Содержание

Благодарности **9**

Вступление **10**

О Node.js 10
Об этой книге 10

Предыстория **12**

Веб-страница, веб-сервис, веб-приложение 12
Асинхронность как необходимость 12
Решения – Twisted, Tornado и другие 14

Странный язык – JavaScript **16**

Волшебные замыкания 17
Что такое замыкание? 17
Применение замыканий 19
ООП в JavaScript 21
Явление Node 26

Хватит теории! Начало работы с Node.js **27**

Установка Node 27
Веб-сервер из пяти строк 29
Сайт на Node.js 32

Node Core **37**

Как все работает? Event loop в Node.js 37
Глобальные объекты (Globals) 37
Global 38
Объект Console 38
Require и псевдоглобальные объекты 39
Процессы 40
Свойства и методы объекта Process 40
Метод process.nextTick() 42
Процессы ввода/вывода 43
Signal Events 44
Child Process – параллелизм в Node.js 45
Понятие буфера 49
Таймеры 52

События **54**

Слушаем! 54
Объект EventEmitter 56

Модули 58
Пакетный менеджер npm 59
Создаем собственный модуль 63
А теперь по-взрослому – пишем Си++ Addons 70

Работа с файлами 75
В дебри файловой системы 75
Маленький полезный модуль – Path 81
Бродим по папкам 84
Работа с файлами – вот она, асинхронность 85
Читаем файлы 85
Watching Files 88
Потоки – унифицируем работу с источниками данных 90
ReadStream/WriteStream 90
Веб-сервер на потоках 93

Сервер HTTP, и не только 95
Создаем TCP-сервер 95
UDP – тоже полезный протокол 104
Переходим на прикладной уровень – реализация HTTP 106
IncomingMessage – входящий HTTP-запрос 107
ServerResponse 108
HTTP-клиент (грабим Центробанк) 110
HTTPS – шифруемся! 114
Запускаем HTTPS-сервер 115
И секретный клиент 115

WebSockets – стандарт современного веба 118
Браузер – веб-сервер. Надо что-то менять 118
WebSockets – окончательное решение? 120
Простой способ – модуль ws 121
Начинаем работу с ws 121
Реализация WebSocket-чата 123
Обмен бинарными данными 127
Socket.io – webSockets для пролетариата 129
Реальное время для всех! 129
Начинаем работать с socket.io 130
Простой чат на socket.io 132
Усложняем 134
Совершенствуем приложение – дополнительные возможности socket.io 136
Пространства имен 140
«Летучие» сообщения 141
Извещения (acknowledgements) 142
Конфигурация 142

Пирамиды судьбы – асинхронный поток выполнения и как с ним бороться 145
Начинаем строить пирамиды 146
Долой анонимость! 150
Node.js control-flow 151
Async – берем поток исполнения в свои руки 153
Инструменты async 153
Control Flow средствами async 157
Живительный водопад (async.waterfall) 161

Node.js и данные. Базы данных 165

MySQL и Node.js 166
Четыре буквы – CRUD 167
Preparing Queries 168
Чтение, обновление и удаление данных 169
Работа с пулом соединений 171

ORM-система Sequelize 173
Начинаем работать с Sequelize 173
CRUD на Sequelize 176
Связи 179

NoSQL 182

NodeJS и Memcached 183
Основы Memchashed 183
Реализация 184
Создаем приложение 186
MemcacheDB – все-таки DB 190

Redis – очень полезный овощ 192
Redis – что он умеет? 192
Основы работы с Redis 193
Модуль Redis для Node.js 194
Хэши (Hashes) 196
Множества (Sets) 199
Упорядоченные множества (Sorted Sets) 201
Механизм Publish/Subscribe 202
Очередь сообщений с помощью Redis 205

MongoDB: JavaScript – он везде! 208
Для чего? 209
Основы работы с MongoDB 210
Native MongoDB 214

Рики-тики-тави: Mongoose для MongoDB....219
Основы работы с Mongoose....220
CRUD по-мангустски....221
Сеттеры, гетттеры и прочие приятные вещи....224

Переходим на сторону клиента....226

Мыслим шаблонами....227

Mustache – усатый и нелогичный....227
EJS – пришелец из мира RoR....232
Синтаксис EJS-шаблонов....233
Помощники (Helpers)....233
EJS и Node.js....234
Фильтры....237
Jade – нечто нефритовое....239
Начинаем работу с Jade....240
Include – собираем шаблон по частям....247
Примеси....248

CSS-препроцессоры – решение проблем стиля....250

LESS – больше, чем Sass....250
Вложенные блоки....251
Переменные....252
Операции и функции....253
Примеси....255
Расширения....257
Работаем с LESS в Node.js....257
Stylus....260
Возьмем CSS и отсечем лишнее....260
И тут примеси....262
Функции и переменные....263
Stylus и Node.js....267

Поднимаем разработку на новый уровень....270

Чего нам не хватает?....270

Connect – middleware framework для node.js....272

Что такое middleware? (и зачем все это?)....272
Connect на практике....273
Статический сайт одной строчкой (почти)....275
Совершенствуем сайт....275
Пишем свое СПО....278
Еще немного Connect....280

Веб-каркас для node (node.js web-framework'и)....282

Что такое web-framework?....282

Express....283
Начало работы с Express....283
Закат солнца вручную....285
Подключаем шаблонизатор....289
Задействуем статику....291
Подключаем CSS....291
Разработка RESTFul-приложения на основе Express....293
Немного о REST-архитектуре....293
Приступаем к реализации RESTFul API....294
Подключаем источник данных....299
А теперь – на три буквы (на MVC)....306
Немного об архитектуре MVC....306
Структурируем код....307
Добавляем новую сущность....313

Практика разработки приложений Node.js....316

Nodemon – друг разработчика....316
Отладка Node.js-приложений (debug-режим)....320
Node Inspector – отладка на стороне клиента....324

Тестирование Node.js-приложений....325

Что такое Unit-тестирование?....325
TDD/BDD....325
Accert – утвердительный модуль....326
Should – BDD-style тестовый фреймворк....330
Цепочки утверждений....330
Chai – BDD/TDD-библиотека утверждений....335
Chai TDD-стиль....335
Chai BDD....335
Mocha – JavaScript тест-фреймворк....336
Начинаем работать с Mocha....337
Exports....340
QUnit....341
Асинхронный код....341
Jasmine – ароматный BDD-фреймворк....342
Основы работы с Jasmine....342
Jasmine и асинхронный код....345
Spies – шпионим и эмуллируем....346

Grunt – The JavaScript Task Runner....349

Grunt – начало работы....350
Инструменты Grunt....352
Grunt watch – задача-наблюдатель....359
Grunt connect web server....361

Альтернативы JavaScript и Node.js....363

CoffeeScript – зависимость с первой чашки....363

Краткий курс по кофе 364
Классы, наследование, полиморфизм, генераторы! 367
CoffeScript и Node.js 368
Пишем сервер на CoffeScript 370
TypeScript – типа JavaScript от Microsoft 372
Node.js как TypeScript-компилятор 373
Аннотации типов 373
Классы! настоящие классы! 374
Интерфейсы 376
Модули 377
Что еще? 378
Dart – дротик в спину JavaScript от Google 378
Экосистема Dart 378
Знакомство с Dart 382
ООП – чтим традиции! 383
Область видимости и библиотеки 385
Изоляторы 386
А в общем-то... 387

Будущее уже сейчас – ECMAScript.next и Node 388

ECMAScript 5 – уже стандарт 388
Всякие строгости – Strict Mode в JavaScript 388
JSON 390
Массивы 392
Объекты 394
Who's Next? ECMAScript 6 399
ECMAScript 6 в браузерах и в Node.js 400
Вот теперь настоящие классы! 400
Модульная структура 402
Цикл for-of 403
Функции – let it block! 403
Arrow function 404
Обещания 406
Proxy – займемся метапрограммированием на JavaScript 407
Константы 409
Генераторы 409

Заключение – что дальше? 412

Приложение – полезные ресурсы по платформе Node.js 413

Список литературы 414

Предметный указатель 415

Благодарности

В первую очередь огромное спасибо Райану Далу, TJ Holowaychuk и другим разработчикам, подарившим мне отличный инструмент, благодаря которому я могу на своем любимом языке писать полноценные серверные приложения.

Огромное спасибо Дмитрию Мовчану, который буквально вынудил меня взяться за этот труд и поддерживал в процессе. Спасибо издательству «ДМК Пресс» и отдельно Галине Синяевой.

Спасибо редакции журнала «Системный администратор», Галине Положенец, Полине Гвоздь – без вас мне было бы очень трудно.

Особая благодарность тем, кто помогал мне в работе над книгой, – Александу Календареву, Валентину Синицину, Александру Слесареву, Алексею Вторникову.

Спасибо тем специалистам, советы и рекомендации которых помогли в работе над материалом. Это Дмитрий Бородин, Андрей Шетухин, Олег Царев, Илья Кантор, Евгений Зиндер, Борис Богданов, а также Константин Полянский, Александр Байрак, Артем Демин «and last but not least» – Андрей Бешков. Огромное спасибо Александру Смирнову и сообществу phpclub.ru.

Спасибо моему сыну Роману и моим друзьям за поддержку.

И самое главное спасибо – моей музе и невесте Лиде. Любимая, без твоего терпения и поддержки вообще бы ничего не состоялось.

Вступление

О Node.js

Node.js – это серверная JavaScript-платформа, предназначенная для создания масштабируемых распределенных сетевых приложений, использующая событийно-ориентированную архитектуру и неблокирующее асинхронное взаимодействие. Она основана на JavaScript-движке V8 и использует этот же JavaScript для создания приложений. Node.js хоть и достигла в своем развитии только цифр 0.10.30 в номере версии, но уже активно используется в реальных проектах.

Помимо эффектной асинхронной модели работы, неблокирующих процессов, высокой производительности, Node.js делает то, что считалось принципиально невыполнимым, – дает возможность разработчику создавать как server-side/backend-, так и frontend-приложения, пользуясь единой технологией! Да-да, теперь на JavaScript можно написать обработчик http-запросов, да что там, настоящий, полнофункциональный веб-сервер! Можно работать с SQL- (и NoSQL-) базами данных, сетью, файловой системой. Еще недавно все это казалось труднодостижимым.

На самом деле, чего там скрывать, когда 5 лет назад новая технология только появилась, автору этих строк и многим его коллегам она казалась забавной игрушкой – интересной, но без шансов промышленного применения. Я рад, мы ошибались – Node.js доказала свою состоятельность, и сейчас её «боевое» использование – не экзотика, а нормальная практика, особенно в пресловутых высоконагруженных проектах. Node.js сейчас тем или иным образом используют такие известные участники IT-рынка, как Groupon, SAP, LinkedIn, Microsoft, Yahoo!, Walmart, PayPal. По-моему, достойная компания, к которой не грех присоединиться.

Об этой книге

Мне всегда нравились рассказы о новых технологиях, тесно завязанные на практические примеры, на реальный, работающий код. И сам я попытался создать нечто подобное. В книге не слишком много общих фраз, но очень много кода и пояснений к нему.

Сначала, после небольшого вводного обзора, мы установим Node.js, начнем работу и даже, с места в карьер, напишем первый Node.js-сайт (это не займет много времени). Далее мы подробно познакомимся

с ядром Node.js, освоим его основные компоненты – событийную модель, процессы, понятие буфера, таймеры.

Далее мы познакомимся с понятием модуля, освоим менеджер пакетов Node.js – Node Packaged Manager и даже напишем собственный Node.js-модуль (а то и пару).

Затем, изучив работу с файлами и потоками (Stream), примемся за сетевую ипостась платформы. Мы узнаем, как на Node.js можно создавать TCP/UDP/HTTP-серверы и организовать работу сети. В следующем разделе мы будем изучать работу Node.js с хранилищами данных, как реляционными (mysql), так и NoSql – Memcached, Redis, MongoDB. Если вы до этого мало имели дело с NoSQL, не беда, мы постараемся более или менее подробно разобрать работу каждого хранилища.

Далее рассказывается о различных реализациях WebSocket-сервера и инструментах Node.js для работы с протоколом Websocket – ws, socket.io. Веб-сокеты – это уже стандарт современного веба, и платформа Node.js имеет все средства для их воплощения.

Следующий раздел посвящен клиентской стороне веб-разработки – изучаем шаблонизаторы и CSS-препроцессоры и выбираем лучший. Выбрать есть из чего – рассмотрена работа Mustache, EJS, Jade, LESS, Stylus.

Вторая половина книги возвращает нас на сторону сервера, но уже на новом уровне – будут рассмотрены midware-фреймворк Connect, платформа Express, создано первое RESTful API Node.js-приложение. В разделе «Практика разработки приложений Node.js» будут рассмотрены средства отладки, профилирования, сборки и развертывания Node.js-приложений. Отдельная глава посвящена инструментам тестирования – модулям assert, chai, should, фреймворкам Mocha, Jasmine.

Последний раздел посвящен будущему основного инструмента платформы Nodep.js – языка JavaScript. Рассмотрены такие его модификации/заменители, как CoffeeScript, TypeScript и Dark, а также подробно рассмотрены настоящие и будущие нововведения языка, привнесенные стандартами EcmaScript5 и EcmaScript6 (Harmony), – на платформе Node.js сейчас они доступны почти все.

Для нормального восприятия книги достаточно начальных знаний языка javaScript, общего представления об устройстве Всемирной сети и желания разобраться в самых современных веб-технологиях.

Предыстория

За что я люблю веб-разработку? За то, что веб развивается прямо у нас на глазах. Это живая среда, которая растет бурно и зачастую непредсказуемо, меняясь и меняя нашу жизнь, увлекая ее за собой. Я отлично помню то время, когда получить на свой компьютер простую HTML-страничку с какого-нибудь заморского сервера было маленьким чудом, а сейчас... даже описывать что-либо не имеет смысла. Сегодня www – это значительная часть нашей рабочей, личной и социальной сферы, и похоже, это только начало.

Впрочем, это все лирика, нас, суровых технарей, больше интересует именно техническая сторона этой непрерывной веб-революции.

Веб-страница, веб-сервис, веб-приложение

Не будем сейчас особо вдаваться в детали, но если кратко: всё развивалось в несколько этапов. Сначала были плоские HTML-странички, и задачей веб-сервера было только исправно отдавать их клиентам. Затем появился cgi и аналогичные механизмы, страницы стали генериться с помощью различных интерпретаторов (perl, php и т. д.), затем они (интерпретаторы) стали стандартом, как модули к веб-серверу. Появилась возможность использовать базы данных, было написано множество библиотек и фреймворков, выводящих веб-разработку на новый уровень.

Параллельно развивалась и клиентская часть www. HTML-страницы с помощью JavaScript и DOM перестали быть статичными, они «ожили», превратив с появлением технологии Ajax сайты из набора связанных страниц в полноценные приложения.

А потом... а потом наступила эра highload, и многих возможностей старого веба стало не хватать, причем нехватка эта носила принципиальный, архитектурный характер. Требовались новые подходы, такие как использование NoSQL-хранилищ данных, очередей сообщений, балансировки нагрузки и асинхронного доступа.

Асинхронность как необходимость

Что такое асинхронно событийная модель и зачем она нужна? Строго говоря, второй вопрос не требует ответа. Если у вас, при разработке вашего проекта, не возникла потребность в неблокирующем асинхронном взаимодействии, то, скорее всего, это вам просто не нужно.

Пока. Но если вы создаете систему, которая поддерживает взаимодействие по множеству одновременных подключений, причем взаимодействие двунаправленное, то рано или поздно вы придёте к необходимости организации асинхронного доступа.

Да, именно доступа – под асинхронностью и понимают асинхронный доступ к ресурсам – файлам, сокетам, данным. Как он происходит по классической схеме работы веб-приложения? Все просто и состоит из нескольких этапов:

- подключиться к ресурсу (источнику данных);
- считать из него первую порцию данных;
- ждать, пока на него не будет готова вторая порция данных;
- считать вторую порцию данных;
- ждать третью порцию данных;
- ...; завершить считывание данных;
- разорвать соединение и продолжить работу.

При этом при исполнении программы возникает «блокировка», вызванная тем, что установка соединения с ресурсом и (особенно) чтение из него данных требуют времени. Во время этой блокировки поток исполнения простаивает, и это, безусловно не самое рациональное использование ресурсов. Для преодоления данной проблемы была разработана хорошо всем известная многопоточная модель. Суть ее работы – в том, что приложение создает некоторое количество потоков, передавая каждому из них задачу и данные для обработки. Все задачи выполняются параллельно, и если они не имеют общих данных и не нуждаются в синхронизации, их обработка происходит достаточно быстро. В такую модель работы почти идеально укладывается обработка веб-сервером запросов от многочисленных клиентов (браузеров, запрашивающих веб-страницы).

Собственно, все еще самый популярный в Интернете веб-браузер – Apache – так и работает. На каждый http-запрос создаётся отдельный поток (ну, если точнее, может просто забираться свободный поток из пула, но это уже детали реализации), в котором работает, например, php-скрипт. Все хорошо, и такая модель способна обеспечить стабильную работу при относительном множестве запросов. Но... но вот Hihgload ведь. Что, если одновременных запросов будет не просто множество, а тысяча? Десять тысяч? Сто тысяч? Да, в общем, ничего особенного – просто создаваемые потоки, в конце концов, заполнят всю оперативную память и уронят север. С этим, конечно, можно бороться, наращивая объем памяти, вводя в бой дополнительные ресурсы, но есть и другие способы справляться с высокими нагрузками.

Асинхронная модель решает проблемы доступа принципиально по-другому. Она построена на зацикленной очереди событий (event-loop). При возникновении некоторого события (пришел запрос, ответ от базы данных, считалась порция данных из файла) оно помещается в конец очереди. Поток исполнения, обрабатывающий этот цикл, выполняет код, связанный со следующим событием, и, в свою очередь, помещает его в конец. Это происходит до тех пор, пока очередь не опустеет.

Реализации такой схемы работы написаны для различных языков программирования. Это фреймворки Perl AnyEvent, Ruby EventMachine, Python Twisted и веб-сервер Tornado. По такой же схеме работает и Node.js, имея в своем воплощении event loop ряд особенностей, которые, возможно, позволят этой среде получить ключевое конкурентное преимущество. Все их мы со временем постараемся рассмотреть.

Решения – Twisted, Tornado и другие

Twisted – событийно-ориентированный сетевой фреймворк, написанный на Python. Он поддерживает все основные сетевые протоколы (HTTP, TCP, UDP, SSL/TLS, IP Multicast, Unix domain sockets и т. д.). Основной модуль фреймворка, twisted.internet.reactor, представляет реализованный собой тот самый цикл событий (event loop), который занимается выполнением обработчиков событий и возможных ошибок.

По умолчанию редактор веб-сервера (как и фреймворк в целом) использует механизм распределения событий для неблокирующих сокетов. **Tornado** – это расширяемый, неблокирующий серверный веб-фреймворк, написанный тоже на Python. Он разработан для использования в проекте FriendFeed, который был приобретен Facebook в 2009 году, после чего исходные коды Tornado были открыты. Tornado был создан для обеспечения высокой производительности, и он действительно отлично справляется с той задачей на легких запросах, но вот на сложных проектах требует дополнительных компонентов – прежде всего это веб-сервер.

Ключевые различия между этими двумя платформами следующие: Twisted – это набор библиотек для асинхронного программирования на питоне, сложный и многофункциональный; Tornado – это веб-сервер, который может запускать приложения (как асинхронные, так и самые обычные).

Ну, хватит с Python, есть что-нибудь на других языках? Конечно! **EventMachine** – легкий фреймворк, использующий асинхронный, событийно-ориентированный механизм обработки сетевых соединений, предназначенный для сетевого взаимодействия. EventMachine написан на языке Ruby и имеет в своем составе довольно много средств – веб-сервер, механизмы подписки и отложенного выполнения, набор стандартных классов. Это не единственная платформа на Ruby, но самая известная.

Впрочем, и языки – ветераны веб-разработки имеют собственные решения для асинхроной модели. **AnyEvent** – Perl-модуль, популярный фреймворк, реализующий так называемую концепцию событийно-ориентированного программирования (СОП) в Perl. Близок к рассматриваемой нами асинхронной событийной модели и является наиболее оптимальным выбором для программирования современных сетевых приложений в Perl.

И, надо сказать, это почти все. Нельзя сказать, что перечисленные решения не получили признания, но все же не сильно распространены, однако они достаточно нишевые, и это естественным образом ограничивает их применение.

Странный язык – JavaScript

Этот странный, рожденный «на коленке» язык программирования собрал огромное количество критики и нареканий. И это естественно – язык никогда не проектировался для такого применения и вообще развивался «от практики». Поначалу и применение его на веб-страницах (и только на них!) ограничивалось простыми эффектами вроде подсветки кнопок при наведении мыши и тому подобными мелкими «красивостями». Но постепенно JavaScript брал на себя все большую роль по взаимодействию www и человека, а с появлением суммы технологий, обозначенных красивым термином Ajax, эта роль стала вообще ключевой. Еще интенсивней JavaScript стал использоваться с распространением библиотек/фреймворков, делающих разработку сложных сценариев, приложений (да-да, речь идет уже о JavaScript-приложениях!) и веб-интерфейсов простым и приятным занятием. Я говорю прежде всего о библиотеке Prototype, заложившей основы нескольких JavaScript-фреймворков, о jQuery, ставшей практически стандартом разработки, о Ext.js, задающем новый уровень веб-интерфейсов, наконец, о MVC-среде – Backbone.js, предоставляющей каркас для создания полноценных REST-приложений.

Но этого было мало. Новый стандарт языка разметки веб-страниц – HTML5 – подразумевает наличие множества JavaScript API, которые, не являясь частью языка, творят настоящие чудеса, как на веб-странице, так и за ее пределами. Я говорю про Geolocation API, WebRTC, WebGL и подобные технологии, находящиеся сейчас на переднем крае развития www.

Обилие возможностей породило JavaScript-безумие, которое длится и по сей день, – на этом языке стали писать все – 3D-игры, компиляторы, DE-окружения и даже операционные системы! Правда, пока эти творения скорее более ориентированы на демонстрацию возможностей, чем на практическое использование, но зато демонстрация получается ну очень впечатляющая.

И вот появилась платформа Node.js – JavaScript проник на сервер! Похоже, скоро в Интернете ничего, кроме JavaScript, и не останется.

И тут я бы хотел рассказать про этот удивительный язык, но, во-первых, это точно займет много времени, а во-вторых, рядовой веб-разработчик JavaScript, как правило, знает. Ну или думает, что знает.

В любом случае, я хочу прояснить всего два аспекта этого языка. Они очень важны для дальнейшего изложения материала, но если замыкания и ООП-практика JavaScript вам хорошо знакомы, смело пропускайте следующие два раздела.

Волшебные замыкания

Замыкание (*closure*) – это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции, причем в качестве её параметров, то есть функция, которая ссылается на некие сущности в своём контексте.

Что такое замыкание?

Ничего не понятно? Хм. Я бы тоже не понял. Можно сказать, что замыкание – это особый вид функции, которая определена в теле другой функции и создаётся каждый раз во время её выполнения. При этом вложенная внутренняя функция может содержать ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра замыкания, с новыми ссылками на переменные внешней функции.

Замыкания поддерживаются в таких языках, как Ruby, Python, Scheme, Haskell, Java (с помощью анонимных классов). Не так давно их реализация стала возможна в C++ (стандарт C++11) и PHP (с версии 5.3). Конечно, многие программисты при написании JavaScript-сценариев обходятся без замыканий вовсе, но это не значит, что мы должны следовать их примеру. Во-первых, замыкания здорово упрощают жизнь, во-вторых, некоторые вещи без их применения просто невозможно реализовать. В современном JavaScript замыкания – важный элемент языка, без которого нам не обойтись, поэтому давайте разберемся, что это и как с этими конструкциями обращаться.

Начнем с примера:

```
function localise(greeting) {
    return function(name) {
        console.log(greeting + ' ' + name);
    };
}

var english = localise('Hello');
var russian = localise('Привет');

english('closure');
russian('closure');
```

Такой код выведет в консоль:

```
>Hello closure
>Привет closure
```

Фактически мы создали две разные функции, в которых фраза приветствия «замкнута» вмещающей функцией. Но это еще не все. Как известно, в JavaScript областью видимости локальной переменной является тело функции, внутри которой она определена. Если вы объявляете функцию внутри другой функции, первая получает доступ к переменным и аргументам последней и сохраняет их даже после того, когда внешняя функция отработает:

```
function counter(n) {
    var count = n;
    return function() {
        console.log(count++);
    };
}
var ct = counter(5);
ct(); //5
ct(); //6
ct(); //7
ct(); //8
ct(); //9
```

Функция ct() возвращает количество собственных вызовов, причем начальное значение задается при ее создании.

Магия? Да ничего подобного. Вернее, да, магия, но она заложена в самой концепции JavaScript. Возьмем, например, такой код:

```
var i = 5
function plusOne(){
     i++;
     return i;
}
console.log(plusOne()); //6
console.log(plusOne()); //7
console.log(i); //7
```

Тут, наверное, никого не смущает, что функция plusOne() имеет доступ к глобальной переменной i. А ведь это тоже замыкание, только по глобальной области видимости. Переменные же, объявленные внутри функций, имеют область видимости, ограниченную рамками этой функции, и замыкание происходит именно по ней – пусть даже и сама внешняя функция завершена. Это делает возможными следующие конструкции:

```
var outerVar = 4;
var handle;
function outer() {
    var innetVar = 5;
    function inner() {
        console.log(outerVar);
         console.log(innetVar);
    };
    handle = inner;
}
outer();
handle(); //4
          //5
```

(Разумеется, если **handle()** вызывать до **outer()**, скрипт завершится с ошибкой, ведь handle еще не ссылается на функцию.)

Применение замыканий

Самым очевидным применением замыканий будет конструкция вроде «фабрики функций» (не буду называть это Abstact Factory, дабы не вызвать гнев приверженцев чистоты терминологии):

```
function createFunction(n) {
   return function(x) {
      return x*n;
   }
}

var twice = createFunction(2);
var threeTimes = createFunction(3);
var fourTimes = createFunction(4);
console.log(twice(4)+" "+fourTimes(5)); //8 20
```

Это, конечно, очень простой пример, но оцените саму возможность. Практически метапрограммирование!

Следующее важное применение – сохранение состояния функции между вызовами. Мы уже рассматривали это на примере функции, возвращающей количество вызовов. Ее можно сделать еще проще:

```
var ct = (function counter() {
    var count = 5;
    return function() {
        console.log(count++);
    };
})();
ct(); //5
ct(); //6
ct(); //7
```

Так у нас нет необходимости отдельно вызывать внешнюю функцию.

Как известно, в JavaScript отсутствуют модификаторы доступа, а оградить переменную от нежелательного влияния иногда ой как необходимо. Опять же, использование замыкания нам может организовать это не хуже, чем модификатор private:

```
function counter() {
    var count = 0;
    this.getCount = function(){
      return count;
    }
    this.setCount = function(n){
      count = n;
    }
    this.incrCount = function(){
      count++;
    }
}

var ct = new counter();
ct.setCount(5);
ct.incrCount();
console.log(ct.getCount()); //6
console.log(ct.count); //undefined
```

Как видите, цель достигнута – прямой доступ к «приватной» переменной count невозможен. Её можно получить, только используя методы counter, которые даже могут быть приватными!

```
function counter() {
   var count = 0;
   function setCount(n) {
     count = n;
   }
   return {
      safeSetCount: function(n) {
      if(n!=13){
         setCount(n);
      } else {
         console.log("Bad number!");
      }
      },
    getCount: function(){
      return count;
    },
    incrCount: function(){
      count ++ ;
    }
```

```
  }
}
ct = new counter();
ct.safeSetCount(5);
ct.safeSetCount(13); //Bad number!
ct.incrCount(13);
console.log(ct.getCount()); //6
```

Тут попытка вызвать метод **ct.SetCount()** снаружи немедленно приведет к ошибке.

ООП в JavaScript

Поддерживает ли JavaScript парадигму объектно-ориентированного программирования? С одной стороны, да, безусловно, с другой – при изучении реализации ООП в JavaScript большинство специалистов, привыкших к разработке на C++ или Java, в лучшем случае отмечают своеобразие этой самой реализации. Ну действительно, есть объекты, но где классы? Как наследовать? А инкапсулировать? Как с этим вообще можно работать?

На самом деле работать можно, и даже очень эффективно, просто надо помнить, что ООП в JavaScript прототипное. То есть в нем отсутствует понятие класса, а наследование производится путём клонирования существующего экземпляра объекта – прототипа. С этим обстоятельством не надо бороться, его надо использовать.

Объекты в JavaScript представляют собой в первую очередь ассоциативный массив. То есть набору строковых ключей соответствует одно любое значение. Значением массива может быть и функция, в таком случае она является методом этого объекта.

Как мы обычно создаем «с нуля» объект в JavaScrit? Например, так:

```
var user_vasya = { name: "Vasya",
         id: 51,
         sayHello: function(){
               console.log("Hello "+ this.name);
               }
         };
user_vasya.sayHello(); // Hello Vasya
```

Или так:

```
var user_masha = {};
user_masha.name = "Masha";
user_masha.uid = 5;
user_masha.sayHello = function(){ console.log("Hello "+ this.name);}
user_masha.sayHello();  // Hello Masha
```