февраль 2009
habradigest
web * дизайн * разработка
№8

habradigest

web * дизайн * разработка

Добрый день, читатель. Перед вами восьмой выпуск журнала habradigest. Этот номер неоднозначный. Во-первых, он небольшой, во-вторых, особенной темы, которую можно было бы выделить, в нем нет.

В целом выпуск, хоть и получился небольшим, но все равно содержит интересный и разнообразный материал. И мне остается только надеяться на то, что в марте авторы habrahabr проявят большую активность и напишут больше материала, чем написали в феврале.

Небольшому выпуску - небольшое вступление. До следующего номера, читатель.

Владимир "ХаосCPS" Юнев

HASKELL →

Классы типов, монады 2

WEB-РАЗРАБОТКА →

Кроссбраузерная одноцветная полупрозрачность 8

RUBY →

3 простых совета, которые сделают ваше Rails приложение быстрее 10

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ →

DNS Amplification (DNS усиление) 12

TWISTED FRAMEWORK →

Асинхронное программирование: концепция Deferred 15

Deferred: все подробности 18

JAVASCRIPT →

ActionWeb. Асинхронный интернет 21

.NET →

LINQ to SQL: паттерн Repository 24

DRUPAL →

Безопасный код в Друпале: Подделка межсайтовых запросов 28

Безопасный код в Друпале: Работа с базой данных 30

ПОСЛЕДНЯЯ СТРАНИЦА →

Статистика 33

Классы типов, монады

voidex

Темой сегодняшней статьи будут классы типов, некоторые стандартные из них, синтаксический сахар с их использованием и класс монад. Классы привносят динамический полиморфизм, как и интерфейсы в традиционных императивных языках, а также могут быть использованы как замены отсутствующей в Хаскеле перегрузки функций. Я расскажу, как определить класс типов, его экземпляры (instance) и как это всё устроено внутри.

Классы типов

Предположим, вы написали свой тип данных и хотите написать для него оператор сравнения. Проблема в том, что перегрузки функций в Хаскеле нет и потому использовать для этих целей (==) простым способом не получится. Более того, придумывать для каждого типа новое имя — не вариант. Но вы можете определить класс типов «сравниваемый». Тогда, если ваш тип данных принадлежит к этому классу, значения этого типа можно будет сравнивать.

```
class MyEq a where
        myEqual :: a -> a -> Bool
        myNotEqual :: a -> a -> Bool
```

MyEq — это имя класса типов (как и типы данных, оно должно начинаться с заглавной буквы), a — некий принадлежащий к данному классу тип. Параметров у класса может быть несколько (FooClass a b c), но в данном случае только один. Тип a принадлежит к классу MyEq, если для него определены соотстветствующие функции. В классе можно дать определение функции по умолчанию. Например, функции myEqual и myNotEqual могут быть выражены друг через друга:

```
myEqual x y = not (myNotEqual x y)
myNotEqual x y = not (myEqual x y)
```

Такие определения приведут к бесконечной рекурсии, но в экземпляре класса достаточно определить хотя бы одну из них. Теперь напишем экземпляр класса для `Bool`:

```
instance MyEq Bool where
        myEqual True True = True
        myEqual False False = True
        myEqual _ _ = False
```

Определение экземпляра начинается с ключевого слова instance, затем вместо переменной типа a в определении самого класса мы пишем тот тип, для которого определяется экзмепляр, т.е. Bool. Определяем одну функцию myEqual и теперь можно проверить в интерпретаторе результат:

```
ghci> myEqual True True
True
ghci> myNotEqual True False
True
ghci> :t myEqual
myEqual :: (MyEq a) => a -> a -> Bool
```

Видим, что тип функции `myEqual` накладывает ограничение (constraints) на тип — он должен принадлежать к классу `MyEq`. Такие же ограничения можно накладывать и при объявлении самого класса:

```
class (MyEq a) => SomeClass a where
-- ...
```

Классы чем-то похожи на интерфейсы — мы объявляем фукнции, для которых потом предоставляем реализации. Другие функции могут использовать эти функции только если для некоторого типа есть такая реализация. Однако есть существенные отличия как в возможностях, так и в реализации самого механизма:

1. Как видно по функции `myEqual`, она принимает два значения типа `a`, тогда как виртуальная функция принимает только один скрытый параметр `this`. Даже если у нас есть `instance MyEq Bool` и `instance MyEq Int`, вызов функции `myEqual True 5` приведёт к неудаче:

```
ghci> myEqual True (5::Int)

<interactive>:1:14:
Couldn't match expected type `Bool' against inferred
type `Int'
In the second argument of `myEqual', namely `(5::Int)'
In the expression: myEqual True (5 :: Int)
In the definition of `it': it = myEqual True (5 :: Int)
ghci>
```

Компилятор (интерпретатор) знает, что параметры myEqual должны иметь один тип и потому предотвращает такие попытки.

2. Экземпляр класса может быть определён в любой момент, что также очень удобно. Наследование от интерфейсов же указывается при определении самого класса.

3. Функция может потребовать принадлежность типа данных сразу к нескольким классам:

```
foo :: (Eq a, Show a, Read a) => a -> String -> String
```

Как это сделать, например, в C++/C#? Создавать композитный IEquableShowableReadable? Но от него не отнаследуешься. Передавать аргумент три раза с приведением к разным интерфейсам и полагать вну-

три функции, что это один и тот же объект, а ответственность лежит на вызывающей стороне?

Раз уж упомянул C++, то заодно скажу, что в новом стандарте C++0x concept и concept_map есть суть классы типов, но используемые на этапе компиляции.

Но есть и недостаток. Как нам завести список объектов, которые, к примеру, принадлежат к классу Show (функция show :: a -> String)? Способ есть, но он нестандартен. В начало файла необходимо добавить соответствующие опции GHC:

```
{-#
OPTIONS_GHC
-XExistentialQuantification
#-}
-- Не знаю, почему, но при выравнивании TAB'ом GHC опции игнорировал
```

Теперь мы можем объявить такой тип данных:

```
data ShowObj = forall a. Show a => ShowObj a
```

(Прочитать подробнее про existential types)
И заодно определить для него экземляр класса `Show`, чтобы он сам также к нему принадлежал:

```
instance Show ShowObj where
show (ShowObj x) = show x
```

Проверим:

```
ghci> [ShowObj 4, ShowObj True, ShowObj (ShowObj 'x')]
[4,True,'x']
```

Хотя функцию `show` я явно не вызывал, интерпретатор использует именно её, так что можно быть уверенным, что всё работает.

Как это всё реализуется?

В функцию, которая накладывает ограничения на тип, передаётся скрытый параметр (на каждый класс и тип — свой) — словарь со всеми необходимыми функциями. Фактически, `instance` — это определение экземпляра словаря для конкретного типа. Чтобы было проще это понять, я просто приведу псевдореализацию для класса `Eq` на Haskell'е и на C++:

```
-- class MyEq
data MyEq a = MyEq {
myEqual :: a -> a -> Bool,
myNotEqual :: a -> a -> Bool}
-- instance MyEq Bool
myEqBool = MyEq {
myEqual = \x y -> x == y,
myNotEqual = \x y -> not (x == y)}
-- foo :: (MyEq a) => a -> a -> Bool
foo :: MyEq a -> a -> a -> Bool
foo dict x y = (myEqual dict) x y
-- foo True False
fooResult = foo myEqBool True False
```

То же самое на C++:

```
#include <iostream>

// class MyEq a
class MyEq
{
public:
virtual ~MyEq() throw() {}
// принимаем void const *, так как сам тип в базовом классе неизвестен
virtual bool unsafeMyEqual(void const * x, void const * y) const = ;
virtual bool unsafeMyNotEqual(void const * x, void const * y) const = ;
};

// Шаблонная обёртка, знающая о типе и потому определяющая
// безопасные виртуальные функции
template <typename T>
class MyEqDictBase : public MyEq
{
virtual bool unsafeMyEqual(void const * x, void const * y) const
{ return myEqual(*static_cast<T const *>(x), *static_cast<T const *>(y)); }
virtual bool unsafeMyNotEqual(void const * x, void const * y) const
{ return myNotEqual(*static_cast<T const *>(x), *static_cast<T const *>(y)); }
public:
virtual bool myEqual (T const & x, T const & y) const {
return !myNotEqual(x, y); }
virtual bool myNotEqual (T const & x, T const & y)
const { return !myEqual(x, y); }
};

// Экземпляры классов. Определяться будут через специализацию.
template <typename T>
class MyEqDict;

// Создать словарь для определённого экземпляра класса
template <typename T>
MyEqDict<T> makeMyEqDict() { return MyEqDict<T>(); }

// instance MyEq Bool
// Экземпляр класса MyEq для bool
template <>
class MyEqDict<bool> : public MyEqDictBase<bool>
{
public:
virtual bool myEqual(bool const & l, bool const & r)
const { return l == r; }
};

// Функция принимает словарь и два параметра
// То, что эти параметры на самом деле bool, гарантируется компиляторов Haskell'я
bool fooDict(MyEq const & dict, void const * x, void const * y)
{
```

```
return dict.unsafeMyNotEqual(x, y); // myNotEqual
}

// Вспомогательная функция
// foo :: (MyEq a) => a -> a -> Bool
template <typename T>
bool foo (T const & x, T const & y)
{
return fooDict(makeMyEqDict<T>(), &x, &y);
}

int main()
{
std::cout << foo(true, false) << std::endl; // 1
std::cout << foo(false, false) << std::endl; // 0
return ;
}
```

Некоторые стандартные классы и синтаксический сахар

Enum — перечисление. Определяет функции для получение следующего/предыдущего значения, а также значения по номеру. Используется в синтаксическом сахаре для списков `[1 .. 10]`, фактически, это означает `enumFromTo 1 10`, `[1,3 .. 10] => enumFromThenTo 1 3 10`

Show — преобразование в строку, основная функция `show :: a -> String`. Используется, например, интерпретатором для вывода значений.

Read — преобразование из строки, основная функция `read :: String -> a`. Не знаю, почему выбрали возвращение значения, а не `Maybe a` (опциональное значение), чтобы сигнализировать об ошибке «чистым» способом, а не «грязным» исключением.

Eq — сравнение, операторы `(==)` и `(/=)` (как бы перечёркнутый знак равенства)

Ord — упорядоченные типы, операторы `(<) (>) (<=) (>=)`. Требует принадлежности типа к классу Eq.

Более полный список различных классов можно посмотреть здесь.

Functor — функтор, функция `fmap :: (a -> b) -> f a -> f b`. Чтобы было понятно, приведу примеры применения этой функции:

```
ghci> fmap (+1) [1,2,3]
[2,3,4]
ghci> fmap (+1) (Just 6)
Just 7
ghci> fmap (+1) Nothing
Nothing
ghci> fmap reverse getLine
hello
"olleh"
```

Т.е. для списка это просто `map`, для опционального значения `Maybe a` функция (+1) применяется, если само значение есть, а для ввода-вывода функция применяется к результату этого ввода-вывода. Подробнее про ввод-вывод я напишу далее, сейчас можно просто запомнить, что `getLine` не возвращает строку, так что применить к нему `reverse` напрямую не получится.

Чтобы каждый раз не определять экземпляры для вновь написанных типов данных, Haskell умеет делать это автоматически для некоторых классов.

```
data Test a = NoValue | Test a a deriving (Show, Read,
Eq, Ord)
data Color = Red | Green | Yellow | Blue | Black |
Write deriving (Show, Read, Enum, Eq, Ord)

ghci> NoValue == (Test 4 5)
False
ghci> read "Test 5 6" :: Test Int
Test 5 6
ghci> (Test 1 100) < (Test 2 )
True
ghci> (Test 2 100) < (Test 2 )
False
ghci> [Red .. Black]
[Red,Green,Yellow,Blue,Black]
```

Великие и ужасные монады

Класс Monad представляет собой «вычисление», т.е. он позволяет описывать способ комбинирования вычислений и результатов. Вряд ли я напишу статью лучше, чем есть уже написанные, так что я просто дам ссылки на лучшие (по моему мнению) статьи для понимания, для чего эти монады нужны.
— Монады — статья с SPbHUG о монадах на русском языке и с аналогиями на привычных императивных языках.
— IO inside — статья на английском о вводе-выводе с использованием монад
— Yet Another Haskell Tutorial — книга по Хаскелю, в разделе Monads очень хорошо написано на примере создания класса Computation, который суть и есть монада.

Здесь я напишу очень вкратце, чтобы можно было потом подглядеть. Допутим, мы хотим описать последовательные вычисления, где каждое следующее зависит от результатов предыдущего (некоторым не заданным наперёд способом). Для этого можно определить соответствующий класс, в котором должны быть как минимум две функции:

```
class Computation c where
return :: a -> c a
bind :: c a -> (a -> c b) -> c b
```

Первая функция по сути принимает некоторое значение и возвращает вычисление, которые при выполнении просто вернёт это же самое значение.
Вторая функция принимает 2 аргумента:
1. вычисление, которое при выполнении вернёт значение типа `a`

2. функцию, которое примет значение типа a и вернёт новое вычисление, возвращающее значение типа b
Результатом будет вычисление, которое вернёт значение типа b.

Зачем всё это может быть нужно, я покажу на примере опционального значения

```
data Maybe a = Just a | Nothing
```

Допустим, у нас есть несколько функций, каждая из которых может завершиться неудачей и вернуть Nothing. Тогда, используя их напрямую, мы рискуем получить такой слабочитаемый код:

```
funOnMaybes x =
case functionMayFail1 x of
Nothing -> Nothing -- первая функция ничего нам не вернула, и мы тоже ничего не вернём
Just x1 -> -- отлично, первая функция вернула некоторое значение, продолжим с ним работу
case functionMayFail2 x1 of
Nothing -> Nothing -- теперь вторая функция ничего не вернула (точнее вернула "ничего")
Just x2 -> -- и так далее
```

Т.е. вместо простого последовательного вызова этих функций приходится каждый раз проверять значение. Но ведь у нас язык позволяет передавать функции в качестве аргументов и возвращать так же, воспользуемся:

```
combineMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
combineMaybe Nothing _ = Nothing
combineMaybe (Just x) f = f x
```

Функция combineMaybe принимает опциональное значение и функцию, а возвращает результат применения этой функции к опциональному значнию, либо неудачу. Перепишем функцию funOnMaybes:

```
funOnMaybes x = combineMaybe (combineMaybe x functionMayFail1)
functionMayFail2 --...
```

Или, используя то, что функцию можно вызвать инфиксно:

```
funOnMaybes x = x `combineMaybe` functionMayFail1
`combineMaybe` functionMayFail2 --...
```

Можно заметить, что тип функции combineMaybe в точности повторяет тип bind, только вместо c стоит Maybe.

```
instance Computation Maybe where
return x = Just x
bind Nothing f = Nothing
bind (Just x) f = f x
```

Именно так и определена монада Maybe, только bind там называется (>>=), плюс есть дополнительный оператор (>>), который не использует результат предыдущего вычисления. Кроме того, для монад определён синтаксический сахар, который значительно упрощает их использование:

```
funOnMaybes x = do
x1 <- functionMayFail1 x
x2 <- functionMayFail2 x1
if x2 == (0)
then return (0)
else do
x3 <- functionMayFail3 x2
return (x1 + x3)
```

Обратите внимание на дополнительный do внутри else и на его отсутствие в then. do — это всего лишь синтаксический сахар, который комбинирует несколько вычислений в одно, а так как в ветке then вычисление и так одно (return (0)), то do там не нужен; в ветке else вычисления два подряд, и чтобы их скомбинировать, надо снова использовать do. Специальный синтаксис с обратной стрелкой (<-) преобразуется очень просто:

1. do {e} -> e
do с одним вычислением есть само это вычисление, комбинировать ничего не нужно
2. do {e; es} -> e >> do {es}
несколько подряд идущих вычислений соединяются оператором (>>)
3. do {let decls; es} -> let decls in do {es}
внутри do можно заводить дополнительные декларации наподобие let ... in
4. do {p < — e; es} -> let ok p = do {es}; ok _ = fail "..." in e >>= ok
если результат первого вычисления используется в дальнейшем, то для комбинации используется оператор (>>=). В последнем случае такая конструкция используется потому, что в качестве p может выступать образец, который может и не совпасть. Если он совпадёт, то будет выполнено дальнейшее вычисление, в противном случае будет возвращена ошибка со строковым описанием. Функция fail — ещё одна дополнительная функция в монаде, которая вообще говоря к концепции монад отношения не имеет.

Какие ещё экземпляры монад есть в стандартной библиотеке?

State — вычисления с состоянием. При помощи функций get/put, которые знают о внутреннем устройстве State, можно получать и устанавливать состояние.

```
import Control.Monad.State

fact' :: Int -> State Int Int -- тип состояния - Int, тип результата - тоже Int
fact' = do
acc <- get -- получаем накопленный результат
return acc -- возвращаем его
fact' n = do
acc <- get -- получаем аккумулятор
put (acc * n) -- домножаем его на n и сохраняем
fact' (n - 1) -- продолжаем вычисление факториала
```

```
fact :: Int -> Int
fact n = fst $ runState (fact' n) 1 -- начальное значение состояния - 1
```

`runState` вычисляет функцию с состоянием, возвращает кортеж с результатом и изменённым состоянием. Нам нужен только результат, поэтому `fst`.

Список — тоже монада. Последующие вычисления применяются ко всем результатам предыдущего.

```
dummyFun contents = do
l <- lines contents -- получаем все строки
if length l < 3 then fail "" -- строки менее 3 символов игнорируем
else do
w <- words l -- разбиваем строку на слова
return w -- возвращаем слово
ghci> dummyFun "line1\nword1 word2 word3\n \n \nline5"
["line1","word1","word2","word3","line5"]
```

Continuations (продолжения) в Хаскеле — тоже монада — Cont. Про продолжения можно почитать на русской вики и английской вики и далее по ссылкам. Они заслуживают отдельного внимания, но всё я не умещу, да и к монадам они непосредственного отношения не имеют. Хорошая статья про продолжения в Scheme есть у пользователя palm_mute в живом журнале.

Ввод-вывод тоже реализован с использованием монады. Например, тип функции `getLine`:

```
getLine :: IO String
```

IO-действие, которое вернёт нам строку. `IO` можно понимать так:

```
getLine :: World -> (String, World)
```

где `World` — состояние мира, т.е. любая функция ввода-вывода как бы принимает состояние мира, а возвращает некоторый результат и новое состояние мира, которое затем используется последующими функциями. Разумеется, это всего лишь мысленное представление. Так как в отличие от списков и Maybe у нас нет конструкторов типа `IO`, то мы никогда не сможем результат типа `IO String` разобрать на составляющие, а обязаны будем только использовать его в других вычислениях, таким образом гарантируется, что использование ввода-вывода будет отражено в типе функции.

«Никогда» — это громко сказано, на самом деле есть unsafePerformIO :: IO a -> a, но на то и unsafe, чтобы использоваться только с пониманием дела и когда это крайне необходимо. Я лично ни разу не использовал.

Стоит упомянуть Monad transformers (трансформаторы монад). Если нам нужно состояние, мы можем использовать `State`, если ввод-вывод — `IO`. А что если наша функция хочет иметь состояния и при этом осуществлять ввод-вывод? Для этого предназначен трансформер StateT, который соединяет `State` и другую монаду. Для выполнения вычислений в этой другой монаде используется функция `lift`. Посмотрим на примере факториала, который мы изменим так, чтобы он печатал аккумулятор

```
import Control.Monad.State

fact' :: Int -> StateT Int IO Int -- добавили IO
fact' = do
acc <- get
return acc
fact' n = do
acc <- get
lift $ print acc -- print acc - вычисление типа IO (), поэтому его мы передаём функции lift
put (acc * n)
fact' (n - 1)

fact :: Int -> IO Int -- Пришлось поставить IO и здесь, никуда не деться :)
fact n = do
(r, _) <- runStateT (fact' n) 1
return r
ghci> fact 5
1
5
20
60
120
120
```

Кроме StateT есть также ListT (для списка).

Полный список монад и трансформеров монад.

Для удобства над монадами определены обобщённые функции. Их названия говорят за себя, большинство из них дублируют списочные функции, так что я просто перечислю некоторые из их и дам эту ссылку

```
sequence :: Monad m => [m a] -> m [a]
-- выполнить последовательно все вычисления и вернуть список результатов
mapM f = sequence.map f
forM = flip mapM
-- forM [1..10] $ \i -> print i
forever :: Monad m => m a -> m b -- думаю, пояснять не надо
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a] -- filter
foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a -- foldl
when :: Monad m => Bool -> m () -> m () -- if без ветки else
```

List comprehensions

List comprehensions позволяет конструировать списки в соответствии с математической нотацией:

$$S = \{\, 2 \cdot x \mid x \in \mathbb{N},\ x^2 > 3 \,\}$$

```
ghci> take 5 $ [2*x | x <- [1..], x^2 > 3]
```

```
[4, 6, 8, 10, 12]
ghci> [(x, y) | x <- [1..3], y <- [1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```

Видно, что последний список перебирается чаще.
y может зависеть от x:

```
ghci> [x + y | x <- [1..4], y <- [1..x], even x]
[3,4,5,6,7,8]
```

List comprehensions тоже синтаксический сахар и разворачивается в (последний пример):

```
do
x <- [1..4]
y <- [1..x]
True <- return (even x)
return (x + y)
```

Конечно, он разворачивается напрямую, но так видна связь между монадическими вычислениями над списками и list comprehensions.

Кроссбраузерная одноцветная полупрозрачность

Panya

В этой статье я рассмотрю метод создания блоков с одноцветным полупрозрачным фоном. Сразу оговорюсь, что я не буду использовать opacity и абсолютное позиционирование, чтобы разместить контент поверх полупрозрачного блока.

Итак, нам нужно сделать блок с однотонным полупрозрачным фоном. Для этого можно использовать свойство opacity, но все знают, что оно применяется не только к самому элементу, но и к его детям. Можно, конечно, схитрить и применить opacity к элементу-подложке, а сам контент разместить в другом блоке, и, затем, переместить этот блок на элемент-подложку с помощью абсолютного позиционирования. Но у этого метода есть существенные недостатки, во-первых, необходимо точно знать размеры блока, а, во-вторых, добавляются лишние несемантические элементы.

Всех этих недостатков можно избежать, если вместо opacity использовать однопиксельную картинку нужного цвета с заданной прозрачностью. Но, в таком случае, будет происходить лишний http-запрос, что нежелательно.

В CSS3 появилась возможность задавать цвет фона элементу при помощи RGBA это, по сути, тот же RGB но с возможностью указать значение прозрачности.

```
.opacity {
background: rgba(0, 0, 0, 0.5);
}
```

пример (RGBA)

Но, к сожалению, задание цвета фона через RGBA поддерживается только в новых версиях Safari (Chrome тоже) и Firefox. Для старых версий Safari, Firefox а также для Opera и IE8 (Ура!), чтобы избавиться от лишнего http-запроса, можно использовать Data:URI:

```
.opacity {
background:url(data:image/png;base64,iVBORw0KGg...);
}
```

пример (Data:URI)

Где, `iVBORw0KG...` это наша однопиксельная полупрозрачная картинка, закодированная base64. Такое представление файла получить достаточно просто. Можно, например, использовать Data: URI image encoder.

Объединив вместе эти два способа получим:

```
.opacity {
background:url(data:image/png;base64,iVBORw0KGg...);
background:rgba(0, 0, 0, 0.5);
}
```

пример (RGBA + Data:URI)

Этот пример уже работает для FF 1.5+, Opera 7.2+, Safari 2+, Chrome, Konqueror, IE 8.

Но что делать с IE 7 и IE 6? Здесь нам поможет фильтр Alpha и один маленький трюк. Дело в том, что если к элементу применить фильтр Alpha и потом дать всем потомкам этого элемента position: relative;, то они чудесным образом становятся полностью непрозрачными:

```
.opacity {
zoom:1; /* hasLayout чтобы фильтр применился */
background:#000;
filter:alpha(opacity=50);
}

.opacity * {
position:relative;
}
```

пример (IE 6 и 7)

Итак, совмещая все вместе получим:

```
<!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>Opacity Block</title>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8"/>
<style type="text/css">
* {
margin:0;
padding:0;
}

html {
font-size:100.01%;
}
```

```
body {
font:.8em Arial, Helvetica, sans-serif;
color:#fff;
background:url(bg_pattern.jpg);
}

a {
color:#fff;
}

h1 {
font-weight:normal;
margin:0 0 .5em;
}

p {
margin:0 0 .5em;
}

.opacity {
margin:40px;
padding:20px;
background:url(data:image/png;base64,iVBORw0KG...);
background:rgba(0, 0, 0, 0.5);
}
</style>
<!--[if lte IE 7]>
<style type="text/css">
.opacity {
zoom:1;
background:#000;
filter:alpha(opacity=50);
}

.opacity * {
position:relative;
}
</style>
<![endif]-->
</head>
<body>
<div class="opacity">
<h1>Привет!</h1>
<p>Это полупрозрачный блок.</p>
</div>
</body>
</html>
```

конечный результат

Пример протестирован и корректно работает в IE 6+, FF 1.5+, Opera 7.2+, Safari 2+, Chrome, Konqueror 4.1. Из недостатков, не работает в IE < 6.

3 простых совета, которые сделают ваше Rails приложение быстрее

Arion

Я знаю, что уже много людей писали руководства, помогающие вашему веб-приложению работать быстрее. Но я постараюсь сосредоточиться на самых простых, но наиболее эффективных методах, которые помогут вам существенно ускорить ваше приложение без потери какого-либо функционала из Ruby on Rails.

Совет #1: Приберите ваш статический контент

Часто бывает что одно веб-приложение подгружает сразу несколько javascripts и css стилей. Что существенно замедляет загрузку страницы так как веб-браузер открывает каждый раз новое соединение для нового файла. Решение заключается в том чтобы уменьшить количество внешних ресурсов вашей страницы объединив их все в один файл. Поможет нам в этом плагин AssetPackager

Ставим:

```
script/plugin install git://github.com/sbecker/asset_
packager.git
```

Прмер config/asset_packages.yml:

```
---
javascripts:
- base:
- prototype
- effects
- controls
- dragdrop
- application
- secondary:
- foo
- bar
stylesheets:
- base:
- screen
- header
- secondary:
- foo
- bar
```

И запускаем rake таск:

```
rake asset:packager:build_all
```

Дальше для javascript пишем

```
<%= javascript_include_merged :base %>
```

или

```
<%= javascript_include_merged ‘prototype’, ‘effects’,
‘controls’, ‘dragdrop’, ‘application’ %>
```

Для стилей пишем:

```
<%= stylesheet_link_merged :base %>
```

или

```
<%= stylesheet_link_merged ‘screen’, ‘header’ %>
```

В итоге получаем для режима разработки наш старый код например:

```
<script type=”text/javascript” src=”/javascripts/
prototype.js”></script>
<script type=”text/javascript” src=”/javascripts/
effects.js”></script>
<script type=”text/javascript” src=”/javascripts/
controls.js”></script>
<script type=”text/javascript” src=”/javascripts/
dragdrop.js”></script>
<script type=”text/javascript” src=”/javascripts/
application.js”></script>
<link href=”/stylesheets/screen.css” type=”text/css” />
<link href=”/stylesheets/header.css” type=”text/css” />
```

А в режиме продакшена будет:

```
<script type=”text/javascript” src=”/javascripts/base_
packaged.js?123456789”></script>
<link href=”/stylesheets/base_packaged.css?123456789”
type=”text/css” />
```

Теперь чтобы сделать нагрузку еще меньше переносим все свои статические файлы на другой хост (почему здесь) В рельсах это очень просто сделать, достаточно добавить в config/environments/production.rb такую строчку:

```
config.action_controller.asset_host = “http://assets.
example.ru”
```

Теперь все image_tag, javascript_include_tag и т.д. будут указывать на этот хост.

UPD: Вместо плагина можно использовать <%= javascript_include_tag :all, :cache => true %>, подробнее здесь. Спасибо grossu

Совет #2: Уберите все лишнее

Аутентификация, обращение к сессиям, провер-

ка прав пользователя и лишнии запросы к базе. Действительно ли нужно выполнять все это при каждом запросе к серверу?

Вы можете отключить сессии у конкретных Action просто указав:

```
session :off, :only => :index
```

Это существенно повышает производительность и является отличной альтернативой для кэширования, если вам необходима обратиться к базе данных но при этом нет необходимости проверки прав пользователя.

Если вы используете RestfulAuthentication или ActsAsAuthenticated плагины, вы можете отключить проверку прав пользователя для некоторых Action, что сохранит для вас один запрос к базе.

```
skip_filter :login_from_cookie
```

или

```
skip_filter :login_required
```

ну или, что там у вас еще…

Используя fragment caching для ваших Partials пропускайте запросы к базе в вашем контроллере через read_fragment:

```
@users = User.find('all) unless read_fragment('unique_
cache_key')
```

Кроме того не забывайте правильно использовать опцию :include в ваших запросах для подгрузки ассоциаций

```
Post.find(:all, :include => :user)
```

Это сократит число запросов в 2 раза.

Совет #3: Кэшируйте всю страницу

Этот, последний совет, является наиболее эффективным. Веб-сервер кэширует страницу полностью, а затем отдает лишь статический контент. Что бы начать работать с caches_page достаточно просто посмотреть замечательный railscasts.

Необходимо помнить что после полного кэширования страницы, она будет отображаться одинаково для всех пользователей, не будет производиться не каких проверок или запросов к базе. Поэтому следует избавиться в странице от всех конструкций вида:

```
<%= ... if logged_in? %>
```

```
Вы все ещё можете использовать JavaScript для того что-
бы показывать или скрывать код для зарегистрированных
пользователей. Вот небольшой пример:
var CurrentUser = {
loggedIn: false,
author: false,
admin: false,

init: function() {
this.loggedIn = Cookie.get('token') != null;
this.admin = Cookie.get('admin') != null;
}
};

var Application = {
init: function() {
CurrentUser.init();
},

onBodyLoaded: function() {
if (CurrentUser.loggedIn) {
$$('.if_logged_in').invoke('show');
$$('.unless_logged_in').invoke('hide');
}
if (CurrentUser.admin) {
$$('.if_admin').invoke('show');
}
}
};
```

Так же, вы больше не сможете полноценно использовать <%= flash[:notice] %>. Однако это не проблема, есть замечательный плагин Cacheable Flash

Ставим:

```
ruby script/plugin install svn://rubyforge.org/var/svn/
pivotalrb/cacheable_flash/trunk
```

В ApplicationController пишем:

```
include CacheableFlash
```

В контроллере:

```
flash[:notice] = "Welcome to Eternity" if current_user
```

А в layout:

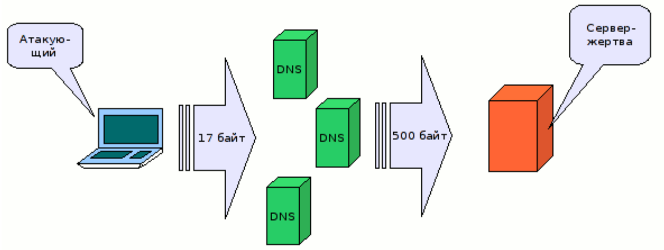
```
<div id="error_div_id" class="flash flash_error"></div>
<div id="notice_div_id" class="flash flash_notice"></div>
<script type="text/javascript">
Flash.transferFromCookies();
Flash.writeDataTo('error', $('error_div_id'));
Flash.writeDataTo('notice', $('notice_div_id'));
</script>
```

Теперь все flash сообщения будут записываться в Куки. Кстати для работы этого плагина необходимо поставить gem json, но в моей Ubuntu 8.10 с этим возникли проблемы, этот гем упорно не хотел вставать, как потом выяснилось подобная проблема не только у меня. ешил я эту проблему так, поставил ruby-json, и заменил в /vendor/plugins/cachable-flash/init.rb строчку gem "json" на gem "ruby-json". И все заработало как часы.

DNS Amplification (DNS усиление)

Nast

Не так давно столкнулся с проблемой (и ее решением) учитывая актуальность этой темы в последнее время, а также то, сколько людей сейчас страдают от этой беды, решил объединить информацию в одну статью. Может быть кому-то еще она будет полезной.



Начало

Пару недель назад я заметил странную активность, направленную на мой DNS-сервер. Сразу скажу, что использую шлюз на Linux, соответственно там установлен DNS-сервер bind. Активность заключалась в том, что на порт 53 (DNS) моего сервера сыпалось по несколько UDP пакетов в секунду с различных IP-адресов:

```
10:41:42.163334 IP 89.149.221.182.52264 > MY_IP.53:
22912+ NS?. (17)
10:41:42.163807 IP MY_IP.53 > 89.149.221.182.52264:
22912 Refused- 0/0/0 (17)
```

На что, как видно из лога, сервер отвечал отказами. Естественно мне стало интересно, что за IP-адреса долбят мой DNS. Посмотрев несколько адресов через whois я определил, что это крупные хостинговые компании, я написал просьбу прекратить атаку на мой сервер в техподдержку некоторых из них. В ответ я получил отписку, что этот тип атаки относится к тем, что они не могут блокировать, и что они сами они страдают от этой аномальной активности. Было решено со всем разбираться самому.

DNS Amplification (DNS усиление). Теория

Сам тип атаки не нов — о нем было известно еще в 2006 году, подробности на английском языке можно посмотреть здесь , однако активно использовать его начали относительно недавно. В январе-феврале 2009 года несколько интернет-изданий опубликовало информацию о крупномасштабном использовании киберпреступниками этого вида DDOS, о чем можно узнать здесь и на английском языке здесь Объясняя простым языком, суть усиления заключается в том, что злоумышленник посылает (обычно короткий) запрос уязвимому DNS-серверу, который отвечает на запрос уже значительно большим по размеру пакетом. Если использовать в качестве исходного IP-адреса при отправке запроса адрес компьютера жертвы (ip spoofing), то уязвимый DNS-сервер будет посылать в большом количестве ненужные пакеты компьютеру-жертве, пока полностью не парализует его работу.

Практика

Наиболее эффективен этот тип атаки на старом (непропатченном) или неправильно сконфигурированном DNS-сервере, который, как уже было сказано, отвечает на короткие запросы злоумышленников большими по размеру пакетами. Вот пример такого взаимодействия (кстати именно такие запросы чаще всего используют атакующие): отправляем серверу NS запрос командой:

```
#dig @SERVER_IP NS
```

где SERVER_IP — IP-адрес сервера. В результате в логе по 53 порту сервера получаем:

```
11:08:47.994604 IP MY_IP.47816 > SERVER_IP.53: 5655+
NS?. (17)
```

т.е. как раз те 17 байт запроса, что мы хотели послать. В ответ в то же самом логе видим:

```
11:08:47.995853 IP 192.168.100.254.53 >
192.168.100.100.47816: 5655 13/0/6 NS C.ROOT-SERVERS.
NET.,[|domain]
```

т.е. сервер ответил нам подсказкой в виде адресов корневых DNS-серверов, что составляет уже 360 байт. Это длины DNS запроса и ответа, общая же длина пакетов 60 и 402 байта соответственно. Усиление налицо.

Что делать?

Во-первых конечно же проверить актуальность версии вашего DNS-сервера вне зависимости от того, на какой платформе он работает. Во-вторых, убедиться, что настроен сервер достаточно безопасно и не отвечает на «левые» запросы всем подряд. Об этом в сети можно найти множество мануалов и рекомендаций, упомяну здесь только об одном документе, который нашел не так давно.

Что еще можно сделать?

В моем случае делать было уже особенно нечего. Если посмотреть самый первый лог, который я привел, то видно, что атакующий отправляет запрос длиной 17 байт и получает REJECT той же самой длины (17 байт), т.е. никакого усиления не происходит. Но, видимо, «ddos-еры» особенно не торопятся

убирать из своих списков адреса DNS-серверов, не подверженных этой уязвимости, и продолжают беспокоить их своей долбежкой… Эта ситуация меня не устраивала. Неприятно что от моего адреса кто-то получает на свой сервер ненужный трафик (даже пусть я в этом и не виноват).

Поначалу я начал ставить в black-лист адреса отправителей, но не тут-то было, со старых адресов атака прекращалась, но появлялись все новые и новые. Было решено использовать более изощренные методы фильтрации и задействовать на моем Linux-сервере модули iptables, до которых раньше у меня никак руки не доходили. Убить, так сказать, сразу двух зайцев — и атакующим сделать -1 и разобраться с парой модулей iptables.

Модуль String

Закрыть 53 порт (DNS) полностью я конечно же не мог — у меня много клиентов которым он нужен. Я решил фильтровать пакеты по содержимому DNS-запросов и убирать те из них, которые содержат запросы атакующих, а они все как один содержали в себе «NS». Для этой задачи подходит модуль iptables string, который как раз позволяет заглянуть в содержимое пакетов.

Для того, чтобы понять что фильтровать, посмотрим на пакет атакующего через wireshark.

```
0000  00 07 e9 13 ff 45 00 5a  40 02 55 86 08 00 45 00   .....E.Z @.U...E.
0010  00 2d 0e 55 00 00 34 11  5c 24 59 95 dd b6 50 59   .-.U..4. \$Y...PY
0020  94 a2 b0 22 00 35 00 19  45 b2 e9 69 01 00 00 01   ...".5.. E..i....
0030  00 00 00 00 00 00 00 00  02 00 01 00               ........ ....
```

Здесь и здесь можно почитать о структуре UDP пакетов и формате кадра DNS соответственно. Для краткости скажу, что первые 14 байт пакета занимают данные протокола Ethernet (на рис. красным), затем идут 20 байт заголовка протокола IP (на рис. синим), затем 8 байт — заголовок UDP (на рис. зеленым), после которого начинаются данные протокола DNS (на рис. желтым). Первые 12 байт кадра DNS занимает заголовок, после которого, наконец, начинается поле DNS Query (т.е. непосредственно сам запрос, на рис. оранжевым). В пакетах, присылаемых атакующим начиная с 54-ого (14+20+8+12) байта идут следующие данные: 00 00 02 00 01 (в шестнадцатеричном коде), что соответствует запросу «NS», о котором я говорил раньше. Таким образом нужно выделить пакеты, которые начиная с 54 байта содержат эти байты. Это будет выглядеть так:

```
iptables -A INPUT --in-interface eth1 --protocol udp
--dport 53 --match state --state NEW --match string
--algo kmp --hex-string “|00 00 02 00 01|” --from 40
--to 45 --jump DROP
```

Немного поясню:
--in-interface — указывает на каком интерфейсе отлавливать пакеты. Нужно поставить только внешний интерфейс, на который идет атака (незачем ущемлять пользователей во внутри сети).
--match state --state NEW — отлавливаем пакеты только со состоянием NEW, чтобы не проверять все транзакции подряд, а только инициирующие пакеты (мало ли что может передаваться по 53 порту).
Дальше идет самое интересное — задействуем модуль sting. Мы используем следующие параметры:
--algo — указывает алгоритм поиска, по сути не важен я указал kmp, но можно указать и bm;
--hex-string — записывается та самая строка в шестнадцатеричном виде, которую мы ищем;
--from 40 — ищем начиная с 40 байта (заметьте, не с 54 потому что string начинает поиск, а соответственно и отсчет от первого байта протокола IP, т.е. выбрасывается протокол Ethernet, длина которого 14 байт(на рис. сверху красным). Итого 54 — 14 = 40);
--to 45 — соответственно искать до 45 байта пакета.

Модуль Recent

На этом уже можно было бы остановиться. Пакеты уже не будут доходить до bind, но меня еще не утешала мысль, что я закрыл для ВСЕХ возможность обращаться с запросами NS к моему DNS-серверу, поэтому я решил задействовать еще один модуль iptables — recent.

Этот модуль позволяет создавать динамические таблицы IP-адресов в зависимости от определенных условий, а затем устанавливать разрешающие и запрещающие правила для этих таблиц. Рассмотрим простой пример использования recent, состоящий из двух строк:

```
iptables -A INPUT --protocol tcp --match state --state
NEW --dport 22 --match recent --update --seconds 30
--name SSHT --jump DROP
iptables -A INPUT --protocol tcp --match state --state
NEW --dport 22 --match recent --set --name SSHT --jump
ACCEPT
```

Начнем разбираться со второго правила. Каждый кто пытается зайти (именно зайти, т.к. мы используем --state NEW) на порт 22 (SSH) пропускается (--jump ACCEPT), но его IP-адрес попадает в таблицу с именем SSHT. Когда с этого адреса пытаются соединиться еще раз, начинает работать первое правило, смысл которого состоит в том, чтобы обновить (--update) запись в таблице и заодно проверить когда эта запись была установлена/обновлена в последний раз. Если запись была установлена/обновлена меньше 30 секунд назад (--seconds 30), то срабатывает --jump DROP и пакет отбрасывается. Таким образом брутфорсеры, пытающиеся долбиться на порт SSH будут отбрасывается, если частота отправки их пакетов будет превышать 1 пакет в 30 секунд. Попробуем использовать recent для наших нужд:

```
iptables -A INPUT --in-interface eth1 --protocol udp
--dport 53 --match state --state NEW --match string
--algo kmp --hex-string “|00 00 02 00 01|” --from 40
--to 45 --match recent --name DNST --update --seconds
600 --jump DROP
iptables -A INPUT --in-interface eth1 --protocol udp
--dport 53 --match state --state NEW --match string
--algo kmp --hex-string “|00 00 02 00 01|” --from 40
```

```
--to 45 --match recent --name DNST --set --jump ACCEPT
```

Таким образом я разрешаю делать запросы NS на внешний интерфейс не чаще, чем 1 раз в 10 минут.

После добавления этих правил в /proc/net/xt_recent моего сервера появился файл DNST, в котором начали записываться IP-адреса атакующих. А DNS-сервер перестал поддаваться на провокации:

```
14:10:19.011818 IP 89.149.221.182.40320 > MY_IP.53:
23508+ NS?. (17)
14:10:25.243499 IP 89.149.221.182.64984 > MY_IP.53:
25306+ NS?. (17)
14:11:08.832827 IP 89.149.221.182.15864 > MY_IP.53:
48029+ NS?. (17)
14:11:15.063058 IP 89.149.221.182.30959 > MY_IP.53:
64444+ NS?. (17)
```

Через несколько дней работы правил количество пакетов со стороны атакующих снизилось в несколько раз. Сейчас я получаю 2-3 пакета в минуту, которые тут же отбрасываются фаерволом.

Асинхронное программирование: концепция Deferred

smira

Асинхронная концепция программирования заключается в том, что результат выполнения функции доступен не сразу же, а через некоторое время в виде некоторого асинхронного (нарушающего обычный порядок выполнения) вызова. Зачем такое может быть полезно? Рассмотрим несколько примеров.

Первый пример — сетевой сервер, веб-приложение. Чаще всего как таковых вычислений на процессоре такие приложения не выполняют. Большая часть времени (реального, не процессорного) тратится на ввод-вывод: чтение запроса от клиента, обращение к диску за данными, сетевые обращение к другим подсистемам (БД, кэширующие сервера, RPC и т.п.), запись ответа клиенту. Во время этих операций ввода-вывода процессор простаивает, его можно загрузить обработкой запросов других клиентов. Возможны различные способы решить эту задачу: отдельный процесс на каждое соединение (Apache mpm_prefork, PostgreSQL, PHP FastCGI), отдельный поток (нить) на каждое соединение или комбинированный вариант процесс/нить (Apache mpm_worker, MySQL). Подход с использованием процессов или нитей перекладывает мультиплексирование процессора между обрабатываемыми соединениями на ОС, при этом расходуется относительно много ресурсов (память, переключения контекста и т.п.), такой вариант не подходит для обработки большого количества одновременных соединений, но идеален для ситуации, когда объем вычислений достаточно высок (например, в СУБД). К плюсам модели нитей и процессов можно добавить потенциальное использование всех доступных процессоров в многопроцессорной архитектуре.

Альтернативой является использование однопоточной модели с использованием примитивов асинхронного ввода-вывода, предоставляемых ОС (select, poll, и т.п.). При этом объем ресурсов на каждое новое обслуживаемое соединение не такой большой (новый сокет, какие-то структуры в памяти приложения). Однако программирование существенно усложняется, т.к. данные из сетевых сокетов поступают некоторыми "отрывками", причем за один цикл обработки данные поступают от разных соединений, находящихся в разных состояниях, часть соединений могут быть входящими от клиентов, часть — исходящими к внешним ресурсам (БД, другой сервер и т.п.). Для упрощения разработки используются различные концепции: callback, конечные автоматы и другие. Примеры сетевых серверов, использующих асинхронный ввод-вывод: nginx, lighttpd, HAProxy, pgBouncer, и т.д. Именно при такой однопоточной модели возникает необходимость в асинхронном программировании. Например, мы хотим выполнить запрос в БД. С точки зрения программы выполнение запроса — это сетевой ввод-вывод: соединение с сервером, отправка запроса, ожидание ответа, чтение ответа сервера БД. Поэтому если мы вызываем функцию "выполнить запрос БД", то она сразу вернуть результат не сможет (иначе она должна была бы заблокироваться), а вернет лишь нечто, что позволит впоследствие получить результат запроса или, возможно, ошибку (нет соединения с сервером, некорректный запрос и т.п.) Этим возвращаемым значением удобно сделать именно Deferred.

Второй пример связан с разработкой обычных десктопных приложений. Предположим, мы решили сделать аналог Miranda (QIP, MDC, …), то есть свой мессенджер. В интерфейсе программы есть контакт-лист, где можно удалить контакт. Когда пользователь выбирает это действие, он ожидает что контакт исчезнет на экране и что он действительно удалится из контакт-листа. На самом деле операция удаления из серверного контакт-листа опирается на сетевое взаимодействие с сервером, при этом пользовательский интерфейс не должен быть заблокирован на время выполнения этой операции, поэтому в любом случае после выполнения операции потребуется некоторое асинхронное взаимодействие с результатом операции. Можно использовать механизм сигналов-слотов, callback'ов или что-то еще, но лучше всего подойдет Deferred: операция удаления из контакт-листа возвращает Deferred, в котором обратно придет либо положительный результат (всё хорошо), либо исключение (точная ошибка, которую надо сообщить пользователю): в случае ошибки контакт надо восстановить контакт в контакт-листе.

Примеры можно приводить долго и много, теперь о том, что же такое Deferred. Deferred — это сердце framework'а асинхронного сетевого программирования Twisted в Python. Это простая и стройная концепция, которая позволяет перевести синхронное программирование в асинхронный код, не изобретая велосипед для каждой ситуации и обеспечивая высокое качества кода. Deferred — это просто возвращаемый результат функции, когда этот результат неизвестен (не был получен, будет получен в другой нити и т.п.) Что мы можем сделать с Deferred? Мы можем "подвеситься" в цепочку обработчиков, которые будут вызваны, когда результат будет получен. При этом Deferred может нести не только положительный результат выполнения, но и исключения, сгенерированные функцией или обработчиками, есть

возможность исключения обработать, перевыкинуть и т.д. Фактически, для синхронного кода есть более-менее однозначная параллель в терминах Deferred. Для эффективной разработки с Deferred оказываются полезными такие возможности языка программирования, как замыкания, лямбда-функции.

Приведем пример синхронного кода и его альтернативу в терминах Deferred:

```
try:
    # Скачать по HTTP некоторую страницу
    page = downloadPage(url)
    # Распечатать содержимое
    print page
except HTTPError, e:
    # Произошла ошибка
    print "An error occured: %s", e
```

В асинхронном варианте с Deferred он был бы записан следующим образом:

```
def printContents(contents):
    """
    Callback, при успешном получении текста страницы,
    распечатываем её содержимое.
    """
    print contents

def handleError(failure):
    """
    Errback (обработчик ошибок), просто распечатываем
    текст ошибки.
    """

    # Мы готовы обработать только HTTPError, остальные
    # исключения проваливаются" ниже.
    failure.trap(HTTPError)
    # Распечатываем само исключение
    print "An error occured: %s", failure

# Теперь функция выполняется асинхронно и вместо
# непосредственного результата мы получаем Deferred
deferred = downloadPage(url)
# Навешиваем на Deferred-объект обработчики
# успешных результатов и ошибок
# (callback, errback).
deferred.addCallback(printContents)
deferred.addErrback(handleError)
```

На практике обычно мы возвращаем Deferred из функций, которые получают Deferred в процессе своей работы, навешиваем большое количество обработчиков, обрабатываем исключения, некоторые исключения возвращаем через Deferred (выбрасываем наверх). В качестве более сложного примера приведем код в асинхронном варианте для примера атомарного счетчика из статьи про [структуры данных в memcached](), здесь мы предполагаем, что доступ к memcached как сетевому сервису идет через Deferred, т.е. методы класса Memcache возвращают Deferred (который вернет либо результат операции, либо ошибку):

```
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}
    @return: Deferred, результат операции
    """
  def tryAdd(failure):
    # Обрабатываем только KeyError, всё остальное
    # "вывалится" ниже
    failure.trap(KeyError)

    # Пытаемся создать ключ, если раз его еще нет
    d = self.mc.add(self.key, value, 0)
    # Если вдруг кто-то еще создаст ключ раньше нас,
    # мы это обработаем
    d.addErrback(tryIncr)
    # Возвращаем Deferred, он "вклеивается" в цепочку
    # Deferred, в контексте которого мы выполняемся
    return d

  def tryIncr(failure):
    # Всё аналогично функции tryAdd
    failure.trap(KeyError)

    d = self.mc.incr(self.key, value)
    d.addErrback(tryAdd)
    return d

  # Пытаемся выполнить инкремент, получаем Deferred
  d = self.mc.incr(self.key, value)
  # Обрабатываем ошибку
  d.addErrback(tryAdd)
  # Возвращаем Deferred вызывающему коду, он может
  # тем самым:
  # а) узнать, когда операция действительно завершится
  # б) обработать необработанные нами ошибки
  # (например,   разрыв соединения)
  return d

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

  @return: текущее значение счетчика
  @rtype: C{int}
  @return: Deferred, значение счетчика
  """
  def handleKeyError(failure):
    # Обрабатываем только KeyError
    failure.trap(KeyError)
```

```
        # Ключа нет — возвращаем 0, он станет результатом
        # вышележащего Deferred
        return 0

    # Пытаемся получить значение ключа
    d = self.mc.get(self.key)
    # Будем обрабатывать ошибку отсутствия ключа
    d.addErrback(handleKeyError)
    # Возвращаем Deferred, наверх там можно будет
    # повеситься
    # на его callback и получить искомое значение
    # счетчика
    return d
```

Приведенный выше код можно написать “короче”, объединяя часто используемые операции, например:

```
return self.mc.get(self.key).addErrback(handleKeyError)
```

Практически для каждой конструкции синхронного кода можно найти аналог в асинхронной концепции с Deferred:
— последовательности синхронных операторов соответствует цепочка callback с асинхронными вызовами;
— вызову одной подпрграммы с вводом-выводом из другой соответствует возврат Deferred из Deferred (ветвление Deferred);
— глубокой цепочки вложенности, распространению исключений по стеку соответствует цепочка функций, возвращающие друг другу Deferred;
— блокам try..except соответствуют обработчики ошибок (errback), которые могут “пробрасывать” исключения дальше, любое исключение в callback переводит выполнение в errback;
— для “параллельного” выполнения асинхронных операций есть DeferredList.

Нити часто применяются в асинхронных программах для осуществления вычислительных процедур, осуществления блокирующегося ввода-вывода (когда не существует асинхронного аналога). Всё это легко моделируется в простой модели ‘worker’, тогда нет необходимости при грамотной архитектуре в явной синхронизации, при этом всё элегантно включается в общий поток вычислений с помощью Deferred:

```
def doCalculation(a, b):
  """
  В этой функции осуществляются вычисления, синхронные
  операции ввода-вывода,
  не затрагивающие основной поток.
  """

  return a/b

def printResult(result):
  print result
def handleDivisionByZero(failure):
  failure.trap(ZeroDivisionError)

  print “Ooops! Division by zero!”
```

```
deferToThread(doCalculation, 3, 2) \
  .addCallback(printResult) \
  .addCallback(lambda _: deferToThread( \
    doCalculation, 3, 0) \
  .addErrback(handleDivisionByZero))
```

В приведенном выше примере функция deferToThread переносит выполнение указанной функции в отдельную нить и возвращает Deferred, через который будет асинхронно получен результат выполнения функции или исключение, если они будет выброшено. Первое деление (3/2) выполняется в отдельной нити, затем распечатывается его результат на экран, а затем запускается еще одно вычисление (3/0), которое генерирует исключение, обрабатываемое функцией handleDivisionByZero.

В одной статье не описать и части того, что хотелось бы сказать о Deferred, мне удалось не написать ни слова о том, как же они работают. Если успел заинтересовать — читайте материалы ниже, а я обещаю написать еще.

Дополнительные материалы

Документация Twisted Framework:

— Основы асинхронного программирования
— Использование Deferred, Откуда берутся Deferred?, Детальное описание Deferred

Deferred в других языках программирования:

— JavaScript: Использование Deferred в JavaScript, Deferred в qooxdoo, Deferred в Dojo, Deferred в MochiKit
— C++: 1, 2, 3

Александр Бурцев о Twisted

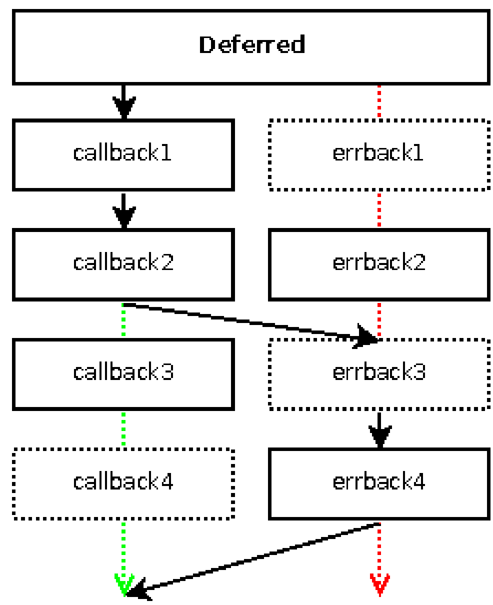
Deferred: все подробности

smira

В предыдущей статье были описаны основные принципы работы Deferred и его применение в асинхронном программировании. Сегодня мы постараемся рассмотреть в деталях функционирование Deferred и примеры его использования.

Итак, Deferred — это отложенный результат, результат выполнения, который станет известен через некоторое время. Результатом, хранящимся в Deferred, может быть произвольное значение (успешное выполнение) или ошибка (исключение), которое произошло в процессе выполнения асинхронной операции. Раз нас интересует результат операции и мы получили от некоторой асинхронной функции Deferred, мы хотим выполнить действия в тот момент, когда результат выполнения будет известен. Поэтому Deferred кроме результата хранит еще цепочку обработчиков: обработчиков результатов (callback) и обработчиков ошибок (errback).

Рассмотрим поподробнее цепочку обработчиков:



Обработчики располагаются по "слоям" или уровням, выполнение происходит четко по уровням сверху вниз. При этом на каждом уровне расположены обработчики callback и errback, один из элементов может отсутствовать. На каждом уровне может быть выполнен либо callback, либо errback, но не оба. Выполнение обработчиков происходит только один раз, повторного входа быть не может.

Функции-обработчики callback являются функциями с одним аргументом — результатом выполнения:

```
def callback(result):
…
```

Функции-обработчики errback принимают в качестве параметра исключение, "завернутое" в класс Failure:

```
def errback(failure):
…
```

Выполнение Deferred начинается с того, что в Deferred появляется результат: успешное выполнение или исключение. В зависимости от результата выбирается соответствующая ветвь обработчиков: callback или errback. После этого происходит поиск ближайшего уровня обработчиков, в котором существует соответствующий обработчик. В нашем примере на рисунке был получен успешный результат выполнения и результат был передан обработчику callback1.

Дальнейшее выполнение приводит к вызову обработчиков на нижележащих уровнях. Если callback или errback завершается возвратом значения, которое не является Failure, выполнение считается успешным и полученный результат отправляется на вход callback-обработчику на следующем уровне. Если же в процессе выполнения обработчика было выкинуто исключение или возвращено значение типа Failure, управление будет передано errback на следующем уровне, который получит исключение в качестве параметра.

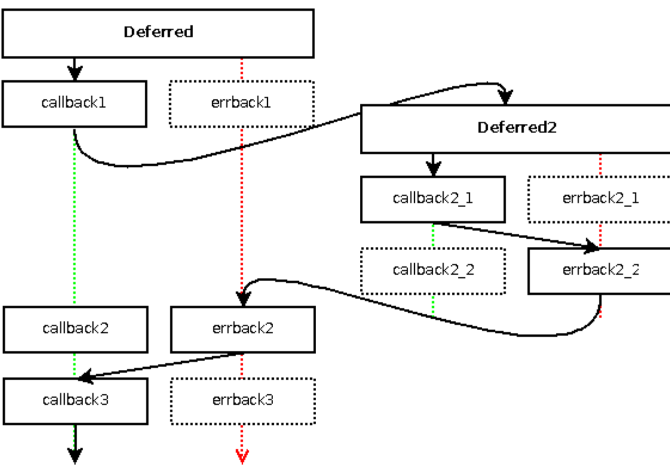
В нашем примере обработчик callback1 выполнился успешно, его результат был передан обработчику callback2, в котором было выкинуто исключение, которое привело к переходу к цепочке errback-обработчиков, на третьем уровне обработчик errback отсутствовал и исключение было передано в errback4, который обработал исключение, вернул успешный результат выполнения, который теперь является результатом Deferred, однако больше обработчиков нет. Если к Deferred будет добавлен еще один уровень обработчиков, они смогут получить доступ к этому результату.

Как и все другие объекты Python, объект Deferred живет до тех пор, пока на него есть ссылки из других объектов. Обычно объект, вернувший Deferred, сохраняет его, т.к. ему надо по завершении асинхронной операции передать в Deferred полученный результат. Чаще всего другие участники (добавляющие обработчики событий) не сохраняют ссылки на Deferred, таким образом объект Deferred будет уничтожен по окончании цепочки обработчиков. Если происходит уничтожение Deferred, в котором осталось необработанное исключение (выполнение завершилось исключением и больше обработчиков

нет), на экран печатается отладочное сообщение с traceback исключения. Эта ситуация аналогична "выскакиванию" необработанного исключения на верхний уровень в обычной синхронной программе.

Deferred в квадрате

Возвращаемым значением callback и errback может быть также другой Deferred, тогда выполнение цепочки обработчиков текущего Deferred приостанавливается до окончания цепочки обработчиков вложенного Deferred.



В приведенном на рисунке примере обработчик callback2 возвращает не обычный результат, а другой Deferred — Deferred2. При этом выполнение текущего Deferred приостанавливается до получения результата выполнения Deferred2. Результат Deferred2 — успешный или исключение — становится результатом, передаваемым на следующий уровень обработчиков первого Deferred. В нашем примере Deferred2 завершился с исключением, которое будет передано на вход обработчику `errback2` первого Deferred.

Обработка исключений в errback

Каждый обработчик исключений errback является аналогом блока try..except, а блок except обычно реагирует на некоторый тип исключений, такое поведение очень просто воспроизвести с помощью Failure:

```
def errback(failure):
  """
  @param failure: ошибка (исключение), завернутое в
  failure
  @type failure: C{Failure}
  """
  failure.trap(KeyError)

  print "Got key error: %r" % failure.value

  return 0
```

Метод `trap` класса `Failure` проверяет, является ли завернутое в него исключение наследником или непосредственно классом `KeyError`. Если это не так, оригинальное исключение снова выбрасывается, прерывая выполнение текущего errback, что приведет к передаче управления следующему errback в цепочке обработчиков, что имитирует поведение блока `except` при несоответствии типа исключения (управление передается следующему блоку). В свойстве `value` хранится оригинальное исключение, которое можно использовать для получения дополнительной информации об ошибке.

Необходимо обратить внимание, что обработчик errback должен завершиться одним из двух способов:
— Вернуть некоторое значение, которое станет входным значением следующего callback, что по смыслу означает, что исключение было обработано.
— Выкинуть оригинальное или новое исключение — исключение не было обработано или было перевыброшено новое исключение, цепочка обработчиков errback продолжается.

Существует и третий вариант — вернуть Deferred, тогда дальнейшее выполнение обработчиков будет зависеть от результата Deferred.

В нашем примере мы исключение обработали и передали в качестве результата 0 (например, отсутствие некоторого ключа эквивалентно его нулевому значению).

Готовимся к асинхронности заранее

Как только появляется асинхронность, то есть некоторая функция вместо непосредственного значения возвращает Deferred, асинхронность начинает распространяться по дереву функций выше, заставляя возвращать Deferred из функций, которые раньше были синхронными (возвращали результат непосредственно). Рассмотрим условный пример такого превращения:

```
def f():
  return 33

def g():
  return f()*2
```

Если по каким-то причинам функция `f` не сможет вернуть результат сразу, она начнет возвращать Deferred:

```
def f():
  return Deferred().callback(33)
```

Но теперь и функция `g` вынуждена возращать Deferred, зацепляясь за цепочку обработчиков:

```
def g():
  return f().addCallback(lambda result: result*2)
```

Аналогичная схема "превращения" происходит и с реальными функциями: мы получаем результаты в виде Deferred от нижележащих в дереве вызовов функции, навешиваем на их Deferred свои обработчики callback, которые соответствуют старому, синхронному коду нашей функции, если у нас были обработчики исключений, добавляются и обработчики errback.

На практике лучше сначала выявить те места кода, которые будут асинхронными и будут использовать Deferred, чем переделывать синхронный код в асинхронный. Асинхронный код начинается с тех вызовов, которые не могут построить результат непосредственно:
— сетевой ввод-вывод;
— обращение к сетевым сервисам СУБД, memcached;
— удаленные вызовы RPC;
— операции, выполнение которых будет выделено в нить в модели Worker и т.п.

В процессе написания приложения часто ясно, что в данной точке будет асинхронное обращение, но его еще нет (не реализован интерфейс с СУБД, например). В такой ситуации можно использовать функции `defer.success` или `defer.fail` для создания Deferred, в котором уже содержится результат. Вот как можно короче переписать функцию `f`:

```
from twisted.internet import defer

def f():
  return defer.success(33)
```

Если мы не знаем, будет ли вызываемая функция возвращать результат синхронно или вернет Deferred, и не хотим зависеть от её поведения, мы можем завернуть обращение к ней в `defer.maybeDeferred`, которое любой вариант сделает эквивалентным вызову Deferred:

```
from twisted.internet import defer

def g():
  return defer.maybeDeferred(f)
    .addCallback(lambda result: result*2)
```

Такой вариант функции `g` будет работать как с синхронной, так и с асинхронной `f`.

Вместо заключения

Рассказывать о Deferred можно еще очень долго, в качестве дополнительного чтения могу опять порекомендовать список материалов в конце предыдущей статьи.

ActionWeb. Асинхронный интернет

nikitaeremin

С момента первого упоминания об AJAX в статье Джесси Джеймса Гарретта «Новый подход к разработке веб- приложений» 18 февраля 2005 года прошло уже 4 года. За это время наверно каждый веб- разработчик хоть раз испробовал эту технологию, и теперь обычного пользователя интернет уже не удивить динамической валидацией форм, автодополнением запросов в строке поиска, всплывающим контекстным меню и прочими ещё недавно диковинными вебдванольными интерактивными радостями, а современные JavaScript фрэймворки делают процесс разработки таких скриптов на порядок быстрей, эффективней и приятней для программиста. Но почему до сих пор подавляющее большинство разрабатываемых веб- сайтов придерживаются стандартной модели загрузки контента и не перешли полностью на AJAX платформу? Зачем каждый раз перезагружать всю страницу при нажатии на внутреннюю ссылку, когда нужно изменить только блок контента, а JavaScript файлы, CSS и большая часть HTML разметки не требует обновления?

Причин несколько:

— Каждый раз открывая в браузере AJAX ресурс, пользователь начинает работу с «отправной точки», т.к. состояние такого приложения в большинстве случаев не отображается в адресной строке браузера
— Динамически подгружаемый контент не обрабатывается поисковыми роботами
— Разработка асинхронных веб- приложений на порядок сложнее и длительнее, чем при использовании статической модели
— Отсутствие интеграции AJAX приложений со стандартными инструментами браузера. По такому ресурсу нельзя перемещаться, используя кнопки браузера back и forward, нельзя добавить страницу сайта в закладки, если она подгружается динамически, нельзя открыть ссылку на динамический контент в новой вкладке браузера
— Использование «толстого» клиента существенно увеличивает нагрузку на процессор компьютера пользователя и требуемый объём оперативной памяти
— AJAX приложение окажется не работоспособным при выключенном JavaScript в браузере пользователя.

Для преодоления перечисленных выше проблем потребуется совершенно новая концепция, которую назовём **ActionWeb**. По своей сути ActionWeb (как и AJAX в своё время) не предлагает новых технологий, а только лишь изменяет принцип использования уже известных. Выдвинем основные тезисы концепции:

1. одни и те же данные не загружаются более одного раза без особой необходимости, полная перезагрузка страницы происходит только в исключительных случаях
2. ActionWeb приложение с точки зрения пользователя и поисковых систем не отличается от обычного веб- сайта
3. разделение презентационного слоя и слоя бизнес-логики клиентской части приложения
4. слой бизнес-логики не виден из презентационного слоя, разработчик от проекта к проекту меняет только презентационную составляющую

Рассмотрим определённые выше проблемы AJAX приложений, и механизмы их решения, реализованные при создании сайта nikitaeremin.com с учётом использования концепции ActionWeb.

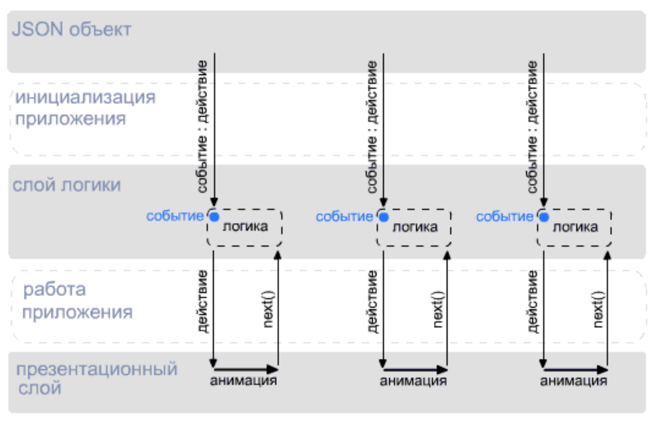
Проблема: Текущее состояние AJAX ресурса не отображается в адресной строке браузера, пользователь каждый раз начинает работу с «отправной точки».
Решение: При изменении url в адресной строке браузера происходит перезагрузка страницы. Из этого правила есть только одно исключение: без перезагрузки можно менять хэш адресной строки (location.hash), проще говоря добавлять\убирать символ # и всё что стоит за ним. В адресной строке будет что-то вроде nikitaeremin.com/#projects/persik/, а ActionWeb приложение при инициализации вычленяет из неё хэш и соответствующим образом формирует своё состояние.

Проблема: динамически подгружаемый контент не обрабатывается поисковыми роботами
Решение: просматривая веб- ресурс поисковые роботы не воспринимают JavaScript. Сканируя исходный HTML код страницы, они путешествуют по внутренним ссылкам, при этом адрес nikitaeremin.com/#projects/persik/ для них будет означать переход на главную страницу. Выход- в исходном коде оставить ссылки как они есть, при этом разделить их на внутренние и внешние. Разделить можно, например, путём добавления css класса к ссылке. При инициализации ActionWeb приложения на ссылки с заданным классом прикрепляются обработчики событий, которые динамически подгружают контент и меняют хэш адресной строки браузера. Но что будет, когда пользователь перешёл с поисковика по ссылке nikitaeremin.com/projects/persik/, проиндексированной в исходном коде ActionWeb приложения? На соответствующей странице ставится редирект на nikitaeremin.com/#projects/persik/, происходит переход на главную страницу и приложение формирует своё состояние, исходя из хэша адресной строки.

Проблема: Отсутствие интеграции AJAX приложений со стандартными инструментами браузера
Решение: Если использовать описанную выше тех-

нику изменения адресной строки, проблема добавления в закладки решается сама собой. Так же решается проблема невозможности открыть внутреннюю ссылку AJAX приложения в новой вкладке браузера, т.к. в ActionWeb приложении нажимая правой кнопкой мыши на внутреннюю ссылку, пользователь нажимает на обычную ссылку, и браузер работает с ней точно так же, как он работал бы с ней на статическом сайте. Восстановить же функции браузера back и forward можно двумя путями. Первый способ- использовать плагин history для jQuery, написанный как раз для этих целей, но данный скрипт не отличается кроссбраузерностью и некорректно работает прежде всего в Internet Explorer. Второй путь- сохранять все изменения адресной строки в JavaScript массиве и с помощью несложных манипуляций реализовать кнопки back и forward прямо на странице приложения.

Проблема: Разработка AJAX приложений на порядок сложнее, чем при использовании статической модели
Решение: Разделение презентационного слоя и слоя логики клиентской части приложения. При таком подходе от проекта к проекту меняется только презентационная составляющая, все функции взаимодействия с сервером, получения и обработки информации «зашиты» внутрь движка, обеспечивая достаточный уровень абстракции, и разработчику не надо задумываться о них. Некоторые возможности, связанные с работой бизнес-модели приложения, такие как кэширование подгружаемых контентных страниц, имя класса для внутренних ссылок ресурса и т.п. можно конфигурировать в JSON объекте, передаваемом движку при инициализации



Проблема: Использование RIA существенно увеличивает нагрузку на процессор клиентского компьютера
Решение: При использовании JavaScript нагрузка на процессор происходит в основном из-за насыщенности презентационного слоя, в особенности когда несколько функций анимации исполняются одновременно друг с другом или с функциями слоя бизнес- логики, а при чрезмерной нагрузке браузер может просто «повиснуть». Выход- реализация цепочечного вызова функций на критических участках приложения. Цепочечный вызов означал бы не допущение исполнения нескольких функций одновременно, предотвращая тем самым чрезмерную нагрузку на компьютер пользователя. Но для реализации этого механизма в JavaScript не достаточно просто расположить вызовы функций один под другим, это просто не сработает, если в таком списке последовательного вызова будет хоть одна функция анимации, т.к. JavaScript интерпретатор сталкиваясь с отсроченными во времени операциями (проще говоря с таймаутами) не останавливается в ожидании их завершения, а идёт дальше по коду. Приведём простой пример, написанный на jQuery. Допустим, есть функция анимации и функция логики, функцию логики необходимо выполнить только после завершения работы функции анимации:

```
function animation(){
    $("#elem").animate({"width" : "300px"}, 200,
"linear", function(){
        alert("animation first!");
    })
}

function logic(){
    alert("logic first!");
}

animation();
logic();
```

При исполнении приведённого выше кода первый алерт будет «logic first!», т.к. завершение функции анимации отсрочено по времени. А для достижения нашей цели код следовало бы переписать следующим образом:

```
function animation(){
    $("#elem").animate({"width" : "300px"}, 200,
"linear", function(){
        alert("animation first!");
        logic();
    })
}

function logic(){
    alert("logic first!");
}

animation();
```

Но при таком подходе не выполнялся бы 4й тезис концепции ActionWeb. Для реализации анонимного вызова функций бизнес-логики в движке Persik потребовалось написание метода chain, связывающего компоненты бизнес-модели и презентационного слоя. В слое бизнес-логики использование метода chain выглядит следующим образом:

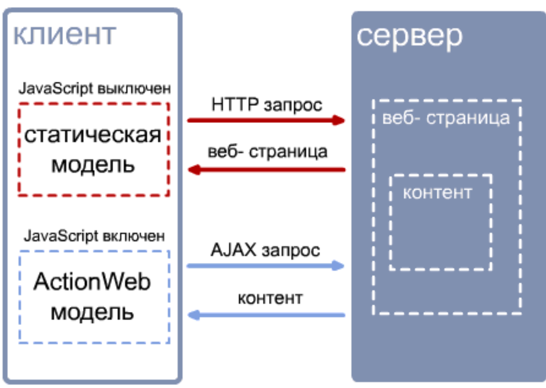
```
engine.chain(func1, [vars1]).chain(func2, [vars2]).
chain(func3, [vars3]).start();
```

При этом в каждой функции цепочки должна быть явно определена точка выхода:

```
function animation(){
    $("#elem").animate({"width" : "300px"}, 200,
"linear", function(){
        alert("animation first!");

        animation.next();
    })
}

function logic(){
    alert("logic first!");
    logic.next();
}

persik.chain(animation).chain(logic).start();
```

Как видно из примера, функции цепочки не знают откуда они были вызваны и каким функциям они передают управление. Такой подход даёт 2 основных преимущества:
1. Разработчик уверен в том, что функция цепочки начнёт своё выполнение только после завершения предыдущей. При этом он сам определяет, насколько последовательным должно быть выполнение функций, и может поставить вызов next() в любом месте функции анимации.
2. Реализуется принцип анонимного вызова функций, обеспечивающий максимальное отделение бизнес- модели от презентационного слоя.

Проблема: AJAX приложение окажется не работоспособным при выключенном JavaScript в браузере пользователя
Решение: В сущности, что из себя представляет ActionWeb ресурс? Это интерактивное веб- приложение, использующее метод асинхронной загрузки данных, состояние которого в большинстве случаев однозначно определяется через url. Что изменится, при выключении JavaScript? Состояние приложения не будет формироваться на стороне клиента, возможность асинхронной загрузки данных станет недоступной, функции анимации не будут выполнены. В таком случае формировать состояние приложения можно на стороне сервера, из функций анимации вычленить конечные состояния объектов (открыто\закрыто), выделить их в отдельные классы и формировать состояние DOM объектов с помощью классов (простейший пример- подсветка текущего пункта меню), асинхронная загрузка контента сменится на традиционную, и в итоге получится самый обыкновенный с точки зрения архитектуры веб- ресурс. На сервере нужно будет определить, включён ли js в браузере пользователя, и исходя из этого выдавать либо ActionWeb вариант сайта, либо обычный статический. Конечно реализация такого подхода возможна не для всех AJAX приложений, и в статическом варианте будут доступны только функции загрузки и отображения контента, но в итоге разработчик будет уверен в том, что пользователь гарантированно получит минимальный объём функциональности сайта, не зависимо от настроек его браузера.



В итоге созданный ресурс распознаётся поисковыми роботами, не теряет работоспособности при выключенном JavaScript в браузере пользователя, быстрее работает за счёт AJAX и отличается высокой презентационной привлекательностью за счёт применения JavaScript анимации.

LINQ to SQL: паттерн Repository

alexeyb

В этой статье будет рассмотрен один из вариантов реализации паттерна репозиторий на базе LINQ to SQL. Сегодня LINQ to SQL – это одна из технологий Microsoft, предназначенная для решения проблемы объектно-реляционного отображения (object-relational mapping). Альтернативная технология Entity Framework является более мощным инструментом, однако у LINQ to SQL есть свои преимущества – относительная простота и низкоуровневость. Данная статья — это попытка продемонстрировать сильные стороны LINQ to SQL. Паттерн репозиторий отлично ложится на парадигму LINQ to SQL.

Репозиторий

Для начала вспомним, что такое репозиторий.

```
public interface IRepository<T> where T: Entity
{
IQueryable<T> GetAll();
bool Save(T entity);
bool Delete(int id);
bool Delete(T entity);
}
```

Репозиторий – это фасад для доступа к базе данных. Весь код приложения за пределами репозитория работает с базой данных через него и только через него. Таким образом, репозиторий инкасулирует в себе логику работы с базой данных, это слой объектно-реляционного отображения в нашем приложении. Более точно, репозиторий, или хранилище, это интерфейс для доступа к данным одного типа – один класс модели, одна таблица базы данных в простейшем случае. Доступ к данным организуется через совокупность всех репозиториев. Обратите внимание, что интерфейс репозитория задается в терминах модели приложения: Entity – базовый класс для всех классов модели приложения (POCO-объекты).

```
public abstract class Entity
{
protected Entity()
{
Id = -1;
}

public int Id { get; set; }

public bool IsNew()
{
return Id == -1;
}
}
```

Вообще говоря, атрибут Id необходим только на уровне базы данных. На уровне модели приложения уникальность объектов может разрешаться без использования явного идентификатора. Таким образом, предлагаемое решение не совсем честное решение проблемы объектно-реляционного отображения с теоретической точки зрения. Однако на практике использование атрибута первичного ключа в модели приложения часто приводит к получению даже более гибких схем. Предлагаемое решение — компромисс между уровнем абстракции слоя базы данных и гибкостью архитектуры.

Методы интерфейса IRepository обеспечивают полный набор CRUD-операций.

GetAll – возвращает всю совокупность объектов данного типа, хранимых в БД. Фильтрация, сортировка и другие операции над выборкой объектов осуществляются на более высоком уровне, благодаря использованию интерфейса IQueryable<T>. Подробнее в разделе «Фильтры и конвейер».

Save – сохраняет объект модели в базе данных. В случае, если он новый, выполняется операция INSERT, иначе – UPDATE.

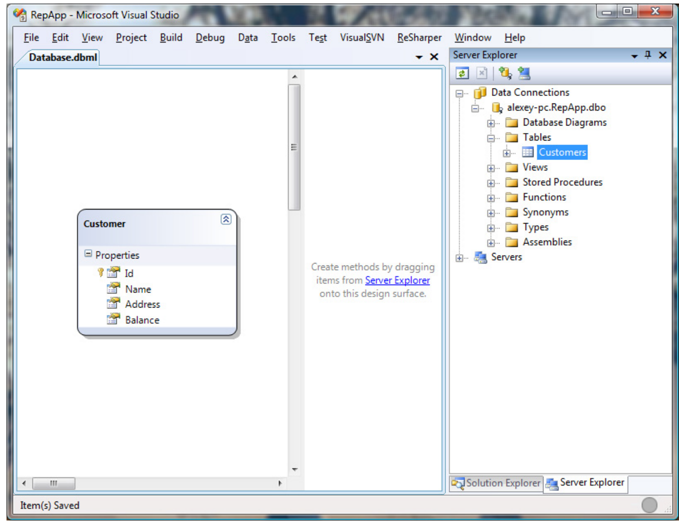
Delete – удаляет объект из базы данных. Предусмотрены два варианта вызова функции: с параметром id удаляемой записи и с параметром объектом класса модели приложения.

Реализация

Пусть у нас есть БД, состоящая из одной таблицы Customers.

```
CREATE TABLE dbo.Customers
(
[Id] int IDENTITY(1,1) NOT NULL PRIMARY KEY,
[Name] nvarchar(200) NOT NULL,
[Address] nvarchar(1000) NULL,
[Balance] money NOT NULL
)
```

Для начала добавим в проект файл dbml, в котором будут задаваться классы объектов модели базы данных и свойства их отображения. Для этого надо воспользоваться контекстным меню Solution Explorer (New Item…->Data->LINQ to SQL Classes) в Visual Studio. После появления окна дизайнера следует открыть Server Explorer и перетащить таблицу Customers в окно дизайнера. Вот что должно получиться:



В результате, Visual Studio сгенерирует класс Customer модели базы данных. Модель самого приложения в общем случае отличается от модели базы данных, но в данном примере они практически совпадают. Ниже приведено описание класса Customer модели приложения:

```
public class Customer : Entity
{
        public string Name { get; set; }
        public string Address { get; set; }
        public decimal Balance { get; set; }
}
```

Пришло время заняться реализацией CustomersRepository – репозитория объектов типа Customer. Для того, чтобы избежать дублирования кода при создании репозиториев для других классов модели, большая часть функциональности вынесена в базовый класс.

```
public abstract class RepositoryBase<T, DbT> :
                                IRepository<T>
where T : Entity where DbT : class, IDbEntity, new()
{
protected readonly DbContext context = new DbContext();

public IQueryable<T> GetAll()
{
        return GetTable().Select(GetConverter());
}

public bool Save(T entity)
{
        DbT dbEntity;

        if (entity.IsNew())
        {
                dbEntity = new DbT();
        }
        else
        {
                dbEntity = GetTable()
                .Where(x => x.Id == entity.Id)
                .SingleOrDefault();
                if (dbEntity == null)
                {
                        return false;
                }
        }

        UpdateEntry(dbEntity, entity);

        if (entity.IsNew())
        {
                GetTable().InsertOnSubmit(dbEntity);
        }

        context.SubmitChanges();

        entity.Id = dbEntity.Id;
        return true;
}

public bool Delete(int id)
{
        var dbEntity = GetTable()
                .Where(x => x.Id == id)
                .SingleOrDefault();

        if (dbEntity == null)
        {
                return false;
        }

        GetTable().DeleteOnSubmit(dbEntity);

        context.SubmitChanges();
        return true;
}

public bool Delete(T entity)
{
        return Delete(entity.Id);
}
protected abstract Table<DbT> GetTable();
protected abstract
        Expression<Func<DbT, T>> GetConverter();
protected abstract void UpdateEntry(DbT dbEntity,
        T entity);
}
```

Все классы модели LINQ to SQL имеют общий интерфейс IDbEntity:

```
public interface IDbEntity
{
        int Id { get; }
}
```

К сожалению, средства визуального дизайнера не позволяют указать базовый класс для объектов LINQ to SQL. Для этого необходимо открыть файл dbml в редакторе XML (Open with...) и указать атрибут EntityBase у элемента Database:

```
<Database EntityBase="Data.Db.IDbEntity" ...>
```

Далее приведено описание класса CustomersRepository.

```
public class CustomersRepository :
RepositoryBase<Customer, Db.Entities.Customer>
{
protected override Table<Db.Entities.Customer>
GetTable()
{
        return context.Customers;
}

protected override Expression<Func<Db.Entities.
Customer, Customer>> GetConverter()
{
        return c => new Customer
        {
                Id = c.Id,
                Name = c.Name,
                Address = c.Address,
                Balance = c.Balance
        };
}

protected override void UpdateEntry(Db.Entities.
Customer dbCustomer, Customer customer)
{
        dbCustomer.Name = customer.Name;
        dbCustomer.Address = customer.Address;
        dbCustomer.Balance = customer.Balance;
}
}
```

Фильтры и конвейер

Метод GetAll репозиториев возвращает объект, реализующий интерфейс IQueryable<T>. Это позволяет применять к выборке объектов операции фильтрации (метод Where), сортировки и любые другие операции, определенные над IQueryable<T>. Для удобства часто употребляемые операции могут быть вынесены в extension-методы. Например, фильтрация по имени клиента.

```
public static IQueryable<Customer> WithNameLike(this
IQueryable<Customer> q, string name)
{
        return q.Where(customer =>
                customer.Name.  StartsWith(name));
}
```

Теперь мы можем использовать репозиторий следующим образом.

```
IRepository<Customer> rep = new CustomersRepository();
foreach (var cust in rep.GetAll()
                .WithNameLike("Google")
                .OrderBy(x => x.Name)) { … }
```

Неважно, какой сложности фильтры или другие операции, которые мы используем. Неважно сколько их. В результате будет выполнен ровно один запрос к базе данных. Этот принцип называется отложенным выполнением запросов (deferred execution) – итоговый SQL-запрос генерируется и исполняется только в момент, когда требуется получить итоговую выборку. В данном случае, это происходит непосредственно перед выполнением первой итерации цикла foreach. Важное преимущество архитектуры – фильтры, как и всё приложение за исключением слоя репозиториев, работают над моделью приложения, а не над моделью базы данных.

Анализ

Далее проводится анализ генерируемых LINQ to SQL запросов к базе данных при выполнении той или иной операции над репозиторием.

GetAll. В случае примера:

```
rep.GetAll().WithNameLike("Google")
                .OrderBy(x => x.Name)
```

Делается единственный запрос:

```
exec sp_executesql N'SELECT [t0].[Name], [t0].
[Address], [t0].[Balance], [t0].[Id]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Name] LIKE @p0
ORDER BY [t0].[Name]',N'@p0 nvarchar(7)',@p0=N'Google%'
```

Метод Save для нового объекта выполняет единственный запрос INSERT. Например:

```
exec sp_executesql N'INSERT INTO [dbo].[Customers]
([Name], [Address], [Balance])
VALUES (@p0, @p1, @p2)
SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]',N'@p0
nvarchar(6),@p1 nvarchar(3),@p2 money',@p0=N'Google',@
p1=N'USA',@p2=$10000.0000
```

В случае вызова Save для существующего объекта или Delete выполняются два запроса. Первый – извлечение записи из базы данных. Например:

```
exec sp_executesql N'SELECT [t0].[Id], [t0].[Name],
[t0].[Address], [t0].[Balance]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Id] = @p0',N'@p0 int',@p0=29
```

Второй запрос – непосредственное выполнение операций UPDATE или DELETE, соответственно. Пример для DELETE:

```
exec sp_executesql N'DELETE FROM [dbo].[Customers]
WHERE ([Id] = @p0) AND ([Name] = @p1) AND ([Address] =
@p2) AND ([Balance] = @p3)',N'@p0 int,@p1 nvarchar(6),@
p2 nvarchar(3),@p3 money',@p0=29,@p1=N'Google',@
p2=N'USA',@p3=$10000.0000
```

В случае UPDATE и DELETE первый запрос является избыточным, однако без него не удастся сохранить или удалить объект, используя стандартные средства LINQ to SQL. Один из вариантов избавления от ненужного запроса – использование хранимых процедур.

Заключение

Основная цель статьи – дать общее представление о паттерне репозиторий и его реализации на LINQ to SQL. Рассмотренный пример применения подхода слишком прост. В реальных приложениях возникает множество проблем при реализации данной архитектуры. Вот некоторые из них.
— Преобразование между объектом модели базы данных и объектом модели приложения может быть значительно более сложным. В таких случаях, невозможно реализовать фильтры над моделью приложения так, чтобы итоговый запрос можно было транслировать в SQL.
— Часто в качестве результата выборки необходимо получить результат соединения (JOIN) нескольких таблиц, а не данные лишь одной таблицы.
— Не все SQL-операции и функции имеют свой эквивалент в LINQ.
— Большинство проблем решаемы, но эти вопросы выходят за рамки данной статьи.

Исходный код к статье (проект ASP.NET MVC).

Ссылки по теме

Паттерн Repository (Martin Fowler)

Статьи по LINQ to SQL от Scott Guthrie

Storefront MVC (screencasts):

Repository Pattern
Pipes and Filters

Безопасный код в Друпале: подделка межсайтовых запросов

neochief

Поводом к написанию этой статьи послужило нахождение мною уязвимости в одном довольно известном модуле. Так как по правилам обнаружения уязвимостей, я пока не вправе распространяться о деталях, то расскажу об уязвимости в общих чертах, а также о методах борьбы с ней. Итак, подделка межсайтовых запросов (анг. Cross Site Request Forgery, или, сокращенно, CSRF): что это такое и с чем его едят.

CSRF — это вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP. Если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер (например, на сервер платёжной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счёт злоумышленника). Для осуществления данной атаки, жертва должна быть авторизована на том сервере, на который отправляется запрос, и этот запрос не должен требовать какого-либо подтверждения со стороны пользователя.

Данный тип атак, вопреки распространённому заблуждению, появился достаточно давно: первые теоретические рассуждения появились в 1988 году, а первые уязвимости были обнаружены в 2000 году.

Одно из применений CSRF — эксплуатация пассивных XSS, обнаруженных на другом сервере. Так же возможны отправка спама от лица жертвы и изменение каких-либо настроек учётных записей на других сайтах(например, секретного вопроса для восстановления пароля).

Живой пример

Например, нам нужно сделать небольшой модуль, который должен аяксом удалять ноды. Это можно реализовать служебной ссылкой ноды, при нажатии которой, отправляется аякс запрос на друпаловский путь. К этому пути прицеплен обработчик, который и удаляет ноду. Вот примерно таким модулем все и делается:

node_destroy.module

```
/**
* Реализация hook_menu(). Регистрирует наш коллбек в системе меню.
*/
function node_destroy_menu() {
        $menu['node/%node/destroy'] = array(
        'page_callback' => 'node_destroy',
        'page_arguments' => array(1),
        'access_arguments' => array('administer nodes'),
        'type' => MENU_CALLBACK,
);
}
/**
* Реализация коллбека.
*/
function node_destroy($node) {
        if ($node->nid) {
                node_delete($node->nid);
                print('SUCCESS');
        }
// в коллбеках для аякса почти всегда надо принудительно завершать скрипт,
// чтобы не выводить оформление сайта вместе с вашими данными
        exit();
}
/**
* Реализация hook_link(). Добавляем свою ссылку в служебные ссылки ноды.
*/
function node_destroy_link($type, $node = NULL,
                                $teaser = FALSE) {
switch ($type) {
case 'node':
// если эта функция вызывается, значит мы выводим ссылки ноды,
// а это значит, что нам и скрипты нужны
$path = drupal_get_path('module', 'node_destroy');
drupal_add_js($path .'/node_destroy.js');
// собственно, добавление ссылки
$links['node_destroy'] = array(
'title' => t('Destroy node'),
'href' => "node/$node->nid/destroy",
'attributes' => array('class' => 'node_destroy_link'),
);
break;
}
return $links;
}
```

node_destroy.js

```
// Таким нехитрым путем правильно инициализировать некие действия
// вместо обычного $(document).ready(function() { ... })
Drupal.behaviors.node_destroy = function(context) {
// Мы перебираем все наши ссылочки и навешиваем на них аякс запросы.
// Заметьте необычный селектор. Он предотвратит двойное
```

```
навешивание обработчиков.
$('.node_destroy_link:not(.processed)', context).
addClass('processed').click(function(){
href = $(this).attr('href');
$.ajax({
        type: "GET",
        url: href,
        success: function(result){
// SUCCESS нам возвращает наш коллбек меню, если все
замечательно
        if (result != 'SUCCESS') {
                alert('Error');
        }
}
});
});
}
```

И все бы хорошо, но в один солнечный день, на сайт приходит злой тролль… Или более жизненная ситуация — озлобленный бывший сотрудник приходит на сайт и пытается его поломать. Помня старый опыт, он пробует зайти по адресу site.ru/node/123/destroy, но получает от ворот поворот, так как уже не имеет прав на удаление материалов.

И тут, в порыве деструктивного креатива, он создает ноду с таким контетом:

```
<img src=”http://site.ru/node/123/destroy” />
```

Что происходит в этот момент? Никакая картинка, естественно, не подгрузится, но браузер тролля выполнит запрос на этот путь с прежним результатом.

Смирившись с неудачей, тролль уходит с сайта. Через день, администратор сайта замечает эту мусорную ноду, заходит в нее и удаляет. А вернувшись в список материалов, не находит в нем ноды с айдишником 123. Атака удалась. Занавес.

Для тех, кто не понял, когда администратор зашел в ноду, его браузер тоже ломанулся по ссылке картинки. Но здесь уже прав доступа хватило, и нода была успешно удалена, а админ даже ничего не заметил.

Как избежать CSRF уязвимостей?

Ответ — использовать уникальные ссылки для действий по изменению данных. Как это возможно? В друпале используется метод токенизации ссылок. Это означает, что к ссылке активного действия, прибавляется уникальный параметр, который проверяется при осуществлении самого действия. В друпале сгенерировать такой параметр можно функцией drupal_get_token(). Проверить —drupal_valid_token(). Токен генерируется на основе подаваемого значения, сессии пользователя, а также приватного ключа сайта, что практически сводит на ноль вероятность генерации вредителем правильного токена.

Внесем изменения в наш модуль. Начнем с выставления правильной ссылки:

```
function node_destroy_link($type, $node = NULL,
        $teaser = FALSE) {
switch ($type) {
case 'node':
$path = drupal_get_path('module', 'node_destroy');
drupal_add_js($path .'/node_destroy.js');
$links['node_destroy'] = array(
        'title' => t('Destroy node'),
        'href' => "node/$node->nid/destroy",
        'attributes' =>
                array('class' => 'node_destroy_link'),
// query – это все GET параметры, т.е. все что в ссылке
находится после знака вопроса
// мы добавляем параметр token
'query' =>
'token='. drupal_get_token('node_destroy_'. $node->nid)
);
break;
}
return $links;
}
```

Как вы помните, мы шлем аякс запрос по адресу, который зашит в ссылке, поэтому в коллбеке нам остается только проверить $_GET стандартным способом.

```
function node_destroy($node) {
if ($node->nid && isset($_GET['token']) &&
drupal_valid_token($_GET['token'], 'node_destroy_'.
$node->nid)) {
        node_delete($node->nid);
        print('SUCCESS');
}
exit();
}
```

via DrupalDance

Безопасный код в Друпале: работа с базой данных

neochief

Друпал предоставляет свои собственные средства для доступа к базе данных. Во-первых, это позволяет не зависеть от используемого типа СУБД. К слову, на сегодняшний момент, полностью функционирует прослойка для MySQL и PostgreeSQL. В седьмом Друпале этот список будет расширен Ораклом и SQLite. Во-вторых же, прослойка БД позволяет защититься от SQL инъекций.

Самая первая функция, о которой следует узнать при работе с базой — db_query(). Начну, пожалуй, с примера, в стиле которого пишут почти все начинающие друпаллеры:

```
/**
* Пример 1 - небезопасный
* Пример должен отобразить список заголовков нод типа
$type (например, поступающего из поля формы)
*/
$result =
db_query("SELECT nid, title FROM node WHERE type =
'$type'");

$items = array();
while ($row = db_fetch_object($result)) {
       $items[] = l($row->title, "node/{$row->nid}");
}
return theme('item_list', $items);
```

В этом примере сразу несколько вещей в корне неправильны.

Псевдонимы названий таблиц

Название таблиц следует заключать в фигурные скобки, а также присваивать им псевдонимы, которые рекомендуется всегда использовать при обращении к колонкам. Измененный вызов будет выглядеть так:

```
$result = db_query("SELECT n.nid, n.title FROM {node} n
WHERE n.type = '$type'");
```

Что нам это даст? Это обеспечит простоту обработки таблиц с префиксами. То есть, если у вас все таблицы в базе называются «pr_node», «pr_users» и т.д., Друпал автоматически будет подставлять корректные префиксы к таблицам, заключенным в скобки. Указание псевдонимов при этом избавит от надобности использовать фигурные скобки больше одного раза.

Фильтрация аргументов

Отсутствует фильтрация аргументов запроса. Это прямой путь к SQL инъекции. Если в $type окажется значение story' UNION SELECT s.sid, s.sid FROM {sessions} s WHERE s.uid = 1/*, то весь запрос будет уже таким:

```
SELECT n.nid, n.title FROM {node} n
WHERE n.type = 'story'
UNION
SELECT s.sid, s.sid FROM {sessions} s
WHERE s.uid = 1/*'
```

что позволит мошеннику завладеть айдишниками сессий, и в свою очередь, при создании корректной куки сессии, получить прямой админский доступ к сайту.

Защититься от этого довольно просто, используя параметризацию запроса. При формировании запроса, Друпал использует синтаксис функции sprintf. В строке запроса вставляются заглушки, которые заменяются параметрами, которые идут отдельно. При этом параметры проходят проверку и экранирование, так что вы можете забыть об инъекциях, используя данный подход. Вот некоторые примеры:

```
db_query("SELECT n.nid FROM {node} n WHERE n.nid > %d",
$nid);
db_query("SELECT n.nid FROM {node} n WHERE n.type =
'%s'", $type);
db_query("SELECT n.nid FROM {node} n WHERE n.nid > %d
AND n.type = '%s'", $nid, $type);
db_query("SELECT n.nid FROM {node} n WHERE n.type =
'%s' AND n.nid > %d", $type, $nid);
```

Список заменителей:

— %d — для целых чисел (integers)
— %f — для чисел с плавающей запятой, т.е. дробных (floats)
— %s — для строк (однако, обратите внимание, что в запросе, вокруг строки выставляются кавычки)
— %b — двоичные данные (не нужно оборачивать в кавычки)
— %% — заменяется на % (например, для LIKE %monkey%)

Для конструкций IN (... , ... , ...), используйте функцию db_placeholders(), которая создаст нужную последовательность заменителей, по заданному массиву параметров, например:

```
$nids = array(1, 5, 449);
db_query('SELECT * FROM {node} n WHERE n.nid IN ('. db_
placeholders($nids) .')', $nids);
```

Если вы используете модуль Devel, у вас есть очень простой способ получения конечных запросов для отладочных целей. Просто вызовите функцию db_queryd() с точно такими же параметрами, как вы вызываете db_query().

Теперь, наш запрос будет выглядеть так:

```
$result = db_query("SELECT n.nid, n.title FROM {node} n
WHERE n.type = '%s'", $type);
```

Ранжирование результатов запроса

Наш пример на большом сайте выведет большущий список нодов. Что если нам можно ограничиться всего первым десятком? Первым позывом будет использовать SQL конструкцию LIMIT, например

```
SELECT n.nid, n.title FROM {node} n
WHERE n.type = '%s' LIMIT 0, 10
```

и вроде бы все хорошо, но на Postgree SQL этот код приведет к ошибке, так как с этим сервером управления, вам нужно использовать конструкцию OFFSET 0 LIMIT 10. А еще на каком-нибудь Оракле, синтаксис опять другой. Что же делать?

Ответ — использовать db_query_range() для лимитирования количества результатов запроса. Его использование аналогично db_query, за исключением того, что в после всех аргументов, вам нужно указать два параметра — номер первой строки, и количество результатов. Наш запрос преобразится в следующее:

```
// выведет первых 10 результатов
$result = db_query_range("SELECT n.nid, n.title FROM
{node} n WHERE n.type = '%s'", $type, 0, 10);
```

И на последок, если вам ко всему еще нужен постраничный вывод, используйте функцию pager_query(). Она отличается от db_query_range() наличием всего одного необязательного параметра, о котором вы можете почитать на странице документации. С этой функцией вывод листалки страниц прост как дважды два:

```
/**
* Пример 2 - безопасный, с листалкой
*/
// изменяем сам запрос
$result = pager_query("SELECT n.nid, n.title FROM
{node} n WHERE n.type = '%s'", $type, 0, 10);

$items = array();
while ($row = db_fetch_object($result)) {
$items[] = l($row->title, "node/{$row->nid}");
}
$output = theme('item_list', $items);

// добавляем листалку
$output .= theme('pager');

return $output;
```

Как видите, всего две строчки изменений. Всю рутину по подхватыванию текущей страницы, обработке и т.д. полностью берет на себя Друпал.

Возможность изменения запроса модулями

Довольно часто имеет смысл предоставить другим модулям возможность повлиять на ваш запрос. В Друпале это реализуется связкой функции db_rewrite_sql(), и реализациями хука hook_db_rewrite_sql() в модулях. Наш запрос будет выглядеть так:

```
$result = pager_query(db_rewrite_sql("SELECT n.nid,
n.title FROM {node} n WHERE n.type = '%s'", 'n',
'nid'), $type, 0, 10);
```

а вот и пример реализации хука, для того, чтобы у вас было представление, что происходит:

```
// Модуль отсеет все ноды, авторы которых сутки не были
на сайте
function my_module_db_rewrite_sql($query, $primary_
table, $primary_field, $args) {
        switch ($primary_field) {
        case 'nid':
        if ($primary_table == 'n') {
        $return['join'] =
"LEFT JOIN {users} u ON $primary_table.uid = u.uid";
        $return['where'] = 'u.login > '. time() -
                60 * 60 * 24;
        }
        return $return;
        break;
}
}
```

Возвращенные из хука 'join' элементы, будут прикреплены к нашему запросу, 'where' — добавлены к списку условий, и наш запрос после обработки будет таким:

```
SELECT n.nid, n.title FROM {node} n
LEFT JOIN {users} u ON n.uid = u.uid
WHERE n.type = '%s' AND u.login > 199976743
```

После этого, он, собственно, поступит в pager_query() и будет обработан как обычно.

Финальный код примера

```
/**
* Пример 3 - безопасный, с листалкой и возможностью пе-
резаписи запроса
*/
// добавляем db_rewrite_sql
$result = pager_query(db_rewrite_sql("SELECT n.nid,
```

```
n.title FROM {node} n WHERE n.type = '%s'", 'n',
'nid'), $type, 0, 10);

$items = array();
while ($row = db_fetch_object($result)) {
$items[] = l($row->title, "node/{$row->nid}");
}

$output = theme('item_list', $items);

$output .= theme('pager');

return $output;
```

Полезные ссылки

— Функции работы с базой данных
— Стандарты кодирования SQL в друпале

Статьи цикла «Безопасный код»

— Подделка межсайтовых запросов
— Работа с базой данных

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:

voidex, Panya, Arion, Nast, smira, nikitaeremin, alexeyb, neochief

Орфография и пунктуация авторская с небольшими правками.

Проект habradigest

Владимир "ХаосCPS" Юнев - автор, редактор, верстка, сайт

Владимир "OnTheFly" Синельников - дизайн, хостинг, домен

habradigest, февраль 2009