Урок N2

Консольные программы

Node.js хорошо подходит для создания кроссплатформенных консольных приложений. Его просто установить, а готовые программы легко распространять с помощью npm. Все чаще многие разработчики устанавливают node просто для использования написанных на нем утилит. Например, с помощью Node.js созданы популярные системы сборки веб-проектов Grunt.js и Gulp.js

---

Содержание урока

Традиционно самым простым способом управления консольными приложениями является передача параметров из консольной строки при их запуске.

Переданные в скрипт параметры доступны в массиве process.args:
["node", "/путь/к/скрипту.js", "параметр 1", "параметр 2", …]

process.argv.slice(2) вернет все разделенные пробелами параметры:
["параметр 1", "параметр 2", …]

Поскольку обрабатывать вручную различные комбинации параметров и их различные форматы (например: a=b —debug file.txt) неудобно, для этих целей обычно используют тот или иной npm-модуль. Один из простых и удобных - **minimist**, наследник популярного optimist, в котором оставлен только основной функционал.

Вот как он работает:

```
var argv = require('minimist')(process.argv.slice(2));
console.dir(argv);
```

———————————————————————————

```
$ node example/parse.js -a beep -b boop
{ _: [], a: 'beep', b: 'boop' }
```

———————————————————————————

```
$ node example/parse.js -x 3 -y 4 -n5 -abc --beep=boop foo bar baz
{ _: [ 'foo', 'bar', 'baz' ],
  x: 3,
  y: 4,
  n: 5,
  a: true,
  b: true,
  c: true,
  beep: 'boop' }
```

Помимо аргументов командной строки, полезную информацию можно получать из переменных окружающей среды, которые доступны всем программам в системе. Например, перменная PATH содержит пути поиска исполняемых файлов для выполнения команд в консоли.

Также пользователи могут настраивать переменные окружения по своему усмотрению. Например, в переменную NODE_ENV можно записать значения для определения режима запуска программы: dev - при локальной разработке, staging - при тестировании на стенде, production - при работе на "боевом" сервере и т.п.

Все переменные окружения можно прочитать из ассоциативного массива **process.env** . Также через этот массив можно менять текущие значения переменных окружения и добавлять новые, но этим изменения будут действительны только для текущего процесса и дочерних процессов, до момента окончания программы. Так что обычно переменные окружения устанавливаются пользователем, а скрипты просто их считывают.

Если необходимо переопределить значение переменной непосредственно для конкретной программы на время ее выполнения, можно выполнить следующую команду (*NIX):
NODE_ENV=dev node script.js

console.log(arg1, arg2, …) - выводит строковые аргументы

console.error(arg1, arg2, …) - выводит строковые аргументы

Для чего нужны 2 одинаковые по результату работы команды? Дело в том, что любая программа работает как минимум с 3 стандартными потоками ввода-вывода.

Поток stdin предназначен для ввода данных. При этом если программа запущена из консоли, то он соответствует вводу из консоли. Если же поток ввода перенаправлен из другой программы или из файла, данные будут поступать из соответствующего источника напрямую в stdin поток. О работе с потоком stdin мы поговорим позже.

Поток stdout предназначен для вывода сообщений для пользователя. При запуске из консоли это вывод в консоль. Также этот поток можно перенаправить на вход другой программы или в файл.

Поток stderr предназначе для вывода сообщений об ошибках для разработчика. При запуске из консоли по умолчанию это вывод в консоль. Также этот поток можно перенаправить на вход другой программы или в файл, что чаще всего и делается при запуске программ node с помощью менеджеров процессов.

Таким образом, можно отдельно выводить или логировать сообщения для пользователя и сообщения об ошибках для разработчика.

util.inspect(arg) - возвращает отформатированное в строку представление аргумента (простой переменной, массива, объекта и т.п.). Обычно используется совместно с **console.log(util.inspect(arg))** для вывода данных о сложных объектах. По умолчанию данные будут выводиться только до второго уровня вложенности. Чтобы вывести данные с заданным уровнем вложенности, используется **util.inspect(arg, {depth: N})**. Для вывода структуры целиком, без ограничений на вложенность: **util.inspect(arg, {depth: null})**

console.dir(arg) - эквивалентно console.log(util.inspect(arg, {depth: 2})

Одним из самых простых и удобных способов консольного ввода является построчный ввод данных. Для этого используется стандартный модуль readline .

Инициализация:

```
var readline = require('readline');

var rl = readline.createInterface({
        input: process.stdin, // ввод из стандартного потока
        output: process.stdout // вывод в стандартный поток
});
```

Обработка каждой введенной строки:

```
rl.on('line', function (cmd) {
        console.log('You just typed: '+cmd);
});
```

Получение ответа на вопрос (аналогично prompt в браузере):

```
rl.question('What is your favorite food?', function(answer) {
        console.log('Oh, so your favorite food is ' + answer);
});
```

Пауза (блокирование ввода):

```
rl.pause()
```

Разблокирование ввода:

```
rl.resume()
```

Окончание работы с интерфейсом readline:

```
rl.close()
```

Дополнительные npm-модули для ввода: prompt, cli

Для работы с файлами используется стандартный модуль fs.

Инициализация:

var fs = require('fs');

fs.exists(path, callback) - проверка существования файла
fs.readFile(filename, [options], callback) - чтение файла целиком
fs.writeFile(filename, data, [options], callback) - запись файла целиком
fs.appendFile(filename, data, [options], callback) - добавление в файл
fs.rename(oldPath, newPath, callback) - переименование файла
fs.unlink(path, callback) - удаление файла

Функции **callback** принимают как минимум один параметр err, которые равен null при успешном выполнении команды или содержит информацию об ошибке. Помимо этого при вызове readFile передается параметр data, который содержит объект типа Buffer, содержащий последовательность прочитанных байтов. Чтобы работать с ним как со строкой, нужно его сконвертировать:

```
fs.readFile('/etc/passwd', function (err, data) {
        if (err) throw err;
        console.log(data.toString());
});
```

Чтобы обеспечить правильную работу с файлами в различных кодировках и конвертацию их в строки, в опциях команд работы с файлами можно указывать кодировку, по умолчанию она не определена: {encoding: null}. Node по умолчанию без проблем работает с кодировкой UTF8. Если требуется работать с другими кодировками, например, windows-1251, для этого обычно используются дополнительные модули: iconv-lite или iconv.

Также почти все методы модуля fs имеют синхронные версии функции, оканчивающиеся на Sync. Этим функциям не нужны callback, т.к. они являются блокирующими и поэтому не рекомендованы к применению в большинстве случаев.

Чтобы "поймать" и обработать ошибку в виде исключения (например, при попытке чтения несуществующего файла) и продолжить дальнейшую работу, используется обычный блок try..catch

```
try {
	throw new Error('my silly error');
} catch (err) {
	console.error(err);
}
```

Также можно установить глобальный обработчик исключений. Обратите внимание, что после выполнения этого обработчика программа будет завершена!

```
process.on('uncaughtException', function(err) {
	console.log('Caught exception: ' + err);
});
```

Продвинутым новым и пока еще нестабильным способом обработки ошибок является использование модуля domain . Основное преимущество этого метода в том, что можно создать несколько различных областей (доменов), которые будут обрабатывать ошибки аналогично process.on('uncaughtException', callback) но с той разницей, что программа будет продолжать работу и вы получите доступ к информации о стеке вызова в месте возникновения ошибки.

```
var d = require('domain').create();
d.on('error', function(er) {
	console.error('Caught error!', er);
});
d.run(function() {
	process.nextTick(function() {
		setTimeout(function() { // simulating async stuff
			fs.open('non-existent file', 'r', function(er, fd) {
				if (er) throw er;
				// proceed...
			});
		}, 100);
	});
});
```

Данные библиотеки ощутимо облегчают жизнь как при разработке клиентского JavaSrcipt, так и при использовании совместно с node . Конечно же, они легко устанавливаются с помощью npm (для node) или bower (для браузера).

underscore - в основном для работы с массивами и объектами
sugar - для работы со строками, массивами, объектами и датами
async - для организации взаимодействия асинхронных функций

Самоконтроль

✓ Полезные свойства глобального объекта process
✓ Основные методы вывода и вывода данных в консоль
✓ Блокирующие и неблокирующие файловые операции
✓ Особенности 3 способов обработки ошибок и исключений
✓ Применение callback’ов для неблокирующих команд

Домашнее задание

1) Написать консольную игру "Орел или решка", в которой надо будет угадывать выпадающее число (1 или 2). В качестве аргумента программа может принимать имя файла для логирования результатов каждой партии. В качестве более продвинутой версии задания можете реализовать простейшую версию игры Blackjack.

2) Сделать программу-анализатор игровых логов. В качестве аргумента программа получает путь к файлу. Выведите игровую статистику: общее количество партий, количество выигранных / проигранных партий и их соотношение, максимальное число побед / проигрышей подряд.