**АРХИТЕКТУРА И ФИЛОСОФИЯ NODEJS**

**Основные принципы**

Каждая платформа имеет свою собственную философию — набор принципов, которые приняты сообществом разработчиков, и которые влияют на развитие платформы и то, как разрабатываются приложения. Некоторые из этих принципов вытекают из самой технологии, некоторые — всего лишь тенденции в сообществе разработчиков, а другие — эволюции различных идеологий.

В Node.js часть принципов исходят непосредственно от создателя, Райана Дала, от всех людей, которые внесли свой вклад в разработку ядра, а какие-то из принципов унаследованы от культуры JavaScript или под влиянием философии Unix.

Ни один из этих принципов не обязателен, и их всегда следует применять в соответствии со здравым смыслом, однако они могут оказаться чрезвычайно полезными при принятии решений относительно архитектуры и дизайна приложений.

**Small core**

Или принцип компактного ядра. Суть этого принципа состоит в том, что ядро должно содержать минимально необходимый набор функционала, весь остальной функционал должен находиться в пользовательских модулях. Этот принцип оказывает огромное влияние на культуру Node.js, поскольку он дает свободу сообществу, чтобы экспериментировать и быстро добавлять необходимый функционал к ядру вместо того, чтобы ждать, когда соответствующий функционал появится непосредственно в ядре.

Сохранение основного набора функционала максимально компактным, становится удобным не только с точки зрения поддержки, но и с точки зрения положительного воздействия, которое приводит к эволюции всей экосистемы.

Например, в ядре находится низкоуровневая функциональность, которая необходима почти всем сетевым программам: TCP, HTTP, DNS, файловая система, дочерние процессы и несколько других вещей. Что-то более объемное определенно не должно быть частью ядра.

**Small modules**

Nodejs использует концепцию модулей.

Модуль — это блок функционала для структурирования сложных приложений и изоляции приватных функций и переменных, которые явно не предназначенные для экспорта.

С точки зрения разработчиков Nodejs модуль должен быть максимально компактным и должен иметь одну четко определенную ответственность.

Такой подход имеет один очень большой недостаток — это проблема зависимостей в приложении. Если приложение достаточно сложное оно может зависеть от множества модулей. Причем эта зависимость может быть транзитивной, когда приложение зависит от модуля А, а модуль А в свою очередь зависит от модулей В и С. В таком случае построить дерево зависимостей вручную решая при этом возникающие конфликты между разными версиями модулей становится очень сложно.

Nodejs предлагает хороший инструмент, позволяющий решить подобные проблемы — это NPM менеджер пакетов. NPM устанавливает каждый пакет со своим собственным отдельным набором зависимостей, что позволяет программе зависеть от множества пакетов без конфликтов. Фактически, такой подход поднимает повторное использование кода на новый уровень, в результате чего приложения состоят из большого числа небольших, хорошо определенных зависимостей. Хотя это можно считать непрактичным или даже совершенно неосуществимым других платформах, в Node.js эта практика поощряется.

**Small surface area**

Помимо небольшого размера и объема, модули Node.js обычно также предоставляют минимальный набор функциональных возможностей для окружающего мира. Основное преимущество этого принципа в повышении удобства использования API, что означает, что API становится более понятным и менее подверженным ошибочному использованию.

Другая характеристика многих модулей Node.js — это то, что они созданы для использования, а не для расширения, то есть разработчики не должны пытаться наследоваться от компонентов модуля. Отсутствие возможности расширения может казаться негибким, но на самом деле оно имеет преимущество сокращения случаев использования модуля, упрощение его реализации, поддержки и повышения удобства его использования.

**Simplicity and pragmatism**

Проектирование простого, а не идеального полнофункционального программного обеспечения является хорошей практикой по нескольким причинам:

* — требуется меньше усилий для реализации,
* — позволяет ускорить реализацию с меньшими ресурсами,
* — легче адаптироваться и легче поддерживать и понимать.

Эти факторы способствуют вкладу сообщества разработчиков и позволяют самому программному обеспечению расти и улучшаться.

В Node.js этот принцип также поддерживается JavaScript, который является очень прагматичным языком программирования. Это не редкость для джаваскрипт видеть простые функции, замыкания и объектные литералы заменяющие сложные иерархии классов.

Чистые объектно-ориентированные проекты часто пытаются копировать реальный мир, используя математические термины компьютерной системы без учёта несовершенства и сложности реального мира.

Правда в том, что наше программное обеспечение всегда является приближением к реальности, и нам, вероятно, лучше получить что-то работающее, пораньше и с разумной сложностью, вместо того, чтобы пытаться создать совершенное программное обеспечение с огромными усилиями и тоннами кода.

**Особенности архитектуры Nodejs**

Node.js — это кроссплатформенная среда (т.е. среда способная работать более чем на одной операционной системе) с открытым исходным кодом (т.е. вы можете скачать код из репозитория и самостоятельно сделать сборку) для выполнения JavaScript на стороне сервера. В основе Nodejs лежит **событийно-ориентированное** асинхронное программирование с **неблокирующим вводом/выводом**.

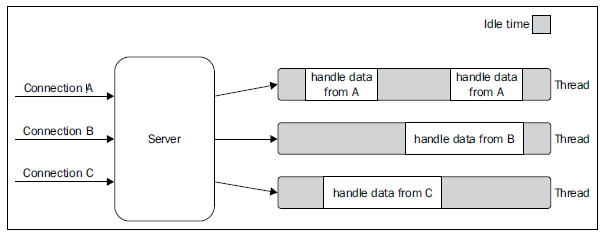
Рассмотрим недостатки блокирующего ввода/вывода.

Операции ввода/вывода определенно являются самыми медленными в сравнении с другими операциями компьютера. При этом операции ввода/вывода обычно не дорогостоящие с точки зрения процессорного времени, но они добавляет задержку между моментом начала и моментом завершения операции.

В традиционном программировании с блокирующим вводом-выводом вызов соответствующей функции, в ходе выполнения которой необходимо осуществить операцию ввода-вывода, блокирует выполнение потока до завершения операции. Это может занять от нескольких миллисекунд, в случае доступа к диску, до нескольких минут или даже больше, в случае, если данные генерируются из действий пользователя, например, ожидания нажатия клавиши.

Следует отметить, что веб-сервер, реализованный с использованием блокирующих операций ввода-вывода, не будет иметь возможности обрабатывать несколько соединений в одном потоке, каждая операция ввода / вывода будет блокировать обработку любого другого соединения.

По этой причине традиционный подход для обработки параллелизма в веб-серверах — это запуск потока (или повторное использование потока, взятого из пула) для каждого параллельного соединения, которые необходимо обработать. Таким образом, когда поток заблокируется операцией ввода-вывода это не повлияет на доступность сервера для выполнения других запросов, поскольку они обрабатываются в отдельных потоках.



На данной схеме показано количество времени, в течение которого каждый поток простаивает, ожидая получения новых данных от соответствующего соединения. Теперь, если мы примем во внимание, что любая операция ввода-вывода может блокировать поток, например, во время взаимодействия с базой данных или с файловой системой, то поток может блокироваться много раз, ожидая результат операции ввода-вывода.

К сожалению, поток не является дешевым с точки зрения системных ресурсов, он потребляет память и вызывает переключение контекста, поэтому поток, в котором будет выполняться продолжительная операция ввода/вывода будет простаивать большую часть времени, и это не лучшее решение с точки зрения эффективности.

**Неблокирующий ввод/вывод**

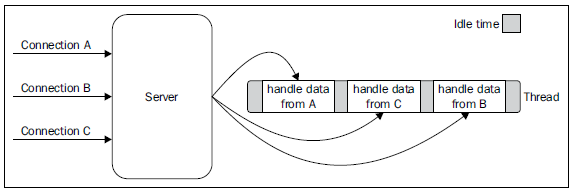
Современные операционные системы предлагают другой механизм доступа к ресурсам, который называется неблокирующий ввод/вывод. При выполнении таких операций управление возвращается сразу в вызывающий код без ожидания результата операции ввода/вывода.

Чтобы получить результат операции неблокирующего ввода/вывода существует два подхода.

В одном случае мы в цикле будем обращаться к системному ресурсу проверяя наличие результата нашей операции. Такой подход называется busy-waiting.  
Второй подход называется Event demultiplexing. Суть этого подхода в том, что специальный компонент event demultiplexer создает очередь событий, генерируемых операциями ввода/вывода.

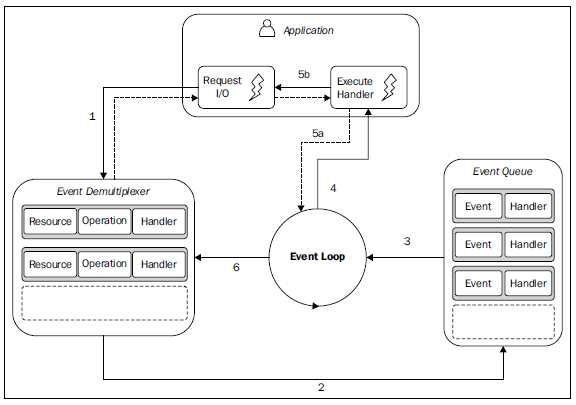
Каждое такое событие обрабатывается в так называемом цикле событий. И это позволяет выполнять несколько операций ввода/вывода в одном потоке и сократить время простоя этого потока до минимума. Поэтому такой подход называется событийно-ориентированным асинхронным программированием.

На следующем рисунке изображено как веб-сервер мог бы обрабатывать запросы из нескольких соединений используя event demultiplexer и один поток:



Обобщая все вышесказанное можно подчеркнуть, что существует специальный шаблон проектирования, полностью описывающий подобный подход и который называется Reactor.

Основная идея этого шаблона заключается в том, чтобы иметь обработчик (который в Node.js представлен функцией обратного вызова callback), связанный с каждой операцией ввода / вывода и который будет вызываться, как только событие будет создано и обработано в цикле событий.

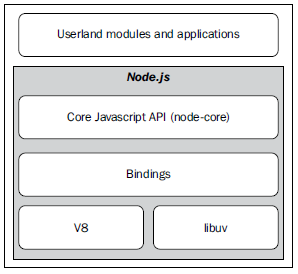


Можно пошагово описать, что происходит в приложении, использующем шаблон реактор:

1. Приложении генерирует новые операции ввода/вывода отправляя запрос к Event Demultiplexer. Приложение также указывает обработчик, который должен быть вызван, когда операция завершиться, в таком обработчике обычно добавляется код, который должен обработать результат операции. Запрос к Event Demultiplexer это неблокирующий вызов, поэтому управление тут же возвращается в приложение.
2. По завершению операции ввода/вывода Event Demultiplexer помещает соответствующее событие в очередь событий.
3. Очередь событий итерируется по всем доступным событиям.
4. Для каждого события вызывается обработчик.
5. По завершению обработки результата операции ввода/вывода обработчик может передать управление в цикл событий сразу или вызвать новую асинхронную операцию, которая будет отправлена в Event Demultiplexer перед тем, как управление будет возвращено в цикл событий.
6. Когда все события будут обработаны цикл событий заблокируется пока Event Demultiplexer не запустит новый цикл.

Следует отметить, что каждая операционная система предлагает свой собственный интерфейс для Event Demultiplexer. Кроме того, каждая операция ввода /вывода может иметь свои особенности в зависимости от типа ресурса даже на одной и той же операционной системе. Именно поэтому Node.js содержит специальную библиотеку libuv, которая совместима с основными платформами и предлагает высокоуровневую абстракцию реализации интерфейса для Event Demultiplexer.

**Финальная архитектура Nodejs**



Шаблон реактор и libuv являются основными строительными блоками Node.js, но также необходимы следующие компоненты для построения полной платформы:

* — Bindings которые служат оберткой для libuv и другой низкоуровневой функциональности и предоставляют интерфейс для JavaScript.
* — V8, JavaScript движок, первоначально разработанный Google для Chrome. Это одна из причин, почему Node.js настолько быстр и эффективен. V8 известен благодаря революционному дизайну, скорости и эффективному управлению памятью. Он транслирует JavaScript в машинный код и позволяет добавить новый фичи в JavaScript. Например, print(‘hello world’) не валидный код в Node.js и мы получим ошибку, если попробуем скомпилировать его. Но мы можем добавить свою собственную имплементацию этой функции на С++ для V8. Это позволит JavaScript понимать больше чем предполагает ECMAScript стандарт. Возможность писать код на C ++ делая его доступным для JavaScript, позволяет добавлять дополнительные функции в JavaScript.
* — Ядро — JS-библиотека, которая реализует высокоуровневый API Node.js.