январь 2009

habradigest

web * дизайн * разработка

№7

Ruby

habradigest

web * дизайн * разработка

Добрый день, читатель! В этом месяце из-за новогодних каникул всего один номер. Но зато номер вышел самым большим из всех: 20 статей общим объемом в 99 страниц.

По обложке вы уже, наверняка, поняли, что темой номера в этот раз является язык программирования Ruby. И это так. Седьмой номер содержит четыре статьи на тему Ruby, в том числе объемное введение в язык, в оригинале состоящее из одиннадцати частей от автора MaxEcl. Только у нас все эти статьи объединены в одну.

Темой Ruby этот выпуск не ограничивается, вы сможете найти множество статей на другие интересные темы: LaTex, JavaScript, .NET, Python и другие. Обязательно обратите внимание на интересную статью про влияние свойства hasLayout в Internet Explorer от автора Panya. Огромный интерес, так же представляет статья о реализации технологии Liquid Resize от Kotter. Кроме того, в выпуске есть статьи про генерацию паролей и учет пользователей с ограниченными способностями при создании сайтов. А автор smira написал очередную порцию материала про memcached в двух частях. Не пропустите.

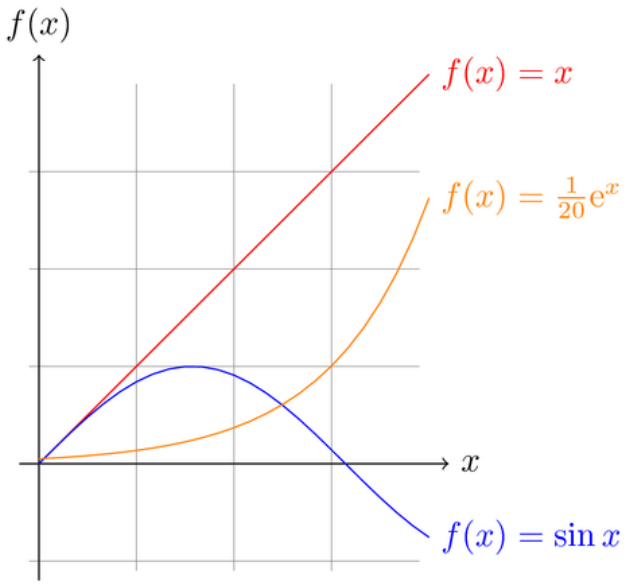
Владимир "ХаосCPS" Юнев

Векторная графика в LaTeX. Пакет PGF/TikZ

adrianopol

Доброго времени суток. Давно собирался рассказать о возможностях векторной графики в LaTeX, предоставляемых низкоуровневым макропакетом PGF и его расширением TikZ, а выход предыдущей статьи о пакете Xy-pic для создания диаграмм и графов и появление свободного времени сделали возможным начать работу :-).

Мне в своё время понадобилось найти и изучить какое-то гибкое средство для создания качественных векторных изображений, потому что уже достали криво масштабированные, вставленные с ужасным расширением картинки растровых форматов, портящие всё впечатление от документа, да и увеличивающие его размер в два раза из-за одной большой картинки с прямоугольником и несколькими подписями к нему. Имеющиеся возможности встроенного окружения **picture** весьма скудны; пакет PStricks ориентирован на язык PostScript (не работает с pdflatex, который мне необходим), хотя и может кое-что, чего не может PGF; система MetaPost, пожалуй, является наиболее мощной из всех в этой области, но функционирует с использованием отдельного интерпретатора со всеми вытекающими из этого последствиями. Таким образом, выбор пал на PGF/TikZ.



Основной целью сей статьи считаю наличие достаточного числа ссылок и небольшого количества примеров с описанием на русском языке. Этого в большинстве случаев достаточно, чтобы понять, нужно оно вам, или нет.

Очень приятно, когда для какого-то инструмента есть исчерпывающий список примеров, что он может делать и как этого достичь. Для обсуждаемого пакета этот список можно посмотреть, набрав в поисковике «pgf examples» и перейдя по первой же ссылке. Здесь все примеры одним списком: www.texample.net/tikz/examples/all. Нет смысла пытаться описать всё, на что способен пакет. Пожалуй, остановлюсь на основных концепциях.

В статье рассматривается использование pgf/tikz вместе с макропакетом LaTeX, как самым распространённым, однако pgf/tikz также работает и с другими. Текст статьи базируется преимущественно на руководстве TikZ & PGF. Manual for Version 2.00 (560 с.). Некоторые примеры взяты из него. Также в книге есть небольшая, но полезная глава Guidelines on Graphics (с. 65), посвящённая общим принципам работы над документами с векторной графикой, включая примеры того, как надо и как не надо делать; к ней стоит обратиться независимо от того, будете ли вы в дальнейшем использовать pgf/tikz или что-то другое.

Установка

В GNU/Linux надо поставить пакет pgf (и зависимый xcolor версии >= 2.00). Если такого нет, достаточно разархивировать исходники (локально или глобально) и сделать texhash (подробности см. в руководстве). В Windows pgf/tikz входит в состав дистрибутива MiKTeX, так что ничего дополнительного устанавливать не требуется.

Трансляция

Для получения pdf используем

```
pdflatex file.tex
```

Для получения ps используем две команды:

```
latex file.tex
dvips file.dvi
```

Также интересна возможность конвертации в HTML и SVG.

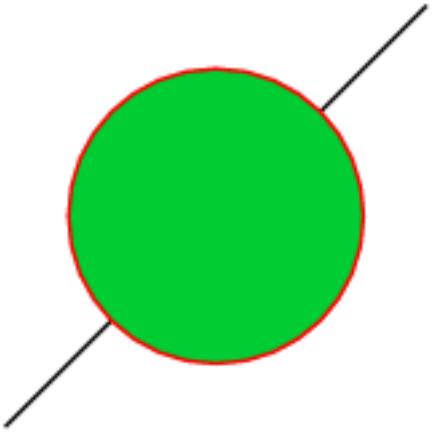
Использование

Для использования базовых возможностей пакета надо подключить его в преамбуле командой

```
\usepackage{tikz}
```

Для вставки команд pgf/tikz используется либо команда \tikz с одним аргументом (обычно для вставки рисунка в пределах строки), либо окружение \begin[options]{tikzpicture}...\end{tikzpicture}. Если в аргументе к \tikz надо указать несколько команд рисования, они окружаются фигурными скобками (\tikz[options]{...}). Каждая команда оканчивается точкой с запятой.

```
\documentclass{memoir}
\pagestyle{empty}
\usepackage{tikz}

\begin{document}
\tikz{\draw (-1,-1) -- (1,1); \path[fill=green!80!bl
ue,draw=red] (0,0) circle (7mm);}
\end{document}
```



Очень удобно, что не надо указывать размер холста. Программа сама вычислит его за вас.

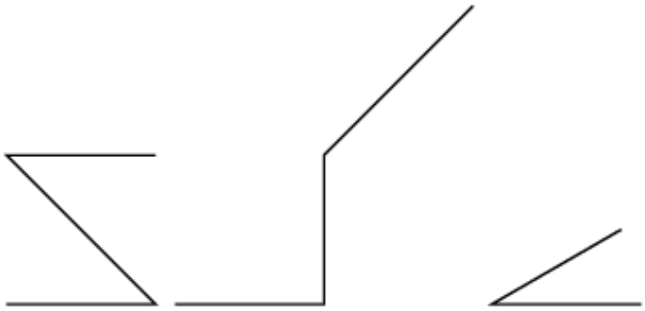
Точки

Точки задаются абсолютно или относительно.

Примеры декартовых и полярных координат:

— (2,0) — точка с заданными координатами в текущих единицах измерения (по умолчанию — 1 см);
— (2cm,-3pt) — 2 см в направлении Ox, –3 пункта в направлении Oy;
— (30:5cm) — точка на расстоянии 5 см от текущего положения в направлении 30 градусов;
— +(2,0) — точка, отстоящая от текущего положения на 2 единицы вправо (текущее положение не меняется);
— ++(2,0) — точка, отстоящая от текущего положения на 2 единицы вправо, которая становится новым текущим положением.

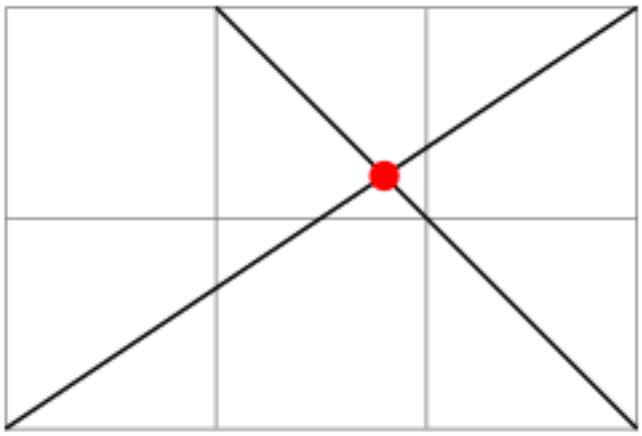
```
\tikz \draw (0,0) --  +(1,0) --  +(0,1) --  +(1,1);
\tikz \draw (0,0) -- ++(1,0) -- ++(0,1) -- ++(1,1);
\tikz \draw (1,0) -- (0,0) -- (30:1);
```



Также могут использоваться барицентрические координаты, система координат узлов (см. ниже), тангенциальная система.

Система координат пересечений позволяет оперировать с координатами пересечения (однако, пока только для комбинаций прямых и окружностей). Для этого задаётся первый и второй объекты (или линии их координатами) и номер пересечения (когда их несколько).

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) coordinate (A) -- (3,2) coordinate
(B)
        (1,2) -- (3,0);
  \fill[red] (intersection of A--B and 1,2--3,0)
circle (2pt);
\end{tikzpicture}
```



Можно задавать координаты с использованием относительных или абсолютных модификаторов длины или проекций. Это делается следующими способами:

— coordinate!number!angle:second_coordinate
— coordinate!dimension!angle:second_coordinate
— coordinate!projection_coordinate!angle:second_coordinate

Например,

— (1,2)!.25!(3,4) означает координату, которая находится на расстоянии одной четверти пути из (1,2) в

(3,4)
— (1,2)!1cm!(3,4) означает координату, которая находится на расстоянии 1 см от точки (1,2) на прямой (1,2)--(3,4)
— (1,2)!(0,5)!(3,4) означает координату, которая является проекцией точки (0,5) на прямую (1,2)--(3,4)

Траектории

Траектория — это последовательность прямых и кривых линий. Они создаются командой \path. Если она вызывается без аргументов, то ничего нарисовано не будет, что не очень интересно. Аргументами могут быть draw, fill, shade, clip или любая их комбинация. Вместо \path[draw], \path[fill], \path[shade,draw],… можно использовать сокращения \draw, \fill, \shadedraw.

Основная команда для создания прямых — это (a) -- (b), где (a) и (b) — некоторые точки.

Кубическая кривая Безье задаётся командой (a) .. controls (x) and (y) .. (b), где (a) и (b) — некоторые точки, (x) и (y) — опорные точки, влияющие на форму кривой. Если часть and (y) пропустить, считается, что (y) = (x).

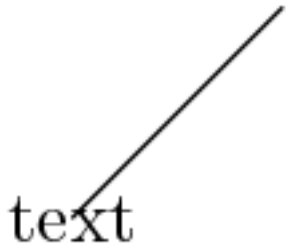
Графические параметры

Они указываются в квадратных скобках (как необязательные аргументы в LaTeX) в виде наборов ключ-значение: key=value. Например, \tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (1,0) -- cycle;. Иногда часть key= можно опускать, если ключ для данного значения трактуется однозначно.

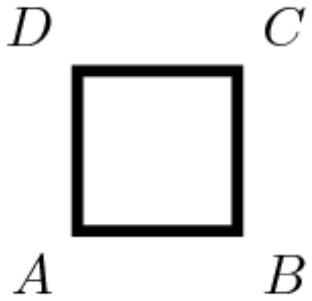
Узлы

Текст и подписи добавляются к рисунку в виде узлов (nodes). Они привязываются к текущему положению на плоскости (в пространстве), либо явно привязываются к некоторой точке. В качестве параметров можно указать якорь привязки узла (где он будет относительно точки: справа, слева, справа сверху и т.п.), форму узла (круг, квадрат, эллипс, ...), способ его отображения (нарисовать, залить) и другие.

```
\tikz \draw (1,1) node {text} -- (2,2);
```




```
\begin{tikzpicture}[line width=2pt]
  \draw (0,0) node [below left]  {$A$} --
        (1,0) node [below right] {$B$} --
        (1,1) node [above right] {$C$} --
        (0,1) node [above left]  {$D$} --
        cycle;
\end{tikzpicture}
```



Области видимости

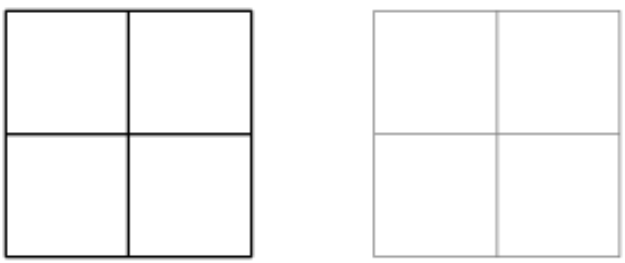
С помощью окружения scope можно создавать области видимости со своими параметрами; остальные наследуются из содержащей данную. Фактически, окружение tikzpicture ведёт себя аналогично scope. С определёнными ограничениями области видимости можно также применять при рисовании траектории.

```
\tikz \draw (0,0) -- (1,1)
            {[rounded corners] -- (2,0) -- (3,1)} --
            (3,0) -- (2,1);
```

Стили

Стили указываются при перечислении графических параметров. Есть набор уже определённых, которые можно переопределять. Можно определять свои.

```
\begin{tikzpicture}[scale=0.7]
  \draw             (0,0) grid +(2,2);
  \draw[help lines] (3,0) grid +(2,2);
\end{tikzpicture}
```



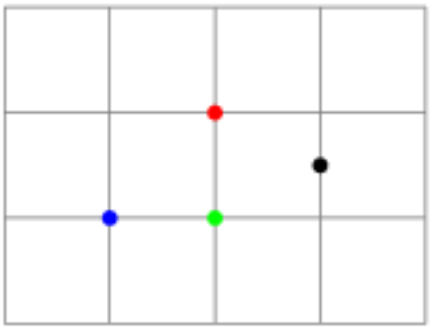
Вычисления

При подключении библиотеки calc командой в преамбуле

```
\usetikzlibrary{calc}
```

можно использовать некоторые математические вычисления для определения координат, например,

```
\begin{tikzpicture}[scale=0.5]
  \draw[help lines] (0,0) grid (4,3);
  \fill [red]   ($2*(1,1)$)              circle (2pt);
  \fill [green] (${1+1}*(1,.5)$)         circle (2pt);
  \fill [blue]  ($cos(0)*sin(90)*(1,1)$) circle (2pt);
  \fill [black] (${3*(4-3)}*(1,0.5)$)    circle (2pt);
  \end{tikzpicture}
```



Заключение

Возможности пакета можно перечислять ещё очень долго (например, я ничего не упоминал про графы, деревья и цепи), но такому перечислению не место в одной статье, тем более, что существует вышеупомянутый сайт и прекрасное руководство с исчерпывающими примерами использования каждой вкусности пакета pgf/tikz. Использовать его или нет — решать вам. Я с ним работаю меньше шести месяцев (да и узнал о нём не намного раньше), но уже активно использую и не жалею; во многих случаях он сильно экономил время и силы. Если у народа возникнет интерес, рассказ может быть продолжен.

PS.
Изображения делались так:

```
$ latex qq.tex && dvips -K -o qq.ps qq.dvi && convert
-antialias -seed 0 -density 200 -trim qq.ps qq.png
```

Делаем Liquid Resize своими руками

Kotter

Вы наверное уже слышали о технологии масштабирования Liquid Resize, которая учитывает содержимое изображения. Если вам интересно как оно все работает и как можно реализовать все это самому, то читайте далее

Что такое Seam Carving?

Seam Carving — это алгоритм изменения размеров изображения, который учитывает содержимое изображения (как называют авторы — Content-Aware Image Resizing Algorithm). Впервые был продемонстрирован в 2007 году и вызвал немалый интерес. Наверное многие читатели Хабра про него слышали, поскольку тут уже были статьи на эту тему.

Возникает тогда вопрос: как же этот алгоритм учитывает содержимое изображений? В теории тут все просто — сначала вычисляется важность разных участков изображения, после чего изменяются в размере только те части, которые, по мнению алгоритма, являются не такими важными. На практике же для реализации этого алгоритма надо решить ряд вопросов: как определить насколько важными являются части изображения, как найти и сжать менее важные участки изображения и т. д.

В этой статье я попытался ответить на эти, а также на ряд других вопросов, возникающих при решении данной задачи.

Те, кто уже сейчас хотят посмотреть что у меня в итоге получилось, могут в конце статьи найти примеры изображений и демо-программу с прокомментированными исходниками.

Несколько замечаний по прочтению статьи

Перед самой статьей хочется оговорить некоторые условности.

Во-первых. В статье я часто использую термин «энергия», хотя и не уверен в правильности этого термина. А использую я его потому, что сами авторы называют это английским словом «energy». Энергия точки означает ее важность в изображении. Чем больше энергия — тем важнее точка.

Во-вторых. В статье рассматривается только изменение размера по-горизонтали. При одновременном изменении размера по обоих направлениях принцип остается тот-же, только мы теперь рассматриваем не только вертикальные цепочки пикселей, но и горизонтальные.

В-третьих. У меня нету опыта написания под C#/.NET. Выбор платформы/языка написания программы был обусловлен простотой кода для понимания и широким охватом аудитории. Поэтому если я что-то написал в программе неэффективно или не «по дотнетовски», то не обращайте на это внимание. Не в этом суть программы.

В-четвертых. Русский для меня не родной язык, поэтому извините за возможные ошибки.

И последнее. Код программы не оптимален. Между простотой кода и оптимальностью кода я выбирал простоту, поэтому если вы захотите где-то использовать этот алгоритм — оптимизируйте его под свою задачу. Благо простор для оптимизаций там есть.

Вот, собственно, с прелюдией закончили, теперь переходим к самому интересному...

Общая схема алгоритма

Весь алгоритм состоит из таких составных частей:

1. Нахождение энергии каждой точки. На этом этапе нам надо узнать какие части изображения являются более важными, а какие — менее важными, исходя их этих данных мы в последствии будем менять размер изображения.
2. Нахождение такой вертикальной цепочки пикселей, чтобы суммарная энергия пикселей, которые входят в эту цепочку была минимальной. Цепочка пикселей — это такой набор пикселей, что в каждой строке выбрано ровно по одному пикселю, и соседние пикселы в ней соединены сторонами или углами. Если мы найдем такую цепочку то сможем ее удалить из изображения, при этом минимально затронув информационное наполнение изображения.
3. Когда мы нашли цепочку с минимальной энергией, то нам остается ее только удалить, если нам надо уменьшить изображение, или растянуть, если надо увеличить изображение.

Теперь рассмотрим каждый пункт более детально.

Вычисление энергии точек

Первым делом нам надо решить какие участки изображения важны, а какие не очень.

Для решения этой задачи авторы алгоритма предлагают для каждого пикселя посчитать его «энергию» — то есть какой-то условный показатель (в попугаях), который буде показывать важен ли данный пиксел в данном изображении, или не очень.

В принципе способов это сделать много: от самых простых (например посчитать изменения цвета по сравнению с соседними) до достаточно сложных (например сделать анализ акцентирования внимания человека). По словам самих авторов, они проверили множество функций энергии, и одна из самых простых функций дала один из лучших результатов. Поэтому мы воспользуемся сейчас этой функцией. Вот она:

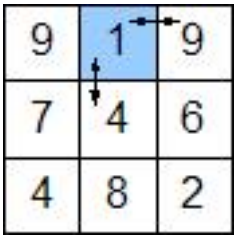
$$e_1(\mathbf{I}) = \left|\frac{\partial}{\partial x}\mathbf{I}\right| + \left|\frac{\partial}{\partial y}\mathbf{I}\right|$$

Если вы при виде данной формулы с производными испугались, что вам надо будет вспоминать матанализ, то зря. На самом деле она формулируется достаточно просто: энергия пикселя равна изменению цвета соседних пикселей по сравнению с данным пикселом. То есть чем больше разница в цвете между данным пикселем и соседними — тем больше его энергия.

Теперь рассмотрим реализацию вычислений энергии на практике. Пусть точки изображения характеризуются своей интенсивностью. У нас есть следующее изображение размером 3x3 точки:

9	1	9
7	4	6
4	8	2

Сначала посчитаем по модулю разницу интенсивностей между пикселем и его соседями (справа и снизу), а потом вычислим энергию этого пикселя как среднее арифметическое этих значений.



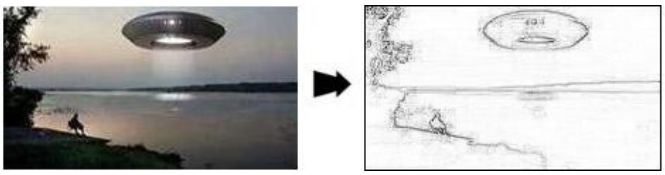
Для выделенного пикселя: разница между ним и соседом справа — 8, соседом снизу — 3. Среднее арифметическое — (8+3)/2 = 5.5. Но поскольку с целыми числами работать удобнее и быстрее, а такая точность излишняя, то отбросим дробную часть. То есть энергия выделенного пикселя равна 5.

Аналогично посчитаем для остальных пикселей. При этом пиксели, которые крайние справа и крайние снизу будут иметь только соседа справа или снизу, поэтому для них значение разницы и будет средним арифметическим. Для пикселя в правом-нижнем углу таких соседей вообще нет, поэтому мы просто примем его энергию за 0. Хотя теоретически можно и для него посчитать энергию, но не будем усложнять себе жизнь, поскольку на практике им можно пренебречь.

В результате получим следующую матрицу энергий пикселей:

5	5	3
3	3	4
4	6	0

Если же у нас пиксели характеризуются не просто интенсивностью, а значением интенсивности отдельно для R, G, B, то энергия пикселя равна сумме энергий по каждой из компонент.
Пример карты энергий для изображения:



На этой картинке чем темнее цвет на карте энергий — тем больше энергия.

Я реализовал нахождение энергий пикселей с помощью следующей функции:

```
private void FindEnergy()
{
  for (int i = 0; i < imgHeight; i++)
    for (int j = 0; j < imgWidth; j++)
    {
      energy[i, j] = 0;

        // Считаем отдельно для компонент R, G, B
      for (int k = 0; k < 3; k++)
      {
        int sum = 0, count = 0;

          // Если пиксел не крайний снизу, то добавляем
в sum разность между текущим пикселом и соседом снизу
```

```
        if (i != imgHeight - 1)
        {
          count++;
          sum += Math.Abs((int)image[i, j, k] - (int)
image[i + 1, j, k]);
        }

          // Если пиксел не крайний справа, то добав-
ляем в sum разность между поточным пикселом и соседом
справа
        if (j != imgWidth - 1)
        {
          count++;
          sum += Math.Abs((int)image[i, j, k] - (int)
image[i, j + 1, k]);
        }

          // В массив energy добавляем среднее арифме-
тическое разностей пикселя с соседями по k-той компо-
ненте (то есть по R, G или B)
        if (count != 0)
          energy[i, j] += sum / count;
      }
    }
}
```

Функция работает с глобальными переменными: energy — массив, в который записываются энергии и image — массив, в котором хранится изображение.

Нахождение цепочки с минимальной суммарной энергией

У нас уже есть значения энергий для каждого пикселя, но теперь нам надо выбрать те пиксели у которых значение энергии минимальное. Но если мы начнем забирать/добавлять произвольные пиксели, то само изображение у нас просто расползется, и деформируется до неузнаваемости. Конечно такой результат нам не подходит. Поэтому сначала надо выбрать цепочку пикселей, а потом уж их удалять или прибавлять. При чем не произвольную цепочку, а «правильную».

«Правильная» цепочка — это набор точек, такой, что в каждой строчке изображения выбран ровно 1 пиксел, а пикселы в соседних строчках «соединены» или сторонами, или углами.

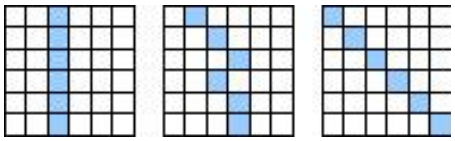


Рисунок №1. Пример «правильных» цепочек

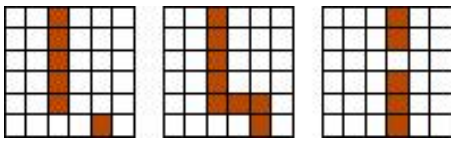


Рисунок №2. Пример «неправильных» цепочек

Теперь мы знаем, что удалять надо «правильные» цепочки, тогда мы не сильно испортим изображение. Но какую цепочку тогда выбрать? Авторы алгоритма предлагают выбрать для удаления/растяжение те цепочки, сумма энергий пикселей, которые входят в нее, минимальна.

Отсюда возникает вполне естественный вопрос: как нам найти такую цепочку?

Вариант первый — перебираем все цепочки, их суммируем, и находим с минимальной суммой. Вариант, конечно, интересный, но тогда для обработки даже небольшого изображения нам надо будет ждать вечность :) Если у нас есть вечность в запасе, то пишем код полного перебора, запускаем и ждем. Если вы хотите увидеть результат за более короткий промежуток времени — то читаем дальше.

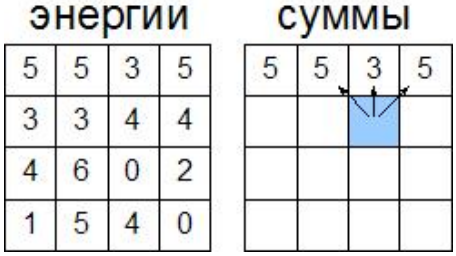
Тут нам на помощь приходить динамическое программирование. Не буду углубляться в детали, и показывать в чем именно «динамичность» программирования (кстати термин «программирование» тут отношения к написанию кода не имеет), и сразу перейду непосредственно к алгоритму. Если кто-то не знает, что такое «динамическое программирование» и хочет узнать — может почитать об этом на Википедии, или других источниках.

Сначала мы создадим новый массив, который по размеру равен массиву с энергиями пикселей. В этот массив мы для каждого пикселя запишем сумму элементов минимальной цепочки пикселей, которая начинается у верхнего края изображения и заканчивается на данном пикселе.

Покажем процесс вычисления на примере. Пусть у нас есть массив с значениями энергий для каждого пикселя:

5	5	3	5
3	3	4	4
4	6	0	2
1	5	4	0

Будем вычислять элементы этого массива по строчкам — от первой к последней. Для верхней строчки элементы данного массива будут равны элементам массива, поскольку из каждого такого элемента можно только 1 цепочку построить (все единичной длины):



Перейдем к третьей строчке. В принципе, третья строчка считается аналогично второй. Снова рассмотрим выделенную ячейку. От нее мы можем образовать такие цепочки:

— эта ячейка плюс минимальная цепочка длинны 8;
— эта ячейка плюс минимальная цепочка длинны 6;
— эта ячейка плюс минимальная цепочка длинны 7;

Из этих вариантов опять выбираем минимальный (6) и прибавляем к энергии самой ячейки (6). Получаем значение этой ячейки равно 12. Аналогично считаем остальные элементы третьей строчки.

После вычисления таким образом всех строчек мы получим в результате следующий массив:

энергии

5	5	3	5
3	3	4	4
4	6	0	2
1	5	4	0

суммы

5	5	3	5
8	6	7	7
10	12	6	9
11	11	10	6

Если формализовать вычисление этого массива, то получим следующую формулу:

$$s[i,j] = \begin{cases} e[i,j] & \text{if } i = 0 \\ e[i,j] + Min(s[i-1,j-1], s[i-1,j], s[i-1,j+1]) & \text{if } i \neq 0 \end{cases}$$

где s — наш массив сумм, а e — массив энергий.

Мой вариант реализации вычисления массива сумм:

```
private void FindSum()
{
     // Для верхней строчки значение sum и energy бу-
дут совпадать
   for (int j = 0; j < imgWidth; j++)
     sum[0, j] = energy[0, j];

     // Для всех остальных пикселей значение элемента
(i,j) массива sum будут равны
     //  energy[i,j] + MIN ( sum[i-1, j-1], sum[i-1,
j], sum[i-1, j+1])
   for (int i = 1; i < imgHeight; i++)
     for (int j = 0; j < imgWidth; j++)
     {
       sum[i, j] = sum[i - 1, j];
       if (j > 0 && sum[i - 1, j - 1] < sum[i, j])
sum[i, j] = sum[i - 1, j - 1];
       if (j < imgWidth - 1 && sum[i - 1, j + 1] <
sum[i, j]) sum[i, j] = sum[i - 1, j + 1];

       sum[i, j] += energy[i, j];
     }
}
```

Функция работает с глобальными переменными: energy — массив, в котором записаны энергии и sum — массив, в который записываются значения сумм. Теперь, когда у нас уже есть этот массив, настало время задуматься — а для чего он вообще нам надо? Отвечу — по этому массиву мы теперь можем быстро найти цепочку с минимальной суммой энергий.

Сначала мы найдем какой пиксел из нижней строчки изображения принадлежит этой цепочке: элемент, что мы ищем, будет иметь наименьшее значение в массиве сумм среди элементов нижней строчки. Почему? Вспомните, что в данном массиве записаны значения сумм элементов минимальных цепочек от верхнего края до данного пиксела. Цепочки, которые нас интересуют заканчиваются на пикселе из последнего ряда. Соответственно для всего рисунка минимальная цепочка будет выбрана как минимальная из всех минимальных цепочек, которые заканчиваются на пикселях из нижней строчки.

Для нашего примера это будет следующий элемент:

энергии

5	5	3	5
3	3	4	4
4	6	0	2
1	5	4	0

суммы

5	5	3	5
8	6	7	7
10	12	6	9
11	11	10	6

Мы уже знаем на каком пикселе заканчивается минимальная цепочка, теперь мы можем найти пиксел из предпоследней строчки. Как уже было написано, из нижнего пиксела мы можем пойти только в 3 соседние пиксели на строчку выше: слева-сверху, сверху или справа-сверху. Среди этих пикселей выбираем пиксел с минимальным значением в массиве сумм, и переходим к нему. Продолжаем, пока на дойдем до верхней строчки. Процесс показан на следующем рисунке:

энергии

5	5	3	5
3	3	4	4
4	6	0	2
1	5	4	0

суммы

5	5	3	5
8	6	7	7
10	12	6	9
11	11	10	6

После выполнения всех этих операций мы получим то, чего и хотели — набор пикселей, которые мы можем изменить с минимальными потерями для изображения.

Пример минимальной цепочки на рисунке:

Программная реализация алгоритма поиска цепочки:

```
private int[] FindShrinkedPixels()
{
    // Номер последней строчки
  int last = imgHeight - 1;
    // Выделяем память под массив результатов
  int[] res = new int[imgHeight];

    // Ищем минимальный элемент массива sum, кото-
рый находиться в нижней строке и записываем результат в
res[last]
  res[last] = 0;
  for (int j = 1; j < imgWidth; j++)
    if (sum[last, j] < sum[last, res[last]])
      res[last] = j;

    // Теперь вычисляем все элементы массива от предпо-
следнего до первого.
  for (int i = last - 1; i >= 0; i--)
  {
      // prev - номер пикселя цепочки из предыдущей
строки
       // В этой строке пикселями цепочки могут быть
только (prev-1), prev или (prev+1), поскольку цепочка
должна быть связанной
    int prev = res[i + 1];

      // Здесь мы ищем, в каком элементе массива sum,
из тех, которые мы можем удалить, записано минимальное
значение и присваиваем результат переменной res[i]
    res[i] = prev;
    if (prev > 0 && sum[i, res[i]] > sum[i, prev - 1])
res[i] = prev - 1;
    if (prev < imgWidth - 1 && sum[i, res[i]] > sum[i,
prev + 1]) res[i] = prev + 1;
  }

  return res;
}
```

Уменьшение рисунков

Мы, наконец-то, нашли цепочку пикселей, с которыми будем работать. Теперь можно вспомнить зачем вообще мы ее искали. Наша задача — изменение размера изображения. Увеличение и уменьшение изображение немного отличаются, поэтому сначала рассмотрим уменьшение, а потом уже увеличение.

Если нам надо уменьшить ширину картинки на 1 пиксел, то тут все просто: находим вертикальную цепочку, как это описывается выше, и просто удаляем ее из изображение. Я реализовал эту операцию с помощью следующего цикла:

```
for (int i = 0; i < imgHeight; i++)
   for (int j = cropPixels[i]; j < imgWidth; j++)
   {
     pImage[i, j] = pImage[i, j + 1];
   }
```

В i-м элементе массива cropPixels записан номер столбика пикселя, который мы удаляем из i-й строчки. А сам цикл делает следующее: все пиксели, которые записаны справа от тех, которых удаляем, мы сдвигаем на одну точку влево. В результате этой операции наша цепочка пикселей будет удалена из изображения.

Если нам надо уменьшить изображение не на 1 пиксел, а на большее значение, то выполняем операцию удаления цепочки столько раз, сколько надо (каждый раз при этом нам надо будет эту цепочку искать).

Хочу обратить внимание, что операция уменьшения изображения на n пикселей и операция уменьшения изображения на 1 пиксел n раз не являются эквивалентными. Их отличие в том, что если мы уменьшаем на 1 пиксел n раз, то мы для каждой из промежуточных изображений ищем энергию пикселей, а если сразу уменьшаем на n, то энергии пикселей мы берем из исходного изображения.

Увеличение рисунков

Уменьшать изображения мы уже умеем, теперь надо выяснить, как же их надо увеличивать.

Первое, что приходит на ум — это точно так же, как и при уменьшении, выбирать цепочки и их «растягивать». Но если мы это реализуем мы получим такой результат:

Как мы видим, программа просто взяла один столбик и его растянула, а это не совсем то, что нам надо.

Следующая мысль может быть, например, такой: надо брать не одну минимальную цепочку, а столько, сколько надо, чтобы дополнить рисунок до нужной ширины. Допустим мы реализовали данный алгоритм, но что будет, если мы увеличим рисунок в 2 раза?

Как видно из этого рисунка — мы просто выбрали все точки, и растянули их по горизонтали, но это тоже не совсем то, что нам надо, поскольку этот способ не отличается от обычного растягивания.

Но в принципе сама идея последнего способа правильная, но с небольшим изменением: нам надо увеличивать изображение поэтапно, так, чтобы на

каждом этапе охватывалось как можно больше «не важных» частей изображения, и как можно меньше «важных». Тогда возникает вопрос, как разбить процесс увеличение на этапы? Вариантов много — от разбиения на этапы вручную, до написания каких-то эвристических анализаторов. Но в своей программе я написал просто — разбивание на этапы происходит таким образом, чтобы на каждом этапе изображение не увеличивалось больше, чем на 50%. Иногда это дает приемлемый результат, иногда не очень, но, как я уже написал, вариантов реализации разбиения на этапы можно придумать много.

Если увеличить таким образом нашу картинку с НЛО, то получим следующий результат:

Как видим, НЛО, человек и дерево остались не измененными.

Примеры

Все примеры масштабирования рисунков были получены с помощью программы, которую я написал параллельно с этой статьей, реализовав алгоритмы, которые здесь описываются.

Исходники и бинарник программы можете скачать внизу статьи.

Рисунок №3. Классический пример Content-Aware Resizing'a

Рисунок №4. Ярко выражены «важные» объекты и фон

(оригинал) (уменьшено) (увеличено)

Рисунок №5. Рисунок звездного неба. При изменении размера, форма звезд почти не изменяется.

(оригинал) (уменьшено) (увеличено)

И еще пример масштабирования двух фотографий, сделанных gmm .

(оригинал) (уменьшено) (увеличено)

(оригинал) (уменьшено) (увеличено)

Итоги

Итак, что же мы получили в результате:

— Знание принципа работы такого интересного и

полезного алгоритма, как Seam Carving;
— Программку, в которой можно поиграться в изменителя размеров изображений (конечно есть еще Photoshop и http://rsizr.com/, но в этих случаях у вас нет исходников);
— Простор для воображения по возможным улучшениям алгоритма (например можно детектор лиц к нему прицепить, результаты, думаю, будут получше);
— Знание, что Content-Aware Rescale в Photoshop это не особая фотошоповская магия, а обычный алгоритм, который поддается пониманию.

Спасибо тем, кто дочитал статью до конца, надеюсь было интересно.

P. S.

Исходники (VS 2008)
Бинарник (.NET)

Полезные линки по теме:

— PDF, в котором авторы описывают алгоритм (англ, ~18 Мб).
— Улучшение способа подсчета энергии Sobol operator(англ).
— Динамическое программирование на русской Википедии. ДП использовалось в этом алгоритме, но имеет куда большую сферу применения, поэтому очень рекомендуется почитать для тех, кто с ним не знаком.
— По просьбе prokoudine , упомяну о некоторых свободных реализациях алгоритма:
• Расширение Liquid Rescale для GIMP и ImageMagick, основаны на библиотеке liblqr,
• SeamCarvingGUI основанной на библиотеке CAIR

Генерация мнемонически сильных паролей

stboris

В своей жизни я часто сталкиваюсь с необходимостью придумать пароль для своего (а также и не своего) нового аккаунта/логина. Пароли должны быть достаточно сложны, иначе они могут быть легко подобраны (привет счастливым обладателям паролей god, sex, love). Также они должны по возможности быть разными, т.к. если вы даже и придумали очень сложный пароль, но пользуетесь им везде, то это легко может привести к компрометации. Запомнить несколько десятков паролей, состоящих из случайного набора букв разных регистров, цифр, спецсимволов не такая уж тривиальная задача.

Моя память работает по какому-то своему особому принципу, что-то вроде «тут помню, тут не помню». С девушкой одного моего друга я знакомился 4 раза, не мог ее запомнить и все тут. Но есть у моей памяти и бонус — она очень хорошо работает с ассоциациями. А еще, раньше я весьма много играл в разные игрушки: Кваку, Линягу, BoB и много еще чего сетевого и не очень и давно заметил, что для записи ников часто используют не только буквы, но и цифры и специальные символы.

И вот, однажды, сопоставив это все, я и придумал свой метод генерации паролей (возможно, его кто-то придумал до меня, но я пока не видел). Я уже достаточно давно им пользуюсь, но до сих пор не сделал четких правил для него. Так что решил написать эту статью и с ее помощью довести метод до ума и заодно поделиться им с людьми.

Задача

Создать метод, по которому было бы легко придумывать надежные, но легко запоминающиеся пароли.

Решение

Допустим, я решил поиграть в онлайн-игру «Овощной магнат» (игра выдумана, все совпадения случайны). Создаю там аккаунт и мне нужен пароль.

Осмысленность

Для легкого запоминания пароль должен быть осмысленным, т.е. это должно быть слово или фраза. Много кому известно, что человек хорошо запоминает ассоциации, поэтому в идеале это слово/фраза должны быть ассоциативно привязаны к тому, для чего мы создаем пароль.

У меня возникла ассоциация «баклажан»

Латиница

Во избежание зла (проблемы с кодировкой), пароль должен использовать английскую раскладку. Также слово/фраза в пароле не должно быть записано по-русски (по-украински, по-арабски, по-китайски и т.д.) с использованием английской раскладки иначе можно очень сильно встрять при отсутствии русской (подставить по выбору) клавиатуры. Также нельзя использовать транслит, т.к. нет единого стандарта транслита. Отсюда мораль – самым логичным решением будет перевести слово на язык, использующий только латинский алфавит (никаких диакритических знаков, кириллицы, иероглифов и т.п.). Например, английский или латынь. Кстати, использование латыни в данном случае очень интересно – никто никогда в здравом уме не будет делать словарь для брутфорсера под латынь.

Поскольку я не знаю латыни — перевожу на английский. Получилось — aubergine

Регистр

В пароле должны присутствовать буквы в разных регистрах. Для простоты запоминания мы переводим половину слова в верхний регистр. Тут также есть несколько вариантов: правая — левая половины и, если количество букв нечетное, со средней или без.

Я беру первую половину со средней буквой и перевожу в верхний регистр. Получилось — AUBERgine

Спецсимволы

Также в пароле должны присутствовать цифры и специальные символы. Для этого надо для части букв алфавита придумать замену из зрительно похожих символов и цифр. Пароль после этого останется вполне удобочитаемым. Кстати для кул-хацкеров и про-геймеров в этом нет ничего нового, многие из них свои ники именно так и записывают. Например, a=@, e=3, i=! и т.д. Нюанс — не нужно заменять весь алфавит, достаточно лишь некоторой части, т.к. может получиться, что пароль будет состоять из одних лишь спецсимволов – тоже не очень хорошо.
Есть вариант использовать спецсимволы только из цифровых клавиш, т.к. если в каком-либо сервисе не будет возможности использовать спецсимволы, вы

просто «дауншифтите» их на соответствующие им цифры. Например @=2, !=1 и т.д.

Применяем замену. Получилось — @UB3Rg!n3
В случае с сервисом, не позволяющем использовать спецсимволы. Получилось — 2UB3Rg1n3

Длина пароля

Для этого вводятся понятия «открывающего» и «закрывающего» символа. Иногда нужно сделать пароль короче (бывает, ставят ограничение на длину) для этого мы можем убрать «символы». Некоторые сервисы криво работают с паролями, начинающимися со спецсимволов. В частности, Aurvote для дистрибутива Arch Linux некорректно обрабатывает пароли, начинающиеся с символа «$». Поэтому может быть логичным «открывающий» символ делать цифрой, а «закрывающий» — спецсимволом.

Добавляю «открывающую» цифру «1» и «закрывающий» спецсимвол «)». Получилось — 1@UB3Rg!n3)

Чтож, помоему не так уж и плохо — одиннадцатисимвольный пароль, содержащий все типы символов и при этом достаточно свободно читается, по большому счету требуется помнить только слово-ассоциацию.

Итого

В этой статье я специально не давал готовых решений, т.к. каждый, кто хочет использовать данную систему, должен сам выбрать некоторые моменты в зависимости от того, что ему по душе и что он лучше запомнит.

Плюсы

Относительная устойчивость к подбору, легкость запоминания, быстрота генерации пароля (я бывало, по часу раньше сидел, пытаясь придумать хороший пароль), модульность (можно отбросить те правила, которые вам не нравятся или добавить новые).

Минусы

Могут быть небольшие проблемы, если вы печатаете на нестандартной клавиатуре, но это решаемо (находите в интернете картинку стандартной раскладки). Данный пароль слабее, чем случайно сгенерированный пароль, т.к. можно составить словарь на основе этих правил, но никто не мешает вам добавить свои правила, о которых никто кроме вас не знает.

В любом случае – это не панацея, а, как и очень многое в нашей жизни, компромисс между надежностью и удобством.

Рекомендации

Регулярная смена паролей повышает безопасность. Смена вариантов правил повышает надежность системы, но усложняет восстановление пароля в памяти через долгое время. Одинаковые пароли в разных местах очень сильно снижают безопасность. Если кто-то знает ответ на ваш контрольный вопрос, никакой пароль не поможет. Есть еще много вариантов, как украсть ваш пароль так, что сильный пароль не панацея и нужно следить еще много за чем, но это выходит за рамки этой статьи.

Программы для хранения паролей

В этой статье нет ни слова о программах для хранения паролей в силу того, что статья не о них =) Тем более, что я пока не пользуюсь ни одной из них, т.к. просто не было случая/периодически забываю флешку где-нибудь/не везде можно подрубить флешку, а пароли все равно вводить надо. Да и вообще одно другому не мешает.

IronPython как движок для макросов в .NET приложениях

alek_sys

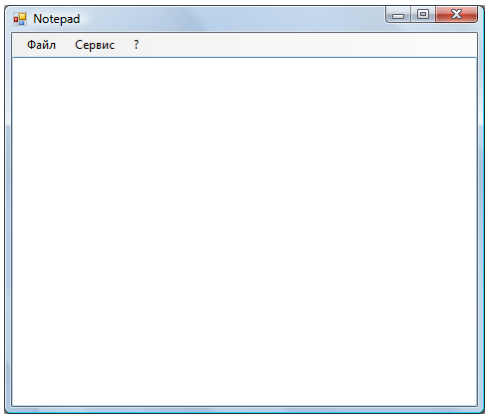
Подозреваю, многие из вас задумывались — как можно в .NET приложение добавить поддержку макросов — чтобы можно было расширять возможности программы без ее перекомпиляции и предоставить сторонним разработчикам возможность легко и просто получить доступ к API вашего приложения? В статье рассмотрено, как в качестве основы для выполнения макросов использовать IronPython — реализацию языка Python на платформе .NET.

Для начала, следует определится — что мы будем иметь в виду под словом «макрос» — это скрипт, который без перекомпиляции проекта позволял бы получить доступ к определенному API. Т.е. вытаскивать значения с формы, модифицировать их — и все это в режиме run-time, без модификации приложения.

Первым вариантом, который приходит на ум будет создание собственного интерпретатора для простенького скрипт-языка. Вторым — будет динамическая компиляция какого-нибудь .NET языка (того же C#) — с динамической же подгрузкой сборок и выполнением через Reflection. И третий — использование интерпретируемых .NET языков (DLR) — IronPython или IronRuby.

Создавать свой язык + интерпретор к нему с возможностью .NET interoperability — задача нетривиальная, оставим ее для энтузиастов.
Динамическая компиляция — слишком громоздко и тащит за собой использование Reflection. Однако, этот метод не лишен преимуществ — написанный макрос компилируется единожды и в дальшейшем может использоватся многократно — в виде полноценной .NET сборки. Итак — финалист — метод номер три — использование существующих DLR языков. В качестве такого языка выбираем IronPython (примите это как факт :). Текущая версия IPy — 2.0, взять можно на codeplex.com/IronPython

Перейдем непосредственно к кодированию.
Для начала, рассмотрим интерфейс тестового приложения «Notepad».



В меню «Сервис» и разместим пункт «Макросы». Для примера рассмотрим простейший вариант формирования списка макросов — в каталоге с программой создадим папку «Macroses» файлы из этой папки станут пунктами меню.

```
private void Main_Load(object sender, EventArgs e)
    {
      MacrosToolStripMenuItem itm = null;
      string[] files = Directory.GetFiles(@».\
Macroses»);
      foreach (string file in files)
      {
        itm = new MacrosToolStripMenuItem(Path.GetFileN
ameWithoutExtension(file)) { MacrosFileName = file };
        itm.Click += new EventHandler(macroToolStripMen
uItem_Click);
        макросыToolStripMenuItem.DropDownItems.
Add(itm);
      }
    }

internal class MacrosToolStripMenuItem :
ToolStripMenuItem
{
public MacrosToolStripMenuItem(string FileName) :
base(FileName) { }
public string MacrosFileName { get; set; }
}
```

MacrosToolStripMenuItem — класс-наследник от ToolStripMenuItem отличающийся только свойством MacrosFileName

Для начала, создадим макрос, который просмотрит текст в textBox'e и найдет все e-mail адреса вида «vpupkin@mail.ru». В папке Macroses создаем файл SaveEmail.py, запускаем приложение — и смотрим, что в меню Макросы появился пункт SaveEmail.

Теперь собственно ключевой момент — выполнение IPy скрипта и доступ его к интерфейсу. Добавляем к проекту ссылку на сборку IronPython.dll. И создаем класс MacroRunner — выполняющий скрипт.

```
public class MacroRunner
  {
    public static Form CurrentForm;
```

```
    public string FileName { get; set; }

    public MacroRunner() { }

    public void Execute()
    {
      // собственно среда выполнения Python-скрипта
      IronPython.Hosting.PythonEngine pyEngine = new
IronPython.Hosting.PythonEngine();
      // важный момент - к среде выполнения подключаем
текущую выполняемую сборку, т.к.
      // в ней собственно и объявлена форма, к которой
необходимо получит доступ
      pyEngine.LoadAssembly(System.Reflection.Assembly.
GetExecutingAssembly());

      try
      {
        pyEngine.ExecuteFile(FileName);
      }
      catch (Exception exc)
      {
        MessageBox.Show(exc.Message);
      }
    }
  }
```

Ключевой момент — подключение к выполняющей среде IPy текущей сборки — для доступа к форме. Когда сборка подключена, в IPy скрипте появится возможность использовать классы пространства имен Notepad. Так же, через LoadAssebmly можно добавить и другие необходимые сборки — типа System.Windows.Forms — чтобы работать с формами. Класс готов, теперь модифицируем обработчик клика на пунктах подменю Макросы

```
protected void macroToolStripMenuItem_Click(object
sender, EventArgs e)
    {
      MacrosToolStripMenuItem item = sender as
MacrosToolStripMenuItem;

      MacroRunner runner = new MacroRunner() { FileName
= item.MacrosFileName };
      MacroRunner.CurrentForm = this;
      runner.Execute();
    }
```

Здесь следует отметить следующий момент — чтобы передать в IPy-скрипт форму, из которой собственно вызывается макрос — используется статическое поле CurrentForm. В скрипте форма будет доступна как Notepad.MacroRunner.CurrentForm. В идеале, скрипт, разумеется, не должен иметь полного доступа к интерфейсу формы — а должен пользоваться только предоставленным API — и ограничиваться только им. Но сейчас этим заморачиваться не будем — и просто сделаем textBox открытым (Modifier = Public). Ну и кроме текстового поля, разрешим скрипту доступ к пункту меню Сервис (Modifier = Public). Работа с формой закончена, собираем проект и открываем файл SaveEmail.py — теперь работаем только с макросами.

Итак, первый макрос — SaveEmail.py:

```
from Notepad import *
import re

text = MacroRunner.CurrentForm.textBox.Text
links = re.findall(«\w*@\w*\.\w{2,4}», text)
file = open(«emails.txt», «w»)
file.write(«\n».join(links))
file.close()
```

Т.к. сборка подключена к среде выполнения — то доступно пространство имен Notepad — в котором объявлены классы приложения. Как раз сейчас потребуется статический метод класса MacroRunner — чтобы получить доступ к активной форме (еще раз оговорюсь — что правильнее было бы предоставить не прямой доступ, а через класс-посредник — которые ограничит доступ определенным API). Ну а дальше все просто — получаем текст, регулярным выражением вытаскиваем email — и сохраняем их в файл в текущем каталоге.

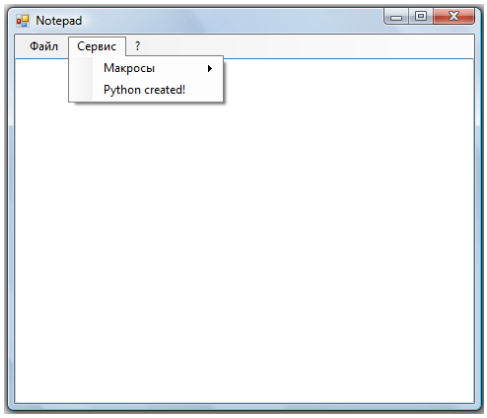
Можно запустить приложение, ввести произвольный текст, содежащий email — и убедиться, что после того, как макрос отработал — в папке с выполняемой программой появился файл emails.txt.

Теперь еще один пример, что может сделать макрос — чуть интереснее предыдущего. Итак, создаем в папке Macroses файл UIModifier.py. Как можно догадаться по названию — макрос будет изменять элементы интерфейса приложения. Конкретно — добавит новый пункт в меню Сервис. Для того, чтобы можно было работать с элементами управления WinForms необходимо в среде выполнения IPy подключить сборку System.Windows.Forms. Это можно сделать в момент запуска скрипта из приложения — добавить еще один вызов LoadAssembly. Но мы решили — никаких перекомпиляций, пусть IronPython обходится своими силами. Ну что ж, силы есть :). Чтобы подключить сборку используется метод AddReference класса clr.

```
from Notepad import *
main = MacroRunner.CurrentForm

import clr
clr.AddReference(«System.Windows.Forms»)
from System.Windows.Forms import *

def pyHello(s,e):
  MessageBox.Show(«Hello from IPy!»)

item = ToolStripMenuItem()
item.Name = «pybtn»
item.Text = «Python created!»
item.Click += pyHello

main.сервисToolStripMenuItem.DropDownItems.Add(item)
```

Все просто — получаем текущую форму, подключаем сборку System.Windows.Forms и импортируем из пространства имен System.Windows.Forms все — пригодится.
pyHello — простенький обработчик события — при щелчке на созданном пункте меню — будет выводится сообщение.

Запускаем приложение, выполняем макрос.
Смотрим пункт меню Сервис:



При щелчке на пункт меню «Python created!» появится стандартный MessageBox — собственно, чего и добивались.

LINQ to SQL и пространственные данные SQL Server

RollingStone

Начиная с версии 2008 (и пока что заканчивая ей) MS SQL Server имеет встроенную поддержку пространственных данных. Прекрасно! На данный момент времени уже существует несколько СУБД, предлагающих индексированное хранение пространственных данных. Наверное, самые популярные из них, это: «народная» MySql и PostGIS. Программируя на c#, естественно, в очень многих случаях, отдаёшь предпочтение продуктам и решениям Microsoft. Причины просты: полнее поддержка одних технологий другими, хорошая документация, более полная реализация, например провайдеров данных, и гораздо меньшая глючность. Я выбрал SQL Server. Заодно захотелось освоить LINQ в общем и LINQ to SQL в. частности.

Поначалу всё было хорошо. Для меня хороший старт сделала, обнаруженная на msdn, статья «LINQ to SQL: .NET Language-Integrated Query for Relational Data». Но я не сильно удивился, когда «всё хорошо» закончилось.

Для хранения геометрических данных в SQL Server были введены два дополнительных типа: geometry и geography. Первый используется для хранения геометрических объектов, описанных в декартовой системе координат, а второй — для геометрических объектов заданных географическими координатами (широта/долгота).

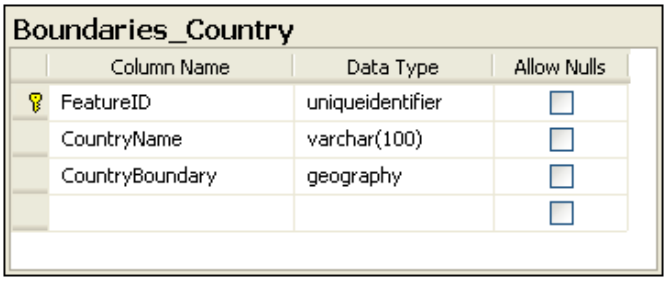
Такое разделение, по всей видимости, пришлось сделать из-за того, что пространственный индекс реализован в SQL Server на основе B-деревьев. При использовании этого индекса пространство шаблонно разбивается сеткой несколько раз и в «ячейки» этой сетки сохраняются ссылки на геометрические объекты. И оказалось невозможно строить универсальное разбиение и для прямоугольной системы координат и для эллипсоидальной. В MySql, например, выбран другой алгоритм индексирования, основанный на R-деревьях, работающий на совершенно другом принципе, и используется один тип данных. Какой способ индексирования лучше, а какой хуже —не очевидно, так что пока не понятно на кого ругаться и стоит ли.

Оказалось, что LINQ to SQL не понимает этих типов данных и работать с ними, а также со встроенными геометрическими функциями, отказывается. Хотя, наверное, правильнее сказать, что их не понимает провайдер. В любом случае, уверен, что поддерживаться эти данные будут, но сейчас такой поддержки нет.

Я не смог найти в интернет решения, обходящего эту проблему, поэтому пришлось изобрести его самому. Здесь нет никаких удивительных ходов, но есть детали, которые, думаю, будут интересны. Также в этой большой заметке, для вашего интереса, я чуть-чуть опишу работу с LINQ to SQL.

База данных

Для примера будем использовать следующую таблицу.



Для её создания использовался следующий скрипт:

```
USE ExampleDatabase;
GO
 --Create table
CREATE TABLE Boundaries_Country(
 FeatureID UNIQUEIDENTIFIER NOT NULL PRIMARY KEY,
 CountryName VARCHAR(100) NOT NULL UNIQUE,
 CountryBoundary GEOGRAPHY NOT NULL
)
 CREATE SPATIAL INDEX SpatialIndex
 ON Boundaries_Country (CountryBoundary);
GO
```

В строке 11 для поля CountryBoundary с типом данных geography, создаётся пространственный индекс с настройками по умолчанию.

Чтобы было с чем работать, я заполнил таблицу мультиполигонами стран мира, shape-файл для которых был найден на просторах интернет. Несколько стран не захотели конвертироваться — я не стал разбираться почему, было не важно, хотя за Россию, безусловно, обидно.

SQL Server имеет симпатичный встроенный просмотрщик (рисунок №1).

Начало работы с LINQ to SQL

Для работы с LINQ to SQL в проект нужно добавить

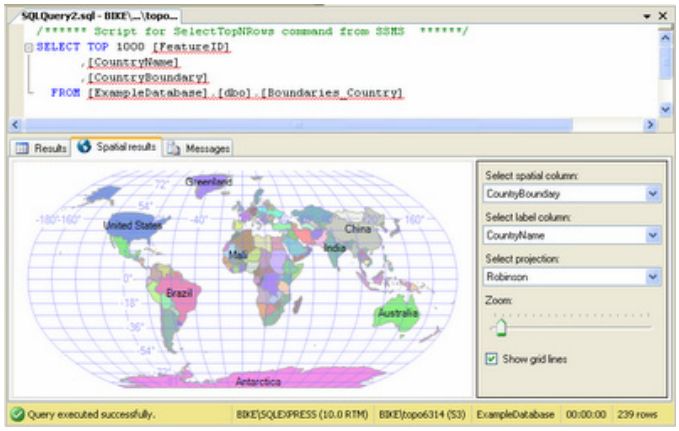


Рисунок №1. Просмотр гео-данных при работе с SQL Server

ссылки на две сборки: System.Data.Linq и Microsoft.SqlServer.Types. Если с первой библиотекой проблем нет (её можно найти на вкладке «.NET» формы «Add Reference» — добавления ссылки на используемую в проекте библиотеку), то вторую нужно будет поискать в директории «C:\Program Files\Microsoft SQL Server\100\SDK\Assemblies\». Для того, чтобы последняя сборка впредь отображалась во вкладке «.NET» формы добавления сборок, нужно её зарегистрировать один раз с помощью утилиты gacutil.

Первый шаг в использовании LINQ to SQL — это создание классов-отображений для таблиц базы данных.

На одну таблицу — один класс.

```
using System;
using System.Data.Linq.Mapping;
using Microsoft.SqlServer.Types;
namespace MyNamespace
{
 [Table()]
 public sealed class Boundaries_Country
 {
    [Column(AutoSync = AutoSync.OnInsert, DbType =
“uniqueidentifier”, IsPrimaryKey = true, IsDbGenerated =
true, UpdateCheck = UpdateCheck.Never)]
    public Guid FeatureID;
    [Column(DbType = “varchar(100)”, CanBeNull =
false)]
    public string CountryName;
    [Column(/*DbType = “geography”, */CanBeNull =
false)]
    public SqlGeography CountryBoundary;
 }
}
```

Над объявлением класса и над полями расставлены атрибуты. Например, в строке 7, атрибут Table указывает, что этот класс ассоциирован с таблицей в базе данных. Если имя класса совпадает с именем таблицы, то атрибут можно записывать так, как у меня, а если нет, то нужно будет указать дополнительное свойство Name: [Table(Name = «Boundaries_Country»)].

В строке 16, при описании атрибута для поля, содержащего пространственные данные, по идее, я должен указать тип данных geography, но поскольку поддержки этого типа данных ещё нет, то я его и не указываю.

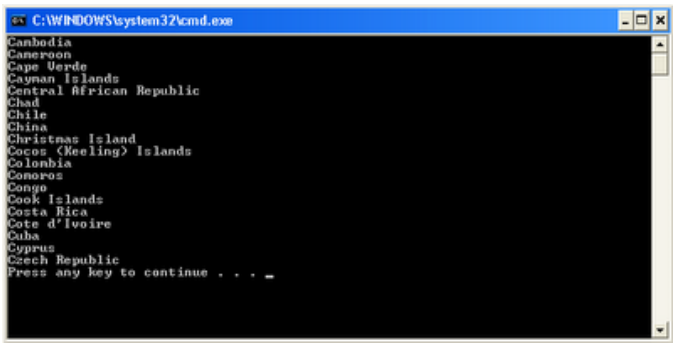
Ещё один класс нужен для создания контекста базы данных. Мы можем использовать уже имеющийся DataContext, но лучше сделать своего наследника для строгой типизации.

```
using System.Data.Linq;

namespace MyNamespace
{
 public class ExampleDatabase: DataContext
 {
    public Table<Boundaries_Country> BoundariesCountry;
    public ExampleDatabase(string connectionString)
     : base(connectionString)
    {

    }
 }
}
```

Пример, выгребем из базы данных всё, что есть, но так, чтобы название страны начиналось с буквы «C».

```
static void Main(string[] args)
{
  ExampleDatabase db = new ExampleDatabase(@»...»);

  var q = from item in db.BoundariesCountry
          where item.CountryName.StartsWith(“C”)
          select item;

  foreach (var item in q)
     Console.WriteLine(item.CountryName);
}
```

Получилось не так уж и много:



Один интересный момент. Если в режиме отладки остановить выполнение программы на строке 9 и просмотреть содержание переменной q, то мы увидим сформированный LINQ to SQL запрос.

LINQ to SQL: работа с пространственными данными

Рассмотрим запрос, выбирающий из базы данных страны, попавшие в заданный прямоугольник, и название которых начинается на букву «C».

Прямоугольник задан полигоном (WKT-представление): POLYGON ((40 -28, 40 30, 5 30, 5 -28, 40 -28)).

```
var q = from item in db.BoundariesCountry
        where item.CountryName.StartsWith("C") &&
                    item.CountryBoundary.
STIntersects(sqlEnvelope).Value
        select item;

foreach (var item in q)
 Console.WriteLine(item.CountryName);
```

Этот код скомпилируется, но работать не будет. Во время выполнения, на строке 5, когда от LINQ to SQL потребуется отправить запрос на сервер, будет выброшено исключение: «Method 'System.Data.SqlTypes.SqlBoolean STIntersects(Microsoft.SqlServer.Types.SqlGeography)' has no supported translation to SQL.».

Чтобы решить эту проблему мы будем использовать хранимые процедуры и table-valued функции, а пересылать геометрические объекты на сервер будем в хорошо понятном SQL Server'у бинарном формате WKB.

Хранимые процедуры

Для выборки геометрических фигур по критерию попадания в заданный прямоугольник для одной конкретной таблицы, достаточно создать следующую простую хранимую процедуру.

```
CREATE PROCEDURE [dbo].[sp_bbx_Boundaries_Country]
 @boundingBox varbinary(max)
AS
BEGIN
 SET NOCOUNT ON;
 SELECT *
    FROM dbo.Boundaries_Country
    WHERE GEOGRAPHY::STGeomFromWKB(@boundingBox,4326).
STIntersects(CountryBoundary) = 1;
 RETURN;
END
```

Входной параметр — это прямоугольник (заданный полигоном) в WKB-формате. В строке 8 он преобразуется статическим методом STGeomFromWKB в объект типа данных geography и уже на нём вызывается функция STIntersects, осуществляющая проверку на попадание конкретной границы в прямоугольник.

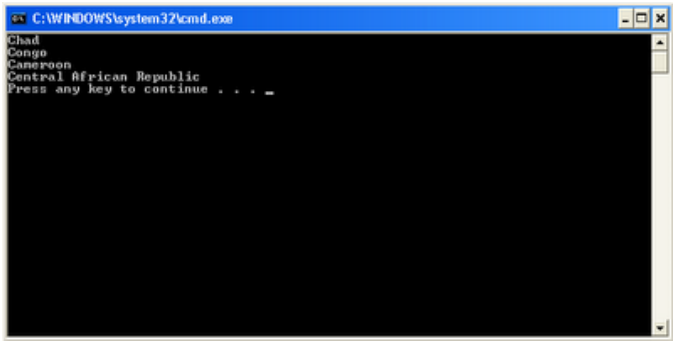
В программе, в классе, реализующем DataContext (у нас этот класс называется ExampleDatabase), опишем обёртку для вызова этой процедуры.

```
[Function()]
public ISingleResult<Boundaries_Country> sp_bbx_
Boundaries_Country(
[Parameter(DbType = "varbinary(max)")] byte[] bounding-
Box)
{
 IExecuteResult execResult = this.
ExecuteMethodCall(this, ((MethodInfo)
 (MethodInfo.GetCurrentMethod())), boundingBox);
 ISingleResult<Boundaries_Country> result =
((ISingleResult<Boundaries_Country>)execResult.
ReturnValue);
 return result;
}
```

Здесь, также как и для таблиц, описываются атрибуты для функции и параметров. В строке 4 вызывается хранимая процедура и результат сохраняется в execResult, затем, в строке 5, он преобразуется к требуемому типу данных и возвращается в основную программу. Пользуемся этой «радостью» следующим образом:

```
var q = from item in db.sp_bbx_Boundaries_Country(
 sqlEnvelope.STAsBinary().Buffer)
        where item.CountryName.StartsWith("C")
        select item;
foreach (var item in q)
 Console.WriteLine(item.CountryName);
```

Результат на консоль:



Замечание, если у вас, как у меня в рабочем проекте, несколько таблиц с географическими данными, то имеются следующие варианты.

1. Для каждой таблицы создать свою хранимую процедуру, а в программе для каждой хранимой процедуры свою функцию-обёртку. Можно упростить жизнь пользователю api, если написать «центральный» generic-метод в котором, по актуальному типу, используемому при вызове generic-метода, будут вызываться приватные метод-обёртки хранимых процедур и выполняться необходимые приведения типов.
2. Написать одну хранимую процедуру с использованием динамического sql. В программе нужно будет сделать один generic-метод, из которого также, как и в предыдущем варианте будут вызывать специализированные (по типу данных) методы-обёртки вокруг одной и той же процедуры (с наскока уйти от этого не удалось, воевать не стал).

Хранимые процедуры — это хорошо, но при использовании в LINQ to SQL, в описанной манере, у них есть один существенный недостаток: хранимые процедуры исполняются сразу и с сервера на клиент пересылаются все, попавшие в заданный регион, страны и уже потом над этим массивом выполняется дополнительная фильтрация. Т.е. трансляции в SQL всего LINQ-выражения не происходит. Для ухода от этой проблемы мы можем использовать inline-функции SQL Server'a.

Table-valued функции

Table-valued функция, извлекающая из таблицы базы данных записи по критерию попадания в заданный регион, может быть создана следующим образом.

```
CREATE FUNCTION [dbo].[f_bbx_Boundaries_Country]
(
 @boundingBox varbinary(max)
)
RETURNS TABLE
AS
RETURN
(
 SELECT *
    FROM dbo.Boundaries_Country
    WHERE GEOGRAPHY::STGeomFromWKB(
                    @boundingBox, 4326).
STIntersects(CountryBoundary) = 1
)
```

Т.е. по содержанию полностью аналогично хранимой процедуре, описанной раньше. Для функции также нужно будет создать свою обёртку.

```
[Function(IsComposable = true)]
public IQueryable<Boundaries_Country> f_bbx_Boundaries_
Country(
                [Parameter(DbType = "varbinary(max)")]
byte[] boundingBox)
{
 return this.CreateMethodCallQuery<Boundaries_
Country>(this,
               ((MethodInfo)(MethodInfo.
GetCurrentMethod())), boundingBox);
}
```

В атрибуте метода указываем свойство IsComposable, которое говорит, что сейчас мы будем запускать функцию на SQL Server, а не хранимую процедуру. Для вызова функции используется метод CreateMethodCallQuery. Смотрим пример:

```
var q = from item in db.f_bbx_Boundaries_Country(
                                          sqlEnvelope.
STAsBinary().Buffer)
        where item.CountryName.StartsWith("C")
        select item;
foreach (var item in q)
 Console.WriteLine(item.CountryName);
```

Результат тот же, что и при использовании хранимой процедуры.

И в отладке видим прекрасную картину (всё linq-выражение было транслировано в sql-запрос):

Первые шаги с Unity: DI/IoC & AOP

butaji

Если Вы когда-нибудь слышали такие слова, как IoC, DI, AoP, но не имеете четкого понимания этих терминов, надеюсь, эта статья поможет в них разобраться на примере работы с Microsoft Unity контейнером.

Inversion of Control (IoC) и Dependency Injection (DI)

За определением IoC обратимся к википедии:

Обращение контроля* (Inversion of Control, IoC) — важный принцип объектно-ориентированного программирования, который используется для уменьшения связности в компьютерных программах.

IoC также известен как Dependency Injection Principle. Приём Dependency Injection используется почти во всех framework'ах. Он применяется программистами использующими такие объектно-ориентированные языки программирования как Smalltalk, C++, Java или языки платформы .NET.

**Сразу же хотелось оговориться, что мне больше по душе не употреблять для IoC русскоязычного перевода, но всё же если меня заставят это сделать, что я употреблю его, как «инверсия управления»*

За определением DI так же обратимся к википедии:

Внедрение зависимости (англ. Dependency injection) обозначает процесс предоставления внешней зависимости программному компоненту и является специфичной формой «обращения контроля (англ. Inversion of control)», где изменение порядка связи является путём получения необходимой зависимости.

Классический пример для определения, на сколько крепко объекты вашего приложения увязаны друг с другом, является попытка скопировать класс из вашей предметной области в новое приложения и попробовать запустить его, если это не получится, то скопировать связанные классы. Обычно в результате данных манипуляций ваше старое приложение практически целиком копируется в новое.

Для разрешения проблем зависимостей одного объекта от другого обычно на объект, имеющий зависимость возлагают ответственность за создание/получение необходимого ему объекта (Хотелось бы к слову обратить Ваше внимание на такие шаблоны, как Factory, Registry).

Примеров реализации с приведением псевдо-кода IoC и DI великое множество, в связи с этим не очень хочется, чтобы статья разрасталась из-за них, а лучшего понимания, чем при практическом использовании, я врядли смогу вам предоставить.

Думаю, тут будет полезно привести список .NET DI/IoC контейнеров:

— Castle Windsor
— StructureMap
— Spring.NET
— Autofac
— Unity
— Puzzle.NFactory
— Ninject
— S2Container.NET
— PicoContainer.NET
— LinFu

Aspect-oriented programming (AoP)

Признаю, я дотошный почитатель википедии:

Аспектно-ориентированное программирование (АОП) — парадигма программирования, основанная на идее разделения функциональности, особенно сквозной функциональности, для улучшения разбиения программы на модули.

Хотелось бы добавить, что Аспектно-ориентированное программирование (АОП) так же призвано прийти на помощь ООП для того, чтобы сократить количество написанного в приложениях кода.

Чаще всего использование АОП сводится к осуществлению повседневных рутинных задач, таких как логгирование, кэширование, валидация, так же управление доступом. Можно возразить и сказать, что можно решить эту проблему можно при помощи обволакивания ваших бизнес-объектов шаблона Decorator (рус.), но дело в том, что при этом возникает необходимость изменения кода потребителя, а АОП позволяет сделать это легко и непринужденно.

Известные framework'и:

— PostSharp
— Aspect.NET
— LOOM.NET
— Puzzle.NAspect
— AspectDNG
— Aspect#
— Encase
— Compose*
— Seasar.NET
— DotSpect (.SPECT)
— Как часть функциональности The Spring.NET Framework

О Microsoft Enterprise Library

Как известно, в октябре 2008 года Microsoft обновили версию Enterprise Library до 4.1, соотвественно одну из её составляющих Unity Application Block до версии 1.2. Разработкой Microsoft Enterprise Library занимается группа patterns & practices. Microsoft Enterprise Library представляет собой набор функциональных блоков, призваных помочь разработчикам в выполнении обычных рутинных задач. Функциональные блоки — что-то типа руководства, предоставляемый в их составе исходный код можно использовать «как есть», расширять или изменять разработчикам в соответствии с их нуждами. Microsoft Enterprise Library состоит из следующих блоков:

— Caching Application Block. Разработчики могут использовать этот блок для реализации кэширования в своих приложениях. Поддерживает так же подключаемые кэш провайдеры.
— Cryptography Application Block. Разработчики могут использовать этот блок для реализации хеширования и симметричного шифрования в приложениях.
— Data Access Application Block. Разработчики могут использовать этот блок для реализации стандартных действий с базой данных в своих приложениях.
— Exception Handling Application Block.
Разработчики и редакторы правил могут использовать этот блок для реализации последовательной стратегии обработки исключений, которые происходят на всех архитектурных слоях корпоративных приложений.
— Logging Application Block. Разработчики могут использовать этот блок для реализации стандартных функций логирования.
— Policy Injection Application Block. Разработчики могут использовать этот блок для реализации политик перехвата для того, чтобы насытить приложение такими возможностями, как логирование, кеширование, обработка исключений и валидация на уровне всего приложения.
— Security Application Block. Разработчики могут использовать этот блок для реализации механизма авторизации и безопасности приложений.
— Unity Application Block. Разработчики могут использовать этот блок, как легковесный, расширяемый dependency injection контейнер с поддержкой внесения зависимостей в конструктор, свойство и вызов методов, таким же образом, как и реализация перехвата, через расширения.
— Validation Application Block. Разработчики могут использовать этот блок для реализации правил валидации для бизнес-объетов, используемых в приложении.

О Unity Application Block

Легковесный, расширяемый dependency injection контейнер с поддержкой внесения зависимостей в конструктор, свойство и вызов методов, таким же образом, как и реализация перехвата, через расширения. Вы можете использовать его вместе с Enterprise Library (EntLib) для генерации объектов, как ваших собсвенных, так и Enterprise Library. Однако, Unity Application Block отличается от остальных функциональных блоков EntLib в нескольких ключевых моментах:

— Вы можете использовать Unity Application Block как автономный механизм внесения зависимостей, не нуждаясь в установленной EntLib.
— Unity Application Block может быть сконфигурирован как с помощью специализированных файлов конфигурации, так и в run-time режиме с помощью вашего кода.
— Unity Application Block независим от ядра EntLib, ровно как от системы конфигурации EntLib. Он содержит свой собственный механизм для конфигурирования, хотя, при желании вы можете использовать механизм конфигурации EntLib.

Долгожданный In Action. Требования

Для работы приведенного ниже примера нам понадобятся:

1. Установленный .NET Framework 3.5 SP1
2. Ваша любимая IDE (Notepad..Visual Studio 2008 SP1)
3. Установленный Unity
4. 15-20 мин времени и немного интереса

Процесс

Для начала, создадим ConsoleApplication под названием UnityInAction1.

Далее добавим ссылки на следующие сборки:

— Microsoft.Pratices.ObjectBulder2
— Microsoft.Pratices.Unity
— Microsoft.Pratices.Unity.Interception.

Создадим в нашем приложении интерфейс ILogger:

```
namespace UnityInAction1
{
    interface ILogger
    {
        void Write(string msg);
    }
}
```

Далее имплементируем его в классе Logger:

```
using System;

namespace UnityInAction1
{
    class Logger : ILogger
    {
        #region ILogger Members

        public void Write(string msg)
        {
```

```
            Console.WriteLine();
            Console.WriteLine(«*** In logger ***»);
            Console.WriteLine(String.Format(“message
{0}”, msg));
        }

        #endregion
    }
}
```

Так бы выглядело наше обычное приложение:

```
using System;

namespace UnityInAction1
{
    class Program
    {
        static void Main(string[] args)
        {
            ILogger logger = new Logger();
            logger.Write(«Моё сообщение»);
            Console.ReadKey();
        }
    }
}
```

Однако, с точки зрения подхода IoC/DI нас это не совсем устраивает, в связи с этим мы хотим поместить наш Logger в контейнер (к примеру для того, чтобы заменить [причём заменить нужно будет только в месте конфигурирования контейнера, а не по всему покрытию кода, как это обычно бывает] его на заглушку при проведении тестирования):

```
using System;
using Microsoft.Practices.Unity;

namespace UnityInAction1
{
    class Program
    {
        static void Main(string[] args)
        {
            IUnityContainer container = new
UnityContainer();
            container.RegisterType<ILogger, Logger>(new
ContainerControlledLifetimeManager());

            var logger = container.Resolve<ILogger>();
            logger.Write(«Моё сообщение»);
            Console.ReadKey();
        }
    }
}
```

Отлично, думаю данного примера вполне достаточно для старта.

Теперь добавим требование замера времени выполнения метода ILogger.Write() для всех имплементаций этого интерфейса, понятно, что с помощью ООП реализация данного условия превращается в затруднее, в связи с этим посмотрим на данный пример с точки зрения АОП. Для начала немного переопределим описание ILogger с помощью введения нового атрибута .NET:

```
namespace UnityInAction1
{
    interface ILogger
    {
        [Stopwatch]
        void Write(string msg);
    }
}
```

Описание атрибута Stopwatch:

```
using System;
using Microsoft.Practices.Unity.InterceptionExtension;

namespace UnityInAction1
{
    class Stopwatch : HandlerAttribute
    {
        public override ICallHandler
CreateHandler(Microsoft.Practices.Unity.IUnityContainer
container)
        {
            return new StopwatchCallHandler();
        }
    }

    public class StopwatchCallHandler : ICallHandler
    {
        public int Order {get; set;}

        public StopwatchCallHandler() : this(0) {}

        public StopwatchCallHandler(int order)
        {
            Order = order;
        }

        #region ICallHandler Members

        public IMethodReturn Invoke(IMethodInvocation
input, GetNextHandlerDelegate getNext)
        {
            System.Diagnostics.Stopwatch sw = new
System.Diagnostics.Stopwatch();
            sw.Start();

            var result = getNext().Invoke(input, get-
Next);

            sw.Stop();

            Console.WriteLine();
            Console.WriteLine(String.Format(“Прошло
времени {0} мс», sw.ElapsedMilliseconds));

            return result;
        }
```

```
        #endregion
    }
}
```

Ну и добавим расширение для нашего контейнера:

```
using System;
using Microsoft.Practices.Unity;
using Microsoft.Practices.Unity.InterceptionExtension;

namespace UnityInAction1
{
    class Program
    {
        static void Main(string[] args)
        {
            IUnityContainer container = new
UnityContainer();
            container.RegisterType<ILogger, Logger>(new
ContainerControlledLifetimeManager());
            container.AddNewExtension<Interception>();
            container.Configure<Interception>()
                .SetInterceptorFor<ILogger>(new
TransparentProxyInterceptor());

            var logger = container.Resolve<ILogger>();
            logger.Write(«Моё сообщение»);
            Console.ReadKey();
        }
    }
}
```

Теперь каждая имплементация ILogger.Write() будет выводить замеры времени выполнения этого метода, разве это не отлично?

Материалы

Написание это статьи было инициировано просмотром небольшого скринкаста от David Hayden. Примеры кода с небольшими поправками позаимствованы из него.

При написании статьи, в основном, я руководствовался базовыми знаниями, почерпнутыми когда-то из замечательной книги “Applying Domain-Driven Design and Patterns” by Jimmy Nilsson.
Так же замечательной кладезью знаний Википедия.

В завершение

Надеюсь в данной статье дано хорошее начало для ваших изысканий в области качественного кода.

Microsoft Object Builder

acerv

Столкнувшись сегодня в ленте на описание Java-фреймворков Spring и Tapestry, решил просмотреть Хабру и найти ценителей «конкурирующих» фреймворков от Microsoft – в частности Composite UI Application Block (CAB) и Unity. К моему удивлению, ничего не нашел. Увидев в комментах в статье про Java-фреймворки просьбу описать механизм инъекций зависимостей, решил начать свой цикл статей про .Net-фреймворки именно с разъяснения вопросов IoC. Итак, встречайте – краеугольный камень (в прошлом) замечательного движка CAB – Microsoft Object Builder.

Что это такое и где используется

Object Builder (далее OB) – это механизм неявной автоматической композиции множества зависимых объектов в связанный граф. OB был предоставлен компанией Microsoft в мае 2005 года через свою кузницу «сложных проектов» с открытыми кодами Patterns and Practices в тандеме с каркасом десктоп-приложений – CAB (или SmartClient). Набор основополагающих классов CAB использует OB как механизм инъекции зависимостей при создании пользователем объектов движка (пользовательских контролов, сервисов, мест расширения, и т.п.).

Как поставить себе

Скачать CAB отсюда и поставить. Инсталлятор добавит в GAC три сборки, одна из них OB.

Из чего состоит Object Builder и как он работает

По сути, OB это агрегация нескольких паттернов программирования. Вкратце опишу структуру.
— **Builder** – представляет из себя фасад, точку входа в механизм построителя объектов. Предоставляет пользователю методы построения (`BuildUp`), метода разрушения (`TearDown`). **Builder** содержит в себе цепочку стратегий (**Strategy**), каждая из которых является атомарной частью построителя. К тому же, **Builder** обладает возможностью вызвать процесс построения объекта нужным пользователю методом – через потребление т.н. политик (**Policy**) построения. Когда пользователь хочет построить объект, он вызывает `BuildUp` и передает в метод тип создаваемого объекта. На выходе получает инстанс этого типа.
— **Strategy** – частичка, из которой состоит OB. Стратегия имплементирует какой-то способ построения объекта. Во время своей работы, **Builder** идет по цепочке и применяет каждую стратегию из нее к создаваемому объекту – тем самым реализуя связи между зависимыми объектами. Пользователь может создать свою стратегию и кастомизировать процесс создания объектов.
— **Policy** – политика построения объектов. Каждая стратегия может обладать какой-то политикой. Политика предоставляет стратегии контекст (т.е. объектное окружение) в момент создания объектов. Политика может служить для связи стратегий друг с другом. Достаточно сложная штука, но, с высоты прожитых лет, могу сказать, что Microsoft сделало функционал OB достаточно богатым и (обычно) создание новых политик построения не требуется – достаточно встроенных.
— **Locator** и **LifeTimeContainer** – для того, чтобы знать, откуда брать уже построенные объекты зависимостей, OB требуется контейнер уже созданных объектов. **Locator** – это специальный класс, который может найти в **LifeTimeContainer** объект и представить его **Builder** для вставки зависимости через WeakReference. Соответственно, **LifeTimeContainer** управляет жизненным циклом объектов (т.е. он знает, когда объекты надо удалить из памяти, а когда не надо).
— **Stage** – в OB есть жизнный цикл построения объекта. Это несколько этапов:

- Перед созданием (PreCreation)
- Создание (Creation)
- Инициализация (Initialization)
- Пост-инициализация (PostInitialization)
- Каждая стратегия OB работает на каком-то этапе.

Душа требует примеров! Для примера, создадим объект – микросервис `IProgramService`:

```
interface IProgramService
{
  void DoWork();
}
```

А вот его реализация:

```
class ProgramService : IProgramService
{
  #region IProgramService Members

  public void DoWork()
  {
    Console.WriteLine(string.Format("[{0}] Program
Service: DoWork()", _guid));
  }

  #endregion

  private Guid _guid = Guid.NewGuid();
  public Guid Guid
  {
```

```
    get { return _guid; }
  }
}
```

Тогда код создания будет такой:

```
Builder builder = new Builder();
Locator locator = new Locator();
IProgramService service = builder.BuildUp<ProgramServic
e>(locator, null, null);

service.DoWork();
```

Вот так просто. Перейдем к рассмотрению стратегий вставки зависимостей.

Применение

Если посмотреть на код создания билдера, можно увидеть, как он добавляет в себя жизненно-необходимые стратегии:

```
public Builder(IBuilderConfigurator<BuilderStage>
configurator)
{
  Strategies.AddNew<TypeMappingStrategy>(BuilderStage.
PreCreation);
  Strategies.AddNew<SingletonStrategy>(BuilderStage.
PreCreation);
  Strategies.AddNew<ConstructorReflectionStrategy>(Build
erStage.PreCreation);
  Strategies.AddNew<PropertyReflectionStrategy>(BuilderS
tage.PreCreation);
  Strategies.AddNew<MethodReflectionStrategy>(BuilderSta
ge.PreCreation);
  Strategies.AddNew<CreationStrategy>(BuilderStage.
Creation);
  Strategies.AddNew<PropertySetterStrategy>(BuilderSta
ge.Initialization);
  Strategies.AddNew<MethodExecutionStrategy>(BuilderSta
ge.Initialization);
  Strategies.AddNew<BuilderAwareStrategy>(BuilderStage.
PostInitialization);

  Policies.SetDefault<ICreationPolicy>(new
DefaultCreationPolicy());

  if (configurator != null)
    configurator.ApplyConfiguration(this);
}
```

Разберемся, какими методами можно создавать связи между зависимостями и как работают эти стратегии.

TypeMappingStrategy (PreCreation) – Политика соответствия типов. Например, в прошлом примере мы явно указали, какой тип нам создавать. Но что, если мы не хотим каждый раз указывать, какой класс реализует интерфейс IProgramService? Что, если мы хотим создать IProgramService, а кто будет конкретным типом, знать не хотим – пусть об этом голова болит у программиста Васи? Воспользуемся политикой создания ITypeMappingPolicy. Эта политика укажет локатору соответствие между интерфейсом и конкретным типом:

```
class TypeMappingPolicy : ITypeMappingPolicy
{
  #region ITypeMappingPolicy Members

  public DependencyResolutionLocatorKey Map(DependencyR
esolutionLocatorKey incomingTypeIDPair)
  {
    return new DependencyResolutionLocatorKey(typeof(Pr
ogramService), null);
  }

  #endregion
}
```

Регистрация политики происходит таким способом:

```
builder.Policies.Set<ITypeMappingPolicy>(new
TypeMappingPolicy(), typeof(IProgramService), null);
IProgramService service = builder.BuildUp<IProgramServi
ce>(locator, null, null);
```

SingletonStrategy (PreCreation) – Стратегия создания объекта как синглтона. Это значит, что создаваемый объект для указанного типа создастся всего один раз. Все последующие попытки создания такого типа будут замещаться подстановкой уже существующего объекта. Для этого нужно явно пометить создаваемый тип специальным ключем, т.н.

```
DependencyResolutionLocatorKey:
IProgramService service = builder.BuildUp<ProgramServic
e>(locator, null, null);
DependencyResolutionLocatorKey key = new DependencyReso
lutionLocatorKey(typeof(ProgramService), null);
locator.Add(key, service);

IProgramService service2 = builder.BuildUp<ProgramServi
ce>(locator, null, null);
```

В этом случае service и service2 указывают на один и тот же объект.

ConstructorReflectionStrategy (PreCreation) – стратегия создания объекта через конструктор. Название говорит само за себя – стратегия выберет конструктор объекта, помеченный атрибутом [InjectionConstructor] и для каждого помеченного атрибутом [Dependency] параметра найдет его через локатор. После чего вызовет конструктор, передав туда отрезолвленные параметры.

```
[InjectionConstructor]
public ServiceConsumer([Dependency]IProgramService
service)
{
  _service = service;
}
```

Существует несколько способов вставки параме-

тров. Через свойства атрибута Dependency можно указать создаваемый тип, так же определить поведение, если разыскиваемый тип не найден. Например, создать новый (CreateNew), или бросить исключение:

```
[InjectionConstructor]
public ServiceConsumer(
  [Dependency(CreateType=typeof(ProgramService), NotPre
sentBehavior=NotPresentBehavior.Throw)]IProgramService
service)
{
  _service = service;
}
```

Само собой разумеется, если мы создаем один объект, у которого есть зависимость на другой, то другой объект должен резолвится в локаторе:

```
IProgramService service = builder.BuildUp<ProgramServic
e>(locator, null, null);

DependencyResolutionLocatorKey key = new DependencyReso
lutionLocatorKey(typeof(IProgramService), null);
locator.Add(key, service);

ServiceConsumer consumer = builder.BuildUp<ServiceConsu
mer>(locator, null, null);
```

CreationStrategy (Creation) – это обычная стратегия создания. В самом первом примере выше объект строится как раз через стратегию создания. Как она работает – сначала стратегия ищет пользовательскую политику построения ICreationPolicy, и, если не находит политику, пользуется «родной». Родная политика создания инструктирует билдер определить первый конструктор объекта (через рефлексию) и запустить цепочку построения на каждую зависимость (параметр) в конструкторе. Затем исполнить его через Activator. Особенностью этой стратегии служит то, что в нее можно передать уже созданный объект. Тогда стратегия делать ничего не будет :) Просто передаст управление другим стратегиям, у которых Stage следующий за Creation. Еще раз последнее, но другими словми — мы можем создать объект через new(), передать его билдеру. В этом случае билдер не будет создавать объект, а просто вставит в него зависимости.

PropertyReflectionStrategy + PropertySetterStrategy (Initialization). Стратегии (на самом деле, семантически, это одна стратегия), позволяющие вставлять зависимости в свойства объекта. Вкратце, работа такая – PropertyReflectionStrategy определяет свойства объекта, помеченные атрибутом [Dependency], узнает типы объектов, которые туда надо вставить, создает политику IPropertySetterPolicy, где перечисляет эту информацию. Далее, идя по цепочке, начинает работать PropertySetterStrategy, которая находит политику, из нее определяет свойства, нуждающиеся в вставке объектов, резолвит объекты и устанавливает их в нужные места. Фишка разделения стратегий здесь для того, чтобы пользователь смог использовать свои IPropertySetterPolicy.

```
[Dependency(CreateType=typeof(ProgramService),
  NotPresentBehavior=NotPresentBehavior.CreateNew)]
public IProgramService ProgramService
{
  set { _service = value; }
  get { return _service; }
}
```

Этот код можно переписать проще:

```
[CreateNew]
public ProgramService ProgramService2
{
  set { _service = value; }
}
```

MethodReflectionStrategy + MethodExecutionStrategy (Initialization). По сути дела, эти две стратегии изоморфны стратегиям Property-зависимостей. Только, вместо установки свойства, происходит вызов метода, помеченного атрибутом [InjectionMethod].

```
[InjectionMethod]
public void DoWork([Dependency]IProgramService service)
{
  service.DoWork();
}
```

BuilderAwareStrategy (PostInitialization). В OB есть такой специальный интерфейс IBuilderAware, в котором есть два метода – OnBuiltUp, OnTearingDown. Можно реализовать их в своем классе, тогда при построении (или уничтожении) объекта, стратегия их вызовет. Очень полезно, если нужно запустить что-то сразу же после создания объекта (когда все зависимости будут на своем месте).

```
#region IBuilderAware Members
public void OnBuiltUp(string id)
{
  Console.WriteLine("Built Up!");
}
public void OnTearingDown()
{
  Console.WriteLine("Tearing down!");
}
#endregion
```

Я перечислил все «родные» стратегии OB. В движках разработчики добавляют свои стратегии, чтобы можно было связывать специализированные аспекты фреймворка. К сожалению, полностью разобрать все аспекты OB — так, скажем, не затронуты именованые инстансы, не развернута полностью тема политик и т.п. Читатель, думаю, если захочет, разберется сам.

Как это использовать в реальных приложениях, плюсы и минусы

Ответ прост – как инфраструктурный аспект. Если вы хотите внедрить у себя в приложении архитек-

туру, подразумевающую инъекцию зависимостей, то OB – то, что доктор прописал. Сам OB изначально не подразумевался для прямого использования. В движке CAB его методы оборачиваются специальными коллекциями – фасадами OB, что очень облегчает работу программиста.

Минусы: Самым, пожалуй, большой минус решения – OB тяжелый. Он загромождает код атрибутами (без них не обойтись!), а его метод BuildUp, сами понимаете, не из самых быстрых. Решить эту проблему можно оптимизацией связей в графе классов – собственно, минимизировать количество зависимостей при максимальной изолированности кода.

Плюсы: Низкая связность кода, объектов и все отсюда вытекающие. Это и простота юнит-тестов, и автономная (когда ручками ничего не надо выставлять) микросервисная архитектура и тому подобное.

Будущее Object Builder и заключение.

Сейчас OB уже устарел. Да, простите, я писал статью по устаревшей технологии, но, мне кажется, полезно начать с основ. К тому же, будет сложно говорить о Microsoft Object Builder 2, не понимая, чем же был сам Microsoft Object Builder. Microsoft замещает движок CAB на более современный – Unity. Unity Application Block – это еще более гибкая платформа с долгожданной поддержкой WPF. Собственно, в следующих статьях я расскажу, что такое CAB и постепенно перейду на Unity. Кому не терпится узнать, что это такое, вот ссылки:
CAB: msdn.microsoft.com/en-us/library/aa480450.aspx
CAB: www.codeplex.com/smartclient
Unity: msdn.microsoft.com/en-us/library/cc468366.aspx
Unity: www.codeplex.com/unity
Отличие CAB от Unity: www.codeplex.com/CompositeWPF/Release/ProjectReleases.aspx?ReleaseId=16941

Use Case Driven Development и Composite UI Application Block

acerv

В этой статье приведено описание фреймворка для разработки десктоп-приложений на платформе Microsoft .NET — Composite UI Application Block. Разработчики Java, если им интересно, могут сравнить этот фреймворк с Java Spring. В статье так же рассказывается о use case driven разработке — что это такое и как реализуется при помощи фреймворка.

Пара слов про описания фреймворков

Я буду сейчас рассказывать про системы, имея в виду front-end десктоп-приложений WinForms .NET 2.0. Вообще говоря, описывать фреймворки — очень тяжело. Основная загвоздка в том, что все сразу не опишешь, а рассказывать про прям все, пусть даже в нескольких статьях — это во-первых, очень трудоемко, во-вторых, легче махнуть рукой и дать ссылку на документацию, в которой уже все разжевано. В общем, чтобы не попасть в эту ловушку, я соберусь духом и расскажу все так, как будто бы вы только что пришли работать в команду и совершенно ничего не знаете о фреймворке. Поехали.

Use Case Driven Development

Из Википедии: прецедент (англ. Use Case, а также: вариант использования, сценарий использования) — спецификация последовательности действий (варианты последовательности), которые может осуществлять система, подсистема или класс, взаимодействуя с внешними акторами (англ. Actors).

Проблема

Разрабатываемую систему можно разложить на доменную модель и некоторый набор связанных друг с другом юс-кейсов. Это работа аналитиков. Работа программиста (команды программистов) заключается в том, чтобы перенести аналитическую информацию в код, запрограммировав систему так, чтобы она своей логикой соответствовала модели прецедентов. Основная проблема, которая возникает при разработке — воплощение юс-кейсов в виде какой-то объектной модели. Самое очевидное решение, предлагаемое Microsoft — взаимодействие объектов UserControl с бизнес-логикой, имплементированной в самом классе пользовательского контрола. Т.е., другими словами, программист помещает логику прецедента в самом пользовательском контроле через дизайнер Visual Studio — пишет в него методы, создает в нем другие пользовательские контролы и тому подобное. Monkey-style, позволяющий получить в короткие сроки работающую систему, фронт-энд которой достаточно тяжело поддерживать и код которого, по своей сути, является одноразовым. Это хорошо, если систему надо сдать быстро и забыть про нее, и это очень плохо, если система разрабатывается с упором на дальнейшее развитие, а аналитическое окружение предполагает частую смену требований (обычная ситуация в сфере госзаказов).

Решение

Использовать какой-то фреймворк, базис которой предлагает повторное использование кода, дает возможность покрыть бизнес-логику фронт-энда тестами и обеспечивает loose-coupling сущностей (для быстрого удовлетворения новых или измененных требований). Таким фреймворком является Composite UI Aplication Block (далее я буду называть его движком или CAB).

Composite UI Application Block

Суть движка — предложить пользователю набор объектов (вышеупомянутый базис), основным из которых является объектное представление юс-кейса. Тогда, при наличии проработанной аналитиками модели прецендентов, программистам достаточно преобразовать юс-кейсы в набор взаимодействующих друг с другом объектов (преобразование почти всегда один к одному). Еще раз, другими словами — основной фишкой CAB является возможность работать с юс-кейсами, как с объектами, при этом дизайн фронт-энда подразумевает разделение логики юс-кейсов от view-логики. Разработчик использует модель прецедентов для разработки системы. Меняется модель прецедентов — меняется система. Поэтому разработка use case driven :) (Простите, что я не привожу тут диаграм модели прецедентов — предполагается, что читатель знает, что такое UML)

Помимо этого, CAB предлагает:
— Модульность. Модули подгружаются динамически и имеют возможность взаимодействовать друг с другом
— Паттерны Event Broker и Command, для loose-coupling коммуникаций объектов друг с другом
— Паттерн Model-View-Presenter (разновидность MVC) для разделения и тестирования view-логики
— Микросервисную архитектуру, для подключения всего, что можно — сервисов аутентификации, поиска модулей, бизнес-сервисов (работа с БД, с доменной моделью) и т.п.

Базис системы

CAB предлагает свою семантику, тесно коррелирующую с семантикой .NET WinForms, только уровнем выше. Я перечислю каждый термин и расскажу, что он означает и как используется.

WorkItem (воркайтем) — это и есть объект юс-кейса. Так же это узел логических связок — для обеспечения связи между объектами, воркайтем является контейнером других юс-кейсов, сервисов, пользовательских контролов, объектов состояния, команд, мест расширения, рабочих зон (о них ниже). Воркайтем знает все аспекты своего юс-кейса и предоставляет в него точку доступа. У воркайтема есть ссылки на RootWorkItem, на Parent (родительский воркайтем) и на всех своих детей (sub-use cases).

RootWorkItem (главный воркайтем) — это точка входа в юс-кейс нулевого уровня. Самый изначальный юс-кейс — контейнер всех других юс-кейсов.

Service (сервис, или микросервис). Это объект, который обладает public интерфейсом. Сервис реализует какую-то логику системы (например, получения доменной модели, или аутентификации), в большинстве случаев является зависимостью для view-логики и может передаваться между воркайтемами.

Module (модуль) — набор логически связанных воркайтемов, т.н. unit of deployment. Модуль подключается к системе динамически, имеет точку входа и исполняется после добавления. Например, запускает свои воркайтемы (юс-кейсы), или расширяет контролы элементами управления, позволяющими запускать их. Физически — сборка .NET.

ProfileCatalog — каталог конфигурации. Приложение может быть сконфигурированно с целью загрузки одного определенного набора модулей в одном случае и другого набора в другом. Например, администратору — модули, в которых хранятся административные юс-кейсы, а пользователю с правами на просмотр — модуль поиска объектов и модуль их карточек. В базовой версии каталог представляет собой xml-файл, но может быть переопределен пользовательским сервисом IModuleEnumerator — например, чтобы подгружать конфигурацию приложения из базы данных или через ремоутинг от приложения-аутентификатора.

ModuleLoader — загрузчик модулей. Класс, который получает каталог, находит модули, динамически их подгружает и исполняет. Как и почти все, может быть перегружен другим загрузчиком, достаточно переопределить нужный интерфейс.

ModuleInit — точка входа в модуль (класс). В нем регистрируются все нужные модулю и его юс-кейсам сервисы и резолвятся объекты view-logic. То, что исполняет ModuleLoader и что определяет сборку .NET как модуль системы.

Shell Application — приложение системы. Ответственно за инициализацию RootWorkItem. В нем определяются инфрастуктурные объекты движка, которые можно переопределять и регистрируются основные view-logic объекты (шелл-форма, например)

Shell — шелл-форма. Представляет внешний вид системы. Главная форма приложения. В нее будут вставлятся рабочие зоны юс-кейсов других модулей (или же она сама изначально ими владеет).

Workspace — рабочая зона приложения. Хостер для view-элементов системы (так называемых SmartPart-ов). Любой контрол может быть рабочей зоной — при наличии имплементации интерфейса IWorkspace. В CAB изначально есть четыре рабочих зоны — о них в других частях статей:)

UIExtensionSite — место расширения пользовательского интерфейса. По своей сути, это контролы, которые содержат элементы управления — тулбары, менюшки, статусбары и т.п. RootWorkItem регистрирует их в себе на старте приложения, а модули расширяют кнопками и другими элементами — которые, в свою очередь, выполняют команды. Для того, чтобы обеспечить возможность регистрировать любые контролы как места расширения и расширять их чем угодно, в CAB поднята довольно нехилая инфраструктура — тоже тема для отдельной статьи.

Command — паттерн «Команда». Команды создаются в виде объектов и хранятся в WorkItem-ах. Это запускаемые команды в виде объектов. Так же существуют декларативные команды, которые, в свою очередь, являются методами в классах, помеченные специальными аттрибутами [CommandHandler].

EventPublication и EventSubscription — публикация и подписка на события. «Щупальца» паттерна Event Broker. Класс, который бросает эвент, помечает его как публикацию со строковым ключем. Класс, который хочет обработать этот эвент, помечает нужный метод «подпиской» с тем же ключем и типом аргументов. В итоге, обеспечивается запуск-обработка событий между объектами, которые друг о друге совсем ничего не знают (loose-coupling). Тоже достаточно обширная тема, достойная отдельной статьи.

Model — модель. То, что в MVC является моделью:) Ну, или доменные объекты (жучок, червячок, пользователь, документ, и т.п.).

View — представление. Юзер-контрол с кнопками, гридами и т.п. Класс, который инкапсулирует в себе исключительно UI-логику (обработку кнопок, перемещений строки выделения, установку списка объектов с listbox и т.д.)

Presenter — презентер. Частный случай Controller в паттерне MVC. Инкапсуляция клиентской бизнес-логики. Знает о юс-кейсе (воркайтеме), в котором находится. В него инъектятся объекты сервисов. В него перебрасывает свои эвенты View. Опять же, ме-

тодика создания пользовательских контролов, достойная тема для отдельной статьи.

SmartPart — это тройка Model-View-Presenter. Объект «части» системы. Его показывают рабочие зоны (пример чуть ниже в коде).

Microsoft Object Builder — композиционный механизм движка. Фабрика сущностей и инъектор зависимостей, который позволяет фабрике автоматически создать и проинициализировать зависимости создаваемой сущности. Про то, как он работает, я достаточно подробно написал вот здесь.

Примеры и заключение

Вы знаете, я закончил описание базиса и перенес написанное в черновики в Хабру. Перечитал и стало страшно, потому что поставил себя на ваше место и понял, что сказанное, во-первых, мало похоже на интересный триллер, а во-вторых, уже очень большое по объему. Поэтому давайте я быстро закруглюсь, приведу один пример и перейду к заключению.

Любой десктоп-клиент можно сделать в CAB. Возьмем, для примера, некий абстрактный Microsoft Word2, рассмотрим юс-кейс… хм, ну, чтобы уж очень по-быстрому, сохранения файла (через диалог с пользователем).

Для этого нам понадобятся два сервиса — читатель контента с рабочей области ворда, и сохраняльщик файлов:

```
public interface IFileSaver
{
  void SaveFile(byte[] content, string fileName);
}

public interface IContentReader
{
  byte[] GetContent();
}
```

IFileSaver сохраняет файл, IContentReader считывает байт-массив информации из ворда (имеется в виду, что у нас уже есть конкретные типы для этих интерфейсов в воркайтеме выше уровнем).

Понадобится SmartPart диалога сохранения. Как мы помним, это тройка MVP (а физически, интерфейс для вью, сам вью, и презентер):

```
public interface IFileSaveDialog
{
  event EventHandler OkClicked;
  string GetFileName();
}

[SmartPart]
public class FileSaveDialog : UserControl
{
  public FileSaveDialog()
  {
    InitializeComponent();
  }

  [CreateNew]
  public FileSaveDialogPresenter Presenter
  {
    get { return _presenter; }
    set { _presenter = value; _presenter.View = this; }
  }

  private FileSaveDialogPresenter _presenter;
}

public class FileSaveDialogPresenter :
Presenter<IFileSaveDialog>
{
  [InjectionConstructor]
  public FileSaveDialogPresenter(
    [ServiceDependency]IFileSaver fileSaver,
    [ServiceDependency]IContentReader contentReader)
  {
    _fileSaver = fileSaver;
    _contentReader = contentReader;
  }

  protected override void OnViewSet()
  {
    base.OnViewSet();
    View.OkClicked += SaveFile;
  }

  public void SaveFile(object o, EventArgs e)
  {
    _fileSaver.SaveFile(_contentReader.GetContent(),
View.GetFileName());
  }
}
```

Ну и сам воркайтем сохранения файла:

```
public class FileSaveWorkItem : WorkItem
{
  protected override void InitializeServices()
  {
    base.InitializeServices();
    Services.AddNew<FileSaver, IFileSaver>();
  }

  protected override void OnInitialized()
  {
    base.OnInitialized();
    Commands.AddNew<Command>(CommandNames.SaveFile);
    CommandBarButton invoker = Commands[CommandNames.
SaveFile].AddInvoker(new CommandBarButton("Сохранить
файл»), "ItemClick");
    UIExtensionSites[SiteNames.MainToolbar].
Add<CommandBarButton>(invoker);
  }

  [CommandHandler(CommandNames.SaveFile)]
  public void HandleSave(object a, object b)
  {
    object smartPart = SmartParts.
```

```
AddNew<FileSaveDialog>();
    Workspaces[WorkspaceNames.ModalWorkspace].
Show(smartPart);
  }
}
```

Как это все работает: программист добавляет этот воркайтем в модуль работы с файлами. Модуль на этапе инициализации приложением создает и инициирует этот воркайтем:

```
    WorkItems.AddNew<FileSaveWorkItem>().Run();
```

Как видим, воркайтем сохранения файлов на этапе инициализации добавляет кнопку сохранения файлов в тулбар приложения. По этой кнопке запускается команда — ее обработчик находится в воркайтеме. Обработчик создает смарт-парт диалога сохранения, находит модальную рабочую зону (воркспейс, который может показывать контролы в модальном режиме) и отображает его пользователю. Пользователь кликает на кнопку Ok, тогда обработчик кнопки в презентере контрола сохраняет файл с помощью сервисов. Как видно, нет нужды ручками прописывать зависимости (например, зависимость контрола от презентера, или сервисные зависимости в презентере — фабрики сделают это автоматически с использованием **Object Builder**).

Это очень простой пример. Можно даже сказать, нереальный. Но наглядный:)

Плюсы и минусы

CAB — тяжеленная конструкция, ее трудно поднять, но потом очень легко поддерживать. Если у вас приложение на два формы и одно модальное окошко — то здесь, разумеется, нет нужды использовать какие-то фреймворки. Но если это приложение на несколько АРМ-ов с пересекающимся функционалом, интеграцией и т.п., то CAB — то, что доктор прописал. Особенно, если людей мало, тестеров нет, и требования меняются. В него достаточно тяжело въехать новичку, поэтому несколько дней придется поводить новичка за ручку. Его ненавидят отдаленные от технологий менеджеры, потому что им кажется, что это очень большое усложнение системы — все глюки, которые появятся в проекте, будут сразу записываются на ваш счет, даже если к вам и CABу никакого отношения не имеют. Для него надо продумать юскейсы, причем хорошо продумать, т.е. нужна работа аналитиков. Но при наличие сплоченной команды в два-три человека и тех-лида по фронтэнду, разработка превращается в одно удовольствие.

Где взять

Скачать CAB отсюда и поставить.

Что дальше

Не совсем понятно, что дальше. Может, вы подскажете, про что дальше писать? Если не подскажете, то я буду писать про все те аспекты, которые пометил «нуждаются в отдельной статье». Вообще говоря, могу взять какой-то простенький пример и сделать полноценное CAB-приложение.

Учет пользователей с ограниченными возможностями при создании веб-сайта

Skaizer, перевод

Установлено, что 75% населения США и 65% населения Великобритании имеют доступ к сети Интернет дома. Перед веб-дизайнерами ставится сложная задача — спроектировать интерфейс так, чтобы он был понятен и доступен абсолютно всем. Указания WCAG могут помочь веб-дизайнерам создавать понятные сайты, но мы должны использовать нечто большее, чем эти указания, если мы хотим создавать сайты, доступные пользователям с ограниченными возможностями, использующих специальные средства. В этой статье мы объясним несколько простых методов, которые, при внедрении в дизайн веб-сайта, увеличат его доступность и полезность для людей со зрительными, слуховыми, физическими, познавательными недостатками или необучаемостью.

Небольшая статистика

В Великобритании:

— у 2 млн. жителей нарушения зрения;
— 9 млн. жителей имеют нарушения слуха;
— 3.4 млн. людей имеют физические нарушения;
— 1.5 млн. людей имеют умственные нарушения;
— у 6 млн. человек дислексия.

В США:

— у 10 млн. жителей нарушения зрения;
— 28 млн. жителей имеют нарушения слуха;
— 8 млн. людей имеют физические нарушения;
— 6.8 млн. людей имеют умственные нарушения;
— у 25 млн. человек дислексия.

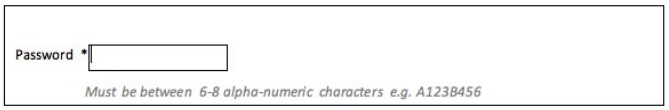
Обратите внимание — методы, применяемые для пользователей с ограниченными возможностями, помогут сделать сайт проще в использовании для каждого.

Нарушение зрения — слепота

Большинство слепых людей используют особое программное обеспечение для просмотра информации с экрана, такое как JAWS или Windows-Eyes, чтобы читать страницы веб-сайта. Кроме того, некоторые пользователи получают доступ к сайтам с помощью устройства Брайля, преобразующего текст веб-сайта в шрифт Брайля. Однако, люди, владеющие азбукой Брайля, встречаются намного реже — многие слепые люди потеряли зрение во время жизни, а не с рождения, а следовательно не обучены системе Брайля.

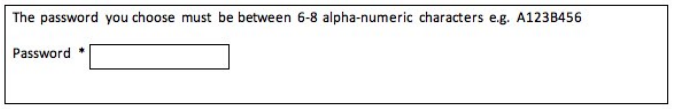
Ключевые методы доступного и практичного дизайна для слепых пользователей. Располагайте инструкции перед полями формы

Так как программное обеспечение для чтения информации с экрана просматривает содержимое в линейном порядке, необходимо размещать важную информацию о требованиях к заполнению поля формы перед самим полем. Если вы сделаете наоборот, слепые пользователи обнаружат специальные требования после того, как они заполнят поле формы.



Другой пример, при котором появляется та же проблема, возникает, когда звездочка «обязательное к заполнению поле» ставится после поля ввода. Пользователи со слепотой не буду знать, что поле формы обязательно для заполнения, пока не переместятся к следующему.

Для повышения доступности и практичности для пользователей, страдающих слепотой, требования к заполнению поля формы должны располагаться перед самим полем. Если пользователю необходимо ввести данные в поле формы, то звездочку необходимо поставить внутри подписи к полю. Для большего увеличения доступности и практичности можно поставить перед формой заметку, объясняющую пользователям, что звездочка обозначает обязательное для заполнения поле.

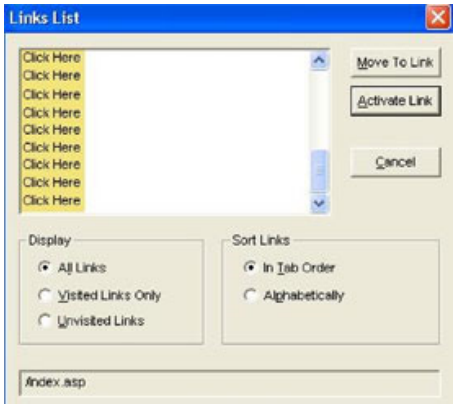


Создание ссылки «перейти к основному контенту»

Для пользователей, страдающих слепотой, очень полезной будет ссылка «перейти к основному контенту», позволяющая перепрыгнуть через длинную навигацию к главной информации веб-страницы, уменьшая количество содержимого, которое им нужно прослушать. Более того, если на сайте есть часто используемые функции, например, поиск, также будет полезно иметь ссылки для перехода к этим функциям.

Убедитесь, что текст ссылок информативен

Слепые пользователи, использующие ПО JAWS, могут слушать называния ссылок на веб-странице через опцию «список ссылок». Если текст ссылки неинформативен — в нем используются лишь фразы вида «нажмите сюда» или «подробнее», слепые пользователи никак не смогут определить, куда приведет их эта ссылка.



При использовании этой опции рекомендуется писать в тексте ссылки место её назначения. Например, вместо «нажмите здесь, чтобы узнать больше об Agoo IT» было бы лучше написать: «узнайте больше о Agoo IT». Слова «нажмите здесь» убраны из второго варианта, так как это термин, относящийся к работе с мышкой.

Создавайте информативные заголовки веб-страниц

Первое, что услышит пользователь читающего ПО при открытии страницы — это ее <title>. Следовательно, важно использовать заголовок, который отражает содержимое веб-страницы. Например, страница с контактной информацией должна иметь заголовок «Наши контакты». Слепые пользователи после прослушивания заголовка страницы должны точно определить, попали ли они на правильную страницу и хотят ли прослушать ее содержимое.

Используйте сжатый альтернативный текст

Слепые пользователи узнают, что изображено на картинке по ее альтернативному тексту. Когда изображение несет важную информацию, к нему необходимо добавить краткий подходящий альтернативный текст. Например, логотип веб-сайта компании «ЭЮЯ» должен быть обозначен, как alt=»Компания ЭЮЯ». Размещение слишком большого количества информации, как, например, «Компания ЭЮЯ, синий фон с фиолетовым текстом и желтой подсветкой» может запутать слепых пользователей.

В случае, если изображение чисто декоративное или использовано для разметки, к нему нужно добавить пустой альтернативный текст: alt=«». Тогда этого читающее экран ПО не будет произносить название файла с изображением.

Создавайте информативные заголовки

Важно создавать наглядные заголовки, которые семантически отмечаются тегами от <h1> до <h6>. Слепые пользователи часто прослушивают заголовки отдельно от основного содержимого страницы, с помощью списка заголовков. Это позволяет получить быстрый доступ именно к той информации, которая нужна пользователю, вместо прослушивания всей веб-страницы.

Тег <h1> должен содержать главный заголовок, чтобы можно было быстро определить о чем рассказывается на странице.

Создавайте аудио описание и транскрипции видео файлов

Слепые пользователи используют аудио описания, чтобы получить дополнительную информацию о важной визуальной составляющей, показываемой в видеофайле. Например, в сцене погони, когда в качестве звукового сопровождения используется только музыкальный фрагмент, необходимо использовать аудио описание, чтобы передать происходящие события, такие как «два вора спускаются по лестнице, спасаясь от полиции» — потому что слепые пользователи не смогут понять, о чем рассказывается в видео файле, слушая только его звуковое сопровождение.

Другая полезная опция для аудио и видео содержимого — это транскрипции. Транскрипции — это записанные сводки аудио и видео контента, которые могут содержать дополнительную информацию, такую как комментарии или описания, чтобы может помочь лучше понять содержимое. Транскрипции позволяют слепым пользователям получать доступ к аудио и видео контенту альтернативным способом.

Помните, что слепые пользователи не могут использовать мышь

Важно не забывать, что слепые пользователи не используют мышь — они применяют клавиатуру для навигации по веб-сайту. Поэтому необходимо, чтобы весь функционал и информация сайта, включая

управление видео в медиа-плеерах на сайте, были доступны с помощью клавиатуры.

Нарушения зрения — слабое зрение

Люди со слабым зрением используют увеличительное программное обеспечение, чтобы облегчить просмотр веб-сайта. В зависимости от серьезности нарушения зрения, они могут комбинировать увеличительное и читающее экран программное обеспечение, используя такие продукты, как Supernova или ZoomText. При небольших проблемах со зрением пользователи могут просто увеличить размер текста по умолчанию в настройках своих браузеров или изменить цвет, чтобы сделать содержимое сайта удобнее для чтения.

Ключевые принципы доступного и практичного дизайна для пользователей со слабым зрением. Не используйте изображения с текстом

Если это возможно, избегайте изображений с текстом, вместо них лучше применять специально настроенный HTML. Изображения с текстом мешают всем пользователям с нарушениями зрения. Для тех, кто использует увеличивающее ПО, качество изображения потеряется при увеличении, а это сделает текст трудночитаемым. Кроме того, на изображения с текстом не будут влиять цвета, настроенные в веб-браузере или увеличение текста.

Убедитесь, что размер текста можно изменять

Если создавать контент, размер которого может меняться, пользователи с нарушениями зрения будут менять величину текста на веб-странице, чтобы сделать его более читаемым.

Размещайте ключевую информацию в специальных местах экрана

Пользователи, которые используют увеличивающее ПО могут видеть только небольшую часть веб-страницы в каждый момент времени и буду искать контент и функционал сайта в специальных местах. Например, функция поиска по веб-сайту обычно располагается в правом верхнем углу страницы, если она находится где-нибудь ещё, то ее могут пропустить.

Используйте цвета с высоким контрастом

Для повышения читабельности сайта для людей с нарушениями зрения, убедитесь, что цвета, применяемые на сайте, имеют хороший контраст. Этого можно достичь, проверив выбранные цвета с помощью Juicy Studio color contrast analyzer. Тестируя различные сочетания основного и фонового цветов можно определить, соответствуют ли выбранные цвета рекомендациям WCAG. Кроме того, можно сделать несколько различных цветовых схем, чтобы облегчить читаемость.

Нарушения слуха

Люди с нарушениями слуха обычно не пользуются вспомогательным программным обеспечением для улучшения просмотра сайтов. Вместо этого, им требуются сайты, представляющие аудио контент в альтренативных форматах, таких как титры или расшифровки.

Делая аудио контент доступным для пользователей с нарушениями слуха, вы также повышаете его доступность для остальных пользователей, которые находится в ситуации, при которой аудио информацию нельзя прослушать. Например, они могут смотреть видео с выключенным звуком в библиотеке или находиться в шумном месте, где сложно услышать что-либо, могут пользоваться компьютером без аудио колонок.

Ключевые принципы доступного и практичного дизайна для пользователей со нарушениями слуха. Создавайте титры для любого видео контента

Титры создаются вместе с самим видео контентом. Они должны быть синхронизированы с аудио так, чтобы содержимое было целиком понятно без звука.

Создавайте расшифровки исполняемого аудио

Если контент роговаривается без видео, как например в подкастах, важно сделать расшифровку. Рекомендуется создавать расшифровки с помощью простого и доступного HTML, а не в виде документа Microsoft Word или Adobe PDF, чтобы как можно более широкая публика получила доступ к ним.

Физические нарушения

Люди с физическими нарушениями варьируются от тех, кто имеет временные повреждения, например сломаную руку, до людей с параличем, которые не могут пользоваться ни одной конечностью. В зависимости от тяжести физического нарушения эти пользователи могут заходить на веб-сайты, используя распознающее голос ПО, например Dragon Naturally Speaking. Однако имеется то, что объединяет всех пользователей с физическими нарушениями — ограничение или отсутствие возможности использования мыши. Это означает, что контент вебсайта, требующий клика мышкой или хорошей моторики недоступен таким пользователям.

Ключевые принципы доступного и практичного дизайна для пользователей с физическими нарушениями.Убедитесь, что весь контент может быть доступен с помощью клавиатуры

Пользователи с физическими нарушениями имеют ограничения или отсутствие возможности поль-

зования мышкой, и поэтому перемещаются по веб-сайтам с помощью клавиатуры. Функции, зависимые от мышки, такие как обработчик события onchange, обычно используемое с выпадающими меню, осложняют работу при использовании клавиатуры (как только пользователь перемещается на выпадающее меню и нажимает стрелку вниз для выбора, событие onchange мгновенно запускается, и в результате автоматически выбирается первая строка, вне зависимости от того, что хотел пользователь).

Создавайте состояние focus для ссылок

Так как пользователи с физическими нарушениями постоянно пользуются клавиатурой для всех действий, важно создавать состояние :focus для ссылок. Это можно осуществить, присваивая другой стиль текста ссылке, из-за чего при наведении курсора цвет или что-либо другое изменяется и делает понятным то, где находится пользователь на странице. Если вставить в CSS следующий пример, ссылки будут становиться черными при наведении курсора либо мышкой, либо клавиатурой:

```
a:hover, a:focus, a:active {color: #000000;}
```

В результате получится:

Создавайте видимые ссылки перехода

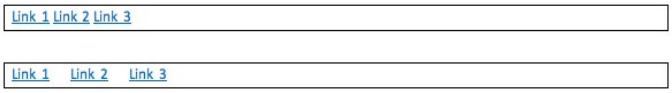
Ссылки перехода — это ссылки, становящиеся видимыми при наведении и помогающие пользователям с физическими нарушениями. Пользователи, использующие клавиатуру, должны проходить всю страницу, чтобы добраться до ссылки, которая их интересует, а ссылки перехода позволяют обойти долгую навигацию и уменьшить количество нажатий клавиш, требующееся для нажатия нужной ссылки.

Убирайте двигающиеся объекты

Не используйте двигающиеся цели, такие как маятники, потому что они могут показаться пользователям с физическими нарушениями слишком трудными для использования. При пониженной моторике сложно четко контролировать движение мышки и нажимать на объект, пока он перемещается; пользователям, работающим с помощью клавиатуры, может не хватить времени для наведения на нужный объект, пока он не скроется из виду.

Создавайте большие активные области

Некоторые пользователи с физическими нарушениями будут использовать мышь для доступа к веб-сайтам. Так как у них снижена моторика, важно, чтобы текст мог быть увеличен для облегчения нажатия на ссылку. Для большего увеличения удобства, сделаете достаточно пробелов между ссылками. Это позволит понизить предел погрешности, а значит пользователь не будет случайно нажимать неправильную ссылку.



Правильно называйте ссылки на изображения и элементы управления

Пользователи, использующие ПО, распознающее голос, перемещаются по веб-сайтам, говоря слова для активирования ссылок и элементов управления. Произносимые слова в основном не отличаются от тех, что показаны на веб-сайте. Поэтому важно убедиться, что все кнопки и изображения на форме корректно созданы, и их название совпадает с текстом или картинкой. Например, если на сайте есть кнопка «Купить», пользователь скажет слово «Купить» для нажатия этой кнопки, но если имя кнопки «Приобрести», то пользователь не сможет нажать на нее.

Нарушения познавательных способностей и необучаемость

Люди с нарушениями познавательных способностей и необучаемостью могут иметь проблемы с памятью, решением задач, восприятием и и мышлением. Кроме того, страдающие необучаемостью люди могут испытывать трудности при чтении и понимании (дислексия. В целях повышения доступности веб-сайта важно, чтобы содержание было написано простым языком, разметка страницы была несложной, навигация — понятной и единообразной, а движущиеся объекты отсутствовали и не нарушали восприятие.

Есть несколько других важных принципов доступного и практичного дизайна для пользователей с нарушениями познавательных способностей и необучаемостью:

Создайте единоообразный дизайн

Придерживайтесь одного внешнего вида и смысла на всех страницах веб-сайта. Проследите, чтобы вся навигация и главное содержимое находились в одних местах на каждой странице. Кроме того, выберите цветную раскраску различных секций веб-сайта. Пользователи с нарушениями познавательных способностей и необучаемостью смогут легче перемещаться по секциям, раскрашенным в разные цвета. Этого можно достичь, используя одинаковый цвет фона или баннер наверху каждой страницы в секции.

Сделайте карту сайта

Карта сайта позволит пользователям с нарушениями познавательных способностей и необучаемостью

четко понять состав всего контента, расположенного на сайте. Она также поможет пользователю напрямую открыть любую страницу сайта, если он потерялся.

Используйте выровненный слева шрифт без засечек с изменяющимся размером

С целью увеличения читаемости для пользователей с нарушениями познавательных способностей и необучаемостью, используйте шрифт без засечек, размер которого можно менять. Кроме того, используйте выравнивание слева. Текст труднее читать из-за неровных промежутков между словами. Курсив и заглавные буквы тоже необходимо свести к минимуму для повышения читабельности.

Создавайте полезные сообщения об ошибках

Если пользователь делает ошибки, помогите ему, создав полезные сообщения о них. Например, если обязательные поля формы не были заполнены, сообщение об ошибке должно сказать пользователю какие поля необходимо заполнить.

Предложите словесный вывод

Например, на сайтах организаций Browse Aloud и Textic имеется возможность вслух проговорить содержимое веб-сайта при выделении слов на странице. Использование этой функции особенно полезно для пользователей, которым трудно прочитать большие объемы текста.

Создавайте различные цветовые схемы

Людям с нарушениями познавательных способностей может помочь выбор различных цветовых схем.

Заключение

Я надеюсь, что эта статья немного объяснила то, как пользователи с различными нарушениями просматривают веб-сайты, и как простые изменения играют большую роль в помощи этим просмотрам. Пока этот список не полон, методы, описанные в статье, могут помочь повысить доступность и удобство веб-сайтов для людей со зрительными, слуховыми, физическими, познавательными нарушениями и необучаемостью.

Алгоритм Шинглов — поиск нечетких дубликатов текста

Skaizer

В этой статье я расскажу об алгоритме поиска нечетких дубликатов под названием «Алгоритм Шинглов». А так же реализую данный алгоритм на языке Python. Почему я решил изучить данный алгоритм? Сам я являюсь SEO-шником, занимаюсь продвижением сайтов и так далее… Соответственно, моя работа заключается в изменении выдачи поисковой системы по определенному запросу. Но проработав более года в этом направлении меня заинтересовала внутренняя часть поисковых систем. Как они борются с поисковым спамом, как ранжируют документы и т.д. Поиск нечетких дубликатов позволяет поисковой системе исключить из выдачи клоны или частично похожие страницы (под словом частично я подразумеваю некоторое значение, при котором в конкретной поисковой системе два документа будут определяться как почти одинаковыми). Одним из таких алгоритмов является «Алгоритм Шинглов» (шингл на английском означает чешуйка).

Немного теории

Поиск нечетких дубликатов позволяет предположить, являются ли два объекта частично одинаковыми или нет. Под объектом могут пониматься текстовые файлы и другие типы данных. Мы будем работать с текстом, но поняв, как работает алгоритм, вам не составит труда перенести мою реализацию на необходимые вам объекты.

Обратите внимание, задачей не стоит определить абсолютное значение схожести объектов, а так же выделения в каждом из объектов схожих частей. Нам необходимо только предположить, являются ли объекты почти дубликатами или нет.

Где может применяться данный алгоритм?

Как я уже писал выше, он может быть применен в поисковой системе для очистки поисковой выдачи. Так же данный алгоритм может использоваться для кластеризации документов по их схожести.

Почти одинаковый текст

Рассмотрим задачу алгоритма на примере текста. Допустим, мы имеем файл с текстом в 8 абзацев. Делаем его полную копию, а затем переписываем только последний абзац. 7 из 8 абзацев второго файла являются полной копией оригинала.

Другой пример, посложнее. Если мы в копии оригинального текста перепишем каждое 5-6е предложение, то текст по-прежнему будет являться почти дублем.

Представим, что мы имеем большой форум или портал, где контент составляется пользователями. Как показали наблюдения, пользователи имеют привычку заниматься копи-пастом, и кражей контента, лишь немного изменив его. На нашем сайте имеется поисковый механизм. Пользователь вводит какой-либо запрос, и на первых позициях получает десяток текстов, которые отличаются только одним или двумя предложениями. Естественно ему не интересно просматривать их все, и он понимает, что поисковая система работает некорректно.

Поступим логичнее. В поисковой выдаче вместо десятка практически одинаковых документов покажем один из них, например, наиболее релевантный запросу, а ниже дадим ссылочку на список так же найденных страниц, похожих на него.

И таким образом, при наличии множества почти одинакового текста мы распределим его на группы и предоставим пользователю ссылку на эти группы, вместо того, чтобы сваливать все в кучу.

Как работает алгоритм Шинглов?

Итак, мы имеем 2 текста и нам нужно предположить, являются ли они почти дубликатами. Реализация алгоритма подразумевает несколько этапов:

— канонизация текстов;
— разбиение текста на шинглы;
— нахождение контрольных сумм;
— поиск одинаковых подпоследовательностей.

Теперь поконкретнее. В алгоритме шинглов реализовано сравнение контрольных сумм текстов. В своей реализации я использую CRC32, но применимы и другие, например SHA1 или MD5 и т.д. Как известно, контрольные суммы статических функций очень чувствительны к изменениям. Например, контрольные суммы двух следующих текстов будут в корне отличаться:

— Текст: «My war is over.» Контрольная сумма: 1759088479
— Текст: «My war is over!» Контрольная сумма: -127495474

Как сказал Джон Рэмбо: «My war is over». Но сказать он это мог по-разному, громко восклицая или

тихо шепча, так и у нас, отличие второй фразы от первой заключается всего лишь в восклицательном знаке в конце фразы, но как видим, контрольные суммы абсолютно разные.

При поиске почти дубликата нас не интересуют всякие там восклицательные знаки, точки, запятые и т.д. Нас интересуют только слова (не союзы, предлоги и т.д., а именно слова).

Следовательно, ставится задача: очистить текст от ненужных нам знаков и слов, которые не несут смысла при сравнении, это и называется «привести текст к канонической форме».

Итак, нам необходимо создать набор символов, которые нужно исключить из обоих текстов, т.е. приведения его к форме, пригодной для нашего сравнения.

Давайте что-нибудь напрограммируем.
Допустим, мы имеем два текста, занесем их в 2 переменных: source1 и source2. Нужно их почистить от ненужных символов и слов. Определим наборы стоп-слов и стоп-символов.

```
stop_symbols = '.,!?:;-\n\r()'

stop_words = (u'это', u'как', u'так',
  u'и', u'в', u'над',
  u'к', u'до', u'не',
  u'на', u'но', u'за',
  u'то', u'с', u'ли',
  u'а', u'во', u'от',
  u'со', u'для', u'о',
  u'же', u'ну', u'вы',
  u'бы', u'что', u'кто',
  u'он', u'она')
```

В стоп-символы мы включили знаки препинаний, знаки переноса строки и табуляции. В стоп-лова мы включили союзы, предлоги и прочие слова, которые при сравнении не должны влиять на результат.

Создадим функцию, которая будет производить канонизацию текста:

```
def canonize(source):
  stop_symbols = '.,!?:;-\n\r()'

  stop_words = (u'это', u'как', u'так',
    u'и', u'в', u'над',
    u'к', u'до', u'не',
    u'на', u'но', u'за',
    u'то', u'с', u'ли',
    u'а', u'во', u'от',
    u'со', u'для', u'о',
    u'же', u'ну', u'вы',
    u'бы', u'что', u'кто',
    u'он', u'она')

  return ( [x for x in [y.strip(stop_symbols) for y in
source.lower().split()] if x and (x not in stop_words)]
)
```

Функция canonize очищает текст от стоп-символов и стоп-слов, приводит все символы строки к нижнему регистру и возвращает список, оставшихся после чистки слов.

Вообще, приведение текста к единой канонической форме не ограничено действиями, которые я описал. Можно, например, каждое из существительных приводить к единственному числу, именительному падежу и т.д., для этого нужно подключать морфологические анализаторы русского языка (или других языков, где необходимы эти действия).

Итак, тексты у нас очищены от всего лишнего. Теперь необходимо разбить каждый из них на подпоследовательности — шинглы. На практике я применяю подпоследовательности длинной в 10 слов. Из выделенных нами шинглов далее будут находиться контрольные суммы.

Разбиение на шинглы будет происходить внахлест через одно слово, а не встык, т.е., имеем текст:

«Разум дан человеку для того, чтобы он разумно жил, а не для того только, чтобы он понимал, что он неразумно живет.»© В. Г. Белинский

После обработки нашей функцией текст примет следующий вид:

разум дан человеку того чтобы разумно жил того только чтобы понимал неразумно живет

Это и есть каноническая форма. Теперь нужно разбить ее на шинглы длиной в 10 слов. Так как шинглы составляются внахлест, то всего шинглов len(source)-(shingleLen-1), т.е. количество слов в тексте минус длина шинглов плюс 1.

Шинглы будут выглядеть следующим образом:

— Sh1 = разум дан человеку того чтобы разумно жил того только чтобы
— Sh2 = дан человеку того чтобы разумно жил того только чтобы понимал
— Sh3 = человеку того чтобы разумно жил того только чтобы понимал неразумно
— Sh4 = того чтобы разумно жил того только чтобы понимал неразумно живет

Напишем функцию, которая производит разбиение текста на шинглы:

```
def genshingle(source):
  shingleLen = 10 #длина шингла
  out = []
  for i in range(len(source)-(shingleLen-1)):
    out.append (' '.join( [x for x in
source[i:i+shingleLen]] ).encode('utf-8'))

  return out
```

Но так как нас интересую именно контрольные

суммы шинглов, то соответственно изменим нашу функцию. Мы будем использовать алгоритм хэширования CRC32, подключим библиотеку binascii, которая содержит в себе нужный нам метод. Измененная функция:

```
def genshingle(source):
import binascii
shingleLen = 10 #длина шингла
out = []
for i in range(len(source)-(shingleLen-1)):
out.append (binascii.crc32(' '.join( [x for x in
source[i:i+shingleLen]] ).encode('utf-8')))

return out
```

Наш пример выдаст следующие наборы контрольных сумм:

```
[1313803605, -1077944445, -2009290115, 1772759749]
```

Таким образом, через наши 2 функции нужно прогнать 2 сравниваемых текста и мы получим 2 множества контрольных сумм шинглов, теперь необходимо найти их пересечения.

```
def compaire (source1,source2):
  same = 0
  for i in range(len(source1)):
    if source1[i] in source2:
    same = same + 1

  return same*2/float(len(source1) + len(source2))*100
```

Вот и все, функция нам вернет процент схожести двух текстов.

Скомпонованный код

```
# -*- coding: UTF-8 -*-
if __name__ == '__build__':
    raise Exception

def canonize(source):
        stop_symbols = '.,!?:;-\n\r()'

        stop_words = (u'это', u'как', u'так',
        u'и', u'в', u'над',
        u'к', u'до', u'не',
        u'на', u'но', u'за',
        u'то', u'с', u'ли',
        u'а', u'во', u'от',
        u'со', u'для', u'о',
        u'же', u'ну', u'вы',
        u'бы', u'что', u'кто',
        u'он', u'она')

        return ( [x for x in [y.strip(stop_symbols) for
y in source.lower().split()] if x and (x not in stop_
words)] )

def genshingle(source):
    import binascii
    shingleLen = 10 #длина шингла
    out = []
    for i in range(len(source)-(shingleLen-1)):
        out.append (binascii.crc32(' '.join( [x for x
in source[i:i+shingleLen]] ).encode('utf-8')))

    return out

def compaire (source1,source2):
    same = 0
    for i in range(len(source1)):
        if source1[i] in source2:
            same = same + 1

    return same*2/float(len(source1) + len(source2))*100

def main():
    text1 = u'' # Текст 1 для сравнения
    text2 = u'' # Текст 2 для сравнения

    cmp1 = genshingle(canonize(text1))
    cmp2 = genshingle(canonize(text2))

    print compaire(cmp1,cmp2)

# Start program
main()
```

Я показал именно реализацию ядра алгоритма, но его можно дорабатывать до бесконечности, увеличивая его производительность и точность сравнения.

Сам по себе алгоритм достаточно ресурсоемкий, поэтому не помешает для него создать механизм кеширования, который будет особенно актуален для сравнения наборов файлов, чтобы не высчитывать контрольные суммы каждый раз.

Так же, для увеличения производительности при обработке больших объемов текста можно сравнивать не все полученные контрольные суммы, а только те, которые, например, делятся на 25, или любое целое число в пределах от 10 до 40. Как показали тесты, это дает **значительный(!)** прирост скорости и не сильно уменьшает точность, но только при обработке больших объемов.

Существуют модифицированные версии «Алгоритма шинглов» — «Алгоритм супершинглов» и «Алгоритма мегашинглов», их реализацию я представлю позже.

Кузявые ли бутявки, т.е. пишем морфологический анализатор на Python

kmike

Морфологический анализатор для русского языка — это что-то заумное? Программа, которая приводит слово к начальной форме, определяет падеж, находит словоформы — непонятно, как и подступиться? А на самом деле все не так и сложно. В статье — как я писал аналог mystem, lemmatizer и phpmorphy на Python, и что из этого получилось. Сразу скажу, получилась библиотека для морфологического анализа на Python, которую Вы можете использовать и дорабатывать согласно лицензии MIT.

Первый вопрос — зачем все это?

Потихоньку делаю один хобби-проект, назрело решение писать его на Python. А для проекта был жизненно необходим более-менее качественный морфологический анализатор. Вариант с mystem не подошел из-за лицензии и отсутствием возможности поковыряться, lemmatizer и phpmorphy меня слегка напрягали своими словарями не в юникоде, а приличного аналога на python я почему-то не нашел. Вообщем причин много, все, если честно, не очень серьезные, на самом деле просто захотелось. Решил я, в итоге, изобрести велосипед и написать свою реализацию.

За основу взял наработки с сайта aot.ru. Словари (LGPL) для русского и английского, а также идеи — оттуда. Там были описаны и конкретные алгоритмы реализации, но в терминах теории автоматов. Я знаком с теорией автоматов (даже диплом по формальным грамматикам защитил), но мне показалось, что тут можно прекрасно обойтись без нее. Поэтому реализация — независимая, что имеет как плюсы, так и минусы.

Для начала, библиотека должна, как минимум, уметь:
1. Приводить слово к нормальной форме (например, в ед.ч., И.п. для существительных) — для слова «ЛЮДЕЙ» она должна вернуть «ЧЕЛОВЕК»
2. Определять форму слова (или формы). Например, для слова «ЛЮДЕЙ» она должна как-то дать знать, что это существительное, во множественном числе, может оказаться в родительном или винительном падежах.

Разбираемся со словарями

Словари с сайта aot.ru содержат следующую информацию:
1. парадигмы слов и конкретные правила образования;
2. ударения;
3. пользовательские сессии;
4. набор префиксов (продуктивных приставок);
5. леммы (незменяемые части слова, основы);
и еще есть 6. грамматическая информация — в отдельном файле.

Из этого всего нам интересны правила образования слов, префиксы, леммы и грамматическая информация.

Все слова образуются по одному принципу:
[префикс]+[приставка]+[основа]+[окончание]

Префиксы — это всякие «мега», «супер» и т.д. Набор префиксов хранится просто списком.

Приставки — имеются в виду приставки, присущие грамматической форме, но не присущие неизменяемой части слова («по»,«наи»). Например, «наи» в слове «наикрасивейший», т.к. без превосходной степени будет «красивый».

Правила образования слов — это то, что надо приписать спереди и сзади основы, чтобы получить какую-то форму. В словаре хранятся пары «приставка — окончание», + «номер» записи о грамматической информации (которая хранится отдельно).

Правила образования слов объединены в парадигмы. Например, для какого-нибудь класса существительных может быть описано, как слово выглядит во всех падежах и родах. Зная, что существительное принадлежит к этому классу, мы сможем правильно получить любую его форму. Такой класс — это и есть парадигма. Первой в парадигме всегда идет нормальная форма слова (если честно, вывод эмпирический))

Леммы — это неизменяемые части слов. В словаре хранится информация о том, какой лемме соответствуют какие парадигмы (какой набор правил для образования грамматических форм слова). Одной лемме может соответствовать несколько парадигм.

Грамматическая информация — просто пары («номер» записи, грам. информация). «Номер» в кавычках, т.к. это 2 буквы, просто от балды, но все разные.

Файл со словарем — обычный текстовый, для каждого раздела сначала указано число строк в нем, а потом идут строки, формат их описан, так что написать парсер труда не составило.

Поняв структуру словаря, уже несложно написать первую версию морфологического анализатора.

Пишем морфологический анализатор

По сути, нам дано слово, и его надо найти среди всех разумных комбинаций вида
<префикс>+<приставка>+<лемма>+<окончание>
и
<приставка>+<лемма>+<окончание>

Дело упрощает то, что оказалось (как показала пара строчек на питоне), что «приставок» у нас в языке (да и в английском вроде тоже) всего 2. А префиксов в словаре — порядка 20 для русского языка. Поэтому искать можно среди комбинаций **<префикс>+<лемма>+<окончание>**, объединив в уме список приставок и префиксов, а затем выполнив небольшую проверочку.

С префиксом разберемся по-свойски: если слово начинается с одного из возможных префиксов, то мы его (префикс) отбрасываем и пытаемся морфологически анализировать остаток (рекурсивно), а потом просто припишем отброшенный префикс к полученным формам.

В итоге получается, что задача сводится к поиску среди комбинаций **<лемма>+<окончание>**, что еще лучше. Ищем подходящие леммы, потом смотрим, есть ли для них подходящие окончания*.

Тут я решил сильно упростить себе жизнь, и задействовать стандартный питоновский ассоциативный массив, в который поместил все леммы. Получился словарь вида lemmas: {base -> [rule_id]}, т.е. ключ — это лемма, а значение — список номеров допустимых парадигм. А дальше поехали — сначала считаем, что лемма — это первая буква слова, потом, что это 2 первых буквы и т.д. По лемме пытаемся получить список парадигм. Если получили, то в каждой допустимой парадигме пробегаем по всем правилам и смотрим, получится ли наше слово, если правило применить. Получается — добавляем его в список найденных форм.

**Еще был вариант — составить сразу словарь всех возможных слов вида <лемма>+<окончание>, получалось в итоге где-то миллионов 5 слов, не так и много, но вариант, вообщем, мне не очень понравился.*

Дописываем морфологический анализатор

По сути — все готово, мы написали морфологический анализатор, за исключением некоторых деталей, которые заняли у меня 90% времени)

Деталь первая

Если вспомнить пример, который был в начале, про «ЛЮДЕЙ» — «ЧЕЛОВЕК», то станет понятно, что есть слова, у которых неизменяемая часть отсутствует. И где тогда искать формы этих слов — непонятно. Пробовал искать среди всех окончаний (точно так же, как и среди лемм), ведь для таких слов и «ЛЮДЕЙ», и «ЧЕЛОВЕКУ» в словаре будут храниться как окончания. Для некоторых слов работало, для некоторых выдавало, кроме правильного результата, еще и кучу мусора. В итоге, после долгих экспериментов выяснилось, что есть в словаре такая хитрая магическая лемма «#», которая и соответствует всем пустым леммам. Ура, задача решена, ищем каждый раз еще и там.

Деталь вторая

Хорошо бы иметь «предсказатель», который смог бы работать и со словами, которых нет в словаре. Это не только неизвестные науке редкие слова, но и просто описки, например.

Тут есть 2 несложных, но вполне работающих подхода:
1. Если слова отличаются только тем, что к одному из них приписано что-то спереди, то, скорее всего, склоняться они будут однаково
2. Если 2 слова оканчиваются одинаково, то и склоняться они, скорее всего, будут одинаково.

Первый подход — это угадывание префикса.
Реализуется очень просто: пробуем считать сначала одну первую букву слова префиксом, потом 2 первых буквы и т.д. А то, что осталось, передаем морфологическому анализатору. Ну и делаем это только для не очень длинных префиксов и не очень коротких остатков.

Второй (предсказание по концу слова) подход чуть сложнее в реализации (так-то сильно сложнее, если нужна хорошая реализация)) и «поумнее» в плане предсказаний.

Первая сложность связана с тем, что конец слова может состоять не только из окончания, но и из части леммы. Тут я тоже решил «срезать углы» и задействовал опять ассоциативный массив с предварительно подготовленными всеми возмоными окончаниями слов (до 5 букв). Не так и много их получилось, несколько сот тысяч. Ключ массива — конец слова, значение — список возможных правил. Дальше — все как при поиске подходящей леммы, только у слова берем не начало, а 1, 2, 3, 4, 5-буквенные концы, а вместо лемм у нас теперь новый монстромассив.

Вторая сложность — получается много заведомого мусора. Мусор этот отсекается, если учесть, что полученные слова могут быть только существительными, прилагательными, наречиями или глаголами.
Даже после этого у нас остается слишком много немусорных правил. Для определенности, для каждой части речи оставляем только самое распространенное правило. По идее, если слово не было предсказано как существительное, хорошо бы добавить вариант с неизменяемым существительным в ед.ч. и.п., но это в TODO.

Идеальный текст для проверки работы предсказателя — конечно же, «Сяпала Калуша по напушке», про

то, как она там увазила бутявку и что из этого вышло:

Сяпала Калуша по напушке и увазила бутявку. И волит:
— Калушата, калушаточки! Бутявка!
Калушата присяпали и бутявку стрямкали. И подудонились.
А Калуша волит:
— Оее, оее! Бутявка-то некузявая!
Калушата бутявку вычучили.
Бутявка вздребезнулась, сопритюкнулась и усяпала с напушки.
А Калуша волит:
— Бутявок не трямкают. Бутявки дюбые и зюмо-зюмо некузявые. От бутявок дудонятся.
А бутявка волит за напушкой:
— Калушата подудонились! Калушата подудонились! Зюмо некузявые! Пуськи бятые!

Из текста предсказатель не справился с именем собственным Калуша, с «Калушата»(они стали мужиками «Калуш» и «Калушат»), с междометием «Оее», загадочным «зюмо-зюмо», и вместо «Пуська» опять выдал мужика «Пусек», остальное все, на мой взгляд, определил правильно. Вообщем есть куда стремиться, но уже неплохо.

О ужас, «хабрахабр» предсказывается тоже никак. А вот «хабрахабра» — уже понимает, что «хабрахабр».

Тут можно было бы, в принципе, подвести итог, если бы компьютеры работали мгновенно. Но это не так, поэтому есть

Деталь №3: ресурсы процессора и оперативной памяти

Скорость работы первого варианта меня вполне устроила. Получалось, по прикидкам, тыщ до 10 слов в секунду для простых русских слов, около тыщи для навороченных. Для английского — еще быстрее. Но было 2 очевидных (еще до начала разработки) «но», связанных с тем, что все данные грузились в оперативную память (через pickle/cPickle).
1. первоначальная загрузка занимала 3-4 секунды
2. кушалось порядка 150 мегабайт оперативной памяти с psyco и порядка 100 без (+удалось немного сократить, когда привел всякие там питоновские set и list к tulpe, где возможно).

Не долго думая, провел небольшой рефакторинг и вынес все данные в отдельную сущность. А дальше мне на помощь пришла магия Python и duck typing. Вкратце — в алгоритмах использовались данные в виде ассоциативных и простых массивов. И все будет работать без переписывания алгоритмов, если вместо «настоящих» массивов подсунуть что-нибудь, что ведет себя как массив, а конкретнее, для нашей задачи, — поддерживает [] и in. Все) В стандартной поставке питона обнаружились такие «массивные» интерфейсы к нескольким нереляционным базам данных. Эти базы (bsddb, ndbm, gdbm), по сути, и есть ассоциативные массивы на диске. Как раз то, что нужно. Еще там обнаружилась пара высокоуровневых надстроек над этим хозяйством (anydbm и shelve). Остановился на том, что унаследовался от DbfilenameShelf и добавил поддержку ключей в юникоде и ключей-целых чисел. А, еще добавил кеширование результата, которое почему-то есть в Shelf, но убито в DbfilenameShelf.

Данные по скорости на тестовых текстах (995 слов, русский словарь, macbook):

```
get_pickle_morph('ru', predict_by_prefix = False): 0.214
CPU seconds
get_pickle_morph('ru'): 0.262 CPU seconds
get_shelve_morph('ru', predict_by_prefix = False): 0.726
CPU seconds
get_shelve_morph('ru'): 0.874 CPU seconds
```

Памяти вариант shelve, можно сказать, не кушал совсем.

Варианты shelve похоже, работали, когда словари уже сидели в дисковом кеше. Без этого время может быть и 20 секунд с включенным предсказателем. Еще замечал, что медленнее всего работает предсказание по префиксу: до того, как прикрутил кеширование к своим наследникам от DbfilenameShelf, тормозило это предсказание раз в 5 больше, чем все остальное вместе взятое. А в этих тестах вроде не сильно уже.

Кстати, пользуясь случаем, хочу спросить, вдруг кто-то знает, как в питоне можно узнать количество занятой текущим процессом памяти. По возможности кроссплатформенно как-нибудь. А то ставил в конец скрипта задержку и мерил по списку процессов.

Пример использования

```
import pymorphy
morph = pymorphy.get_shelve_morph('ru')
#слова должны быть в юникоде и ЗАГЛАВНЫМИ
info = morph.get_graminfo(unicode('Вася').upper())
```

Так все-таки, зачем?

В самом начале я уже объяснил, зачем стал писать морфологический анализатор. Теперь время объяснить, почему я стал писать эту статью. Я ее написал в надежде на то, что такая библиотека интересна и полезна, и вместе мы эту библиотеку улучшим. Дело в том, что функционал, который уже реализован, вполне достаточен для моих нужд. Я буду ее поддерживать и исправлять, но не очень активно. Поэтому эта статья — не подробная инструкция по применению, а попытка описать, как все работает.

В поставке

pymorphy.py — сама библиотека;

shelve_addons.py — наследники от shelf, может кому пригодится;

encode_dicts.py — утилита, которая преобразовывает словари из формата AOT в форматы pymorphy. Без параметров, работает долго, ест метров 200 памяти, запускается 1 раз. Сконвертированные словари не распространяю из-за возможной бинарной несовместимости и большого объема;

test.py — юнит-тесты для pymorphy;

example.py — небольшой пример и тексты с теми 995 словами;

dicts/src — папка с исходными словарями, скачанными с aot.ru;

dicts/converted — папка, куда encode_dicts.py будет складывать сконвертированные словари.

Напоследок

ссылки:
www.assembla.com/wiki/show/pymorphy
hg.assembla.com/pymorphy
trac-hg.assembla.com/pymorphy

Лицензия — MIT.

проверял только под Python 2.5.

Функциональное программирование для землян: функции и списки

uj2

Сразу скажу, почему я не буду рассказывать о чистом ФП. Чистое ФП подразумевает отсутствие состояния вычисления и немодифицируемость памяти (отсутствие побочных эффектов выполнения подпрограмм). Грубо говоря, вся программа в чистом ФП представляет собой одну большую формулу. Чисто императивные концепции, вроде последовательности вычислений, ввода-вывода и изменяемого состояния, в них реализуются различными красивыми способами вроде монад в Haskell.

Кроме того, в ФП включают различные концепции более высокого порядка:

— **совпадение по шаблону (pattern matching)** — наподобие перегрузки функций, только более продуманно и более гибко
— **продолжения (continuation)** — возможность останавливать вычисление и продолжать его в другое время (т.е. «консервировать» состояние вычисления и потом возобновлять его). Вид продолжения мы видим в работе оператора yield
— **ленивые вычисления** — при такой модели, грубо говоря, аргументы функций вычисляются только тогда, когда они реально необходимы, а не при входе в функцию
— **алгебраические типы данных, рекурсивные типы данных, автоматический вывод типов** — и т.д. и т.п.

На этом я концентрироваться не буду, поскольку а) сам не применял этих концепций на практике (или применял, но ограниченно), б) их применимость для «обычного» программиста на «обычных» языках пока недоступна. Поэтому мы начнем с более простых понятий.

Функции

В ФП все завязано вокруг функций, поэтому **функция должна быть объектом первого рода (first-class object)**. Это значит, что функцию можно создать (анонимная функция), можно присвоить переменной, можно передать в функцию, можно вернуть из функции. Вновь созданные функции должны обладать свойством **замыкания** (closure) — т.е. новая функция должна захватывать окружающий контекст (объявленные переменные как локальной, так и глобальной области видимости). Простой пример (полный код к данному посту вы можете найти по ссылкам внизу поста):

```
# encoding: utf-8
def get_writer(type, params):
  # вывод данных в HTML
  def html_writer(filename):
    f = open(filename + '.' + type, 'w');
    f.write(«»»
<html>
  <head>
    <title>%s</title>
  </head>
  <body>
    <h1>Hello</h1>
  </body>
«»» % params['title'])
    f.close()
  # вывод данных в PLAIN TEXT
  def text_writer(filename):
    f = open(filename + '.' + type, 'w');
    f.write(«»»
%s
================================
Hello
«»»)
    f.close()
  # определим, какой тип данных запросили, и вернем соответсвующую функцию
  if type == 'html':
    return html_writer
  elif type == 'txt':
    return text_writer
params = { 'title': 'Header' }
# выведем в HTML
f = get_writer('html', params)
f('file1')
# выведем в PLAIN TEXT
f = get_writer('txt', params)
f('file2')
```

Обратите внимание на то, что внутри html_writer и text_writer используются аргументы get_writer (type и params). Как такое может быть? Ведь после возврата из get_writer ее аргументы, по идее, перестают существовать? В этом и заключается суть замыкания: если одна функция возвращает другую, то к последней добавляется еще и так называемый контекст — совокупность всех доступных переменных (локальных, глобальных, аргументов) с их значениями на момент вызова. Таким образом, при возврате функции из функции вы возвращаете не просто функцию (простите за тавталогию), а **замыкание** — функцию + контекст.

Функции высшего порядка

Теперь представьте такой пример. Мы пишем программу построения графиков определенных функ-

ций. Определим пару таких функций:

```
# encoding: utf-8
import math

# y = k * x + b
def get_linear(k, b):
  return lambda x: k * x + b

# y = k * x^2 + b

def get_sqr(k, b):
  return lambda x: k * x ** 2 + b

# y = A * sin(k * x + phi)
def get_sin(amplitude, phi, k):
  return lambda x: amplitude * math.sin(math.radians(k
* x  + phi))

# y = A * e^(k*x)
def get_exp(amplitude, k, b):
  return lambda x: amplitude * math.exp(k * x + b)
```

Это простые функции. Как мы можем их использовать:

```
# y = 5 * sin(0.3 * x + 30)
y = get_sin(5, 30, 0.3)

# y(90) = 4.19
print y(90)
print
# результат применения y к интервалу от 0 до 180
print [ y(x) for x in range(0, 180) ]
```

Но, вы видите, каждая из этих функций поддерживает операцию сдвига функции по оси X. А ведь это отдельная функция и мы можем ее выделить! И точно так же мы можем выделить функции масштабирования по X и по Y:

```
def shifter(func, b):
  return lambda x: func(x + b)

def x_scaler(func, k):
  return lambda x: func(k*x)

def y_scaler(func, A):
  return lambda x: A * func(x)

def combine(func1, func2):
  return lambda x: func2(func1(x))
```

shifter, x_scaler, y_scaler, combine — это **функции высшего порядка** (high-order functions), т.к. они принимают не только скалярные аргументы, но и другие функции, модифицируя их поведение. Combine — крайне полезная общая функция, позволяющая применить одну функцию к другой. Теперь мы можем переписать наши предыдущие функции следующим образом:

```
def identity(x):
  return x

def sqr(x):
  return x ** 2
```

Уже интересней. Мы переименовали две функции, а две вообще убрали. Зачем переименовали? Дело в том, что избавившись от «шелухи» вроде масштабирования и переноса, мы получили еще более общие функции. Первая из них называется identity — функция идентичности — очень важное понятие в ФП. Очень важное, но очень простое — возвращает свой аргумент и все. Вторая функция просто возводит аргумент в квадрат. Теперь любую конфигурацию, которую мы могли описать нашими функциями из первого примера, мы можем получить путем комбинирования простых функций и функций высшего порядка — главное комбинировать их в корректной последовательности. Для демонстрации того, что значит в корректной последовательности, приведу такое выражение:

```
y = x_scaler(shifter(x_scaler(sqr, 5), 6), 7)
```

Какую результирующую функцию мы получим? Сначала к аргументу применяется… нет, не x_scaler(5) и не sqr, а самый внешний x_scaler. Затем shifter. затем внутренний x_scaler. Затем sqr. Покрутите это немного у себя в голове, к этому нужно немного «привыкнуть». Порядок такой: первым применяется самый внешний модификатор. Результат будет полностью аналогичен следующей функции:

```
def y(x):
  return sqr(((x * 7) + 6) * 5)
```

С тем различием лишь, что мы фактически создали эту функцию вручную по кусочкам. Теперь давайте для закрепления попробуем написать y = 5 * sin(0.3 * x + 30):

```
# самым последним должен применяться y_scaler(5)
y = y_scaler(math.sin, 5)
# предпоследним -- превращение угла в радианы
y = combine(math.radians, y)
# далее -- shifter(30)
y = shifter(y, 30)
# наконец -- x_scaler(0.3)
y = x_scaler(y, 0.3)
```

Очевидно, результат будет полностью аналогичен примеру без комбинаторов.

И последний финт функциями

Поднаторев в комбинировании, мы довольно просто напишем, например, модулятор одной функции при помощи другой:

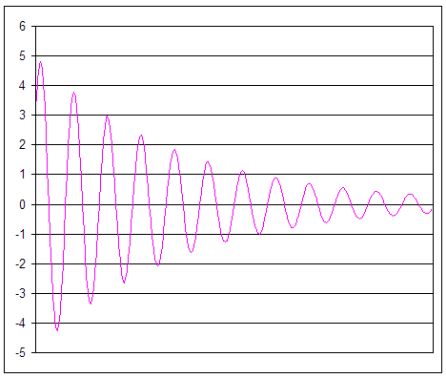
```
def modulate(mod, src):
  return lambda x: mod(x) * src(x)
```

Теперь мы можем описать затухающие гармонические колебания:

```
# y1 = 5 * sin(15 * x + 30) -- исходная функция
```

```
y1 = \
  x_scaler(
    shifter(
      combine(
        math.radians,
        y_scaler(
          math.sin,
          5)),
      30),
  15)

# y2 = exp(-0.01 * x) -- модулирующая функция
y2 = x_scaler(math.exp, -0.01)

y = modulate(y2, y1)

print [ y(x) for x in range(0, 180) ]
```

По-моему, неплохо:



На этом, пожалуй, пост закончу. В следующий раз будут списки, техника map-reduce и list comprehensions как синтаксический сахар для данной техники и **как это все помогает нам более ясно выражать свои мысли в коде**.

Код к этому посту: 1 2 3 4

Продолжаю свое небольшое введение в функциональное программирование. Речь пойдет о списках и методах их обработки в функциональном стиле.

Итак, список. Почему ему придается такое большое значение в мире ФП? Ответ на этот вопрос лежит в концептуальной основе языка Lisp. Список (в том или ином виде) является необходимой сематической единицей языка программирования. Без наличия списка мы не сможем получить произвольное количество единиц информации в программе. С другой стороны, добавление только списков позволяет нам реализовать сколь угодно сложные — рекурсивные, даже бесконечные (при наличии в языке поддержки ленивых вычислений) — структуры данных. Список + простые типы данных — тот необходимый минимум, который необходим любому языку программирования. Все остальные сложные типы данных — словари, деревья, графы и т.д. — можно реализовать при помощи списка.

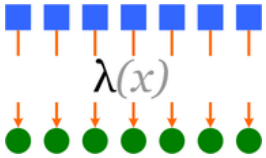
Так решили создатели Lisp… и сделали язык, в котором программы представляют собой список :) Да-да, любая программа на Lisp — суть просто список. Вызов функции — список, в котором первым идет имя функции, а следом — значения аргументов. Определение функции — список, в котором первым идет ключевое слово defun, затем имя функции, затем вложенный список с именами аргументов, затем список операторов. И так далее. На первый взгляд, это кажется довольно глупым и неудобным — многие слышали упреки в сторону Lisp за невероятное количество скобочек (списки там ограничиваются скобочками). Но на второй взгляд… если у программы такая упрощенная синтаксическая структура, то мы вполне можем модифицировать программу непосредственно в рантайме.

И для этого у Lisp есть механизм макросов — функций, результат выполнения которых заново выполняется (наподобие eval в динамических языках программирования, только гораздо гибче). Благодаря механизму макросов Lisp можно изменить до неузнаваемости, можно ввести новый синтаксис и использовать его. Новые операторы (хотели когда-нибудь более удобную форму цикла for? посмотрите на великолепный, гибкий макрос for в Common Lisp — это, не побоюсь сказать, цветок в грубом мире циклов for обычных языков программирования). Своя объектная система, синтаксически встроенная в язык (посмотрите на CLOS — это тоже просто набор макросов, а смотрится в языке как влитая). Вот такая вот гибкость. Хотя, конечно, нужен редактор с подсветкой скобочек — обязательно :)

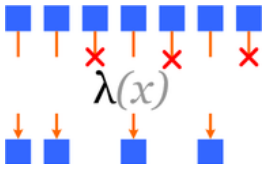
Теперь вернемся с Lisp к обычным, императивным языкам программирования — что здесь для нас список? Обработка списков (массивов, словарей) также составляет львиную долю программ на Python. Это и обработка выборки данных из БД, и расчет функции для построения, и обработка списка файлов в файловой системе, и обработка списка строк в файле, а также многое, многое другое.

В таких языках мы обычно обрабатываем списки при помощи разного рода циклов — for, while, do... while. Это как бы не является проблемой, но цикл сам по себе не семантичен. Т.е. он не говорит, что конкретно делается со списком. Нам приходится читать код тела цикла и разбирать, что же он делает. ФП в лице Lisp предлагает нам более изящные методы работы со списком (сюда не входят распространенные операции модификации списка — сортировка, обращение, конкатенация и т.п.):

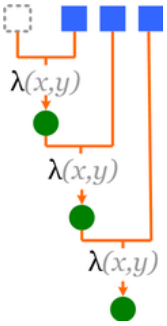
— отображение (map) — к каждому элементу списка применяется некоторая функция, результат которой подставляется вместо этого элемента:



— фильтрация (filter) — каждый элемент списка проверяется на соответствие функции-предикату, и если элемент не соответствует, он выбрасывается из списка:



— свертка (reduce) — здесь чуть посложнее. Процесс таков: берется некоторое базовое значение и первый элемент списка и к ним применяется функция-редуктор; затем берется результат действия этой функции и второй элемент списка и к ним снова применяется функция-редуктор; затем снова берется возвращаемое значение функции и третий элемент списка — процесс повторяется до тех пор, пока не закончится список. Результатом свертки будет одно значение. Например, таким образом можно реализовать суммирование всех элементов. Или что-нибудь посложней (например, интерполяцию, или обращение списка). Визуально это можно представить так:



Обычно к этим трем функциям сводится большинство проблем, связанных с обработкой списков. Но не всегда. Бывает, например левосторонняя и правосторонняя свертка. Бывает необходимость не в фильтрации списка, а в разделении его по некоторому признаку на две части. Да мало ли чего. Суть в том, что есть некая функция, которая на вход принимает список и на выходе выдает либо список, либо какое-то простое значение, не модифицируя при этом исходный список.

В Python вышеописанные функции присутствуют как в виде одноименных функций, так и в виде синтаксического сахара под странным названием list comprehension (не возьмусь корректно перевести, что-то вроде постижение списка).

List comprehensions (LC)

Простейший синтаксис LC таков:

```
[ expression(a) for a in x ]
```

где x — список, a — элемент списка, а expression(a) — некоторое выражение, в котором обычно участвует a. LC — это выражение, и его результатом является список. В смысловом плане вышеописанное LC соответствует функции map следующего вида:

```
map(lambda a: expression(a), x)
```

Далее, перед самой последней квадратной скобочкой может стоять еще ветка if:

```
[ expression(a) for a in x if condition(a) ]
```

Как вы догадались, это аналог filter. При помощи функций мы можем переписать это выражение следующим образом:

```
map(lambda a: expression(a),
  filter(lambda a: condition(a), x))
```

Для reduce синтаксического аналога нет, т.к. основная, первичная цель LC — конструирование списков. Стоит отметить еще один интересный момент: функция map может принимать несколько списков. В этом случае каждый раз при вызове функции-преобразователя ей будет передаваться несколько аргументов: первый аргумент будет значением текущего элемента из первого списка, второй аргумент — значением текущего элемента второго списка и т.д.:

```
map(
  lambda a1, a2, a3: a1 + a2 + a3,
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9])
```

В LC присутствует, казалось бы, похожая конструкция:

```
[ expression(a1, a2) for a1 in x1, for a2 in x2 ]
```

Но, увы, это не то же самое. Результатом действия данной конструкции будет не слишком часто применяемое на практике декартово произведение списков. Для примера:

```
[ a1 + a2 for a1 in [ 'a', 'b', 'c'] for a2 in ['x',
'y'] ]

    => ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
```

На практике, LC удобно применять для простых, невложенных операций, вроде получения квадратов чисел от 1 до 10. В иных случаях (см. сложный пример ниже) лучше использовать функции.

Простые примеры

Код ко всем примерам данного поста смотрите ниже. Давайте возьмем список следующего вида:

```
x = [ 2, 3, 4, 5, 7, 5 ]
```

Начнем с чего-нибудь простого; например, возведем все элементы списка в квадрат:

```
map(lambda a: a ** 2, x)

# то же самое, но при помощи LC
[ a ** 2 for a in x ]

    => [4, 9, 16, 25, 49, 25]
```

Теперь применим фильтрацию — отсеем все четные числа:

```
filter(lambda a: a % 2 == 1, x)

# то же самое, но при помощи LC
[ a for a in x if a % 2 == 1 ]

    => [3, 5, 7, 5]
```

Теперь скомбинируем — выведем нечетные квадраты чисел списка:

```
filter(lambda a: a % 2 == 1,
  map(lambda a: a ** 2,
    x))
# то же самое, но при помощи LC
[ a ** 2 for a in x if a ** 2 % 2 == 1 ]

    => [9, 25, 49, 25]
```

Как видите, в первом случае сначала производим отображение списка, а затем — фильтрацию результата. Теперь поиграем с reduce. Для начала выведем сумму всех чисел списка:

```
reduce(lambda a, b: a + b, x, 0)

    => 26
```

Первый параметр, как вы уже поняли, это функция-редуктор (в данном случае, сумматор). Второй параметр — это наш список, и третий — начальное значение, или инициализатор. Чтобы показать важность правильного выбора инициализатора, приведем тот же пример, но для умножения:

```
reduce(lambda a, b: a * b, x, 0)

    => 0
```

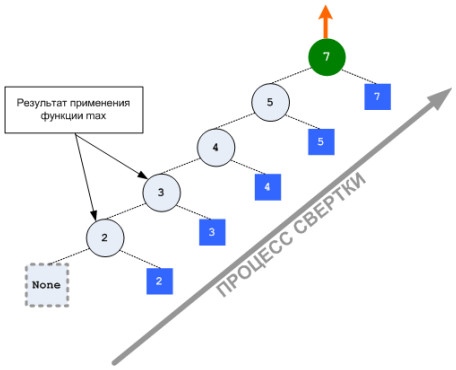
Здесь мы получили 0. И ведь правильно: получается, выполняется следующее выражение: ((((((0 * 2) * 3) * 4) * 5) * 7) * 5). Исправим этот пример, установив значение инициализатора в единицу:

```
reduce(lambda a, b: a * b, x, 1)

    => 4200
```

Теперь мы получим корректное значение. Теперь попробуем получить максимальное значение из списка:

```
reduce(lambda a, b: max(a, b), x)

    => 7
```

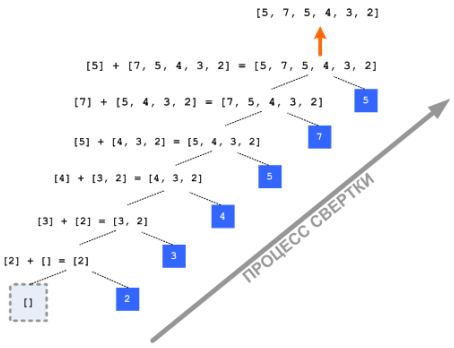
Здесь уже нет инициализатора. Когда инициализатор не указан, reduce подставляет на его место None. Работу этого кода легче всего пояснить визуально:



И наконец, возьмем задачу обращения списка посредством свертки. Напомню, у нас имеется список чисел 2, 3, 4, 5, 7, 5. Обращенный список будет таков: 5, 7, 5, 4, 3, 2. Давайте расставим скобочки, чтобы увидеть, какую операцию нам нужно будет применить в функции-редукторе: (5, (7, (5, (4, (3, (2)). Очевидно, что это операция добавления каждого нового элемента списка в начало результата с предыдущего шага свертки. Инициализатор же должен быть пустым списком: (5, (7, (5, (4, (3, (2, [])). Теперь запишем конкретный код:

```
reduce(lambda a, b: [ b ] + a, x, [])

    => [5, 7, 5, 4, 3, 2]
```

Еще раз, для понятности отобразим визуально:



В этом посте я рассмотрел только простые, «нежизненные» примеры работы со списками, но у меня в запасе есть пример пожизнеспособней, и я его выложу в ближайшее время в новом посте, вместе с некоторыми соображениями по поводу производительности, применимости, о том, какое это находит отражение в других областях компьютерной науки, и вообще. Надеюсь, это будет кому-то интересно. За сим хочу откланяться, спасибо за внимание.

Баги IE.Наличие или отсутствие hasLayout

Panya

Этим постом я планирую начать серию статей о багах IE и возможных вариантах их исправления. Цикл статей решил написать в первую очередь для себя, дабы как-то систематизировать и сохранить свои знания, но я надеюсь, что он будет полезен как новичкам так и опытным верстальщикам. Первым постом хочу затронуть одно из фундаментальных понятий при исправлении багов IE — hasLayout.

Что такое hasLayout?

В движке рендеринга IE каждый элемент, по сути, не отвечает за свое расположение. Расположение элемента зависит лишь от его положения в исходном коде и расположения в нормальном потоке. При этом, все дочерние элементы некоторго блока-контейнера располагаются, на самом деле, не относительно своего непосредственного родителя, а в зависимости от наиболее близкого предка имеющего «layout».

hasLayout это проприетарное свойство IE, которое определяет имеет ли элемент «layout» т.е. то, как элемент располагается в потоке, его размеры, позиционирование, реакцию на события и влияние на другие элементы.

HTML элементы, которые по умолчанию имеют «layout» (hasLayout = true): <html>, <body>, <table>, <tr>, <td>, <th>, <img>, <input>, <button>, <textarea>, <select>, <fieldset>, <legend>, <hr>, <iframe>, <embed>, <object>, <applet>, <marquee>.

Свойство hasLayout в IE нельзя напрямую установить, но на его наличие можно косвенно влиять различными значениями некоторых css свойств. Следующие значения перечисленных свойств дают элементу «layout» (hasLayout = true):

— position: absolute
— float: left или right
— height, width: любое значение кроме auto
— display: inline-block
— zoom: любое значение кроме normal (**невалидное свойство**)
— writing-mode: tb-rl
— overflow, overflow-x, overflow-y: auto|scroll|hidden (только в IE7)
— position: fixed (только в IE7)
— min-width, min-height: любое значение (только в IE7)
— max-width, max-height: любое значение кроме none (только в IE7)

Чтобы скинуть hasLayout нужно указать значение отличное от перечисленных выше (например: width: auto или float: none).

Более подробно о том, что такое hasLayout читайте в замечательной статье On having layout — the concept of hasLayout in IE/Win (перевод).

Как и зачем устанавливать hasLayout

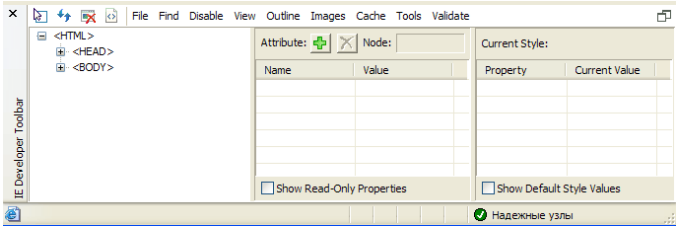
Придание элементам hasLayout может исправить множество багов IE. Большинство багов, связанных с неправильным позиционированием, отображением или измерением связаны с наличием или отсутствием у элемента hasLayout.

Чтобы успешно исправлять эти баги прежде всего нужно узнать включен ли у элемента hasLayout. Это можно проверить аналитическим или опытным путем.

Первый подразумевает просмотр исходного кода стилей, применяемых к проблемному элементу, и сканирование их на наличие свойств-триггеров, перечисленных выше. Если у элемента, к примеру, явно задана высота (свойство height имеет значение отличное от auto) то свойство hasLayout установлено (его значение true). Если же свойств-триггеров не найдено, либо они имеют значения, которые сбрасывают hasLayout и элемент не относится к списку элементов имеющих «layout» по умолчанию, то свойство hasLayout не установлено.

Второй способ заключается в непосредственном просмотре значения свойства hasLayout элемента с помощью DOM инспектора в плагине Internet Explorer Developer Toolbar.

После того как вы установите плагин, на панели IE появится кнопка , при нажатии на которую откроется панель плагина:



Также, открыть панель можно через меню View в IE6 (View → Explorer Bar → IE Developer Toolbar) или через меню Tools в IE7 (Tools → Toolbars → Explorer Bar

→ IE Developer Toolbar).

Для того чтобы выбрать элемент, у которого мы хотим проверить наличие свойства hasLayout, нужно нажать на кнопку  и мышкой выделить его (аналогично тому как выделяются элементы в Firebug после нажатия на кнопку «Inspect»). После этого в панели «Current Style» будут показаны все стили, применяемые к выделенному элементу браузером. Если среди них нет свойства hasLayout значит элемент не имеет «layout». Если же среди них оно есть и его значение равно -1 (true), значит свойство hasLayout установлено, и элемент имеет «layout».

После того, как мы узнали как проверить включено ли свойство hasLayout самое время научиться правильно его устанавливать. Все перечисленные выше свойства-триггеры так или иначе будут влиять на отображение элемента. Если, к примеру, мы укажем элементу float: left; то, кроме необходимого нам эффекта (установки hasLayout), элемент будет вести себя в соответствии прямому назначению этого свойства (будет вырван из нормального потока и его будут обтекать другие элементы). Хорошо, если это совпадает с нашей задумкой, но чаще всего нам нужно просто включить элементу hasLayout.

Поэтому нам нужно такое свойство-триггер, которое, в идеале, никак прямо не влияет на поведение элемента. Таких свойств немного, в некоторых ситуациях таким свойством может быть height со значением 1% или display: inline-block, но все они имеют ограничения. Если использовать height: 1%, то в случае явно установленной высоты родительского блока, высота элемента, естественно, будет вычисляться относительно выставленного значения. На мой взгляд, наиболее безопасным можно считать свойство zoom с установленным значением 1 (масштаб 100%). Но, так как это свойство невалидно, лучше всего использовать его в отдельном файле стилей для IE, который подключать с помощью Conditional Comments. Все бы хорошо, но это свойство не поддерживается в IE < 6 так что, если вам необходима поддержка этих браузеров для них можно использовать display: inline-block или height: 1%.

Алгоритм поиска багов, которые можно исправить с помощью hasLayout таков:

1. Найти несоответствие в отображении конкретного участка страницы в IE.
2. Просмотреть кусок HTML кода этого участка с целью запоминания иерархии элементов.
3. Найти стили в CSS, которые применяются к элементам на этом участке.
4. Проанализировать все применяемые стили на наличие свойств-триггеров hasLayout как у проблемного элемента так и у его предков.
5. Поочередно применять zoom: 1; к элементам, у которых не установлен hasLayout, начиная от самого верхнего в HTML дереве, при этом, после каждого изменения CSS файла проверять результат в IE.

Если, после проведения всех этих действий, так и не удалось исправить изначальный баг значит дело не в hasLayout :)

Примеры: положительное влияние hasLayout

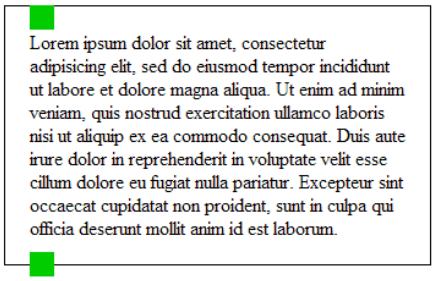
Предположим, что нам нужно расположить два блока внутри некоторого элемента. Один в левом верхнем углу, а другой в нижнем левом углу.
Код будет выглядеть примерно так:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//
EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.
dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="ru">
<head>
 <title>Пример 1. Неправильное позиционирование элемен-
тов с position:absolute.</title>
 <meta http-equiv="Content-Type" content="text/
html;charset=utf-8"/>
 <style type="text/css">
  * {
   margin:0;
   padding:0;
  }

  .zoom {
   zoom:1;
  }

  div.parent {
   position:relative;
   border:1px solid #000;
   padding:20px;
   margin:10px;
  }

  div.parent div {
   position:absolute;
   width:20px;
   height:20px;
   top:0;
   left:0;
   background:#0c0;
  }

  div.parent div.bottom {
   top:auto;
   bottom:0;
  }
 </style>
</head>
<body>
 <div class="parent">
  <p>Lorem ipsum dolor sit amet, consectetur adipisic-
ing elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua. Ut enim ad minim veniam, quis nos-
trud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehen-
derit in voluptate velit esse cillum dolore eu fugiat
nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim
id est laborum.</p>
```

```
  <div></div>
  <div class="bottom"></div>
 </div>
</body>
</html>
```

Пример 1

В нормальных браузерах все отображается как и задумывалось. Два зеленых квадратика, один из которых в левом верхнем углу элемента с рамкой, а другой в левом нижнем. Но, в IE6 все выглядит так:



Для того чтобы исправить это поведение нужно дать «layout» родительскому элементу (div.parent) для этого я добавлю ему специально заготовленный для этого класс «zoom». После этого все встанет на свои места:

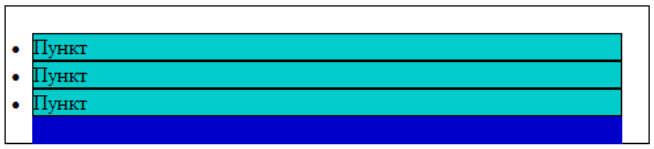
Пример 1 (баг исправлен)

Рассмотрим следующий пример. Его код:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//
EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.
dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="ru">
<head>
 <title>Пример 2. Неправильная высота списка</title>
 <meta http-equiv="Content-Type" content="text/
html;charset=utf-8"/>
 <style type="text/css">
  * {
   margin:0;
   padding:0;
  }

  .zoom {
   zoom:1;
  }

  div.parent {
   border:1px solid #000;
   padding:20px;
   margin:10px;
  }

  div.parent ul {
   background:#00c;
  }

  div.parent ul li {
   border:1px solid #000;
   background:#0cc;
  }
 </style>
</head>
<body>
 <div class="parent">
  <ul>
   <li>Пункт</li>
   <li>Пункт</li>
   <li>Пункт</li>
  </ul>
 </div>
</body>
</html>
```

Пример 2

В нормальных браузерах, список внутри блока с рамкой заканчивается ровно там, где заканчивается последний элемент списка. В IE 6 и 7 нижняя граница списка совпадает с нижней границей блока с рамкой (это четко видно из-за того, что я указал списку синий бекграунд):



Опять же, после включения hasLayout у родительского элемента (div.parent) баг исчезает:

Пример 2 (баг исправлен)

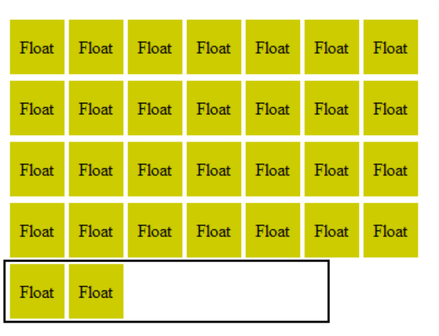
Следующий пример. Есть некоторый общий контейнер в котором несколько зафлоаченных элементов:

```
<!DOCTYPE html PUBLIC «-//W3C//DTD XHTML 1.0 Strict//
EN» «http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.
dtd»>
<html xmlns=»http://www.w3.org/1999/xhtml»
xml:lang=»ru»>
<head>
 <title>Пример 3. Выпадение зафлоаченных блоков из кон-
тейнера</title>
 <meta http-equiv="Content-Type" content="text/
html;charset=utf-8"/>
 <style type="text/css">
  * {
   margin:0;
   padding:0;
  }

  .zoom {
   zoom:1;
  }

  div.parent {
   border:2px solid #000;
   margin:10px 20%;
  }
  div.parent div.float {
```

```
    background:#cc0;
    float:left;
    margin:2px;
    text-align:center;
    width:50px;
    height:50px;
    line-height:50px;
  }
 </style>
</head>
<body>
 <div class=”parent”>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div class=”float”>Float</div>
  <div style=”clear:both;”></div>
 </div>
</body>
</html>
```

Пример 3

IE снова глючит:



После добавления класса zoom для родительского дива баг исчезает:

Пример 3 (баг исправлен)

Помимо исправления подобных багов, включение у элемента hasLayout полезно также для следующего:

— Создание нового контекста форматирования (Пример 4).
— В следствиии создания нового контекста форматирования запрет на обтекание плавающих элементов (Пример 5).
— Фильтры применяются только к элементам с установленным hasLayout (Пример 6).
— Исправление бага при котором display:inline-block применяется только к изначально строчным элементам. (Пример 7).

Негативное влияние hasLayout

Помимо положительного влияния, включение hasLayout может также негативно влиять на отображение. Чаще всего это выражается в несхлопывании вертикальных маржинов, появлении лишних отступов, пропадании частей элементов или «прыгании» элементов при изменении размеров окна или наведении курсора на ссылку. В таких случаях нужно либо отказаться от hasLayout, либо, если без него никак, попробовать обернуть проблемный элемент в другой, установить у оберточного элемента hasLayout а у исходного его сбросить.

Кроме этого, нужно учесть, что применение hasLayout плохо влияет на производительность.

Структуры данных в memcached/MemcacheDB

smira

Достаточно часто нам приходится хранить данные в memcached или MemcacheDB. Это могут быть относительно простые данные, например, закэшированные выборки из базы данных, а иногда необходимо хранить и обрабатывать более сложные структуры данных, которые обновляются одновременно из нескольких процессов, обеспечивать быстрое чтение данных и т.п. Реализация таких структур данных уже не укладывается в комбинацию команд memcached get/set. В данной статье будут описаны способы хранения некоторых структур данных в memcached с примерами кода и описанием основных идей.

Memcached и MemcacheDB в данной статье рассматриваются вместе, потому что имеют общий интерфейс доступа и логика работы большей части структур данных будет одинаковой, далее будем называть их просто "memcached". Зачем нам нужно хранить структуры данных в memcached? Чаще всего для распределенного доступа к данным из разных процессов, с разных серверов и т.п. А иногда для решения задачи хранения данных достаточно интерфейса, предоставляемого MemcacheDB, и необходимость в использовании СУБД отпадает.

Иногда проект разрабатывается изначально для нераспределенного случая (работа в рамках одного сервера), однако предполагая будущую необходимость масштабирования, лучше использовать сразу такие алгоритмы и структуры данных, которые могут обеспечить легкое масштабирование. Например, даже если данные будут храниться просто в памяти процесса, но интерфейс к доступа к ним повторяет семантику memcached, то при переходе к распределенной и масштабируемой архитектуре достаточно будет заменить обращения к внутреннему хранилищу на обращения к серверу (или кластеру серверов) memcached.

Всего будет рассмотрено 6 структур данных:
— блокировка;
— счетчик;
— счетчик посетителей;
— лог событий;
— массив;
— таблица.

С первой по третью в этой части, а с четвертой по шестую — в следующей. Примеры кода будут написаны на Python, в них будет использоваться следующий интерфейс доступа к memcached:

```
class Memcache(object):
    def get(self, key):
        """
        Получить значение ключа.

        @param key: ключ
        @type key: C{str}
        @return: значение ключа
        @raises KeyError: ключ отсутствует
        """

    def add(self, key, value, expire=0):
        """
        Установить значение ключа, если он еще не существует.

        @param key: ключ
        @type key: C{str}
        @param value: значение ключа
        @param expire: "срок годности" ключа в секундах (0 — бесконечно)
        @type expire: C{int}
        @raises KeyError: ключ уже существует
        """

    def incr(self, key, value=1):
        """
        Увеличить на единицу значение ключа.

        @param key: ключ
        @type key: C{str}
        @param value: инкремент
        @type value: C{int}
        @return: новое значение ключа (после увеличения)
        @rtype: C{int}
        @raises KeyError: ключ не существует
        """

    def append(self, key, value):
        """
        Добавить суффикс к значению существующего ключа.

        @param key: ключ
        @type key: C{str}
        @param value: суффикс
        @raises KeyError: ключ не существует
        """

    def delete(self, key):
        """
        Удалить ключ.
```

```
        @param key: ключ
        @type key: C{str}
        """
```

Все наши структуры данных будут наследниками базового класса следующего вида:

```
class MemcacheObject(object):
    def __init__(self, mc):
        """
        Конструктор.

        @param mc: драйвер memcached
        @type mc: L{Memcache}
        """
        self.mc = mc
```

Блокировка. Задача

Блокировка (mutex, или в данной ситуации скорее spinlock) — это не совсем структура данных, но она может быть использована как "кирпичик" при построении сложных структур данных в memcached. Блокировка используется для исключения одновременного доступа к некоторым ключам, возможности блокировки отсутствуют в API memcached, поэтому она эмулируется с помощью других операций.

Итак, блокировка определяется своим именем, работает распределённо (то есть из любого процесса, имеющего доступ к memcached), имеет автоматическое "умирание" (на случай, если процесс, поставивший блокировку, не сможет её снять, например, по причине аварийного завершения).

Операции над блокировкой:
— попробовать заблокироваться (try-lock);
— разблокировать (unlock).

Решение

```
class MCLock(MemcacheObject):
    def __init__(self, mc, name, timeout=5):
        """
        Конструктор.

        @param name: имя блокировки
        @type name: C{str}
        @param timeout: таймаут аварийного снятия блокировки (секунды)
        @type timeout: C{int}
        """
        super(MCLock, self).__init__(mc)
        self.key = 'lock_' + name
        self.locked = False
        self.timeout = timeout

    def try_lock(self):
        """
        Попробовать заблокироваться.

        @return: удалось ли заблокироваться?
        @rtype: C{bool}
        """
        assert not self.locked
        try:
            self.mc.add(self.key, 1, self.timeout)
        except KeyError:
            return False

        self.locked = True
        return True

    def unlock(self):
        """
        Снять блокировку.
        """
        assert self.locked
        self.mc.delete(self.key)
        self.locked = False
```

Обсуждение

Корректность работы блокировки основывается на операции `add`, которая гарантирует что ровно один процесс сможет "первым" установить значение ключа, все остальные процессы получат ошибку. Приведенный выше класс может быть обернут более удобно, например, для with-обработчика Python. Более подробно вопрос блокировок обсуждался в посте про одновременное перестроение кэшей.

Атомарный счетчик. Задача

Необходимо вычислять количество событий, которые происходят распределённо на разных серверах, также получать текущее значение счетчика. При этом счетчик не должен "терять" события и начислять "лишние" значения.

Тип данных: целое число.

Начальное значение: ноль.

Операции:
— увеличить значение счетчика на указанное значение;
— получить текущее значение счетчика.

Решение

```
class MCCounter(MemcacheObject):
    def __init__(self, mc, name):
        """
        Конструктор.
        @param name: имя счетчика
        @type name: C{str}
        """
        super(MCCounter, self).__init__(mc)
        self.key = 'counter' + name

    def increment(self, value=1):
        """
        Увеличить значение счетчика на указанное значение.
        @param value: инкремент
```

```
        @type value: C{int}
        """
        while True:
            try:
                self.mc.incr(self.key, value)
                return
            except KeyError:
                pass

            try:
                self.mc.add(self.key, value, 0)
                return
            except KeyError:
                pass

    def value(self):
        """
        Получить значение счетчика.

        @return: текущее значение счетчика
        @rtype: C{int}
        """
        try:
            return self.mc.get(self.key)
        except KeyError:
            return 0
```

Обсуждение

Реализация счетчика достаточно очевидная, самый сложный момент — это "вечный" цикл в методе increment. Он необходим для корректной инициализации значения счетчика в условиях конкурентного доступа к нему. Если операция incr завершилась ошибкой отсутствия ключа, нам необходимо создать ключ счетчика, но этот код могут одновременно выполнять несколько процессов, тогда одному из них удастся add, а второй получит ошибку и инкрементирует уже созданный счетчик с помощью incr. Зацикливание невозможно, т.к. одна из операций incr или add должна быть успешной: после создания ключа в memcached операция incr будет успешной всё время существования ключа.

Как быть с ситуацией, когда ключ со счетчиком из memcached пропадет? Возможны две ситуации, когда ключ исчезнет:
— memcached не хватает памяти для размещения других ключей;
— процесс memcached аварийно завершился (сервер "упал").

(В случае MemcacheDB, настроенной репликации и т.п. падение сервера не должно приводить к потере ключей.) В первом случае надо лишь правильно обслуживать memcached — памяти должно хватать для счетчиков, иначе мы начинаем терять значения, а это неприемлемо в данной ситуации (но совершенно нормально, например, для хранения кэшей в memcached). Второй случай всегда возможен, если такая ситуация часто возникает, можно дублировать счетчики в нескольких memcached-серверах.

Счетчик посетителей. Задача

Необходимо вычислить количество уникальных обращений к сайту в течение некоторого периода T. Например, T может быть равно 5 минутам. Пусть мы можем сказать, когда было последнее обращение данного посетителя к сайту — более T минут назад или менее (то есть является ли оно уникальным в течение периода T).

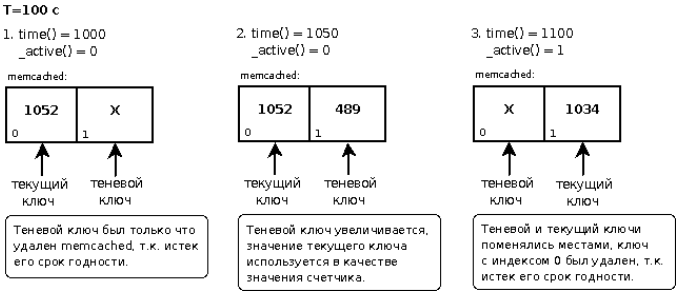
Операции над счетчиком:
— увеличить значение счетчика на единицу (обращение уникального в течение периода T посетителя);
— получить текущее число посетителей.

Решение

```
def time():
    """
    Текущее время в секундах с любого момента времени
(например, UNIX Epoch).

    @return: текущее время в секундах
    @rtype: C{int}
    """

class MCVisitorCounter(MemcacheObject):
    def __init__(self, mc, name, T):
        """
        Конструктор.

        @param name: имя счетчика
        @type name: C{str}
        @param T: период счетчика, секунды
        @type T: C{int}
        """
        super(MCVisitorCounter, self).__init__(mc)
        self.keys = ('counter' + name + '_0',
'counter' + name + '_1')

    def _active(self):
        """
        Получить индекс текущего счетчика.

        @return: 0 или 1
        """
        return 0 if (time() % (2*T)) < T else 1

    def increment(self):
        """
        Увеличить значение счетчика.
        """
        active = self._active()
        while True:
            try:
                self.mc.incr(self.keys[1-active], 1)
                return
            except KeyError:
                pass

            try:
                self.mc.add(self.keys[1-active], 1,
```

```
2*T)   # строка 43
                return
            except KeyError:
                pass

    def value(self):
        """
        Получить значение счетчика.

        @return: текущее значение счетчика
        @rtype: C{int}
        """
        try:
            return self.mc.get(self.keys[self._
active()])
        except KeyError:
            return 0
```

Обсуждение



Основная структура работы со счетчиком в memcached в данной задаче повторяет решение задачи атомарного счетчика. Базовая идея — использование теневого и текущего счетчика: теневой счетчик получает обновления (увеличивается), а текущий используется для получения значения числа посетителей (его значение было накоплено, когда она был теневым). Каждый период T текущий и теневой счетчик меняются местами. Ключ теневого счетчика создается со сроком годности 2*T секунд, то есть его время жизни — T секунд, пока он был теневым (и увеличивался), и T секунд, пока его значение использовалось для чтения. К моменту того, как этот ключ снова станет теневым счетчиком, он должен быть уничтожен memcached, т.к. срок его годности истек. Идея теневого и текущего счетчика напоминает, например, двойную буферизацию при выводе на экран в играх.

Необходимо обратить внимание на функцию _active(): она реализована именно через вычисление остатка от деления текущего времени в секундах, а не, например, через периодическое (раз в T секунд) смену значения active, т.к. если время на всех серверах синхронизировано с разумной точностью, то и результат функции _active() будет на всех серверах одинаковым, что важно для корректного распределенного функционирования данной структуры данных.

Описанный подход будет работать корректно только при достаточно большом количестве посетителей, когда временной интервал между очередной сменой значения функции _active() и обращением к функции increment() будет как можно меньше. То есть когда increment() вызывается часто, то есть когда посетителей достаточно много, например, 1000 человек при периоде T=3 минуты. Иначе создание и уничтожение ключей не будет синхронизировано с моментом смены значения _active() и начнут пропадать посетители, накопленные в теневом счетчике, который будет уничтожен в процессе накопления значений и создан заново. В этой ситуации его время жизни 2*T будет "сдвинуто" относительно моментов 0, T функции time() % (2*T). В силу того, что memcached рассматривает срок годности ключей дискретно относительно секунд (с точностью не более одной секунды), любой сдвиг времени жизни ключа может составлять 1,2,...,T-1 секунду (отсутствие сдвига — 0 секунд). Дискретность срока годности означает, что если ключ был создан в 25 секунд 31 миллисекунду со сроком годности 10 секунд, то он будет уничтожен на 35-й (или 36-й) секунде 0 миллисекунд (а не 31-й миллисекунде). Для компенсация сдвига в целое число секунд необходимо исправить строчку 43 примера следующим образом:

```
self.mc.add(self.keys[1-active], 1, 2*T – time() % T)
```

Значение счетчика посетителей не изменяется в течение периода времени T, для более приятного отображения можно к значению счетчика при выводе добавить нормально распределенную величину с небольшой дисперсией и математическим ожиданием около 5% от значения счетчика.

Немного более сложный вариант такого счетчика (с интерполяцией во времени), но с точно такой же идеей был описан в посте про атомарность операции и счетчики в memcached.

Лог событий. Задача

Задача этой структуры данных — хранение событий, произошедших в распределенной системе за последние T секунд. Каждое событие имеет момент времени, когда оно произошло, остальное содержимое события определяется логикой приложения.

Операции над логом событий:

— добавить сообщение в лог событий (должна быть максимально быстрой);
— получить события, произошедшие в период времени от Tmin до Tmax (должна быть эффективной, но вызывается реже, чем добавление).

Решение

```
def time():
    """
    Текущее время в секундах с любого момента времени
(например, UNIX Epoch).

    @return: текущее время в секундах
    @rtype: C{int}
```

```
    """

class Event:
    """
    Событие, помещаемое в лог событий.
    """

    def when(self):
        """
        Момент времени, когда произошло событие (секун-
ды).
        """

    def serialize(self):
        """
        Сериализовать событие.

        @return: сериализованное представление
        @rtype: C{str}
        """

    @static
    def deserialize(serialized):
        """
        Десериализовать набор событий.

        @param serialized: сериализованное представле-
ние одного
                           или нескольких событий
        @type serialized: C{str}
        @return: массив десериализованных событий
        @rtype: C{list(Event)}
        """

class MCEventLog(MemcacheObject):
    def __init__(self, mc, name, timeChunk=10,
numChunks=10):
        """
        Конструктор.
        @param name: имя лога событий
        @type name: C{str}
        @param timeChunk: емкость одного ключа лога в
секундах
        @type timeChunk: C{int}
        @param numChunks: число выделяемых ключей под
лог
        @type numChunks: C{int}
        """
        super(MCEventLog, self).__init__(mc)
        self.keyTemplate = 'messagelog' + name + '_%d';
        self.timeChunk = timeChunk
        self.numChunks = numChunks

    def put(self, event):
        """
        Поместить событие в лог.

        @param event: событие
        @type event: L{Event}
        """
        serialized = event.serialize()
        key = self.keyTemplate % (event.when() // self.
timeChunk % self.numChunks)
        while True:
            try:
                self.mc.append(key, serialized)
                return
            except KeyError:
                pass

            try:
                self.mc.add(key, serialized, self.
timeChunk * (self.numChunks-1))
                return
            except KeyError:
                pass

    def fetch(self, first=None, last=None):
        """
        Получить события из лога за указанный период
(или все события).

        @param first: минимальное время возвращаемого
сообщения
        @type first: C{int}
        @param last: максимальное время возвращаемого
сообщения
        @type last: C{int}
        @return: массив событий
        @rtype: C{list(Event)}
        """
        if last is None or last > time():
            last = time()

        if first is None or last < first or
           (last-first) > self.timeChunk * (self.
numChunks-1):
            first = time() - self.timeChunk * (self.
numChunks-1)

        firstKey = first / self.timeChunk % self.
numChunks
        lastKey = last / self.timeChunk % self.
numChunks

        if firstKey < lastKey:
            keyRange = range(firstKey, lastKey+1)
        else:
            keyRange = range(firstKey, self.numChunks) +
range(0, lastKey+1)

        keys = [self.keyTemplate % n for n in keyRange]
        result = []
        for key in keys:
            try:
                events = Event.deserialize(self.
mc.get(key))
            except KeyError:
                continue

            result.extend(filter(lambda e: e.when() >=
first and
                                                    e.
when() <= last, l))
        return result
```

Обсуждение

Основная идея лога событий — кольцевой буфер, состоящий из numChunks ключей в memcached. Каждый ключ активен (то есть дополняется значениями) в течение timeChunk секунд, после чего активным становится следующий ключ (если активным был последний ключ, эта роль переходит к первому ключу). Полный цикл буфера, т.е. период времени между двумя использованиями одного ключа составляет numChunks * timeChunk секунд, а время жизни каждого ключа — (numChunks - 1) * timeChunk секунд, таким образом при любом сдвиге времени создания ключа по модулю timeChunk к моменту времени следующего использования ключ гарантированно будет уничтожен. Таким образом, ёмкость лога событий (или период времени, за который сохраняются события) составляет (numChunks - 1) * timeChunk секунд. Такое разбиение лога на ключи позволяет при получении событий из лога вынимать лишь те ключи, которые соответствуют интересному нам временному отрезку.

Выбор параметров timeChunk и numChunks зависит от применения лога событий: сначала определяется желаемый срок хранения событий, затем по частоте событий выбирается такое значение timeChunk, чтобы размер каждого ключа лога событий был относительно небольшим (например, 10-20Кб). Из этих соображений можно найти значение и второго параметра, numChunks.

В примере используется некоторый класс Event, который обладает единственным интересным для нас свойством — временем, когда произошло событие. В методе put лога событий предполагается, что событие event, переданное в качестве параметра, произошло "недавно", то есть с момента event.when() прошло не более чем (numChunks - 1) * timeChunk секунд (емкость лога). При работе put вычисляется ключ, в который должна быть помещена информация о событии, в соответствие с его временной меткой. После этого с помощью уже знакомой по предыдущим примерам техники ключ либо создается, либо к значению уже существующего ключа дописывается сериализованное представление события.

Метод fetch вычисляет потенциальный набор ключей лога, в которых могут находиться события, произошедшие во временной интервал от first до last. Если временные рамки не заданы, last считается равным текущему моменту времени, а first — моменту времени, отстоящему от текущего на емкость лога. Набор ключей вычисляется с учетом кольцевой структуры метода, после чего выбираются соответствующие ключи, десериализуются последовательно записанные в них события и проводится дополнительная фильтрация на попадание в отрезок [first, last].

Приведенная выше сигнатура метода позволяет последовательными обращениями выводить новые события из лога:

1. Первый раз вызывается events = fetch(). Вычисляется lastSeen как max(events.when()).
2. Все последующие обращения выглядят следующим образом: events = fetch(first=lastSeen), при этом lastSeen каждый раз перевычисляется.

Массив. Задача 1

В массиве хранится список значений произвольного типа, относительно редко происходит обновление списка, гораздо чаще происходит получение списка целиком.

Операции над массивом:

— изменить массив (редкая операция);
— получить массив целиком (частая операция).

Решение 1

```
def serializeArray(array):
    """
    Сериализовать массив в бинарное представление.
    """

def deserializeArray(str):
    """
    Десериализовать массив из бинарного представления.
    """

class MCArray1(MemcacheObject):
    def __init__(self, mc, name):
        """
        Конструктор.

        @param name: имя массива
        @type name: C{str}
        """
        super(MCArray1, self).__init__(mc)
        self.lock = MCLock(name)
        self.key = 'array' + name

    def fetch(self):
        """
        Получить текущее значение массива.

        @return: массив
        @rtype: C{list}
        """
        try:
            return deserializeArray(self.mc.get(self.
key))
        except KeyError:
            return []

    def change(self, add_elems=[], delete_elems=[]):
        """
        Изменить значение массива, добавив или удалив
из него
        элементы.

        @param add_elems: элементы, которые надо доба-
вить
        @type add_elems: C{list}
        @param delete_elems: элементы, которые надо
```

```
удалить
        @type delete_elems: C{list}
        """
        while not self.lock.try_lock():
            pass

        try:
            try:
                array = deserializeArray(self.
mc.get(self.key))
            except KeyError:
                array = []
            array = filter(lambda e: e not in delete_
elems, array) + add_elems
            self.mc.set(self.key,
serializeArray(array), 0)
        finally:
            self.lock.unlock()
```

Обсуждение 1

Приведенный выше способ решения на самом деле не имеет никакого отношения к массивам, а может быть применен для любой структуры данных. Он основан на модели reader-writer, когда есть много читателей и относительно мало писателей. Читатели в любой момент с помощью метода fetch получают содержимое массива, при этом важно, что "писатель" change записывает содержимое одной командой memcached, то есть в силу внутренней атомарности операций get и set в memcached и несмотря на отсутствие синхронизации между методами fetch и change, результат fetch всегда будет консистентным: это будет значение до или после очередного изменения. Писатели блокируются от одновременного изменения массива с помощью блокировки MCLock, описанной выше.

В данной ситуации можно было бы избежать использования блокировки и воспользоваться командами gets, cas и add из протокола memcached для того, чтобы гарантировать атомарность изменений с помощью функции change.

Задача 2

Массив хранит список значений некоторого типа, часто происходит операция вида "добавить значение в массив". Относительно редко массив запрашивается целиком. Для простоты реализации в дальнейшем будет рассматриваться массив целых чисел, хотя для решения задачи тип данных не имеет существенного значения.

Операции над массивом:
— добавить значение в массив (частая операция);
— получить массив целиком.

Решение 2

```
def serializeInt(int):
    """
    Сериализовать целое число в бинарное представление
(str).
    """

def deserializeIntArray(str):
    """
    Десериализовать массив целых чисел из бинарного
представления.
    """

class MCArray2(MemcacheObject):
    def __init__(self, mc, name):
        """
        Конструктор.

        @param name: имя массива
        @type name: C{str}
        """
        super(MCArray2, self).__init__(mc)
        self.key = 'array' + name

    def fetch(self):
        """
        Получить текущее значение массива.

        @return: массив
        @rtype: C{list}
        """
        try:
            return deserializeIntArray(self.
mc.get(self.key))
        except KeyError:
            return []

    def add(self, element):
        """
        Добавить элемент в массив.

        @param element: элемент, который необходимо до-
бавить в массив
        @type element: C{int}
        """
        element = serializeInt(element)
        while True:
            try:
                self.mc.append(self.key, element)
            except KeyError:
                return

            try:
                self.mc.add(self.key, element, 0)
            except KeyError:
                return
```

Обсуждение 2

Эта реализация практически повторяет аналогичный код для лога событий, только упрощенный в силу наличия всего одного ключа. По сравнению с первым вариантом реализации типа данных "массив" уменьшилось число операций memcached, все изменяющие массив процессы могут выполняться без задержек (отсутствие блокировок). Как и в первом варианте, не проверяется наличие дубликатов при добавлении элемента в массив (может быть и

хорошо, и плохо, в зависимости от применения).

Возможны следующие улучшения (или расширения) описанного примера:
— использование нескольких ключей для хранения массива вместо одного, распределение элементов по ключам с использованием хэширования; такой вариант позволит ограничить размер каждого ключа, при условии что массив большой (содержит много элементов);
— реализация в том же стиле операции удаления элемента из массива, тогда массив можно представить
как последовательность операций "удалить" и "добавить", например сериализованное представление +1 +3 +4 -3 +5 будет после десериализации образовывать массив [1, 4, 5]; при этом как
операция добавления элемента, так и удаления, будет приводить к дописыванию байт в конец сериализованного представления (атомарная операция append).

Таблица. Задача

Необходимо хранить множество строк. Операции над множеством:
— проверка принадлежности строки множеству (самая частая операция);
— получение множества целиком, добавление элемента, удаление элемента — редкие операции.

Можно рассматривать данную структуру данных как таблицу, в которой осуществляется быстрый поиск нужной строки. Или как хэш, хранящийся в распределенной памяти.

Решение

```
def serializeArray(array):
    """
    Сериализовать массив в бинарное представление.
    """

def deserializeArray(str):
    """
    Десериализовать массив из бинарного представления.
    """

class MCTable(MemcacheObject):
    def __init__(self, mc, name):
        """
        Конструктор.

        @param name: имя таблицы
        @type name: C{str}
        """
        super(MCTable, self).__init__(mc)
        self.lock = MCLock(name)
        self.key = 'table' + name

    def has(self, key):
        """
        Проверка наличия ключа в таблице.

        @param key: ключ
        @type key: C{str}
        @rtype: C{bool}
        """
        try:
            self.mc.get(self.key + '_v_' + key)
            return True
        except KeyError:
            return False

    def fetch(self):
        """
        Получить целиком значение элементов таблицы.

        @return: значение таблицы
        @rtype: C{list(str)}
        """
        try:
            return deserializeArray(self.mc.get(self.
key + '_keys'))
        except KeyError:
            pass

    def add(self, key):
        """
        Добавить ключ в таблицу.

        @param key: ключ
        @type key: C{str}
        """
        while not self.lock.try_lock():
            pass

        try:
            try:
                array = deserializeArray(self.
mc.get(self.key + '_keys'))
            except KeyError:
                array = []
            if key not in array:
                array.append(key)
            self.mc.set(self.key + '_v_' + key, 1, 0)
            self.mc.set(self.key + '_keys',
serializeArray(array), 0)
        finally:
            self.lock.unlock()

    def delete(self, key):
        """
        Удалить ключ из таблицы.

        Реализация аналогична методу add().
        """
```

Обсуждение

Вообще говоря memcached представляет собой огромную хэш-таблицу, правда в ней отсутствует одна операция, которая необходима для нашей структуры данных: получение списка ключей. Поэтому реализация таблицы использует отдельные ключи для хранения каждого элемента таблицы, и отдельно еще один ключ для хранения списка всех её элементов.

Реализация хранения списка всех элементов фактически совпадает с реализацией "массива 1". Для сериализации доступа к списку всех элементов используется блокировка, при этом методы fetch и add не синхронизированы друг с другом, т.к. список всех элементов меняется атомарно и при чтении ключа мы всегда получим некоторое консистентное состояние.

Проверка наличия ключа в таблице выполняется максимально быстро: проверяется наличие соответствующего ключа в memcached. Любое изменение списка элементов всегда происходит одновременно и в ключе, хранящем весь список, и в отдельных ключах для каждого элемента (которые используются только для проверки).

На основе приведенной схемы можно реализовать полноценный хэш, когда для каждого элемента таблицы будет храниться связанное значение, это значение необходимо будет записывать только в отдельные ключи, соответствующие элементам, а список элементов не будет содержать значений.

Заключение

Итак, приведем список "приемов" или "трюков", описанных в данной статье:

— атомарность операций с помощью memcached (пара add/set и т.п.);
— блокировки;
— теневые ключи;
— кольцевой буфер с автоматическим "отмиранием" ключей;
— блокировки и модель reader-writer.

В статье не рассматривались вопросы оптимизации, специфичной для memcached, например, использование multi-get запросов. Это делалось сознательно, чтобы не перегружать исходный код и рассказ. Во многих ситуациях приведенные выше примеры следует рассматривать скорее как псевдокод, чем как пример идеальной реализации на Python.

Если Вы нашли ошибку, хотите предложить более ясное, более оптимальное решение поставленным задачам, хотите предложить реализацию для какой-то еще структуры данных — буду рад комментариям и критике.

Работа с объектами в JavaScript: теория и практика

depp

В этой статье я хочу по возможности полно и последовательно рассказать о том, что такое объект в JavaScript, каковы его возможности, какие взаимоотношения могут строиться между объектами и какие способы «родного» наследования из этого вытекают, как это все влияет на производительность и что вообще со всем этим делать. В статье не будет ни слова про: эмуляцию традиционной класс-объектной парадигмы, синтаксический сахар, обертки и фреймворки. Сложность материала будет нарастать от начала к концу статьи, так что для профи первые части могут показаться скучными и банальными, но дальше будет намного интереснее

Объекты в JavaScript

Во многих статьях встречается фраза «В JavaScript — всё объект». Технически это не совсем верно, однако производит должное впечатление на новичков :)

Действительно, многое в языке является объектом, и даже то, что объектом не является, может обладать некоторыми его возможностями.

Важно понимать, что слово «объект» употребляется здесь не в смысле «объект некоторого класса». Объект в JavaScript — это в первую очередь просто коллекция свойств (если кому проще, может называть это ассоциативным массивом или списком), состоящая из пар ключ-значение. Причем ключом может быть только строка (даже у элементов массива), а вот значением — любой тип данных из перечисленных ниже.

Итак, в JavaScript есть 6 базовых типов данных — это Undefined (обозначающий отсутствие значения), Null, Boolean (булев тип), String (строка), Number (число) и Object (объект). При этом первые 5 являются примитивными типами данных, а Object — нет. Кроме того, условно можно считать, что у типа Object есть «подтипы»: массив (Array), функция (Function), регулярное выражение (RegExp) и другие. Это несколько упрощенное описание, но на практике обычно достаточное.

Кроме того, примитивные типы String, Number и Boolean определенным образом связаны с непримитивными «подтипами» Object: String, Number и Boolean соответственно. Это означает, что строку 'Hello, world', например, можно создать и как примитивное значение, и как объект типа String. Если вкратце, то это сделано для того, чтобы программист мог и в работе с примитивными значениями использовать методы и свойства, как будто это объекты. А подробнее об этом можно будет прочитать в соответствующем разделе данной статьи.

Работа по ссылке

Ссылка — это средство доступа к объекту под различными именами. Работа с любыми объектами ведется исключительно по ссылке. Продемонстрируем это на примере:

```
test=function() {alert('Hello!')} //Создадим функцию
{alert('Hello!')} (а функция, как мы помним, является
полноправным объектом) и сделаем переменную test ссыл-
кой на нее
test_link=test; //test_link теперь тоже ссылается на
нашу функцию
test(); //Hello!
test_link(); //Hello!
```

Как мы видим, и первая ссылка, и вторая дают один и тот же результат. Необходимо осознать, что у нас нет никакой функции с именем test, и что переменная test не является какой-то «главной» или «основной» ссылкой, а «test_link» — второстепенной. Наша функция, как и любой другой объект — просто область в памяти, и все ссылки на эту область абсолютно равнозначны. Более того, объект может вообще не иметь ссылок — в таком случае он называется анонимным, и может быть использован только непосредственно сразу после создания (например, передан в функцию), иначе доступ к нему получить будет невозможно и в скором времени он будет уничтожен сборщиком мусора (garbage collection), который и занимается тем, что удаляет объекты без ссылок.

Посмотрим, почему так важно это понимать:

```
test={prop: 'sometext'} //Создаем объект со свойством
prop
test_link=test; //Создаем еще одну ссылку на этот объ-
ект

alert(test.prop); //sometext
alert(test_link.prop); //sometext

//Изменяем свойство объекта
test_link.prop='newtext';

alert(test.prop); //newtext
alert(test_link.prop); //newtext
/*Можно было бы сказать, что свойство изменилось и там
и тут - но это не так.
```

```
Объект-то один. Так что свойство изменилось в нем один раз, а ссылки просто продолжают указывать туда, куда и указывают. */

//Добавляем новое свойство и удаляем старое
test.new_prop='hello';
delete test.prop;

alert(test_link.prop); //undefined - такого свойства больше нет
alert(test_link.new_prop); //hello - что и следовало ожидать

//Удаляем ссылку
delete test;
alert(test.new_prop);
/*В этом месте скрипт выкинет ошибку, потому что test уже не существует, и test.new_prop не существует тем более */
alert(test_link.new_prop); //hello
/* а вот тут все в порядке, ведь мы удалили не сам объект, а лишь ссылку на него. Теперь на наш объект указывает единственная ссылка test_link */

//Создаем новый объект
test=test_link; //Сперва снова создадим ссылку test
test_link={prop: 'sometext'} //А вот и новый объект

alert(test_link.prop); //sometext
alert(test.prop); //undefined
/* Создание нового объекта разрывает ссылочную связь, и теперь test и test_link указывают на разные объекты. Фактически, это равносильно удалению ссылки test_link и созданию ее заново, но уже указывающей на другой объект */
alert(test.new_prop); //hello - теперь test содержит ссылку на наш самый первый объект
```

Такое поведение объектов часто вызывает массу вопросов у начинающих разработчиков, так что надеюсь данный текст внесет некоторую ясность. Если же мы хотим создать действительно новую, независимую копию объекта, а не ссылку — то единственный способ сделать это — создать новый объект и скопировать туда требуемые свойства.

Также стоит отметить, что работа с объектами по ссылке, помимо вышеперечисленных забавных эффектов дает также значительную экономию памяти, что немаловажно при широком использовании одного объекта в различных местах программы.

Примитивные значения

Как я упоминал выше, типы данных String и Number могут быть как объектами, так и примитивными значениями.

```
obj=new String('hello'); //Создаем строку как объект
simple='hello'; //Создаем примитивное значение

alert(obj); //hello
alert(simple); //hello - пока все предсказуемо
alert(obj.length); //6 - у объекта типа String есть свойство length, хранящее длину строки
alert(simple.length); //6
/* Хотя simple - не объект, мы можем обращаться к тому же набору свойств, что и у объекта типа String. Это довольно удобно */

obj.prop='text';
simple.prop='text';

alert(obj.prop); //text - раз obj обычный объект, то мы можем запросто придать ему еще одно свойство
alert(simple.prop); //undefined - а вот simple не объект, и этот номер у нас не пройдет
```

Все то же самое справедливо и для типа Number, и для Boolean (ну, кроме того, что в них нет свойства length, а есть ряд других замечательных свойств). Использование строк и чисел как объектов не несет в себе никакой практической пользы, т.к. примитивные значения удобнее в работе, но сохраняют при этом весь необходимый функционал. Тем не менее, для полноты картины необходимо понимать этот механизм.

Не стоит путать использование примитивных значений с использованием литералов — например, независимо от того, создаем мы массив как «test=new Array()» или как «test=[]», в результате все равно будет один и тот же объект. Никаких примитивных значений мы не получим.

Создание и использование объектов

Итак, в отличии от языков, где реализована класс-объектная парадигма, нам не нужно создавать сначала класс, чтобы потом создать объект класса. Мы можем сразу создать объект, что и сделаем в следующем примере:

```
test={
 simple_property: ‘Hello’,
 object_property: {
  user_1: ‘Петя',
  user_2: ‘Вася'
 },
 function_property: function(user) {
  alert(this.simple_property + ‘, ‘ + this.object_
property[user]);
 }
}

test.function_property(‘user_1’); //Hello, Петя.
```

Перед нами объект test, имеющий 3 свойства, названия которых, как я надеюсь, говорят сами за себя. Больше всего нас в нем интересует свойство function_property, содержащее функцию. Такую функцию можно назвать методом объекта.

В нашей функции дважды используется ключевое слово this, которое является указателем (т.е. ссылкой) на объект, из которого вызывается функция. Таким образом, this.simple_property=test.simple_property=’Hello’, а this.object_property[user]=test.

object_property[user]=’Петя'.

Необходимо четко осознавать, this всегда указывает именно на объект, из которого вызвана функция, а не на объект, к которому она принадлежит. Хотя в данном примере это один и тот же объект, это не всегда так.

```
test.function_property(‘user_1’); //Hello, Петя.

test2=new Object(); //Еще одна форма создания нового объекта, аналогичная test2={}

test.function_property.call(test2, 'user_1'); //ошибка
/* Метод call позволяет вызвать функцию от имени другого объекта. В данном случае, мы вызываем метод function_property объекта test, и его this указывает уже не на объект test, а на объект test2. А т.к. в нем нет свойства object_property, то при попытке получить this.object_property[user]скрипт выдаст ошибку */

//попробуем исправить ситуацию
test2.simple_property='Good day';
test2.object_property=test.object_property; //В данном случае воспользуемся указанием объекта по ссылке, чтобы не дублировать код

test.function_property.call(test2, 'user_1'); //Good day, Петя.
```

Из примера также должно быть видно, что нет четких этапов создания и использования объекта. Объект может быть как угодно модифицирован в любое время — до, после и даже во время использования. Это тоже важное отличие от «традиционного» ООП.

Конструктор

В примере выше мы создавали 2 объекта, обладающих некой схожестью. И там и там имелись свойства simple_property и object_property. Очевидно, что при написании реального кода также нередко встает задача создания одинаковых или просто похожих объектов. И разумеется, мы не должны каждый такой объект создавать вручную.

На помощь нам придет конструктор. Конструктор в JavaScript — это не часть класса (потому что здесь нет классов), а просто самостоятельная функция. Самая обычная функция.

```
make_me=function(_name) {
alert('меня запустили');
this.name=_name;
this.show_name=function() {alert(this.name);}
}

child=new make_me('Вася'); //меня запустили
/* Давайте разберемся, что здесь происходит.
Интерпретатор видит оператор new и проверяет, что находится справа от него. Т.к. make_me - это функция, и она может быть использована в качестве контруктора, то создается новый объект в памяти и запускается на выполнение функция make_me, причем ее this указывает как раз на этот новый объект. Далее этому объекту добавляется свойство name, которому присваивается значение из аргумента _name, и метод show_name. Также (не знаю в какой именно момент, но это и не важно) переменная child начинает указывать на наш новенький, только что рожденный объект */

alert(child.name); //Вася
child.show_name(); //Вася

child2=new make_me('Петя');
child2.show_name(); //Петя

child2.show_name=function() {alert('Не буду говорить свое имя');} //Не забываем, что можем изменять наши объекты в любой момент
child2.show_name(); //Не буду говорить свое имя

child.show_name(); //Вася - дети никак не влияют друг на друга
```

Также можно сравнить конструктора с отцом — он порождает ребенка, наделяя его определенными качествами, но сразу после создания ребенок становится полностью независим от родителя и может стать очень непохожим на своих братьев.

Если мы вспомним про описание типов данных в начале статьи, то становится понятно, что Object и его подтипы (Function, Array и другие) — это на самом деле конструкторы, придающие создаваемому объекту возможности функции, массива и т.д.

Итак, это уже намного лучше. У нас теперь есть возможность создавать объекты по некоторому образцу. Однако, не все еще хорошо. Во-первых, каждый созданный нами объект и все его свойства и методы занимают отдельное место в памяти, хотя во многом они повторяются. Во-вторых, как быть, если мы хотим сохранить связь между родителем и ребенком, и иметь возможность менять все дочерние объекты разом. На помощь нам придет прототип.

Прототип

Как у каждого ребенка есть отец и мать (хотя бы в биологическом смысле), также они есть и у каждого объекта в JavaScript. И если отец, как мы определелись, работает конструктором, то мать — это как раз прототип. Посмотрим, как это происходит:

```
make_me=function(_name) {
alert('меня запустили');
this.name=_name;
this.show_name=function() {alert(this.name);}
}
/*
Видя ключевое слово function, интерпретатор проверяет код справа от него, и т.к. все ок - создает новый объект в памяти, который одновременно является нашей функ-
```

```
цией. Затем, автоматически (без участия программиста) для этой функции создается свойство prototype, ссылающееся на пустой объект. Если бы мы это делали вручную, это выглядело бы как make_me.prototype=new Object();

Затем, данному объекту (на который указывает свойство prototype) также автоматически добавляется свойство constructor, указывающее обратно на функцию. Получается такая вот циклическая ссылка.

Теперь этот объект, который можно описать как {constructor: ...здесь ссылка на фунцию...} - и есть прототип функции.
*/

alert(typeof make_me.prototype); //Object - действительно, объект
alert(typeof make_me.prototype.constructor); //Function - это наша функция
alert(make_me.prototype.constructor === make_me); //true

make_me.prototype.set_name=function(_name) {this.name=_name;} //Добавляем в прототип функции make_me новый метод

child=new make_me('Вася'); //меня запустили
/* Теперь помимо всего того, что описано в предыдущем примере, дополнительно в объекте child создается скрытое свойство [[Prototype]], которое указывает на тот же объект, что и make_me.prototype. Т.к. свойство скрыто, мы не можем ни просмотреть его значение, ни изменить его - однако оно играет важную роль в дальнейшей работе */

alert(child.name); //Вася
child.show_name(); //Вася

child.set_name('Коля');
/* Сначала, интерпретатор ищет метод set_name в объекте child. Так как его там нет, он продолжает поиск в свойстве child.[[Prototype]], находит его там и запускает. */
child.show_name(); //Коля - теперь Васю зовут Коля :)

make_me.prototype.show_name2=function() {alert('Привет, ' + this.name;} //Т.к. прототип - это обычный объект, мы точно также можем его менять на лету

child2=new make_me('Петя');
child2.show_name2(); //Привет, Петя
child.show_name2(); //Привет, Коля - изменения в прототипе влияют не только на вновь созданные объекты, но и на все старые

child2.show_name2=function() {alert('Не буду говорить свое имя');} //Мы по прежнему можем изменить сам объект, при этом новый метод show_name2 в данном объекте (и только в нем) как бы «затрет» старый метод из прототипа
child2.show_name2(); //Не буду говорить свое имя - т.к. у нас теперь есть собственный метод show_name2, то он и вызывается, и поиск в прототипе не происходит
child.show_name2(); //Привет, Коля - здесь все по прежнему

make_me.prototype={prop: 'hello'} //Попробуем пересоздать прототип заново

alert(child.prop); //undefined
child.show_name2(); //Привет, Коля
/* Если вспомнить, что такое работа по ссылке, то все понятно. Пересоздание прототипа рвет связь, и теперь свойство [[Prototype]] у объектов child и child2 указывают на один объект (который раньше был прототипом функции make_me), а свойство make_me.prototype - на другой объект, который является новым прототипом функции make_me */

child3=new make_me('Олег');
alert(child3.prop); //hello - что и следовало ожидать
```

Как видно из примера, пока отец сохраняет верность матери (т.е. пока протип функции остается прежним), все дети зависят от матери и чутко реагируют на все изменения в ней. Однако, стоит только родителям развестись (конструктор меняет прототип на другой) — дети тут же разбегаются кто куда и больше связи с ними нет.

Немного о терминологии. До тех пор, пока первичная связь между конструктором и прототипом не разорвана, мы можем наблюдать следующую картину:

```
make_me=function(_name) {
alert('меня запустили');
this.name=_name;
this.show_name=function() {alert(this.name);}
}

make_me.prototype.set_name=function(_name) {this.name=_name;}
child=new make_me('Вася');

alert(typeof make_me.prototype); //object - у функции есть свойство prototype
alert(typeof child.prototype); //undefined - у созданного объекта НЕТ свойства prototype
alert(child.constructor.prototype === make_me.prototype); //true - зато у объекта есть свойство constructor, которое указывает на функцию-конструктор make_me, у которой, в свою очередь, есть свойство prototype
```

Как я заметил после чтения многочисленных форумов на эту тему, основные проблемы возникают у людей, когда они путают свойство prototype у функции и скрытое свойство [[Prototype]] у объекта, созданного с помощью этой функции. Оба этих свойства являются ссылкой на один и тот же объект (до тех пор, пока первичная связь прототипа с конструктором не нарушена), но это тем не менее разные свойства, с разными именами, одно из них доступно для программиста, а другое нет.

Необходимо всегда четко понимать, что если речь

идет о прототипе конструктора — то это всегда свойство prototype, а если о прототипе созданного объекта — то это скрытое свойство [[Prototype]].

Наследование

Теперь мы знаем, что у каждого объекта есть скрытая ссылка на прототип, а каждый прототип — это обычный объект. Наиболее чуткие читатели уже уловили запах рекурсии. Действительно, т.к. прототип — это обычный объект, то и он в свою очередь имеет ссылку на свой прототип, и так далее. Таким образом реализуется иерархия прототипов.

```
bird=function() {} //Это конструктор птички
bird.prototype.cry=function(){alert('Кри!');} //Птичка умеет кричать
bird.prototype.fly=function(){alert('Я лечу!');} //и летать

duck=function() {}
duck.prototype=new bird();
duck.prototype.cry=function(){alert('Кря-кря!');} //Утка кричит по другому
duck.prototype.constructor=duck; //Принудительно устанавливаем свойство prototype.constructor в duck, т.к. иначе оно будет ссылаться на bird

billy = new duck(); //Билли - это наша утка
billy.fly(); //Я лечу! - Билли может летать, потому что он птица
billy.cry(); //Кря-кря! - Билли кричит кря-кря, потому что он утка
```

Так можно реализовывать иерархию любого уровня вложенности.

Задача на звездочку

Теперь, раз уж мы столько знаем обо всем этом, давайте попробуем разобраться, сколько всего происходит в этих трех строчках.

```
make_me=function() {}
child=new make_me();
alert(child.toString()); //выводит [object]
```

В первой строке мы создаем новую функцию и переменную make_me, которая указывает на эту функцию. При этом создается прототип функции, make_me.prototype, в котором содержится свойство constructor, указывающее на make_me. Но это далеко не все. Т.к. функция make_me — это тоже объект, то он в свою очередь имеет папу и маму, т.е. конструктор и прототип. Его конструктор — это родная функция языка Function(), а прототип — объект, содержащий в себе методы call, apply и т.д. — именно благодаря этому прототипу мы и можем пользоваться этими методами в любой функции. Таким образом, у функции make_me появляется свойство [[Prototype]], указывающее на Function.prototype.

В свою очередь, прототип конструктора Function — тоже объект, конструктором которого является (сюрприз!) Object (т.е. Function.prototype.[[Prototype]].constructor===Object), а прототипом — объект, содержащий стандартные свойства и методы объекта, такие как toString, hasOwnProperty и другие (другими словами — Function.prototype.[[Prototype]]['hasOwnProperty'] — это как раз тот самый метод, которым мы можем пользоваться во всех производных объектах — причем это именно собственной метод данного объекта, а не наследованный). Вот таким вот интересным образом мы обнаруживаем, что все виды объектов являются производными от Object.

Можем ли мы продолжить дальше? Оказывается, нет. Object.prototype именно потому и содержит базовые свойства объекта, что не имеет собственного прототипа. Object.prototype.[[Prototype]]=null; В этом месте путешествие по цепочке прототипов в поиске свойства или метода прекращается.

Еще один интересный факт — конструктором Object является Function. T.e. Object.[[Prototype]].constructor===Function. Налицо еще одна циклическая ссылка — конструктор Object это Function, а конструктор Function.prototype — это Object.

Вернемся к нашему примеру. Как создается функция мы уже поняли, теперь перейдем ко второй строке. Там мы создаем объект child, конструктором которого является функция make_me, а прототипом — make_me.prototype.

Ну и в третей строчке мы видим, как интепретатор поднимается по цепочке, от child к child.[[Prototype]] (он же make_me.prototype), затем к child.[[Prototype]].[[Prototype]] (он же Object.prototype), и уже там находит метод toString, который и запускает на выполнение.

Примеси

Может показаться, что наследование через прототипы — единственный способ, возможный в JavaScript. Это не так. Мы имеем дело с очень гибким языком, который предоставляет не столько правила, сколько возможности.

Например, при желании мы можем вообще не использовать прототипы, а программировать с помощью концепции примесей. Для этого нам пригодятся наши старые добрые друзья — конструкторы.

```
//Это конструктор человека
man=function() {
this.live=function(){alert('Я живу');} //Человек умеет жить
this.walk=function(){alert('Я иду');} //Человек умеет ходить
}

//Это конструктор поэта
poet=function(){
this.kill=function(){alert('Поэт убил человека');} //Поэт может убить человека
```

```
this.live=function(){alert('Я мертв');} //От этого человек умрет
}

vladimir=new man(); //Владимир - человек
vladimir.live(); //Я живу - он жив
vladimir.walk(); //Я иду - он ходит

poet.call(vladimir); //Выполняем конструктор poet для объекта vladimir
vladimir.kill(); //Поэт убил человека
vladimir.live(); //Я мертв

//А теперь фокус
man.call(vladimir);
vladimir.live(); //Я живу
```

Что мы видим в данном примере? Во-первых, это возможность наследования от нескольких объектов, не находящихся в одной иерархии. В примере их 2, но может быть сколько угодно. Во-вторых, это отсутствие какой-либо иерархии вообще. Переопределение свойств и методов определяется исключительно поряком вызова конструкторов. В-третьих, это возможность еще более динамически менять объект, причем именно отдельный объект, а не всех потомков, как при изменении прототипа.

Замыкания и приватные свойства

Чтобы не раздувать эту и без того немаленькую статью, даю ссылку на пост Замыкания в JavaScript, где про это довольно подробно написано.

Что теперь со всем этим делать

Как я уже говорил выше, и произвольное изменение отдельных объектов, и использование конструкторов, и примеси, и гибкость прототипов — это лишь инструменты, возможности, которые позволяют программисту создать как ужасный, так и прекрасный во всех отношениях код. Важно лишь понимать, какие задачи мы решаем, какими средствами, какие цели достигаем и какую цену платим за это.

Причем вопрос о цене довольно нетривиален, особенно если мы говорим о разработке под браузер Internet Explorer 6 и 7 версий.
1. Память — тут все просто. Во всех браузерах наследование на прототипах отнимает в разы меньше памяти, чем при создании методов через конструкторы. Причем, чем больше методов и свойств у нас есть, тем больше разница. Однако, стоит помнить, что если у нас не тысяча одинаковых объектов а всего лишь один, то расходы памяти в любом случае будут небольшими, т.к. здесь стоит учитывать другие факторы.
2. Процессорное время — здесь основные тонкости связанны именно с браузерами от Microsoft.
С одной стороны, объекты, где методы и свойства создаются через конструктор — могут создаваться в разы (в некоторых случаях в десятки и сотни раз) медленнее, чем через прототип. Чем больше методов — тем медленнее. Так что если у вас в IE замирает на несколько секунд во время инициализации скрипта — есть повод копать в эту сторону.

С другой стороны, собственные методы объекта (созданные через конструктор) могут выполняется немного быстрее, чем прототипные. В случае, если позарез необходимо ускорить именно выполнение какого-то метода в этом браузере, то нужно это учесть. Имейте ввиду, ускоряется именно вызов метода (т.е. поиск его в объекте), а не его выполнение. Так что если сам метод у вас выполняется секунду, то особого увеличения быстродействия вы не заметите.

В других браузерах подобных проблем наблюдается, там время создания объектов и вызова их методов примерно одинаково для обоих подходов.

P.S. Обычно в статьях подобного рода автор предлагает некую обертку, либо пытающуюся реализовать класс-объектное наследование на базе прототипного, либо просто синтаксический сахар для прототипного наследования. Я не делаю этого намеренно, т.к. считаю, что человек, понявший смысл данной статьи, способен сам для себя написать любую обертку, и еще много интересных вещей :)

Стыкуем компоненты в JavaScript

sunnybear, webo.in

После заметки Стыкуем асинхронные скрипты и предложенного решения от Steve Souders я подумал о модульной загрузке какого-то сложного JavaScript-приложения. И понял, что предложенный подход в таком случае будет довольно громоздким: нам нужно будет в конец каждого модуля вставлять загрузчик следующих модулей. А если нам на разных страницах требуются различные наборы модулей и разная логика их загрузки? Тупик? Ан нет. Не зря Steve упоминает в самом начала своей заметки о событии onload / onreadystatechange для скриптов. Используя их, мы можем однозначно привязать некоторый код к окончанию загрузки конкретного модуля. Дело за малым: нам нужно определить этот самый код каким-либо образом.

Решение первое: дерево загрузки

В качестве наиболее простого способа определить порядок загрузки модулей на конкретной странице можно предложить глобальный массив, содержащий в себе дерево зависимостей. Например, такой:

```
var modules = [
    [0, 'item1', function(){
            alert('item1 is loaded');
    }],
    [1, 'item2', function(){
            alert('item2 is loaded');
    }],
    [1, 'item3', function(){
            alert('item3 is loaded');
    }]
];
```

В качестве элемента этого массива у нас выступает еще один массив. Первым элементом идет указание родителя (в том случае, если элемент является корнем и должен быть загружен сразу же), далее имя файла или его алиас. Последней идет произвольная функция, которую можно выполнить по загрузке.

Давайте рассмотрим, каким образом можно использовать данную структуру:

```
/* перебор и загрузка модулей */
function load_by_parent (i) {
    i = i || 0;
    var len = modules.length,
            module;
/* перебираем дерево модулей */
    while (len--) {
            module = modules[len];
/* и загружаем требуемые элементы */
            if (!module[0]) {
                    loader(len);
            }
    }
}

/* объявляем функцию-загрузчик */
function loader (i) {
    var module = modules[i];
/* создаем новый элемент script */
        var script = document.createElement('script');
        script.type = 'text/javascript';
    /* задаем имя файла */
        script.src = module[1] + '.js';
    /* задаем текст внутри тега для запуска по загрузке */
        script.text = module[2];
    /* запоминаем текущий индекс модуля */
        script.title = i + 1;
    /* выставляем обработчик загрузки для IE */
        script.onreadystatechange = function() {
                if (this.readyState === 'loaded') {
    /* перебираем модули и ищем те, которые нужно загрузить */
                        load_by_parent(this.title);
                }
        };

    /* выставляем обработчик загрузки для остальных */
        script.onload = function (e) {
    /* исполняем текст внутри тега (нужно только для Opera) */
                if (/opera/i.test(navigator.userAgent))
                {
                        eval(e.target.innerHTML);
                }
    /* перебираем модули и ищем те, которые нужно загрузить */
                        load_by_parent(this.title);
        };
    /* прикрепляем тег к документу */
        document.getElementsByTagName('head')[0].
appendChild(script);
    }

    /* загружаем корневые элементы */
    load_by_parent();
```

Мы можем вынести загрузку корневых элементов в событие загрузки страницы, а сами функции — в какую-либо библиотеку, либо объявлять прямо на странице. Задавая на каждой странице свое дерево, мы получаем полную гибкость в асинхронной загрузке любого количества JavaScript-модулей. Стоит отметить, что зависимости в таком случае разрешаются «от корня — к вершинам»: мы сами должны знать, какие базовые компоненты загрузить, а потом загрузить более продвинутые.

Решение второе: загрузка через DOM-дерево

Кроме того, я вспомнил, что подобной проблемой уже занимался Андрей Сумин и даже предложил свое решение в виде библиотеки JSX, которая позволяет назначать список зависимостей через DOM-дерево. Для этого у элементов, которые требуют загрузки каких-либо модулей для взаимодействия с пользователем, назначается класс с префиксом jsx-component, а далее идет уже список компонентов. Сама библиотека обходит DOM-дерево, находит все модули, которые нужно загрузить, и последовательно их загружает. Просто замечательно.

Но что, если нам требуется поменять обработчик по загрузке этого модуля? Как его задавать? Сама JSX использует атрибуты искомых узлов DOM-дерева сугубо для определения параметров этих модулей. Это достаточно удобно: ведь таким образом можно назначить и инициализатор модуля.

Также библиотека позволяет отслеживать повторную загрузку модулей, осуществлять догрузку модулей в случае плохого соединения и даже объединять разные модули в один исходный файл через систему алиасов. Таким образом, проблема асинхронной загрузки произвольного дерева модулей оказывается решенной. В случае JSX задача разрешается в обратном порядке: мы указываем основной файл (вершину дерева зависимостей), а он уже загружает все необходимые ему модули либо проверяет, что модули загружены.

Это все?

Решение третье: JSX + YASS

Почти. После недолгих раздумий JSX была взята за основу для построения модульной системы, которая могла бы стать основой для гибких и динамических клиентских приложений. Удалось совместить оба описанных выше подхода, что обеспечило все видимые функциональные требования к такого рода системе.

Для примера можно рассмотреть следующий участок HTML-кода:

```
    <div id=»item1» class=»yass-module-utils-base-dom»>
        <span id=»item2» class=»yass-module-dom»
title=»_('#item2')[0].innerHTML = 'component is
loading...';»></span>
    </div>
```

Давайте разберемся, какую логику загрузки он обеспечивает:

1. YASS при инициализации обходит DOM-дерево документа и выбирает все узлы с классом yass-module-*.
2. После этого формируется 2 потока загрузки модулей: для utils-base-dom и для dom. Причем в последнем случае загрузки, фактически, не будет: загрузчик дождется, пока состояние компонента dom будет выставлено в loaded, а только потом запустит (через eval) код, записанный в title этого элемента (в данном случае это span).
3. Первый поток загрузки асинхронно вызовет 3 файла с сервера: yass.dom.js, yass.base.js и yass.utils.js. По загрузке всех этих модулей (ибо они вызваны в цепочке зависимостей, в данном случае dom зависит от base, который зависит от utils) будет вызваны соответствующие инициализационные функции (если они определены). Таким образом возможны два типа обработчиков: непосредственно по загрузке компонента (будет вызвано для всех компонентов в цепочке) и после загрузки всей заданной цепочки компонентов (в нашем случае это utils-base-dom).
4. Если мы хотим каким-то образом расширить нашу цепочку, то может в конце каждого из указанных файлов прописать загрузку какой-либо другой цепочки (например, base-callbacks), которая «заморозит» загрузку модуля base до получения callbacks. Сделать это можно (имея в виду, что расширяем зависимости модуля base) следующим образом:

```
        _.load('callbacks-base');
```

5. Предыдущий шаг может быть также выполнен при помощи самого DOM-Дерева: нам нужно будет прописать для произвольного элемента класс yass-module-callbacks-base. Это добавит в дерево зависимостей искомую цепочку.

Для большей ясности описанное выше конечное дерево загружаемых модулей можно представить так:

```
dom
  -> base
      -> utils
      -> callbacks
```

Более того, на следующей страницы представлен процесс загрузки более сложного варианта дерева. Осторожно: задержек никаких нет, поэтому может работать очень быстро.

Естественно, весь указанный функционал уже добавлен в последнюю версию YASS. Можно начинать использовать и писать отзывы.

Покорим Ruby вместе!

MaxElc

Собственно в этой серии статей мы будем капля за каплей наполнять стаканчик знаниями о Руби, наполнять будем вместе — для меня это также станет дополнительным стимулом не бросать занятия. Надеюсь, что и вам будет интересно! Начнем?

Руби, Рельсы...

Что такое Ruby? Руби — это полноценный интерпретируемый язык программирования высокого уровня для объектно-ориентированного программирования. Интерпретируемый — значит что код программы хранится в виде обычного текста, который передается исполняющему его интерпретатору. Можно сравнить с PHP и C++ — как и в PHP достаточно написать код, условно загрузить его на сервер-хостинг, где его будет выполнять интерпретатор. В противовес C++ необходимо компилировать, но зато мы получаем готовое приложение, которое выполняется само по себе, никуда код передавать не надо, но и посмотреть в исходники, заглянуть в программу, увы, не выйдет — там нечитаемый бинарный код. Возвращаясь к Руби, делаем вывод, что язык этот замечателен для разработки веб-приложений, скриптов, в ОС же его использование ограничено скоростью приложения (как подсказали ниже, для фронтендов и небольших утилит Ruby очень даже подходит).

Что такое Ruby on Rails? Дословно: «Руби на Рельсах». На какие такие рельсы поставили этот ЯП? Rails — это наиболее известный фрэймворк для языка Ruby. Мы можем писать скрипты и на «чистом» Ruby (подобно большинству скриптов на PHP), однако рельсы позволяют повысить скорость, производительность написания кода, кроме того, этот фрэймворк реализует архитектуру MVC (Model — View — Controller) — о том, что это такое и как это использовать, поговорим позже.

Концепции Ruby и Rails

Ruby и Ruby on Rails следуют паре принципов для того, чтобы помочь писать код чище и красивее. Первый: DRY (Don't repeat yourself). Это означает, что мы должны писать необходимый код только один раз и только в строго определенном месте. Второй: CoC (Conventions over Configuration) — общие соглашения важнее собственной конфигурации. Для большинства необходимых в кодинге методов уже есть замечательные автоматические установки, которые и следует использовать, а в крайних случаях, если умолчания нас не устраивают, мы просто переписываем их под себя. Всё, для того, чтобы код был чистым и лаконичным, и при этом мы затрачивали минимум усилий.

Почему Ruby?

— Язык предельно лёгок в изучении в сравнении с другими ЯП
— Полностью объектно-ориентированный
— Архитектура MVC у Rails
— Нет необходимости писать много кода
— Очень расширяемый
— Open Source
— Несмотря на то, что фрэймворк Rails молодой, он многофункционален и содержит совсем немного багов

Среда для разработки

Как и в других интерпретируемых языках создавать готовые программы можно и в Блокноте, однако же мы всерьёз схватились за Руби ;) Стоило бы, наверное, погуглить IDE для языка, провести какое-то тестирования производительности и функциональности. Но мы же непрофессионалы, так что ищем наиболее универсальное, надежное, качественное, проверенное решение. И оно есть: это Ruby in Steel для Microsoft Visual Studio 2005/2008. Да, это сугубо коммерческий, закрытый продукт только для Windows. Но, во-первых, это знакомая IDE, которая верно будет вам служить в будущем, во-вторых она не базируется на Java, что положительно отражается на скорости работы (это важно для слабых компов, вроде моего), в-третьих, полнофункциональная trial-версия работает 60 дней, чего, я надеюсь, нам будет достаточно для нашего курса.

Если же вы отдаете предпочтение другим ОС, то несомненно стоит взглянуть на свободный кроссплатформенный NetBeans IDE 6.0 Ruby.

Выбранная вами среда разработки в дальнейшем особого значения не имеет, однако процесс установки компонентов я буду освещать для Ruby in Steel — он достаточно прост, так как вы можете скачать All-in-One Installer — готовый пакет с бесплатным Visual Studio 2008 и всем необходимым для нашего учебного места.

После загрузки распакуйте архив и запустите Setup. Отметьте нужные пакеты (в большинстве случаев стоит оставить все по умолчанию), нажмите Install и только успевайте нажимать Next и Finish в появляющихся окнах установщиков. В процессе установки будут загружены и установлены последние Rails, поэтому консольные окна не закрываем и ждем, когда появится «Press any key to continue...». Для MySQL можем задать запуск в виде сервиса Windows и установить пароль рута.

Никакой дополнительной конфигурации не требует-

ся — просто запускаем свежеустановленный Visual Studio.

Первый проект

Проект в Ruby in Steel создаем как обычно в VS: File — New — Project — Ruby In Steel (слева) — Ruby Project (справа) — Name (задаем имя проекта) — OK.

Стоит уделить внимание иерархии организации работы с кодом — проект (Project) содержит в себе один или несколько файлов в одной или нескольких папках, в то же время Solution (решение) может содержать несколько проектов.

После создания проекта он уже содержит в себе один пустой файл rubyfile.rb, новые файлы добавляются через контекстное меню проекта. Двойной щелчок по имени файла открывает его в редакторе. Дальнейший код будем писать прямо в rubyfile.rb — это наш тестовый файл.

Первая программа

Традиционно это Hello World:

```
puts «Hello World!»
```

Упс, это всё… Ctrl+F5 в среде — и результат, так сказать, налицо!

Игры со строками

```
puts «Hello World!\nPrivet Mir!»
```

Как вы заметили прямо в строках работают управляющие последовательности (Escape Sequences), в т.ч. \n (новая строка), \t (табуляция), \s (пробел) и др.

```
puts 'Hello World!\nPrivet Mir!'
```

А вот строка, заключенная в апострофы «глупая» — она не понимает ES, поэтому в большинстве случаев используем именно кавычки. Однако если заранее известно, что ES в строке не нужны, то апострофы позволяют выиграть в производительности приложения.

```
puts 'It\'s Ruby'
```

Хотя нет, «глупая» строка понимает \' и больше ничего.

```
print «Hello World!»
print «Hello World!»
```

В отличие от оператора puts print не делает перенос строки после вывода переменной.

```
puts «2 x 2 = #{2*2}»
```

Замечательная возможность Ruby — регулярное выражение в переменной — вы можете вставлять кусочки кода Ruby прямо в строки используя конструкцию #{code}.

```
puts «#{«Ruby! «*4}»
```

«Умножаем» строку на четыре и получаем слова песни Kaiser Chiefs ;) Причем делаем все это в одной строке кода!

```
puts «\n\t#{(1 + 2) * 3}\nGoodbye»
```

Попробуйте представить, что выведет этот код и проверьте ваше догадки

Числа и Выражения

В программировании выражением (expression) называется комбинация чисел, операторов и переменных, которая, будучи понята компьютером, дает результат в какой-либо форме. Вот несколько примеров:

```
8
2 + 3
«a» + «b» + «c»
100 - 5 * (2 - 1)
x + y
```

Рассмотрим и разберем простой код:

```
x = 5
y = x * 20
x += y
puts x
```

Придаём переменной x значение 5, для y устанавливаем значение x * 20 (100). Затем прибавляем y к x и выводим результат. Все передельно ясно, единственная заминка может возникнуть на третьей строке. Казалось бы адекватнее написать x = x + y, однако это — кусочек сахара Ruby. Также можем использовать x *= y и x /=, а x += 1 увеличит переменную на единицу.

Операторы сравнения и Условия

Еще код:

```
age = 22
puts «Sovsem Molodoi!» if age < 25
```

Компьютер не только выполняет заданные операции (иначе это был бы просто калькулятор), он использует логику для определения направления работы. Результат исполнения кода предсказуем. Разберем вторую строку. Прежде всего обратим внимание на age < 25 — известные всем «больше/меньше» (а также ==, >=, <=, <=>, !=) работают и в Руби. Дальше видим знакомое из других языков условие: if. А где же then, end? Еще один сахарок Руби, на самом деле вторая строка полностью аналогична коду:

```
if age < 25 then
puts «Sovsem Molodoi!»
end
```

Условия можно комбинировать. Чтобы получить противоположный эффект можно использовать слово unless:

```
age = 24
puts «You're NOT a teenager» unless age > 12 && age <
20
```

Код понятен. Идем дальше.

Итераторы

Вот что ожидает нас здесь:

```
4.times do puts «Ruby» end
```

Давайте попробуем запустить код — результатом будет четырёхкратное выполнение puts. Анализируем. В первую очередь берётся число 4, затем для него вызывается метод times, применимый ко всем числам в Руби (метод? что такое метод? это действие — об этом поговорим позже). Вместо того, чтобы передавать в метод данные, мы передаём ему код, находящийся между do и end. Метод times затем 4 раза выполняет этот самый код. Как всегда Руби хочет, чтобы нам было удобнее писать программки, и поэтому дает еще одно послабление. Полностью аналогичный код, без do и end:

```
4.times { puts «Ruby» }
```

В Руби (да и в других ЯП тоже) одним из механизмов создания циклов является итератор — это некий объект, который позволяет перебирать все элементы какого-либо набора. В нашем случае он задает цикл, или итерирует, в четыре повтора. Посмотрим другие итераторы, применимые к числам в Руби:

```
1.upto(5) { ...код цикла... }
10.downto(5) { ...код цикла... }
0.step(50, 5) { ...код цикла... }
```

Думаю, назначение каждого из них понятно. Снова код:

```
1.upto(5) { |number| puts number }
```

В результате программа «посчитает до пяти» — мы передали состояние итерации в код. В начале циклируемого кода число «от 1 до 5» отправляется в переменную number.

Плавающая точка

Наверняка вы заметили, что во всех написанных ранее программах нам не приходилось объявлять переменные, определять их тип — Руби делает это за нас (продолжаем лизать сахар). Попробуем разделить: puts 10 / 3. Результат — 3. Так как входящие числа целые, то и результат остался целым. Руби хотел помочь, но у него не получилось. Давайте подскажем ему: puts 10.0 / 3.0. Решение оказалось простым!

Иногда мы оказываемся в такой ситуации, что не можем контролировать входящие числа, однако и тут есть решение:

```
x = 10
y = 3
puts x.to_f / y.to_f
```

Для целых чисел есть специальный метод (т.е. действие) .to_f, преобразующий их в числа с плавающей точкой на лету. Обратное действие (округление до целой части) выполняет метод .to_i.

Немного об ООП

В прошлой капле я говорил о том, что нам пока удается уходить от ООП. На самом деле я лукавил — в Руби всё, с чем мы манипулируем — это объекты. ВСЁ! Это отличает язык от C++ и Java, где присутствуют примитивные типы и фразы, не возвращающие значения.

При написании кода на Ruby, как и любого другого ОО кода, мы прежде всего создаем модели, а не пытаемся повторить весь процесс в коде. Например, если вам нужно написать приложение для кулинарной книги, вы, скорее всего, захотите создать список рецептов. Для моделирования вам придется использовать некоторые способы сортировки, чтобы воспроизводить различные виды данных, синхронизировать позиции необходимых данных в каждом списке и др. нонсенс. ООП делает работу легче, предлагая вам создавать классы и объекты, чтобы моделировать необходимые составляющие. В нашем примере вы можете создать класс Рецепты со строчными атрибутами название и автор и атрибутом в виде массива ингридиенты. Назначение класса — моделировать какую-либо вещь в вашей программе. Класс — это прототип, чертёж «существительных» в вашем прокте: объектов. Экземпляры класса или объекты (взаимозаменяемые термины) затем берут эти прототипы и запускают их в действие. В примере объекты могут быть созданы для каждого рецепта в списке, которые будут экземплярами класса Рецепты, который в свою очередь будет содержать данные и делать вещи, относящиеся к рецептам (например, держать список ингридиентов, добавлять ингридиенты в этот список, и т. п.) и следить за целостностью рецептов (например, что только числа в количествах ингридиентов, что каждый рецепт должен иметь название и др.) — все эти действия над объектами, «глаголы», и есть методы.

Вернемся к началам этой капли. Да, действительно, любое число — также полноценный объект. В примере 4.times {puts «Ruby»} число 4 — объект, является экземпляром класса Integer (мы об этом не заявляем в коде, Руби делает это за нас), к нему применяется метод times, который прописан в классе (опять мы этого не видим, но это подразумевается). Более того, даже в выражении x = 1 + x единица — это объект, а + — его метод! Вот так, не зная принципов ООП, мы уже, сами того не заметив, стали «жертвами» объектно-ориентированного программирова-

ния.

Классы, объекты, методы

Вместо того, чтобы углубляться дальше в синтаксис Ruby, отставим циклы, типы, модули и др. — мы вернёмся к ним позже. Сейчас же мы посмотрим, как создавать классы, объекты и методы.

Как и в других объектных языках объекты определяются классом. Например этот класс определяет собак:

```
class Dog
   def set_name( aName )
     @myname = aName
   end
end
```

Ну а чем вам собаки — не класс? ;) Смотрим код. Определение класса начинается с ключевого слова class и названия самого класса, которое обязано должно начинаться с большой буквы. В классе определен метод set_name (def...end), который будет называть каждый объект-собаку. Он берёт входящий аргумент aName и назначает его значение переменной @myname.

Переменные, начинающиеся с @, — это переменные экземпляра класса. Важно то, что значение такой переменной доступно только внутри экземпляра, к которому она принадлежит. Создадим два объекта (две конкретные собаки) с помощью метода new:

```
moya = Dog.new
tvoya = Dog.new
```

Напоминаю и запоминаем: класс называем только с большой буквы, объект — только со строчной. Используем метод set_name, чтобы назвать собак:

```
moya.set_name( 'Lassie' )
tvoya.set_name( 'Rex' )
```

Как теперь вывести клички собак? Мы не можем «вытащить» переменную @name из объекта, так как внутренние данные объекта известны только ему (см. выше). Это основной принцип ООП: данные объекта приватны; это называется скрытие данных и является частью принципа инкапсуляции.

Для решения этой задачки просто добавим новый метод в класс, который будет выводить переменную:

```
def get_name
   return @myname
end
```

Слово return необзательно, Руби возвратит последнее полученное значение, однако писать его — хорошая привычка.

Заставим собаку лаять (для этого напишем еще один метод) и соберём всё в кучу:

```
class Dog
   def set_name( aName )
     @myname = aName
   end

   def get_name
     return @myname
   end

   def gav
     return 'r-r-r-r!'
   end
end

dog1 = Dog.new
dog1.set_name( 'Fido' )
puts(dog1.get_name)
puts(dog1.gav)
```

Представим, что у нас есть два класса (например, кошки и собаки), в каждом определено по объекту (конкретные кот и пёс) и методу с одинаковыми названиями («говорить», но кот мяукает, а пёс лает). Получается, что на один и тот же метод объекты реагируют по-разному. Возможность иметь несколько классов, содержащих одноименные методы, называется полиморфизм. Узнали… и забыли это грозное слово.

Атрибуты

Для нашей задачи есть другой подход — создадим необходимые методы с помощью атрибутов класса. Усложним поставленное задание и создадим зоопарк

```
class Cat
   attr_accessor :name, :age, :gender, :color
end
class Dog
   attr_accessor :name, :age, :gender, :color
end
class Snake
   attr_accessor :name, :age, :gender, :color, :length
end
```

Рассмотрим вторую строку. Она предоставляет три атрибута для класса Cat. Каждая кошка имеет свою кличку, возраст, пол и окрас — и код создает эти атрибуты. _accessor означает «сделайте эти атрибуты доступными для создания и изменения».

Создадим объект и применим атрибуты:

```
cat_object = Cat.new
cat_object.name = «Pussy»
.....................
puts cat_object.name
```

Наследование

Посмотрев на последний код с объявлением трёх классов, легко заметить, что они предельно похожи. Лишь у змеи дополнительно определена длина. Вспомним принципы программирования на Ruby

из первой капли. Один из них: DRY (Don't repeat yourself), который означает, что мы должны писать необходимый код только один раз, а у нас сплошные повторения. Нам просто необходимо обратить внимание на одну из лучших возможностей ООП: наследование.

Наследование позволяет разным классам соотноситься друг с другом и группироваться по схожестям в структуре. В нашем примере коты, собаки, змеи — всё это питомцы (pets). Так почему бы нам не создать класс Pet, который будет «родительским» для остальных, которые, в свою очередь, будут наследовать общие для всех питомцев свойства? Пишем:

```
class Pet
   attr_accessor :name, :age, :gender, :color
end
class Cat < Pet
end
class Dog < Pet
end
class Snake < Pet
   attr_accessor :length
end
```

Принцип и правила объявления наследованных классов, думаю, понятны, теперь попробуйте создать объекты, задать параметры и вывести результат. Заметим, что более простые классы стоят выше сложных в иерархии классов.

Диапазоны значений

Иногда полезно иметь возможность сохранить «концепт» простого списка, причём хочется, чтобы объявить его мы могли бы максимально просто, например: список из букв от A до Z, или числа от 1 до 25. С помощью диапазонов это возможно, они в Руби максимально интуитивно понятны. Вот простые числовые диапазоны:

```
digits = 0..9
scale1 = 0..10
scale2 = 0...10 #digits = scale2
```

Оператор .. включает конечное значение, а ... — нет (глупо — казалось, что должно быть наоборот). Однако сами по себе диапазоны мало используются.

Массивы

Простые переменные порой не годятся для реального программирования. Каждый современный ЯП, в т.ч. и Руби, поддерживает более сложные формы структурированных данных. В Руби вы можете представлять упорядоченные наборы объектов с помощью массивов.

Массивы в Руби динамические: можно (но не обязательно) указывать размер массива при создании. После этого он может расти без участия со стороны программиста, поэтому в практике редко приходится даже узнавать его размер. Массивы в Руби гетерогенные: они могут хранить данные самых разных типов данных (если быть точным, то массив хранит только ссылки на объекты). Три одинаковых массива:

```
a = Array.[](1,2,3,4)
b = Array[1,2,3,4]
c = [1,2,3,4]
```

Для создания массивов используется специальный метод класса []. Также есть метод new, который берет 0, 1 или два параметра: первый — количество элементов, второй — их значение. Смотрим пример:

```
d = Array.new # Создаем пустой массив
e = Array.new(3) # [nil, nil, nil]
f = Array.new(3, «ruby») # [«ruby», «ruby», «ruby»]
```

Частая ошибка начинающих — считаем, что элементы независимы друг от друга. Однако, как было сказано раньше, в массиве лишь ссылки на один объект. Чтобы избежать такого поведения используем блок (мы его уже встречали во второй капле — это код между {}):

```
f[0].capitalize! # f теперь: [«Ruby», «Ruby», «Ruby»]
g = Array.new(3) { «ruby» } # [«ruby», «ruby», «ruby»]
g[0].capitalize! # g теперь: [«Ruby», «ruby», «ruby»]
```

Обращение к элементам и установление их значений выполняются с помощью методов [] и []= соответственно. Каждый может принимать один или два (начало и длина) параметра или диапазон. Обратный отсчет с конца массива начинается с -1. Считывать значения можно и специальным простым методом at — он принимает только один параметр и поэтому работает немного быстрее, delete_at удалит элемент. Смотрим:

```
a = [1, 2, 3, 4, 5, 6]
b = a[0] # 1
c = a.at(0) # 1
d = a[-2] # 5
e = a.at(-2) # 5
f = a[9] # nil
g = a.at(9) # nil
h = a[3,3] # [4, 5, 6]
i = a[2..4] # [3, 4, 5]
j = a[2...4] # [3, 4]

a[1] = 8 # [1, 8, 3, 4, 5, 6]
a[1,3] = [10, 20, 30] # [1, 10, 20, 30, 5, 6]
a[0..3] = [2, 4, 6, 8] # [2, 4, 6, 8, 5, 6]
a[-1] = 12 # [2, 4, 6, 8, 5, 12]

a.delete_at(3) # [1, 2, 4, 5, 6]
a.delete_at(9) # nil
```

Другой способ — толкание (pushing) данных. Метод join «сливает» элементы массива в одну переменную, в качестве параметра указываем разделитель:

```
x = []
x << «Word»
```

```
x << «Play» << «Fun»
puts x.join(', ') # Word, Play, Fun
```

Методы first и last возвращают первый и последний элемент массива соответственно, a values_at — массив, содержащий только выбранные элементы:

```
x = [«alpha», «beta», «gamma», «delta», «epsilon»]
a = x.first # alpha
b = x.values_at(0..2,4) # [alpha, beta, gamma, epsilon]
```

Метод length и его алиас size возвратят количество элементов в массиве, a nitems не будет считать пустые элементы (nil), compact уберёт nil из массива вообще:

```
y = [1, 2, nil, nil, 3, 4]
c = y.size # 6
e = y.nitems # 4
d = y.compact # [1, 2, 3, 4]
```

Самый простой метод для сортировки массива — это sort (попробуйте сами). Сортировка работает только в массивах, содержащих элементы, которые поддаются сравнению — с массивами со смешанными типами данных метод возвратит ошибку. Отсортируем смешанный массив, налету преобразовывая его элементы в строки (преобразовываем методом to_s):

```
a = [1, 2, «three», «four», 5, 6]
b = a.sort {|x,y| x.to_s <=> y.to_s}
# b теперь [1, 2, 5, 6, «four», «three»]
```

Как это работает? <=> — один из методов сравнения (см. третью каплю). Блок возвращает -1, если первый их сравниваемой пары элемент меньше (тогда метод меняет элементы местами), если элементы равны, 1 — если больше (в последних двух случаях оставляем элементы на местах). Поэтому для убывающего порядка надо просто поменять местами сравниваемые элементы ({|x,y| y.to_s <=> x.to_s}). Думаю, что мы всё поняли.

Для выборки элементов из массива по критериям используем detect (find — его синоним) и find_all (select — то же самое), выражение критерия засовываем в блок:

```
x = [5, 8, 12, 9, 4, 30]
# find покажет нам только первый элемент, кратный шести
x.find {|e| e % 6 == 0 } # 12
# A select покажет все подходящие элементы
x.select {|e| e % 6 == 0} # [12, 30]
```

Метод reject обратен select — он удалит каждый элемент, удовлетворяющий блоку:

```
x = [5, 8, 12, 9, 4, 30]
d = c.reject {|e| e % 2 == 0} # [5, 9]
```

delete удалит все элементы, содержащие определенные данные:

```
c = [«alpha», «beta», «gamma», «delta»]
c.delete(«delta»)
# Теперь c = [«alpha», «beta», «gamma»]
```

В классе массивов определен итератор each — работает он просто, догадайтесь, что делает этот код:

```
[1, «test», 2, 3, 4].each { |element| puts element.to_s
+ «X» }
```

Можно создать копию массива и изменить её налету с помощью итератора collect:

```
[1, 2, 3, 4].collect { |element| element * 2 } #[2, 4,
6, 8]
```

Объединяем массивы:

```
x = [1, 2, 3]
y = [«a», «b», «c»]
z = x + y #[1, 2, 3, «a», «b», «c»]

a = [1, 2, 3, 4]
b = [3, 4, 5, 6]
a – b # [1, 2] – разность массивов
a & b # [3, 4] – пересечение массивов
a | b # [1, 2, 3, 4, 5, 6] – объединение с удалением
дупликатов
a*2 # [1, 2, 3, 4, 1, 2, 3, 4] - повторение
```

Наше приложение: Текстовый анализатор

Собственно программа простая: она будет считывать текстовый файл, анализировать его по некоторым паттернам, считать статистику и выводить результат. Руби замечательно подходит для анализа документов и текстов с помощью регулярных выражений и методов scan и split. В этом приложении мы сконцентрируемся на простом и быстром программировании и не будем организовывать объектно-ориентированную структуру.

Основные возможности

Вот список возможностей, которые мы должны реализовать:

— подсчет символов
— подсчет символов без пробелов
— подсчет строк
— подсчет слов
— подсчет абзацев
— среднее число слов в предложении
— среднее число предложений в абзаце

Реализация

В начале разработки новой программы полезно представить неоходимы ключевые шаги. Давайте выделим основные ступени:

— Загрузить файл, содержащий текст, который мы хотим проанализировать.
— Так как мы загружаем файл по строками, сразу будем считать их количество для необходимой статистики.
— Вставить весь текст в одну строку и измерить ее

длину для подсчета символов.
— Временно удалить все пробелы и посчитать длину новой строки без них.
— Разбить строку по пробелам, чтобы узнать количество слов.
— Разбить строку по знакам препинания, чтобы посчитатать количество предложений
— Разбить по двойным переносам строки, чтобы узнать количество абзацев.
— Произвести подсчеты, чтобы узнать средние величины для статистики.

Создайте новый, пустой файл исходников и сохраните его как analyzer.rb.

Ищем какой-нибудь текст

Перед тем, как начать писать код, неоходимо найти кусок текста для тестов. В нашем примере мы будем использовать первую часть рассказа Оливер Твист, которую вы можете загрузить здесь. Сохраните файл под именем text.txt там же, где находится analyzer.rb.

Загрузка файла и подсчет строк

Настало время для кодинга! Первый шаг — загрузка файла. Руби предоставляет достаточный список методов для файловых манипуляций в классе File. Вот код, который откроет наш text.txt:

```
File.open(«text.txt»).each { |line| puts line }
```

Внесите код в analyzer.rb и запустите программу. В результате вы увидите бегущие по экрану строки текста.

Вы запрашиваете класс File открыть text.txt, а затем, как в случае с массивами, вызываете метод each прямо на файл, заставляя передавать каждую строку одну за другой во внутренний блок, где puts отпраляет их на экран. Отредактируйте код, чтобы он выглядел так:

```
line_count = 0
File.open(«text.txt»).each { |line| line_count += 1 }
puts line_count
```

Вы определяете переменную line_count, чтобы хранить в ней счетчик строк, затем окрываете файл и итерируете по строкам, увеличивая line_count на 1 каждый раз. В конце вы выводите результат на экран (около 127 в нашем примере). У нас есть первый кусочек для статистики!

Вы посчитали строки, однако у нас по-прежнему нет возможности посчитать слова, предложения, абзацы. Это легко исправить. Давайте немного изменим код и добавим переменную text, чтобы налету собрать в нее все строки:

```
text=''
line_count = 0
File.open(«text.txt»).each do |line|
   line_count += 1
   text << line
end

puts «#{line_count} lines»
```

В отличие от предыдущего кода, этот представляет переменную text и добавляет в нее каждую строку. Когда итерация окончена, в text находится весь наш текст.

Казалось бы код максимально лаконичен и прост, однако в File также существуют другие методы, которые могут быть использованы в нашем случае. Например, мы можем переписать наш код так:

```
lines = File.readlines(«text.txt»)
line_count = lines.size
text = lines.join

puts «#{line_count} lines»
```

Намного проще! Метод readlines считывает весь файл в массив, строка за строкой.

Считаем символы

Так как мы собрали весь файл в text, мы можем использовать применимый для строк метод length, который вернет размер (количество символов) в строке, и, соответственно, во всем нашем тексте. Допишем код:

```
total_characters = text.length
puts «#{total_characters} characters»
```

Еще один элемент статистики, который нам необходим — это подсчёт символов без пробелов. Для этого воспользуемся методами замены символов. Вот пример:

```
puts «foobar».sub('bar', 'foo') #foofoo
```

Метод sub нашел набор символов, переданный в первом параметре, и заменил их на символы из второго. Однако sub находит и изменяет только одно, первое встретившееся вхождение символов, метод gsub производит все возможные замены за раз.

Регулярные выражения

А как насчет замены более сложных патернов? Для этого используются регулярные выражения (regular expression). Целые книги были написаны на эту тему, поэтому мы ограничимся лишь кратким обзором этого мощного инструмента для работы с текстом. В Руби регулярные выражения создаются с помощью заключения патерна между слэшами (/pattern/). И в Руби, естественно, это также объекты. Например, вы можете задать следующий патерн для того, чтобы выбрать строки, содержащие текст Perl или текст Python: /Perl|Python/. В слэшах заключен наш патерн, состоящий из двух необходимых нам слов, разделенных прямой чертой (пайпом, pipe, |). Это символ означает «или то, что слева, или то, что справа». Вы также можете использовать скобки, как в число-

вых выражениях: /P(erl|ython)/.

В патернах можно реализовать повторения: /ab+c/ (это не сложение, рассматриваем как a, затем b+, затем c). Такому патерну соответствует строка, содержащая вхождение a, затем одного или более b, и, наконец, c. Заменим плюс на звездочку, теперь /ab*c/ соответствует строка содержащая a, ноль или больше b и c. + и * — это так называемые квантификаторы, чьё назначение, я думаю, понятно

Мы также можем выбирать строки, содержащие определенные символы. Наиболее простой пример — это патерны классов символов, например, \s означает whitespace (это пробелы, табы, переносы строки и т. п.), под \d попадают все числа, и др. Вот сводка, взятая из учебника по Руби на Wikibooks:

Короткие записи популярных символьных классов

Короткая запись	Полная запись	Описание
\s	[\f\t\n\r]	Пробельный символ
\S	[^\f\t\n\r]	Любой символ, кроме пробельного
\d	[0-9]	Цифра
\D	[^0-9]	Любой символ, кроме цифры
\w	[a-zA-Z0-9]	Латиница или цифра
\W	[^a-zA-Z0-9]	Любой символ, кроме латиницы или цифры
.	[^\n\r]	Любой символ, кроме перевода строки
\b		Граница слова
\B		Не граница слова
\A		Начало строки
\Z		Конец строки

Продолжаем считать символы

Итак, теперь мы знаем как убрать из строки все ненужные символы:

```
total_characters_nospaces = text.gsub(/\s+/, '').length
puts «#{total_characters_nospaces} characters excluding spaces»
```

Добавим этот код в конец нашего файла и переходим к подсчету слов.

Считаем слова

Для подсчета количества слов есть два подхода:

1. Посчитать число групп непрерывных символов, используя метод scan
2. Разбить текст по пробельным символам и посчитать получившиеся фрагменты с помощью split и size.

Пойдем по второму пути. По умолчанию (без параметров) split разобьет строку по пробельным символам и поместит фрагменты в массив. Нам остается только узнать длину массива. Дописываем код:

```
word_count = text.split.length
puts «#{word_count} words»
```

Считаем предложения и абзацы

Если нам понятно, как был реализован подсчет символов, то с предложениями и абзацами сложностей не будет. Единственное отличие в патерне, по которому мы разбиваем текст. Для предложения это точка, вопросительный и восклицательный знаки, для абзацев — двойной перенос строки. Вот код:

```
paragraph_count = text.split(/\n\n/).length
puts «#{paragraph_count} paragraphs»
sentence_count = text.split(/\.|\?|!/).length
puts «#{sentence_count} sentences»
```

Думаю, код понятен. Единственную сложность может составить патерн у предложений. Однако, он только выглядит страшно. Мы не можем просто ставить символы . и ? — мы их «экранируем» наклонной чертой.

Подсчет остальных значений

У нас уже есть количество слов, абзацев и предложений в переменных word_count, paragraph_count и sentence_count соответственно, поэтому дальше работает только арифметика:

```
puts «#{sentence_count / paragraph_count} sentences per paragraph (average)»
puts «#{word_count / sentence_count} words per sentence (average)»
```

Исходный код

Мы дополняли исходный код шаг за шагом, поэтому логика и вывод на экран у нас смешались. Давайте расставим все на свои места. Далее только косметические изменения:

```
lines = File.readlines(«text.txt»)
line_count = lines.size
text = lines.join
word_count = text.split.length
character_count = text.length
character_count_nospaces = text.gsub(/\s+/, '').length
paragraph_count = text.split(/\n\n/).length
sentence_count = text.split(/\.|\?|!/).length

puts «#{line_count} lines»
puts «#{character_count} characters»
puts «#{character_count_nospaces} characters excluding spaces»
puts «#{word_count} words»
puts «#{paragraph_count} paragraphs»
puts «#{sentence_count} sentences»
puts «#{sentence_count / paragraph_count} sentences per paragraph (average)»
puts «#{word_count / sentence_count} words per sentence (average)»
```

Ввод и inspect

```
puts «What is your name?»
STDOUT.flush
chompname = gets.chomp
puts «Again, what is your name?»
name = gets
puts «Hello, « + name
puts «Hi, « + chompname
puts 'But name = ' + name.inspect + ' and chompname = '
+ chompname.inspect
```

STDOUT — стандартная глобальная константа, обозначающая стандартный канал вывода. Метод flush очищает все данные во внутреннем буфере ввода/вывода Руби. Использование этой строки кода необязательно, но рекомендуется. Помним, что все константы должны начинаться с заглавной буквы.

gets принимает одну строку введенных данных и передает его переменной. chomp — это метод класса String. Несмотря на то, что результат мы видим одинаковый, необходимо помнить, что gets возвращает строку и \n, в то время как chomp удаляет этот \n (метод также удаляет возврат каретки \r и комбинацию \r\n).

Легко продемонстрировать это с помощью метода inspect, роль которого «заглядывать» в переменные, классы — в общем в любые объекты Руби.

Пунктуация в методах

Восклицательный знак в конце метода означает, что он не только возвращает результат, но и изменяет объект, к которому он применен (это так называемый «опасный» метод). Метод strip убирает пробелы в конце строки:

```
string = 'Bring, bring     '
a = string.strip!
puts string
puts a
```

Методы с вопросительным знаком, так называемые предикаты, возвращают только true или false, например, метод массивов empty? вернет true, если если в массиве нет элементов:

```
a = []
puts «empty» if a.empty?
```

Метод any? наоборот вернет true, если в массиве элементы присутствуют, а nonzero?, определенный в классе Numeric, выдаст nil, если число, на которое он вызван, равно нулю, в противном случае вернет это число.

Используем %w

Порой создание массива из слов может быть большой головной болью, однако в Руби есть упрощение для этого: %w{} делает то, что нам нужно:

```
pets1 = [ 'cat', 'dog', 'snake', 'hamster', 'rat' ]
pets2 = %w{ cat dog snake hamster rat } # pets1 = pets2
```

Ветвление по условиям

Мы уже говорили о том, что выражения условий позволяют контролировать направления выполнения кода. Повторим эти выражения и узнаем новые.

```
if

a = 7
if a == 4
   a = 9
end
```

Но Руби не был бы Руби, если б не упрощал нам задачу. Полностью аналогичный цикл:

```
a = 7
a = 9 if a == 4
```

if-elsif-else

Пример условия:

```
a = 7
if a == 4
   a = 9
else
   if a == 7
      a = 10
   end
end
```

elsif максимально упростит это условие и получаем:

```
a = 7
if a == 4
   a = 9
elsif a == 7
   a = 10
end
```

Трёхместный оператор

```
a = 7
plus_minus = '+'
print «#{a} #{plus_minus} 2 = « + (plus_minus == '+' ?
(a+2).to_s : (a-2).to_s)
```

Конструкция [? (expr) : (expr)] называется трёхместным (ternary) оператором (единственный трехместный в Руби) и служит для подсчета выражения и возвращения результата. Рекомендуется использовать только для второстепенных задач, так как подобный код тяжело воспринимается. Подсчитывается сначала первый операнд перед ?, если его значение не false и не nil, значением выражения становится значение второго операнда, иначе — третьего (после :).

while

while в Руби синтаксически похож на if и while в других ЯП:

```
a = 0
while a < 5
   puts a.to_s
   a += 1
end
```

Как обычно цикл можно поместить в одну строку: <...код...> while <выражение>

Символы (Symbols)

В коде Symbol выглядит как имя переменной, только начинающееся с :, например, :action. Symbol — самый простой объект в Руби, который возможно создать — у него есть только имя и ID. Symbol'ы более эфективны, производительны, чем строки, — данное имя для symbol ссылается на один объект на протяжении всей программы, в то время как две строки с одинаковым содержанием являются разными объектами — это сохраняет время и память:

```
ruby_know = :yes
if ruby_know == :yes
   puts «You're a good guy!»
else
   puts 'Learn Ruby!'
end
```

В этом примере :yes — symbol, он не содержит значений или объектов, вместо этого он используется как постоянное имя в коде. Мы можем преспокойно заменить :yes на строку «yes», результат будет такой же, но программа будет менее производительной. Более подробно о теме вы можете узнать в замечательной статье от Kane «Различие между символами и строками».

Ассоциативные массивы

Асоциативные массивы (далее хэши, hashes) похожи на массивы, так как они также содаржат упорядоченный набор объектов. Однако в массиве объекты индексируются числами, а в хэше индексом может быть любой объект. Простой пример:

```
h = {'dog' => 'sobaka',
'cat' => 'koshka',
'donkey' => 'oslik'}

puts h.length  # 3
puts h['dog']  # 'sobaka'
puts h         # catkoshkadonkeyoslikdogsobaka
```

Как видим элементы хэша не упорядочиваются, поэтому хэши непригодны для списков, очередей и т.д.

Везде, где вы хотите поставить строку в кавычках, задумайтесь о применении symbol:

```
users = Hash.new
users[:nickname] = 'MaxElc'
users[:language] = 'Russian'
puts users[:nickname] #MaxElc
```

new и initialize

Давайте взглянем еще раз на код из четвертой капли:

```
class Dog
        def set_name( aName )
                @myname = aName
        end

        def get_name
                return @myname
        end

        def gav
                return 'r-r-r-r!'
        end
end

dog1 = Dog.new
dog1.set_name( 'Fido' )
puts(dog1.get_name)
puts(dog1.gav)
```

Похожая программа:

```
class Dog
        def initialize(name, age)
                @name = name
                @age = age
        end

        def get_name
                return @name
        end

        def get_age
                return @age
        end
end

d = Dog.new('Fido', 2)
puts «My name is #{d.get_name} and I'm #{d.get_age}»
```

Когда определяется новый класс (обычно используя class Name ... end), создается объект типа Class. Когда вызывается Name.new для создания нового объекта, вызывается метод экземляра new из Class, который в свою очередь активизирует allocate, чтобы выделить память для объекта перед тем, как будет вызван метод initialize нового объекта. Фазы конструкции и инициализации объекта отдельны и могут быть переписаны. Инициализация происходит через метод экземпляра initialize, конструкция — через new. initialize — не конструктор (спорно, конечно, но в разных источниках — разное мнение).

Использование initialize имеет два явных преимущества над установкой переменных экземляра, используя методы типа set_name. Прежде всего сложный класс может содержать множество переменных экземпляра и все их можно объявить с помощью одной строки с initialize без необходимости пи-

сать методы для каждой. Также если все переменные объявляются во время создания объекта (как написано выше непосредственно после new вызывается именно initialize), у вас никогда не останется неопределенных переменных (nil).

Еще проще

В коде мы определили два метода (get_name и get_age), чтобы возвращать две переменные экземпляра. Так как это простая и часто используемая идиома, Руби предоставляет упрощение: attr_reader определит эти методы за нас:

```
class Dog
        attr_reader :name, :age
        def initialize(name, age)
                @name = name
                @age = age
        end
end

d = Dog.new(«Fido», 2)
puts «My name is #{d.name} and I'm #{d.age}»
```

Заметим, что в attr_reader используются symbol'ы (см. прошлую каплю) и то, как изменился запрос значения в выводе. attr_writer определит set-методы (set_name в первом листинге, например), а attr_accessor комбинирует возможности ридера и райтера (см. пример с зоопарком в четвертой капле).

initialize + наследование

В четвертой капле мы уже говорили о наследовании в «зоопарке», однако там мы использовали attr_accessor. Как же будет выглядеть наследование, если мы от них откажемся и вернемся к initialize и методам? Не намного сложнее:

```
class Pet
        def initialize(name, age)
                @name = name
                @age = age
        end

        def get_name
                return @name
        end

        def get_age
                return @age
        end

end

class Dog < Pet
        def initialize(name, age)
                @name = name
                @age = age
                super
        end
end

class Snake < Pet
        def initialize(name, age, length)
                @name = name
                @age = age
                @length = length
                super(name, age)
        end

        def get_length
                return @length
        end

end

d = Dog.new('Fido', 2)
s = Snake.new('Lili', 2, 85)
puts «Dog: My name is #{d.get_name} and I'm #{d.get_
age}»
puts «Snake: My name is #{s.get_name}, I'm #{s.get_age}
& I'm #{s.get_length} cm»
```

В этом примере мы отказались еще и от кошек :) Как всегда самый простой класс выше по иерархии. Единственное отличие — это ключевое слово super. Им обозначаются переменные, которые классы-потомки должны передать вышестоящему классу. Просто super передаст все переменные, super() — ни одной. Общие методы уходят в класс-родитель, у потомков остается только инициализация переменных и собственные методы.

Методы объектов

Даже новосозданный, пустой объект уже «откликается» на ряд методов. Выведем список этих методов кодом: puts d.methods.sort, где d — любой объект. Из всех стоит выделить object_id (у всех объектов Руби есть уникальный номер, который и выведет метод), class (выведет класс, которому принадлежит объект) и обратный instance_of? (вернет true, если объект принадлежит классу из параметра, например, puts 10.instance_of?(Fixnum))

Вы можете узнать, может ли объект ответить на сообщение, которое вы хотите отправить ему с помощью метода respond_to?. Чаще всего он используется в условии:

```
if d.respond_to?(«gav»)
        d.gav
else
        puts «Sorry, don't understand.»
end
```

Proc в Руби

Блок (do ... end или {...}) объектом не является, но он может быть преобразован в объект класса Proc с помощью метода lambda. Активизирует блок метод call из Proc. Методы могут содержать в себе proc'и:

```
def method proc
        puts 'Start of method'
        proc.call
```

```
        puts 'End of method'
end

say = lambda {puts 'Hello'}

method say
```

Замораживание объектов

Метод freeze в классе Object защищает объект от изменений, превращая его в константу. После «заморозки» объекта, любая попытка изменить его превратится в ошибку TypeError. Метод frozen? позволит узнать, заморожен ли объект:

```
a = b = 'Original String'
b.freeze
puts a.frozen? # true
puts b.frozen? # true
a = 'New String'
puts a
puts b
puts a.frozen? # false
puts b.frozen? # true
```

Переменые a и b сначала ссылаются на один объект, поэтому true в двух первых случаях.

Иногда Руби самостоятельно копирует объекты и замораживает копии. Например, когда вы используете строку в качестве ключа хэша, Руби замораживает ее копию и в дальнейшем использует именно ее. Поэтому даже если строка изменится, на ключ это не повлияет.

Сериализация объектов

В Java есть возможность сериализировать объекты, позволяя сохранять их в бинарном виде в файл и извлекать по мере надобности. Руби называет этот вид сериализации маршалингом (marshaling), при этом используется встроенная библиотека Marshal. Основные методы – dump и load:

```
f = File.open( 'peoples.sav', 'w' )
Marshal.dump( [«bred», «bert», «kate»], f )
f.close
File.open( 'peoples2.sav', 'w' ){ |friendsfile|
    Marshal.dump( [«anny», «agnes», «john» ],
friendsfile )
}
myfriends = Marshal.load(File.open('peoples.sav' ))
morefriends = Marshal.load(File.open('peoples2.sav' ))
puts myfriends
puts morefriends
```

В отличие от Marshal, библиотека YAML позволяет сохранять данные в текстовом формате, о ней как-нибудь в другой раз.

Модули и Примеси

Модули в Руби похожи на классы в том, что они содержат набор методов, константы, другие модули и определения классов. Модули задаются как классы, только слово module используется вместо class. В отличие от классов создать объекты на основе модуля нельзя, модуль не может иметь подклассы. Вместо этого вы добавляете недостающую функциональность класса или отдельного объекта с помощью модуля. Модули – одиночки, нет иерархии и наследования. (Вообще класс Module имеет суперкласс — Object, однако любой созданный модуль суперкласса не имеет).

Есть два предназначения модулей. Во-первых, они служат централизованного хранения констант и методов, например:

```
module Trig
        PI = 3.1416
        # методы класса
        def Trig.sin(x)
                # ...
        end
        def Trig.cos(x)
                # ...
        end
end
```

Во-вторых, модули позволяют делить функциональность между классами, при включении (include) модуля в класс, его методы добавляются в класс. Такой способ называется примесью (mixin):

```
module MyModule
   GOODMOOD = «happy»
   BADMOOD = «grumpy»

        def greet
                return «I'm #{GOODMOOD}. How are you?»
        end

        def MyModule.greet
                return «I'm #{BADMOOD}. How are you?»
        end
end

class MyClass
   include MyModule

        def sayHi
                puts( greet )
        end
end

ob = MyClass.new
ob.sayHi
puts(ob.greet)
```

Руби (в отличие от C++) не разрешает множественное наследование, заменяют его mixin'ы.

Множественные конструкторы

Что если мы хотим иметь несколько конструкторов для объекта? Ничто не мешает нам создать дополнительные методы класса, которые возвращают но-

вые объекты. В следующем примере мы описываем котов, имеющих пять параметров: вес, высота, и три на окрас. Мы создадим дополнительные методы, которые определят некоторые типы котов “по умолчанию” (например, черный кот или толстый кот):

```
class SuperCat

    def initialize(height, weight, tail_color, head_
color, legs_color)
        @height, @weight, @tail_color, @head_color, @
legs_color = height, weight, tail_color, head_color,
legs_color
    end

    def SuperCat.white_cat(height, weight)
        new(height, weight, «white», «white», «white»)
    end

    def SuperCat.black_cat(height, weight)
        new(height, weight, «black», «black», «black»)
    end

    def SuperCat.big_cat(tail_color, head_color, legs_
color)
        new(100, 100, tail_color, head_color, legs_
color)
    end

end
a = SuperCat.new(10, 15, «white», «black», «white»)
b = SuperCat.black_cat(13, 20)
c = SuperCat.big_cat(«white», «red», «red»)
p(a); p(b); p(c)
```

Имеет ли здесь место слово “конструктор”? Мы оставим этот вопрос для юристов.

Развитие конструкторов

Объекты становятся все сложнее, они собирают все больше атрибутов, которые необходимо инициализировать при создании объекта. Отвечающий за это конструктов становится длинным и обременительным, заставляя нас пересчитывать параметры и переносить строки.

Один из способов справиться с этой сложностью – передать блок методу initialize. Затем мы можем использовать блок для инициализации объекта. Будем использовать метод instance_eval вместо eval:

```
class HyperCat
        attr_accessor :name,
        :height, :weight, :age,
        :tail_color, :head_color, :legs_color

    def initialize(&block)
        instance_eval &block
    end

    # Другие методы...
end
pussy = HyperCat.new do
    self.name = «Pussy»
    self.height = 10
    self.weight = 12
    self.age = 3.2
    self.tail_color = «gray»
    self.head_color = «gray»
    self.legs_color = «white»
end

p pussy
```

Мы используем аксессоры в атрибутах, так легче передавать им значения. Также нам необходимо ссылаться на self, потому что метод назначения значения всегда берет явного приемника чтобы создать локальную переменную.

Контроль доступа к методам

В Руби объект в основном определяется предоставляемым им интерфейсом, методом, с помощью которого он становится доступным другим. Однако, при написании класса нам часто нужны вспомогательные методы, используемые внутри класса, но опасные, если они доступны извне. Вот где нам поможет метод private класса Module.

Мы можем использовать private двумя способами. Если вызвать private без параметров в теле класса, все методы ниже станут приватными. Или вы можете передать список методов в виде символов в качестве параметров private:

```
class Bank
    def open_safe
        # ...
    end

    def close_safe
        # ...
    end

    private :open_safe, :close_safe

    def make_withdrawal(amount)
        if access_allowed
            open_safe
            get_cash(amount)
            close_safe
        end
    end
    # остальное - приватное
    private

    def get_cash
        # ...
    end

    def access_allowed
        # ...
    end
end
```

Копирование объектов

Методы clone и dup создают копии вызывающего элемента. Метод dup копирует только содержание объекта, в то время как clone берет и такие вещи, как синглтон классы, связанные с объектом:

```
s1 = «cat»

def s1.upcase
    «CaT»
end

s1_dup = s1.dup
s1_clone = s1.clone
s1                    #=> «cat»
s1_dup.upcase         #=> «CAT» (синглтон метод не ско-
пировался)
s1_clone.upcase     #=> «CaT»
```

Углубляясь в модули

Перелистнем на одну каплю назад и вспомним, что такое модули и как их применять, в частности, обратим внимание на примеси и пример, приведенный там. Но что случится при смешивании с нашими методами модуля? Если вы думаете, что будут включены как методы класса, то Руби так не поступает. Для этого можно сделать так:

```
module MyMod

    def meth
        puts «Метод экземпляра модуля»
        puts «может стать методом класса.»
    end

end

class MyClass

    class << self    # Здесь self это MyClass
        include MyMod
    end

end
```

Здесь нам пригодится метод extend – с ним пример становится гораздо проще:

```
class MyClass
    extend MyMod
end
MyClass.meth
```

Создание Struct'ов

Иногда нам нужно просто сгруппировать некоторые данные без дальнейшей обработки. Мы можем создать класс:

```
class ExtraCat

    attr_accessor :name, :age, :weight

    def initialize(name, age, weight)
        @name, @age, @weight = name, age, weight
    end

end

lucky = ExtraCat.new(«Lucky», 2, 4)
```

Это, конечно, работает, но здесь одни повторения. Вот почему пригодился класс Struct. Как attr_accessor за нас определяет необходимые методы, так и Struct определяет классы, содержащие одни атрибуты. Эти классы называются структурными шаблонами (structure templates).

```
ExtraCat = Struct.new(«ExtraCat», :name, :age, :weight)
lucky = ExtraCat.new(«Lucky», 2, 4)
```

Описывая и обсуждая компьютерные программы, мы часто используем образные и человеческие метафоры. Например, мы говорим, что находимся в классе или возвращаемся из вызова метода. Иногда имеет смысл говорить во втором лице, например, object.respond_to?(«x»): "Эй, объект, ты ответишь на x? И пока программа интепретируется контекст меняется снова и снова.

Некоторые объекты везде означают одно и тоже, например, числа и ключевые слова вроде def и class. Однако значение большинства элементов зависит от контекста.

self и текущий объект

Один из краеугольных камней Руби – это текущий объект, доступный через ключевое слово self. В каждый момент работы программы есть только один self, узнать этот объект можно с помощью нескольких правил. Прежде всего необходимо знать контекст, типов его немного, вот они: высший уровень (top level), блоки определения классов, блоки определения модулей и блоки определения методов.

Высший уровень относится к коду, написанному вне каких-либо классов и модулей. Например, если мы откроем новый .rb файл и запишем только x=1, то мы создадим локальную переменную высшего уровня x. Если запишем

```
def m
end
```

то получим метод высшего уровня. В высшем уровне Руби предоставляет нам начальный (start up) self и, при попытке идентифицировать его, вернет main. main – это специальный термин, который использует объект self по умолчанию при обращении к самому себе.

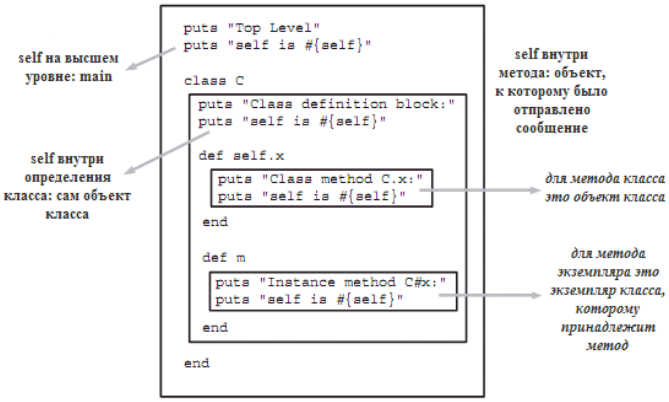
В определении класса или модуля self – сам объект класса, вот демо:

```
class C
    puts «Started class C:»
```

```
        puts self     # C
        module M
            puts «Module C::M:»
            puts self     # C::M
        end
        puts «Back C:»
        puts self    # C
    end
```

self внутри определения метода экземпляра хитрый, и вот в чем дело. Когда интерпретатор считывает блок def/end, он определяет метод незамедлительно. Однако код внутри метода исполняется только при его вызове объектом, который и будет self для этого метода.

Вот небольшая сводка поведения self:



БД в .txt

Многим приложениям необходимо хранить и манипулировать данными. Иногда для этого открываются файлы, делаются нужные изменения и результат выводится на экран либо сохраняется назад в файл. В некоторых ситуациях, однако, требуется база данных. База данных – это система для организации и хранения данных на компьютере в систематическом виде. База данных может быть простой как текстовый файл, который манипулируется самой программой или сложной, например, состоять из террабайтов данных, распределенных на выделенных серверах. Между прочим Руби можно использовать и в этом случае, однако мы рассмотрим вариант с самой простой БД.

База данных может представлять собой простой текстовый файл, как в случае с CSV (Comma-Separated Values), где для каждого пункта данных можно хранить атрибуты, разделенные запятыми. Давайте создадим файл db.txt, в который запишем данные CSV, например, такие:

```
Fred Bloggs,Manager,Male,45
Laura Smith,Cook,Female,23
Debbie Watts,Professor,Female,38
```

Каждая строка представляет собой отдельного человека и атрибуты каждого разделены запятыми.

Руби содержит стандартную библиотеку cvs, позволяющую использовать текстовые файлы, содержащие CSV данные, как простую базу данных, с которой легко манипулировать. Библиотека содержит класс CSV, который и поможет нам в работе. Загрузим данные из CVS файла в массив с помощью метода read класса CSV и прочитаем и изменим что-нить из массива:

```
require 'csv'
a=CSV.read('db.txt')
puts a.inspect
puts a[0][0] # Fred Bloggs
puts a[1][0] # Laura Smith
puts a[2][0] # Debbie Watts
a[1][1] = "Marine"
p(a[1]) # ["Laura Smith", "Marine", "Female", "23"]
```

Теперь, когда мы получили необходимые нам данные, нужно сохранить CSV обратно, модуль csv сделает все за нас:

```
CSV.open('bd.txt', 'w') do |csv|
    a.each do |a|
        csv << a
    end
end
```

Исключения

В любой программе случаются моменты, когда дела идут не по нужному пути: люди вводят неверные данные, файлы, на которые вы расчитываете не существуют (или у вас нет прав доступа к ним), заканчивается память и др. Есть несколько выходов из подобных ситуаций. Проще всего выйти из программы, когда что-то случилось не так.

Менее радикальное решение – каждый метод должен возвращать что-то вроде статусных данных, в которых указано, была ли обработка успешна, а затем необходимо протестировать полученные от метода данные. Однако тестирование каждого результата сделает код плохочитаемым.

Еще один альтернативный подход – использовать исключения. Когда что-то идет не так (т.е. появляется условие исключения) выдаются исключения. На высоком уровне в программе будет кусочек кода (обработчик исключений), который будет следить за появлением такого сигнала и реагировать на него определенным способом.

Также в одной программе может быть множество обработчиков исключений, каждый из которых обрабатывает определенные типы ошибок. Исключение проходит через все обработчики, пока не встретит требуемый, если его нет, то программа закрывается. Такое поведение есть в C++, Java и Руби.

Представим себе текстовый редактор. Пользователь должен ввести имя в диалоге SaveAs и нажать OK. Так как пользователь сам решает, какие данные вводить, мы не можем знать, имеет ли он права для записи этого файла, есть ли свободное место на диске.

Будем использовать исключения:

```
text = editor()
location = ask_user()
begin
    File.open(location, w) do |file|
        save_work(file, text)
end
rescue
    puts “Сохранение не удалось. Ошибка: #{$!}”
end
```

Теперь если что-то пойдет не так, то программа не завершит работу, данные не потеряются и у нас будет второй шанс. Все, что находится между begin и rescue защищено. Если появляется исключение, то контроль передается блоку между rescue и end. Глобальная переменная $! передает сообщение об ошибке и оно выводится на экран. Для того, чтобы контролировать только отдельные виды исключений, мы записываем из в rescue. Например, чтобы обрабатывать только ошибки при записи файла, используем выражение rescue IOError. Если мы хотим перехватывать несколько видов исключений в один обработчик, то перечисляем их через запятую, либо (что удобнее) написать обработчик для каждого вида:

```
rescue IOError
puts Не удалось записать на диск -- $!.
rescue SystemCallError
puts Произошла ошибка системного вызова -- $!.
end
```

DSL и динамические вкусности Ruby

iv_s

В этой статье я проиллюстрирую основные возможности Ruby для построения Domain Specific Languages(DSL). DSL, это небольшие, узкоспециализированные языки для решения конкретных задач. В отличие от языков общего назначения, таких как C++ или Java, DSL обычно очень компактны, и обладают высокой выразительностью в контексте решаемой задачи.

Различные DSL широко распространены в библиотеках и фреймворках для Ruby. Например в Rails DSL используются для создания миграций. А теперь, давайте посмотрим какие возможности Ruby предоставляет для построения DSL

Пусть нам нужен простой формат для описания комплектации компьютера. Простой пример:

```
Процессор: 2.2 гигагерц
Память: 1 гигабайт
Диск: 250 гигабайт
```

Теперь с помощью Ruby построим удобный DSL для таких описаний.

Этап 1

Трансформируем даное описание в Ruby код, например так(пусть память мы храним в мегабайтах а частоту в мегагерцах):

```
comp = Computer.new
comp.cpu = 2.2 * 1024
comp.ram = 2 * 1024
comp.disk = 1 * 1024
```

Код класса элементарный:

```
class Computer
    attr_accessor :cpu
    attr_accessor :ram
    attr_accessor :disk
end
```

В Ruby все переменные экземпляра(такие переменные начинаются с @) являются приватными, т.е. доступны только внутри методов объекта. Чтобы сделать аттрибут, мы должны объявить два метода для установки и получения значения этого аттрибута:

```
def cpu
    @cpu
end

def cpu= val
    @cpu = val
end
```

или проще, вызвать: attr_accessor :cpu, который и сгенерирует нам эти методы

Этап 2

Ну и что? ничего нового тут нет, просто используем объект с аттрибутами. Попробуем немного усовершенствовать наш код. Первое что бросается в глаза, мы должны самостоятельно переводить гигабайты в мегабайты и тд. Исправим!

```
comp = Computer.new
comp.cpu = 2.2.ghz
comp.ram = 2.gb
comp.disk = 1.gb
```

Для этого примешаем методы ghz и gb к классу Numeric:

```
class Numeric
    def ghz
        self*1000
    end
    def gb
        self*1024
    end
    def mhz
        self
    end
    def mb
        self
    end
end
```

Еще я добавил два метода mhz и mb, cpu.ram = 512.mb вместо cpu.ram = 512.

В Ruby есть возможность примешать(mixin) новый метод к любому классу. Т.е. мы можем расширять класс даже после его создания. После того как мы примешали метод к классу, он становится доступным для всех его экземпляров.

```
class String
    def cool
        self + « is cool!»
    end
end
```

В методе cool self — это указатель на значения самого объекта, а так как возвращаемое значение метода это результат выполнения его последней строки, то

```
puts «my string».cool
```

выведет на экран «My string is cool!»

Методы ghz и тд я примешал к классу Numeric, потому что это родительский класс для всех чисел в Ruby. И для целых и для дробных

Этап 3

Уже лучше. Но еще смущает тот факт, что перед каждым параметром мы должны указывать «comp.». Сделаем немного по-другому:

```
comp = Computer.new do
    cpu 2.2.ghz
    ram 2.gb
    disk 1.gb
end
```

Выглядит намного лучше, не правда ли? Но вопрос будет ли это работать? Давайте разберемся. На вид, это валидный Ruby код. cpu, ram и disk это уже не методы, а функции, так как вызываются не у экземпляра класса Computer. Что то наподобии этого:

```
def cpu val
    comp.cpu = val
end
```

Но как нам передать в эту функцию переменную comp? cpu 2.ghz, comp? но тогда потяряется вся выразительность. Вот если бы мы могли выполнить эти методы в контексте этого объекта… И ведь мы можем! Ruby дает нам такую возможность с помощью метода instance_eval.

Теперь посмотрим на новую реализацию класса Computer

```
class Computer
    #метод initialize это конструктор
    def initialize &block #&block означает что методу
передается блок кода
        instance_eval &block #вызываем волшебный метод
instance_eval и передаем ему блок
    end

    #тут я объявил методы вместо аттрибутов, чтобы вме-
сто cpu= 2.ghz писать cpu 2.ghz
    def cpu val
        @cpu_clock = val
    end

    def ram val
        @ram_size = val
    end

    def disk val
        @disk_size = val
    end

    #в установке значения теперь нет нужды, так как
есть методы cpu и тд
    #по этому я вызываю attr вместо attr_accessor, он
не будет генерировать метод для установки значения
    #но есть небольшое ограничение, т.к. в динамических
языках нельзя перегружать методы
    #(а attr_accessor по настоящему создает методы)
    #то для аттрибутов нужно выбрать другие имена
    attr :cpu_clock
    attr :ram_size
    attr :disk_size
end
```

Что же делает этот метод? Все очень просто. Как уже сказал выше, он выполняет блок(а можно и строку с кодом) в контексте данного объекта. А так как в классе Computer объявлены методы cpu и тд, то они и вызовутся. И именно для этого объекта.

Этап 4

Теперь представим, что у нашего компьютера неограниченно много различных характеристик. Например, размер BIOS'a или разрадность шины:

```
comp = Computer.new do
    bios 0.5.mb
    bus 100
end
```

Все мы предугадать не можем, но хочется чтобы была возможность добавления таких характеристик. И тут нам снова приходят на помощь возможности Ruby, а именно метод method_missing.

method_missing - специальный метод объекта, который вызывается при попытке вызвать несуществующий метод. Пример:

```
class Test
    def method_missing name, *args, &block #name - имя
метода, *args - его аттрибуты, &block - блок кода если
есть.
        puts name.to_s + « called»
    end
end

t = Test.new
t.some_method #напечатает «some_method called»
t.asdf #»asdf called»
```

Теперь вернемся к классу Computer:

```
class Computer
    def initialize &block
        instance_eval &block
    end
    def method_missing name, *args, &block
        instance_variable_set(«@#{name}».to_sym,
args[0]) #создаем переменную экземпляра и присваеваем
ей значение
        self.class.send(:define_method, name, proc {
instance_variable_get(«@#{name}»)}) #создаем метод для
доступа к этой переменной
    end
end
```

Теперь как это работает. Мы передаем блок конструктору нашего класса. Этот блок выполняется в контексте экземпляра этого класса. В процессе выполнения вызываются несуществующие методы, вместо них выполняется method_missing с параметром name равным имени метода и массивом аргументов args. Теперь мы создаем переменную экземпляра с именем, совпадающим с методом, и значением, равным первому аттрибуту вызванного метода. А также метод для получения значения этой переменной.

Кстати, method_missing используется в Rails в ActiveRecord, для создание тысяч методов типо Person.find_all_by_name

Получившийся код можно посмотреть тут.

Дальнейшие усовершенствования

Что еще можно придумать, чтобы наш DSL стал еще удобнее?
Так как DSL могут быть предназначены не только для программистов, но и для людей незнакомых с программированием, то логично вынести описания в файл, который может редактировать даже не программист. А потом загружать и интерпретировать этот файл.

my_pc.conf:

```
cpu 1.8.mgh
ram 512.mb
disk 40.gb
```

Сделать это очень просто, как я уже писал выше, методу instance_eval можно передать строку с кодом вместо блока.

Во-вторых, для фанатов русского языка, можно писать так:

```
comp = Computer.new do
процессор 2.2.ghz
память 2.gb
диск 1.gb
end
```

Чтобы это сработало, надо просто добавить параметр -Ku при вызове интерпретатора. Например так:

```
ruby -Ku test.rb
```

Выше мы построили простой DSL, который является валидным Ruby кодом. Но можно отказаться от валидности. Например избавиться от точки.
Вместо cpu 1.ghz писать cpu 1ghz. Тогда придется произвести небольшой препроцессинг. Добавить эти точки, например с помощью регулярных выражений.

А теперь, если мы скомбинируем эти улучшения, то мы сможем проинтерпритировать пример, который я дал в самом начале:

```
Процессор: 2.2 гигагерц
Память: 1 гигабайт
Диск: 250 гигабайт
```

Попробуйте сами.

Теперь немножко саморекламы. Когда я сам разбирался с этой методикой, я написал простую библиотеку с DSL для генерации валидного XHTML. Смысл в том, что если мы пытаемся написать чтото невалидное, то получаем ошибку прямо в процессе генерации. Я оформил ее как gem пакет, так что можно поставить так:
— в Windows: gem install rml
— в *nix системах: sudo gem install rml

Страница проекта: http://rubyforge.org/projects/rml/.
Примеры и документацию можно посмотреть: http://rml.rubyforge.org/.

Интересные приёмы в Ruby, которые вы можете использовать в своём коде

Kane, перевод

В этом посте я хочу представить вам 21 интересную особенность Ruby. Многие из этих особенностей могут быть не очевидны новичку, в то время как более опытный разработчик использует их каждый день. Для выполнения примеров кода удобнее всего использовать «интерактивный» Ruby — irb. Если же у вас ещё нет своего Ruby, то воспользуйтесь онлайн интерактивным интерпретатором. Итак, приступим.

1. Быстрое получение совпадений из регулярного выражения

Чтобы получить данные из текста используя регулярные выражения, как правило применяют метод match. Сейчас я покажу как использовать его наиболее эффективно:

```
email = «Fred Bloggs <fred@bloggs.com>»
email.match(/<(.*?)>/)[1]            # => «fred@bloggs.
com»
email[/<(.*?)>/, 1]                  # => «fred@bloggs.
com»
email.match(/(x)/)[1]                # => NoMethodError
[:(]
email[/(x)/, 1]                      # => nil
email[/([bcd]).*?([fgh])/, 2]        # => «g»
```

В комментариях мне делают замечание, что тема регулярных выражений не раскрыта. Раскрываю:

```
name, mail = *email.match(/(.+)\s<(.*)>/)[1..2]
puts name               # «Fred Bloggs»
puts mail               # «fred@bloggs.com»
```

2. Синоним для Array#join

Многие знают, что если использовать метод Array#* для «умножения» массива на число, то массив будет увеличен в заданное количество раз путём дублирования его элементов. Но не всем известно, что если вместо числа, «умножить» массив на строку, то это сработает как метод Array#join:

```
%w{this is a test} * «, «      # => «this, is, a, test»
h = { :name => «Fred», :age => 77 }
h.map { |i| i * «=» } * «\n»   # => «age=77\nname=Fred»
```

3. Быстрое форматирование десятичных чисел.

Если вы хотите сделать форматированный вывод, например, для указания цен, скорее всего будете использовать метод sprintf, но не спешите, Ruby предлагает более интересную возможность:

```
money = 9.5
«%.2f» % money                       # => «9.50»
```

4. Окружаем текст тегами

Конечно предыдущий пример можно использовать не только для чисел, но и для строк:

```
«[%s]» % «same old drag»          # => «[same old drag]»
```

И даже для массивов:

```
x = %w{p hello p}
«<%s>%s</%s>» % x                 # => «<p>hello</p>»
```

5. Удаляем директорию со всем содержимым

Нет-нет, не спешите писать код на bash, в Руби есть библиотека fileutils:

```
require 'fileutils'
FileUtils.rm_r 'somedir'
```

6. Разбиваем массивы и хэши

Можно использовать «*» чтобы «разбить» перечисляемые типы данных (массивы и хэши). Чтобы было понятно о чём идёт речь, вот несколько примеров:

```
a = %w{a b}
b = %w{c d}
[a + b]                      # => [[«a», «b», «c», «d»]]
[*a + b]                     # => [«a», «b», «c», «d»]

a = { :name => «Fred», :age => 93 }
[a]                 # => [{:name => «Fred», :age =>93}]
[*a]                # => [[:name, «Fred»], [:age, 93]]

a = %w{a b c d e f g h}
b = [0, 5, 6]
a.values_at(*b).inspect          # => [«a», «f», «g»]
```

Получаем из массива хэш:

```
fruit = [«apple»,»red»,»banana»,»yellow»] # [«apple»,
«red», «banana», «yellow»]
Hash[*fruit]   # {«apple»=>»red», «banana»=>»yellow»}
```

7. Убираем объявление переменных.

Вместо объявления переменных с каким-то, зача-

стую пустым, начальным значением, можно это сделать «на лету». Таким образом можно совместить объявление переменной и какие-то манипуляции над ней. Приём особенно полезен, когда не известно, была ли объявлена эта переменная раньше:

```
(z ||= []) << 'test'
```

8. Использование не строковых ключей хэша

Такое применение можно встретить не часто, но иногда это может оказаться действительно полезно:

```
does = is = { true => 'Yes', false => 'No' }
does[10 == 50]                        # => «No»
is[10 > 5]                            # => «Yes»
```

9. Логические and и or могут объединить операторы в одну строку

Многие разработчики используют это приём чтобы сократить код, убрав ненужное разбиение на строки:

```
queue = []
%w{hello x world}.each do |word|
queue << word and puts «Added to queue» unless word.
length <  2
end
puts queue.inspect

# Output:
#   Added to queue
#   Added to queue
#   [«hello», «world»]
```

10. Выполнение кода, только если если файл запущен непосредственно

Вы можете использовать один и тот же код как в качестве самостоятельного скрипта, так и библиотеки. Тогда у вас может возникнуть необходимость выполнять части кода только если они использованы как самостоятельный скрипт или наоборот, подключаются с помощью require:

```
if __FILE__ == $0
# Этот участок будет выполнен, только если файл выполнен как самостоятельная программа
end
```

11. Быстрое массовое присвоение

Что может быть проще:

```
a, b, c, d = 1, 2, 3, 4
```

12. Диапазоны вместо нескольких сравнений

Больше никаких if x > 1000 && x < 2000, теперь можно писать так:

```
year = 1972
puts  case year
when 1970..1979: «Семидесятые»
when 1980..1989: «Восьмидесятые»
when 1990..1999: «Девяностые»
end
```

13. Перечисляемые структуры вместо повторения кода

```
%w{rubygems daemons eventmachine}.each { |x| require x
}
```

14. Тернарный оператор

О тернарном операторе обычно узнают на ранних этапах изучения Руби, но почему-то используют не часто. Тернарный оператор — не панацея, но иногда способен сделать многое проще:

```
puts x == 10 ? «x равно десяти» : «x не равно десяти»
# Конечно, можно сделать и присвоение:
LOG.sev_threshold = ENVIRONMENT == :development ?
Logger::DEBUG : Logger::INFO
```

15. Несколько тернарных операторов

Безусловно, это может вызвать проблемы с читаемостью кода, но всё же:

```
qty = 1
qty == 0 ? 'none' : qty == 1 ? 'one' : 'many'
```

Да, немного запутанно… Так будет понятнее:

```
(qty == 0 ? 'none' : (qty == 1 ? 'one' : 'many'))
```

16. Избавляемся от избыточных логических выражений

Частенько можно увидеть что-нибудь вроде этого:

```
def odd?(x)
  if x % 2 == 0
    return false
  else
    return true
  end
end
```

Но это слишком длиииино! Воспользуемся тернарным оператором:

```
def odd?(x)
  # Кстати, return можно опустить
  x % 2 == 0 ? false : true
end
```

Уже неплохо, но это тоже длинно, зачем нам использовать тернарный оператор, если «==» и так возвращает true или false:

```
def odd?(x)
  x % 2 != 0
end
```

17. Полный backtrace исключения

```
def do_division_by_zero; 5 / 0; end
begin
  do_division_by_zero
rescue => exception
  puts exception.backtrace
end
```

18. Перебираем в итераторе как одиночные объекты, так и перечисляемые

```
[*items].each do |item|
  # ...
end
```

В этом примере, если items — одиночный объект, то он будет преобразован в массив, благодаря заключению в «[]», если же это массив, то с ним ничего не произойдёт, потому что «[]» и «*» скомпенсируют друг-друга.

19. Перед блоком rescue не обязательно писать begin

```
def x
  begin
    # ...
  rescue
    # ...
  end
end


def x
  # ...
rescue
  # ...
end
```

20. Блоки комментариев

Для комментирования можно использовать не только «#», но и такую конструкцию:

```
puts «x»
=begin
  this is a block comment
  You can put anything you like here!

  puts «y»
=end
puts «z»
```

21. Rescue для спасения

Можно использовать однострочную форму rescue когда важно во что бы то ни стало вернуть значение:

```
h = { :age => 10 }
h[:name].downcase                       # ERROR
h[:name].downcase rescue «No name»      # => «No
name»
```

Бонус 22. Параметр, заданный по-умолчанию

Как можно опознать, задан ли параметр метода вызывающей программой или было взято значение по-умолчанию. Один из возможных вариантов:

```
def foo(a, b=(flag=true; 666))
  puts «b равно #{b}, flag равно #{flag.inspect}»
end

foo 100      # b равно 666, flag равно true
foo 100, 666 # b равно 666, flag равно nil
foo 100, 400 # b равно 400, flag равно nil
```

Бонус 23. Переводим из 10 системы счисление в любую другую с базисом от 2 до 36

```
1234567890.to_s(2)  # «1001001100101100000001011010010»
1234567890.to_s(8)  # «11145401322»
1234567890.to_s(16) # «499602d2»
1234567890.to_s(24) # «6b1230i»
1234567890.to_s(36) # «kf12oi»
```

Ruby on Rails шаг за шагом

MaxElc

Целью первой части уроков по Ruby on Rails будет создание некоторого многопользовательского блога (аля Хабр). Также хочется отметить, что для этой первой части желательно иметь познание о Руби хотя бы на начальном уровне. Хочется поскорей приступить к кодингу, но начинать все равно придется с теории.

Что такое Ruby on Rails (далее RoR)? Самый распространненый ответ – это базирующийся на ЯП Ruby (далее Руби) фрэймворк, который реализует шаблон (далее паттерн) MVC. Выделим два главных пункта из ответа:

— Это фрэймворк на базе Ruby
— Он реализует паттерн MVC

Разберем каждый отдельно.

RoR – базирующийся на Руби фрэймворк

ЯП Руби – простой и мощный, возможность мета-программинга, блоков, итераторов, а также обработки исключений делает язык замечательной основой для фрэймворка. Собственно так и посчитал Дэвид Хэйнемеер Ханссон, создатель RoR, и в июле 2004 года фрэймворк вышел на свет. Увидеть, как помогает мета-программирование Руби, можно увидеть в ORM компоненте RoR Active Record. Основываясь на имени класса, RoR считывает схему (schema) и налету создает объекты класса на базе таблицы БД. Можно перейти к следующему аспекту.

RoR реализует MVC

Возожно, вы уже слышали о MVC в отношении других фрэймворков, однако что представляет собой MVC в RoR? MVC – это паттерн архитектуры приложения, четко разделяющий три его компонента:

— Model (далее Модель) является «сутью» приложения и отвечает за непосредственные алгоритмы, расчёты и тому подобное внутреннее устройство приложения. Также предоставляет линк к хранилищу данных.
— View (Представление, дальше Вид) предназначен для вывода данных, предоставленных Моделью. Это единственная часть MVC, которая непосредственно контактирует с пользователем.
— Controller (Поведение, далее Контроллер) получает данные от пользователя и передаёт их в Модель. Кроме того, он получает сообщения от Модели и передаёт их в Вид.

Вот схема MVC (рисунок №1).

Исходя из этого RoR использует три компонента:

— Active Record
— Action View
— Action Controller

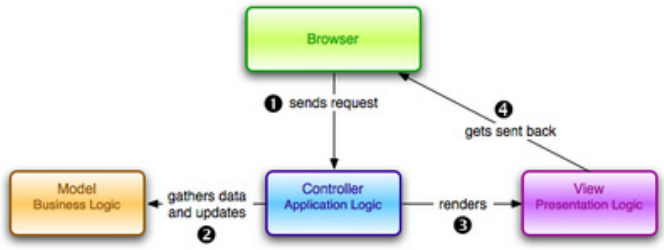


Рисунок №1

Сочетание последних двух известно как Action Pack. Рассмотрим каждый компонент поближе, и узнаем, почему Руби так подходит для реализации фрэймворка.

Active Record

Active Record – это Модель в RoR. Модель хранит данные и предоставляет базу для работы с данными. Кроме этого Active Record также является ORM фрэймворком. ORM значит Object-relational mapping (Объектно-реляционная проекция). Собственно Active Record делает следующие вещи:

— **Проекция таблицы на класс.** Каждая таблица проецируется на один или несколько классов по принципу convention over configuration (соглашение выше конфигурации). Одно из таких соглашений – имя таблицы должно быть во множественном числе, а название класса – в единственном. Атрибуты таблицы налету проецируются в атрибуты экземпляра Руби. После того, как все проекции сделаны, каждый объект ORM класса представляет определенную строку таблицы, с которой класс был спроецирован.
— **Соединение с БД.** Вы можете подключиться к базе данных, используя API, предоставляемый Active Record, который создает необходимый вам запрос непосредственно в движок БД при помощи адаптеров. У Active Record есть адаптеры для MySQL, Postgres, MS SQLServer, DB2, и SQLite. Необходимо лишь записать параметры доступа к БД в файле database.yml.
— **Операции CRUD.** Это операции create (создание), retrieve (получение), update (обновление) и delete (удаление) над таблицей. Так как Active Record – это ORM фрэймворк, вы всегда работаете с объектами. Чтобы создать новую строку таблицы, вы создаете новый объект класса и заполняете его переменные экземпляра значениями. Стоит заметить, что все это Active Record делает за вас.
— **Проверка данных.** Проверка данных перед по-

мещением их в таблицу – это первый шаг в безопасности вашего проекта. Active Record предоставляет проверку Модели. Данные могут быть проверены автоматически с помощью множества готовых методов, которые, в случае необходимости, можно переписать под собственные нужды.

Action View

Вид включает в себя логику, необходимую для вывода данных Модели. Роль Вида в RoR играет Action View. Наилее часто используемые функции Action View:

— **Шаблоны (Templates).** Шаблоны – это файлы, содержащие заполнители (placeholders), которые буду заменены на контент. Шаблоны могут содержать HTML-код и код Ruby, встраиваемый в HTML с использованием синтакса встроенного (embedded) Ruby (ERb).
— **Помощники** (helper, далее хелпер) **форм и форматирования**. Хелперы форм позволяют создавать такие элементы страниц, как чекбоксы, списки, используя готовые методы. В свою очередь хелперы форматирования позволяют форматировать данные необходимым нам способом, методы существуют для дат, валют и строк.
— **Макет**. Макеты (layouts) определяют, как контент будет расположен на странице. Динамически создаваемая страница может содержать вложение из нескольких страниц, даже без использования таблиц и фрэймов, используя API Макета.

Action Controller

В веб-приложении Контроллер регулирует поток логики приложения. Он находится на границе программы, перехватывая все запросы, на основе которых он изменяет какой-то объект Модели и вызывает Вид, чтобы отобразить обновленные данные. В RoR Action Controller является Контроллером, вот его основные функции:

— **Поддержка сессий.** Сессия – это период времени, проведенный пользователем на сайте. Его можно отследить с помощью cookie или объекта сессии. Cookie – небольшой файл, он не может содержать объекты, в отличие от объекта сессии.
— **Фильтрация.** Бывают ситуации, когда необходимо вызвать определенный код, перед тем как исполнять логику Контроллера или после него, например, аутентификация пользователей, логирование событий, предоставление персонального ответа. Помогают в таких случаях фильтры, предоставляемые Action Controller. Существуют три основных фильтра: before, after и around. О них – позже.
— **Кэширование.** Кэширование – это процесс, при котором наиболее запрашиваемый контент сохраняется в кэше, чтобы не было необходимости запрашивать его вновь и вновь.

Три среды

RoR поощряет использование отдельных сред для каждого из этапов цикла жизни приложения: разработка (development), тестирование (testing) и эксплуатация (production), для каждого из которых создается отдельная БД. Рассмотрим каждую среду.

— **development.** В среде разработки ставка делается на немедленное отображение нового варианта при изменении кода – достаточно обновить страницу в браузере. Скорость в этой среде не важна. Когда случается ошибка, она выводится на экран.
— **test.** При тестировании мы обычно каждый раз наполняем БД каким-нибудь глупым текстом, чтобы убедиться, что нормальное поведение не зависит от содержания БД. Процедуры юнит-тестинга и теста функциональности в RoR автоматизированы и производятся через консоль. Тестовая среда предоставляет отдельное пространство, в которых оперируют эти процедуры.
— **production.** В конце концов ваше приложение выходит к финальной черте, пройдя тесты и избавившись от багов. Теперь обновления кода будут происходить редко и можно сконцентрироваться на производительности, включить кэширование. Нет необходимости писать огромные логи ошибок и пугать пользователей сообщениями об этих ошибках в браузере. Для вас – среда production.

Instant Rails

Установка рабочего комплекта для Windows максимально проста, а все из-за того, что один щедрый программист по имени Курт Хибс уже собрал для нас все необходимое для разработки приложений RoR в одном пакете под название Instant Rails. В пакеты собраны:

— интерпретатор Ruby,
— движки БД SQLite и MySQL,
— веб-сервер Apache (который, собственно, нам не нужен…)
— и сами Rails

Скачаем Instant Rails (IR) с официального сайта, распакуем содержимое архива в любую папку (лучше, если расположить ближе к корню диска, а также в пути не должно быть пробелов, рекомендую C:\ruby). В архиве около 18000 файлов, так что придется немного подождать. После запускаем InstantRails.exe и в появившемся запросе нажимаем ОК для того, чтобы прописались пути – вообще само это приложение нам не нужно – мы будем использовать специальный веб-сервер, написанный на Руби, а в качестве БД нам будет служить SQLite. Спасибо, Курт, за старания, но мы останавливаем апач и мускул и закрываем все это добро.

Однако при загрузке комплекта вы, скорее всего, заметили, что последняя его версия датируется аж концом позапрошлого года, так что нам необходимо обновлять компоненты. Прежде всего, устарел фрэймворк RoR – за его обновление мы и возьмемся. Как это сделать? На помощь приходит gem.

gem

Все дополнения, библиотеки и программы для Руби находятся во всемирном репозитории Ruby Application Archive (RAA), в котором уже находятся более 1700 проектов. Для установки (как удаленно, так и локальной) и контроля этих дополнений как раз и используется программа gem. Добро пожаловать в консоль (Пуск->Все программы->Выполнить->вводим cmd->OK). В ней переходим в папку с IR (cd c:/ruby) и запускаем use_ruby (use_ruby.bat) – этот bat файл пропишет нам необходимые пути в PATH, что сделает работу с RoR еще удобнее, что сделает работу с RoR еще удобнее. Работать мы будем всегда только через use_ruby.bat, поэтому не забываем запускать его. Командуем: gem list и видим список всех установленных «джемов». Команда gem update должна нам обновить их все, однако этого не произойдет, потому что система обновлений уже устарела, не оставив обратной совместимости. Поэтому порядок команд такой (для того, чтобы не путаться, мы сразу удалим старые версии джемов):

— gem install rubygems-update
— update_rubygems
— gem uninstall rails activerecord activeresource activesupport actionpack actionmailer
— gem install rails --include-dependencies

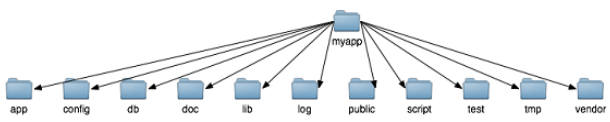
Думаю, что тут все интуитивно понятно и объяснять, что, как и почему, не стОит. Дальше ждем, пока все компоненты загрузятся и установятся (особенно долго создается документация, но нам спешить некуда).

Установка «с нуля»

Если вы любите брать все под свой контроль или у вас уже установлен MySQL, то для вас подойдет развертывание среды вручную. Это совсем несложно, начнем с офсайта Ruby и скачаем пакет Ruby 1.8.6 One-Click Installer — он содержит дополнительные библиотеки под Windows, устанавливаем все по умолчанию. Затем заходим в Пуск — Ruby-186-xx — Ruby Gems — Ruby Gems package manager и здесь, как и было описано выше, устанавливаем необходимые джемы (rails --include-dependencies, mysql, mongrel).

Структура приложения

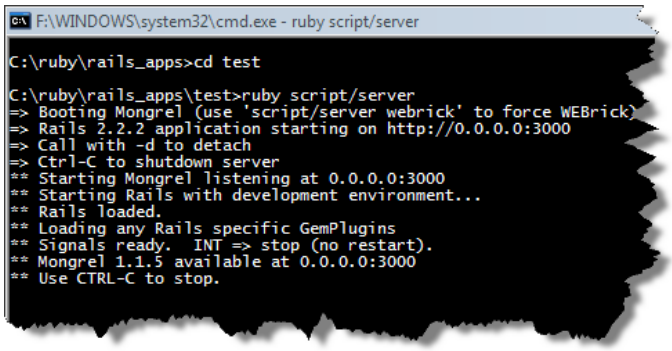
Как мы уже говорили, RoR должен следовать определенным соглашениям. Одно из них – приложение на RoR всегда имеет строгую структуру папок. Вот ее первый уровень:



Но не торопитесь создавать папки – как обычно, RoR делает рутинную работу за нас.

Тестовое приложение

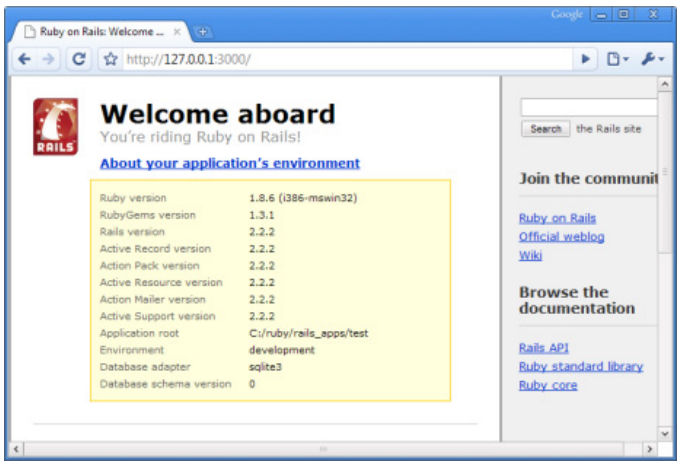
При инсталляции RoR было установлено и приложение rails, которое будет генерировать структуру папок для приложения и создавать базовые файлы с кодом. У нас по-прежнему должна быть открыта «рельсовая консоль», с приглашением в папке rails_apps (в ней и будут находиться приложения). Мы создадим приложение test, командуем: rails test. Ждем окончания генерации. Как видим, была создана отдельная папка, в которой и поместилась вышеупомянутая структура. В следующий раз мы узнаем, для чего нужна каждая папка, а пока просто запустим веб-сервер Mongrel, чтобы посмотреть результат. В консоли перейдем в папку с приложением (cd test) и скомандуем: ruby script/server. В результате и будет запущен Mongrel:



Как видим, сервер поселился на 3000 порту. Что ж, открываем браузер и переходим на

```
http://127.0.0.1:3000/
```

– и тут нас уже ждет тестовая страница с приветствием. Да, вы уже ездите по Рельсам вместе с Руби ;) Нажав на “About your application’s environment”, вы узнаете параметры вашего окружения:



Вот и первое «demo» у нас уже есть. Дальше опять вернемся и углубимся в теорию — расскажу об особенностях RoR, как фрэймворка на Руби, ЯП с возможностью мета-программинга, узнаем о принципах REST и начнем проектировать наш многопользовательский блог.

Как мы уже говорили, приложение RoR имеет строго определенную структуру, давайте посмотрим, для чего предназначены его папки:

— app: динамическая часть вашего приложения
  controllers: интерфейс и деловая логика
  helpers: код для поддержки Видов
  models: объекты Модели, соотносящиеся с БД и используемые в приложении
  views: шаблоны для отсылки удаленному пользователю
  layouts: файлы макетов страниц
— config: настройки среды, базы данных и приложения
  enviroments: конфигурации, относящиеся к средам
  initializers: конфигурационные файлы, обрабатывающиеся при запуске сервера
— db: схема базы данных в виде миграций Active Record
— doc: документация вашего веб-приложения
— lib: библиотеки Руби, относящиеся к приложению
— log: файлы логов
— public: статическая часть вашего приложения
  stylesheets: CSS файлы
  javascripts: файлы JavaScript
  images: файлы изображений
— script: Shell-скрипты для управления проектами RoR
— test: код и данные для тестирования вашего приложения
  fixtures: даные, загружаемые в БД при тестировании
  functional: фунциональные тесты для проверки Контроллеров
  integration: интеграционные тесты
  unit: Юнит-тесты для проверки Моделей
— tmp: здесь Rais хранит временные данные
— vendor: внешние библиотеки, используемые приложением
  plugins: плагины RoR

Доброй традицией стало “Hello World” в качестве примера работы любого ЯП или движка. Что ж, давайте сделаем страничку, которая будет приветствовать нас. Мы знаем, что приложение на RoR должно состоять из трех компонентов: Вида, Контроллера и Модели. Сказать привет – это просто, достаточно передать браузеру HTML-код. Несложно догадаться, что HTML должен содержаться в Виде, однако RoR не позволяет создавать виды напрямую, так как Вид должен быть непосредственно ассоциирован с Контроллером. Чтобы сгенерировать что-нибудь в RoR, нужно идти в командную строку (мы уже знаем, как это делается) и переместиться в папку приложения. Тут мы и командуем:

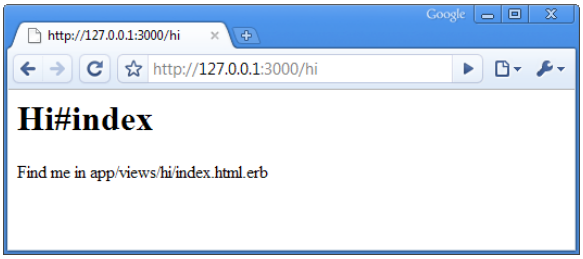
```
ruby script/generate controller Hi index
```

script/generate – это Shell-скрипт generate в папке script, написанный на Руби, поэтому запускать скрипт нужно через интерпретатор (для этого мы добавила ruby в начале команды). Так же мы передали скрипту два аргумента: controller определяет, что необходимо сгенерировать код Контроллера, в данном случае он будет назваться Hi – это говорит второй аргумент. А index в конце команды привяжет к контроллеру Вид index, что, собственно, нам и требовалось. Немого подождем, и вот что выдаст нам скрипт:

```
exists   app/controllers/
exists   app/helpers/
create   app/views/hi
exists   test/functional/
create   app/controllers/hi_controller.rb
create   test/functional/hi_controller_test.rb
create   app/helpers/hi_helper.rb
create   app/views/hi/index.html.erb
```

Строки, начинающиеся с exists, показывают папки и файлы, который генератор хотел создать, но они уже были на месте. Зная о структуре папок, мы уже можем сказать, что были созданы: папка hi в Видах, Контроллер, шаблон для создания тестов, хэлпер и файл Вида index.html.erb (формат .html.erb – это сочетание HTML и кода Руби).

Теперь index стал доступен для приложения. Давайте запустим сервер и заглянем в `http://127.0.0.1:3000/hi/`:



Немногословно, но страничка подсказывает нам, что файл находится в app/views/hi/index.html.erb. Открыв его, вы заметите, что в коде даже нет базовой структуры HTML. Давайте немного дополним файл и приведем его в более порядочный вид:

```
<html>
<head><title>Hi Habrahabr!</title></head>
<body>
<h1>Hello!</h1>
<p>Это приветствие прибыло из app/views/hi/index.html.
erb</p>
</body>
</html>
```

Сохраним файл и обновим страничку:

Наверное, все эти генерации множества папок выглядят достаточно сомнительно для того, чтобы создать заурядную HTML страничку. Для того, чтобы начать использовать силу RoR, поработаем с Контроллером app/controllers/hi_controller.rb. Вот, что в нем находится:

```
class HiController < ApplicationController
def index
end

end
```

Мы с вами уже знаем Руби, поэтому кодом нас не испугаешь: объявлены класс HiController (заметим, что название класса было сгенерировано на основе имени Контроллера — это CoC и мы еще будем говорить почему и зачем), который является потомком класса ApplicationController и пустой метод index. Давате пофантазируем с методом, задавая в нем переменные и отображая их в Виде. Для этого в RoR используются переменные экземпляра (те самые, который начинаются с @):

```
class HiController < ApplicationController
def index
     @habr = 'Habrahabr'
     @message = 'Это сообщение пришло из Контроллера'
end
end
```

Соответственно отредактируем Вид, чтобы показать переменные:

```
<html>
<head><title>Hi <%= @habr %>!</title></head>
<body>
<h1>Hello!</h1>
<p>Это приветствие прибыло из app/views/hi/index.html.
erb</p>
<p><%= @message %></p>
</body>
</html>
```

И тут мы видим, как код Руби внедряется в HTML. Для вывода значения переменной в HTML-код мы используем тэги <%= … %> – это так называемое выражение (expression).

Для того, чтобы получить текущее время в Руби используется метод Time.now. Достаточно вставить в HTML <%= Time.now %>, однако стоит использовать преимущества MVC и делать расчеты в Контроллере. Попробуйте “считать” время в Контроллере и отбразить его на странице Вида.

Логика в Виде

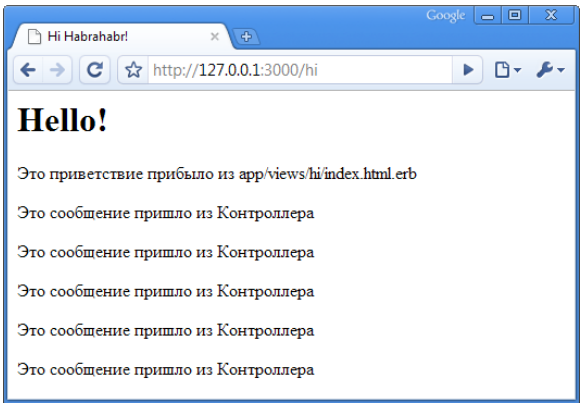
Мы также можем добавлять некоторую логику в файлы Вида, например, чтобы создавать списки, мы можем использовать итераторы. Допустим, мы хотим вывести переменную @message пять раз в параграфах. Мы знаем, как сделать это на чистом Руби:

```
5.times do
puts «<p> #{@message} </p>» # немного неверно, но неважно
end
```

Осталось только переделать код в формат .erb, получается даже несколько проще:

```
<% 5.times do%>
<p><%= @message %></p>
<% end %>
```

puts заменили тэги <%= … %>. А когда вывод кода нам не нужен мы просто опускаем = в тэгах. Вот что у нас получилось.



Как это работает?

Когда запускается код, RoR интерпретирует запрос 127.0.0.1:3000/hi/ как вызов Контроллера Hi. У RoR есть редактируемый список правил роутинга запросов, по умолчанию первая часть запроса — имя Контроллера, вторая — метод в нем. Опять же, по умолчанию вызывается метод index, поэтому мы не уточняли метод в запросе. Метод определяет необходимые базовые переменные. На этом работа Контроллера закончена и RoR передает данные в Вид. Как он знает, в какой Вид нужно передать данные? Работают conventions — магия договоренностей об именовании.

Эпилог

Мы поверхностно рассмотрели взаимодействие Контроллера и Вида, узнали, как оно проходит и почему работает, научились разделять логику между Видом и Контроллером. Мы еще углубимся в эту тему — здесь много интересных и важных подробностей, но чуть позже. Комментарии приветствуются в больших количествах!

Статистика

Владимир Юнев

Работая в web и создавая в нем проекты, всегда интересно знать, что представляет собой средний пользователь интернета. Эта страничка будет содержать небольшую статистику по тому, как представлен наш читатель во всемирной паутине.

Операционные системы пользователей

Windows	84.87%
Linux	10.09%
Macintosh	4.54%
Unknown	0.32%
BSD	0.12%
Symbian OS	<0.1 %
Sun Solaris	<0.1 %
Unknown Unix system	<0.1 %
OS/2	<0.1 %

Браузеры пользователей

Firefox	56.12%
Opera	23.9%
Chrome	11.16%
MS Internet Explorer	5.49%

RSS-readers

Google Feedfetcher	70%
Opera RSS Reader	12%
FeedDeamon	4%
Windows RSS Platform	4%
Thunderbird	1%
NetNewsWire	1%

Страны

Russia	63.43%
Ukraine	20.74%
Belarus	5.86%
Kazakhstan	1.83%
Germany	1.13%
(not set)	0.88%
United States	0.76%
Latvia	0.76%
Moldova	0.69%
Estonia	0.63%

Все статьи в настоящем издании публикуются с ресурса Хабрахабр, с любезного согласия авторов:

adrianopol, Kotter, MaxElc, iv-s, alek-sys, Kane, depp, sunnybear, RollingStone, kmike, uj2, Skaizer, Panya, smira, butaji, acerv, stboris

Орфография и пунктуация авторская с небольшими правками.

Проект habradigest

Владимир "XaocCPS" Юнев - автор, редактор, верстка, сайт

Владимир "OnTheFly" Синельников - дизайн, хостинг, домен

habradigest, январь 2009