник

Self като собственик на променливи на екземпляра

Последици от правилото „класовете също са обекти“

Променлив обхват и видимост за локални, глобални и променливи на класа

Правилата за търсене и препращане към константи

Нива на достъп до методите на Ruby (публични, частни, защитени)

Писане и работа с дефиниции на методи от най-високо ниво

Техниките в тази глава са от голямо значение за Руби. Концепции като различните

съотношение между променливи на екземпляра в дефиниция на клас и променливи на инстанции в

определението на метода на екземпляр са от решаващо значение. Лесно е да разгледате Ruby програма и да получите a

общо усещане за случващото се. Но да се разбере програма в дълбочина - и да се напише

добре организирани, стабилни програми - трябва да знаете как да откриете къде са различните

местните обхвати започват и свършват; как константи, променливи на екземпляра и други идентификатори

са разрешени; и как да оценим въздействието на непрекъснато променящата се роля на себе си.

Тази глава ви показа как да се ориентирате в Ruby програма. То е също

ви показа някои техники, които можете да използвате по-точно и ефективно във вашия код

по силата на това, че се ориентирате. Но има още какво да се изследва, свързано с това, което вие

може да направи в пейзажа на програма, отвъд разбирането му. Следващата глава, на

предметът на контролния поток, ще разгледа някои от тези техники.

|  |
| --- |
| **Страница 152** |

**152**

*Техники за контрол на потока*

Както вече видяхте в случая на извиквания на методи - където се контролира програмата

скача от мястото, където се извършва повикването, към тялото на метода

ция - програмите не се изпълняват по права линия. Вместо това заповедта за изпълнение се определя

добит от различни правила и програмни конструкции, наричани заедно

техники за *контрол на потока* .

Техниките за контрол на потока на Ruby включват следното:

*Условно изпълнение* - *Изпълнението* зависи от истинността на даден израз.

*Цикъл* - Един сегмент от код се изпълнява многократно.

*Итерация* - Извикването на метод се допълва с сегмент от код, който

методът може да извика един или повече пъти по време на собственото си изпълнение.

*Изключения* - Условията за грешки се обработват от специални правила за контрол на потока.

***Тази глава обхваща***

Условно изпълнение

Цикли и техники на цикли

Итератори

Изключения и обработка на грешки

|  |
| --- |
| **Страница 153** |

**153**

***Условно изпълнение на код***

Ще разгледаме всеки от тях на свой ред. Всички те са необходими както за разбирането

и практиката на Руби. Първото, условно изпълнение (ако и приятели), е фундамент-

мисловен и ясен инструмент за програмиране на почти всеки език за програмиране.

Циклирането е по-специализирана, но тясно свързана техника и Ruby ви предоставя

няколко начина да го направите. Когато стигнем до итерация, ще бъдем в истинската територия на Ruby.

Техниката не е уникална за Ruby, но е сравнително рядко приложение

това е на видно място в Ruby. Накрая ще разгледаме обширния механизъм на Руби

за обработка на условия за грешки чрез изключения. Изключенията спират потока на про-

грам, или изцяло, или докато не бъде разгледано условието за грешка. Изключения са

обекти и можете да създадете свои собствени класове изключения, наследяващи от вградените

към Ruby, за специализирано боравене със състояния на грешки във вашите програми.

***6.1***

***Условно изпълнение на код***

*Разрешете на потребителя достъп до сайт, ако паролата е правилна. Отпечатайте съобщение за грешка, освен ако не е поискано*

*елемент съществува. Признайте поражението си, ако кралят е поставен в шах.* Списъкът с приложения за контрол на потока

на програма условно - изпълнение на конкретни редове или сегменти от код само ако е

всички условия са изпълнени - е безкрайно. Без да ставаме твърде философски, може би дори

казват, че вземането на решения въз основа на непредсказуеми, но забележими условия е толкова

Mon в програмирането, както е в живота.

Ruby ви дава редица начини за контрол на програмния поток условно.

Най-важните попадат в две категории:

ако и свързани ключови думи

Декларации по делото

Ще разгледаме и двете в този раздел.

***6.1.1***

***Ключовата дума if и приятели***

Работният кон с условно изпълнение, не е изненадващо, е ключовата дума if. ако

клаузи могат да приемат няколко форми. Най-простото е следното:

ако *условие*

# код тук, изпълнява се, ако *условието* е вярно

край

Кодът вътре в условния може да бъде с всякаква дължина и може да включва вложени условни

национални блокове.

Можете също да поставите цяла клауза if на един ред, като използвате тогавашната ключова дума след

условието:

ако x> 10, тогава поставя x край

Можете също да използвате точка и запетая, за да имитирате прекъсванията на редове и да зададете ключовата дума край:

ако x> 10; поставя x; край

Условното изпълнение често включва повече от един клон; може да искате да го направите

нещо, ако състоянието успее и друго, ако не успее. Например, *ако паролата е*

|  |
| --- |
| **Страница 154** |

**154**

C ГЛАВА 6 ***Техники за контрол на потока***

*правилно, оставете потребителя да влезе; в противен случай отпечатайте съобщение за грешка.* Руби прави пълни провизии за мулти-

тройни условни клонове, използвайки else и elsif.

T HE останали и ELSIF КЛЮЧОВИ ДУМИ

Можете да предоставите клон else във вашия оператор if, както следва:

ако *условие*

# код се изпълнява, ако *условието* е вярно

друго

# код се изпълнява, ако *условието* е невярно

край

Има и ключова дума elsif (изписва се така, без второ *e* ). elsif ви позволява да

направете вашата условна логика на повече нива, отколкото можете, само ако и друго:

ако *условие1*

# код се изпълнява, ако *условието1* е вярно

*условие на* elsif2

# код се изпълнява, ако *условието1* е невярно

# и условие2 е вярно

elsif *състояние3*

# код се изпълнява, ако нито едно от *условията1*

# нито *условие2* е вярно, но *условие3* е

край

Можете да имате произволен брой клаузи elsif в даден оператор if. Кодовият сегмент

съответстващ на първия успешен, ако или elsif е изпълнен, а останалата част от състоянието

мента се игнорира:

print "Въведете цяло число:"

n = gets.to\_i

ако n> 0

поставя "Вашият номер е положителен."

elsif n <0

поставя "Вашият номер е отрицателен."

друго

поставя "Вашият номер е нула."

край

Имайте предвид, че можете да използвате окончателно else, дори ако вече имате един или повече elsifs. The

клаузата else се изпълнява, ако нито един от предишните тестове за истина не е успял. Ако няма

на условията е вярно и няма друга клауза, цялото изразено терми

нати без действие.

Понякога искате условието if да бъде отрицателно: *ако нещо не е вярно, тогава изпълнете*

*сладък даден сегмент от код* . Можете да направите това по няколко начина.

N УСЛОВИЯ ЗА ЕГАТИРАНЕ С НЕ И !

Един от начините да се отрече условие е да се използва ключовата дума not:

ако не (x == 1)

Можете също да използвате отрицанието! (удивителен знак или *взрив* ) оператор:

ако! (x == 1)

|  |
| --- |
| **Страница 155** |

**155**

***Условно изпълнение на код***

И двата примера използват скоби, за да отделят тествания израз. Вие

не се нуждаят от тях в първия пример; можете да направите това:

ако не x == 1

Но вие *се* нуждаете скобите във втория пример, тъй като отрича! опера-

ator има по-висок приоритет от оператора ==. С други думи, ако направите това

ако! x == 1

вие всъщност сравнявате отрицанието на x с цялото число 1:

ако (! x) == 1

Най-добрата практика е да използвате скоби през повечето или дори през цялото време, когато пишете кон

структури като тази. Дори и да не са строго необходими, те могат да ви улеснят

и други, за да разберат вашия код и да го модифицират по-късно, ако е необходимо.

Третият начин за изразяване на отрицателно състояние е с освен.

T HE ОСВЕН KEYWORD

Ключовата дума never предоставя по-естествен начин за изразяване на същото

семантика като че ли не или ако!:

освен ако x == 1

Но вземете „естествено звучене“ със зърно сол. Програмите Ruby са написани на Ruby,

не английски и трябва да се стремите към добрия стил на Руби, без да се притеснявате излишно

за това как кодът ви се чете като английска проза. Не че английският не може от време на време

направлявам те; например последователността освен / else, която връща назад от

отрицателно към положително, което обикновено не се свързва с употребата на думата, *освен ако не* може

бъдете малко трудни за следване:

освен ако x> 100

поставя "Малък номер!"

друго

поставя "Голямо число!"

край

Като цяло, ако / else чете по-добре, освен освен ако / else - и като обърнете логиката на

условие, винаги можете да замените последното с първото:

ако x <= 100

поставя "Малък номер!"

друго

поставя "Голямо число!"

край

Ако попаднете на случай, при който отричането на логиката изглежда по-неудобно от сдвояването

освен ако не е с друго, след това запазете освен. В противен случай, ако имате клауза else, ако е gen-

или по-добър избор, освен ако.

|  |
| --- |
| **Страница 156** |

**156**

C ГЛАВА 6 ***Техники за контрол на потока***

Можете също така да поставите условни тестове в позиция на *модификатор* , директно след изявление.

С НЕОБХОДИМИ МОДИФИКАТОРИ

Не е необичайно да видите условен модификатор в края на израз в случай

като този:

поставя "Голямо число!" ако x> 100

Това е същото като

ако x> 100

поставя "Голямо число!"

край

Можете също да направите това, освен ако:

поставя "Голямо число!" освен ако x <= 100

Живот без висящата **друга** неяснота

В някои езици не можете да разберете коя друга клауза върви с коя, ако клауза с-

извън специално правило. В C например изразът if може да изглежда така:

ако (x)

if (y) {изпълни този код}

else { *изпълнете този код* };

Но изчакайте: Кодът поведе ли се така, както показва отстъпът (другото принадлежи

към втория ако)? Или работи така?

ако (x)

if (y) {изпълни този код}

else {изпълнете този код};

Всичко, което се промени, е отстъпът на третия ред (което няма значение за C

компилатор; отстъпът просто прави неяснотата визуално очевидна). Което ако има

другото принадлежи? И как да разберете?

Казвате, като знаете правилото в C: висящо друго върви с последния несравним if

(първото от двете поведения в този пример). Но в Ruby имате нужда от помощ

вие навън:

ако x> 50

ако x> 100

поставя "Голямо число"

друго

поставя "Среден номер"

край

край

Сингълът else в това изявление трябва да принадлежи на втория if, защото това if

все още не е достигнал края си. Първият if и последният край винаги принадлежат заедно, вторият

ако и последният край винаги принадлежат заедно и т.н. Двойките if / end

обградете това, което им принадлежи, включително и друго. Разбира се, това означава, че трябва

поставете крайните си ключови думи правилно.

**x е вярно,**

**но не е**

**x не е вярно**

|  |
| --- |
| **157** |

**157**

***Условно изпълнение на код***

Условните модификатори имат разговорен тон. Няма край за притеснение. Вие

не мога да направя толкова много с тях (няма друго или разклоняване elsif, например), но когато вие

се нуждаят от прост условен, те често са подходящи. Опитайте се да избягвате наистина дълги изявления

които завършват с условни модификатори; те могат да бъдат трудни за четене и трудно

дръжте в главата си, докато чакате модификатора в края:

поставя "готово"&& return (x> y && a <b), освен ако c == 0

Подобно на други изрази в Ruby, всеки оператор if оценява обект. Нека да разгледаме

как се разиграва това.

T HE СТОЙНОСТ НА If конструкции

Ако операторът if успее, целият оператор оценява това, което е представено

от кода в успешния клон. Въведете този код в irb и ще видите този принцип

пле в действие:

x = 1

ако x <0

"отрицателен"

elsif x> 0

"положителен"

друго

"нула"

край

Както irb ще ви каже, стойността на целия този оператор if е низът "положителен".

Оператор if, който никъде не успява, връща нула. Ето пълен изпит за IRB-

пле на такъв случай:

>> x = 1

=> 1

>> ако x == 2

>>"това е 2!"

>> elsif x == 3

>>"това е 3!"

>> край

=> нула

Условните изявления взаимодействат с други аспекти на синтаксиса на Ruby по няколко начина

че трябва да сте наясно - по-специално със синтаксиса на присвояване. Струва си да се погледне

в някои подробности за това как се държат условните в задачите, защото това включва някои

интересни моменти за това как Ruby анализира кода.

***6.1.2***

***Синтаксис на заданието в състояния и тестове***

Синтаксисът на заданието и условните изрази се пресичат в две точки: в телата

на условни изрази, където заданията могат да се случат или да не се случат изобщо, и

в самите условни тестове:

**Потенциално объркващо**

**прихващане на освен ако**

**до вече дълга линия**

**Цял, ако изразът оценява**

**да нула, защото не успява**

|  |
| --- |
| **Страница 158** |

**158**

C ГЛАВА 6 ***Техники за контрол на потока***

ако x = 1

y = 2

край

Какво се случва (или не), когато използвате тези идиоми? Ще разгледаме и двете, започвайки

с присвояване на променлива в тялото на условната - по-специално локална променлива

възлагане, което показва някакво може би неочаквано поведение в този контекст.

L OCAL VARIABLE ПРЕХВЪРЛЯНЕ НА ТЯЛОТО условно

Ruby не очертава толкова ясна граница, колкото компилираните езици, между „времето за компилиране“

и „runtime“, но интерпретаторът прави синтактичен анализ на вашия код, преди да го стартира и потвърждава

по време на този процес се вземат отделни решения. Важно е разпознаването и

разпределение на локални променливи.

Когато анализаторът на Ruby вижда *идентификатора* на последователността *, знака за равенство* и *стойността* , както в това

израз,

x = 1

тя разпределя място за локална променлива, наречена x. Създаването на променливата - не

присвояване на стойност към него, но вътрешното създаване на променлива винаги се извършва

в резултат на този вид израз, дори ако кодът не се изпълнява!

Помислете за този пример:

ако е невярно

x = 1

край

px

py

Присвояването на x не се изпълнява, защото е увито в неуспешен условен

тест. Но анализаторът на Ruby вижда последователността x = 1, от която извежда, че

програмата включва локална променлива x. Анализаторът не се интересува дали x е изобщо

присвоява стойност. Неговата работа е просто да измие кода за локални променливи, за което пространство

трябва да бъде разпределено.

Резултатът е, че x обитава странен вид променлива крайност. Донесено е

в битие и инициализирано до нула. В това отношение тя се различава от променлива, която няма

съществуване изобщо; както можете да видите в примера, изследването на x ви дава стойността nil,

като има предвид, че опитът да се провери несъществуващата променлива y води до фатална грешка. Но

въпреки че x съществува, той не е играл никаква роля в програмата. Той съществува само като артефакт

на процеса на синтактичен анализ.

Нищо от това не се случва с клас, екземпляр или глобални променливи. И трите

променливите типове се разпознават по външния им вид (@@ x, @x, $ x). Но локални променливи

изглеждат точно като извиквания на методи. Руби трябва да приложи някаква логика при анализиране, за да разбере

какво какво, доколкото може.

Също така трябва да държите внимателно за себе си, когато използвате синтаксиса на присвояване в теста

част от условно.

**Възлагане в**

**условен тест**

**Възлагане в**

**условно тяло**

**Изход: нула**

**Фатална грешка:**

**y е неизвестен**

|  |
| --- |
| **Страница 159** |

**159**

***Условно изпълнение на код***

A TEST SSIGNMENT ПО условно

В този пример имайте предвид, че условният тест е задание (x = 1), а не

тест за равенство (който би бил x == 1):

ако x = 1

поставя "Здравей!"

край

Присвояването работи както обикновено заданията: x се задава на 1. Тестът, следователно,

намалява до if 1, което е вярно. Следователно тялото на условния се изпълнява и

низът "Здравей!" се отпечатва.

Но получавате и предупреждение:

предупреждение: намерено = в условно, трябва да бъде ==

Мисленето на Руби в случай като този е следното. Тестовият израз, ако x = 1 ще

винаги успее и условното тяло винаги ще бъде изпълнено. Това означава

няма възможна причина програмист някога да пише, ако x = 1. Следователно,

Руби заключава, че почти сигурно сте искали да напишете нещо друго и проблеми

предупреждението, за да ви предупреди за вероятната грешка. По-конкретно, предупредителната

гестира оператора ==, който произвежда реален тест (т.е. тест, който не е необходим-

ily винаги вярно).

Особено хубавото на този предупредителен механизъм е, че Руби е умна

достатъчно, за да не ви предупреждава в случаите, когато не е сигурно, че състоянието ще бъде вярно.

Ако дясната страна на заданието сама по себе си е променлива или извикване на метод, тогава вие

не получавайте предупреждението:

ако x = y

За разлика от x = 1, изразът на присвояване x = y може или не може да бъде успешен като условен

тест. (Ще бъде невярно, ако y е невярно.) Следователно не е неправдоподобно да сте тествали това

израз, така че Руби не ви предупреждава.

Защо бихте искали да използвате задание в условен тест? Със сигурност никога

трябва; винаги можете да направите това:

x = y

ако x

# и т.н.

Но понякога е удобно да правите задаването и тестването едновременно, особено

когато използвате метод, който връща нула при неуспех и някаква друга стойност при успех

цес. Често срещан пример е съвпадението на модела с метода на съвпадение. Този метод,

което ще видите много повече в глава 11, тества низ срещу регулярен израз,

връщане на нула, ако няма съвпадение и екземпляр на MatchData, ако има такъв. The

Обектът MatchData може да бъде заявен за информация относно спецификата на съвпадението.

Обърнете внимание на използването на буквален регулярен израз, / la /, в хода на тестване за съвпадение

срещу името на низа:

**Без предупреждение**

|  |
| --- |
| **Страница 160** |

**160**

C ГЛАВА 6 ***Техники за контрол на потока***

name = "Дейвид А. Блек"

ако m = /la/.match(name)

поставя "Намерих съвпадение!"

print "Ето ненадминатото начало на низа:"

поставя m.pre\_match

print "Ето ненадминатия край на низа:"

поставя m.post\_match

друго

поставя "Няма съвпадение"

край

Резултатът от този фрагмент е

Намерих съвпадение!

Ето ненадминатото начало на низа: Дейвид А. Б

Ето ненадминатия край на низа: ck

Методът на съвпадение търси модела la в низа "David A. Black". Вариацията

способният m е присвоен в условния B и ще бъде нулев, ако няма съвпадение. Колода е

подредени в примера, разбира се: има съвпадение, така че m е обект MatchData и може

да бъде поискано. В примера го питаме за частите на низа, които са възникнали

преди и след съответстващата част на низа и тя ни дава съответните поднизове.

Както винаги, можете да пренапишете заданието и условния тест по следния начин:

m = /la/.match(name)

ако m

# и т.н.

Не е нужно да ги комбинирате в един израз. Но поне в този случай има

някаква семантична тежест за това: изразът може или не може да премине условния

тест, така че е разумно да го тествате.

Въпреки че ако и приятелите са условните ключови думи на Руби за хляб и масло, те са

не единствените. След това ще разгледаме изявленията по случая.

***6.1.3***

***казуси по делото***

Изразът за случай започва с израз - обикновено единичен обект или променлива, но всеки

израз може да се използва - и го превежда през списък с възможни съвпадения. Всеки възможен

match се съдържа в оператор when, състоящ се от едно или повече възможни съвпадения

обекти и сегмент от код. Когато един от термините в дадена клауза съвпада,

че когато се счита, че е „спечелил“, и неговият кодов сегмент се изпълнява. Само един

мач, най-много, може да спечели.

изявленията за случаи са по-лесни за разбиране чрез пример, отколкото чрез описание. Следното

списъкът показва делото, което тества линия от въвеждане на клавиатура и базирани клонове

върху неговата стойност.

print "Изход от програмата? (да или не):"

отговор = gets.chomp

отговор на случая

Листинг 6.1 Интерпретиране на потребителски вход с изявление за **случай**

б

**Изрязва крайната линия**

**знак на входния низ**

б

|  |
| --- |
| **Страница 161** |

**161**

***Условно изпълнение на код***

когато "да"

поставя "Сбогом!"

изход

когато "не"

поставя „Добре, ще продължим“

друго

поставя "Това е неизвестен отговор - ако приемем, че сте имали предвид" не "

край

поставя "Продължаване с програма ...."

Делото започва с ключова дума B , продължава през всички кога

блокира c и клауза d (по избор) else и завършва с ключовата дума e . В

повечето, един мач ще успее и кодът му ще бъде изпълнен. Ако е този, който принадлежи

"да", тогава програмата излиза. Всеки друг вход е или "не", или някаква друга стойност,

което тази програма интерпретира като еквивалент на "не", което кара програмата да

продължи да работи.

Можете да поставите повече от едно възможно съвпадение в едно, когато, както показва този фрагмент:

отговор на случая

когато "y", "да"

поставя "Сбогом!"

изход

# и т.н.

Запетаята между множество условия в клауза, когато е вид оператор „или“;

този код ще каже „Сбогом!“ и излезте, ако отговорът е "y" или "да".

Нека да разгледаме по-нататък как кога клаузите работят под капака. Няма да се изненадате

за да научите, че е изпратено съобщение.

H OW ПРИ РАБОТИ

Основната идея на случая / когато структурата е да вземете обект и каскада

чрез поредица от тестове за мач, предприемане на действия въз основа на теста, който е успешен. Но

какво означава *съвпадение* в този контекст? Какво означава, в нашия пример, да кажем това

отговор отговаря на думата *да* , или думата *не* , или нито едно?

Руби има конкретна дефиниция на *съвпадение,* когато става въпрос за кога.

Всеки обект на Ruby има метод за *равенство* на *случай* , наречен === (три знака за равенство, някои-

пъти наречен *оператор threequal* ). Резултатът от извикването на метода === определя

дали клаузата за кога е съвпаднала.

Можете да видите това ясно, ако погледнете първо изявление по дело и след това превод

на това твърдение в три равни условия. Погледнете отново делото в списъка-

6.1. Ето същото нещо, пренаписано, за да се използва изрично операторът threequal:

ако "да" === отговор

поставя "Сбогом!"

изход

elsif "не" === отговор

поставя „Добре, ще продължим“

друго

поставя „Това е неизвестен отговор - ако приемем, че сте имали предвид„ не ““

край

° С

д

д

|  |
| --- |
| **162 серия** |

**162**

C ГЛАВА 6 ***Техники за контрол на потока***

=== в позиция на инфиксния оператор (т.е. между ляв термин и дясна ръка

терминът) е наистина синтактична захар за извикване на метод:

ако е "да". === (отговор)

Оператор when обгръща извикването на този метод в още повече захар: не е нужно да използвате ===

изрично в позиция на оператор или метод. Това е направено за вас.

Това е логиката на синтаксиса. Но защо го прави

"да" === отговор

връща true, когато отговорът съдържа "да"?

Извикването на метода връща true, тъй като за метода threequal е дефиниран

струни. Когато поискате низ да се уравни с друг низ (string1 ===

string2), вие го молите да сравнява собственото си съдържание символ по знак срещу

другият низ и докладвайте true за перфектно съвпадение или false по друг начин.

Най-важният момент в това обяснение е фразата „за струни *.* ”Всеки клас

(и на теория всеки отделен обект, въпреки че обикновено се обработва на ниво клас)

може да дефинира свой собствен === метод и по този начин собствената си логика за равенство на случаи. За низове и,

наистина, за всеки обект, който не го отменя, === работи по същия начин като == (основният

string-equals-some-other-string метод за изпитване). Но други класове могат да дефинират threequal

тествайте по какъвто начин искат.

случай / когато логиката е наистина обект === други\_обективна логика; и

object === other\_object е наистина обект. === (other\_object) прикрит. По дефиниция-

Използвайки метода threequal, както искате за вашите собствени класове, можете да упражнявате

пълен контрол върху начина, по който вашите обекти се държат в рамките на случай.

P ROGRAMMING ОБЕКТИ " CASE ОБЯВЯВАНЕ ПОВЕДЕНИЕ

Да приемем, че решаваме, че обектът на Ticket трябва да съвпада с клауза когато в състояние на случай-

въз основа на мястото му на провеждане. Можем да постигнем това, като напишем съответния threequal

метод. Следващият списък показва такъв метод, в комплект с достатъчно функции за билети-

да се направи пълен работен пример.

клас Билет

attr\_accessor: място,: дата

def инициализиране (място, дата)

self.venue = място

self.date = дата

край

def === (друга\_билет)

self.venue == other\_ticket.venue

край

край

ticket1 = Ticket.new ("Town Hall", "07/08/13")

ticket2 = Ticket.new ("Конферентен център", "07/08/13")

ticket3 = Ticket.new ("Town Hall", "08/09/13")

поставя „билет1 е за събитие на адрес: # {ticket1.venue}.“

случай билет1

Обява 6.2 Изпълнителна **случай** поведение изявление за **продажба на билети** клас

б

|  |
| --- |
| **163 серия** |

**163**

***Условно изпълнение на код***

когато билет2

поставя „Същото местоположение като билет2!“

когато билет3

поставя „Същото местоположение като билет3!“

друго

поставя "Няма съвпадение"

край

Резултатът от този списък е както следва:

билет1 е за събитие в: Town Hall.

Същото местоположение като билет3!

Съвпадението се намира чрез неявното използване на метода === на

Билет клас B . Вътре в изявлението case, първото, когато израз c задейства hid-

den извикване към ===, еквивалентно на това:

ако билет2 === билет1

Тъй като методът === връща true или false въз основа на сравнение на местата, и

Мястото на Tick2 не е същото като Tick1, сравнението между двата билета

връща false. Следователно тялото на съответната клауза когато не се изпълнява.

След това се провежда следващият тест: още едно сравнение между три билета между билет1

и билет3 d . Този тест връща true; изразът когато успее и кодът в

тялото му е изпълнено.

Този вид взаимодействие между дефинициите на метода (===) и кода, който не го прави

изглежда, че методите за извикване (случай / кога) са типични за Ruby. Случаят / кога структура

осигурява елегантен начин за извършване на каскадни условни тестове; и факта, че е

куп === повиквания означава, че можете да го накарате да прави това, от което се нуждаете, като дефинирате ===

метод във вашите класове.

Делото също се предлага в леко съкратена форма, която ви позволява да тествате

директно за стойност на истината: случай без израз на случай.

T HE простия случай TEST ИСТИНАТА

Ако започнете изявление за случай с ключова дума case от само себе си - т.е. без тест

израз - последван от някои клаузи, когато клаузата, чието условие е

true ще бъде победителят. Ако приемем, че потребител на обект е с първо и фамилно име

методи, можете въображаемо да напишете изявление за случай като това:

случай

когато user.first\_name == "Дейвид", user.last\_name == "Черен"

поставя „Може да си Дейвид Блек“.

когато Time.now.wday == 5

поставя "Ти не си Дейвид Блек, но поне е петък!"

друго

поставя „Ти не си Дейвид Блек и не е петък“.

край

Простата ключова дума по този начин е алтернативен начин за писане на оператор if.

Всъщност всяко изявление за казус може да бъде написано като изявление if. казуси с

изричните аргументи по делото често са значително по-кратки от тези, ако техните колеги,

° С

д

**Пореден номер**

**за ден от седмицата**

**(Неделя е 0)**

|  |
| --- |
| **164 серия** |

**164**

C ГЛАВА 6 ***Техники за контрол на потока***

които трябва да прибегнат до обаждане === или други методи за сравнение. Тези без

изричните тестови аргументи обикновено не са по-кратки от еквивалентните if; за

Например, предишният пример ще бъде написан по този начин, като се използва

ако user.first\_name == "Дейвид" или user.last\_name == "Черен"

поставя „Може да си Дейвид Блек“.

elsif Time.now.wday == 5

поставя "Ти не си Дейвид Блек, но поне е петък!"

друго

поставя „Ти не си Дейвид Блек и не е петък“.

край

Предимството на декларацията за непроверен случай е, че не ви ограничава до това, което сте

може да тества с === на даден обект. Когато имате избор между случай без тест или

ако, вашият избор ще се основава на вашето усещане за това, което изглежда и чете най-добре.

Т се върне СТОЙНОСТ НА CASE ОТЧЕТИ

Важен последен момент, който трябва да имате предвид при изявленията по случая, е всеки случай

израз изчислява до един обект. Ако има успешна клауза кога или друго,

връщаната стойност на целия оператор на случай е стойността, върната от кода в това

клауза. В противен случай, ако изявлението за случай не успее да намери съвпадение, цялото изявление

оценява на нула, подобно на ако операторите, които се провалят.

По този начин можете например да пренапишете безусловния пример по следния начин:

поставя дело

когато user.first\_name == "Дейвид", user.last\_name == "Черен"

- Може да си Дейвид Блек.

когато Time.now.wday == 5

"Ти не си Дейвид Блек, но поне петък е!"

друго

- Ти не си Дейвид Блек и не е петък.

край

В тази версия извикванията към путове са извлечени от клаузите when; цялото

case изявлението е посветено на намирането на аргумент за единичното извикване отляво.

Този аргумент ще бъде който и от трите низа да върне изявлението case.

Условни като ако и случай / кога ви позволяват да контролирате потока на програмата, като го направите

нещо вместо друго. Но понякога трябва да изпълните отново една-единствена задача

и отново. Този вид повторение може да се осъществи с цикли, които ще разгледаме

на следващия.

***6.2***

***Повтарящи се действия с цикли***

Удобствата на Ruby за многократно циклиране чрез код също ви позволяват да включите

условна логика: можете да циклирате, *докато* дадено условие е вярно (като променлива същество

равна на дадена стойност) и можете да завъртате, *докато* дадено условие е вярно. Можете също

излизане от цикъл *безусловно* , прекратяване на цикъла в определена точка и

възобновяване на изпълнението на програмата след цикъла.

Ще разгледаме няколко начина за цикъл - започвайки, по подходящ начин, с метод, наречен цикъл.

|  |
| --- |
| **165** |

**165**

***Повтарящи се действия с цикли***

***6.2.1***

***Безусловен цикъл с метода на цикъла***

Методът на цикъла не приема никакви нормални аргументи: просто го извиквате. Прави, как-

някога, вземете кодов блок - тоест, ограничен набор от програмни инструкции, написани като

част от викането на метода (повикването към цикъл) и е налице за да бъде изпълнена *от* най

метод. (Ще разгледаме кодовите блокове много по-подробно по-късно в тази глава. Можете

се справете само с нивото на знанието за запазване тук.) Анатомията на призива към

цикъл, тогава изглежда така:

цикъл кодов блок

Блоковете на кода могат да бъдат написани по един от двата начина: или във фигурни скоби ({}), или с

ключовите думи правят и завършват. Следните два фрагмента са еквивалентни:

цикъл {поставя "Looping forever!" }

цикъл правя

поставя "Looping forever!"

край

Разхлабената конвенция гласи, че едноредови кодови блокове използват къдравите скоби и многоредови

блокове използват do / end. Но Руби не прилага тази конвенция. (Скобите и

do / end двойка всъщност се различават леко един от друг по отношение на предимството. Вие

не трябва да се притеснявате за това сега.)

Като цяло не искате цикъл да се завърта завинаги; искате да спре в даден момент. Вие

обикновено може да спре, като натиснете Ctrl-C, но има и други, по-програмни начини.

C КОНТРОЛИРАНЕ НА ПЪТАТА

Един от начините да се спре цикълът е с ключовата дума break, както е в тази многословна дума

подход за настройка n на 10:

n = 1

цикъл правя

n = n + 1

прекъсване, ако n> 9

край

Друга техника преминава към следващата итерация на цикъла, без да завърши курса

итерация под наем. За да направите това, използвате ключовата дума next:

n = 1

цикъл правя

n = n + 1

следваща, освен ако n == 10

почивка

край

Тук контролът пада до оператора break, само ако n == 10 е вярно. Ако n == 10 е

*не е* вярно (освен ако n == 10), следващото се изпълнява и контролът скача обратно към началото-

ning на цикъла, преди да достигне прекъсване.

Можете също така да *приключвате* условно: *докато* дадено условие е вярно или *до* условие

става истина.

|  |
| --- |
| **166 серия** |

**166**

C ГЛАВА 6 ***Техники за контрол на потока***

***6.2.2***

***Условен цикъл с ключови думи докато и докато***

Условното циклиране се постига чрез ключовите думи докато и до. Тези ключови думи могат да бъдат

използва се по всеки от няколко начина, в зависимост от това как точно искате да се повтори цикъла.

T HE ДОКАТО KEYWORD

Ключовата дума while ви позволява да изпълните цикъл, докато дадено условие е вярно. Блок

започвайки с докато трябва да завърши с край. Кодът между while и end е тялото

на цикъла while. Ето пример:

n = 1

докато n <11

поставя н

n = n + 1

край

поставя "Готово!"

Този код отпечатва следното:

1

2

3

4

5

6

7

8

9

10

Свършен!

Докато условието n <11 е вярно, цикълът се изпълнява. С всяка итерация на

цикъл, n се увеличава с 1. Единадесетият път, когато условието се тества, е невярно (n е

не по-малко от 11) и изпълнението на цикъла се прекратява.

Можете също да поставите в края на цикъл. В този случай трябва да използвате ключа-

двойка думи започва / завършва, за да отбележи къде е цикълът (в противен случай Руби няма да знае как

много от редовете, предхождащи времето, което искате да включите в цикъла):

n = 1

започнете

поставя н

n = n + 1

край, докато n <11

поставя "Готово!"

Резултатът от този пример е същият като изходът от предишния пример.

Има разлика между поставянето в началото и поставянето в

край. Ако поставите while в началото и ако условието while е false, кодът

не се изпълнява:

n = 10

докато n <10

поставя н

край

|  |
| --- |
| **Страница 167** |

**167**

***Повтарящи се действия с цикли***

Тъй като n е по-голямо от 10, когато тестът n <10 се извършва за първи път,

тялото на изявлението не се изпълнява. Но ако поставите теста за време в края,

n = 10

започнете

поставя н

край, докато n <10

отпечатва се числото 10. Очевидно е, че n не е по-малко от 10 във всяка точка. Но тъй като

докато тестът е позициониран в края на изявлението, тялото се изпълнява веднъж преди това

тестът се извършва.

Като ако и освен ако ключовите думи с условен цикъл идват като двойка: while и till.

T HE ДО KEYWORD

Ключовата дума till се използва по същия начин, както докато, но с обратна логика. Ето

друг трудоемък начин за разпечатване на целите числа от 1 до 10, този път илюстриращ

използването до:

n = 1

докато n> 10

поставя н

n = n + 1

край

Тялото на цикъла (отпечатването и увеличаването на n, в този пример) е изпълнено

изрязва се многократно, докато условието е вярно.

Можете също да използвате докато не е в позиция след блокиране, заедно с начало / край

сдвояване. Както и докато, блокът ще се изпълни веднъж, преди да бъде тествано условието.

Подобно на техните братовчеди, ако и освен ако, докато и докато може да се използва в модификатор

ция в едноредови изявления.

Т ВО при и до МОДИФИКАТОРИ

Ето малко по-кратък начин за броене до 10, като се използва докато не е в позиция на модификатор:

n = 1

n = n + 1, докато n == 10

поставя "Достигнахме 10!"

Вместо оператора till, можете също да използвате докато n <10.

Имайте предвид, че едноредовите модификаторски версии на while и until не се държат на

по същия начин като пост-позиционираните докато и докато не използвате с начален / краен блок. В

с други думи, в случай като този

a = 1

a + = 1 до истина

а ще остане 1; изразът a + = 1 няма да бъде изпълнен, защото true вече е true.

Но в този случай

a = 1

започнете

a + = 1

край до истина

|  |
| --- |
| **168 серия** |

**168**

C ГЛАВА 6 ***Техники за контрол на потока***

тялото на блока начало / край се изпълнява веднъж.

В допълнение към цикъла безусловно (цикъл) и условно (докато, докато),

можете да прегледате списък със стойности, като изпълните цикъла веднъж за всяка стойност. Руби

предлага няколко начина за това, един от които е ключовата дума за.

***6.2.3***

***Цикли на базата на списък със стойности***

Да предположим, че искате да отпечатате диаграма на Фаренхайт еквиваленти на стойности на Целзий. Можеш

направете това, като поставите стойностите на Целзий в масив и след това прегледате масива

използване на двойката ключови думи for / in. Цикълът се изпълнява веднъж за всяка стойност в масива; всеки

от време на време, тази стойност се присвоява на променлива, която сте посочили:

Целзий = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

поставя "Целзий \ tFahrenheit"

за c в Целзий

поставя "# {c} \ t # {Temperature.c2f (c)}"

край

Тялото на цикъла (операторът put) се изпълнява 11 пъти. За първи път,

стойността на c е 0. Вторият път, c е 10; третият път е 20; и така нататък.

защото е мощен инструмент. Колкото и да е странно, при по-внимателен оглед се оказва

това е просто алтернативен начин да се направи нещо още по-мощно.

***6.3***

***Итератори и кодови блокове***

Техниките за контрол на потока, които разгледахме досега, включват контрол колко пъти,

или при какви условия се изпълнява сегмент от код. В този раздел ще разгледаме

в различен вид съоръжение за управление на потока. Техниките, които ще обсъдим тук, не само

изпълнете операция за изпълнение или пропускане на сегмент от код; те отскачат контрола върху

програма от един обхват в друг и обратно, чрез *итерация* .

***6.3.1***

***Съставките на итерацията***

Фокусирайки се върху движението между местните обхвати, може да звучи така, сякаш сме се върнали

говорейки за извиквания на методи. В края на краищата, когато извикате метод върху обект, контролът е

предадени в тялото на метода (различен обхват); и когато методът има фин-

изпълнено, контролът се връща към точката точно след точката, където методът

обаждането се състоя.

Наистина сме отново в територията за извикване на метода, но изследваме нови аспекти от нея,

не само преразглеждане на старото. Говорим за нова конструкция, наречена *кодов блок* и a

ключова дума по името на добив.

В раздел 6.2.1 видяхте примерен код, който изглеждаше така:

цикъл {поставя "Looping forever!" }

Думата цикъл и съобщението в низа ви подсказват какво получавате, ако изпълните

то: това съобщение, отпечатано завинаги. Но какво *точно* се случва? Защо това поставя

изобщо се изпълнява - и защо се изпълнява в цикъл?

**Заглавка за диаграма**

**(\ t отпечатва раздел)**

|  |
| --- |
| **169** |

**169**

***Итератори и кодови блокове***

Отговорът е, че цикълът е *итератор* . Итераторът е метод на Ruby, който има екстра

съставка в извикващия синтаксис: очаква да му предоставите кодов блок. Къдравата

скоби в примера на цикъла ограничават блока; кодът в блока се състои от

поставя изявление.

Методът на цикъла има достъп до кода вътре в блока: методът може да *извика*

(изпълни) блока. За да направите това от собствен итератор, използвате ключовата дума

добив. Заедно кодовият блок (предоставен от повикващия код) и добив (извикан

от метода) са основните съставки на итерацията.

цикълът е написан на C (и използва функция C, за да постигне същия ефект като

добив). Но цялата идея на цикъла предполага интересно упражнение: reimplement-

контур в чист Ruby. Това упражнение ще ви даде първи поглед върху добива в действие.

***6.3.2***

***Итерация, домашен стил***

Работата на цикъла е да дава контрол на кодовия блок, отново и отново, завинаги. Ето

как можете да напишете вашата собствена версия на цикъл:

def my\_loop

докато е вярно

добив

край

край

Или още по-кратко:

def my\_loop

добив, докато е вярно

край

Тогава бихте го извикали точно както се обаждате

my\_loop {поставя "My-looping forever!" }

и съобщението ще се отпечатва отново и отново.

Предоставяйки кодов блок, вие давате на my\_loop нещо - парче код

което може да даде контрол. Когато методът отстъпва на блока, кодът в блока

работи и след това контролът се връща към метода. Добивът не е същото като връщането

от метод. Добивът се извършва, докато методът все още работи *.* След кода

блокът се изпълнява, контролът се връща към метода в оператора, непосредствено след него

призивът за отстъпване.

Кодовият блок е част от извикването на метода - тоест част от неговия синтаксис. Това е

важен момент: кодният блок не е аргумент. Аргументите на методите са

аргументи. Кодовият блок е кодовият блок. Те са две отделни конструкции. Вие

можете да видите логиката на разграничението, ако погледнете пълната картина на метода

обажданията се събират.

|  |
| --- |
| **170 серия** |

**170**

C ГЛАВА 6 ***Техники за контрол на потока***

***6.3.3***

***Анатомията на извикване на метод***

Всяко извикване на метод в Ruby има следния синтаксис:

Обект или променлива на приемника (по подразбиране самостоятелно, ако отсъства)

Точка (задължително, ако има изричен приемник; забранено е друго)

Име на метод (задължително)

Списък с аргументи (по избор; по подразбиране е ())

Кодов блок (по избор; няма по подразбиране)

По-специално обърнете внимание, че списъкът с аргументи и кодовият блок са отделни. Тяхното съществуване-

тенцията варира независимо. Всичко това са синтактично легитимни извиквания на Ruby метод:

цикъл {поставя "Здравей"}

цикъл () {поставя "Здравей"}

string.scan (/ [^,] + /)

string.scan (/ [^,] + /) {| дума | поставя дума}

(Последният пример показва параметър на блока, дума. Ще се върнем към параметрите на блока

в момента.) Разликата между извикване на метод с блок и извикване на метод с-

от блок се свежда до това дали методът може да даде или не. Ако има блок,

тогава може; ако не, не може, защото няма на какво да се отстъпи.

Освен това са написани някои методи, така че те поне ще направят *нещо* , независимо дали

предавате им кодов блок или не. String # split, например, разделя приемника си (a

низ, разбира се) на разделителя, който въвеждате и връща масив от разделения елемент

моменти. Ако му подадете блок, split също дава разделените елементи на блока, един в a

време. След това вашият блок може да прави каквото поиска с всеки подниз: отпечатвайте го, скрийте

това в колона на база данни и т.н.

Ако се научите да мислите за кодовия блок като синтактичен елемент на извикването на метода,

вместо като един от аргументите, ще можете да поддържате нещата прави, както виждате

още вариации на основната итерационна тема.

По-рано видяхте накратко, че блоковете на кода могат да бъдат разграничени или от къдрави скоби, или

от двойката ключови думи do / end. Нека сега разгледаме по-отблизо как тези два разделителя

опциите се различават помежду си.

***6.3.4***

***Фигурни скоби срещу do / end в синтаксиса на кодовия блок***

Разликата между двата начина за разграничаване на кодов блок е разлика в предварително

цеденс. Погледнете този пример и ще започнете да виждате как се играе това:

>> масив = [1,2,3]

=> [1, 2, 3]

>> array.map {| n | n \* 10}

=> [10, 20, 30]

>> array.map do | n | n \* 10 край

=> [10, 20, 30]

>> поставя array.map {| n | n \* 10}

10

20.

30

б

° С

д

|  |
| --- |
| **Страница 171** |

**171**

***Итератори и кодови блокове***

=> нула

>> поставя array.map do | n | n \* 10 край

# <Изброител: 0x00000101132048>

=> нула

Методът map работи през масив по един елемент, като извиква кодовия блок

веднъж за всеки елемент и създаване на нов масив, състоящ се от резултатите от всички тях

обаждания към блока. Картографиране на нашия масив [1,2,3] чрез блок, който умножава всеки

т. по 10 резултата в новия масив [10,20,30]. Освен това, за проста операция на карта-

не е важно дали използваме къдрави скоби B или правим / край c . Резултатите са

същото.

Но вижте какво се случва, когато използваме резултата от операцията на картата като

аргумент за поставя. Версията с къдрава скоба отпечатва масива [10,20,30] (един елемент

на ред, в съответствие с начина, по който се обработват масиви) d . Но версията do / end се връща

изброител - точно това прави картата, когато е извикана *без* кодов блок e .

(Ще научите повече за изброителите в глава 10. Съответната точка тук е тази

двата блокови синтаксиса дават различни резултати.)

Причината е, че предимството е различно. Първото изявление за путове е

Предполагам така:

поставя (array.map {| n | n \* 10})

Вторият се интерпретира по следния начин:

put (array.map) do | n | n \* 10 край

Във втория случай кодовият блок се интерпретира като част от повикването към путове, а не

обаждането към картата. И ако извикате путове с блок, той игнорира блока. Така че направете / края

версията е наистина еквивалентна на

поставя array.map

И затова получаваме преброител.

Извикването за картографиране с помощта на кодов блок в стил стил / край илюстрира факта, че ако предоставите

код, но методът, който извиквате, не го вижда (или не го търси), без грешка

се случва: методите не са задължени да отстъпват и много методи (включително карта) имат добре

дефинирано поведение за случаи, в които има кодов блок и случаи, когато няма. Ако

изглежда пренебрегва блок, който очаквате да отстъпи, погледнете отблизо

правила за приоритет и се уверете, че блокът наистина е достъпен за метода.

Ще продължим да разглеждаме итератори и итерации, като правим с няколко вградени Ruby

итератори какво направихме с цикъл: изследване на метода и след това внедряване на нашия

собствен. Ще започнем с метод, който е леко усъвършенстване на loop: times.

***6.3.5***

***Време за изпълнение***

Методът times е метод на екземпляр от класа Integer, което означава, че го извиквате

като метод за цели числа. Той изпълнява кодовия блок *n* пъти, за всяко цяло число *n* , и при

в края на метода връщаната стойност е *n* .

д

|  |
| --- |
| **172 серия** |

**172**

C ГЛАВА 6 ***Техники за контрол на потока***

Можете да видите както изходната, така и връщаната стойност, ако стартирате пример за време в irb:

>> 5. пъти {поставя "Писане на това 5 пъти!" }

Пишете това 5 пъти!

Пишете това 5 пъти!

Пишете това 5 пъти!

Пишете това 5 пъти!

Пишете това 5 пъти!

=> 5

Извикването на метода включва кодов блок B, който се изпълнява пет пъти. The

върнатата стойност на *целия* метод е обектът, с който започнахме: цялото число 5 c .

Поведението на времената добре илюстрира факта, че отстъпването на блок и връщането-

от метода са две различни неща. Методът може да даде на своя блок всяко число

ber на пъти, от нула до безкрайност (последното в случай на цикъл). Но всеки метод

връща се точно веднъж (при условие, че няма фатални грешки), когато приключи всичко, което прави

ще направя. Това е малко като скок във фигурното пързаляне. Излитате, изпълнявате малко въртене

въздух и земя. И колкото и ротации да изпълните, вие само

излети веднъж и само веднъж кацнете. По подобен начин извикването на метод кара метода да се изпълнява

веднъж и да се върне веднъж. Но между тях, като въртене във въздуха, методът може

контрол на добива обратно към блока (ако има такъв) нула или повече пъти.

Преди да приложим времената, нека разгледаме друга негова характеристика. Всеки път пъти

отстъпва на своя блок, дава нещо. Разбира се, кодовите блокове, като методите, могат

вземете аргументи. Когато методът дава, той може да даде една или повече стойности.

Блокът взема аргумента чрез параметрите си. В случая на време, вие

може да предостави един параметър и този параметър ще бъде обвързан с каквато и да е стойност

се отстъпва на блока при всяка итерация. Както се досещате, стойностите, получени от

пъти са целите числа от 0 до *n -* 1:

>> 5. пъти {| i | поставя "Аз съм на итерация # {i}!" }

Аз съм на итерация 0!

Аз съм на итерация 1!

Аз съм на итерация 2!

Аз съм на итерация 3!

Аз съм на итерация 4!

=> 5

Всеки път през - т.е. всеки път, когато се отстъпва на кодовия блок - той дава

следваща стойност и тази стойност се поставя в променливата i.

Готови сме да внедрим пъти - или по-скоро my\_times - и ето как изглежда:

клас Цяло число

def my\_times

c = 0

докато c == самостоятелно

добив (c)

c + = 1

край

себе си

край

край

б

° С

|  |
| --- |
| **Страница 173** |

**173**

***Итератори и кодови блокове***

Ако искате да опитате пример в irb, можете или да въведете предишния код във вашия

irb сесия или я поставете във файл - да речем, my\_times.rb - и след това издайте командата require

"./my\_times.rb" в irb. След това можете да видите резултатите от опита:

>> 5.my\_times {| i | поставя "Аз съм на итерация # {i}!" }

Аз съм на итерация 0!

Аз съм на итерация 1!

Аз съм на итерация 2!

Аз съм на итерация 3!

Аз съм на итерация 4!

=> 5

Работи точно като времената. Прилагането на my\_times в никакъв случай не е най-много

възможно е сбито изпълнение, но работи. Ще научите много повече итерации и

техники за колективно манипулиране, които можете да използвате, за да направите дефинициите на вашия метод

едновременно кратко и ясно.

Говорейки за това, следващата ни спирка е всеки метод. Както ще видите тук и в

още по-голяма дълбочина в по-късните глави, където изследваме обектите на колекцията

Силно, всеки е зает, основен метод.

***6.3.6***

***Важността да бъдеш всеки***

Идеята на всеки е проста: стартирате всеки метод върху обект на колекция и всеки

връща всеки елемент от колекцията към вашия кодов блок, един по един. Руби има няколко

класове за събиране и дори повече класове, които са достатъчно подходящи за събиране, за да ги поддържат

всеки метод. Ще видите две глави, посветени на колекциите Ruby. Ето, ще го направим

наемете скромния масив за нашите примери.

Ето една проста операция:

масив = [1,2,3,4,5]

array.each {| e | поставя "Блокът току-що получи # {e}." }

Резултатът от всяко повикване изглежда така в irb сесия:

>> масив.each {| e | поставя "Блокът току-що получи # {e}." }

Блокът току-що получи 1.

Блокът току-що бе връчен 2.

Блокът току-що бе предаден 3.

Блокът току-що получи 4.

Блокът току-що получи 5.

=> [1, 2, 3, 4, 5]

Последният ред не е изходен метод; това е възвръщаемата стойност на всеки, отразен обратно от irb.

Връщаната стойност на всеки, когато му бъде даден блок, е неговият приемник, оригиналният масив.

(Когато не му бъде даден блок, той връща изчислител; ще научите за тези в

глава 10.) Както пъти, всеки няма вълнуваща възвръщаема стойност. Всички лихви

се крие във факта, че дава стойности на блока.

За да приложим my\_each, ще направим още една стъпка по линия на итерацията refine-

мента. С my\_loop повторихме завинаги. С my\_times повторихме *n* пъти. С

|  |
| --- |
| **Страница 174** |

**174**

C ГЛАВА 6 ***Техники за контрол на потока***

my\_each, броят на итерациите - броят пъти, които методът дава - зависи

върху размера на масива.

Нуждаем се от брояч, за да следим къде сме в масива и да продължаваме да отстъпваме

докато приключим. Удобно е, че масивите имат метод за размер, което го прави лесно

определете колко повторения (колко „завъртания във въздуха“) трябва да бъдат извършени

формиран. Като възвръщаема стойност за метода ще използваме оригиналния обект на масив:

клас Array

def my\_each

c = 0

докато c == размер

добив (самостоятелно [c])

c + = 1

край

себе си

край

край

Пробното пускане на my\_each води до резултата, към който се стремим:

>> масив = [1,2,3,4,5]

>> array.my\_each {| e | поставя "Блокът току-що получи # {e}." }

Блокът току-що получи 1.

Блокът току-що бе връчен 2.

Блокът току-що бе предаден 3.

Блокът току-що получи 4.

Блокът току-що получи 5.

=> [1, 2, 3, 4, 5]

Успешно внедрихме поне проста версия на всеки. Хубавото нещо

за всеки е, че е толкова ванилов: всичко, което прави, е да хвърля стойности в кодовия блок, едно в a

време, докато изтече. Важно значение за това е, че е възможно да се изгради

произволен брой по-сложни, семантично богати итератори *отгоре на* всеки. Ще приключим

това повторно изпълнение с един такъв метод: карта, в която видяхте накратко

раздел 6.3.4. Научаването на малко за картата също ще ни отведе в някои допълнителни нюанси на

писане и използване на блокови кодове.

Допълнителен кредит: Определете **my\_each** от гледна точка на **my\_times**

Интересно упражнение е да се определи my\_each, като се използва съществуващата дефиниция на

my\_times. Можете да използвате метода за размер, за да определите колко итерации са ви необходими

и след това ги изпълнете с любезното съдействие на my\_times, по следния начин:

клас Array

def my\_each

size.my\_times do | i |

добивам себе си [i]

край

себе си

край

край

**Използвайте [], за да получите ток**

**елемент на масив**

|  |
| --- |
| **Страница 175** |

**175**

***Итератори и кодови блокове***

***6.3.7***

***От всяка на карта***

Както всеки, картата преминава през масив по един елемент и дава всеки елемент

към кодовия блок. Разликата между всяка и картата се крие във връщаната стойност: всяка

връща своя приемник, но map връща нов масив. Новият масив винаги е с еднакъв размер

като оригинален масив, но вместо оригиналните елементи, новият масив съдържа

натрупани връщани стойности на кодовия блок от итерациите.

Ето пример за карта. Забележете, че връщаната стойност съдържа нови елементи; не е

само масивът, с който започнахме:

>> имена = ["Дейвид", "Алън", "Черен"]

=> [„Дейвид“, „Алън“, „Черен“]

>> names.map {| име | name.upcase}

=> ["DAVID", "ALAN", "ЧЕРНО"]

Резултатът от картографирането е нов масив, всеки от елементите на който съответства на елемента

в същото положение в оригиналния масив, но обработен чрез кодовия блок.

Частта от пъзела, която картата добавя към нашия анализ на итерацията, е идеята на

кодов блок, връщащ стойност *към* метода, който му е дал. И наистина го прави: просто

тъй като методът може да даде стойност, така и блокът може да върне стойност. Връщаната стойност

се връща като стойността на поканата за доходност.

За да приложим my\_map, трябва да уредим масив от акумулатори в

които ще изпуснем върнатите стойности на последователните извиквания към блока с кодове. Добре тогава

върнете масива на акумулатора като резултат от цялото извикване на my\_map.

Нека започнем с предварително, но не окончателно изпълнение, при което не го правим

надграждайте my\_each, но напишете my\_map от нулата. Целта е да се илюстрира точно

как картографирането се различава от обикновената итерация. След това ще усъвършенстваме изпълнението.

Първото изпълнение изглежда така:

клас Array

def my\_map

c = 0

в съответствие = []

докато c == размер

в съответствие << добив (самостоятелно [c])

c + = 1

край

съгл

край

край

***(продължение)***

Използването на my\_times ви спестява неприятностите при писане на код за брояч на цикли в my\_each. Но

това е малко назад: много от итераторите на Ruby са изградени върху всеки, а не върху другия

наоколо. Като се има предвид дефиницията на my\_each в основния текст, как бихте го използвали в

изпълнение на my\_times?

За разлика от първото упражнение, това наистина ще бъде оставено на вас да опитате сами!

**Инициализира**

**акумулаторен масив**

**Улавя връщаната стойност от**

**блок в масива на акумулатора**

**Се завръща**

**акумулаторен масив**

|  |
| --- |
| **176** |

**176**

C ГЛАВА 6 ***Техники за контрол на потока***

Сега получаваме същите резултати от my\_map, които направихме от картата:

>> names.my\_map {| име | name.upcase}

=> ["DAVID", "ALAN", "ЧЕРНО"]

Подобно на my\_each, my\_map дава на свой ред всеки елемент от масива. За разлика от my\_each,

my\_map съхранява стойността, която се връща от блока. Така натрупва

картографиране на старите стойности към новите стойности: новите стойности се основават на старата стойност

ues, обработени през блока.

Но нашата реализация на my\_map не успява да изпълни обещанието на my\_each -

обещава се, че всеки служи като ванилов итератор, отгоре на който колкото по-

могат да бъдат изградени плекс итератори. Нека да изпълним отново картата. Този път ще напишем my\_map

условия на my\_each.

Б КАРТА ЗА ИЗПЪЛНЕНИЕ НА ВСЯКА ВСЕКА

Изграждането на карта върху всяка от тях е почти поразително просто:

клас Array

# Поставете тук дефиницията на my\_each

def my\_map

в съответствие = []

my\_each {| e | в съответствие << добив (д)}

съгл

край

край

Превключваме ваниловия итератор, позволявайки на my\_each да направи разходката през

масив. Няма нужда да поддържате явен брояч или да пишете цикъл до.

Вече имаме тази логика; това е въплътено в my\_each. При писането на my\_map това прави

смисъл да се възползвате от него.

Има много, много повече да се каже за итератори и по-специално за начините, по които Руби

надгражда върху всеки, за да осигури изключително богат набор от методи за обработка на събиране.

Ще слезем по този път в глава 10. Междувременно, нека се задълбочим още малко

дълбоко в някои от гайките и болтовете на итераторите - започвайки от заданието и

правила за обхват, които регулират използването им на параметри и променливи.

***6.3.8***

***Параметри на блока и обхват на променливата***

Виждали сте, че параметрите на блоковете са заобиколени от тръби, а не от скоби като

параметрите на метода са. Но можете да използвате наученото за аргументите на метода

за създаване на списъци с параметри на блокове. Спомняте ли си метода args\_unleashed от глава 2?

def args\_unleashed (a, b = 1, \* c, d, e)

поставя "Аргументи:"

pa, b, c, d, e

край

Ето блокова версия на метода:

def block\_args\_unleashed

добив (1,2,3,4,5)

край

|  |
| --- |
| **Страница 177** |

**177**

***Итератори и кодови блокове***

block\_args\_unleashed направи | a, b = 1, \* c, d, e |

поставя "Аргументи:"

pa, b, c, d, e

край

Обвързването на параметрите и изходът на програмата са същите, както са били с оригинала

nal версия:

Аргументи:

1

2

[3]

4

5

Ами обхватът? Дефиницията на метод, както знаете, стартира нов локален обхват. Блокове

са малко по-сложни.

Нека започнем с прост случай: вътре в блок, вие се позовавате на променлива (а не на блок

параметър; просто променлива), наречена x, и вече имате обхват, наречен x

преди да напишете блока:

def block\_scope\_demo

x = 100

1. пъти

поставя х

край

край

Когато стартирате метода (който включва удобен оператор за поставяне), ще видите това

x вътре в блока е същият като x, който вече е съществувал:

block\_scope\_demo

Сега, какво ще кажете за присвояване на променливата вътре в блок? Отново се оказва, че

променливата вътре в блока е същата като тази, която е съществувала преди блока,

както можете да видите, като го промените вътре в блока и след това го разпечатате след

блокът е завършен:

def block\_scope\_demo\_2

x = 100

1. пъти

x = 200

край

поставя х

край

block\_scope\_demo\_2

С други думи, блоковете имат пряк достъп до променливи, които вече съществуват (като x in

примера). Въпреки това, блокирайте параметрите (имената на променливите между тръбите)

се държат по различен начин от непараметричните променливи. Ако имате променлива на дадено

име в обхват и също използвайте това име като един от вашите параметри на блока, след това двата

променливите - тази, която вече съществува и тази в списъка с параметри - *не* са

същите като всеки друг.

**Единична итерация служи за**

**създайте контекст на блока на кода**

**Изход: 100**

**Изход: 200**

|  |
| --- |
| **178** |

**178**

C ГЛАВА 6 ***Техники за контрол на потока***

ЗАБЕЛЕЖКА Въпреки че е важно само по себе си, фактът, че блоковете споделят локално

обхват с кода, който ги предшества, ще придобие допълнително значение, когато

ние разглеждаме Proc обекти и *затваряния* в глава 14. Ще научите, че блоковете могат

служат като тела на анонимни функционални обекти и тези обекти запазват

локалните променливи, които са в обхвата по време на създаването им - дори ако

функционалните обекти се връчват около други локални обхвати.

Погледнете променливите с име x в този пример:

def block\_local\_parameter

x = 100

[1,2,3] .всеки | x |

поставя "Параметър x е # {x}"

x = x + 10

поставя „Преназначен на x в блок; сега е # {x}“

край

поставя „Външният x все още е # {x}“

край

Резултатът от извикване на този метод е

Параметър x е 1

Преназначен на x в блок; сега е 11

Параметър x е 2

Преназначен на x в блок; сега е 12

Параметър x е 3

Преназначен на x в блок; сега е 13

Външният x все още е 100

X вътре в блока не е същото като x извън блока, защото x се използва като

параметър на блок. Дори преназначаването на x вътре в блока не презаписва

„Външен“ x. Това поведение ви позволява да използвате всяко име на променлива, което искате за вашия

блокират параметрите, без да се притеснявате дали променлива от същото

име вече е в обхвата.

Понякога може да искате да използвате временна променлива в даден блок, дори и да не е

един от параметрите, които се присвояват при извикване на блока. И когато го направите

това е хубаво да не се притеснявате, че случайно използвате повторно променлива от out-

отстрани на блока. Ruby предоставя специална нотация, указваща, че искате един или повече

променливи, които да са локални за блока, дори ако променливи със същото име вече съществуват: a

точка и запетая в списъка с параметри на блока.

Ето един пример. Забележете точка и запетая в списъка с параметри:

def block\_local\_variable

x = "Оригинал x!"

3. времена | i; x |

x = i

поставя "x в блока е вече {{x}"

край

поставя "x след завършването на блока е # {x}"

край

block\_local\_variable

**Външен х**

**(преди блока)**

**Блок**

**параметър x**

**Възлагане на**

**x вътрешен блок**

|  |
| --- |
| **179 серия** |

**179**

***Обработка на грешки и изключения***

Точката с запетая, последвана от x, показва, че блокът се нуждае от свой собствен x, несвързан с

всяко x, което може да е било създадено вече в обхвата извън блока. В изпита-

ple, ние присвояваме на x вътре в блока, но тези назначения не влияят на x that

съществували вече. Резултатът показва, че оригиналът x оцелява:

x в блока вече е 0

x в блока вече е 1

x в блока вече е 2

x след приключване на блока е Original x!

Със сигурност оригиналът x е защитен от промяна.

Променливите, изброени след точка и запетая, не се считат за параметри на блока; те

не се обвързвайте с нищо, когато блокът бъде извикан. Те са *запазени имена* -names

искате да можете да използвате като временни променливи вътре в блока, без да се налага

проверете за сблъсъци на имена извън блока.

В обобщение са ви достъпни три основни „вкуса“ на променливата на блока:

Локални променливи, които съществуват вече при създаването на блока

Блокови параметри, които винаги са локално блокирани

Истински локални блокове, които са изброени след запетаята и не са присвоени

но защитават всички едноименни променливи от външния обхват

С тези инструменти под ръка бихте могли да проектирате вашите блокове, така че те да правят какво

имате нужда от тях по отношение на променливите и обхвата, така че да не „клопате“ никакви

променливи от външния обхват, които не искате да клонирате.

Итераторите и кодовите блокове на Ruby ви позволяват да пишете и да използвате методи, които са инж

се споделиха да споделят собствената си функционалност със своите обаждащи се. Методът съдържа някои

логика и процедура, но когато извикате метода, предоставяте допълнителен код, който

попълва логиката и индивидуализира конкретното обаждане, което правите. Това е елегантно

функция с безкрайни приложения. Ще се върнем към итераторите, когато разгледаме

лекционни обекти в детайли в глави 10 и 11.

Но сега ще разгледаме друг механизъм за контрол на потока. Досега сме били оперирани

в кооперативен, ефективен пейзаж. Не винаги работи обаче така; и

един от най-важните аспекти на контролния поток, който трябва да разберете, е

без значение какво се случва, когато нещата се объркат.

***6.4***

***Обработка на грешки и изключения***

Още в глава 1 разгледахме как да тестваме кода за синтаксични грешки:

$ ruby ​​-cw име на файл.rb

Подаването на теста -cw означава, че Ruby може да стартира вашата програма. Но това не означава нищо

ще се обърка, докато програмата ви работи. Можете да напишете синтактично правилно

програма - програма, която интерпретаторът ще приеме и изпълни - която прави всичко

видове неприемливи неща. Ruby се справя с неприемливо поведение по време на изпълнение от

*повдигане на изключение* .

|  |
| --- |
| **180. стр** |

**180**

C ГЛАВА 6 ***Техники за контрол на потока***

***6.4.1***

***Повишаване и спасяване на изключения***

Едно *изключение* е специален вид на обекта, копие на изключението за клас или descen-

dant от този клас. *Повишаването* на изключение означава спиране на нормалното изпълнение на програмата

грам и или да се справите с проблема, с който сте се сблъскали, или да излезете от

програма напълно.

Кое от тези се случва - справяне с проблема или прекъсване на програмата -

зависи от това дали сте предоставили спасителна клауза. Ако не сте предоставили такъв

клауза, програмата се прекратява; ако имате, контролът преминава към спасителната клауза.

За да видите изключения в действие, опитайте да разделите на нула:

$ ruby ​​-e '1/0'

Руби прави изключение:

-e: 1: в `/ ': разделено на 0 (ZeroDivisionError)

от -e: 1: в "<main>"

ZeroDivisionError е името на това конкретно изключение. По-технически, това е

име на клас - низходящ клас на класа Изключение. Руби има цяло семейство

дърво на класовете на изключенията, всички те се връщат в крайна сметка към Exception.

С ОМЕ ОБЩИ ИЗКЛЮЧЕНИЯ

Таблица 6.1 показва някои общи изключения (всяко от които е клас, произхождащ от

Изключение), заедно с често срещаните причини, поради които са повдигнати, и пример за код

ще вдигне всеки един.

Таблица 6.1 Чести изключения

Име на изключение

Често срещани причини

Как да го вдигна

RuntimeError

Изключението по подразбиране, повдигнато от

рейз метод.

Повишаване

NoMethodError

На обект се изпраща съобщение, което не може

разрешаване на име на метод; на

по подразбиране метод\_пропускане повишава

това изключение.

a = Object.new

a.some\_unknown\_method\_name

NameError

Интерпретаторът го удря в идентификатор

не може да се разреши като променлива или метод

име.

a = some\_random\_identifier

IOError

Причинени от четене на затворен поток,

запис в поток само за четене и

подобни операции.

STDIN.puts ("Не пиши на

СТДИН! ")

Errno :: *грешка*

Семейство от грешки се отнася до I / O файл.

File.open (-12)

TypeError

Методът получава аргумент it

не може да се справи.

a = 3 + "не може да добави низ

до номер! "

ArgumentError

Причинени от грешен брой на

аргументи.

def m (x); край; м (1,2,3,4,5)

|  |
| --- |
| **181** |

**181**

***Обработка на грешки и изключения***

Можете да опитате тези примери в irb; ще получите съобщение за грешка, но сесията не трябва

прекратявам. irb е добър да направи потенциално фатални грешки нефатални - и можете да го направите

нещо подобно и във вашите програми.

***6.4.2***

***Спасителната ключова дума за спасяване!***

Това, че е повдигнато изключение, не означава, че програмата ви се прекратява. Можеш

обработвайте изключения - справете се с проблема и поддържайте програмата работеща - от

средства на ключовата дума за спасяване. Спасяването включва спасителен блок, който е ограничен

с ключови думи начало и край и има клауза за спасяване в средата:

print "Въведете номер:"

n = gets.to\_i

започнете

резултат = 100 / n

спасяване

поставя "Вашият номер не работи. Беше ли нула ???"

изход

край

поставя "100 / # {n} е # {резултат}."

Ако стартирате тази програма и въведете 0 като свой номер, операцията за разделяне (100 / n)

повдига ZeroDivisionError. Тъй като сте направили това в начален / краен блок с a

спасителна клауза, контролът се предава на спасителната клауза. Отпечатва се съобщение за грешка,

и програмата излиза.

Ако въведете нещо различно от 0 и разделянето е успешно, програмирайте контрола

прескача спасителния отчет и блокира и изпълнението се възобновява след това (с

обаждането към путове).

Можете да прецизирате тази техника, като посочите изключението, което искате да хванете.

Вместо обща инструкция за спасяване, която спасява всяка грешка, която е потомък

клас на StandardError, вие казвате на спасяването какво да спасите:

спасяване ZeroDivisionError

Това улавя един вид изключение, но не и други. Предимството е, че не сте

по-дълъг риск от неволно прикриване на някакъв друг проблем чрез спасяване

твърде жадно.

Спасяването на изключения в тялото на метода или блока на кода има няколко различни

функции, които си струва да се отбележат.

U пее RESCUE МЕТОДИ вътре и КОД БЛОКОВЕ

Началото на метод или кодов блок предоставя имплицитен контекст начало / край.

Следователно, ако използвате ключовата дума за спасяване вътре в метод или кодов блок, не го правите

трябва да кажем започнете изрично - ако приемете, че искате клаузата за спасяване да управлява

цял метод или блок:

def open\_user\_file

print "Файл за отваряне:"

име на файл = gets.chomp

fh = File.open (име на файл)

б

|  |
| --- |
| **182 серия** |

**182**

C ГЛАВА 6 ***Техники за контрол на потока***

добив fh

fh.close

спасяване

поставя "Не можах да отворя файла ви!"

край

Ако операцията B за отваряне на файл задейства изключение, контролът скача директно към спасяването

клауза в . Ключовите думи def / end служат за ограничаване на обхвата на спасителната операция.

Но може да искате да получите малко по-подробна информация за това, кои линии спасявате

клаузата се прилага за. В предишния пример клаузата за спасяване се задейства дори ако

изключение се повдига по причини, които нямат нищо общо с опита за отваряне на файла. За

например, ако извикването към get създаде изключение по някаква причина, спасителната клауза се изпълнява.

За да получите по-фино зърно, трябва да се върнете към използването на явна обвивка за начало / край:

def open\_user\_file

print "Файл за отваряне:"

име на файл = gets.chomp

започнете

fh = File.open (име на файл)

спасяване

поставя "Не можах да отворя файла ви!"

връщане

край

добив fh

fh.close

край

В тази версия клаузата за спасяване урежда само това, което идва между клавиша за започване -

дума Б и спасяване в . Освен това е необходимо да се даде изрична команда за връщане

в спасителната клауза d, защото в противен случай методът ще продължи да се изпълнява.

Досега разглеждахме как да хванем изключенията, създадени от Руби - и вие ще го направите

научете повече техники за улавяне на изключения. Но нека се обърнем сега към другата страна на

монета: как сами да създавате изключения.

***6.4.3***

***Изрично повишаване на изключенията***

Що се отнася до традиционната гъвкавост и компактна мощност на кодиране на Ruby, изключения

така да се каже, не са изключение. Можете да създадете изключения в собствения си код и вие

може да създаде нови изключения за повишаване.

За да създадете изключение, използвате рейз плюс името на изключението, което искате

повишаване. Ако не предоставите име на изключение (и ако не повдигате друго

вид изключение, както е описано в раздел 6.4.4), Ruby повдига доста общото

RuntimeError. Можете също така да дадете повиши втори аргумент, който се използва като съобщение

мъдрец низ, когато се повдигне изключението:

def fussy\_method (x)

повиши ArgumentError, "Имам нужда от число под 10", освен ако x <10

край

fussy\_method (20)

° С

б

° С

д

|  |
| --- |
| **183** |

**183**

***Обработка на грешки и изключения***

Ако се стартира от файл, наречен fussy.rb, този код отпечатва следното:

fussy.rb: 2: в `fussy\_method ': Имам нужда от номер под 10 (ArgumentError)

от fussy.rb: 5: в `<main>'

Можете също да използвате спасяване в такъв случай:

започнете

fussy\_method (20)

спасяване ArgumentError

поставя "Това не беше приемливо число!"

край

Хубаво ощипване е, че ако дадете повишаване на съобщение като единствен аргумент, а не като

вторият аргумент, когато клас на изключение е първият аргумент, повишава цифрите

че искате да предизвика RuntimeError, използвайки предоставеното съобщение. Тези два реда

са еквивалентни:

повдигнете "Проблем!"

повиши RuntimeError, "Проблем!"

Във вашите спасителни клаузи е възможно да уловите обекта за изключение в променлива и

заявете го за евентуално полезна информация.

***6.4.4***

***Заснемане на изключение в спасителна клауза***

За да присвоите обекта на изключение на променлива, вие използвате специалния оператор => заедно с

командата за спасяване. Обектът за изключение, както всеки обект, отговаря на съобщения.

Особено полезни са методите за обратно проследяване и съобщения. backtrace връща

масив от низове, представляващи стека повиквания по време на издигане на изключението:

имена на методи, имена на файлове и номера на редове, показващи пълна пътна карта на кода, който

е изпълнен по пътя към изключението. message връща низа на съобщение pro-

vided за рейз, ако има такъв.

За да видите тези съоръжения в действие, поставете предходната дефиниция на fussy\_method в

файла fussy.rb (ако още не сте го направили) и след това добавете следния блок начало / край:

започнете

fussy\_method (20)

спасяване ArgumentError => д

поставя "Това не беше приемливо число!"

поставя "Ето обратната следа за това изключение:"

поставя e.backtrace

поставя "И ето съобщението на обекта за изключение:"

поставя e.message

край

В спасителната клауза присвояваме обекта за изключение на променливата e B и след това питаме

обектът за изключение, за да покаже обратното си проследяване c и съобщението си d . Ако приемем, че сте

има един празен ред между fussy\_method и ключовата дума begin, ще видите следното

намаляване на изхода (и във всеки случай ще видите нещо почти идентично, въпреки че

номерата на редовете могат да се различават), когато стартирате fussy.rb:

б

° С

д

|  |
| --- |
| **184** |

**184**

C ГЛАВА 6 ***Техники за контрол на потока***

Това не беше приемливо число!

Ето обратната следа за това изключение:

fussy.rb: 2: в `fussy\_method '

fussy.rb: 6: в "<main>"

И ето съобщението на обекта за изключение:

Трябва ми номер под 10

Обратното проследяване ви показва, че бяхме в метода fussy\_method на ред 2 от

fussy.rb, когато беше повдигнато изключението и че преди това бяхме на ред 6 от

един и същ файл в контекста <main> - с други думи, на най-горното ниво на програмата (out-

страна на произволен клас, модул или дефиниция). Съобщението: „Имам нужда от номер

под 10 ”идва от призива за рейз вътре в fussy\_method.

Вашата спасителна клауза също може да повдигне отново изключението, което я е задействало.

R E - НАДИГАНЕ НА ИЗКЛЮЧЕНИЕ

Не е необичайно да искате да повдигнете отново изключение, като разрешите следващото местоположение на

извикайте стека, за да се справите, след като спасителният ви блок го е обработил. Може, например,

искате да регистрирате нещо за изключението, но все пак да го третира като изключение от

повикващия код.

Ето втора версия на блока начало / край от метода open\_user\_file

няколко примера назад. Тази версия предполага, че имате метод на регистрационен файл, който

връща записващ файл манипулатор на регистрационен файл:

започнете

fh = File.open (име на файл)

Какво се повдига: Изключение или клас на изключение?

Езикът за повишаване на изключения е базиран на класа: по-скоро повдигнете ZeroDivisionError

отколкото вдигнете ZeroDivisionError.new. Но наистина, екземпляри на класовете на изключения

се вдигат. Синтаксисът ви позволява да вдигнете клас, защото това изглежда по-добре и абстрактно

далеч факта, че е включена инстанция.

Можете да видите изключването на класа / екземпляра, ако изследвате обекта, който сте заснели

в спасителната клауза:

започнете

повишаване на ArgumentError

спасяване => д

р д. клас

край

Класът на обекта е ArgumentError; самият обект е екземпляр на Argument-

Грешка, а не класът ArgumentError.

Получавате отсрочка от въвеждането на .new и вашият код изглежда добре на високо ниво,

предоставяне на достатъчно информация, за да ви покаже какво се случва, без излишно

подробности за домакинството.

**ArgumentError**

|  |
| --- |
| **185 серия** |

**185**

***Обработка на грешки и изключения***

спасяване => д

logfile.puts ("Потребителят се опита да отвори # {име на файл}, # {Time.now}")

logfile.puts ("Изключение: # {e.message}")

повишаване

край

Идеята тук е да се прихване изключението, да се отбележи в лог файла и

след това го повишете отново, като извикате рейз. (Въпреки че няма аргумент за повдигане, от

в клауза за спасяване се установява, че искате да повдигнете отново изключението

обработва, а не обичайната обща RuntimeError.) Мястото в програмата, което

наречен open\_user\_file на първо място, след това трябва да се справи с изключението - или не,

ако е по-добре да му позволите да спре изпълнението на програмата.

Друго усъвършенстване на управлението на потока за управление с изключения е осигуряването

клауза, която изпълнява безусловно, независимо какво друго се случва, когато

ния се повдига.

***6.4.5***

***Клаузата за осигуряване***

Да предположим, че искате да прочетете ред от файл с данни и да създадете изключение, ако редът

не включва определен подниз. Ако все пак включва подниза, вие искате

върнете линията. Ако не стане, искате да повдигнете ArgumentError. Но каквото и да се случи,

искате да затворите манипулатора на файла, преди методът да приключи.

Ето как можете да постигнете това, като използвате клауза за осигуряване:

def line\_from\_file (име на файл, подниз)

fh = File.open (име на файл)

започнете

линия = fh.gets

повишаване на ArgumentError, освен ако line.include? (substring)

спасяване ArgumentError

поставя "Невалиден ред!"

повишаване

осигурете

fh.close

край

обратна линия

край

В този пример блокът начало / край обгръща реда, който чете от файла, и

спасителната клауза обработва само ArgumentError - което означава, че ако нещо друго

се обърка (като файла, който не съществува), той не е спасен. Но ако ArgumentError е

повдигнато въз основа на теста за включване на подниза в линията на низ, спасяването

клауза се изпълнява.

Освен това клаузата за осигуряване се изпълнява, независимо дали е повдигнато изключение или не.

уверете се, че е фиксирана към началната / крайната структура, от която е част, и нейното изпълнение е

безусловен. В този пример искаме да гарантираме, че манипулаторът на файла се затваря.

Клаузата за осигуряване се грижи за това, каквото и друго да се е случило.

|  |
| --- |
| **186** |

**186**

C ГЛАВА 6 ***Техники за контрол на потока***

ЗАБЕЛЕЖКА Има по-добър начин за отваряне на файл, включващ кодов блок, който се увива

файловите операции и се грижи за затварянето на файла вместо вас. Но едно нещо в

време; ще видите тази техника, когато разгледаме техниките за файлове и I / O в

глава 12.

Един дълготраен проблем с метода line\_from\_file е, че ArgumentError не е

най-доброто име за изключението, което повдигаме. Най-доброто име би било нещо

като InvalidLineError, който не съществува. За щастие можете да създадете свой собствен

класове за изключения и им дайте име, каквото искате.

***6.4.6***

***Създаване на собствени класове за изключения***

Създавате нов клас на изключение чрез наследяване от Exception или от потомък

клас на изключение:

клас MyNewException <Изключение

край

повиши MyNewException, "възникна някакъв нов вид грешка!"

Тази техника предлага две основни предимства. Първо, като ви позволи да дадете нови имена на

класове изключения, той изпълнява функция за самодокументиране: когато MyNewException получи

повдигнат, той се различава от, да речем, ZeroDivisionError или обикновена ванилия RuntimeError.

Второ, този подход ви позволява да определите точно вашите спасителни операции. След като създадете

ated MyNewException, можете да го спасите по име:

клас MyNewException <Изключение

край

започнете

поставя „Предстои да се направи изключение ...“

повиши MyNewException

спасяване MyNewException => д

поставя „Току-що е направило изключение: # {e}“

край

Резултатът от този фрагмент е както следва:

На път да повдигнете изключение ...

Току-що повдигнах изключение: MyNewException

Само грешките MyNewException ще бъдат заловени от тази спасителна клауза. Ако друг

първо се повдига по някаква причина, това ще доведе до прекратяване на програмата без спасяване.

Ето как би изглеждал нашият метод line\_from\_file с персонализиран excep-

- заедно с кода, който създава потребителски клас на изключение. Ще наследим от

StandardError, суперкласът на RuntimeError:

клас InvalidLineError <StandardError

край

def line\_from\_file (име на файл, подниз)

fh = File.open (име на файл)

линия = fh.gets

повишаване на InvalidLineError, освен ако line.include? (substring)

обратна линия

|  |
| --- |
| **187** |

**187**

***Обобщение***

спасяване InvalidLineError

поставя "Невалиден ред!"

повишаване

осигурете

fh.close

край

Този път напълно определихме изключението, което искаме да засечем.

Просто чрез наследяване от StandardError, InvalidLineError предоставя значение-

пълно име на изключение и усъвършенства семантиката на спасителната операция. Персонализирани

класовете за производство са лесни и евтини и могат да добавят значителна стойност. Самата Руби

има много класове изключения - така че вземете намека и не се колебайте да създадете свой собствен

по всяко време чувствате, че нито едно от вградените изключения не изразява точно това, от което се нуждаете.

И не забравяйте, че изключенията са класове, класовете са константи, а константите могат

да бъдат пространствени имена, с любезното съдействие на гнезденето:

модул TextHandler

клас InvalidLineError <StandardError

край

край

def line\_from\_file (име на файл, подниз)

fh = File.open (име на файл)

линия = fh.gets

повиши TextHandler :: InvalidLineError, освен ако line.include? (подниз)

Изключенията за променливи имена по този начин са учтиви, в смисъл, че позволяват на други хора да назовават имена

изключения, както им харесва, без да се страхуват от сблъсъци с имената. Веднъж се превръща и в необходимост

започвате да създавате повече от много малък брой класове изключения.

С нашето проучване на изключенията и начина, по който се обработват, стигнахме до

края на това изследване на контролния поток. Както видяхте, контролът може да скача около a

справедлива сума - но ако имате предвид различните видове скачане (условни,

цикли, итератори и изключения), ще можете да следвате всеки Ruby код и да пишете

код, който използва продуктивно многото налични техники, свързани с потока.

***6.5***

***Обобщение***

В тази глава сте виждали

Условни условия (ако / освен ако и случай / кога)

Цикли (цикъл, за, докато и докато)

Итератори и кодови блокове, включително параметри на блокове и променливи

Примери за внедряване на Ruby методи в Ruby

Изключения и обработка на изключения

Тази глава обхваща няколко широкообхватни теми, свързани заедно, защото те

имат общо факта, че те включват контролен поток. Кондиционерите преместват контрол

на основата на истината или лъжата на изразите. Циклите повтарят сегмент от

кодирайте безусловно, условно или веднъж за всеки елемент в списък. Итератори - meth-

коефициенти, които се отнасят до кодов блок, който предоставяте заедно с извикването на метода - са

**Хубаво пространство с имена**

**име на изключение!**

|  |
| --- |
| **188** |

**188**

C ГЛАВА 6 ***Техники за контрол на потока***

сред най-отличителните черти на Руби. Научихте как да пишете и да наричате итера-

tor, техники, с които ще се сблъсквате често по-късно в тази книга (и след това).

Изключение правят механизмът на Ruby за обработка на невъзстановими условия на грешка.

*Невъзстановимото* е относително: можете да спасите състояние на грешка и да продължите изпълнението, но

трябва да организирате умишлена намеса чрез спасителен блок и по този начин да отклоните и спечелите

контрол на програмата, където в противен случай тя би прекратила. Можете също така да създадете своя

притежават класове на изключения чрез наследяване от вградените класове на изключения Ruby.

На този етап ще се задълбочим във вградената функционалност на Ruby, започвайки с някои

всеобхватни характеристики и техники и преминаване към специфични класове и модификации

ules. Не че вече не сте виждали и използвали много вградени функции; но е време да

станете по-систематични и по-задълбочено в работата на вградените модули.

|  |
| --- |
| **189** |

*Част 2*

*Вградени класове*

*и модули*

В част 2 стигаме до сърцевината на езика Ruby: вградени класове и

модули.

Голяма част от това, което ще направите като рубист, ще включва вградените в него руби. Вие сте

вече видяни примери, включващи много от тях: низове, масиви, файлове и т.н.

Ruby ви предоставя богат набор от инструменти, които са вградени в кутията

можете да използвате и върху които можете да надграждате.

Това е нещото: когато проектирате свои собствени класове и модули, често

установете, че това, от което се нуждаете, е нещо подобно на съществуващ клас Ruby. Ако ти си

например, като напишете клас DeckOfCards, една от първите ви мисли ще бъде

колода карти е много като масив. Тогава бихте искали да помислите

дали класът ви на карти трябва да бъде подклас на Array - или може би всяка колода

object може да съхранява масив в променлива на екземпляр и да поставя картите там - и

и така нататък. Въпросът е, че вградените класове на Ruby ви осигуряват отправни точки

за вашия собствен клас и обект дизайн, както и с набор от класове изключително използване-

пълни сами по себе си.

Ще започнем част 2 с разглеждане на вградените основни неща (глава 7). Целта на

тази глава е да ви предостави набор (така да се каже!) от техники и инструменти

че ще намерите полезно в общи линии в следващите глави. Да уча

низове, масиви и хешове, например, полезно е да знаете как се справя Ruby

концепциите за *вярно* и *невярно -* понятия , които не са обвързани с нито една вградена

клас, но че трябва да разберете най-общо.

|  |
| --- |
| **190** |

Следвайки най-важното, ще се обърнем към конкретни класове, но групирани в по-високо ниво

категории: *скаларни* обекти първо (глава 8), а след това *колекции* (глава 9). Скаларите са

атомни обекти, като низове, числа и символи. Всеки скаларен обект представлява един

стойност; скаларите не съдържат други обекти. (Низовете съдържат символи, разбира се; но

в Ruby няма отделен клас символи, така че низовете все още са скаларни.) Collection

обектите съдържат други обекти; основните класове за събиране в Ruby са масиви и

хешове. Проучването за събиране също ще включва *диапазони* , които са хибридни обекти, които

може (но не винаги) да служи за представяне на колекции от обекти. Накрая ще разгледаме

набори, които са внедрени в стандартната библиотека (а не в ядрото на Ruby), но

които заслужават изключение от общото правило, че фокусът ни е върху самото ядро.

По значение по отношение на специфичните класове за събиране са съоръженията, които всички събират

лекции в Ruby share: съоръжения, въплътени в модула Enumerable. Изброими

дарява колекционни обекти със знанието как да ги прехвърля и трансформира-

аз в много начини. Глава 10 ще бъде посветена на изброимите модули

ule и неговите последици за мощността на програмиране на Ruby.

Част 2 продължава в глава 11 с поглед към регулярните изрази - свързани с низове

тема, която въпреки това заслужава известно пространство - и завършва в глава 12

с проучване на файлове и I / O операции: четене от и запис във файлове и

I / O потоци и свързани подтеми като обработка на грешки и заявки за състояние на файла. Не

замислено Ruby третира всички тези неща, включително регулярни изрази и I / O потоци,

като обекти.

До края на част 2 ще имате богато работно познание за основните класове на Ruby,

и вашите собствени хоризонти Руби ще се разширят драстично.

|  |
| --- |
| **Страница 191** |

**191**

*Вградени основни неща*

По-късните глави в тази част на книгата ще обхващат специфични вградени класове: какво

те са, какво можете да правите с тях и какви методи имат техните екземпляри. Това

Глава ще обсъди селекция от теми, които обхващат редица вградени класове.

Това е нещо повече от това: това е и вид грамотност от следващо поколение Ruby

ръководство, по-задълбочена и по-широка версия на глава 1. Както глава 1, тази глава има две

цели: да се даде възможност да се вземе определено количество материал за даденост по-късно

глави, където тя ще възникне на различни места в различна степен; и ви представя

с информация за Ruby, която е важна и използваема сама по себе си. През-

от тази глава ще изследвате и богатството, което се крие във всеки обект на Ruby

***Тази глава обхваща***

Конструктори на буквални обекти

Синтактична захар

„Опасни“ и / или разрушителни методи

Семейството на методите за преобразуване to\_ \*

Булеви състояния и обекти и нула

Техники за сравнение на обекти

Проверка по време на изпълнение на възможностите на обектите

|  |
| --- |
| **192** |

**192**

C ГЛАВА 7 ***Вградени основни неща***

някои от синтактичните и семантичните подсистеми, които правят езика толкова интересен

и гъвкав.

Главата преминава през редица теми, така че вероятно си заслужава да имате

погледнете предварително какво ще видите. Ето леко коментирано резюме:

Буквални *конструктори -* начини за създаване на определени обекти със синтаксис, а не с a

обадете се на нов

*Синтактична захар* - Нещата, които Ruby ви позволява да направите вашия код да изглежда по-хубав

„*Опасни“ и / или разрушителни методи* - Методи, които променят постоянството на техните приемници

и други съображения за „опасност“

*Семейството на методите за преобразуване to\_ \** - Методи, които произвеждат преобразуване от

обект към обект от различен клас и синтактичните характеристики, които се свързват

в тези методи

*Булеви състояния и обекти, и нула* - Поглед отблизо на истински и неверни и свързани

концепции в Ruby

*Техники за сравнение на обекти - Техники за* рубин, както по подразбиране, така и персонализирани -

в състояние, за сравнение обект-обект

*Проверка по време на изпълнение на възможностите на обектите* - важен набор от техники за изпълнение

рефлексия върху възможностите на даден обект

Всички тези теми ще ви бъдат полезни, докато четете и / или пишете Ruby код в работа

чрез тази книга и след това.

Може да искате да стартирате irb сесия за тази глава; той често използва

irb формат на сесията за примерите на кода и често можете да опитате примерите с

малки вариации, за да усетите как се държи Руби.

***7.1***

***Буквалните конструктори на Ruby***

Ruby има много вградени класове. Повечето от тях могат да бъдат създадени чрез new:

str = String.new

arr = Array.new

Някои не могат; например не можете да създадете нов екземпляр на класа Integer. Но за

в по-голямата си част можете да създавате нови копия на вградените класове.

В допълнение, късметлия, избрани няколко вградени класа се ползват с привилегията да имат *буквални*

*конструктори* . Това означава, че можете да използвате специална нотация, вместо да извиквате нова, за да създавате

нов обект от този клас.

Класовете с литерални конструктори са показани в таблица 7.1. Когато използвате един от

тези литерални конструктори, вие въвеждате нов обект в съществуване. (Въпреки че не е така

очевидно от таблицата, заслужава да се отбележи, че няма нов конструктор за Symbol

обекти. Единственият начин за генериране на обект Symbol е с литералния конструктор.)

Ще разгледаме доста подробно много функционалност във всички тези класове.

Междувременно започнете да свиквате с нотацията, за да можете да разпознавате тези типове данни

на поглед. Буквалните конструктори никога не са единственият начин да се създаде екземпляр на обект от дадена

клас, но те са много често използвани.

|  |
| --- |
| **Страница 193** |

**193**

***Повтаряща се синтактична захар***

След това ще разгледаме някои от синтактичната захар, която Ruby ви предоставя на разположение

спектъра на обектите.

***7.2***

***Повтаряща се синтактична захар***

Както знаете, Ruby понякога ви позволява да използвате захарно обозначение вместо обичайното

object.method (args) синтаксис за извикване на метод. Това ви позволява да правите хубави неща,

като например използване на знак плюс между две числа, като оператор

x = 1 + 2

вместо странния еквивалент на метода:

x = 1. + (2)

Докато задълбавате по-задълбочено в Ruby и вградените в него методи, имайте предвид, че някои

методи винаги получават това лечение. Последицата е, че можете да определите как вашият

обектите се държат в код по този начин

my\_object + my\_other\_object

Таблица 7.1 Вградени Ruby класове с литерални конструктори

Клас

Буквален конструктор

Примери)

Струна

Кавички

"нов низ"

'нов низ'

Символ

Водещо дебело черво

: символ

: "символ с интервали"

Масив

Квадратни скоби

[1,2,3,4,5]

Хеш

Къдрави скоби

{"Ню Йорк" =>"Ню Йорк",

"Орегон" =>"ИЛИ"}

Обхват

Две или три точки

0..9 или 0 ... 10

Regexp

Наклонени черти напред

/ ([az] +) /

Proc (ламбда)

Тире, стрелка, скоби, скоби

-> (x, y) {x \* y}

Буквални конструкторни знаци с повече от едно значение

Някои от обозначенията, използвани за литерални конструктори, имат повече от едно значение в

Руби. Много обекти имат метод, наречен [], който прилича на литерален конструктор на масив

но не е. Блоковете на кода, както видяхте, могат да бъдат разграничени с къдрави скоби - но

те все още са кодови блокове, а не хеш литерали. Този вид претоварване на нотация е a

последица от крайния брой символи на клавиатурата. Винаги можете да кажете

какво означава обозначението под неговия контекст и има малко контексти, с които

малко практика, ще бъде лесно да се разграничи.

|  |
| --- |
| **Страница 194** |

**194**

C ГЛАВА 7 ***Вградени основни неща***

просто чрез дефиниране на метода +. Виждали сте този процес на работа, особено в

връзка с равенство на делата и дефиниране на метода ===. Но сега да разгледаме повече

широко с тази елегантна техника.

***7.2.1***

***Определяне на оператори чрез дефиниране на методи***

Ако дефинирате метод + за вашия клас, тогава обектите от вашия клас могат да използват захарирания

синтаксис за добавяне. Освен това няма такова нещо като определяне на смисъла на това

синтаксис отделно от дефинирането на метода. Операторът е методът. Просто изглежда

по-хубав като оператор.

Не забравяйте също, че семантиката на методи като + изцяло се основава на конвенционални

ция. Руби не знае, че + означава добавяне. Нищо (освен добра преценка)

спира ви да пишете напълно неприсъединителни + методи:

obj = Object.new

def obj. + (other\_obj)

"Опитваш се да ми добавиш нещо, а?"

край

поставя obj + 100

Знакът плюс в оператора put е извикване на метода + на obj с цялото число

100 като единичен (игнориран) аргумент.

Наслоена върху захарта в стил оператор е пряката захар: x + = 1 за x = x + 1.

За пореден път автоматично прибирате реколтата от захар, ако определите съответната

метод (и). Ето пример - клас на банкова сметка с методи + и -:

клас Акаунт

attr\_accessor: баланс

def инициализиране (количество = 0)

self.balance = сума

край

def + (x)

self.balance + = x

край

def - (x)

self.balance - = x

край

def to\_s

баланс.до\_с

край

край

acc = Account.new (20)

в съответствие - = 5

поставя съгл

Чрез дефиниране на метода - екземпляр B , получаваме пряк път - = и можем да изваждаме от

акаунта, използващ тази нотация c . Това е прост, но поучителен пример за факта

че Руби ви насърчава да се възползвате от същото „окабеляване“, което

guage използва, така че да интегрира вашите програми възможно най-гладко в

основната технология.

**Без добавяне,**

**просто изход**

б

° С

**Изход: 15**

|  |
| --- |
| **Страница 195** |

**195**

***Повтаряща се синтактична захар***

Автоматизираните захарирани методи са събрани в таблица 7.2.

Спомнянето си кои методи получават лечение със захар не е трудно. Те попадат в седем-

различни отделни категории, както показва таблица 7.2. Тези категории са за удобство на

само обучение и справка; Руби не категоризира методите и отговорностите

Прилагането на смислена семантика зависи от вас. Наименованията на категориите

кажете как тези имена на методи се използват във вградените класове на Ruby и как са най-много

често се използват по конвенция, когато програмистите ги прилагат в нови класове.

(Не забравяйте също оператора за условно присвояване || =, както и неговите рядко спот-

ted cousin && =, и двете предоставят същия вид пряк път като псевдо-оператора

методи, но се основават на оператори, а именно || и &&, че не можете да отмените.)

Широкото използване на този вид синтактична захар - където нещо *прилича* на

оператор, но *е* извикване на метод - разказва ви много за философията зад Ruby като a

Таблица 7.2 Методи със синтактична нотация за извикване на захар в операторски стил

Категория

Име

Пример за дефиниция

Пример за повикване

Захарна нотация

Аритметичен метод /

оператори

+

def + (x)

obj. + (x)

obj + x

-

def - (x)

obj .- (x)

obj - x

\*

def \* (x)

obj. \* (x)

obj \* x

/

def / (x)

obj./(x)

obj / x

% (по модул)

def% (x)

obj.% (x)

obj% x

\*\* (експонента) def \*\* (x)

obj. \*\* (x)

obj \*\* x

Вземете / задайте / добавете данни

[]

def [] (x)

obj. [] (x)

obj [x]

[] =

def [] = (x, y)

obj. [] = (x, y) obj [x] = y

<<

def << (x)

obj. << (x)

obj << x

Метод за сравнение /

оператори

<=>

def <=> (x)

obj. <=> (x)

obj <=> x

==

def == (x)

obj. == (x)

obj == x

>

def> (x)

obj.> (x)

obj> x

<

def <(x)

obj. <(x)

obj <x

> =

def> = (x)

obj.> = (x)

obj> = x

<=

def <= (x)

obj. <= (x)

obj <= x

Оператор за равенство на делата

===

def === (x)

obj. === (x)

obj === x

Побитови оператори

| (ИЛИ)

def | (x)

obj. | (x)

obj | х

& (И)

def & (x)

obj. & (x)

obj & x

^ (XOR)

def ^ (x)

obj. ^ (x)

obj ^ x

|  |
| --- |
| **196 страница** |

**196**

C ГЛАВА 7 ***Вградени основни неща***

програмен език. Фактът, че можете да дефинирате и дори да предефинирате елементи като

знак плюс, знак минус и квадратни скоби означава, че Руби има много

гъвкавост. Но има ограничения за това, което можете да предефинирате в Ruby. Не можете да предефинирате

всеки от конструкторите на литерални обекти: {} винаги е хеш литерал (или кодов блок, ако е

се появява в този контекст), "" винаги ще ограничава низ и т.н.

Но има много неща, които можете да направите. Можете дори да дефинирате някои унарни оператори чрез

дефиниции на метода.

***7.2.2***

***Персонализиране на унарни оператори***

Унарните оператори + и - се срещат най-често като знаци за числа, както в -1. Но

те могат да бъдат дефинирани; можете да посочите поведението на изразите + obj и -obj за

вашите собствени обекти и класове. Правите това, като дефинирате методите + @ и - @.

Да кажем, че искате + и - да означават главни и малки букви за низ като

обект. Ето как определяте подходящото поведение на унарния оператор, използвайки банер

клас като пример:

клас Банер

def инициализиране (текст)

@text = текст

край

def to\_s

@текст

край

def + @

@ text.upcase

край

def - @

@ text.downcase

край

край

Сега създайте банер и манипулирайте неговия случай с помощта на унарните оператори + и -:

banner = Banner.new („Яжте при Дейвид!“)

поставя банер

поставя + банер

поставя -банер

Основният изходен низ за текста на банера, непроменен, се предоставя от to\_s con-

версия метод Б , който ще видите отблизо в раздел 7.4.1.

Можете също така да дефинирате! (логически *не* ) оператор, като дефинира! метод. Всъщност,

определяне на! метод ви дава и унарните! а ключовата дума не. Нека добавим a

определение към Банер:

клас Банер

деф!

обратен

край

край

б

**Резултат: Яжте при Дейвид!**

**Изход: ХРАНИ СЕ НА ДЕЙВИД!**

**Резултат: яжте при Дейвид!**

|  |
| --- |
| **197 стр** |

**197**

***Методи и "опасност"***

Сега разгледайте банера, „отхвърлен“. Ще трябва да използваме скоби около not

версия за изясняване на предимството на изразите (в противен случай поставя мисли, че се опитваме

не печат):

поставя! банер

поставя (не банер)

Както често се случва, Ruby ви дава обектно-ориентиран метод, базиран на метода, за да персонализирате какво

първоначално може да си помислите, че са синтактични характеристики с твърди кабели - дори унарните оператори харесват!

Унарното отрицание не е единственото използване на Ruby на удивителен знак.

***7.3***

***Методи и "опасност"***

Рубинните методи могат да завършват с удивителен знак (!) Или да взривят. Взривът няма значение

вътрешно присъствие на Руби; bang методите се извикват и изпълняват, както всеки друг

методи. Но по споразумение взривът обозначава метод като „опасен“ - по-специално като

опасният еквивалент на метод със същото име, но без взрив.

*Опасно* може да означава каквото иска човекът, който пише метода, да означава. В

в случая на вградените класове, това обикновено означава *този метод, за разлика от неговия небанг еквивалент* ,

*трайно модифицира своя приемник* . Не винаги обаче: излезте! е опасна алтернатива

тивен за излизане, в смисъл, че не пуска никакви финализатори на изхода от про-

грам. Опасността в под! (метод, който замества низ за замяна на

съвпадащ модел в низ) е отчасти това, че променя приемника си и отчасти това

връща нула, ако не е настъпила промяна - за разлика от sub, която винаги връща копие на

оригиналният низ с направена подмяна (или без замяна).

Ако „опасността“ е твърде мелодраматична за вас, можете да се сетите! в имената на методите като

един вид „Внимание!“ И с много малко, много специализирани изключения, всеки взрив

метод трябва да се появи в двойка с несвързан еквивалент. Ще се върнем към въпросите на

най-добрата практика за наименуване на методи, след като разгледахме някои взривни методи в действие.

***7.3.1***

***Деструктивни (променящи приемника) ефекти като опасност***

Без съмнение повечето от методите за взрив, които ще срещнете в основния език Ruby

да ги чуете, защото те са разрушителни: те променят обекта, върху който

те се наричат. Извикването на upcase на низ ви дава нов низ, състоящ се от

nal низ в главни букви; но upcase! превръща оригиналния низ в собствена главна буква

еквивалент, на място:

>> str = "Здравейте"

=>"Здравей"

>> str.upcase

=>"ЗДРАВЕТЕ"

>> ул

=>"Здравей"

>> str.upcase!

=>"ЗДРАВЕТЕ"

>> ул

=>"ЗДРАВЕТЕ"

**Изход:! S'divaD ta taE**

**Изход:! S'divaD ta taE**

б

° С

|  |
| --- |
| **198 стр** |

**198**

C ГЛАВА 7 ***Вградени основни неща***

Разглеждането на оригиналния низ след преобразуването му в главни букви показва, че горната-

версията на случая беше копие; оригиналния низ е непроменена B . Но операцията взрив

е променил съдържанието на самата str c .

Основните класове на Ruby са пълни с разрушителни (променящи приемника) методи за взрив

сдвоени с техните неразрушаващи колеги: сортиране / сортиране! за масиви, лента / лента!

(лента, водеща и последваща празно пространство) за низове, обратно / обратно! за струни и

масиви и много други. Във всеки случай, ако извикате небанг версията на метода

върху обекта, получавате нов обект. Ако се обадите на bang версията, вие оперирате на място

същия обект, на който сте изпратили съобщението.

Винаги трябва да сте наясно дали методът, който извиквате, променя своя

приемник. Нито една от двете опции не винаги е правилна или грешна; кое е най-доброто зависи от това какво сте

правиш. Едно съображение, претегляне отстрани на модифициращи обекти вместо на създаване

при привличането на нови е ефективност: създаване на нови обекти (като втори низ, който е идентичен

до първата с изключение на една буква) е скъпа по отношение на паметта и обработката.

Това няма значение, ако имате работа с малък брой обекти. Но когато получите

в, да речем, обработка на данни от големи файлове и използване на цикли и итератори за това, създаване

новите обекти могат да бъдат източване на ресурси.

От друга страна, трябва да внимавате да модифицирате обектите на място,

тъй като други части на програмата могат да зависят от тези обекти, за да не се променят. За

Например, да предположим, че имате база данни с имена. Прочетете имената от данните -

база в масив. В даден момент трябва да обработите имената за печатни резултати -

всичко с главни букви. Можете да направите нещо подобно:

имена.всеки | име |

capped = name.upcase

# ... код, който прави нещо с ограничено ...

край

В този пример ограничението е нов обект - главен дубликат на име. Когато ти

преминете през същия масив по-късно, в ситуация, в която *не* искате имената

главни букви, като например да ги запазите обратно в базата данни, имената ще бъдат както са

са били първоначално.

Чрез създаване на нов низ (с ограничение), който да представя версията с главни букви на всеки

име, вие избягвате страничния ефект от промяната на имената за постоянно. Операцията

изпълнявате върху имената, постига целите си, без да променя основното състояние на

данни. Понякога ще искате да промените обект за постоянно, а понякога и вие

няма да иска. В това няма нищо лошо, стига да знаете кой сте

прави и защо.

Освен това, не приемайте пряка връзка между взривните методи и

деструктивни методи. Те често съвпадат, но не са едно и също нещо.

***7.3.2***

***Деструктивността и „опасността“ варират независимо***

Това, което следва тук, е някои коментари на конвенциите и най-добрите практики. Руби

не му пука; Руби е щастлив да изпълнява методи, чиито имена завършват на! дали

|  |
| --- |
| **199** |

**199**

***Методи и "опасност"***

те са опасни, безопасни, сдвоени с небанг метод, не сдвоени - каквото и да е. The

стойност на! нотация като знак за комуникация между автор на метод и a

потребителят на този метод се намира изцяло в конвенциите. Струва си да спечелите солидна

положението на тези конвенции и защо имат смисъл.

Най-добрият съвет за това кога да се използват имена на методи, прекратени с взрив, е ...

D ON " T УПОТРЕБА ! С изключение на М / М ! ЧЕТКИ НА МЕТОД

The! нотация за име на метод трябва да се използва само когато има метод на

същото име без!, когато връзката между тези два метода е тази

и двамата правят по същество едно и също и когато версията с взрив има и страна

ефекти, различна възвръщаема стойност или някакво друго поведение, което се отклонява от неговото не-

взрив колега.

Не използвайте! само защото смятате, че вашият метод е опасен в някаква неяснота,

абстрактен начин. Всички методи правят нещо; това само по себе си не е опасно. The! е

предупреждение, че може да се случва повече, отколкото подсказва името - и това, в

turn, има смисъл само ако името се използва за метод, който няма дан-

щедро поведение.

Не назовавайте метод за запазване! само защото пише във файл. Извикайте този метод за запазване,

и след това, ако имате друг метод, който записва във файл, но (да речем) не архивира

оригиналния файл (ако приемем, че save го прави), продължете и извикайте този save !.

Ако откриете, че пишете един метод за запис във файла, и поставяте! в

край, защото се притеснявате, че методът е твърде мощен или твърде несигурен, трябва

преразгледайте именуването на метода. Всеки опитен рубист, който вижда спасяване! метод

документираното ще иска да знае как се различава от запазването. Удивителен знак

не означава нищо изолирано; има смисъл само в края на една от двойка

иначе идентични имена на методи.

D ON ' T EQUATE ! Нотация С деструктивно поведение , или обратно

Опасността в смисъл на взрив обикновено означава поведение, променящо обекта или „разрушително“ поведение. Това е

следователно не е необичайно да чуете хората да твърдят, че! означава разрушително. От

там не е голям скок да започнете да се чудите защо някои деструктивни методи "

имената не завършват с!

Този ред на мисли е проблематичен от самото начало. Взривът не означава

разрушително; това означава опасно, евентуално неочаквано поведение. Ако имате метод

наречен upcase и искате да напишете разрушителна версия на него, можете да го извикате

destructive\_upcase; нито едно правило не казва, че трябва да добавите a! към оригиналното име. Това е просто a

конвенция, но е изразителна.

Деструктивните методи не винаги завършват с!, Нито това би имало смисъл. Много

nonbang методите имат имена, които ви карат да *очаквате* приемника да се промени. Тези

методите нямат неразрушаващи аналози. (Какво би означавало да имаш не-

структурна версия на String # clear, която премахва всички символи от низ и оставя

равно на ""? Ако не променяте низа на място, защо просто не напишете ""

на първо място?) Ако име на метод без взрив вече предлага модификация на място

или друг вид „опасно поведение“, тогава това не е опасен метод.

|  |
| --- |
| **Страница 200** |

**200**

C ГЛАВА 7 ***Вградени основни неща***

Почти със сигурност ще откриете, че конвенционалната употреба на! нотацията е

най-елегантното и логично използване. Най-добре е да не пляскате бретон по имена, освен ако не играете-

заедно с тези конвенции.

Оставяйки опасност зад себе си, ще разгледаме следващите съоръжения, които Руби предоставя

прехвърляне на един обект към друг.

***7.4***

***Вградени и персонализирани to\_ \* (преобразуване) методи***

Ruby предлага редица вградени методи, чиито имена се състоят от to\_ плюс индикатор

tor на клас, *в* който методът преобразува обект: to\_s (в низ), to\_sym (в sym-

bol), to\_a (за масив), to\_i (за цяло число) и to\_f (за плаване). Не всички обекти реагират

всички тези методи. Но много обекти реагират на много от тях и принципът е такъв

достатъчно последователни, за да се наложи да ги гледаме колективно.

***7.4.1***

***Преобразуване на низове: to\_s***

Най-често използваният метод to\_ вероятно е to\_s. Всеки Ruby обект - освен

екземпляри на BasicObject — отговаря на to\_s и по този начин има начин да се покаже като

струна. Какво прави to\_s, както показват следните извадки от irb, варира от нищо

повече от връща собствения си приемник, когато обектът вече е низ

>> „Вече съм низ!“. To\_s

=>"Вече съм низ!"

за връщане на низ, съдържащ кодово представяне на обект

>> [„едно“, „две“, „три“, 4, 5, 6] .to\_s

=>"[\" един \ ", \" два \ ", \" три \ ", 4, 5, 6]"

(където кавичките с обратна наклонена черта означават, че има буквални кавички

вътре в низа) за връщане на информативен, ако е загадъчен, описателен низ за обект:

>> Object.new.to\_s

=>"# <Обект: 0x000001030389b0>"

Основната точка на to\_s е, че се използва по определени методи и в определен синтак-

tic контексти за предоставяне на канонично представяне на низ на обект. Путовете

метод, например, извиква to\_s на своите аргументи. Ако напишете свои to\_s за

клас или да го замени върху обект, вашите to\_s ще изплуват, когато дадете обекта си на

поставя. Можете да видите това ясно, ако е малко безсмислено, като използвате общ обект:

>> obj = Object.new

=> # <Обект: 0x000001011c9ce0>

>> поставя obj

# <Обект: 0x000001011c9ce0>

=> нула

>> def obj.to\_s

>>"Аз съм обект!"

>> край

=>: до\_с

>> поставя obj

Аз съм обект!

=> нула

б

° С

д

д

е

|  |
| --- |
| **Страница 201** |

**201**

***Вградени и персонализирани to\_ \* (преобразуване) методи***

Представянето на низа по подразбиране на обекта е обичайният клас и местоположение на паметта

екран сметище B . Когато се обадите, поставяте обекта, това виждате c . Но ако вие

дефинирайте персонализиран метод to\_s за обекта d , последващите извиквания към put отразяват

нова дефиниция f . (Обърнете внимание, че самата дефиниция на метода изчислява на символ,

: to\_s, представляващо името на метода д .)

Също така получавате изхода на to\_s, когато използвате обект в интерполация на низове:

>>"Моят обект казва: # {obj}"

=>"Моят обект казва: Аз съм обект!"

Не забравяйте също, че можете да извикате изрично to\_s. Не е нужно да чакате Руби

иди го търси. Но голям процент обаждания до tos са автоматични, зад

сценични обаждания от името на путове или механизма за интерполация.

ЗАБЕЛЕЖКА Когато става въпрос за генериране на низови представяния на техните екземпляри,

масивите правят нещата малко по-различно от нормата. Ако се обадите, поставяте масив,

получавате циклично представяне въз основа на извикване на to\_s на всеки от елементите

в масива и извежда по един на ред. Това е специално поведение; то

не съответства на това, което получавате, когато извикате to\_s в масив - а именно,

низово представяне на масива в квадратни скоби.

Докато разглеждаме низови представяния на обекти, нека разгледаме няколко свързани

методи. Може би се отдалечаваме малко от категорията to\_ \*, но това са всички

методи, които генерират низове от обекти и тяхното разглеждане е там-

предварително навреме.

В Орн ДА бъде преодоляно : INSPECT

Всеки Ruby обект - за пореден път, с изключение на копията на BasicObject - има

метод за проверка. По подразбиране - освен ако даден клас отменя inspect - проверката

string е мини-изваждане на екрана на местоположението на паметта на обекта:

>> Object.new.inspect

=>"# <Обект: 0x007fe24a292b68>"

Всъщност irb използва inspect за всяка стойност, която отпечатва, така че можете да видите проверката

низове от различни обекти, без дори изрично да извиквате inspect:

>> Object.new

=> # <Обект: 0x007f91c2a8d1e8>

>>"abc"

=>"abc"

>> [1,2,3]

=> [1, 2, 3]

>> / регулярен израз /

=> / регулярен израз /

Ако искате полезен низ за проверка за вашите класове, трябва да дефинирате изрично:

клас Личност

def инициализиране (име)

@ име = име

край

|  |
| --- |
| **Страница 202** |

**202**

C ГЛАВА 7 ***Вградени основни неща***

def инспектира

@ име

край

край

david = Person.new ("Дейвид")

поставя david.inspect

# Резултат: Дейвид

(Обърнете внимание, че заменянето на\_s и заменянето на проверката са две различни неща. Преди

Ruby 2, инспектирайте piggybacked на to\_s, за да можете да замените и двете, като замените един.

Това вече не е така.)

Друг, по-рядко използван метод генерира и показва низ, представящ

ция на обект: дисплей.

U Синг DISPLAY

Няма да видите много дисплей. Това се случва само веднъж, при последно броене, във всички Ruby про-

грам файлове в цялата стандартна библиотека. (проверката се извършва 160 пъти.) Това е специализирано

изходен метод.

display приема аргумент: изходящ поток за запис, под формата на Ruby I / O

обект. По подразбиране използва STDOUT, стандартният изходен поток:

>>"Здравей" .дисплей

Здравейте => нула

Обърнете внимание, че дисплеят, за разлика от путове, но като печат, не вмъква автоматично нов ред

характер. Ето защо => nil се изпълнява заедно на един ред с изхода.

Можете да пренасочите изхода на дисплея, като предоставите например отворен файл

дръжка като аргумент:

>> fh = File.open ("/ tmp / display.out", "w")

=> # <Файл: /tmp/display.out>

>>"Hello" .display (fh)

=> нула

>> fh.close

=> нула

>> поставя (File.read ("/ tmp / display.out"))

Здравейте

Низът "Hello" се "показва" директно във файла B , както потвърждаваме, като прочетем

съдържанието на файла и ги отпечатва c .

Нека оставим територията на низ в този момент и да разгледаме как играят техниките за преобразуване

в случая с класа Array.

***7.4.2***

***Преобразуване на масив с to\_a и оператора \****

Методът to\_a (към масив), ако е дефиниран, осигурява подобно на масив представяне на

обекти. Една от най-поразителните характеристики на to\_a е, че той автоматично се свързва с \*

оператор. Операторът \* (произнася се „звезда“, „неравномерно“ или, причудливо

наклонен, „splat“) прави своеобразно разгъване на своя операнд в неговите компоненти, тези

компоненти, които са елементите на неговото представяне на масив.

б

° С

|  |
| --- |
| **203** |

**203**

***Вградени и персонализирани to\_ \* (преобразуване) методи***

Вече сте виждали оператора звезда, използван в списъците с параметри на метода, където той

обозначава параметър, който запълва незадължителните аргументи в масив. В

по-общ случай, звездата превръща всеки масив или всеки обект, който отговаря на to\_a, в

еквивалент на гол списък.

Терминът *гол списък* означава няколко идентификатора или буквални обекти, разделени със запетаи.

Голите списъци са валиден синтаксис само в определени контексти. Например можете да поставите гол списък

вътре в квадратните скоби на конструктора на масив:

[1,2,3,4,5]

Това е фино разграничение, но обозначението, което лежи между скобите, не е масив; това е

list и масивът се изгражда от списъка, благодарение на скобите.

Звездата има своеобразен ефект на премахване на скоби или разсейване. Какво започва като

масив се превръща в списък. Можете да видите това, ако конструирате масив от звезден масив:

>> масив = [1,2,3,4,5]

=> [1, 2, 3, 4, 5]

>> [\* масив]

=> [1, 2, 3, 4, 5]

Масивът в масива е понижен, така да се каже, от масив до гол списък,

Теси на звездата. Сравнете това с това, което се случва, ако не използвате звездата:

>> [масив]

=> [[1, 2, 3, 4, 5]]

Тук списъкът, от който се изгражда новият масив, съдържа един елемент: обектът

масив. Този обект не е добиван заради вътрешните си елементи, както беше в примера

със звездата.

Един извод е, че можете да използвате звездата пред аргумент на метод, за да се обърнете

го от масив в списък. Правите това в случаите, когато имате обекти в масив, който

трябва да изпратите до метод, който очаква разбит списък с аргументи:

дефинирайте съчетани имена (първо име, фамилия)

first\_name + "" + last\_name

край

names = ["David", "Black"]

поставя комбинация\_имена (\* имена)

Ако не използвате звездата без матрица, ще изпратите само един аргумент - масив - на

метод и методът няма да бъде щастлив.

Нека се обърнем към числата.

***7.4.3***

***Числово преобразуване с to\_i и to\_f***

За разлика от някои езици за програмиране, като Perl, Ruby не поддържа автоматично

vert от низове към числа или числа към низове. Не можете да направите това

>> 1 + "2"

**Изход:**

**Дейвид Блек**

**TypeError: String не може**

**да бъде принуден в Fixnum**

|  |
| --- |
| **204** |

**204**

C ГЛАВА 7 ***Вградени основни неща***

защото Руби не знае как да добавя низ и цяло число заедно. И вие ще го направите

вземете изненада, ако направите това:

print "Въведете номер:"

n = gets.chomp

поставя n \* 100

Ще видите низовата версия на номера, отпечатана 100 пъти. (Този резултат също казва

вие, че Ruby ви позволява да умножавате низ, но винаги се третира като низ, дори ако е така

се състои от цифри.) Ако искате числото, трябва изрично да го превърнете в число:

n = gets.to\_i

Както ще видите, ако експериментирате с преобразуване на низове в цели числа (което можете да направите

лесно в irb с изрази като "здравей" .to\_i), стойността на преобразуване to\_i на низовете

които нямат разумен цяло число еквивалент (включително "Hello") е винаги 0. Ако вашият

низ започва с цифри, но не е съставен изцяло от цифри ("123hello"), без цифра

частите се игнорират и преобразуването се извършва само на водещите цифри.

Преобразуването to\_f (да плава) ви дава, предсказуемо, еквивалент на плаваща запетая на

всяко цяло число. Правилата, отнасящи се до несъответстващи знаци, са подобни на правилата

erning преобразуване на низ към цяло число: "hello" .to\_f е 0.0, докато "1.23hello" .to\_f е

1.23. Ако извикате to\_f на плувка, получавате същия плувка обратно. По същия начин извикването на to\_i в

integer връща това цяло число.

Ако правилата за преобразуване на низове ви се струват малко отпуснати - ако не искате низове

като "-5xyz", за да успеят да се преобразуват в цели числа или с плувка - имате a

няколко по-строги техники за преобразуване, достъпни за вас.

S ПО-ТРИКТОРНИ КОНВЕРСИИ С I NTEGER И F LAT

Ruby предоставя методи, наречени Integer и Float (и да, те изглеждат като константи,

но това са методи с имена, които съвпадат с тези на класовете, към които те

конвертирате). Тези методи са подобни на to\_i и to\_f, съответно, но малко по-строги:

ако ги храните с нещо, което не отговаря на целевия тип на преобразуване, те

повдигнете изключение:

>>"123abc" .to\_i

=> 123

>> Цяло число ("123abc")

ArgumentError: невалидна стойност за Integer (): "123abc"

>> Float ("3")

=> 3.0

>> Float ("- 3")

=> -3,0

>> Float ("- 3xyz")

ArgumentError: невалидна стойност за Float (): "-3xyz"

(Обърнете внимание, че преобразуването от цяло число в плаващо е допустимо. Това са буквите, които причиняват

проблемът.)

Ако искате да бъдете строги относно това, което се преобразува и какво се отхвърля, Integer

и Float може да ви помогне.

|  |
| --- |
| **Страница 205** |

**205**

***Вградени и персонализирани to\_ \* (преобразуване) методи***

Връщане към семейството на преобразуватели to\_ \*: в допълнение към директното

методи за преобразуване на обекти, Ruby ви дава няколко to\_ \* метода, които имат малко

допълнителна интелигентност за това какво се очаква да направи тяхната стойност.

***7.4.4***

***Ролеви методи на\_ \****

Донякъде е против зърното в програмирането на Ruby да се тревожите много за какво

клас, към който принадлежи обектът. Важно е само какво може да направи обектът - какви методи

може да изпълни.

Но в няколко случая, включващи основните класове, се обръща стриктно внимание на класа на

обекти. Не мислете за това като за план за „рубиновия начин“ на мислене за обектите.

Това е по-скоро като целесъобразност, която ви зарежда в света на основните обекти в

такъв начин, че след като започнете, можете да отделите по-малко мисли на класа на вашите обекти

членства.

S TRING ROLE - ИГРА СЪ С \_ STR

Ако искате да отпечатате обект, можете да дефинирате метод to\_s за него или да използвате каквото и да било

to\_s поведение, с което е надарен от своя клас. Но какво, ако имате нужда от обект, който да *бъде*

струна?

Отговорът е, че вие ​​дефинирате метод to\_str за обекта. Обектът е to\_str

представяне влиза в картината, когато извикате основен метод, който изисква това

аргументът е низ.

Класическият пример е добавяне на низ. Ruby ви позволява да добавяте два струни заедно, про-

извличане на трети низ:

>>"Здравей" + "там."

=>"Здравейте."

Ако се опитате да добавите nonstring към низ, ще получите грешка:

>>"Здравей" + 10

TypeError: няма неявно преобразуване на Float в String

Преобразуване спрямо типово излъчване

Когато извиквате методи като to\_s, to\_i и to\_f, резултатът е нов обект (или

приемник, ако го конвертирате в собствен клас). Това не е съвсем същото като typecast-

инг на C и други езици. Не използвате обекта като низ или цяло число;

искате обекта да предостави втори обект, който съответства на идеята му за

себе си (така да се каже) в една от тези форми.

Разграничението между преобразуване и типово излъчване засяга някои важни

аспекти на сърцето на Руби. В известен смисъл всички обекти се типографират,

постоянно. Всеки път, когато извикате метод върху обект, вие питате обекта да се държи

като определен тип. Съответно, типът на обекта всъщност е съвкупността от

всичко, което може да направи в определен момент.

Най-близкото до Руби до традиционното излъчване на шрифтове (и то не е много близо) е ролята-

методи за преобразуване на възпроизвеждане, описани в раздел 7.4.4.

|  |
| --- |
| **206 серия** |

**206**

C ГЛАВА 7 ***Вградени основни неща***

Тук влиза to\_str. Ако обект реагира на to\_str, неговият to\_str представя-

tion ще се използва, когато обектът се използва като аргумент на String # +.

Ето пример, включващ прост клас Person. Методът to\_str е wrap-

за около метода на името:

клас Личност

attr\_accessor: име

def to\_str

име

край

край

Ако създадете обект Person и го добавите към низ, to\_str започва с низа на името:

david = Person.new

david.name = "Дейвид"

поставя "Дейвид е наречен" + Дейвид + "."

Преобразуването to\_str се използва и за аргументи към << (добавяне към низ)

метод. И масивите, като низовете, имат метод за ролево преобразуване.

А RRAY РОЛЯ - играе с ДА \_ ARY

Обектите могат да се маскират като масиви, ако имат метод to\_ary. Ако такъв метод е

понастоящем се извиква за обекта в случаите, когато масив и само масив ще направи -

например в операция за конкатенация на масив.

Ето още едно изпълнение на Person, където ролята на масива се играе от масив

съдържащ три атрибута на човек:

клас Личност

attr\_accessor: име,: възраст,: имейл

def to\_ary

[име, възраст, имейл]

край

край

Обединяването на обект Person в масив води до добавяне на име, възраст и

изпращайте стойности по имейл към целевия масив:

david = Person.new

david.name = "Дейвид"

david.age = 55

david.email = "Дейвид @ където и да е"

масив = []

array.concat (Дейвид)

p масив

Подобно на to\_str, to\_ary предоставя начин за обект да влезе в ролята на обект на

определен основен клас. Както обикновено в Ruby, разумното използване на конвенциите е оставено до

Вие. Възможно е да се напише метод to\_ary, например, който прави нещо друго

отколкото да върнете масив - но почти сигурно ще получите съобщение за грешка, когато се опитате да го направите

използвайте го, тъй като Ruby изглежда to\_ary за масив. Така че, ако ще използвате ролевата игра

to\_ \* методи, не забравяйте да играете в топката на Руби.

**Изход: Дейвид е**

**на име Дейвид.**

**Изход: ["Дейвид", 55,**

**"Дейвид @ където и да е"]**

|  |
| --- |
| **207** |

**207**

***Булеви състояния, булеви обекти и нула***

Сега ще се обърнем към темата за булевите състояния и обекти в Ruby, тема, която имаме

вече е потопен, но такъв, който заслужава по-задълбочено проучване.

***7.5***

***Булеви състояния, булеви обекти и нула***

Всеки израз в Ruby оценява на обект и всеки обект има булева стойност

или *вярно,* или *невярно* . Освен това, true и false са обекти. Тази идея не е толкова

колкото и да звучи. Ако true и false не са обекти, тогава чист булев израз

сион като

100> 80

няма да има обект, за да се оцени *до* . (И> е метод и следователно трябва да се върне

предмет.)

В много случаи, когато искате да получите стойност на истина / лъжа, като например, ако

или сравнение между две числа, не е нужно да манипулирате тези данни

граждански обекти директно. В такива ситуации можете да мислите за истината и лъжата като *състояния* ,

а не предмети.

Ще разглеждаме истинното и фалшивото както като състояния, така и като специални обекти, заедно с

специален обект нула.

***7.5.1***

***Вярно и невярно като състояния***

Всеки израз в Ruby е или вярно, или невярно, в логически или булев смисъл. Най-доброто

начинът да се справите с това е да мислите по отношение на условни изявления. За всеки

израз *e* в Ruby, можете да направите това

ако д

и Руби може да го осмисли.

За много изрази условният тест е разтягане; но може да е поучително да опитате

това е на различни изрази, както показва следващият списък.

if (клас MyClass; край)

поставя "Определението на празен клас е вярно!"

друго

поставя "Определението на празен клас е невярно!"

край

if (клас MyClass; 1; край)

поставя "Определението на класа с числото 1 е вярно!"

друго

поставя "Определението на класа с числото 1 е невярно!"

край

if (def m; return false; end)

поставя "Определението на метода е вярно!"

друго

поставя "Определението на метода е невярно!"

край

Листинг 7.1 Тестване на булева стойност на изрази, използвайки **if** конструкции

б

° С

д

|  |
| --- |
| **208** |

**208**

C ГЛАВА 7 ***Вградени основни неща***

ако "низ"

поставя "Струните изглеждат верни!"

друго

поставя "Изглежда, че струните са неверни!"

край

ако 100> 50

поставя "100 е по-голямо от 50!"

друго

поставя "100 не е по-голямо от 50!"

край

Ето изхода от този списък (минус предупреждение за използване на низов литерал в

условно):

Определението на празен клас е невярно!

Определението на класа с номер 1 в него е вярно!

Определението на метода е вярно!

Изглежда струните са верни!

100 е по-голямо от 50!

Както можете да видите, празните дефиниции на клас B са неверни; непразни дефиниции на клас

изядени до същата стойност като последната стойност, която съдържат c (в този пример числото 1);

определения метод са истински г (дори и на повикване *към* метода ще се върне невярно);

низовете са верни e (не се притеснявайте за литерала на низа в предупреждение за състояние); и 100

е по-голямо от 50 f . Можете да използвате тази проста техника if, за да изследвате Boolean

стойност на всеки Ruby израз.

Примерите if показват, че всеки израз в Ruby е или true, или false в

усещане или за преминаване, или за неуспешен тест if. Но тези примери не показват какво

изразите оценяват на. Това е, което тестът if тества: той оценява израз

(като клас MyClass; край) и продължава въз основа на това дали стойността про-

изведени от тази оценка е вярно.

За да видим какви стойности се връщат от изразите, чиято истинска стойност сме били

тестване, можете да извлечете тези стойности в irb:

>> клас MyClass; край

=> нула

>> клас MyClass; 1; край

=> 1

>> def m; върнете false; край

=>: m

>>"низ литерал!"

=>"низ литерал!"

>> 100> 50

=> вярно

Дефиницията на празния клас B се оценява на нула, което е специален обект (обсъден в

раздел 7.5.3). Всичко, което трябва да знаете за момента за nil, е, че той има Boolean

стойност на false (както можете да откриете от поведението на клаузите if, които са се занимавали с него

в списък 7.1).

д

е

б

° С

д

д

е

|  |
| --- |
| **209 серия** |

**209**

***Булеви състояния, булеви обекти и нула***

Определението на класа с числото 1 в него c оценява на числото 1, тъй като

всеки блок с дефиниция на клас оценява до последния израз, съдържащ се в него, или нула

ако блокът е празен.

Дефиницията на метода изчислява символа: m d , представляващ името на

методът, който току-що беше дефиниран.

Низовият литерал e оценява на себе си; това е буквален обект и не трябва да бъде

изчислени или обработени в друга форма, когато се оценяват. Стойността му като израз

sion е себе си.

И накрая, изразът за сравнение 100> 50 f оценява на true - не само на някои -

нещо, което има булева стойност true, но за обекта true. Обектът true прави

имат булева стойност true. Но заедно с false, той има специална роля в

царството на истината и лъжата и как те са представени в Руби.

***7.5.2***

***вярно и невярно като обекти***

Булевите обекти true и false са специални обекти, всеки от които е единственият екземпляр

на клас, специално създаден за него: TrueClass и FalseClass, съответно. Можеш

помолете true и false да ви кажат имената на техните класове и те ще:

поставя true.class

поставя false.class

Термините true и false са ключови думи. Не можете да ги използвате като променлива или метод

имена; те са запазени за изключителната употреба на Ruby.

Можете да предавате обектите true и false, да ги присвоявате на променливи и

разгледайте ги като всеки друг предмет. Ето една сесия на irb, която поставя истината през своите

крачки в качеството си на Ruby обект:

>> a = вярно

=> вярно

>> a = 1 освен ако a

=> нула

>> а

=> вярно

>> b = a

=> вярно

Понякога ще виждате true и false, използвани като аргументи на метода. Например, ако вие

искате клас да ви покаже всички свои методи на екземпляр, но да изключи дефинираните в

родови класове, можете да предоставите аргумента false на вашата заявка:

>> String.instance\_methods (false)

Проблемът с булевите аргументи е, че е много трудно да се запомни какво правят.

Те са доста загадъчни. Затова е най-добре да ги избягвате в собствения си код, освен ако

има случай, в който истинското / фалшивото разграничение е много ясно.

Нека обобщим истинската / фалшивата ситуация в Ruby с поглед върху булевите състояния вер-

sus булеви стойности.

**Изход: TrueClass**

**Изход: FalseClass**

|  |
| --- |
| **210** |

**210**

C ГЛАВА 7 ***Вградени основни неща***

Т RUE / FALSE : S ЧЛЕНКИ VS . СТОЙНОСТИ

Както вече знаете, всеки Ruby израз е вярно или невярно в булев смисъл (като

поставени от теста if), а има и обекти, наречени true и false. Този двойник

използването на истинската / фалшивата терминология понякога е източник на объркване: когато казвате

че нещо е вярно, не винаги е ясно дали имате предвид, че има булева истина

стойност или че това е обектът true.

Не забравяйте, че всеки израз има булева стойност - включително израза

true и изразът false. Може да изглежда неудобно да се каже: „Обектът е истина

истина е." Но тази допълнителна стъпка дава възможност моделът да работи последователно.

Надграждайки върху тази точка и върху някои от случаите, които видяхте в малко по-различна форма

в таблица 7.1, таблица 7.3 показва картографиране на някои примерни изрази към двата изхода

идват от тяхната оценка и тяхната булева стойност.

Обърнете внимание по-специално, че нула и празни низове (както и празни масиви и хешове)

имат логическа стойност на true. Единствените обекти, които имат булева стойност false са

фалшив и нулев.

И по въпроса за нула: време е да разгледаме по-внимателно този уникален обект.

Таблица 7.3 Съпоставяне на примерни изрази с резултатите от тяхната оценка и булеви стойности

Израз

Обект, който изразът оценява

Булева стойност на израза

1

1

Вярно

0

0

Вярно

1 + 1

2

Вярно

вярно

вярно

Вярно

невярно

невярно

Невярно

нула

нула

Невярно

"низ"

"низ"

Вярно

""

""

Вярно

поставя "низ"

нула

Невярно

100> 50

вярно

Вярно

x = 10

10

Вярно

def x; край

:х

Вярно

клас С; край

нула

Невярно

клас С; 1; край

1

Вярно

|  |
| --- |
| **Страница 211** |

**211**

***Булеви състояния, булеви обекти и нула***

***7.5.3***

***Специалният обект е нула***

Специалният обект nil всъщност е обект (това е единственият екземпляр на клас, наречен

NilClass). Но на практика това също е вид обект. Булевата стойност на nil е

невярно, но това е само началото на неговата безпредметност.

nil означава липса на каквото и да било. Можете да видите това графично, когато попитате

в стойността например на променлива на екземпляр, която не сте инициализирали:

поставя @x

Тази команда отпечатва нула. (Ако опитате това с локална променлива, ще получите грешка;

локалните променливи не се инициализират автоматично за нищо, дори и за нула.) nil също е

стойността по подразбиране за несъществуващи елементи на обекти на контейнери и колекции. За

Например, ако създадете масив с три елемента и след това се опитате да получите достъп до

десети елемент (при индекс 9, тъй като индексирането на масива започва от 0), ще откриете, че е нула:

>> [„едно“, „две“, „три“] [9]

=> нула

нула понякога е труден за разбиране обект. Всичко е свързано с отсъствие и не

тенция; но nil съществува и той отговаря на извикванията на методи като други обекти:

>> nil.to\_s

=>""

>> nil.to\_i

=> 0

>> nil.object\_id

=> 8

Преобразуването to\_s на nil е празен низ (""); целочисленото представяне на нула

е 0; и идентификаторът на обекта на nil е 8. (nil няма специална връзка с 8; това просто се случва с

е номерът, посочен като негов идентификатор .)

Не е точно да се каже, че нула е празна, защото това би означавало, че е

характеристики и измерение, като число или колекция, което не се предполага

да се. Опитът да хванете нула може да ви отведе в някаква трънлива философска територия. Можеш

мислете за нула като за обект, който съществува и който е снабден с комплект за оцеляване от

методи, но това служи за представяне на отсъствие и състояние на битието

неопределен.

В пълния кръг не забравяйте, че nil има булева стойност false. нулев и фалшив

са единствените два обекта, които го правят. Те не са единствените два *израза,* които го правят; 100 <50

има булева стойност false, тъй като оценява на обекта false. Но нула и

false са единствените два *обекта* в Ruby с булева стойност false. Всички останали Руби

обектите - числа, низове, екземпляри на MyCoolClass - имат логическа стойност true.

Тествани директно, всички те преминават теста if.

Булевите стойности и тестването дават насока към следващата тема: сравнения

между обектите. Ще разгледаме тестове, включващи два обекта и начини за определяне

дали са равни - и ако не са, дали могат да бъдат класирани като по-големи /

по-малко и въз основа на какви критерии.

|  |
| --- |
| **Страница 212** |

**212**

C ГЛАВА 7 ***Вградени основни неща***

***7.6***

***Сравняване на два обекта***

Ruby обектите са създадени с капацитет да се сравняват с други обекти за

равенство и / или ред, като се използва някой от няколкото метода. Тестовете за равенство са най-много

общи сравнителни тестове и ще започнем с тях. След това ще разгледаме вградения Ruby

модул, наречен сравним, който ви дава бърз начин за предаване на знания за ком-

parison операции за вашите класове и обекти и това се използва за тази цел от a

брой вградени класове Ruby.

***7.6.1***

***Тестове за равенство***

В клас Object всички методи за тестване на равенство правят едно и също: те ви казват

дали два обекта са абсолютно един и същ обект. Ето ги в действие:

>> a = Object.new

=> # <Обект: 0x00000101258af8>

>> b = Object.new

=> # <Обект: 0x00000101251d70>

>> a == a

=> вярно

>> a == b

=> невярно

>> a! = b

=> вярно

>> a.eql? (a)

=> вярно

>> a.eql? (b)

=> невярно

>> a.equal? ​​(a)

=> вярно

>> a.equal? ​​(b)

=> невярно

И трите метода за тест за положително равенство (==, eql? И равни?) Дават едни и същи

резултати в тези примери: когато тествате a срещу a, резултатът е верен и когато вие

тествайте а срещу b, резултатът е невярен. (Методът за неравен или отрицателен тест за равенство! = Е

обратното на метода ==; всъщност, ако дефинирате ==, вашите обекти автоматично ще се

имаме метода! =.) Имаме много начини да установим, че a е a, но не b.

Но няма много смисъл да се правят три теста, които правят едно и също нещо. Освен това

надолу по пътя, в класове, различни от Object, == и / или eql? обикновено са предефинирани към

вършете смислена работа за различни обекти. Равният? методът обикновено се оставя сам, така че

че винаги можете да го използвате, за да проверите дали два обекта са абсолютно един и същ обект.

Ето пример, включващ низове. Имайте предвид, че те са == и eql ?, но не са равни ?:

>> string1 = "текст"

=>"текст"

>> string2 = "текст"

=>"текст"

>> string1 == string2

=> вярно

>> string1.eql? (string2)

|  |
| --- |
| **213. стр** |

**213**

***Сравняване на два обекта***

=> вярно

>> string1.equal? ​​(string2)

=> невярно

Освен това, Ruby ви предоставя набор от инструменти за сравняване на обекти, и то не винаги

просто сравнение за равенство. След това ще разгледаме как тестовете за равенство и тяхното предефиниране

тези се вписват в общата картина за сравнение.

***7.6.2***

***Сравнения и Сравним модул***

Най-често предефинираният метод за тест за равенство и този, който ще видите, се използва най-често

често е ==. Това е част от по-голямото семейство методи за тест за равенство и също е част от a

семейство методи за сравнение, което включва ==,! =,>, <,> = и <=.

Не всеки клас обект се нуждае или трябва да има всички тези методи. (Трудно е

представете си какво би означавало за един велосипед да бъде по-голям или равен на друг.

Gears?) Но за класове, които се нуждаят от пълна функционалност за сравнение, Ruby предоставя a

удобен начин да го получите. Ако искате обектите от клас MyClass да имат пълния набор от

методи за сравнение, всичко, което трябва да направите, е следното:

1

Смесете модул, наречен Comparable (който идва с Ruby) в MyClass.

2

Дефинирайте метод за сравнение с името <=> като метод на екземпляр в

Моят клас.

Метод за сравнение <=> (обикновено се нарича *оператор* на *космически кораб* или *метод на космически кораб* )

е сърцевината на въпроса. Вътре в този метод вие определяте какво имате предвид под *по-малко от* ,

*равно на* и *по-голямо от* . След като направите това, Ruby има всичко необходимо, за да предостави

съответните методи за сравнение.

Да приемем например, че приемате оферти за работа и използвате Ruby скрипт в помощ

проследявате какви оферти са постъпвали. Решавате, че би било удобно да можете

сравнете всеки два обекта за наддаване въз основа на атрибут за оценка, използвайки прости сравнения

син оператори като> и <. *По-голямо от* означава да се искат повече пари и *по-малко от*

означава да поискате по-малко пари.

Една проста версия на класа Bid може да изглежда като следния списък.

клас Оферта

включват Сравним

attr\_accessor: оценка

def <=> (други\_бъд)

ако само.оценка <друго\_об.оценка

-1

elsif self.estimate> other\_bid.estimate

1

друго

0

край

край

край

Листинг 7.2 Пример за клас, който се смесва в модула за **сравнение**

б

|  |
| --- |
| **214** |

**214**

C ГЛАВА 7 ***Вградени основни неща***

Методът на космическия кораб B се състои от каскадно изражение if / elsif / else. Зависи-

на кой клон се изпълнява, методът връща отрицателно число (по конвенция

ция, –1), положително число (по конвенция, 1) или 0. Тези три върнати стойности са

предварително дефинирани, предварително подредени сигнали към Руби. Вашият <=> метод трябва да върне един от тях

три стойности всеки път, когато се извика - и те винаги означават по-малко от, равно на и

по-голяма от, съответно.

Можете да съкратите този метод. Оценките на офертите са или числа с плаваща запетая, или

цели числа (последните, ако не се занимавате с центовете на частите от фигурата или ако съхранявате

сумите като центове, а не като долари). Числата вече знаят как да ги сравняват

самите помежду си, включително цели числа до плувки. Следователно методът на наддаване <=> може

piggyback за съществуващите <=> методи на класовете Integer и Float, като този:

def <=> (други\_бъд)

самооценяване <=> друго\_об.оценка

край

В тази версия на метода на космическия кораб ние пунт; казваме, че ако искате

да знаете как две оферти се сравняват една с друга, да повдигнете въпроса до приблизителните стойности

за двете оферти и използвайте това сравнение като основа за сравнение между офертата и офертата.

Изплащането за определяне на оператора на космически кораб и включване на „Сравнимо“ е

че можете оттогава да използвате целия набор от методи за сравнение по двойки

вашите обекти. В този пример bid1 печели договора; това е по-малко от (както е определено

от <) bid2:

>> bid1 = Bid.new

=> # <Оферта: 0x000001011d5d60>

>> bid2 = Bid.new

=> # <Оферта: 0x000001011d4320>

>> bid1.estimate = 100

=> 100

>> bid2.estimate = 105

=> 105

>> оферта1 <оферта2

=> вярно

Методът <(заедно с>,> =, <=, ==,! = И между?) Е дефиниран по отношение на <=>,

вътре в модула за сравнение. (б. между? (а, в) ви казва дали *b*>*a* и *b*<*c* .)

Всички числови класове на Ruby включват сравними и имат дефиниция за <=>. The

същото важи и за класа String; можете да сравните низовете, като използвате пълния асортимент от

Сравним метод / оператори. Сравним е удобен инструмент, който ви дава много функции-

националност в замяна на по същество една дефиниция на метода.

Сега ще се обърнем към темата за проверка на обект по време на изпълнение. В съответствие с

дух на тази глава, ще разгледаме достатъчно техники, за да ви поддържаме през повечето от

останалата част от книгата. Имайте предвид обаче, че главата 15, последната в книгата, ще го направи

върнете се към темата за проверка по време на работа (наред с други). Така че можете да приемете това като

първият, но не и последният съществен поглед върху темата.

|  |
| --- |
| **215 стр** |

**215**

***Проверка на възможностите на обекта***

***7.7***

***Проверка на възможностите на обекта***

*Инспекцията* и *размисълът се* отнасят общо за различните начини, по които можете да получите Руби

предмети, за да ви разкажат за себе си през живота си. Голяма част от наученото

по-рано за получаване на обекти, които да показват низови представяния на себе си, може да бъде

описано като проверка. В този раздел ще разгледаме различен вид отражение по време на изпълнение

ция: техники за питане на обекти за методите, които те могат да изпълнят.

Как ще направите това зависи от обекта и от точно това, което търсите.

Всеки обект може да ви каже какви методи можете да извикате, поне към момента

вие го питате. Освен това обектите от клас и модул могат да ви дадат разбивка на

методи, които те предоставят за обектите, които използват тези методи (като екземпляри или

чрез включване на модул).

***7.7.1***

***Изброяване на методи на обект***

Най-простият и често срещан случай е, когато искате да знаете какви съобщения са

обектът разбира - тоест какви методи можете да извикате при него. Руби ви дава типично

лесен начин да направите това. Въведете това в irb:

>>"Аз съм обект String" .methods

Ще видите голям набор от имена на методи. Най-малкото ще искате да ги сортирате така

можете да намерите това, което търсите:

>>"Аз съм обект String" .methods.sort

Методът с методи работи и с обекти на клас и модул. Но не забравяйте, че

ви показва на какво отговаря обектът (класът или модулът), а не на какви екземпляри на

клас или обекти, които използват модула, отговарят. Например да поискате irb за

>> String.methods.sort

показва списък с методи, на които обектът Class отговаря на String. Ако видите елемент в

този списък, знаете, че можете да го изпратите директно на String.

Методите, които виждате, когато извиквате методи на обект, включват неговия сингълтон

методи - тези, които сте написали само за този обект - както и всички методи, които може

извикване по силата на включването на един или повече модули навсякъде в неговия произход. всичко

тези методи са представени като равни: изброяването на методи изравнява метода

пътека за търсене и отчита само за какви методи знае обектът, независимо от

където са дефинирани.

Можете да проверите това в irb. Ето пример, в който се добавя единичен метод

струна. Ако включите извикването към str.methods.sort в края, ще видите този вик

сега е сред методите на низа:

>> str = "Обикновен стар низ"

=>"Обикновен стар низ"

>> def str.shout

>> self.upcase + "!!!"

>> край

|  |
| --- |
| **216 серия** |

**216**

C ГЛАВА 7 ***Вградени основни неща***

=> нула

>> str.shout

=>"РАВНА СТРАНА !!!"

>> str.methods.sort

Удобно е, че можете да поискате само единични методи на обект:

>> str.singleton\_methods

=> [: вик]

По същия начин, ако смесите модул в клас с include , екземплярите на този клас ще го направят

се отчитат като способни да извикват методите на екземпляра от този модул. Между-

очаквано ще получите същия резултат, дори ако включите модула след екземпляра

вече съществува. Започнете нова irb сесия (за да изчистите паметта от предишния пример),

и опитайте този код. Вместо да разпечатваме всички методи, ще използваме няколко по-малко

разхвърляни техники, за да разберете дали str има метода на вика:

>> str = "Още един обикновен стар низ."

=>"Още един обикновен стар низ."

>> модул StringExtras

>> def вика

>> self.upcase + "!!!"

>> край

>> край

=>: вик

>> клас String

>> включва StringExtras

>> край

=> Низ

>> str.methods.include? (: shout)

=> вярно

Включването на модула засяга низове, които вече съществуват, защото когато поискате низ

за да извика, той търси пътя си за търсене на метод за метод на вика и го намира в

модул. Низът наистина не се интересува кога или как модулът е вмъкнат

пътеката за търсене.

Всеки обект може да ви каже какви методи знае. Освен това клас и модул

обектите могат да ви дадат информация за методите, които предоставят.

***7.7.2***

***Заявка за обекти на клас и модул***

Един от методите, които ще намерите в списъка, генериран от командата irb String.

methods.sort е instance\_methods. Той ви казва всички методи на екземпляра, които

екземплярите на String са надарени с:

>> String.instance\_methods.sort

Полученият списък е същият като списъка с методи, както е показано от методи, за всеки даден

низ (освен ако не сте добавили единични методи към този низ).

Можете да направите подобна заявка за модул:

>> Enumerable.instance\_methods.sort

**Прави низове**

**на викачи**

|  |
| --- |
| **Страница 217** |

**217**

***Обобщение***

В допълнение към ясните списъци с методи и екземпляри, Ruby предоставя и

определен брой ощипвания, за да ви помогнат да направите по-фини заявки.

***7.7.3***

***Филтрирани и избрани списъци с методи***

Понякога ще искате да видите методите на екземпляра, дефинирани в определен клас с-

навън, занимавайки се с методите, които всеки обект има. В крайна сметка вече знаете, че вашият

обектът има *тези* методи. Можете да прегледате методите на екземпляра на класа без тези на

предците на класа с помощта на леко тайнствената техника, въведена по-рано, на

въвеждане на аргумента false към метода instance\_methods:

String.instance\_methods (false) .sort

По този начин ще видите много по-малко методи, защото разглеждате списък само с тези

дефинирани в класа String, без тези, дефинирани в някой от родовите класове на String

или модули. Този подход ви дава ограничена картина на наличните методи

низови обекти, но е полезно за разглеждане по по-фин начин как и къде

дефинициите на метода зад даден обект са позиционирани.

Други методи за изброяване на методи включват следното (от което сте виждали

singleton\_methods вече):

obj.private\_methods

obj.public\_methods

obj.protected\_methods

obj.singleton\_methods

Освен това класовете и модулите ви позволяват да разгледате техните методи на екземпляр:

MyClass.private\_instance\_methods

MyClass.protected\_instance\_methods

MyClass.public\_instance\_methods

Последният от тях, public\_instance\_methods, е синоним на instance\_methods.

Механизмите за изследване на методите на обектите са обширни. Както винаги, бъдете ясни

в собствения си ум какъв е обектът (по-специално клас / модул или „обикновен“ обект)

че заявявате и какво искате да ви каже.

Стигнахме до края на нашата сесия за зареждане на midbook, комплект за оцеляване, грамотност

ръководство. ... Както и да го наречете (дори „глава 7“!), това ни поставя в добра позиция

разгледайте внимателно редица важни основни класове, които ще направим през следващите се

ерални глави.

***7.8***

***Обобщение***

В тази глава сте виждали

Буквалните конструктори на Ruby

Синтактични методи за преобразуване на захар в оператори

„Деструктивни“ методи и взривни методи

|  |
| --- |
| **Страница 218** |

**218**

C ГЛАВА 7 ***Вградени основни неща***

Методи за преобразуване (to\_s и приятели)

Методите за проверка и показване

Булеви стойности и булеви обекти

Специалният обект е нула

Сравняване на обекти и сравним модул

Изследване на методите на обекта

Тази глава обхваща няколко теми, които се отнасят до множество вградени класове и

ules. Виждали сте литералните конструктори на Ruby, които дават кратка алтернатива на

извикване на ново за определени вградени класове. Виждали сте също как Ruby осигурява синтактика

захар за конкретни имена на методи, включително голям брой методи с имена

които съответстват на аритметични оператори.

Разгледахме значението на методите, които променят собствените си приемници, които

правят много вградени методи (много от тях разбиват методи, които завършват с!). Ние също

разгледа методите to\_ \*: вградени методи за извършване на преобразувания от един

основен клас към друг.

Също така сте научили редица важни моменти и техники, свързани с

Булеви (истински / неверни) стойности и сравнение между обекти. Виждали сте това всеки

обект в Ruby има булева стойност и че Ruby също има специални булеви обекти

(true и false), които представляват тези стойности в най-простата им форма. Трета специална

обект, нула, представлява състояние на неопределеност или отсъствие. Ние също обсъдихме

niques за сравняване на обекти с помощта на стандартния оператор за сравнение (<=>) и

Сравним модул.

Накрая разгледахме начини да накараме обектите на Ruby да ви кажат какви методи използват

реагирайте на - вид техника за размисъл, която може да ви помогне да видите и разберете

какво се случва в даден момент от вашата програма. Ще разгледаме по-задълбочено интроспек-

ция и размисъл в глава 15.

Материалът в тази глава ще ви постави в силна позиция да усвоите това, което сте

срещат по-късно, в останалата част на тази книга и след това. Когато четете твърдения като

„Този ​​метод има алтернатива за взрив“, ще разберете какво означават те. Когато видите doc-

Umentation, която ви казва, че даден аргумент на метода по подразбиране е нулев, ще знаете

какво значи това. И фактът, че сте научили за тези повтарящи се теми, ще го направи

помогнете ни да спестим от повторение в предстоящите глави за вградените Ruby класове

и модули и вместо това се концентрирайте върху движението напред.

|  |
| --- |
| **219 серия** |

**219**

*Струни, символи и*

*други скаларни обекти*

Терминът *скалар* означава *едноизмерен* . Тук се отнася до обекти, които представляват греховно-

gle стойности, за разлика от колекцията или контейнерните обекти, които съдържат множество стойности.

Има някои нюанси на сивото: струните например могат да се разглеждат като колекции от

символи, освен че са единични единици текст. Скаларът е до известна степен в

око на гледащия. И все пак, като добро първо приближение, можете да разгледате класовете

обсъждани в тази глава като класове на едномерни обекти с размер на хапка, в съзвучие

преминете към колекционните обекти, които ще бъдат фокусът на следващата глава.

Вградените обекти, които ще разгледаме в тази глава, включват следното:

Струни, които са стандартният начин за работа с текстови материали на всеки

дължина

***Тази глава обхваща***

Създаване и манипулиране на низови обекти

Методи за трансформиране на низове

Семантика на символи

Сравнение на низ / символ

Цели числа и плувки

Обекти за час и дата

|  |
| --- |
| **220 серия** |

**220**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Символи, които са (освен всичко друго) и друг начин за представяне на текст

в Руби

Цели числа

Числа с плаваща запетая

Обектите Time, Date и DateTime

Всички тези иначе доста различни обекти са скаларни - те са едномерни,

неконтейнерни обекти без други обекти, които се дебнат вътре в тях, както масивите

имат. Това не означава, че скаларите не са сложни и богати на своята семантика; както и вие

вижте, те са.

***8.1***

***Работа със струни***

Ruby предоставя два вградени класа, които между тях осигуряват цялата функционалност на

текстово представяне и манипулация: класът String и клас Symbol. Струни

и символите са дълбоко различни един от друг, но те са достатъчно сходни по своите

споделен капацитет да представят текст, който заслужава да бъде обсъден в същата глава.

Ще започнем с низове, които са стандартният начин за представяне на тела от текст на arbi-

проследяване на съдържанието и дължината. Вече сте виждали струни в много контексти; ето, ще получим

по-дълбоко в някои от тяхната семантика и способности. Първо ще разгледаме как пишете

низове, след което ще обсъдим редица начини, по които можете да манипулирате низове,

запитвайте ги (например по отношение на дължината им), сравнявайте ги помежду си и транс-

оформете ги (от малки до големи и т.н.). Също така ще разгледаме още

подробности за процеса на преобразуване на низове с to\_i и свързани методи.

***8.1.1***

***Струнна нотация***

А *символни низове,* обикновено, затворени в кавички:

„Това е низ.“

Могат да се използват и единични кавички:

„Това също е низ.“

Но един цитиран низ се държи по различен начин, при някои обстоятелства, отколкото двойно-

цитиран низ. Основната разлика е, че интерполацията на низове не работи

низове с един кавички. Опитайте тези два фрагмента и ще видите разликата:

поставя "Две плюс две е # {2 + 2}."

поставя „Две плюс две е # {2 + 2}.“

Както ще видите, ако поставите тези редове в irb, ще получите два много различни резултата:

Две плюс две е 4.

Две плюс две е # {2 + 2}.

Единичните кавички деактивират механизма за интерполация # {...}. Ако имате нужда от този меха-

nism, не можете да използвате единични кавички. И обратно, ако е необходимо, можете да *избягате* (и

|  |
| --- |
| **221 серия** |

**221**

***Работа със струни***

по този начин деактивира) механизма за интерполация на низове в двоен кавичен низ,

използване на обратни наклонени черти:

поставя "Избягала интерполация: \" \ # {2 + 2} \ "."

Струните с единични и двойни кавички също се държат различно по отношение на необходимостта

избягайте от определени герои. Следните изявления документират и демонстрират

разлики. Погледнете отблизо кои са едноцитатни и кои са двойно цитирани,

и как се използва обратната наклонена черта:

поставя "Наклонените наклонени черти (\\) трябва да бъдат избегнати в двойни кавички."

поставя „Можете просто да въведете \ веднъж в един цитиран низ.“

поставя „Но който и да е вид кавички, които използвате ...“

поставя "... трябва да избягате от котировъчния символ, като \". "

поставя „Това се отнася за \“ и в низове с единични кавички. “

поставя „Backslash-n просто изглежда \ n между отделни кавички.“

поставя "Но това означава нов ред \ nв низ с двойни кавички."

поставя „Същото с \ t, което излиза като \ t с единични кавички ... '

поставя "... но вмъква знак за табулатор: \ tinside двойни кавички."

поставя "Можете да избягате от наклонената черта, за да получите \\ n и \\ t с двойни кавички."

Ето резултата от този поток от котировки. Не се подрежда ред за ред

с кода, но можете да разберете защо, ако погледнете изявлението, което извежда ново-

ред символ:

Наклонените наклонени черти (\) трябва да бъдат избегнати в двойни кавички.

Можете просто да въведете \ веднъж в един цитиран низ.

Но който и вид кавички да използвате ...

... трябва да избягате от котировъчния символ, като ".

Това важи и за „в низове с единични кавички.

Backslash-n просто изглежда \ n между отделни кавички.

Но това означава нов ред

в низ с двоен кавички.

Същото е с \ t, което излиза като \ t с единични кавички ...

... но вмъква знак за табулация: вътре в двойни кавички.

Можете да избягате от наклонената наклонена черта, за да получите \ n и \ t с двойни кавички.

Ще видите други случаи на интерполация на низове и избягване на символи, докато продължаваме.

Междувременно най-добрият начин да усетите това поведение от първа ръка е да опитате

мент с низове в irb.

Ruby ви дава няколко начина за писане на низове в допълнение към единична и двойна квота-

знаци.

O ТЕХНИТЕ МЕХАНИЗМИ ЗА ЦИТИРАНЕ

Алтернативните механизми за цитиране приемат формата% *char* {text}, където *char* е един от

няколко специални знака и къдравите скоби представляват разделител на вашия избор -

инж. Ето пример за един от тези механизми:% q, който произвежда едно-

цитиран низ:

поставя% q {Не е необходимо да избягвате апострофите, когато използвате% q.}

Както посочва примерното изречение, тъй като не използвате символа с една кавичка

ter като знак за кавичка, можете да го използвате без изваждане вътре в низа.

|  |
| --- |
| **222 серия** |

**222**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Също така на ваше разположение са% Q {}, който генерира двоен кавичка низ, и обикновен

% {} (знак за процент и разделител), който също генерира двойно кавиран низ. Natu-

рали, не е нужно да избягвате символа с двойни кавички вътре в низове, които са повторени

възмутен с някоя от тези обозначения.

Разделителят за нотации в стил% може да бъде почти всичко, което искате, стига да

тъй като началният разделител съвпада със затварящия. *Съответствието* в този случай означава и двете

съставяне на лява / дясна двойка скоби (къдрава, извита или квадратна) или да са две от

един и същ герой. По този начин всички от следните са приемливи:

% qА низ-

% Q / Друг низ /

% [Още един низ]

Не можете да използвате буквено-цифрови знаци като разделители, но ако ви се иска

неясен, можете да използвате интервал. Трудно е да се види в пример, така че цялото следващо

пример е заобиколен от квадратни скоби, които не трябва да въвеждате, ако въвеждате

примера в irb сесия или програмен файл Ruby:

[% q Здравейте! ]

Ограниченият от пространството пример, освен че е глупав (макар и поучителен), води до

имайте предвид въпроса какво се случва, ако използвате разделителя вътре в низа (защото

много низове имат интервали вътре в тях). Ако разделителят е единичен знак, имате

за да избягате от него:

[% q Здравейте \ там! ]

% q-По-добре избягайте от \ - вътре в този низ! -

Ако използвате ляво / дясно съвпадащи скоби и Руби вижда лява в

низ, той приема, че скобата е част от низа и търси съответстващ десен

ръка първа. Ако искате да включите несравнима скоба от същия тип като тези

използвате за разделители, трябва да избягате от него:

% Q [Мога да сложа [] тук незасегнат.]

% q (Трябва да избягам \ (ако го използвам сам тук.)

% Q (И същото важи и за \).)

Всеки от механизмите за котиране в стил % *char* генерира или единичен, или двоен

цитиран низ. Това разграничение пронизва струнността; всеки низ е един или друг,

irb не играе добре с част от този синтаксис

irb има собствен парсер Ruby, който трябва да се бори с факта, че докато анализира един

линия, няма начин да знае какъв ще бъде следващият ред. Резултатът е, че irb го прави

нещата са малко по-различни от интерпретатора на Ruby. В случай на котировъчни механизми,

може да откриете, че в irb, избягването на несравним квадрат и други скоби произвежда

странни резултати. Като цяло е по-добре да включите тези примери в командата-

ред формат ruby ​​–e 'поставя% q [Пример: \ []' и подобни.

|  |
| --- |
| **223 серия** |

**223**

***Работа със струни***

без значение коя нотация използвате - включително следващата, която ще разгледаме, „тук“

синтаксис на документ.

ДОКУМЕНТИ „H ERE “

Документът *"тук"* или *тук-документ* е низ, обикновено многоредов низ, който често отнема

форма на шаблон или набор от редове за данни. Казва се, че е „тук“, защото е физически

присъства в програмния файл, не се чете от отделен текстов файл.

Тук документите се появяват чрез оператора <<, както е показано в този откъс от irb:

>> text = << EOM

Това е първият ред текст.

Това е вторият ред.

Сега сме готови.

EOM

=>"Това е първият ред от текста. \ NТова е вторият ред. \ N

➥ Сега приключихме. \ N "

Изразът << EOM означава *текста, който следва, до, но без да включва следващото появяване*

*на „EOM.“* Разделителят може да бъде всеки низ; EOM (край на съобщението) е често срещан избор.

Той трябва да бъде изравнен вляво и трябва да е единственото нещо на линията, където се случва. Вие

може да изключи изискването за изравняване наляво, като постави тире пред оператора <<:

>> text = << - EOM

EOM не трябва да бъде оставен настрана!

EOM

=>"EOM не трябва да бъде оставен настрана! \ N"

МНИ, която спира четенето на този документ тук (този път само документ от един ред)

е в средата на линията.

По подразбиране тук-документите се четат като двойно кавирани низове. По този начин те могат да включват

интерполация на низове и използване на екраниращи символи като \ n и \ t. Ако искате едно-

цитиран тук-doc, поставете затварящия разделител в единични кавички, когато стартирате документа-

мента. За да направите разликата по-ясна, този пример включва путове от тук-документа:

>> text = << - 'EOM'

Едноцитирани!

Обърнете внимание на литерала \ n.

И буквалът # {2 + 2}.

EOM

=>"Единичен кавичка! \ NЗабележете литерала \\ n. \ NИ литерала \ # {2 + 2}. \ N"

>> поставя текст

Едноцитирани!

Обърнете внимание на литерала \ n.

И буквалът # {2 + 2}.

<< EOM (или еквивалент) не трябва да е последното нещо на неговата линия. Където и да е

се случва, той служи като заместител за предстоящия тук-документ. Ето един, който получава

вертиран до цяло число и умножен по 10:

a = << EOM.to\_i \* 10

5

EOM

поставя а

**Изход: 50**

|  |
| --- |
| **224 серия** |

**224**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Можете дори да използвате here-doc в конструктор на литерални обекти. Ето пример, когато a

string се поставя в масив, създавайки низа като тук-документ:

масив = [1,2,3, << EOM, 4]

Това е тук-док!

Става масив [3].

EOM

p масив

Изходът е

[1, 2, 3, „Това е тук-док! \ NТой става масив [3]. \ N“, 4]

И можете да използвате << EOM нотацията като аргумент на метода; аргументът става

тук-doc, който следва реда, на който се появява извикването на метода. Това може да е полезно

ако искате да избегнете натъпкването на твърде много текст във вашия списък с аргументи:

do\_something\_with\_args (a, b, << EOM)

http: // some\_very\_long\_url\_or\_other\_text\_best\_put\_on\_its\_own\_line

EOM

В допълнение към създаването на низове, трябва да знаете какво можете да правите с тях. Вие

може да направи много и ще разгледаме голяма част от него в детайли, като започнем с основите на низа

манипулация.

***8.1.2***

***Основна манипулация на низове***

*Basic* в този контекст означава манипулиране на обекта на най-ниските нива: извличане и

задаване на поднизове и комбиниране на низове помежду си. От гледна точка на Руби,

тези техники не са по-основни от тези, които идват по-късно в нашето проучване на

струни; но концептуално те са по-близо до струнния метал, така да се каже.

G ВЪВЕЖДАНЕ И НАСТРОЙКА НА ПОДСТРОЙКИ

За да извлечете *n* -ия символ в низ, използвате оператора /] [], като му присвоявате

индекс, на база нулев произход, за знака, който искате. Индекс на отрицателни числа

от края на низа:

>> string = "Ruby е страхотен език."

=>"Ruby е страхотен език."

>> низ [5]

=>"i"

>> низ [-12]

=>"o"

Ако предоставите втори целочислен аргумент, *m* , ще получите подниз от *m* знака,

като се започне от посочения от вас индекс:

>> низ [5,10]

=>"е готино"

Можете също да предоставите обект от един *диапазон* като аргумент. Ще разгледаме диапазоните в

по-голяма дълбочина по-късно; засега можете да мислите за *n..m* като за всички стойности между *n* и *m* ,

включително (или без *m* , ако използвате три точки вместо две). Асортиментът може да използва

|  |
| --- |
| **Страница 225** |

**225**

***Работа със струни***

отрицателни числа, които се броят от края на низа назад, но вторият

index винаги трябва да е по-близо до края на низа от първия индекс; индекса

логиката преминава само отляво надясно:

>> низ [7..14]

=>"готино"

>> низ [-12 ..- 3]

=>"стар език"

>> низ [-12..20]

=>"стар език"

>> низ [15 ..- 1]

=>"език."

Можете също така да вземете поднизове въз основа на изрично търсене на поднизове. Ако поднизът е

намерено, връща се; ако не, върнатата стойност е нула:

>> низ ["готин език"]

=>"готин език"

>> низ ["много готин език"]

=> нула

Също така е възможно да се търси съвпадение на шаблон, използвайки техниката [] с обикновен

израз - [] е метод и това, което е вътре в него, са аргументите, така че може да прави каквото-

някога е програмирано да прави:

>> низ [/ c [ol] + /]

=>"готин l"

Ще разгледаме регулярните изрази отделно в глава 11, след което ще получите a

усещане за възможностите на този начин за търсене на поднизове.

Методът [] е достъпен и под името slice. Освен това, приемник-

промяна на версията на среза, а именно slice !, премахва символа (ите) от низа

постоянно:

>> string.slice! ("готино")

=>"готино"

>> низ

=>"Ruby е език."

За да зададете част от низ на нова стойност, използвате метода [] =. Отнема същите видове

на индексиране на аргументи като [], но променя стойностите на това, което сте посочили. Поставянето на

предшестващ малък низ през неговите стъпки, ето някои примери за настройка на низове,

с изследване на променения низ след всеки:

>> string = "Ruby е страхотен език."

=>"Ruby е страхотен език."

>> string ["cool"] = "страхотно"

=>"страхотно"

>> низ

=>"Ruby е страхотен език."

>> низ [-1] = "!"

=>"!"

>> низ

|  |
| --- |
| **226 серия** |

**226**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

=>"Ruby е страхотен език!"

>> string [-9 ..- 1] = "нещо, което да научите!"

=>"нещо за научаване!"

>> низ

=>"Ruby е страхотно нещо за научаване!"

Целите числа, диапазоните, низовете и регулярните изрази могат да работят като индекс или под-

низ спецификатори. Ако се опитате да зададете част от низа, която не съществува - т.е. твърде висока

или твърде нисък цифров индекс, или низ или регулярен израз, който не съответства на

низ - получавате фатална грешка.

В допълнение към промяната на отделни низове, можете също да комбинирате низове с

взаимно.

C ОМБИНИРАЩИ СТРОНИ

Има няколко техники за комбиниране на струни. Тези техники се различават по отношение

дали вторият низ е добавен за постоянно към първия или нов, трети

низ се създава от първите два - с други думи, дали операцията се променя

приемника.

За да създадете нов низ, състоящ се от два или повече струни, можете да използвате +

метод / оператор за изпълнение на оригиналните низове заедно. Ето какво трябва да каже irb

добавяне на низове:

>>"a" + "b"

=>"ab"

>>"a" + "b" + "c"

=>"abc"

Низът, който получавате обратно от +, винаги е нов низ. Можете да тествате това, като зададете a

низ към променлива, като се използва в операция + и се проверява каква е стойността след нея

операцията:

>> str = "Здравей"

=>"Здравей"

>> str + "там."

=>"Здравей."

>> ул

=>"Здравей"

Изразът str + "там." (което е синтактична захар за извикването на метода str. +

("там")) оценява на новия низ "Здравей там."B, но оставя str непроменен c .

За да добавите (добавите) втори низ за постоянно към съществуващ низ, използвайте <<

метод, който също има синтактична захар, псевдо-операторна форма:

>> str = "Здравей"

=>"Здравей"

>> str <<"там."

=>"Здравей."

>> ул

=>"Здравей."

В този пример първоначалният низ е добавил новия низ, както вие

може да видите от оценката на ул в края B .

б

° С

б

|  |
| --- |
| **227 серия** |

**227**

***Работа със струни***

Интерполацията на низове е (наред с други неща) друг начин за комбиниране на низове.

Вече сте го виждали в действие, но нека използваме възможността да разгледаме няколко

подробности за това как работи.

S TRING КОМБИНАЦИЯ ЧРЕЗ ИНТЕРПОЛАЦИЯ

В най-простата си интерполация на низове включва пускането на един съществуващ низ в друг:

>> str = "Здравей"

=>"Здравей"

>>"# {str} там."

=>"Здравей."

Резултатът е нов низ: „Здравей.“ Въпреки това е добре да имате предвид, че

интерполацията може да включва всеки израз на Ruby:

>>"Сумата е # {2 + 2}."

=>"Сумата е 4."

Кодът в къдравите скоби може да бъде всякакъв. (Те трябва да са къдрави скоби;

не е като% q {}, където можете да изберете свой собствен разделител.) Необичайно е да правите

кодът е ужасно сложен, защото това намалява структурата и четливостта на

програмата - но Руби е доволен от всеки интерполиран код и задължително ще го постави

низово представяне на стойността на кода във вашия низ:

>>"Казвам се # {class Person

attr\_accessor: име

край

d = Person.new

d.name = "Дейвид"

d.name

}. "

=>"Казвам се Дейвид."

Наистина, *наистина* не искате да правите това, но е важно да разберете, че можете

интерполирайте всеки код, който искате. Руби търпеливо чака всичко да изтича и след това хваща

крайна стойност на цялото нещо (d.name, в този случай, защото това е последният израз

вътре в блока за интерполация) и го интерполира.

Има много по-хубав начин да постигнете нещо подобно. Руби интерполира от

извикване на to\_s на обекта, към който оценява кодът за интерполация. Можеш да вземеш

предимството на това за рационализиране на конструкцията на низове, чрез дефиниране на вашия собствен to\_s meth-

подходящо:

>> клас Личност

>> attr\_accessor: име

>> def to\_s

>> име

>> край

>> край

=>: до\_с

>> david = Person.new

=> # <Лице: 0x00000101a73cb0>

>> david.name = "Дейвид"

|  |
| --- |
| **228. страница** |

**228**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

=>"Дейвид"

>>"Здравейте, # {david}!"

=>"Здравей, Дейвид!"

Тук обектът David служи като интерполиран код и произвежда резултата от него

операция to\_s, която се дефинира като обвивка около метода за получаване на име. Използвайки

куката to\_s е полезен начин за контрол на поведението на вашите обекти в интерполирани низове.

Не забравяйте обаче, че можете също да кажете (в предишния пример) david.name. Така

ако имате по-широко използване на to\_s на клас от много специфичен сценарий за интерполация,

обикновено можете да го настаните.

След като създадете и евентуално промените низ, можете да поискате от него значителен брой

количество информация за себе си. Сега ще разгледаме как да търсим низове.

***8.1.3***

***Заявка за низове***

Струнните заявки се предлагат с няколко вкуса. Някои ви дават булева (вярно или невярно)

отговор, а някои ви дават вид доклад за състоянието на текущото състояние на низа.

Ще организираме нашето изследване на методите за низови заявки по тези линии.

Б ОЛЕАН СТРУДНИ ЗАПИТВАНИЯ

Можете да попитате низ дали включва даден подниз, като използвате include ?. Дадено

низът, използван по-рано („Ruby е страхотен език.“), може да изглеждат заявки за включване

като този:

>> string.include? ("Ruby")

=> вярно

>> string.include? ("английски")

=> невярно

Можете да тествате за дадено начало или край на низ със start\_with? и end\_with ?:

>> string.start\_with? ("Ruby")

=> вярно

>> string.end\_with? ("!!!")

=> невярно

И можете да тествате за отсъствие на съдържание - тоест за наличие на някакви символи

изобщо - с празните? метод:

>> string.empty?

=> невярно

>>"" .празен?

=> вярно

Имайте предвид, че празното пространство се брои като знаци; низът "" не е празен.

С ЪДЪРЖАНИЕ ЗАЯВКИ

Методите за размер и дължина (те са синоними на един и същ метод) правят това, което правят

имената предполагат, че го правят:

>> низ.размер

=> 24

|  |
| --- |
| **229 серия** |

**229**

***Работа със струни***

Ако искате да знаете колко пъти дадена буква или набор от букви се срещат в низ, използвайте

броя. За да преброите появата на една буква, посочете тази буква като аргумент. Все още

използвайки низа "Ruby е страхотен език.", има три повторения на "a":

>> string.count ("a")

=> 3

За да преброите колко от набор от букви има, можете да използвате тире, разделено

обхват:

>> string.count ("gm")

=> 5

Спецификациите на символите са чувствителни към малки и големи букви:

>> string.count ("AZ")

=> 1

Можете също така да предоставите изписан набор от знаци, които искате да преброите:

>> string.count ("aey.")

=> 10

За да се отрече търсенето - тоест да се преброи броят на символите, които не съответстват на

тези, които посочите - поставете ^ (карета) в началото на вашата спецификация:

>> string.count ("^ aey.")

=> 14

>> string.count ("^ gm")

=> 19

(Ако сте запознати с регулярните изрази, ще разпознаете техниката на карета като a

близък братовчед на отрицание на класове на регулярни изрази.) Можете да комбинирате

спецификации синтаксиси и дори предоставят повече от един аргумент:

>> string.count ("ag-m")

=> 8

>> string.count ("ag-m", "^ l")

=> 6

Друг начин за търсене на низове по отношение на тяхното съдържание е методът на индекса. индексът е

нещо като обратното на използването на [] с числов индекс: вместо да търсите под-

низ в определен индекс, той връща индекса, при който възниква даден подниз. The

връща се първото появяване отляво. Ако искате първото появяване от

вдясно, използвайте rindex:

>> string.index ("готино")

=> 10

>> string.index ("l")

=> 13

>> string.rindex ("l")

=> 15

**Три букви плюс точка**

**и космически символи**

**Пребройте „a“ и „gm“**

**с изключение на „l“**

|  |
| --- |
| **Стр. 230** |

**230**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Въпреки че низовете са съставени от символи, Ruby няма отделен клас символи.

Едносимволните низове могат да ви кажат техния порядков код, благодарение на метода ord:

>>"a" .ord

=> 97

Ако вземете реда на по-дълъг низ, ще получите кода за първия символ:

>>"abc" .ord

=> 97

Обратната операция е достъпна като метод chr за цели числа:

>> 97.chr

=>"а"

Задаване на число, което не съответства на нито един знак за неговия chr еквивалент

причинява фатална грешка.

В допълнение към предоставянето на информация за себе си, низовете могат да ги сравняват -

аз с други струни, за да тестваме за равенство и ред.

***8.1.4***

***Сравнение на низове и подреждане***

Класът String се смесва в модула Comparable и дефинира метод <=>. Струни

следователно е добре да се направи, когато става въпрос за сравнения, базирани на символен код

( ASCII или по друг начин) поръчка:

>>"a"<=>"b"

=> -1

>>"b">"a"

=> вярно

>>"a">"A"

=> вярно

>>".">","

=> вярно

Не забравяйте, че методът / операторът на космическия кораб връща -1, ако е правилният обект

по-голямо, 1, ако левият обект е по-голям, и 0, ако двата обекта са равни. В първия случай

в предишната последователност връща -1, защото низът "b" е по-голям от низа

"а". Но "a" е по-голямо от "A", тъй като редът се извършва по стойност на символа и

символните стойности за "a" и "A" са съответно 97 и 65 в кодирането по подразбиране на Ruby-

създаване на UTF-8 . По същия начин низът „.“ е по-голямо от „,“, тъй като стойността за a

периодът е 46, а този за запетая е 44. (Вижте раздел 8.1.7 за повече информация относно кодирането.)

Подобно на Ruby обектите като цяло, низовете имат няколко метода за тестване на равенството.

С OMPARING два низа за равенство

Най-често срещаният метод за сравнение на низове е ==, който тества за равенство на

съдържание съдържание:

>>"низ" == "низ"

=> вярно

>>"низ" == "къща"

=> невярно

|  |
| --- |
| **Страница 231** |

**231**

***Работа със струни***

Двата буквални "низ" обекта са различни обекти, но имат едно и също съдържание.

Следователно те преминават теста ==. Низът "къща" има различно съдържание и е там-

fore не се считат за равни, базирани на ==, с "низ".

Друг метод за тест за равенство, String # eql?, Тества два низа за идентично съдържание.

На практика обикновено връща същия резултат като ==. (Има фини разлики в

изпълнението на тези два метода, но можете да използвате и двата. Ще откриете, че ==

е по-често.) Трети метод, String # jednaк ?, се държи като равни? обикновено прави:

той проверява дали два низа са един и същ обект - или по този въпрос дали низ

и всеки друг обект е същият обект:

>>"a" == "a"

=> вярно

>>"a" .equal? ​​("a")

=> невярно

>>"a" .equal? ​​(100)

=> невярно

Първият тест е успешен, тъй като двата низа имат едно и също съдържание. Вторият тест

се проваля, тъй като първият низ не е същият обект като втория низ. (И разбира се

нито един низ не е същият обект като цяло число 100, така че и тестът се проваля.) Това е добро

напомняне за факта, че струните, които изглеждат идентични на окото, за Руби могат да имат различни

идентичности на предходния обект.

Следващите два раздела ще представят низови *трансформации* и *преобразувания* в това

поръчка. Разликата между двете е, че трансформацията включва прилагане на някои

вид алгоритъм или процедура към съдържанието на низ, докато преобразуването означава

извличане на втори, несвързан обект - обикновено дори и низ - от низа.

***8.1.5***

***Струнна трансформация***

Струнните трансформации в Ruby се разделят неофициално в три категории: случай, форматиране,

и трансформации на съдържание. Ще разгледаме всеки по ред.

С ASE ТРАНСФОРМАЦИИ

Струните ви позволяват да повдигате, спускате и сменяте кутията им. Всички методи за промяна на регистъра имат

еквиваленти на взрив, модифициращ приемника:

>> низ = "Дейвид А. Блек"

=>"Дейвид А. Блек"

>> string.upcase

=>"ДЕЙВИД А. ЧЕРЕН"

>> string.downcase

=>"Дейвид а. черен"

>> string.swapcase

=>"dAVID a. bLACK"

Има и хубаво усъвършенстване, което ви позволява да пишете низа с главни букви:

>> низ = "Дейвид"

=>"Дейвид"

>> string.capitalize

=>"Дейвид"

|  |
| --- |
| **232 серия** |

**232**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Подобно на другите трансформатори на случаи, капитализирането има еквивалент на място, главни букви !.

Можете да извършите редица трансформации във формат на низ, повечето от

които са предназначени да ви помогнат да направите струните си да изглеждат по-хубави.

F ОРМАТИРАНЕ НА ТРАНСФОРМАЦИИТЕ

Строго погледнато, трансформациите на формати са подмножество на трансформациите на съдържание; ако

последователността от символи, представени от низа, не се е променила, няма да бъде

голяма трансформация. Ще групираме под заглавието на форматирането някои трансформации

мации, чиято основна цел е да подобри представянето на низове.

Методите rjust и ljust разширяват размера на вашия низ до дължината ви

предоставете в първия аргумент, допълвайки празни интервали, ако е необходимо:

>> низ = "Дейвид А. Блек"

=>"Дейвид А. Блек"

>> string.rjust (25)

=>"

Дейвид А. Блек "

>> string.ljust (25)

=>"Дейвид А. Блек

"

Ако предоставите втори аргумент, той се използва като подложка. Този втори аргумент може да бъде

повече от един символ:

>> string.rjust (25, '.')

=>"........... Дейвид А. Блек"

>> string.rjust (25, '><')

=>"><><><><><> Дейвид А. Блек"

Моделът за подплънки се повтаря толкова пъти, колкото е подходящ, като се отрязва последното място -

ако е необходимо.

И за да закръгли нещата в сферата на оправданието, има централен метод, който

се държи като rjust и ljust, но поставя символите на низа в центъра:

>>"Средата" .center (20, "\*")

=>"\*\*\*\*\* Средната \*\*\*\*\*"

Нечетните петна за подплата се правят вдясно:

>>"Средата" .center (21, "\*")

=>"\*\*\*\*\* Средната \*\*\*\*\*\*"

И накрая, можете да украсите струните си, като премахнете празно пространство от едната или от двете страни,

използвайки методите strip, lstrip и rstrip:

>> низ = "Дейвид А. Блек"

=>"Дейвид А. Блек"

>> string.strip

=>"Дейвид А. Блек"

>> string.lstrip

=>"Дейвид А. Блек"

>> низ.rstrip

=>"Дейвид А. Блек"

И трите метода за отстраняване на низове имат! версии, които променят низ perma-

нежно на място.

|  |
| --- |
| **Страница 233** |

**233**

***Работа със струни***

С ЪДЪРЖАНИЕ ТРАНСФОРМАЦИИ

Ще разгледаме някои, макар и не всички начини, по които можете да трансформирате низ чрез промяна

неговото съдържание.

Методите chop и chomp са свързани с премахването на символи от

краищата на струните - но те го правят по различен начин. Основната разлика е, че котлет

премахва символ безусловно, докато chomp премахва целевия подниз, ако намери

този подниз в края на низа. По подразбиране целевият подниз на chomp е новият ред

знак, \ n. Можете да замените целта, като предоставите на chomp аргумент:

>>"Дейвид А. Блек" .chop

=>"Дейвид А. Блак"

>>"Дейвид А. Блек \ n" .chomp

=>"Дейвид А. Блек"

>>"Дейвид А. Блек" .chomp ('ck')

=>"Дейвид А. Бла"

Далеч най-честата употреба или на котлет, или на chomp е използването на chomp за премахване

нови редове от краищата на низове, обикновено низове, които идват в програмата в

форма на редове от файл или клавиатура.

И chop, и chomp имат еквиваленти, които променят низа на място.

На по-радикалния край на премахването на символите стои ясният метод, който

изпразва низ от всички негови символи, оставяйки празния низ:

>> низ = "Дейвид А. Блек"

=>"Дейвид А. Блек"

>> string.clear

=>""

>> низ

=>""

String # clear е чудесен пример за метод, който променя приемника си, но не го прави

завърши с! характер. Името ясен, ясно казва, така да се каже, че нещо

се случва със струната. Няма смисъл да имаме ясен метод, който

не е променил низа на място; това би било просто дълъг начин да се каже "" (

празен низ).

Ако искате да замените всичките си знаци, без непременно да напускате низа си

лишени от съдържание, можете да използвате replace, който приема низов аргумент и замества

текущото съдържание на низа със съдържанието на този аргумент:

>> низ = "(да бъде наречен по-късно)"

=>"(да бъде наречен по-късно)"

>> string.replace ("Дейвид А. Блек")

=>"Дейвид А. Блек"

Както е ясно, методът замяна постоянно променя низа - както се предлага,

още веднъж, с името.

Можете да насочите определени символи за премахване от низ с изтриване. Аргументът-

моментите за изтриване спазват същите правила като аргументите за преброяване (вж. раздел 8.1.3):

|  |
| --- |
| **Страница 234** |

**234**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

>>"Дейвид А. Блек" .delete ("abc")

=>"Dvid A. Blk"

>>"Дейвид А. Блек" .delete ("^ abc")

=>"aac"

>>"Дейвид А. Блек" .delete ("ae", "^ c")

=>"Dvi A. Blck"

Друга специализирана низова трансформация е крипта, която извършва криптиране на данни

Стандартно ( DES ) криптиране на низа, подобно на функцията на библиотеката на Unix crypt (3)

ция. Единственият аргумент на crypt е двусимволен солен низ:

>>"Дейвид А. Блек" .crypt ("34")

=>"347OEY.7YRmio"

Уверете се, че сте прочели надеждността на всички техники за криптиране, които използвате,

включително крипта.

Последната техника на трансформация, която ще разгледаме, е увеличаване на низовете. Можеш

вземете следващия най-висок низ с метода succ (наличен и под името

следващия). Подреждането на низовете е проектирано така, че да има смисъл, дори за сметка на

строг ред на символен код: "a" идва след """(обратния знак), както в

ASCII , но след "z" идва "aa", а не "{". Увеличаването продължава, по одометър,

в цялата азбука:

>>"a" .succ

=>"b"

>>"abc" .succ

=>"abd"

>>"azz" .succ

=>"baa"

Възможността за увеличаване на низове е полезна в случаите, когато се нуждаете от генерирани партиди

уникални низове, може би за използване като имена на файлове.

Както вече видяхте, низовете (подобно на други обекти) могат да се преобразуват с meth-

os в семейството to\_ \*. След това ще разгледаме някои допълнителни подробности за преобразуването на низове.

***8.1.6***

***Преобразуване на низове***

Методът to\_i, който видяхте в последната глава, е един от наличните методи за преобразуване

в състояние да струни. Този метод предлага допълнителна функция: ако му дадете положителна интеграция

ger аргумент в диапазона 2–36, низът, който преобразувате, се интерпретира като

представляващ число в основата, съответстващо на аргумента.

Например, ако искате да интерпретирате 100 като базово число 17, можете да го направите по следния начин:

>>"100" .to\_i (17)

=> 289

Резултатът е десетичен еквивалент на 100, основа 17.

Основа 8 и основа 16 се считат за специални случаи и имат специални методи за това

не е нужно да отивате по маршрута to\_i. Тези методи са съответно oct и hex:

|  |
| --- |
| **Страница 235** |

**235**

***Работа със струни***

>>"100" .окт

=> 64

>>"100" .hex

=> 256

Други методи за преобразуване, достъпни за низове, включват to\_f (да плава), to\_s (да низ;

той връща своя приемник) и to\_sym или intern, което преобразува низа в символ

обект. Нито едно от тях не прави особени изненади:

>>"1.2345" .до\_f

=> 1,2345

>>"Здравей" .to\_s

=>"Здравей"

>>"abcde" .to\_sym

=>: abcde

>>"1.2345и някои думи" .to\_f

=> 1,2345

>>"само някои думи" .to\_i

=> 0

Всеки низ се състои от последователност от байтове. Байтовете се преобразуват в символи. *Как* точно

те се преобразуват в символи - колко байта съставляват символ и какви са тези символи

ters са - е въпрос на *кодиране* , който сега ще разгледаме накратко.

***8.1.7***

***Кодиране на низове: Кратко въведение***

Темата за кодирането на знаци е интересна, но обширна. Кодирането е много, и

има далеч от глобален консенсус относно един-единствен най-добър. Ruby 1.9 добави много

на кодиране на интелигентност и функционалност в низове. Голямата промяна в Ruby 2 беше

използването на UTF-8 , вместо US-ASCII , като кодиране по подразбиране за Ruby скриптове.

Кодирането в Ruby продължава да бъде област на непрекъснато обсъждане и развитие.

Тук няма да го изследваме дълбоко, но ще го поставим на радара си и ще разгледаме някои важни

техники, свързани с tant кодиране.

S настройвате кодирането на файла SOURCE

Като начало изходният ви код използва определено кодиране. По подразбиране изходните файлове на Ruby

използвайте UTF-8 кодиране. Можете да определите това, като помолите Ruby да покаже стойността

\_\_КОДИРАНЕ\_\_. Поставете този ред във файл и го стартирайте:

поставя \_\_ENCODING\_\_

Трябва да поставите реда във файл, защото може да получите различни резултати, ако стартирате

команда директно от командния ред. Причината за разликата е, че файл-

по-малко Ruby run взема кодирането си от текущата локална настройка. Можете да проверите това чрез

наблюдение на ефекта от изпълнението на една и съща команда с варианта на LANG среда

може да зададе различна стойност:

LANG = bg.iso885915 ruby ​​-e 'поставя \_\_ENCODING\_\_'

**Изход: UTF-8**

**Изход: US-ASCII**

|  |
| --- |
| **236 серия** |

**236**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

За да промените кодирането на изходен файл, трябва да използвате *вълшебен коментар* в горната част на

файлът. Вълшебният коментар приема формата

# кодиране: *кодиране*

където *кодирането* е идентификатор за кодиране. Например за кодиране на изходен файл

в US-ASCII поставяте този ред в горната част на файла:

# кодиране: ASCII

Тази линия (което може да се използва думата *кодиране* , а не думата *кодиране* , ако предпочитате)

понякога се нарича „вълшебен коментар“.

В допълнение към вашия изходен файл, можете също да направите заявка и да зададете кодирането на individ-

ual струни.

Е КОДИРАНЕ НА ЛИЧНИ СТРУНИ

Низовете ще ви кажат кодирането им:

>> str = "Тестов низ"

=>"Тестов низ"

>> str.encoding

=> # <Кодиране: UTF-8>

Можете да кодирате низ с различно кодиране, стига преобразуването от

оригиналното кодиране към новото - *прекодирането* - е разрешено (което зависи от

възможността на низа с новото кодиране):

>> str.encode ("US-ASCII")

=>"Тестов низ"

Ако трябва, можете да принудите кодиране с метода force\_encoding, който

заобикаля таблицата на „разрешените“ кодирания и кодира байтовете на низа с

кодирането, което посочите, безусловно.

Взривната версия на кодирането променя кодирането на низа постоянно:

>> str.encode! ("US-ASCII")

=>"Тестов низ"

>> str.encoding

=> # <Кодиране: US-ASCII>

Кодирането на низ също се влияе от наличието на определени символи в

низ и / или чрез изменение на низа с определени знаци. Можете да представите

изпрати произволни символи в низ, използвайки или \ x екранираща последователност, за двуцифрена

шестнадесетично число, представляващо байт или \ u избягващата последователност, която е след

намален от UTF-8 код и вмъква съответния знак.

Ефектът върху кодирането на низа зависи от характера. Дадено кодиране

на US-ASCII , добавяйки екраниран символ в диапазона 0–127 (0x00-0x7F в шестнадесетичен

mal) оставя кодирането непроменено. Ако знакът е в диапазона 128–255 (0xA0-

0x FF ), кодирането преминава към UTF-8 . Ако добавите UTF-8 знак в диапазона

0x0000–0x007F, кодирането на ASCII низа не е засегнато. UTF-8 кодове по-високи от

0x007F карат кодирането на низа да премине към UTF-8 . Ето пример:

|  |
| --- |
| **Страница 237** |

**237**

***Символи и тяхното използване***

>> str = "Тестов низ"

=>"Тестов низ"

>> str.encode! ("US-ASCII")

>> str <<". Евро символ: \ u20AC"

=>"Тестов низ. Евро символ: €"

>> str.encoding

=> # <Кодиране: UTF-8>

Последователността \ u избягване B ви позволява да вмъкнете всеки UTF-8 знак, независимо дали можете да го въведете

директно или не.

Има много повече по темата за кодирането на символи и низове, но вие сте

в този момент се вижда достатъчно, за да се знаят видовете операции, които са на разположение. Как

в крайна сметка изследването на кодирането ще зависи от вашите нужди като разработка на Ruby-

опера. Отново, имайте предвид, че кодирането обикновено е във фокуса на особено интензивно

дискусия и развитие в Ruby (и другаде).

На този етап ще завършим нашето проучване на низови методи и ще се обърнем към клас с

някои силни афинитети към класа String, но също така и някои интересни разлики:

Клас на символа.

***8.2***

***Символи и тяхното използване***

*Символите* са екземпляри на вградения Ruby клас Symbol. Те имат литерален конструктор:

водещото дебело черво. Винаги можете да разпознаете буквален символ (и да го разграничите от

низ, име на променлива, име на метод или нещо друго) от този маркер:

: а

:Книга

: "Ето как да направите символ с интервали в него."

Можете също така да създадете символ програмно, като извикате метода to\_sym (също

известен със синонима intern) на низ, както видяхте в последния раздел:

>>"a" .to\_sym

=>: a

>>"Преобразуване на низ в символ с стажант ...."

=>: "Преобразуване на низ в символ с стажант ...."

Обърнете внимание на издайническите водещи двоеточия на резултатите от оценката, върнати от irb.

Можете лесно да конвертирате символ в низ:

>>: a.to\_s

=>"а"

Това обаче е само началото. Символите се различават от низовете по важни начини. Нека да

погледнете символите според собствените им условия и след това се върнете към сравнителния поглед върху символите

бол и струни.

***8.2.1***

***Основни характеристики на символите***

Символите са твърд орех за много хора, които учат Ruby. Те не са съвсем като

нещо друго и те не отговарят точно на типовете данни, които повечето хора са дошли

по-рано. В някои отношения те са доста подобни на струни, но в същото време те

б

|  |
| --- |
| **Страница 238** |

**238**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

имат много общо с цели числа. Определено си струва да разгледаме внимателно техния началник

характеристики: неизменност и уникалност.

I МУТАБИЛНОСТ

Символите са неизменни. Няма такова нещо като добавяне на символи към символ;

след като символът съществува, това е. Никога няма да видите: abc <<: d или нещо подобно.

Това не означава, че няма символ: abcd. Има, но е напълно различно-

входящ символ от: abc. Не можете да промените: abc себе си. Подобно на цяло число, символ не може да бъде

променен. Когато искате да се позовете на 5, не променяте обекта 4, като добавяте 1 към него.

Можете да добавите 1 до 4, като извикате 4. + (1) (или 4 + 1), но не можете да накарате обекта 4 да бъде

обектът 5. По същия начин, въпреки че можете да използвате символ като подсказка за Руби за генералите

на друг символ, не можете да промените даден символ.

U НИКВЕНОСТ

Символите са уникални. Винаги, когато видите: abc, виждате представяне на

един и същ обект. Отново символите в това отношение приличат повече на цели числа, отколкото на низове. Кога

виждате обозначението "abc" на две места, гледате изображения на две различни

предходни низови обекти; конструкторът на литерални низове създава нов низ. Но: abc е

винаги един и същ обект Symbol, както 100 винаги е един и същ обект.

Можете да видите разликата между низовете и символите по отношение на уникалността

чрез запитване на обекти към техния object\_id, който е уникален за всеки отделен обект:

>>"abc" .object\_id

=> 2707250

>>"abc" .object\_id

=> 2704780

>>: abc.object\_id

=> 160488

>>: abc.object\_id

=> 160488

Нотацията "abc" създава нов низ всеки път, както можете да видите от факта, че всеки

такъв низ има различен идентификатор на обект . Но обозначението: abc винаги представлява едно и също

обект; : abc се идентифицира със същия идентификационен номер, независимо колко пъти го попитате.

Тъй като символите са уникални, няма смисъл да има конструктор за тях; Руби

няма символ # нов метод. Не можете да създадете символ повече, отколкото можете да създадете

ново цяло число. И в двата случая можете да се позовавате само на тях.

Думата *символ* има широки конотации; звучи така, сякаш може да се отнася до всяка идентификация

фиер или жетон. Важно е да получите манипулатор за връзката между символните обекти

и символи в по-общ смисъл.

***8.2.2***

***Символи и идентификатори***

Този код включва един обект Symbol (: x) и един идентификатор (и) на локална променлива:

s =: x

Но не е необичайно да се отнася към s като символ. И това *е* символ, в смисъл, че той

представлява нещо различно от себе си. Всъщност една от потенциалните причини за объркване

|  |
| --- |
| **239 серия** |

**239**

***Символи и тяхното използване***

заобикалящ клас Symbol и символни обекти е фактът, че символните обекти *не го правят*

представляват нещо различно от себе си. В известен смисъл името на променлива е по-скоро „сим-

болич ”от символ.

И съществува връзка между символни обекти и символни идентификатори. Между-

накрая, Ruby използва символи, за да следи всички имена, които е създал за променливи, meth-

коефициенти и константи. Можете да видите списък с тях, като използвате класа Symbol.all\_symbols

метод. Бъди предупреден; има много от тях! Ето върха на айсберга:

>> Symbol.all\_symbols

=> [: проверка,: стажант,: object\_id,: const\_missing,: method\_missing,

: method\_added,: singleton\_method\_added,: method\_removed,

: singleton\_method\_removed,

И продължава, изброявайки повече от 3000 символа.

Когато присвоявате стойност на променлива или константа, или създавате клас или пишете a

метод, избраният от вас идентификатор влиза във вътрешната таблица със символи на Ruby. Можеш

проверете за доказателства за това с някои техники за сондиране на масиви:

>> Symbol.all\_symbols.size

=> 3118

>> abc = 1

=> 1

>> Symbol.all\_symbols.size

=> 3119

>> Symbol.all\_symbols.grep (/ abc /)

=> [: abc]

Можете да видите от измерването на размера на масива, върнат от all\_symbols

че нараства с 1, когато направите задача за abc. Освен това символът: abc

сега присъства в масива, както е показано от операцията grep.

Таблицата със символи е точно това: таблица със символи. Това не е обектна таблица. Ако използвате iden-

tifier за повече от една цел - да речем, като локална променлива, а също и като име на метод -

съответният символ ще се появи само веднъж в таблицата със символи:

>> def abc; край

=>: abc

>> Symbol.all\_symbols.size

=> 3119

Тестовете за включване на символи винаги са верни

grep е основан на регулярен израз начин за търсене на съвпадащи елементи в масив.

Защо просто не кажа това?

>> Symbol.all\_symbols.include? (: Abc)

Защото винаги ще е истина! Самият акт на писане: abc в включва? тест поставя

символа: abc в таблицата със символи, така че тестът преминава, дори ако не е имало предварително

присвояване на идентификатор abc.

**Използвайте grep, вместо да включвате?**

**(виж следната бележка)**

**Използва повторно идентификатора „abc“**

**Дефинициите на метода се връщат**

**техните имена като символи**

**Същия размер; : abc е в**

**маса само веднъж**

|  |
| --- |
| **Страница 240** |

**240**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Руби следи за какви символи трябва да знае, за да може да ги търси

бързо. Включването на символ в таблицата със символи не ви казва нищо

за какво служи символът.

Идвайки в пълен кръг, можете също да видите, че когато присвоявате символ на променлива,

този символ се добавя към таблицата:

>> abc =: my\_symbol

=>: my\_symbol

>> Symbol.all\_symbols.size

=> 3020

>> Symbol.all\_symbols.grep (/ my\_symbol /)

=> [: my\_symbol]

В таблицата се поставят не само символи, съответстващи на имена на променливи и методи; всякакви символи

bol Ruby вижда навсякъде в програмата е добавен. Фактът, че: my\_symbol се съхранява

в таблицата със символи поради това, че сте го използвали, означава, че следващия път, когато използвате

Руби ще може бързо да го търси. И за разлика от символ, който съответства на

идентификатор, на който сте присвоили по-сложен обект, като низ или масив, символ

bol, която използвате чисто като символ, като: my\_symbol, не изисква повече

погледни нагоре. Това е само себе си: символът: my\_symbol.

Ruby ви позволява, програмистът, да използвате същия механизъм за съхранение на символи, който и вие

Ruby използва за проследяване на идентификатори. Само че не проследявате идентификатори; използваш sym-

болс за вашите собствени цели. Но все пак получавате предимствата на Руби, излагайки цялото

символен механизъм за използване от страна на програмиста.

Какво правите със символите?

***8.2.3***

***Символи на практика***

Символите имат редица приложения, но повечето изяви попадат в една от двете категории:

аргументи на метода и хеш ключове.

С ИМБОЛИ КАТО МЕТОДНИ АРГУМЕНТИ

Редица основни методи на Ruby вземат символи като аргументи. Много такива методи

може да приема и струни. Вече сте видели няколко примера от attr\_ \*

семейство методи:

attr\_accessor: име

attr\_reader: възраст

Методът за изпращане, който изпраща съобщение до обект без точката, може да вземе символ:

"abc" .send (: upcase)

Обикновено не е необходимо изпращане, ако предварително знаете цялото име на метода. Но

урокът тук е, че изпращането може да вземе символ, който остава верен, дори ако символът

bol се съхранява в променлива, а не се записва и / или се определя динамично

по време на изпълнение.

В същото време повечето методи, които вземат символи, също могат да поемат низове. Можеш

замени: upcase с "upcase" в предишния пример за изпращане и ще работи. The

|  |
| --- |
| **Страница 241** |

**241**

***Символи и тяхното използване***

Разликата е, че предоставяйки: upcase, вие спестявате на Руби неприятностите при превода

низът се превръща в символ вътрешно по пътя за намиране на метода.

Възможно е да прекалите. Понякога ще виждате код като този:

some\_object.send (method\_name.to\_sym)

Предприема се допълнителна стъпка (преобразуването to\_sym) по пътя към предаване на аргумент на

изпрати. Няма смисъл да правите това, освен ако извиканият метод не може да се справи само

символи. Ако може да обработва низове и имате низ, предайте низа. Оставете метода

обработва преобразуването, ако е необходимо.

Следва: символи като хеш ключове. Няма да разглеждаме задълбочено хешовете до глава 9, но

използването на символи като хеш ключове е изключително широко разпространено и си струва да се постави на нашия

радар сега.

S YMBOLS КАТО ХАШ КЛЮЧОВЕ

А *хашиш* е изнервящ структура на данните: поставите ценности в нея чрез присвояване на стойността на един клавиш

и извличате стойност, като предоставяте препратка към ключ. Руби не поставя никакви ограничения

хеш ключове. Можете да използвате масив, клас, друг хеш, низ или всеки обект, който харесвате като

хеш ключ. Но в повечето случаи е вероятно да използвате низове или символи.

Ето създаването на хеш със символи като ключове, последвано от извличането на такъв

от стойностите:

>> d\_hash = {: name =>"Дейвид",: възраст => 55}

=> {: name =>"Дейвид",: възраст => 55}

>> d\_hash [: възраст]

=> 55

И ето подобен хеш с низови ключове:

>> d\_hash = {"name" =>"Дейвид", "възраст" => 55}

=> {"name" =>"Дейвид", "възраст" => 55}

>> d\_hash ["име"]

=>"Дейвид"

Няма нищо ужасно в използването на низове като хеш ключове, особено ако вече имате

колекция от струни на ръка и трябва да ги включите в хеш. Но символи

имат няколко предимства в отдела за хеш ключове.

Помислете дали да не разрешите символи или низове като аргументи на метода

Когато пишете метод, който ще вземе аргумент, който може да бъде a

низ или символ, често е хубаво да разрешите и двете. Не е необходимо в случаите, когато сте

работа с генерирани от потребителя произволни низове или където е текстът, прочетен от файл

участващи; те така или иначе няма да бъдат под формата на символи. Но ако имате метод, който очаква,

да речем, име на метод или може би стойност от крайна таблица с етикети или етикети, това е учтиво

за да разрешите или низове, или символи. Това означава да се избягва да се прави нещо на обекта, който

изисква тя да бъде една или друга и това ще доведе до грешка, ако е грешната. Вие

може да нормализира аргумента с извикване на to\_sym (или to\_s, ако искате да нормализирате до

низове), така че каквото и да се предаде да се впише в операциите, които трябва да изпълните.

|  |
| --- |
| **Страница 242** |

**242**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Първо, Ruby може да обработва символи по-бързо, така че ако правите много хеш търсения,

ще спестите малко време. Няма да забележите разлика, ако обработвате само малки

количества данни, но ако трябва да промените за ефективност, символните хеш ключове вероятно са

добра идея.

Второ, символите изглеждат добре като хеш ключове. Да изглеждаш добре, разбира се, не е техника

кал характеристика, а мнението за това, което изглежда добре, варира в широки граници. Но символите го правят

имат вид замразен, подобен на етикет вид, който се поддава добре на случаите, когато вашият хеш

ключовете са предназначени да бъдат статични идентификатори (като: name и: age), докато низовете имат a

податливост, която е подходяща за представяне на произволни ценности (като нечия

име). Може би това е случай на проектиране на техническата основа на двата обекта -

струните са променливи, символите не - в естетическата равнина. Както и да е, Руби

програмистите са склонни да използват символи повече от низове като хеш ключове.

Третата причина да се използват символи, а не низове като хеш ключове, когато е възможно, е

че Ruby позволява специална форма на представяне на символ в позиция хеш ключ, с

двоеточието след символа вместо преди него и стрелката за разделител на хеш премахнати.

С други думи,

хеш = {: name =>"Дейвид",: възраст => 55}

може да се запише и като

хеш = {име: "Дейвид", възраст: 55}

Както често се случва, Руби се старае да ви позволява да пишете неща в претрупано,

прост начин. Разбира се, ако предпочитате версията със стандартната символна нотация и

хеш стрелките, все още можете да използвате тази форма.

Досега и по дизайн разглеждахме символите главно от светлината на това как те се различават

от струни. Но ще забележите, че струните влизат редовно в дискусията, не

без значение колко се опитваме да разделяме двете. Струва си да центрирате прожекторите

върху символите, но сега нека го разширим и да разгледаме някои конкретни точки за сравнение

между символи и низове.

***8.2.4***

***Струни и символи в сравнение***

Символите стават все по-подобни на низ в последователните версии на Ruby. Това не е

да кажат, че са изхвърлили своите забележителни черти; те все още са неизменни и уникални. Но

те представят значително по-низовиден интерфейс, отколкото преди.

Като груба демонстрация на промените, ето два списъка с методи.

Първият идва от Ruby 1.8.6:

>> Symbol.instance\_methods (false) .sort

=> ["===", "id2name", "inspect", "to\_i", "to\_int", "to\_s", "to\_sym"]

Вторият е от Ruby 2:

>> Symbol.instance\_methods (false) .sort

=> [: <=>,: ==,: ===,: = ~,: [],: изписване с главни букви,: casecmp,: downcase,: празно ?,

: кодиране,: id2name,: inspect,: intern,: length,: match,: next,: size,

: slice,: succ,: swapcase,: to\_proc,: to\_s,: to\_sym,: upcase]

|  |
| --- |
| **Страница 243** |

**243**

***Числени обекти***

Някъде по линията символите са се научили да правят много нови неща, най-вече от

низовия домейн. Но имайте предвид, че няма версии на взрива на различните променящи се букви

и методи за увеличаване. За струни, upcase! означава да се *надградите на място* . Sym-

болите, от друга страна, са неизменни; символът: a може да ви покаже символа: A,

но не може да бъде символът: A.

По принцип семантиката на методите, подобни на низ, е същата като

низови еквиваленти, включително нарастване:

>> sym =: Дейвид

=>: Дейвид

>> sym.upcase

=>: DAVID

>> sym.succ

=>: Дейви

>> sym [2]

=>"v"

>> sym.casecmp (: Дейвид)

=> 0

Имайте предвид, че индексирането в символ връща подниз B , а не символ. От про-

граматическа перспектива, символите признават факта, че са представяне на

текст, като ви даде редица начини за манипулиране на тяхното съдържание. Но всъщност не е

палатка; : Дейвид не съдържа „Дейвид“, а повече от 100 съдържа „100“. Това е въпрос на

интерфейса и на характерно за Руби сходство на обектната теория и про-

практичност на граматирането.

Отдолу символите приличат повече на цели числа, отколкото на низове. (Таблицата със символи е основно-

наречен хеш, базиран на цяло число.) Те споделят с цели числа не само неизменност и

уникалност, но също и непосредственост: променлива, към която е обвързан символ, осигурява

действителна стойност на символа, а не препратка към нея. Ако сте озадачени как точно символите

работа, или защо и низовете, и символите съществуват, когато изглежда се дублират

усилията на другия в представянето на текст, мислете за символите като цялоподобни обекти

облечени в герои. Звучи странно, но обяснява много.

Целочислените качества на символите също осигуряват хубав преход към темата за

числови обекти.

***8.3***

***Числени обекти***

В Ruby числата са обекти. Можете да изпращате съобщения до тях, точно както можете до всеки обект:

n = 99,6

m = n.около

поставя m

x = 12

ако x.zero?

поставя "x е нула"

друго

поставя "x не е нула"

край

поставя "ASCII символът, еквивалентен на 97, е # {97.chr}"

б

б

° С

д

|  |
| --- |
| **Страница 244** |

**244**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

Както ще видите, ако стартирате този код, числата с плаваща запетая знаят как да ги закръглят-

аз B (нагоре или надолу). Числата като цяло знаят c дали са нула. И инте-

gers могат да се преобразуват в символа, който съответства на тяхната ASCII стойност d .

Числата са обекти; следователно те имат класове - цяло родословно дърво от тях.

***8.3.1***

***Числови класове***

Няколко класа съставят числовия пейзаж. Фигура 8.1 показва малко опростена

изглед (смесени модули не са показани) на тези класове, илюстриращи наследяването

отношенията между тях.

Най-добрият клас в йерархията на числовите класове е Numeric; всички останали

слез от него. Първият клон в дървото е между плаваща запетая и интеграл

числа: класовете Float и Integer. Целите числа са разделени на два класа: Fixnum

и Bignum. Както може да предположите, Bignums са големи цели числа. Когато използвате или изчислявате

късно цяло число, което е достатъчно голямо, за да бъде Bignum, а не Fixnum, Ruby обработва

преобразуването автоматично за вас; не е нужно да се притеснявате за това.

***8.3.2***

***Извършване на аритметични операции***

В по-голямата си част числата в Ruby се държат като правилата за аритметика и обичайните

конвенциите на аритметичната нотация ви карат да очаквате. Примерите в таблица 8.1

трябва да бъде успокояващо в тяхната скучност.

Имайте предвид, че когато разделяте цели числа, резултатът винаги е цяло число. Ако искаш

разделение с плаваща запетая, трябва да подадете Ruby числа с плаваща запетая (дори ако всички

което правите е да добавите .0 в края на цяло число).

Таблица 8.1 Общи аритметични изрази и техните оценителни резултати

Израз

Резултат

Коментари

1 + 1

2

Събиране

10/5

2

Целочислено разделение

16/5

3

Цяло число (без автоматично преобразуване с плаваща запетая)

10 / 3.3

3.3333333333

Деление с плаваща запетая

1,2 + 3,4

4.6

Добавяне с плаваща запетая

Числови

Цяло число

Плувка

Bignum

Fixnum

Фигура 8.1 Числена йерархия на класове

|  |
| --- |
| **Страница 245** |

**245**

***Числени обекти***

Ruby също ви позволява да манипулирате числа в недексимални бази. Шестнадесетични цели числа

са обозначени с водещ 0x. Ето някои оценки на irb на шестнадесетично цяло число

изрази:

>> 0x12

=> 18

>> 0x12 + 12

=> 30

Вторият 12 в последния израз Б е десетичен 12; префиксът 0x се отнася само за

номерата, на които се появява.

Целите числа, започващи с 0, се интерпретират като *осмични* (основа 8):

>> 012

=> 10

>> 012 + 12

=> 22

>> 012 + 0x12

=> 28

Както видяхте в раздел 8.1.6, можете също да използвате метода to\_i на низове за конвертиране

числа във всяка основа до десетична. За да извършите такова преобразуване, трябва да предоставите

основата, която искате да конвертирате *от* като аргумент, за да to\_i. След това низът се интер-

изведени като цяло число в тази основа и целият израз връща десетичната еквивалентна

алент. Можете да използвате всяка основа от 2 до 36 включително. Ето няколко примера:

>>"10" .to\_i (17)

=> 17

>>"12345" .to\_i (13)

=> 33519

>>"рубин" .to\_i (35)

=> 1194794

Имайте предвид, че повечето аритметични оператори, които виждате в Ruby, са *методи* . Те

не изглеждайте по този начин заради синтактичната захар, подобна на оператора, която им дава Руби.

Но те са методи и те могат да бъдат наречени като методи:

>> 1. + (1)

=> 2

>> 12./(3)

=> 4

>> -12 .- (- 7)

=> -5

-12 - -7

–5

Изваждане

10% 3

1

По модул (остатък)

Таблица 8.1 Общи аритметични изрази и техните оценителни резултати ***(продължение)***

Израз

Резултат

Коментари

б

|  |
| --- |
| **Страница 246** |

**246**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

На практика никой не пише аритметични операции по този начин; винаги ще виждате синтак-

тик еквиваленти на захар (1 + 1 и т.н.). Но виждайки примери за извикване на метод

form е добро напомняне за факта, че те са методи, а също и за факта, че ако

дефинирате, да речем, метод, наречен + в собствен клас, можете да използвате операторския

синтактична захар. (И ако видите аритметични оператори да се държат странно, може да е това

някой е предефинирал основните си методи.)

Ще се обърнем сега към следващата и последна категория скаларни обекти, които ще обсъдим в това

глава: обекти за час и дата.

***8.4***

***Времена и дати***

Ruby ви дава много начини да манипулирате часове и дати. Всъщност степента и вариацията

клас на класовете, които представляват времена и / или дати, и методът на класа и екземпляра

на разположение чрез тези класове, може да бъде объркващо. Така могат и различните начини

кои случаи на различните класове представляват себе си. Искате ли да знаете какво

денят, който наричаме 24 април 1705 г., би бил наречен в Англия преди календара

реформа от 1752 г.? Заредете пакета с дати и след това попитайте

>> изисква 'дата'

=> вярно

>> Date.parse ("24 април 1705"). England.strftime ("% B% d% Y")

=>"13 април 1705"

От по-малко екзотичната страна можете да извършите редица полезни и удобни манипулации.

лации върху обекти за час и дата.

Времената и датите се манипулират чрез три класа: Time, Date и DateTime.

(За удобство екземплярите на всички тези класове могат да се наричат ​​колективно като

*обекти за дата / час* .) За да извлечете пълните им предимства, трябва да изтеглите едната или и двете дати

и времеви библиотеки във вашата програма или irb сесия:

изисква "дата"

изисква "време"

Тук първият ред предоставя класовете Date и DateTime, а вторият ред

подобрява класа Time. (Всъщност, дори ако не ви е необходима „дата“, ще можете

вижте класа Date. Но все още не може да направи нищо.) В някакъв момент от бъдещето, всички

наличната свързана с дата и час функционалност може да бъде обединена в една библиотека и

предоставени на програми по подразбиране. Но за момента трябва да направите

изискват операции, ако искате пълната функционалност.

По-нататък ще разгледаме голяма част от операциите за дата / час - не всички

тях, но повечето от най-често срещаните и достатъчно, за да ви даде основание за по-нататъшно

развитие. По-конкретно, ще разгледаме как да създадем екземпляр на обекти за дата / час, как да

запитвайте ги и как да ги конвертирате от една форма или формат в друга.

***8.4.1***

***Инсталиране на обекти с дата / час***

Начинът на създаване на пример за обект дата / час зависи от това точно кой обект е включен.

Ще разгледаме класовете Date, Time и DateTime в този ред.

|  |
| --- |
| **Страница 247** |

**247**

***Времена и дати***

C ОТГОВОР НА ОБЕКТИ НА ДАТА

Можете да получите днешната дата с конструктора Date.today:

>> Дата.до днес

=> # <Дата: 02.11.2013 ((2456599j, 0s, 0n), + 0s, 2299161j)

Можете да получите по-прост низ, като стартирате to\_s на датата или като го поставите:

>> поставя Date.today

2013-11-02

Можете също така да създавате обекти за дата с Date.new (също като Date.civil). Изпрати

през година, месец и ден:

>> поставя Date.new (1959,2,1)

1959-02-01

Ако не е посочено, месецът и денят (или само денят) по подразбиране са 1. Ако не предоставите аргумент

по подразбиране годината е –4712 - вероятно не е най-полезната стойност.

И накрая, можете да създадете нова дата с конструктора на разбор, който очаква a

низ, представляващ дата:

>> поставя Date.parse ("2003/6/9")

2003-06-09

По подразбиране Руби разширява века за вас, ако предоставите едно- или двуцифрено число

бер. Ако числото е 69 или повече, тогава добавеното отместване е 1900; ако е между 0 и

68, изместването е 2000. (Това разграничение е свързано с началото на Unix

„Епоха“ в началото на 1970 г.)

>> поставя Date.parse ("03/6/9")

2003-06-09

>> поставя Date.parse ("33/6/9")

2033-06-09

>> поставя Date.parse ("77/6/9")

1977-06-09

Date.parse прави усилия, за да осмисли каквото хвърлите по него, и то доста

добър в работата си:

>> поставя Date.parse ("2 ноември 2013 г.")

2013-11-02

>> поставя Date.parse ("2 ноември 2013 г.")

2013-11-02

>> поставя Date.parse ("2 ноември 2013 г.")

2013-11-02

>> поставя Date.parse ("2013/11/2")

2013-11-02

Можете да създадете Julian и търговски (базирани на понеделник, а не на неделя

броене на ден от седмицата) Обекти с дата съответно с методите jd и commercial.

Можете също да сканирате низ срещу спецификация на формат, генерирайки обект Дата, с

strptime. Тези конструкторски техники са по-специализирани от останалите и ние

**Приема година /**

**поръчка месец / ден**

|  |
| --- |
| **Страница 248** |

**248**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

няма да навлиза в тях подробно тук; но ако вашите нужди са също толкова специализирани, датата

клас може да се обърне към тях.

Класът Time, подобно на класа Date, има множество конструктори.

C ПОВТОРЕНО ОБЕКТИ НА ВРЕМЕТО

Можете да създадете времеви обект, използвайки всеки от няколко конструктора: new (известен още като сега), at,

local (известен още като mktime) и синтактичен анализ. Това множество конструктори, макар и прекомерно

може да изглежда на пръв поглед, предоставя различни функционалности, всички те полезни. Тук са

някои примери, irb-стил:

>> Time.new

=> 2013-11-02 12:16:21 +0000

>> Time.at (100000000)

=> 1973-03-03 09:46:40 +0000

>> Time.mktime (2007,10,3,14,3,6)

=> 2007-10-03 14:03:06 +0100

>> изисква 'време'

=> вярно

>> Time.parse ("22 март 1985 г., 22:35 ч.")

=> 1985-03-22 22:35:00 +0000

Time.new (или Time.now) създава обект, представящ текущото време B .

Time.at (секунди) ви дава времеви обект за броя секунди от епохата

(полунощ на 1 януари 1970 г., GMT ), представено от секундния аргумент c .

Time.mktime (или Time.local) очаква година, месец, ден, час, минута и секунда

аргументи. Не е нужно да ги предоставяте; докато изпускате аргументи от

вдясно, Time.mktime се запълва с разумни настройки по подразбиране (1 за месец и ден; 0 за час,

минута и секунда) d .

За да използвате Time.parse, трябва да заредите библиотеката за време e . След като го направите, Time.parse

има колкото се може повече смисъл от аргументите, които му давате, подобно на Date.parse f .

C ПОВТОРЕНО ОБЕКТИ НА ДАТА / ВРЕМЕ

DateTime е подклас на Date, но неговите конструктори са малко по-различни благодарение на

някои замествания. Най-често срещаните конструктори са нови (достъпни и като граждански),

сега и анализирайте:

>> поставя DateTime.new (2009, 1, 2, 3, 4, 5)

2009-01-02T03: 04: 05 + 00: 00

=> нула

>> поставя DateTime.now

2013-11-03T04: 44: 52-08: 00

=> нула

>> поставя DateTime.parse ("23 октомври 1973 г., 10:34 ч.")

1973-10-23T10: 34: 00 + 00: 00

DateTime включва и специализирания jd (юлианска дата), търговски и strptime

конструктори, споменати по-рано във връзка с класа Date.

Нека се обърнем сега към различните начини, по които можете да запитвате обекти от дата / час.

б

° С

д

д

е

|  |
| --- |
| **Страница 249** |

**249**

***Времена и дати***

***8.4.2***

***Методи за запитване за дата / час***

Като цяло обектите за час и дата имат методите за заявка, които бихте очаквали

имат. Времевите обекти могат да бъдат запитвани за тяхната година, месец, ден, час, минута и секунда

ond, както и обектите с дата / час. Обектите за дата могат да бъдат запитвани за тяхната година, месец,

и ден:

>> dt = DateTime.now

=> # <DateTime: 2014-02-21T06: 33: 38-05: 00

((2456710j, 41618s, 552942000n), - 18000s, 2299161j)>

>> dt.година

=> 2014

>> dt.hour

=> 6

>> dt.minute

=> 33

>> dt.секунда

=> 38

>> t = Time.now

=> 2014-02-21 06:33:50 -0500

>> t.месец

=> 2

>> т. сек

=> 50

>> d = Date.today

=> # <Дата: 2014-02-21 ((2456710j, 0s, 0n), + 0s, 2299161j)>

>> д. ден

=> 21

Имайте предвид, че обектите за дата / час имат втори метод, както и сек. Времевите обекти имат

само сек.

Някои удобни методи от деня на седмицата работят еднакво и за трите класа. През

тях, можете да определите дали датата / часът е или не е определен ден от

седмицата:

>> d.понеделник?

=> невярно

>> dt.friday?

=> вярно

Други налични заявки включват булеви такива за високосна година (скок?) И лятно запазване

измерване на времето (dst ?, само за времеви обекти).

Както видяхте, низовите представяния на обектите дата / час се различават значително,

в зависимост от това какво точно сте поискали и с кой от трите класа се занимавате-

инжектиране с. На практика представянето на низовете по подразбиране не се използва много. Вместо това

обектите обикновено се форматират с помощта на методи, създадени за тази цел.

***8.4.3***

***Методи за форматиране на дата / час***

Всички обекти за дата / час имат метода strftime, който ви позволява да форматирате техните

полета по гъвкав начин, използвайки форматиращи низове, в стила на Unix strftime (3) sys-

tem библиотека:

|  |
| --- |
| **Страница 250** |

**250**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

>> t = Time.now

=> 2014-02-21 06:37:59 -0500

>> t.strftime ("% m-% d-% y")

=>"02-21-14"

В примера използваните спецификатори на формат са% m (двуцифрен месец),% d (двуцифрен

ден) и% Y (четирицифрена година). Тиретата между полетата се възпроизвеждат в

изход като буквални тирета.

Някои полезни спецификатори на формат за strftime са показани в таблица 8.2.

ПРЕДУПРЕЖДЕНИЕ Спецификаторите% c и% x, които включват комбинации от удобства

на други спецификатори, може да се различава в различните локали; например някои

системи поставят деня преди месеца във формат% x. Това е добре, защото

това означава, че стилът на определена държава не е кодиран твърдо в тези формати. Но

трябва да сте наясно с това, така че да не разчитате на конкретно поведение, което

не винаги може да получите. Когато се съмнявате, можете да използвате форматиран низ, съставен

на по-малки спецификатори.

Ето още няколко примера за спецификаторите на формата в действие:

>> t.strftime ("Днес е% x")

=>"Днес е 11/03/13"

>> t.strftime ("Иначе известен като% d-% b-% y")

=>"Иначе известно като 03-ноември-13"

>> t.strftime ("Или дори ден% e от% B,% Y.")

=>"Или дори ден 3 ноември 2013 г."

>> t.strftime ("Времето е% H:% m.")

=>"Часът е 04:11."

Таблица 8.2 Общи спецификатори на формат на час и дата

Спецификатор

Описание

% Y

Година (четири цифри)

% y

Година (последните две цифри)

% b , % B

Кратък месец, пълен месец

% m

Месец (брой)

%д

Ден от месеца (подплатен с нули вляво)

% д

Ден от месеца (подплатено със заготовки вляво)

% a , % A

Име на кратък ден, име на цял ден

% H , % I

Час (24-часов часовник), час (12-часов часовник)

% М

Минута

%С

Второ

%° С

Еквивалентно на „% a% b% d% H:% M:% S% Y“

%х

Еквивалентно на "% m /% d /% y"

|  |
| --- |
| **251 серия** |

**251**

***Времена и дати***

В допълнение към удобствата, предоставени от strftime, класовете Date и DateTime дават

имате шепа предварително приготвени изходни формати за специализирани случаи като RFC 2822 (имейл)

съответствие и HTTP формат, посочен в RFC 2616:

>> Date.today.rfc2822

=>"Неделя, 3 ноември 2013 г. 00:00:00 +0000"

>> DateTime.now.httpdate

=>"Нед, 03 ноември 2013 г. 12:49:48 GMT"

По един или друг начин можете да получите вашите часове и дати, за да изглеждате по начина, по който ги искате

да се. Обектите за дата / час също позволяват преобразувания от различен вид, от един клас на

възрази на друг.

***8.4.4***

***Методи за преобразуване на дата / час***

Всички класове дата / час позволяват конвертиране един към друг; тоест времето има

to\_date и to\_datetime методи, Date има to\_time и to\_datetime и Date-

Времето е to\_time и to\_date. Във всички случаи, когато целевият клас има повече информация

За разлика от клас източник, липсващите полета са зададени на 0 - по същество, полунощ,

защото и трите класа имат информация за дата, но само два имат информация за времето.

Можете също да се придвижвате в календара с определени аритметични методи във времето

и оператори.

D АРИТЕМИКА НА ВРЕМЕ / ВРЕМЕ

Обектите за време ви позволяват да добавяте и изваждате секунди от тях, връщайки нов времеви обект:

>> t = Time.now

=> 2013-11-03 04:50:49 -0800

>> t - 20

=> 2013-11-03 04:50:29 -0800

>> t + 20

=> 2013-11-03 04:51:09 -0800

Обектите за дата и дата интерпретират + и - като ежедневни операции и те позволяват

за месечни преобразувания с << и >>:

>> dt = DateTime.now

=> # <DateTime: 2013-11-03T04: 51: 05-08: 00 ...>

>> поставя dt + 100

2014-02-11T04: 51: 05-08: 00

>> поставя dt >> 3

2014-02-03T04: 51: 05-08: 00

>> поставя dt << 10

2013-01-03T04: 51: 05-08: 00

Можете също така да продължите напред, като използвате следващия (известен още като succ) метод. Цяло семейство от

методите next\_unit и prev\_unit ви позволява да се движите напред-назад по ден (и), месец (и),

или година (и):

>> d = Date.today

=> # <Дата: 2013-11-03 ((2456600j, 0s, 0n), + 0s, 2299161j)>

>> поставя d.next

2013-11-04

|  |
| --- |
| **252 серия** |

**252**

C ГЛАВА 8 ***Струни, символи и други скаларни обекти***

>> поставя d.next\_year

2014-11-03

>> поставя d.next\_month (3)

2014-02-03

>> поставя d.prev\_day (10)

2013-10-24

Освен това обектите за дата и дата / час ви позволяват да итерирате в редица от тях,

използвайки методите upto и downto, всеки от които отнема време, дата или дата / час

обект. Ето един нов пример:

>> d = Date.today

=> # <Дата: 2013-11-03 ((2456600j, 0s, 0n), + 0s, 2299161j)>

>> следваща\_седмица = d + 7

=> # <Дата: 2013-11-10 ((2456607j, 0s, 0n), + 0s, 2299161j)>

>> d.upto (следваща\_седмица) {| дата | поставя "# {date} е # {date.strftime ("% A ")}"}}

2013-11-03 е неделя

2013-11-04 е понеделник

2013-11-05 е вторник

2013-11-06 е сряда

2013-11-07 е четвъртък

2013-11-08 е петък

2013-11-09 е събота

2013-11-10 е неделя

Класовете за дата / час предлагат много повече от това, което сте виждали тук. Но характеристиките

ние сме разгледали са най-често срещаните и, по всяка вероятност, най-полезните. Не забравяйте

че винаги можете да използвате инструмента за команден ред ri, за да получите информация за методите!

Ако опитате ri дата например, ще получите информация за класа, както и списък с

налични методи на клас и екземпляр - всеки от които можете да стартирате ri поотделно.

Стигнахме до края на нашето проучване на скаларни обекти в Ruby. След това, в глава 9,

ще разгледаме колекции и обекти на контейнери.

***8.5***

***Обобщение***

В тази глава сте виждали

Създаване и манипулиране на струни

Работата на символите

Числови обекти, включително плувки и цели числа

Дата, час и обекти на дата / час и как да ги търсите и манипулирате

Накратко, ние разгледахме основите на най-често срещаните и важни скаларни обекти в

Руби. Някои от тези теми включват консолидиране на точките, изложени по-рано в книгата;

други бяха нови в тази глава. Във всеки момент ние разгледахме селекция от важни,

общи методи. Също така разгледахме как се отнасят някои от класовете скаларни обекти

взаимно. Низовете и символите представляват текст и въпреки че са различни

видове обекти, преобразуването от един в друг е лесно и често. Числа и

низовете също си взаимодействат. Преобразуванията не са автоматични, както са (например) в Perl; но

Ruby предлага методи за преобразуване, за да премине от низ към числов обект и обратно, както

|  |
| --- |
| **253 серия** |

**253**

***Обобщение***

както и начини за конвертиране на низове в цели числа в толкова бази, колкото 10 цифри и 26 букви

на английската азбука може да побере.

Обектите за час и дата имат стъпка както в низ, така и в числови лагери. Можеш

извършете изчисления върху тях, като добавете *n* месеца към дадена дата и можете

също ги поставяйте през техните крачки като низове, използвайки техники като Time # strftime

метод във връзка с спецификаторите на изходния формат.

Светът на скаларни обекти в Ruby е богат и динамичен. Повечето от това, с което правите

и Ruby, и Rails ще произлязат от това, което сте научили тук за скаларни обекти:

директна манипулация на тези обекти, манипулация на обекти, които споделят някои от техните

черти (например CGI параметри, чието съдържание са низове), или колекции от mul-

обекти от тези категории. Скаларните обекти не са всичко, но те лежат в

корен на почти всичко останало. Обиколката, която направихме на важни скаларни класове и

методите в тази глава ще ви помогнат, докато продължим да разглеждаме

лекции и контейнери - двуизмерните (а понякога и повече) измерени граждани на Руби

обектния свят.

|  |
| --- |
| **254 серия** |

**254**

*Събиране и*

*контейнерни обекти*

При програмирането обикновено се занимавате не само с отделни обекти, но и с *колекциониране*

*ции* на обекти. Търсите в колекции, за да намерите обект, който съответства на

стриктни критерии (като обект на списание, съдържащ определена статия); вие сортирате

колекции за по-нататъшна обработка или визуално представяне; филтрирате колекции към

включва или изключва определени елементи; и така нататък. Всички тези операции и подобни

големите, зависят от обектите, които са достъпни в колекции.

Ruby представлява колекции от обекти, като ги поставя вътре в *контейнерни* обекти.

В Ruby два вградени класа доминират пейзажа на контейнер-обект: *масиви* и

*хешове* . Ще започнем тази глава, като разгледаме класовете Array и Hash: първо в com-

съпоставяне помежду си, за да се установи цялостно разбиране и след това поотделно.

Ще разгледаме два други класа: Range и Set. Гамите са малко хибридни: те

работят отчасти като булеви филтри (в смисъл, че можете да извършите истински / фалшив тест като

***Тази глава обхваща***

Последователно подредени колекции с масиви

Ключови колекции с хешове

Тестове за включване и членство с диапазони

Уникални, неподредени колекции с комплекти

Именувани аргументи, използващи хеш синтаксис

|  |
| --- |
| **Страница 255** |

**255**

***Масиви и хешове в сравнение***

до това дали дадена стойност се намира в даден диапазон), но също така, в някои контексти, като

лекции. Комплектите са колекции през и през. Единствената причина класът Set

изисква специално въведение е, че не е основен клас Ruby; това е стандартна библиотека

клас и въпреки че не разглеждаме много от тези, които са задълбочени в тази книга, декорите падат

добре на място до масиви и хешове и заслужават нашето внимание.

Докато четете тази глава, имайте предвид, че тя представлява първо преминаване през a

вид мега-тема, която ще посетим и в следващата глава. Ruby прилага колекционерски

главно чрез техниката на дефиниране на класове, които се смесват в Enumerable

модул. Този модул ви дава пакетна сделка за методи, които сортират, пресяват, филтрират, броят,

и трансформира колекции. В тази глава ще разгледаме преди всичко какво можете да направите

с основните групи за събиране на *други* , отколкото да ползват техните Enumerable природата.

Глава 10 ще се занимава директно с Enumerable и как се използва. Ще разгледаме достатъчно

за изброяване тук, за да заредим тази глава и след това ще се върнем към нея в

следващият.

И накрая, тази глава включва връщане към глава 2. В тази глава разгледахме

в дълбочина в списъка с параметри на метода и аргументи и в това как се свързват аргументите

параметри. След като разгледаме по-отблизо хешовете, ще попълним празнина в глава 2 от

разглеждайки *именуваните аргументи* на Ruby , които използват хеш синтаксис.

Освен това имайте предвид, че колекциите са сами по себе си обекти. Изпращате им съобщения

мъдреци, присвоявайте ги на променливи и т.н., по обичайния обект. Те просто имат

допълнително измерение, отвъд скалара.

***9.1***

***Масиви и хешове в сравнение***

Един *масив* е подредена съвкупност от обекти- *нареди* в смисъл, че можете да изберете

обекти от колекцията въз основа на последователен, последователен числов индекс. Вие ще

забелязахме, че вече сме използвали масиви в някои от примерите по-рано в

Книга. Трудно е *да не* се използват масиви в Ruby. Задачата на масива е да съхранява други обекти. Всякакви

обект може да се съхранява в масив, включително други масиви, хешове, манипулатори на файлове, класове,

вярно и невярно ... изобщо всеки обект. Съдържанието на масив винаги остава в същото

поръчка, освен ако изрично не премествате обекти (или не ги добавяте или премахвате).

*Хешовете* в последните версии на Ruby също са подредени колекции - и това е голямо

промяна от предишните версии, където хешовете са неподредени (в смисъл, че те

нямат представа какъв е техният първи, последен или *n* -ти елемент). Хешовете съхраняват обекти по двойки,

всяка двойка, състояща се от *ключ* и *стойност* . Извличате стойност с помощта на ключа.

Хешовете запомнят реда, в който са били вмъкнати ключовете им; това е редът

който хешът се преиграва за вас, ако прегледате двойките в него или отпечатате a

низово представяне на екрана.

Всеки Ruby обект може да служи като хеш ключ и / или стойност, но ключовете са уникални за

хеш: можете да имате само една двойка ключ / стойност за всеки даден ключ. Хешове (или подобни данни

видове съхранение) понякога се наричат *речници* или *асоциативни масиви* на други езици.

Те предлагат изключително - понякога изненадващо - мощен начин за съхранение и

извличане на данни.

|  |
| --- |
| **Страница 256** |

**256**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

Масивите и хешовете са тясно свързани. Масивът в известен смисъл е хеш, където

ключовете са последователни цели числа. Хешовете са в известен смисъл масиви, където индексите

могат да бъдат всичко, а не само цели числа. Ако все пак използвате последователни цели числа като хеш

ключовете, масивите и хешовете започват да се държат по подобен начин, когато правите справки:

масив = ["рубин", "диамант", "изумруд"]

хеш = {0 =>"рубин", 1 =>"диамант", 2 =>"изумруд"}

поставя масив [0] # рубин

поставя хеш [0] # рубин

Дори и да не използвате цели числа, хешовете показват вид свойство „метаиндекс“, базирано

върху факта, че имат определен брой двойки ключ / стойност и че тези двойки могат

да се отчитат последователно. Можете да видите това свойство в действие, като преминете

хеш с метода with\_index, който дава противоположна стойност на блока заедно

с ключа и стойността:

хеш = {"червен" =>"рубин", "бял" =>"диамант", "зелен" =>"изумруд"}

hash.each.with\_index {| (ключ, стойност), i |

поставя „Pair # {i} is: # {key} / # {value}“

}

Резултатът от този кодов фрагмент е

Двойка 0 е: червено / рубинено

Двойка 1 е: бял / диамант

Двойка 2 е: зелено / изумрудено

На *индекса* е брояч число, поддържани като двойките минават. Двойките са действителните

съдържание на хеш.

СЪВЕТ Скобите в блоковите параметри (ключ, стойност) служат за разделяне

освен масив. Всяка двойка ключ / стойност идва в блока като масив от два елемента

моменти. Ако параметрите бяха ключ, стойност, i, тогава ключът на параметъра би

в крайна сметка обвързан с целия масив [ключ, стойност]; стойността ще бъде обвързана с

индекс; и щях да съм нула. Това очевидно не е това, което искате. Родителят-

теоретичното групиране на (ключ, стойност) е сигнал, че искате масивът да бъде изключен

отчитани през тези два параметъра, елемент по елемент.

Общите преобразувания между масиви и хешове са често срещани. Някои такива

версиите са автоматични: ако извършвате определени операции по избор или извличане на

двойки от хеш, ще получите обратно масив. Други преобразувания изискват изрични инструкции

като превръщането на плосък масив ([1,2,3,4]) в хеш ({1 => 2, 3 => 4}). Ще видите

добро количество назад и напред между тези два класа на събиране, както тук в

тази глава и в много Ruby код.

В следващите два раздела ще разгледаме масивите и хешовете в дълбочина. Нека започнем с масиви.

***9.2***

***Работа с колекции с масиви***

Масивите са начинът за обработка на колекции от обекти. Ще сложим масиви

чрез техните стъпки в този раздел: ще разгледаме разнообразните техники, налични за

|  |
| --- |
| **Страница 257** |

**257**

***Работа с колекции с масиви***

създаване на масиви; как да вмъкнете, извлечете и премахнете елементи от масив; комбиниране на масиви

един с друг; трансформиране на масиви (например изравняване на вложен масив в

едномерен масив); и заявка на масиви за техните свойства и състояние.

***9.2.1***

***Създаване на нов масив***

Можете да създадете масив по един от четирите начина:

С метода Array.new

С конструктора на литерален масив (квадратни скоби)

С метод от най-високо ниво, наречен Array

Със специалните нотации% w {...} и% i {...}

Ще видите всички тези техники при силно въртене в кода на Ruby, така че всички те си струват

знаейки. Ще разгледаме всеки по ред.

А RRAY . НОВО

Новият метод за клас масив работи по обичайния начин:

a = Array.new

След това можете да добавяте обекти към масива, използвайки техники, които ще разгледаме по-късно.

Array.new ви позволява да посочите размера на масива и, ако желаете, да инициализирате неговата

палатки. Ето един обмен на irb, който илюстрира и двете възможности:

>> Array.new (3)

=> [нула, нула, нула]

>> Array.new (3, "abc")

=> ["abc", "abc", "abc"]

Ако дадете един аргумент на Array.new B , ще получите масив от размера, който сте поискали,

с всички елементи, зададени на нула. Ако дадете два аргумента c , ще получите масив от размера

сте поискали, като всеки елемент е инициализиран, за да съдържа втория аргумент.

Можете дори да предоставите кодов блок на Array.new. В този случай елементите на

масива се инициализират чрез повтарящи се извиквания към блока:

>> n = 0

=> 0

>> Array.new (3) {n + = 1; n \* 10}

=> [10, 20, 30]

В този пример новият масив има размер 3. Всеки от трите елемента е зададен на

върната стойност на кодовия блок. Кодът в блока Б , изпълнен три пъти, предлага

умножава стойностите 10, 20 и 30 - и това са началните стойности в масива c .

ПРЕДУПРЕЖДЕНИЕ Когато инициализирате множество елементи от масив, като използвате втори

аргумент на Array.new - както в Array.new (3, "abc") - всички елементи на

масив се инициализират към същия обект. Ако направите a = Array.new (3, "abc");

a [0] <<"def"; поставя [1], ще откриете, че вторият елемент от масива е

сега "abcdef", въпреки че сте добавили "def" към първия елемент. Това е

защото първата и втората позиция в масива съдържат един и същ низ

б

° С

б

° С

|  |
| --- |
| **258 серия** |

**258**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

обект, а не два различни низа, които се случват и на двата, се състоят от "abc". За да

изяде масив, който вмъква различен низ "abc" във всеки слот, трябва да използвате

Array.new (3) {"abc"}. Кодовият блок се изпълнява три пъти, всеки път, когато

привличане на нов низ (същите символи, различен низ обект).

Предварителната инициализация на масиви не винаги е необходима, тъй като масивите ви растат, докато добавяте елементите

моменти към тях. Но ако и кога имате нужда от тази функционалност - и / или ако я виждате в

използвайте и искате да го разберете - той е там.

Т ИЗПЪЛНИТЕЛ НА ЛИТЕРАЛЕН МАСИВ : []

Вторият начин за създаване на масив е чрез използване на *литералния конструктор на масив* [] (квадрат

скоби):

a = []

Когато създавате масив с литералния конструктор, можете да поставите обекти в

масив едновременно:

a = [1,2, „три“, 4, []]

Забележете, че последният елемент в този масив е друг масив. Това е напълно законно;

можете да влагате масиви на толкова нива, колкото желаете.

Квадратните скоби могат да означават много различни неща в Ruby: изграждане на масив,

индексиране на масив (както и индексиране на низове и хеш), класове символи в редовно

изрази, разделители в% q [] - стилова нотация, дори извикването на анонимен

функция. Можете да направите първоначално разделяне на различните употреби на квадратни скоби чрез

разграничаване на случаите, когато те са семантична конструкция от случаите, когато те са името

на метод. Струва си да се упражнявате на няколко примера като този, за да усетите начина

квадратните скоби се играят в различен контекст:

[1,2,3] [1]

Сега обратно към създаването на масив.

T HE А МЕТОД RRAY

Третият начин за създаване на масив е с *метод* (въпреки че изглежда като клас

име!) наречен Array. Както знаете от това, че сте виждали методите Integer и Float,

законно е да се дефинират методи, чиито имена започват с главни букви. Тези имена изглеждат

точно като константите, а в самия Ruby самите методи с главни букви обикновено имат

същите имена като класовете, с които са свързани.

Някои по-вградени методи, които започват с главни букви

В допълнение към метода Array и двата метода за преобразуване в стил главни букви

вече сте виждали (Integer и Float, „придирчивите“ версии на to\_i и to\_f),

Ruby предоставя няколко други метода от най-високо ниво, чиито имена изглеждат като имена на класове:

Сложни, рационални и струнни. Във всеки случай методът връща обект на

клас, който изглежда името му.

**Индекс 1 върху масив [1,2,3]**

|  |
| --- |
| **259 серия** |

**259**

***Работа с колекции с масиви***

Методът Array създава масив от единичния си аргумент. Ако аргументът обект

има дефиниран метод to\_ary, след което Array извиква този метод върху обекта, който трябва да генерира

масив. (Не забравяйте, че to\_ary е методът за преобразуване на масив от квазитипичен тип.) Ако

няма метод to\_ary, той се опитва да извика to\_a. Ако и to\_a не е дефинирано, Array се увива

обекта в масив и връща, че:

>> низ = "низ"

=>"Низ"

>> низ.отговорете\_ към? (: към\_я)

=> невярно

>> низ.отговори\_то? (: до\_а)

=> невярно

>> Масив (низ)

=> ["Низ"]

>> def string.to\_a

>> разделяне (//)

>> край

=> нула

>> Масив (низ)

=> ["A", "", "s", "t", "r", "i", "n", "g"]

В този пример първият опит за стартиране на Array на низа B води до едноелементен

масив, където един елемент е низът. Това е така, защото низовете нямат нито to\_ary

нито метод to\_a. Но след като to\_a е дефиниран за низа c , резултатът от извикването

Масивът е различен: сега той изпълнява метода to\_a и го използва като своя върната стойност. (The

методът to\_a разделя низа на отделни символи.)

Сред различните конструктори на масиви литералът [] е най-често срещаният, следван

от Array.new и метода Array, в този ред. Но всеки има своето място. Буквалният

конструкторът е най-лаконичният; когато научите какво означава, това ясно съобщава

„Масив“, когато го видите. Методът Array се ограничава от необходимостта да съществува

наличен метод to\_ary или to\_a.

Т ОН % W И % W КОНСТРУКТОРИ ЗА МАСИВ

Като специално разпределение, което ви помага да създавате масиви от низове, Ruby предоставя% w

оператор, много в същото семейство като% q операторите в стил, които вече сте виждали, това

***(продължение)***

Методът String е обвивка около to\_s, което означава String (obj) е еквивалентен

до obj.to\_s. Комплексът и Rational съответстват на методите to\_c и to\_r

налични за числа и низове - с изключение на сложни и рационални, като Integer и

Float, са суетливи: те не приемат любезно нечислови низове. "abc" .to\_c дава

вие (0 + 0i), но Complex ("abc") повдига ArgumentError, и Rational и to\_r

държат се по подобен начин.

Тук не покриваме рационални и комплексни числа, но сега знаете как да генерирате

изтрийте ги, в случай че те представляват интерес!

б

° С

|  |
| --- |
| **Страница 260** |

**260**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

автоматично генерира масив от низове от разделените с интервал низове, които сте поставили

вътре в него. Можете да видите как работи, като го използвате в irb и погледнете резултата:

>>% w {Дейвид А. Блек}

=> ["Дейвид", "А.", "Черен"]

Ако някой низ в списъка съдържа пробел, трябва да избягате от този символ

ter с обратна наклонена черта:

>>% w {Дейвид \ А. \ Блек е рубист. }

=> [„Дейвид А. Блек“, „е“, „а“, „Рубист.“]

Низовете в списъка се анализират като низове с единични кавички. Но ако имате нужда от двойно

цитираните низове, можете да използвате% W вместо% w:

>>% W {Дейвид е на # {2014 - 1959} години. }

=> ["Дейвид", "е", "55", "години", "стар."]

T ОН % I И % I КОНСТРУКТОРИ ЗА МАСИВ

Точно както можете да създавате масиви от низове, използвайки% w и% W, можете също да създавате масиви от

символи, използващи% i и% I. Разграничението i / I, подобно на разграничението w / W, се отнася до

тълкуване на единични срещу двойни кавички:

>>% i {abc}

=> [: a,: b,: c]

>> d = "Дейвид"

=>"Дейвид"

>>% I {"# {d}"}

=> [: "\" Дейвид \ ""]

Нека да пристъпим сега към въпроса за обработката на масивни елементи.

В **try\_convert** семейството на методи

Всеки от няколкото вградени класа в Ruby има метод на клас, наречен try\_convert,

което винаги взема един аргумент. try\_convert търси метод за преобразуване на

аргументния обект. Ако методът съществува, той се извиква; ако не, опитайте\_конвертирате връща

нула. Ако методът на преобразуване връща обект от клас, различен от класа

коя конверсия се опитва, това е фатална грешка (TypeError).

Класовете, изпълняващи try\_convert (и имената на необходимото преобразуване

методи) са Array (to\_ary), Hash (to\_hash), IO (to\_io), Regexp (to\_regexp),

и String (to\_str). Ето пример за обект, поставящ Array.try\_convert

през неговите крачки. (Другите методи try\_convert работят по подобен начин.)

>> obj = Object.new

=> # <Обект: 0x000001028033a8>

>> Array.try\_convert (obj)

=> нула

>> def obj.to\_ary

>> [1,2,3]

>> край

|  |
| --- |
| **261 серия** |

**261**

***Работа с колекции с масиви***

***9.2.2***

***Вмъкване, извличане и премахване на елементи от масив***

Масивът е числово подредена колекция. Всеки обект, който добавите към масива, отива

в началото, в края или някъде по средата. Най-общата технология

nique за вмъкване на един или повече елементи в масив е методът на задаване [] =

(квадратни скоби и знак за равенство). Това изглежда странно като име на метод в средата

на параграф като този, но благодарение на синтактичния си захарен еквивалент, [] = работи

плавно на практика.

За да използвате [] =, трябва да знаете, че всеки елемент (или елемент) в масив заема a

номерирана позиция. Първият елемент е в позиция *нула* (не позиция *първа* ). Секундата

елемент е в позиция едно и т.н.

За да вмъкнете елемент с метода [] = - използвайки синтактичната захар, която позволява

да избягвате обичайната точка за извикване на методи - направете това:

a = []

a [0] = "първи"

Вторият ред е синтактична захар за a. [] = (0, "първи"). В този пример се озовавате

с масив от един елемент, чийто първи (и единствен) елемент е низът „first“.

Когато имате обекти в масив, можете да ги извлечете с помощта на []

метод, който е еквивалентът на гетъра на метода [] = setter:

a = [1,2,3,4,5]

па [2]

В този случай вторият ред е синтактична захар за a. [] (2). Искате третото

елемент (въз основа на индексирането с нулев произход), което е цялото число 3.

Можете също да изпълнявате тези методи за получаване и задаване на повече от един елемент в

време.

S настройвате или получаване ПОВЕЧЕ ОТ ЕДИН елемент на масива в момент

Ако дадете или Array # [], или Array # [] = (методът get или set) втори аргумент,

той се третира като дължина - редица елементи за задаване или извличане. В случай че

извличане, резултатите се връщат в нов масив.

Ето малко диалогов диалог, илюстриращ многоелементните операции на [] и

[] = методи:

***(продължение)***

=>: to\_ary

>> Array.try\_convert (obj)

=> [1, 2, 3]

>> def obj.to\_ary

>>"Не е масив!"

>> край

=>: to\_ary

>> Array.try\_convert (obj)

TypeError: не може да конвертира обект в масив (обект # to\_ary дава String

|  |
| --- |
| **262 серия** |

**262**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

>> a = ["червено", "оранжево", "жълто", "лилаво", "сиво", "индиго", "виолетово"]

=> ["червено", "оранжево", "жълто", "лилаво", "сиво", "индиго", "виолетово"]

>> а [3,2]

=> ["лилаво", "сиво"]

>> a [3,2] = "зелено", "синьо"

=> ["зелено", "синьо"]

>> а

=> ["червено", "оранжево", "жълто", "зелено", "синьо", "индиго", "виолетово"]

След инициализиране на масива a, ние хващаме два елемента B , започвайки от индекс 3 (четвъртия

елемент) на a. Двата елемента се връщат в масив. След това задаваме четвъртия и

пети елементи, използвайки [3,2] нотация c , към нови стойности; тези нови стойности са тогава

присъства в целия масив d, когато поискаме от irb да го покаже в края.

Има синоним на метода []: slice. Подобно на [], среза взема два аргумента:

начален индекс и незадължителна дължина. В допълнение, метод, наречен слайс! премахва

нарязаните елементи за постоянно от масива.

Друга техника за извличане на множество елементи от масив е values\_at

метод. values\_at приема един или повече аргументи, представляващи индекси и връща

масив, състоящ се от стойностите, съхранявани при тези индекси в масива на приемника:

масив = ["the", "dog", "изяде", "the", "cat"]

статии = array.values\_at (0,3)

p статии

Можете да извършвате операции по задаване и получаване на елементи навсякъде в масив. Но опера-

Най-често се появяват специфично засягащи началото и края на масивите.

Съответно съществуват редица методи със специалната цел за добавяне на елементи към или

премахването им от началото или края на масив, както сега ще видите.

S СПЕЦИАЛНИ МЕТОДИ ЗА МАНИПУЛИРАНЕ НА НАЧАЛИТЕ И КРАЙИТЕ НА МАСИВИТЕ

За да добавите обект в началото на масив, можете да използвате unshift. След тази операция

a = [1,2,3,4]

a.unshift (0)

масивът a сега изглежда така: [0,1,2,3,4].

За да добавите обект в края на масив, използвате push. Правейки това

a = [1,2,3,4]

а.бутане (5)

води до масив а, който има пети елемент: [1,2,3,4,5].

Можете също да използвате метод, наречен << (два знака по-малко от), който поставя обект

в края на масива. Подобно на много методи, чиито имена приличат на оператори, <<

предлага синтактичната захар от използването като инфикс оператор. Следващият код добавя 5 като

петият елемент на a, точно като операцията за натискане в последния пример:

a = [1,2,3,4]

a << 5

б

° С

**Синтактична захар за**

**а. [] = (3,2, ["зелено", "синьо"])**

д

**Изход:**

**[„The“, „the“]**

|  |
| --- |
| **263 серия** |

**263**

***Работа с колекции с масиви***

Методите << и push се различават по това, че push може да отнеме повече от един аргумент.

Кодът

a = [1,2,3,4,5]

а.бутане (6,7,8)

добавя три елемента към a, което води до [1,2,3,4,5,6,7,8].

Съответстващи на отместване и натискане, но с обратен ефект са shift и pop.

shift премахва един обект от началото на масива (като по този начин "измества"

останалите обекти вляво от една позиция), а pop премахва обект от края

на масива. shift и pop и двете връщат елемента на масива, който са премахнали, като този

пример показва:

a = [1,2,3,4,5]

print "Оригиналният масив:"

па

изскочи = a.pop

print "Изскачащият елемент:"

поставя изскача

print "Новото състояние на масива:"

па

изместен = a.shift

print "Преместеният елемент:"

поставя изместен

print "Новото състояние на масива:"

па

Изходът е

Оригиналният масив: [1, 2, 3, 4, 5]

Изскачащият елемент: 5

Новото състояние на масива: [1, 2, 3, 4]

Преместеният елемент: 1

Новото състояние на масива: [2, 3, 4]

Както можете да видите от текущия коментар в изхода, възвръщаемата стойност на

pop и shift е елементът, който е премахнат от масива. Масивът е перма-

неотдавна променени от тези операции; елементите се премахват, а не само се препращат

на или заловени.

shift и pop може да премахне повече от един елемент наведнъж. Просто предоставете

ger аргумент и този брой елементи ще бъдат премахнати. Премахнатите елементи ще

да се върне като масив (дори ако номерът, който предоставяте, е 1):

>> a =% w {една две три четири пет}

=> ["едно", "две", "три", "четири", "пет"]

>> a.pop (2)

=> ["четири", "пет"]

>> а

=> ["едно", "две", "три"]

>> a.shift (2)

=> ["един", "два"]

>> а

=> [„три“]

|  |
| --- |
| **264 серия** |

**264**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

След това ще преминем от манипулиране на един масив към търсене на начини за комбиниране на два или

още масиви.

***9.2.3***

***Комбиниране на масиви с други масиви***

Няколко метода ви позволяват да комбинирате множество масиви по различни начини - нещо

това се оказва често и полезно, когато започнете да манипулирате много данни в

списъци. Не забравяйте, че във всеки случай, въпреки че имате работа с двама (или повече)

масиви, един масив винаги е получателят на съобщението. Другите масиви, участващи в

операцията са аргументи на метода.

За да добавите съдържанието на един масив към друг масив, можете да използвате concat:

>> [1,2,3] .concat ([4,5,6])

=> [1, 2, 3, 4, 5, 6]

Имайте предвид, че concat се различава по важен начин от push. Опитайте да замените concat с

натиснете примера и вижте какво се случва.

concat променя за постоянно съдържанието на своя приемник. Ако искате да комбинирате

два масива в трети, нов масив, можете да го направите с метода +:

>> a = [1,2,3]

=> [1, 2, 3]

>> b = a + [4,5,6]

=> [1, 2, 3, 4, 5, 6]

>> а

=> [1, 2, 3]

Получателят на съобщението + - в този случай масивът a - остава непроменен от

операция (както irb ви казва B ).

Друг полезен метод за комбиниране на масиви, поне при доста либерална интерпретация

понятието „комбиниране“ е заместител. Както подсказва името, замени замества

съдържанието на един масив със съдържанието на друг:

>> a = [1,2,3]

=> [1, 2, 3]

>> a.replace ([4,5,6])

=> [4, 5, 6]

>> а

=> [4, 5, 6]

Оригиналното съдържание на a е изчезнало, заменено B със съдържанието на аргумента

масив [4,5,6]. Не забравяйте, че операцията по замяна е различна от преназначаването. Ако

правиш това

a = [1,2,3]

a = [4,5,6]

второто задание кара променливата a да се отнася до съвсем различен обект на масив

от първия. Това не е същото като заместването на елементите на един и *същ* обект на масив. Това

започва да има значение, по-специално, когато имате друга променлива, която се отнася до

nal масив, както в този код:

б

б

|  |
| --- |
| **265 серия** |

**265**

***Работа с колекции с масиви***

>> a = [1,2,3]

=> [1, 2, 3]

>> b = a

=> [1, 2, 3]

>> a.replace ([4,5,6])

=> [4, 5, 6]

>> б

=> [4, 5, 6]

>> a = [7,8,9]

=> [7, 8, 9]

>> б

=> [4, 5, 6]

След като изпълните заданието от a до b B , замествайки съдържанието на средство

заменили сте съдържанието на b c , защото двете променливи се отнасят до един и същ масив.

Но когато преназначавате на d , прекъсвате обвързването между a и масива; a и b

сега се обърнете към различни обекти на масив: b към същия стар масив e , a към нов.

В допълнение към комбинирането на множество масиви, можете също да трансформирате отделни масиви

към различни форми. След това ще разгледаме техниките по този ред.

***9.2.4***

***Трансформации на масиви***

Полезна трансформация на масив се изравнява, което прави разгъване на вътрешни масиви.

Можете да посочите колко нива на изравняване искате, като по подразбиране е пълно

разгъване.

Ето тройно вложен масив, който се изравнява от различни нива:

>> масив = [1,2, [3,4, [5], [6, [7,8]]]]

=> [1, 2, [3, 4, [5], [6, [7, 8]]]]

>> array.flatten

=> [1, 2, 3, 4, 5, 6, 7, 8]

>> array.flatten (1)

=> [1, 2, 3, 4, [5], [6, [7, 8]]]

>> array.flatten (2)

=> [1, 2, 3, 4, 5, 6, [7, 8]]

Има и изравняване на място! метод, който прави промяната за постоянно в

масива.

Друг метод за преобразуване на масив е обратен, който прави точно това, което казва:

>> [1,2,3,4] .обръщане

=> [4, 3, 2, 1]

Подобно на нивото си на низ, Array # reverse също има версия bang (!), Която перма-

нежно обръща масива, който го извиква.

Друг важен метод за преобразуване на масив е join. Връщаната стойност на присъединяване

не е масив, а низ, състоящ се от низово представяне на всички елементи на

масивът, нанизан заедно:

>> ["abc", "def", 123] .join

=>"abcdef123"

б

° С

д

д

**Изравнява напълно**

**Изравнява от**

**едно ниво**

**Изравнява от**

**две нива**

|  |
| --- |
| **266 серия** |

**266**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

join приема незадължителен аргумент; ако е даден, аргументът се поставя между всяка двойка

от елементи:

>> ["abc", "def", 123] .join (",")

=>"abc, def, 123"

Присъединяването със запетаи (или интервал със запетая, както в последния пример) е доста често срещано

операция.

В чудесен пример за стила на дизайн на Ruby има друг начин да се присъедините към масив:

\* метод. Изглежда, че умножавате масива по низ, но всъщност сте

формиране на операция за присъединяване:

>> a =% w {една две три}

=> ["едно", "две", "три"]

>> а \* "-"

=>"един-два-три"

Можете също да трансформирате масив с uniq. uniq ви дава нов масив, състоящ се от

елементи на оригиналния масив с премахнати всички дублиращи се елементи:

>> [1,2,3,1,4,3,5,1] .uniq

=> [1, 2, 3, 4, 5]

Състоянието на дубликат се определя чрез тестване на двойки елементи с метода ==. Всякакви

два елемента, за които тестът == връща true, се считат за дубликати на всеки

други. uniq също има версия на взрив, uniq !, която премахва дубликати за постоянно

от оригиналния масив.

Понякога имате масив, който включва едно или повече появявания на nil и

искате да се отървете от тях. Може например да имате масив от пощенските кодове

на всички членове на организация. Но може би някои от тях нямат ZIP

кодове. Ако искате да направите хистограма на пощенските кодове, бихте искали да се отървете от

първо нулеви.

Можете да направите това с компактния метод. Този метод връща нов идентификатор на масив

cal към оригиналния масив, с изключение на това, че всички случаи на nil са премахнати:

>> zip\_codes = ["06511", "08902", "08902", nil, "10027",

"08902", нула, "06511"]

=> ["06511", "08902", "08902", нула, "10027", "08902", нула, "06511"]

>> zip\_codes.compact

=> ["06511", "08902", "08902", "10027", "08902", "06511"]

Отново има налична версия (компактна!).

В допълнение към трансформирането на масиви по различни начини, можете да заявявате масиви по вари-

нашите критерии.

***9.2.5***

***Заявка за масив***

Няколко метода ви позволяват да събирате информация за масив от масива.

Таблица 9.1 обобщава някои от тях. Други методи за заявки произтичат от включването на Array

на модула Enumerable и следователно ще се появи в следващата глава.

|  |
| --- |
| **Страница 267** |

**267**

***Хеш***

В случаите на first, last и sample, ако не предадете аргумент, получавате справедливо

един елемент обратно. Ако все пак предадете аргумент *n* , ще получите масив от *n* елемента

обратно - дори ако *n* е 1.

Следва: хешове. Те са ни прекосили пътя тук-там по пътя и сега

ще ги разгледаме подробно.

***9.3***

***Хеш***

Подобно на масив, хешът е колекция от обекти. Хешът се състои от двойки ключ / стойност,

където всеки ключ и всяка стойност могат да бъдат всеки Ruby обект. Хешовете ви позволяват да извършвате търсене

операции, базирани на ключове. В допълнение към простото извличане на стойност въз основа на ключ, можете също

извършват по-сложни операции за филтриране и подбор.

Типично използване на хеш е съхраняването на пълни низове заедно с техните съкращения.

Ето хеш, съдържащ селекция от имена и двубуквени съкращения на държавата

с някакъв код, който го упражнява. Операторът => свързва ключ вляво с

стойност, съответстваща на него вдясно:

state\_hash = {"Кънектикът" =>"CT",

"Делауеър" =>"DE",

"Ню Джърси" =>"Ню Джърси",

"Вирджиния" =>"VA"}

print "Въведете името на държава:"

състояние = gets.chomp

съкращение = състояние\_хаш [състояние]

поставя „Съкращението е # {abbr}.“

Когато стартирате този фрагмент (ако приемем, че въведете едно от състоянията, дефинирани в хеша),

виждате съкращението.

Хешовете запомнят реда на вмъкване на ключовете им. Редът за вмъкване не винаги е

страшно важно; една от достойнствата на хеш е, че той осигурява бързо търсене в

по-добро от линейно време. И в много случаи елементите се добавят към хешове без никакви конкретни

голям ред; нареждането, ако има такова, идва по-късно, когато искате да превърнете, да речем, хеш от имена

и рождени дни, които сте създали с течение на времето, хронологично или по азбучен ред

Таблица 9.1 Резюме на често срещаните методи за заявки в масив

Име на метод / Примерен разговор

Значение

a.size (синоним: дължина)

Брой елементи в масива

а.празен?

Вярно, ако a е празен масив; false, ако има някакви елементи

a.include? (артикул)

Вярно, ако масивът включва елементи; false в противен случай

a.count (артикул)

Брой появявания на елемент в масива

а.първа (n = 1)

Първи *n* елемента на масива

a. последен (n = 1)

Последни *n* елемента от масив

а.проба (n = 1)

*n* произволни елементи от масива

|  |
| --- |
| **268 серия** |

**268**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

сортиран масив. И все пак, колкото и полезно да е или не е полезно за тях да го правят, хешове

запомнете техния ред за вмъкване на ключове и спазвайте този ред, когато повторите

или ги разгледайте.

Подобно на масивите, хешовете могат да се създават по няколко различни начина.

***9.3.1***

***Създаване на нов хеш***

Има четири начина за създаване на хеш:

С литералния конструктор (къдрави скоби)

С метода Hash.new

С метода Hash. [] (Метод на клас на квадратна скоба на Hash)

С метода от най-високо ниво, чието име е Hash

Тези техники за създаване на хеш са изброени тук, доколкото е възможно, в низходящ ред

ред на честотата. С други думи, ще започнем с най-често срещаната техника

и продължете от там.

C ПОВТОРЕНЕ НА ЛИТЕРАЛЕН ХАШ

Когато въведете буквален хеш в къдравите скоби, отделяте ключовете от стойностите

с оператора => (освен ако не използвате специалния синтаксис {key: value} за символ

клавиши). След всяка пълна двойка ключ / стойност поставяте запетая (с изключение на последната двойка,

където е по избор).

Буквалният конструктор на хеш е удобен, когато имате стойности, които искате да хеширате

това няма да се промени; ще ги въведете в програмния файл веднъж и ще се обърнете към

ги от програмата. Държавните съкращения са добър пример.

Можете да използвате литералния конструктор на хеш, за да създадете празен хеш:

h = {}

Вероятно бихте искали да добавите елементи към празния хеш в даден момент; техники за

това ще бъде направено в раздел 9.3.2.

Вторият начин за създаване на хеш е с традиционния нов конструктор.

T HE H ASH . НОВ КОНСТРУКТОР

Hash.new създава празен хеш. Но ако предоставите аргумент на Hash.new, това е така

третира се като стойност по подразбиране за несъществуващи хеш ключове. Ще се върнем към въпроса

стойности по подразбиране и някои камбани на Hash.new, след като разгледахме ключ /

вмъкване и извличане на стойност.

Т О Н ПЕПЕЛ . [] МЕТОД НА КЛАСА

Третият начин за създаване на хеш включва друг метод на клас от клас Hash: методът

[] (квадратни скоби). Този метод взема разделен със запетая списък с елементи и, ако приемем

има четен брой аргументи, третира ги като редуващи се ключове и стойности, които

използва се за конструиране на хеш. Благодарение на синтактичната захар на Ruby можете да изложите аргументите

да [] директно в скобите и да ги разпределите с точката за извикване на метод:

>> Hash [„Кънектикът“, „CT“, „Делауеър“, „DE“]

=> {"Кънектикът" =>"CT", "Делауеър" =>"DE"}

|  |
| --- |
| **269 ​​серия** |

**269**

***Хеш***

Ако предоставите нечетен брой аргументи, възниква фатална грешка, тъй като е нечетен

брой аргументи не могат да бъдат съпоставени с поредица двойки ключ / стойност. Ти обаче

може да премине в масив от масиви, където всеки подмасив се състои от два елемента. Хеш. []

ще използва вътрешните масиви като двойки ключ / стойност:

>> Хеш [[[1,2], [3,4], [5,6]]]

=> {1 => 2, 3 => 4, 5 => 6}

Можете също така да предадете всичко, което има метод, наречен to\_hash. Новият хеш ще бъде

резултатът от извикването на този метод.

Друга техника за създаване на хеш включва метода на най-високото ниво на хеш.

T HE Н МЕТОД ПЕПЕЛ

Методът на Hash има леко идиосинкратично поведение. Ако се извика с празен масив ([])

или нула, връща празен хеш. В противен случай той извиква to\_hash на своя единствен аргумент. Ако

аргументът няма метод to\_hash, повдига се фатална грешка (TypeError).

Вече видяхте редица начини за създаване на хешове. Не забравяйте, че те са вътре

приблизителен низходящ ред по честота. Ще видите много по-буквални хеш-кон-

конструктори и обаждания към Hash.new, отколкото вие, останалите от представените техники. Все още,

добре е да знаете какво се предлага и как работят различните техники.

Сега нека се обърнем към въпроса за манипулиране на съдържанието на хеш. Ще последваме

почти същия път, както направихме с масиви, гледайки операцията за вмъкване и извличане

комбиниране на хешове с други хешове, хеш трансформации и заявки

хешове. По пътя ще разгледаме отделно и задаването на стойности по подразбиране за

несъществуващи хеш ключове.

***9.3.2***

***Вмъкване, извличане и премахване на двойки хеш***

Както ще видите, хешовете имат много общо с масивите, когато става въпрос за get- и

операции в стил-набор - въпреки че има някои важни разлики и някои технически

niques, които са специфични за всеки.

А DDING КЛЮЧ / такава двойка в разбъркващата

За да добавите двойка ключ / стойност към хеш, използвате по същество същата техника като за добавяне

добавяне на елемент към масив: [] = метод плюс синтактична захар.

За да добавите състояние към state\_hash, направете това

state\_hash ["Ню Йорк"] = "Ню Йорк"

което е захаризираната версия на това:

state\_hash. [] = ("Ню Йорк", "Ню Йорк")

Можете също да използвате хранилището на синонимни методи за тази операция. магазин отнема две

аргументи (ключ и стойност):

state\_hash.store ("Ню Йорк", "Ню Йорк")

Когато добавяте към хеш, имайте предвид важния принцип, който са ключовете

единствен по рода си. Можете да имате само един запис с даден ключ. Стойностите на хеш не трябва да бъдат

|  |
| --- |
| **270 серия** |

**270**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

единствен по рода си; можете да присвоите една и съща стойност на два или повече клавиша. Но не можете да имате дубли-

кейт ключове.

Ако добавите двойка ключ / стойност към хеш, който вече има запис за ключа, който сте

добавяйки, старият запис се заменя. Ето пример:

h = Hash.new

h ["a"] = 1

h ["a"] = 2

поставя h ["a"]

Този код присвоява две стойности на ключа "a" на хеша h. Второто задание clob-

извежда първия, както показва операторът put, като извежда 2.

Ако преназначите на даден хеш ключ, този ключ все още запазва мястото си във вмъкването

ред на хеш. Промяната в стойността, сдвоена с ключа, не се счита за нова

вмъкване в хеша.

R ПОЛУЧАВАНЕ НА СТОЙНОСТИ ОТ ХАШ

Техниката на работния кон за извличане на хеш стойности е методът []. Например, до

извлечете "CT" от state\_hash и го присвойте на променлива, направете това:

conn\_abbrev = state\_hash [„Кънектикът“]

Използването на хеш ключ прилича много на индексиране на масив - с изключение на това, че индексът (ключът) може

бъде каквото и да е, докато в масив това винаги е цяло число.

Хешовете също имат метод за извличане, който ви дава алтернативен начин за извличане

стойности по ключ:

conn\_abbrev = state\_hash.fetch ("Кънектикът")

fetch се различава от [] по начина, по който се държи, когато го помолите да търси несъществуващ

key: fetch създава изключение, докато [] ви дава нула или по подразбиране сте

посочени (както е обсъдено в следващия раздел). Ако предоставите втори аргумент на хеш,

този аргумент ще бъде върнат, вместо да бъде повдигнато изключение, ако ключът не е

намерен. Например този код

state\_hash.fetch ("Небраска", "Неизвестно състояние")

оценява до низа "Неизвестно състояние".

Можете също така да извлечете стойности за множество ключове в една операция, с values\_at:

two\_states = state\_hash.values\_at ("Ню Джърси", "Делауеър")

Този код връща масив, състоящ се от ["NJ", "DE"] и го присвоява на променливата

two\_states.

Сега, когато имате представа за механиката на навлизане и излизане на информация

хеш, нека кръгнем назад и да разгледаме въпроса за предоставяне на стойност по подразбиране (или по подразбиране

код блок), когато създавате хеш.

**Изход: 2**

|  |
| --- |
| **271 серия** |

**271**

***Хеш***

***9.3.3***

***Посочване на хеш стойности и поведение по подразбиране***

По подразбиране, когато поискате хеш за стойността, съответстваща на несъществуващ ключ, вие

получи нула:

>> h = Hash.new

=> {}

>> h ["няма такъв ключ!"]

=> нула

Но можете да посочите различна стойност по подразбиране, като предоставите аргумент на Hash.new:

>> h = Hash.new (0)

=> {}

>> h ["няма такъв ключ!"]

=> 0

Тук връщаме стойността по подразбиране на хеш, 0, когато използваме несъществуващ ключ. (Можеш

също така задайте по подразбиране на вече съществуващ хеш с метода по подразбиране.)

Важно е да запомните, че каквото и да посочите като стойност по подразбиране, това е какво

получавате, когато посочите *несъществуващ* ключ - и че ключът остава несъществуващ до

вие му присвоявате стойност. С други думи, казването на h [„бла“] не означава, че h сега има

клавиш "бла". Ако искате този ключ в хеша, трябва да го поставите там. Можете да проверите

фактът, че хешът h няма ключове, като го проверява след извършване на несъществуващото

търсене на ключове в последния пример:

>> ч

=> {}

Ако искате препратките към несъществуващи ключове да предизвикат съществуването на ключовете,

можете да уредите това, като предоставите кодов блок на Hash.new. Кодовият блок ще бъде

изпълнява се всеки път, когато се посочва несъществуващ ключ. Два обекта ще бъдат предадени на

block: хеш и (несъществуващ) ключ.

Тази техника ви дава крак във вратата, когато става въпрос за автоматична настройка на ключове

cally, когато се използват за първи път. Това не е най-елегантната или рационализирана техника в

Руби, но работи. Написвате блок, който грабва хеша и ключа, и правите a

задайте операция.

Например, ако искате всеки несъществуващ ключ да бъде добавен към хеша с

стойност 0, създайте своя хеш по този начин:

h = Hash.new {| хеш, ключ | хеш [ключ] = 0}

Когато хешът h бъде изискан да извлече стойността за ключ, който няма, блокът е

изпълнява се с хеш, зададен на самия хеш и ключ, зададен на несъществуващия ключ. И

благодарение на кода в блока, ключът все пак се добавя към хеша със стойността 0.

Като се има предвид това присвояване на нов хеш на h, можете да задействате блока така:

>> h [„нов ключ!“]

=> 0

>> ч

=> {"нов ключ!" => 0}

б

° С

|  |
| --- |
| **272 серия** |

**272**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

Когато се опитате да потърсите ключа "нов ключ!"Б , няма го; но благодарение на блока,

той се добавя със стойността 0. След това, когато поискате irb да ви покаже целия хеш c ,

съдържа автоматично добавената двойка.

Тази техника има много приложения. Позволява ви да правите предположения за това какво има в хеш,

дори да няма нищо, с което да започнете. Той също така ви показва още един аспект на обширния Ruby

репертоар от техники за динамично програмиране и гъвкавостта на хешовете.

Сега ще се обърнем към начините, по които можете да комбинирате хешове помежду си, както направихме с

низове и масиви.

***9.3.4***

***Комбиниране на хешове с други хешове***

Процесът на комбиниране на две хешове в едно се състои от два вкуса: разрушителен

аромат, където първият хеш има добавени двойки ключ / стойност от втория хеш

директно; и неразрушаващия вкус, където се създава нов, трети хеш, който комбинира

съчетава елементите на оригиналните два.

Деструктивната операция се извършва с метода за актуализиране. Записи в

първият хеш се презаписва за постоянно, ако вторият хеш има съответния ключ:

h1 = {"Smith" =>"John",

"Джоунс" =>"Джейн"}

h2 = {"Smith" =>"Jim"}

h1.актуализация (h2)

поставя h1 ["Смит"]

В този пример записът "Smith" на h1 е променен (актуализиран) до стойността, която има

h2. Искате опресняване на вашия хеш, за да отрази съдържанието на втория хеш.

Това е разрушителната версия на комбинирането на хешове.

За да извършите неразрушително комбиниране на две хешове, използвайте метода на сливане,

което ви дава трети хеш и оставя оригинала непроменен:

h1 = {"Smith" =>"John",

"Джоунс" =>"Джейн"}

h2 = {"Smith" =>"Jim"}

h3 = h1.merge (h2)

p h1 [„Смит“]

Тук двойката "Smith" / "John" на h1 не се заменя от двойката "Smith" / "Jim" на h2. Вместо,

създава се нов хеш с двойки от двете други. Този хеш ще изглежда

това, ако се изследва:

{"Smith" =>"Jim", "Jones" =>"Jane"}

Имайте предвид, че h3 трябва да вземе решение: кой от двата записа на Смит трябва да съдържа?

Отговорът е, че когато двете хешове, които се обединяват, споделят ключ, вторият хеш

(h2, в този пример) печели. Стойността на h3 за ключа "Smith" ще бъде "Jim".

Между другото, merge! - взривната версия на merge - е синоним на актуализация. Вие

може да използва всяко име, когато искате да извършите тази операция.

Освен че се комбинират с други хешове, хешовете също могат да бъдат трансформирани

по няколко начина, както ще видите по-нататък.

**Изход: Джим**

**Резултат: Джон**

|  |
| --- |
| **273 серия** |

**273**

***Хеш***

***9.3.5***

***Хеш трансформации***

Можете да извършите няколко трансформации на хешове. *Трансформация* , в този контекст,

означава, че методът се извиква върху хеш и резултатът от операцията (

връщаната стойност на метода) е хеш. В глава 10 ще видите друго филтриране и изберете-

методите за хеширане, които връщат резултатите си в масиви. Тук разглеждаме

хеш-към-хеш операции.

S ИЗБОР И ОТХВЪРЛЯНЕ НА ЕЛЕМЕНТИ ОТ ХАШ

За да извлечете подхеш от съществуващ хеш, използвайте метода select. Двойки ключ / стойност

ще бъдат предадени последователно на кодовия блок, който предоставяте. Всяка двойка, за която

block връща истинска стойност ще бъде включена в резултата хеш:

>> h = хеш [1,2,3,4,5,6]

=> {1 => 2, 3 => 4, 5 => 6}

>> h.изберете {| k, v | k> 1}

=> {3 => 4, 5 => 6}

Отхвърлянето на елементи от хеш работи по обратния начин - тези двойки ключ / стойност за

които блокът връща true са изключени от резултата хеш:

>> h.reject {| k, v | k> 1}

=> {1 => 2}

изберете и отхвърлете има еквиваленти на място (версии, които променят оригиналния хеш

постоянно, вместо да връща нов хеш): изберете! и отхвърлете !. Тези двамата

методи връщат нула, ако хешът не се промени. За да направите операция на място, която се връща

оригиналният ви хеш (дори и да е непроменен), можете да използвате keep\_if и delete\_if.

Аз NVERTING хеш

Hash # инвертира обръща клавишите и стойностите. Стойностите се превръщат в ключове, а ключовете се превръщат в стойности:

>> h = {1 =>"един", 2 =>"два"}

=> {1 =>"един", 2 =>"два"}

>> h.invert

=> {"две" => 2, "една" => 1}

Бъдете внимателни, когато обръщате хешове. Тъй като хеш ключовете са уникални, но стойностите не са,

когато превръщате дублирани стойности в ключове, една от двойките се изхвърля:

>> h = {1 =>"един", 2 =>"повече от 1", 3 =>"повече от 1"}

=> {1 =>"един", 2 =>"повече от 1", 3 =>"повече от 1"}

>> h.invert

=> {"one" => 1, "повече от 1" => 3}

Само една от двете стойности „повече от 1“ може да оцелее като ключ, когато инверсията е

изпълнени; другият се изхвърля. Трябва да инвертирате хеш само когато сте сигурни

стойностите, както и ключовете са уникални.

C НАУЧВАНЕ НА ХАШ

Hash # clear изпразва хеша:

>> {1 =>"един", 2 =>"два"} .clear

=> {}

|  |
| --- |
| **274 серия** |

**274**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

Това е операция на място: празният хеш е същият хеш (същия обект) като

такъв, на който изпращате ясното съобщение.

R ЗАМЯНА НА СЪДЪРЖАНИЕТО НА ХАШ

Подобно на низове и масиви, хешовете имат метод за замяна:

>> {1 =>"един", 2 =>"два"}. Замени ({10 =>"десет", 20 =>"двадесет"})

=> {10 =>"десет", 20 =>"двадесет"}

Това също е операция на място, както подсказва името замяна.

Ще се обърнем до методите за хеш запитване.

***9.3.6***

***Заявка за хеш***

Подобно на масивите (и много други обекти на Ruby), хешовете предоставят редица методи

с които можете да попитате за състоянието на обекта. Таблица 9.2 показва някои общи хешове

методи за заявка.

Нито един от методите в таблица 9.2 не трябва да предлага никакви изненади в този момент; те са подобни-

голям по дух, а в някои случаи и по писмо, на тези, които сте виждали за масиви. С изключение на

при размер, всички те връщат или вярно, или невярно. Единствената изненада може да бъде колко от

те са синоними. Четири метода тестват за наличие на определен ключ: has\_key ?,

включва ?, ключ ?, и член ?. Може да се направи случай, че това е две или дори три синхронизации

оними твърде много. has\_key? изглежда най-популярният от четирите и е най-много

до точката по отношение на това, за което методът тества.

Has\_value? метод има един синоним: стойност ?. Както при ключовия си аналог,

has\_value? изглежда по-популярен.

Другите методи - празни? и размер - казват ви дали хешът е празен и

какъв е размерът му. (размер може да се нарече и дължина.) Размерът на хеш е броят на

двойки ключ / стойност, които съдържа.

Хешовете получават специално разпределение в списъците с аргументи на метода, както ще видите по-нататък.

Таблица 9.2 Общи методи за хеш заявки и техните значения

Име на метод / Примерен разговор

Значение

h.has\_key? (1)

Вярно, ако h има ключа 1

h.include? (1)

Синоним на has\_key?

h.key? (1)

Синоним на has\_key?

ч. член? (1)

Синоним на has\_key?

h.has\_value? ("три")

Вярно, ако която и да е стойност в h е "три"

h.value? ("три")

Синоним на has\_value?

ч. празен?

Вярно, ако h няма двойки ключ / стойност

h.размер

Брой двойки ключ / стойност в h

|  |
| --- |
| **275 серия** |

**275**

***Хеш***

***9.3.7***

***Хешове като окончателни аргументи на метода***

Ако извикате метод по такъв начин, че *последният* аргумент в списъка с аргументи е

хеш, Ruby ви позволява да пишете хеш без къдрави скоби. Това може би тривиално

звучането на специално правило може на практика да направи списъците с аргументи да изглеждат много по-хубави от

те иначе биха.

Ето един пример. Първият аргумент към add\_to\_city\_database е името на

градът; вторият аргумент е хеш от данни за града, написани без къдрава

скоби (и с помощта на специалния ключ: нотация на символ на стойност):

add\_to\_city\_database ("Ню Йорк",

щат: "Ню Йорк",

население: 7000000,

псевдоним: "Голяма ябълка")

Методът add\_to\_city\_database трябва да свърши повече работа, за да получи достъп до данните

да му бъде предадено, отколкото ако би било обвързващи параметри за аргументи от ляво до

правилен ред чрез списък:

def add\_to\_city\_database (име, информация)

c = City.new

c.name = име

c.state = информация [: състояние]

в. население = информация [: население]

# и т.н.

Разбира се, точният процес, включен в разгъването на хеша, ще варира в различните случаи

друг. (Може би обектите на City съхраняват информацията си като хеш; това би направило

работата на метода е малко по-лесна.) Но по един или друг начин методът трябва да се справи с хеша.

Имайте предвид, че въпреки че трябва да оставите къдравите скоби на хеш литерала

когато това е последното нещо в списъка с аргументи, можете да имате толкова хешове, колкото желаете

аргументи на метода във всяка позиция. Само не забравяйте, че това е само когато хеш е в

последна позиция на аргумента, която можете да освободите от скобите.

Докато не се появи Ruby 2, хеш аргументите от този вид бяха най-близките

стигнете до имена или аргументи на ключови думи. Това обаче се промени. Руби вече има истински

Хешовете като първи аргументи

В допълнение към изучаването на специалния синтаксис, наличен за използване на хешове като

финални аргументи на метода без къдрави скоби, струва си да се отбележи капка от използването на хеш

като първи аргумент на метод. Правилото в този случай е, че трябва не само да сложите

фигурни скоби около хеша, но също така поставете целия списък с аргументи в скоби. Ако

не го правите, Руби ще си помисли, че вашият хеш е кодов блок. С други думи, когато правите това

my\_method {"NY" =>"Ню Йорк"}, 100, "друг аргумент"

Руби интерпретира израза в скоби като блок. Ако искате да изпратите хеш заедно

като аргумент в тази позиция трябва да използвате скоби около целия аргумент

списък, за да стане ясно, че къдравите скоби са свързани с хеш, а не с блокове.

|  |
| --- |
| **276 серия** |

**276**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

именувани аргументи. Синтаксисът им е много хеш, поради което ги разглеждаме

тук, а не в глава 2.

***9.3.8***

***Отклонение обратно към синтаксиса на аргумента: Аргументи с име (ключова дума)***

Използването на именувани аргументи ви спестява проблема с „разгъването“ на хешовете във вашите методи.

Ето един пример с неопределени кости, който показва най-основната версия на именуваните аргументи:

>> def m (a :, b :)

>> па, б

>> край

=>: m

>> m (a: 1, b: 2)

1

2

=> [1, 2]

В края на метода има два параметъра, завършващи с двоеточие. На повикване

в крайна сметка има нещо, което много прилича на хеш. Руби съвпада с всичко, така че

стойностите за a и b се свързват според очакванията. Няма нужда да търсите в хеш.

В предходния пример a и b посочват необходимите аргументи на ключови думи. Вие

не мога да извикам метода без тях:

>> м

ArgumentError: липсват ключови думи: a, b

>> m (a: 1)

ArgumentError: липсваща ключова дума: b

Можете да направите аргументите на ключовите думи незадължителни, като предоставите стойности по подразбиране за вашия

наименовани параметри - което прави списъка с параметри да изглежда още по-хеш:

>> def m (a: 1, b: 2)

>> па, б

>> край

=>: m

>> м

1

2

=> [1, 2]

>> m (a: 10)

10

2

=> [10, 2]

Когато извикате m без аргументи B , се включват настройките по подразбиране за a и b. Ако предоставите

a a, но не b c , получавате a, които сте предоставили и b по подразбиране.

Ами ако отидете в другата посока и извикате метод, използвайки аргументи на ключови думи

че методът не декларира? Ако списъкът с параметри на метода включва двойно-

име със звезда, променливата на това име ще разпръсне всички неизвестни ключови думи аргу-

в хеш, както следва:

>> def m (a: 1, b: 2, \*\* c)

>> pa, b, c

>> край

б

° С

|  |
| --- |
| **277** |

**277**

***Обхвати***

=>: m

>> m (x: 1, y: 2)

1

2

{: x => 1,: y => 2}

=> [1, 2, {: x => 1,: y => 2}]

Ако няма параметър за гъба на ключова дума, извикване на метод като m (x: 1, y: 2) просто преминава

заедно с хеш, който може или не може да се провали, в зависимост от аргументите на метода

очаква.

И разбира се, можете да комбинирате аргументи за ключови думи и ключови думи:

>> def m (x, y, \* z, a: 1, b :, \*\* c, & block)

>> px, y, z, a, b, c

>> край

=>: m

>> m (1,2,3,4,5, b: 10, p: 20, q: 30)

1

2

[3, 4, 5]

1

10

{: p => 20,: q => 30}

=> [1, 2, [3, 4, 5], 1, 10, {: p => 20,: q => 30}]

Тук методът m

Взема два задължителни позиционни аргумента (x и y, обвързани с 1 и 2)

Има параметър „гъба“ (z), който се грижи за допълнителни аргументи след

позиционни (3, 4, 5)

Има един незадължителен и един задължителен аргумент на ключова дума (a и b, съответно,

обвързани с 1 и 10)

Има ключова дума „гъба“ (c), за да абсорбира неизвестни именувани аргументи (p и

q хеш)

Има променлива за обвързване с кодовия блок, ако има такъв (блок)

Рядко ще видите подписи на методи с тази сложност, така че ако можете да следите

елементи в този, вероятно сте готови!

След това ще разгледаме диапазони - които не са точно обекти на колекция, може би, но

които се оказват много общи неща с колекционни обекти.

***9.4***

***Обхвати***

А *гама* е обект с начална точка и крайна точка. Семантиката на диапазон опера-

те включват две основни концепции:

*Включване* - Дадена стойност попада ли в обхвата?

*Изброяване* - Обхватът се третира като колекция от отделни елементи, която може да се проследи.

Логиката на включване важи за всички диапазони; винаги можете да тествате за включване. Логиката

на изброяването започва само с определени диапазони - а именно тези, които включват краен

|  |
| --- |
| **278 серия** |

**278**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

брой дискретни, разпознаваеми стойности. Не можете да итерирате в диапазон, който се намира между

две числа с плаваща запетая, тъй като обхватът включва безкраен брой вал

ues. Но можете да итерирате в диапазон между две цели числа.

Ще запазим допълнителен анализ на итерацията на обхвата и логиката на изброяване за следващия

глава, където ще разгледаме задълбочено изброяването и модула Enumerable. В

в този раздел ще разгледаме предимно другата семантична концепция: логиката на включване. Добре

започнете с някои техники за създаване на обхват.

***9.4.1***

***Създаване на диапазон***

Можете да създавате обекти от диапазон с Range.new. Ако го направите в irb, ще бъдете възнаградени с

изглед на синтаксиса за изграждане на буквален диапазон:

>> r = Range.new (1100)

=> 1..100

Буквалният синтаксис може, разбира се, също да се използва директно за създаване на диапазон:

>> r = 1..100

=> 1..100

Когато видите диапазон с две точки между стойностите на началната и крайната точка, както в

в предишния пример виждате *обхващащ* диапазон. Диапазон с три точки в

средата е *изключителен* диапазон:

>> r = 1 ... 100

=> 1 ... 100

Разликата се състои в това дали крайната точка се счита, че се намира в диапазона.

Отивайки в пълен кръг, можете също да посочите приобщаващо или изключително поведение, когато създавате

яде диапазон с Range.new: по подразбиране е включен диапазон, но можете да принудите

изключителен обхват чрез предаване на трети аргумент на true на конструктора:

>> Range.new (1100)

=> 1..100

>> Range.new (1100, вярно)

=> 1 ... 100

За съжаление няма начин да запомните кое поведение е по подразбиране и кое е

задействано от истинския аргумент, освен за запаметяването му.

Също така е трудно да се запомни кой брой точки се съчетава с кой тип

от обхвата.

R EMEMBERING .. VS . ...

Ако следите дискусионните форуми на Ruby, периодично ще виждате съобщения и публикации от

хора, на които им е трудно да запомнят кое е кое: две срещу три точки, включително

sive спрямо изключителен диапазон.

Един от начините да запомните е да мислите за диапазон като винаги достигащ до представената точка

от каквото и да следва втората точка. В обхващащ диапазон, точката след втората точка

е крайната стойност на диапазона. В този пример стойността 100 е включена в диапазона:

1..100

|  |
| --- |
| **279 серия** |

**279**

***Обхвати***

Но в този изключителен диапазон, стойността 100 е извън ефективния край на диапазона:

1 ... 100

С други думи, можете да мислите за 100 като за „избутани“ надясно в такъв

по този начин, че сега е извън обхвата.

Сега ще се обърнем към логиката за включване на обхвата - раздел, който тясно съответства на

„Запитвайте“ раздели от дискусиите на низове, масиви и хешове, защото повечето от

това, което правите с диапазони, включва тяхното задаване на критерии за включване.

***9.4.2***

***Логика за включване на обхвата***

Диапазоните имат методи за начало и край, които отчитат началните и крайните си точки:

>> r = 1..10

=> 1..10

>> r.begin

=> 1

>> r. край

=> 10

Асортиментът също така знае дали това е ексклузивен (три точки) диапазон:

>> r.exclude\_end?

=> невярно

С поставените цели, можете да започнете да тествате за включване.

Налични са два метода за тестване на включването на стойност в диапазон: покритие? и

включва? (което също е с псевдоним като член?).

T Esting RANGE ВКЛЮЧВАНЕ с обложка ?

Корицата? метод извършва прост тест: ако аргументът на метода е по-голям

от началната точка на диапазона и по-малка от крайната му точка (или равна на нея, за

sive range), тогава се казва, че обхватът *покрива* обекта. Тестовете се извършват с помощта на

Булеви тестове за сравнение, с невярен резултат в случаите, когато сравнението прави

няма смисъл.

Всички следващи сравнения имат смисъл; един от тях се проваля, защото артикулът

не е в обхвата:

>> r = "a" .. "z"

=>"a" .. "z"

>> r.cover? ("a")

=> вярно

>> r.cover? ("abc")

=> вярно

>> r.cover? ("A")

=> невярно

Но този следващ тест е неуспешен, тъй като тестваният елемент за включване не е сравним

с началните и крайните точки на диапазона:

>> r.cover? ([])

=> невярно

**вярно: "a"> = "a"**

**и „a“ <= „z“**

**вярно: “abc”> = “a”**

**и „abc“ <= „z“**

**false: „A“ <„a“**

|  |
| --- |
| **280** |

**280**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

Безсмислено е да питаме дали масивът е по-голям от низа „а“. Ако опитате такъв

сравнение самостоятелно, ще получите фатална грешка. За щастие диапазоните са по-консервативни-

вативен подход и ще ви кажа, че артикулът не е обхванат от обхвата.

Като има предвид покритие? извършва сравнения между начална и крайна точка, а другото включва

тест, включва? (или член?), възприема по-базиран на колекцията подход.

T Esting RANGE ВКЛЮЧВАНЕ, включват ?

Включването? тест третира диапазона като един вид крипто-масив - тоест колекция от

стойности. Обхватът "a" .. "z", например, се счита, че включва (както се измерва с

включва?) само 26-те стойности, които се намират включително между "а" и "z".

Затова включете? дава резултати, които се различават от тези на корицата ?:

>> r.include? ("a")

=> вярно

>> r.include? ("abc")

=> невярно

В случаите, когато диапазонът не може да се тълкува като крайна колекция, като диапазон от

плува, включва? метод се връща към числения ред и сравнение:

>> r = 1.0..2.0

=> 1.0..2.0

>> r.include? (1.5)

=> вярно

Има ли обхвати назад?

Антиклиматичният отговор на въпроса за изостаналите граници е следният: да и не. Вие

може да създаде обратен диапазон, но няма да направи това, което вероятно искате:

>> r = 100 ... 1

=> 100 ... 1

>> r.include? (50)

=> невярно

Асортиментът изпълнява обичайния за вас тест за включване. Тестът изчислява

дали кандидатът за включване е по-голям от началната точка на диапазона

и по-малко от крайната точка. Защото 50 не е нито по-голямо от 100, нито по-малко

от 1, тестът е неуспешен. И се проваля безшумно; това е логическа грешка, а не фатален синтаксис или

грешка по време на изпълнение.

Обратните диапазони се показват в един определен набор от случаи на употреба: като индекси на аргументи

към низове и масиви. Те обикновено са под формата на положителна начална точка и отрицателна

родна крайна точка, като отрицателната крайна точка се брои отдясно:

>>"Това е примерен низ" [10 ..- 5]

=>"проба st"

>> ['a', 'b', 'c', 'd'] [0 .. - 2]

=> ["a", "b", "c"]

|  |
| --- |
| **281 серия** |

**281**

***Комплекти***

Ще видите повече за диапазоните като квази-колекции в следващата глава, както беше обещано. В

в тази глава имаме да разгледаме още един основен клас за колекция: класът Set.

***9.5***

***Комплекти***

Set е единственият клас, който се обсъжда в тази глава, който, строго погледнато, не е Ruby

основен клас. Това е стандартен библиотечен клас, което означава, че за да го използвате, трябва да направите това:

изисква 'set'

Общото правило в тази книга е, че по-скоро разглеждаме основния език, отколкото

стандартната библиотека, но класът Set прави достойно изключение, защото се вписва в него

добре с другите класове за контейнери и събиране, които разгледахме.

А *набор* е уникална колекция от предмети. Обектите могат да бъдат всякакви - низове, инте-

gers, масиви, други набори - но нито един обект не може да се появи повече от веднъж в набора. Единствен по рода си-

ността се прилага и на ниво съдържание на здравия разум: ако комплектът съдържа низа

"Ню Йорк", не можете да добавите низа "Ню Йорк" към него, въпреки че двата низа могат

технически да са различни обекти. Същото важи и за масиви с еквивалентно съдържание.

ЗАБЕЛЕЖКА Вътрешно, наборите използват хеш, за да наложат уникалността на своето съдържание.

Когато елемент се добави към набор, вътрешният хеш за този набор получава нов

ключ. Следователно, всеки два обекта, които биха се считали за дубликати, ако се използват като хеш

ключовете не могат да се появят заедно в набор.

Нека да разгледаме сега как да създаваме набори.

***9.5.1***

***Създаване на набор***

За да създадете набор, използвате конструктора Set.new. Можете да създадете празен набор или вие

може да премине в обект на колекция (дефиниран като обект, който отговаря на всеки или

всеки\_вход). В последния случай всички елементи на колекцията се поставят индивидуално

съюзник в комплекта:

>> new\_england = ["Кънектикът", "Мейн", "Масачузетс",

"Ню Хемпшир", "Род Айлънд", "Върмонт"]

=> ["Кънектикът", "Мейн", "Масачузетс",

"Ню Хемпшир", "Род Айлънд", "Върмонт"]

>> state\_set = Set.new (new\_england)

=> # <Set: {"Кънектикът", "Мейн", "Масачузетс",

"Ню Хемпшир", "Род Айлънд", "Върмонт"}>

***(продължение)***

Можете дори да използвате изключителен обхват назад:

>> ['a', 'b', 'c', 'd'] [0 ... - 2]

=> ["a", "b"]

В тези случаи това, което не работи (поне по начина, който може да сте очаквали) в

диапазонът сам по себе си работи, когато се прилага към низ или масив.

|  |
| --- |
| **282 серия** |

**282**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

Тук създадохме масив new\_england и го използвахме като аргумент на конструктора

за създаване на набора state\_set. Имайте предвид, че няма конструктор на литерални набори (не

еквивалентно на [] за масиви или {} за хешове). Не може да има: наборите са част от стандарта

dard библиотека, а не ядрото и основният синтаксис на езика вече е налице

преди зададената библиотека да се зареди.

Можете също да предоставите кодов блок на конструктора, като в този случай всеки елемент в

обектът за събиране, който доставяте, се предава през блока (отстъпва му) с

получената стойност се вмъква в набора. Например, ето начин за инициализиране на набор

към списък с главни низове:

>> имена = ["Дейвид", "Юкихиро", "Чад", "Ейми"]

=> ["Дейвид", "Юкихиро", "Чад", "Ейми"]

>> name\_set = Set.new (имена) {| name | name.upcase}

=> # <Задайте: {"AMY", "YUKIHIRO", "CHAD", "DAVID"}>

Вместо да използва масива от имена като първоначални стойности, конструкторът на задания дава

всяко име в блока и вмъква какво получава обратно (главна версия на

низ) в набора.

Сега, когато имаме набор, можем да го манипулираме.

***9.5.2***

***Манипулиране на елементи от набор***

Подобно на масивите, наборите имат два режима на добавяне на елементи: или вмъкване на нов елемент

в комплекта или чертеж върху друг обект на колекция като източник за множество нови елементи

моменти. В света на масивите това е разликата между << и concat. За комплекти,

разграничението се отразява в различни методи, които ще разгледаме тук.

А DDING / ИЗВАЛЯВАНЕ НА ЕДИН ПРЕДМЕТ КЪМ / ОТ КОМПЛЕКТ

За да добавите единичен обект към набор, можете да използвате << оператора / метода:

>> tri\_state = Set.new (["Ню Джърси", "Ню Йорк"])

=> # <Set: {"New Jersey", "New York"}>

>> tri\_state <<"Кънектикът"

=> # <Набор: {"Ню Джърси", "Ню Йорк", "Кънектикът"}>

Тук, както при масиви, низове и други обекти, << означават добавянето към колекция

или променлив обект. Ако се опитате да добавите обект, който вече е в комплекта (или обект

това е съдържание, равно на това, което е в комплекта), нищо не се случва:

>> tri\_state <<"Кънектикът"

=> # <Набор: {"Ню Джърси", "Ню Йорк", "Кънектикът"}>

За да премахнете обект, използвайте delete:

>> tri\_state <<"Пенсилвания"

=> # <Комплект: {"Ню Джърси", "Ню Йорк", "Кънектикът", "Пенсилвания"}>

>> tri\_state.delete ("Кънектикът")

=> # <Набор: {"Ню Джърси", "Ню Йорк", "Пенсилвания"}>

Изтриването на обект, който не е в комплекта, не води до грешка. Както при добавянето на дубли-

cate обект, нищо не се случва.

**Упс,**

**само две!**

**Добавя трето**

**Втори път**

|  |
| --- |
| **283 серия** |

**283**

***Комплекти***

Методът << е наличен и като добавяне. Съществува и метод, наречен add ?, който

се различава от add по това, че връща nil (вместо да връща самия набор), ако комплектът е

непроменен след операцията:

>> tri\_state.add? ("Пенсилвания")

=> нула

Можете да тествате връщаната стойност на добавяне? за да се определи дали да се вземат различни условия

национален клон, ако елементът, който сте се опитали да добавите, вече е там.

S ET пресичане , СЪЮЗ , и различие

Комплектите имат концепция за собствено пресичане, обединение и разлика с други набори -

и наистина с други изброими обекти. Класът Set се доставя с необходимото

методи за извършване на тези операции.

Тези методи имат английски имена и символни псевдоними. Имената са

пресичане, с псевдоним като &

съюз, с псевдоним като + и |

разлика, с псевдоним като -

Всеки от тези методи връща нов набор, състоящ се от първоначалния набор, плюс или минус

подходящите елементи от обекта, предоставен като аргумент на метода. The

оригиналният комплект е незасегнат.

Нека преместим нашата група от три държави обратно на Изток и да разгледаме някои определени операции:

>> tri\_state = Set.new (["Кънектикът", "Ню Джърси", "Ню Йорк"])

=> # <Set: {"Кънектикът", "Ню Джърси", "Ню Йорк"}>

# Изваждане (разлика / -)

>> state\_set - tri\_state

=> # <Комплект: {"Мейн", "Масачузетс", "Ню Хемпшир", "Род Айлънд",

"Върмонт"}>

# Добавяне (съюз / + / |)

>> state\_set + tri\_state

=> # <Set: {"Кънектикът", "Мейн", "Масачузетс", "Ню Хемпшир",

"Род Айлънд", "Върмонт", "Ню Джърси", "Ню Йорк"}>

# Пресичане (&)

>> state\_set & tri\_state

=> # <Задайте: {"Кънектикът"}>

>> state\_set | tri\_state

=> # <Set: {"Кънектикът", "Мейн", "Масачузетс", "Ню Хемпшир",

"Род Айлънд", "Върмонт", "Ню Джърси", "Ню Йорк"}>

Има и ексклузивен или оператор ^, който можете да използвате, за да вземете изключителния съюз

между множество и изброима - т.е. набор, състоящ се от всички възникващи елементи

или в множеството, или в изброените, но не и в двете:

>> state\_set ^ tri\_state

=> # <Комплект: {"Ню Джърси", "Ню Йорк", "Мейн", "Масачузетс",

"Ню Хемпшир", "Род Айлънд", "Върмонт"}>

Можете да разширите съществуващ набор, използвайки техника, много подобна по ефект на Set.new

техника: методът на сливане, който може да вземе като аргумент всеки обект, който отговаря

|  |
| --- |
| **Страница 284** |

**284**

C ГЛАВА 9 ***Обекти за събиране и контейнери***

към всеки или всеки\_вход. Това включва масиви, хешове и диапазони - и, разбира се,

други комплекти.

М ERGING колекция В друг набор

Какво се случва, когато обедините друг обект в набор, зависи от това какво

идеята на обекта за итерация върху себе си се състои от. Ето пример за масив, включително

проверете на object\_id, за да потвърдите, че оригиналният комплект е променен на място:

>> tri\_state = Set.new (["Кънектикът", "Ню Джърси"])

=> # <Set: {"Кънектикът", "Ню Джърси"}>

>> tri\_state.object\_id

=> 2703420

>> tri\_state.merge (["Ню Йорк"])

=> # <Set: {"Кънектикът", "Ню Джърси", "Ню Йорк"}>

>> tri\_state.object\_id

=> 2703420

Обединяването на хеш в набор води до добавяне на двуелементни масиви ключ / стойност към

множеството - защото по този начин хешовете се разграждат, когато прегледате

тях. Ето малко нереален пример, който демонстрира технологията:

>> s = Set.new ([1,2,3])

=> # <Задаване: {1, 2, 3}>

>> s.merge ({"New Jersey" =>"NJ", "Maine" =>"ME"})

=> # <Набор: {1, 2, 3, ["Ню Джърси", "Ню Джърси"], ["Мейн", "МЕН"]}>

Ако предоставите хеш аргумент на Set.new, поведението е същото: получавате нов набор

с двуелементни масиви, базирани на хеш.

Може да искате да обедините само ключовете на хеш, а не целия хеш, в a

комплект. В крайна сметка, зададеното членство се основава на уникалността на хеш ключа, под капака. Вие

може да направи това с метода ключове:

>> state\_set = Set.new (["Ню Йорк", "Ню Джърси"])

=> # <Набор: {"Ню Йорк", "Ню Джърси"}>

>> state\_hash = {"Maine" =>"ME", "Vermont" =>"VT"}

=> {"Maine" =>"ME", "Vermont" =>"VT"}

>> state\_set.merge (state\_hash.keys)

=> # <Комплект: {"Ню Йорк", "Ню Джърси", "Мейн", "Върмонт"}>

Изпробвайте някои пермутации на сливане на множество и ще видите, че това е доста отворено

(точно като създаването на набор), стига аргументът да е обект с всеки или всеки

\_входен метод.

Наборите няма да бъдат множества без подмножества и супермножества, а обектите на Руби са под-

и супер осъзнат.

***9.5.3***

***Подмножества и супермножества***

Можете да тествате за връзки подмножество / супермножество между множествата (и аргументите имат

да бъдат набори, а не масиви, хешове или друг вид изброяване или събиране) с помощта на

изненадващо именувани методи за подмножество и супермножество:

|  |
| --- |
| **285 серия** |

**285**

***Обобщение***

>> small\_states = Set.new (["Кънектикът", "Делауеър", "Род Айлънд"])

=> # <Set: {"Кънектикът", "Делауеър", "Род Айлънд"}>

>> tiny\_states = Set.new (["Делауеър", "Род Айлънд"])

=> # <Set: {"Delaware", "Rhode Island"}>

>> tiny\_states.subset? (small\_states)

=> вярно

>> small\_states.superset? (tiny\_states)

=> вярно

Методите Proper\_subset и Proper\_superset също са на ваше разположение. А *правилното*

*подмножество* е подмножество, което е по-малко от родителския набор от поне един елемент. Ако двете

множествата са равни, те са подмножества един на друг, но не са правилни подмножества. По същия начин, *правилно*

*супермножество* на набор е втори набор, който съдържа всички елементи от първия набор плюс поне

един елемент не присъства в първия набор. „Правилната“ концепция е начин за филтриране

случаят, когато даден набор е надмножество или подмножество на себе си - защото всички набори са и двете.

Ще вземем зададената нишка в следващата глава, където ще вземем още един пропуск

чрез колекционни предмети в интерес на по-дълбокото навлизане в изброимите

модул и услугите, базирани на събиране, които предоставя.

***9.6***

***Обобщение***

В тази глава сте виждали

Как да създавате, манипулирате и трансформирате колекционни обекти, включително

- Масиви

- Хеш

- Обхвати

- Комплекти

Назовани аргументи

Разгледахме отблизо основните класове основни контейнери на Ruby, Array и Hash. Ние сме

също така разгледа диапазоните, които по принцип действат като критерии за включване, но знаят

как да се държат като колекции, когато гримът им позволява (момент, който ще го направи

има повече смисъл, след като сте видели повече за модула Enumerable). След диапазони,

разгледахме набори, които са дефинирани в стандартната библиотека и добавихме още едно важно

tant tool към набора от инструменти за събиране на Ruby. Изходният код за класа Set е написан на

Рубин; което ни даде възможност да разгледаме истински производствен Ruby код.

Освен това предприехме заобикаляне на посочени аргументи, които ви пречат да се налага

използвайте хеш ключове като псевдо-ключови думи и хешове на аргументи „разархивиране“ във вашите методи.

Концепцията за *колекцията* в Ruby е тясно свързана с модула Enumerable

ule и неговия принцип на зависимост от всеки метод. В следващата глава ще го направим

отидете по-дълбоко в Enumerable - което означава да разгледате многото търсещи, филтриращи-

операции за сортиране, сортиране и преобразуване, налични за обекти, чиито класове се смесват

този модул.

|  |
| --- |
| **286 серия** |

**286**

*Централни колекции:*

*Изброими и*

*Изброител*

Всички колекционни обекти не са създадени еднакво, но ужасно много от тях имат много

общи характеристики. В Ruby, общи характеристики сред много обекти

са склонни да живеят в модули. Колекциите не са изключение: обекти на колекция в Ruby

обикновено включват модула Enumerable.

Класовете, които използват Enumerable, сключват един вид договор: класът трябва да дефинира

метод на екземпляр, извикан всеки, а в замяна Enumerable дарява обектите на

класът с всякакви поведения, свързани с колекцията. Методите зад тях

поведението се определя по отношение на всяко. В някои отношения може да се каже цялото

Концепцията за „колекция“ в Ruby е обвързана с модула Enumerable и

методи, които той дефинира отгоре на всеки.

***Тази глава обхваща***

Смесване на Enumerable във вашите класове

Използването на изброими методи в

колекционни обекти

Низовете като квази-изброими обекти

Сортиране на изброими със Сравним

модул

Изброители

|  |
| --- |
| **Страница 287** |

**287**

***Получаване на изброеност чрез всеки***

Вече сте виждали по малко от всеки в действие. Тук ще видите много повече. Имайте

имайте предвид обаче, че въпреки че всеки основен клас на колекция участва в Изброимите

модул, всеки от тях също има свои собствени методи. Методите на масив не са

идентични с тези на комплект; тези от диапазон не са идентични с тези на хеш. И

понякога класовете за събиране споделят имена на методи, но методите не се справят точно

едно и също нещо. Не винаги *могат да* правят едно и също; целият смисъл е да има много

класове за събиране на типове, но за да извлечете възможно най-често срещано поведение в

общ модул.

Можете да смесите Enumerable в собствените си класове:

клас С

включват Изброими

край

Само по себе си това не прави много. За да се възползвате от предимствата на Enumerable, трябва

дефинирайте метод на всеки екземпляр във вашия клас:

клас С

включват Изброими

def всеки

# подходящ код тук

край

край

В този момент обектите от клас C ще имат способността да извикват всеки дефиниран метод на екземпляр

в Изброими.

В допълнение към модула Enumerable, в тази глава ще разгледаме внимателно

свързан клас, наречен Enumerator. *Изброителите* са обекти, които капсулират знанията за

как да прегледате определена колекция. Чрез опаковане на итерационната интелигентност в

обект, който е отделен от самата колекция, изброителите добавят още и

мощно измерение на вече значителните съоръжения за манипулиране на Ruby.

Нека започнем, като разгледаме по-отблизо всяка и нейната роля като двигател за изброяване-

бле поведение.

***10.1 Получаване на изброеност чрез всеки***

Всеки клас, който се стреми да бъде изброен, трябва да има всеки метод, чиято работа е да

дават елементи на предоставен кодов блок, един по един.

Точно това, което всеки прави, ще варира в различните класове. В случай на масив,

всеки дава първия елемент, след това втория и т.н. В случай на хеш, то

дава двойки ключ / стойност под формата на масиви от два елемента. В случай на манипулатор на файл,

той дава по един ред от файла наведнъж. Обхватът се итерира, като първо се решава дали да се повтори

е възможно (което не е, например, ако началната точка е плаваща) и след това се преструва

да бъде масив. И ако дефинирате всеки в свой клас, това може да означава каквото и да е

искате да означава - стига да дава нещо. Така че всеки има различна семантика

за различни класове. Но както и да е внедрен всеки, методите в Enumerable

модул зависи от възможността да го извикате.

|  |
| --- |
| **288 серия** |

**288**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Можете да разберете добре как работи Enumerable, като напишете малко доказателство за

концептуален клас, който го използва. Следният списък показва такъв клас: Rainbow. Това

class има всеки метод, който дава по един цвят наведнъж. Защото класът се смесва

Изброими, неговите екземпляри автоматично се предоставят с дефинираните методи на екземпляра

в този модул.

клас Дъга

включват Изброими

def всеки

добив "червен"

добив "портокал"

добив "жълт"

добив "зелен"

добив "син"

добив "индиго"

добив "виолетов"

край

край

Всеки екземпляр на Rainbow ще знае как да преминава през цветовете. В най-простия

случай, можем да използваме всеки метод:

r = Rainbow.new

r.each do | цвят |

поставя "Следващ цвят: # {цвят}"

край

Резултатът от тази проста итерация е както следва:

Следващ цвят: червен

Следващ цвят: оранжев

Следващ цвят: жълт

Следващ цвят: зелен

Следващ цвят: син

Следващ цвят: индиго

Следващ цвят: виолетов

Но това е само началото. Тъй като Rainbow се смесва в модула Enumerable,

дъгите автоматично са надарени с цял набор от методи, изградени върху

всеки метод.

Ето пример: find, който връща първия елемент в изброен обект

за които предоставеният кодов блок връща true. Да кажем, че искаме да намерим първия цвят

който започва с буквата у. Можем да го направим с find, по следния начин:

r = Rainbow.new

y\_color = r.find {| цвят | color.start\_with? ('y')}

поставя „Първият цвят, започващ с„ y “, е # {y\_color}.“

намерете произведения, като извикате всеки. всеки от тях дава елементи и find използва кодовия блок, който имаме

даде го да тества тези елементи един по един за съвпадение. Когато всеки се приближи до добив-

жълто, find го прекарва през блока и той преминава теста. Променливата y\_color

Листинг 10.1 **Изброителен** клас и неговото внедряване на **всеки** метод

**Изход: Стартиране на първия цвят**

**с 'y' е жълто.**

|  |
| --- |
| **289 серия** |

**289**

***Изброими булеви заявки***

следователно получава стойността жълто. Забележете, че няма нужда да дефинирате find. Това е част

от Изброими, които сме смесили. Той знае какво да прави и как да използва всеки, за да го направи.

Определянето на всеки, заедно със смесването в Enumerable, ви купува много функции-

националност за вашите обекти. Голяма част от функционалността за търсене и заявки, която виждате в

Ruby масиви, хешове и други обекти за събиране идва директно от Enumerable. Ако

искате да знаете кои методи предоставя Enumerable, попитайте го:

>> Enumerable.instance\_methods (false) .sort

=> [: всички ?,: всякакви ?,: парче,: събиране,: събиране\_котка,: брой,: цикъл,: откриване,

: drop,: drop\_ while,: each\_cons,: each\_entry,: each\_slice,: each\_with\_index,

: each\_with\_object,: entries,: find,: find\_all,: find\_index,: first,

: flat\_map,: grep,: group\_by,: include ?,: inject,: lazy,: map,: max,: max\_by,

: member ?,: min,: min\_by,: minmax,: minmax\_by,: none ?,: one ?,: дял,

: намалете,: отхвърлете,: reverse\_each,: select,: slice\_before,: sort,: sort\_by,

: take,: take\_ while,: to\_a,: to\_h,: zip]

Благодарение на фалшивия аргумент списъкът включва само методите, дефинирани в

Изброителен модул. Всеки от тези методи е изграден върху всеки от тях.

В следващите раздели ще видите примери за много от тези методи. Някои от

останалите ще изникнат в следващите глави. Примерите в останалата част от това

Главата ще се опира на четирите основни класа на колекция - Array, Hash, Range и

Задайте - повече или по-малко произволно. Глава 9 ви запозна индивидуално с тези класове.

Въоръжени с усещането за това какво кара всеки от тях да кърлежи, вие сте в добра позиция

изучавайте общото между тях.

Някои от методите в изброените класове на Ruby всъщност са заменени в тях

класове. Например ще намерите изпълнения на map, select, sort и други

Изброими методи на екземпляр във файла с изходен код array.c; класът Array не симулира

ply осигурете всеки метод и смесете в Enumerable (макар че го прави и печели

поведение по този начин). Тези презаписи се правят или защото даден клас изисква специални

поведение пред лицето на даден Изброителен метод или в името на ефективността.

Няма да разглеждаме всички замествания. Основната точка тук е да се изследва

начини, по които всички класове за събиране споделят поведение и интерфейс.

В следващото ще разгледаме няколко категории методи от Enumerable. Добре

започнете с някои булеви методи.

***10.2 Изброими булеви заявки***

Редица изброими методи връщат true или false в зависимост от това дали един или

повече елемент отговаря на определени критерии. Даден масив състояния, съдържащи имената

от всички щати в Съединените американски щати, ето как можете да изпълните някои

от тези булеви заявки:

# Включва ли масивът Луизиана?

>> States.include? ("Луизиана")

=> вярно

# Включват ли всички държави интервал?

>> States.all? {| държава | състояние = ~ / /}

=> невярно

|  |
| --- |
| **290 серия** |

**290**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

# Включва ли *някоя* държава интервал?

>> States.any? {| държава | състояние = ~ / /}

=> вярно

# Има ли една и само една държава с име "Запад" в името си?

>> States.one? {| държава | състояние = ~ / Запад /}

=> вярно

# Няма ли държави с "Изток" в имената им?

>> States.none? {| държава | състояние = ~ / Изток /}

=> вярно

Ако състоянията вместо това бяха хеш с имена на състояния като ключове и съкращения като стойности,

можете да провеждате подобни тестове, въпреки че ще трябва да се приспособите към факта, че Hash # всеки

дава както ключ, така и стойност всеки път. Hash # включва? проверки на методите

за включване на ключове, както видяхте в глава 9, но другите методи в предишната

пример обработва двойки ключ / стойност:

# Хешът включва ли Луизиана?

>> States.include? ("Луизиана")

=> вярно

# Включват ли всички държави интервал?

>> States.all? {| държава, съкратено | състояние = ~ / /}

=> невярно

# Има ли една и само една държава с име "Запад" в името си?

>> States.one? {| държава, съкратено | състояние = ~ / Запад /}

=> вярно

Във всички тези случаи можете да вземете масив чрез States.keys и да извършите тестовете на

този масив директно:

# Включват ли всички държави интервал?

>> States.keys.all? {| държава, съкратено | състояние = ~ / /}

=> невярно

Генериране на целия масив ключове предварително, вместо да се разхождате из хеша

това вече е там, е малко загуба на памет. И все пак новият масив съдържа ключа

обекти, които вече съществуват, така че той само „губи“ паметта, предназначена за опаковане на ключовете

в масив. Паметта, заета от самите клавиши, не се увеличава.

Хешовете се итератират с двуелементни масиви

Когато итерирате през хеш с всеки или друг вграден итератор, хешът е

отстъпва на вашия кодов блок по една двойка ключ / стойност едновременно - и двойките са двуелементни

масиви. Можете, ако желаете, да предоставите само един параметър на блока и да уловите цялото

малък масив:

hash.each {| двойка | ...}

В такъв случай ще намерите ключа при двойка [0] и стойността при двойка [1]. Обикновено това

има повече смисъл да вземете ключа и стойността в отделни параметри на блока. Но всички

това, което се случва е, че двете са обвити в двуелементен масив и този масив

се дава. Ако искате да работите с данните в тази форма, можете.

**включва? консултира**

**ключове на хеш**

**Хеш дава ключ / стойност**

**двойки за блокиране**

|  |
| --- |
| **Страница 291** |

**291**

***Безброй търсене и избор***

Ами наборите и диапазоните? Set iteration работи подобно на итерация на масив за Boolean

заявка (и повечето други) цели: ако състоянията са набор, можете да стартирате абсолютно същото

заявки като тези в примера със същите резултати. С диапазони, изброеност

става малко по-сложен.

По-смислено е да разглеждате някои диапазони като безброй - като колекции от предмети

през които можете да преминете - от другите. Включването? метод работи за всеки диапазон.

Но другите Boolean Enumerable методи налагат проблема с изброяването: ако

обхватът може да бъде изразен като списък на дискретни елементи, тогава тези методи работят; но ако

не може, както при набор от плувки, след това извикването на някой от методите задейства фатална грешка:

>> r = Range.new (1, 10)

=> 1..10

>> r.one? {| n | n == 5}

=> вярно

>> r.none? {| n | n% 2 == 0}

=> невярно

>> r = Range.new (1.0, 10.0)

=> 1.0..10.0

>> r.one? {| n | n == 5}

TypeError: не може да итерира от Float

>> r = Range.new (1, 10.3)

=> 1..10.3

>> r.any? {| n | n> 5}

=> вярно

Като се има предвид диапазон, обхващащ две цели числа, можете да стартирате тестове като едно? и никой? Б, защото

диапазонът може лесно да се подхлъзне, за да се държи като колекция: всъщност диапазонът 1..10

приема API на съответния масив, [1,2,3,4,5,6,7,8,9,10].

Но диапазонът между два плувки c не може да се държи като крайна колекция от дискретни

стойности. Безсмислено е да се произвежда "всеки" поплавък в диапазон. Асортиментът има всеки

метод, но методът е написан по такъв начин, че да откаже итерация над плувки d .

(Фактът, че грешката е TypeError, а не NoMethodError, показва, че

всеки метод съществува, но не може да функционира в този диапазон.)

Можете да използвате плувка като крайна точка на диапазона и все пак да получите изброяване, стига че

началната точка е цяло число e . Когато извикате всеки (или един от методите, изградени върху

всеки), диапазонът се държи като колекция от цели числа, започвайки от началната точка и

завършваща в крайната точка, закръглена надолу до най-близкото цяло число. Това цяло число се счита

да бъдат включени в обхвата, независимо дали обхватът е включителен или изключителен (защото,

в края на краищата официалната крайна точка е поплавък, който е по-висок от цялото число под него).

В допълнение към отговорите на различни верни / неверни въпроси относно тяхното съдържание, enu-

измерими обекти превъзхождат при извършване на операции за търсене и избор. Ще се обърнем към

тези сега.

***10.3 Безброй търсене и избор***

Обичайно е да искате да филтрирате колекция от обекти въз основа на една или повече селекции

критерии. Например, ако имате база данни с хора, регистриращи се за конференция,

и искате да изпратите напомняния за плащане на хората, които не са платили, можете

б

° С

д

д

|  |
| --- |
| **292 серия** |

**292**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

филтрирайте пълен списък въз основа на състоянието на плащане. Или може да се наложи да стесните списък с

числа само до четните. И така нататък; случаите на използване за избор на елементи

от изброени обекти са неограничени.

Модулът Enumerable предоставя няколко средства за филтриране на колекции и за

търсене в колекции за намиране на един или повече елементи, които отговарят на един или повече критерии.

Тук ще разгледаме няколко метода за филтриране и търсене. Всички те са итератори:

всички те очакват да предоставите код код. Кодовият блок е филтър за избор. Вие

дефинирайте критериите си за избор (вашите тестове за включване или изключване) вътре в блока.

Връщаната стойност на целия метод може, в зависимост от това кой метод използвате

и върху това, което открие, бъде един обект, масив (вероятно празен) от обекти, съответстващи на вашия

критерии или нула, което показва, че критериите не са били изпълнени.

Ще започнем с търсене с един обект с помощта на find и след това ще си проправим път през sev-

ерални техники за извличане на набор от множество обекти от изброена заявка.

***10.3.1 Получаване на първото съвпадение с find***

find (наличен и като синонимно откриване) локализира първия елемент в масив

за които кодовият блок, когато е извикан с този елемент като аргумент, връща true.

Например, за да намерите първото число, по-голямо от 5 в масив от цели числа, можете

използвайте find по този начин:

>> [1,2,3,4,5,6,7,8,9,10] .намерете {| n | n> 5}

=> 6

find итерации през масива, като всеки елемент от своя страна се връща към блока. Ако

block връща всичко с булева стойност на true, елементът дава "победи" и

find спира да се итерира. Ако find не успее да намери елемент, който преминава теста за кодов блок, той

връща нула. (Опитайте да промените n> 5 на n> 100 в примера и ще видите.) Това е

esting, за да размишлявате върху случая, когато вашият масив има nil като един от неговите елементи, а вашият

код блок търси елемент, равен на нула:

[1,2,3, нула, 4,5,6] .намерете {| n | n.nil? }

При тези обстоятелства търсенето винаги връща нула - независимо дали търсенето е успешно или не!

Това означава, че тестът е безполезен; не можете да разберете дали е успял. Можете да работите

около тази ситуация с други техники, като включване? метод, с който

можете да разберете дали масив има nil като елемент. Можете също така да предоставите „нищо-

функцията за откриване ”- обект на Proc - като аргумент за намиране, в този случай тази функция

ще се извика, ако операцията за намиране не успее. Не сме разглеждали Proc обекти в

дълбочина, въпреки че сте виждали някои примери за тях във връзка с хан-

dling на кодови блокове. За бъдещи справки, ето пример за това как да се намери търсене

с функция за справяне с неизправности:

>> грешка = ламбда {11}

=> # <Proc: 0x434810 @ (irb): 6 (ламбда)>

>> over\_ten = [1,2,3,4,5,6] .намерете (неуспех) {| n | n> 10}

=> 11

б

|  |
| --- |
| **Страница 293** |

**293**

***Безброй търсене и избор***

В този пример анонимната функция (обектът Proc) връща 11 B , така че дори

няма число по-голямо от 10 в масива, все пак получавате едно. (Ще видите агнешко

das и Proc отблизо в глава 14.)

Въпреки че find винаги връща един обект, find\_all, известен също като select, винаги

връща масив, както и неговото отрицателно еквивалентно отхвърляне.

***10.3.2 Получаване на всички съвпадения с find\_all (известен още като select) и отхвърляне***

find\_all (същият метод като select) връща нова колекция, съдържаща всички

елементи от оригиналната колекция, които отговарят на критериите в кодовия блок, а не само

първият такъв елемент (като при търсене). Ако не са намерени съвпадащи елементи, find\_all

връща празен обект за събиране.

В общия случай - например, когато използвате Enumerable в собствените си класове -

„колекцията“, върната от select, ще бъде масив. Руби прави специални уговорки

за хешове и набори обаче: ако изберете хеш или набор, ще получите обратно хеш или набор.

Това е подобрено поведение, което не е строго част от Enumerable.

Тук ще се придържаме към примери за масиви:

>> a = [1,2,3,4,5,6,7,8,9,10]

=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> a.find\_all {| артикул | елемент> 5}

=> [6, 7, 8, 9, 10]

Доминирането на масива

Масивите обикновено служат като контейнери за повечето резултати, които се връщат от

безброй операции по избор и филтриране, независимо дали обектът е избран или не

от или филтриран е масив. Има някои изключения от това квази правило, но то важи

вярно широко.

Най-ясният начин да го видите е да създадете свой собствен клас и да наблюдавате

инг. какво получавате обратно от избраните от вас заявки. Погледнете отново класа Rainbow в

изброяване 10.1. Сега вижте какво получавате обратно, когато изпълнявате някои заявки:

>> r = Rainbow.new

=> # <Rainbow: 0x45b708>

>> r.изберете {| цвят | color.size == 6}

=> ["оранжево", "жълто", "индиго", "виолетово"]

>> r.map {| цвят | цвят [0,3]}

=> ["червено", "ora", "yel", "gre", "blu", "ind", "vio"]

>> r.drop\_ while {| цвят | color.size <5}

=> ["оранжево", "жълто", "зелено", "синьо", "индиго", "виолетово"]

Във всеки случай наборът от резултати се връща в масив.

Масивът е най-общият контейнер и следователно логичният кандидат за

роля на универсален формат на резултата. Възникват няколко изключения. Хеш връща хеш от a

изберете или отхвърлете операцията. Задава масиви за връщане от картата, но можете да извикате карта! На

набор за промяна на елементите на комплекта на място. В по-голямата си част обаче enumer-

възможните операции по избор и филтриране се връщат при вас в масиви.

б

|  |
| --- |
| **Страница 294** |

**294**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

>> а.изберете {| артикул | елемент> 100}

=> []

Първата операция find\_all връща масив от всички елементи, преминали теста

блока: всички елементи, които са по-големи от 5 B . Втора операция също се връща

масив, този път на всички елементи в оригиналния масив, които са по-големи от 10.

Няма такива, така че се връща празен масив c .

(Масивите, хешовете и наборите имат версия на взрив, изберете !, която намалява колекцията

постоянно само за онези елементи, които са преминали теста за подбор. Няма find\_all!

синоним; трябва да използвате select !.)

Точно както можете да изберете елементи, така можете да отхвърлите елементи, което означава, че ще разберете

кои елементи от масив не връщат истинска стойност, когато се предават на блока.

Използвайки масива от предишния пример, можете да направите това, за да получите масива минус

всички и всички елементи, които са по-големи от 5:

>> a.reject {| артикул | елемент> 5}

=> [1, 2, 3, 4, 5]

(Още веднъж има взрив, версия на място, отхвърли !, специално за масиви, хешове,

и комплекти.)

Ако някога сте използвали помощната програма на командния ред grep, следващият метод ще звъни.

Ако не сте, така или иначе ще го разберете.

***10.3.3 Избор на мачове от три нива с grep***

Методът Enumerable # grep ви позволява да избирате от изброим обект въз основа на

операторът за равенство на случаи, ===. Най-честото приложение на grep е това, което

съответства най-тясно на общата работа на помощната програма на командния ред на

същото име, съвпадение на модел за низове:

>> цветове =% w {червено оранжево жълто зелено синьо индиго виолетово}

=> ["червено", "оранжево", "жълто", "зелено", "синьо", "индиго", "виолетово"]

>> colors.grep (/ o /)

=> ["оранжево", "жълто", "индиго", "виолетово"]

Но общността на === ви позволява да правите някои фантастични неща с grep:

>> miscellany = [75, „здравей“, 10 ... 20, „сбогом“]

=> [75, „здравей“, 10 ... 20, „довиждане“]

>> miscellany.grep (низ)

=> [„здравей“, „сбогом“]

>> miscellany.grep (50..100)

=> [75]

String === обектът е вярно за двата низа в масива, така че масив от тези два

струни е това, което можете да получите обратно от grepping за струнен B . Обхватите се изпълняват === като

тест за включване. Обхватът 50..100 включва 75; оттук и резултатът от грепване

смесени за този диапазон c .

° С

б

° С

|  |
| --- |
| **Страница 295** |

**295**

***Безброй търсене и избор***

По принцип изявлението enumerable.grep (израз) е функционално еквивалентно

отпуснат на това:

enumerable.select {| елемент | израз === елемент}

С други думи, той избира стойност на истината въз основа на извикване ===. В допълнение, grep може

вземете блок, в който случай той дава всеки елемент от резултата си, зададен на блока преди

връщане на резултатите:

>> цветове =% w {червено оранжево жълто зелено синьо индиго виолетово}

=> ["червено", "оранжево", "жълто", "зелено", "синьо", "индиго", "виолетово"]

>> colors.grep (/ o /) {| цвят | color.capitalize}

=> ["Оранжево", "Жълто", "Индиго", "Виолетово"]

Пълният синтаксис на grep

enumerable.grep ( *израз* ) {| *артикул* | ...}

по този начин действа на практика по следния начин:

enumerable.select {| *артикул* | *израз* === *елемент* } .map {| *артикул* | ...}

Отново, най-вече ще виждате (и вероятно най-вече използвате) grep като избрана низ, базирана на шаблон

тор. Но имайте предвид, че грепирането е обвързано с равенството между регистрите (===) и може да се използва

съответно в различни ситуации.

Независимо дали се извършва като select или grep или някаква друга операция, сценарии за избор-

ios често изискват групиране на резултатите в клъстери или категории. Изброимите

Методите #group\_by и #partition правят удобни разпоредби точно за този вид

на групиране.

***10.3.4 Организиране на резултатите от селекцията с group\_by и дял***

Операцията group\_by за изброим обект взема блок и връща хеш. The

блок се изпълнява за всеки обект. За всяка уникална стойност на връщания блок, хешът на резултата

получава ключ; стойността за този ключ е масив от всички елементи на изброеното за

които блокът връща тази стойност.

Един пример трябва да изясни операцията:

>> цветове =% w {червено оранжево жълто зелено синьо индиго виолетово}

=> ["червено", "оранжево", "жълто", "зелено", "синьо", "индиго", "виолетово"]

>> colors.group\_by {| цвят | цвят.размер}

=> {3 => ["червено"], 6 => ["оранжево", "жълто", "индиго", "виолетово"],

5 => [„зелено“], 4 => [„синьо“]}

Блокът {| цвят | color.size} връща цяло число за всеки цвят. Хешът

върнато от цялата операция group\_by се въвежда в различни размери (3, 4, 5, 6) и

стойностите са масиви, съдържащи всички низове от оригиналния масив, които са от

размер, представен от съответните клавиши.

Методът на разделяне е подобен на group\_by, но разделя елементите на enu-

merable в два масива въз основа на това дали кодовият блок връща true за елемента.

|  |
| --- |
| **296 серия** |

**296**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Няма хеш, просто масив от два масива. Двата масива винаги се връщат

вярно / невярно нареждане.

Помислете за клас Person, където всеки човек има възраст. Класът също определя

екземпляр метод тийнейджър ?, което е вярно, ако възрастта на човека е между 13 и

19, включително:

клас Личност

attr\_accessor: възраст

def инициализиране (опции)

self.age = опции [: възраст]

край

def тийнейджър?

(13..19) === възраст

край

край

Сега нека генерираме масив от хора:

хора = 10. стъпка (25,3) .карта {| i | Person.new (: възраст => i)}

Този код прави итерация от 10 до 25 на стъпки от 3 (10, 13, 16, 19, 22, 25), преминавайки

всяка от стойностите към блока на свой ред. Всеки път се създава нов Човек

с възрастта, съответстваща на нарастването. Благодарение на картата обектът на човека е

всички се натрупват в масив, който се присвоява на хора. (Веригата на итера-

Картата на стъпката на итератора е възможна поради факта, че стъпката връща изброяване

тор. Понастоящем ще научите повече за изброителите.)

Имаме нашите шестима души; сега нека ги разделим на тийнейджъри и не-тийнейджъри:

тийнейджъри = хора.раздел {| лице | person.teenager? }

Масивът за тийнейджъри има следното съдържание:

[[# <Лице: 0x000001019d1a50 @ age = 13>, # <Лице: 0x000001019d1a50 @ age = 13>, # <Лице: 0x000001019d1a50 @ age = 13>, # <Лице: 0x000001019d1a50 @ age = 13>,

# <Person: 0x000001019d1988 @ age = 19>], [# <Person: 0x000001019d1ac8 @ age = 10>,

# <Лице: 0x000001019d1910 @ age = 22>, # <Лице: 0x000001019d1910 @ age = 25>]]

Имайте предвид, че това е масив, съдържащ два подмасива. Първият съдържа тези хора за

кого person.teenager? върнато истина; вторият е не-тийнейджърите.

Вече можем да използваме информацията, например, за да разберем колко тийнейджъри и

не-тийнейджъри имаме:

поставя "# {тийнейджъри [0] .size} тийнейджъри; # {тийнейджъри [1] .size} не-тийнейджъри"

Резултатът от това изявление отразява факта, че половината от нашите хора са тийнейджъри и

половината не са:

3 тийнейджъри; 3 не-тийнейджъри

Нека да разгледаме сега някои „елементарни“ операции - методи, които включват относително

фино манипулиране на конкретни елементи на колекцията.

|  |
| --- |
| **Страница 297** |

**297**

***Елементарно изброими операции***

***10.4 Елементарно изброими операции***

Колекциите са родени, за да бъдат прехвърлени, но те също така съдържат лица със специален статус

обекти: първият или последният в колекцията и най-големият (най-големият) или най-малкият (най-малкият).

Многобройни обекти идват с няколко инструмента за обработка на елементи по тези линии.

***10.4.1 Първият метод***

Първо число, както подсказва името, връща първия елемент, който се среща при

итерация върху изброеното:

>> [1,2,3,4] .първо

=> 1

>> (1..10) .първо

=> 1

>> {1 => 2, "one" =>"two"}. Първо

=> [1, 2]

Обектът, върнат от first, е същият като първия обект, който получавате при итерация

чрез родителския обект. С други думи, това е първото нещо, дадено от всеки. Поддържа-

с факта, че хешовете дават двойки ключ / стойност в двуелементни масиви, като вземат

first element of a hash ви дава двуелементен масив, съдържащ първата двойка, която е била

вмъкнати в хеша (или първия вмъкнат ключ и новата му стойност, ако сте променили

тази стойност във всяка точка):

>> хеш = {3 =>"три", 1 =>"едно", 2 =>"две"}

=> {3 =>"три", 1 =>"едно", 2 =>"две"}

>> hash.first

=> [3, „три“]

>> хеш [3] = "trois"

=>"trois"

>> hash.first

=> [3, „trois“]

Може би най-забележителният момент за Enumerable # first е, че няма

Изброими # последни. Това е така, защото намирането на края на итерацията не е толкова просто

отделение като намиране на началото. Помислете за случай, в който повторението продължава вечно.

Ето малък клас Die (умрете като в единствено число на зарове). Той се повтаря чрез валцуване на матрицата за

винаги и давайки резултат всеки път:

клас Die

включват Изброими

def всеки

цикъл правя

добив (6) + 1

край

край

край

Цикълът използва метода Kernel # rand. Извикан без аргумент, този метод общо

събира произволно число с плаваща запетая *n* такова, че *0 <= n <1* . С аргумент *i* , то

връща произволно цяло число *n* такова, че *0 <= n <i.* По този начин rand (6) произвежда цяло число в

**първото означава първо**

**вмъкнат**

**Новата стойност не**

**промяна на реда за вмъкване**

|  |
| --- |
| **Страница 298** |

**298**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

диапазона (0..5). Добавянето на едно към това число дава число между 1 и 6, което

съответства на това, което получавате, когато хвърлите матрица.

Но основното е, че Die # всеки продължава вечно. Ако използвате клас Die,

трябва да вземете мерки за излизане от цикъла. Ето една малка игра, където сте

спечелете веднага щом матрицата се появи 6:

поставя „Добре дошли в„ Печелиш, ако хвърлиш 6 “!“

d = Die.new

d.всеки | ролка |

поставя „Превърнахте # {ролка}.“

ако ролка == 6

поставя "Вие печелите!"

почивка

край

край

Типично бягане може да изглежда така:

Добре дошли в „Печелиш, ако хвърлиш 6“

Хвърлихте 3.

Хвърлихте 2.

Хвърлихте 2.

Хвърлихте 1.

Хвърлихте 6.

Ти печелиш!

Настрана тривиалността на играта, въпросът е, че би било безсмислено да се обаждам последен

върху вашия предмет на матрицата, защото няма последно хвърляне на матрицата. За разлика от приемането на първата еле-

възприемането на последния елемент от изброима няма обобщаващо значение.

По същата причина - недостижимостта на края на изброяването - enu-

merable клас с безкрайно отстъпващ всеки метод не може да направи много с методите

като select и map, които не връщат резултатите си, докато основната итерация не бъде

завършен. Поводите за безкрайна итерация във всеки случай са малко; но спазвайки

поведението и въздействието на безкраен всеки може да бъде поучително за това, което разкрива

по-често срещаният, краен случай.

Имайте предвид обаче, че някои изброени класове имат последен метод:

по-специално Array и Range. Нещо повече, всички изброими имат метод за вземане, един вид

обобщение на първо и метод за придружаване, наречен drop.

***10.4.2 Методи за вземане и пускане***

Изброимите знаят как да „вземат“ определен брой елементи от началото на

себе си и обратно как да „пуснат“ определен брой елементи. Вземете и

операциите за пускане в основата правят едно и също нещо - те разделят колекцията на конкретно

точка - но те се различават по това, което връщат:

>> състояния =% w {NJ NY CT MA VT FL}

=> ["NJ", "NY", "CT", "MA", "VT", "FL"]

>> States.take (2)

=> ["NJ", "NY"]

>> States.drop (2)

=> ["CT", "MA", "VT", "FL"]

**Хваща първите две**

**елементи**

**Събиране на грабване освен**

**първите два елемента**

|  |
| --- |
| **299 серия** |

**299**

***Елементарно изброими операции***

Когато вземате елементи, получавате тези елементи. Когато пуснете елементи, получавате

оригинална колекция минус елементите, които сте изпуснали. Можете да ограничите приема

и пуснете операции, като предоставите блок и използвате вариантните форми take\_ while и

drop\_ while, които определят размера на „take“ не от целочислен аргумент, а от

истинността на блока:

>> States.take\_ while {| s | /N/.match(s)}

=> ["NJ", "NY"]

>> States.drop\_ while {| s | /N/.match(s)}

=> ["CT", "MA", "VT", "FL"]

Операциите за вземане и пускане са вид хибрид на first и select. Те са

закрепени към началото на итерацията и се прекратяват, след като са изпълнили

изискване за количество или е възникнала неизправност на блок.

Можете също така да определите минималните и максималните стойности в една изброима

колекция.

***10.4.3 Методите min и max***

Методите min и max правят това, което звучат така:

>> [1,3,5,4,2] .макс

=> 5

>>% w {Ruby C APL Perl Smalltalk} .min

=>"APL"

Минимумът и максимумът се определят от <=> (операция за сравнение на космически кораб-

tor) логика, която за масива от низове поставя "APL" на първо място във възходящ ред. Ако ти

искате да извършите минимален или максимален тест, базиран на критерии без подразбиране, можете

предоставете код код:

>>% w {Ruby C APL Perl Smalltalk} .min {| a, b | a.size <=> b.size}

=>"C"

По-рационализиран базиран на блокове подход обаче е да се използва min\_by или max\_by, които

изпълнете сравнението имплицитно:

>>% w {Ruby C APL Perl Smalltalk} .min\_by {| lang | lang.size}

=>"C"

Има и метод minmax (и съответния метод minmax\_by), който

ви дава чифт стойности, една за минималната и една за максималната:

>>% w {Ruby C APL Perl Smalltalk} .minmax

=> ["APL", "Smalltalk"]

>>% w {Ruby C APL Perl Smalltalk} .minmax\_by {| lang | lang.size}

=> ["C", "Smalltalk"]

Имайте предвид, че семейството min / max от изброени методи е винаги на разположение, дори

когато го използвате не е добра идея. Не бихте искали да направите това, например:

die = Die.new

поставя die.max

**Няма нужда да сравнявате**

**два параметъра**

**изрично в кодов блок**

|  |
| --- |
| **Страница 300** |

**300**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Безкрайният цикъл, с който се изпълнява Die # each, няма да позволи максимална стойност

някога да бъде определено. Програмата ви ще увисне.

В случай на хешове, min и max използвайте клавишите за определяне на подреждането. Ако искаш

за да използвате стойности, \* \_от членовете на семейството min / max могат да ви помогнат:

>> state\_hash = {"New York" =>"NY", "Maine" =>"ME",

"Аляска" =>"АК", "Алабама" =>"АЛ"}

=> {"Ню Йорк" =>"Ню Йорк", "Мейн" =>"МЕН", "Аляска" =>"АК", "Алабама" =>"АЛ"}

>> state\_hash.min

=> ["Алабама", "AL"]

>> state\_hash.min\_by {| име, съкращение | име}

=> ["Алабама", "AL"]

>> state\_hash.min\_by {| име, съкращение | съкратен}

=> ["Аляска", "АК"]

И разбира се, можете, ако желаете, да извършите изчисления вътре в блока, които включват

както ключът, така и стойността.

На този етап разгледахме примери за всеки метод и как те се свързват с a

брой методи, които са изградени върху тях. Време е сега да разгледаме някои методи-

кодове, които са подобни на всеки, но малко по-специализирани. Най-важният от тях

е карта. Всъщност картата е достатъчно важна, за да я разгледаме отделно в нейния собствен раздел

ция. Първо, нека обсъдим някои други роднини.

***10.5 Роднини на всеки***

Enumerable ви предлага няколко метода, които са подобни на всеки от тях

те преминават през цялата колекция и добиват елементи от нея, като не спират до

те са преминали целия път (и в един случай, дори и тогава!). Всеки член на

това семейство методи има своя специфична семантика и ниша. Методите

включват reverse\_each, each\_with\_index, each\_slice, each\_cons, цикъл и инжектиране.

Ще ги разгледаме в този ред.

***10.5.1 reverse\_each***

Методът reverse\_each прави това, което звучи така: той се повтаря назад

чрез безброй. Например кода

[1,2,3] .reverse\_each {| e | поставя e \* 10}

произвежда този резултат:

30

20.

10

Трябва да внимавате с reverse\_each: не го използвайте на безкраен итератор, тъй като

Концепцията за движение в обратна посока зависи от концепцията за знанието на последния елемент

е - което е безсмислено понятие в случай на безкраен итератор. Опитайте да се обадите

reverse\_each на екземпляр от клас Die, показан по-рано, но бъдете готови да натиснете Ctrl-c

за да се измъкнем от безкрайния цикъл.

**Минимална двойка, по ключ**

**Същото като мин**

**Минимална двойка по стойност**

|  |
| --- |
| **Страница 301** |

**301**

***Роднини на всеки***

***10.5.2 Методът всеки\_с\_индекс (и всеки.с\_индекс)***

Безброй # всеки\_с\_индекс се различава от всеки по това, че дава допълнителен елемент всеки

време през колекцията: а именно цяло число, представляващо поредното положение на

елемента. Този индекс може да бъде полезен за етикетиране на обекти, наред с други цели:

>> имена = ["Джордж Вашингтон", "Джон Адамс", "Томас Джеферсън",

"Джеймс Медисън"]

=> ["Джордж Вашингтон", "Джон Адамс", "Томас Джеферсън",

"Джеймс Медисън"]

>> имена.each\_with\_index do | pres, i |

?> поставя „# {i + 1}. # {pres}“

>> край

1. Джордж Вашингтон

2. Джон Адамс

3. Томас Джеферсън

4. Джеймс Медисън

Във всеки\_с индекса е включена аномалия: всеки обект, който може да се изброява, го има, но не

всеки изброен обект има знания за това какво е индекс. Можете да видите това, като попитате

изброими за извършване на операция всеки\_индекс (за разлика от всяка\_с\_индекс).

Резултатите варират от едно до друго:

>>% w {abc} .each\_index {| i | поставя i}

0

1

2

=> ["a", "b", "c"]

Следователно масивите имат основно усещане за индекс. Хешовете не го правят - макар че го правят

имам усещане за с индекс:

>> букви = {"a" =>"ay", "b" =>"пчела", "c" =>"виж"}

=> {"a" =>"ay", "b" =>"пчела", "c" =>"виж"}

>> букви.each\_with\_index {| (ключ, стойност), i | поставя i}

0

1

2

=> {"a" =>"ay", "b" =>"пчела", "c" =>"виж"}

>> букви.each\_index {| (ключ, стойност), i | поставя i}

NoMethodError: неопределен метод `each\_index 'за {" a "=>" ay ",

"b" =>"пчела", "c" =>"виж"}: Хеш

Бихме могли да твърдим, че ключовете на хеш са неговите индекси и че поредните номера са общи

от итерацията every\_with\_index са допълнителни или метаиндекси. Това е интересно

теоретичен въпрос; но на практика това в крайна сметка няма голямо значение, защото е така

изключително необичайно е да се налага да се извършва операция всеки\_с\_индекс на хеш.

Изброими # всеки\_с\_индекс работи, но е донякъде остарял. Вместо това,

sider, използвайки метода #with\_index на изброителя, който получавате обратно при извикване

всеки. Вече сте виждали тази техника в глава 9:

>> масив =% w {червено жълто синьо}

=> ["червено", "жълто", "синьо"]

**Добавя 1 за избягване**

**0-ти запис в списъка**

|  |
| --- |
| **302 серия** |

**302**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

>> масив.each.with\_index прави | цвят, i |

?> поставя "Цветовият номер # {i} е # {color}."

>> край

Това е толкова просто, колкото промяната на долната черта на период ... макар че има малко повече

към него под капака, както ще видите, когато научите малко повече за изброителите

по късно. (Вижте раздел 10.11.2 за повече информация за всеки\_индекс.) Използването на всеки\_индекс също купува

имате някаква функционалност: можете да предоставите аргумент, който ще се използва като първи

стойност на индекса, като по този начин се избягва необходимостта от добавяне на такава към индекса в случай като предишния

нашия списък на президенти:

>> имена.each.with\_index (1) do | pres, i |

?> поставя „# {i} # {pres}“

>> край

Друго подсемейство на всеки роднина е двойката методи each\_slice и each\_cons.

***10.5.3 Методите each\_slice и each\_cons***

Методите each\_slice и each\_cons са специализации на всеки, който ходи

чрез колекция определен брой елементи наведнъж, давайки масив от това

много елементи към блока на всяка итерация. Разликата между тях е тази

всеки\_слойк обработва всеки елемент само веднъж, докато всеки\_конз взема нова група-

при всеки елемент и по този начин се получават припокриващи се получени масиви.

Ето илюстрация на разликата:

>> масив = [1,2,3,4,5,6,7,8,9,10]

=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> array.each\_slice (3) {| срез | p филия}

[1, 2, 3]

[4, 5, 6]

[7, 8, 9]

[10]

=> нула

>> array.each\_cons (3) {| минуси | p минуси}

[1, 2, 3]

[2, 3, 4]

[3, 4, 5]

[4, 5, 6]

[5, 6, 7]

[6, 7, 8]

[7, 8, 9]

[8, 9, 10]

=> нула

Операцията each\_slice дава колекцията прогресивно в резени с размер *n* (или по-малко

от *n* , ако останат по-малко от *n* елемента). За разлика от тях, всеки\_кон се движи през

събиране по един елемент в даден момент и във всяка точка дава масив от *n* елемента, стоп-

ping, когато последният елемент в колекцията е получен веднъж.

Още един общ начин за итерация през изброима е с метода цикъл.

|  |
| --- |
| **303 серия** |

**303**

***Роднини на всеки***

***10.5.4 Циклов метод***

Изчислим # цикъл дава всички елементи в обекта отново и отново в цикъл. Ако

предоставите аргумент с цяло число, цикълът ще се изпълнява толкова много пъти. Ако не го направиш

ще се изпълнява безкрайно.

Можете да използвате цикъл, за да решите динамично колко пъти искате да повторите

чрез колекция - по същество, колко всякакви подобни изпълнения искате да извършите

последователно. Ето пример, включващ тесте игрални карти:

клас PlayingCard

КОСТЮМИ =% w {клубове диаманти сърца пика}

РАНГОВЕ =% w {2 3 4 5 6 7 8 9 10 JQKA}

клас Дек

attr\_reader: карти

def инициализиране (n = 1)

@cards = []

SUITS.cycle (n) do | s |

RANKS.cycle (1) do | r |

@cards <<"# {r} от # {s}"

край

край

край

край

край

Класът PlayingCard определя константи, представляващи костюми и класира B , докато

PlayingCard :: Deck клас моделира тестето. Картите се съхраняват в масив в тестето

Примерна променлива @cards, налична и като четец атрибут c . Благодарение на цикъла, това е

лесно да се организира възможността за комбиниране на две или повече палуби. Deck.new отнема

аргумент, по подразбиране 1 d . Ако замените по подразбиране, процесът, чрез който

Попълва се масив @cards, който се увеличава.

Например, тази команда създава двойно тесте карти, съдържащи две от

всяка карта за общо 104:

deck = PlayingCard :: Deck.new (2)

Това е така, защото методът преминава два пъти през костюмите, като преминава през редиците

веднъж на итерация на костюма д . Цикълът на ранговете винаги се прави само веднъж f ; цикъл (1) е, в

ефект, друг начин да се каже всеки. За всяка пермутация, нова карта, представена от a

описателен низ, се вмъква в тестето g .

Последното в обиколката на метода за всяко семейство е инжектиране, известно още като намаляване.

***10.5.5 Значително намаляване с инжектиране***

Методът на инжектиране (известен също като метод за намаляване и подобен на методите за „сгъване“ във функционалността

languages) работи чрез инициализиране на обект акумулатор и след това итерация през

колекция (изброим обект), извършваща изчисление за всяка итерация

и възстановяване на акумулатора за целите на следващата итерация до резултата от

това изчисление.

б

° С

д

д

е

ж

|  |
| --- |
| **Страница 304** |

**304**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Класическият пример за инжектиране е сумирането на числа в масив. Ето

как да го направим:

>> [1,2,3,4] .inject (0) {| acc, n | в съответствие + n}

=> 10

И ето как работи:

1

Акумулаторът се инициализира до 0, благодарение на аргумента 0 за инжектиране.

2

Първият път през итерацията - кодовият блок - е равен на 0 и n е зададен на 1

(първият елемент в масива). Резултатът от изчислението вътре в блока е 0 +

1 или 1.

3

Вторият път, acc е зададен на 1 (резултатът на блока от предишния

време през), а n е зададено на 2 (вторият елемент в масива). Блокът

следователно оценява на 3.

4

Третият път, acc и n са 3 (резултат от предишния блок) и 3 (следващ

стойност в масива). Блокът оценява на 6.

5

Четвъртият път, acc и n са 6 и 4. Блокът оценява на 10.

Тъй като това е последният път, стойността от блока служи като

връща стойност на цялото повикване за инжектиране. По този начин цялото повикване се оценява на 10, като

показано от irb.

Ако не предоставите аргумент за инжектиране, той използва първия елемент в изброеното

обект като начална стойност за съгл. В този пример това би довело до същия резултат,

тъй като първата итерация добави 0 към 1 и така или иначе е съобразена с 1.

Ето един измислен пример, с някои коментари, отпечатани на всяка итерация

за да можете да видите какво се случва:

>> [1,2,3,4] .inject do | acc, n |

поставя "добавяне на # {acc} и # {n} ... # {acc + n}"

в съответствие с n

край

добавяне на 1 и 2 ... 3

добавяне на 3 и 3 ... 6

добавяне на 6 и 4 ... 10

=> 10

Изявлението пут е чист страничен ефект (и сам по себе си оценява на нула), така че вие

все още трябва да завършите блока с ac + n, за да сте сигурни, че блокът оценява на съответния

правоъгълна стойност.

Запазихме може би най-важния роднина на всеки за последно: Безброй # карта.

***10.6 Методът на картата***

Методът на картата (извиква се също като събиране) е един от най-мощните и важни

tant изброими или операции по събиране, налични в Ruby. Срещали сте го и преди (в

глава 6), но има какво още да се види, особено сега, когато сме вътре в общата тема

на изброяване.

|  |
| --- |
| **Страница 305** |

**305**

***Методът на картата***

С каквото и изброяване да започне, map винаги връща масив. Върнатият масив

винаги е със същия размер като оригиналния брой. Неговите елементи се състоят от

закъснял резултат от извикване на кодовия блок на всеки елемент от оригиналния обект на свой ред.

Например, ето как картографирате масив от имена в техните главни еквиваленти:

>> имена =% w {Дейвид Юкихиро Чад Ейми}

=> ["Дейвид", "Юкихиро", "Чад", "Ейми"]

>> names.map {| име | name.upcase}

=> ["DAVID", "YUKIHIRO", "CHAD", "AMY"]

Новият масив е със същия размер като оригиналния масив и всеки от неговите елементи съответства

спондира към елемента в същото положение в оригиналния масив. Но всеки елемент има

са преминали през блока.

Може да е очевидно, но е важно да се отбележи, че това, което има значение за картата, е нейната възвръщаема стойност.

***10.6.1 Връщаната стойност на картата***

Възвръщаемата стойност на картата и полезността на тази възвръщаема стойност е това, което отличава

карта от всеки. Възвръщаемата стойност на всеки няма значение. Почти никога няма да видите това:

резултат = масив.each {| x | # код тук ...}

Защо? Защото всеки връща своя приемник. Можете също така да направите това:

резултат = масив

array.each {| x | ...}

От друга страна, map връща нов обект: картографиране на оригиналния обект в a

нов обект. Така че често ще виждате - и правите - неща като това:

резултат = array.map {| x | # код тук ...}

Разликата между картата и всяка една е добро напомняне, че всяка съществува чисто за

страничните ефекти от изпълнението на блока. Стойността, върната от блока всеки

времето се изхвърля. Ето защо всеки връща своя приемник; няма

нещо друго да се върне, защото не е спасило нищо. карта, от друга страна,

задържа масив от акумулатор на резултатите от блока.

Това не означава, че картата е по-добра или по-полезна от всяка. Това означава, че те са различни

ferent по някои важни начини. Но семантиката на map наистина означава, че трябва да бъдете

внимавайте за страничните ефекти, които правят всеки полезен.

Използване на символен аргумент като блок

Можете да използвате символ като: upcase с & пред него в метод-аргумент

позиция и резултатът ще бъде същият, както ако сте използвали кодов блок, който извиква

метод със същото име като символа на всеки елемент. По този начин можете да пренапишете

блока в последния пример, който извиква upcase за всеки елемент, по следния начин:

names.map (&: upcase)

Ще видите задълбочено обяснение на този идиом, когато прочетете за извикващи се обекти

в глава 14.

|  |
| --- |
| **306 серия** |

**306**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Б Е ВНИМАТЕЛНО С БЛОКОВА ОЦЕНКА

Погледнете този код и вижте дали можете да предскажете какво ще съдържа резултатът от масива

когато кодът се изпълнява:

масив = [1,2,3,4,5]

резултат = array.map {| n | поставя n \* 100}

Отговорът е, че резултатът ще бъде следният:

[нула, нула, нула, нула, нула]

Защо? Тъй като възвръщаемата стойност на путове винаги е нула. Това е всичко, за което се грижи картата. Да,

петте стойности, представени с n \* 100, ще бъдат отпечатани на екрана, но това е така

кодът в блока се изпълнява. Резултатът от операцията - картографирането

себе си - е всичко нула, защото всяко повикване към този конкретен блок ще връща нула.

На място има версия на карта за масиви и набори: map! (известен още като събиране!).

***10.6.2 Картографиране на място с карта!***

Помислете отново за масива с имена:

имена =% w {Дейвид Юкихиро Чад Ейми}

За да промените масива от имена на място, прекарайте го през map !, разрушителната версия на map:

names.map! (&: upcase)

Картата! методът на Array е дефиниран в Array, а не в Enumerable. Защото картата е

обикновено връщат масиви, независимо от класа на техния приемник, като правят ин-

map maping няма смисъл, освен ако обектът вече не е масив. Би било

трудно е, например, да си представим какво би се съставило на място картографиране на диапазон

на. Но картата Set #! метод прави на място картографиране на набор обратно към себе си - което

има смисъл, като се има предвид, че даден набор в много отношения е подобен на масив.

След това ще разгледаме клас, който не може да се изброи: String. Струните са малко

като диапазони в това, че те се държат и не се държат като колекции. В случай на диапазони,

техните свойства, подобни на колекцията, са достатъчни, че класът гарантира смесването на

Изброими. В случай на струни, Enumerable не е в игра; но семантиката на низовете,

когато ги третирате като повторяеми последователности от символи или байтове, е достатъчно подобен на

многобройна семантика, която ще разгледаме тук.

***10.7 Низове като квазиизброими***

Можете да прегледате необработените байтове или символите на низ, като използвате удобно

итератор методи, които третират низа като колекция от байтове, символи, кодови точки,

или линии. Всеки от тези четири начина за итерация през низ има всеки стил

метод, свързан с него. За да прегледате байтове, използвайте всеки\_байт:

str = "abcde"

str.each\_byte {| b | pb}

**Вижте съвет, по-горе!**

|  |
| --- |
| **307 серия** |

**307**

***Струни като квазиизброими***

Резултатът от този код е

97

98

99

100

101

Ако искате всеки символ, а не неговия байтов код, използвайте each\_char:

str = "abcde"

str.each\_char {| c | настолен компютър }

Този път изходът е

"а"

"б"

"° С"

"д"

"д"

Итерацията по кодова точка предоставя кодове на символи (цели числа) със скорост точно една

на знак:

>> str = "100 \ u20ac"

=>"100 €"

>> str.each\_codepoint {| cp | p cp}

49

48

48

8364

Сравнете този последен пример с това, което се случва, ако прегледате същия байт на низ

по байт:

>> str.each\_byte {| b | pb}

49

48

48

226

130

172

Поради кодирането, броят на байтовете е по-голям от броя на кодовите точки

(или броят на знаците, който е равен на броя на кодовите точки).

И накрая, ако искате да преминете ред по ред, използвайте each\_line:

str = "Този низ \ nима три \ nлинии"

str.each\_line {| l | поставя "Следващ ред: # {l}"}

Резултатът от този пример е

Следващ ред: Този низ

Следващ ред: има три

Следващ ред: редове

|  |
| --- |
| **308 серия** |

**308**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Низът се разделя в края на всеки ред - или, по-строго казано, при всяко възникване -

rence на текущата стойност на глобалната променлива $ /. Ако промените тази променлива, вие сте

промяна на разделителя за това, което Ruby счита за следващия ред в низ:

str = "Дейвид! Алън! Черен"

$ / = "!"

str.each\_line {| l | поставя "Следващ ред: # {l}"}

Сега концепцията на Ruby за „линия“ ще се основава на! характер:

Следващ ред: Дейвид!

Следващ ред: Алън!

Следващ ред: Черен

Въпреки че Ruby струните не са изброими в технически смисъл (String не

включват Enumerable), по този начин езикът ви предоставя необходимите инструменти за трасиране

стих ги като символи, байтове, кодови точки и / или редови колекции, когато трябва.

Описаните тук четири метода за всеки стил работят чрез създаване на изброител.

Ще научите повече за изброителите в раздел 10.9. Важният урок за

моментът е, че имате друг набор от опции, ако просто искате масив от всички

байтове, символи, кодови точки или редове: пуснете всеки\_ и плурализирайте метода

име. Например, ето как бихте получили масив от всички байтове в низ:

низ = "Здравей"

p string.bytes

Изходът е

[72, 101, 108, 108, 111]

Можете да направите по същия начин с методите символи , кодови точки и линии .

Търсихме, трансформирахме, филтрирахме и разпитвахме различни обекти за колекция

използвайки още по-голямо разнообразие от методи. Единственото нещо, за което не съм направил е *нещо като* колективното

ния. Ще направим това по-нататък.

***10.8 Сортиране на изброими***

Ако имате клас и искате да можете да подредите няколко екземпляра от него по ред,

трябва да направите следното:

1

Дефинирайте метод за сравнение за класа (<=>).

2

Поставете множество екземпляри в контейнер, вероятно масив.

3

Сортирайте контейнера.

Ключовият момент е, че въпреки че възможността за сортиране се предоставя от Enumerable, вашият клас

не трябва да се смесва в Enumerable. По-скоро поставяте обектите си в контейнер

обект, който се смесва в Enumerable. Този обект контейнер, като изброима, има две

методи за сортиране, sort и sort\_by, които можете да използвате за сортиране на колекцията.

В по-голямата част от случаите контейнерът, в който поставяте обекти, които искате

сортирано ще бъде масив. Понякога това ще бъде хеш, в този случай резултатът ще бъде

|  |
| --- |
| **309 серия** |

**309**

***Сортиране на изброими***

масив (масив от двуелементни масиви двойка ключ / стойност, сортирани по ключ или някои

друг критерий).

Обикновено не е необходимо да създавате масив от елементи изрично, преди да сортирате

тях. По-често сортирате колекция, която вашата програма вече е генерирала автоматично

матично. Например можете да извършите операция за избор на колекция от обекти

и сортирайте избраните от вас. Ръчното напълване на списъци с обекти на квадрат

скоби за създаване на примери за масиви в този раздел следователно е малко измислено. Но

целта е да се съсредоточим директно върху техниките за сортиране и това ще направим.

Ето един прост пример за сортиране, включващ масив от цели числа:

>> [3,2,5,4,1] .sort

=> [1, 2, 3, 4, 5]

Това е лесно, когато имате числа или дори низове (където сортирането ви дава

азбучен ред). Масивът, в който ги поставяте, има механизъм за сортиране и интеграцията

gers или струните имат известни познания за това какво означава да си в ред.

Но какво, ако искате да сортирате, да речем, масив от Painting обекти?

>> [pa1, pa2, pa3, pa4, pa5] .sort

За да имате картините достатъчно знания, за да участвате в операция за сортиране, имате

за да дефинирате оператора на космическия кораб (вижте раздел 7.6.2): ​​Painting # <=>. Всяка картина ще

тогава знайте какво означава да бъдете по-големи или по-малки от друга картина и това ще даде възможност

масива за сортиране на съдържанието му. Не забравяйте, че това е масивът, който сортирате, а не всяка боя -

ing; но за да сортират масива, неговите елементи трябва да имат представа за това как се сравняват

взаимно. (Не е нужно да смесвате модула за сравнение; просто се нуждаете от

метод на космически кораб. Скоро ще се върнем към Сравнимо.)

Да предположим, че искате картините да се сортират в нарастващ ред на цената, и да предположим

картините имат атрибут цена. Някъде в класа си по живопис бихте направили това:

def <=> (друго\_ рисуване)

self.price <=> other\_painting.price

край

Сега всеки масив от картини, които сортирате, ще излезе в сортиран по цена ред:

сортирано\_цена = [pa1, pa2, pa3, pa4, pa5] .sort

Ruby прилага теста <=> към тези елементи, по два наведнъж, изграждайки достатъчно информация

мация за извършване на пълното сортиране.

По-подробен разказ за участващите стъпки може да изглежда така:

1

Научете предметите си как да се сравняват помежду си, като използвате <=>.

2

Поставете тези обекти в изброим обект (вероятно масив).

3

Помолете този обект да се сортира сам. Това се прави, като се иска от обектите да ги сравняват -

себе си един с друг с <=>.

Ако имате предвид това разделение на труда, ще разберете как работи и сортирането

как се отнася към Изброими. Но какво ще кажете за сравним?

|  |
| --- |
| **310 серия** |

**310**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

***10.8.1 Къде Сравнимият модул се вписва в безброй сортиране***

***(или не)***

Когато за първи път се сблъскахме с оператора на космически кораб, това беше в контекста на включването

Сравним и позволява на този модул да изгради своите различни методи (>, <и т.н.) отгоре

от <=>. Но при подготовката на обектите за сортиране в безброй контейнери имаме всичко

готово е дефиниране <=>; не сме се смесвали в Сравнимо.

Цялата картина се вписва заедно, ако я разглеждате като няколко отделни, слоести техники:

Ако дефинирате <=> за клас, тогава копията на този клас могат да бъдат поставени вътре в

масив или друго изброимо за сортиране.

Ако не дефинирате <=>, пак можете да сортирате обекти, ако ги поставите в масив

и осигурете кодов блок, който казва на масива как трябва да класира всеки два от

обекти. (Това е обсъдено по-нататък в раздел 10.8.2.)

Ако дефинирате <=> и също включите Comparable във вашия клас, тогава ще получите сортиране-

възможност в масив *и* можете да извършвате всички операции за сравнение

между всеки два от вашите обекти (>, <и т.н.), според обсъждането на

Сравним в глава 9.

С други думи, методът <=> е полезен както за класове, чиито екземпляри искате

сортиране и за класове, чиито екземпляри искате да сравните помежду си в повече

фин начин, използващ пълния набор от оператори за сравнение.

Обратно се връщаме към безброй сортиране - и по-специално към варианта на сортиране

където предоставяте кодов блок вместо метод <=>, за да укажете как трябва да бъдат обектите

да бъдат сравнени и подредени.

***10.8.2 Определяне на логика за сортиране с блок***

В случаите, когато за тези обекти не е дефиниран метод <=>, можете да предоставите блок

в движение, за да посочите как искате да бъдат сортирани вашите обекти. Ако има метод <=>, вие

може да го замени за текущата операция за сортиране, като предостави блок.

Да кажем например, че сте дефинирали Painting # <=> по такъв начин, че да сортира

по цена, както по-рано. Но сега искате да сортирате по години. Можете да принудите сортиране по година

с помощта на блок:

year\_sort = [pa1, pa2, pa3, pa4, pa5] .sort do | a, b |

a.year <=> b.year

край

Блокът взема два аргумента, a и b. Това дава възможност на Ruby да използва блока колкото се може повече

пъти, колкото е необходимо за сравняване на една картина с друга. Кодът вътре в блока

прави <=> сравнение между съответните години на двете картини. За това обаждане

за сортиране се използва кодът в блока вместо кода в метода <=> на

Клас по рисуване.

Можете да използвате тази форма на сортиране на кодов блок за обработка на случаи, когато вашите обекти не

имат <=> метод и следователно не знаят как да се сравняват с всеки

други. Може да бъде полезен и когато сортираните обекти са от различни класове

|  |
| --- |
| **Страница 311** |

**311**

***Преброители и следващото измерение на изброяемостта***

и по подразбиране не знаят как да се сравняват помежду си. Цели числа и

низове, например, не могат да се сравняват директно: израз като "2"<=> 4 причинява a

фатална грешка. Но ако първо направите преобразуване, можете да го извършите:

>> ["2", 1,5, "3", 4, "6"]. Сортиране {| a, b | a.to\_i <=> b.to\_i}

=> [1, "2", "3", 4, 5, "6"]

Елементите в сортирания изходен масив са същите като тези във входния масив: a

смес от низове и цели числа. Но те са подредени, както биха били, ако бяха всички

цели числа. Вътре в кодовия блок и низовете, и целите числа се нормализират на цяло число

форма с to\_i. Що се отнася до механизма за сортиране, той извършва сортиране въз основа на

поредица от целочислени сравнения. След това прилага реда, който е измислил, към произхода

nal масив.

сортирането с блок по този начин може да ви помогне там, където съществуващите методи за сравнение няма

свърши работата. И има още по-кратък начин за сортиране на колекция с

кодов блок: методът sort\_by.

***10.8.3 Кратко сортиране с sort\_by***

Подобно на sort, sort\_by е метод на екземпляр на Enumerable. Основната разлика е в това

sort\_by винаги взема блок и изисква само да му покажете как да го третирате

артикул в колекцията. sort\_by установява, че искате да направите едно и също нещо и на двамата

елементи всеки път, когато сравнява чифт обекти.

Предишният пример за сортиране на масив може да бъде написан по този начин, като се използва sort\_by:

>> ["2", 1,5, "3", 4, "6"]. Sort\_by {| a | a.to\_i}

=> [1, "2", "3", 4, 5, "6"]

Всичко, което трябва да направим в блока, е да покажем (веднъж) към какво действие трябва да се извърши

подгответе всеки обект за операцията за сортиране. Не е нужно да извикваме to\_i за два обекта; нито

трябва ли да използваме изрично метода <=>.

В допълнение към модула Enumerable и все още в сферата на изброяването,

Ruby предоставя клас, наречен Enumerator. Преброителите добавят цяло измерение на col-

мощност за манипулиране на лекция към Ruby. Сега ще ги разгледаме задълбочено.

***10.9 Изброители и следващото измерение на изброяването***

Преброителите са тясно свързани с итератори, но те не са едно и също нещо. Итера-

tor е метод, който дава една или повече стойности на кодов блок. Преброителят е

*обект* , а не метод.

В основата си преброителят е прост изброим обект. Той има всеки метод,

и той използва модула Enumerable, за да дефинира всички обичайни методи - изберете,

инжектирайте, карта и приятели - директно върху всяка от тях.

Обратът в сюжета обаче е как е проектиран всеки метод на изброителя.

Преброителят не е обект на контейнер. Той няма „естествена” основа за всяка операция

ция, начина, по който го прави масивът (започнете от елемент 0; дайте го; преминете към елемент 1; дайте го; и така

На). Всяка итерационна логика на всеки изброител трябва да бъде изрично посочена. След

**Или сортиране\_ по (&: to\_i)**

|  |
| --- |
| **312 серия** |

**312**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

вие сте му казали как да направите всеки, изброителят поема от там и измисля

как да направите карта, да намерите, вземете, пуснете и всичко останало.

Преброителят е като мозък в научно-фантастичен филм, седнал на маса без

връзка с тяло, но все пак способно да мисли. Просто се нуждае от алгоритъм „всеки“, така че да го направи

може да задейства нещата, които вече знае как да прави. И това може да научи в едно

от два начина: или извиквате Enumerator.new с кодов блок, така че кодовият блок

съдържа всяка логика, която искате да следва преброителят; или създавате списък-

tor, базиран на съществуващ обект, който може да се изброява (масив, хеш и т.н.) в такъв

начин, по който всеки метод на изброителя извлича своите елементи за итерация от спе

конкретен метод за този изброим обект.

Ще започнем, като разгледаме подхода на блока на кода за създаване на изброители. Но

по-голямата част от дискусията на изброителите ще се съсредоточи върху втория подход,

където „свързвате“ преброител с итератор на друг обект. (Ако намерите

блокова техника, която е трудна за следване, няма да навреди, ако прегледате раздел 10.9.1

засега и се съсредоточете върху раздел 10.9.2.) Кои техники използвате и как

bine тях в крайна сметка ще зависи от вашите точни нужди в дадена ситуация.

***10.9.1 Създаване на изброители с кодов блок***

Ето един прост пример за създаване на екземпляр на преброител с кодов блок:

e = Enumerator.new do | y |

y << 1

y << 2

y << 3

край

Първо, първо: какво е y?

y е *yielder* , екземпляр на Enumerator :: Yielder, който автоматично се предава на вашия

блок. Yielders капсулират сценария за добив, към който искате вашият изброител

последвам. В този пример казваме, *когато вие (изброителят) получите* всяко *обаждане,*

*моля, вземете това, за да означава, че трябва да дадете 1, след това 2, след това 3* . Методът << (в инфикс

позиция на оператора, както обикновено) служи за инструктиране на подрастващия за това какво трябва да даде.

(Можете също така да напишете y.yield (1) и т.н., въпреки че приликата на добива

метод към ключовата дума yield може да е по-объркващ, отколкото си струва.) След като сте

помолен да повтори, изброителят се консултира с по-младия и прави следващия ход -

следващ добив - въз основа на инструкциите, които е съхранил Yielder.

Какво се случва, когато използвате e, изброителят? Ето една irb сесия, където е

пуснати през неговите крачки (като се има предвид, че кодът в примера вече е изпълнен):

>> e.to\_a

=> [1, 2, 3]

>> e.map {| x | x \* 10}

=> [10, 20, 30]

>> д. изберете {| x | x> 1}

=> [2, 3]

>> д. вземете (2)

=> [1, 2]

**Представяне на масив на получени елементи**

**Картиране, въз основа на всеки**

**Избор, въз основа на всеки**

**Взема първите два елемента отстъпва**

|  |
| --- |
| **Страница 313** |

**313**

***Преброители и следващото измерение на изброяемостта***

Преброителят e е изчислителна машина. Не съдържа обекти; има код

свързан с него - оригиналният кодов блок - който му казва какво да прави, когато е адресиран

по отношение на термините, които той разпознава като идващи от модула Enumerable.

Изброителят се повтаря веднъж за всеки път, когато << (или методът на добив) е

- извика младшият. Ако поставите повиквания към << вътре в цикъл или друг итератор вътре в

код, можете да въведете почти всяка итерационна логика, която искате. Ето пренаписване

от предишния пример, използвайки итератор вътре в блока:

e = Enumerator.new do | y |

(1..3) .всеки {| i | y << i}

край

Поведението на e ще бъде същото, като се има предвид тази дефиниция, както е в предишния изпит -

плес. Уредили сме << да се извика три пъти; това означава, че e.each ще направи три

итерации. Отново, поведението на изброителя може да бъде проследено в крайна сметка до повикванията

до << вътре в кодовия блок, с който е инициализиран.

Обърнете специално внимание, че не отстъпвате от блока; тоест *не* правите това:

e = Enumerator.new do

добив 1

добив 2

добив 3

край

По-скоро попълвате вашия yielder (y, в първите примери) със спецификации за

как искате итерацията да продължи по време, когато извикате итеративен метод

на преброителя.

Всеки път, когато извикате метод на итератор в преброителя, кодовият блок получава exe-

излъчен веднъж. Всички променливи, които инициализирате в блока, се инициализират веднъж в началото на

всяко такова извикване на метод. Можете да проследите последователността на изпълнението, като добавите някои гласове-

ity и извиква множество методи:

e = Enumerator.new do | y |

поставя "Стартиране на блока!"

(1..3) .всеки {| i | y << i}

поставя "Излизане от блока!"

край

р е. до\_а

p д. изберете {| x | x> 2}

Резултатът от този код е

Стартиране на блока!

Излизане от блока!

[1, 2, 3]

Стартиране на блока!

Излизане от блока!

[3]

Можете да видите, че блокът се изпълнява веднъж за всеки итератор, извикан на e.

**Грешно!**

**Ето какво**

**ти не правиш!**

**Обадете се на to\_a**

**Обадете се, за да изберете**

|  |
| --- |
| **Страница 314** |

**314**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Възможно е също да включите други обекти в блока на кода за изброител.

Ето малко абстрактен пример, в който изброителят извършва изчисление-

ния, включващ елементите на масив, като същевременно премахвате тези елементи от

масив за постоянно:

a = [1,2,3,4,5]

e = Enumerator.new do | y |

общо = 0

докато а.празен?

общо + = a.pop

y << общо

край

край

Сега нека разгледаме съдбата на бедния a, в irb:

>> д. вземете (2)

=> [5, 9]

>> а

=> [1, 2, 3]

>> e.to\_a

=> [3, 5, 6]

>> а

=> []

Операцията take вземе резултат от масив от два елемента (стойността на total for

две последователни итерации) и оставя a с три елемента. Обаждам се to\_a на e, при това

point, кара отново оригиналния код да бъде изпълнен, защото извикването to\_a

не е част от същата итерация като обаждането. Следователно общата сума започва отново от 0,

и цикълът while се изпълнява с резултата, че се получават три стойности и a е

оставен празен.

Не е честно да се прави засада на отделен обект, като се отстраняват неговите елементи като страничен ефект от

извикване на преброител. Но примерът ви показва механизма - и той също така предлага

добавя разумен преглед към другата половина на темата за създаване на изброители: cre-

привличане на изброители, чиито всеки метод е обвързан със специфични методи на съществуващи

изброими обекти.

***10.9.2 Прикачване на изброители към други обекти***

Другият начин да дадете на изброител с всяка логика е да свържете преброителя

към друг обект - по-конкретно, към итератор (често всеки, но потенциално всеки метод

което дава една или повече стойности) за друг обект. Това дава основа на преброителя

за собствена итерация: когато трябва да даде нещо, той получава необходимата стойност до

задействане на следващия добив от обекта, към който е прикрепен, чрез посочения

метод. По този начин изброителят действа като частичен прокси, частичен паразит, като определя всеки свой собствен

по отношение на итерация на друг обект.

Създавате изброител с този подход, като извикате enum\_for (известен още като to\_enum)

върху обекта, от който искате изброителят да изготви своите итерации. Вие предоставяте

като първи аргумент името на метода, към който ще се прикачи изброителят

|  |
| --- |
| **Страница 315** |

**315**

***Преброители и следващото измерение на изброяемостта***

всеки метод. Този аргумент по подразбиране е: всеки, въпреки че е обичайно да се прикачва

enumerator към различен метод, както в този пример:

имена =% w {Дейвид Блек Юкихиро Мацумото}

e = names.enum\_for (: изберете)

Посочване: select като аргумент означава, че искаме да обвържем този изброител

методът за избор на масива от имена. Това означава, че всеки от изброителите ще служи като

един вид преден край за избор на масив:

д. всеки {| n | n.include? ('a')}

Можете също така да предоставите допълнителни аргументи на enum\_for. Всички такива аргументи се предават

до метода, към който е прикрепен изброителят. Например,

ето как да създадете изброител за инжектиране, така че при инжектиране да се извика

стойности за подаване към всяка от изброителя, тя се извиква с начална стойност на "Имена:":

>> e = names.enum\_for (: инжектиране, "Имена:")

=> # <Изброител: ["Дейвид", "Черен", "Юкихиро", "Мацумото"]: инжектиране ("Имена:")>

>> e.each {| низ, име | низ <<"# {name} ..."}

=>"Имена: Дейвид ... Блек ... Юкихиро ... Мацумото ..."

Но внимавай! Този начален низ „Имена:“ е добавил някои имена към него, но е така

все още жив в преброителя. Това означава, че ако изпълните една и съща операция за инжектиране

отново се добавя към същия низ (редът в изхода в следния код е счупен

през два реда, за да го побере):

>> e.each {| низ, име | низ <<"# {name} ..."}

=>"Имена: Дейвид ... Блек ... Юкихиро ... Мацумото ...

Дейвид ... Черен ... Юкихиро ... Мацумото ... "

Когато създавате изброителя, аргументите, които му давате с цел доставка-

Своят проксидиран метод с аргументи са аргументите - обектите, които той ще използва

постоянно. Така че следете за странични ефекти. (В този конкретен случай можете да избегнете страната

ефект чрез добавяне на низове — низ + "# {name} ..." - вместо добавяне към низа

с <<, тъй като операцията за добавяне създава нов низ обект. И все пак, предпазливостта-

ary tale обикновено е полезен.)

ЗАБЕЛЕЖКА Можете да извикате Enumerator.new (obj, method\_name, arg1, arg2 ...) като

еквивалент на obj.enum\_for (име\_на\_метод, arg1, arg2 ...). Но използвайки

тази форма на Enumerator.new се обезкуражава. Използвайте enum\_for за метода-

сценарий на прикачване и Enumerator.new за описания блоков сценарий

в раздел 10.9.1.

Сега знаете как да създадете изброители и от двата вида: от вида, чиито познания

как да се итерира, му се предава в кодов блок и вида, който получава това знание

от друг обект. Преброителите също се създават имплицитно, когато направите блок-

по-малко извиквания към определени итераторски методи.

**Изход: ["Дейвид",**

**"Черно", "Мацумото"]**

|  |
| --- |
| **316 серия** |

**316**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

***10.9.3 Имплицитно създаване на изброители чрез безблокови извиквания на итератори***

По дефиниция итераторът е метод, който дава една или повече стойности на блок. Но

ами ако няма блок?

Отговорът е, че повечето вградени итератори връщат преброител, когато са

извикана без блок. Ето пример от класа String: всеки\_байт

метод (виж раздел 10.7). Първо, ето класическо използване на итератор на метода, без

изброител, но с блок:

>> str = "Здравейте"

=>"Здравей"

>> str.each\_byte {| b | поставя b}

72

101

108

108

111

=>"Здравей"

всеки\_байт прелита байтовете в низа и връща своя приемник (низът). Но

ако извикате всеки\_байт без блок, ще получите изброител:

>> str.each\_byte

=> # <Изчислител: "Здравей": всеки\_байт>

Преброителят, който получавате, е еквивалентен на това, което бихте получили, ако направите това:

>> str.enum\_for (: всеки\_байт)

Ще откриете, че много методи от Enumerable return enumerators, когато се обадите

тях без блок (включително всеки, карта, избор, инжектиране и други). Основното

случай на употреба за тези автоматично върнати изброители е *верига* : извикване на друг

метод незабавно на изброителя. Ще разгледаме веригирането като част от корицата-

възраст на семантиката на изброител в следващия раздел.

***10.10 Семантика и употреби на изброител***

Сега, след като знаете как се свързват изброителите и как да ги създадете, ще продължим

за да разгледаме как се използват - и защо се използват.

Може би най-трудното нещо при изброителите, защото е най-трудно да се

Pret визуално, това е как нещата се развиват, когато извикате всеки метод. Ще започнем с търсене

на това; след това ще разгледаме практическите възможности на изброителите, по-специално начините в

кой изброител може да защити обект от промяна и как можете да използвате enu-

мератор за извършване на фино контролирани итерации. След това ще разгледаме как се вписват изброителите

като цяло във веригите от методи и ще видим няколко важни конкретни случая.

***10.10.1 Как да използваме всеки метод на изброител***

Всеки метод на изброител е свързан с метод на друг обект, вероятно a

метод, различен от всеки. Ако го използвате директно, той се държи като този друг метод,

включително по отношение на възвръщаемата му стойност.

|  |
| --- |
| **Страница 317** |

**317**

***Семантика и употреби на изброител***

Това може да доведе до някои странни резултати, при които повикванията към всяко връщане се филтрират,

сортирани или картографирани колекции:

>> масив =% w {котка куче заек}

=> ["котка", "куче", "заек"]

>> e = array.map

=> # <Изброител: ["котка", "куче", "заек"]: карта>

>> e.each {| животно | animal.capitalize}

=> ["Котка", "Куче", "Заек"]

Тук няма нищо загадъчно. Преброителят не е същият обект като масива; то

има свои собствени идеи за това какво означава всеки. Все пак общият ефект от свързването на enu-

merator към метода map на масив е, че получавате всяка операция с масив

картографиране като негова възвръщаема стойност. Както виждате, съществува всяка обичайна итерация на масив

главно за страничните му ефекти и връща приемника (масива). Но на преброител

всеки служи като вид проводник към метода, от който извлича своите стойности и

се държи по същия начин по отношение на връщаната стойност.

Друга характеристика на изброителите, която трябва да знаете е фактът, че

те изпълняват един вид *не-висш* от методи в Enumerable.

T HE ООН - по-висш ФЕНОМЕН

Ако клас определя всеки и включва Enumerable, неговите екземпляри автоматично получават карта,

изберете, инжектирайте и всички останали методи на Enumerable. Всички тези методи са

дефинирани по отношение на всеки.

Но понякога даден клас вече е заменил версията на a

метод със свой собствен. Добър пример е Hash # select. Стандартната, нестандартна

select метод от Enumerable винаги връща масив, независимо от класа на

обект, който го използва, може да бъде. От друга страна, операция за избор на хеш се връща

хеш:

>> h = {"котка" =>"котешка", "куче" =>"кучешка", "крава" =>"говеда"}

=> {"котка" =>"котешка", "куче" =>"кучешка", "крава" =>"говеда"}

>> h.select {| ключ, стойност | ключ = ~ / c /}

=> {"котка" =>"котешка", "крава" =>"говеда"}

Засега добре (и нищо ново). И ако закачим преброител към избрания

метод, той ни дава всеки метод, който работи като този метод:

>> e = h.enum\_for (: изберете)

=> # <Изброител: {"cat" =>"котешки", "куче" =>"кучешки", "крава" =>"говеда"}: изберете>

>> e.each {| ключ, стойност | ключ = ~ / c /}

=> {"котка" =>"котешка", "крава" =>"говеда"}

Но какво ще кажете за изброител, свързан не с метода за избор на хеш, а с

хеш всеки метод? Можем да го получим, като използваме to\_enum и оставим целевия метод

по подразбиране за всеки:

>> e = h.to\_enum

=> # <Изброител: {"cat" =>"котешки", "куче" =>"кучешки", "крава" =>"говеда"}: всеки>

**Се завръща**

**картографиране**

|  |
| --- |
| **Страница 318** |

**318**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Хеш # всеки, извикан с блок, връща хеша. Същото важи и за изброителя

всеки - защото това е само преден край на всеки хеш. Блоковете в тези примери

са празни, защото се занимаваме само с връщаните стойности:

>> ч. всеки {}

=> {"котка" =>"котешка", "куче" =>"кучешка", "крава" =>"говеда"}

>> e.each {}

=> {"котка" =>"котешка", "куче" =>"кучешка", "крава" =>"говеда"}

Засега изглежда, че всеки от преброителя е вграден за всеки хеш. Но какво

се случва, ако използваме това за извършване на избрана операция?

>> д. изберете {| ключ, стойност | ключ = ~ / c /}

=> [["котка", "котешка"], ["крава", "говеда"]]

Отговорът, както виждате, е, че получаваме обратно масив, а не хеш.

Защо? Ако e.each е привързан към h.each, как получава връщаната стойност на e.select

unpegged от връщаната стойност на h.select?

Ключът е, че повикването за избор в последния пример е повикване към метода за избор

на *преброителя* , а не хеш. И методът за избор на изброителя е изграден

директно върху всеки метод на изброителя. Всъщност методът на избор на изброителя

е Enumerable # select, който винаги връща масив *.* Фактът, че Hash # изберете

не връща масив не представлява интерес за изброителя.

В този смисъл изброяващият добавя изброеност към хеша, въпреки че

hash вече е изброим. Също така е неотменим Enumerable # select; избраното

предоставено от изброителя е Enumerable # select, дори ако изборът на хеш не е бил.

(Технически това не е отмяна, но създава усещането, че enu-

merator затваря логиката за избор на оригиналния хеш.)

Урокът е, че е важно да запомните, че преброителят е различен

обект от колекцията, от която той сифонира своите итерирани обекти. Въпреки че тази разлика

съотношението между обекти може да доведе до някои евентуално странни резултати, като избрано същество

пренасочен през модула Enumerable, определено е полезен поне в един

важен начин: достъп до колекция чрез преброител, а не чрез

самата колекция, защитава обекта на колекция от промяна.

***10.10.2 Защита на обекти с изброители***

Помислете за метод, който очаква, да речем, масив като свой аргумент. (Да, малко е не-Ruby-

бихте искали да се фокусирате върху класа на обекта, но ще видите, че това не е основната точка тук.)

def give\_me\_an\_array (масив)

Ако предадете обект на масив на този метод, методът може да промени този обект:

масив <<"нов елемент"

Ако искате да защитите оригиналния масив от промяна, можете да го дублирате и да го предадете

по дубликата - или вместо това можете да предадете изброител:

give\_me\_an\_array (array.to\_enum)

|  |
| --- |
| **319 серия** |

**319**

***Семантика и употреби на изброител***

Преброителят с радост ще позволи повторения през масива, но няма да поеме

промени. (Той ще отговори с фатална грешка, ако опитате да се обадите << на него.) С други думи,

изброителят може да служи като вид шлюз към обект за събиране, който позволява

итерация и изследване на елементи, но забранява разрушителните операции.

Кодът на тестето карти от раздел 10.5.4 предоставя добра възможност за някои

обектна защита. В този код класът Deck има четец атрибутни карти. Когато

deck е създадена, нейната променлива @cards instance се инициализира в масив, съдържащ всички

карти. Тук има уязвимост: Ами ако някой се добере до масива @cards

чрез атрибута четец на карти и го променя?

deck = PlayingCard :: Deck.new

deck.cards <<"JOKER !!"

В идеалния случай бихме искали да можем да четем от масива от карти, но не и да го променяме. (Бихме могли

замразете го с метода на замразяване, който предотвратява по-нататъшни промени на обекти, но

ще трябва да сменим тестето в класа Deck, когато е раздадено от него.) Enumera-

торси осигуряват решение. Вместо четец атрибут, нека направим метода на картите

връщане на изброител:

клас PlayingCard

КОСТЮМИ =% w {клубове диаманти сърца пика}

РАНГОВЕ =% w {2 3 4 5 6 7 8 9 10 JQKA}

клас Дек

def карти

@ cards.to\_enum

край

def инициализиране (n = 1)

@cards = []

SUITS.cycle (n) do | s |

RANKS.cycle (1) do | r |

@cards <<"# {r} от # {s}"

край

край

край

край

край

Все още е възможно да се вмъкнете в масива @cards и да го объркате, ако сте решени. Но

изброителят осигурява значително ниво на защита:

deck = PlayingCard :: Deck.new

deck.cards <<"Жокер !!"

Разбира се, ако искате кодът за повикване да може да адресира картите като масив,

връщането на изброител може да има обратен ефект. (И поне още една друга технология

nique защитава обекти при такива обстоятелства: return @ cards.dup.) Но ако е

добро прилягане, защитните качества на преброител могат да бъдат удобни.

Тъй като изброителите са обекти, те имат състояние. Освен това те използват своето състояние

да проследяват собствения си напредък, за да можете да спрете и да стартирате техните повторения. Ще разгледаме сега

техниките за контрол на изброителите по този начин.

**NoMethodError: недефиниран метод '<<'**

**за # <Изброител: 0x000001020643b8>**

|  |
| --- |
| **Страница 320** |

**320**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

***10.10.3 Фино повторена итерация с изброители***

Изброителите поддържат състояние: те проследяват къде се намират в своето изброяване.

Няколко метода използват директно тази информация. Помислете за този пример:

имена =% w {Дейвид Юкихиро}

e = names.to\_enum

поставя e.next

поставя e.next

д. назад

поставя e.next

Резултатът от тези команди е

Дейвид

Юкихиро

Дейвид

Преброителят ви позволява да се движите в забавен каданс, така да се каже, през изброяването-

ация на масива, спиране и рестартиране по желание. В това отношение това е като едно от тях

редактиране на таблици, където филмов редактор ръчно завърта филма. За разлика от проектор, който

включите се и го оставите да свърши своето, таблицата за редактиране ви позволява да повлияете на програмата

рес на филма, докато продължава.

Тази точка също хвърля светлина върху разликата между изброител и итератор

тор. Преброителят е обект и следователно може да поддържа състояние. То помни

където е в изброяването. Итераторът е метод. Когато го повикате, обаждането е

атомна; цялото обаждане се случва и след това приключва. Благодарение на кодовите блокове има

Разбира се, определена полезна сложност при извикванията на метода Ruby: методът може да се върне обратно

блока и могат да се вземат решения, които влияят на резултата. Но все пак това е метод.

Итераторът няма състояние. Изброителят е изброим обект.

Интересното е, че можете да използвате изброител на обект, който не се изброява. Всички вие

необходимо е вашият обект да има метод, който дава нещо, за да може изброителят

да приеме този метод като основа за своя собствен метод. В резултат на това неизброимите

обектът всъщност се изброява.

***10.10.4 Добавяне на изброеност с преброител***

Преброителят може да добави изброяване към обекти, които го нямат. Въпрос на wir-

ing: ако свържете всеки метод на преброител към който и да е итератор, тогава можете да използвате

enumerator за извършване на изброими операции върху обекта, който притежава итератора,

независимо дали този обект се смята за изброим или не.

Когато свържете преброител към метода String # bytes, вие сте ефективно

добавяне на изброеност към обект (низ), който го няма, в смисъл че

String не се смесва в Enumerable. Можете да постигнете почти същия ефект с класове

на собствения си. Помислете за следния клас, който не се смесва в Enumerable, но го прави

има един метод на итератор:

модул Музика

клас Скала

ЗАБЕЛЕЖКИ =% w {cc # dd # eff # gaa # b}

|  |
| --- |
| **321 серия** |

**321**

***Семантика и употреби на изброител***

def play

ЗАБЕЛЕЖКИ.всеки {| забележка | бележка за добива}

край

край

край

Като се има предвид този клас, е възможно да се извърши итерация през нотите на скала

scale = Music :: Scale.new

scale.play {| бележка | поставя "Следващата бележка е # {бележка}"}

с резултата

Следващата бележка е c

Следващата бележка е c #

Следващата бележка е d

и така нататък. Но мащабът не е технически изброим. Стандартните методи от

Enumerable няма да работи, защото класът Music :: Scale не се смесва в Enumerable и

не дефинира всеки:

scale.map {| бележка | note.upcase}

Резултатът е

NoMethodError: неизвестен метод `map 'за # <Music :: Scale: 0x3b0aec>

Сега на практика, ако искате везните да бъдат напълно изброени, бихте почти сигурно

смесете в Enumerable и променете името на играта на всяка. Но можете да направите и

мащабируемо, като го свържете с изброител.

Ето как да създадете изброител за мащабния обект, обвързан с метода на възпроизвеждане:

enum = scale.enum\_for (: игра)

Преброителят, enum, има всеки метод; този метод изпълнява същата итерация

че методът на възпроизвеждане на мащаба изпълнява. Освен това, за разлика от мащаба, изброяването

tor *е* изброим обект; има карта, изберете, инжектирайте и всички останали стандартни

методи от Enumerable. Ако използвате изброителя, получавате изброими операции

върху фундаментално неизброим обект:

p enum.map {| бележка | note.upcase}

p enum.select {| бележка | note.include? ('f')}

Изходът на първия ред е

["C", "C #", "D", "D #", "E", "F", "F #", "G", "A", "A #", "B"]

и изходът на втория ред е

["f", "f #"]

След това един изброител се прикрепя към определен метод за определен обект

и използва този метод като основен метод - всеки - за цялото изброяване -

способен набор от инструменти.

**Или scale.to\_enum (: игра)**

|  |
| --- |
| **322 серия** |

**322**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Прикачването на преброител към обект, който не може да се изброява като обекта на мащаба е a

добро упражнение, защото илюстрира разликата между оригиналния обект и

преброителят толкова рязко. Но в по-голямата част от случаите обектите, за които

изброителите са създадени, самите те са изброими: масиви, хешове и т.н.

Повечето от примерите в следващото ще включват изброени обекти (изключението

нето са низове). В допълнение към това да ни отведе в сферата на най-често срещаните

практики, това ще ни позволи да разгледаме по-широко възможните предимства на

използване на изброители.

През цялото време имайте предвид урока на обекта Music :: Scale и неговото изброяване-

ator: изброителят е изброим обект, чийто всеки метод работи като вид

на сифона, извличане на стойности от итератор, дефиниран на различен обект.

Ще завършим нашата проверка на изброителите с поглед върху техниките, които

включват верижни преброители и извиквания на методи.

***10.11 Верига на метода на изброяване***

Веригирането на методи е често срещана техника в програмирането на Ruby. Това е често срещано отчасти

защото е толкова лесно. Искате да отпечатате списък с главни имена, разделен със запетая

започвайки с A до N? Просто нанижете няколко метода заедно:

поставя names.select {| n | n [0] <'M'} .map (&: upcase) .join (",")

Стилът на обработка на данните от ляво надясно на конвейерната лента е мощен и за повечето

част, пряка. Но това има своя цена: създаването на междинни обекти.

Веригата на методи обикновено създава нов обект за всяка връзка във веригата. В предишната

ous код, приемайки, че имената са масив от низове, Руби накрая създава още два

масиви (един като изход на select, един от map) и низ (от join).

Изброителите не решават всички проблеми на веригата на методите. Но те смятат, че

порта в някои случаи създаването на междинни обекти. И изброител-

базирана верига има някаква семантика за себе си, че е добре да се справите.

***10.11.1 Икономия на междинни обекти***

Не забравяйте, че много методи от модула Enumerable връщат преброител

ако им се обадите без блок. В повечето такива случаи няма причина да веригирате

преброител директно към друг метод. names.each.inject, например, може като

добре бъдете имена.инжектирайте. По същия начин names.map.select не ви купува нищо

имена.изберете. Преброителят на картата няма никакви познания за това към каква функция

карта към; следователно не може да направи много друго освен да предаде първоначалния масив от стойности

веригата.

Но помислете за имена.each\_slice (2). Преброителят, генериран от този израз

sion носи полезна информация; знае, че се очаква да произведе дву-

дълги на елемента резени на масива от имена. Ако го поставите във верига от методи, той има

ефект:

**Изход (даден**

**предишен списък на**

**имена): DAVID, ЧЕРНО**

|  |
| --- |
| **323 серия** |

**323**

***Верига на метода на изброител***

>> имена =% w {Дейвид Блек Юкихиро Мацумото}

=> ["Дейвид", "Черен", "Юкихиро", "Мацумото"]

>> names.each\_slice (2) .map do | first, last |

"Име: # {first}, фамилия: # {last} \ n"

край

=> ["Име: Дейвид, фамилия: Черен \ n",

"Име: Юкихиро, фамилия: Мацумото \ n"]

Кодовият блок, прикрепен към операцията на картата, получава подадени елементи от имената

масив две наведнъж, поради изчислителя each\_slice (2). Изброителят може

продължете по „мързелив“ начин: вместо да създавате цял масив от двуелементни резени

памет, той може да създава срезовете, както са необходими за операцията на картата.

Изброими методи, които вземат аргументи и връщат изброители, като всеки\_срез,

са кандидати за този вид компресия или оптимизация, дори ако са изброени

метод не връща изброител, можете да създадете такъв за него, като включите

аргумент, така че да бъде запомнен от изброителя. Виждали сте пример за това

техника, подходена от малко по-различен ъгъл, в раздел 10.9.2:

e = names.enum\_for (: инжектиране, "Имена:")

Преброителят помни не само, че е прикрепен към метода на инжектиране на имена

но също така, че представлява покана за инжектиране с аргумент "Имена".

В допълнение към общата практика за включване на преброители във вериги от методи,

специализиран метод with\_index - един от малкото, които класът Enumerator прилага

отделно от тези в Enumerable - добавя значителна стойност към изброяванията.

Грамотност на преброителя

Едно от последиците от начина, по който работят изброителите и от автоматичното им връщане

По принцип от безблоковите извиквания на итератори е необходимо малко практика за четене на преброител

код правилно. Помислете за този фрагмент, който връща масив от цели числа:

string = "Произволен низ"

string.each\_byte.map {| b | b + 1}

Вероятно не е полезна бизнес логика ... но въпросът е, че тя много прилича

string.each\_byte връща масив. Наличието на карта като следваща операция,

макар и да не е убедително доказателство за масив, със сигурност предизвиква присъствието на

лекция вляво.

Нека го кажем по друг начин. Съдейки по външния му вид, може да очаквате, че ако обелите

от цялото обаждане на картата ще ви остане колекция.

Всъщност string.each\_byte връща изчислител. Най-важното е, че Enumerator *е*

колекция. Това е изброим обект, колкото е масив или хеш. Просто може

вземете малко да свикнете.

|  |
| --- |
| **324 серия** |

**324**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

***10.11.2 Индексиране на изброими с with\_index***

В дните, когато рубистите използваха метода each\_with\_index, редица от нас лобираха

за съответния метод map\_with\_index. Никога не сме го получили - но в крайна сметка го получихме

нещо още по-добро. Изброителите имат метод with\_index, който добавя числови

индексиране, като втори параметър на блока, към всяко изброяване. Ето как бихте го направили

използвайте with\_index, за да направите картографиране на буквите / цифрите:

('a' .. 'z'). map.with\_index {| буква, i | [писмо, i]}

Имайте предвид, че е map.with\_index (два метода, оковани), а не map\_with\_index (ком-

име на метода на позицията). И with\_index може да бъде свързан към всеки изброител. Помня

музикалната скала от раздел 10.10.4? Да кажем, че изброяваме метода на възпроизвеждане:

def play

ЗАБЕЛЕЖКИ.to\_enum

край

Оригиналният пример за преминаване през бележките вече ще работи без създаването

на междинен изброител.

scale.play {| бележка | поставя "Следваща бележка: # {бележка}"}

И сега и това ще работи:

scale.play.with\_index (1) {| забележка, i | поставя „Бележка № {i}: # {Бележка}“}

Резултатът ще бъде номериран списък с бележки:

Забележка 1: c

Забележка 2: c #

Забележка 3: d

# и т.н.

По този начин методът with\_index обобщава това, което иначе би било ограничено

функционалност.

Ще разгледаме още един пример за свързване на преброители, който добре събира

няколко техники за изброяване и итерация, а също така представя няколко нови

методи, които може да ви бъдат полезни.

***10.11.3 Изключително - или операции върху низове с изброители***

Изпълнението на ексклузивна или (или *XOR* ) операция на низ означава XOR -инг на всяка от неговите

байтове с някаква стойност. XOR -ing байт е битова операция: всеки байт е представен

от цяло число, а резултатът от операцията XOR е изключително или ин-гиране на това цяло число

с друго число.

Ако вашият низ е "а", например, той съдържа един байт със стойността 97. Двоичната

представяне на 97 е 1100001. Да кажем, че искаме да го XOR с символа #,

който има ASCII стойност 35 или 100011 в двоичен формат. Разглеждайки го чисто числено,

а не по отношение на низове, правим 97 ^ 35 или 1100001 ^ 100011 в двоични термини.

**Изход: [["a", 0],**

**["b", 1] и др.]**

**Посочете 1 като**

**първата стойност**

**за индекса**

|  |
| --- |
| **325 серия** |

**325**

***Верига на метода на изброител***

Един XOR произвежда резултат, че в двоично представяне (тоест, от гледна точка на неговите бита)

съдържа 1, където някой от номерата на източника, *но не и двата* , съдържа 1 и 0

където и двата номера на източника съдържат една и съща стойност, независимо дали 0 или 1. В

в случай на нашите две числа, операцията XOR дава 1000010 или 66.

Отличително свойство на битовите XOR операции е, че ако изпълните

същата операция два пъти, получавате обратно първоначалната стойност. С други думи, (a ^ b) ^ b == a.

По този начин, ако xor 66 с 35, получаваме 97. Това поведение прави xor-ing низовете полезни

техника за замъгляване, особено ако xor или дълъг байт низ за байт срещу секунда

низ. Кажете, че вашият низ е „Това е низ“. Ако зададете xor символ за символ

срещу, да речем, #%. 3u, повтаряйки xor низ, колкото е необходимо, за да се достигне дължината на

оригинален низ, получавате доста обезсърчаващия резултат wMG @ UJV \ x0ERUPQ \\ Z \ eD \ v. Ако ти

xor тази чудовищност срещу #%. 3u отново получавате обратно "Това е низ."

Сега нека напишем метод, който ще направи това. Ще го добавим към класа String - а не не-

най-добрият начин да промените функционалността на основните Ruby обекти (като

ще видите в глава 13), но е целесъобразно за целите на илюстрацията. Следният списък-

ing показва метода на екземпляра String # ^.

клас String

def ^ (ключ)

kenum = ключ.each\_byte.cycle

each\_byte.map {| байт | байт ^ kenum.next} .pack ("C \*")

край

край

Методът приема един аргумент: низът, който ще се използва като основа на xor

работа (на *ключ* ) B . Трябва да се справим със случаи, когато ключът е по-кратък от

оригинален низ чрез прелистване на ключа толкова пъти, колкото е необходимо за осигуряване

достатъчно символи за цялата операция. Там влизат изброителите.

Променливата kenum е свързана с изчислител въз основа на свързване на два метода

ключовият низ: всеки\_байт, който сам по себе си връща преброител, обхождащ байта на низа

по байт и цикъл, който се повтаря отново и отново чрез колекция, възобновявайки се в

началото, когато стигне до края c . Кнумът на изброителя въплъщава и двете

тези операции: всяка итерация през него осигурява друг байт от низа; и

когато приключи с предоставянето на всички байтове, той се връща в началото на низа

и отново се итерира над байтовете. Точно това поведение искаме, за да сме сигурни

имаме достатъчно байта, за да съответстваме на какъвто и да е низ, дори ако това е низ

по-дълъг от ключа. Всъщност направихме ключовия низ безкрайно дълъг.

Сега идва действителната операция xor d . Тук използваме всеки\_байт за итерация над

байта на низ, който е xor'ed. Преброителят, върнат от всеки\_байт получава

окован в карта. Вътре в блока на картата всеки байт от оригиналния низ е xor'ed с

„Следващ“ байт от изброителя, който се движи безкрайно през байтовете на ключа

низ. Тогава цялата операция с карта създава масив от xor'ed байтове. Всичко това

остава да върнете тези байтове обратно в низ с резултати.

Листинг 10.2 Изключителен или метод за низове

б

° С

д

|  |
| --- |
| **326 серия** |

**326**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Въведете метода на пакета. Този метод превръща масив в низ, тълкувайки

всеки елемент от масива по начин, посочен от аргумента. В този случай

аргументът е "C \*", което означава да се *третира всеки елемент от масива като неподписано цяло число*

*изпращане на един знак* (това е „C“) *и обработка на всички тях* (това е „\*“). Опаковане

по този начин масивът в низ от символи е еквивалентът на трансформиране на всеки масив

елемент в символ и след това прави обединяване на целия масив.

Сега можем да xor низове. Ето как изглежда процесът:

>> str = "Хубав малък низ."

=>"Хубав малък низ."

>> ключ = "тайна!"

=>"тайна!"

>> x = str ^ ключ

=>"= \ f \ x00 \ x17E \ x18H \ a \ x11 \ x0F \ x17E \ aU \ x01 \ f \ r \ x15K"

>> orig = x ^ ключ

=>"Хубав малък низ."

Както можете да видите, XOR -ing два пъти с един и същ ключ ви връща към първоначалния низ.

И всичко това благодарение на двуредов метод, който използва три изброители!

Изброителите добавят изцяло нов инструмент към вече богатия набор от инструменти за събиране на Ruby

управление и итерация. Те са концептуално и технически различни от итера-

tors, но ако ги изпробвате при техните условия, със сигурност ще намерите приложения за тях

заедно с другите техники, свързани с колекцията, които сте виждали.

Ще завършим погледа си към изброителите с вариант, наречен мързелив преброител.

***10.12 Мързеливи изброители***

Мързеливите изброители улесняват селективното изброяване на безкрайно големи колекции

ния. За да илюстрираме какво означава това, нека започнем със случай, в който дадена операция се опитва

изброява над безкрайно голяма колекция и засяда. Ами ако искате да знаете

първите 10 кратни на 3? За да използваме безкрайна колекция, ние ще създадем диапазон, който върви

от 1 до специалната стойност Float :: INFINITY. Използването на такъв диапазон, първи подход към

разглежданата задача може да бъде

(1..Float :: INFINITY) .изберете {| n | n% 3 == 0} .първа (10)

Принудително кодиране

String # ^, както е реализиран в предишния фрагмент, е уязвим за кодиране

въпроси: ако xor, да речем, UTF-8 низ срещу ASCII низ два пъти, ще получите обратно

низ, кодиран в ASCII-8BIT. За да се предпазите от това, добавете повикване към force\_encoding:

each\_byte.map {| байт | байт ^ kenum.next} .pack ("C \*").

force\_encoding (самокодиране)

Това ще гарантира, че байтовата последователност, генерирана от картографирането, ще бъде кодирана

кодирането на оригиналния низ.

|  |
| --- |
| **327 серия** |

**327**

***Мързеливи изброители***

Но този ред код работи завинаги. Операцията за избор никога не завършва, така че

chained-on първата команда никога не се изпълнява.

Можете да получите краен резултат от безкрайна колекция с помощта на мързелив преброител.

Извикването на мързеливия метод директно върху обект от диапазон ще доведе до мързелив преброител

над този диапазон:

>> (1..Float :: INFINITY) .lazy

=> # <Изброител :: Мързелив: 1..Бесконечност>

След това можете да свържете този мързелив изброител, за да изберете, създавайки каскада от мързеливи

изброители:

>> (1..Float :: INFINITY) .lazy.select {| n | n% 3 == 0}

=> # <Enumerator :: Lazy: # <Enumerator :: Lazy: 1..Infinity>: select>

Тъй като сега лениво изброяваме, е възможно да вземем набори от резултати от нашата операционна система

без да чакате изпълнението на безкрайни задачи. По-конкретно, вече можем да попитаме

за първите 10 резултата от избрания тест от безкрайния списък, а безкрайният списък е

щастлив да изброя само толкова, колкото е необходимо, за да се получат тези 10 резултата:

>> (1..Float :: INFINITY) .lazy.select {| n | n% 3 == 0} .първа (10)

=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

Като вариант на същата тема, можете да създадете мързеливия избор на изброител и

след това използвайте поемете го. Това ви позволява да изберете колко кратни на 3 искате да видите

без твърдо кодиране на номера. Имайте предвид, че трябва да призовете сила за резултата от

предприеме; в противен случай ще се окажете с поредния мързелив изброител, а не с действителен

набор от резултати:

>> my\_enum = (1..Float :: INFINITY) .lazy.select {| n | n% 3 == 0}

=> # <Enumerator :: Lazy: # <Enumerator :: Lazy: 1..Infinity>: select>

>> my\_enum.take (5) .force

=> [3, 6, 9, 12, 15]

>> my\_enum.take (10) .force

=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

Мързеливите изброители са малко специализиран инструмент и вероятно няма да имате нужда от тях

твърде често. Но те са много удобни, ако имате безкрайна колекция и искате да се занимавате

само с краен набор от резултати от операции върху тази колекция.

***10.12.1 FizzBuzz с мързелив преброител***

Проблемът с FizzBuzz в класическата му форма включва отпечатване на целите числа от 1 до

100 ... с изключение на това, че прилагате следните правила:

Ако числото се дели на 15, отпечатайте "FizzBuzz".

В противен случай, ако числото се дели на 3, отпечатайте "Fizz".

В противен случай, ако числото се дели на 5, отпечатайте "Buzz".

В противен случай отпечатайте номера.

|  |
| --- |
| **328 серия** |

**328**

С ГЛАВА 10 ***Централни колекции: Изброими и изброител***

Можете да използвате мързелив преброител, за да напишете версия на FizzBuzz, която може да се справи с всяка

диапазон от числа. Ето как може да изглежда:

def fb\_calc (i)

дело 0

когато i% 15

"FizzBuzz"

когато i% 3

"Fizz"

когато i% 5

„Бъз“

друго

i.to\_s

край

край

def fb (n)

(1..Float :: INFINITY) .lazy.map {| i | fb\_calc (i)} .first (n)

край

Сега можете да разгледате, да речем, изхода на FizzBuzz за първите 15 положителни числа като този:

p fb (15)

Резултатът ще бъде

["1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz", "Buzz", "11",

"Fizz", "13", "14", "FizzBuzz"]

Без да се създава мързелив преброител на обхвата, операцията с картата ще продължи за-

някога. Вместо това, мързеливото изброяване гарантира, че целият процес ще спре, след като сме го направили

получихме това, което искаме.

***10.13 Обобщение***

В тази глава сте виждали

Модулът Enumerable и неговите методи за екземпляри

Използване на Enumerable във вашите собствени класове

Основи на изброителя

Създаване на изброители

Итерация над низове

Мързеливи изброители

Тази глава се фокусира върху модула Enumerable и класа Enumerator, две цялостни

връзки с тесни връзки. Първо, изследвахме методите на екземпляра на Enumerable, които

са дефинирани по отношение на всеки метод и които са достъпни за вашите обекти като

стига тези обекти да отговарят на всеки и вашият клас се смесва в Enumerable. Второ,

разгледахме изброители, обекти, които капсулират процеса на итерация на друг

обект, обвързвайки се - конкретно, всеки техен метод - с определен метод

на друг обект и използвайки това паразитно всяко свързване, за да достави пълната гама от

изброима функционалност.

|  |
| --- |
| **329 серия** |

**329**

***Обобщение***

Преброителите могат да бъдат трудни. Те надграждат изцяло върху Изброими; и в случаите, когато

преброител се свързва с обект, който е заменил някои от Enumerable

методи, важно е да запомните, че изброителят ще има свои собствени идеи за

какви са тези методи. Това не е прокси с общо предназначение за друг обект; то сифонира

изключване на стойности от единия метод за другия обект.

По един или друг начин - било то чрез използването на изброители или използването на повече

класически стил на итерация и управление на колекции в Ruby - почти със сигурност ще използвате

свързаните с изброяването съоръжения на езика практически всеки път, когато пишете Ruby

програма. Струва си да се запознаете отблизо с Enumerable; това е толкова мощна единица

функционалност, както има навсякъде в Ruby.

Ще се обърнем до темата за регулярните изрази и съвпадението на шаблоните. Като

ще видите, има някаква печалба да се гледат както низовете, така и обектите за събиране преди

изучаване на регулярни изрази: редица методи за съвпадение на шаблони, изпълнени на

низовете ви връщат резултатите си под формата на колекция и следователно се поддават

повторение. Разглеждането на регулярни изрази ще ви помогне да разработите пълнофункционален инструментариум

за обработка на низове и тела.

|  |
| --- |
| **Страница 330** |

**330**

*Регулярни изрази*

*и базирани на regexp*

*низ операции*

В тази глава ще разгледаме съоръженията на Ruby за съвпадение на шаблони и текстов процес-

инг, центриране около използването на регулярни изрази. А *регулярен израз* в Ruby

служи на същите цели, както на други езици: той определя модел на характер

ters, модел, който може или не може правилно да предскаже (т.е. да съвпада) даден низ.

Операциите за съвпадение на шаблони се използват за условно разклоняване (съвпадение / липса на съвпадение),

точно определяне на поднизове (части от низ, които съответстват на части от шаблона) и вари-

нашите техники за филтриране на текст.

Регулярните изрази в Ruby са обекти. Изпращате съобщения *до* редовен експрес

сион. Регулярните изрази добавят нещо към пейзажа на Ruby, но като обекти

те също се вписват добре в пейзажа.

Ще започнем с преглед на регулярните изрази. Оттам ще преминем към

подробностите за това как да ги напишете и, разбира се, как да ги използвате. В последния кате-

gory, ще разгледаме използването на регулярни изрази както в прости операции за съвпадение, така и в

***Тази глава обхваща***

Синтаксис на регулярен израз

Операции за съвпадение на шаблони

Класът MatchData

Вградени методи, базирани на съвпадение на шаблони

|  |
| --- |
| **331 серия** |

**331**

***Писане на регулярни изрази***

в методи, при които те играят роля в по-голям процес, като филтриране на колекция или

многократно сканиране на низ.

***11.1 Какво представляват регулярните изрази?***

Регулярните изрази се появяват в много езици за програмиране, с малки разлики

сред въплъщенията. Тяхната цел е да определят модели на символи, които

често са определени да съвпадат (или да не съвпадат) низове. Съвпадение на шаблони, от своя страна,

служи като основа за операции като анализиране на регистрационни файлове, тестване на въвеждането от клавиатурата за валидни-

ity и изолиращи поднизове - операции, с други думи, чести и значителни

използвайте за всеки, който трябва да обработва низове и текст.

Регулярните изрази имат странна репутация. Използването им е мощен, концентриран

уловена техника; те изгарят през голямо подмножество проблеми с обработката на текст като

киселина през катинар. Те също са, по мнението на много хора (включително хора

които ги разбират добре), труден за използване, труден за четене, непрозрачен, неподдържан-

способни и в крайна сметка контрапродуктивни.

Трябва да прецените сами. Единственото нещо, което не трябва *да* правите, е да се отбягвате

научавайки поне основите на това как работят регулярните изрази и как да използваме Ruby

методи, които ги използват. Дори ако решите, че не сте „регулярен израз на

синко “, имате нужда от четене на знания за тях. И в никакъв случай няма да бъдете сами, ако сте

в крайна сметка ги използвайте в собствените си програми повече, отколкото сте очаквали.

Редица вградени в Ruby методи приемат регулярни изрази като аргументи и

извършете избор или модификация на един или повече низови обекти. Регулярни изрази

се използват, например, за *сканиране* на низ за множество появявания на шаблон, за *заместване*

*настройване* на заместващ низ за подниз и за *разделяне* на низ на множество поднизове

въз основа на съвпадащ разделител.

Ако сте запознати с регулярните изрази от Perl, sed, vi, Emacs или други

източник, може да искате да прегледате или пропуснете материала на изложението тук и да го вземете за секунди

11.5, където говорим за руби методи, които използват регулярни изрази. Но забележете

че регулярните изрази на Ruby не са идентични с тези на който и да е друг език. Вие ще

почти със сигурност ще можете да ги прочетете, но може да се наложи да проучите разликите (такива

като дали скобите са специални по подразбиране или специални при избягване), ако влезете

писането им.

Нека се обърнем сега към писането на някои регулярни изрази.

***11.2 Писане на регулярни изрази***

Регулярните изрази се пишат с познати знаци - разбира се, но трябва

да се научат да ги четат и пишат като неща за себе си. Те не са струни, а техните

значението не винаги е толкова очевидно, колкото това на низовете. Те са представяне на *модели* .

***11.2.1 Виждане на модели***

Регулярният израз (regexp или regex) указва модел. За всеки такъв модел,

всеки низ в света или съвпада с модела, или не му съответства. Рубинът

|  |
| --- |
| **332 серия** |

**332**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

методите, които използват регулярни изрази, ги използват, за да определят дали даден

низ съответства на даден модел или за да се определи това, а също така вземете някои

действие въз основа на отговора.

Образците от вида, определени от регулярни изрази, се разбират най-лесно,

първоначално на обикновен език. Ето няколко примера за модели, изразени по този начин:

Буквата а, последвана от цифра

Всяка главна буква, последвана от поне една малка буква

Три цифри, последвани от тире, последвани от четири цифри

Моделът може също да включва компоненти и ограничения, свързани с позиционирането вътре

низът:

Началото на ред, последвано от един или повече пробели

Характерът . (точка) в края на низ

Главна буква в началото на дума

Компоненти на шаблони като „началото на ред“, които по-скоро отговарят на условие

отколкото символ в низ, въпреки това се изразяват със знаци или последователности от

символи в регулярния израз.

Регулярните изрази предоставят език за изразяване на модели. Научете се да пишете

те се състоят главно в това да научат как различни неща се изразяват в рамките на регулярния израз.

Най-често прилаганите правила за изграждане на регулярни изрази са доста лесни за научаване. Вие

просто трябва да запомните, че регулярният израз, въпреки че съдържа символи, не е низ. Това е

специална нотация за изразяване на модел, който може или не може да опише правилно

някои или всички от даден низ.

***11.2.2 Просто съвпадение с буквални регулярни изрази***

Регулярните изрази са екземпляри на Regexp класа, който е един от класовете Ruby

който има литерален конструктор за лесно създаване на инстанции. Буквеният конструктор на regexp е

двойка наклонени черти напред:

//

Колкото и странно да изглежда това, наистина е регулярно изражение, ако е скелетно. Можете да проверите дали е така

ви дава екземпляр на Regexp класа в irb:

>> //.клас

=> Regexp

Специфичните особености на регулярния експрес са между наклонените линии. Ще започнем да изграждаме няколко сим-

попълнете регулярни изрази, докато разглеждаме основите на процеса на съвпадение.

Всяка операция за съвпадение на шаблони има два основни играча: регулярно изражение и низ. The

regexp изразява прогнози за низа. Или низът изпълнява тези прогнози

tions (съответства на модела) или не.

Най-простият начин да разберете дали има съответствие между шаблон и a

string е с метода на съвпадението. Можете да направите това в двете посоки - regexp обекти

|  |
| --- |
| **333 серия** |

**333**

***Изграждане на шаблон в регулярен израз***

и string обектите отговарят на съвпадение и двата примера успяват и

print "Match!":

поставя "Мач!" ако /abc/.match("Абетката започва с abc. ")

поставя "Мач!" ако "Азбуката започва с abc.". съвпадение (/ abc /)

Низовата версия на match (вторият ред от двете) се различава от регулярния израз

версия с това, че превръща низ аргумент *за* по-рег.израз. (Ще се върнем към това малко

по-късно.) В примера аргументът вече е регулярно изражение (/ abc /), така че няма преобразуване

необходимо е.

В допълнение към метода на съвпадение, Ruby има и оператор за съвпадение на шаблони,

= ~ (знак за равенство и тилда), което преминава между низ и регулярно изражение:

поставя "Мач!" if / abc / = ~ "Азбуката започва с abc."

поставя "Мач!" ако "Азбуката започва с abc." = ~ / abc /

Както се досещате, този „оператор“, който съвпада с модела, е метод на екземпляр и на двете

класовете String и Regexp. Това е един от многото методи на Ruby, които предоставят

синтактична захар от стил на използване на инфикс-оператор.

Методът на съвпадение и операторът = ~ са еднакво полезни, когато търсите сим-

ple да / не отговор на въпроса дали има съвпадение между низ и a

модел. Ако няма съвпадение, връщате нула. Това е удобно за специалисти; и четиримата

от предишните примери тестват резултатите от техните операции по съвпадение с тест if.

Къде мач и = ~ различават един от друг предимно е в това, което те се връщат, когато има *е*

съвпадение: = ~ връща числовия индекс на символа в низа, където

match започна, докато match връща екземпляр на класа MatchData:

>>"Азбуката започва с abc" = ~ / abc /

=> 25

>> /abc/.match("Абетката започва с abc. ")

=> # <MatchData "abc">

Първият пример намира съвпадение в позиция 25 на низа. Във втория пример,

създаването на обект MatchData означава, че е намерено съвпадение.

Ще разгледаме обектите MatchData малко по-нататък. За момента ще бъдем

главно с получаване на отговор да / не на опит за съвпадение, така че

показаните досега niques ще работят. В името на последователността и защото ще бъдем

по-загрижени за MatchData обекти, отколкото числови индекси на поднизове, повечето

от примерите в тази глава ще се придържаме към метода за съвпадение на Regexp #.

Сега, нека разгледаме по-подробно състава на регулярния израз.

***11.3 Изграждане на шаблон в регулярен израз***

Когато пишете регулярно изражение, поставяте дефиницията на вашия шаблон между напред

наклонени черти. Не забравяйте, че това, което поставяте там, не е низ, а набор от прогнози

и ограничения, които искате да търсите в низ.

Възможните компоненти на регулярния израз включват следното:

|  |
| --- |
| **334 серия** |

**334**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

*Буквални знаци* , означаващи „съвпадат с този знак“

The *Dot заместващия знак (.)* , Което означава "всички съответстващи знаци" (с изключение на \ Н, на новообза-

ред символ)

*Класове знаци* , означаващи „съвпадение с един от тези знаци“

Ще обсъдим всеки от тях на свой ред. След това ще използваме това знание, за да разгледаме по-дълбоко

при мачови операции.

***11.3.1 Буквални знаци в шарки***

Всеки буквален знак, който поставите в регулярно изражение, съвпада със *себе си* в низа. По този начин регулярното изражение

/ а /

съответства на всеки низ, съдържащ буквата a.

Някои символи имат специални значения за синтактичния анализатор на regexp (както ще видите подробно

скоро). Когато искате да съчетаете един от тези специални знаци като себе си, трябва

избягайте с обратна наклонена черта (\). Например, за да съответства на героя? (въпросителен знак),

трябва да напишете това:

/ \? /

Обратната наклонена черта означава „не третирайте следващия знак като специален; третирайте го като себе си. "

Специалните знаци включват тези, изброени между скобите тук: (^ $?. /

\ [] {} () + \*). Както можете да видите, сред тях е точката, която е специален знак в

регулярни изрази.

***11.3.2 Точковият заместващ знак (.)***

Понякога ще искате да съвпаднете с *който и* да *е символ* в даден момент от вашия модел. Ти правиш

това със специалния точков заместващ знак (.). Точка съвпада с всеки символ с

изключение на нов ред. (Има начин да го направите да съвпада и с нови редове, което ще видите

малко по-късно.)

Шаблонът в този регулярен израз съответства както на „отпаднал“, така и на „отхвърлен“:

/.ejected/

Той също така съвпада с „% изхвърлени“ и „8ejected“:

поставя "Мач!" ако /.ejected/.match("%ejected ")

Точката с заместващ знак е удобна, но понякога ви дава повече съвпадения, отколкото искате.

Можете да наложите ограничения върху съвпаденията, като същевременно позволявате множество възможни

низове, като се използват класове символи.

***11.3.3 Класове знаци***

А *клас характер* е изрично списък от символи, поставени във вътрешността на регулярния израз в квадратни скоби:

/ [dr] изхвърлен /

|  |
| --- |
| **335 серия** |

**335**

***Изграждане на шаблон в регулярен израз***

Това означава „съвпадение или *d* или *r* , последвано от *изхвърлено* .“ Този нов модел съвпада или

„Унищожен“ или „отхвърлен“, но не „& изхвърлен“. Класът на символите е вид частичен или

ограничен заместващ символ: позволява множество възможни знаци, но само ограничен

брой от тях.

В клас символи можете също да вмъкнете *набор* от символи. Често срещан случай е

това, за малки букви:

/ [az] /

За да съответствате на шестнадесетична цифра, можете да използвате няколко диапазона в клас от символи:

/ [A-Fa-f0-9] /

Това съвпада с всеки символ от *a* до *f* (главна или малка буква) или произволна цифра.

Понякога трябва да съвпадате с който и да е символ, *с изключение на* тези от специален списък. Може,

например, търсете първия символ в низ, който *не* е валиден шестнадесетичен

малка цифра.

Вие извършвате този вид отрицателно търсене, като отричате клас знаци. За да направите това,

поставяте каретка (^) в началото на класа. Например, тук е клас на символите

който съвпада с който и да е символ с изключение на валидна шестнадесетична цифра:

/ [^ A-Fa-f0-9] /

И ето как можете да намерите индекса на първото появяване на не-шестнадесетичен знак

ter в низ:

>> string = "ABC3934 е шестнадесетично число."

=>"ABC3934 е шестнадесетично число."

>> низ = ~ / [^ A-Fa-f0-9] /

=> 7

Класът на символите, положителен или отрицателен, може да съдържа всякакви знаци. Някакъв характер

класовете са толкова често срещани, че имат специални съкращения.

S СПЕЦИАЛНИ ПОСЛЕДОВАТЕЛСТВА ЗА ЕКСКАП ЗА ОБЩИ КЛАСОВЕ НА ХАРАКТЕРИТЕ

За да съответствате на която и да е цифра, можете да направите това:

/ [0-9] /

Можете също така да постигнете едно и също нещо по-кратко с помощта на специалното бягство

последователност \ d:

/\д/

Класовете знаци са по-дълги от това, което съвпадат

Дори кратък клас символи като [a] заема повече от едно място в регулярния израз. Но

не забравяйте, че всеки клас символи съвпада с *един символ* в низа. Когато погледнете

в клас от символи като / [dr] /, може да изглежда, че ще съвпада с подниза dr.

Но не е: ще съответства или на d, или на r.

|  |
| --- |
| **336 серия** |

**336**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

Забележете, че тук няма квадратни скоби; просто \ d. Две други полезни бягства

последователностите за предварително дефинирани класове символи са следните:

\ w съвпада с която и да е цифра, азбучен знак или долно (\_).

\ s съответства на всеки празен знак (интервал, раздел, нов ред).

Всеки от тези предварително дефинирани класове знаци също има отрицателна форма. Можете да съчетаете всеки

знак, който не е цифра, като направите това:

/\Д/

По същия начин \ W съответства на всеки символ, различен от буквено-цифров или недостатъчен

резултат и \ S съответства на всеки символ, който не е празен *.*

Успешното повикване за съвпадение връща обект MatchData. Нека да разгледаме MatchData

обекти и техните възможности отблизо.

***11.4 Съвпадение, улавяне на поднизове и MatchData***

Досега разгледахме основните операции по съвпадение:

regex.match (низ)

string.match (регулярно изражение)

Това са по същество тестове true / false: или има съвпадение, или няма. Сега ще го направим

изследвайте какво се случва при успешни и неуспешни мачове и какъв мач

операция може да направи за вас отвъд отговора да / не.

***11.4.1 Заснемане на подсъвпадения със скоби***

Една от най-важните техники за изграждане на регулярни изрази е използването на скоби

ses за уточняване на *улавяне* .

Идеята е тази. Когато тествате за съвпадение между низ - да речем, линия от a

файл - и шаблон, това е обикновено защото искате да направите нещо със низа или,

по-често с част от низа *.* Нотацията за улавяне ви позволява да изолирате

и запазете поднизовете на низа, които съответстват на определени подпатерни.

Да приемем например, че имаме низ, съдържащ информация за човек:

Пийл, Ема, г-жа, талантлив любител

От този низ трябва да съберем фамилията и титлата на човека. Ние знаем

полетата са разделени със запетая и ние знаем в какъв ред идват: фамилия, първо

име, титла, професия.

За да създадем шаблон, който съответства на такъв низ, може да помислим на английски заедно

следните редове:

*Първо* няколко азбучни символа,

*след това* запетая,

*след това* някои азбучни символи,

*след това* запетая,

*тогава* или „господин“ или „госпожа“

|  |
| --- |
| **337 серия** |

**337**

***Съвпадение, улавяне на поднизове и MatchData***

Поддържаме го просто: без тирета, без лекари или професори, без напускане

от крайния период за г-н и г-жа (което ще бъде направено в британска употреба). The

regexp, тогава може да изглежда така:

/[A-Za-z]+,[A-Za-z]+, госпожо?\./

(Въпросителният знак след *s* означава *съвпадение на нула или едно s* . Изразявайки го по този начин ни позволява

съвпада или с „господин“ и „госпожа“ кратко.) Моделът съвпада с низа, както irb удостоверява:

>> /[A-Za-z]+,[A-Za-z]+,Mrs?\./.match("Peel,Emma,Mrs.,talented amateur ")

=> # <MatchData "Peel, Emma, ​​Mrs.">

Получихме MatchData обект, а не нула; имаше мач.

Но сега какво? Как да изолираме поднизовете, които ни интересуват ("Peel" и

"Госпожо")?

Тук влизат групите в скоби. Искаме две такива групи: една

около подвида, който съвпада с фамилното име, и един около подплата

което съответства на заглавието:

/([A-Za-z]+),[A-Za-z]+,(Mrs?\.)/

Сега, когато изпълняваме мача

/([A-Za-z]+),[A-Za-z]+,(Mrs?\.)/.match("Peel,Emma,Mrs.,talented amateur ")

случват се две неща:

Получаваме MatchData обект, който ни дава достъп до подменютата (обсъдени в

момент).

Ruby автоматично попълва серия от променливи за нас, които също ни дават

достъп до тези подсъвпадения.

Променливите, които Ruby попълва, са глобални променливи и техните имена са базирани

на числа: $ 1, $ 2 и т.н. $ 1 съдържа подниза, съответстващ на subpat-

tern в *първия* набор от скоби отляво в регулярния израз. Разглеждане на $ 1

след предишния мач (например с путове $ 1) показва Peel. $ 2 съдържа

подниз, съчетан с *втория* подпатерн; и така нататък. Като цяло правилото е следното:

след успешна операция на съвпадение, променливата $ *n* (където *n* е число) съдържа

поднизът, съчетан с подшаблон в *n* -ия набор от скоби от

ляво в регулярния израз.

ЗАБЕЛЕЖКА Ако сте използвали Perl, може да сте видели променливата $ 0, която представлява

изпраща не конкретен уловен под образец, а целия подниз, който е бил

успешно съвпада. Ruby използва $ 0 за нещо друго: съдържа името

на програмния файл Ruby, от който е била инициирана текущата програма или скрипт

първоначално стартиран. Вместо $ 0 за съвпадение на шаблони, Ruby предоставя метод;

извиквате низ на обекта MatchData, върнат от съвпадението. Ще видите

пример за низ метод в раздел 11.4.2.

|  |
| --- |
| **338 серия** |

**338**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

Можем да комбинираме тези техники с интерполация на низове, за да генерираме поздрав

за писмо, базирано на изпълнение на мача и грабване на променливите $ 1 и $ 2:

line\_from\_file = "Пил, Ема, госпожо, талантлив любител"

/([A-Za-z]+),[A-Za-z]+,(Mrs?\.)/.match(line\_from\_file)

поставя „Уважаеми # {$ 2} # {$ 1},“

Променливите в стил $ *n* са удобни за грабване на подсъвпадения. Но можете да постигнете

същото по по-структуриран, програмен начин чрез заявка на обекта MatchData

върнати от операцията ви за съвпадение.

***11.4.2 Съвпадение на успех и неуспех***

Всяка операция на мача е или успешна, или неуспешна. Нека започнем с по-простия случай: неуспех-

урета. Когато се опитате да съчетаете низ с шаблон и низът не съвпада, символът

резултатът винаги е нулев:

>> /a/.match("b ")

=> нула

За разлика от nil, обектът MatchData, върнат от успешно съвпадение, има булева стойност

на true, което го прави удобен за прости тестове за съвпадение / несъвпадение. Освен това, то също

съхранява информация за мача, която можете да извлечете със съответния метод

ods: къде съвпадението е започнало (с какъв символ в низа), колко от низа

обхващаше това, което беше уловено в групите с скоби и т.н.

За да използвате обекта MatchData, първо трябва да го запазите. Помислете за пример, където

искате да изтръгнете телефонен номер от низ и да запазите различните части от него

(код на района, обмен, номер) в групи. Следващият списък показва как сте

може да направи това. Също така е написано като клиника за това как да използвате някои от MatchData повече

общи методи.

string = "Моят телефонен номер е (123) 555-1234."

phone\_re = / \ ((\ d {3}) \) \ s + (\ d {3}) - (\ d {4}) /

m = phone\_re.match (низ)

освен ако m

поставя "Нямаше съвпадение - съжалявам."

изход

край

print "Целият низ, с който започнахме:"

поставя m.string

print "Цялата част от низа, която съвпада:"

поставя m [0]

поставя "Трите улавяния:"

3. пъти правя | индекс |

поставя "Capture ## {index + 1}: # {m.captures [index]}"

край

поставя „Ето още един начин да стигнете до първото улавяне:“

print "Capture # 1:"

поставя m [1]

Листинг 11.1 Съвпадение на телефонен номер и заявки за получения обект **MatchData**

**Резултат: Уважаеми**

**Госпожо Пийл,**

**Прекратява**

**програма**

б

° С

д

д

|  |
| --- |
| **339 серия** |

**339**

***Съвпадение, улавяне на поднизове и MatchData***

В този код използвахме низовия метод на MatchData B , който връща цялото

низ, върху който е извършена операцията за съвпадение. За да получите частта от низа, която

съвпада с нашия модел, ние се обръщаме към обекта MatchData с квадратни скоби, с

индекс от 0 c . Също така използваме метода d, за да повторим точно три пъти

чрез кодов блок и разпечатайте подсъвпаденията (в скоби) в

наследяване. Вътре в този кодов блок метод, наречен улавяне, изважда поднизовете

които съответстваха на скобите в шаблоните. Накрая поглеждаме още веднъж

първото улавяне, този път чрез различна техника e : индексиране на MatchData

обект директно с квадратни скоби и положителни цели числа, като всяко цяло число съответства

до залавяне.

Ето резултата от списъка:

Целият низ, с който започнахме: Моят телефонен номер е (123) 555-1234.

Цялата част от низа, която съвпада: (123) 555-1234

Трите улавяния:

Заснемане # 1: 123

Заснемане # 2: 555

Заснемане # 3: 1234

Ето още един начин да стигнете до първото улавяне:

Заснемане # 1: 123

Това ви дава вкус на видовете данни за съвпаденията, които можете да извлечете от MatchData

обект. Можете да видите, че има два начина за извличане на заснемания. Нека увеличим

тези техники.

***11.4.3 Два начина за получаване на улавянията***

Един от начините за получаване на скобите от обект MatchData е директно

индексиране на обекта, стил на масив:

м [1]

м [2]

# и т.н.

Първият ред ще покаже първото улавяне (първият набор от скоби отляво),

вторият ред ще покаже второто улавяне и т.н.

Както показва списък 11.1, индекс 0 ви дава целия низ, който е бил съчетан.

От 1 нататък индексът на *n* ви дава *n* -то заснемане въз основа на отваряне за броене

скоби отляво. (И *n* , където *n*> 0, винаги съответства на числото в

глобалната променлива $ *n* .)

Другата техника за получаване на скоби от обект MatchData

е методът captures, който връща всички заловени поднизове в един масив.

Тъй като това е редовен масив, първият елемент в него - по същество, същият като глобалната вариация

$ 1 - е позиция 0, а не позиция 1. С други думи, важат следните еквивалентности:

m [1] == m. улавя [0]

m [2] == m. улавя [1]

и така нататък.

|  |
| --- |
| **Стр. 340** |

**340**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

Дума за това повтарящо се „броене на скоби отляво“. Някои рег-

ular изразите могат да бъдат объркващи по отношение на техните скоби за улавяне, ако не знаете

правилото. Вземете този, например:

/((a)((b)c))/.match("abc ")

Какво ще има в различните улавяния? Е, просто пребройте отварящите скоби от

наляво. За всяка отваряща скоба намерете съответствието си вдясно. Всичко вътре

тази двойка ще бъде номер за заснемане *n* , за каквото и *n* сте стигнали.

Това означава, че първото улавяне ще бъде "abc", защото това е частта от низа, която

съвпада с модела между най-отдалечените скоби. Следващите скоби

кръг "а"; това ще бъде второто улавяне. Следва "bc", последвано от "b". И

това е последната от отварящите скоби.

Представянето на низове на обекта MatchData, което получавате от това съвпадение, ще

задължително ви показва улавянията:

>> /((a)((b)c))/.match("abc ")

=> # <MatchData "abc" 1: "abc" 2: "a" 3: "bc" 4: "b">

Разбира се, те отговарят стриктно на това, което е съвпаднало между двойките

скоби отброени отляво.

N AMED ПЛАНИ

Заснемането на субекспресии, индексирани по номер, със сигурност е удобно, но има и друго,

понякога по-лесен за читателите начин, чрез който можете да обозначавате субекспресии: наречени улавяния.

Ето един пример. Този регулярен израз ще съответства на име на формата „Дейвид

А. Блек “или, също така,„ Дейвид Блек “(без средно инициал):

>> re = /(?<first>\w+)\s+((?<middle>\w\.)\s+)?(?<last>\w+)/

Какви са думите първи, среден и последен, които правят там? Те предоставят име

улавя: скоби, които можете да възстановите от обекта MatchData

използване на думи вместо числа.

Ако извършите съвпадение, използвайки този регулярен израз, ще видите доказателства за

наименовани заснемания в изхода на екрана, представляващи обекта MatchData:

>> m = re.match ("Дейвид А. Блек")

=> # <MatchData "Дейвид А. Блек" първо: "Дейвид" средна: "А". последно: "Черно">

Сега можете да заявите обекта за неговите имена за улавяне:

>> m [: първо]

=>"Дейвид"

Имените улавяния могат да натрупват вашите регулярни изрази, но с възвръщаемост, която

семантиката на извличане на заснемания от съвпадението става по-скоро базирана на думи, отколкото число

базирани на ber и следователно потенциално по-ясни и по-самодокументиращи се. Вие също не го правите

трябва да преброите двойки скоби, за да извлечете препратка към вашите заснети поднизове.

Обектите MatchData предоставят информация отвъд скобите в скоби, информация

мация, която можете да вземете и използвате, ако имате нужда от нея.

|  |
| --- |
| **341 серия** |

**341**

***Съвпадение, улавяне на поднизове и MatchData***

***11.4.4 Друга информация за MatchData***

Кодът в следващия списък, който е проектиран да бъде присаден към списък 11.1,

дава някои бързи примери за няколко допълнителни метода на MatchData.

print "Частта от низа преди съответстващата част е:"

поставя m.pre\_match

print "Частта от низа след съответстващата част е:"

поставя m.post\_match

print "Второто заснемане започна с характер"

поставя m.begin (2)

print "Третото заснемане завърши с характер"

поставя m.end (3)

Резултатът от този допълнителен код е както следва:

Низът до съответстващата част беше: Моят телефонен номер е

Низът след съответстващата част беше:.

Второто улавяне започна на 25-ти символ

Третото улавяне приключи на знак 33

Методите pre\_match и post\_match, които виждате в този списък, зависят от факта

че когато успешно съвпадате с низ, тогава низът може да се счита за такъв

съставен от три части: частта преди частта, която съвпада с модела; частта

които съвпадаха с модела; и частта след частта, която съвпада с модела. Всяка или

всичко това може да бъде празен низ. В този списък те не са: pre\_match и

низовете post\_match и двете съдържат символи (макар и само един знак в случая на

post\_match).

Можете също да видите методите за начало и край в този списък. Тези методи ви казват

където различните улавяния в скоби, ако има такива, започват и завършват. За да получите информация-

за улавяне на *n* , вие предоставяте *n* като аргумент за начало и / или край.

Обектът MatchData е своеобразен център за информация за информация какво се случва

pened, когато моделът отговаря на низа. С това знание на място, нека продължим

разглеждайки техники, които можете да използвате за изграждане и използване на регулярни изрази. Ще започнем

с няколко важни компонента на regexp: квантори, анкери и модификатори.

Изучаването на тези компоненти ще ви помогне и на двамата с написването на вашия собствен регистър

ular изрази и с вашата грамотност regexp. Ако съвпадението / abc / има смисъл за вас

сега, съвпадението /^x?[yz]{2}.\*\z/i ще ви има смисъл скоро.

Листинг 11.2 Допълнителен код за операции за съвпадение на телефонен номер

Глобалният обект **MatchData $ ~**

Всеки път, когато изпълнявате успешна операция на съвпадение, използвайки или съвпадение, или = ~, Ruby

задава глобалната променлива $ ~ на MatchData обект, представляващ съвпадението. Върху

неуспешен мач, $ ~ се настройва на нула. По този начин винаги можете да получите MatchData

обект, за аналитични цели, дори ако използвате = ~.

|  |
| --- |
| **342 серия** |

**342**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

***11.5 Фина настройка на регулярни изрази с квантори,***

***анкери и модификатори***

*Кванторите* позволяват да посочите колко пъти подред искате нещо да съвпада.

*Котвата* ви позволява да посочите, че съвпадението се случва в определена структурна точка в низ

(начало на низ, край на ред, при граница на дума и т.н.). *Модификаторите* са като

превключватели, които можете да обърнете, за да промените поведението на регулярния експрес; например от

което го прави без регистрация или променя начина, по който се справя с празното пространство.

Тук ще разгледаме кванторите, анкерите и модификаторите в този ред.

***11.5.1 Ограничаващи мачове с квантори***

Синтаксисът на Regexp ви дава начини да посочите не само какво искате, но и колко:

точно един от определен герой, 5–10 повторения на подпатерн и т.н.

Всички квантори работят или с един символ (който може да бъде представен

изречен от клас знаци) или в скобена група. Когато посочите, че вие

искат да съвпадат, да речем, три последователни повторения на даден подпатерн, това

по този начин subpattern може да бъде само един символ или може да бъде поставен по-дълъг subpattern

вътре в скоби.

Z ЕРО ИЛИ ЕДНО

Вече сте виждали пример за квантор с нула или един. Нека да го прегледаме и да отидем малко

по-дълбоко в него.

Искате да съответствате или на „г-н“ или на „г-жа“ - и само за да го направите по-интересно, вие

искате да поберете както американските версии, които завършват с точки, така и

Британски версии, които не го правят. Можете да опишете модела по следния начин:

знакът M, последван от символа r, последван от

*нула или един* от символите, последван от

*нула или един* от символа '.'

Regexp нотацията има специален знак, който представя ситуацията нула или едно:

въпросителен знак (?). Моделът, който току-що беше описан, ще бъде изразен в нотация на регулярни изрази

както следва:

/ Госпожо?

Въпросителният знак след *s* означава, че низ със *s* в тази позиция ще съвпада

модела, както и низ без *s* . Същият принцип важи и за буквала

период (обърнете внимание на обратната наклонена черта, указвайки, че това е действителен период, а не специален wild-

карта точка), последвана от въпросителен знак. Тогава целият модел ще съответства на „г-н“

„Госпожо“, „Господин“ или „Госпожо“ (Също така ще съответства на „ ABCM r.“ И „Госпожо!“, Но ще видите как да го направите

ограничете съвпадението по-точно, когато разгледаме анкери в раздел 11.5.3.)

Въпросният знак често се използва с класове символи, за да се посочи нула или един от

всеки от определен брой знаци. Ако търсите една или две цифри подред,

например, можете да изразите тази част от вашия модел по следния начин:

\ d \ d?

|  |
| --- |
| **343 серия** |

**343**

***Фина настройка на регулярни изрази с квантори, анкери и модификатори***

Тази последователност ще съответства на „1“, „55“, „03“ и т.н.

Заедно с нула или едно, има нула или повече квантор.

Z ЕРО ИЛИ ПОВЕЧЕ

Доста често срещан случай е случаят, в който низ, който искате да съчетаете, съдържа бяло-

пространство, но не сте сигурни колко. Да приемем, че се опитвате да съвпадате със затваряне </poem>

тагове в XML документ. Такъв маркер може или не може да съдържа празно пространство. Всички

те са еквивалентни:

</poem>

</ стихотворение>

</ стихотворение>

</ стихотворение

>

За да съответствате на маркера, трябва да разрешите непредсказуеми пропуски

по вашия модел - включително нито един.

Това е случаят с *нулевия или повече* квантора - звездичката или звездата (\*):

/ <\ s \* \ / \ s \* стихотворение \ s \*> /

Всеки път, когато се появи, последователността \ s \* означава, че низът, на който се съчетава, е разрешен

съдържат нула или повече празни знаци в този момент от съвпадението. (Обърнете внимание на

необходимост от избягване на наклонената черта в шаблона с обратна наклонена черта. В противен случай

ще се тълкува като наклонена черта, сигнализираща края на регулярния израз.)

Трябва да се отбележи, че регулярните изрази не могат да направят всичко. По-специално, това е a

често срещано и правилно наблюдение, че не можете да анализирате произволен XML с редовен

изрази, поради причини, свързани с влагането на елементи и начините в

кои данни от символи са представени. И все пак, ако сканирате документ, защото вие

искате да получите приблизително преброяване на броя на стихотворенията в него и вие съвпадате и броите

етикети на стихотворения, вероятността да получите информацията, която търсите, е голяма.

Следващата сред кванторите е една или повече.

О НЕ ИЛИ ПОВЕЧЕ

Един или повече квантор е знак плюс (+), поставен след символа или

тетичното групиране, на което искате да съответства една или повече. Съвпадението е успешно, ако низът

съдържа поне една поява на посочения подпатерн в подходящата точка.

Например моделът

/ \ d + /

съответства на всяка последователност от една или повече последователни цифри:

/\d+/.match("Тук има някаква цифра… ")

/\d+/.match(" Тук няма цифри. Придвижете се. ")

/\d+/.match("Digits-R-Us 2345 ")

Разбира се, ако хвърлите скоби, можете да разберете какво е съвпаднало:

/(\d+)/.match("Digits-R-Us 2345 ")

поставя $ 1

**Успева**

**Не успява**

**Успева**

|  |
| --- |
| **344 серия** |

**344**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

Резултатът тук е 2345.

Ето въпрос обаче. Работата на шаблона \ d + е да съвпада с една или повече цифри.

Това означава веднага щом двигателят на regexp (частта от преводача, която прави всичко това

pattern matching) вижда, че низът има цифрата 2 в себе си, той има достатъчно информация за

заключете, че да, има съвпадение. И все пак явно продължава; не спира да съвпада

низът, докато стигне чак до 5. Можете да изведете това от стойността на $ 1:

фактът, че $ 1 е 2345, означава, че подвиразът \ d +, който е в първия набор от

скоби, се счита, че съвпада с този подниз от четири цифри.

Но защо да съвпадате с четири цифри, когато всичко, което трябва да докажете, че сте прав, е една цифра? The

отговорът, както често се случва в живота, както и регексп анализът, е алчността.

***11.5.2 Алчни (и ненаситни) квантори***

Кванторите \* (нула или повече) и + (една или повече) са *алчни* . Това означава, че те

съвпадат с възможно най-много знаци, съобразени с това, че позволяват останалата част от шаблона

да съвпадне.

Вижте какво. \* Съвпада в този фрагмент:

низ = "abc! def! ghi!"

match = /.+!/.match(string)

поставя съвпадение [0]

Поискахме един или повече знаци (с помощта на заместваща точка), последвани от

удивителен знак. Може да очаквате да получите обратно подниза "abc!", Който се вписва

това описание.

Вместо това получаваме "abc! Def! Ghi!". Кванторът + лакомо изяжда толкова от

низ, както може и спира само на *последния* удивителен знак, а не на първия.

Можем да направим +, както и \* в не-алчни квантори, като поставим въпросителен знак

след тях. Вижте какво се случва, когато го направим с последния пример:

низ = "abc! def! ghi!"

match = /.+?!/.match(string)

поставя съвпадение [0]

Тази версия казва: „Дайте ми един или повече заместващи знака, но само толкова, колкото вие

вижте до *първия* удивителен знак, който също трябва да бъде включен. " Разбира се,

този път получаваме "abc!".

Ако добавим въпросителния знак към квантора в примера с цифри, той ще спре

след като види 2:

/(\d+?)/.match("Digits-R-Us 2345 ")

поставя $ 1

В този случай изходът е 2.

Какво означава да се каже, че алчните квантори ви дават толкова символи, колкото

те могат, „в съответствие с позволяването на останалата част от модела да съвпада“?

Помислете за това съвпадение:

/\d+5/.match("Цифри-R-Us 2345 ")

**Изход:**

**abc! def! ghi!**

**Изход: abc!**

|  |
| --- |
| **Страница 345** |

**345**

***Фина настройка на регулярни изрази с квантори, анкери и модификатори***

Ако алчността на един или повече квантора е абсолютна, \ d + ще съответства и на четирите

цифри - и тогава 5-те в шаблона няма да съвпадат с нищо, така че цялото съвпадение

ще се провали. Но алчността винаги се подчинява на осигуряването на успешен мач.

Това, което се случва, в този случай е, че след като мачът не успее, двигателят на regexp се връща назад:

той се сравнява с 5 и опитва модела отново. Този път успява: удовлетворява

както изискването \ d + (с 234), така и изискването 5 да следват цифрите, които

\ d + съвпада.

За пореден път можете да получите информативна рентгенова снимка на производството чрез заснемане

части от съответстващия низ и изследване на това, което сте заснели. Нека оставим irb и

Обектът MatchData ни показва съответните заснемания:

>> /(\d+)(5)/.match("Digits-R-Us 2345 ")

=> # <MatchData "2345" 1: "234" 2: "5">

Първото улавяне е "234", а второто е "5". Един или повече квантор, обаче

алчен, се е съгласил да получи само три цифри, вместо четири, в интерес на

инжектиране на regexp двигателя, за да намери начин да накара целия модел да съвпада с низа.

В допълнение към използването на модификаторите с нулев / един или повече стил, можете да изискате и

точен брой или числов диапазон от повторения на даден подпатерн.

S КОНКРЕТНИ БРОЯ НА ПОВТОРЕНИЯ

За да посочите точно колко повторения на част от вашия модел искате да съвпадат,

поставете числото в фигурни скоби ({}) веднага след съответния подизраз, тъй като това

пример показва:

/ \ d {3} - \ d {4} /

Този пример съвпада точно с три цифри, тире и след това с четири цифри: 555-1212

и други последователности, подобни на телефонен номер.

Можете също така да посочите диапазон вътре в скобите:

/ \ d {1,10} /

Този пример съвпада с всеки низ, съдържащ 1–10 последователни цифри. Едно число

последвано от запетая се интерпретира като минимум ( *n* или повече повторения). Можеш

следователно съответствайте на „три или повече цифри“ по следния начин:

/ \ d {3,} /

Двигателят на regexp на Ruby е достатъчно умен, за да ви уведоми дали вашият обхват е невъзможен;

ще получите фатална грешка, ако се опитате да съвпаднете, да речем, {10,2} (поне 10, но не повече от 2)

поява на подпатерн.

Можете да посочите, че повторението се брои не само за единични знаци или знаци

класове, но също така и за всеки *атом на* regexp - по-техническият термин за „част от вашия pat-

крачка. " Атомите включват скоби в скоби и класове символи, както и отделни

ual символи. По този начин можете да направите това, за да съответствате на пет последователни главни букви:

/([AZ]){5}/.match("David BLACK ")

Но в подобни случаи има важна потенциална ловушка.

|  |
| --- |
| **346 серия** |

**346**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

T HE ОГРАНИЧЕНИЕ В скоби

Ако изпълните последния ред код и погледнете за какво ви разказва обектът MatchData

при първото улавяне може да очаквате да видите „ЧЕРНО“. Но вие не:

>> /([AZ]){5}/.match("David BLACK ")

=> # <MatchData "ЧЕРНО" 1: "K">

Това е просто "К". Защо „ЧЕРНО“ не е уловено изцяло?

Причината е, че скобите не „знаят“, че се повтарят пет

пъти. Те просто знаят, че са първите скоби отляво (в частност

case) и че това, което са заснели, трябва да бъде скрито в първия слот за улавяне ($ 1, или

улавя [1] на обекта MatchData). Изразът в скобите, [AZ],

може да съответства само на един знак. Ако съвпада с един знак пет пъти подред, все още е

само едно по едно - и ще запомни само последното.

С други думи, съвпадение на един знак пет пъти не е същото като съвпадение на пет

символи еднократно.

Ако искате да уловите всичките пет знака, трябва да преместите скобите, така че те

приложете целия мач от пет части:

>> /([AZ]{5})/.match("David BLACK ")

=> # <MatchData "ЧЕРНО" 1: "ЧЕРНО">

Бъдете внимателни и с буквално мислене, когато става въпрос да разберете какво ще бъде заловено.

След това ще разгледаме начините, по които можете да посочите условията, при които искате

да се появят съвпадения, а не съдържанието, което очаквате да има низът.

***11.5.3 Котви и твърдения за регулярни изрази***

Твърденията и котвите са различни видове същества от герои. Когато ти

съвпадат с характер (дори спрямо клас характер или маска), което казва, че *кон-*

*сумиране* на символ в низ, който отговаряте. Твърдение или котва на

от друга страна, не консумира никакви символи. Вместо това изразява *ограничение* : условие

трябва да бъде изпълнено преди съвпадението на символите да бъде разрешено да продължи.

Най-често срещаните котви са *началото на реда* (^) и *края на реда* ($). Може би

използвайте котвата в началото на реда за задача като премахване на всички редове за коментар от

програмен файл на Ruby. Можете да постигнете това, като преминете през всички редове във файла

и отпечатвате само тези, които не *са* започнали с хеш-знак (#) или с празно пространство

последвано от знак за хеш. За да определите кои редове са редове за коментари, можете да използвате

това регулярно изражение:

/ ^ \ s \* # /

^ (Карета) в този модел *закрепва* мача в началото на ред. Ако останалата част от

моделът съвпада, но *не* в началото на реда, това не се брои - като вас

можете да видите с няколко теста:

>> comment\_regexp = / ^ \ s \* # /

=> / ^ \ s \* # /

>> comment\_regexp.match ("# Чист коментар!")

|  |
| --- |
| **Страница 347** |

**347**

***Фина настройка на регулярни изрази с квантори, анкери и модификатори***

=> # <MatchData "#">

>> comment\_regexp.match ("x = 1 # Код плюс коментар!")

=> нула

Съвпада само редът, който започва с някакво празно пространство и хеш символът

моделът на коментарите. Другият ред не съответства на модела и следователно

няма да бъде изтрито, ако сте използвали този регулярен израз за филтриране на коментари от файл.

Таблица 11.1 показва редица котви, включително начало и край на линия и начало

и край на низа.

Имайте предвид, че \ z съответства на абсолютния край на низа, докато \ Z съответства на края на

низ, с изключение на незадължителен последващ нов ред. \ Z е полезен в случаите, когато сте

не сте сигурни дали вашият низ има символ на нов ред в края - може би последния ред

прочетете от текстов файл - и не искате да се притеснявате за това.

Ръка за ръка с анкери се *твърдят* , че по същия начин казват на regexp процесора

че искате даден мач да се брои само при определени условия.

L УКАЗАНИЯ ЗА OOKAHEAD

Да приемем, че искате да съчетаете поредица от числа само ако тя завършва с точка. Но

не искате самият период да се брои като част от мача.

Един от начините да направите това е с *твърдение* за *търсене -* или, за да бъде завършено, с нулева ширина,

положително твърдение. Тук, последвано от допълнително обяснение, е как го правите:

str = "123 456. 789"

m = /\d+(?=\.)/.match(str)

В този момент, m [0] (представляващ целия участък от низа, който рисува

съвпада) съдържа 456 - едната последователност от числа, последвана от точка.

Ето малко повече коментари за някои от терминологиите:

*Нулевата ширина* означава, че не консумира никакви символи в низа. Председателят

ence of the period е отбелязан, но все пак можете да сравните периода, ако вашият модел

продължава.

Таблица 11.1 Котви за регулярен израз

Нотация

Описание

Пример

Примерен низ за съвпадение

^

Начало на линията

/ ^ \ s \* # /

"# Коментарен ред в Ruby с

водещи пространства "

$

Край на реда

/\.$/

"един \ nдве \ nтри. \ nчетири"

\ A

Начало на низ

/ \ AЧети резултат /

"Четири точки"

\ z

Край на низа

/ от земята. \ z / "от земята."

\ Z

Край на низа (с изключение на

за окончателен нов ред)

/ от земята. \ Z / "от земята \ n"

\ b

Граница на думите

/ \ b \ w + \ b /

"!!! дума \*\*\*" ( съвпада с "дума")

|  |
| --- |
| **348 серия** |

**348**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

*Положително* означава, че искате да посочите периода да присъства. Също така има

*отрицателни* главни глави; те използват (?! ...), а не (? = ...).

*Твърдението Lookahead* означава, че искате да знаете, че посочвате какво *би* било

следващ, без да го съвпада.

Когато използвате твърдение lookahead, скобите, в които поставяте looka-

главна част от мача не се брои; $ 1 няма да бъде зададен от операцията за съвпадение в

пример. И точката след 6 няма да бъде погълната от мача. (Запазете това последно

имайте предвид, ако някога сте озадачени от поведението на главата; озадачаването често

идва от забравянето, че гледането напред не е същото като движение напред.)

L OOKBEHIND ТВЪРДЕНИЯ

Твърденията на lookahead имат задни еквиваленти. Ето един регулярен израз, който съвпада

низът ЧЕРЕН само когато е предшестван от „Дейвид“:

re = / (? <= David) ЧЕРНО /

И обратно, ето един, който му отговаря, само когато не е предшестван от „Дейвид“:

re = / (? <! David) ЧЕРНО /

Още веднъж, имайте предвид, че това са твърдения с нулева ширина. Те представляват кон-

стрейтове на низа ( „*Дейвид“ трябва да е преди него, или този „ ЧЕРЕН “ не се брои за съвпадение* ),

но те не съвпадат или консумират никакви символи.

Има и такова нещо като *условен мач* .

С НЕОБХОДИМИ МАЧОВЕ

Макар че вероятно няма да бъде сред ежедневните ви практики за регулярно изразяване, това е така

интересно е да се отбележи съществуването на условни съвпадения в редовните изрази на Ruby 2.0

sion engine (име на проекта Onigmo). Тестове за условно съвпадение за определена граница

ture (по номер или име) и съответства на един от двата под израза въз основа на това дали

или не е намерено улавянето.

Незахващащи скоби

Ако искате да съчетаете нещо - не просто твърдете, че е следващото, но всъщност го сравнявайте -

използване на скоби, но не искате да се брои като един от номерираните скоби-

ical улавя в резултат на съвпадението, използвайте конструкцията (?: ...). Всичко вътре в

(? :) групирането ще бъде съпоставено въз основа на групирането, но няма да бъде запазено в заснемане.

Имайте предвид, че обектът MatchData, произтичащ от следващото съвпадение, има само две капачки

каси; групирането def не се брои, поради?: notation:

>> str = "abc def ghi"

=>"abc def ghi"

>> m = / (abc) (?: def) (ghi) /. съвпадение (str)

=> # <MatchData "abc def ghi" 1: "abc" 2: "ghi">

За разлика от твърдението с нулева ширина, групата (? :) наистина консумира символи. Просто няма

запишете ги като улавяне.

|  |
| --- |
| **349 серия** |

**349**

***Фина настройка на регулярни изрази с квантори, анкери и модификатори***

Ето един прост пример. Условният израз (? (1) b | c) съвпада с b, ако cap-

номер 1 е съчетан; в противен случай съвпада с c:

>> re = / (a)? (? (1) b | c) /

=> / (a)? (? (1) b | c) /

>> re.match ("ab")

=> # <MatchData "ab" 1: "a">

>> re.match ("b")

=> нула

>> re.match ("c")

=> # <MatchData "c" 1: нула>

Регулярният израз re съвпада с низа "ab"B , като "a" е първият скоби-

улавяне на кал и условния подизраз, съвпадащ с "a". Повторно обаче не го прави

съвпада с низа "b"c . Тъй като няма първо улавяне в скоби, условието

национален подизраз се опитва да съвпадне с "c" и не успява d . Това е и причината отново *е* да съответства на

низ "c": условието (? (1) ...) не е изпълнено, така че изразът се опитва да съответства на "else"

част от себе си, което е субекспресията / c /.

Можете също така да напишете условни регулярни изрази, като използвате именувани улавяния. Предварително

пример за отстъпване ще изглежда така:

/ (? <първи> а)? (? (<първи>) b | c) /

и резултатите от различните мачове биха били едни и същи.

Котвите, твърдения и условни съвпадения добавят богатство и детайлност към

език на шаблона, с който изразявате търсените съвпадения. Също така в

категорията за обогатяване на езика са модификатори на regexp.

***11.5.4 Модификатори***

*Модификаторът на* regexp е буква, поставена след последната, затваряща наклонена черта на литерала на regex:

/ abc / i

Модификаторът i, показан тук, води до съвпадение на операциите за съвпадение, включващи този регекс

безчувствен. Другият най-често срещан модификатор е m. Модификаторът m (многоредов) има

ефект, че точката на заместващия знак, която обикновено съвпада с *всеки символ, с изключение на new-*

*линия* , ще съответства на *всеки символ, включително нов ред* . Това е полезно, когато искате да заснемете

всичко, което се намира между, да речем, отваряща скоба и затваряща, и вас

не знам (или не се интересувам) дали са на една линия.

Ето пример; обърнете внимание на вградените символи от нов ред (\ n) в низа:

str = "Това (включително \ n какво е в парените \ n) заема три реда."

m = /\(.\*?\)/m.match(str)

Не-алчният подсистема за заместващи символи. \*? съвпадения:

(включително \ n какво има в парените \ n)

Без модификатора m, точката в подпатерна не би съответствала на символа за нов ред

ters. Операцията за съвпадение би ударила първата нова линия и, без да намери а) знак от

този момент, ще се провали.

б

° С

д

|  |
| --- |
| **Страница 350** |

**350**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

Друг често използван модификатор на regexp е x. Модификаторът x променя начина на

regexp парсер третира празно пространство. Вместо да го включите буквално в модела, то

игнорира го, освен ако не е избегнато с обратна наклонена черта. Смисълът на модификатора x е да ви позволи

добавете коментари към вашите регулярни изрази:

/

\ ((\ d {3}) \) # 3 цифри в литерални парени (код на областта)

\с

# Един интервал

(\ d {3}) # 3 цифри (обмен)

-

# Тире

(\ d {4}) # 4 цифри (втора част от числото

/х

Предишното регулярно изражение е точно същото като това, но с разширен синтаксис и

коментари:

/ \ ((\ d {3}) \) \ s (\ d {3}) - (\ d {4}) /

Бъдете внимателни с модификатора x. Когато го откриете за първи път, е изкушаващо да унищожите всичките си

модели широко отворени:

/ (? <= Дейвид \) ЧЕРНО / x

(Обърнете внимание на обратната наклонена черта буквален интервал, единственият такъв знак, който ще

да се счита за част от модела.) Но не забравяйте, че много програмисти имат

са се обучили да разбират регулярните изрази без много привидно потребителски

добавено приятелско допълнително празно пространство. Не е лесно да деинсталирате регексп, докато го четете, ако

сте свикнали със стандартния синтаксис.

В по-голямата си част модификаторът x е най-добре запазен за случаите, когато искате да се счупите

регулярният израз на няколко реда с цел добавяне на коментари, както в теле-

пример за телефонен номер. Не приемайте, че празното пространство автоматично се прави редовно

изрази по-четливи.

След това ще разгледаме техниките за преобразуване напред-назад между две различни

но тясно свързани класове: String и Regexp.

***11.6 Конвертиране на низове и регулярни изрази***

***един на друг***

Фактът, че регулярните изрази не са низове, е лесен за възприемане с един поглед в случая

на регулярни изрази като този:

/ [ac] {3} /

Със своя специален синтаксис за класове символи и повторение, този модел не изглежда много

като всеки от низовете, които съответства ("aaa", "aab", "aac" и т.н.).

Става малко по-трудно *да не* се види пряка връзка между регулярния израз и низ когато

изправен пред регулярно изражение като това:

/ abc /

|  |
| --- |
| **351 серия** |

**351**

***Преобразуване на низове и регулярни изрази един в друг***

Този регулярен израз не е низът "abc". Освен това съвпада не само с „abc“, но и с всеки низ

с подниза „abc“ някъде вътре в него (като „Сега знам своите abcs.“). Няма

уникална връзка между низ и подобен на регекс експрес.

И все пак, въпреки че визуалната прилика между някои струни и някои редовни

изрази не означава, че са едно и също нещо, регулярните изрази и низовете го правят

взаимодействат по важни начини. Нека да разгледаме някакъв поток в посока низ към регулярно изражение

а след това някои вървят по обратния път.

***11.6.1 Идиоми за низ към регулярно изражение***

Като начало можете да извършите интерполация на низ (или стил на низ) в рамките на регулярно изражение.

Правите го с познатата техника за интерполация # {...}:

>> str = "def"

=>"def"

>> / abc # {str} /

=> / abcdef /

Стойността на str се изпуска в регулярния експрес и става част от него, точно както би било, ако

използвахте същата техника, за да го интерполирате в низ.

Техниката на интерполация става по-сложна, когато низът сте

интерполирането съдържа специални символи за регулярни изрази. Например, помислете за низ с кон-

задържане на точка (.). Както знаете, точката или точката имат специално значение в редовно

изрази: отговаря на всеки отделен символ, с изключение на нов ред. В низ, това е просто точка.

Когато става въпрос за интерполиране на низове в регулярни изрази, това има потенциал

да предизвика объркване:

>> str = "ac"

=>"променлив ток"

>> re = / # {str} /

=> / ac /

>> re.match ("ac")

=> # <MatchData "ac">

>> re.match ("abc")

=> # <MatchData "abc">

И двата мача успяват; те връщат обекти MatchData, а не нула. Точката в

pattern съвпада с точка в низа "ac". Но също така съвпада с b в "abc". Точката,

която започна живота като просто точка вътре в str, придобива специален смисъл, когато стане

част от регулярния израз.

Но можете да *избягате* от специалните знаци в низ, преди да изпуснете низа

в регулярно изражение. Не е нужно да правите това ръчно: Regexp класът предоставя a

Regexp.escape метод на класа, който го прави вместо вас. Можете да видите какво прави този метод

като го пуснете на няколко низа изолирано:

>> Regexp.escape ("ac")

=>"a \\. c"

>> Regexp.escape ("^ abc")

=>"\\ ^ abc"

|  |
| --- |
| **352 серия** |

**352**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

(irb удвоява обратната наклонена черта, защото извежда двойно кавирани низове. Ако желаете, вие

can поставя изразите и ще ги видите в истинския им вид с единични наклонени черти.)

В резултат на този вид избягване можете да ограничите вашите регулярни изрази до

съвпадат точно с низовете, които интерполирате в тях:

>> str = "ac"

=>"променлив ток"

>> re = /#{Regexp.escape(str)}/

=> /a\.c/

>> re.match ("ac")

=> # <MatchData "ac">

>> re.match ("abc")

=> нула

Този път опитът да се използва точката като символ за съвпадение на заместващ знак е неуспешен; "abc" не е

съвпадение за избягал, интерполиран низ.

Възможно е също да създадете екземпляр на регулярно изражение от низ, като го предадете на

Regexp.new:

>> Regexp.new ('(. \*) \ S + Black')

=> /(.\*)\s+Black/

Прилага се обичайната логика за избягване на символи и / или избягване на регулярни изрази:

>> Regexp.new („Г-н \. Дейвид Блек“)

=> / Г-н \. Дейвид Блек /

>> Regexp.new (Regexp.escape ("Г-н. Дейвид Блек"))

=> / Г-н \. \ Дейвид \ Блек /

Забележете, че буквалните интервали са избягали с наклонени черти - не

строго необходимо, освен ако не използвате модификатора x, но не и вредно.

Можете също така да предадете буквален regexp на Regexp.new, в който случай получавате обратно нов,

идентични регулярни изрази. Тъй като винаги можете просто да използвате буквалния регулярно изражение на първо място,

Regexp.new е по-често използван за конвертиране на низове в регулярни изрази.

Използването на низове с един кавички прави ненужно удвояването на гърба

наклонени черти. Ако използвате двойни кавички (което може да се наложи в зависимост от вида

интерполация, която трябва да направите), не забравяйте, че трябва да напишете Mr \\. така че отзад-

наклонена черта е част от низа, предаден на конструктора на regexp. В противен случай ще има само

ефектът от поставянето на буквална точка в низа - което така или иначе щеше да се случи -

и тази точка ще влезе в регулярния израз без наклонена черта и следователно ще бъде

зададен като заместваща точка.

Сега нека разгледаме някои техники за преобразуване в другата посока: regexp към

низ. Това е нещо, което ще правите най-вече за отстраняване на грешки и анализи.

***11.6.2 Преминаване от регулярен израз към низ***

Както всички обекти на Ruby, регулярните изрази могат да се представят под формата на низове.

Начинът, по който правят това, в началото може да изглежда странно:

>> поставя / abc /

(? -mix: abc)

|  |
| --- |
| **353 серия** |

**353**

***Често използвани методи, които използват регулярни изрази***

Това е алтернативна нотация на регулярни изрази - такава, която рядко вижда бял свят, освен когато

генерирани от метода to\_s instance на regexp обекти. Това, което изглежда като *микс,* е списък

на модификаторите (m, i и x) със знак минус отпред, показващ, че модификаторите са

всички изключени.

Можете да играете с поставянето на регулярни изрази в irb и ще видите повече за

как работи тази нотация. Тук няма да го преследваме, отчасти защото има друг начин

за да получите низово представяне на regexp, което прилича по-скоро на това, което вероятно сте

въведени - чрез извикване на inspect или p (което от своя страна извиква inspect):

>> /abc/.inspect

=>"/ abc /"

Преминаването от регулярни изрази към низове е полезно преди всичко, когато учите

и / или отстраняване на неизправности с регулярни изрази. Това е добър начин да се уверите, че сте редовни

изразите са това, което мислите, че са.

На този етап ще представим пълен кръг с регулярни изрази, като разгледаме ролите, които те имат

играйте в някои важни методи от други класове. Стигнахме дотук, използвайки мача

метод почти изключително; но съвпадението е само началото.

***11.7 Общи методи, които използват регулярни изрази***

Отплатата за получаване на съоръжение с регулярни изрази в Ruby е възможността да се използва

методи, които приемат регулярни изрази като аргументи и правят нещо с тях.

Като начало винаги можете да използвате операция за съвпадение като тест, да речем, при намиране или

find\_all операция върху колекция. Например, за да намерите всички низове, по-дълги от 10

знаци и съдържащи поне 1 цифра от масив от низове, наречен масив, вие

мога да направя това:

array.find\_all {| e | e.size> 10 и /\d/.match(e)}

Но редица методи, най-вече отнасящи се до низовете, се базират по-директно на

използване на регулярни изрази. Ще разгледаме няколко от тях в този раздел.

***11.7.1 Сканиране на низ #***

Методът за сканиране преминава отляво надясно през низ, като многократно се тества за a

съвпадат с модела, който сте посочили. Резултатите се връщат в масив.

Например, ако искате да съберете всички цифри в низ, можете да направите това:

>>"тестване 1 2 3 тестване 4 5 6" .scan (/ \ d /)

=> ["1", "2", "3", "4", "5", "6"]

Имайте предвид, че сканирането прескача неща, които не съвпадат с неговия модел, и търси съвпадение

по-късно в низа. Това поведение е различно от това на мача, който спира завинаги

когато завърши съвпадението с модела напълно веднъж.

Ако използвате групи в скоби в регулярния експрес, който давате за сканиране, операцията

връща масив от масиви. Всеки вътрешен масив съдържа резултатите от едно сканиране

низът:

|  |
| --- |
| **354 серия** |

**354**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

>> str = "Леополд Ауер беше учител на Jascha Heifetz."

=>"Леополд Ауер беше учител на Яша Хайфец."

>> цигулари = str.scan (/ ([AZ] \ w +) \ s + ([AZ] \ w +) /)

=> [["Leopold", "Auer"], ["Jascha", "Heifetz"]]

Този пример ви свързва масив от масиви, където всеки вътрешен масив съдържа първия

име и фамилия на човек. Като всяко пълно име се съхранява в свое собствено

array улеснява прелистването на целия списък с имена, който сме удобно

скрит в променливата цигулари:

цигулари.всеки | fname, lname |

поставя "Името на # {lname} е # {fname}."

край

Резултатът от този фрагмент е както следва:

Първото име на Ауер беше Леополд.

Първото име на Хайфец беше Яша.

Regexp, използван за имена в този пример, разбира се, е твърде прост: той пренебрегва

тирета, бащини имена и т.н. Но това е добра илюстрация за това как да използвате

скации.

Сканирането на низ # също може да вземе кодов блок - и тази техника понякога може да запази

ти стъпка. сканирането дава своите резултати на блока и детайлите на получаването зависят

за това дали използвате скоби в скоби. Ето пренаписване на базирано на сканиране блок

предишния код:

str.scan (/ ([AZ] \ w +) \ s + ([AZ] \ w +) /) do | fname, lname |

поставя "Името на # {lname} е # {fname}."

край

Всеки път през низа блокът получава улавянията в масив. Ако не сте

като прави улавяне, блокът последователно получава съвпадащите поднизове. Сканиране

за бучки от символи \ w (\ w е класът символи, състоящ се от букви, цифри,

и подчертаване) може да изглежда така

"едно две три" .scan (/ \ w + /) {| n | поставя "Следващ номер: # {n}"}

които биха произвели този резултат:

Следващ номер: един

Следващо число: две

Следващо число: три

Имайте предвид, че ако предоставите блок, сканирането не съхранява резултатите в масив и не ги връща

тях; той изпраща всеки резултат към блока и след това го изхвърля. По този начин можете да сканирате

чрез дълги струни, като правите нещо с резултатите по пътя и избягвайте да вземате

нагоре памет с поднизовете, които вече сте виждали и използвали.

Друга често срещана низова операция, базирана на regexp, е разделена.

|  |
| --- |
| **355 серия** |

**355**

***Често използвани методи, които използват регулярни изрази***

***11.7.2 Разделяне на низ #***

В съответствие с името си, split разделя низ на множество поднизове, връщайки се

тези поднизове като масив. split може да вземе или regexp, или обикновен низ като sep-

аратор за операцията за разделяне. Обикновено се използва за получаване на масив, състоящ се от всички

символи в низ. За да направите това, използвате празен регулярен израз:

>>"Ruby" .split (//)

=> ["R", "u", "b", "y"]

split често се използва в процеса на конвертиране на плоски, базирани на текст конфигурационни файлове в

Структури от данни на Ruby. Обикновено това включва преминаване през файл ред по ред и кон-

вертиране на всеки ред. Преобразуването от един ред може да изглежда така:

line = "first\_name = david; last\_name = black; country = usa"

запис = line.split (/ = |; /)

Това оставя запис, съдържащ масив:

["first\_name", "david", "last\_name", "black", "country", "usa"]

Още повече сканиране на **низове с** класа **StringScanner**

Стандартната библиотека включва разширение, наречено strscan, което предоставя

Клас StringScanner. StringScanner обектите разширяват наличния набор от инструменти за сканиране

ning и изследване на струни. Обект StringScanner поддържа указател в

низ, позволяващ движение напред-назад през струната, използвайки позиция и

семантика на указателя.

Ето някои примери за методите в StringScanner:

>> изисква 'strscan'

=> вярно

>> ss = StringScanner.new ("Тестване на сканиране на низове")

=> # <StringScanner 0/23 @ "Testi ...">

>> ss.scan\_until (/ ing /)

=>"Тестване"

>> ss.pos

=> 7

>> ss.peek (7)

=>"низ"

>> ss.unscan

=> # <StringScanner 0/23 @ "Testi ...">

>> ss.pos

=> 0

>> ss.skip (/ Тест /)

=> 4

>> ss. почивка

=>"сканиране на низ"

Използвайки понятието за указател в низа, StringScanner ви позволява да преминете през

низа, както и да разгледате какво вече е съвпаднало и какво остава. Низ -

Скенерът е полезно допълнение към вградените съоръжения за сканиране на низове.

**Зарежда библиотека на скенера**

**Създава скенер**

**Сканира низ до**

**regexp съвпада**

**Разглежда нови**

**позиция на показалеца**

**Разглежда следващите 7 байта (но**

**не напредва показалеца)**

**Отменя предишното сканиране**

**Премества показалеца покрай регулярния израз**

**Разглежда част от низа**

**вдясно от показалеца**

|  |
| --- |
| **356 серия** |

**356**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

С малко повече работа можете да попълните хеш със записи от този вид:

данни = []

запис = хеш [\* line.split (/ = |; /)]

data.push (запис)

Ако направите това за всеки ред във файл, ще имате масив от хешове, представляващи всички

записи. Този масив от хешове от своя страна може да се използва като точка на въртене към следващ оператор

ация - може би вграждане на информацията в отчет или подаване в библиотека

tine, който може да го запише в таблица на базата данни като последователност от двойки колона / стойност.

Можете да предоставите втори аргумент за разделяне; този аргумент ограничава броя на

върнати елементи. В този пример

>>"a, b, c, d, e" .split (/, /, 3)

=> ["a", "b", "c, d, e"]

split спира да се разделя, след като има три елемента за връщане и поставя всичко, което е

вляво (запетаи и всички) в третия низ.

В допълнение към разбиването на низ на части чрез сканиране и разделяне, можете също

променяйте части от низ с операции за заместване, както ще видите по-нататък.

***11.7.3 под / под! и gsub / gsub!***

sub и gsub (заедно с взрива им, еквиваленти на място) са най-често срещаните инструменти

за промяна на съдържанието на низове в Ruby. Разликата между тях е тази

gsub ( *гр* сплитка кепър *под* конституция) прави промени в целия низ, докато под прави най-

най-много една замяна.

S ИНГЛИРАЙТЕ ЗАМЕНИТЕЛИ С SUB

sub взема два аргумента: регулярно изражение (или низ) и заместващ низ. Както и да е

част от низа съответства на регулярния израз, ако има такъв, се премахва от низа и се заменя

със заместващия низ:

>>"типографска грешка" .sub (/ i /, "o")

=>"печатна грешка"

Можете да използвате кодов блок *вместо* аргумента за заместващ низ. Блокът е

извикан (отстъпен), ако има съвпадение. Повикването преминава в низа, който се заменя като

аргумент:

>>"главна буква на първата гласна" .sub (/ [aeiou] /) {| s | s.upcase}

=>"cАпитализиране на първата гласна"

Ако сте направили някакво групиране в скоби, глобалните $ *n* променливи са зададени и налични

за използване вътре в блока.

G ЛОБАЛНИ ЗАМЕСТИТЕЛИ С GSUB

gsub е като sub, освен че продължава да замества, докато моделът съвпада някъде

в низа. Например, ето как можете да замените първата буква от всяка дума в

низ със съответната главна буква:

>>"изписване с главни букви на всяка дума" .gsub (/ \ b \ w /) {| s | s.upcase}

=>"Изписване с главни букви на всяка дума"

**Използвайте \*, за да превърнете масива в**

**гол списък за подаване в Hash []**

|  |
| --- |
| **Страница 357** |

**357**

***Често използвани методи, които използват регулярни изрази***

Както при sub, gsub ви дава достъп до променливите за улавяне на $ *n в* скоби в

кодов блок.

U пее улавя в ПОДМЯНА STRING

Можете да получите достъп до скобите в скоби, като използвате специална нотация, състояща се от

числа, избягали от наклонената черта. Например можете да коригирате появата на по-ниско-

главна буква, последвана от главна буква (ако приемем, че се справяте със ситуация

където това е грешка) като това:

>>"aDvid" .sub (/ ([az]) ([AZ]) /, '\ 2 \ 1')

=>"Дейвид"

Обърнете внимание на използването на единични кавички за заместващия низ. С двойно

кавички, ще трябва да удвоите обратната наклонена черта, за да избегнете обратната наклонена черта.

За да удвоите всяка дума в низ, можете да направите нещо подобно, но с помощта на gsub:

>>"удвояване на всяка дума" .gsub (/ \ b (\ w +) /, '\ 1 \ 1')

=>"двойно удвояване на всяка дума дума"

Ще завършим нашия поглед върху инструментите, базирани на regexp, с две техники,

mon тяхната зависимост от оператора за равенство на случаи (===): изявления за case (които

не са извиквания на методи, но които включват повиквания към оператора threequal) и

Безброй # grep.

***11.7.4 Равенство между делата и греп***

Както знаете, всички обекти на Ruby разбират съобщението ===. Ако не е било отменено-

den в даден клас или за даден обект, това е синоним на ==. Ако е било отменено,

това е каквото и да е новата версия.

Равенството на случаите за регулярни изрази е тест за съвпадение: за всеки даден *регулярен израз* и *низ* ,

*regexp* === *низ* е вярно, ако *низът* съответства на *regexp* . Можете да използвате === изрично като тест за съвпадение:

поставя "Мач!" ако re.match (низ)

поставя "Мач!" ако низ = ~ re

поставя "Мач!" ако re === низ

Глобална променлива за улавяне

Внимавайте: Можете да използвате глобалните променливи за улавяне ($ 1 и т.н.) във вашия заместващ низ,

но те може да не правят това, което мислите, че ще направят. По-конкретно, ще бъдете уязвими за ляво-

над стойностите за тези променливи. Помислете за този пример:

>> /(abc)/.match("abc ")

=> # <MatchData "abc" 1: "abc">

>>"aDvid" .sub (/ ([az]) ([AZ]) /, "# {$ 2} # {$ 1}")

=>"abcvid"

Тук $ 1 от предишния мач ("abc") в крайна сметка прониква в низа за заместване

във втория мач. Като цяло, придържайки се към препратките в стила \ 1 към вашите снимки

е по-безопасно, отколкото използването на глобалните променливи за улавяне в подменютата за под и gsub.

|  |
| --- |
| **358 серия** |

**358**

C ГЛАВА 11 ***Регулярни изрази и низови операции, базирани на регулярни изрази***

И, разбира се, трябва да използвате който и тест да ви даде това, от което се нуждаете: нула или

MatchData обект за съвпадение; нулево или цяло число отместване за = ~; вярно или невярно за ===.

В случай на изявления, === се използва имплицитно. За да тествате за различни съвпадения на шаблони в a

делото, продължете по следните линии:

print "Продължаване? (г / н)"

отговор = получава

отговор на случая

когато / ^ y / i

поставя "Страхотно!"

когато / ^ n / i

поставя "Чао!"

изход

друго

поставя "А?"

край

Всяка клауза когато е повикване към ===: / ^ y / i === отговор и т.н.

Другата техника, която сте виждали и която използва метода === / оператор, също предполага

itly, е Enumerable # grep. Можете да се върнете към раздел 10.3.3. Тук ще поставим мястото-

осветление на няколко аспекта на това как се справя с низовете и регулярните изрази.

grep прави операция за филтриране от безброй обекти въз основа на случая

оператор за равенство (===), връщащ всички елементи в изброеното, които връщат a

истинска стойност, когато три се равнява на аргумента на grep. Така ако аргументът за grep

е регулярно изражение, изборът се основава на съвпадения на шаблоните според поведението на

Regexp # ===:

>> [„САЩ“, „Великобритания“, „Франция“, „Германия“]. Grep (/ [az] /)

=> ["Франция", "Германия"]

Можете да постигнете същото с select, но е малко по-сложно:

[„САЩ“, „Великобритания“, „Франция“, „Германия“]. Изберете {| c | / [az] / === c}

grep използва обобщената техника threequal, за да направи специализирана избрана опера-

, включително, но не само тези, които включват низове, кратки и удобни.

Можете също така да предоставите кодов блок на grep, като в този случай получавате комбиниран

операция за избор / картографиране: резултатите от операцията за филтриране се представят един по един

към блока, а върнатата стойност на цялото извикване на grep е кумулативният резултат от

тези добиви. Например, за да изберете държави и след това да ги съберете с главни букви, вие

мога да направя това:

>> [„САЩ“, „Великобритания“, „Франция“, „Германия“]. Grep (/ [az] /) {| c | c.upcase}

=> ["ФРАНЦИЯ", "ГЕРМАНИЯ"]

Имайте предвид, че grep избира въз основа на оператора за равенство на букви (===), така че няма

изберете нещо различно от низове, когато му дадете регулярно изражение като аргумент - и

няма автоматично преобразуване между числа и низове. По този начин, ако опитате това

[1,2,3] .grep (/ 1 /)

|  |
| --- |
| **359 серия** |

**359**

***Обобщение***

получавате обратно празен масив; масивът няма низов елемент, който да съответства на регулярното изражение

/ 1 /, няма елемент, за който е вярно, че / 1 / === елемент.

Това ни води до края на нашето проучване на регулярни изрази и някои от

методи, които ги използват. Има още какво да научите; съвпадението на шаблона е разтегнат под-

джект. Но тази глава ви запозна с голяма част от това, което вероятно ще ви е необходимо

вижте, докато продължите с проучването и използването на Ruby.

***11.8 Обобщение***

В тази глава сте виждали

Основните принципи на съвпадението на модела на регулярния израз

Техники за съвпадение и = ~

Класове знаци

Парентетични улавяния

Квантори

Котви

MatchData обекти

Интерполация и преобразуване на низ / регулярно изражение

Ruby методи, които използват регулярни изрази: сканиране, разделяне, grep, под, gsub

Тази глава ви запозна с основите на регулярните изрази в Ruby,

включително класове символи, скоби и скоби. Видяхте това

ular изразите са обекти - по-специално обекти от класа Regexp - и че те

отговаряйте на съобщения (като „съвпадение“). Разгледахме класа MatchData, екземпляри на

които съдържат информация за резултатите от мачова операция. Вие също сте научили

как да се интерполират низове в регулярни изрази (избягали или не ескапирани, в зависимост

дали искате специалните знаци в низа да бъдат третирани като специални в

regexp), как да създадете екземпляр на регулярно изражение от низ и как да генерирате низ, представящ

усещане за регулярно изражение.

Методи като String # scan, String # split, Enumerable # grep и "sub" семейството

на String методите използват регулярни изрази и съвпадение на шаблони като начин за определяне

копаят как трябва да се прилагат техните действия. Придобиване на знания за редовни

sions ви дава достъп не само до относително прости методи за съвпадение, но и до a

набор от инструменти за обработка на низове, които иначе не биха били използваеми.

Докато продължаваме разследването на вградените съоръжения на Ruby, ще преминем към глава 12

към предмета на I / O операциите като цяло и по-специално обработката на файлове.

|  |
| --- |
| **Страница 360** |

**360**

*Файлови и I / O операции*

Както ще видите, след като се гмурнете, Ruby поддържа четен файл и обекти на I / O операции-

ориентирана. Входни и изходни потоци, като стандартния входен поток или, за този мат

ter, всеки дескриптор на файлове, са обекти. Някои I / O-свързани команди са по-процедурни:

поставя например системния метод, който ви позволява да изпълнявате системна команда.

Но путовете са процедурни само когато работят на стандартния изходен поток.

Когато поставяте ред във файл, вие изрично изпращате съобщението „put“ към обект File.

Пространството на паметта на програма Ruby е един вид идеализирано пространство, където

обектите възникват и разговарят помежду си. Предвид факта, че I / O и sys-

изпълнението на командата включва излизане извън това идеализирано пространство, Руби прави a

много, за да запазите обектите в микса.

Ще видите повече обсъждане на стандартните библиотеки (за разлика от основните) в

тази глава от където и да е другаде в книгата. Това е така, защото обработката на файлове

***Тази глава обхваща***

Вход за клавиатура и изход на екрана

Класовете IO и File

Стандартни библиотечни файлове, включително

FileUtils и Pathname

Функциите на библиотеката StringIO и open-uri

|  |
| --- |
| **361 серия** |

**361**

***Как се събира I / O системата на Ruby***

съоръжения в стандартната библиотека - подчертани от FileUtils, Pathname и

Пакетите StringIO са толкова мощни и толкова гъвкави, че са постигнали един вид

квазиядрен статус. Шансовете са, че ако правите някаква файлоемка Ruby програма-

ming, ще стигнете до точката, в която зареждате тези пакети почти без да мислите

за това.

***12.1 Как се събира системата за вход / изход на Ruby***

Класът IO обработва всички входни и изходни потоци самостоятелно или чрез неговия низходящ

класове, особено File. До голяма степен API на IO се състои от обвивки около sys-

разговори в библиотеката, с някои подобрения и модификации. Колкото сте по-познати

са със стандартната библиотека C, колкото по-у дома ще се чувствате с методи като търси,

getc и eof ?. По същия начин, ако сте използвали друг език на високо ниво, който също има

доста плътно прилепващ API на обвивката около тези библиотечни методи, ще разпознаете техните

еквиваленти в Ruby. Но дори и да не сте програмист или програмист на C, ще получите

закачете го бързо.

***12.1.1 Класът IO***

IO обектите представляват четими и / или записваеми връзки към дискови файлове, клавиатури,

екрани и други устройства. Третирате IO обект като всеки друг обект: изпращате го

съобщения и изпълнява методи и връща резултатите.

Когато се стартира програма Ruby, тя е наясно със стандартния вход, изход и грешка

потоци. И трите са капсулирани в случаи на IO. Можете да ги използвате, за да разберете

за това как работи един прост IO обект:

>> STDERR.клас

=> IO

>> STDERR.puts ("Проблем!")

Проблем!

=> нула

>> STDERR.write ("Проблем! \ N")

Проблем!

=> 9

Константите STDERR, STDIN и STDOUT (всички те ще бъдат разгледани подробно след

12.1.3) се настройват автоматично при стартиране на програмата. STDERR е IO обект B .

Ако IO обект е отворен за писане (което е STDERR, защото цялата цел е в

извеждане на състояние и съобщения за грешки към него), можете да се обадите поставя на него и каквото и да е

put ще бъде записан в изходния поток на този IO обект c . В случай на STDERR - в

най-малкото, в стартиращата ситуация по подразбиране - това е изискан начин да се каже, че ще бъде записан-

десет на екрана.

В допълнение към путове, IO обектите имат метод за печат и метод за запис. Ако ти

пишете в IO обект, няма автоматично извеждане на нов ред (записването е по-скоро като печат

отколкото поставя в това отношение), а върнатата стойност е броят на байтовете, записани d .

IO е клас Ruby и като клас има право да се смесва в модули. И така става. В

по-специално, IO обектите са изброими.

б

° С

д

|  |
| --- |
| **362 серия** |

**362**

C ГЛАВА 12 ***Файлови и I / O операции***

***12.1.2 IO обекти като изброими***

Както знаете, една изброима трябва да има всеки метод, за да може да итерира. IO

обектите се итерират въз основа на глобалния разделител на входни записи, който, както видяхте в

връзката със низове и техния метод всеки\_ ред в раздел 10.7, се съхранява в

глобална променлива $ /.

В следващите примери изходът на Ruby е обозначен с **получер** шрифт; обикновен тип

показва въвеждане от клавиатурата. Кодът извършва итерация на стандартния STDIN

входящ поток. (Ще разгледаме по-отблизо STDIN и приятели в следващия раздел.)

първо, STDIN третира символа на нов ред като сигнал, че една итерация е завършила; то

по този начин отпечатва всеки ред, докато го въвеждате:

**>>** STDIN.each {| ред | p линия}

Това е ред 1

**"Това е ред 1 \ n"**

Това е ред 2

**"Това е ред 2 \ n"**

Всички разделени с $ /, което е символ на нов ред

**"Всички разделени с $ /, което е символ на нов ред \ n"**

Но ако промените стойността на $ /, идеята на STDIN за това какво представлява итерация също се променя.

Прекратете първата итерация с Ctrl-d (или Ctrl-c, ако е необходимо!) И опитайте този пример:

**>>** $ / = "СЛЕДВАЩ"

**=>"СЛЕДВАЩ"**

**>>** STDIN.each {| ред | p линия}

Първа линия

СЛЕДВАЩИЯ

**"Първи ред \ nСЛЕДВАЩ"**

Следващ ред

където „линия“ наистина означава

докато не видим ... СЛЕДВАЩ

**"\ nСледващ ред \ n където \" ред \ "наистина означава \ nдокато не видим ... СЛЕДВАЩ"**

Тук Ruby приема въвеждане от клавиатурата, докато не удари низа „NEXT“, след което го

счита вписването на записа за пълно.

Така $ / определя усещането на IO обекта за „всеки.“ И тъй като IO обектите са enu-

merable, можете да извършвате обичайните изброими операции върху тях. (Можете да предположите

че $ / е върната към първоначалната си стойност в тези примери.) Нотация ^ D

показва, че машинописката е въвела Ctrl-d в този момент:

**>>** STDIN.изберете {| ред | линия = ~ / \ A [AZ] /}

Ние се интересуваме само от

редове, които започват с

Главни букви

^ D

**=> ["Ние се интересуваме само от \ n", "Големи букви \ n"]**

**>>** STDIN.map {| ред | line.reverse}

сенил esehT

terces a niatnoc

.egassem

^ D

**=> ["\ nТези редове", "\ nсъдържат тайна", "\ nсъобщение."]**

|  |
| --- |
| **363 серия** |

**363**

***Как се събира I / O системата на Ruby***

Ще се върнем към изброеното поведение на IO обектите в контекста на файл han-

dling в раздел 12.2. Междувременно трите основни IO обекта - STDIN, STDOUT и

STDERR - струва си да ги разгледате отблизо.

***12.1.3 STDIN, STDOUT, STDERR***

Ако сте написали програми и / или скриптове на черупки, които използват какъвто и да е тип I / O тръбопроводи, тогава

вероятно сте запознати с концепцията за *стандартния* вход, изход и грешка

потоци. Те са основно по подразбиране: освен ако не е казано друго, Руби приема, че всички входни данни

ще дойде от клавиатурата и всички нормални изходи ще отидат в терминала. *Ако приемем* ,

в този контекст означава, че необузданите, процедурни I / O методи, като путове и

получава, оперирайте съответно на STDOUT и STDIN.

Съобщенията за грешки и STDERR са малко по-ангажирани. Нищо не отива на STDERR

освен ако някой не му каже. Така че, ако искате да използвате STDERR за изход, трябва да посочите име

изрично:

ако е счупен?

STDERR.puts "Има проблем!"

край

В допълнение към трите константи, Ruby ви дава и три глобални променливи: $ stdin,

$ stdout и $ stderr.

T HE СТАНДАРТЕН I / O глобални променливи

Основната разлика между STDIN и $ stdin (и другите двойки също) е, че

не трябва да преназначавате на константата, но можете да преназначите на променливата.

Променливите ви дават начин да модифицирате стандартното поведение на I / O потока без

загуба на оригиналните потоци.

Например, може би искате всички изходи да отидат във файл, включително стандартния изход

и стандартна грешка. Можете да постигнете това с някои задания към глобалната вариация

ables. Запазете този код в outputs.rb:

запис = File.open ("/ tmp / запис", "w")

old\_stdout = $ stdout

$ stdout = запис

$ stderr = $ stdout

поставя "Това е запис"

z = 10/0

Първата стъпка е да отворите файла, към който искате да напишете. (Ако нямате / tmp

директория на вашата система, можете да промените името на файла, така че да сочи към различно

path, стига да имате разрешение за запис.) След това запишете текущия $ stdout в a

променлива, в случай че искате да се върнете към нея по-късно.

Сега идва малкият танц на I / O дръжките. Първо, $ stdout е предефиниран като

запис на изходната дръжка. След това $ stderr се задава еквивалентно на $ stdout. В този момент всеки

plain-old поставя резултатите от оператора в изхода, който се записва във файл / tmp / запис,

тъй като plain поставя извлечения в $ stdout - и там сега е $ stdout

сочещ. $ stderr изход (като съобщението за грешка в резултат на разделяне на нула)

също отива във файла, тъй като $ stderr също е преназначен за този манипулатор на файла.

|  |
| --- |
| **364 серия** |

**364**

C ГЛАВА 12 ***Файлови и I / O операции***

Резултатът е, че когато стартирате програмата, не виждате нищо на екрана си; но

/ tmp / record изглежда така:

Това е рекорд

outputs.rb: 6: в `/ ': разделено на 0 (ZeroDivisionError)

от outputs.rb: 6: в "<main>"

Разбира се, можете също да изпратите стандартен изход към един файл и стандартна грешка към

друг. Глобалните променливи ви позволяват да манипулирате потоците по какъвто и да е начин.

Скоро ще преминем към файлове, но докато говорим за I / O като цяло и

по-специално стандартните потоци, нека разгледаме по-отблизо клавиатурата.

***12.1.4 Малко повече за въвеждането от клавиатурата***

Въвеждането на клавиатурата се извършва в по-голямата си част с get и getc. Както и вие

видяна, получава връща един ред вход. getc връща един символ.

Една разлика между тези два метода е, че в случай на getc, трябва

назовете изрично входящия си поток:

линия = получава

char = STDIN.getc

И в двата случая входът е буфериран: трябва да натиснете Enter, преди да се случи нещо. Това е

възможно да накарате getc да се държи по буфериран начин, така че да приема входа си като

веднага щом героят бъде ударен, но няма преносим начин да се направи това на платформата Ruby

форми. (На UNIX -ish платформи можете да настроите терминала в „суров“ режим със stty

команда. За целта трябва да използвате системния метод, описан в глава 14

отвътре Руби.)

Ако по някаква причина сте настроили $ stdin на нещо различно от клавиатурата,

все още можете да четете въвеждането от клавиатурата, като използвате STDIN изрично като получател на get:

линия = STDIN.gets

Ако приемем, че сте следвали съветите в предишния раздел и сте изпълнили всичките си стандарти

dard I / O поток жонглиране чрез използването на глобалните променливи, а не на con-

стойности, STDIN пак ще бъде входният поток на клавиатурата, дори ако $ stdin не е.

На този етап ще се обърнем към съоръженията на Ruby за четене, писане и

манипулиране на файлове.

***12.2 Основни файлови операции***

Вграденият клас File предоставя възможности за манипулиране на файлове в Ruby. Файлът е

подклас на IO, така че File обектите споделят определени свойства с IO обекти, въпреки че

Файловият клас добавя и променя определено поведение.

Първо ще разгледаме основните файлови операции, включително отваряне, четене, писане и

затваряне на файлове в различни режими. След това ще разгледаме по-“Rubyish” начин за обработка на файлове

четене и писане: с кодови блокове. След това ще влезем по-дълбоко в enu-

издръжливост на файлове и след това завършете раздела с преглед на някои от често срещаните

изключения и съобщения за грешки, които можете да получите в процеса на манипулиране на файлове.

|  |
| --- |
| **Страница 365** |

**365**

***Основни файлови операции***

***12.2.1 Основите на четенето от файлове***

Четенето от файл може да се извършва по един байт наведнъж, определен брой байтове

в даден момент или по един ред в даден момент (където *редът* се определя от $ / разделителя). Можете също

променете позицията на следващата операция за четене във файла, като се придвижите напред или назад

защитава определен брой байтове или чрез преместване на вътрешния указател на обекта File до a

конкретно изместване на байта във файла.

Всички тези операции се извършват с любезността на File обекти. И така, първата стъпка е

за да създадете обект File. Най-простият начин да направите това е с File.new. Предайте име на файл на

този конструктор и, ако приемем, че файлът съществува, ще получите обратно манипулатор на файл, отворен за

четене. Следващите примери включват файл, наречен ticket2.rb, който съдържа кода

в списък 3.2 и това се съхранява в директория, наречена код:

>> f = File.new ("код / ​​билет2.rb")

=> # <Файл: code / ticket2.rb>

(Ако файлът не съществува, ще бъде повдигнато изключение.) На този етап можете да използвате файла

екземпляр за четене от файла. На ваше разположение са редица методи. Абсо-

лютне най-прост е методът за четене; той чете в целия файл като един низ:

>> е. четете

=>"Билет за клас \ n def инициализиране (място, дата) \ n

@venue = място \ n @ дата = дата \ n край \ n \ n *и т.н.*

Въпреки че използването на четене е примамливо в много ситуации и подходящо в някои, може

бъдете неефективни и малко подобни на чук, когато имате нужда от повече гранулираност във вашия

четене и обработка на данни.

Тук ще разгледаме голям избор от методи за четене на файлове на Ruby, които се справят с тях

в групи: първо базирани на редове методи за четене и след това байтови методи за четене.

***12.2.2 Линейно четене на файлове***

Най-лесният начин да прочетете следващия ред от файл е с get:

>> е. получава

=>"Билет за клас \ n"

>> е. получава

=>"def инициализира (място, дата) \ n"

>> е. получава

=>"@venue = място \ n"

Затворете манипулаторите на вашите файлове

Когато приключите с четенето от и / или писането във файл, трябва да го затворите. Файл

обектите имат близък метод (например f.close) за тази цел. Ще научите

за начин за отваряне на файлове, така че Ruby да се справя със затварянето на файла, като обхване

цяла файлова операция с кодов блок. Но ако го правите по старомоден начин, както

в примерите, включващи File.new в тази част на главата, трябва да затворите вашия

файлове изрично. (Те ще се затворят, когато излезете и от irb, но е добра практика да затворите

тези, които сте отворили.)

|  |
| --- |
| **366 серия** |

**366**

C ГЛАВА 12 ***Файлови и I / O операции***

Методът за четене прави голяма част от това, което получава: чете един ред от файла.

Разликата се състои в това как двата метода се държат, когато се опитвате да четете отвъд

край на файл: получава връща нула и readline поражда фатална грешка. Можете да видите разликата-

ence, ако направите четене на обект File, за да стигнете до края на файла и след това опитайте двете

методи за обекта:

>> е. четете

=>"def инициализира (място, дата) \ n @venue = място \ n

@ дата = дата \ n край \ n \ n

*и т.н.*

>> е. получава

=> нула

>> f.readline

EOFError: достигнат е краят на файла

Ако искате да получите целия файл наведнъж като масив от редове, използвайте readlines (близък rel-

ative of read). Обърнете внимание и на операцията за пренавиване, която премества вътрешния обект на файла

позиционен указател обратно към началото на файла:

>> е. превъртане назад

=> 0

>> f.readlines

=> ["Class Ticket \ n", "def инициализира (място, дата) \ n",

"@venue = място \ n", "@date = дата \ n"*и т.н.*

Имайте предвид, че File обектите са изброими. Това означава, че можете да прегледате

редовете един по един, вместо да чете целия файл в паметта. Всеки от тях

метод на File обекти (известен също като синоним всеки\_ ред) служи за тази цел:

>> f.each {| ред | поставя "Следващ ред: # {ред}"}

Следващ ред: билет за клас

Следващ ред: def инициализиране (място, дата)

Следващ ред: @venue = място

и т.н.

ЗАБЕЛЕЖКА В предишния пример и няколко следващи, пренавиване на файла

обект се приема. Ако следите в irb, ще искате да напишете f.rewind

за да се върнете в началото на файла.

Изброяемостта на обектите File заслужава собствена дискусия и ние ще я разгледаме

скоро. Междувременно нека да разгледаме байтовите прости операции за четене.

***12.2.3 Четене на байтови и символни файлове***

Ако цял ред е твърде много, какво ще кажете за един символ? Методът getc чете и

връща един символ от файла:

>> f.getc

=>"c"

Можете също така да „откачите“ символ - т.е. да поставите определен символ обратно във файла -

входен поток, така че това е първият прочетен знак при следващото четене:

>> f.getc

=>"c"

|  |
| --- |
| **367 серия** |

**367**

***Основни файлови операции***

>> f.ungetc ("X")

=> нула

>> е. получава

=>"Билет за Xlass \ n"

Всеки знак е представен от един или повече байта. Как байтовете се преобразуват в символи

зависи от кодирането. Каквото и да е кодирането, можете да премествате и байтове

символно чрез файл, използвайки getbyte. В зависимост от кодирането, номерът

от байта и броят на знаците във вашия файл може да са равни или не, и getc

и getbyte, на дадена позиция във файла, може или не може да върне едно и също нещо.

Точно както четенето се различава от това, че четенето поражда фатална грешка, ако го използвате

в края на файл методите readchar и readbyte се различават от getc и getbyte,

съответно по същия начин. Ако приемем, че вече сте прочели до края на файла

обект f, получавате следните резултати:

>> f.getc

=> нула

>> f.readchar

EOFError: достигнат е краят на файла

>> f.getbyte

=> нула

>> f.readbyte

EOFError: достигнат е краят на файла

По време на всички тези операции обектът File (както всеки IO обект) има усещане къде се намира

е във входния поток. Както видяхте, можете лесно да превъртите този вътрешен указател към

началото на файла. Можете също така да манипулирате показалеца в някои по-фини

зърнени начини.

***12.2.4 Търсене и запитване на позицията на файла***

Обектът File има усещане къде във файла е спрял четенето. Можете и двете

прочетете и променете изрично този вътрешен указател, като използвате позицията на обекта на файла (позиция)

атрибут и / или метод на търсене.

С pos можете да разберете къде във файла сочи указателят в момента:

>> е. превъртане назад

=> 0

>> f.pos

=> 0

>> е. получава

=>"Билет за клас \ n"

>> f.pos

=> 13

Тук позицията е 0 след пренавиване и 13 след четене на един 13-байтов ред. Можеш

присвоява на стойността на позицията, която премества показалеца на определено място във файла:

>> f.pos = 10

=> 10

>> е. получава

=>"et \ n"

|  |
| --- |
| **368 серия** |

**368**

C ГЛАВА 12 ***Файлови и I / O операции***

Върнатият низ е това, което обектът File счита за „линия“ от байт 10: всичко

от тази позиция нататък до следващото появяване на нов ред (или, строго погледнато,

от $ /).

Методът за търсене ви позволява да се придвижвате във файл, като премествате показалеца на позицията към

ново място. Местоположението може да бъде конкретно отместване във файла или да е относително

или към текущата позиция на показалеца, или към края на файла. Вие посочвате какво искате

използвайки специални константи от IO класа:

f.seek (20, IO :: SEEK\_SET)

f.seek (15, IO :: SEEK\_CUR)

f.seek (-10, IO :: SEEK\_END)

В този пример първият ред се стреми към байт 20. Вторият ред премества показалеца

15 байта от текущата си позиция, а последният ред търси 10 байта преди края на

файлът. Използването на IO :: SEEK\_SET не е задължително; обикновен f.seek (20) прави същото (като

прави f.pos = 20).

Разгледахме няколко начина за четене от файлове, започвайки с четенето наведнъж

метод, прогресирайки през подхода по ред и ликвидирайки с най-много

финозърнести четения въз основа на характер и позиция. Всички тези технологии за четене на файлове

niques включват File обекти - т.е. екземпляри от класа File. Този клас също

предлага някои техники за четене.

***12.2.5 Четене на файлове с методи за клас на файл***

Малко по-късно ще видите повече от наличните съоръжения като класови методи на File. За

сега ще разгледаме два метода, които се справят с четенето на файлове на ниво клас: File.read

и File.readlines.

Тези два метода правят едно и също нещо, което им дава едноименният екземпляр

terparts правят; но вместо да създавате екземпляр, вие използвате клас File, методът

име и името на файла:

full\_text = File.read ("myfile.txt")

lines\_of\_text = File.readlines ("myfile.txt")

В първия случай получавате низ, съдържащ цялото съдържание на файла. През сек-

ond case, получавате масив от редове.

Тези два метода от клас съществуват само за удобство. Те се грижат за отварянето

и затваряне на дръжката на файла за вас; не е нужно да правите домакинство на системно ниво-

инж. През повечето време ще искате да направите нещо по-сложно и / или по-ефективно

cient от четенето на цялото съдържание на файл в низ или масив едновременно.

Като се има предвид, че дори методите за екземпляри read и readlines са относително груби

гранулирани инструменти, ако решите да прочетете файл наведнъж, можете също да отидете докрай

и използвайте версиите на метода на класа.

Вече имате добър набор от инструменти за четене на файлове и справяне с резултатите. В този

точка, ще се обърнем към другата страна на уравнението: писане във файлове.

|  |
| --- |
| **369 серия** |

**369**

***Основни файлови операции***

***12.2.6 Записване във файлове***

Записването във файл включва използване на путове, печат или запис на обект на файл, който се отваря в

режим на писане или добавяне. Режимът на запис е посочен с w като втори аргумент на нов. В

този режим, файлът се създава (ако приемем, че имате разрешение да го създадете); ако съществуваше

вече старата версия е заменена. В режим на добавяне (обозначен с а), каквото и да сте

записът във файла се добавя към това, което вече е там. Ако файлът все още не съществува, отворете-

създаването му в режим на добавяне.

Този пример изпълнява някои прости операции за запис и добавяне, като прави пауза

начинът за използване на могъщия File.read за проверка на съдържанието на файла:

>> f = File.new ("data.out", "w")

=> # <Файл: data.out>

>> f.puts "Дейвид А. Блек, рубист"

=> нула

>> е. затвори

=> нула

>> поставя File.read ("data.out")

Дейвид А. Блек, рубист

=> нула

>> f = File.new ("data.out", "a")

=> # <Файл: data.out>

>> f.puts "Yukihiro Matsumoto, създател на Ruby"

=> нула

>> е. затвори

=> нула

Методи за входно / изходни ниски нива

В допълнение към различните I / O и File методи ще разгледаме отблизо тук, IO

class ви дава инструментариум от методи на системно ниво, с които можете да правите I / O на ниско ниво

операции. Те включват sysseek, sysread и syswrite. Тези методи коректуват

спонд към системните повиквания, върху които са изградени някои от методите на по-високо ниво.

Методите sys- изпълняват сурови, недеферирани операции с данни и не трябва да се смесват

с методи на по-високо ниво. Ето пример за това какво да не се прави:

File.open ("output.txt", "w") do | f |

f.print ("Здравей")

f.syswrite ("там!")

край

поставя File.read ("output.txt")

Ако стартирате тази малка програма, ето какво ще видите:

syswrite.rb: 3: предупреждение: syswrite за буфериран IO

там! Здравейте

В допълнение към предупреждение получавате и втория низ (този, написан със syswrite)

заседнал във файла преди първия низ. Това е така, защото syswrite и print не го правят

действайте по същите правила и не играйте добре заедно. Най-добре е да се придържате

с методите от по-високо ниво, освен ако нямате конкретна причина да използвате другите.

|  |
| --- |
| **Стр. 370** |

**370**

C ГЛАВА 12 ***Файлови и I / O операции***

>> поставя File.read ("data.out")

Дейвид А. Блек, рубист

Юкихиро Мацумото, създател на Руби

Връщаната стойност на повикване за поставяне на обект File е същата като връщаната стойност на

всяко обаждане към puts: nil. Същото важи и за печата. Ако използвате запис от по-ниско ниво

метод, който е метод на екземпляр от IO класа (и следователно е достъпен за File

обекти, тъй като File наследява от IO), върнатата стойност е броят на записаните байтове

десет към файла.

Ruby ви позволява да спестите от изрично затваряне на обекти на файл и ви позволява да го направите

поддържайте кода си добре капсулиран - като предоставите начин за извършване на файлови операции

вътре в кодов блок. Ще разгледаме тази елегантна и често срещана техника по-нататък.

***12.2.7 Използване на блокове за обхват на файлови операции***

Използването на File.new за създаване на обект File има недостатъка, че в крайна сметка имате

да затворите файла сами. Ruby предоставя алтернативен начин за отваряне на файлове, който поставя

домакинската задача за затваряне на файла в ръцете на Ruby: File.open с

кодов блок.

Ако извикате File.open с кодов блок, блокът получава обекта File като негов sin-

аргумент gle. Използвате този обект File вътре в блока. Когато блокът приключи,

Файловият обект се затваря автоматично.

Ето пример, в който се отваря файл и се чете ред по ред за обработка.

Първо създайте файл, наречен records.txt, съдържащ по един запис на ред:

Пабло Казалс | каталонски | виолончело | 1876-1973

Jascha Heifetz | руско-американски | цигулка | 1901-1988

Емануел Фойерман | австрийско-американски | виолончело | 1902-1942

Сега напишете кода, който ще чете този файл, ред по ред и докладвайте за това, което намира. То

използва базираната на блокове версия на File.open:

File.open ("records.txt") do | f |

докато запис = f.gets

име, националност, инструмент, дати = record.chomp.split ('|')

поставя "# {name} (# {дати}), който е # {националност},

➥ играе # {инструмент}. "

край

край

Програмата се състои изцяло от извикване на File.open заедно с кодовия му блок. (Ако ти

извикайте File.open без блок, той действа като File.new.) Параметърът на блока f получава

обекта File. Вътре в блока файлът се чете по един ред с помощта на f. Докато

тестът е успешен, стига да идват редове от файла. Когато програмата удари

края на входния файл, получава връща нула и условието while се проваля.

Вътре в цикъла while текущата линия е изрязана, за да се премахне окончателният нов

ред символ, ако има такъв, и разделен на символа на тръбата. Получените стойности се съхраняват в

четирите локални променливи вляво и тези променливи след това се интерполират в a

красив отчет за изход:

|  |
| --- |
| **Страница 371** |

**371**

***Основни файлови операции***

Пабло Казалс (1876-1973), който е каталунец, свири на виолончело.

Jascha Heifetz (1901-1988), който беше руско-американец, свиреше на цигулка.

Емануел Фойерман (1902-1942), който е австроамериканец, свири на виолончело.

Често се използва използването на кодов блок за обхват на операция File.open. Понякога

води обаче до недоразумения. По-специално, не забравяйте, че блокът, който предлага

vides с обекта File не прави нищо друго. Няма неявен цикъл. Ако

искате да прочетете какво има във файла, пак трябва да направите нещо като цикъл while

с помощта на обекта File. Просто е хубаво, че можете да го направите вътре в кодов блок и това

не е нужно да се притеснявате за затваряне на обекта File след това.

И не забравяйте, че File обектите са изброими.

***12.2.8 Изброяване на файлове***

Благодарение на факта, че Enumerable е сред предците на File, можете да го замените

идиома while в предишния пример с всеки:

File.open ("records.txt") do | f |

е. всеки | запис |

име, националност, инструмент, дати = record.chomp.split ('|')

поставя "# {name} (# {дати}), който е # {националност},

➥ играе # {инструмент}. "

край

край

Руби грациозно спира итерацията, когато удари края на файла.

Като изброими, File обектите могат да изпълняват много от същите функции, които масивите,

хешове и други колекции. Разбирането как работи изброяването на файлове изисква

малко по-различен умствен модел: докато масив вече съществува и преминава през него

елементи в хода на итерацията, обектите на файла трябва да управляват четене ред по ред

зад кулисите, когато преглеждате през тях. Но приликата на идиомите -

честото използване на методите от Enumerable - означава, че не е нужно да мислите

много подробности за процеса на четене на файлове, когато преглеждате файл.

Най-важното, не забравяйте, че можете да преглеждате файлове и да ги адресирате като

изброими. Изкушаващо е да прочетете цял файл в масив и след това да го обработите

масив. Но защо просто да не повторите файла и да не изгубите мястото, необходимо за задържане

съдържанието на файла в паметта?

Можете например да прочетете цял файл от записи с обикновен текст и след това да

оформете операция инжектиране върху получения масив, за да получите средната стойност на дадено поле:

# Примерен запис в members.txt:

# Дейвид Блек мъж 55

брой = 0

total\_ages = File.readlines ("members.txt"). inject (0) do | общо, ред |

брой + = 1

полета = линия.сплит

възраст = полета [3] .to\_i

общо + възраст

край

поставя „Средна възраст на групата: # {total\_ages / count}.“

|  |
| --- |
| **372 серия** |

**372**

C ГЛАВА 12 ***Файлови и I / O операции***

Но можете също да изпълните операцията за инжектиране директно върху обекта File:

брой = 0

total\_ages = File.open ("members.txt") do | f |

f.inject (0) do | общо, линия |

брой + = 1

полета = линия.сплит

възраст = полета [3] .to\_i

общо + възраст

край

край

поставя „Средна възраст на групата: # {total\_ages / count}.“

С този подход не се създава междинен масив. Обектът File работи сам.

По един или друг начин, определено ще се сблъскате със случаи, когато нещо се обърка

с вашите файлови операции. Руби ще ви остави без съмнение, че има проблем,

но е полезно да видите предварително какви са някои от възможните проблеми и как

те са докладвани.

***12.2.9 Изключения и грешки във файловите I / O***

Когато нещо се обърка с файловите операции, Ruby създава изключение. Повечето от

грешки, които ще получите по време на работа с файлове, могат да бъдат намерени в пространството от имена Errno:

Errno :: EACCES (разрешението е отказано), Errno :: ENOENT (няма такъв обект - файл или дирек-

тори), Errno: EISDIR (е директория - грешка, която получавате, когато се опитате да отворите директория като

ако беше файл) и други. Винаги ще получавате съобщение заедно с изключението:

>> File.open ("no\_file\_with\_this\_name")

Errno :: ENOENT: Няма такъв файл или директория - no\_file\_with\_this\_name

>> f = File.open ("/ tmp")

=> # <Файл: / tmp>

>> е. получава

Errno :: EISDIR: Това е директория - / tmp

>> File.open ("/ var / root")

Errno :: EACCES: Разрешението е отказано - / var / root

Семейството грешки Errno включва не само грешки, свързани с файлове, но и друга система

грешки. Основната система обикновено картографира грешки в цели числа (например на

Linux, грешката „не директория“ е представена от макроса C ENOTDIR, който е

дефиниран като числото 20). Класът Errno на Ruby обгръща тези съпоставяния с грешка в число

в пакет от класове изключения.

Всеки клас на изключение Errno съдържа знания за цялото число, на което съответства

карти на системни грешки. Можете да получите тези числа чрез константата на Errno на

всеки клас Errno - и ако това звучи неясно, пример ще го направи по-ясен:

>> Errno :: ENOTDIR :: Errno

=> 20

Рядко, ако изобщо някога, ще се наложи да се занимавате с картографирането на изключението на Errno на Ruby

класове към целите числа, на които вашата операционна система съпоставя грешки. Но трябва да бъдете

знаейки, че всяко изключение на Errno е основно системна грешка, проникваща чрез Ruby.

|  |
| --- |
| **Страница 373** |

**373**

***Заявка за IO и File обекти***

Това не са специфични за Ruby грешки, като синтаксисни грешки или липсващи грешки на метода; те включват

нещата се объркват на системно ниво. В тези ситуации Руби е само пратеникът.

Нека се върнем към това, което можете да направите, когато нещата тръгнат правилно. След това ще разгледаме някои

начини, по които можете да поискате от IO и File обекти информация за себе си и

тяхната държава.

***12.3 Заявки за IO и File обекти***

Обектите IO и File могат да бъдат заявявани по множество критерии. Класът IO включва някои

методи за заявки; класът File добавя още.

Един акт и един модул, тясно свързани с File, също влизат в действие: File :: Stat

и FileTest. File :: Stat връща обекти, чиито атрибути съответстват на полетата на

структурата на статията, дефинирана от извикването на библиотеката C stat (2). Някои от тези полета са системни

специфични и без значение за всички платформи. Модулът FileTest предлага множество

методи за получаване на информация за състоянието на файловете.

Класът File също има някои методи за заявка. В някои случаи можете да получите същото

информация за файл по няколко начина:

>> File.size ("code / ticket2.rb")

=> 219

>> FileTest.size ("код / ​​билет2.rb")

=> 219

>> Файл :: Stat.new ("code / ticket2.rb"). Размер

=> 219

По-нататък ще разгледаме голям избор от методи за заявки. В някои случаи те са

налични по повече от един начин.

***12.3.1 Получаване на информация от класа File и модула FileTest***

File и FileTest предлагат множество методи за заявки, които могат да ви дадат много информация

за файл. Това са основните категории на заявката: *Какво е това? Какво може да направи?*

*Колко голям е?*

Наличните методи като класни методи на File и FileTest са почти идентични;

те са предимно псевдоними един на друг. Примерите ще използват само FileTest, но вие можете

използвайте и File.

Ето някои въпроси, които може да искате да зададете за даден файл, заедно с

техники за тяхното питане. Всички тези методи връщат или true, или false, освен

size, което връща цяло число. Имайте предвид, че тези методи за тестване на файлове са

с удоволствие ще вземе директории, връзки, сокети и други подобни на файлове обекти като свои аргументи.

Те не са ограничени до обикновени файлове:

*Съществува ли файл?*

FileTest.exist? ("/ Usr / local / src / ruby ​​/ README")

*Файлът директория ли е? Редовен файл? Символична връзка?*

FileTest.directory?("/home/users/dblack/info ")

FileTest.file? ("/ Home / users / dblack / info")

FileTest.symlink? ("/ Home / users / dblack / info")

|  |
| --- |
| **Страница 374** |

**374**

C ГЛАВА 12 ***Файлови и I / O операции***

Това семейство от методи за заявки включва също blockdev ?, pipe ?, chardev ?, и

гнездо ?.

*Четим ли е файл? За записване? Изпълним?*

FileTest.четлив? ("/ Tmp")

FileTest.writable? ("/ Tmp")

FileTest.executable? ("/ Home / users / dblack / setup")

Това семейство от методи за заявки включва world\_readable? и world\_writable ?,

които тестват за по-разрешителни разрешения. Включва и варианти на основното

три метода с добавен \_real. Те тестват разрешенията на скрипта

действителен идентификатор по време на изпълнение, за разлика от неговия ефективен потребителски идентификатор .

*Какъв е размерът на този файл? Файлът празен ли е (нула байта)?*

FileTest.size ("/ home / users / dblack / setup")

FileTest.zero?("/tmp/tempfile ")

В допълнение към заявката и булевите методи, достъпни чрез FileTest (и File),

можете също така да се консултирате с обекти от класа File :: Stat за информация за файла.

***12.3.2 Извличане на информация за файл с File :: Stat***

File :: Stat обектите имат атрибути, съответстващи на структурата stat в стандарта

C библиотека. Можете да създадете обект File :: Stat по един от двата начина: с новия

метод или със метода stat на съществуващ обект File:

>> Файл :: Stat.new ("code / ticket2.rb")

=> # <Файл :: Stat dev = 0x1000002, ino = 11531534, mode = 0100644,

nlink = 1, uid = 501, gid = 20, rdev = 0x0, размер = 219, blksize = 4096,

блокове = 8, атиме = 2014-03-23 ​​08:31:49 -0400,

mtime = 2014-02-25 06:24:43 -0500, ctime = 2014-02-25 06:24:43 -0500>

>> File.open ("code / ticket2.rb") {| f | f.stat}

Получаване на информация за файла с тест за ядро ​​#

Сред методите от най-високо ниво на ваше разположение (т.е. частни методи на ядрото

модул, който можете да извикате навсякъде без приемник, като путове) е метод, наречен

тест. Използвате тест, като му предадете два аргумента: първият представлява теста и

вторият е файл или директория. Изборът на тест се обозначава със знак. Вие

може да представлява стойността, използвайки нотацията? c, където c е знакът или като едно-

низ от символи.

Ето пример, който установява дали / tmp съществува:

тест? e, "/ tmp"

Други често срещани тестови символи включват? D (тестът е истина, ако вторият аргумент е

директория),? f (вярно, ако вторият аргумент е обикновен файл) и? z (вярно, ако сек-

аргументът ond е файл с нулева дължина). За всеки тест, достъпен чрез Kernel # test,

обикновено има начин да се получи резултатът, като се извика метод на един от класовете dis-

изброени в този раздел. Но тестовата нотация на ядрото # е по-кратка и може да бъде полезна

поради това.

**Един и същ**

**изход**

|  |
| --- |
| **375 серия** |

**375**

***Манипулация на директории с клас Dir***

Екранният изход от метода File :: Stat.new ви показва атрибутите на

обект, включително времето му на създаване (ctime), последната модификация (mtime) и последното

достъп (атиме).

СЪВЕТ Кодовият блок, даден на File.open в този пример, {| f | f.stat}, eval-

uates до последния израз вътре в него. Защото последният (наистина, само)

sion е f.stat, стойността на блока е обект File :: Stat. Като цяло кога

използвате File.open с кодов блок, извикването на File.open връща последното

стойност от блока. Наречен без блок, File.open (като File.new)

връща новосъздадения обект File.

Голяма част от информацията, достъпна от File :: Stat, е изградена от UNIX- подобни показатели,

като номер на inode, режим на достъп (разрешения) и идентификатор на потребител и група . Съобщението

тази информация зависи от вашата операционна система. Няма да влизаме в

подробности тук, защото не е междуплатформена; но каквато и информация да има вашата система

поддържа за файлове е на разположение, ако имате нужда от него.

Манипулирането и заявката на файлове често включва извършване по същия начин на директории.

Ruby осигурява съоръжения за операции с директории в класа Dir. Ще видите и такива

операции в някои от стандартните библиотечни инструменти, които ще обсъдим малко по-късно. Първо, нека

погледнете Dir.

***12.4 Манипулация на директории с клас Dir***

Подобно на File, класът Dir предоставя полезни методи за клас и екземпляр. За да създадете Dir

екземпляр, предавате път на директория на new:

>> d = Dir.new ("/ usr / local / src / ruby ​​/ lib / minitest")

=> # <Dir: / usr / local / src / ruby ​​/ lib / minitest>

Най-често срещаната и полезна техника, свързана с Dir, е итерация през записите

(файлове, връзки, други директории) в директория.

***12.4.1 Четене на записи в директорията***

Можете да стигнете до записите по един от двата начина: с помощта на метода на записите или с помощта на

техниката глобус. Основната разлика е, че *глобирането* на директорията не се връща

скрити записи, които в много операционни системи (включително всички UNIX- подобни системи)

означава записи, чиито имена започват с точка. Globbing също позволява заместване

съвпадение и за рекурсивно съвпадение в поддиректории.

Т МЕТОД HE ЗАПИСИ

Както самият клас Dir, така и екземплярите на класа Dir могат да ви дадат директория

записи. Предвид екземпляра на Dir, създаден по-рано, можете да направите това:

>> d.entries

=> [".", "..", ".document", "autorun.rb", "benchmark.rb", "hell.rb",

"mock.rb", "parallel\_each.rb", "pride.rb", "README.txt", "spec.rb",

"unit.rb"]

**При необходимост коригирайте пътя**

**за вашата система**

|  |
| --- |
| **376 серия** |

**376**

C ГЛАВА 12 ***Файлови и I / O операции***

Или можете да използвате метода на метода на класа:

>> Dir.entries ("/ usr / local / src / ruby ​​/ lib / minitest")

=> [".", "..", ".document", "autorun.rb", "benchmark.rb", "hell.rb",

"mock.rb", "parallel\_each.rb", "pride.rb", "README.txt", "spec.rb",

"unit.rb"]

Имайте предвид, че записите с една и две точки (текуща директория и родителска директория,

съответно), както и скритият .document запис. Ако искате да повторите

чрез записите, само за обработка на файлове, трябва да сте сигурни, че сте филтрирали

имена, започващи с точки.

Да кажем, че искаме да съберем размерите на всички не скрити обикновени файлове в директория.

Ето първа итерация (по-късно ще разработим по-кратка):

d = Dir.new ("/ usr / local / src / ruby ​​/ lib / minitest")

записи = d.вписвания

entries.delete\_if {| влизане | влизане = ~ /^\./}

entries.map! {| запис | File.join (d.path, запис)}

entries.delete\_if {| влизане | ! File.file? (Запис)}

print "Общо байтове:"

поставя entries.inject (0) {| общо, запис | общо + File.size (запис)}

Първо, ние създаваме обект Dir за целевата директория и вземаме записите му. Следва а

последователност от манипулации върху масива от записи. Използване на масива delete\_if

метод, премахваме всичко, което започва с точка. След това правим картографиране на място на

entry масив, така че всеки запис да включва пълния път до файла. Това е постигнато

с два полезни метода: методът на екземпляра Dir # path, който връща оригинала

път на директорията, лежащ в основата на този конкретен екземпляр на Dir (/ usr / local / src / ruby ​​/ lib /

минитест); и File.join, който свързва пътя към името на файла с правилното разделяне

rator (обикновено /, но донякъде зависи от системата).

Сега, когато записите са масажирани, за да представят пълни имена на пътеки, ние го правим

друга операция delete\_if за изтриване на всички записи, които не са обикновени файлове, като

sured от File.file? метод на тестване. Масивът от записи вече съдържа пълни имена на пътища

на всички обикновени файлове в оригиналната директория. Последната стъпка е да съберете техните размери, a

задача, за която инжектирането е напълно подходящо.

Наред с другите начини за съкращаване на този код, можете да използвате глобиране на директория вместо

методът на записите.

D ДИРЕКТОРЕН ГЛОБИНГ

Globbing в Ruby взема своята семантика до голяма степен от globbing на черупката, синтаксиса, който позволява

правите неща като това в черупката:

$ ls \* .rb

$ rm \*.? xt

$ за f в [AZ] \* # и т.н.

Детайлите се различават от една черупка до друга, разбира се; но въпросът е, че цялото това

семейството техники за разширяване на имената е мястото, където Ruby получава своя глобиращ синтаксис. Звезда

isk представлява съвпадение на заместващ знак за произволен брой знаци; въпросник представлява

изпраща един заместващ символ. Налични са класове в стил Regexp за съвпадение.

|  |
| --- |
| **377** |

**377**

***Манипулация на директории с клас Dir***

За да глобите директория, можете да използвате метода Dir.glob или Dir. [] (Квадратна скоба-

ets). Версията на метода с квадратни скоби ви позволява да използвате синтаксис в стил индекс, като

бихте направили с метода квадратна скоба за масив или хеш. Връщате масив

съдържащ набор от резултати:

>> Dir ["/ usr / local / src / ruby ​​/ include / ruby ​​/ r \* .h"]

=> ["/usr/local/src/ruby/include/ruby/re.h", "/ usr / local / src / ruby ​​/ include /

ruby / regex.h "," /usr/local/src/ruby/include/ruby/ruby.h "]

Методът glob е до голяма степен еквивалентен на метода [], но е малко по-гъвкав: вие

може да му даде не само глобален модел, но и един или повече аргументи на флага, които контролират неговия

поведение. Например, ако искате да направите безчувствен глобус, можете да предадете

Файл :: FNM\_CASEFOLD флаг:

Dir.glob ("информация \*") # []

Dir.glob ("информация", файл :: FNM\_CASEFOLD # ["информация", "ИНФОРМАЦИЯ"]

Друг полезен флаг е FNM\_DOTMATCH, който включва скрити точки в резултатите.

Ако искате да използвате два флага, вие ги комбинирате с побитовия ИЛИ оператор, който

се състои от единичен знак. В този пример се откриват постепенно повече файлове

като се добавят по-разрешителните флагове:

>> Dir.glob ("\* информация \*")

=> []

>> Dir.glob ("\* информация \*", файл :: FNM\_DOTMATCH)

=> [".information"]

>> Dir.glob ("\* информация \*", Файл :: FNM\_DOTMATCH | Файл :: FNM\_CASEFOLD)

=> [".information", ".INFO", "Info"]

Знамената са буквално цифри. Стойността на File :: FNM\_DOTMATCH, например, е 4.

Конкретните числа нямат значение (те произтичат в крайна сметка от флаговете в системата

функция на библиотеката fnmatch). Важен е фактът, че те са експоненти на две

отчита използването на операцията ИЛИ за тяхното комбиниране.

ЗАБЕЛЕЖКА Както можете да видите от първите два реда на предишния пример, a

glob операцията в директория не може да намери нищо и пак да не се оплаква. То

ви дава празен масив. Ненамеряването на нещо не се счита за провал

когато глобирате.

Globbing с квадратни скоби е същото като globbing без предоставяне на никакви флагове. В

с други думи, прави това

Dir ["\* информация \*"]

е като да правиш това

Dir.glob ("\* информация \*", 0)

което, тъй като по подразбиране е, че нито един от флаговете не е в сила, е като да направите това:

Dir.glob ("\* информация \*")

|  |
| --- |
| **Страница 378** |

**378**

C ГЛАВА 12 ***Файлови и I / O операции***

Методът с квадратни скоби на Dir ви дава един вид стенография за най-

мон случай. Ако се нуждаете от повече детайлност, използвайте Dir.glob.

По подразбиране глобирането не включва имена на файлове, които започват с точки. Също така, както можете

вижте, globbing връща пълни имена на пътища, а не само имена на файлове. Заедно тези факти ни позволяват

отрежете пример за общ размер на файла:

dir = "/ usr / local / src / ruby ​​/ lib / minitest"

entries = Dir ["# {dir} / \*"]. изберете {| entry | File.file? (Запис)}

print "Общо байтове:"

поставя entries.inject (0) {| общо, запис | общо + File.size (запис)}

С тяхното изключване на файлове с точки и включването им на пълни пътеки, резултатите от глоби често

отговарят по-близо от резултатите на Dir.entries на начините, по които много от нас се справят

с файлове и директории ежедневно.

В управлението на директории има нещо повече, отколкото просто да се види какво има. Ще погледнем

следва някои техники, които ви позволяват да влезете по-дълбоко в процеса.

***12.4.2 Манипулиране на директорията и заявки***

Класът Dir включва няколко метода за получаване на информация за насочване

тори или за текущата директория, както и методи за създаване и премахване

директории. Тези методи са, както много други, най-добре илюстрирани с пример.

Тук ще създадем нова директория (mkdir), ще отидем до нея (chdir), ще добавим и прегледаме

в файл и изтрийте директорията (rmdir):

newdir = "/ tmp / newdir"

newfile = "newfile"

Dir.mkdir (newdir)

Dir.chdir (newdir) направи

File.open (нов файл, "w") do | f |

f.puts "Примерен файл в нова директория"

край

поставя "Текуща директория: # {Dir.pwd}"

поставя "Списък с директории:"

p Dir.entries (".")

File.unlink (нов файл)

край

Dir.rmdir (newdir)

print "Съществува ли все още # {newdir}?"

ако File.exist? (newdir)

поставя "Да"

друго

поставя "Не"

край

След инициализиране на няколко удобни променливи B , ние създаваме новата директория с

mkdir. С Dir.chdir преминаваме към тази директория; също, като се използва блок с chdir

означава, че след излизането на блока се връщаме в предишната директория c . (Използване на chdir

без блок променя текущата директория, докато изрично не бъде променена обратно.)

Като вид стъпка за попълване на директория на символи, ние създаваме един файл с един

линия в него d . След това изследваме текущото име на директория с помощта на Dir.pwd и разглеждаме a

б

° С

д

д

е

ж

з

|  |
| --- |
| **379 серия** |

**379**

***Файлови инструменти от стандартната библиотека***

списък на записите в директорията e . След това прекратяваме връзката (изтриваме) наскоро създаденото

файл f , след което блокът chdir е завършен.

Обратно във всяка директория, в която стартирахме, премахваме примерната директория с помощта

Dir.rmdir (извиква се също като прекратяване на връзката или изтриването) g . И накрая, тестваме за съществуването на

newdir, напълно очаквайки отговор на Не (защото rmdir щеше да доведе до фатална грешка

ако не беше намерил директорията и успешно я премахна) h .

Както беше обещано във въведението към тази глава, сега ще разгледаме някои стандарти

библиотечни съоръжения за манипулиране и обработка на файлове.

***12.5 Файлови инструменти от стандартната библиотека***

Работата с файлове е област, в която предложенията на стандартната библиотека са особено богати.

Съответно тук ще се задълбочим в тези предложения по-задълбочено, отколкото където и да било другаде

книгата. Това не означава, че до останалата част от стандартната библиотека не си струва да стигате

знам, но че разширенията, налични за манипулиране на файлове, са толкова важни за това как

повечето хора правят манипулации с файлове в Ruby, че не можете да получите стабилна основа в

процес без тях.

Първо ще разгледаме гъвкавия пакет FileUtils и след това по-специализирания

но полезен клас Pathname. След това ще срещнете StringIO, клас, чиито обекти са по същество

най-общо, низове с I / O интерфейс; можете да ги превъртите назад, да ги търсите, getc

от тях и т.н. Накрая ще проучим open-uri, пакет, който ви позволява

„Отварят“ URI и ги четат в низове толкова лесно, сякаш са локални файлове.

***12.5.1 Модулът FileUtils***

Модулът FileUtils предоставя някои практични и удобни методи, които го правят

лесно да се манипулират файлове от Ruby по кратък начин по начини, които съответстват на

познати системни команди. Имената на методите ще бъдат особено познати на потребителите

на UNIX и UNIX- подобни операционни системи. Те могат лесно да бъдат научени от тези, които

не ги познавам вече.

Много от методите в FileUtils са именувани в чест на системните команди с

конкретни опции на командния ред. Например FileUtils.rm\_rf емулира rm -rf

команда (принуди безусловно рекурсивно премахване на файл или директория). Можете да създавате

символна връзка от *име* на *файл* към *име* на *връзка* с FileUtils.ln\_s (име на файл, име на връзка),

много по начина на командата ln -s.

Както можете да видите, някои от методите в FileUtils са специфични за операционната система. Ако

вашата система не поддържа символни връзки, тогава ln\_s няма да работи. Но по-голямата част от

методите на модула са преносими. Тук ще разгледаме примери за някои от най-много

полезни.

С OPYING , ПРЕМЕСТВАНЕ , И изтриване на файлове

FileUtils предоставя няколко кратки метода на високо ниво за тези операции. Cp

метод емулира традиционния UNIX метод със същото име. Можете да cp един файл

към друг или няколко файла в директория:

|  |
| --- |
| **380 серия** |

**380**

C ГЛАВА 12 ***Файлови и I / O операции***

>> изискват 'fileutils'

=> вярно

>> FileUtils.cp ("baker.rb", "baker.rb.bak")

=> нула

>> FileUtils.mkdir ("архивиране")

=> ["архивиране"]

>> FileUtils.cp (["осигурете.rb", "супер.рб"], "архивиране")

=> ["осигурете.rb", "супер.рб"]

>> Dir ["backup / \*"]

=> ["backup / sure.rb", "backup / super.rb"]

Този пример също илюстрира метода mkdir B, както и използването на Dir # [] c to

проверете наличието на копираните файлове в новата директория за архивиране.

Точно както можете да копирате файлове, можете също да ги премествате, поотделно или поотделно:

>> FileUtils.mv ("baker.rb.bak", "архивиране")

=> 0

>> Dir ["backup / \*"]

=> ["backup / baker.rb.bak", "backup / sure.rb", "backup / super.rb"]

И можете лесно да премахвате файлове и директории:

>> File.exist? ("Backup / super.rb")

=> вярно

>> FileUtils.rm ("./ backup / super.rb")

=> ["./backup/super.rb"]

>> File.exist? ("Backup / super.rb")

=> невярно

Методът rm\_rf рекурсивно и безусловно премахва директория:

>> FileUtils.rm\_rf ("архивиране")

=> ["архивиране"]

>> File.exist? ("Архивиране")

=> невярно

FileUtils ви предоставя полезен набор от инструменти за бърза и лесна поддръжка на файлове. Но си отива

по-нататък: позволява ви да опитате команди, без да ги изпълнявате.

T Н. D RY R ООН и N O W RITE МОДУЛИ

Ако искате да видите какво би се случило, ако трябва да стартирате определен FileUtils com-

mand, можете да изпратите командата на FileUtils :: DryRun. Резултатът от метода

обаждането е представяне на системна команда в стил UNIX , еквивалентна на това, което бихте направили

get, ако сте извикали същия метод на FileUtils:

>> FileUtils :: DryRun.rm\_rf ("архивиране")

rm -rf архивиране

=> нула

>> FileUtils :: DryRun.ln\_s ("архивиране", "архивиране\_връзка")

ln -s backup backup\_link

=> нула

Ако искате да сте сигурни, че не случайно изтривате, презаписвате или премествате файлове, вие

може да дава вашите команди на FileUtils :: NoWrite, който има същия интерфейс като

FileUtils, но не извършва никакви операции за запис на диск:

б

° С

|  |
| --- |
| **381 серия** |

**381**

***Файлови инструменти от стандартната библиотека***

>> FileUtils :: NoWrite.rm ("backup / super.rb")

=> нула

>> File.exist? ("Backup / super.rb")

=> вярно

Почти със сигурност ще намерите FileUtils полезен в много ситуации. Дори и да не сте

запознат с командите в стил UNIX, на които са наречени много от методите на FileUtils

са базирани, ще ги научите бързо и ще ви спести да се задълбочите

по-ниско ниво на входно-изходни и файлови библиотеки, за да изпълнявате задачите си.

След това ще разгледаме друго, свързано с файлове предложение от стандартната библиотека: path-

разширение на име.

***12.5.2 Класът Pathname***

Класът Pathname ви позволява да създавате обекти на Pathname и да правите заявки и да ги манипулирате

така че можете да определите например основното име и разширението на име на път или

прегледайте пътя, докато се издига структурата на директорията.

Обектите на Path също имат голям брой методи, които се проксират от File,

Dir, IO и други класове. Тук няма да разглеждаме тези методи; ще се придържаме към тези

които са уникално на Pathname.

Първо започнете с обект Pathname:

>> изискват 'pathname'

=> вярно

>> path = Pathname.new ("/ Users / dblack / hacking / test1.rb")

=> # <Име на пътя: /Users/dblack/hacking/test1.rb>

Когато извиквате методи за обект на Pathname, често получавате обратно друго Pathname

обект. Но новият обект винаги има своето представяне на низове, видимо в него

инспектиране на низ. Ако искате да видите низа самостоятелно, можете да използвате to\_s или да направите путове

на името на пътя.

Ето два начина за изследване на основното име на пътя:

>> path.basename

=> # <Име на пътя: test1.rb>

>> поставя path.basename

test1.rb

Можете също така да разгледате директорията, която съдържа файла или директорията, представени от

името на пътя:

>> path.dirname

=> # <Име на пътя: / Users / dblack / hacking>

Ако последният сегмент на пътя има разширение, можете да го получите от

Обект на име на път:

>> path.extname

=>".rb"

Обектът Pathname може също така да разшири структурата на своите файлове и директории, като се отсече

отдясно на всяка итерация, използвайки метода ascend и кодов блок:

|  |
| --- |
| **382 серия** |

**382**

C ГЛАВА 12 ***Файлови и I / O операции***

>> path.ascend do | dir |

?> поставя „Следващо ниво нагоре: # {dir}“

>> край

Ето резултата:

Следващо ниво нагоре: /Users/dblack/hacking/test1.rb

Следващо ниво нагоре: / Users / dblack / hacking

Следващо ниво нагоре: / Users / dblack

Следващо ниво нагоре: / Потребители

Следващо ниво нагоре: /

Основната поведенческа черта на обектите на Pathname е, че те връщат други Pathname

обекти. Това означава, че можете да разширите логиката на вашите операции с имена на пътища без

трябва да конвертирате напред и назад от чисти низове. Като илюстрация, тук е

последен пример отново, но променен, за да се възползва от факта, че това, което предстои

чрез в параметъра на блока dir на всяка итерация не е низ (въпреки че

отпечатва като един), но обект на Pathname:

>> path = Pathname.new ("/ Users / dblack / hacking / test1.rb")

=> # <Име на пътя: /Users/dblack/hacking/test1.rb>

>> path.ascend do | dir |

?> поставя „Възходящо до # {dir.basename}“

>> край

Изходът е

Изкачен на test1.rb

Възкачен на хакерство

Изкачен до dblack

Издигнат до потребители

Изкачен до /

Фактът, че dir винаги е обект на Pathname означава, че е възможно да се извика

basename метод върху него. Вярно е, че винаги можете да извикате File.basename (низ)

всеки низ. Но класът Pathname посочва конкретните знания, които има път

може да се приеме, че капсулира за себе си и го прави достъпен за вас чрез sim-

извиквания на метода ple.

След това ще разгледаме различен и мощен стандартен клас на библиотека: StringIO.

***12.5.3 Класът StringIO***

Класът StringIO ви позволява да третирате низове като IO обекти. Можете да търсите чрез

ги, пренавийте ги и т.н.

Предимството, предоставено от StringIO, е, че можете да пишете методи, които използват IO

object API и тези методи ще могат да обработват низове. Това може да бъде полезно за

тестване, както и в реални ситуации по време на изпълнение.

Да кажем, например, че имате модул, който декомментира файл: той чете

от един файл и записва всичко, което не е коментар, в друг файл. Ето какво

такъв модул може да изглежда така:

|  |
| --- |
| **383 серия** |

**383**

***Файлови инструменти от стандартната библиотека***

модул DeCommenter

def self.decomment (infile, outfile, comment\_re = / \ A \ s \* # /)

infile.each do | inline |

outfile.print вграден, освен ако не е вграден = ~ comment\_re

край

край

край

Методът DeCommenter.decomment очаква две манипулатори на отворен файл: една, която може да чете

от и един, на когото може да пише. Той също така приема регулярен израз, който има стойност по подразбиране.

Регулярният израз определя дали всеки ред във входа е коментар. Всеки

ред, който *не* съответства на регулярния израз, се отпечатва в изходния файл.

Типичен случай на използване на модула DeCommenter ще изглежда така:

File.open ("myprogram.rb") do | inf |

File.open ("myprogram.rb.out", "w") do | outf |

DeCommenter.decomment (inf, outf)

край

край

В този пример изваждаме коментарите от хипотетичния програмен файл

myprogram.rb.

Ами ако искате да напишете тест за модула DeCommenter? Тестване на файл трансформация

мации може да бъде трудно, защото трябва да поддържате входния файл като част от теста

и също така се уверете, че можете да пишете в изходния файл - който след това трябва да прочетете обратно

in. StringIO улеснява, като позволява на целия код да остане на едно място без

трябва да четете или пишете действителни файлове.

За да използвате декомментатора със StringIO, запазете модула в decommenter.rb. Тогава,

създайте втори файл, decomment-demo.rb, в същата директория и със следното

ing съдържание:

изисква 'stringio'

require\_relative 'decommenter'

низ = << EOM

# Това е коментар.

Това не е коментар.

# Това е.

# И това е.

Тестване с помощта на реални файлове

Ако искате да стартирате тестове за въвеждане и извеждане на файлове с помощта на реални файлове, tempfile на Ruby

клас може да ви помогне. Това е функция за стандартна библиотека, така че трябва да изисквате 'temp-

файл '. След това създавате временни файлове с конструктора, като подавате име, което

Ruby се променя в уникално име на файл. Например:

tf = Tempfile.new ("my\_temp\_file").

След това можете да пишете и четете от файла с помощта на обекта File tf.

б

° С

|  |
| --- |
| **384 серия** |

**384**

C ГЛАВА 12 ***Файлови и I / O операции***

Това също не е коментар.

EOM

infile = StringIO.new (низ)

outfile = StringIO.new ("")

DeCommenter.decomment (infile, outfile)

поставя "Тестът е успешен", ако outfile.string == << EOM

Това не е коментар.

Това също не е коментар.

EOM

След зареждане както на стринг библиотеката, така и на кода за декомментиране B , програмата

задава низ на низ от пет реда (създаден с помощта на документ тук), съдържащ комбинация от

редове за коментар и редове, които не коментират c . След това се създават два обекта StringIO:

един, който използва съдържанието на низ като негово съдържание, и един, който е празен d . The

празният представлява изходния файл.

След това идва обаждането до DeCommenter.decomment e . Модулът третира своите два аргумента

като обекти на файл или IO, четене от едното и отпечатване на другото. StringIO

обектите се държат щастливо като IO обекти и филтрирането се извършва между тях.

Когато филтрирането приключи, можете да проверите изрично, за да сте сигурни, че написаното

към изхода „файл“ е това, което очаквахте f . Оригиналното и промененото съдържание са

и двете физически присъстват в един и същ файл, което улеснява да се види какъв е тестът

и също така е по-лесно да го промените.

Друга полезна стандартна функция на библиотеката е библиотеката open-uri.

***12.5.4 Библиотеката на open-uri***

Стандартният библиотечен пакет open-uri ви позволява да извличате информация от мрежата

работи с HTTP и HTTPS протоколи толкова лесно, сякаш четете локални файлове.

Всичко, което правите, е да изисквате библиотеката (изисква 'open-uri') и да използвате Kernel # open

метод с URI като аргумент. Получавате обратно обект StringIO, съдържащ

резултати от вашата заявка:

изискват 'open-uri'

rubypage = отворен ("http://rubycentral.org")

поставя rubypage.gets

Получавате декларацията за типа на документа от началната страница на Ruby Central - не най-много

искрящо четене, но демонстрира лекотата, с която open-uri ви позволява да импортирате

мрежови материали.

***12.6 Обобщение***

В тази глава сте виждали

I / O (keyboa0rd и екран) и файлови операции в Ruby

Файлови обекти като изброими

Обектите STDIN, STDOUT и STDERR

Модулът FileUtils

Модулът Pathname

° С

д

д

е

|  |
| --- |
| **385 серия** |

**385**

***Обобщение***

Класът StringIO

Модулът open-uri

I / O операциите се основават на IO класа, от които File е подклас. Голяма част от това, което IO

и Файловите обекти се състоят от обвити разговори в библиотеката; те са основно API библиотеки, които

седнете на върха на системните I / O съоръжения.

Можете да итерирате през манипулаторите на файлове на Ruby, сякаш са масиви, като използвате всяка карта,

отхвърляне и други методи от модула Enumerable и Ruby ще се погрижи

подробности за обработката на файлове. Ако и когато имате нужда, можете също да се обърнете към IO и

Файлови обекти с команди от по-ниско ниво.

Някои от стандартните библиотечни средства за манипулиране на файлове са незаменими и

разгледахме няколко: модулът FileUtils, който предоставя обогатен набор от инструменти за

файлови и дискови операции; класът StringIO, който ви позволява да адресирате низ като че ли е

на I / O поток; разширението Pathname, което позволява лесни, разширени операции на

низове, представляващи пътища на файловата система; и open-uri, което улеснява „отварянето“

документи в мрежата.

Също така разгледахме входа на клавиатурата и изхода на екрана, които се обработват чрез IO

обекти-специално, стандартната вход, изход, и грешки I / O дръжки. Руби позволява

преразпределяте ги, за да можете да пренасочвате входа и изхода при необходимост.

С тази глава стигнахме до края на част 2 от книгата и по този начин до края на

нашето проучване на вградените функции и класове на Ruby. Ще се обърнем в част 3 към широката и

дълбока материя на динамиката на Ruby, започвайки с поглед към един от най-простите, но най-про-

намерени помещения на Руби: предпоставката, че обектите, дори обектите от същия клас, могат

действайте и реагирайте индивидуално.

|  |
| --- |
| **386 серия** |

|  |
| --- |
| **387 серия** |

*Част 3*

*Динамика на рубина*

*Руби е динамична, като човешката природа.*

—Matz, в RubyConf 2001

T той фраза *динамика Ruby* е почти излишен: всичко за Руби е

динамичен. Променливите не се интересуват с какъв клас обект ги свързвате, което означава

не е нужно (наистина, не можете) да декларирате техния тип предварително. Обектите получават

възможности от класовете, които ги създават, но могат и да се разклонят от

техните класове, като им се добавят индивидуални методи. Класове и модули

могат да бъдат отворени и модифицирани след първоначалните им дефиниции. Нищо задължително

остава същият през жизнения цикъл на работещата програма.

И тези примери са само началото. В тази последна част на книгата ще го направим

погледнете по-дълбоко и широко, отколкото все още имаме на начините, по които Руби позволява

да променяте обстоятелствата за изпълнение на вашата програма във вашата програма.

Първо, в глава 13 ще разгледаме *индивидуализацията* на *обекта* , навлизайки в детайлите на

как Руби прави възможно отделните обекти да живеят собствения си живот и

развиват свои собствени характеристики и поведение извън базовата на класа класа

характеристики, с които са „родени“. По този начин ще се върнем към една от най-ранните книги

теми: добавяне на методи към отделни обекти. Но тук, оборудвани с

ръба от интервенционния материал, ще обнулим много по-отблизо на

основните механизми на индивидуализация на обекта.

Глава 14 разглежда *извикващи се обекти* : обекти, които можете да изпълните. Виждал си мет-

вече, разбира се - но не сте виждали *обекти на* методите , които ще обсъдим

тук, както и анонимни функции под формата на Proc обекти. Струните не са

|  |
| --- |
| **388 серия** |

се извикват, но можете да оцените низ по време на изпълнение като част от кода на Ruby,

и глава 14 ще включва тази (понякога съмнителна) техника. Главата ще

също така ще ви запознаем с Ruby нишки, които ви позволяват да стартирате паралелно сегменти от код.

И накрая, глава 15 разглежда съоръженията, които Ruby предоставя за *размисъл* по време на изпълнение : изпит-

създаване и манипулиране на състоянието на вашата програма и вашите обекти, докато програмата

се изпълнява и обектите съществуват. Ruby ви позволява да поискате информация за вашите обекти

себе си, като например какви методи могат да изпълняват по време на изпълнение; и редица

Предлагат се куки под формата на методи, които можете да пишете, като използвате специални резервирани

имена, за да прихваща събития по време на изпълнение като наследяване на клас и включване на модул. Тук

навлизаме на територията на динамично отражение и вземане на решения, което дава

Ruby е неговото характерно и поразително качество на гъвкавост и мощност.

Глава 13 включва също информация - и съвети - за процеса на вземане

промени в основния език на Ruby във вашите собствени програми. Всъщност цялата тази част на

книгата включва много указатели за най-добри практики (и указатели далеч от някои не толкова

най-добри практики). Това не е изненадващо, като се има предвид вида на земята, която тези глави покриват.

Това е мястото, където вашите програми могат да се различават, за добро или лошо, по отношение на

естеството и качеството на тяхното използване на либералния набор от инструменти за динамично програмиране на Ruby. Това определено

безкрайно си плаща да помислите не само как и защо, но и *дали* някои от тях

тези мощни техники при определени обстоятелства. Използва се разумно и разумно,

Динамичните възможности на Ruby могат да ви отведат до нови и очарователни висоти.

|  |
| --- |
| **389 серия** |

**389**

*Индивидуализация на обекта*

Един от крайъгълните камъни на дизайна на Руби е индивидуализацията на обекта - тоест способността

на отделни обекти да се държат различно от другите обекти от същия клас.

Всеки обект е пълноправен гражданин на света на изпълнение на програмата и може

да живеят живота, от който се нуждае.

Свободата на обектите да се отклоняват от условията на тяхното раждане има a

вид философски пръстен към него. От друга страна, тя има някои важни техники

кал. последици. Забележителен брой функции и характеристики на Ruby

произтичат от или се сближават по индивидуалността на обектите. Голяма част от Руби е

създаден, за да направи възможна индивидуализацията на обекта. В крайна сметка индивидуализацията е такава

по-важно от инженерството: Мац повтаря отново и отново, че

принципът на индивидуалната индивидуалност е това, което има значение и как Руби го прилага

***Тази глава обхваща***

Единични методи и класове

Методи на класа

Методът на удължаване

Отмяна на поведението на Ruby ядрото

Класът BasicObject

|  |
| --- |
| **390. стр** |

**390**

C ГЛАВА 13 ***Индивидуализация на обекта***

втори. И все пак, изпълнението на индивидуализацията на обекти има някои мощни и

полезни компоненти.

Ще разгледаме в тази глава как Руби позволява на обектите да придобиват мет-

и поведения за всеки обект и как частите на Ruby, които правят

възможно поведението на обекта може да се използва с най-голямо предимство. Ще започнем с разглеждане в

детайлизираме *единични методи - методи,* които принадлежат на отделни обекти - в контекста на

*singleton класове* , където се съхраняват дефинициите на singleton-метод. След това ще

cuss class методи, които са в основата сингълтон методи, прикрепени към обекти на класа.

Друга ключова техника при изработването на поведение на обект е методът extend, който

прави нещо подобно на включването на модула, но за един обект наведнъж. Ще разгледаме

как можете да използвате разширяване за индивидуализиране на вашите обекти.

Може би най-важната тема, свързана по някакъв начин с индивидуализацията на обекта, е

промяна на основното поведение на класовете Ruby. Добавяне на метод към клас, който вече

съществува, като Array или String, е форма на индивидуализация на обекта, защото класовете са

обекти. Това е мощна и рискована техника. Но има начини да го направите със сравнение

малко малък риск - и начини за това по обект (по-скоро добавяне на поведение към един низ

отколкото към класа String) - и ще преминем през пейзажа на ядрото по време на изпълнение

промени с оглед на това как техниките за всеки обект могат да ви помогнат да извлечете максимума от

Понякога изненадващо отвореният модел на Руби.

И накрая, ще подновим едно по-ранно запознанство: класът BasicObject. BasicObject

примери предоставят около най-чистата лаборатория, която може да се представи за създаване на отделни обекти,

и ще разгледаме как тази индивидуализация на клас и обект се допълват взаимно.

***13.1 Къде са сингълтон методите: Единичният клас***

Повечето от това, което се случва в Ruby, включва класове и модули, съдържащи дефиниции на

екземплярни методи

клас С

def talk

поставя "Здравей!"

край

край

и впоследствие инстанцирането на класове и извикването на тези методи на екземпляра:

c = C. нов

в. разговор

Но както видяхте по-рано (дори по-рано, отколкото видяхте методите на инстанции в класовете), вие

може също така да дефинира единични методи директно върху отделни обекти:

obj = Object.new

def obj.talk

поставя "Здравей!"

край

obj.talk

**Изход: Здравейте!**

**Изход: Здравейте!**

|  |
| --- |
| **391 серия** |

**391**

***Къде са сингълтън методите: Единичният клас***

И също така сте видели, че най-често срещаният тип единичен метод е класът

метод - метод, добавен към обект от клас поотделно:

клас Автомобил

def self.makes

% w {Хонда Форд Тойота Шевролет Волво}

край

край

Но всеки обект може да има добавени единични методи. (Почти всеки обект; вижте страничен

Възможността да се дефинира поведението на базата на обект е един от отличителните белези на

Дизайнът на Руби.

Методите на екземпляра - тези, които са достъпни за всеки и всички екземпляри от даден клас - живеят вътре

клас или модул, където те могат да бъдат намерени от обектите, които са в състояние да ги извикат.

Но какво да кажем за единичните методи? Къде живее метод, ако този метод съществува

само да бъде извикан от един обект?

***13.1.1 Двойно определяне чрез единични класове***

Руби, вярна на характера, има прост отговор на този труден въпрос: единичен обект

ton методи живеят в *единичния клас* на обекта . Всеки обект в крайна сметка има два класа:

Класът, чийто екземпляр е

Неговият сингъл клас

Обектът може да извиква методи на екземпляр от първоначалния си клас и може да извиква meth-

os от неговия сингълтон клас. Има и двете. Възможностите на метода за извикване на обекта

сума, всички заедно, до сумата на всички методи на екземпляра, дефинирани в тези два

класове, заедно с методи, достъпни чрез родови класове (суперкласът на

клас на обект, суперклас на този клас и т.н.) или чрез всякакви модули, които имат

е бил смесен или добавен към някой от тези класове. Можете да мислите за греха на обекта

клас gleton като ексклузивно скривалище от методи, специално пригодени за този обект, а не

споделено с други обекти - дори с други екземпляри от класа на обекта.

Някои обекти са по-индивидуализирани от други

Почти всеки обект в Ruby може да има добавени методи към него. Изключенията са

екземпляри на някои числови подкласове, включително целочислени класове и плувки, и

символи. Ако опитате това

def 10.some\_method; край

ще получите грешка в синтаксиса. Ако опитате това

клас << 10; край

ще получите грешка в типа и съобщение с надпис „Не може да се дефинира сингълтон“. Същото е

вярно и в двата случая на числа и символи с плаваща запетая.

|  |
| --- |
| **392 серия** |

**392**

C ГЛАВА 13 ***Индивидуализация на обекта***

***13.1.2 Изследване и модифициране на сингълтон клас директно***

Единичните класове са анонимни: въпреки че са обекти на класа (екземпляри на класа

Клас), те изникват автоматично, без да им се дава име. Въпреки това, вие

може да отвори тялото с дефиниция на клас на единичен клас и да добави методи на екземпляр, клас

методи и константи към него, както бихте направили с обикновен клас.

Правите това със специална форма на ключовата дума class. Обикновено следва константа

тази ключова дума:

клас С

# метод и константни дефиниции тук

край

Но за да влезете в тялото на дефиницията на единичен клас, използвате специална нотация:

клас <<*обект*

# метод и константни дефиниции тук

край

Нотацията на *обекта*<< означава анонимен, единичен клас на обект. Когато си

вътре в тялото на сингълтон клас-дефиниция можете да дефинирате методи - и тези meth-

ods ще бъдат единични методи на обекта, в чийто единичен клас се намирате.

Помислете за тази програма, например:

str = "Аз съм низ"

клас << ул

def два пъти

self + "" + self

край

край

поставя str.tice

Изходът е

Аз съм струна Аз съм струна

Методът два пъти е единичен метод на низа str. Точно като че ли сме имали

направи това:

def str.twice

self + "" + self

край

Разликата е, че ние отворихме сингълтонния клас на str и дефинирахме

метод там.

Разликата между def obj.meth и клас << obj; def meth

Често възниква този въпрос: Има ли някаква разлика между директното дефиниране на метод

върху обект (използвайки нотация def obj.some\_method) и добавяне на метод към

единичен клас на обект изрично (чрез извършване на клас << obj; def some\_method)? The

отговорът е, че има една разлика: константите се решават по различен начин.

|  |
| --- |
| **393 серия** |

**393**

***Къде са сингълтън методите: Единичният клас***

Класирането на обекта <<<< има известна известност като загадъчно или объркващо. То

не трябва да бъде нито. Помислете за това по този начин: това е ключовата дума за класа и тя е готова да приеме

или константа, или израз на обект <<. Новото тук е концепцията за греха-

клас на глетон. Когато се чувствате добре с идеята, че обектите имат единичен

класове, има смисъл да можете да отваряте тези класове с ключовата дума class.

Нотацията на обекта << е начинът, по който се изразява концепцията “единичен клас обект”

когато класът го изисква.

Безспорно най-честото използване на нотация на клас << обект за въвеждане на сингълтон-

class class е във връзка с дефинициите на class-method.

D ОПРЕДЕЛЯНЕ НА МЕТОДИ ЗА КЛАС С КЛАС <<

Ето един идиом, който ще виждате често:

клас Билет

клас << себе си

def most\_expensive (\* билети)

tickets.max\_by (&: цена)

край

край

край

***(продължение)***

Ако имате константа от най-високо ниво N, можете също да дефинирате N вътре в единичната единица на обекта

тон клас:

N = 1

obj = Object.new

клас << obj

N = 2

край

Като се има предвид тази последователност от инструкции, двата начина за добавяне на единичен метод към

obj се различават, при което N се вижда от дефиницията на метода:

def obj.a\_method

поставя N

край

клас << obj

def another\_method

поставя N

край

край

obj.a\_method

obj.another\_method

Сравнително необичайно е тази разлика във видимостта на константите да повлияе на вас

код; в повечето случаи можете да разгледате двете означения за сингълтън метод

определение като взаимозаменяема. Но си струва да знаете за разликата, защото

това може да има значение в някои ситуации и може също да обясни неочаквани резултати.

**Изход: 1**

**(външно ниво N)**

**Изход: 2 (N принадлежащи**

**към единичния клас на obj)**

|  |
| --- |
| **394** |

**394**

C ГЛАВА 13 ***Индивидуализация на обекта***

Този код води до създаването на метода на класа Ticket.most\_expensive - много

същия метод като този, дефиниран в раздел 3.6.3, но по това време направихме това:

def Ticket.most\_expensive (\* билети) # и т.н.

В текущата версия използваме идиома на обекта клас <<, отваряйки единичната

тон клас на обекта; и в този конкретен случай участващият обект е класът

object Ticket, което е стойността на self в точката в кода, където клас <<

се извиква себе си. Резултатът от дефинирането на метода най-скъпо в класа-

Блокът за дефиниция е, че той се дефинира като единичен метод в Ticket - което е

да речем, метод на клас.

Същият метод на клас също може да бъде дефиниран по този начин (ако приемем, че този код идва

в точка от програмата, където класът на Ticket вече съществува):

клас << Билет

def most\_expensive (билети)

# и т.н.

Тъй като self е Ticket вътре в тялото на дефиницията на Ticket class, class << self *inside*

тялото е същото като клас << Билет *извън* тялото. (Технически бихте могли дори

do class << Билет в тялото на класа Ticket, но на практика обикновено ще видите

class << self, когато обектът, чийто единичен клас се нуждае от отваряне, е self.)

Фактът, че клас << self се появява често във връзка със създаването на

методите на класа понякога водят до фалшивото впечатление, че класът << обект nota-

не може да се използва само за създаване на методи на клас или това е единственият израз, който можете

законно поставен отдясно е себе си. Всъщност class << self вътре в блок с дефиниция на клас е

само един конкретен случай на употреба за обект клас <<. Техниката е обща: поставя ви

в дефиниционен блок за единичния клас обект, какъвто и да е обектът.

В глава 4 разгледахме стъпките, които обектът предприема, докато търси метод

сред дефинираните в неговия клас, класа на неговия клас и т.н. Сега имаме нов артикул

на радара: сингълтон клас. Какъв е ефектът от този допълнителен клас върху метода-

процес на търсене?

***13.1.3 Единични класове по пътя за търсене на метод***

Спомнете си, че търсенето на методи се изкачва по веригата на наследяване на класа, с отклонения за всеки

модули, които са били смесени или добавени. Когато за първи път обсъдихме този процес,

не бяхме говорили за класове и методи за сингъл и те не присъстваха в

диаграма. Сега можем да преразгледаме диаграмата, за да ги обхване, както е показано на фигура 13.1.

Кутията, съдържаща обект клас << представлява сингълтон клас на обект. В

търсенето на метода x, обектът търси първо всички модули, добавени към неговия единичен

тон клас; след това изглежда в самия клас singleton. След това изглежда във всички модули, които

класът сингълтън е включил. (В диаграмата има един: модул N.) След това,

търсенето продължава до първоначалния клас на обекта (клас D) и т.н.

Обърнете внимание по-специално, че е възможно един клас да добавя или включва a

модул. Все пак това е клас.

|  |
| --- |
| **395 серия** |

**395**

***Къде са сингълтън методите: Единичният клас***

I питомен модул ПРИ SINGLETON КЛАС

Нека да изградим малка програма, която илюстрира ефекта от включването на модул в едно-

тон клас. Ще започнем с прост клас Person и няколко екземпляра от този клас:

клас Личност

attr\_accessor: име

край

david = Person.new

david.name = "Дейвид"

matz ​​= Person.new

matz.name = "Matz"

ruby = Person.new

ruby.name = "Рубин"

Сега да кажем, че някои лица - т.е. някои обекти на Личност - не обичат да разкриват

техните имена. Логичен начин да добавите този вид секретност към отделни обекти е да добавите a

единична версия на метода на името за всеки от тези обекти:

клас Обект

(

)

*вграден*

включва ядрото

модул Ядро

(

)

*вграден*

клас D <C

добавете M

включват N

край

клас BasicObject

(

)

*вграден*

клас С

край

клас << обект

добавете X

включва Y

край

обект = D.нови

**обект.x**

модул М

край

модул N

край

модул Y

край

модул X

край

Фигура 13.1 Ред за търсене на метод, преработен, за да включва единични класове

|  |
| --- |
| **396 серия** |

**396**

C ГЛАВА 13 ***Индивидуализация на обекта***

def david.name

"[Не е наличен]"

край

В този момент Мац и Руби разкриват имената си, но Дейвид е потаен. Когато ние

направете поименно повикване

поставя „Имаме един човек на име # {matz.name},“

поставя "един с име # {david.name},"

поставя "и един с име # {ruby.name}."

получаваме само две имена:

Имаме един човек на име Мац,

едно име [не е налично],

и един на име Руби.

Дотук добре. Но какво, ако повече от един човек реши да бъде потаен? Би било

неприятност да се написва def person.name ... за всеки такъв човек.

Начинът около това е да се използва модул. Ето как изглежда модулът:

модул Секретно

def име

"[Не е наличен]"

край

край

Сега нека направим Руби потайна. Вместо да използвате def за дефиниране на нова версия на

метода на името, ще включим модула в единичния клас на Ruby:

клас << рубин

включват Secretive

край

Поименното обаждане сега показва, че Руби е отишла в тайния лагер; стартиране на

предишните оператори за пускане отново дават следния изход:

Имаме един човек на име Мац,

едно име [не е налично],

и един с име [не е наличен].

Какво се случи в случая с Руби? Изпратихме съобщението “name” на обекта ruby. The

обект, зададен да намери метода. Първо изглеждаше в собствения си сингъл клас, където го имаше

не намери метод за име. След това той погледна в модулите, смесени в неговия сингълтон

клас. Класът сингълтон на рубин, смесен в модула Secretive, и, разбира се,

този модул съдържа екземпляр метод, наречен име. В този момент методът

се изпълнява.

Дадено разбиране за реда, в който обектите търсят своите пътеки за търсене

за методи можете да разберете коя версия на метод (т.е. кой клас или мод-

ule версията на метода) обектът ще намери първо. Примерите също помагат, особено за

илюстрират разликата между включването на модул в единичен клас и в регистър

клас ular.

|  |
| --- |
| **397 серия** |

**397**

***Къде са сингълтън методите: Единичният клас***

S ИНГЛЕТОН МОДУЛ ВКЛЮЧВАНЕ ПРОТИВ . ОРИГИНАЛ - ВКЛЮЧВАНЕ НА КЛАСОВ МОДУЛ

Когато смесвате модул в единичен клас на обект, имате работа с това

възразите конкретно; методите, които той научава от модула, имат предимство пред всички

методи със същото име в първоначалния си клас. Следният списък показва

механиката и резултатът от извършването на този вид включва операция.

клас С

def talk

поставя "Здравей от оригиналния клас!"

край

край

модул М

def talk

поставя "Здравей от модул!"

край

край

c = C. нов

в. разговор

клас << c

включват М

край

в. разговор

Резултатът от този списък е както следва:

Здравейте от оригиналния клас!

Здравейте от модула!

Първото повикване за разговор B изпълнява метода на екземпляр на разговор, дефиниран в класа на c, C.

След това смесваме модула M, който също така дефинира метод, наречен talk, в c's single-

тон клас c . В резултат на това следващия път, когато извикаме talk на c d , токът, който се изпълнява

cuted (този, който c вижда първо) е този, дефиниран в M.

Всичко е въпрос на това как класовете и модулите в търсенето на метода на обекта

пътеката са подредени. Модулите, включени в сингълтон класа, се срещат преди

оригинален клас и преди всички модули, включени в оригиналния клас.

Можете да видите това графично, като използвате метода предци, който ви дава списък

на класовете и модулите в йерархията за наследяване и включване на всеки клас или

модул. Започвайки от след дефинициите на класа и модула в предишния пример,

опитайте да използвате предци, за да видите как изглежда йерархията:

c = C. нов

клас << c

включват М

п предци

край

Получавате масив от предци - по същество, пътят за търсене на метод за екземпляри на

този клас. Тъй като това е сингълтон клас на c, разглеждането на предците му означава

Листинг 13.1 Включване на модул в единичен клас

б

° С

д

|  |
| --- |
| **398 серия** |

**398**

C ГЛАВА 13 ***Индивидуализация на обекта***

търсене на пътя за търсене на метода за c. Имайте предвид, че сингълтон класът c е на първо място

списъкът на предците:

[# <Клас: # <C: 0x007fbc8b9129f0 >>, M, C, Object, Kernel, BasicObject]

Сега вижте какво се случва, когато не само смесите M в сингълтон клас c, но и

смесете го в класа на c (C). Вземете след предишния пример,

клас С

включват М

край

клас << c

п предци

край

Този път виждате следния резултат:

[# <Клас: # <C: 0x007fbc8b9129f0 >>, M, C, M, Object, Kernel, BasicObject]

Модулът М се появява два пъти! Два различни класа - единичен клас на c и

клас С - смесили са го. Всеки микс е отделна транзакция. Това е частният бизнес-

за всеки клас; класовете не се консултират помежду си. (Можете дори да смесите M

в Object и ще го получите три пъти в списъка с предци.)

Препоръчваме ви да вземете тези примери, да ги промените, да ги обърнете по този начин и

и изследвайте резултатите. Класовете също са обекти - така че вижте какво се случва, когато вие

вземете сингълтон класа на единичен клас на обект. Ами смесването на модули в

други модули? Опитайте някои примери и с prepend. Възможни са много пермутации

кървене; можете да научите много неща чрез експерименти, използвайки това, което сме разгледали тук като a

начална точка.

Основният урок е, че поведението на всеки обект в Ruby се основава на същия принцип

ples като редовно поведено от клас поведение на обекта: дефиницията на методите на екземпляра в

класове и модули, смесването на модули с класове и следването на a

път за търсене на метод, състоящ се от класове и модули. Ако овладеете тези понятия

и се върнете към тях, когато нещо изглежда размито, вашето разбиране ще се мащабира

нагоре успешно.

***13.1.4 Методът singleton\_class***

За да се обърнете директно към сингълтон класа на обект, използвайте метода singleton\_class.

Този метод може да ви спести някакъв клас << обект обиколки.

Ето как бихте използвали този метод, за да получите предците на единичния клас на обект:

низ = "низ"

p string.singleton\_class.ancestors

Сега да се върнем назад и да разгледаме специален случай в света на сингълтон методите (специални,

защото е често и полезно): методите на класа.

|  |
| --- |
| **399 серия** |

**399**

***Къде са сингълтън методите: Единичният клас***

***13.1.5 Класови методи в (дори повече) дълбочина***

Методите на класа са единични методи, дефинирани за обекти от клас Class. По много начини,

те се държат като всеки друг сингъл метод:

клас С

край

def C.a\_class\_method

поставя "Singleton метод, дефиниран на C"

край

C.a\_class\_method

Но методите на класа показват и специално поведение. Обикновено, когато дефинирате едно-

ton метод за обект, никой друг обект не може да служи като приемник при извикване на това

метод. (Това е, което прави единичните методи единични, или за обект.) Клас

методите са малко по-различни: метод, дефиниран като единичен метод на клас

object може да бъде извикан и в подкласове от този клас. Предвид предишния пример,

с C можете да направите това:

клас D <C

край

D.a\_class\_method

Ето доста объркващия изход (объркващ, защото обектът на класа, който изпратихме

съобщението до е D, а не C):

Единичен метод, дефиниран на C

Имате право да извиквате единични методи на C в подклас на C в допълнение към C, защото

на специална настройка, включваща единичните класове обекти на класа. В нашия пример,

клас на единичен C (където методът a\_class\_method живее) се счита за супер-

клас на сингълтон клас на D.

Когато изпращате съобщение до обекта на класа D, се следва обичайният път за търсене—

с изключение на това, че след единичния клас на D се търси суперкласът на единичния клас на D.

Това е единичният клас на суперкласа на D. И там е методът.

Фигура 13.2 показва връзките между класовете във връзка за наследяване

и техните сингълтон класове.

**Изход: Singleton**

**метод, дефиниран на C**

клас С

Единичен клас на C

Единичен клас на

Суперклас на

Единичен клас на D

клас D

Фигура 13.2 Връзки между класовете

в наследствени отношения и техните

единични класове

|  |
| --- |
| **Страница 400** |

**400**

C ГЛАВА 13 ***Индивидуализация на обекта***

Както можете да видите от фигура 13.2, единичният клас на детето C, D, се счита за дете

(подклас) от сингълтон клас на C.

Единичните класове на обекти на класове понякога се наричат *мета-класове* . Понякога ще

чуйте термина *мета-клас,* приложен към единични класове като цяло, въпреки че няма нищо

особено мета за тях и сингъл клас е по-описателен общ термин.

Можете да третирате това обяснение като бонус тема. Едва ли има спешна нужда

разберете, че ще възниква често. И все пак това е чудесен пример за това как се основава дизайнът на Ruby

на относително малък брой правила (като всеки обект с единичен клас,

и начина, по който се търсят методите). Класовете са обекти със специална обвивка; след всичко,

те са фабрични обекти, както и обекти сами по себе си. Но в Руби има малко

това не произтича естествено от основните принципи на езиковия дизайн - дори

специалните случаи.

Тъй като класовете и модулите на Ruby са обекти, промените, които правите в тези класове

и модулите са промени за всеки обект. По този начин дискусия за това как, кога и дали трябва

да направи промени в основните класове и модули на Ruby има място в тази дискусия на

индивидуализация на обекта. След това ще проучим основните промени.

КЛАСОВЕ СИНГЛЕТ И МОДЕЛ НА ЕДИНИЦА Думата „сингълтон“ има a

второ, различно значение в Ruby (и другаде): то се отнася до синглона

шаблон, който описва клас, който има само един екземпляр. Стандартът Ruby

библиотеката включва изпълнение на единичен модел (достъпен чрез

команда изискват 'единичен'). Имайте предвид, че единичните класове не са

пряко свързана с модела на единични; думата „сингълтън“ е малко прекалена

зареден. Като цяло от контекста става ясно кое значение е предназначено.

***13.2 Модифициране на основните класове и модули на Ruby***

Откритостта на класовете и модулите на Ruby - фактът, че вие, програмистът, можете

влезте под капака на езика и променете това, което прави - е един от най-много

важни характеристики на Ruby, а също и една от най-трудните за примиряване. Това е като

да можете да ядете ястията заедно с храната в ресторант. Откъде знаеш

където единият завършва, а другият започва? Как да разберете кога да спрете? Можете ли да ядете

покривката също?

Да се ​​научиш как да се справяш с откритостта на Руби е малко свързано с техниката на програмиране

и много за най-добрите практики. Не е трудно да се правят модификации на основната мрежа

манометър; трудната част е да знаеш кога трябва, кога не трябва и как да вървиш

за това безопасно.

В този раздел ще разгледаме пейзажа на основните промени: как, какво и какво

защо (и защо не). Ще разгледаме значителните клопки, възможните

предимства и начини да мислите за обектите и тяхното поведение, които ви позволяват да имате

най-доброто от двата свята: гъвкавост и безопасност.

Ще започнем с няколко приказки.

|  |
| --- |
| **401 серия** |

**401**

***Модифициране на основните класове и модули на Ruby***

***13.2.1 Рисковете от промяна на основната функционалност***

Проблемът с извършването на промени в основните класове на Ruby е, че тези промени са

глобално: докато програмата ви работи, промените, които сте направили, ще бъдат в сила.

Ако промените начина на работа на метода и този метод се използва някъде другаде (вътре

Ruby или в библиотека, която зареждате), сте дестабилизирали целия интерпретатор от

промяна на правилата на играта в средата на потока.

Все пак е изкушаващо да персонализирате Ruby по ваш вкус, като промените ядрото

методи в световен мащаб. Все пак можете. Но това е най-малко безопасно и най-малко препоръчително

подход за персонализиране на поведението на основен обект. Ние го гледаме само за да можете да го получите

усещане за същността на проблема.

Един често цитиран кандидат за ad hoc промяна е класът Regexp.

С висящи R EGEXP # MATCH ( а защо не и ДА )

Както ще си спомните от глава 11, когато операция на съвпадение, използваща метода на съвпадение

не успее, получавате обратно нула; когато успее, получавате обратно обект MatchData. Този резултат

е дразнещо, защото не можете да правите същите неща с нула, които можете с мач-

Обект на данни.

Този код, например, успява, ако първото улавяне е създадено от съвпадението:

some\_regexp.match (some\_string) [1]

Но ако няма съвпадение, получавате обратно nil - и тъй като nil няма метод [], вие

получите фатално изключение NoMethodError, когато опитате операцията [1]:

string = "Тестов низ"

re = / A (примерен) низ /

подниз = re.match (низ) [1]

Може да се изкуши да направите нещо подобно, за да избегнете грешката:

клас Regexp

псевдоним \_\_old\_match\_\_ съвпадение

def match (низ)

\_\_old\_match \_\_ (низ) || []

край

край

Този код първите създава псевдоним на мач, с любезното съдействие на псевдоним дума за B . Тогава

код предефинира съвпадение. Новият мач се свързва с оригиналната версия на мача

(чрез псевдонима) и след това връща или резултата от извикването на оригиналната версия, или

(ако това обаждане връща нула) празен масив.

ЗАБЕЛЕЖКА Един *псевдоним* е синоним на името на метод. Извикване на метод чрез псевдоним

не включва промяна в поведението или промяна на метода

процес на търсене. Изборът на псевдоним в предишния пример се основава на

доста конвенционална формула: добавянето на думата *стар* плюс водещата

и последващо подчертаване. (Може да се направи случай, че формулата е твърде съобразена

ventional и че трябва да създавате имена, за които е по-малко вероятно да бъдат избрани

други заместители, които също познават конвенцията!)

**NoMethodError: недефинирано**

**метод [] за нула: NilClass**

б

|  |
| --- |
| **402 серия** |

**402**

C ГЛАВА 13 ***Индивидуализация на обекта***

Вече можете да направите това:

/abc/.match("X")[1]

Въпреки че мачът се проваля, програмата няма да се взриви, защото неуспешният мач

сега връща празен масив, а не нула. Най-лошото, което можете да направите с новия мач

е опитайте да индексирате празен масив, което е законно. (Резултатът от операцията с индекс ще бъде

nil, но поне не се опитвате да индексирате nil.)

Проблемът е, че човекът, който използва вашия код, може да зависи от операцията за съвпадение

за връщане на нула при неуспех:

ако regexp.match (низ)

направи нещо

друго

направи нещо друго

край

Тъй като масивът (дори празен) е истина, докато nil е false, връщайки масив

за неуспешна операция на съвпадение означава, че тестът true / false (както е въведен в if /

else statement) винаги връща true.

Може би промяната на Regexp # match, за да не върнете нула при неуспех, е нещо ваше

инстинктите ще ви кажат да не правите така или иначе. И никой не се застъпва за това; това е повече от това

някои нови потребители на Ruby не свързват точките и следователно не виждат промяната на

основният метод на едно място го променя навсякъде.

Друг често срещан пример и този, който е малко по-фин (и по отношение на това какво

прави и защо не е добра идея), включва String # gsub! метод.

T се върне стойността на S Tring # GSUB ! И ЗАЩО ТРЯБВА ДА ОСТАНЕ ТАКА

Както си спомняте, String # gsub! прави глобална операция за замяна на приемника си, спестявайки

промените в оригиналния обект:

>> string = "Здравейте!"

=>"Здравейте!"

>> string.gsub! (/ e /, "E")

=>"Здравейте!"

>> низ

=>"Здравейте!"

Както можете да видите, връщаната стойност на обаждането до gsub! е низовият обект с

промени, направени B . (И проверката на обекта отново чрез променливата низ потвърждава

че промените наистина са постоянни c .)

Интересното е обаче, че когато gsub се случва нещо различно! операция

не води до промени в низа:

>> string = "Здравейте!"

=>"Здравейте!"

>> string.gsub! (/ zzz /, "xxx")

=> нула

>> низ

=>"Здравейте!"

б

° С

|  |
| --- |
| **403 серия** |

**403**

***Модифициране на основните класове и модули на Ruby***

Няма съвпадение на / zzz /, така че низът не се променя - и връщаната стойност на

обадете се на gsub! е нула.

Подобно на nil return от операция за съвпадение, nil return от gsub! има

потенциал да накарате нещата да се взривят, когато предпочитате да не го правят. По-конкретно, това означава

не можете да използвате gsub! надеждно във верига от методи:

>> string = "Здравейте!"

=>"Здравейте!"

>> string.gsub! (/ e /, "E").

=>"! ErEht ollEH"

>> string = "Здравейте!"

=>"Здравейте!"

>> string.gsub! (/ zzz /, "xxx").

NoMethodError: недефиниран метод `обратен! ' за нула: NilClass

Този пример прави нещо подобно (но не съвсем същото) два пъти. Първият път

през, окованите призовавания към gsub! и обратно! B върнете новия gsub! D и

обърнат низ c . Но вторият път веригата от обаждания води до фатална грешка d :

gsub! call не променя низа, така че връща nil - което означава, че сме извикали

обратен! на нула, а не на низ.

Един от възможните начини за справяне с неудобството от необходимостта да се заобиколи нулата

връщане от gsub! е да се приеме, че обикновено не е подходящо да се веригира метод

и без това се обажда твърде много. И винаги можете да избегнете проблеми, свързани с веригата, ако

не верига:

>> string = "Здравейте!"

=>"Здравейте!"

>> string.gsub! (/ zzz /, "xxx")

=> нула

>> string.reverse!

=>"! ereht olleH"

Методът на чешмата

Методът с докосване (извикващ се за всеки обект) изпълнява малко странното, но потенциално

полезна задача за изпълнение на кодов блок, привеждане на приемника към блока и връщане

приемника. По-лесно е да се покаже това, отколкото да се опише:

>>"Здравей" .tap {| низ | поставя string.upcase} .reverse

ЗДРАВЕЙТЕ

=>"olleH"

Извикан на приемника "Здравейте", методът на докосване връща този низ обратно към неговия код

блок, както е потвърдено от отпечатването на главна версия на низа. Тогава

tap връща целия низ - така че се извършва обратната операция върху низа. Ако

викате gsub! на низ вътре в кран, няма значение дали връща нула,

защото докосването връща низа. Бъдете внимателни обаче. Използване на кран за заобикаляне на нула

връщане на gsub! (или други подобни поведение на взрив методи) може да въведе сложни

само по себе си, особено ако правите множество вериги, където се изпълняват някои методи

операции на място и други връщат копия на обекти.

б

° С

д

|  |
| --- |
| **404 серия** |

**404**

C ГЛАВА 13 ***Индивидуализация на обекта***

Все пак редица потребители на Ruby са били ухапани от връщаната стойност nil, било защото

те очакваха gsub! да се държи като gsub (версията без взрив, която винаги се връща

приемника му, независимо дали е имало промяна или не) или защото не са предвидили

случай, когато низът не би се променил. Така че gsub! и неговата нулева стойност на връщане се превърна в изскачащ

ular кандидат за промяна.

Промяната може да се извърши по следния начин:

клас String

псевдоним \_\_old\_gsub\_bang\_\_ gsub!

def gsub! (\* аргументи и блокиране)

\_\_old\_gsub\_bang \_\_ (\* аргументи, & блокиране)

себе си

край

край

Първо оригиналния gsub! получава псевдоним; което ще ни позволи да извикаме оригиналната версия

от вътрешната страна на новата версия. Новият gsub! взема произволен брой аргументи (

самите аргументи нямат значение; ще ги предадем на стария gsub!) и a

кодов блок, който ще бъде уловен в променливия блок. Ако не е предоставен блок - и

gsub! може да се извика със или без блок - блокът е нула.

Сега наричаме старата версия на gsub !, предавайки й аргументите и използвайки повторно

кодов блок. И накрая, новият gsub! прави нещото, за което е писано: връща се

self (низът), независимо дали извикването на \_\_old\_gsub\_bang\_\_ връща

низ или нула.

И сега, причините да не се прави това.

Промяна на gsub! този начин вероятно е по-малко вероятно, като въпрос на статистика, да ви вкара

проблем, отколкото промяната на Regexp # съвпадение е. Все пак е възможно някой да пише

код, който зависи от документираното поведение на gsub !, по-специално от return-

nil, когато низът не се променя. Ето пример - и въпреки че е

trived (както повечето примери за този сценарий трябва да бъдат), това е валидно Ruby и

в зависимост от документираното поведение на gsub !:

>> посочва = {"NY" =>"Ню Йорк", "NJ" =>"Ню Джърси",

➥ "ME" =>"Мейн"}

=> {"NY" =>"Ню Йорк", "Ню Джърси" =>"Ню Джърси", "МЕН" =>"Мейн"}

>> string = "Източните щати включват NY, NJ и ME."

=>"Източните щати включват NY, NJ и ME."

>> ако string.gsub! (/ \ b ([AZ] {2}) \ b /) {състояния [$ 1]}

>> поставя "Заместване настъпи"

>> друго

?> поставя "String непроменен"

>> край

Настъпи заместване

Започваме с хашиш на държавните съкращения и пълните имена B . След това идва низ, който

използва държавни съкращения c . Целта е съкращенията да бъдат заменени с пълните

имена, използвайки gsub! операция, която улавя две последователни главни букви

заобиколен от граници на думи (\ b) и ги заменя със стойността от хеша

съответстващ на двубуквения подниз d . По пътя си отбелязваме дали

б

° С

д

д

|  |
| --- |
| **405 серия** |

**405**

***Модифициране на основните класове и модули на Ruby***

всякакви такива замествания се правят. Ако има такива, gsub! връща новата версия на низ.

Ако не се правят замени, gsub! връща нула. Резултатът от процеса се отпечатва

в края e .

Щетите тук са относително леки, но урокът е ясен: не променяйте документа

поведение на основните Ruby методи. Ето още една версия на hash на state

пример, използвайки sub! а не gsub !. В тази версия неуспехът да върне нула, когато

низ не променя задейства безкраен цикъл. Ако приемем, че имаме хеш на държавите и

оригиналната версия на низ, можем да направим еднократно заместване, където всеки под-

отчитане на състоянието:

>> string = "Източните щати включват NY, NJ и ME."

=>"Източните щати включват NY, NJ и ME."

>> докато string.sub! (/ \ b ([AZ] {2}) \ b /) {състояния [$ 1]}

>> поставя „Замяна на # {$ 1} с # {състояния [$ 1]} ...“

>> край

Замяна на Ню Йорк с Ню Йорк ...

Замяна на Ню Джърси с Ню Джърси ...

Замяна на мен с Мейн ...

Ако string.sub! винаги връща ненулева стойност (низ), след това условието while

никога няма да се провали и цикълът ще се изпълни завинаги.

Това, което *не* трябва *да* правите, е да пренапишете основните методи, така че те да не правят какво

други очакват от тях. Няма изключение от това. Това е нещо, което трябва

никога не го прави, въпреки че можеш.

Това ни оставя въпроса как безопасно да променим основната функционалност на Ruby.

Ще разгледаме четири техники, които можете да разгледате. Първите три са адитивни

промяна, закачане или преминаване през промяна и промяна по обект. Само един от тях е

наистина безопасни, въпреки че и трите са достатъчно безопасни за използване при много обстоятелства. The

четвъртата техника са *усъвършенстванията* , които са модулни промени в класовете и кои

може да ви помогне да определите основните промени в Ruby, за да не преливат в

код за закръгляване и в самия Ruby.

По пътя ще разгледаме примери по поръчка, както и някои примери от

библиотеката Active Support, която обикновено се използва като част от уеб приложението Rails

рамка за развитие. Активната поддръжка предоставя добри примери за първите два вида

на основната промяна: добавка и преминаване. Ще започнем с добавка.

***13.2.2 Промени в добавките***

Най-често срещаната категория промени във вградените класове Ruby е *промяната* на *добавките* :

добавяне на метод, който не съществува. Ползата от промяната на добавката е, че тя не го прави

clobber съществуващи методи на Ruby. Опасността, присъща на него, е, че ако двама програмисти

напишете добавени методи с едно и също име и двамата се включват в интерпретатора

по време на изпълнението на определена библиотека или програма, един от двамата ще изтрие

други. Няма начин да намалим този риск до нула.

Добавените методи често служат за осигуряване на функционалност, която е голяма

брой хора искат. С други думи, не всички са написани за специализирана употреба в

|  |
| --- |
| **406 серия** |

**406**

C ГЛАВА 13 ***Индивидуализация на обекта***

една програма. Има цифри за безопасност: ако хората са обсъждали дадено

метод от години и ако де факто изпълнението на метода се носи

света на Руби, шансовете са големи, че ако напишете метода или използвате съществуващ

изпълнение, няма да се сблъскате с нищо, което някой друг може да е написал.

Библиотеката за активна поддръжка и по-специално нейната подбиблиотека с разширение на ядрото добавя много

на методите за ядро ​​на Ruby класове. Добавките към класа String осигуряват някои добри резултати

примери. Активната поддръжка се доставя с набор от „флексии“ на String, с методи

като плурализиране и заглавие. Ето няколко примера (ще трябва да стартирате gem

инсталирайте activesupport, за да ги стартирате, ако вече не сте инсталирали скъпоценния камък):

>> изисква 'active\_support / core\_ext'

=> вярно

>>"човек" .pluralize

=>"хора"

>>"little\_dorritt" .titleize

=>"Малкият Дорит"

Всеки път, когато добавяте нови методи към основните класове на Ruby, рискувате някой

else ще добави метод със същото име, който се държи по различен начин. A

библиотека като Active Support зависи от добросъвестността на потребителите и от собствената й репутация

тация: ако използвате активна поддръжка, вероятно знаете, че влизате

един вид неписан договор, който да не замени методите му или да зареди други библиотеки, които го правят

така. В този смисъл Активната поддръжка е защитена от собствената си репутация и широта

използване. Със сигурност можете да използвате Активна поддръжка, ако тя ви дава нещо, което искате или имате нужда,

но не го приемайте като сигнал, че обикновено е добре да добавяте методи към основните класове. Вие

трябва да бъдете доста предпазливи по отношение на това.

Друг начин за добавяне на функционалност към съществуващите класове и модули на Ruby е с pas-

sive закачане или преминаваща техника.

***13.2.3 Промяна на преминаването***

А *проходна* промяна метод включва първостепенни съществуващ метод по такъв начин,

че в крайна сметка първоначалната версия на метода се извиква заедно с новата версия

сион. Новата версия прави всичко, което трябва, и след това предава аргументите си

заедно с оригиналната версия на метода. Той разчита на оригиналния метод за

вижте възвръщаема стойност. (Както знаете от съвпадението и gsub! Заместете примери, call-

оригиналната версия на метод не е достатъчна, ако ще промените основния

интерфейс на метода чрез промяна на връщаната му стойност.)

Можете да използвате заместващи преминавания за редица цели, включително регистриране

и отстраняване на грешки:

клас String

псевдоним \_\_old\_reverse\_\_ обратен

деф обратен

$ stderr.puts "Обръщане на низ!"

\_\_old\_reverse\_\_

край

край

поставя "Дейвид" .обръщане

|  |
| --- |
| **407 серия** |

**407**

***Модифициране на основните класове и модули на Ruby***

Резултатът от този фрагмент е както следва:

Обръщане на низ!

divaD

Първият ред се отпечатва на STDERR, а вторият ред се отпечатва на STDOUT. Изпитът-

ple зависи от създаването на псевдоним за оригиналния реверс и след това извикване на този псевдоним на

края на новия реверс.

Възможно е да се напишат методи, които съчетават добавката и преминаващата философия

фиес. Някои примери от Active Support демонстрират как да направите това.

А DDITIVE / PASS - ЧРЕЗ ХИБРИДИ

Един *добавка / преминаване през хибрид* е метод, който има същото име като съществуващ ядро

метод, извиква старата версия на метода (така че това не е заместител out-out-out),

и добавя нещо към интерфейса на метода. С други думи, това е замяна

предлага супер набор от функционалността на оригиналния метод.

Активната поддръжка разполага с редица добавъчни / преминаващи хибридни методи. A

добър пример е методът to\_s от класа Time. Непроменено, Time # to\_s предоставя a

хубав човекоразчитаем низ, представящ времето

>> Time.now.to\_s

=>"2013-12-31 08:37:32 -0500"

Активната поддръжка добавя към метода, така че да може да вземе аргумент, указващ спе-

вид на форматиране. Например (ако приемем, че сте изисквали активна\_поддръжка)

можете да форматирате обект Time по начин, подходящ за вмъкване на база данни по следния начин:

>> Time.now.to\_s (: db)

=>"2013-12-31 08:37:40"

Псевдоним и неговите псевдоними

В допълнение към ключовата дума alias, Ruby има метод, наречен alias\_method, който е

метод на частен екземпляр на модул. Резултатът е, че можете да създадете псевдоним за

метод или по този начин

клас String

псевдоним \_\_old\_reverse\_\_ обратен

край

или по този начин:

клас String

alias\_method: \_\_ old\_reverse\_\_,: reverse

край

Тъй като това е метод, а не ключова дума, alias\_method се нуждае от обекти, а не

голи имена на методи като негови аргументи. Може да приема символи или низове. Обърнете внимание и на това

аргументите за псевдоним нямат запетая помежду им. Ключовите думи трябва да свършат

подобни неща, но методите не.

|  |
| --- |
| **408 серия** |

**408**

C ГЛАВА 13 ***Индивидуализация на обекта***

Ако искате датата да бъде представена като число, попитайте за: числовия формат:

>> Time.now.to\_s (: номер)

=>"20131231083748"

Аргументът: rfc822 свързва време, форматирано в стил RFC822 , стандартната дата за-

изтривалка за дати в заглавките на имейли. Подобен е на метода Time # rfc822:

>> Time.now.to\_s (: rfc822)

=>"Вторник, 31 декември 2013 г. 08:38:00 -0500"

Различните формати, добавени към Time # to\_s, работят с помощта на strftime, което обвива sys-

извикване със същото име и ви позволява да форматирате пъти по голям брой начини. Така

в модифицирания Time # to\_s няма нищо, което да не можете да направите сами. По избор

за ваше удобство е добавен аргумент (и разбира се удобен за базата данни: db for-

mat представлява интерес главно, ако използвате активна поддръжка заедно с обект-

релационна библиотека, като ActiveRecord). Резултатът е супер набор от Време # до\_с. Вие

може да игнорира добавките и методът ще работи както винаги.

Както при чистото добавяне на метод (като String # pluralize), видът superset-

управляваното заместване на основните методи, представени от тези примери, води до известен риск:

по-конкретно, рискът от сблъсък. Възможно ли е в крайна сметка да заредите две библиотеки, които

и двете добавят незадължителен аргумент: db към Time # to\_s? Не, това е малко вероятно - но е възможно.

Още веднъж, библиотека като Активна поддръжка е защитена от своя висок профил: ако я заредите,

вероятно сте запознати с това, което прави и ще знаете, че няма да замени замените.

И все пак, отдалечено е възможно друга библиотека, която зареждате, да се сблъска с Active-

Поддържа. Както винаги е трудно или невъзможно да се намали рискът от сблъсък до нула.

Трябва да се предпазите, като се запознаете с това, което всяка библиотека прави и

като тествате кода си достатъчно.

Последният основен подход за отмяна на основното поведение на Ruby, който ще разгледаме - и

най-безопасният начин да го направите - е добавянето на функционалност строго за всеки обект, използвайки

Обект # удължаване.

***13.2.4 Промени за всеки обект с удължаване***

Object # extension е вид завръщане у дома по отношение на потока от теми. Скитахме се

външните граници на модифициране на основните класове - и разширяване ни връща към централното

процес в основата на всички такива промени: промяна на поведението на индивида

обект. Също така ни връща към по-ранна тема от тази глава: смесването на a

модул в единичен клас на обект. Това е по същество това, което разширява.

А DDING към обект " S функционалност с EXTEND

Погледнете отново раздел 13.1.3 и по-специално примера на Личност, където сме

смеси секретния модул в единичните класове на някои обекти Person. Като

напомняне, техниката беше следната (където рубинът е личност):

клас << рубин

включват Secretive

край

|  |
| --- |
| **409 серия** |

**409**

***Модифициране на основните класове и модули на Ruby***

Ето как би изглеждал примерът Person, използвайки разширяване вместо изрично отворен

създаване на сингълтонния клас на рубиновия обект. Нека използваме и разширение за Дейвид (вместо

на дефиницията на сингълтън метод с def):

модул Секретно

def име

"[Не е наличен]"

край

край

клас Личност

attr\_accessor: име

край

david = Person.new

david.name = "Дейвид"

matz ​​= Person.new

matz.name = "Matz"

ruby = Person.new

ruby.name = "Рубин"

david.extend (Тайно)

ruby.extend (Потайно)

поставя "Имаме един човек на име # {matz.name}," +

"един с име # {david.name},"

+

"и един с име # {ruby.name}."

По-голямата част от тази програма е същата като първата версия, както и изходът. Ключовите разлики-

ence е използването на разширение B , което води до добавяне на секретния модул към

пътеките за търсене на отделните обекти Дейвид и Рубин, като ги смесва в техните

съответни сингълтон класове. Този процес на включване се случва, когато удължите клас

обект също.

А DDING КЛАС методи с EXTEND

Ако напишете единичен метод върху обект на клас, ето така

клас Автомобил

def self.makes

% w {Хонда Форд Тойота Шевролет Волво}

край

край

или така

клас Автомобил

клас << себе си

def прави

% w {Хонда Форд Тойота Шевролет Волво}

край

край

край

или с някой от другите налични варианти на нотация, добавяте екземпляр

метод към единичния клас на обекта на класа. От това следва, че можете да постигнете това през

в допълнение към другите начини, като се използва разширяване:

б

|  |
| --- |
| **Страница 410** |

**410**

C ГЛАВА 13 ***Индивидуализация на обекта***

създатели на модули

def прави

% w {Хонда Форд Тойота Шевролет Волво}

край

край

клас Автомобил

удължете производителите

край

Ако е по-подходящо в дадена ситуация, можете да разширите обекта на класа след него

вече съществува:

Car.extend (производители)

Така или иначе резултатът е, че обектът на класа Car вече има достъп до метода make.

Както при некласовите обекти, разширяването на обект от клас с модул означава смесване на

модул в единичен клас на класа. Можете да проверите това с метода на предците:

p Car.singleton\_class.ancestors

Резултатът от този фрагмент е

[# <Class: Car>, Makers, # <Class: Object>, # <Class: BasicObject>, Class, Module,

Обект, ядро, BasicObject]

Странно изглеждащите записи в списъка са единични класове. Класът сингъл на Car

самата е включена; така са и единичният клас на Object (който е суперкласът на

сингълтон клас на Car) и сингълтон клас BasicObject (който е суперкласът

от единичния клас на Object). Основната цел на нашата цел е, че Makers е

включени в списъка.

Не забравяйте също, че подкласовете имат достъп до методите на класа на своя суперклас. Ако

подклас Car и погледнете предците на сингълтон класа на новия клас, ще видите

Създатели в списъка.

Първоначалната ни цел в разглеждането на разширението беше да проучим начин да добавим към Руби

основна функционалност. Нека се обърнем сега към тази цел.

М ОДИФИЦИРАНЕ НА ОСНОВНОТО ПОВЕДЕНИЕ С РАЗШИРЯВАНЕ

Вероятно сте събрали парчетата до този момент. Модулите ви позволяват да дефинирате само-

съдържащи се многократно използвани колекции от методи. Kernel # extension ви позволява да дадете индивидуално

обекти за достъп до модули, с любезното съдействие на сингълтон класа и смесения механизъм

низма. Съберете всичко и имате компактен, безопасен начин за добавяне на функционалност

към основните обекти.

Нека да разгледаме още веднъж String # gsub! загадка - а именно, че се връща

нула, когато низът не се променя. Чрез дефиниране на модул и използване на удължаване е възможно

ble, за да промените поведението на gsub! по ограничен начин, като правите само промените, от които се нуждаете

и не повече. Ето как:

модул GsubBangModifier

def gsub! (\* аргументи и блокиране)

супер || себе си

край

б

|  |
| --- |
| **411 серия** |

**411**

***Модифициране на основните класове и модули на Ruby***

край

str = "Здравейте!"

str.extend (GsubBangModifier)

str.gsub! (/ zzz /, "abc").

поставя ул

В модула GsubBangModifier дефинираме gsub !. Вместо технологията за псевдоним и повикване

nique, ние наричаме супер, връщайки или стойността, върната от това обаждане, или самостоятелно - lat-

трето, ако обаждането към супер връща нула B . (Ще си спомните, че супер задейства изпълнението на

следващата версия на текущия метод нагоре по пътя за търсене на метод. Дръж го

мисъл.…)

След това създаваме низ от str и го разширяваме с GsubBangModifier c . Обаждане

str.gsub! d изпълнява gsub! в GsubBangModifier, защото str се среща

GsubBangModifier в пътя си за търсене на метод, преди да срещне класа String -

който, разбира се, съдържа и gsub! определение. Призивът към супер вътре в GsubBang-

Модификатор # gsub! скача нагоре по пътя и изпълнява оригиналния метод, String # gsub !,

предавайки му оригиналните аргументи и кодовия блок, ако има такива. (Това е ефектът от обаждането

супер без аргументи и без празен списък с аргументи.) И резултатът от извикването на

super е или самият низ, или нула, в зависимост от това дали са направени някакви промени

към низа.

По този начин можете да промените поведението на основните обекти - низове, масиви, хешове и т.н.

напред - без да отварят отново своите класове и без да въвеждат промени в глобален

ниво. Наличието на обаждания за удължаване в кода ви помага да покажете какво се случва. Промяна на a

метод като gsub! вътре в самия клас String има недостатък не само да бъде

глобален, но също така и за вероятност да бъде скрит някъде в библиотечен файл, правейки

бъгове, които трудно могат да бъдат открити за хора, които са ухапани от глобалната промяна.

Има още една важна част от пъзела за това как да промените основния обект

поведения: нова функция, наречена *усъвършенстване* .

***13.2.5 Използване на усъвършенствания, за да повлияе на поведението на ядрото***

Подобрения бяха добавени към Ruby 2.0, но се смятаха за „експериментални“ до

2.1 освобождаване. Идеята на усъвършенстването е да се направи временна промяна с ограничен обхват

клас (който, макар и да не е необходимо, може да бъде основен клас).

Ето пример, в който методът на shout се въвежда в клас String, но

само на ограничена основа:

модул Shout

прецизиране на String do

def вик

self.upcase + "!!!"

край

край

край

клас Личност

attr\_accessor: име

използвайки Shout

° С

д

**Изход:**

**! ereht olleH**

б

° С

|  |
| --- |
| **412 серия** |

**412**

C ГЛАВА 13 ***Индивидуализация на обекта***

def обяви

поставя „Обявяване # {name.shout}“

край

край

david = Person.new

david.name = "Дейвид"

david.съобщавам

Тук се появяват два различни метода и те работят ръка за ръка: прецизирайте B и

използвайки c . Методът за прецизиране взема име на клас и кодов блок. Вътре в кода

блок, вие дефинирате поведението, което искате да приеме класът, който прецизирате. В нашата

Например, ние прецизираме класа String, добавяйки метод на shout, който връща

подобрена версия на низа, последвана от три удивителни знака.

Методът на използване превключва ключа: след като „използвате” модула, в който сте

дефинира желаното от вас усъвършенстване, целевият клас възприема новото поведение. В изпита-

ple, използваме модула Shout в класа Person. Това означава, че за дура-

ция на този клас (от оператора за използване до края на дефиницията на класа), низове

ще бъдат „усъвършенствани“, така че да имат метода на викане.

Ефектът от „използването“ на усъвършенстване приключва с края на класа (или

дефиниция, в която декларирате, че използвате усъвършенстването. Можеш

всъщност използвайте, използвайки извън определение на клас или модул, в този случай ефектът от

усъвършенстването продължава до края на файла, в който се появява призивът за използване.

Уточненията могат да ви помогнат да направите временни промени в основните класове в относително

безопасен начин. Други програмни файлове и библиотеки, които вашата програма използва по време на изпълнение, няма да бъдат

засегнати от вашите усъвършенствания.

Ще завършим тази глава с поглед върху малко странна тема: BasicObject

клас. BasicObject не е изключително тема за индивидуализация на обекти (както знаете от

след като прочетох уводния материал за това в глава 3). Но това се отнася до

родословие на всички обекти - включително тези, чието поведение се разклонява от тяхното

оригинални класове - и могат да изиграят важна роля за динамиката, която Руби

прави възможно.

***13.3 BasicObject като предшественик и клас***

BasicObject се намира в горната част на дървото на класа на Ruby. За всеки обект Ruby *obj* , следното

истина е:

*obj* .class.ancestors.last == BasicObject

С други думи, най-високият предшественик на всеки клас е BasicObject. (Освен ако не смесите

модул в BasicObject - но това е пренамерен сценарий.)

Както ще си спомните от глава 3, екземплярите на BasicObject имат няколко метода - само a

комплект за оцеляване, така да се каже, за да могат да участват в дейности, свързани с обекти. Ще го намерите

трудно е да се получи екземпляр BasicObject, който да ви каже какво може да направи:

>> BasicObject.new.methods.sort

NoMethodError: недефинирани `методи 'на метода за # <BasicObject: 0x007fafa308b0d8>

**Резултат: Обявяване**

**ДЕЙВИД !!!**

|  |
| --- |
| **413 серия** |

**413**

***BasicObject като предшественик и клас***

Но BasicObject е клас и се държи като такъв. Можете да получите информация директно от

it, използвайки познати методи на ниво клас:

>> BasicObject.instance\_methods (false) .sort

=> [:!,:! =,: ==,: \_\_ id\_\_,: \_\_ send\_\_,: равно ?,: instance\_eval,

: instance\_exec]

Какъв е смисълът от BasicObject?

***13.3.1 Използване на BasicObject***

BasicObject ви позволява да създавате обекти, които не правят нищо, което означава, че можете

научете ги да правят всичко - без да се притеснявате от сблъсък със съществуващите методи.

Обикновено това предполага интензивно използване на method\_missing. Чрез дефиниране на method\_missing за

BasicObject или клас, който пишете, който наследява от него, можете да проектирате обекти

за чието поведение сте изцяло натоварени и които са малко или никак не предубедени

усещане за това как трябва да се държат.

Най-известният пример за използване на обект с почти никакви методи е

Библиотека на строителя от Джим Уейрих. Builder е инструмент за писане на XML , който извежда XML тагове

съответстващи на съобщенията, които изпращате до обект, който разпознава малко съобщения. The

магията се случва благодарение на method\_missing.

Ето един прост пример за използване на Builder (и цялото използване на Builder е просто; това е

точка на библиотеката). Този пример предполага, че сте инсталирали скъпоценния камък на builder.

изисква 'строител'

xml = Builder :: XmlMarkup.new (: target => STDOUT,: indent => 2)

xml.instruct!

xml.friends правят

xml.friend (: source =>"колеж") правя

xml.name ("Джо Смит")

xml.address do

xml.street ("123 Main Street")

xml.city („Навсякъде, САЩ 00000“)

край

край

край

XML е Builder :: XmlMarkup обект B . Обектът е програмиран да изпраща изхода си

до -STDOUT и отстъп с две интервали. Инструкцията! команда c казва на XML

builder да започне с XML декларация. Всички методи на Builder :: Xml-

Маркирането завършва с гръм и трясък (!). Те нямат аналози, които не са взривове - което гръмва

в повечето случаи трябва да има методи, но в този случай взривът служи за разграничаване

тези методи от методи с подобни имена, които може да искате да използвате за генериране

XML тагове чрез method\_missing. Предполага се, че може да искате XML елемент

наречен инструктирайте, но няма да имате нужда от такъв, наречен инструктирайте !. Така взривът обслужва

специфична за домейна цел и има смисъл да се отклоним от обичайната конвенция на Ruby -

ция за неговото използване.

Резултатът от нашия скрипт на Builder е следният:

б

° С

|  |
| --- |
| **414 серия** |

**414**

C ГЛАВА 13 ***Индивидуализация на обекта***

<? xml version = "1.0" encoding = "UTF-8"?>

<приятели>

<приятел източник = "колеж">

<name> Джо Смит </name>

<адрес>

<street> 123 Main Street </street>

<city> Навсякъде, САЩ 00000 </city>

</address>

</friend>

</friends>

Различните XML тагове вземат имената си от извикванията на метода. Всеки липсващ метод

води до етикет, а кодовите блокове представляват XML влагане. Ако предоставите аргумент за низ

на липсващ метод, низът ще се използва като текстов контекст на елемента.

Атрибутите се предоставят в хеш аргументи.

Builder използва BasicObject, за да върши работата си. Интересното е, че Builder е съществувал и преди

BasicObject го направи. Оригиналните версии на Builder използваха направен по поръчка клас, наречен

BlankSlate, който вероятно е послужил като вдъхновение за BasicObject.

Как бихте внедрили прост клас, базиран на BasicObject?

***13.3.2 Прилагане на подклас на BasicObject***

*Simple* , в току-що зададения въпрос, означава по-просто от Builder :: XmlMarkup (който

прави писането на XML просто, но само по себе си е доста сложно). Нека напишем малка библиотека, която

работи на подобен принцип и извежда списък с елементи с отстъп. Ще избягваме хав-

ing, за да осигурите затварящи тагове, което улеснява нещата много по-лесно.

Класът Lister, показан в следващия списък, ще наследи от BasicObject. То

ще дефинира method\_missing по такъв начин, че всеки липсващ метод да се приема като a

заглавие за списъка, който генерира. Вложените кодови блокове ще управляват отстъп.

клас Lister <BasicObject

attr\_reader: списък

def инициализира

@list = ""

@ ниво = 0

край

деф отстъп (низ)

"" \* @ level + string.to\_s

край

def method\_missing (m, & block)

@list << отстъп (m) + ":"

@list <<"\ n"

@ ниво + = 2

@list << отстъп (добив (самостоятелно)) ако блок

@ ниво - = 2

@list <<"\ n"

връщане ""

край

край

Листинг 13.2 Клас **Lister** : Генерира **отстъпени** списъци от подклас **BasicObject**

б

° С

д

д

е

ж

з

|  |
| --- |
| **415 серия** |

**415**

***BasicObject като предшественик и клас***

При инициализация се задават две променливи на екземпляра B : @list ще служи като точен низ

mulator за целия списък, а @level ще насочва отстъп. Метод на отстъп c

взема низ (или всичко, което може да бъде преобразувано в низ; той извиква to\_s на аргумента си

ment) и връща този низ с отстъп вдясно от интервали @level.

Повечето от действията са в method\_missing d . Символът m представлява липсващия

име на метод - вероятно отговаря на заглавка или елемент от списъка. Съгласие-

по този начин първата стъпка е да добавите m (с отстъп и последвано от двоеточие) към @list, заедно с

символ от нов ред e . След това увеличаваме нивото на отстъп f и добива g . (Това

стъпка се случва само ако блокът не е нулев. Обикновено можете да тествате за наличие на блок

с block\_given ?, но екземплярите BasicObject не разполагат с този метод!) Отстъпване може

задействат още липсващи извиквания на методи, в който случай те се обработват и техните резултати

добавен към @list на новото ниво на отстъп. След получаване на резултатите от добива, ние

намалете нивото на отстъп и добавете нов нов ред към @list.

В края method\_missing връща празен низ h . Целта тук е да се избегне

свързване на @list със себе си. Ако method\_missing завърши с израз, оценяващ на

@list (като @list <<"\ n"), след това вложени извиквания на method\_missing вътре в yield instruc-

tions ще върне @list и ще го добави към себе си. Празният низ прекъсва цикъла.

Ето пример за Lister в употреба:

списък = Lister.new

списък с хранителни стоки | артикул |

item.name {"Ябълки"}

item.quantity {10}

item.name {"Захар"}

item.quantity {"1 lb"}

край

lister.freeze do | f |

f.name {"Сладолед"}

край

lister.inspect do | i |

i.item {"кола"}

край

lister.sleep do | s |

с. часа {8}

край

lister.print do | документ |

document.book {"Глава 13"}

document.letter {"за редактор"}

край

поставя lister.list

Резултатът от това изпълнение е както следва:

бакалия:

име:

Ябълки

количество:

10

име:

Захар

|  |
| --- |
| **416 серия** |

**416**

C ГЛАВА 13 ***Индивидуализация на обекта***

количество:

1 lb

замръзване:

име:

Сладолед

инспектирайте:

вещ:

кола

сън:

часа:

8

печат:

Книга:

Глава 13

писмо:

към редактор

Разбира се, не е толкова удовлетворителен като Builder, но можете да проследите добивите и липсващите

извиква метода и вижте как се възползвате от екземпляр BasicObject. И ако погледнете

имената на методите, използвани в примерния код, ще видите някои, които са вградени методи

на (неосновни) обекти. Ако не наследите от BasicObject, ще получите грешка, когато

опитате се да се обадите на замразяване или инспекция. Също така е интересно да се отбележи, че сън и печат,

които са частни методи на ядрото и следователно обикновено не се извикват с

явен приемник, задейства метод\_пропускане, въпреки че строго погледнато те са частни

а не липсва.

Нашият поглед към BasicObject ни води до края на това проучване на индивидуалните обекти-

ция. Ще преминем към друга тема, която също е дълбоко свързана с Руби

динамика: извикващи и изпълними обекти.

***13.4 Обобщение***

В тази глава сте виждали

Единични класове и как да добавяме методи и константи към тях

Методи на класа

Методът на удължаване

Няколко подхода за промяна на основното поведение на Руби

BasicObject и как да го използвате

Разгледахме начините, по които обектите на Ruby отговарят на философията на Ruby, която

е, че това, което се случва по време на изпълнение, е свързано с отделни обекти и това, което те могат да направят

във всяка дадена точка. Рубиновите предмети се раждат в определен клас, но способността им да

съхраняване на отделни методи в специален единичен клас означава, че всеки обект може да направи

почти нищо.

Виждали сте как да отваряте дефиниции на единичен клас и да манипулирате вътрешностите на

отделни обекти, включително обекти от клас, които използват интензивно метода singleton

техники във връзка с методите на класа (които по същество са единични методи

os за обекти от клас). Видяхте и част от силата, както и рисковете на

|  |
| --- |
| **417 серия** |

**417**

***Обобщение***

способността на Ruby ви дава възможност да отворите не само вашите собствени класове, но и ядрото на Ruby

класове. Това е нещо, което трябва да правите пестеливо, ако изобщо - и то също е нещо

трябва да сте наясно с това, което правят други хора, за да можете да оцените рисковете от всеки

код на трета страна, който използвате, който променя основното поведение.

Завършихме с изследване на BasicObject, върховният предшественик на всички класове

и клас, който можете да използвате в случаите, когато дори ванилов Ruby обект не е достатъчно ванилов.

Следващата глава ще ни отведе в областта на извикващите се и изпълними обекти:

тиони (Proc обекти), нишки, eval блокове и др. Фактът, че можете да създавате

обекти, които въплъщават изпълним код и манипулират тези обекти, както бихте направили

object добавя още един основен слой към общата тема за динамиката на Ruby.

|  |
| --- |
| **418 серия** |

**418**

*Извиква се и*

*управляеми обекти*

В допълнение към основното, методът за хляб и масло призовава това да обяснява по-голямата част

какво се случва във вашата програма, Ruby предоставя обширен набор от инструменти за правене

нещата се случват по най-различни начини. За да стартирате, са ви необходими две или повече части от кода

паралелно? Създайте няколко нишки и ги стартирайте при необходимост. Искате да изберете

измежду набор от възможни функции за изпълнение и нямат достатъчно информация

предварително да се напишат методи за тях? Създайте масив от Proc обекти - anon-

ymous функции - и се обадете на тази, от която се нуждаете. Можете дори да изолирате методите като

обекти или изпълнява динамично създадени низове като код.

***Тази глава обхваща***

Proc обекти като анонимни функции

Ламбда методът за генериране на функции

Кодови блокове

Методът Symbol # to\_proc

Обект на метода

Обвързвания

Фамилното семейство методи

Конци

Изпълнение на външни програми

|  |
| --- |
| **419 серия** |

**419**

***Основни анонимни функции: Класът Proc***

Тази глава е за обекти, които можете да извиквате, изпълнявате или изпълнявате: нишки, анонимни

функции, низове и дори методи, които са превърнати в обекти. Ще разгледаме изобщо

на тези конструкции, заедно с някои помощни инструменти - ключови думи, обвързване на променливи, код

блокове - които правят възможно включването на Ruby на извикващи се, изпълними обекти.

Бъдете предупредени: управляемите обекти са в челните редици на трудни и променливи

теми в последните версии на Ruby. Няма как да се заобиколи фактът, че има много

на несъгласие относно начина, по който трябва да работят, и има много сложност

в това как работят. Извикващите се и изпълними обекти се различават помежду си, и в двете

синтаксис и цел, а групирането им в една глава е малко препоръчително

вх. Но това е и поучителен начин за разглеждане на тези обекти.

***14.1 Основни анонимни функции: Класът Proc***

В най-ясната си представа за *извикваем обект* е въплътена в Ruby

чрез обекти, към които можете да изпратите съобщение, с очакването, че

ще бъде изпълнен някакъв код, свързан с обектите. Основните извикващи се обекти в

Ruby са обекти на Proc, ламбди и обекти на методи. Proc обектите са самостоятелни

кодови последователности, които можете да създавате, съхранявате, предавате като аргументи на метода и,

когато желаете, изпълнете с метода на повикване. Lambdas са подобни на Proc обекти.

Истината е, че ламбда *е* Proc обект, но с малко специален вътрешен двигател

черпящ. Разликите ще се появят, докато разглеждаме всеки поотделно. Обект на метода

представляват методи, извлечени в обекти, които можете по същия начин да съхранявате, предавате,

и изпълнете.

Ще започнем нашето проучване на извикващи се обекти с Proc обекти.

ЗАБЕЛЕЖКА За сбитост терминът *proc* (с обикновен шрифт) ще се използва

текстът да означава Proc обект, както *низът се* отнася до екземпляр на класа

Струна. *Ламбда* ще означава екземпляр на ламбда стила на обекта Proc.

(Не се притеснявайте; скоро ще видите какво означава това!) Терминът *функция* е общ

термин за самостоятелни единици код, които вземат вход и връщат стойност. Има

няма Функционален клас в Ruby. Тук обаче понякога ще видите използваната *функция*

за да се позовем на procs и lambdas. Това е просто още един, малко по-абстрактен начин

идентифициране на тези обекти.

***14.1.1 Proc обекти***

Разбирането на Proc обектите задълбочено означава да сте запознати с няколко неща:

основи на създаване и използване на procs; начина, по който procs обработват аргументи и променлива bind-

ings; ролята на процесорите като *затваряне* ; връзката между procs и кодови блокове; и

разликата между създаването на procs с Proc.new, метода proc, ламбда

метод и литералния ламбда конструктор ->. Тук се случват много неща, но всичко това

пасва заедно, ако го вземате един по един слой.

Нека започнем с основния извикваем обект: екземпляр на Proc, създаден с Proc.new.

Създавате Proc обект чрез създаване на инстанция на клас Proc, включително кодов блок:

pr = Proc.new {поставя „Вътре в блока на Proc“}

|  |
| --- |
| **420 серия** |

**420**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

Кодовият блок става тяло на процеса; когато се обадите на proc, блокирайте ви

предоставеното се изпълнява. По този начин, ако се обадите на pr

pr.call

той докладва, както следва:

Вътре в блок на Proc

Това е основният сценарий: кодов блок, предоставен на повикване към Proc.new, се превръща в

тяло на обекта Proc и се изпълнява, когато извикате този обект. Всичко друго

което се случва или може да се случи, включва допълнения и вариации по тази тема.

Не забравяйте, че procs са обекти. Това означава, че можете да ги присвоите на променливи,

поставете ги вътре в масиви, изпратете ги като аргументи на метода и обикновено лекувайте

като всеки друг обект. Те имат познания за парче код (

блока с кода, с който са създадени) и възможността за изпълнение на този код при поискване.

Но те все още са обекти.

Може би най-важният аспект на процесите, за да се справите, е връзката между тях

procs и кодови блокове. Тази връзка е интимна и се оказва важен ключ

за по-нататъшно разбиране.

***14.1.2 Процеси и блокове и как се различават***

Когато създавате Proc обект, винаги предоставяте кодов блок. Но не всеки код

блок служи като основа на проц. Фрагментът

[1,2,3] .всеки {| x | поставя x \* 10}

включва кодов блок, но не създава proc. И все пак сюжетът е малко по-дебел от

че. Методът може да улови блок, обективиран в процес, използвайки специалния параметър

тер синтаксис, който видяхте накратко в глава 9:

def call\_a\_proc (& block)

block.call

край

call\_a\_proc {поставя "Аз съм блокът ... или Proc ... или нещо подобно." }

Методът proc

Методът proc взема блок и връща обект Proc. По този начин можете да кажете proc

{поставя "Здравей!" } вместо Proc.new {поставя "Здравей!" } и да получите същия резултат.

Proc.new и proc бяха малко по-различни един от друг, като proc обслужваха

като синоним на ламбда (виж раздел 14.2) и методите proc / lambda

създаване на специализирани Proc обекти, които не бяха съвсем същите като тези на Proc.new

принуден. Да, беше объркващо. Но сега, въпреки че все още има два варианта на Proc

object, Proc.new и proc правят същото, докато lambda произвежда другото

вариант. Поне именуването се подрежда по-предсказуемо.

|  |
| --- |
| **421 серия** |

**421**

***Основни анонимни функции: Класът Proc***

Резултатът не е изненадващ:

Аз съм блокът ... или Proc ... или нещо такова.

Но също така е възможно proc да служи вместо блока на кода в извикване на метод,

използвайки подобен специален синтаксис:

p = Proc.нови {| x | поставя x.upcase}

% w {Дейвид Блек} .each (& p)

Ето изхода от това обаждане за всеки:

ДЕЙВИД

ЧЕРЕН

Но въпросът остава: точно какво се случва по отношение на procs и блокове?

Защо и как присъствието на (& p) убеждава всеки, че не се нуждае от действително

код блок?

До голяма степен връзката между блоковете и процесорите се свежда до въпрос на

синтаксис спрямо обекти.

S YNTAX ( БЛОКОВЕ ) И ОБЕКТИ ( ПРОЦЕСИ )

Важен и често погрешно разбран факт е, че кодовият блок на Ruby не е обект.

Този познат тривиален пример има приемник, точков оператор, име на метод и a

код блок:

[1,2,3] .всеки {| x | поставя x \* 10}

Приемникът е обект, но кодовият блок не е. По-скоро кодовият блок е част от

синтаксис на извикването на метода.

Можете да поставите кодови блокове в контекст, като мислите за аналогията със списъци с аргументи.

В извикване на метод с аргументи

поставя c2f (100)

аргументите са обекти, но самият списък с аргументи - цялото (100) нещо - не е

предмет. Няма клас ArgumentList и няма клас CodeBlock.

Нещата стават малко по-сложни в случай на синтаксис на блокове, отколкото в случая на

списъци с аргументи, обаче, поради начина, по който блоковете и процесите взаимодействат. Екземпляр

на Proc е обект. Кодовият блок съдържа всичко, което е необходимо за създаване на процес.

Ето защо Proc.new взема кодов блок: по този начин открива какво предлага процедурата

е готов да го направи, когато се обади.

Едно важно значение на факта, че кодовият блок е синтактична конструкция

а не обект е, че кодовите блокове не са аргументи на метода. Въпросът на прови-

аргументирането на метод е независимо от това дали е наличен кодов блок, точно както

присъствието на блок е независимо от наличието или отсъствието на списък с аргументи.

Когато предоставяте кодов блок, не изпращате блока на метода като

аргумент; предоставяте кодов блок и това е нещо само за себе си. Да вземем

друг, по-внимателен поглед сега на механизмите за преобразуване, които позволяват да бъдат кодови блокове

заснети като procs и procs, които трябва да бъдат използвани вместо кодови блокове.

|  |
| --- |
| **422 серия** |

**422**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

***14.1.3 Преобразувания в блок-процес***

Преобразуването между блокове и procs е лесно - което не е твърде изненадващо, защото

целта на кодов блок е да бъде изпълнена, а proc е обект, чиято задача е да

предоставят достъп за изпълнение на предварително дефиниран кодов блок. Първо ще разгледаме блок-

to-proc преобразувания и след това при използване на procs вместо блокове.

C ПРИЛАГАНЕ НА КОДЕН БЛОК КАТО P ROC

Нека започнем с друг прост метод, който улавя своя кодов блок като Proc обект

и впоследствие извиква този обект:

def capture\_block (& ​​block)

block.call

край

capture\_block {поставя „Вътре в блока“}

Това, което се случва, е вид неявно извикване на Proc.new, използвайки същия блок. Процесът

така създадената е обвързана с блока на параметрите.

Фигура 14.1 предоставя изобразяване на художник за това как кодов блок се превръща в proc. The

първото събитие (в долната част на фигурата) е извикването на метода capture\_block

с кодов блок. По пътя се създава нов Proc обект (стъпка 2), използвайки същия

блок. Това е този Proc обект, към който е свързан променливият блок, вътре в метода

тяло (стъпка 3).

По този начин синтактичният елемент (кодовият блок) служи като основа за създаването

на обект. Обяснява се и „фантомната“ стъпка за създаване на процеса от блока

необходимостта от специалния и базиран синтаксис. Извикването на метод може да включва както аргумент

списък с ментове и кодов блок. Без специално знаме като &, Руби няма как да знае

че искате да спрете обвързването на параметрите с редовни аргументи и вместо това да изпълните

преобразуване на блок в процес и запазване на резултатите.

& Също се появява, когато искате да направите преобразуването другия

начин: използвайте Proc обект вместо кодов блок.

def capture\_block (& ​​block)

поставя "Got block as proc"

block.call

край

Proc.new {поставя „Вътре в блока“}

capture\_block {поставя „Вътре в блока“}

2

1

3

Фигура 14.1 Фантомната инсталация на **Proc се** намесва между извикване на метод и

метод.

|  |
| --- |
| **423 серия** |

**423**

***Основни анонимни функции: Класът Proc***

U пее PROCs при блокове

Ето как можете да извикате capture\_block, използвайки proc вместо кодов блок:

p = Proc.new {поставя „Този ​​аргумент на proc ще служи като кодов блок.“ }

capture\_block (& ​​p)

Изходът е

Този аргумент proc ще служи като кодов блок.

Ключът към използването на proc като блок е, че всъщност го използвате вместо блок: you

изпратете proc като аргумент на метода, който извиквате. Точно както маркирате

параметър в дефиницията на метода със знака &, за да покаже, че трябва

конвертирате блока в proc, така че вие ​​също използвате & от страната на извикване на метод в indi-

cate, че proc трябва да свърши работата на кодов блок.

Имайте предвид, че тъй като процесът, маркиран с &, служи за кодов блок, вие

не може да изпрати кодов блок в същото извикване на метод. Ако го направите, ще получите грешка. Повикването

capture\_block (& ​​p) {поставя "Това е изричният блок"}

води до грешка „даден аргумент на блока и действителен блок.“ Руби не може да реши кое

субектът - proc или блокът - служи като блок, така че можете да използвате само един.

Тук се случва интересна подпрограма. Подобно на много Ruby оператори, & in & p е

обвивка около метод: а именно методът to\_proc. Извикване на to\_proc на a

Proc обектът връща самия Proc обект, по-скоро като извикване на to\_s на низ или to\_i

върху цяло число.

Но имайте предвид, че все още имате нужда от &. Ако направите това

улавяне\_блок (п)

или това

улавяне\_блок (p.to\_proc)

proc служи като редовен аргумент на метода. Вие не задействате специалното

поведение, при което аргумент proc върши работата на кодов блок.

По този начин & in capture\_block (& ​​p) прави две неща: задейства повикване към p's to\_proc

метод и той казва на Ruby, че полученият обект Proc служи като стенд на кодов блок

ин. И тъй като to\_proc е метод, е възможно да се използва по-общо.

G ENERALIZING ДА \_ PROC

На теория можете да определите to\_proc във всеки клас или за всеки обект, както и техниката &

след това ще работи за засегнатите обекти. Вероятно няма да ви се налага да правите това много; на

два класа, където to\_proc е най-полезен, са Proc (обсъдени по-рано) и Symbol (dis-

cussed в следващия раздел) и поведението to\_proc вече е вградено в тези класове.

Но разглеждането на това как да превъртите to\_proc във вашите собствени класове може да ви даде представа за

динамична сила, която се намира под повърхността на езика.

Ето един доста странен, но поучителен код:

|  |
| --- |
| **424 серия** |

**424**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

клас Личност

attr\_accessor: име

def self.to\_proc

Proc.new {| човек | person.name}

край

край

d = Person.new

d.name = "Дейвид"

m = Person.new

m.name = "Matz"

поставя [d, m] .map (& Person)

Най-добрата отправна точка, ако искате да следвате следите от галета през това

код, е последният ред e . Тук имаме масив от два обекта Person. Правим a

map операция върху масива. Както знаете, Array # map взема кодов блок. В такъв случай,

вместо това използваме обект Proc. Този процес е обозначен в списъка с аргументи като

& Личност. Разбира се, Person не е процес; това е клас. За да разбере какво вижда, Руби

иска от Person да се представи като процес, което означава имплицитен разговор към Person

to\_proc метод c .

Този метод от своя страна създава прост Proc обект, който взема един аргумент и

извиква метода на името на този аргумент. Лице обекти имат име атрибути B . И

обектите Person, създадени с цел изпробване на кода, със сигурност имат

имена d . Всичко това означава, че картографирането на масива от обекти на Person ([d, m])

ще събере атрибутите на имената на обектите и целият получен масив ще бъде

разпечатан (благодарение на путове).

Това е дълъг път наоколо. И дизайнът е малко свободен; в крайна сметка всеки метод, който

отнема блок, който може да използва & Person, което може да стане странно, ако се касае за не-човек

обекти, които нямат метод за име. Но примерът ви показва това to\_proc

може да служи като мощна кука за преобразуване. И това е, което прави в класа Symbol,

както ще видите по-нататък.

***14.1.4 Използване на символ # to\_proc за сбитост***

Вграденият метод Symbol # to\_proc влиза в действие в ситуации като тази:

% w {Дейвид Черно} .map (&: капитализиране)

Резултатът е

[„Дейвид“, „Черен“]

Символът: капитализиране се интерпретира като съобщение, което трябва да бъде изпратено до всеки елемент от

масив от своя страна. По този начин предишният код е еквивалентен на

% w {Дейвид черно} .map {| str | str.capitalize}

но, както виждате, по-кратък.

Ако току-що сте видели &: capitalize или подобна конструкция в кода, може би си мислите, че е така

загадъчен. Но знаейки как се анализира - знаейки, че: изписването с главни букви е символ и & е a

спусък to\_proc — позволява ви да го интерпретирате правилно и да оцените неговата изразителност.

б

° С

д

д

|  |
| --- |
| **425 серия** |

**425**

***Основни анонимни функции: Класът Proc***

Ситуацията Symbol # to\_proc се поддава добре на премахването на скобите:

% w {david black} .map &: главни букви

Като свалите скобите, можете да направите символа, който е посочен, да изглежда така, както е в него

позиция на кодовия блок. Няма нужда от това, разбира се, и трябва да запазите

имайте предвид, че когато използвате индикатора to\_proc &, изпращате процеса като

аргумент, маркиран с & и не предоставящ буквален кодов блок.

Символ # to\_proc е, наред с други неща, чудесен пример за нещо, което Ruby

прави за вас това, което бихте могли, ако трябва, да направите лесно сами. Ето как.

Аз MPLEMENTING S със символ # ДА \_ PROC

Ето отново казус на to\_proc:

% w {Дейвид Черно} .map (&: капитализиране)

Знаем, че е еквивалентно на това:

% w {Дейвид черно} .map {| str | str.capitalize}

И същото може да се напише и така:

% w {Дейвид черно} .map {| str | str.send (: капитализиране)}

Обикновено не бихте го написали по този начин, защото няма нужда да се занимавате с проблема

за извършване на изпращане, ако можете да извикате метода, използвайки синтаксис на обикновена точка. Но

версията, базирана на изпращане, посочва пътя към внедряване на Symbol # to\_proc. Работата

на блока в този пример е да изпрати символа: капитализиране на всеки елемент от

масив. Това означава, че Proc, произведен от: capitalize # to\_proc, трябва да изпрати: capitalize

към неговия аргумент. Обобщавайки от това, можем да измислим това просто (почти анти-

климатично, може да се каже) изпълнение на Symbol # to\_proc:

клас Символ

def to\_proc

Proc.new {| obj | obj.send (самостоятелно)}

край

край

Този метод връща Proc обект, който приема един аргумент и изпраща самостоятелно (което

ще бъде какъвто и символ да използваме) за този обект.

Можете да опитате новото внедряване в irb. Нека хвърлим поздрав от

метод, така че е ясно, че използваната версия е тази, която току-що дефинирахме:

клас Символ

def to\_proc

поставя "В новия символ # to\_proc!"

Proc.new {| obj | obj.send (самостоятелно)}

край

край

Запазете този код във файл, наречен sym2proc.rb, и от директорията, в която сте запазили

го, издърпайте го в irb, като използвате флага –I (включете пътя в пътя за зареждане) и флага -r (изискване):

irb --simple-prompt –I. -r sym2proc

|  |
| --- |
| **426 серия** |

**426**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

Сега ще видите новия to\_proc в действие, когато използвате техниката *символ*& :

>>% w {david black} .map (&: изписване с главни букви)

В новия символ # to\_proc!

=> [„Дейвид“, „Черен“]

Вие не сте задължени да използвате пряк път Symbol # to\_proc (да не говорим за изпълнение

това), но е полезно да знаете как работи, за да можете да решите кога е подходящо да се използва.

Един от най-важните аспекти на обектите на Proc е тяхното обслужване като *затваряне* :

ymous функции, които запазват локалните обвързващи променливи, които са в сила, когато

създават се procs. След това ще разгледаме как процесорите работят като затваряне.

***14.1.5 Процедури като затваряне***

Вече видяхте, че локалните променливи, които използвате в тялото на метода, не са

същото като локалните променливи, които използвате в обхвата на извикването на метода:

def talk

a = "Здравей"

поставя а

край

a = "Сбогом"

говоря

поставя а

Идентификаторът a е бил присвоен два пъти, но двете присвоявания (двете вариа-

ables) не са свързани помежду си.

Също така сте виждали, че кодовите блокове запазват променливите, които са съществували в

времето, когато са създадени. Всички кодови блокове правят това:

m = 10

[1,2,3] .всеки {| x | поставя x \* m}

Това поведение става значимо, когато кодовият блок служи като тяло на повикване-

способен обект:

def multiply\_by (m)

Proc.new {| x | поставя x \* m}

край

mult = multiply\_by (10)

многократно обаждане (12)

В този пример методът multiply\_by връща процес, който може да бъде извикан с всеки

аргумент, но това винаги се умножава по числото, изпратено като аргумент на multiply\_by.

Променливата m, независимо от нейната стойност, се запазва вътре в кодовия блок, предаден на

Proc.new и следователно служи като множител всеки път, когато Proc обектът се върне

от multiply\_by се извиква.

Proc обектите поставят малко по-различно завъртане на обхвата. Когато конструирате кода

блок за извикване на Proc.new, локалните променливи, които сте създали, все още са в обхват (както при

всеки кодов блок). И тези променливи остават в обхвата в процеса, без значение

къде или когато го извикате.

Погледнете следния списък и следете двете променливи, наречени a.

**Изход:**

**Здравейте**

**Изход:**

**Довиждане**

**Ouptut:**

**120**

|  |
| --- |
| **427 серия** |

**427**

***Основни анонимни функции: Класът Proc***

def call\_some\_proc (pr)

a = "без значение 'a' в обхвата на метода"

поставя а

pr.call

край

a = "'a' да се използва в Proc block"

pr = Proc.new {поставя}

pr.call

call\_some\_proc (pr)

Както в предишния пример, в дефиницията на метода B има a и a в

външен (извикващ) обхват d . Вътре в метода е извикване на процес. Кодът за този процес,

случайно знаем, се състои от поставя a. Забележете, че когато proc е извикан отвътре

методът c , разпечатаният a не е дефинираният в метода; това е a

от обхвата, където първоначално е създаден proc:

'a', който ще се използва в Proc блок

без значение „a“ в обхвата на метода

'a', който ще се използва в Proc блок

Обектът Proc носи своя контекст наоколо. Част от този контекст е променлива

наречен a, на който е определен определен низ. Тази променлива живее вътре в Proc.

Код, който носи контекста на създаване около себе си по този начин, се нарича *clo-*

*разбира се* . Създаването на затваряне е като опаковане на куфар: където и да отворите куфара, той

задържа това, което сте поставили, когато сте го опаковали. Когато отворите затваряне (като го извикате), то

съдържа това, което сте вложили в него, когато е било създадено. Затварянията са важни, защото те

запазване на частично работещо състояние на програма. Променлива, която излиза извън обхвата, когато a

връщането на метода може да има нещо интересно да се каже по-късно - и със затваряне вие

може да запази тази променлива, за да може да продължи да предоставя информация или резултати от изчисленията.

Класическият пример за затваряне е брояч. Ето метод, който връща затваряне (a

proc със запазени локални променливи обвързвания). Процесът служи като брояч; увеличава се

изменя своята променлива всеки път, когато се извика:

def make\_counter

n = 0

върнете Proc.new {n + = 1}

край

c = make\_counter

поставя c.call

поставя c.call

d = make\_counter

поставя d.call

поставя c.call

Изходът е

1

2

1

3

Листинг 14.1 **Proc** обект, запазващ локален контекст

б

° С

д

б

° С

д

д

|  |
| --- |
| **428 серия** |

**428**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

Логиката в процеса включва добавяне на 1 към n B ; така че първият път, когато е извикан proc, то

оценява на 1; втори път до 2; и така нататък. Извикване на make\_counter и след това повикване-

изготвянето на процеса, който той връща, потвърждава това: първо се отпечатва 1, а след това 2 c . Но нова страна

ter започва отново от 1; второто повикване към make\_counter d генерира нов, локален n,

което се запазва в различен процес. Разликата между двата брояча е

стана ясно от третото обаждане до първия брояч, което отпечатва 3 д . Той вдига къде е

спря, като използва n променливата, запазена вътре в нея по време на нейното създаване.

Както всеки кодов блок, блокът, който предоставяте, когато създавате Proc обект, може да поеме

аргументи. Нека разгледаме подробно как работят аргументите и параметрите на блока в

курс на създаване на Proc.

***14.1.6 Proc параметри и аргументи***

Ето екземпляр на Proc, с блок, който приема един аргумент:

pr = Proc.new {| x | поставя "Извикан с аргумент # {x}"}

pr.call (100)

Изходът е

Извиква се с аргумент 100

Процесите се различават от методите по отношение на обработката на аргументи по това, че не им пука

дали получават точния брой аргументи. Процес с един аргумент, като този

>> pr = Proc.new {| x | px}

=> # <Proc: 0x000001029a8960 @ (irb): 1>

може да бъде извикан с произволен брой аргументи, включително нито един. Ако се извика с не

аргументи, единичният му параметър се задава на нула:

>> pr.call

нула

Ако се извика с повече от един аргумент, единичният параметър е обвързан с първия

аргумент, а останалите аргументи се отхвърлят:

>> pr.call (1,2,3)

1

(Не забравяйте, че разпечатаната единична стойност е стойността на променливата x.)

Можете, разбира се, да използвате и аргументи „гъба“ и всички останали параметри-

избройте принадлежности, за които вече сте научили. Но имайте предвид точката, че procs

са малко по-малко придирчиви от методите за броя на аргументите им - тяхната *същност* . И все пак, Руби

предоставя начин за създаване на по-сутринни функции: ламбда методът.

***14.2 Създаване на функции с ламбда и ->***

Подобно на Proc.new, ламбда методът връща Proc обект, използвайки предоставения код

блок като тяло на функцията:

>> lam = lambda {поставя "A lambda!" }

=> # <Proc: 0x0000010299a1d0 @ (irb): 2 (ламбда)>

|  |
| --- |
| **429 серия** |

**429**

***Създаване на функции с ламбда и ->***

>> lam.call

Ламбда!

Както инспектиращият низ предполага, обектът, върнат от ламбда, е от клас Proc. Но

обърнете внимание на (ламбда) нотацията. Няма клас Lambda, но има различен ламбда fla-

преди клас Proc. А ламбда ароматизираните проки са малко по-различни от ванилията си

братовчеди, по три начина.

Първо, ламбдите изискват изрично създаване. Където Ruby създава Proc обекти

имплицитно, те са обикновени procs, а не ламбда. Това означава главно, че когато вие

вземете кодов блок в метод, като този

def m (& block)

обектът Proc, който сте грабнали, е обикновен процес, а не ламбда.

Второ, ламбдите се различават от procs по това как се отнасят към ключовата дума return. връщане

вътре в ламбда задейства изход от тялото на ламбда към контекста на кода

веднага съдържащ ламбда. return вътре в proc задейства връщане от

метод, при който процесът се изпълнява. Ето илюстрация на разликата:

def return\_test

l = ламбда {връщане}

л. повикване

поставя "Все още тук!"

p = Proc.new {return}

p.call

поставя „Няма да видите това съобщение!“

край

return\_test

Резултатът от този фрагмент е „Все още тук!“ Никога няма да видите второто съобщение d

разпечатва се, защото извикването на обекта Proc c предизвиква връщане от

return\_test метод. Но повикването към ламбда B задейства връщане (изход) от

тялото на ламбда и изпълнението на метода продължава там, където е спряно.

ПРЕДУПРЕЖДЕНИЕ, тъй като връщането отвътре (с не-ламбда-ароматизиран) процес задейства a

return от метода за затваряне, извиквайки proc, който съдържа return when

не сте вътре, нито един метод не води до фатална грешка. За да видите демонстрация на това

грешка, опитайте от командния ред: ruby ​​-e 'Proc.new {return} .call'.

И накрая, и най-важното, ламбда-ароматизираните procs не обичат да бъдат извиквани с

грешен брой аргументи. Те са суетливи:

>> lam = ламбда {| x | px}

=> # <Proc: 0x000001029901f8 @ (irb): 3 (ламбда)>

>> lam.call (1)

1

=> 1

>> lam.call

ArgumentError: грешен брой аргументи (0 за 1)

>> lam.call (1,2,3)

ArgumentError: грешен брой аргументи (3 за 1)

В допълнение към ламбда метода има ламбда литерален конструктор.

б

° С

д

|  |
| --- |
| **Страница 430** |

**430**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

T HE " Камата Ламбда " ИЗПЪЛНИТЕЛ , ->

Ламбда конструкторът (с прякор „стабилна ламбда“) работи по следния начин:

>> lam = -> {поставя "здравей"}

=> # <Proc: 0x0000010289f140 @ (irb): 1 (lambda)>

>> lam.call

здравей

Ако искате вашата ламбда да взема аргументи, трябва да поставите параметрите си в родителски

тези след ->, *не* във вертикални тръби вътре в кодовия блок:

>> мулти = -> (x, y) {x \* y}

=> # <Proc: 0x00000101023c38 @ (irb): 7 (ламбда)>

>> mult.call (3,4)

=> 12

Малко история: стабилната ламбда съществува на първо място, защото по-стари версии на

Руби имаше проблеми при синтактичния анализ на синтаксиса на аргумента в стил стил във вертикалните тръби. За

например в Ruby 1.8 не можете да използвате синтаксис по подразбиране като този:

ламбда {| a, b = 1 | „Не работи в Ruby 1.8 - синтаксична грешка!“ }

Проблемът беше, че Руби не знаеше дали втората тръба е втора граница

iter или битов оператор OR . Стабилната ламбда е въведена, за да стане възможно

използвайте пълноценни аргументи в стил метод с ламбда:

-> (a, b = 1) {"Работи в Ruby 1.8!" }

В крайна сметка ограничението на анализатора беше преодоляно; вече можете да използвате метод-аргумент

синтаксис, в цялата му слава, между вертикалните тръби в кодов блок. Строго погледнато,

следователно стабилната ламбда вече не е необходима. Но това привлече малко след-

и ще видите, че се използва доста широко.

На практика нещата, които най-често се обаждате в Ruby, не са procs или lambdas, а

методи. Досега сме разглеждали извикването на методи като нещо, което правим на едно ниво

на remove: изпращаме съобщения до обекти и обектите се изпълняват по подходящ начин

наименован метод. Но е възможно да се обработват методи като обекти, както ще видите по-нататък.

***14.3 Методи като обекти***

Методите не се представят като обекти, докато не им кажете. Методи за лечение

тъй като обектите включват *обективирането* им.

***14.3.1 Заснемане на обекти на Метод***

Можете да се сдобиете с обект на метод, като използвате метода на метода с името на

метод като аргумент (под формата на низ или символ):

клас С

def talk

поставя "Тест за улавяне на метод! self е # {self}."

край

край

c = C. нов

meth = c.method (: беседа)

|  |
| --- |
| **Страница 431** |

**431**

***Методи като обекти***

В този момент имате обект на метод - по-конкретно, *обвързан* обект на метод: той не е

метод на разговор в абстрактно, а по-скоро метод на разговор, конкретно обвързан с обекта

° С. Ако изпратите съобщение за повикване до meth, той знае да се извика с c в ролята на себе си:

meth.call

Ето резултата:

Тест за грабване на методи! self е # <C: 0x00000101201a00>.

Можете също така да освободите метода от неговия обект и след това да го свържете с друг обект, като

стига този друг обект да е от същия клас като оригиналния обект (или подклас):

клас D <C

край

d = D.нови

несвързан = meth.unbind

unbound.bind (d) .call

Тук изходът ви казва, че методът наистина е свързан с D обект (d) в

времето, когато е изпълнено:

Тест за грабване на методи! self е # <D: 0x000001011d0220>.

За да получите директно обект на несвързан метод, без да се налага да извиквате unbind на a

bound метод, можете да го получите от класа, а не от конкретен екземпляр на

класът, използващ метода instance\_method. Този единичен ред е еквивалентен на метод

обаждане плюс отмяна на повикване:

несвързан = C.instance\_method (: беседа)

След като имате метод за несвързан в плен, така да се каже, можете да използвате bind за свързване

тя към всеки екземпляр на C или C подклас като D.

Но защо бихте?

***14.3.2 Обосновката на методите като обекти***

Няма съмнение, че методите за развързване и обвързване са специализирана техника и

не е вероятно да се нуждаете от повече от четене на знания за него. Но освен

принцип, че поне познанията за четене на каквото и да е в Ruby не могат да бъдат лоша идея

в някои случаи най-добрият отговор на въпрос „как да“ е „С необвързани методи“.

Ето един пример. Следният въпрос възниква периодично във форумите на Ruby:

*Да предположим, че имам йерархия на класа, където методът се предефинира:*

клас А

def a\_method

поставя "Определение в клас А"

край

край

клас B <A

def a\_method

поставя "Определение в клас B (подклас на A)"

край

край

|  |
| --- |
| **432 серия** |

**432**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

клас C <B

край

*И имам екземпляр на подкласа:*

c = C. нов

*Има ли начин да накарате този екземпляр от най-ниския клас да отговори на съобщението (* a\_method *) чрез*

*изпълнение на версията на метода в класа два класа нагоре по веригата?*

По подразбиране, разбира се, инстанцията не прави това; изпълнява първото съвпадение

метод, който намира, докато пресича пътя на търсене на метода:

c.a\_method

Изходът е

Определение в клас B (подклас на A)

Но можете да принудите проблема чрез операция за развързване и свързване:

A.instance\_method (: a\_method) .bind (c) .call

Тук изходът е

Определение в клас А

Можете дори да скриете това поведение в метод в клас C: \

клас С

def call\_original

A.instance\_method (: a\_method) .bind (self) .call

край

край

и след това извикайте call\_original директно на c.

Това е пример за техника на Ruby с парадоксален статус: тя е в рамките на

царството на нещата, които трябва да разберете, като някой, който придобива владеене на динамиката на Руби

ics, но това е извън сферата на всичко, което вероятно би трябвало да направите. Ако откриете себе си

принуждавайки Ruby обектите да реагират на методи, които вече сте предефинирали, трябва

прегледайте дизайна на вашата програма и намерете начин да накарате обектите да правят това, което искате

резултат от, а не въпреки йерархията клас / модул, която сте създали.

И все пак методите са обекти, които могат да се извикат и могат да бъдат отделени (несвързани) от

техните екземпляри. Като привърженик на динамиката на Ruby, трябва поне да имате признание-

ниво на знания за този вид операция.

Ще се задържим в динамичната стратосфера за известно време, като погледнем следващата евална фамилия

ily of methods: малка шепа методи със специални правомощия, които ви позволяват да изпълнявате низове като

кодирайте и манипулирайте обхвата и себе си по някои интересни, управлявани от случая начини.

|  |
| --- |
| **Страница 433** |

**433**

***Фамилното семейство методи***

***14.4 Евал семейството методи***

Подобно на много езици, и Ruby има средство за изпълнение на код, съхранен под формата на

низове по време на изпълнение. Всъщност Руби има куп техники за това, всяка от които

служи за определена цел, но всички те работят на сходен принцип: този на

казвайки в средата на програма, „Каквито и да са кодови низове, които сте прочели

от програмния файл, преди да започнете да изпълнявате тази програма, изпълнете *този* код

низ в момента. "

Най-ясният метод за оценяване на низ като код, а също и най-много

опасно, е методът eval. Другите методи на eval-family са малко по-меки, не

защото те също не оценяват низовете като код, но защото това не е всичко, което правят.

instance\_eval води до временна промяна в стойността на self и class\_eval

(известен също като синоним module\_eval) ще ви отведе на ad hoc странично пътуване в

контекст на блок с дефиниция на клас. Тези методи на eval-family могат да работят върху низове,

но те могат да бъдат извикани и с кодов блок; по този начин те не винаги работят толкова откровено

като eval, който изпълнява низове.

Нека разопаковаме това описание, като разгледаме по-отблизо eval и другите eval методи.

***14.4.1 Изпълнение на произволни низове като код с eval***

eval изпълнява низа, който сте му дали:

>> eval ("2 + 2")

=> 4

eval е отговорът или поне един отговор на редица често задавани въпроси,

като например „Как да напиша метод и да му дам име, което някой въвежда?“ Можете да направите

така по този начин:

Алтернативни техники за извикване на извикващи се обекти

Досега използвахме изключително метода на повикване за извикване на обекти, които могат да се извикат. Направете, как-

някога, имайте няколко други опции.

Единият е методът / операторът с квадратни скоби, който е синоним на повикване. Вие поставяте

всякакви аргументи в скобите:

mult = ламбда {| x, y | x \* y}

дванадесет = мулти [3,4]

Ако няма аргументи, оставете скобите празни.

Можете също да извикате извикващи се обекти, използвайки метода ():

дванадесет = мулти (3,4)

Забележете точката преди отварящата скоба. Методът () трябва да бъде извикан с помощта на

точка; не можете просто да добавите скобите към обект на Proc или Method по начина, по който вие

би с име на метод. Ако няма аргументи, оставете скобите празни.

|  |
| --- |
| **Страница 434** |

**434**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

print "Име на метод:"

m = gets.chomp

eval ("def # {m}; поставя 'Здравей!'; край")

eval (m)

Този код извежда

Здравей!

Пише се нов метод. Да предположим, че стартирате кода и въведете abc. Низът

впоследствие използвате eval на is

def abc; поставя „Здравей!“; край

След като приложите eval към този низ, съществува метод, наречен abc. Вторият eval exe-

изрязва низа abc - който, като се има предвид създаването на метода в предишния ред,

представлява призив към abc. Когато се извика abc, „Вътре в нов метод!“ се разпечатва.

eval ви дава много сила, но също така крие опасности - според мнението на някои хора,

достатъчно опасност, за да се изключи като използваема техника.

Класът на обвързване и изравняващ код с подвързване

Ruby има клас, наречен Binding, чиито екземпляри капсулират локалната променлива

обвързвания в сила в даден момент на изпълнение. И метод от най-високо ниво, наречен обвързване

връща каквото е текущото обвързване.

Най-честото използване на Binding обекти е в позицията на втори аргумент на

eval. Ако предоставите обвързване в тази позиция, низът, който се изчиства, се изпълнява

в контекста на даденото обвързване. Всички локални променливи, използвани вътре в евалния низ, са

тълкувано в контекста на това обвързване.

Ето един пример. Методът use\_a\_binding приема обект Binding като аргумент

и го използва като втори аргумент за призив към eval. Операцията eval,

следователно използва локалните обвързващи променливи, представени от обекта Binding:

def use\_a\_binding (b)

eval ("поставя str", b)

край

str = "Аз съм низ в обвързване от най-високо ниво!"

use\_a\_binding (обвързване)

Резултатът от този фрагмент е „Аз съм низ в обвързване от най-високо ниво!“. Този низ е

обвързан с променливата от най-високо ниво str. Въпреки че str не е в обхвата в use\_a\_

метод на обвързване, той е видим за eval благодарение на факта, че eval получава обвързващ аргумент

на връзката от най-високо ниво, в която str е дефиниран и обвързан.

По този начин низът "поставя str", което в противен случай би довело до грешка (тъй като str не е

дефиниран), може да бъде премахнат успешно в контекста на даденото обвързване.

|  |
| --- |
| **Страница 435** |

**435**

***Фамилното семейство методи***

***14.4.2 Опасностите от eval***

Изпълнението на произволни низове носи значителна опасност - особено (макар и не изключващо

низове), които идват от потребители, взаимодействащи с вашата програма. Например, то

би било лесно да се инжектира разрушителна команда, може би системно извикване на rm –rf / \*

предишния пример.

eval може да бъде съблазнителен. Това е толкова динамично, колкото и динамичната техника на програмиране

можете да получите: оценявате низове от код, които вероятно дори не са съществували, когато сте

написа програмата. Навсякъде, където Руби поставя някаква бариера за абсолютно, лесно

манипулиране на състоянието на нещата по време на изпълнение на програма, eval изглежда предлага a

начин да пресечете бюрокрацията и да правите каквото искате.

Но както виждате, eval не е панацея. Ако използвате eval на низ, вие сте

написано, обикновено не е по-малко сигурно от това да стартирате програмен файл, който сте написали. Но всякакви

когато е включен несигурен, динамично генериран низ, опасността е гъбата.

По-специално е трудно да се изчисти потребителското въвеждане (включително въвеждането от уеб формуляри

и файлове) до такава степен, че можете да се чувствате в безопасност при стартирането на eval върху него. Руби поддържа

глобална променлива, наречена $ SAFE, която можете да зададете на голямо число (по скала от 0 до 4)

за да спечелите защита от опасности като измамни заявки за писане на файлове. $ SAFE прави живота с

eval много по-безопасно. И все пак най-добрият навик да се възползвате е навикът да не използвате eval.

Не е трудно да се намерят опитни и експертни програмисти на Ruby (както и

грамати на други езици), които никога не използват eval и никога няма. Трябва да решите

как се чувствате по въпроса, въз основа на вашите познания за клопките.

Нека да преминем сега към по-широкото семейство методи. Тези методи могат да направят

същия вид груба сила оценка струна, че eval прави, но те също имат по-добро,

по-нежно поведение, което ги прави използваеми и полезни.

***14.4.3 Методът instance\_eval***

instance\_eval е специализиран братовчед на eval. Той оценява низа или кодовия блок

вие го давате, променяйки себе си, за да бъде получателят на повикването към instance\_eval:

р себе си

a = []

a.instance\_eval {p self}

Този фрагмент извежда две различни самостоятелни:

основен

[]

instance\_eval е предимно полезен за разбиване на това, което обикновено би било друго

частни данни на обекта - особено променливи на екземпляра. Ето как да видите стойността на

инстанция променлива, принадлежаща на който и да е стар обект (в този случай променливата на инстанцията на @x

на C обект):

клас С

def инициализира

@x = 1

край

край

|  |
| --- |
| **436 серия** |

**436**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

c = C. нов

c.instance\_eval {поставя @x}

Този вид вглеждане в състоянието на друг обект обикновено се счита за неучтив; ако an

обект иска да знаете нещо за състоянието му, той предоставя методи чрез

които можете да попитате. Въпреки това, тъй като динамиката на Ruby се основава на промените

Идентичността на себе си не е лоша идея езикът да ни даде техника

манипулиране на себе си директно.

Може би най-честата употреба на instance\_eval е в услуга за разрешаване на опростяване

фиден код за присвояване по следния начин:

david = Person.new do

име "Дейвид"

възраст 55

край

Това изглежда малко като че използваме аксесоари, само че няма изричен приемник и не

знаци за равенство. Как бихте накарали този код да работи?

Ето как може да изглежда класът Person:

клас Личност

def инициализиране (& блокиране)

instance\_eval (& block)

край

деф име (име = нула)

@ име || = име

край

def age (възраст = нула)

@age || = възраст

край

край

Методът instance\_exec

instance\_eval има близък братовчед, наречен instance\_exec. Разликата между

двете е, че instance\_exec може да приема аргументи. Всички аргументи, които предадете, ще го направят

да бъдат предадени от своя страна на кодовия блок.

Това ви позволява да правите неща като това:

string = "Примерен низ"

string.instance\_exec ("s") {| разделител | self.split (delim)}

(Не че ще трябва, ако вече знаете разделителя; но това е основната технология

nique.)

За съжаление кой метод е кой - кой от двамата взема аргументи и кой

не - просто трябва да се запомни. В термините eval или exec няма нищо

да ти помогна. И все пак е полезно да имате и двете под ръка.

**Изход: ["A",**

**"достатъчно", "тринг"]**

б

° С

д

|  |
| --- |
| **Страница 437** |

**437**

***Фамилното семейство методи***

Ключът тук е извикването на instance\_eval B , което използва повторно кодовия блок, който има

вече е предаден на нов. Тъй като кодовият блок е включен instance\_eval'd

обекта на новия човек (имплицитното аз в дефиницията за инициализиране), повикванията

за име и възраст се решават в рамките на класа Person. Тези методи от своя страна действат като

хибриден сетер / гетери c : те приемат незадължителен аргумент, който по подразбиране е нулев, и задават

съответните променливи на екземпляра, условно, към стойността на този аргумент. Ако ти

извикайте ги без аргумент, те просто връщат текущата стойност на своя екземпляр

променливи d .

Резултатът е, че можете да кажете име "Дейвид" вместо person.name = "Дейвид". Много

на рубистите намират този вид миниатюрен DSL (специфичен за домейна език) доста мотиви-

изключително компактен и елегантен.

instance\_eval (и instance\_exec) също с удоволствие ще вземе низ и ще го оцени

в превключен самостоятелен контекст. Тази техника обаче има същите подводни камъни като оценката

низове с eval и трябва да се използват разумно, ако изобщо се използват.

Последният член на семейството методи на eval е class\_eval (синоним:

module\_eval).

***14.4.4 Използване на class\_eval (известен още като module\_eval)***

По същество class\_eval ви поставя в тяло с дефиниция на клас:

c = Class.new

c.class\_eval правя

def some\_method

поставя "Създаден в class\_eval"

край

край

c\_instance = c.new

c\_instance.some\_method

Но можете да правите някои неща с class\_eval, които не можете да правите с обикновения клас

ключова дума:

Оценете низ в контекст на дефиниция на клас

Отворете дефиницията на клас за анонимен клас

Използвайте съществуващи локални променливи в тялото на определението на класа

Третият елемент от този списък е особено забележителен.

Когато отворите клас с ключовата дума class, стартирате нова локална променлива

обхват. Но блокът, който използвате с class\_eval, може да вижда променливите, създадени в

обхват около него. Погледнете разликата между лечението на var, an

локална променлива на външния обхват, в редовно тяло с дефиниция на клас и блок, даден на

class\_eval:

>> var = "инициализирана променлива"

=>"инициализирана променлива"

>> клас С

>> поставя var

>> край

**Резултат: Създаден**

**в class\_eval**

|  |
| --- |
| **Страница 438** |

**438**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

NameError: недефинирана локална променлива или метод `var 'за C: Class

от (irb): 3

>> C.class\_eval {поставя var}

инициализирана променлива

Променливата var е извън обхвата в стандартния блок за дефиниция на клас, но все още е в

обхват в кодовия блок, предаден на class\_eval.

Графиката се уплътнява, когато дефинирате метод на екземпляр в блока class\_eval:

>> C.class\_eval {def talk; поставя var; край }

=> нула

>> C.new.talk

NameError: недефинирана локална променлива или метод `var 'за # <C: 0x350ba4>

Както всеки def, def вътре в блока стартира нов обхват - така променливата var е no

по-дълго видими.

Ако искате да обуете променлива на външен обхват в метод на екземпляр, имате

да се използва различна техника за създаване на метода: методът define\_method. Вие

hand define\_method името на метода, който искате да създадете (като символ или

низ) и предоставят кодов блок; кодовият блок служи като тяло на метода.

За да получите външната променлива var в метод на екземпляр от клас C, правите това:

>> C.class\_eval {define\_method ("talk") {поставя var}}

=>: разговор

Връщаната стойност на define\_method е символ, представляващ името на ново

дефиниран метод.

В този момент методът на екземпляр на разговор на C ще има достъп до варианта на външния обхват

способен вар:

>> C.new.talk

инициализирана променлива

Няма да видите техники като този, използвани толкова често, колкото стандартните клас и метод-

техники за дефиниране. Но когато ги видите, ще разберете, че те означават сплеснатост

обхват за локални променливи, а не новия обхват, задействан от по-често срещаните

class и def ключови думи.

define\_method е метод на екземпляр на класа Module, така че можете да го извикате на всеки

екземпляр на модул или клас. По този начин можете да го използвате в обичайно тяло с дефиниция на клас

(където самоприемникът по подразбиране е обектът на класа), ако искате да скриете променлива

локално към тялото в метод на екземпляр. Това не е често срещана сцена

nario, но това не е нечувано.

Ruby ви позволява да правите леко паралелно програмиране с помощта на нишки. Ще разгледаме

нишки следващи.

***14.5 Паралелно изпълнение с нишки***

Нишките на Ruby ви позволяват да правите повече от едно нещо наведнъж във вашата програма, чрез

форма на споделяне на времето: една нишка изпълнява една или повече инструкции и след това преминава

контрол към следващата нишка и т.н. Как точно играе едновременността на нишките

|  |
| --- |
| **439 серия** |

**439**

***Паралелно изпълнение с нишки***

out зависи от вашата система и вашата Ruby реализация. Руби ще се опита да използва

местни съоръжения за резби на операционна система, но ако такива не са налични, ще го направи

се върнете към *зелени* нишки (нишки, изпълнени изцяло вътре в интерпретатора).

Тук ще поставим чек в полето за зелена срещу родна нишка; нашата грижа ще бъде

pally с техники за резби и синтаксис.

Създаването на нишки в Ruby е лесно: създавате инстанция за класа Thread. Нова нишка

започва да се изпълнява незабавно, но изпълнението на кода около нишката не го прави

Спри се. Ако програмата приключва, докато една или повече нишки се изпълняват, тези нишки

са убити.

Ето един пример отвътре навън, който ще ви помогне да започнете с теми чрез show-

да ви информирам как се държат, когато програмата приключи:

Thread.new правя

поставя "Стартиране на нишката"

сън 1

поставя "В края на нишката"

край

поставя "Извън нишката"

Thread.new взема кодов блок, който представлява изпълнимия код на нишката. В това

например нишката отпечатва съобщение, спи за една секунда и след това отпечатва друга

съобщение. Но извън нишката времето върви напред: основната част на програмата

отпечатва съобщение незабавно (това не се влияе от командата за заспиване в

и след това програмата завършва. Освен ако отпечатването на съобщение не отнеме повече от секунда

ond - в този случай трябва да проверите хардуера си! Второто съобщение

от нишката никога няма да се види. Ще видите само това:

Стартиране на нишката

Извън конеца

Сега, какво, ако искаме да позволим на нишката да завърши изпълнението? За да направим това, трябва да използваме

методът на екземпляра се присъедини. Най-лесният начин да използвате присъединяването е да запазите нишката в различна

може и извиква присъединяване на променливата. Ето как можете да промените предишния пример

по тези линии:

t = Thread.new do

поставя "Стартиране на нишката"

сън 1

поставя "В края на нишката"

край

поставя "Извън нишката"

присъединете се

Тази версия на програмата извежда следния изход с пауза от една секунда

между отпечатването на първото съобщение от нишката и отпечатването на

последно съобщение:

Стартиране на нишката

Извън конеца

В края на конеца

**Пауза като програма**

**чака конец да**

**завърши изпълнението**

|  |
| --- |
| **Страница 440** |

**440**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

В допълнение към присъединяването към нишка, можете да я манипулирате по различни други начини,

включително да го убиете, да го приспите, да го събудите и да го принудите да премине контрола

следващата нишка, планирана за изпълнение.

***14.5.1 Убиване, спиране и стартиране на нишки***

За да убиете нишка, вие й изпращате съобщението kill, exit или terminate; и трите са екви-

алент. Или, ако сте вътре в нишката, извиквате kill (или един от неговите синоними) в class-

метод на самата нишка.

Може да искате да убиете нишка, ако в нея възникне изключение. Ето пример,

несъмнено донякъде измислено, но достатъчно кратко, за да илюстрира ефективно процеса.

Идеята е да прочетете съдържанието на три файла (part00, part01 и part02) в

низ текст. Ако някой от файловете не е намерен, нишката завършва:

поставя „Опит за четене в някои файлове ...“

t = Thread.new do

(0..2) .всеки | n |

започнете

File.open ("part0 # {n}") do | f |

текст << f.readlines

край

спасяване Errno :: ENOENT

поставя „Съобщение от нишка: Неуспешно на n = # {n}“

Thread.exit

край

край

край

присъединете се

поставя "Готово!"

Резултатът, ако приемем, че part00 съществува, но part01 не съществува, е следният:

Опит за четене в някои файлове ...

Съобщение от нишка: Неуспешно на n = 1

Готово!

Можете също така да спрете и стартирате нишки и да проучите тяхното състояние. Конец може да спи или

буден и жив или мъртъв. Ето пример, който поставя конец през няколко от неговите

стъпва и илюстрира някои от наличните техники за изследване и манипулиране

състояние на резбата:

t = Thread.new do

поставя "[Начална нишка]"

Thread.stop

поставя "[Възобновяване на нишка]"

край

поставя "Състояние на нишката: # {t.status}"

поставя „Нишката спря ли? # {t.stop?}“

поставя "Жива ли е нишката? # {t.alive?}"

поставя

поставя „Събуждане на нишка и присъединяване към нея ...“

т. събуждане

присъединете се

**[Начална нишка]**

**Състояние на нишката:**

**сън**

**Е конец**

**спряна? вярно**

**Конецът жив ли е?**

**вярно**

**[Възобновяване на нишката]**

|  |
| --- |
| **Страница 441** |

**441**

***Паралелно изпълнение с нишки***

поставя

поставя "Жива ли е нишката? # {t.alive?}"

поставя „Проверете низ за нишка: # {t.inspect}“

Нека продължим да изследваме нишките с няколко мрежови примера: сървър за дати

и малко по-амбициозно - сървър за чат.

***14.5.2 Сребърен сървър за дати***

Сървърът за дати, който ще напишем, зависи от Ruby съоръжение, което все още не сме разглеждали:

TCPS сървър. TCPServer е клас, базиран на сокет, който ви позволява да стартирате почти сървър

невероятно лесно: създавате инстанция на класа и подавате номер на порт. Ето един прост

Фибри: Усукване на нишки

Освен нишки, Ruby има и клас Fiber. Влакната са като реентрантни кодови блокове:

те могат да се отстъпват назад и назад към своя призоваващ контекст няколко пъти.

Създава се влакно с конструктора Fiber.new, който взема кодов блок. Нищо

се случва, докато не кажете на влакното да се възобнови, в този момент кодовият блок започва да работи.

От блока можете да спрете влакното, връщайки контрола към извикващия

текст, с метода на класа Fiber.yield.

Ето един прост пример, включващ *говорещо* влакно, което алтернативно контролира няколко

пъти със своя контекст на извикване:

f = Fiber.нови правят

поставя „Здравей“.

Fiber. добив

поставя „Хубав ден“.

Fiber. добив

поставя "Чао!"

край

е. резюме

поставя „Назад към влакното:“

е. резюме

поставя „Последно съобщение от влакното:“

е. резюме

поставя "Това е всичко!"

Ето изхода от този фрагмент:

Здравей

Обратно към влакното:

Хубав ден.

Последно съобщение от влакното:

Чао!

Това е всичко!

Освен всичко друго, влакната са техническата основа на изброителите, които използват влакна

да приложат собствените си операции за спиране и стартиране.

**Конецът жив ли е? невярно**

**Проверете низ за нишка:**

**# <Тема: 0x28d20 мъртъв>**

|  |
| --- |
| **442 серия** |

**442**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

пример за TCPServer в действие, обслужващ текущата дата на първия човек, който

свързва се с него:

изискват 'сокет'

s = TCPServer.new (3939)

conn = s.приемам

conn.puts "Здравей. Ето датата."

conn.puts `дата`

свързване.затваряне

s.close

Поставете този пример във файл, наречен dateserver.rb, и го стартирайте от командния ред. (Ако

порт 3939 не е наличен, сменете номера на нещо друго.) Сега, от различен-

ent конзола, свържете се към сървъра:

3939

Ще видите изход, подобен на следния:

Опитва се 127.0.0.1 ...

Свързан с localhost.

Символът за бягство е '^]'.

Здравей Ето датата.

Събота, 18 януари 07:29:11 EST 2014

Връзката е затворена от чужд хост.

Сървърът е изпратил заявката и е отговорил.

Какво ще стане, ако искате сървърът да изпрати множество заявки? Лесно: не затваряйте гнездото,

и продължавайте да приемате връзки.

изискват 'сокет'

s = TCPServer.new (3939)

докато е вярно

conn = s.приемам

conn.puts "Здравей. Ето датата."

conn.puts `дата`

свързване.затваряне

край

Сега можете да поискате датата повече от веднъж и всеки път ще получите отговор.

Нещата стават по-сложни, когато искате да изпратите информация *до* сървъра. Правя го

работата за един потребител е ясна; сървърът може да приеме въвеждане чрез извикване получава:

изискват 'сокет'

s = TCPServer.new (3939)

докато е вярно

conn = s.приемам

conn.print "Здравей. Как се казваш?"

name = conn.gets.chomp

conn.puts "Здравей, # {name}. Ето датата."

conn.puts `дата`

свързване.затваряне

край

**дата в обратни отметки се изпълнява**

**командата за системна дата**

**Приема ред на клавиатурата**

**вход от клиента**

|  |
| --- |
| **Страница 443** |

**443**

***Паралелно изпълнение с нишки***

Но ако втори клиент се свърже със сървъра, докато сървърът все още чака първия

клиентски вход, вторият клиент не вижда нищо - дори как се казваш? - защото

сървърът е зает.

Тук идва резбата. Ето сървър за дати с резба, който приема въвеждане

от клиента. Нишките предотвратяват блокирането на цялото приложение, докато то е

чака един клиент да предостави вход:

изискват 'сокет'

s = TCPServer.new (3939)

докато (conn = s.accept)

Thread.new (conn) do | c |

c.print "Здравей. Как се казваш?"

name = c.gets.chomp

c.puts "Здравей, # {name}. Ето датата."

в. извежда `дата`

в. затвори

край

край

В тази версия, сървърът слуша непрекъснато за връзки B . Всеки път, когато получава

един, той поражда нова нишка c . Значението на аргумента за Thread.new е

че ако предоставите такъв аргумент, той ще ви бъде върнат като параметър на блока.

В този случай това означава обвързване на връзката с параметъра c. Въпреки че това

техниката може да изглежда странно (изпращане на аргумент към метод, само за да го върне, когато

блокът се нарича), той гарантира, че всяка нишка има препратка към собствената си връзка

вместо да се биете за променливата conn, която живее извън всяка нишка.

Дори ако даден клиент чака няколко минути, преди да въведе име d ,

сървърът все още слуша нови връзки и новите нишки все още се раждат. The

подходът с резби позволява на сървъра да мащабира, като същевременно включва двупосочно предаване

сион между себе си и един или повече клиенти.

Следващото ниво на сложност е чат сървърът.

***14.5.3 Писане на чат сървър с помощта на сокети и нишки***

Този път ще започнем първо с код. Листинг 14.2 показва кода на чат-сървъра. Много от това

does е подобно на това, което прави сървърът за дати. Основната разлика е, че сървърът за чат

поддържа списък (масив) от всички входящи връзки и използва този списък за излъчване

входящите съобщения в чата.

изискват 'сокет'

def добре дошли (бърборене)

chatter.print "Добре дошли! Моля, въведете името си:"

chatter.readline.chomp

край

def излъчване (съобщение, бърборене)

чат.всеки | бърборене |

chatter.puts съобщение

край

край

Листинг 14.2 Чат сървър, използващ **TCPServer** и нишки

б

° С

д

б

° С

д

|  |
| --- |
| **Страница 444** |

**444**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

s = TCPServer.new (3939)

бърборене = []

докато (бърборене = s.accept)

Thread.new (бърборене) do | c |

име = добре дошли (бърборене)

излъчване ("# {name} се присъедини", бърборене)

бърборене << бърборене

започнете

цикъл правя

линия = c.readline

излъчване ("# {name}: # {line}", бърборене)

край

спасяване EOFError

в. затвори

chatters.delete (c)

излъчване („# {name} е напуснало“, бърборене)

край

край

край

В този списък има много код, така че ще го вземем в реда, в който се изпълнява. Първо идва

задължителната товаренето на гнездо библиотека на B . Следващите няколко реда определят някои

необходими помощни методи; ще се върнем към тези, след като видим за какво помагат-

инжектиране с. Истинското начало на действието е създаването на TCPServer и

инициализация на масива от бърборене д .

Сървърът отива в линия, докато подобно на линия в срок сървъра на F . Когато

бърборене свързва, сървърът го приветства (него или нея, наистина, но *тя* ще направи) G . Добре

Процесът come включва метода за приветствие C , който приема бърборене - обект на сокет -

като свой аргумент отпечатва хубаво приветствено съобщение и връща ред от клиентски вход. Сега

време е да уведомите всички настоящи бърборещи, че е пристигнал нов разговор. Това включва

метод на излъчване D , който е сърцевината на функцията за чат на програмата: това е

отговорен за преминаването през редица бърборене и изпращане на съобщение до всеки от тях.

В този случай съобщението гласи, че новият клиент се е присъединил към чата.

След като бъде обявен, новият бърборене се добавя към масива на бъбривите. Това означава

тя ще бъде включена в бъдещи предавания на съобщения.

Сега идва частта за чат. Състои се от безкраен цикъл, увит в начало /

спасителна клауза Н . Целта е да приемате съобщения от този клиент завинаги, но да приемате

действие, ако клиентският сокет отчита края на файла. Съобщенията се приемат чрез четене I ,

който има предимството пред получава (при всяка ситуация в тази ситуация), че създава

в края на файла. Ако бърборенето напусне чата, следващият опит за четене на ред

от това бърборене се повдига EOFError. Когато това се случи, контролът отива на помощ

блок J , където напусналият бърборене се отстранява от масива на бъбривите и

известието се излъчва в смисъл, че бъбривите са напуснали 1) .

Ако няма EOFError, съобщение тракане се излъчва на всички чатъри I .

Когато използвате нишки, е важно да знаете как действат правилата за обхват на променливи и

видимостта се играе вътре в нишките - и в разглеждането на тази тема, която ще направим по-нататък,

ще разберете и за специална категория специфични за нишки променливи.

д

е

ж

з

i

j

1)

|  |
| --- |
| **Страница 445** |

**445**

***Паралелно изпълнение с нишки***

***14.5.4 Нишки и променливи***

Нишките се изпълняват с помощта на кодови блокове и кодовите блокове могат да виждат вече създадените променливи

в техния локален обхват. Ако създадете локална променлива и я промените в кода на нишката

блок, промяната ще бъде постоянна:

>> a = 1

=> 1

>> Thread.new {a = 2}

=> # <Тема: 0x390d8c стартиране>

>> а

=> 2

Можете да видите интересен и поучителен ефект, ако спрете нишка, преди тя да се промени

променлива и след това изпълнете нишката:

>> t = Thread.new {Thread.stop; a = 3}

=> # <Тема: 0x3e443c стартиране>

>> а

=> 2

>> t.run

=> # <Тема: 0x3e443c мъртъв>

>> а

=> 3

Глобалните променливи остават глобални, в по-голямата си част, в лицето на нишките. Това върви

за вградени глобални, като $ / (разделител на входния запис), както и тези, които създавате

изяде се:

>> $ /

=>"\ n"

>> $ var = 1

=> 1

>> Thread.new {$ var = 2; $ / = "\ n \ n"}

=> # <Тема: 0x38dbb4 стартиране>

>> $ /

=>"\ n \ n"

>> $ var

=> 2

Но някои глобали са глобални *локални нишки - по-* специално, $ 1, $ 2, ..., $ *n,* които са присвоени

стойностите на скобите от най-новите регулярни изрази - съвпадение

операция. Получавате различна доза от тези променливи във всяка нишка. Ето фрагмент

което илюстрира факта, че променливите $ *n* в различни нишки не се сблъскват:

/(abc)/.match("abc ")

t = Thread.new do

/(def)/.match("def ")

поставя „$ 1 в нишка: # {$ 1}“

край.присъединете се

поставя „$ 1 извън нишката: # {$ 1}“

Обосновката за това поведение е ясна: не можете да имате представа на една нишка за над 1 $

засенчване на $ 1 от различна нишка, или ще получите изключително странни резултати. The

**Изход: $ 1 инч**

**нишка: def**

**Изход: $ 1 отвън**

**резба: abc**

|  |
| --- |
| **446 серия** |

**446**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

$ *n* променливите всъщност не са глобални, след като ги видите в контекста на езика

като има конци.

Освен че имат достъп до обичайния набор от променливи на Ruby, темите също имат

собствена променлива скривалище - или, по-точно, вграден хеш, който им позволява да се асоциират

символи или низове със стойности. Тези ключове за нишки могат да бъдат полезни.

***14.5.5 Манипулиране на ключове за резби***

*Ключовете за нишки* са основно хеш за съхранение на стойности, специфични за нишката. Ключовете трябва да са

символи или низове. Можете да стигнете до ключовете, като индексирате обекта на нишката директно с

стойности в квадратни скоби. Можете също така да получите списък с всички ключове (без техните стойности)

използвайки метода на ключовете.

Ето един прост сценарий за задаване и получаване, използващ ключ на нишка:

t = Thread.new do

Thread.current [: message] = "Здравейте"

край

присъединете се

p т. клавиши

поставя t [: съобщение]

Изходът е

[: съобщение]

Здравейте

Изглежда, че нишките изглеждат големи в игрите, така че нека използваме пример за игра, за да изследваме нишката

допълнителни ключове: игра с резба, мрежа, скала / хартия / ножица ( RPS ). Ще започнем с

(без резба) RPS логика в клас RPS и използвайте получената RPS библиотека като основа

за кода на играта.

А BASIC ROCK / хартия / ножица LOGIC ИЗПЪЛНЕНИЕ

Следващият списък показва клас RPS, който е обвит в модул Games (защото

RPS звучи така, сякаш може да се сблъска с друго име на клас). Запазете този списък във файл

наречен rps.rb.

модул Игри

клас RPS

включват Сравним

ПОБЕДИ = [% w {ножици за скала},

% w {ножица хартия},

% w {хартия}}

attr\_accessor: преместване

def инициализиране (преместване)

@move = move.to\_s

край

def <=> (друго)

ако преместите == друго.движете се

0

Листинг 14.3 RPS логика на играта, въплътена в **Games :: RPS** клас

б

° С

д

д

е

|  |
| --- |
| **Страница 447** |

**447**

***Паралелно изпълнение с нишки***

elsif WINS.include? ([move, other.move])

1

elsif WINS.include? ([other.move, move])

-1

друго

повиши ArgumentError, "Нещо не е наред"

край

край

def play (друго)

ако себе си> други

себе си

elsif друго> себе си

други

друго

невярно

край

край

край

край

Класът RPS включва Сравним модул B ; това служи като основа за определяне

в крайна сметка кой печели играта. Константата WINS съдържа всички възможни печеливши

бинации в три масива; първият елемент във всеки масив бие втория елемент C .

Освен това има и ход атрибут, който съхранява в движение за тази инстанция на RPS D . The

инициализираният метод E съхранява преместването като низ (в случай че се появи като символ).

RPS има оператор кораб (<=>) определение метод е , което определя какво-щастлив

писалки, когато този екземпляр на RPS се сравнява с друг екземпляр. Ако двамата имат

равни ходове, резултатът е 0 - сигналът, че двата термина на сравнение на космически кораб

са равни. Останалата част от логиката търси печеливши комбинации с помощта на WINS масива,

връщане -1 или 1 в зависимост от това дали този или другият екземпляр има

Спечелени. Ако не установи, че някой от играчите има победа и резултатът не е равен, той вдига

изключение.

Сега, когато RPS обектите знаят как да се сравняват, е лесно да ги играете

един срещу друг, което прави методът на възпроизвеждане g . Това е просто: в зависимост от това

играчът е по-висок е победителят и ако е равен, методът връща false.

Вече сме готови да включим RPS класа в резбована, мрежова версия на

играта, ключовете за нишки и всичко останало.

U пее RPS класа в резбован ИГРА

Следният списък показва мрежовата RPS програма. Изчаква двама души

присъединете се, получава ходовете си, отчита резултата и излиза. Не лъскав - но добър начин да се види

как ключовете за нишки могат да ви помогнат.

изискват 'сокет'

require\_relative 'rps'

s = TCPServer.new (3939)

теми = []

Листинг 14.4 Нишка, свързана в мрежа RPS програма, използваща нишкови ключове

ж

б

° С

|  |
| --- |
| **Страница 448** |

**448**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

2. времена | n |

conn = s.приемам

теми << Thread.new (conn) do | c |

Thread.current [: number] = n + 1

Thread.current [: player] = c

c.puts "Добре дошли, играч # {n + 1}!"

c.print "Вашият ход? (камък, хартия, ножици)"

Thread.current [: move] = c.gets.chomp

c.puts "Благодаря ... изчакайте."

край

край

a, b = нишки

а.присъединете се

b.присъединете се

rps1, rps2 = Игри :: RPS.new (a [: преместване]), Игри :: RPS.new (b [: преместване])

победител = rps1.play (rps2)

ако победител

резултат = победител.движете се

друго

резултат = "ВРЪЗКА!"

край

нишки.всеки | t |

t [: player] .puts "Победителят е # {резултат}!"

край

Тази програма зарежда и използва класа Games :: RPS, така че се уверете, че имате RPS

код във файла rps.rb в същата директория като самата програма.

Както в примера за сървър за чат, започваме със сървър B заедно с масив, в който

нишките се съхраняват c . Вместо да зациклим завинаги, ние събираме само две нишки,

с любезното съдействие на цикъла 2.times и метода за приемане на сървъра d . За всеки от двете

връзки, ние създаваме нишка e .

Сега съхраняваме някои стойности в ключовете на нишката: число за този плейър (базирано на

цикъла на време, добавяйки 1, така че да няма играч 0) и връзката. След това ние

елате на плейъра и съхранявайте хода в клавиша: move на нишката.

След като и двамата играчи са играли, хващаме двете нишки в удобния вариант

ables a и b и се присъединяват към двете нишки f . След това обменяме обектите с нишки, които

имат памет за движенията на играчите в два RPS обекта g . Победителят се определя

като играете един срещу друг. Крайният резултат от играта е или победителят, или, ако

играта се върна фалшиво, равен h .

Накрая докладваме резултатите и на двамата играчи i . Можете да получите по-любител, като въведете

техните имена или повтаряне на играта и водене на резултат. Но основната точка на тази версия

Сионът на играта е да илюстрира полезността на нишките. Дори и след нишките

приключи, те запомнят информация и това ни позволява да играем

цяла игра, както и да изпращате допълнителни съобщения през гнездата на играчите.

В края на нашия поглед към нишките на Ruby. Струва си да се отбележи, че нишките са

област, която е претърпяла и продължава да претърпява много промени и развитие.

Но каквото и да се случи, можете да надграждате земята, която сте получили тук, както и вие

изследвайте и използвайте теми по-нататък.

д

д

е

**Използвайте паралелно задание**

**синтаксис за присвояване на две**

**променливи от масив**

ж

з

i

|  |
| --- |
| **Страница 449** |

**449**

***Издаване на системни команди от вътрешни програми на Ruby***

Следващата в дневния ред и последна за тази глава е темата за издаване на системни

Mands от Руби.

***14.6 Издаване на системни команди от вътрешни програми на Ruby***

Можете да издавате системни команди по няколко начина в Ruby. Ще разгледаме предимно две от

тях: системният метод и техниката „(обратни връзки). Другите начини за ком-

комуникират със системни програми включват малко по-ниско ниво на програмиране и са

по-зависими от системата и следователно донякъде извън обхвата на тази книга. Добре

все пак ги разгледайте накратко и ако изглежда, че са нещо, от което се нуждаете, вие

може да ги изследва по-нататък.

***14.6.1 Системният метод и обратни връзки***

Системният метод извиква системна програма. Backticks (") извиква системна програма и

върнете изхода му. Изборът зависи от това какво искате да направите.

E XECUTING СИСТЕМА ПРОГРАМИ с метода СИСТЕМА

За да използвате системата, изпратете й името на програмата, която искате да стартирате, с всякакви аргументи.

Програмата използва текущите STDIN, STDOUT и STDERR. Ето три прости

примери. cat и grep изискват натискане на Ctrl-d (или каквото и да е включен бутонът „в края на файла“

вашата система), за да ги прекратите и да върнете контрола на irb. За по-голяма яснота изходът на Ruby е

с получер шрифт и въвеждането от потребителя е с обикновен шрифт:

**>>** система ("дата")

**Събота, 18 януари 07:32:11 EST 2014**

**=> вярно**

**>>** система ("котка")

Пиша на екрана за командата на котката.

**Пиша на екрана за командата на котката.**

**=> вярно**

**>>** система ('grep "D"')

един

две

Дейвид

**Дейвид**

Когато използвате система, глобалната променлива $? е зададен на обект Process :: Status, който

съдържа информация за повикването: по-специално, идентификатора на процеса на процеса, който току-що

run и състоянието му на излизане. Ето обаждане до днес и едно за котка, последното прекратено с

Ctrl-c. Всеки от тях е последван от проверка на $ ?:

>> система ("дата")

Събота, 18 януари 07:32:11 EST 2014

=> вярно

>> $?

=> # <Процес :: Състояние: pid 28025 изход 0>

>> система ("котка")

^ C => невярно

>> $?

=> # <Процес :: Състояние: pid 28026 SIGINT (сигнал 2)>

|  |
| --- |
| **Страница 450** |

**450**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

И ето призив към несъществуваща програма:

>> система ("datee")

=> нула

>> $?

=> # <Процес :: Състояние: pid 28037 изход 127>

$? променливата е локална за нишки: ако извикате програма в една нишка, връщаната й стойност

засяга само $? в тази тема:

>> система ("дата")

Събота, 18 януари 07:32:11 EST 2014

=> вярно

>> $?

=> # <Процес :: Състояние: pid 28046 изход 0>

>> Thread.new {система ("datee"); p $? }.присъединяване

# <Процес :: Състояние: pid 28047 изход 127>

=> # <Тема: 0x3af840 мъртъв>

>> $?

=> # <Процес :: Състояние: pid 28046 изход 0>

Процесът :: Съобщението за обект на състоянието за обаждането до момента се съхранява в $? в основното

резба B . Новата нишка извиква несъществуваща програма c и тази нишка

версия на $? отразява проблема d . Но основната нишка е $? е непроменен e . The

Поведението на глобалната променлива на нишки-локали работи подобно на случая с $ *n*

регулярни изрази за улавяне на променливи - и по подобни причини. И в двата случая не го правите

искам една нишка да реагира на състояние на грешка, което не е причинило и което не

отразяват действителния му програмен поток.

Техниката на беккит е близък роднина на системата.

C ПРОГРАМИ ЗА СИГНАЛНА СИСТЕМА С ОБРАТНИ ЗАДЪЛЖЕНИЯ

За да издадете системна команда с обратни връзки, поставете командата между обратни връзки. The

основната разлика между системата и обратните отметки е, че възвръщаемата стойност на обратния отметка

call е изходът на програмата, която стартирате:

>> d = `дата`

=>"Събота 18 януари 07:32:11 EST 2014 \ n"

>> поставя d

Събота, 18 януари 07:32:11 EST 2014

=> нула

>> изход = `котка`

Пиша в котка. Тъй като използвам обратни връзки,

Няма да виждам всеки ред да се ехови, докато го въвеждам.

Вместо това изходът на котката отива в

променлива продукция.

=>"Пиша в котка. Тъй като използвам обратни връзки, \ nНе искам и *т.н.*

>> поставя изход

Пиша в котка. Тъй като използвам обратни връзки,

Няма да виждам всеки ред да се ехови, докато го въвеждам.

Вместо това изходът на котката отива в

променлива продукция.

б

° С

д

д

|  |
| --- |
| **Страница 451** |

**451**

***Издаване на системни команди от вътрешни програми на Ruby***

Обратните отметки задават $? точно както прави системата. Извикване на несъществуващ метод с обратни връзки

поражда фатална грешка:

>> `Datee`

Errno :: ENOENT: Няма такъв файл или директория - datee

>> $?

=> # <Процес :: Състояние: pid 28094 изход 127>

>> `дата`

=>"Събота 18 януари 07:35:32 EST 2014 \ n"

>> $?

=> # <Процес :: Състояние: pid 28095 изход 0>

Обратните обратни връзки са изключително полезни за улавяне на външен изход на програмата, но не са

единственият начин да го направя. Това ни води до третия начин за стартиране на програми отвътре

програма Ruby: отворена и Open.popen3.

***14.6.2 Комуникация с програми чрез open и popen3***

Използването на отвореното семейство методи за извикване на външни програми е много по-сложно

отколкото използването на система и обратни връзки. Ще разгледаме няколко прости примера, но няма

отвес дълбочините на темата. Тези методи на Ruby се свързват директно с основния

системни библиотеки, които ги поддържат и точното им поведение може да варира в зависимост от системата

tem към друг повече, отколкото повечето Ruby поведение прави.

И все пак - нека да разгледаме. Ще обсъдим два метода: отворен и метод на класа

Open.popen3.

T ALKING към външни програми с OPEN

Можете да използвате отворения метод от най-високо ниво, за да осъществите двупосочна комуникация с външен

програма. Ето стария пример за готовност на котка:

>> d = отворен ("| котка", "w +")

=> # <IO: fd 11>

>> d.puts "Здравей на котка"

=> нула

Някои системни командни звънци и свирки

Има още един начин за изпълнение на системни команди от Ruby:% x oper-

ator. % x {date}, например, ще изпълни командата date. Подобно на обратните отметки,% x

връща низовия изход на командата. Подобно на своите роднини% w и% q (сред другите-

ers),% x позволява всеки разделител, стига разделителите в стил скоби да съвпадат:% x {date},

% x-date- и% x (date) са синоними.

И обратните отметки, и% x позволяват интерполация на низове:

command = "дата"

% x (# {команда})

Това може да е удобно, въпреки че случаите, в които е добре да се обадите

динамично оценяваните низове като системни команди, може би са малко.

б

° С

|  |
| --- |
| **452 серия** |

**452**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

>> d.gets

=>"Здравей на котка \ n"

>> d.close

=> нула

Поканата за отваряне е обща; това може да бъде всеки I / O поток, но в този случай това е двупосочно

връзка към система команда B . Тръбата пред думата котка показва

че искаме да говорим с програма и да не отваряме файл. Дръжката на

външната програма работи подобно на I / O сокет или манипулатор на файл. Отворена е за четене

писане и писане (режимът w +), за да можем да пишем в него c и да четем от него d . И накрая,

ние го затваряме д .

Също така е възможно да се възползвате от блоковата форма на open и да запазите последната стъпка:

>> отворено ("| котка", "w +") {| p | p.puts ("здравей"); p.gets}

=>"здравей \ n"

Малко по-сложен и мощен начин за осъществяване на двупосочна комуникация

между вашата програма Ruby и външна програма е методът Open3.popen3.

W WO - WAY КОМУНИКАЦИЯ С O PEN 3. POPEN 3

Методът Open3.popen3 отваря комуникация с външна програма и

ви дава манипулатори на стандартния вход, стандартен изход на външната програма и

стандартни потоци за грешки. По този начин можете да пишете и да четете от тези дръжки отделно

от аналогичните потоци във вашата програма.

Ето един прост пример за Open.popen3, базиран на котки:

>> изисква 'open3'

=> вярно

>> stdin, stdout, stderr = Open3.popen3 ("котка")

=> [# <IO: fd 10>, # <IO: fd 11>, # <IO: fd 13>,

# <Тема: 0x000001011356f8 сън>]

>> stdin.puts ("Здравей. \ nДа")

=> нула

>> stdout.gets

=>"Здравей. \ N"

>> stdout.gets

=>"Чао \ n"

След зареждане на библиотеката open3 B , ние се обаждаме на Open3.popen3, като я предаваме

име на външната програма c . Връщаме три I / O дръжки и нишка d .

(Можете да игнорирате нишката.) Тези I / O дръжки влизат и излизат от външния про-

грам. По този начин можем да пишем на STDIN дръжката e и да четем редове от STDOUT han-

dle f . Тези дръжки не са същите като потоците STDIN и STDOUT на irb

самата сесия.

Следващият пример показва малко по-сложно използване на Open.popen3. Бъди предупреден:

само по себе си е тривиално. Неговата цел е да илюстрира някои от основните механики на технологията

nique - и използва нишки, така че преилюстрира и някои техники за нишки. Следването-

списъкът показва кода.

д

д

б

° С

д

д

е

|  |
| --- |
| **Страница 453** |

**453**

***Издаване на системни команди от вътрешни програми на Ruby***

изисква 'open3'

stdin, stdout, stderr = Open3.popen3 ("котка")

t = Thread.new do

цикъл {stdin.puts получава}

край

u = Thread.new do

n = 0

str = ""

цикъл правя

str << stdout.gets

n + = 1

ако n% 3 == 0

поставя "-------- \ n"

поставя ул

поставя "-------- \ n"

str = ""

край

край

край

присъединете се

присъединете се

Програмата отваря двупосочна тръба за котка и използва две нишки, за да говори и слуша

тази тръба. Първата нишка, t B , се завърта завинаги, слушайки STDIN - вашият STDIN, не

cat's - и записване на всеки ред в манипулатора STDIN в процеса на котка. Секундата

thread, u c , поддържа брояч (n) и струнен акумулатор (str). Когато

ter удря кратно на 3, както е посочено от модулния тест d , u нишката разпечатва a

хоризонтален ред, трите реда текст, които са натрупани до момента, и още един хоризонтален

линия. След това нулира струнния акумулатор до празен низ и се връща към прослушването.

Ако стартирате тази програма, не забравяйте, че тя се цикли завинаги, така че ще трябва да прекъснете

го с Ctrl-c (или каквото и да е вашата система използва за сигнал за прекъсване). Изходът е, предварително

по диктат, донякъде невъзбуждащо, но ви дава добро, пряко усещане за това, как нишките

си взаимодействат с входящите и изходните манипулатори на вход / изход и помежду си. В този изход,

редовете, въведени от потребителя, са в курсив:

*Едно*

*Две*

*Три*

--------

Едно

Две

Три

--------

*Четири*

*Пет*

*Шест*

--------

Четири

Пет

Шест

--------

Листинг 14.5 Използване на **Open.popen3** и нишки за манипулиране на **котешка** сесия

б

° С

д

|  |
| --- |
| **Страница 454** |

**454**

C ГЛАВА 14 ***Извикващи се и изпълними обекти***

Както беше посочено, няма да навлизаме във всички подробности на Open.popen3. Но можете и

трябва да го имате предвид при ситуации, в които имате нужда от най-голяма гъвкавост при четене

от и писане във външна програма.

***14.7 Обобщение***

В тази глава сте виждали

Proc обекти

Ламбда „вкусът“ на процеса

Кодиране на преобразуване на блок в процес (и обратно)

Символ # до\_процес

Обект на метода

Обвързвания

eval, instance\_eval и class\_eval

Използване на нишки и манипулация

Локални „глобални“ променливи в нишки

Системният метод

Извикване на системни команди с обратни връзки

Основите на съоръженията open и Open.popen3

Обектите в Ruby са продукти на изпълнение на кода по време на изпълнение, но сами могат да имат

правомощия за изпълнение на код. В тази глава разгледахме редица начини, по които

общото понятие за извикващи се и изпълними обекти се играе. Разгледахме обекти на Proc

и ламбда, анонимните функции, които лежат в основата на блоковия синтаксис на Руби. Ние

също обсъди методи като обекти и начини за развързване и обвързване методи и

третирането им отделно от обектите, които ги наричат. Фамилното семейство методи

ни отведе в сферата на изпълнението на произволни низове и също показа някои мощни

и елегантни техники за манипулиране по време на изпълнение на обекта и класа на програмата

пейзаж, използвайки не само eval, но дори повече, class\_eval и instance\_eval с

техните блокови операции.

Нишките са на видно място сред изпълнимите обекти на Ruby; всяка програма се изпълнява

в основна нишка, дори ако не хвърля хайвера си други. Проучихме синтаксиса и семантиката на

нишки и видя как те улесняват проекти като мрежова комуникация за много потребители.

Накрая разгледахме различни начини, по които Ruby ви позволява да изпълнявате външни програми

грамове, включително относително опростения системен метод и техниката на обратен удар, и

малко по-гранулираните и сложни отворени и Open.popen3 съоръжения.

Няма конкретна дефиниция на извикваем или изпълним обект и тази глава

умишлено е подходил доста плавно към разбирането на термините. На един

от друга страна, тази плавност води до съпоставяне на теми, които въображаемо биха могли да бъдат развлечени

разгледани в отделни глави. (Трудно е да се спори за някакво пряко, близко родство между, да речем,

instance\_eval и Open.popen3.) От друга страна, спецификата на Ruby е към a

до голяма степен, прояви на основните принципи и супервенцията и идеята за

|  |
| --- |
| **Страница 455** |

**455**

***Обобщение***

обекти, които участват пряко в динамиката на пейзажа на Руби е важно.

Независимо, че те могат да бъдат в някои отношения, темите в тази глава се подреждат

себе си с този принцип; и доброто заземяване в тях ще допринесе значително за

вашите способности на Ruby.

На този етап ще се обърнем към следващата - и последна - основна тема: размисъл по време на изпълнение,

самоанализ и обратно извикване.

|  |
| --- |
| **456 серия** |

**456**

*Обратни обаждания, куки,*

*и интроспекция по време на работа*

В съответствие с динамичния си характер и насърчаването на гъвкавост, гъвкавост

дизайн на обекти и програми, Ruby предоставя голям брой начини за изследване

какво се случва, докато програмата ви работи и да настроите обратно извикване на събития

и куки - по същество трипъри, които се изтеглят в определено време и за конкретно

причини - под формата на методи със специални, запазени имена, за които можете, ако

пожелайте, предоставете определения. По този начин можете да настроите модул, така че даден

метод се извиква всеки път, когато клас включва този модул, или напишете обратно повикване

метод за клас, който се извиква всеки път, когато класът се наследи и т.н.

***Тази глава обхваща***

Обратни обаждания по време на изпълнение: наследени, включени,

и още

Отговорът\_то? и метод\_пропускане

методи

Самоанализ на списъци с обекти и методи на клас

Прихващане на неразрешени постоянни препратки

Изследване на променливи и константи в обхвата

Анализиране на информация за проследяване на повикващия и стека

|  |
| --- |
| **Страница 457** |

**457**

***Обратни обаждания и куки***

В допълнение към обратните обаждания по време на изпълнение, Ruby ви позволява да изпълнявате по-пасивни, но често

критични актове на изследване: можете да попитате обектите какви методи могат да изпълнят (в

дори повече начини, отколкото вече сте виждали) или какви променливи на екземпляра имат. Вие

могат да търсят класове и модули за техните константи и техните методи на екземпляр. Можеш

изследвайте проследяване на стека, за да определите кой метод ви извиква до определена точка

във вашата програма - и дори получавате достъп до имената на файловете и номерата на редовете на всички

методът извиква по пътя.

Накратко, Руби ви кани на партито: можете да видите какво се случва, значително

детайли, чрез техники за интроспекция по време на изпълнение; и можете да наредите на Ruby да натисне някои

бутони в отговор на събития по време на изпълнение. Тази глава, последната в книгата, ще разгледа различна

на тези интроспективни техники и техники за обратно извикване и ще ви подготвят да приемате все по-големи

предимството на съоръженията, предлагани от този забележителен и изключително динамичен език.

***15.1 Обратни обаждания и куки***

Използването на *обратно извикване* и *куки* е доста често срещана техника за метапрограмиране *.* Тези

се извикват, когато дадено събитие се случи по време на изпълнението на Ruby

грам. Събитието е нещо като

Несъществуващ метод, извикан за обект

Модул, който се смесва с клас или друг модул

Обект, който се разширява с модул

Клас, който се подкласира (наследява се от)

Прави се препратка към несъществуваща константа

Метод на екземпляр, който се добавя към клас

Единичен метод, който се добавя към обект

За всяко събитие в този списък можете (ако решите) да напишете метод за обратно извикване, който ще бъде

изпълнява се, когато се случи събитието. Тези методи за обратно извикване са по обект или по клас,

не е глобален; ако искате метод, извикан, когато билетът за клас се подкласира, вие

трябва да напишете подходящия метод специално за билет за клас.

Това, което следва, са описания на всяка от тези куки за изпълнение по време на изпълнение. Ще разгледаме

ги в реда, посочен по-горе.

***15.1.1 Прихващане на неразпознати съобщения с method\_missing***

Обратно в глава 4 (раздел 4.3) научихте доста много за method\_missing. Да обобщим-

marize: когато изпращате съобщение до обект, обектът изпълнява първия метод

намира в пътя си за търсене на метод със същото име като съобщението. Ако не успее да намери

всеки такъв метод, той поражда изключение NoMethodError - освен ако не сте предоставили

обект с метод, наречен method\_missing. (Вижте раздел 4.3, ако искате

освежете паметта си за това как работи method\_missing.)

Разбира се, method\_missing също заслужава място в тази глава, защото може да се каже

най-често използваното време за изпълнение в Ruby. Вместо да повтаря корицата на глава 4 -

възраст обаче, нека разгледаме няколко специфични метода\_пропускащи нюанси. Ще помислим

|  |
| --- |
| **Страница 458** |

**458**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

използване на method\_missing като техника на делегиране; и ще разгледаме как method\_missing

работи и какво се случва, когато го замените, в горната част на йерархията на класа.

D ELEGATING с метода \_ ЛИПСВА

Можете да използвате method\_missing, за да постигнете автоматично удължаване на начина, по който вашият

обектът се държи. Да приемем например, че моделирате обект, който в някои отношения е

контейнер, но който има и други характеристики - може би готварска книга. Искаш да бъдеш

може да програмира вашата готварска книга като колекция от рецепти, но има и определени характеристики

теристика (заглавие, автор, може би списък с хора, с които сте го споделили или които са

допринесе за това), които трябва да се съхраняват и обработват отделно от рецептите. Поради това

готварската книга е едновременно колекция и хранилище на метаданни за колекцията.

За да направите това по метод, основан на пропускане, ще поддържате набор от рецепти

и след това препраща всички неразпознати съобщения към този масив. Просто изпълнение

може да изглежда така:

клас Кулинарна книга

attr\_accessor: заглавие,: автор

def инициализира

@recipes = []

край

def method\_missing (m, \* args, & block)

@ recipes.send (m, \* аргументи и блокиране)

край

край

Сега можем да извършваме манипулации върху колекцията от рецепти, като се възползваме

всички масивни методи, които желаем. Да приемем, че има клас на рецепта, отделен от

Клас Cookbook и вече създадохме някои обекти за рецепти:

cb = Cookbook.new

cb << рецепта\_за\_кейк

cb << рецепта\_за\_пиле

beef\_dishes = cb.select {| рецепти | recipe.main\_ingredient == "говеждо месо"}

Екземплярът на готварската книга, cb, няма методи, наречени << и select, така че тези съобщения

мъдреците се предават на масива @recipes с любезното съдействие на method\_missing. Все още можем

дефинирайте всички методи, които искаме директно в класа Cookbook - можем дори да заменим

масивни методи, ако искаме по-специфично поведение за готварска книга за някой от тези методи

ods - но method\_missing ни спестява от необходимостта да дефинираме паралелен набор от методи за

обработка на страници като поръчана колекция.

Техники за делегиране на методи на Руби

В този пример за избягване на метода сме *делегирали* обработката на съобщения (

неизвестни) към масива @recipes. Ruby има няколко механизма за делегиране

действия от един обект на друг. Тук няма да влизаме в тях, но вие може да дойдете

както в клас Delegator, така и в клас SimpleDelegator във вашето бъдеще

срещи с Руби.

|  |
| --- |
| **459 серия** |

**459**

***Обратни обаждания и куки***

Това използване на method\_missing е много лесно (макар че можете да го смесвате и съчетавате

с някои от камбаните и свирките от глава 4), но много мощен; добавя страхотно

сделка с интелигентност на клас в замяна на малко усилия. Нека да разгледаме сега другия край

на спектъра: method\_missing не в определен клас, а в горната част на дървото на класа

и най-горното ниво на вашия код.

T HE оригинален : B ASIC О BJECT # МЕТОД \_ ЛИПСВА

method\_missing е един от малкото методи, дефинирани в самия връх на дървото на класа, в

класът BasicObject. Благодарение на факта, че всички класове в крайна сметка произлизат от Basic-

Обект, всички обекти имат метод method\_missing.

По подразбиране method\_missing е доста интелигентен. Вижте разликата между

съобщенията за грешки в тези две борси с irb:

>> а

NameError: недефинирана локална променлива или метод `a 'за main: Object

>> а?

NoMethodError: недефиниран метод `a? ' за основно: Обект

Неизвестният идентификатор a може да бъде или метод, или променлива (ако не беше

неизвестен, т.е.); и въпреки че се обработва чрез method\_missing, съобщението за грешка

отразява факта, че Руби не може в крайна сметка да разбере дали сте имали предвид това като извикване на метод

или препратка към променлива. Вторият неизвестен идентификатор, a ?, може да бъде само метод,

защото имената на променливите не могат да завършват с въпросителен знак. method\_missing вдига

това и прецизира съобщението за грешка (и дори избора на кое изключение да се повдигне).

Възможно е да се замени методът по подразбиране method\_missing по един от двата начина. Първо ти

може да отвори класа BasicObject и да предефинира method\_missing. Вторият, по-ком-

техниката mon (макар, разбира се, не е толкова често срещана) е да се дефинира method\_missing

на най-високо ниво, като по този начин го инсталирате като метод на частен екземпляр на Object.

Ако използвате тази втора техника, всички обекти с изключение на действителните екземпляри на BasicObject

сам ще намери новата версия на method\_missing:

>> def method\_missing (m, \* args, & block)

>> повишете NameError, "Какво, разбира се, имате предвид под # {m}?"

>> край

=> нула

>> а

NameError: Какво, разбира се, разбирате под?

от (irb): 2: в `method\_missing '

>> BasicObject.new.a

NoMethodError: недефиниран метод `a 'за # <BasicObject: 0x4103ac>

(Можете да поставите супер повикване във вашата нова версия, ако искате да го откажете до

версия в BasicObject, може би след регистриране на грешката, вместо да се създаде изключение

себе си.)

Не забравяйте, че ако дефинирате свой собствен метод\_пропускане, вие губите интелигентността

които могат да различат именуването на променливи от именуването на метод:

>> а?

NameError: Какво, разбира се, разбирате под ??

|  |
| --- |
| **Страница 460** |

**460**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

Вероятно няма значение, особено ако все пак ще се обадите на супер - и ако го направите

наистина искам, можете сами да разгледате детайлите на символа m. Но това е интерес-

поглед към тънкостите на йерархията на класовете и семантиката на заместването.

МЕТОД \_ -СКОРО , RESPOND \_ ДА ?, И да отговорят \_ ДА \_ ЛИПСВА ?

Често цитиран проблем с method\_missing е, че той не се привежда в съответствие с response\_to ?.

Помислете за този пример. В класа Person прихващаме съобщения, които започват с set\_,

и ги трансформира в методи за задаване: set\_age (n) става възраст = n и т.н.

Например:

клас Личност

attr\_accessor: име,: възраст

def инициализиране (име, възраст)

@ име, @ възраст = име, възраст

край

def method\_missing (m, \* args, & block)

ако /set\_(.\*)/.match(m)

self.send ("# {$ 1} =", \* аргументи)

друго

супер

край

край

край

И така, има ли обект човек метод set\_age или не? Е, можете да се обадите на това

метод, но лицето обект твърди, че не отговаря на него:

person = Person.new ("Дейвид", 54)

person.set\_age (55)

р човек.възраст

p person.respo\_to? (: set\_age)

Начинът да получите method\_missing и response\_to? да се изравнят помежду си е от

дефиниране на специалния метод response\_to\_missing ?. Ето определение, което можете да добавите

към предходния клас Person:

def response\_to\_missing? (m, include\_private = false)

/set\_/.match(m) || супер

край

Сега новият обект на човек ще отговаря по различен начин при същите заявки:

55

вярно

Можете да контролирате дали частните методи са включени, като използвате втори аргумент за

отговарям на?. Този втори аргумент ще бъде предаден на response\_to\_missing ?.

В примера по подразбиране е false.

Като бонус, методите, които стават видими чрез response\_to\_missing? може също

да бъдат обективирани в обекти на метод, използвайки метод:

person = Person.new ("Дейвид", 55)

p person.method (: set\_age)

**55**

**невярно**

**# <Метод:**

**Лице # set\_age>**

|  |
| --- |
| **461 серия** |

**461**

***Обратни обаждания и куки***

Като цяло, method\_missing е изключително полезен инструмент за улавяне на събития. Но това е далеч от

само един.

***15.1.2 Трапирането включва и подготвя операции***

Знаете как да включите модул в клас или друг модул и знаете как да го направите

добавете модул към клас или модул. Ако искате да заловите тези събития - да задействате a

обратно извикване, когато се случат събитията - можете да определите специални методи, наречени включени и

предварително добавен. Всеки от тези методи получава името на включването или добавянето

клас или модул като единствен аргумент.

Нека да разгледаме отблизо включените, знаейки, че предварително добавените работи почти по същия начин

начин. Можете да направите бърз тест на включен, като го задействате разпечатка на съобщение и

след това извършете операция за включване:

модул М

def self.included (c)

поставя „Току-що ме смесиха с # {c}.“

край

край

клас С

включват М

край

Виждате съобщението „Току-що ме смесиха с C.“ в резултат на изпълнението на

M.включва се, когато M се включва от (смесен в) C. (Тъй като можете да смесите и mod-

ules в модули, примерът също би работил, ако C беше друг модул.)

Кога би било полезно модул да прихване собственото си включване по този начин?

Един често обсъждан случай се върти около разликата между екземпляра и

клас методи. Когато смесвате модул в клас, гарантирате, че всички

методите на екземпляра, дефинирани в модула, стават достъпни за екземпляри от класа.

Но обектът на класа не е засегнат. Често възниква следният въпрос: Ами ако вие

искате да добавите методи на клас към класа чрез смесване в модула заедно с добавяне

методите на екземпляра?

С любезното съдействие на включено, можете да хванете операцията за включване и да използвате случая

добавете методи на клас към класа, който прави включването. Следният списък показва

пример.

модул М

def self.included (cl)

def cl.a\_class\_method

поставя „Сега класът има нов метод на клас.“

край

край

def an\_inst\_method

поставя "Този модул предоставя този метод на екземпляр."

край

край

Листинг 15.1 Използване на **включено** за добавяне на метод на клас като част от операция за смесване

|  |
| --- |
| **462 серия** |

**462**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

клас С

включват М

край

c = C. нов

c.an\_inst\_method

C.a\_class\_method

Резултатът от този списък е

Този модул предоставя този метод на екземпляр.

Сега класът има нов метод на клас.

Когато клас C включва модул M, се случват две неща. Първо, извикан е метод на екземпляр

an\_inst\_method се появява в пътеката за търсене на неговите екземпляри (като c). Второ,

благодарение на включеното обратно извикване на M е дефиниран метод на клас, наречен a\_class\_method

обектът на класа C.

Включен модул # е полезен начин да се включите в инженерирането на клас / модул на

вашата програма. Междувременно, нека разгледаме друг обратен разговор в същата обща област

от интерес: Модул # удължен.

***15.1.3 Прихващане удължаване***

Както знаете от глава 13, разширяването на отделни обекти с модули е един от

най-мощните техники, налични в Ruby, за да се възползвате от гъвкавостта на

обекти и способността им да бъдат персонализирани. Той също така е бенефициент на кука за изпълнение:

използвайки разширения метод Module #, можете да настроите обратно повикване, което ще се задейства

винаги когато обект изпълнява операция за разширяване, която включва въпросния модул.

Следващият списък показва модифицирана версия на списък 15.1, който илюстрира работата-

ings на модул # удължен.

модул М

def self.extended (obj)

поставя „Модул # {self} се използва от # {obj}.“

край

def an\_inst\_method

поставя "Този модул предоставя този метод на екземпляр."

край

край

my\_object = Object.new

my\_object.extend (M)

my\_object.an\_inst\_method

Резултатът от този списък е

Модул М се използва от # <Обект: 0x007f8e2a95bae0>.

Този модул предоставя този метод на екземпляр.

Полезно е да разгледаме как включените и разширените обратни извиквания работят заедно

с единични класове. Тук няма нищо твърде изненадващо; това, което научаваш е колко

устойчивият модел на обект и клас на Ruby е.

Листинг 15.2 Задействане на обратно извикване от **разширено** събитие

|  |
| --- |
| **463 серия** |

**463**

***Обратни обаждания и куки***

S ИНГЛЕТОН - КЛАСНО ПОВЕДЕНИЕ С РАЗШИРЕНО И ВКЛЮЧЕНО

Всъщност разширяването на обект с модул е ​​същото като включването на този модул в

единичен клас на обекта. Както и да го опишете, резултатът е, че моделът

ule се добавя към пътя за търсене на метод на обекта, като влиза във веригата веднага след

единичен клас на обекта.

Но двете операции задействат различни обратни извиквания: разширени и включени. The

следващият списък показва съответното поведение.

модул М

def self.included (c)

поставя „# {self} включено от # {c}.“

край

def self.extended (obj)

поставя „# {self} удължено с # {obj}.“

край

край

obj = Object.new

поставя „Включване на M в единичния клас на обекта:“

клас << obj

включват М

край

поставя

obj = Object.new

поставя "Разширяване на обект с M:"

obj.extend (M)

И двата обратни извиквания са дефинирани в модул M: включени B и разширени c . Всяко обаждане-

back разпечатва отчет за това, което прави. Започвайки с прясно отсечен, генеричен

обект, ние включваме M в единичния клас d на обекта и след това повтаряме процеса,

използване на друг нов обект и разширяване на обекта с M директно e .

Резултатът от този списък е

Включване на M в единичния клас на обекта:

M включено от # <Клас: # <Обект: 0x0000010193c978 >>.

Удължаващ обект с М:

M удължен с # <Обект: 0x0000010193c310>.

Разбира се, включването задейства включеното обратно извикване, а задействането за разширяване

удължен, въпреки че в този конкретен сценарий резултатите от двете операции са

същото: въпросният обект е добавил M към пътя си за търсене на метода. Това е хубаво

илюстрация на някои от тънкостите и прецизността на архитектурата на Ruby и полезен

напомняне, че манипулирането на единичен клас на обект директно не е *съвсем* идентично с

извършване на операции на единично ниво директно върху обекта.

Точно както модулите могат да прихващат операции за включване и разширяване, класовете могат да определят кога

те са подкласирани.

Листинг 15.3 Разширяване на обект и включването му в неговия единичен клас

б

° С

д

д

|  |
| --- |
| **464 серия** |

**464**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

***15.1.4 Прихващане на наследяването с наследен клас #***

Можете да се включите в подкласирането на клас, като дефинирате специален метод на класа

наречен наследен за този клас. Ако наследяването е дефинирано за даден клас, тогава

когато подкласирате класа, наследяваният се извиква с името на новия клас като

неговият единствен аргумент.

Ето един прост пример, където клас C докладва за факта, че е бил

подклас:

клас С

def self.inherited (подклас)

поставя "# {self} току-що е подкласиран от # {subclass}."

край

край

клас D <C

край

Подкласирането на C от D автоматично задейства повикване за наследяване и произвежда

следния изход:

C току-що е подкласиран от D.

наследен е метод на клас, така че потомците на класа, който го дефинира, също могат

наречете го. Действията, които дефинирате в наследена каскада: ако наследявате от подклас,

че подкласът задейства наследения метод и така нататък по веригата от наследствени

танц. Ако направите това

клас E <D

край

сте информирани, че D току-що е полукласифициран от E. Получавате подобни резултати, ако

клас Е и т.н.

Ограниченията на **наследеното** обратно извикване

Всичко има своите граници, включително наследеното обратно извикване. Когато D наследява от C, C

е суперкласът на D; но освен това класът на сингълтон на C е суперкласът на сингълтона на D

клас. Ето как D успява да може да извиква методите на класа на C. Но няма обратен разговор

задейства. Дори ако дефинирате наследен в сингълтон клас C, той никога не се извиква.

Ето тестово поле. Обърнете внимание как наследеното се дефинира в сингълтон класа на C. Но

дори когато D наследява от C - и дори след изричното създаване на сингълтона на D

клас - обратното обаждане не се задейства:

клас С

клас << себе си

def self. наследен

поставя "Сингълтън клас на C току-що е наследен!"

поставя „Но никога няма да видите това съобщение.“

край

край

край

|  |
| --- |
| **465 серия** |

**465**

***Обратни обаждания и куки***

Нека да разгледаме сега как да прихванем препратка към несъществуваща константа.

***15.1.5 Методът # const\_missing метод***

Модул # const\_missing е друг често използван обратен разговор. Както подсказва името, това

метод се извиква всеки път, когато в дадена се посочва неидентифицируема константа

модул или клас:

клас С

def self.const\_missing (const)

поставя "# {const} е недефиниран - задава го на 1."

const\_set (const, 1)

край

край

поставя C :: A

поставя C :: A

Резултатът от този код е

A е недефиниран - задайте го на 1.

1

1

Благодарение на обратното повикване, C :: A се дефинира автоматично, когато го използвате без

инж. Това се грижи по такъв начин, че путовете да могат да отпечатат стойността на кон

постоянство; никога не трябва да знае, че константата не е била дефинирана на първо място. Тогава,

при второто извикване на путове константата вече е дефинирана и const\_missing

не се нарича.

Едно от най-мощните средства за обратно извикване на събития в Ruby е method\_added, което

ви позволява да задействате събитие, когато е дефиниран нов метод на екземпляр.

***15.1.6 Методът\_добавен и единичен\_метод\_добавени методи***

Ако дефинирате method\_added като метод на клас във всеки клас или модул, той ще бъде извикан

когато е дефиниран всеки метод на екземпляр. Ето един основен пример:

***(продължение)***

клас D <C

клас << себе си

поставя "Единичният клас на D вече съществува, но няма обратно извикване!"

край

край

Резултатът от тази програма е

Вече съществува сингълтон клас на D, но няма обратно извикване!

Изключително малко вероятно е някога да попаднете на ситуация, в която това поведение да

ters, но ви дава хубава рентгенова снимка на това как моделът на класа на Ruby си взаимодейства с неговия

слой за обратно извикване.

|  |
| --- |
| **466 серия** |

**466**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

клас С

def self.method\_added (m)

поставя "Метод № {m} току-що беше дефиниран."

край

def a\_new\_method

край

край

Резултатът от тази програма е

Методът a\_new\_method току-що беше дефиниран.

Обратното извикване на singleton\_method\_added прави почти същото нещо, но за singleton

методи. Може би изненадващо, дори се задейства. Ако пуснете този фрагмент

клас С

def self.singleton\_method\_added (m)

поставя "Метод № {m} току-що беше дефиниран."

край

край

ще видите, че обратното повикване - което е единичен метод за обекта от клас C - триг-

създава собствено изпълнение:

Методът singleton\_method\_added беше току-що дефиниран.

Обратното обаждане също ще бъде задействано от дефиницията на друг единичен (клас)

метод. Нека разширим предишния пример, за да включим такава дефиниция:

клас С

def self.singleton\_method\_added (m)

поставя "Метод № {m} току-що беше дефиниран."

край

def self.new\_class\_method

край

край

Новият изход е

Методът singleton\_method\_added беше току-що дефиниран.

Методът new\_class\_method току-що беше дефиниран.

В повечето случаи трябва да използвате singleton\_method\_added с обекти, различни от клас

обекти. Ето как може да се играе използването му с общ обект:

obj = Object.new

def obj.singleton\_method\_added (m)

поставя "Методът на Singleton # {m} беше току-що дефиниран."

край

def obj.a\_new\_singleton\_method

край

Резултатът в този случай е

Единичен метод singleton\_method\_added беше току-що дефиниран.

Единичният метод a\_new\_singleton\_method беше току-що дефиниран.

**Определя обратно повикване**

**Задейства го чрез дефиниране**

**метод на екземпляр**

|  |
| --- |
| **467 серия** |

**467**

***Тълкуване на заявки за възможности на обекти***

Отново получавате донякъде изненадващия ефект, който дефинирането на singleton\_method\_added

задейства собственото изпълнение на обратното повикване.

Съчетавайки подходите, базирани на класове и обекти, можете да постигнете

специфичен за обекта ефект чрез дефиниране на съответните методи в единичния клас на обекта:

obj = Object.new

клас << obj

def singleton\_method\_added (m)

поставя "Методът на Singleton # {m} беше току-що дефиниран."

край

def a\_new\_singleton\_method

край

край

Изходът за този фрагмент е точно същият като за предишния пример. И накрая,

идващ пълен кръг, можете да определите singleton\_method\_added като обикновен екземпляр

метод на клас, в който случай всеки екземпляр на този клас ще следва правилото, че

обратното обаждане ще се задейства от създаването на единичен метод:

клас С

def singleton\_method\_added (m)

поставя "Методът на Singleton # {m} беше току-що дефиниран."

край

край

c = C. нов

def c.a\_singleton\_method

край

Тук дефиницията за обратно извикване B управлява всеки екземпляр на C. Дефиницията на a

singleton метод на такъв екземпляр c следователно задейства обратно извикване, в резултат на което

този изход:

Методът на единичен метод a\_singleton\_method току-що беше дефиниран.

Възможно е да не използвате нито method\_added, нито singleton\_method\_added

често във вашите Ruby приложения. Но експериментирането с тях е чудесен начин да се получи

по-дълбоко усещане за това как различните части на класа, екземпляра и сингъл-класа картини

прилягат заедно.

Сега ще се обърнем към темата за изследване на възможностите на обекта ("abc" .methods и

приятели). Основите на тази тема бяха включени в проучването „Вградени основни неща“ през

глава 7 и както е обещано в тази глава, ще разгледаме по-задълбочено тук.

***15.2 Тълкуване на заявки за възможности на обекти***

На този етап в работата си с Ruby можете да насочите вниманието си към това да правите повече със списъци

на методите на обектите, отколкото да ги изследва и изхвърля. В този раздел ще разгледаме

няколко примера (и ще остане достатъчно място за вас, за да създадете повече, както си

нуждите и интересите изискват) на начини, по които бихте могли да използвате и интерпретирате информацията

в списъците с методи. Рубинът, който научихте, откакто последно се обърнахме към тази тема

б

° С

|  |
| --- |
| **468 серия** |

**468**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

директно ще ви застане на добро място. Ще научите и няколко фини точки на метода-

самите методи за заявки.

Нека започнем от най-познатата отправна точка: изброяване на не частни методи с

методът на методите.

***15.2.1 Изброяване на не частни методи на обект***

За да изброите не частните (т.е. публични или защитени) методи, за които даден обект знае,

използвате метода на метода, който връща масив от символи. Масивите са масиви,

можете да изпълните някои полезни заявки за резултатите от първоначалната заявка. Обикновено,

ще филтрирате масива по някакъв начин, за да получите подмножество от методи.

Ето например как можете да попитате низ, какви методи знае за това

включват промяна на случая:

>> string = "Тестов низ"

=>"Тестов низ"

>> string.methods.grep (/ case /). sort

=> [: casecmp,: downcase,: downcase !,: swapcase,: swapcase !,: upcase,

: upcase!]

Grep филтрира всеки символ, в който няма малки букви. (Не забравяйте, че въпреки това

те не са низове, символите показват редица низоподобни поведения, като например grep-

pable.) Командата за сортиране в края е полезна за повечето операции с изброяване на методи. То

не прави голяма разлика в този пример, защото има само седем метода

os; но когато получите обратно масиви от 100 или повече символа, сортирането им може да помогне много.

Грепването за случай зависи от предположението, разбира се, че свързаният със случая метод

ods ще имат регистър в техните имена. Определено често има елемент на преценка

по пътя на правенето на образовани предположения за това, което мислите, че ще намерите, в много

заявки за способност на метод. Нещата обаче се получават, тъй като Руби е повече от реален

сонарно последователен и конвенционален в избора си на имена на методи.

Някои от методите за случай също са bang (!) Методи. Следвайки тази тема, нека

разберете всички взривни методи, които има низ, отново с помощта на grep операция:

>> string.methods.grep (/.!/). сортиране

=> [: изписване с главни букви !,: chomp !,: chop !,: delete !,: downcase !,: encode !,: gsub !,

: lstrip !,: next !,: reverse !,: rstrip !,: scrub !,: slice !,: squeeze !,

: strip !,: sub !,: succ !,: swapcase !,: tr !,: tr\_s !,: upcase!]

Защо точката преди! в регулярния израз? Целта му е да гарантира, че

има поне един знак преди! в името на метода и по този начин да се изключи

методите!,! = и! ~, които съдържат! но не са взривни методи в обичайните

смисъл. Искаме методи, които завършват с гръм и трясък, но не и такива, които започват с такъв.

Нека използваме методи малко по-нататък. Ето въпрос, на който можем да отговорим, като тълкуваме

резултатите от заявката за метод: имат ли низове някакви взривни методи, които нямат съответстващи

прилагане на методи без взрив?

string = "Тестов низ"

методи = низ.методи

bangs = string.methods.grep (/.!/)

б

|  |
| --- |
| **469 серия** |

**469**

***Тълкуване на заявки за възможности на обекти***

unmatched = bangs.reject do | b |

методи.включете? (b [0 ..- 2] .to\_sym)

край

ако unmatched.empty?

поставя „Всички взривни методи се съчетават с не-взривни методи.“

друго

поставя "Някои методи за взрив нямат партньор, който не е взрив:"

поставя несравним

край

Кодът работи, като събира всички публични методи на низ и, отделно, всички негови

взрив методи B . След това операция за отхвърляне филтрира всички имена на взривни методи за

което съответно име без взрив може да бъде намерено в по-голямото име-метод

списък c . Индексът [0 ..- 2] грабва всичко, освен последния знак на символа -

име на метода минус!, с други думи - и извикването към to\_sym преобразува резултата-

връщане на низ обратно към символ, така че да включва? test може да го потърси в масива на

методи. Ако филтрираният списък е празен, това означава, че няма несравним метод на взрив

имена бяха намерени. Ако не е празно, тогава поне едно такова име е намерено и може да бъде

отпечатан d .

Ако стартирате скрипта такъв, какъвто е, той винаги ще вземе първия (истински) клон на if state-

мента. Ако искате да видите списък с несравними методи за взрив, можете да добавите следното

ред към програмата, непосредствено след първия ред:

def string.surprise !; край

Когато стартирате модифицираната версия на скрипта, ще видите това:

Някои методи за взрив нямат партньор, който не е взрив:

изненада!

Както вече видяхте, писането на bang методи без партньори, които не са взривове, обикновено е

лоша практика - но това е добър начин да видите метода на методите в действие.

Можете, разбира се, да попитате обекти от клас и модул какви са техните методи. След всичко,

те са просто предмети. Но не забравяйте, че методът методи винаги изброява не-

частни методи на самия обект. В случай на класове и модули, това означава

не получавате списък с методите, екземпляри на класа - или екземпляри на

класове, които се смесват в модула - могат да се обаждат. Получавате методите, които класът или

самият модул знае за. Ето (частичен) резултат от извикване на методи за ново

създаден обект на клас:

>> клас С; край

=> нула

>> C.methods.sort

=> [:!,:! =,:! ~,: <,: <=,: <=>,: ==,: ===,: = ~,:>,:> =,: \_\_ id\_\_,: \_\_send\_\_,

: allocate,: предци,: autoload,: autoload ?,: class,: class\_eval,

: class\_exec,: class\_variable\_defined ?,: class\_variable\_get,

: class\_variable\_set,: class\_variables и др.

Обектите на клас и модул споделят някои методи със собствените си екземпляри, защото

всички те са обекти и като цяло обектите споделят определени методи. Но методите вие

° С

д

**Изход: Всички взривове**

**методите са съпоставени**

**по не-взривни методи.**

|  |
| --- |
| **470** |

**470**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

see са тези, които самият клас или модул може да извика. Можете също така да попитате класове и

ules за методите на екземпляра, които те дефинират. Скоро ще се върнем към тази техника.

Първо, нека разгледаме накратко процеса на изброяване на частни и защитени методи на обект.

***15.2.2 Изброяване на частни и защитени методи***

Всеки обект (с изключение на екземпляри на BasicObject) има метод private\_methods и a

метод защитен\_методи. Те работят както очаквате; те предоставят масиви от символи

bols, но съдържащи съответно имена на частни и защитени методи.

Прясно изсечените обекти Ruby имат много частни методи и никакви защитени методи:

$ ruby ​​-e 'o = Object.new; p o.private\_methods.size '

72

$ ruby ​​-e 'o = Object.new; p o.protected\_methods.size '

0

Какви са тези частни методи? Те са методи на частен екземпляр, дефинирани най-вече в

модула Kernel и на второ място в класа BasicObject. Ето как можете

проследете това:

$ ruby ​​-e 'o = Object.new; p o.private\_methods -

BasicObject.private\_instance\_methods (false) -

Kernel.private\_instance\_methods (false) '

[]

Имайте предвид, че след като извадите частните методи, дефинирани в Kernel и BasicObject,

оригиналният обект няма частни методи за изброяване. Частните методи, дефинирани в

Ядрото са методите, които смятаме за „най-високо ниво“, като путове, обвързване и рейз. Възпроизвеждане

наоколо малко с методите за изброяване на методи, които изучавате тук, и ще видите

някои познати методи, изброени.

Естествено, ако сами дефинирате частен метод, той също ще се появи в списъка на

частни методи. Ето пример: прост клас Person, в който се присвоява име

към лицето чрез метода name = задейства метод за нормализиране на име, който

премахва всичко, освен буквите и избраните пунктуационни знаци от

име. Методът normalize\_name е частен:

клас Личност

attr\_reader: име

def name = (име)

@ име = име

normalize\_name

край

частни

def normalize\_name

name.gsub! (/ [^ - a-z '. \ s] / i, "")

край

край

david = Person.new

david.name = "123David !! Bl% a9ck"

повдигнете "Проблем", освен ако david.name == "Дейвид Блек"

поставя "Името е нормализирано."

p david.private\_methods.sort.grep (/ нормално /)

**Определя недефолт**

**записващ аксесоар**

**Нормализира името**

**когато е назначен**

**Премахва нежеланото**

**знаци от име**

**Уверява се**

**работи за нормализиране**

**Отпечатайте съобщение за успех**

**Резултат от частен**

**проверка на метода:**

**[: normalize\_name]**

|  |
| --- |
| **Страница 471** |

**471**

***Тълкуване на заявки за възможности на обекти***

Защитените методи могат да бъдат разгледани по същия начин, като се използва защитеното\_

методи метод.

Освен че питат обектите за какви методи знаят, често е полезен

да попитат класовете и модулите какви методи предоставят.

***15.2.3 Получаване на методи за екземпляри на клас и модул***

Класовете и модулите идват с донякъде измислен набор от методи за заявка на методи

отс. Разглеждането на наличните в String илюстрира пълния списък. Методите

които са специфични за класовете и модулите са **получер** :

>> String.methods.grep (/ methods /). Sort

=> [: **instance\_methods ,:** methods **,: private\_instance\_methods** ,

: private\_methods,: **protected\_instance\_methods** ,: protected\_methods,

**: public\_instance\_methods,: public\_methods** ,: singleton\_methods]

Показаните с удебелен шрифт методи ви дават списъци с дефинирани методи от различни видове

в класа или модула. Четирите метода работят по следния начин:

instance\_methods връща всички публични и защитени методи на екземпляр.

public\_instance\_methods връща всички методи на публична инстанция.

protected\_instance\_methods и private\_instance\_methods връщат всички про-

съответно тектирани и частни методи.

Когато извиквате някой от тези методи, имате възможност да предадете аргумент. Ако

предавате аргумента false, тогава списъкът с методи, които получавате обратно, ще включва

само тези, дефинирани в класа или модула, към който заявявате. Ако предадете някакъв аргумент

с логическа истина (нещо различно от фалшиво или нулево) или ако не подадете аргумент,

списъкът с методи ще включва тези, дефинирани в класа или модула, който заявявате

и всички негови класове и модули на предци.

Например можете да разберете кои методи на екземпляра определя класът Range,

като този:

>> Range.instance\_methods (false) .sort

=> [: ==,: ===,: begin,: bsearch,: cover ?,: each,: end,: eql ?,: exclude\_end ?,

: first,: hash,: include ?,: inspect,: last,: max,: member ?,: min,: size,

: стъпка,: до\_с]]

Отивайки още една стъпка напред, какво ще стане, ако искате да знаете кой от методите, дефинирани в

Безброй модули са заменени в диапазона? Можете да разберете, като изпълните и

(&) работа с двата списъка с методи на екземпляра: тези, дефинирани в Enumerable и

дефинираните в диапазон:

>> Range.instance\_methods (false) & Enumerable.instance\_methods (false)

=> [: първо,: мин,: макс,: член ?,: включи?]

Както можете да видите, Range предефинира пет метода, които Enumerable вече дефинира.

Ще разгледаме накратко последния от методите в стил методи, singleton\_methods.

Но първо, нека създадем програма, която създава списък с всички замествания на всички класове

този микс в Изброими.

|  |
| --- |
| **472 серия** |

**472**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

G настройвате ВСИЧКИ Е може да бъде преброен Overrides

Стратегията тук ще бъде да разберете кои класове се смесват в Enumerable и след това

формират на всеки такъв клас операция и (&) като тази в последния пример, съхраняване

резултатите и накрая, отпечатването им. Следният списък показва кода.

замества = {}

enum\_classes = ObjectSpace.each\_object (Class) .изберете do | c |

c.ancestors.include? (Изброими)

край

enum\_classes.sort\_by {| c | c.name} .each do | c |

замества [c] = c.instance\_methods (false) &

Enumerable.instance\_methods (false)

край

overrides.delete\_if {| c, методи | методи.празни? }

замества.всеки | c, методи |

поставя „Клас № {c} отменя: # {методите.join (", ")}"

край

На първо място, ние създаваме една празна хашиш в променливата замени B . След това получаваме списък с всички

класове, които се смесват в Enumerable. Техниката за получаване на този списък включва

Модул ObjectSpace и неговия метод each\_object c . Този метод отнема единичен

аргумент, представляващ класа на обектите, които искате да намери. В този случай ние сме

се интересуваме от обекти от клас Class, а ние се интересуваме само от тези, които имат Enu-

сред предците им. Методът всеки\_обект връща изчислител,

и повикването за избор на този изброител има желания ефект от филтрирането на списъка с

всички класове до списък само на тези, които са се смесили в Enumerable.

Сега е време да се попълни хешът на отмените. За всеки клас в enum\_classes

(добре сортирано по име на клас), поставяме запис в замествания. Ключът е класът и

стойността е масив от имена на методи - имената на методите Enumerable, които

този клас отменя d . След премахване на всички записи, представляващи класове, които не са

замени всички изброени методи e , ние продължаваме да отпечатваме резултатите, като използваме сортиране

и се присъединете към операции, за да направите изхода да изглежда последователен и ясен f , както е показано тук:

Клас ARGF.class замества: to\_a

Заместване на масив от класове: to\_a, to\_h, first, reverse\_each, find\_index, sort,

събиране, картографиране, избор, отхвърляне, цип, включване ?, броене, цикъл, вземане,

take\_ while, drop, drop\_ while

Заменители на класа на изброяването: всеки\_с\_индекс, всеки\_с\_обект

Преброител на класове :: Мързеливи замествания: map, collect, flat\_map, collect\_concat,

select, find\_all, отхвърляне, grep, zip, take, take\_ while, drop, drop\_ while,

мързелив, парче, парче\_преди

Замяна на хеш клас: to\_h, to\_a, изберете, отхвърлете, включете ?, член?

Клас ObjectSpace :: Замяна на WeakMap: включва ?, член?

Замяна на обхвата на класа: първо, мин, макс, член ?, включва?

Замяна на структурата на класа: to\_a, to\_h, изберете

Първият ред се отнася до малко аномалния обект, обозначен като ARGF.class,

което е уникален, специално проектиран обект, участващ в обработката на програмата

Листинг 15.4 **Замествания на безброй** потомци на методите на **Enumerable** instance

б

° С

д

д

е

|  |
| --- |
| **Страница 473** |

**473**

***Тълкуване на заявки за възможности на обекти***

вход. Останалите редове се отнасят до няколко познати класа, които се смесват в Enumerable. В

във всеки случай виждате кои изброими методи въпросният клас е заменил.

Нека да разгледаме по-нататък как да заявяваме обект по отношение на неговите единични методи.

***15.2.4 Изброяване на единични методи на обекти***

Както знаете, единичен метод е метод, дефиниран единствено за дадено използване

обект (или, ако обектът е клас, за използване на обекта и неговите подкласове) и

съхранявани в единичен клас на този обект. Можете да използвате метода singleton\_methods за

избройте всички такива методи. Имайте предвид, че singleton\_methods изброява публични и защитени single-

тон методи, но не и частни. Ето пример:

клас С

край

c = C. нов

клас << c

def x

край

def y

край

def z

край

защитен: y

частно: z

край

p c.singleton\_methods.sort

Екземпляр на клас C се създава B и неговият единичен клас се отваря c . Три мет-

os са дефинирани в сингълтон класа, по един в публичния (x), защитен (y) и

частни (z) нива d . Разпечатката на единичните методи на c e изглежда така:

[: x,: y]

Единичните методи също се считат за просто методи. Методите: x и: y ще

покажете се, ако извикате и c.methods. Можете да използвате методите за заявки, базирани на клас

за сингълтон класа. Добавете този код в края на последния пример:

клас << c

p private\_instance\_methods (false)

край

Когато го стартирате, ще видите следното:

[: z]

Методът: z е единичен метод на c, което означава, че е метод на екземпляр (a

метод на частен екземпляр, както се случва) на сингълтон клас c.

Можете да попитате клас за неговите единични методи и ще получите единичните методи

определени за този клас и за всички негови предци. Ето илюстрация, базирана на IRB:

>> клас С; край

=> нула

>> клас D <C; край

б

° С

д

д

|  |
| --- |
| **474** |

**474**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

=> нула

>> def C.a\_class\_method\_on\_C; край

=> нула

>> def D.a\_class\_method\_on\_D; край

=> нула

>> D.singleton\_methods

=> [: a\_class\_method\_on\_D,: a\_class\_method\_on\_C]

След като се запознаете с различни методи, ще ги намерите като

пълно за изучаване и проучване как и къде са дефинирани методите. Например вие

може да използва заявки за методи, за да провери как са съставени методите на класа на File. Да се

започнете с, разберете кои методи на клас файлът наследява от своите предци, за разлика от

тези, които той сам определя:

>> File.singleton\_methods - File.singleton\_methods (невярно)

=> [: ново,: отворено,: sysopen,: for\_fd,: popen,: foreach,: readlines,

: read,: select,: pipe,: try\_convert,: copy\_stream]

Извикването на singleton\_methods (false) предоставя само дефинираните singleton методи

на файл. Извикването без фалшивия аргумент предоставя всички сингълтон методи

дефиниран на File и неговите предци. Разликата е в дефинираните от предците.

Суперкласът на File е IO. Интересното, макар и не изненадващо, е, че всички 12 от

родовите сингълтон методи, достъпни за File, са дефинирани в IO. Можете да потвърдите това

с друга заявка:

>> IO.singleton\_methods (false)

=> [: ново,: отворено,: sysopen,: for\_fd,: popen,: foreach,: readlines,

: read,: select,: pipe,: try\_convert,: copy\_stream]

Връзката между класовете - в случая фактът, че File е подклас на IO и

следователно споделя своите единични методи (методите на клас) - се вижда директно в

масиви с име на метод. Различните методи позволяват почти неограничен контрол

ния и проучване от този вид.

Както можете да видите, съоръженията за търсене на методи в Ruby могат да ви кажат много

за обектите, класа и модулите, с които работите. Просто трябва да се свържете

точките чрез прилагане на техники за събиране на заявки и обработка на текст към списъците

те осигуряват. Тълкуването на заявки за метод е хубав пример за вида на обучението

обратна връзка, която Ruby предоставя: колкото повече научавате за езика, толкова повече

можете *да* научите.

Ще се обърнем към въпроса за отражението по време на изпълнение на променливи и константи.

***15.3 Самоанализ на променливи и константи***

Ruby може да ви каже няколко неща за това до кои променливи и константи имате достъп

в даден момент по време на изпълнение. Можете да получите списък на локални или глобални променливи, обект

променливи на екземпляра, променливите на класа на клас или модул и константите на клас

или модул.

|  |
| --- |
| **475 серия** |

**475**

***Самоанализ на променливи и константи***

***15.3.1 Изброяване на локални и глобални променливи***

Локалните и глобалните инспекции на променливи са ясни: вие използвате най-високото ниво

методи local\_variables и global\_variables. Във всеки случай получавате обратно

масив от символи, съответстващи на локалните или глобалните променливи, дефинирани в момента:

x = 1

p local\_variables

[:х]

p global\_variables.sort

[: $ !,: $ ",: $$,: $ &,: $ ',: $ \*,: $ +,: $ ,,: $ - 0,: $ - F,: $ - I,: $ -K,: $ - W,: $ - a,

: $ - d,: $ - i,: $ - l,: $ - p,: $ - v,: $ - w,: $.,: $ /,: $ 0,: $ 1,: $ 2,: $ 3, : $ 4,: $ 5,

: $ 6,: $ 7,: $ 8,: $ 9,: $ :,: $ ;,: $ <,: $ =,: $>,: $ ?,: $ @,: $ DEBUG,: $ FILENAME,

: $ KCODE,: $ LOADED\_FEATURES,: $ LOAD\_PATH,: $ PROGRAM\_NAME,: $ SAFE,: $ VERBOSE,

: $ \,: $ \_,: $ `,: $ stderr,: $ stdin,: $ stdout,: $ ~]

Глобалният списък с променливи включва глобали като $: (пътят за зареждане на библиотеката, наличен и като

$ LOAD\_PATH), $ ~ (глобалният обект MatchData въз основа на най-новия модел-

съответстваща операция), $ 0 (името на файла, в който се изпълнява текущата про-

gram е иницииран), $ FILENAME (името на файла, който се изпълнява в момента) и

други. Списъкът с локални променливи включва всички дефинирани в момента локални променливи.

Имайте предвид, че local\_variables и global\_variables не ви дават стойностите на

променливите, за които докладват; те просто ви дават имената. Същото важи и за

instance\_variables метод, който можете да извикате за всеки обект.

***15.3.2 Изброяване на променливи на екземпляра***

Ето още едно предаване на прост клас Person, който илюстрира в какво участва

заявка за променлива на екземпляр:

клас Личност

attr\_accessor: име,: възраст

def инициализиране (име)

@ име = име

край

край

david = Person.new ("Дейвид")

david.age = 55

p david.instance\_variables

Изходът е

[: @name,: @age]

Обектът David има две променливи на екземпляра, инициализирани по време на заявката. Един от

them, @name, беше присвоена стойност по време на създаването на обекта. Другият, @age, е

присъства поради възрастта на атрибута на достъп. Атрибутите се изпълняват като четене и /

или пишете методи около променливи на екземпляра, така че въпреки че @age не изглежда explic-

itly навсякъде в програмата, тя се инициализира, когато на обекта е определена възраст.

Всички променливи на екземпляра започват с символа @, а всички глобални започват с $. Вие

може да очаква Руби да не се занимава с тези знаци, когато ви дава списъци с варианти

способни имена; но имената, които получавате в списъците, включват началните знаци.

|  |
| --- |
| **476 серия** |

**476**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

След това ще разгледаме техниките за проследяване на изпълнението, които ви помагат да определите метода-

извикване на история в даден момент по време на изпълнение.

***15.4 Изпълнение на проследяване***

Без значение къде се намирате в изпълнението на вашата програма, стигнахте по някакъв начин.

Или сте на най-високо ниво, или сте един или повече методологични обаждания дълбоко. Руби осигурява

информация за това как стигнахте до къде сте. Основният инструмент за изследване на

хронологията за извикване на метод е повикващият метод от най-високо ниво.

***15.4.1 Проучване на стека на стека с повикващия***

Методът на повикващия осигурява масив от низове. Всеки низ представлява една стъпка

проследяването на стека: описание на единично извикване на метод по пътя до мястото, където се намирате

сега. Низовете съдържат информация за файла или програмата, където методът

е извършено обаждането, линията, на която е възникнало извикването на метода, и методът от

който се наричаше текущият метод, ако има такъв.

Ето един пример. Поставете тези редове във файл, наречен tracedemo.rb:

def x

у

край

def y

z

край

def z

поставя "Stacktrace:"

p повикващ

край

х

Всичко, което прави тази програма, е да се погребе в куп извиквания на методи: x извиква y, y извиква z.

Вътре в z получаваме проследяване на стека, с любезното съдействие на повикващия. Ето резултата от изпълнението

tracedemo.rb:

Променливата за подчертаване на irb

Ако стартирате local\_variables в нова irb сесия, ще видите долна черта:

>> местни\_променливи

=> [: \_]

Подчертаването е специална променлива irb: тя представлява стойността на последния израз

оценява се от irb. Можете да го използвате, за да вземете стойности, които иначе ще изчезнат:

>> Person.new ("Дейвид")

=> # <Лице: 0x000001018ba360 @ name = "David">

>> Дейвид = \_

=> # <Лице: 0x000001018ba360 @ name = "David">

Сега обектът Person е обвързан с променливата david.

|  |
| --- |
| **477** |

**477**

***Проследяване на изпълнението***

Проследяване на стека:

["tracedemo.rb: 6: в` y '"," tracedemo.rb: 2: в `x'", "tracedemo.rb: 14: in

`<main>'"]

Всеки низ в масива за проследяване на стека съдържа по една връзка във веригата от извиквания на методи, които

ни стигна до точката, в която беше повикан повикващият. Първият низ представлява най-много

скорошно обаждане в историята: бяхме на ред 6 на tracedemo.rb, вътре в метода y. The

вторият низ показва, че стигнахме до y чрез x. Третият, последен низ ни казва, че сме били в

<main>, което означава, че повикването към x е направено от най-горното ниво, а не отвътре

метод.

Може да разпознаете синтаксиса на проследяване на стека от съобщенията, които сте видели от фатално

грешки. Ако пренапишете z метода, за да изглежда така

def z

повишаване

край

изходът ще изглежда така:

tracedemo.rb: 10: в `z ': необработено изключение

от tracedemo.rb: 6: в `y '

от tracedemo.rb: 2: в `x '

от tracedemo.rb: 13: в `<main>'

Това, разбира се, е само малко измислена версия на масива за проследяване на стека, който получихме

за първи път от обаждащия се.

Проследяванията на стека на Ruby са полезни, но те също така изглеждат накриво, защото се състоят

само от струни. Ако искате да направите нещо с информацията, проследяване на стека

vides, трябва да сканирате или анализирате низа и да извлечете полезната информация. Друг

подходът е да се напише инструмент Ruby за синтактичен анализ на следи от стека и превръщането им в обекти.

***15.4.2 Писане на инструмент за синтактичен анализ на стека на стека***

Като се получи проследяване на стека - масив от низове - ние искаме да генерираме масив от обекти, всеки

от които има познания за програма или име на файл, номер на ред и име на метод

(или <основна>). Ще напишем клас за повикване, който ще представлява една стъпка за проследяване на стека

обект и клас на стека, който ще представлява цяла трасировка на стека, състояща се от една или

още Обекти за повикване. За да сведем до минимум риска от сблъсъци с имената, нека поставим и двата класа

вътре в модул, CallerTools. Нека започнем, като опишем по-подробно какво от всеки

двата класа ще направят.

CallerTools :: Call ще има три четци атрибута: програма, линия и meth.

(По-добре е да използвате meth от метода като име на третия атрибут, защото

класовете вече имат метод, наречен метод и не искаме да го отменяме.)

инициализация, обект от този клас ще анализира низ за проследяване на стека и ще запази

vant поднизове до подходящите променливи на инстанцията за по-късно извличане чрез атрибута

методи за четене на бутове.

CallerTools :: Stack ще съхранява един или повече обекти на повикване в масив, който от своя страна

ще се съхранява в променливата на екземпляра @backtrace. Също така ще напишем метод на доклад,

|  |
| --- |
| **Страница 478** |

**478**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

което ще създаде (разумно) доста печатно представяне на цялата информация

в този конкретен набор от разговори.

Сега, нека напишем класовете.

T HE С ALLER T OOLS :: С ВСИЧКИ КЛАС

Следващият списък показва класа на повикванията заедно с първия ред на цялата програма

грам, който обгръща всичко останало в модула CallerTools.

модул CallerTools

клас Обаждане

CALL\_RE = /(.\*):(\d+):in `(. \*) '/

attr\_reader: program,: line,: meth

def инициализира (низ)

@program, @line, @meth = CALL\_RE.match (низ) .captures

край

def to\_s

"% 30s% 5s% 15s"% [програма, линия, мет]

край

край

Нуждаем се от регулярен израз, с който да анализираме низовете за проследяване на стека; че редовно

изразяване се съхранява в CALL\_RE константа на B . CALL\_RE има три скоби

групови групи, разделени от неуловени буквални поднизове. Ето как редовните

изразът съвпада с типичен низ за проследяване на стека. Удебелен шрифт показва под-

низове, които са уловени от съответните подмодели на регулярни изрази. The

несмелите символи не са включени в заснеманията, но са съвпадащи буквално:

myrubyfile.rb: 234: в `a\_method '

**. \***

: **\ d +** : в '**. \***

'

Класът има, както е посочено, три четци атрибута за трите компонента на

обадете се c . Инициализацията изисква низ аргумент, низът е съпоставен с CALL\_RE,

и резултатите, достъпни чрез метода за улавяне на обекта MatchData, се поставят

в три екземпляра променливи, съответстващи на атрибутите, като се използва паралелно присвояване

среда г . (Получаваме фатална грешка при опит за извикване на улавяне на нула, ако няма съвпадение.

Можете да промените кода, за да се справите директно с това състояние, ако желаете.)

Ние също дефинираме метод to\_s за обекти на повикване e . Този метод влиза в сила през

ситуации, в които е полезно да отпечатате отчет за определен елемент за обратно проследяване. То

включва удобната техника на Ruby%. Отляво на% е форматиране в стил sprintf

низ, а вдясно е масив от заместващи стойности. Може да искате да се повозите

с дължините на полетата в заместващия низ - или за този въпрос напишете вашия

собствен метод to\_s, ако предпочитате различен стил на извеждане.

Сега е време за класа Stack.

T HE С ALLER T OOLS :: S TACK КЛАС

Класът Stack, заедно с инструкцията за затваряне на края за целия CallerTools

модул, е показан в следващия списък.

Листинг 15.5 Начало на модула **CallerTools** , включително класа **Call**

б

° С

д

д

|  |
| --- |
| **479 серия** |

**479**

***Проследяване на изпълнението***

клас Stack

def инициализира

стек = повикващ

stack.shift

@backtrace = stack.map направи | повикване |

Call.new (обаждане)

край

край

def доклад

@ backtrace.map правя | обаждане |

call.to\_s

край

край

def find (& block)

@ backtrace.find (& block)

край

край

При инициализиране, нов Stack обект призовава обаждащия се и спестява получения масив B . То

след това премества този масив, премахвайки първия низ; този низ докладва за повикването към

Stack.new сам по себе си и следователно е просто шум.

Съхраненият @backtrace трябва да се състои от един обект за повикване за всеки низ в

масив my\_caller. Това е работа за карта c . Имайте предвид, че няма атрибут на четеца за обратно проследяване

бут. В този случай всичко, от което се нуждаем, е променливата на екземпляра за вътрешна употреба от обекта.

Следва методът на отчета, който използва map на масива @backtrace към gener-

изяде масив от низове за всички обекти Call в стека d . Този масив на отчета е подходящ

способни за печат или, ако е необходимо, за търсене и филтриране.

Класът Stack включва един последен метод: find e . Той работи, като го препраща

кодов блок към метода за търсене на масива @backtrace. Работи много като някои от

deck-of-cards методи, които сте виждали, които препращат метод към масив, съдържащ

картите, които съставляват тестето. Техники като тази ви позволяват да прецизирате интерфейса

лицето на вашите обекти, като използвате подлежащи обекти, за да им предоставите точно функциите-

националност, от която се нуждаят. (Ще видите конкретната полезност на find скоро.)

Сега, нека изпробваме CallerTools.

U пее C ALLER T OOLS МОДУЛ

Можете да използвате модифицирана версия на демонстрацията „x, y, z“ от раздел 15.4.1, за да изпробвате

CallerTools. Поставете този код във файл, наречен callertest.rb:

require\_relative 'callertools'

def x

у

край

def y

z

край

def z

стек = CallerTools :: Stack.new

Листинг 15.6 **CallerTools ::** Клас на **стека**

б

° С

д

д

|  |
| --- |
| **Страница 480** |

**480**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

поставя stack.report

край

х

Когато стартирате програмата, ще видите този изход:

callertest.rb 12

z

callertest.rb 8

у

callertest.rb 4

х

callertest.rb 16

<основна>

Нищо твърде изискано, но това е хубав програмен начин да се обърне по-скоро към трасирането на стека

отколкото да се налага да манипулирате струните всеки път директно. (Има много празно място в

началото на редовете, но би имало по-малко, ако файловите пътища са по-дълги - и

разбира се можете да настроите форматирането по вкус.)

Следващото в дневния ред и последната спирка за тази глава е проект, който се свързва заедно

редица техники, които разглеждахме: проследяване на стека, заявка за метод,

и обратно извикване, както и някои техники, които познавате от други места в книгата. Добре

напишете тестова рамка.

***15.5 Обратни обаждания и проверка на методи на практика***

В този раздел ще внедрим MicroTest, малка тестова рамка. Няма много

характеристики, но тези, които има, ще демонстрират част от силата и изразителността на

обратно извикване и техники за проверка, които току-що сте научили.

Първо, малко предистория.

***15.5.1 Фон на MicroTest: MiniTest***

Ruby се доставя с тестова рамка, наречена MiniTest. Използвате MiniTest, като напишете

клас, който наследява от класа MiniTest :: Unit :: TestCase и който съдържа meth-

ods, чиито имена започват с теста за низове. След това можете да посочите кой тест

методи, които искате да бъдат изпълнени, или подредете (както ще направим по-долу) за всеки тест с име

метод, който да се изпълнява автоматично, когато стартирате файла. Вътре в тези методи,

пишете *твърдения* . Истинността или лъжата на вашите твърдения определя дали вашите

тестове преминат.

Упражнението, което ще направим тук, е да напишем проста програма за тестване, базирана на някои от

същите принципи като MiniTest. За да ви помогнем да се ориентирате, първо ще разгледаме пълната информация

пример за MiniTest в действие и след това направете упражнението за внедряване.

Ще тестваме карти за раздаване. Следващият списък показва версия на клас за тесте от

карти. Колода се състои от масив от 52 низа, държани в променливата на екземпляра @cards.

Раздаването на една или повече карти означава изскачане на толкова много карти от горната част на тестето.

модул PlayingCards

РАНГОВЕ =% w {2 3 4 5 6 7 8 9 10 JQKA}

КОСТЮМИ =% w {клубове диаманти сърца пика}

клас Дек

Листинг 15.7 Внедряване на тестета с възможности за раздаване на карти

|  |
| --- |
| **Страница 481** |

**481**

***Обратни обаждания и проверка на методи на практика***

def инициализира

@cards = []

РАНГОВЕ.всеки | r |

КОСТЮМИ. Всеки |

@cards <<"# {r} от # {s}"

край

край

@ cards.shuffle!

край

раздаденост (n = 1)

@ cards.pop (n)

край

def размер

@ cards.size

край

край

край

Създаването на нов дек B включва инициализиране на @cards, вмъкване на 52 низа в него и shuf-

хвърляне на масива. Всеки низ приема формата „ *ранг* на *костюма* “, където *рангът* е един от ранга в

постоянният масив RANKS и *костюм* е един от SUITS. В раздаването от палубата c се връщаме

масив от n карти, където n е броят на картите, които се раздават и по подразбиране е 1.

Дотук добре. Сега да го тестваме. Въведете MiniTest. Следващият списък показва теста

код за класа на картите. Тестовият код предполага, че сте запазили кода на картите в a

отделен файл, наречен cards.rb в същата директория като файла с тестовия код (който можете

call cardtest.rb).

изискват „най-малко / единица“

изискват 'минитест / автоматично стартиране'

require\_relative 'карти'

class CardTest <MiniTest :: Unit :: TestCase

настройка на def

@deck = Игрални карти :: Deck.new

край

def test\_deal\_one

@ deck.deal

assert\_equal (51, @ deck.size)

край

def test\_deal\_many

@ deck.deal (5)

assert\_equal (47, @ deck.size)

край

край

Първият бизнес ред е да се изискват както най-малката / единична библиотека, така и

cards.rb файл B . Изискваме също така минимално / автоматично стартиране; тази функция кара MiniTest да

стартирайте тестовите методи, които среща, без да се налага да правим изрични извиквания на методи.

След това създаваме клас CardTest, който наследява от MiniTest :: Unit :: TestCase c . В

този клас дефинираме три метода. Първият е настройка d . Настройката на името на метода е

магия към MiniTest; ако е дефиниран, той се изпълнява преди всеки тест метод в тестовия клас.

Листинг 15.8 cardtest.rb: Тестване на точността на **раздаване** на **PlayingCards :: Deck**

б

° С

б

° С

д

д

е

ж

|  |
| --- |
| **482 серия** |

**482**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

Изпълнението на метода за настройка преди всеки метод на теста допринася за запазването на теста

методи, независими един от друг, и тази независимост е важна част от

архитектурата на тестовите пакети.

Сега идват двата метода за тестване, test\_deal\_one e и test\_deal\_many g .

Тези методи определят действителните тестове. Във всеки случай имаме сделка от колода и

след това изказване за размера на тестето след раздаването.

Не забравяйте, че настройката се изпълнява преди всеки тест метод, което означава @deck con-

осигурява пълен тесте от 52 карти за всеки метод.

Твърденията се извършват с помощта на метода assert\_equal f . Този метод

взема два аргумента. Ако двете са равни (използвайки ==, за да направите сравнението зад

сцени), твърдението успява. Ако не, то се проваля.

Изпълнете cardtest.rb от командния ред. Ето какво ще видите (вероятно с

различно семе и различни измервания на времето):

$ ruby ​​cardtest.rb

Опции за изпълнение: --seed 39562

# Текущи тестове:

..

Завършени тестове за 0.000784s, 2551.0204 теста / s, 2551.0204 твърдения / s.

2 теста, 2 твърдения, 0 неуспехи, 0 грешки, 0 пропуска

Последният ред ви казва, че има два метода, чиито имена започват с test

(2 теста) и общо две твърдения (двете извиквания на assert\_equal). Казва ти кожух

там и двете твърдения са преминали (без грешки) и че нищо не се е объркало драстично

(без грешки; грешката е нещо непоправимо като препратка към неизвестна променлива,

като има предвид, че неуспехът е неправилно твърдение). Той също така съобщава, че не са пропуснати тестове

(пропускането на тест е нещо, което можете да направите изрично с извикване на метода на пропускане).

Най-поразителното при стартирането на този тестов файл е, че в нито един момент нямате

за да създадете *екземпляр* на класа CardTest или изрично да извикате методите за тестване или настройката

метод. Благодарение на зареждането на функцията за автоматично стартиране, MiniTest разбра, че е така

трябва да изпълнява всички методи, чиито имена започват с тест, стартирайки настройката

метод преди всеки от тях. Това автоматично изпълнение - или поне подмножество от него - е

какво ще приложим в нашето упражнение.

***15.5.2 Посочване и внедряване на MicroTest***

Ето какво ще искаме от нашата програма MicroTest:

Автоматично изпълнение на метода за настройка и тестовите методи, въз основа на класа

наследство

Прост метод за твърдение, който или успява, или се проваля

Първата спецификация ще включва по-голямата част от работата.

Нуждаем се от клас, който след наследяване наблюдава новия подклас и изпълнява

методите в този подклас, както са дефинирани. В името на (относителната!) Простота,

ще ги изпълним в реда на дефиниция, което означава, че първо трябва да бъде дефинирана настройката.

|  |
| --- |
| **Страница 483** |

**483**

***Обратни обаждания и проверка на методи на практика***

Ето по-подробно описание на стъпките, необходими за внедряване на MicroTest:

1

Определете класа MicroTest.

2

Определете MicroTest.inherited.

3

Вътре наследен, класът за наследяване трябва ...

4

Дефинирайте свой собствен метод за обратно извикване, който трябва да ...

5

Инстанцирайте класа и изпълнете новия метод, ако започне с test, но

първо ...

6

Изпълнете метода за настройка, ако има такъв.

Ето неработещ, коментиран макет на MicroTest в Ruby:

клас MicroTest

def self.inherited (c)

c.class\_eval правя

def self.method\_added (m)

# Ако m започва с "тест"

# Създайте екземпляр на c

# Ако има метод за настройка

# Изпълнете настройката

# Изпълнете метода m

край

край

край

край

Тук има някаква логическа каскада. Вътре в MicroTest дефинираме self.inherited,

който получава като свой аргумент наследствения клас (новия подклас). След това влизаме

в обхвата на дефиницията на този клас, използвайки class\_eval. Вътре в този обхват ние прилагаме

method\_added, което ще се извиква всеки път, когато в метода се дефинира нов метод.

Писането на пълния код следва директно от коментарите в макета на кода.

Следният списък показва пълната версия на micro\_test.rb. Поставете го в същата посока

тори като callertools.rb.

require\_relative 'callertools'

клас MicroTest

def self.inherited (c)

c.class\_eval правя

def self.method\_added (m)

ако m = ~ / ^ тест /

obj = self.new

ако self.instance\_methods.include? (: настройка)

obj.setup

край

obj.send (m)

край

край

край

край

Листинг 15.9 **MicroTest** , клас за тестване, който емулира някои функционалности на MiniTest

б

° С

д

|  |
| --- |
| **Страница 484** |

**484**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

def твърди (твърдение)

ако твърдение

поставя „Твърдението е прието“

вярно

друго

поставя „Неуспешно твърдение:“

стек = CallerTools :: Stack.new

грешка = stack.find {| повикване | call.meth! ~ / отстояване /}

поставя провал

невярно

край

край

def assert\_equal (очаквано, действително)

резултат = твърди (очаква се == действително)

поставя "(# {действително} не е # {очаквано})" освен ако не е резултат

резултат

край

край

Вътре в обхвата на дефиницията на клас (class\_eval) на новия подклас дефинираме

method\_added и там е по-голямата част от действието. Ако методът се дефинира

започва с тест B , ние създаваме нов екземпляр на класа c . Ако методът за настройка е

дефиниран d , ние го наричаме в този случай. След това (независимо дали е имало настройка или не

метод; това е по избор), ние извикваме новодобавения метод, използвайки send, защото ние

не знам името на метода.

ЗАБЕЛЕЖКА Колкото и странно да изглежда (в светлината на традиционната представа за модел

съвпадение, което включва низове), m в операцията за съвпадение на шаблони m

= ~ / ^ тест / е символ, а не низ. Способността на символните обекти да съвпадат

самите срещу регулярните изрази е част от общия ход, който сме предприели

вече е отбелязано, че символите стават по-лесно взаимозаменяеми с

струни. Имайте предвид обаче важните разлики между двете, като

обяснено в глава 8.

Методът утверждаване проверява истинността на своя единствен аргумент e . Ако аргументът е верен

(в логически смисъл; не е задължително действителният обект да е истина), съобщението е

разпечатан, показващ успех. Ако твърдението се провали, отпечатването на съобщение свети

tle по-сложен. Създаваме обект CallerTools :: Stack и посочваме първия

Извикайте обект в този стек, чието име на метод не съдържа низа assert f .

Целта е да се уверим, че не съобщаваме за неизправността като възникнала в

assert метод, нито в метода assert\_equal (описан скоро). Не е

здрав; може да имате метод с твърдение в него, че сте искали грешка

съобщава от. Но това илюстрира вида на манипулацията, при която методът за намиране

CallerTools :: Stack позволява.

Вторият метод за твърдение, assert\_equal, тества за равенство между двете

аргументи ж . Това се прави чрез извикване на assert при сравнение. Ако резултатът не е верен,

се показва съобщение за грешка, показващо двата сравнени обекта h . Така или иначе - успех

cess or failure - резултатът от извикването на assert се връща от assert\_equal.

д

е

ж

з

|  |
| --- |
| **Страница 485** |

**485**

***Обобщение***

За да изпробвате MicroTest, поставете следния код във файл, наречен microcardtest.rb, и

стартирайте го от командния ред:

require\_relative 'микротест'

require\_relative 'карти'

клас CardTest <MicroTest

настройка на def

@deck = Игрални карти :: Deck.new

край

def test\_deal\_one

@ deck.deal

assert\_equal (51, @ deck.size)

край

def test\_deal\_many

@ deck.deal (5)

assert\_equal (47, @ deck.size)

край

край

Както можете да видите, този код е почти идентичен с тестовия файл на MiniTest, който написахме преди.

Единствените разлики са имената на тестовата библиотека и родителския тестов клас. И когато

стартирате кода, получавате тези малко неясни, но обнадеждаващи резултати:

Твърдението премина

Твърдението премина

Ако искате да видите грешка, променете 51 на 50 в test\_deal\_one:

Твърдението не бе успешно:

microcardtest.rb 11 test\_deal\_one

(51 не е 50)

Твърдението премина

MicroTest няма да замести MiniTest скоро, но прави няколко от най-много

магически неща, които MiniTest прави. Всичко това става възможно от самоанализа на Руби и

съоръжения за обратно извикване, техники, които предоставят изключителна мощ и гъвкавост във вашите ръце.

***15.6 Обобщение***

В тази глава сте виждали

Прихващане на методи с method\_missing

Куки за изпълнение и обратно извикване за обекти, класове и модули

Запитване на обекти за техните методи по различни критерии

Прихващане на препратки към неизвестни константи

Следи от стека

Писане на рамката на MicroTest

В тази глава разгледахме много основания и практикувахме обхванатите техники

тук ще допринесе значително за вашата основа като рубист. Погледнахме прихващането

неизвестни съобщения с method\_missing, заедно с други куки за изпълнение и call-

гръб като Module.included, Module.extended и Class.inherited. Главата

|  |
| --- |
| **486 серия** |

**486**

C ГЛАВА 15 Обратни ***обаждания, куки и самонаблюдение по време на изпълнение***

също ни включи в методологичното запитване в различните му нюанси: публичен, защитен, частен;

клас, екземпляр, единичен. Видяхте няколко примера за това как този вид заявки

може да ви помогне да извлечете информация за това как Ruby прави своя клас, модул и

организация на метода.

Последната обща тема беше обработката на следи от стека, които използвахме в

Модул CallerTools. Главата завърши с разширено упражнение, състоящо се от

внедряване на клас MicroTest, който събра редица теми и

теми от тази глава и другаде.

Прегледахме материала методично и съзнателно, както подобава на

заземяване или подготовка. Но ако погледнете резултатите, особено MicroTest, вие

можете да видите колко мощност ви дава Руби в замяна на относително малко усилия. Това е

защо си струва да се знае дори за това, което може да изглежда като магията или „мета“ частите

Руби. Наистина не са - всичко е Руби и след като усвоите принципите на класа

и обектна структура и взаимоотношения, всичко останало следва.

И това е! Насладете се на своята основаност като рубист и многото структури

ще изградите върху основата, която сте придобили чрез тази книга.

|  |
| --- |
| **Страница 487** |

**487**

*индекс*

Символи

`(обратен удар) [450](https://translate.googleusercontent.com/translate_f#450) –451

^ (Карета); [229](https://translate.googleusercontent.com/translate_f#229)

^ = (битов ИЗКЛЮЧИТЕЛЕН ИЛИ) оператор [95](https://translate.googleusercontent.com/translate_f#95)

\_ (променлива в [долната](https://translate.googleusercontent.com/translate_f#476) черта) [476](https://translate.googleusercontent.com/translate_f#476)

: (единично дебело черво) [193](https://translate.googleusercontent.com/translate_f#193)

:: (двойно дебело черво) [86](https://translate.googleusercontent.com/translate_f#86) –87, 116, 138

! (взрив)

методи [197](https://translate.googleusercontent.com/translate_f#197)

оператор [154,](https://translate.googleusercontent.com/translate_f#154) 196

! @ метод [196](https://translate.googleusercontent.com/translate_f#196)

! = метод [212](https://translate.googleusercontent.com/translate_f#212)

. (точка)

оператор [9,](https://translate.googleusercontent.com/translate_f#9) 37

в регулярни изрази [351](https://translate.googleusercontent.com/translate_f#351)

заместващ характер [334](https://translate.googleusercontent.com/translate_f#334)

... (конструктор на буквален обхват) [193](https://translate.googleusercontent.com/translate_f#193)

'' (конструктор на литерални низове) [193](https://translate.googleusercontent.com/translate_f#193)

- оператор (тире), персонализиране [196](https://translate.googleusercontent.com/translate_f#196)

[] скоби [258](https://translate.googleusercontent.com/translate_f#258)

Конструктор на [литерален](https://translate.googleusercontent.com/translate_f#193) масив [193](https://translate.googleusercontent.com/translate_f#193)

метод на конструктор на хеш [268](https://translate.googleusercontent.com/translate_f#268)

{} къдрави скоби

до / край срещу [170](https://translate.googleusercontent.com/translate_f#170)

Конструктор на [буква](https://translate.googleusercontent.com/translate_f#193) Hash [193](https://translate.googleusercontent.com/translate_f#193)

модел брой повторения [345](https://translate.googleusercontent.com/translate_f#345)

при интерполация на низове [170](https://translate.googleusercontent.com/translate_f#170)

@ (при знак) [67](https://translate.googleusercontent.com/translate_f#67)

[196](https://translate.googleusercontent.com/translate_f#196) метод

\* (звездичка)

разширява масива в гол списък [203](https://translate.googleusercontent.com/translate_f#203)

префикс оператор [202](https://translate.googleusercontent.com/translate_f#202)

до\_а и [202](https://translate.googleusercontent.com/translate_f#202)

размазващ ефект от [203](https://translate.googleusercontent.com/translate_f#203)

различни имена за [202](https://translate.googleusercontent.com/translate_f#202)

нулев или повече квантор [343](https://translate.googleusercontent.com/translate_f#343)

/ (знак с наклонена черта) [74](https://translate.googleusercontent.com/translate_f#74)

// (конструктор на литерал на регулярен израз) [193](https://translate.googleusercontent.com/translate_f#193) ,

[332](https://translate.googleusercontent.com/translate_f#332)

\ (наклонена черта) [140,](https://translate.googleusercontent.com/translate_f#140) 221, 334, 392

&& = (и е равно) оператор [95](https://translate.googleusercontent.com/translate_f#95)

# (знак за хеш) [86](https://translate.googleusercontent.com/translate_f#86)

# {...} интерполация [220](https://translate.googleusercontent.com/translate_f#220) , 351

% (конструктор на литерални низове) [71](https://translate.googleusercontent.com/translate_f#71)

% {} (знак за процент и разделител) [222](https://translate.googleusercontent.com/translate_f#222)

% = (по модул) оператор [95](https://translate.googleusercontent.com/translate_f#95)

+ (знак плюс)

ден-мъдър конверсия [251](https://translate.googleusercontent.com/translate_f#251)

метод [194,](https://translate.googleusercontent.com/translate_f#194) 226

оператор, персонализиране [196](https://translate.googleusercontent.com/translate_f#196)

[196](https://translate.googleusercontent.com/translate_f#196) метод

++ = (повишаване на мощност) оператор [95](https://translate.googleusercontent.com/translate_f#95)

+ = оператор [95](https://translate.googleusercontent.com/translate_f#95)

<< оператор [223,](https://translate.googleusercontent.com/translate_f#223) 226, 262, 282

<=> метод [230,](https://translate.googleusercontent.com/translate_f#230) 310

= (знак за равенство) [70,](https://translate.googleusercontent.com/translate_f#70) 73, 225, 261, 269

== метод [212,](https://translate.googleusercontent.com/translate_f#212) 230, 266

=== метод с три [качества 161–](https://translate.googleusercontent.com/translate_f#161) 162, 194, 357

=> оператор [183](https://translate.googleusercontent.com/translate_f#183) , 267–268

= ~ оператор за съвпадение на шаблони [333](https://translate.googleusercontent.com/translate_f#333)

| (условно съвпадение на регулярен израз) [348](https://translate.googleusercontent.com/translate_f#348)

| = (побитово ИЛИ) оператор [95](https://translate.googleusercontent.com/translate_f#95)

|| = (или-равно) оператор [95](https://translate.googleusercontent.com/translate_f#95)

$? променлива [450](https://translate.googleusercontent.com/translate_f#450) –451

$: глобална променлива [130](https://translate.googleusercontent.com/translate_f#130) , 475

$ / глобална променлива [308](https://translate.googleusercontent.com/translate_f#308)

$ ~ глобална променлива [475](https://translate.googleusercontent.com/translate_f#475)

$$ глобална променлива [130](https://translate.googleusercontent.com/translate_f#130)

$ 0 глобална променлива [130](https://translate.googleusercontent.com/translate_f#130) , 337, 475

|  |
| --- |
| **Страница 488** |

ИНДЕКС

**488**

A

\ Котва в началото на низа [347](https://translate.googleusercontent.com/translate_f#347)

достъп, частен / защитен / публичен [144](https://translate.googleusercontent.com/translate_f#144)

метод за достъп [143](https://translate.googleusercontent.com/translate_f#143)

Библиотека ActiveSupport [405](https://translate.googleusercontent.com/translate_f#405)

адитивна промяна на основната функционалност

[405](https://translate.googleusercontent.com/translate_f#405) –407

псевдоним ключова дума [401](https://translate.googleusercontent.com/translate_f#401) , 407

метод [407](https://translate.googleusercontent.com/translate_f#407)

метод на предците [397](https://translate.googleusercontent.com/translate_f#397)

и-равно оператор [95](https://translate.googleusercontent.com/translate_f#95)

анонимни функции, Proc клас [419](https://translate.googleusercontent.com/translate_f#419)

Archdir [17](https://translate.googleusercontent.com/translate_f#17)

ARGF.class обект [472](https://translate.googleusercontent.com/translate_f#472)

\* args параметър [110](https://translate.googleusercontent.com/translate_f#110)

списък с аргументи [170](https://translate.googleusercontent.com/translate_f#170)

ArgumentError [180,](https://translate.googleusercontent.com/translate_f#180) 185

аргументи

присвояване на променлива [53](https://translate.googleusercontent.com/translate_f#53)

стойност по подразбиране [49](https://translate.googleusercontent.com/translate_f#49)

дефинирани [9,](https://translate.googleusercontent.com/translate_f#9) 38

на име [276](https://translate.googleusercontent.com/translate_f#276) –277

по избор [48](https://translate.googleusercontent.com/translate_f#48)

поръчка [50](https://translate.googleusercontent.com/translate_f#50)

общ преглед [48](https://translate.googleusercontent.com/translate_f#48)

за Proc обекти [428](https://translate.googleusercontent.com/translate_f#428)

справка и [58](https://translate.googleusercontent.com/translate_f#58)

изисква се [48](https://translate.googleusercontent.com/translate_f#48)

правила за [52](https://translate.googleusercontent.com/translate_f#52)

синтаксис [53](https://translate.googleusercontent.com/translate_f#53)

аритметична операция [195](https://translate.googleusercontent.com/translate_f#195) , 244–246

Клас масив

буквално конструктор [193](https://translate.googleusercontent.com/translate_f#193)

Нов метод [257](https://translate.googleusercontent.com/translate_f#257)

Масив # [] = метод [261](https://translate.googleusercontent.com/translate_f#261)

Метод за размер на масив # [174](https://translate.googleusercontent.com/translate_f#174)

масиви [95](https://translate.googleusercontent.com/translate_f#95)

[] метод [261](https://translate.googleusercontent.com/translate_f#261)

добавяне на обект [262](https://translate.googleusercontent.com/translate_f#262)

комбинации [264](https://translate.googleusercontent.com/translate_f#264)

компактен метод [266](https://translate.googleusercontent.com/translate_f#266)

компактен! метод [266](https://translate.googleusercontent.com/translate_f#266)

метод Concat [264](https://translate.googleusercontent.com/translate_f#264)

методи за преобразуване [202](https://translate.googleusercontent.com/translate_f#202)

метод брой [267](https://translate.googleusercontent.com/translate_f#267)

създаване на [257](https://translate.googleusercontent.com/translate_f#257)

метод [376](https://translate.googleusercontent.com/translate_f#376)

определяне дали nil е елемент

[292](https://translate.googleusercontent.com/translate_f#292)

празна? метод [267](https://translate.googleusercontent.com/translate_f#267)

enumerables и [293](https://translate.googleusercontent.com/translate_f#293)

метод за намиране [292](https://translate.googleusercontent.com/translate_f#292)

първи метод [267](https://translate.googleusercontent.com/translate_f#267)

сплескване [265](https://translate.googleusercontent.com/translate_f#265)

метод приемателното [261](https://translate.googleusercontent.com/translate_f#261)

метод на grep [468](https://translate.googleusercontent.com/translate_f#468)

хеш срещу [255](https://translate.googleusercontent.com/translate_f#255) –256

включва? метод [239,](https://translate.googleusercontent.com/translate_f#239) 267

вътрешен, гнездене [265](https://translate.googleusercontent.com/translate_f#265)

на място картографиране [306](https://translate.googleusercontent.com/translate_f#306)

вмъкване на елементи [261](https://translate.googleusercontent.com/translate_f#261) –264

метод на присъединяване [265,](https://translate.googleusercontent.com/translate_f#265) 472

последен метод [267](https://translate.googleusercontent.com/translate_f#267)

метод на дължина [267](https://translate.googleusercontent.com/translate_f#267)

буквално конструктор [258](https://translate.googleusercontent.com/translate_f#258)

метод на картата [479](https://translate.googleusercontent.com/translate_f#479)

карта! метод [306](https://translate.googleusercontent.com/translate_f#306)

общ преглед [256](https://translate.googleusercontent.com/translate_f#256)

метод пакет [326](https://translate.googleusercontent.com/translate_f#326)

поп метод [263](https://translate.googleusercontent.com/translate_f#263)

печат [49](https://translate.googleusercontent.com/translate_f#49)

метод тласък [262](https://translate.googleusercontent.com/translate_f#262)

заявка [266](https://translate.googleusercontent.com/translate_f#266) –267

[294.](https://translate.googleusercontent.com/translate_f#294) отхвърляне

премахване на дублиращи се елементи [266](https://translate.googleusercontent.com/translate_f#266)

премахване на елементи [261](https://translate.googleusercontent.com/translate_f#261) –264

замести метод [264](https://translate.googleusercontent.com/translate_f#264)

извличане на елементи [261](https://translate.googleusercontent.com/translate_f#261) –264

обратен метод [265](https://translate.googleusercontent.com/translate_f#265)

обратен! метод [265](https://translate.googleusercontent.com/translate_f#265)

[261.](https://translate.googleusercontent.com/translate_f#261) настройка / получаване на множество елементи

метод на смяна [263](https://translate.googleusercontent.com/translate_f#263)

метод за размер [267](https://translate.googleusercontent.com/translate_f#267)

метод за отрязък [262](https://translate.googleusercontent.com/translate_f#262)

парче! метод [262](https://translate.googleusercontent.com/translate_f#262)

метод на сортиране [468,](https://translate.googleusercontent.com/translate_f#468) 472

от низове [260](https://translate.googleusercontent.com/translate_f#260)

метод to\_s [200](https://translate.googleusercontent.com/translate_f#200) –201

трансформации [265](https://translate.googleusercontent.com/translate_f#265)

метод Uniq [266](https://translate.googleusercontent.com/translate_f#266)

uniq! метод [266](https://translate.googleusercontent.com/translate_f#266)

като универсален формат на резултата [293](https://translate.googleusercontent.com/translate_f#293)

unshift метод [262](https://translate.googleusercontent.com/translate_f#262)

стойности\_ при метод [262](https://translate.googleusercontent.com/translate_f#262)

индексиране с нулев произход [261](https://translate.googleusercontent.com/translate_f#261)

възходящ метод за имена на пътища [381](https://translate.googleusercontent.com/translate_f#381)

ASCII код на низ [230](https://translate.googleusercontent.com/translate_f#230)

метод на assert\_equal [482](https://translate.googleusercontent.com/translate_f#482)

твърдение [480](https://translate.googleusercontent.com/translate_f#480)

възлагане

в условен тест [159](https://translate.googleusercontent.com/translate_f#159)

синтаксис [157](https://translate.googleusercontent.com/translate_f#157)

променлива [6](https://translate.googleusercontent.com/translate_f) , 74

асоциативен масив [255](https://translate.googleusercontent.com/translate_f#255)

звездичка

разширява масива в гол списък [203](https://translate.googleusercontent.com/translate_f#203)

префикс оператор [202](https://translate.googleusercontent.com/translate_f#202)

до\_а и [202](https://translate.googleusercontent.com/translate_f#202)

размазващ ефект от [203](https://translate.googleusercontent.com/translate_f#203)

|  |
| --- |
| **489 серия** |

ИНДЕКС

**489**

звездичка *(продължение)*

различни имена за [202](https://translate.googleusercontent.com/translate_f#202)

заро-или-повече квантор [343](https://translate.googleusercontent.com/translate_f#343)

при знак [67](https://translate.googleusercontent.com/translate_f#67)

атрибут atime [375](https://translate.googleusercontent.com/translate_f#375)

attr\_ \* методи [74](https://translate.googleusercontent.com/translate_f#74) , 240

attr\_accessor метод [76](https://translate.googleusercontent.com/translate_f#76)

attr\_reader метод [76](https://translate.googleusercontent.com/translate_f#76)

attr\_writer метод [76](https://translate.googleusercontent.com/translate_f#76)

атрибути [74](https://translate.googleusercontent.com/translate_f#74) –75, 77

Б.

\ b котва на границата на думи [347](https://translate.googleusercontent.com/translate_f#347)

обратна наклонена черта [140,](https://translate.googleusercontent.com/translate_f#140) 221, 334, 392

обратни връзки [450](https://translate.googleusercontent.com/translate_f#450) –451

метод на обратното проследяване [183](https://translate.googleusercontent.com/translate_f#183)

bang (!) оператор [154](https://translate.googleusercontent.com/translate_f#154) , 196–197, 231

конвенции и най-добри практики [198](https://translate.googleusercontent.com/translate_f#198)

разглеждането на специфични обекти [468](https://translate.googleusercontent.com/translate_f#468)

не винаги разрушителни [199](https://translate.googleusercontent.com/translate_f#199)

голи списък [203](https://translate.googleusercontent.com/translate_f#203)

стил без говор [10](https://translate.googleusercontent.com/translate_f#10)

метод на базово име [381](https://translate.googleusercontent.com/translate_f#381) –382

BasicObject клас [45](https://translate.googleusercontent.com/translate_f#45) , 80, 101, 412–416

начало / край на блокове с ключова дума

[166](https://translate.googleusercontent.com/translate_f#166) –167

[347.](https://translate.googleusercontent.com/translate_f#347) котва в началото на низа

поведение, капсулиране в модул [94](https://translate.googleusercontent.com/translate_f#94)

Клас Bignum [244](https://translate.googleusercontent.com/translate_f#244)

Клас на подвързване [434](https://translate.googleusercontent.com/translate_f#434)

битови оператори [95](https://translate.googleusercontent.com/translate_f#95) , 195

BlankSlate клас [80](https://translate.googleusercontent.com/translate_f#80)

блок параметър [170,](https://translate.googleusercontent.com/translate_f#170) 176, 179

block\_given? метод [415](https://translate.googleusercontent.com/translate_f#415)

блок, тестване за наличие на [415](https://translate.googleusercontent.com/translate_f#415)

блокове код

улавяне, както Proc [422](https://translate.googleusercontent.com/translate_f#422)

предмети и [421](https://translate.googleusercontent.com/translate_f#421)

Proc обекти и [420](https://translate.googleusercontent.com/translate_f#420) –421

Булево

Булеви обекти [207](https://translate.googleusercontent.com/translate_f#207)

сравнителен тест за диапазон [279](https://translate.googleusercontent.com/translate_f#279)

всеки обект има истинско или фалшиво състояние [207](https://translate.googleusercontent.com/translate_f#207)

изразяващи се в метод [42](https://translate.googleusercontent.com/translate_f#42)

предмети [207](https://translate.googleusercontent.com/translate_f#207)

посочва, [207](https://translate.googleusercontent.com/translate_f#207)

стойности, само nil и false имат false

стойност [211](https://translate.googleusercontent.com/translate_f#211)

скоби

Конструктор на [литерален](https://translate.googleusercontent.com/translate_f#193) масив [193](https://translate.googleusercontent.com/translate_f#193)

метод на конструктор на хеш [268](https://translate.googleusercontent.com/translate_f#268)

ключова дума [165](https://translate.googleusercontent.com/translate_f#165)

Builder библиотека [413](https://translate.googleusercontent.com/translate_f#413)

Builder :: XmlMarkup метод [414](https://translate.googleusercontent.com/translate_f#414)

° С

? c [374](https://translate.googleusercontent.com/translate_f#374)

% c (спецификатор на формат на датата) [221](https://translate.googleusercontent.com/translate_f#221) , 250

C директория с разширения [17](https://translate.googleusercontent.com/translate_f#17)

-c флаг [12](https://translate.googleusercontent.com/translate_f#12) , 23

C стандартна библиотека [361](https://translate.googleusercontent.com/translate_f#361)

метод на повикване [433](https://translate.googleusercontent.com/translate_f#433)

[извикващи се обекти.](https://translate.googleusercontent.com/translate_f#455) *Вижте* изпълними обекти

обратно извикване

дефиниран [457](https://translate.googleusercontent.com/translate_f#457)

използвайки [480](https://translate.googleusercontent.com/translate_f#480)

метод абонати [476](https://translate.googleusercontent.com/translate_f#476)

Модул CallerTools [478](https://translate.googleusercontent.com/translate_f#478) –479

метод с главни букви [231](https://translate.googleusercontent.com/translate_f#231)

метод [422](https://translate.googleusercontent.com/translate_f#422)

Казалс, Пабло [370](https://translate.googleusercontent.com/translate_f#370)

случай

равенство [195](https://translate.googleusercontent.com/translate_f#195) , 357

трансформация [231](https://translate.googleusercontent.com/translate_f#231)

делото

поведение [162](https://translate.googleusercontent.com/translate_f#162)

равенство [161](https://translate.googleusercontent.com/translate_f#161)

общ преглед [160](https://translate.googleusercontent.com/translate_f#160)

върната стойност [164](https://translate.googleusercontent.com/translate_f#164)

модел на тестване съответства на [358](https://translate.googleusercontent.com/translate_f#358)

тест за истина [163](https://translate.googleusercontent.com/translate_f#163)

метод на центъра [232](https://translate.googleusercontent.com/translate_f#232)

верижното [316](https://translate.googleusercontent.com/translate_f#316)

промяна на обект [408](https://translate.googleusercontent.com/translate_f#408)

знаци

буквено-цифров, съответстващ [336](https://translate.googleusercontent.com/translate_f#336)

кодиране [235](https://translate.googleusercontent.com/translate_f#235)

бягство [221,](https://translate.googleusercontent.com/translate_f#221) 334

буквално, в модели [334](https://translate.googleusercontent.com/translate_f#334)

небели, съответстващи на [336](https://translate.googleusercontent.com/translate_f#336)

връщане в потока за въвеждане на файлове [366](https://translate.googleusercontent.com/translate_f#366)

чат сървър, използващ гнезда и нишки

[443](https://translate.googleusercontent.com/translate_f#443) –444

метод на chdir [378](https://translate.googleusercontent.com/translate_f#378)

проверка на синтаксиса [23](https://translate.googleusercontent.com/translate_f#23)

метод на chomp [233](https://translate.googleusercontent.com/translate_f#233)

метод котлет [233](https://translate.googleusercontent.com/translate_f#233)

chr метод [230](https://translate.googleusercontent.com/translate_f#230)

клас << нотация на обект [393](https://translate.googleusercontent.com/translate_f#393) –394

Клас клас [81](https://translate.googleusercontent.com/translate_f#81)

парадокс клас / обект [82](https://translate.googleusercontent.com/translate_f#82)

единични класове като екземпляри от [392](https://translate.googleusercontent.com/translate_f#392)

писане на методи на клас на [391](https://translate.googleusercontent.com/translate_f#391)

ключова дума [63,](https://translate.googleusercontent.com/translate_f#63) 81, 392

клас методи [29](https://translate.googleusercontent.com/translate_f#29) , 399

добавяне с удължаване [409](https://translate.googleusercontent.com/translate_f#409)

дефиниран [83](https://translate.googleusercontent.com/translate_f#83)

дефиниране с клас <<[393](https://translate.googleusercontent.com/translate_f#393)

метод на екземпляр срещу [85](https://translate.googleusercontent.com/translate_f#85)

|  |
| --- |
| **490. стр** |

ИНДЕКС

**490**

методи на клас *(продължение)*

използване на термин [84](https://translate.googleusercontent.com/translate_f#84)

писане [84](https://translate.googleusercontent.com/translate_f#84)

обекти от клас

извикващи методи [82](https://translate.googleusercontent.com/translate_f#82)

създаване [81](https://translate.googleusercontent.com/translate_f#81)

поддържане на състояние по клас със [143](https://translate.googleusercontent.com/translate_f#143)

променливи на класа [7](https://translate.googleusercontent.com/translate_f#7)

класова йерархия и [141](https://translate.googleusercontent.com/translate_f#141)

общ преглед [138](https://translate.googleusercontent.com/translate_f#138) –141

плюсове и минуси [142](https://translate.googleusercontent.com/translate_f#142)

класове

class\_eval метод [433](https://translate.googleusercontent.com/translate_f#433) , 437–438, 483

ядро, модифициращо [400](https://translate.googleusercontent.com/translate_f#400)

дефинирани [10,](https://translate.googleusercontent.com/translate_f#10) 66

дефиниционен блок, локален обхват [132](https://translate.googleusercontent.com/translate_f#132)

дизайн и именуване [114](https://translate.googleusercontent.com/translate_f#114)

разграничаване от модули [117](https://translate.googleusercontent.com/translate_f#117)

изключение [184,](https://translate.googleusercontent.com/translate_f#184) 186

фалшив аргумент към заявка за метод

методи [217](https://translate.googleusercontent.com/translate_f#217)

йерархия и променливи на класа [141](https://translate.googleusercontent.com/translate_f#141)

наследен метод [464,](https://translate.googleusercontent.com/translate_f#464) 483

екземплярни методи

получаване на [471](https://translate.googleusercontent.com/translate_f#471)

гледане [217](https://translate.googleusercontent.com/translate_f#217)

като приемник на съобщения [81](https://translate.googleusercontent.com/translate_f#81)

смесващ модул в [96](https://translate.googleusercontent.com/translate_f#96) –98

модул срещу [115](https://translate.googleusercontent.com/translate_f#115)

разкрой [116](https://translate.googleusercontent.com/translate_f#116)

числови [244](https://translate.googleusercontent.com/translate_f#244)

предмети [81](https://translate.googleusercontent.com/translate_f#81)

общ преглед [63](https://translate.googleusercontent.com/translate_f#63)

Метод private\_instance\_methods [217](https://translate.googleusercontent.com/translate_f#217)

метод [217](https://translate.googleusercontent.com/translate_f#217)

метод [217](https://translate.googleusercontent.com/translate_f#217)

[65](https://translate.googleusercontent.com/translate_f#65) отваряне

subclassing [464](https://translate.googleusercontent.com/translate_f#464)

ясен метод [233](https://translate.googleusercontent.com/translate_f#233)

метод на клониране [59](https://translate.googleusercontent.com/translate_f#59)

метод за затваряне, за файлове [365](https://translate.googleusercontent.com/translate_f#365)

затваряния [178](https://translate.googleusercontent.com/translate_f#178) , 426–428

код

условно изпълнение [153](https://translate.googleusercontent.com/translate_f#153)

повторение, намаляване на [75](https://translate.googleusercontent.com/translate_f#75)

кодови блокове [81](https://translate.googleusercontent.com/translate_f#81) , 165

{} (къдрава скоба) разделители [169](https://translate.googleusercontent.com/translate_f#169)

като затваряне [178](https://translate.googleusercontent.com/translate_f#178)

разделител предимство [171](https://translate.googleusercontent.com/translate_f#171)

do / end срещу {} разделители [165](https://translate.googleusercontent.com/translate_f#165) , 170

игнорирани от методи [171](https://translate.googleusercontent.com/translate_f#171)

в цикъл [165](https://translate.googleusercontent.com/translate_f#165)

итератори и [168](https://translate.googleusercontent.com/translate_f#168)

не е част от списък с аргументи [169](https://translate.googleusercontent.com/translate_f#169)

обвързване на параметри [172](https://translate.googleusercontent.com/translate_f#172)

семантика на параметри спрямо методи [176](https://translate.googleusercontent.com/translate_f#176)

параметри спрямо нови локални променливи [177](https://translate.googleusercontent.com/translate_f#177)

скоби в списъка с параметри [256](https://translate.googleusercontent.com/translate_f#256)

връщане на стойности към метод [175](https://translate.googleusercontent.com/translate_f#175)

като синтактични конструкции [169](https://translate.googleusercontent.com/translate_f#169) –170

временни локални променливи в [178](https://translate.googleusercontent.com/translate_f#178) –179

използвайки за обхват файлови операции [370](https://translate.googleusercontent.com/translate_f#370)

правила за обхват на променливите [177](https://translate.googleusercontent.com/translate_f#177)

отстъпване от метод [170](https://translate.googleusercontent.com/translate_f#170)

колекция, обединяване в набор [284](https://translate.googleusercontent.com/translate_f#284)

дебело черво

постоянно търсене маркер [116](https://translate.googleusercontent.com/translate_f#116)

оператор с постоянна разделителна способност [86](https://translate.googleusercontent.com/translate_f#86) –87

сепаратор [138](https://translate.googleusercontent.com/translate_f#138)

Конструктор на буквален символ [193](https://translate.googleusercontent.com/translate_f#193)

команда, опитвайки се без изпълнение на [380](https://translate.googleusercontent.com/translate_f#380)

превключватели на командния ред [23,](https://translate.googleusercontent.com/translate_f#23) 25

коментар, магия [236](https://translate.googleusercontent.com/translate_f#236)

коментари [7](https://translate.googleusercontent.com/translate_f)

търговски метод [247](https://translate.googleusercontent.com/translate_f#247)

компактен метод [266](https://translate.googleusercontent.com/translate_f#266)

Сравним модул [230](https://translate.googleusercontent.com/translate_f#230)

общ преглед [213](https://translate.googleusercontent.com/translate_f#213) –214

сортиране и [310](https://translate.googleusercontent.com/translate_f#310)

методи за сравнение [213](https://translate.googleusercontent.com/translate_f#213) –214

Комплексен метод [258](https://translate.googleusercontent.com/translate_f#258)

метод Concat [264](https://translate.googleusercontent.com/translate_f#264)

условно

алтернативни обозначения [160](https://translate.googleusercontent.com/translate_f#160)

оператор на присвояване [195](https://translate.googleusercontent.com/translate_f#195)

изпълнение на код [153](https://translate.googleusercontent.com/translate_f#153)

изпълнение [6](https://translate.googleusercontent.com/translate_f)

израз в синтаксиса на присвояване [157](https://translate.googleusercontent.com/translate_f#157)

логика [43](https://translate.googleusercontent.com/translate_f#43)

модификатор [156](https://translate.googleusercontent.com/translate_f#156)

регуларен израз съвпада [348](https://translate.googleusercontent.com/translate_f#348)

тест, задание в [159](https://translate.googleusercontent.com/translate_f#159)

[465](https://translate.googleusercontent.com/translate_f#465) серия

константи [8](https://translate.googleusercontent.com/translate_f#8) , 63

налагане на абсолютен път [137](https://translate.googleusercontent.com/translate_f#137)

интроспекция [474](https://translate.googleusercontent.com/translate_f#474)

справка [137](https://translate.googleusercontent.com/translate_f#137)

промяна [89](https://translate.googleusercontent.com/translate_f#89)

общ преглед [86](https://translate.googleusercontent.com/translate_f#86)

предварително дефиниран [87](https://translate.googleusercontent.com/translate_f#87)

преназначаване [88](https://translate.googleusercontent.com/translate_f#88)

разрешаване и обхват [136](https://translate.googleusercontent.com/translate_f#136)

използвайки [87](https://translate.googleusercontent.com/translate_f#87)

конструктори, буквал [192](https://translate.googleusercontent.com/translate_f#192) –193

контекст за Proc обект [427](https://translate.googleusercontent.com/translate_f#427)

конверсии

масиви [202](https://translate.googleusercontent.com/translate_f#202)

методи за [200](https://translate.googleusercontent.com/translate_f#200)

числен [203](https://translate.googleusercontent.com/translate_f#203)

типизиране срещу [205](https://translate.googleusercontent.com/translate_f#205)

|  |
| --- |
| **Страница 491** |

ИНДЕКС

**491**

основен клас, модифициращ [400](https://translate.googleusercontent.com/translate_f#400)

основна функционалност

адитивни промени [405](https://translate.googleusercontent.com/translate_f#405) –406

добавки / преминаващи хибридни промени

[407](https://translate.googleusercontent.com/translate_f#407)

модифициране безопасно [405](https://translate.googleusercontent.com/translate_f#405)

проходни промени [406](https://translate.googleusercontent.com/translate_f#406) –408

промени на обект с удължаване [408](https://translate.googleusercontent.com/translate_f#408)

рискове от промяна на [401](https://translate.googleusercontent.com/translate_f#401)

надмноженство задвижване замени [408](https://translate.googleusercontent.com/translate_f#408)

основен метод, а не пренаписване за изпълнение

неочаквано [405](https://translate.googleusercontent.com/translate_f#405)

основен модул, модифициращ [400](https://translate.googleusercontent.com/translate_f#400)

основен обект, променящо се поведение на [411](https://translate.googleusercontent.com/translate_f#411)

метод брой [229](https://translate.googleusercontent.com/translate_f#229)

cp метод [379](https://translate.googleusercontent.com/translate_f#379)

метод на крипта [234](https://translate.googleusercontent.com/translate_f#234)

фигурни скоби [165](https://translate.googleusercontent.com/translate_f#165)

до / край срещу [170](https://translate.googleusercontent.com/translate_f#170)

Конструктор на [буква](https://translate.googleusercontent.com/translate_f#193) Hash [193](https://translate.googleusercontent.com/translate_f#193)

модел брой повторения [345](https://translate.googleusercontent.com/translate_f#345)

в интерполация на низове [227](https://translate.googleusercontent.com/translate_f#227)

текущ обект [120](https://translate.googleusercontent.com/translate_f#120)

д

d [374](https://translate.googleusercontent.com/translate_f#374)

% d (спецификатор на формат на датата) [250](https://translate.googleusercontent.com/translate_f#250)

опасните методи [197](https://translate.googleusercontent.com/translate_f#197)

тире оператор [196](https://translate.googleusercontent.com/translate_f#196)

данни, нормализиращи [73](https://translate.googleusercontent.com/translate_f#73)

Клас на дати [246](https://translate.googleusercontent.com/translate_f#246) –247

дати

аритметика [251](https://translate.googleusercontent.com/translate_f#251)

методи за преобразуване [251](https://translate.googleusercontent.com/translate_f#251)

спецификатори на формат [250](https://translate.googleusercontent.com/translate_f#250)

методи за форматиране [249](https://translate.googleusercontent.com/translate_f#249) –251

итерация в обхват [252](https://translate.googleusercontent.com/translate_f#252)

обект, екземпляр [247](https://translate.googleusercontent.com/translate_f#247)

операции на [246](https://translate.googleusercontent.com/translate_f#246)

синтактична конструктор [247](https://translate.googleusercontent.com/translate_f#247)

методи заявка [249](https://translate.googleusercontent.com/translate_f#249)

Време за среща

гражданското конструктор [248](https://translate.googleusercontent.com/translate_f#248)

методи за преобразуване [251](https://translate.googleusercontent.com/translate_f#251)

методи за форматиране [249](https://translate.googleusercontent.com/translate_f#249) –251

итерация в обхват [252](https://translate.googleusercontent.com/translate_f#252)

Новият конструктор [248](https://translate.googleusercontent.com/translate_f#248)

сега конструктор [248](https://translate.googleusercontent.com/translate_f#248)

обект, екземпляр [248](https://translate.googleusercontent.com/translate_f#248)

синтактична конструктор [248](https://translate.googleusercontent.com/translate_f#248)

методи заявка [249](https://translate.googleusercontent.com/translate_f#249)

сек метод [249](https://translate.googleusercontent.com/translate_f#249)

втори метод [249](https://translate.googleusercontent.com/translate_f#249)

Клас DateTime [246](https://translate.googleusercontent.com/translate_f#246) , 248

Лятно часово време [249](https://translate.googleusercontent.com/translate_f#249)

метод на ден от седмицата [249](https://translate.googleusercontent.com/translate_f#249)

отстраняване на грешки с пропускащи заместващи [406](https://translate.googleusercontent.com/translate_f#406)

def ключова дума 37 [,](https://translate.googleusercontent.com/translate_f#37) 132, 182

обект по подразбиране [7](https://translate.googleusercontent.com/translate_f) , 120

като приемник на съобщения [76](https://translate.googleusercontent.com/translate_f#76)

най-високо ниво [121](https://translate.googleusercontent.com/translate_f#121)

метод на дефиниция [438](https://translate.googleusercontent.com/translate_f#438)

делегиране [458](https://translate.googleusercontent.com/translate_f#458)

Делегатор клас [458](https://translate.googleusercontent.com/translate_f#458)

метод за изтриване [233](https://translate.googleusercontent.com/translate_f#233)

за директории [379](https://translate.googleusercontent.com/translate_f#379)

за комплект [282](https://translate.googleusercontent.com/translate_f#282)

метод [376](https://translate.googleusercontent.com/translate_f#376)

desc команда [30](https://translate.googleusercontent.com/translate_f#30)

речник [255](https://translate.googleusercontent.com/translate_f#255)

Дир клас

[] метод [377](https://translate.googleusercontent.com/translate_f#377)

метод на chdir [378](https://translate.googleusercontent.com/translate_f#378)

дефиниран [375](https://translate.googleusercontent.com/translate_f#375)

метод [375](https://translate.googleusercontent.com/translate_f#375)

метод Глоб [377](https://translate.googleusercontent.com/translate_f#377)

на екземпляр [378](https://translate.googleusercontent.com/translate_f#378)

mkdir метод [378](https://translate.googleusercontent.com/translate_f#378)

обект, създаващ [375](https://translate.googleusercontent.com/translate_f#375)

pwd метод [378](https://translate.googleusercontent.com/translate_f#378)

rmdir метод [378](https://translate.googleusercontent.com/translate_f#378)

метод на [разкача 379](https://translate.googleusercontent.com/translate_f#379)

dir, като Pathname обект [382](https://translate.googleusercontent.com/translate_f#382)

Метод Dir.glob [377](https://translate.googleusercontent.com/translate_f#377)

Метод Dir.pwd [378](https://translate.googleusercontent.com/translate_f#378)

Метод Dir # path [376](https://translate.googleusercontent.com/translate_f#376)

директории

създаване на [378](https://translate.googleusercontent.com/translate_f#378)

изтриване [378](https://translate.googleusercontent.com/translate_f#378)

метод [375](https://translate.googleusercontent.com/translate_f#375)

globbing [375](https://translate.googleusercontent.com/translate_f#375)

манипулиране [375](https://translate.googleusercontent.com/translate_f#375)

навигация до [378](https://translate.googleusercontent.com/translate_f#378)

метод на пътя [376](https://translate.googleusercontent.com/translate_f#376)

четене [375](https://translate.googleusercontent.com/translate_f#375)

премахване [380](https://translate.googleusercontent.com/translate_f#380)

метод на показване [202](https://translate.googleusercontent.com/translate_f#202)

до / край срещу къдрави скоби [170](https://translate.googleusercontent.com/translate_f#170)

точка [170](https://translate.googleusercontent.com/translate_f#170)

оператор [9,](https://translate.googleusercontent.com/translate_f#9) 37

в регулярни изрази [351](https://translate.googleusercontent.com/translate_f#351)

заместващ характер [334](https://translate.googleusercontent.com/translate_f#334)

двойно дебело черво [86](https://translate.googleusercontent.com/translate_f#86)

оператор за двойно равен знак [47](https://translate.googleusercontent.com/translate_f#47)

двойни кавички [221](https://translate.googleusercontent.com/translate_f#221)

Модул DryRun [380](https://translate.googleusercontent.com/translate_f#380)

dst? метод [249](https://translate.googleusercontent.com/translate_f#249)

метод на дублиране [58](https://translate.googleusercontent.com/translate_f#58)

дублиране на обекти [58](https://translate.googleusercontent.com/translate_f#58)

|  |
| --- |
| **492 серия** |

ИНДЕКС

**492**

Е.

? e [374](https://translate.googleusercontent.com/translate_f#374)

-e превключвател [23](https://translate.googleusercontent.com/translate_f#23)

всеки метод [30](https://translate.googleusercontent.com/translate_f#30)

сложни итерации [174](https://translate.googleusercontent.com/translate_f#174)

изброеност и [287](https://translate.googleusercontent.com/translate_f#287) –289

прилагане на [173](https://translate.googleusercontent.com/translate_f#173)

за IO обекти [362](https://translate.googleusercontent.com/translate_f#362)

метод на картата срещу [175](https://translate.googleusercontent.com/translate_f#175)

общ преглед [173](https://translate.googleusercontent.com/translate_f#173)

роднини на [300](https://translate.googleusercontent.com/translate_f#300)

върната стойност [173](https://translate.googleusercontent.com/translate_f#173)

просто използване с масив [173](https://translate.googleusercontent.com/translate_f#173)

основната логика [173](https://translate.googleusercontent.com/translate_f#173)

each\_object метод [472](https://translate.googleusercontent.com/translate_f#472)

else ключова дума [154](https://translate.googleusercontent.com/translate_f#154) , 156

ключова дума elsif [154](https://translate.googleusercontent.com/translate_f#154)

празна? метод [228,](https://translate.googleusercontent.com/translate_f#228) 274

метод за кодиране [236](https://translate.googleusercontent.com/translate_f#236)

кодиране

изходни файлове [235](https://translate.googleusercontent.com/translate_f#235)

струни [236](https://translate.googleusercontent.com/translate_f#236)

\_\_ENCODING\_\_ стойност [235](https://translate.googleusercontent.com/translate_f#235)

крайна ключова дума [153](https://translate.googleusercontent.com/translate_f#153)

край на съобщението (EOM) [223](https://translate.googleusercontent.com/translate_f#223)

end\_with? метод [228](https://translate.googleusercontent.com/translate_f#228)

в края на низ котва [347](https://translate.googleusercontent.com/translate_f#347)

Файл English.rb, описания на глобални променливи [131](https://translate.googleusercontent.com/translate_f#131)

осигурете клауза [185](https://translate.googleusercontent.com/translate_f#185)

метод на записи за директории [375](https://translate.googleusercontent.com/translate_f#375)

enum\_за метод [314](https://translate.googleusercontent.com/translate_f#314)

изброеност, добавяне към обекти [320](https://translate.googleusercontent.com/translate_f#320)

Изброителен модул

методи [289](https://translate.googleusercontent.com/translate_f#289)

смесване към класове [289](https://translate.googleusercontent.com/translate_f#289)

отменя, изброяване [472](https://translate.googleusercontent.com/translate_f#472)

изброими

=== (оператор за равенство на случаи) [294](https://translate.googleusercontent.com/translate_f#294)

Булеви заявки [289](https://translate.googleusercontent.com/translate_f#289) –291

метод цикъл [303](https://translate.googleusercontent.com/translate_f#303)

метод капка [298](https://translate.googleusercontent.com/translate_f#298)

[299](https://translate.googleusercontent.com/translate_f#299) серия

всеки метод [287](https://translate.googleusercontent.com/translate_f#287) –289

метод each\_cons [302](https://translate.googleusercontent.com/translate_f#302)

метод [301 на](https://translate.googleusercontent.com/translate_f#301) всеки\_индекс

метод each\_slice [302](https://translate.googleusercontent.com/translate_f#302)

Метод всеки\_с\_индекс [301](https://translate.googleusercontent.com/translate_f#301)

елементите-мъдър операции [297](https://translate.googleusercontent.com/translate_f#297)

намери метод [288,](https://translate.googleusercontent.com/translate_f#288) 292

метод [293](https://translate.googleusercontent.com/translate_f#293)

намиране на минимални и максимални стойности [299](https://translate.googleusercontent.com/translate_f#299)

първи метод [297](https://translate.googleusercontent.com/translate_f#297)

grep метод [294,](https://translate.googleusercontent.com/translate_f#294) 468

[295](https://translate.googleusercontent.com/translate_f#295) серия

инжекционен метод [303](https://translate.googleusercontent.com/translate_f#303)

на място картографиране [306](https://translate.googleusercontent.com/translate_f#306)

IO обекти [362](https://translate.googleusercontent.com/translate_f#362)

метод на картата [304](https://translate.googleusercontent.com/translate_f#304)

максимален метод [299](https://translate.googleusercontent.com/translate_f#299)

[299](https://translate.googleusercontent.com/translate_f#299) серия

метод [299](https://translate.googleusercontent.com/translate_f#299)

минимален метод [299](https://translate.googleusercontent.com/translate_f#299)

[299](https://translate.googleusercontent.com/translate_f#299) серия

метод MinMax [299](https://translate.googleusercontent.com/translate_f#299)

метод minmax\_by [299](https://translate.googleusercontent.com/translate_f#299)

нито един? метод [291](https://translate.googleusercontent.com/translate_f#291)

един? метод [291](https://translate.googleusercontent.com/translate_f#291)

търсене на едно обект [292](https://translate.googleusercontent.com/translate_f#292)

метод дял [295](https://translate.googleusercontent.com/translate_f#295)

метод за намаляване [303](https://translate.googleusercontent.com/translate_f#303)

намаляване [303](https://translate.googleusercontent.com/translate_f#303)

отхвърляне на метод [293](https://translate.googleusercontent.com/translate_f#293)

връщане на всички елементи многократно [303](https://translate.googleusercontent.com/translate_f#303)

връщащи масиви от елементи [302](https://translate.googleusercontent.com/translate_f#302)

връщане на първа позиция [297](https://translate.googleusercontent.com/translate_f#297)

връщащи цели числа с поредни позиции на

елементи [301](https://translate.googleusercontent.com/translate_f#301)

търсене и подбор на [292](https://translate.googleusercontent.com/translate_f#292)

изберете метод [472](https://translate.googleusercontent.com/translate_f#472)

на базата на това дело равенство [294](https://translate.googleusercontent.com/translate_f#294)

метод на сортиране [468](https://translate.googleusercontent.com/translate_f#468)

метод sort\_by [311](https://translate.googleusercontent.com/translate_f#311)

сортиране [308](https://translate.googleusercontent.com/translate_f#308)

логика за сортиране [310](https://translate.googleusercontent.com/translate_f#310)

вземете метод [298](https://translate.googleusercontent.com/translate_f#298)

[299](https://translate.googleusercontent.com/translate_f#299) серия

използвайки grep с [294](https://translate.googleusercontent.com/translate_f#294)

използване на регулярния израз в операция за търсене [353](https://translate.googleusercontent.com/translate_f#353)

with\_index метод [256,](https://translate.googleusercontent.com/translate_f#256) 324

Преброител :: Yielder клас [312](https://translate.googleusercontent.com/translate_f#312)

Enumerator.new [315](https://translate.googleusercontent.com/translate_f#315)

изброители [171,](https://translate.googleusercontent.com/translate_f#171) 311

създавайки [312](https://translate.googleusercontent.com/translate_f#312) –314

създаване имплицитно чрез безблокови извиквания на итератор [316](https://translate.googleusercontent.com/translate_f#316)

всеки метод [316](https://translate.googleusercontent.com/translate_f#316)

метод each\_slice [323](https://translate.googleusercontent.com/translate_f#323)

ексклузивни или операции върху низове [324](https://translate.googleusercontent.com/translate_f#324) –326

фина итерация [320](https://translate.googleusercontent.com/translate_f#320)

инжекционен метод [323](https://translate.googleusercontent.com/translate_f#323)

итератор срещу [320](https://translate.googleusercontent.com/translate_f#320)

мързелив [326](https://translate.googleusercontent.com/translate_f#326) –328

логика на всеки метод [311,](https://translate.googleusercontent.com/translate_f#311) 316

[322](https://translate.googleusercontent.com/translate_f#322) серия

защита на обекти с [318](https://translate.googleusercontent.com/translate_f#318)

изберете метод [318](https://translate.googleusercontent.com/translate_f#318)

семантика [316](https://translate.googleusercontent.com/translate_f#316)

un-overriding Изброими методи [317](https://translate.googleusercontent.com/translate_f#317)

използване за добавяне на изброеност към обекти [320](https://translate.googleusercontent.com/translate_f#320)

with\_index метод [323](https://translate.googleusercontent.com/translate_f#323)

|  |
| --- |
| **Страница 493** |

ИНДЕКС

**493**

<< EOM израз [223](https://translate.googleusercontent.com/translate_f#223) –224

eql? метод [212](https://translate.googleusercontent.com/translate_f#212)

равно? метод [212](https://translate.googleusercontent.com/translate_f#212)

знак за равенство [70,](https://translate.googleusercontent.com/translate_f#70) 73, 225, 261,

[269](https://translate.googleusercontent.com/translate_f#269)

методи за тест за равенство [212](https://translate.googleusercontent.com/translate_f#212)

Errno пространство от имена [372](https://translate.googleusercontent.com/translate_f#372)

Errno :: EACCES [372](https://translate.googleusercontent.com/translate_f#372)

Errno :: ENOENT [372](https://translate.googleusercontent.com/translate_f#372)

Errno :: грешка [180](https://translate.googleusercontent.com/translate_f#180)

Errno: EISDIR [372](https://translate.googleusercontent.com/translate_f#372)

грешка, свързана с I / O файл [372](https://translate.googleusercontent.com/translate_f#372)

бягство [221,](https://translate.googleusercontent.com/translate_f#221) 334

eval семейни методи

class\_eval метод [437](https://translate.googleusercontent.com/translate_f#437) –438

опасности от [435](https://translate.googleusercontent.com/translate_f#435)

eval метод [433](https://translate.googleusercontent.com/translate_f#433) –434

instance\_eval метод [435](https://translate.googleusercontent.com/translate_f#435) –437

преглед на [433](https://translate.googleusercontent.com/translate_f#433)

събитие, примери [457](https://translate.googleusercontent.com/translate_f#457)

изключения

улавяне в спасителна клауза [183](https://translate.googleusercontent.com/translate_f#183)

клас, по поръчка [186](https://translate.googleusercontent.com/translate_f#186)

общи [180](https://translate.googleusercontent.com/translate_f#180)

Errno пространство от имена [372](https://translate.googleusercontent.com/translate_f#372)

файл, свързан с I / O [372](https://translate.googleusercontent.com/translate_f#372)

пространство на имената [187](https://translate.googleusercontent.com/translate_f#187)

NoMethodError [457](https://translate.googleusercontent.com/translate_f#457)

обект, присвоявайки на променлива [183](https://translate.googleusercontent.com/translate_f#183)

издигане и спасяване [180](https://translate.googleusercontent.com/translate_f#180)

повдигане изрично [182](https://translate.googleusercontent.com/translate_f#182)

повторно набиране на [184](https://translate.googleusercontent.com/translate_f#184)

ZeroDivisionError [180](https://translate.googleusercontent.com/translate_f#180)

изключителен или операция [324](https://translate.googleusercontent.com/translate_f#324)

изпълнете буквален скрипт [23](https://translate.googleusercontent.com/translate_f#23)

екзекуция

условно [6](https://translate.googleusercontent.com/translate_f)

проследяване [476](https://translate.googleusercontent.com/translate_f#476)

команда за излизане [28](https://translate.googleusercontent.com/translate_f#28)

изходен метод [440](https://translate.googleusercontent.com/translate_f#440)

примери за израз [39](https://translate.googleusercontent.com/translate_f#39)

удължи събитие, прихваща [462](https://translate.googleusercontent.com/translate_f#462)

разширен метод

добавяне на методи на клас с [409](https://translate.googleusercontent.com/translate_f#409)

добавяне към функционалността на обекта

с [408](https://translate.googleusercontent.com/translate_f#408)

извършване на промени по обект

с [408](https://translate.googleusercontent.com/translate_f#408)

разширения

път на натоварване по подразбиране [19](https://translate.googleusercontent.com/translate_f#19) –20

дефиниран [19](https://translate.googleusercontent.com/translate_f#19)

натоварване [18](https://translate.googleusercontent.com/translate_f#18) –19

изискват изявление [21](https://translate.googleusercontent.com/translate_f#21) –22

изискване\_относително изявление [22](https://translate.googleusercontent.com/translate_f#22)

външен файл, зареждане [18](https://translate.googleusercontent.com/translate_f#18)

F

? е [374](https://translate.googleusercontent.com/translate_f#374)

фалшиви [6](https://translate.googleusercontent.com/translate_f) , 43

обект [207](https://translate.googleusercontent.com/translate_f#207) , 209–210

състояние [207](https://translate.googleusercontent.com/translate_f#207)

състояние спрямо стойност [210](https://translate.googleusercontent.com/translate_f#210)

[209](https://translate.googleusercontent.com/translate_f#209) серия

функция [19](https://translate.googleusercontent.com/translate_f#19)

Фойерман, Емануел [370](https://translate.googleusercontent.com/translate_f#370)

Клас влакна [441](https://translate.googleusercontent.com/translate_f#441)

FIFO. *Вижте* първи вход, първи изход [94](https://translate.googleusercontent.com/translate_f#94)

Файл клас [361](https://translate.googleusercontent.com/translate_f#361)

метод basename [382](https://translate.googleusercontent.com/translate_f#382)

създаване на обект [365](https://translate.googleusercontent.com/translate_f#365)

дефиниран [364](https://translate.googleusercontent.com/translate_f#364)

изброеност [371](https://translate.googleusercontent.com/translate_f#371) –372

изследване на методите на класа от [474](https://translate.googleusercontent.com/translate_f#474)

получаване на информация от [373](https://translate.googleusercontent.com/translate_f#373) –374

[377.](https://translate.googleusercontent.com/translate_f#377) Контролиращи глобуса константи

вътрешен указател [367](https://translate.googleusercontent.com/translate_f#367)

метод [376](https://translate.googleusercontent.com/translate_f#376)

отворен метод [370](https://translate.googleusercontent.com/translate_f#370)

методи заявка [373](https://translate.googleusercontent.com/translate_f#373)

заявки [373](https://translate.googleusercontent.com/translate_f#373)

метод на четене [368](https://translate.googleusercontent.com/translate_f#368) –369

метод за четене [368](https://translate.googleusercontent.com/translate_f#368)

писане на файлове [369](https://translate.googleusercontent.com/translate_f#369) –370

Файл :: FNM\_CASEFOLD флаг [377](https://translate.googleusercontent.com/translate_f#377)

Файл :: Stat [373](https://translate.googleusercontent.com/translate_f#373) –374

$ FILENAME глобална променлива [475](https://translate.googleusercontent.com/translate_f#475)

файлове

bytewise и characterwise четене [366](https://translate.googleusercontent.com/translate_f#366)

затваряне [365](https://translate.googleusercontent.com/translate_f#365)

копиране [379](https://translate.googleusercontent.com/translate_f#379)

атрибут ctime [375](https://translate.googleusercontent.com/translate_f#375)

дефиниран [368](https://translate.googleusercontent.com/translate_f#368)

всеки метод [366](https://translate.googleusercontent.com/translate_f#366)

[366](https://translate.googleusercontent.com/translate_f#366) серия

ефект на кодирането върху четенето [367](https://translate.googleusercontent.com/translate_f#367)

изброеност [371](https://translate.googleusercontent.com/translate_f#371) –372

getbyte метод [367](https://translate.googleusercontent.com/translate_f#367)

метод [366](https://translate.googleusercontent.com/translate_f#366)

получава метод [365](https://translate.googleusercontent.com/translate_f#365)

I / O изключения [372](https://translate.googleusercontent.com/translate_f#372)

итерация през съхранение като масиви [371](https://translate.googleusercontent.com/translate_f#371)

методи за заявки [373](https://translate.googleusercontent.com/translate_f#373)

движещи се [380](https://translate.googleusercontent.com/translate_f#380)

mtime атрибут [375](https://translate.googleusercontent.com/translate_f#375)

отваряне в режим на запис или добавяне [369](https://translate.googleusercontent.com/translate_f#369)

операции, обхват [370](https://translate.googleusercontent.com/translate_f#370)

метод [367](https://translate.googleusercontent.com/translate_f#367)

показалец на позицията, движещ се [368](https://translate.googleusercontent.com/translate_f#368)

указател на позиция, заявка [367](https://translate.googleusercontent.com/translate_f#367)

метод на печат [369](https://translate.googleusercontent.com/translate_f#369)

|  |
| --- |
| **494** |

ИНДЕКС

**494**

файлове *(продължение)*

поставя метод [369](https://translate.googleusercontent.com/translate_f#369)

връщането на символа обратно във файл-вход

поток [366](https://translate.googleusercontent.com/translate_f#366)

метод на четене [365](https://translate.googleusercontent.com/translate_f#365)

метод readbyte [367](https://translate.googleusercontent.com/translate_f#367)

метод на четене [367](https://translate.googleusercontent.com/translate_f#367)

четене

като масив от редове [366](https://translate.googleusercontent.com/translate_f#366)

на ниво клас [368](https://translate.googleusercontent.com/translate_f#368)

следващ ред [365](https://translate.googleusercontent.com/translate_f#365)

един байт [367](https://translate.googleusercontent.com/translate_f#367)

един символ [366](https://translate.googleusercontent.com/translate_f#366)

преглед [14](https://translate.googleusercontent.com/translate_f#14) , 365

метод на четене [366](https://translate.googleusercontent.com/translate_f#366)

метод readlines [366](https://translate.googleusercontent.com/translate_f#366)

премахване [380](https://translate.googleusercontent.com/translate_f#380)

назад метод [366](https://translate.googleusercontent.com/translate_f#366)

търси метод [367](https://translate.googleusercontent.com/translate_f#367)

инструменти от стандартна библиотека [379](https://translate.googleusercontent.com/translate_f#379)

ungetc метод [366](https://translate.googleusercontent.com/translate_f#366)

метод на писане [369](https://translate.googleusercontent.com/translate_f#369)

режим на запис [15](https://translate.googleusercontent.com/translate_f#15)

писане на [15](https://translate.googleusercontent.com/translate_f#15) , 369

Модул FileTest [373](https://translate.googleusercontent.com/translate_f#373) –374

Модул FileUtils [379](https://translate.googleusercontent.com/translate_f#379)

cp метод [379](https://translate.googleusercontent.com/translate_f#379)

Модул DryRun [380](https://translate.googleusercontent.com/translate_f#380)

ln\_s метод [379](https://translate.googleusercontent.com/translate_f#379)

mv метод [380](https://translate.googleusercontent.com/translate_f#380)

Модул NoWrite [380](https://translate.googleusercontent.com/translate_f#380)

rm метод [380](https://translate.googleusercontent.com/translate_f#380)

rm\_rf метод [379](https://translate.googleusercontent.com/translate_f#379) –380

система зависимостта [379](https://translate.googleusercontent.com/translate_f#379)

първи вход, първи изход (FIFO) [94](https://translate.googleusercontent.com/translate_f#94)

Fixnum клас [244](https://translate.googleusercontent.com/translate_f#244)

изравнявам метод [265](https://translate.googleusercontent.com/translate_f#265)

изравнявам! метод [265](https://translate.googleusercontent.com/translate_f#265)

Клас на поплавъка [244](https://translate.googleusercontent.com/translate_f#244)

плувки, освен [244](https://translate.googleusercontent.com/translate_f#244)

FNM\_DOTMATCH флаг [377](https://translate.googleusercontent.com/translate_f#377)

за ключова дума [168](https://translate.googleusercontent.com/translate_f#168)

формален параметър [38](https://translate.googleusercontent.com/translate_f#38)

формат, трансформиращ [232](https://translate.googleusercontent.com/translate_f#232)

наклонена черта [74](https://translate.googleusercontent.com/translate_f#74)

замразяване обекти [59](https://translate.googleusercontent.com/translate_f#59)

функции

дефиниран [419](https://translate.googleusercontent.com/translate_f#419)

ламбда метод [428](https://translate.googleusercontent.com/translate_f#428) –430

G

скъпоценни камъни

команда [31](https://translate.googleusercontent.com/translate_f#31)

дефиниран [18](https://translate.googleusercontent.com/translate_f#18)

директория за [18](https://translate.googleusercontent.com/translate_f#18)

инсталиране [31](https://translate.googleusercontent.com/translate_f#31)

зареждане [32](https://translate.googleusercontent.com/translate_f#32)

метод [32](https://translate.googleusercontent.com/translate_f#32)

родов обект [36](https://translate.googleusercontent.com/translate_f#36) , 45

getbyte метод [367](https://translate.googleusercontent.com/translate_f#367)

getc метод [364,](https://translate.googleusercontent.com/translate_f#364) 366

получава метод 6 [,](https://translate.googleusercontent.com/translate_f) 13

за файлове [365](https://translate.googleusercontent.com/translate_f#365)

клавиатура вход [364](https://translate.googleusercontent.com/translate_f#364)

STDIN и [363](https://translate.googleusercontent.com/translate_f#363)

метод приемателното [261](https://translate.googleusercontent.com/translate_f#261)

glob метод [375,](https://translate.googleusercontent.com/translate_f#375) 377

глобален обхват [130](https://translate.googleusercontent.com/translate_f#130)

глобални променливи [7](https://translate.googleusercontent.com/translate_f#7)

вграден [130](https://translate.googleusercontent.com/translate_f#130)

глобален обхват и [130](https://translate.googleusercontent.com/translate_f#130)

списък на английски.rb [131](https://translate.googleusercontent.com/translate_f#131)

списък [475](https://translate.googleusercontent.com/translate_f#475)

плюсове и минуси [131](https://translate.googleusercontent.com/translate_f#131)

конец локален [130](https://translate.googleusercontent.com/translate_f#130)

метод [475](https://translate.googleusercontent.com/translate_f#475)

глобиране [375](https://translate.googleusercontent.com/translate_f#375) –376

Впиши команда [239](https://translate.googleusercontent.com/translate_f#239)

grep метод и === [358](https://translate.googleusercontent.com/translate_f#358)

З.

-h превключвател [25](https://translate.googleusercontent.com/translate_f#25)

has\_key? метод [274](https://translate.googleusercontent.com/translate_f#274)

Хеш клас

инстанциране с кодов блок по подразбиране [271](https://translate.googleusercontent.com/translate_f#271)

екземпляр със стойност по подразбиране [271](https://translate.googleusercontent.com/translate_f#271)

буквално конструктор [193](https://translate.googleusercontent.com/translate_f#193)

хеш знак [86](https://translate.googleusercontent.com/translate_f#86)

Метод на хеш [269](https://translate.googleusercontent.com/translate_f#269)

хешове

[] метод [270](https://translate.googleusercontent.com/translate_f#270)

=> разделител ключ / стойност [267](https://translate.googleusercontent.com/translate_f#267)

добавяне на двойка ключ / стойност [269](https://translate.googleusercontent.com/translate_f#269)

масив срещу [255](https://translate.googleusercontent.com/translate_f#255) –256

ясен метод [273](https://translate.googleusercontent.com/translate_f#273)

метод на ясна подмяна [274](https://translate.googleusercontent.com/translate_f#274)

[272.](https://translate.googleusercontent.com/translate_f#272) комбиниране с други хешове

съдържание, заместващо [274](https://translate.googleusercontent.com/translate_f#274)

създаване на [268](https://translate.googleusercontent.com/translate_f#268)

метод по подразбиране [271](https://translate.googleusercontent.com/translate_f#271)

стойност по подразбиране, посочваща [271](https://translate.googleusercontent.com/translate_f#271)

дефинирани [140,](https://translate.googleusercontent.com/translate_f#140) 267

delete\_if метод [273](https://translate.googleusercontent.com/translate_f#273)

разрушителни комбинация [272](https://translate.googleusercontent.com/translate_f#272)

празна? метод [274](https://translate.googleusercontent.com/translate_f#274)

метод за извличане [270](https://translate.googleusercontent.com/translate_f#270)

has\_key? метод [274](https://translate.googleusercontent.com/translate_f#274)

has\_value? метод [274](https://translate.googleusercontent.com/translate_f#274)

|  |
| --- |
| **Страница 495** |

ИНДЕКС

**495**

хешове *(продължение)*

Hash.new конструктор [268](https://translate.googleusercontent.com/translate_f#268)

включва? метод [274](https://translate.googleusercontent.com/translate_f#274)

инвертна метод [273](https://translate.googleusercontent.com/translate_f#273)

итерация [290](https://translate.googleusercontent.com/translate_f#290)

[273](https://translate.googleusercontent.com/translate_f#273) серия

ключ? метод [274](https://translate.googleusercontent.com/translate_f#274)

ключ, ред за вмъкване [267](https://translate.googleusercontent.com/translate_f#267)

ключ, несъществуваща [271](https://translate.googleusercontent.com/translate_f#271)

ключ, символ като [241](https://translate.googleusercontent.com/translate_f#241)

ключ, тестващо присъствие на [274](https://translate.googleusercontent.com/translate_f#274)

двойки ключ / стойност [267](https://translate.googleusercontent.com/translate_f#267)

буквално конструктор [268](https://translate.googleusercontent.com/translate_f#268)

член? метод [274](https://translate.googleusercontent.com/translate_f#274)

метод сливат [272](https://translate.googleusercontent.com/translate_f#272)

като аргумент на метода [275](https://translate.googleusercontent.com/translate_f#275)

[272.](https://translate.googleusercontent.com/translate_f#272) неразрушаваща комбинация

поръчани [255](https://translate.googleusercontent.com/translate_f#255)

заявки [274](https://translate.googleusercontent.com/translate_f#274)

отхвърляне на метод [273](https://translate.googleusercontent.com/translate_f#273)

отхвърлете! метод [273](https://translate.googleusercontent.com/translate_f#273)

извличане на множество стойности от [270](https://translate.googleusercontent.com/translate_f#270)

изберете метод [273](https://translate.googleusercontent.com/translate_f#273) , 317–318

изберете! метод [273](https://translate.googleusercontent.com/translate_f#273)

разделяне на ключове и стойности [268](https://translate.googleusercontent.com/translate_f#268)

метод на размера [274](https://translate.googleusercontent.com/translate_f#274)

метод магазин [269](https://translate.googleusercontent.com/translate_f#269)

трансформации [273](https://translate.googleusercontent.com/translate_f#273)

уникалност на ключовете [255](https://translate.googleusercontent.com/translate_f#255)

[275](https://translate.googleusercontent.com/translate_f#275) Серия

актуализация метод [272](https://translate.googleusercontent.com/translate_f#272)

стойност? метод [274](https://translate.googleusercontent.com/translate_f#274)

стойности\_ат метод [270](https://translate.googleusercontent.com/translate_f#270)

Heifetz, Jascha [370](https://translate.googleusercontent.com/translate_f#370)

-помощен превключвател [25](https://translate.googleusercontent.com/translate_f#25)

тук-doc [223](https://translate.googleusercontent.com/translate_f#223) –224

шестнадесетичен метод [234](https://translate.googleusercontent.com/translate_f#234)

шестнадесетично цяло число [245](https://translate.googleusercontent.com/translate_f#245)

кука [457](https://translate.googleusercontent.com/translate_f#457)

Аз

Аз модификатор [349](https://translate.googleusercontent.com/translate_f#349)

% i / I (конструктори на литерални масиви) [260](https://translate.googleusercontent.com/translate_f#260)

I / O (вход / изход)

изключения и грешки [372](https://translate.googleusercontent.com/translate_f#372)

клавиатурен вход [13](https://translate.googleusercontent.com/translate_f#13) –14

операции от ниско ниво [369](https://translate.googleusercontent.com/translate_f#369)

четене от файл [14](https://translate.googleusercontent.com/translate_f#14) –15

писане във файл [15](https://translate.googleusercontent.com/translate_f#15)

[*Вижте също*](https://translate.googleusercontent.com/translate_f#372) IO клас

идентификационен номер на обект [45](https://translate.googleusercontent.com/translate_f#45)

идентификатори

константи [8](https://translate.googleusercontent.com/translate_f#8)

ключови думи [8](https://translate.googleusercontent.com/translate_f#8)

имена на методи [8](https://translate.googleusercontent.com/translate_f#8)

правилата за обхват [129](https://translate.googleusercontent.com/translate_f#129)

видове от [7](https://translate.googleusercontent.com/translate_f#7) –8

променливи [7](https://translate.googleusercontent.com/translate_f#7) –8

ако ключова дума [6](https://translate.googleusercontent.com/translate_f) , 153, 157

if / else неяснота в C [156](https://translate.googleusercontent.com/translate_f#156)

непосредствена стойност [56](https://translate.googleusercontent.com/translate_f#56)

неизменност на символите [238](https://translate.googleusercontent.com/translate_f#238)

в ключова дума [168](https://translate.googleusercontent.com/translate_f#168)

включва? метод [274](https://translate.googleusercontent.com/translate_f#274)

включват срещу добавка [93](https://translate.googleusercontent.com/translate_f#93) , 105–106

включен метод [461](https://translate.googleusercontent.com/translate_f#461)

включване [277](https://translate.googleusercontent.com/translate_f#277)

индексен метод [229](https://translate.googleusercontent.com/translate_f#229)

инфиксни оператори [163](https://translate.googleusercontent.com/translate_f#163)

наследство

верига [77](https://translate.googleusercontent.com/translate_f#77)

на клас срещу смесване в модул [94](https://translate.googleusercontent.com/translate_f#94)

прихващане с наследени [464](https://translate.googleusercontent.com/translate_f#464)

достъп до метод и [149](https://translate.googleusercontent.com/translate_f#149)

множествена [79](https://translate.googleusercontent.com/translate_f#79)

взаимоотношения и сингъл класове [399](https://translate.googleusercontent.com/translate_f#399)

единична [79](https://translate.googleusercontent.com/translate_f#79)

връзка [116](https://translate.googleusercontent.com/translate_f#116) суперклас / подклас [116](https://translate.googleusercontent.com/translate_f#116)

дърво [90](https://translate.googleusercontent.com/translate_f#90)

наследен обратен разговор, ограничения [464](https://translate.googleusercontent.com/translate_f#464)

наследен метод за класове [464](https://translate.googleusercontent.com/translate_f#464)

инициализация, автоматично [69](https://translate.googleusercontent.com/translate_f#69)

инициализирайте метод [68](https://translate.googleusercontent.com/translate_f#68) , 141

входен разделител, за IO обекти [362](https://translate.googleusercontent.com/translate_f#362)

[вход изход.](https://translate.googleusercontent.com/translate_f#372) *Вижте* I / O

проверете метод [201](https://translate.googleusercontent.com/translate_f#201)

инспекция [215](https://translate.googleusercontent.com/translate_f#215)

екземпляр метод [29,](https://translate.googleusercontent.com/translate_f#29) 390

клас метод срещу [85](https://translate.googleusercontent.com/translate_f#85)

получаване на [471](https://translate.googleusercontent.com/translate_f#471)

общ преглед [64](https://translate.googleusercontent.com/translate_f#64)

инстанция променлива [7](https://translate.googleusercontent.com/translate_f#7) , 67

списък [475](https://translate.googleusercontent.com/translate_f#475)

разрешаване чрез себе си [128](https://translate.googleusercontent.com/translate_f#128)

instance\_eval метод [136,](https://translate.googleusercontent.com/translate_f#136) 433, 435–437

instance\_exec метод [136](https://translate.googleusercontent.com/translate_f#136) , 436–437

instance\_methods метод [217,](https://translate.googleusercontent.com/translate_f#217) 431, 471

Метод на instance\_variables [475](https://translate.googleusercontent.com/translate_f#475)

случаи, преглед [63](https://translate.googleusercontent.com/translate_f#63)

екземпляр [10](https://translate.googleusercontent.com/translate_f#10)

инструктирайте! команда [413](https://translate.googleusercontent.com/translate_f#413)

Целочислен клас [244](https://translate.googleusercontent.com/translate_f#244)

цели числа

шестнадесетичен [245](https://translate.googleusercontent.com/translate_f#245)

осмичка [245](https://translate.googleusercontent.com/translate_f#245)

метод [339](https://translate.googleusercontent.com/translate_f#339)

интерактивна интерпретация на Ruby [26](https://translate.googleusercontent.com/translate_f#26) , 28

междинен обект [322](https://translate.googleusercontent.com/translate_f#322)

ординатор метод [235](https://translate.googleusercontent.com/translate_f#235)

|  |
| --- |
| **496 серия** |

ИНДЕКС

**496**

интерполация [42](https://translate.googleusercontent.com/translate_f#42) , 220

преводач

превключватели на командния ред [23](https://translate.googleusercontent.com/translate_f#23)

проверка на синтаксиса [23](https://translate.googleusercontent.com/translate_f#23)

комбиниране на ключове [25](https://translate.googleusercontent.com/translate_f#25) –26

изпълнете скрипт [23](https://translate.googleusercontent.com/translate_f#23) –24

помощ [25](https://translate.googleusercontent.com/translate_f#25)

линия режим [24](https://translate.googleusercontent.com/translate_f#24)

версия за печат [25](https://translate.googleusercontent.com/translate_f#25)

изискват именуван файл [25](https://translate.googleusercontent.com/translate_f#25)

подробен режим [25](https://translate.googleusercontent.com/translate_f#25)

предупреждения [23](https://translate.googleusercontent.com/translate_f#23)

дефиниран [3](https://translate.googleusercontent.com/translate_f#3)

режим на предупреждение [23](https://translate.googleusercontent.com/translate_f#23)

самоанализ [46](https://translate.googleusercontent.com/translate_f#46) , 474

IO клас

дефиниран [361](https://translate.googleusercontent.com/translate_f#361)

eof? инстанционен метод [361](https://translate.googleusercontent.com/translate_f#361)

във Файлов произход [474](https://translate.googleusercontent.com/translate_f#474)

getc модел [361](https://translate.googleusercontent.com/translate_f#361)

методи на ниско ниво [369](https://translate.googleusercontent.com/translate_f#369)

ТЪРСИ константи [368](https://translate.googleusercontent.com/translate_f#368)

метод на търсене на инстанция [361](https://translate.googleusercontent.com/translate_f#361)

[*Вижте също*](https://translate.googleusercontent.com/translate_f#361) I / O

IO глобални променливи [363](https://translate.googleusercontent.com/translate_f#363)

IO обекти [361](https://translate.googleusercontent.com/translate_f#361)

като изброими [362](https://translate.googleusercontent.com/translate_f#362) –363

итерация [362](https://translate.googleusercontent.com/translate_f#362)

заявки [373](https://translate.googleusercontent.com/translate_f#373)

IOError [180](https://translate.googleusercontent.com/translate_f#180)

irb [5](https://translate.googleusercontent.com/translate_f#5) , 26, 28

прекъсване [27](https://translate.googleusercontent.com/translate_f#27)

механизми за котиране и [222](https://translate.googleusercontent.com/translate_f#222)

е? метод [89](https://translate.googleusercontent.com/translate_f#89)

итерация [168](https://translate.googleusercontent.com/translate_f#168)

итератор [30](https://translate.googleusercontent.com/translate_f#30)

кодови блокове и [168](https://translate.googleusercontent.com/translate_f#168)

определение [169](https://translate.googleusercontent.com/translate_f#169)

изброител срещу [320](https://translate.googleusercontent.com/translate_f#320)

J

jd метод [247](https://translate.googleusercontent.com/translate_f#247)

метод на присъединяване [265,](https://translate.googleusercontent.com/translate_f#265) 439

К

Модул на ядрото [101](https://translate.googleusercontent.com/translate_f#101)

вградени методи за частна инстанция [150](https://translate.googleusercontent.com/translate_f#150)

частни методи [374](https://translate.googleusercontent.com/translate_f#374)

Метод за разширяване на ядрото # [408](https://translate.googleusercontent.com/translate_f#408)

Тест # на ядрото [374](https://translate.googleusercontent.com/translate_f#374)

ключ? метод [274](https://translate.googleusercontent.com/translate_f#274)

двойка ключ / стойност [267](https://translate.googleusercontent.com/translate_f#267) , 269

клавиатурен вход [13,](https://translate.googleusercontent.com/translate_f#13) 364

ключови думи

дефинирани [37,](https://translate.googleusercontent.com/translate_f#37) 59

общ преглед [8](https://translate.googleusercontent.com/translate_f#8)

метод убие [440](https://translate.googleusercontent.com/translate_f#440)

L

-l превключвател [24](https://translate.googleusercontent.com/translate_f#24)

ламбда метод [428](https://translate.googleusercontent.com/translate_f#428) –430

последен вход, първи изход (LIFO) [94](https://translate.googleusercontent.com/translate_f#94)

мързеливи изброители [326](https://translate.googleusercontent.com/translate_f#326) –328

скок? метод [249](https://translate.googleusercontent.com/translate_f#249)

високосна година [249](https://translate.googleusercontent.com/translate_f#249)

метод на дължина [228,](https://translate.googleusercontent.com/translate_f#228) 274

библиотека [19](https://translate.googleusercontent.com/translate_f#19)

път библиотека натоварване [475](https://translate.googleusercontent.com/translate_f#475)

поддиректория на библиотеката [17](https://translate.googleusercontent.com/translate_f#17)

LIFO. *Вижте* последен вход, първи изход [94](https://translate.googleusercontent.com/translate_f#94)

линия режим [24](https://translate.googleusercontent.com/translate_f#24)

Linux [372](https://translate.googleusercontent.com/translate_f#372)

литерални конструктори [192](https://translate.googleusercontent.com/translate_f#192) –193

за масиви [258](https://translate.googleusercontent.com/translate_f#258)

за хеш [268](https://translate.googleusercontent.com/translate_f#268)

ламбда конструктор [419](https://translate.googleusercontent.com/translate_f#419)

претоварване на [193](https://translate.googleusercontent.com/translate_f#193)

ljust метод [232](https://translate.googleusercontent.com/translate_f#232)

метод на натоварване [18](https://translate.googleusercontent.com/translate_f#18)

път на натоварване [19](https://translate.googleusercontent.com/translate_f#19)

изявление за натоварване [96](https://translate.googleusercontent.com/translate_f#96)

$ LOAD\_PATH глобална променлива [475](https://translate.googleusercontent.com/translate_f#475)

локален обхват [132,](https://translate.googleusercontent.com/translate_f#132) 135, 177

локална променлива [7](https://translate.googleusercontent.com/translate_f#7) , 54, 133, 179

задание в условно [158](https://translate.googleusercontent.com/translate_f#158)

списък [475](https://translate.googleusercontent.com/translate_f#475)

разпознаване на разбор на време на [158](https://translate.googleusercontent.com/translate_f#158)

метод local\_variables [475](https://translate.googleusercontent.com/translate_f#475)

регистриране с пропускащи промени

[406](https://translate.googleusercontent.com/translate_f#406)

цикъл [169](https://translate.googleusercontent.com/translate_f#169)

условен [166](https://translate.googleusercontent.com/translate_f#166)

контролиращ се [165](https://translate.googleusercontent.com/translate_f#165)

за / в [168](https://translate.googleusercontent.com/translate_f#168)

общ преглед [164](https://translate.googleusercontent.com/translate_f#164)

безусловна [165](https://translate.googleusercontent.com/translate_f#165)

добив и [168](https://translate.googleusercontent.com/translate_f#168)

отстъпване на блок [172](https://translate.googleusercontent.com/translate_f#172)

lstrip метод [232](https://translate.googleusercontent.com/translate_f#232)

Стойност [57](https://translate.googleusercontent.com/translate_f#57) , 72

М

% m [250](https://translate.googleusercontent.com/translate_f#250)

m модификатор [349](https://translate.googleusercontent.com/translate_f#349)

магия коментар [236](https://translate.googleusercontent.com/translate_f#236)

основна ключова дума [123](https://translate.googleusercontent.com/translate_f#123)

|  |
| --- |
| **Страница 497** |

ИНДЕКС

**497**

метод на картата [171,](https://translate.googleusercontent.com/translate_f#171) 174–175

оценка блок [306](https://translate.googleusercontent.com/translate_f#306)

всеки метод срещу [175](https://translate.googleusercontent.com/translate_f#175) –176

прилагане [175](https://translate.googleusercontent.com/translate_f#175)

възвръщаема стойност [175](https://translate.googleusercontent.com/translate_f#175) , 305

карта, надграждайки върху всяка от [176](https://translate.googleusercontent.com/translate_f#176)

съвпадение, ограничаване с квантори [342](https://translate.googleusercontent.com/translate_f#342)

Клас MatchData [159,](https://translate.googleusercontent.com/translate_f#159) 333, 336, 338

[] метод [339](https://translate.googleusercontent.com/translate_f#339)

начален метод [341](https://translate.googleusercontent.com/translate_f#341)

улавя метод [339,](https://translate.googleusercontent.com/translate_f#339) 478

краен метод [341](https://translate.googleusercontent.com/translate_f#341)

все улавя [339](https://translate.googleusercontent.com/translate_f#339)

[341](https://translate.googleusercontent.com/translate_f#341) серия

метод [341](https://translate.googleusercontent.com/translate_f#341)

заявки за екземпляри [159](https://translate.googleusercontent.com/translate_f#159)

математически операции [6](https://translate.googleusercontent.com/translate_f)

член? метод [274](https://translate.googleusercontent.com/translate_f#274)

метод за сливане, за набор [284](https://translate.googleusercontent.com/translate_f#284)

съобщение

дефинирани [8](https://translate.googleusercontent.com/translate_f#8)

приемник [9](https://translate.googleusercontent.com/translate_f#9) , 38

аз като приемник на [126](https://translate.googleusercontent.com/translate_f#126)

изпращане до обект [37,](https://translate.googleusercontent.com/translate_f#37) 47

синтаксис [37](https://translate.googleusercontent.com/translate_f#37)

неразпознат, прихващащ

[457](https://translate.googleusercontent.com/translate_f#457) –461

мета-класове [400](https://translate.googleusercontent.com/translate_f#400)

метод метод [430](https://translate.googleusercontent.com/translate_f#430)

Обект на метод, улавящ [430](https://translate.googleusercontent.com/translate_f#430) –431

[465](https://translate.googleusercontent.com/translate_f#465) серия

method\_missing метод [101,](https://translate.googleusercontent.com/translate_f#101) 109, 413,

[457](https://translate.googleusercontent.com/translate_f#457) –461

комбиниране със супер [110](https://translate.googleusercontent.com/translate_f#110) –114

висш [110](https://translate.googleusercontent.com/translate_f#110)

правила за достъп до метод [144](https://translate.googleusercontent.com/translate_f#144)

път за търсене на метод, единични класове

на [394](https://translate.googleusercontent.com/translate_f#394)

методи [193,](https://translate.googleusercontent.com/translate_f#193) 225

! -прекратени не винаги разрушителни

[197](https://translate.googleusercontent.com/translate_f#197) –199

нива на достъп [144](https://translate.googleusercontent.com/translate_f#144)

правила за достъп и наследяване [149](https://translate.googleusercontent.com/translate_f#149)

Аксесоар [143](https://translate.googleusercontent.com/translate_f#143)

добавяне директно към обекти [91](https://translate.googleusercontent.com/translate_f#91)

анатомия на обажданията до [170](https://translate.googleusercontent.com/translate_f#170)

аргументи

[275](https://translate.googleusercontent.com/translate_f#275) серия

хешира като [275](https://translate.googleusercontent.com/translate_f#275)

на име [276](https://translate.googleusercontent.com/translate_f#276) –277

общ преглед [38,](https://translate.googleusercontent.com/translate_f#38) 48

символ като [240](https://translate.googleusercontent.com/translate_f#240) –241

аритметика [195](https://translate.googleusercontent.com/translate_f#195)

четец на атрибути [74](https://translate.googleusercontent.com/translate_f#74)

атрибут писател [74](https://translate.googleusercontent.com/translate_f#74)

взрив [197](https://translate.googleusercontent.com/translate_f#197)

конвенции и най-добри практики [198](https://translate.googleusercontent.com/translate_f#198)

не винаги разрушителни [199](https://translate.googleusercontent.com/translate_f#199)

обаждане

общ преглед [9](https://translate.googleusercontent.com/translate_f#9)

сценарии за [82](https://translate.googleusercontent.com/translate_f#82)

сходство с локална променлива [59](https://translate.googleusercontent.com/translate_f#59)

синтаксис [170](https://translate.googleusercontent.com/translate_f#170)

[322](https://translate.googleusercontent.com/translate_f#322) серия

клас срещу екземпляр [85](https://translate.googleusercontent.com/translate_f#85)

сравнение [213](https://translate.googleusercontent.com/translate_f#213) –214

преобразуване [200](https://translate.googleusercontent.com/translate_f#200)

опасна [197](https://translate.googleusercontent.com/translate_f#197)

дефинирайки многократно [103](https://translate.googleusercontent.com/translate_f#103) –105

дефиниция [9](https://translate.googleusercontent.com/translate_f#9)

анатомия на [37](https://translate.googleusercontent.com/translate_f#37)

местен обхват [132](https://translate.googleusercontent.com/translate_f#132)

делегиране [458](https://translate.googleusercontent.com/translate_f#458)

тест за равенство [212](https://translate.googleusercontent.com/translate_f#212)

филтриране на списъци от [217](https://translate.googleusercontent.com/translate_f#217)

формален параметър [38](https://translate.googleusercontent.com/translate_f#38)

проверка, на практика [480](https://translate.googleusercontent.com/translate_f#480)

екземпляр, преглед [64](https://translate.googleusercontent.com/translate_f#64)

погледни нагоре

недостатъчност [109](https://translate.googleusercontent.com/translate_f#109)

път за търсене на метод [101](https://translate.googleusercontent.com/translate_f#101) –103

по обект [101](https://translate.googleusercontent.com/translate_f#101)

преглед [100](https://translate.googleusercontent.com/translate_f#100) –101

правила [106](https://translate.googleusercontent.com/translate_f#106)

липсващи [109](https://translate.googleusercontent.com/translate_f#109)

назовавайки 8 [,](https://translate.googleusercontent.com/translate_f#8) 70, 170

неразрушаващи еквиваленти връщат копия [198](https://translate.googleusercontent.com/translate_f#198)

не частни, изброяване [468](https://translate.googleusercontent.com/translate_f#468) –470

нотация [86](https://translate.googleusercontent.com/translate_f#86)

като обекти

улавяне на обекти на Метод [430](https://translate.googleusercontent.com/translate_f#430) –431

цел на [431](https://translate.googleusercontent.com/translate_f#431) –432

обозначение на синтактичната захар в стил оператор [195](https://translate.googleusercontent.com/translate_f#195)

висш [64](https://translate.googleusercontent.com/translate_f#64)

промяна на преминаване [406](https://translate.googleusercontent.com/translate_f#406) –408

добавете [105](https://translate.googleusercontent.com/translate_f#105) –106

частен [145](https://translate.googleusercontent.com/translate_f#145) , 470–471

частен сетер [147](https://translate.googleusercontent.com/translate_f#147)

частен срещу единичен [146](https://translate.googleusercontent.com/translate_f#146)

защитен [148](https://translate.googleusercontent.com/translate_f#148) , 470–471

смяна на приемника (опасно) [197](https://translate.googleusercontent.com/translate_f#197)

приемници [76](https://translate.googleusercontent.com/translate_f#76)

върната стойност [39](https://translate.googleusercontent.com/translate_f#39)

ролеви [205](https://translate.googleusercontent.com/translate_f#205)

път за търсене [102](https://translate.googleusercontent.com/translate_f#102)

търсене и единични класове [394](https://translate.googleusercontent.com/translate_f#394)

сетер [70](https://translate.googleusercontent.com/translate_f#70) , 72

сингълтон [64](https://translate.googleusercontent.com/translate_f#64) , 107, 473–474

като се започва с главна буква [258](https://translate.googleusercontent.com/translate_f#258)

супер ключова дума [107](https://translate.googleusercontent.com/translate_f#107) –109

|  |
| --- |
| **Страница 498** |

ИНДЕКС

**498**

методи *(продължение)*

Най-високо ниво

вграден [150](https://translate.googleusercontent.com/translate_f#150)

определящ [149](https://translate.googleusercontent.com/translate_f#149)

общ преглед [149](https://translate.googleusercontent.com/translate_f#149)

добив [169](https://translate.googleusercontent.com/translate_f#169)

методи метод [215,](https://translate.googleusercontent.com/translate_f#215) 468–470, 473

Помощна програма MicroTest [482](https://translate.googleusercontent.com/translate_f#482) –485

Рамка на MiniTest [480](https://translate.googleusercontent.com/translate_f#480)

MiniTest :: Тест :: Клас единица [482](https://translate.googleusercontent.com/translate_f#482)

MiniTest :: Unit :: TestCase клас [480](https://translate.googleusercontent.com/translate_f#480)

-mix модификатори [352](https://translate.googleusercontent.com/translate_f#352)

микс [114](https://translate.googleusercontent.com/translate_f#114)

mkdir метод [378](https://translate.googleusercontent.com/translate_f#378)

позиция на модификатора за ако и освен

условия [156,](https://translate.googleusercontent.com/translate_f#156) 167

Модулен клас, метод alias\_method

[407](https://translate.googleusercontent.com/translate_f#407)

модул\_eval метод [433](https://translate.googleusercontent.com/translate_f#433)

Модул # разширен метод [462](https://translate.googleusercontent.com/translate_f#462)

Модул # включва метод [462](https://translate.googleusercontent.com/translate_f#462)

модули [95](https://translate.googleusercontent.com/translate_f#95)

метод на предците [410](https://translate.googleusercontent.com/translate_f#410)

обратни обаждания, използвани в единични класове

[463](https://translate.googleusercontent.com/translate_f#463)

клас срещу [115](https://translate.googleusercontent.com/translate_f#115)

[465](https://translate.googleusercontent.com/translate_f#465) серия

ядро, модифициращо [400](https://translate.googleusercontent.com/translate_f#400)

създавайки [93](https://translate.googleusercontent.com/translate_f#93) –94, 99

дефинирани [79](https://translate.googleusercontent.com/translate_f#79)

дефиниционен блок, локален обхват [132](https://translate.googleusercontent.com/translate_f#132)

дизайн и именуване [114](https://translate.googleusercontent.com/translate_f#114)

разграничаване от класове [117](https://translate.googleusercontent.com/translate_f#117)

капсулиращо поведение [96](https://translate.googleusercontent.com/translate_f#96)

разширен метод [462](https://translate.googleusercontent.com/translate_f#462)

включен метод [461](https://translate.googleusercontent.com/translate_f#461) –462

включително в единичен клас [395](https://translate.googleusercontent.com/translate_f#395)

включително повече от веднъж [105](https://translate.googleusercontent.com/translate_f#105)

включване [100](https://translate.googleusercontent.com/translate_f#100)

екземпляр методи, получаване на [471](https://translate.googleusercontent.com/translate_f#471)

метод [216](https://translate.googleusercontent.com/translate_f#216)

метод\_добавен метод [465,](https://translate.googleusercontent.com/translate_f#465) 483

смесване в клас [96](https://translate.googleusercontent.com/translate_f#96) –98, 114

разкрой [116](https://translate.googleusercontent.com/translate_f#116)

използвайки [93](https://translate.googleusercontent.com/translate_f#93) –94

ModuleTester [93](https://translate.googleusercontent.com/translate_f#93)

множествено наследяване [79](https://translate.googleusercontent.com/translate_f#79)

mv метод [380](https://translate.googleusercontent.com/translate_f#380)

н

\ n [223](https://translate.googleusercontent.com/translate_f#223) , 233

име, запазени [179](https://translate.googleusercontent.com/translate_f#179)

именувани аргументи [276](https://translate.googleusercontent.com/translate_f#276) –277

Име Грешка [180](https://translate.googleusercontent.com/translate_f#180)

гнездене

дефиниция на клас в дефиниция на модул [116](https://translate.googleusercontent.com/translate_f#116)

класове и модули [134](https://translate.googleusercontent.com/translate_f#134)

символ на нов ред, като цел на метода на chomp [233](https://translate.googleusercontent.com/translate_f#233)

следваща ключова дума [165](https://translate.googleusercontent.com/translate_f#165)

следващ метод [234,](https://translate.googleusercontent.com/translate_f#234) 251

next\_unit методи [251](https://translate.googleusercontent.com/translate_f#251)

нула [6](https://translate.googleusercontent.com/translate_f) , 43

Булева стойност [44,](https://translate.googleusercontent.com/translate_f#44) 208

обект [211](https://translate.googleusercontent.com/translate_f#211)

регулярни изрази съвпадат с неуспех [338](https://translate.googleusercontent.com/translate_f#338)

премахване от масиви [266](https://translate.googleusercontent.com/translate_f#266)

[211](https://translate.googleusercontent.com/translate_f#211) серия

—Ноехо знаме [27](https://translate.googleusercontent.com/translate_f#27)

NoMethodError [180,](https://translate.googleusercontent.com/translate_f#180) 457

нормализиране на данните [73](https://translate.googleusercontent.com/translate_f#73)

не ключова дума [154](https://translate.googleusercontent.com/translate_f#154)

нотация, претоварване [193](https://translate.googleusercontent.com/translate_f#193)

Модул NoWrite [380](https://translate.googleusercontent.com/translate_f#380)

числа

преобразуване [203](https://translate.googleusercontent.com/translate_f#203)

като предмети [243](https://translate.googleusercontent.com/translate_f#243)

произволен, генериращ [297](https://translate.googleusercontent.com/translate_f#297)

Числов клас [244](https://translate.googleusercontent.com/translate_f#244)

О

[392](https://translate.googleusercontent.com/translate_f#392) серия

Клас на обекти [45,](https://translate.googleusercontent.com/translate_f#45) 79, 150

метод object\_id [45](https://translate.googleusercontent.com/translate_f#45) –46, 136

обекти

добавяне към стека [95](https://translate.googleusercontent.com/translate_f#95)

потекло [79](https://translate.googleusercontent.com/translate_f#79)

основни [45](https://translate.googleusercontent.com/translate_f#45)

поведение [37,](https://translate.googleusercontent.com/translate_f#37) 40

промяна [408](https://translate.googleusercontent.com/translate_f#408)

капсулиране [94](https://translate.googleusercontent.com/translate_f#94)

удължаване [458](https://translate.googleusercontent.com/translate_f#458)

блокове и [421](https://translate.googleusercontent.com/translate_f#421)

клониране [59](https://translate.googleusercontent.com/translate_f#59)

сравнение [212](https://translate.googleusercontent.com/translate_f#212)

създаване [40](https://translate.googleusercontent.com/translate_f#40)

дефиниран [9](https://translate.googleusercontent.com/translate_f#9)

dereferencing [56](https://translate.googleusercontent.com/translate_f#56)

дублиране [58](https://translate.googleusercontent.com/translate_f#58)

половете [46](https://translate.googleusercontent.com/translate_f#46)

разширяване с модул [463](https://translate.googleusercontent.com/translate_f#463)

замразяване [59](https://translate.googleusercontent.com/translate_f#59)

функционалност, добавяйки към с разширение [408](https://translate.googleusercontent.com/translate_f#408)

родово [36,](https://translate.googleusercontent.com/translate_f#36) 45

идентифицирайки уникално [45](https://translate.googleusercontent.com/translate_f#45) –46

индивидуализация [91](https://translate.googleusercontent.com/translate_f#91)

наследство [89](https://translate.googleusercontent.com/translate_f#89)

инициализиране със състояние [68](https://translate.googleusercontent.com/translate_f#68)

междинен [322](https://translate.googleusercontent.com/translate_f#322)

|  |
| --- |
| **499 серия** |

ИНДЕКС

**499**

обекти *(продължение)*

методи като

улавяне на обекти на Метод [430](https://translate.googleusercontent.com/translate_f#430) –431

цел на [431](https://translate.googleusercontent.com/translate_f#431) –432

методи [468](https://translate.googleusercontent.com/translate_f#468)

методи, изброяване [215](https://translate.googleusercontent.com/translate_f#215)

не частни методи, изброяване [468](https://translate.googleusercontent.com/translate_f#468) –470

числово [243](https://translate.googleusercontent.com/translate_f#243)

ориентация [35](https://translate.googleusercontent.com/translate_f#35)

метод на private\_methods [470](https://translate.googleusercontent.com/translate_f#470)

защита с преброители [318](https://translate.googleusercontent.com/translate_f#318)

заявка [46,](https://translate.googleusercontent.com/translate_f#46) 216

връзка с променлива [55](https://translate.googleusercontent.com/translate_f#55)

премахване от стека [95](https://translate.googleusercontent.com/translate_f#95)

отговор на съобщения [37](https://translate.googleusercontent.com/translate_f#37)

изпращане на съобщение до [47](https://translate.googleusercontent.com/translate_f#47)

специална стойност [6](https://translate.googleusercontent.com/translate_f)

състояние [67](https://translate.googleusercontent.com/translate_f#67)

Модул ObjectSpace [472](https://translate.googleusercontent.com/translate_f#472)

окт [234](https://translate.googleusercontent.com/translate_f#234)

осмично цяло число [245](https://translate.googleusercontent.com/translate_f#245)

отворен метод [384,](https://translate.googleusercontent.com/translate_f#384) 451–452

Метод Open3.popen3 [452](https://translate.googleusercontent.com/translate_f#452)

отворен URI библиотека [384](https://translate.googleusercontent.com/translate_f#384)

оператори

побитово [195](https://translate.googleusercontent.com/translate_f#195)

дело за равенство [195](https://translate.googleusercontent.com/translate_f#195)

дефиниране, чрез дефиниране на метод [194](https://translate.googleusercontent.com/translate_f#194) –196

Метод-обвиване [95](https://translate.googleusercontent.com/translate_f#95)

команди за бърз достъп [95](https://translate.googleusercontent.com/translate_f#95)

унар, персонализиране [196](https://translate.googleusercontent.com/translate_f#196)

или оператор, комбиниращ знамена с [377](https://translate.googleusercontent.com/translate_f#377)

метод на ордер [230](https://translate.googleusercontent.com/translate_f#230)

или-равен на оператор [95](https://translate.googleusercontent.com/translate_f#95) , 143

[193.](https://translate.googleusercontent.com/translate_f#193) претоварване

свръхмодулация [114](https://translate.googleusercontent.com/translate_f#114)

заместващи методи [64](https://translate.googleusercontent.com/translate_f#64)

P

p метод [49](https://translate.googleusercontent.com/translate_f#49)

пакети, инсталиране с gem команда [31](https://translate.googleusercontent.com/translate_f#31) -33

паралелно изпълнение с нишки

чат сървър, използващ сокети и нишки [443](https://translate.googleusercontent.com/translate_f#443) –444

нишки за убиване [440](https://translate.googleusercontent.com/translate_f#440) –441

общ преглед [438](https://translate.googleusercontent.com/translate_f#438) –440

изходни нишки [440](https://translate.googleusercontent.com/translate_f#440) –441

спирателни нишки [440](https://translate.googleusercontent.com/translate_f#440) –441

резбови ключове [446](https://translate.googleusercontent.com/translate_f#446) –449

сървър за дати с резба [441](https://translate.googleusercontent.com/translate_f#441) –443

нишки и променливи [445](https://translate.googleusercontent.com/translate_f#445) –446

параметри

блок [170,](https://translate.googleusercontent.com/translate_f#170) 176, 179

официално [38](https://translate.googleusercontent.com/translate_f#38)

поръчка [50](https://translate.googleusercontent.com/translate_f#50)

за Proc обекти [428](https://translate.googleusercontent.com/translate_f#428)

синтаксис [51](https://translate.googleusercontent.com/translate_f#51)

скоби

в условни изявления [155](https://translate.googleusercontent.com/translate_f#155)

стилистични съображения [155](https://translate.googleusercontent.com/translate_f#155)

в submatch улавяне [336](https://translate.googleusercontent.com/translate_f#336)

парсер [158](https://translate.googleusercontent.com/translate_f#158)

преминаваща промяна към основната функционалност

[406](https://translate.googleusercontent.com/translate_f#406) –408

Клас на името на пътя [381](https://translate.googleusercontent.com/translate_f#381)

имена на пътища, [381](https://translate.googleusercontent.com/translate_f#381)

модел за съпоставяне оператор [333](https://translate.googleusercontent.com/translate_f#333)

модели

изразено на ясен език [332](https://translate.googleusercontent.com/translate_f#332)

буквални знаци в [334](https://translate.googleusercontent.com/translate_f#334)

съвпадение, търсене на [225](https://translate.googleusercontent.com/translate_f#225)

Пийл, г-жа Ема [336](https://translate.googleusercontent.com/translate_f#336)

сто-знак техника [71](https://translate.googleusercontent.com/translate_f#71)

поп метод [263](https://translate.googleusercontent.com/translate_f#263)

атрибут pos на обект на файл [367](https://translate.googleusercontent.com/translate_f#367)

prepend vs. включват [105](https://translate.googleusercontent.com/translate_f#105) –106

prev\_unit методи [251](https://translate.googleusercontent.com/translate_f#251)

команда за печат 6 [,](https://translate.googleusercontent.com/translate_f) 12

метод за печат

за файлове [369](https://translate.googleusercontent.com/translate_f#369)

за IO обекти [361](https://translate.googleusercontent.com/translate_f#361)

отпечатване на екран [6](https://translate.googleusercontent.com/translate_f)

частни методи [145](https://translate.googleusercontent.com/translate_f#145) –146, 469–470

метод на частен сетер [147](https://translate.googleusercontent.com/translate_f#147)

private\_instance\_methods метод [151,](https://translate.googleusercontent.com/translate_f#151) 471

метод private\_methods [217,](https://translate.googleusercontent.com/translate_f#217) 470–471

Proc клас [419](https://translate.googleusercontent.com/translate_f#419) –420

Proc обекти

блокове и [420](https://translate.googleusercontent.com/translate_f#420) –421

улавяне на кодов блок като [422](https://translate.googleusercontent.com/translate_f#422)

като затваряния [426](https://translate.googleusercontent.com/translate_f#426) –428

параметри и аргументи [428](https://translate.googleusercontent.com/translate_f#428)

обхват за [427](https://translate.googleusercontent.com/translate_f#427)

Символ # to\_proc [424](https://translate.googleusercontent.com/translate_f#424) –426

to\_proc метод [423](https://translate.googleusercontent.com/translate_f#423) –424

използвайки за блок [423](https://translate.googleusercontent.com/translate_f#423)

Проуч. Нов метод [419](https://translate.googleusercontent.com/translate_f#419)

Процес :: Обект на състоянието [449](https://translate.googleusercontent.com/translate_f#449)

програма

екзекуция, проследяване [476](https://translate.googleusercontent.com/translate_f#476)

бягане 11 [,](https://translate.googleusercontent.com/translate_f#11) 13

съкращаване [14](https://translate.googleusercontent.com/translate_f#14)

правилно подмножество [285](https://translate.googleusercontent.com/translate_f#285)

[285](https://translate.googleusercontent.com/translate_f#285) серия

защитени методи [148](https://translate.googleusercontent.com/translate_f#148) –470

метод [471](https://translate.googleusercontent.com/translate_f#471)

метод защитени\_методи [217,](https://translate.googleusercontent.com/translate_f#217) 470–471

метод на public\_instance\_methods [471](https://translate.googleusercontent.com/translate_f#471)

public\_method\_defined? метод [112](https://translate.googleusercontent.com/translate_f#112)

метод [217](https://translate.googleusercontent.com/translate_f#217)

|  |
| --- |
| **Страница 500** |

ИНДЕКС

**500**

метод public\_send [48](https://translate.googleusercontent.com/translate_f#48)

метод тласък [262](https://translate.googleusercontent.com/translate_f#262)

поставя метод 6 [,](https://translate.googleusercontent.com/translate_f) 12

обаждания към\_s [200](https://translate.googleusercontent.com/translate_f#200)

за файлове [369](https://translate.googleusercontent.com/translate_f#369)

нулева върната стойност [43](https://translate.googleusercontent.com/translate_f#43)

STDOUT и [363](https://translate.googleusercontent.com/translate_f#363)

Въпрос:

% Q (конструктор на литерални низове) [222](https://translate.googleusercontent.com/translate_f#222)

% q (конструктор на литерални низове) [221](https://translate.googleusercontent.com/translate_f#221)

квантори [342](https://translate.googleusercontent.com/translate_f#342)

въпросителен знак, оценяващ true / false [43](https://translate.googleusercontent.com/translate_f#43)

кавички

двойно [221](https://translate.googleusercontent.com/translate_f#221)

механизми [221](https://translate.googleusercontent.com/translate_f#221)

единична [220](https://translate.googleusercontent.com/translate_f#220) –221

R

-r флаг [16](https://translate.googleusercontent.com/translate_f#16) , 25

повишаване до енергиен оператор [95](https://translate.googleusercontent.com/translate_f#95)

полезна програма [29](https://translate.googleusercontent.com/translate_f#29) –30

Rakefile [29](https://translate.googleusercontent.com/translate_f#29)

случайни числа [297](https://translate.googleusercontent.com/translate_f#297)

Клас на обхвата

на екземпляр [278](https://translate.googleusercontent.com/translate_f#278)

буквално конструктор [193](https://translate.googleusercontent.com/translate_f#193)

диапазони

.. срещу ... [278](https://translate.googleusercontent.com/translate_f#278)

назад [280](https://translate.googleusercontent.com/translate_f#280)

като булеви инструменти за включване [279](https://translate.googleusercontent.com/translate_f#279)

Покрийте? метод [279](https://translate.googleusercontent.com/translate_f#279)

създаване на [278](https://translate.googleusercontent.com/translate_f#278)

дефинирано [277](https://translate.googleusercontent.com/translate_f#277)

изброяване [278,](https://translate.googleusercontent.com/translate_f#278) 291

изключителната [278](https://translate.googleusercontent.com/translate_f#278)

включва? метод [279](https://translate.googleusercontent.com/translate_f#279) –280

включване [277](https://translate.googleusercontent.com/translate_f#277) , 279

включително [278](https://translate.googleusercontent.com/translate_f#278)

член? метод [279](https://translate.googleusercontent.com/translate_f#279)

начални и крайни точки [279](https://translate.googleusercontent.com/translate_f#279)

Рационален метод [258](https://translate.googleusercontent.com/translate_f#258)

.rb разширение [11](https://translate.googleusercontent.com/translate_f#11)

rbconfig библиотека [16](https://translate.googleusercontent.com/translate_f#16) –17

RDoc [28](https://translate.googleusercontent.com/translate_f#28)

четене на атрибут [75](https://translate.googleusercontent.com/translate_f#75)

метод на четене [365](https://translate.googleusercontent.com/translate_f#365)

метод readbyte [367](https://translate.googleusercontent.com/translate_f#367)

метод на четене [367](https://translate.googleusercontent.com/translate_f#367)

четене от файл [14](https://translate.googleusercontent.com/translate_f#14)

метод на четене [366](https://translate.googleusercontent.com/translate_f#366)

метод readlines [366](https://translate.googleusercontent.com/translate_f#366)

реалния свят [35](https://translate.googleusercontent.com/translate_f#35)

приемник обект [170](https://translate.googleusercontent.com/translate_f#170)

приемници

по подразбиране [76](https://translate.googleusercontent.com/translate_f#76)

изрично [76](https://translate.googleusercontent.com/translate_f#76)

на съобщение [9](https://translate.googleusercontent.com/translate_f#9)

рекурсия [135](https://translate.googleusercontent.com/translate_f#135)

референции [56](https://translate.googleusercontent.com/translate_f#56)

аргумент на метода и [58](https://translate.googleusercontent.com/translate_f#58)

в присвояване на променлива [57](https://translate.googleusercontent.com/translate_f#57)

уточнения [411](https://translate.googleusercontent.com/translate_f#411) –412

отражение [46](https://translate.googleusercontent.com/translate_f#46) , 215

Regexp клас [332](https://translate.googleusercontent.com/translate_f#332)

метод за бягство [351](https://translate.googleusercontent.com/translate_f#351)

буквално конструктор [193](https://translate.googleusercontent.com/translate_f#193)

промяна [401](https://translate.googleusercontent.com/translate_f#401)

нов метод [352](https://translate.googleusercontent.com/translate_f#352)

предаване на низ към нов метод [352](https://translate.googleusercontent.com/translate_f#352)

Regexp # метод на съвпадение [401](https://translate.googleusercontent.com/translate_f#401) –402

регулярни изрази

? (въпросителен знак) нулев или един квантор [342](https://translate.googleusercontent.com/translate_f#342) , 344

^ (карета) котва за начало на линия [346](https://translate.googleusercontent.com/translate_f#346)

^ (карета) символ за отрицание клас на знак [335](https://translate.googleusercontent.com/translate_f#335)

. (точка) заместващ характер [334](https://translate.googleusercontent.com/translate_f#334)

{} (къдрави скоби) спецификатор на броя [345](https://translate.googleusercontent.com/translate_f#345)

\* (звездичка) нулев или повече квантор [343](https://translate.googleusercontent.com/translate_f#343)

+ (знак плюс) един или повече квантор [343](https://translate.googleusercontent.com/translate_f#343)

= ~ (знак за равенство / тилда) метод [333](https://translate.googleusercontent.com/translate_f#333)

$ (знак на долара) котва за край на линия [346](https://translate.googleusercontent.com/translate_f#346)

\ Котва в началото на низа [347](https://translate.googleusercontent.com/translate_f#347)

анкери [342](https://translate.googleusercontent.com/translate_f#342) , 346–347

твърдения [346](https://translate.googleusercontent.com/translate_f#346)

атома [345](https://translate.googleusercontent.com/translate_f#345)

\ b котва на границата на думата [347](https://translate.googleusercontent.com/translate_f#347)

връщане назад [345](https://translate.googleusercontent.com/translate_f#345)

улавя от [339](https://translate.googleusercontent.com/translate_f#339) –340, 354

улавя, използвайки в низ [357](https://translate.googleusercontent.com/translate_f#357)

улавяне на подсъвпадения със скоби [336](https://translate.googleusercontent.com/translate_f#336)

При равенство [357](https://translate.googleusercontent.com/translate_f#357)

класа герои [334](https://translate.googleusercontent.com/translate_f#334)

коментари за [350](https://translate.googleusercontent.com/translate_f#350)

условни мачове [348](https://translate.googleusercontent.com/translate_f#348)

консумират герои [346](https://translate.googleusercontent.com/translate_f#346)

преобразуване в низове [350](https://translate.googleusercontent.com/translate_f#350)

създават модели [333](https://translate.googleusercontent.com/translate_f#333)

\ d всяка цифра [335](https://translate.googleusercontent.com/translate_f#335)

\ D всякакви нецифрени [336](https://translate.googleusercontent.com/translate_f#336)

представяне на низ по подразбиране [352](https://translate.googleusercontent.com/translate_f#352)

двигател [344](https://translate.googleusercontent.com/translate_f#344)

[335](https://translate.googleusercontent.com/translate_f#335) серия

глобални променливи за уловени подсъвпадения [337](https://translate.googleusercontent.com/translate_f#337)

[344.](https://translate.googleusercontent.com/translate_f#344) алчни и ненаситни квантори

инстанциране от низ [352](https://translate.googleusercontent.com/translate_f#352)

ограничения на скобите при съвпадение [346](https://translate.googleusercontent.com/translate_f#346)

буквални знаци в [332](https://translate.googleusercontent.com/translate_f#332)

[347](https://translate.googleusercontent.com/translate_f#347) серия

|  |
| --- |
| **501** |

ИНДЕКС

**501**

регулярни изрази *(продължение)*

[348](https://translate.googleusercontent.com/translate_f#348) серия

метод мач [332](https://translate.googleusercontent.com/translate_f#332)

мач успеха и провала [338](https://translate.googleusercontent.com/translate_f#338)

модификатори [342](https://translate.googleusercontent.com/translate_f#342) , 349

Отрицателни класа герои [335](https://translate.googleusercontent.com/translate_f#335)

отрицателни твърдения [348](https://translate.googleusercontent.com/translate_f#348)

отрицателен поглед зад твърденията [348](https://translate.googleusercontent.com/translate_f#348)

общ преглед [331](https://translate.googleusercontent.com/translate_f#331)

[347.](https://translate.googleusercontent.com/translate_f#347) положителни твърдения

положителен поглед зад твърденията [348](https://translate.googleusercontent.com/translate_f#348)

предварително определени класове символи [335](https://translate.googleusercontent.com/translate_f#335)

квантори [342](https://translate.googleusercontent.com/translate_f#342)

диапазон от знаци в клас знаци [335](https://translate.googleusercontent.com/translate_f#335)

като представяния на модели [331](https://translate.googleusercontent.com/translate_f#331)

репутация на криптичност [331](https://translate.googleusercontent.com/translate_f#331)

\ S всеки небел знак [336](https://translate.googleusercontent.com/translate_f#336)

\ s всеки празен знак [336](https://translate.googleusercontent.com/translate_f#336)

специални символи [334](https://translate.googleusercontent.com/translate_f#334)

низ методи използвайки [353](https://translate.googleusercontent.com/translate_f#353)

[337](https://translate.googleusercontent.com/translate_f#337) серия

\ w букви или цифри [336](https://translate.googleusercontent.com/translate_f#336)

\ W всеки не-w символ [336](https://translate.googleusercontent.com/translate_f#336)

празно пространство в [350](https://translate.googleusercontent.com/translate_f#350)

писмено [331](https://translate.googleusercontent.com/translate_f#331)

\ Z край на низа (с изключение на крайния нов ред)

котва [347](https://translate.googleusercontent.com/translate_f#347)

\ z котва в края на низа [347](https://translate.googleusercontent.com/translate_f#347)

свойство с нулева ширина на твърдения [347](https://translate.googleusercontent.com/translate_f#347)

отхвърляне на метод [469](https://translate.googleusercontent.com/translate_f#469)

повторение, намаляване на [75](https://translate.googleusercontent.com/translate_f#75)

заменете метода [233,](https://translate.googleusercontent.com/translate_f#233) 264

изискват метод [18](https://translate.googleusercontent.com/translate_f#18)

изискват изявление [21,](https://translate.googleusercontent.com/translate_f#21) 96

изисква\_относително [22](https://translate.googleusercontent.com/translate_f#22)

спасителна клауза [180](https://translate.googleusercontent.com/translate_f#180) , 183

спасителна ключова дума [181](https://translate.googleusercontent.com/translate_f#181)

запазено име [179](https://translate.googleusercontent.com/translate_f#179)

отговарям на? метод 46 [,](https://translate.googleusercontent.com/translate_f#46) 90

връщане на ключова дума 39 [,](https://translate.googleusercontent.com/translate_f#39) 172, 429

обратен метод [265](https://translate.googleusercontent.com/translate_f#265)

обратен\_ всеки метод [300](https://translate.googleusercontent.com/translate_f#300)

метод за пренавиване на файлове [366](https://translate.googleusercontent.com/translate_f#366)

RFC 2616 формат [251](https://translate.googleusercontent.com/translate_f#251)

RFC 2822 формат [251](https://translate.googleusercontent.com/translate_f#251)

ri [28](https://translate.googleusercontent.com/translate_f#28)

метод на rindex [229](https://translate.googleusercontent.com/translate_f#229)

метод rjust [232](https://translate.googleusercontent.com/translate_f#232)

rm метод [380](https://translate.googleusercontent.com/translate_f#380)

rm\_rf метод [380](https://translate.googleusercontent.com/translate_f#380)

rmdir метод [378](https://translate.googleusercontent.com/translate_f#378)

ролеви методи

общ преглед [205](https://translate.googleusercontent.com/translate_f#205)

[206](https://translate.googleusercontent.com/translate_f#206) серия

to\_str [205](https://translate.googleusercontent.com/translate_f#205)

метод [232](https://translate.googleusercontent.com/translate_f#232)

Руби

директория на archdir [17](https://translate.googleusercontent.com/translate_f#17)

основни операции [5](https://translate.googleusercontent.com/translate_f#5)

конзолна програма [5](https://translate.googleusercontent.com/translate_f#5)

директория за скъпоценни камъни [18](https://translate.googleusercontent.com/translate_f#18)

идентификатори

константи [8](https://translate.googleusercontent.com/translate_f#8)

ключови думи [8](https://translate.googleusercontent.com/translate_f#8)

имена на методи [8](https://translate.googleusercontent.com/translate_f#8)

променливи [7](https://translate.googleusercontent.com/translate_f#7) –8

интерактивна интерпретация [26](https://translate.googleusercontent.com/translate_f#26)

грамотност [5](https://translate.googleusercontent.com/translate_f#5)

преглед [4](https://translate.googleusercontent.com/translate_f#4) –5

програма, работеща 11 [,](https://translate.googleusercontent.com/translate_f#11) 13

директория rubylibdir [17](https://translate.googleusercontent.com/translate_f#17)

sitedir директория [17](https://translate.googleusercontent.com/translate_f#17) –18

дърво на изходния код [15](https://translate.googleusercontent.com/translate_f#15)

Документация за Ruby (RDoc) [28](https://translate.googleusercontent.com/translate_f#28)

Рубинен индекс (ri) [28](https://translate.googleusercontent.com/translate_f#28)

RubyGems [18](https://translate.googleusercontent.com/translate_f#18)

управляеми обекти

eval семейни методи

class\_eval метод [437](https://translate.googleusercontent.com/translate_f#437) –438

опасности от [435](https://translate.googleusercontent.com/translate_f#435)

eval метод [433](https://translate.googleusercontent.com/translate_f#433) –434

instance\_eval метод [435](https://translate.googleusercontent.com/translate_f#435) –437

преглед на [433](https://translate.googleusercontent.com/translate_f#433)

функции, използващи ламбда метод [428](https://translate.googleusercontent.com/translate_f#428) –430

методи като обекти

улавяне на обекти на Метод [430](https://translate.googleusercontent.com/translate_f#430) –431

цел на [431](https://translate.googleusercontent.com/translate_f#431) –432

паралелно изпълнение с нишки

чат сървър, използващ сокети и нишки [443](https://translate.googleusercontent.com/translate_f#443) –444

нишки за убиване [440](https://translate.googleusercontent.com/translate_f#440) –441

общ преглед [438](https://translate.googleusercontent.com/translate_f#438) –440

изходни нишки [440](https://translate.googleusercontent.com/translate_f#440) –441

спирателни нишки [440](https://translate.googleusercontent.com/translate_f#440) –441

резбови ключове [446](https://translate.googleusercontent.com/translate_f#446) –449

сървър за дати с резба [441](https://translate.googleusercontent.com/translate_f#441) –443

нишки и променливи [445](https://translate.googleusercontent.com/translate_f#445) –446

Proc клас

обекти [419](https://translate.googleusercontent.com/translate_f#419) –420

преглед на [419](https://translate.googleusercontent.com/translate_f#419)

Proc обекти

блокове и [420](https://translate.googleusercontent.com/translate_f#420) –421

като затваряния [426](https://translate.googleusercontent.com/translate_f#426) –428

конвертиране между блокове и [422](https://translate.googleusercontent.com/translate_f#422) –424

параметри и аргументи [428](https://translate.googleusercontent.com/translate_f#428)

Символ # to\_proc [424](https://translate.googleusercontent.com/translate_f#424) –426

системни команди

отворен метод [451](https://translate.googleusercontent.com/translate_f#451) –452

Метод Open3.popen3 [452](https://translate.googleusercontent.com/translate_f#452)

системен метод [449](https://translate.googleusercontent.com/translate_f#449) –450

с помощта на обратни връзки [450](https://translate.googleusercontent.com/translate_f#450) –451

|  |
| --- |
| **502 серия** |

ИНДЕКС

**502**

Идентификатор на изпълнение, разрешения [374](https://translate.googleusercontent.com/translate_f#374)

RuntimeError [180,](https://translate.googleusercontent.com/translate_f#180) 182

С

скални обекти [219](https://translate.googleusercontent.com/translate_f#219)

обхват

определяне [129](https://translate.googleusercontent.com/translate_f#129)

глобален [130](https://translate.googleusercontent.com/translate_f#130)

местни [132](https://translate.googleusercontent.com/translate_f#132) , 135, 177

за Proc обекти [427](https://translate.googleusercontent.com/translate_f#427)

разрешаване на константи и [136](https://translate.googleusercontent.com/translate_f#136)

скрипт, изпълнете литерал [23](https://translate.googleusercontent.com/translate_f#23)

търсене отрицание [229](https://translate.googleusercontent.com/translate_f#229)

търсене на съвпадение на образец [225](https://translate.googleusercontent.com/translate_f#225)

сек метод [249](https://translate.googleusercontent.com/translate_f#249)

втори метод [249](https://translate.googleusercontent.com/translate_f#249)

метод за търсене на файлове [367](https://translate.googleusercontent.com/translate_f#367)

самостоятелна ключова дума [7](https://translate.googleusercontent.com/translate_f) , 10

в клас [123](https://translate.googleusercontent.com/translate_f#123) , 125, 394

като приемник на съобщения по подразбиране [76](https://translate.googleusercontent.com/translate_f#76) , 126

дефиниран [120](https://translate.googleusercontent.com/translate_f#120)

определящ обект [120,](https://translate.googleusercontent.com/translate_f#120) 123

в пример дефиниция на метод [124](https://translate.googleusercontent.com/translate_f#124) –125

местен обхват и [135](https://translate.googleusercontent.com/translate_f#135)

в модул [123,](https://translate.googleusercontent.com/translate_f#123) 125

разрешаване на променливи на екземпляра чрез [128](https://translate.googleusercontent.com/translate_f#128)

в единичен метод [124](https://translate.googleusercontent.com/translate_f#124)

най-високо ниво [122](https://translate.googleusercontent.com/translate_f#122)

като се използват вместо кодирани имена на класове [125](https://translate.googleusercontent.com/translate_f#125)

точка и запетая [153,](https://translate.googleusercontent.com/translate_f#153) 178

\_\_send\_\_ метод [47](https://translate.googleusercontent.com/translate_f#47)

метод на изпращане [484](https://translate.googleusercontent.com/translate_f#484)

алтернативи на [47](https://translate.googleusercontent.com/translate_f#47)

символ като аргумент [240](https://translate.googleusercontent.com/translate_f#240)

Задайте клас [281](https://translate.googleusercontent.com/translate_f#281)

комплекти

^ (XOR) метод [283](https://translate.googleusercontent.com/translate_f#283)

& (пресичане) метод [283](https://translate.googleusercontent.com/translate_f#283)

+ (пресичане) метод [283](https://translate.googleusercontent.com/translate_f#283)

<< оператор [282](https://translate.googleusercontent.com/translate_f#282)

| метод (пресичане) [283](https://translate.googleusercontent.com/translate_f#283)

добавяне на метод [283](https://translate.googleusercontent.com/translate_f#283)

добавяне? метод [283](https://translate.googleusercontent.com/translate_f#283)

[282.](https://translate.googleusercontent.com/translate_f#282) добавяне / премахване на един обект

промяна на елементи на място [293](https://translate.googleusercontent.com/translate_f#293)

създаване на [281](https://translate.googleusercontent.com/translate_f#281)

метод разлика [283](https://translate.googleusercontent.com/translate_f#283)

изброяване [291](https://translate.googleusercontent.com/translate_f#291)

хешове и [281](https://translate.googleusercontent.com/translate_f#281)

инициализиране с кодов блок [282](https://translate.googleusercontent.com/translate_f#282)

на място картографиране [306](https://translate.googleusercontent.com/translate_f#306)

метод на пресичане [283](https://translate.googleusercontent.com/translate_f#283)

манипулиране [282](https://translate.googleusercontent.com/translate_f#282)

карта! метод [293,](https://translate.googleusercontent.com/translate_f#293) 306

преглед на [281](https://translate.googleusercontent.com/translate_f#281)

метод proper\_subset [285](https://translate.googleusercontent.com/translate_f#285)

метод proper\_superset [285](https://translate.googleusercontent.com/translate_f#285)

метод на подмножество [284](https://translate.googleusercontent.com/translate_f#284)

метод надмножество [284](https://translate.googleusercontent.com/translate_f#284)

метод съюз [283](https://translate.googleusercontent.com/translate_f#283)

уникалност на елементите [281](https://translate.googleusercontent.com/translate_f#281)

сетер метод [70,](https://translate.googleusercontent.com/translate_f#70) 72

на масив [261](https://translate.googleusercontent.com/translate_f#261)

частен [147](https://translate.googleusercontent.com/translate_f#147)

върнати стойности [74](https://translate.googleusercontent.com/translate_f#74)

черупка globbing [376](https://translate.googleusercontent.com/translate_f#376)

метод на смяна [263](https://translate.googleusercontent.com/translate_f#263)

бърз достъп на операторите [95](https://translate.googleusercontent.com/translate_f#95)

Клас SimpleDelegator [458](https://translate.googleusercontent.com/translate_f#458)

- флаг за бърз подсказване [26](https://translate.googleusercontent.com/translate_f#26)

еднократно наследство [79](https://translate.googleusercontent.com/translate_f#79)

единични кавички

бягащи герои [221](https://translate.googleusercontent.com/translate_f#221)

интерполация [220](https://translate.googleusercontent.com/translate_f#220)

Сингълтън клас [107](https://translate.googleusercontent.com/translate_f#107)

поведение с разширено и включено [463](https://translate.googleusercontent.com/translate_f#463)

клас дума нотация [392](https://translate.googleusercontent.com/translate_f#392)

двойно определяне на обект чрез [391](https://translate.googleusercontent.com/translate_f#391)

в наследствени отношения [399](https://translate.googleusercontent.com/translate_f#399)

включително модули [394](https://translate.googleusercontent.com/translate_f#394) –395

на пътя за търсене на метод [394](https://translate.googleusercontent.com/translate_f#394)

модифициране [392](https://translate.googleusercontent.com/translate_f#392)

единичен метод 64 [,](https://translate.googleusercontent.com/translate_f#64) 107, 390

призоваване на подкласове [399](https://translate.googleusercontent.com/translate_f#399)

на обект от клас, извикващ [83](https://translate.googleusercontent.com/translate_f#83)

списък [473](https://translate.googleusercontent.com/translate_f#473) –474

местоположение [391](https://translate.googleusercontent.com/translate_f#391)

частен метод срещу [146](https://translate.googleusercontent.com/translate_f#146)

Сингълтън модул [397](https://translate.googleusercontent.com/translate_f#397)

singleton\_class метод [398](https://translate.googleusercontent.com/translate_f#398)

singleton\_method\_added метод [466](https://translate.googleusercontent.com/translate_f#466) –467

метод на singleton\_methods [217,](https://translate.googleusercontent.com/translate_f#217) 473–474

директория site\_ruby [17](https://translate.googleusercontent.com/translate_f#17)

директория на sitedir [17](https://translate.googleusercontent.com/translate_f#17)

метод на размера [228,](https://translate.googleusercontent.com/translate_f#228) 274

метод на [нарязване 225,](https://translate.googleusercontent.com/translate_f#225) 262

парче! метод [225](https://translate.googleusercontent.com/translate_f#225) , 262

метод на сортиране [468](https://translate.googleusercontent.com/translate_f#468)

дърво на изходния код [15](https://translate.googleusercontent.com/translate_f#15)

метод на космически кораб [230](https://translate.googleusercontent.com/translate_f#230)

оператор на космически кораб [213](https://translate.googleusercontent.com/translate_f#213) –214, 299

обект със специална стойност [6](https://translate.googleusercontent.com/translate_f)

метод на разделяне [74](https://translate.googleusercontent.com/translate_f#74) , 170

метод sprintf [71](https://translate.googleusercontent.com/translate_f#71)

Клас на стека [478](https://translate.googleusercontent.com/translate_f#478)

метод на стека [95](https://translate.googleusercontent.com/translate_f#95)

проследяване на стека

изследване с повикващия [476](https://translate.googleusercontent.com/translate_f#476)

разбор [477](https://translate.googleusercontent.com/translate_f#477)

|  |
| --- |
| **503** |

ИНДЕКС

**503**

stacklikeness [94](https://translate.googleusercontent.com/translate_f#94)

стандартна библиотека [18](https://translate.googleusercontent.com/translate_f#18) , 379

Започни с? метод [228](https://translate.googleusercontent.com/translate_f#228)

държава

инициализиране на обект с [68](https://translate.googleusercontent.com/translate_f#68)

поддържане [68](https://translate.googleusercontent.com/translate_f#68)

поддържане по клас [143](https://translate.googleusercontent.com/translate_f#143)

на обект [67](https://translate.googleusercontent.com/translate_f#67)

STDERR константа [361](https://translate.googleusercontent.com/translate_f#361)

изпълнение на системни програми [449](https://translate.googleusercontent.com/translate_f#449)

изход [363](https://translate.googleusercontent.com/translate_f#363)

$ stderr променлива [363](https://translate.googleusercontent.com/translate_f#363)

STDIN константа [361](https://translate.googleusercontent.com/translate_f#361)

изпълнение на системни програми [449](https://translate.googleusercontent.com/translate_f#449)

получава метод и [363](https://translate.googleusercontent.com/translate_f#363)

$ stdin променлива [363](https://translate.googleusercontent.com/translate_f#363)

STDOUT константа [361](https://translate.googleusercontent.com/translate_f#361)

метод на показване [202](https://translate.googleusercontent.com/translate_f#202)

изпълнение на системни програми [449](https://translate.googleusercontent.com/translate_f#449)

поставя метод и [363](https://translate.googleusercontent.com/translate_f#363)

$ stdout променлива [363](https://translate.googleusercontent.com/translate_f#363)

strftime метод [249](https://translate.googleusercontent.com/translate_f#249) , 408

Струнен клас [193](https://translate.googleusercontent.com/translate_f#193) , 220

Струнен метод [258](https://translate.googleusercontent.com/translate_f#258)

Низ # gsub !, модифициращ [402](https://translate.googleusercontent.com/translate_f#402) –405

StringIO клас [382](https://translate.googleusercontent.com/translate_f#382) –384

струни

[] метод [224](https://translate.googleusercontent.com/translate_f#224)

\* метод (умножение) [204](https://translate.googleusercontent.com/translate_f#204)

+ метод [226](https://translate.googleusercontent.com/translate_f#226)

<=> (космически кораб) метод [230](https://translate.googleusercontent.com/translate_f#230)

== метод [230](https://translate.googleusercontent.com/translate_f#230)

= ~ метод [333](https://translate.googleusercontent.com/translate_f#333)

като се добавя към още [226](https://translate.googleusercontent.com/translate_f#226)

Булеви заявки [228](https://translate.googleusercontent.com/translate_f#228)

байтов метод [308](https://translate.googleusercontent.com/translate_f#308)

с главни букви [231](https://translate.googleusercontent.com/translate_f#231)

улавя в метод на сканиране [354](https://translate.googleusercontent.com/translate_f#354)

При трансформация [231](https://translate.googleusercontent.com/translate_f#231)

центриране [232](https://translate.googleusercontent.com/translate_f#232)

промяна на съдържанието на [356](https://translate.googleusercontent.com/translate_f#356)

метод на символите [308](https://translate.googleusercontent.com/translate_f#308)

метод на chomp [233](https://translate.googleusercontent.com/translate_f#233)

метод котлет [233](https://translate.googleusercontent.com/translate_f#233)

chr метод [230](https://translate.googleusercontent.com/translate_f#230)

ясен метод [199,](https://translate.googleusercontent.com/translate_f#199) 233

метод на кодовите точки [308](https://translate.googleusercontent.com/translate_f#308)

комбинирайки [226](https://translate.googleusercontent.com/translate_f#226) –227

сравнение и поръчка [230](https://translate.googleusercontent.com/translate_f#230)

обединяване [42](https://translate.googleusercontent.com/translate_f#42)

запитвания съдържание [228](https://translate.googleusercontent.com/translate_f#228)

трансформация съдържание [233](https://translate.googleusercontent.com/translate_f#233)

преобразуване на [200,](https://translate.googleusercontent.com/translate_f#200) 234, 237

преброяване писма [229](https://translate.googleusercontent.com/translate_f#229)

създаване на нови от други струни [226](https://translate.googleusercontent.com/translate_f#226)

метод на крипта [234](https://translate.googleusercontent.com/translate_f#234)

изтриване на символи [233](https://translate.googleusercontent.com/translate_f#233)

определяне на ASCII код [230](https://translate.googleusercontent.com/translate_f#230)

двойно цитиран, генериращ [222](https://translate.googleusercontent.com/translate_f#222)

метод downcase [231](https://translate.googleusercontent.com/translate_f#231)

метод each\_byte [306](https://translate.googleusercontent.com/translate_f#306)

each\_char метод [307](https://translate.googleusercontent.com/translate_f#307)

each\_line метод [307](https://translate.googleusercontent.com/translate_f#307)

празна? метод [228](https://translate.googleusercontent.com/translate_f#228)

изпразване [233](https://translate.googleusercontent.com/translate_f#233)

кодиране [235](https://translate.googleusercontent.com/translate_f#235)

шифроване [234](https://translate.googleusercontent.com/translate_f#234)

end\_with? метод [228](https://translate.googleusercontent.com/translate_f#228)

eql? метод [231](https://translate.googleusercontent.com/translate_f#231)

равно? метод [231](https://translate.googleusercontent.com/translate_f#231)

сравнение между половете [230](https://translate.googleusercontent.com/translate_f#230)

бягство знаци за регулярните изрази [351](https://translate.googleusercontent.com/translate_f#351)

ексклузивно - или операция с изброители

[324](https://translate.googleusercontent.com/translate_f#324) –326

глобална смяна [356](https://translate.googleusercontent.com/translate_f#356)

gsub метод [356](https://translate.googleusercontent.com/translate_f#356)

gsub! метод [356](https://translate.googleusercontent.com/translate_f#356)

шестнадесетичен метод [234](https://translate.googleusercontent.com/translate_f#234)

включва? метод [228](https://translate.googleusercontent.com/translate_f#228)

нарастващ [234](https://translate.googleusercontent.com/translate_f#234)

индексен метод [229](https://translate.googleusercontent.com/translate_f#229)

индексиране от края на [225](https://translate.googleusercontent.com/translate_f#225)

индивид, кодираща [236](https://translate.googleusercontent.com/translate_f#236)

ординатор метод [235](https://translate.googleusercontent.com/translate_f#235)

интерполация [220,](https://translate.googleusercontent.com/translate_f#220) 227, 351

итерация

чрез байтове [306](https://translate.googleusercontent.com/translate_f#306)

чрез символи [307](https://translate.googleusercontent.com/translate_f#307)

през редове [307](https://translate.googleusercontent.com/translate_f#307)

оправдание [232](https://translate.googleusercontent.com/translate_f#232)

метод на дължина [228](https://translate.googleusercontent.com/translate_f#228)

метод линии [308](https://translate.googleusercontent.com/translate_f#308)

ljust метод [232](https://translate.googleusercontent.com/translate_f#232)

lstrip метод [232](https://translate.googleusercontent.com/translate_f#232)

като аргументи на метода [241](https://translate.googleusercontent.com/translate_f#241)

методи, използващи регулярни изрази [353](https://translate.googleusercontent.com/translate_f#353)

следващ метод [234](https://translate.googleusercontent.com/translate_f#234)

еквилибристика [325](https://translate.googleusercontent.com/translate_f#325)

обектно идентичности [231](https://translate.googleusercontent.com/translate_f#231)

окт [234](https://translate.googleusercontent.com/translate_f#234)

метод на ордер [230](https://translate.googleusercontent.com/translate_f#230)

като квазибройни [306](https://translate.googleusercontent.com/translate_f#306)

заявки [228](https://translate.googleusercontent.com/translate_f#228)

регулярен израз от [350](https://translate.googleusercontent.com/translate_f#350)

премахване

герои от [225](https://translate.googleusercontent.com/translate_f#225)

крайните символи [233](https://translate.googleusercontent.com/translate_f#233)

[233.](https://translate.googleusercontent.com/translate_f#233) Серия

замести метод [233](https://translate.googleusercontent.com/translate_f#233)

представящ като базов номер [234](https://translate.googleusercontent.com/translate_f#234)

|  |
| --- |
| **504** |

ИНДЕКС

**504**

низове *(продължение)*

метод на rindex [229](https://translate.googleusercontent.com/translate_f#229)

метод rjust [232](https://translate.googleusercontent.com/translate_f#232)

метод [232](https://translate.googleusercontent.com/translate_f#232)

метод на сканиране [353](https://translate.googleusercontent.com/translate_f#353) –354

търсене отрицание [229](https://translate.googleusercontent.com/translate_f#229)

търсене на съвпадение на образец [225](https://translate.googleusercontent.com/translate_f#225)

задаване на част на нова стойност [225](https://translate.googleusercontent.com/translate_f#225)

едноцитиран, генериращ [221](https://translate.googleusercontent.com/translate_f#221)

метод за размер [228](https://translate.googleusercontent.com/translate_f#228)

размер, преобразуващ се [232](https://translate.googleusercontent.com/translate_f#232)

[225](https://translate.googleusercontent.com/translate_f#225) метод

парче! метод [225](https://translate.googleusercontent.com/translate_f#225)

метод на разделяне [355](https://translate.googleusercontent.com/translate_f#355)

Започни с? метод [228](https://translate.googleusercontent.com/translate_f#228)

лентов метод [232](https://translate.googleusercontent.com/translate_f#232)

[232](https://translate.googleusercontent.com/translate_f#232) серия

под метод [356](https://translate.googleusercontent.com/translate_f#356)

под! метод [356](https://translate.googleusercontent.com/translate_f#356)

подниза търсите [228](https://translate.googleusercontent.com/translate_f#228)

поднизове [224](https://translate.googleusercontent.com/translate_f#224) –226

метод Succ [234](https://translate.googleusercontent.com/translate_f#234)

метод swapcase [231](https://translate.googleusercontent.com/translate_f#231)

символи срещу [242](https://translate.googleusercontent.com/translate_f#242) –243

тестване за идентично съдържание [231](https://translate.googleusercontent.com/translate_f#231)

Тестване за начало или край [228](https://translate.googleusercontent.com/translate_f#228)

to\_f метод [204,](https://translate.googleusercontent.com/translate_f#204) 235

to\_i метод [204,](https://translate.googleusercontent.com/translate_f#204) 234, 245

to\_s метод [200,](https://translate.googleusercontent.com/translate_f#200) 235

метод [235](https://translate.googleusercontent.com/translate_f#235)

транскодиране [236](https://translate.googleusercontent.com/translate_f#236)

трансформация [231](https://translate.googleusercontent.com/translate_f#231) –232

upcase метод [231](https://translate.googleusercontent.com/translate_f#231)

StringScanner клас [355](https://translate.googleusercontent.com/translate_f#355)

лентов метод [232](https://translate.googleusercontent.com/translate_f#232)

strptime метод [247](https://translate.googleusercontent.com/translate_f#247)

подклас [77](https://translate.googleusercontent.com/translate_f#77)

подмножества [284](https://translate.googleusercontent.com/translate_f#284) –285

поднизове [224](https://translate.googleusercontent.com/translate_f#224) –226

успешен метод [234,](https://translate.googleusercontent.com/translate_f#234) 251

супер ключова дума [107,](https://translate.googleusercontent.com/translate_f#107) 109

супер метод, съчетаващ с

method\_missing [110](https://translate.googleusercontent.com/translate_f#110) –114

[77.](https://translate.googleusercontent.com/translate_f#77) суперклас

задвижване от суперсет на основната функционалност [408](https://translate.googleusercontent.com/translate_f#408)

суперсетове [284](https://translate.googleusercontent.com/translate_f#284) –285

.svn [376](https://translate.googleusercontent.com/translate_f#376)

превключватели, комбиниращи [25](https://translate.googleusercontent.com/translate_f#25)

Клас на символа [193](https://translate.googleusercontent.com/translate_f#193) , 220

Метод Symbol.all\_symbols [239](https://translate.googleusercontent.com/translate_f#239)

Символ # to\_proc [424](https://translate.googleusercontent.com/translate_f#424) –426

символен идентификатор [239](https://translate.googleusercontent.com/translate_f#239)

символи [237](https://translate.googleusercontent.com/translate_f#237)

добавяне към таблица със символи [240](https://translate.googleusercontent.com/translate_f#240)

метод [243](https://translate.googleusercontent.com/translate_f#243)

преобразуване на низове в [235](https://translate.googleusercontent.com/translate_f#235) , 237

преобразуване в низ [237](https://translate.googleusercontent.com/translate_f#237)

създаване на [237](https://translate.googleusercontent.com/translate_f#237)

дефиниран [75](https://translate.googleusercontent.com/translate_f#75)

като хеш ключ [241](https://translate.googleusercontent.com/translate_f#241)

идентификатори и [238](https://translate.googleusercontent.com/translate_f#238)

неизменност [238](https://translate.googleusercontent.com/translate_f#238)

буквален конструктор [237](https://translate.googleusercontent.com/translate_f#237)

като аргумент на метода [240](https://translate.googleusercontent.com/translate_f#240) –241

низоподобни методи [243](https://translate.googleusercontent.com/translate_f#243)

струни срещу [242](https://translate.googleusercontent.com/translate_f#242) –243

метод Succ [243](https://translate.googleusercontent.com/translate_f#243)

таблица [239](https://translate.googleusercontent.com/translate_f#239)

уникалност [238](https://translate.googleusercontent.com/translate_f#238)

upcase метод [243](https://translate.googleusercontent.com/translate_f#243)

синтактична захар [72](https://translate.googleusercontent.com/translate_f#72) , 162, 261, 268, 275

категории на за оператор / методи [195](https://translate.googleusercontent.com/translate_f#195)

за методи с извикване в стил оператор

нотация [195](https://translate.googleusercontent.com/translate_f#195)

infix оператори за извиквания на методи [193](https://translate.googleusercontent.com/translate_f#193)

за унарни оператори [196](https://translate.googleusercontent.com/translate_f#196)

синтаксични грешки, проверка за [12](https://translate.googleusercontent.com/translate_f#12) , 179

метод [369](https://translate.googleusercontent.com/translate_f#369)

метод [369](https://translate.googleusercontent.com/translate_f#369)

системни команди

отворен метод [451](https://translate.googleusercontent.com/translate_f#451) –452

Метод Open3.popen3 [452](https://translate.googleusercontent.com/translate_f#452)

системен метод [449](https://translate.googleusercontent.com/translate_f#449) –450

с помощта на обратни връзки [450](https://translate.googleusercontent.com/translate_f#450) –451

системен метод [449](https://translate.googleusercontent.com/translate_f#449) –450

метод [369](https://translate.googleusercontent.com/translate_f#369)

т

метод кран [403](https://translate.googleusercontent.com/translate_f#403)

TCPS сървър клас [441](https://translate.googleusercontent.com/translate_f#441) , 444

преобразуване на температурата [168](https://translate.googleusercontent.com/translate_f#168)

Клас на Tempfile [383](https://translate.googleusercontent.com/translate_f#383)

временна променлива [178](https://translate.googleusercontent.com/translate_f#178)

прекрати метод [440](https://translate.googleusercontent.com/translate_f#440)

метод на изпитване [374](https://translate.googleusercontent.com/translate_f#374)

след това ключова дума [153](https://translate.googleusercontent.com/translate_f#153)

Томас, Дейв [28](https://translate.googleusercontent.com/translate_f#28)

Клас на резбата [439](https://translate.googleusercontent.com/translate_f#439)

конци-местен глобални [445](https://translate.googleusercontent.com/translate_f#445)

нишки

чат сървър, използващ сокети и нишки [443](https://translate.googleusercontent.com/translate_f#443) –444

убивайки [440](https://translate.googleusercontent.com/translate_f#440) –441

общ преглед [438](https://translate.googleusercontent.com/translate_f#438) –440

започвайки [440](https://translate.googleusercontent.com/translate_f#440) –441

спиране [440](https://translate.googleusercontent.com/translate_f#440) –441

резбови ключове [446](https://translate.googleusercontent.com/translate_f#446) –449

сървър за дати с резба [441](https://translate.googleusercontent.com/translate_f#441) –443

променливи и [445](https://translate.googleusercontent.com/translate_f#445) –446

метод [threequal 161–](https://translate.googleusercontent.com/translate_f#161) 162, 194, 357

|  |
| --- |
| **505 серия** |

ИНДЕКС

**505**

време

добавяне / изваждане на секунди [251](https://translate.googleusercontent.com/translate_f#251)

аритметика [251](https://translate.googleusercontent.com/translate_f#251)

методи за преобразуване [251](https://translate.googleusercontent.com/translate_f#251)

спецификатори на формат [250](https://translate.googleusercontent.com/translate_f#250)

методи за форматиране [249](https://translate.googleusercontent.com/translate_f#249) –251

методи заявка [249](https://translate.googleusercontent.com/translate_f#249)

Клас по време [66,](https://translate.googleusercontent.com/translate_f#66) 246, 248

файл [66.](https://translate.googleusercontent.com/translate_f#66) time.rb

метод по време [171](https://translate.googleusercontent.com/translate_f#171) –173

Графична библиотека Tk [17](https://translate.googleusercontent.com/translate_f#17)

to\_ \* методи [200,](https://translate.googleusercontent.com/translate_f#200) 205

to\_a метод [202,](https://translate.googleusercontent.com/translate_f#202) 259

to\_ary метод [206,](https://translate.googleusercontent.com/translate_f#206) 259

to\_date метод [251](https://translate.googleusercontent.com/translate_f#251)

Метод to\_datetime [251](https://translate.googleusercontent.com/translate_f#251)

to\_enum метод [314,](https://translate.googleusercontent.com/translate_f#314) 317

to\_f метод [203](https://translate.googleusercontent.com/translate_f#203) –204, 235

to\_i метод 13 [,](https://translate.googleusercontent.com/translate_f#13) 73, 203

преобразуване на числата в десетични [245](https://translate.googleusercontent.com/translate_f#245)

[234](https://translate.googleusercontent.com/translate_f#234) серия

to\_proc метод [423](https://translate.googleusercontent.com/translate_f#423) –424

to\_s метод [235,](https://translate.googleusercontent.com/translate_f#235) 247

на разположение за всеки обект [200](https://translate.googleusercontent.com/translate_f#200)

дефиниране за нов обект [201](https://translate.googleusercontent.com/translate_f#201)

за родов нов обект [200](https://translate.googleusercontent.com/translate_f#200)

за имена на пътища [381](https://translate.googleusercontent.com/translate_f#381)

от Time class [407](https://translate.googleusercontent.com/translate_f#407)

метод to\_str [205](https://translate.googleusercontent.com/translate_f#205)

to\_sym метод [235,](https://translate.googleusercontent.com/translate_f#235) 237

метод TO\_TIME [251](https://translate.googleusercontent.com/translate_f#251)

инструмент, извън кутията [22](https://translate.googleusercontent.com/translate_f#22)

най-високо ниво, определено [122](https://translate.googleusercontent.com/translate_f#122)

методи от най-високо ниво

вграден [150](https://translate.googleusercontent.com/translate_f#150)

дефиниран в ядро [470](https://translate.googleusercontent.com/translate_f#470)

определящ [149](https://translate.googleusercontent.com/translate_f#149)

общ преглед [149](https://translate.googleusercontent.com/translate_f#149)

дефиниран от потребителя [470](https://translate.googleusercontent.com/translate_f#470)

транскодиране [236](https://translate.googleusercontent.com/translate_f#236)

вярно

обект [207](https://translate.googleusercontent.com/translate_f#207) , 209–210

състояние [207](https://translate.googleusercontent.com/translate_f#207)

състояние спрямо стойност [210](https://translate.googleusercontent.com/translate_f#210)

стойност [6](https://translate.googleusercontent.com/translate_f) , 43

TrueClass [209](https://translate.googleusercontent.com/translate_f#209)

стереотипа срещу превръщането [205](https://translate.googleusercontent.com/translate_f#205)

TypeError [180](https://translate.googleusercontent.com/translate_f#180)

U

ф бягство последователност [236](https://translate.googleusercontent.com/translate_f#236)

унарни оператори, персонализиране [196](https://translate.googleusercontent.com/translate_f#196)

метод на развързване [431](https://translate.googleusercontent.com/translate_f#431)

долна променлива [476](https://translate.googleusercontent.com/translate_f#476)

подчертаване, съвпадение [336](https://translate.googleusercontent.com/translate_f#336)

метод Uniq [266](https://translate.googleusercontent.com/translate_f#266)

uniq! метод [266](https://translate.googleusercontent.com/translate_f#266)

уникалност на символите [238](https://translate.googleusercontent.com/translate_f#238)

освен ако ключова дума [155](https://translate.googleusercontent.com/translate_f#155)

метод на развързване за директории [379](https://translate.googleusercontent.com/translate_f#379)

unshift метод [262](https://translate.googleusercontent.com/translate_f#262)

до ключова дума [167](https://translate.googleusercontent.com/translate_f#167)

метод [434](https://translate.googleusercontent.com/translate_f#434)

UTF-8 кодиране [235](https://translate.googleusercontent.com/translate_f#235)

V

-v превключвател [25](https://translate.googleusercontent.com/translate_f#25)

стойност? метод [274](https://translate.googleusercontent.com/translate_f#274)

стойност, непосредствена [56](https://translate.googleusercontent.com/translate_f#56)

стойности\_ при метод [262](https://translate.googleusercontent.com/translate_f#262)

стойности, сравняващи [6](https://translate.googleusercontent.com/translate_f)

променливи

задание [54](https://translate.googleusercontent.com/translate_f#54) –55, 57

блок, аромати [179](https://translate.googleusercontent.com/translate_f#179)

клас [7](https://translate.googleusercontent.com/translate_f#7)

класова йерархия и [141](https://translate.googleusercontent.com/translate_f#141)

общ преглед [138](https://translate.googleusercontent.com/translate_f#138) –141

плюсове и минуси [142](https://translate.googleusercontent.com/translate_f#142)

глобален [7](https://translate.googleusercontent.com/translate_f#7)

вграден [130](https://translate.googleusercontent.com/translate_f#130)

глобален обхват и [130](https://translate.googleusercontent.com/translate_f#130)

списък на английски.rb [131](https://translate.googleusercontent.com/translate_f#131)

списък [475](https://translate.googleusercontent.com/translate_f#475)

плюсове и минуси [131](https://translate.googleusercontent.com/translate_f#131)

пример [7,](https://translate.googleusercontent.com/translate_f#7) 67, 475

интроспекция [474](https://translate.googleusercontent.com/translate_f#474)

местни [7](https://translate.googleusercontent.com/translate_f#7) , 54, 133, 179

задание в условно [158](https://translate.googleusercontent.com/translate_f#158)

списък [475](https://translate.googleusercontent.com/translate_f#475)

сходство с извикване на метод [59](https://translate.googleusercontent.com/translate_f#59)

като етикет на обект [57](https://translate.googleusercontent.com/translate_f#57)

връзка с обект [55](https://translate.googleusercontent.com/translate_f#55)

обхват [54](https://translate.googleusercontent.com/translate_f#54)

временно [178](https://translate.googleusercontent.com/translate_f#178)

нишки и [445](https://translate.googleusercontent.com/translate_f#445) –446

vendor\_ruby директория [18](https://translate.googleusercontent.com/translate_f#18)

vendordir директория [17](https://translate.googleusercontent.com/translate_f#17)

подробен режим [25](https://translate.googleusercontent.com/translate_f#25)

—Версионен превключвател [25](https://translate.googleusercontent.com/translate_f#25)

W

% W (конструктор на литерален масив) [260](https://translate.googleusercontent.com/translate_f#260)

% w (конструктор на литерален масив) [259](https://translate.googleusercontent.com/translate_f#259)

-w флаг [12](https://translate.googleusercontent.com/translate_f#12) , 23

режим на предупреждение [23](https://translate.googleusercontent.com/translate_f#23)

предупреждения [159](https://translate.googleusercontent.com/translate_f#159)

Уейрих, Джим [29,](https://translate.googleusercontent.com/translate_f#29) 80, 413

|  |
| --- |
| **506 серия** |

ИНДЕКС

**506**

когато ключова дума [160](https://translate.googleusercontent.com/translate_f#160) –161

докато ключова дума

в началото срещу края на блок [166](https://translate.googleusercontent.com/translate_f#166)

в края на цикъл [166](https://translate.googleusercontent.com/translate_f#166)

като модификатор [167](https://translate.googleusercontent.com/translate_f#167)

празно пространство

в модели [343](https://translate.googleusercontent.com/translate_f#343)

премахнете от низове [232](https://translate.googleusercontent.com/translate_f#232)

заместващ [335](https://translate.googleusercontent.com/translate_f#335)

with\_index метод [256](https://translate.googleusercontent.com/translate_f#256)

текстообработваща граница котва [347](https://translate.googleusercontent.com/translate_f#347)

метод на писане

за файлове [369](https://translate.googleusercontent.com/translate_f#369)

за IO обекти [361](https://translate.googleusercontent.com/translate_f#361)

писане във файл [15](https://translate.googleusercontent.com/translate_f#15)

х

% x (оператор на системна команда) [250](https://translate.googleusercontent.com/translate_f#250)

x последователност за бягство [236](https://translate.googleusercontent.com/translate_f#236)

x модификатор [350](https://translate.googleusercontent.com/translate_f#350)

оператор% x [451](https://translate.googleusercontent.com/translate_f#451)

XML, Библиотека на Builder като инструмент за писане [413](https://translate.googleusercontent.com/translate_f#413)

операция XOR [324](https://translate.googleusercontent.com/translate_f#324)

Y.

% Y (спецификатор на формат на датата) [250](https://translate.googleusercontent.com/translate_f#250)

ключова дума [312](https://translate.googleusercontent.com/translate_f#312)

връщане срещу [169](https://translate.googleusercontent.com/translate_f#169) , 172

с аргументи [172](https://translate.googleusercontent.com/translate_f#172)

yielder [312](https://translate.googleusercontent.com/translate_f#312)

Z.

? z [374](https://translate.googleusercontent.com/translate_f#374)

\ Z котва в края на низа [347](https://translate.googleusercontent.com/translate_f#347)

\ z котва в края на низа [347](https://translate.googleusercontent.com/translate_f#347)

ZeroDivisionError [180](https://translate.googleusercontent.com/translate_f#180) –181, 184