Principios básicos

Todos los comienzos son difíciles.
–Ovidio

La idea original de JavaScript era aportar más dinamismo a las páginas web. El lenguaje de programación pretendía compensar las debilidades de HTML a la hora de responder a las entradas del usuario. La historia de JavaScript se remonta a 1995, cuando fue desarrollado bajo el nombre en código Mocha por Brendan Eich, un desarrollador de Netscape. Uno de los hechos más destacables de JavaScript es que el primer prototipo de este lenguaje de éxito y uso mundial se desarrolló en tan solo 10 días. En el mismo año de su creación, Mocha pasó a llamarse LiveScript y, finalmente, JavaScript en una colaboración entre Netscape y Sun. Esto se hizo principalmente con fines de marketing, ya que en ese momento se suponía que Java se convertiría en el lenguaje líder en el desarrollo web del lado del cliente.

Node.js ES2015 Support — kangax's compat-table applied only to Node.js — show code examples — Learn more — Nightly requires harmony flag — Created by William Kapke

	23.0.0 (99% complete)	22.6.0 (99% complete)	21.7.3 (99% complete)	21.2.0 (99% complete)	20.16.0 (99% complete)	20.11.1 (99% complete)
optimisation						
proper tail calls (tail call optimisation)						
direct recursion	Error	Error	Error	Error	Error	Error
mutual recursion	Error	Error	Error	Error	Error	Error
syntax						
default function parameters						
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes
explicit undefined defers to the default	Yes	Yes	Yes	Yes	Yes	Yes
defaults can refer to previous params	Yes	Yes	Yes	Yes	Yes	Yes
arguments object interaction	Yes	Yes	Yes	Yes	Yes	Yes
temporal dead zone	Yes	Yes	Yes	Yes	Yes	Yes
separate scope	Yes	Yes	Yes	Yes	Yes	Yes
new Function() support	Yes	Yes	Yes	Yes	Yes	Yes
rest parameters						
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes
function 'length' property	Yes	Yes	Yes	Yes	Yes	Yes
arguments object interaction	Yes	Yes	Yes	Yes	Yes	Yes
can't be used in setters	Yes	Yes	Yes	Yes	Yes	Yes
new Function() support	Yes	Yes	Yes	Yes	Yes	Yes
spread syntax for iterable objects						
with arrays, in function calls	Yes	Yes	Yes	Yes	Yes	Yes
with arrays, in array literals	Yes	Yes	Yes	Yes	Yes	Yes
with sparse arrays, in function calls	Yes	Yes	Yes	Yes	Yes	Yes
with sparse arrays, in array literals	Yes	Yes	Yes	Yes	Yes	Yes
with strings, in function calls	Yes	Yes	Yes	Yes	Yes	Yes
with strings, in array literals	Yes	Yes	Yes	Yes	Yes	Yes
with astral plane strings, in function calls	Yes	Yes	Yes	Yes	Yes	Yes
with astral plane strings, in array literals	Yes	Yes	Yes	Yes	Yes	Yes
with generator instances, in calls	Yes	Yes	Yes	Yes	Yes	Yes

Figura 1: Compatibilidad con las características de JavaScript en Node.js (http://node.green).

Convencidos por el éxito de JavaScript, Microsoft también integró un lenguaje de scripting en Internet Explorer 3 en 1996. Éste fue el nacimiento de JScript, que era en gran parte compatible con JavaScript, pero con características adicionales añadidas.

Hoy en día, la rivalidad mutua de las dos empresas se conoce como la "guerra de los navegadores". El desarrollo aseguró que los dos motores de JavaScript mejoraran constantemente tanto en conjunto de características como en rendimiento, que es la razón principal del éxito de JavaScript en la actualidad.

En 1997, se creó el primer borrador del estándar del lenguaje en Ecma International. Todo el núcleo del lenguaje de script está registrado bajo la designación críptica ECMA-262 o ISO/IEC 16262.
El estándar actual se puede encontrar en:
www.ecmainternational.org/publications/standards/Ecma-262.htm.

Debido a esta estandarización, el JavaScript independiente del proveedor también se conoce como ECMAScript. Hasta hace unos años, el estándar ECMAScript se versionaba en números enteros a partir del 1. Desde la versión 6, las versiones también se proporcionan con números de año. Por lo tanto, la versión 8 de ECMAScript se conoce como ECMAScript 2017. Por lo general, puedes asumir que los fabricantes son bastante compatibles con las versiones anteriores del estándar.

Debe habilitar las funciones más nuevas mediante indicadores de configuración en el navegador o simularlas mediante polyfills (es decir, recrear las funciones en JavaScript). La tabla de compatibilidad de kangax ofrece una buena descripción general de las funciones compatibles actualmente, que se puede encontrar en https://262.ecma-international.org/6.0/#sec-functiondeclarationinstantiation.
Puedes acceder a una versión adaptada para Node.js en http://node.green/.

JavaScript es liviano, relativamente fácil de aprender y tiene un enorme ecosistema de frameworks y bibliotecas. Por estas razones, JavaScript es uno de los lenguajes de programación más exitosos del mundo. Este éxito se puede respaldar con números: desde 2008, JavaScript ha estado en los dos primeros lugares en las tendencias de lenguaje de GitHub. En 2021, JavaScript fue superado por Python en el puesto número uno y ahora está en segundo lugar en las tendencias de lenguaje.

Node.js se basa en este exitoso lenguaje de scripting y ha tenido un ascenso meteórico. Este primer documento servirá como una introducción al mundo de Node.js, mostrándote cómo se construye la plataforma y dónde y cómo puedes usar Node.js.

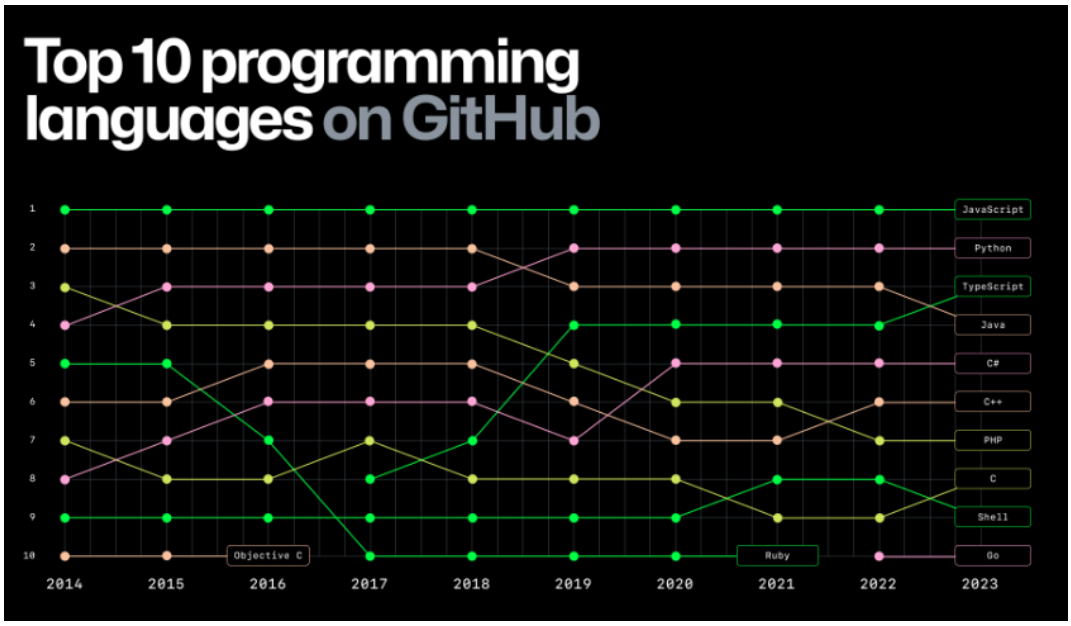


Figura 2: Principales lenguajes en GitHub según las solicitudes de incorporación de cambios (octoverse.github.com)

La historia de Node.js

Para ayudarte a comprender mejor qué es Node.js y cómo surgieron algunas de las decisiones de desarrollo, exploremos la historia de la plataforma.

Orígenes

Node.js fue desarrollado originalmente por Ryan Dahl, un estudiante de doctorado en matemáticas que la pensó mejor abandonó sus esfuerzos y en su lugar prefirió viajar a Sudamérica con un boleto de ida y muy poco dinero en el bolsillo. Allí, se mantuvo a flote enseñando inglés. Durante este tiempo, entró en contacto con PHP y Ruby y descubrió su afición por el desarrollo web. El problema de trabajar con el framework Ruby, llamado Rails, era que no podía manejar solicitudes concurrentes sin ninguna solución alternativa. Las aplicaciones eran demasiado lentas y utilizaban la CPU por completo. Dahl encontró una solución a sus problemas con Mongrel, un servidor web para aplicaciones basadas en Ruby.

A diferencia de los servidores web tradicionales, Mongrel responde a las solicitudes de los usuarios y genera respuestas de forma dinámica, mientras que de lo contrario solo se entregan páginas HTML estáticas.

La tarea que en realidad llevó a la creación de Node.js es bastante trivial desde el punto de vista actual. En 2005, Dahl estaba buscando una forma elegante de implementar una barra de progreso para las cargas de archivos.

Sin embargo, las tecnologías disponibles en ese momento solo permitían soluciones insatisfactorias. En cuanto a las transferencias de archivos, se utilizaba HTTP para archivos relativamente pequeños y el Protocolo de Transferencia de Archivos (FTP) para archivos más grandes. El estado de la carga se consultaba mediante sondeo largo, que es una técnica en la que el cliente envía solicitudes de larga duración al servidor y el servidor utiliza el canal abierto para las respuestas.

El primer intento de Dahl de implementar una barra de progreso tuvo lugar en Mongrel. Después de enviar el archivo al servidor, verificaba el estado de la carga utilizando una gran cantidad de solicitudes JavaScript y XML asíncronas (AJAX) y lo mostraba gráficamente en una barra de progreso. Sin embargo, la desventaja de esta implementación era el enfoque de un solo subproceso de Ruby y la gran cantidad de solicitudes que se requerían.

Otro enfoque prometedor fue la implementación en C. En este caso, las opciones de Dahl no se limitaban a un único subproceso. Sin embargo, C como lenguaje de programación para la web tiene una desventaja decisiva: sólo un pequeño número de desarrolladores se entusiasman con este campo de aplicación. Dahl también se enfrentó a este problema y descartó este enfoque al poco tiempo.

La búsqueda de un lenguaje de programación adecuado para resolver su problema continuó y lo llevó a lenguajes de programación funcionales como Haskell. El enfoque de Haskell se basa en la entrada/salida (E/S) sin bloqueo, lo que significa que todas las operaciones de lectura y escritura son asíncronas y no bloquean la ejecución de un programa. Esto permite que el lenguaje siga siendo un único subproceso en su núcleo y no introduce los problemas que surgen de la programación paralela. Entre otras cosas, no es necesario sincronizar recursos y no se producen problemas debido al tiempo de ejecución de subprocesos paralelos. Sin embargo, Dahl todavía no estaba completamente satisfecho con esta solución y estaba buscando otras opciones.

Nacimiento de Node.js

Dahl encontró entonces la solución con la que finalmente estaba satisfecho: JavaScript. Se dio cuenta de que este lenguaje de scripting podía satisfacer todas sus necesidades. JavaScript ya estaba establecido en la web desde hacía años, por lo que existían motores potentes y una gran cantidad de desarrolladores. En enero de 2009, comenzó a trabajar en su implementación para JavaScript del lado del servidor, lo que puede considerarse como el nacimiento de Node.js. Otra razón para implementar la solución en JavaScript, según Dahl, fue el hecho de

que los desarrolladores de JavaScript no previeron este ámbito de uso. En ese momento, no existía ningún servidor web nativo en JavaScript, no podía manejar archivos en un sistema de archivos y no había implementación de sockets para comunicarse con otras aplicaciones o sistemas. Todos estos puntos hablaban a favor de JavaScript como base para una plataforma para aplicaciones web interactivas porque aún no se habían tomado decisiones en este ámbito y, en consecuencia, tampoco se habían cometido errores.

La arquitectura de JavaScript también abogaba por una implementación de este tipo. El enfoque de las funciones de nivel superior (es decir, funciones que no están vinculadas a ningún objeto, están disponibles libremente y se pueden asignar a variables) ofrece un alto grado de flexibilidad en el desarrollo y permite enfoques funcionales para las soluciones.
Por lo tanto, Dahl seleccionó otras bibliotecas además del motor JavaScript, que es responsable de interpretar el código fuente de JavaScript, y las reunió en una plataforma.
En septiembre de 2009, Isaac Schlueter comenzó a trabajar en un administrador de paquetes para Node.js, el Node Package Manager (npm).

El gran avance de Node.js

Después de que Dahl integrara todos los componentes y creara los primeros ejemplos ejecutables en la nueva plataforma Node.js, necesitaba una forma de presentar Node.js al público. Esto también se volvió necesario porque sus recursos financieros se redujeron considerablemente debido al desarrollo de Node.js, y hubiera tenido que dejar de trabajar en Node.js si no hubiera encontrado patrocinadores. Eligió la conferencia de JavaScript JSConf EU en noviembre de 2009 en Berlín como su plataforma de presentación.

Dahl puso todos los huevos en una canasta. Si la presentación era un éxito y encontraba patrocinadores que apoyaran su trabajo en Node.js, podría continuar su participación; si no, casi un año de trabajo habría sido en vano. En una charla entusiasta, presentó Node.js a la audiencia y mostró cómo crear un servidor web completamente funcional con solo unas pocas líneas de código JavaScript. Como otro ejemplo, presentó una implementación de un servidor de chat de Internet Relay Chat (IRC). El código fuente de esta demostración constaba de unas 400 líneas. Con este ejemplo, demostró la arquitectura y, por tanto, las ventajas de Node.js, al tiempo que lo hacía tangible para el público.

La grabación de esta presentación se puede encontrar en:
www.youtube.com/watch?v=EeYvFl7li9E.
La presentación no perdió el rumbo y llevó a Joyent a participar como patrocinador de Node.js. Joyent es un proveedor de software y servicios con sede en San Francisco que ofrece soluciones de alojamiento e infraestructura en la nube. Con su compromiso, Joyent incluyó el software de código abierto Node.js en su cartera de productos y puso Node.js a disposición de sus clientes como parte de sus ofertas de alojamiento.

Dahl fue contratado por Joyent y se convirtió en un mantenedor a tiempo completo de Node.js a partir de ese momento.

Node.js conquista Windows

Los desarrolladores dieron un paso importante hacia la difusión de Node.js al introducir soporte nativo para Windows en la versión 0.6 en noviembre de 2011. Hasta ese momento, Node.js solo se podía instalar de forma incómoda en Windows a través de Cygwin.
Desde la versión 0.6.3 en noviembre de 2011, npm ha sido una parte integral de los paquetes de Node.js y, por lo tanto, se entrega automáticamente cuando se instala Node.js.

Sorprendentemente, a principios de 2012, Dahl anunció que finalmente se retiraría del desarrollo activo después de tres años de trabajar en Node.js. Le entregó las riendas del desarrollo a Schlueter. Este último, al igual que Dahl, era empleado de Joyent y participó activamente en el desarrollo del núcleo de Node.js. El cambio inquietó a la comunidad, ya que no estaba claro si la plataforma continuaría desarrollándose sin Dahl. Una señal de que la comunidad de Node.js consideró lo suficientemente fuerte como para un desarrollo posterior sólido llegó con el lanzamiento de la versión 0.8 en junio de 2012, que principalmente tenía como objetivo mejorar significativamente el rendimiento y la estabilidad de Node.js.

Con la versión 0.10 en marzo de 2013, una de las interfaces centrales de Node.js cambió: la interfaz de programación de aplicaciones (API) Stream. Con este cambio, se hizo posible extraer datos de forma activa de un flujo. Debido a que la API anterior ya se usaba ampliamente, ambas interfaces continuaron siendo compatibles.

io.js: La bifurcación de Node.js

En enero de 2014, se produjo otro cambio en la gestión del proyecto Node.js. Schlueter, que abandonó el mantenimiento de Node.js para pasar a su propia empresa (llamada npmjs), la que aloja el repositorio npm, fue reemplazado por TJ Fontaine. Bajo su dirección, se publicó la versión 0.12 en febrero de 2014. Una crítica habitual a Node.js en aquel momento era que el framework aún no había alcanzado la versión supuestamente estable 1.0, lo que impedía a numerosas empresas utilizar Node.js para aplicaciones críticas.

Muchos desarrolladores no estaban contentos con Joyent, que había proporcionado mantenedores para Node.js desde Dahl, por lo que la comunidad se fracturó en diciembre de 2014. El resultado fue io.js, una bifurcación de Node.js que se desarrolló por separado de la plataforma original. Como resultado, en febrero de 2015 se fundó la Fundación Node.js, una entidad independiente que se encargó del desarrollo posterior de io.js. Al mismo tiempo, se publicó la versión 0.12 del proyecto Node.js.

Node.js se reúne

En junio de 2015, los dos proyectos io.js y Node.js se fusionaron en la Fundación Node.js. Con la versión 4 del proyecto, se completó la fusión. El desarrollo posterior de la plataforma Node.js ahora está coordinado por un comité dentro de la Fundación Node.js en lugar de por individuos. Como resultado, vemos lanzamientos más frecuentes y una versión estable con soporte a largo plazo (LTS, Long-Term Support).

Deno: una nueva estrella en el cielo de JavaScript

Desde la fusión de io.js y Node.js, las cosas se han vuelto más tranquilas en torno a Node.js. Los lanzamientos regulares, la estabilidad y también la integración de nuevas características, como subprocesos de trabajo, HTTP/2 o ganchos de rendimiento, mantienen el buen humor dentro de la comunidad. Y justo cuando las cosas empezaban a volverse casi demasiado tranquilas en torno a Node.js, un viejo conocido, Dahl, subió al escenario nuevamente en 2018 para presentar una nueva plataforma de JavaScript llamada Deno durante su charla, “10 cosas de las que me arrepiento acerca de Node.js”.

La idea detrás de Deno es crear un Node.js mejor, libre de las restricciones de compatibilidad con versiones anteriores que impiden saltos revolucionarios en el desarrollo. Por ejemplo, Deno se basa en TypeScript de forma predeterminada y agrega un sistema de módulos fundamentalmente diferente. El núcleo de Deno también es bastante diferente de Node.js, ya que está escrito casi en su totalidad en Rust.

Sin embargo, también hay algunas características comunes.
Por ejemplo, Deno se basa en el probado motor V8, que también forma el corazón de Node.js. Y tampoco hay que prescindir de la enorme cantidad de paquetes npm. Para ello, Deno proporciona una capa de compatibilidad.

OpenJS Foundation

En 2015, se creó la Node.js Foundation para coordinar el desarrollo de la plataforma. La fundación era un proyecto subordinado a la Linux Foundation. En 2019, la JS Foundation y la Node.js Foundation se fusionaron para formar la OpenJS Foundation. Además de Node.js, incluye varios otros proyectos populares como webpack, ESLint y Electron.

Organización de Node.js

La comunidad detrás de Node.js ha aprendido las lecciones del pasado. Por este motivo, ya no hay individuos al mando de Node.js, sino un comité de varias personas que dirigen el desarrollo de la plataforma.

Comité de Dirección Técnica

El comité de dirección técnica (TSC, Technical Steering Committee) es responsable del desarrollo de la plataforma. El número de miembros del TSC no está limitado, pero se prevé que sean entre 6 y 12, normalmente seleccionados entre los colaboradores de la plataforma. Las tareas del TSC son las siguientes:

- Establecer la dirección técnica de Node.js
- Realizar el control del proyecto y del proceso
- Definir la política de contribución
- Gestionar el repositorio de GitHub
- Establecer las directrices de conducta
- Gestionar la lista de colaboradores

El TSC celebra reuniones semanales a través de Google Hangouts para coordinar y debatir cuestiones actuales. Muchas de estas reuniones se publican a través del canal de YouTube de Node.js (www.youtube.com/c/nodejs+foundation).

Colaboradores

Node.js es un proyecto de código abierto desarrollado en un repositorio de GitHub. Como sucede con todos los proyectos más grandes de este tipo, un grupo de personas, llamadas colaboradores, tienen acceso de escritura a este repositorio. Además de acceder al repositorio, un colaborador puede acceder a los trabajos de integración continua.

Las tareas típicas de un colaborador incluyen brindar soporte a los usuarios y a los nuevos colaboradores, mejorar el código fuente y la documentación de Node.js, revisar las solicitudes de incorporación de cambios y los problemas (con los comentarios correspondientes), participar en grupos de trabajo y fusionar las solicitudes de incorporación de cambios.

Los colaboradores son designados por el TSC. Por lo general, el rol de un colaborador está precedido por una contribución significativa al proyecto a través de una solicitud de incorporación de cambios.

Comité de la comunidad

Como su nombre lo indica, el Comité de la comunidad (CommComm) se encarga de la comunidad de Node.js con un enfoque especial en la educación y la cultura. El CommComm coordina reuniones periódicas, que se registran en un repositorio de GitHub independiente (https://github.com/nodejs/community committee). El CommComm existe para dar voz a la comunidad y así contrarrestar los intereses comerciales de las corporaciones. Aunque actualmente se dice que el comité está jubilado.

Grupos de trabajo

El TSC establece varios grupos de trabajo para que los expertos aborden temas específicos por separado. Los siguientes son algunos ejemplos de estos grupos de trabajo:

- **Lanzamiento**
  Este grupo de trabajo administra el proceso de lanzamiento de la plataforma Node.js, define el contenido de los lanzamientos y se ocupa de los LTS.

- **Streams**
  El grupo de trabajo de streams trabaja para mejorar la API Stream de la plataforma.

- **Docker**
  Este grupo de trabajo administra las imágenes oficiales de Docker de la plataforma Node.js y se asegura de que se mantengan actualizadas.

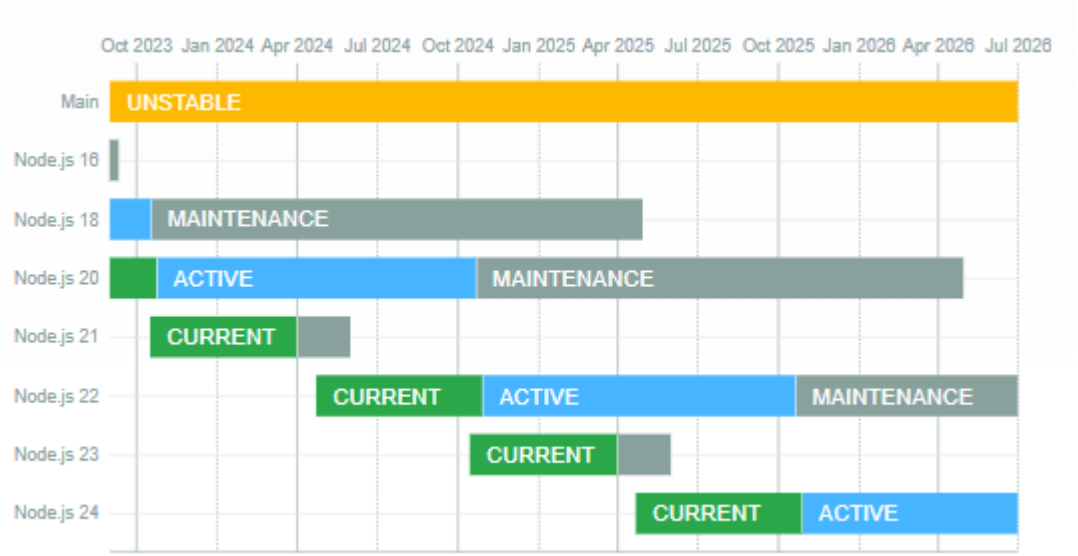
Fundación OpenJS

La Fundación OpenJS constituye el paraguas para el desarrollo de Node.js. Su función es similar a la de la Fundación Linux para el desarrollo del sistema operativo Linux. La Fundación OpenJS se fundó como un organismo independiente para seguir desarrollando Node.js. Su lista de miembros fundadores incluye empresas como IBM, Intel, Joyent y Microsoft. La Fundación OpenJS se financia mediante donaciones y contribuciones de empresas y miembros individuales.

Versiones de Node.js

Uno de los puntos más críticos con Node.js antes de la bifurcación de io.js era que su desarrollo era muy lento. Los lanzamientos regulares y predecibles son un criterio de selección importante, especialmente en el uso empresarial. Por este motivo, después de fusionar Node.js e io.js, los desarrolladores de Node.js acordaron un cronograma de lanzamiento transparente con lanzamientos regulares y una versión LTS que se proporciona con actualizaciones durante un período de tiempo más largo. El cronograma de lanzamiento prevé un lanzamiento principal cada medio año. La tabla 1 muestra el cronograma de lanzamiento de Node.js.

Tabla 1: Cronograma de lanzamiento de Node.js.



Como puedes ver en el cronograma de lanzamiento, las versiones con un número de versión par son lanzamientos LTS, mientras que las impares son lanzamientos con un período de soporte más corto. La figura en la siguiente página muestra más detalles.

Lanzamientos con soporte a largo plazo

Una versión de Node.js con un número de versión par se convierte en un lanzamiento LTS tan pronto como se lanza la siguiente versión impar. La versión LTS se mantiene activa durante un período de 12 meses. Durante este tiempo, la versión recibe lo siguiente:

- Corrección de errores
- Actualizaciones de seguridad
- Actualizaciones menores de npm

- Actualizaciones de documentación
- Mejoras de rendimiento que no comprometen las aplicaciones existentes
- Cambios en el código fuente que simplifican la integración de futuras mejoras

Después de esta fase, la versión entra en una fase de mantenimiento de 12 meses durante la cual la versión seguirá recibiendo actualizaciones de seguridad. Sin embargo, en este caso solo se corrigen errores críticos y brechas de seguridad. En total, los desarrolladores de la plataforma Node.js respaldan una versión LTS durante un período de 30 meses.

Node.js Version	Codename	Release Date	npm	
v22.6.0	-	2024-08-06	v10.8.2	Releases Changelog Docs
v21.7.3	-	2024-04-10	v10.5.0	Releases Changelog Docs
v20.16.0	Iron	2024-07-24	v10.8.1	Releases Changelog Docs
v19.9.0	-	2023-04-10	v9.6.3	Releases Changelog Docs
v18.20.4	Hydrogen	2024-07-08	v10.7.0	Releases Changelog Docs
v17.9.1	-	2022-06-01	v8.11.0	Releases Changelog Docs
v16.20.2	Gallium	2023-08-08	v8.19.4	Releases Changelog Docs
v15.14.0	-	2021-04-06	v7.7.6	Releases Changelog Docs
v14.21.3	Fermium	2023-02-16	v6.14.18	Releases Changelog Docs
v13.14.0	-	2020-04-29	v6.14.4	Releases Changelog Docs
v12.22.12	Erbium	2022-04-05	v6.14.16	Releases Changelog Docs
v11.15.0	-	2019-04-30	v6.7.0	Releases Changelog Docs
v10.24.1	Dubnium	2021-04-06	v6.14.12	Releases Changelog Docs
v9.11.2	-	2018-06-12	v5.6.0	Releases Changelog Docs
v8.17.0	Carbon	2019-12-17	v6.13.4	Releases Changelog Docs
v7.10.1	-	2017-07-11	v4.2.0	Releases Changelog Docs
v6.17.1	Boron	2019-04-03	v3.10.10	Releases Changelog Docs
v5.12.0	-	2016-06-23	v3.8.6	Releases Changelog Docs
v4.9.1	Argon	2018-03-29	v2.15.11	Releases Changelog Docs
v0.12.18	-	2017-02-22	v2.15.11	Releases Changelog Docs

Beneficios de Node.js

El historial de desarrollo de Node.js muestra una cosa muy clara: está conectado directamente a Internet. Con JavaScript como base, tiene la capacidad de lograr resultados visibles muy rápidamente con aplicaciones implementadas en Node.js. La plataforma en sí es muy liviana y se puede instalar en casi cualquier sistema. Como es habitual en los lenguajes de programación, las aplicaciones Node.js también omiten un proceso de desarrollo pesado, por lo que puedes comprobar los resultados directamente.

Además de la rápida implementación inicial, también puede reaccionar de forma muy flexible a los cambios de requisitos durante el desarrollo de aplicaciones web. Debido a que el núcleo de JavaScript está estandarizado por ECMAScript, el lenguaje representa una base confiable con la que se pueden implementar aplicaciones aún más extensas. Las características del lenguaje disponibles están bien documentadas tanto en línea como en libros de referencia.

Además, muchos desarrolladores son competentes en JavaScript y pueden implementar aplicaciones aún más grandes utilizando este lenguaje.
Debido a que Node.js utiliza el mismo motor de JavaScript que Google Chrome, el motor V8, todas las características del lenguaje también están disponibles aquí, y los desarrolladores que dominan JavaScript pueden familiarizarse con la nueva plataforma con relativa rapidez.

La larga historia de desarrollo de JavaScript ha producido una serie de motores de alto rendimiento. Una razón para este desarrollo es que los diversos fabricantes de navegadores siempre estaban desarrollando sus propias implementaciones de motores de JavaScript, por lo que había una sana competencia en el mercado cuando se trataba de ejecutar JavaScript en el navegador. Por un lado, esta competencia ha hecho que ahora JavaScript se interprete muy rápidamente y, por otro, ha hecho que los fabricantes se pongan de acuerdo sobre determinados estándares.

Node.js, como plataforma para JavaScript del lado del servidor, se diseñó como un proyecto de código abierto desde el principio de su desarrollo. Por este motivo, rápidamente se desarrolló una comunidad activa en torno al núcleo de la plataforma, que se ocupa principalmente del uso de Node.js en la práctica, pero también del desarrollo y la estabilización de la plataforma. Los recursos sobre Node.js van desde tutoriales para ayudarte a empezar hasta artículos sobre temas avanzados como el control de calidad, la depuración o el escalado.

La mayor ventaja de un proyecto de código abierto como Node.js es que la información está disponible de forma gratuita y las preguntas y los problemas se pueden resolver con bastante rapidez y competencia a través de una amplia variedad de canales de comunicación o de la propia comunidad.

Áreas de uso de Node.js

Desde una simple herramienta de línea de comandos hasta un servidor de aplicaciones para aplicaciones web que se ejecutan en un clúster con numerosos nodos, Node.js se puede utilizar en cualquier lugar. El uso de una tecnología depende en gran medida del problema a resolver, las preferencias personales y el nivel de conocimiento de los desarrolladores. Por este motivo, no solo debes conocer las características clave de Node.js, sino que también debes tener una idea de cómo trabajar con la plataforma. Solo puedes cumplir con el segundo punto si tienes la oportunidad de unirte a un proyecto Node.js existente o adquirir experiencia en el mejor de los casos con proyectos más pequeños que implementes.

Pero ahora pasemos a los datos más importantes del framework:

- **JavaScript puro**
  Al trabajar con Node.js, no tienes que aprender un nuevo dialecto del lenguaje porque puedes recurrir al núcleo del lenguaje JavaScript. Hay interfaces estandarizadas y bien documentadas disponibles para acceder a los recursos del sistema. Sin embargo, como alternativa a JavaScript, también puedes escribir tu aplicación Node.js en TypeScript, traducir el código fuente a JavaScript y ejecutarlo con Node.js.

- **Motor optimizado**
  Node.js se basa en el motor JavaScript V8 de Google.
  En este caso, se beneficia sobre todo del desarrollo constante del motor, donde las últimas características del lenguaje se admiten ya después de muy poco tiempo.

- **E/S sin bloqueo**
  Todas las operaciones que no se realizan directamente en Node.js no bloquean la ejecución de tu aplicación. El principio de Node.js es que todo lo que la plataforma no tiene que hacer directamente se externaliza al sistema operativo, otras aplicaciones u otros sistemas. Esto le da a la aplicación la capacidad de responder a solicitudes adicionales o procesar tareas en paralelo. Una vez que se completa el procesamiento de una tarea, el proceso de Node.js recibe retroalimentación y puede procesar la información más adelante.

- **Un solo subproceso**
  Una aplicación Node.js típica se ejecuta en un solo proceso.
  Durante mucho tiempo no ha existido el multithreading y la concurrencia inicialmente solo se preveía en forma de la E/S no bloqueante ya descrita. Por lo tanto, todo el código que escribas tú mismo puede bloquear tu aplicación. Por este motivo, debes prestar atención al desarrollo que ahorre recursos. Si aún, así es necesario

procesar tareas en paralelo, Node.js te ofrece soluciones para esto en forma del módulo *child_process*, que te permite crear tus propios procesos secundarios.

Para desarrollar tu aplicación de la mejor manera posible, debes tener al menos una visión general de los componentes y su funcionamiento. El más importante de estos componentes es el motor V8.

El núcleo: motor V8

Para que tú, como desarrollador, puedas evaluar si una tecnología se puede utilizar en un proyecto, debes estar lo suficientemente familiarizado con las características de esa tecnología. Las secciones que siguen ahora profundizan en los detalles internos de Node.js para mostrarte los componentes que conforman la plataforma y cómo puedes usarlos para el beneficio de una aplicación.

El componente central y, por lo tanto, el más importante de la plataforma Node.js es el motor V8 de JavaScript desarrollado por Google (para obtener más información, visita la página del Proyecto V8 en https://code.google.com/p/v8/). El motor de JavaScript es responsable de interpretar y ejecutar el código fuente de JavaScript. No existe un solo motor para JavaScript; en cambio, los diferentes fabricantes de navegadores utilizan sus propias implementaciones.

Uno de los problemas con JavaScript es que cada motor se comporta de manera ligeramente diferente. La estandarización de ECMAScript intenta encontrar un denominador común confiable para que tú, como desarrollador de aplicaciones de JavaScript, tengas menos incertidumbre de la que preocuparte. La competencia entre los motores de JavaScript resultó en una serie de motores optimizados, todos con el objetivo de interpretar el código JavaScript lo más rápido posible. Con el tiempo, varios motores se han establecido en el mercado: JaegerMonkey de Mozilla, Nitro de Apple y el motor V8 de Google, entre otros.
Microsoft, por su parte, utiliza la misma base técnica que Chrome para su navegador Edge, por lo que también utiliza el motor V8.

Node.js utiliza el motor V8 de Google. Este motor ha sido desarrollado por Google desde 2006, principalmente en Dinamarca, en colaboración con la Universidad de Aarhus. El área de uso principal del motor es el navegador Chrome de Google, donde se encarga de interpretar y ejecutar código JavaScript. El objetivo de desarrollar un nuevo motor JavaScript era mejorar significativamente el rendimiento de la interpretación de JavaScript. El motor ahora implementa completamente el estándar ECMAScript ECMA-262 en la quinta versión y gran parte de la sexta versión. El motor V8 en sí está escrito en C++, se ejecuta en varias plataformas y está disponible bajo la licencia Berkeley Source Distribution (BSD) como

software de código abierto para que cualquier desarrollador lo use y mejore. Por ejemplo, puedes integrar el motor en cualquier aplicación C++.

Como es habitual en JavaScript, el código fuente no se compila antes de la ejecución, sino que los archivos que contienen el código fuente se leen directamente cuando se inicia la aplicación.
Al iniciar la aplicación se inicia un nuevo proceso Node.js.
Aquí es donde tiene lugar la primera optimización del motor V8. El código fuente no se interpreta directamente, sino que primero se traduce a código de máquina, que luego se ejecuta. Esta tecnología se conoce como compilación Just-In-Time (JIT) y se utiliza para aumentar la velocidad de ejecución de la aplicación JavaScript. A continuación, la aplicación real se ejecuta sobre la base del código de máquina compilado. El motor V8 realiza otras optimizaciones además de la compilación JIT. Entre otras cosas, estas incluyen una mejor recolección de basura y una mejora en el contexto de acceso a las propiedades de los objetos.

Para todas las optimizaciones que realiza el motor JavaScript, debes tener en cuenta que el código fuente se lee al iniciar el proceso, por lo que los cambios en los archivos no tienen efecto en la aplicación en ejecución. Para que los cambios surtan efecto, debes salir y reiniciar tu aplicación para que se lean nuevamente los archivos de código fuente personalizados.

Modelo de memoria

El objetivo del desarrollo del motor V8 era conseguir la mayor velocidad posible en la ejecución del código fuente de JavaScript. Por este motivo, también se ha optimizado el modelo de memoria. En el motor V8 se utilizan apuntadores etiquetados, que son referencias en memoria que se marcan como tales de una forma especial. Todos los objetos están alineados a 4 bytes, lo que significa que hay 2 bits disponibles para identificar apuntadores. Un apuntador siempre termina en 01 en el modelo de memoria del motor V8, mientras que un valor entero normal termina en 0. Esta medida permite distinguir muy rápidamente los valores enteros de las referencias en memoria, lo que proporciona una ventaja de rendimiento extremadamente significativa.

Las representaciones de objetos del motor V8 en memoria constan cada una de tres palabras de datos. La primera palabra de datos consiste en una referencia a la clase oculta del objeto, sobre la que aprenderás más en documentos posteriores. La segunda palabra de datos es un apuntador a los atributos, es decir, las propiedades del objeto. Por último, la tercera palabra de datos hace referencia a los elementos del objeto.
Estas son las propiedades con una llave numérica. Esta estructura respalda el funcionamiento del motor de JavaScript y está optimizada de tal manera que se puede acceder a los elementos de la memoria muy rápidamente, de modo que se produce un tiempo de espera mínimo al buscar objetos.

Acceso a las propiedades

Como probablemente sepas, JavaScript no conoce clases; el modelo de objetos de JavaScript se basa en prototipos. En los lenguajes basados en clases, como Java o PHP, las clases representan el plano de los objetos. Estas clases no se pueden cambiar en tiempo de ejecución. Los prototipos en JavaScript, por otro lado, son dinámicos, lo que significa que se pueden agregar y eliminar propiedades y métodos en tiempo de ejecución. Al igual que con todos los demás lenguajes que implementan el paradigma de programación orientada a objetos, los objetos se representan por sus propiedades y métodos, donde las propiedades representan el estado de un objeto y los métodos se utilizan para interactuar con el objeto.

En una aplicación, normalmente se accede a las propiedades de los diversos objetos con mucha frecuencia. Además, los métodos en JavaScript también son propiedades de los objetos que se almacenan con una función. En JavaScript, se trabaja casi exclusivamente con propiedades y métodos, por lo que el acceso a ellos debe ser muy rápido.

Prototipos en JavaScript

JavaScript se diferencia de lenguajes como C++, Java o PHP en que no adopta un enfoque basado en clases, sino que se basa en prototipos, como el lenguaje Self. En JavaScript, cada objeto normalmente tiene una propiedad prototipo y, por lo tanto, un prototipo. En JavaScript, como en otros lenguajes, se pueden crear objetos. Sin embargo, para ello no se utilizan clases junto con el operador new, sino que se pueden crear nuevos objetos de varias formas diferentes.

Entre otras cosas, se pueden utilizar funciones constructoras o el método Object.create. Estos métodos tienen en común que se crea un objeto y se le asigna el prototipo. El prototipo es un objeto del que otro objeto hereda sus propiedades. Otra característica de los prototipos es que se pueden modificar en tiempo de ejecución de la aplicación, lo que permite añadir nuevas propiedades y métodos. Mediante el uso de prototipos, se puede crear una jerarquía de herencia en JavaScript.

Normalmente, el acceso a las propiedades en un motor de JavaScript se realiza a través de un directorio en la memoria. Por lo tanto, si accedemos a una propiedad, se busca en este directorio la sección de memoria de la propiedad respectiva y, a continuación, se puede acceder al valor. Ahora imaginemos una gran aplicación que mapea tu lógica de negocios en JavaScript en el lado del cliente y en la que una gran cantidad de objetos se almacenan en paralelo en la memoria, comunicándose constantemente entre sí. Este método de acceso a las propiedades se convertiría rápidamente en un problema.

Los desarrolladores del motor V8 han reconocido esta vulnerabilidad y han desarrollado una solución para ella: las clases ocultas. El verdadero problema con JavaScript es que la

estructura de los objetos solo se conoce en tiempo de ejecución y no durante el proceso de compilación porque tal proceso no existe con JavaScript. Esto se complica aún más por el hecho de que no hay un solo prototipo en la estructura de los objetos, sino que pueden existir en una cadena. En los lenguajes clásicos, la estructura de los objetos no cambia en tiempo de ejecución de la aplicación; las propiedades de los objetos siempre se encuentran en el mismo lugar, lo que acelera significativamente el acceso a ellos.

Una clase oculta no es más que una descripción en la que se pueden encontrar las propiedades individuales de un objeto en la memoria. Para ello, se asigna una clase oculta a cada objeto. Esto contiene el desplazamiento a la sección de memoria dentro del objeto donde se almacena la propiedad respectiva. Tan pronto como se accede a una propiedad de un objeto, se crea una clase oculta para esa propiedad y se reutiliza para cada acceso posterior. Por lo tanto, para un objeto, existe potencialmente una clase oculta separada para cada propiedad.
En el listado 1, puedes ver un ejemplo que ilustra cómo funcionan las clases ocultas.

Listado 1: Acceso a propiedades en una clase.

```
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}
const johnDoe = new Person("John", "Doe");
```

En el ejemplo, se crea una nueva función constructora para el grupo de objetos Person. Este constructor tiene dos parámetros: el nombre y el apellido de la persona. Estos dos valores se deben almacenar en las propiedades firstname y lastname del objeto, respectivamente. Cuando se crea un nuevo objeto con este constructor utilizando el operador new, primero se crea una clase oculta inicial, la clase 0. Esta aún no contiene ningún apuntador a propiedades. Si se realiza la primera asignación, es decir, se establece el nombre, se crea una nueva clase oculta, la clase 1, basada en la clase 0. Esta ahora contiene una referencia a la sección de memoria de la propiedad firstname, relativa al comienzo del espacio de nombres del objeto. Además, se agrega una transición de clase a la clase 0, que indica que se debe usar la clase 1 en lugar de la clase 0 si se agrega la propiedad firstname. El mismo proceso tiene lugar cuando se realiza la segunda asignación para el apellido. Otra clase oculta, la clase 2, se crea en base a la clase 1, que luego contiene el desplazamiento para las propiedades firstname y lastname e inserta una transición que indica que se debe usar la clase 2 cuando se usa la propiedad lastname.

Si se agregan propiedades fuera del constructor, y esto se hace en un orden diferente, se crean nuevas clases ocultas en cada caso. La figura 1.3 aclara este proceso. Cuando se accede a las

propiedades de un objeto por primera vez, el uso de clases ocultas todavía no da como resultado una ventaja de velocidad. Sin embargo, todos los accesos posteriores a la propiedad del objeto ocurren mucho más rápido, porque el motor puede usar directamente la clase oculta del objeto y esta contiene la referencia a la sección de memoria de la propiedad.

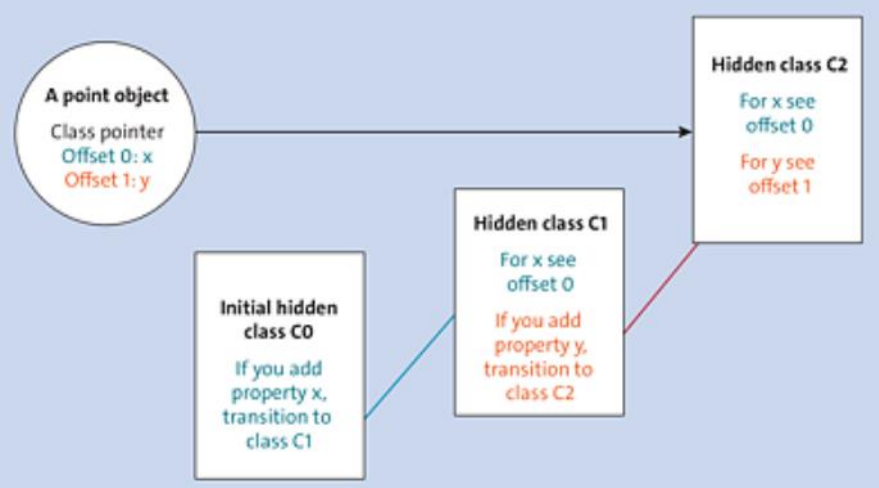


Figura 1.3 Clases ocultas en el motor V8 (https://v8.dev/).

Generación de código de máquina

Como ya sabes, el motor V8 no interpreta directamente el código fuente de la aplicación JavaScript, sino que realiza una compilación JIT en código de máquina nativo para aumentar la velocidad de ejecución. No se realizan optimizaciones en el código fuente durante esta compilación. De esta forma, el código fuente escrito por el desarrollador se convierte uno a uno. Además de este compilador JIT, el motor V8 cuenta con otro compilador que es capaz de optimizar el código máquina. Para decidir qué fragmentos de código optimizar, el motor mantiene estadísticas internas sobre el número de llamadas a funciones y el tiempo de ejecución de cada función. En función de estos datos, se toma la decisión sobre si es necesario optimizar el código máquina de una función.

Ahora probablemente te estarás preguntando por qué no se compila todo el código fuente de la aplicación con el segundo compilador, mucho mejor. Hay una razón muy simple para esto: un compilador que no realiza optimizaciones es mucho más rápido. Debido a que el código fuente se compila JIT, este proceso es muy crítico en términos de tiempo porque cualquier tiempo de espera causado por un proceso de compilación que lleva demasiado tiempo puede tener un impacto directo en el usuario. Por lo tanto, solo se optimizan las secciones de código que justifican este esfuerzo adicional.

Esta optimización del código de máquina tiene un efecto particularmente positivo en aplicaciones más grandes y de mayor duración y en aquellas en las que las funciones se invocan con más frecuencia que una sola vez.

Otra optimización que realiza el motor V8 está relacionada con las clases ocultas y el almacenamiento en caché interno que ya se describió anteriormente. Después de que se inicia la aplicación y se genera el código de máquina, el motor V8 busca la clase oculta asociada cada vez que se accede a una propiedad. Como optimización adicional, el motor supone que los objetos utilizados en este punto tendrán la misma clase oculta en el futuro, por lo que modifica el código de máquina en consecuencia. La próxima vez que se recorra la sección de código, se podrá acceder directamente a la propiedad sin necesidad de buscar primero la clase oculta asociada. Si el objeto utilizado no tiene la misma clase oculta, el motor lo detecta, elimina el código de máquina generado anteriormente y lo reemplaza con la versión corregida.

Este enfoque presenta un problema crítico: imagina que tienes una sección de código en la que siempre se utilizan de forma alternada dos objetos diferentes con clases ocultas diferentes.

En ese caso, la optimización con la predicción de la clase oculta nunca surtiría efecto en la siguiente ejecución. En este caso, se utilizan varios fragmentos de código, que no se pueden utilizar para encontrar la sección de memoria de una propiedad tan rápidamente como con una sola clase oculta, pero el código en este caso es mucho más rápido que sin la optimización porque normalmente es posible seleccionar entre un conjunto muy pequeño de clases ocultas. La generación de código de máquina y las clases ocultas en combinación con los mecanismos de almacenamiento en caché crea posibilidades que son familiares en los lenguajes basados en clases.

Recolección de basura

Las optimizaciones descritas hasta ahora afectan principalmente a la velocidad de una aplicación. Otra característica muy importante es el recolector de basura del motor V8. La recolección de basura se refiere al proceso de limpieza del área de memoria de la aplicación en la memoria principal.
Los elementos que ya no se utilizan se eliminan de la memoria para que el espacio liberado vuelva a estar disponible para la aplicación.

Si te preguntas por qué necesitas un recolector de basura en JavaScript, la respuesta es bastante simple: originalmente, JavaScript estaba destinado a pequeñas tareas en páginas web.

Estas páginas web, y por lo tanto el JavaScript en esta página, tenían una vida útil bastante corta hasta que la página se volvía a cargar, vaciando por completo la memoria que contenía los objetos JavaScript. Cuanto más JavaScript se ejecuta en una página y más complejas se vuelven las tareas a realizar, mayor es el riesgo de que la memoria se llene con objetos que ya no son necesarios. Si ahora supones que tienes una aplicación en Node.js que debe ejecutarse durante varios días, semanas o incluso meses sin reiniciar el proceso, el problema se vuelve claro.

El recolector de basura del motor V8 comprende una serie de características que le permiten realizar sus tareas de manera muy rápida y eficiente. Básicamente, cuando el recolector de basura está en funcionamiento, el motor detiene la ejecución de la aplicación por completo y la reanuda tan pronto como finaliza la ejecución. Estas pausas de la aplicación están en el rango de milisegundos de un solo dígito para que el usuario normalmente no sienta ningún efecto negativo debido al recolector de basura. Para mantener la interrupción del recolector de basura lo más breve posible, no se limpia toda la memoria, sino solo partes de ella. Además, el motor V8 sabe en todo momento en qué parte de la memoria se encuentran los objetos y apuntadores.

El motor V8 divide la memoria disponible en dos áreas: un área para almacenar objetos y otra área para mantener la información sobre las clases ocultas y el código de máquina ejecutable. El proceso de recolección de basura es relativamente simple. Cuando se ejecuta una aplicación, los objetos y apuntadores se crean en el área de corta duración de la memoria del motor V8. Si esta área de memoria está llena, se limpia. Los objetos que ya no se utilizan se eliminan y los objetos que aún se necesitan se mueven al área de larga duración. Durante este desplazamiento, se desplaza el propio objeto y se corrigen los apuntadores a la ubicación de memoria del objeto. La partición de las áreas de memoria hace necesario distintos tipos de recolección de basura.

La variante más rápida está representada por el recolector de basura, que es muy rápido y eficiente y se ocupa sólo del área de corta duración. Existen dos algoritmos de recolección de basura diferentes para la sección de memoria de larga duración, ambos basados en marcado y barrido (mark-and-sweep). Se busca en toda la memoria y se marcan y eliminan los elementos que ya no se necesitan. El verdadero problema con este algoritmo es que crea huecos en la memoria, lo que causa problemas durante un tiempo de ejecución más largo de una aplicación. Por esta razón, existe un segundo algoritmo que también busca en los elementos de la memoria aquellos que ya no se necesitan, los marca y los elimina.

La diferencia más importante entre los dos es que el segundo algoritmo desfragmenta la memoria; es decir, reorganiza los objetos restantes en la memoria para que después, la memoria tenga la menor cantidad de huecos posible.

Esta desfragmentación sólo puede ocurrir porque V8 conoce todos los objetos y apuntadores. A pesar de todos sus beneficios, el proceso de recolección de basura también tiene un inconveniente: lleva tiempo. Lo más rápido que puede ejecutarse la recolección de basura es de aproximadamente 2 ms. A continuación, se realiza el proceso de marcado y barrido sin optimizaciones a 50 ms y, por último, el proceso de marcado y barrido con desfragmentación con un promedio de 100 ms.

En las siguientes secciones, aprenderás más sobre los demás elementos utilizados en la plataforma Node.js además del motor V8.

Bibliotecas en torno al motor

El motor de JavaScript por sí solo no constituye una plataforma todavía. Para que Node.js gestione todos los requisitos, como el manejo de eventos, la E/S o las funciones de soporte, como la resolución o el cifrado del sistema de nombres de dominio (DNS, Domain Name System), se requiere una funcionalidad adicional. Esto se implementa con la ayuda de bibliotecas adicionales. Para muchas de las tareas con las que tiene que lidiar una plataforma como Node.js, ya existen soluciones listas y establecidas. Por este motivo, Dahl decidió construir la plataforma Node.js sobre un conjunto de bibliotecas externas y llenar los vacíos que consideraba que no estaban adecuadamente cubiertos por ninguna solución existente con sus propias implementaciones. La ventaja de esta estrategia es que no es necesario reinventar las soluciones para los problemas estándar; puedes recurrir a bibliotecas probadas y comprobadas.

Un ejemplo destacado que también se basa en esta estrategia es el sistema operativo Unix. En este contexto, los desarrolladores deben ceñirse al siguiente principio: centrarse únicamente en el problema real, resolverlo lo mejor posible y utilizar las bibliotecas existentes para todo lo demás. La mayoría de los programas de línea de comandos en el área Unix implementan esta filosofía.

Una vez que se ha establecido una solución, se puede utilizar en otras aplicaciones para problemas similares. Esto, a su vez, tiene la ventaja de que las mejoras en el algoritmo solo deben realizarse en un punto central. Lo mismo se aplica a las correcciones de errores. Si ocurre un error en la resolución DNS, se corrige una vez y la solución funciona en todos los lugares donde se utiliza la biblioteca. Pero eso también lleva a la otra cara de la moneda: las bibliotecas sobre las que se construye la plataforma deben existir. Node.js resuelve este problema al estar construido sobre solo un pequeño conjunto de bibliotecas que debe proporcionar el sistema operativo. Pero estas dependencias consisten más bien en funciones básicas como la biblioteca de tiempo de ejecución GNU Compiler Collection (GCC) o la

biblioteca C estándar. Las dependencias restantes, como zlib o http_parser, están incluidas en el código fuente.

Ciclo de eventos

El JavaScript del lado del cliente contiene muchos elementos de una arquitectura basada en eventos. La mayoría de las interacciones del usuario provocan eventos a los que se responde con llamadas de función adecuadas. Al utilizar diversas características, como funciones de primera clase y funciones anónimas en JavaScript, puedes implementar aplicaciones completas basadas en una arquitectura basada en eventos.

El término *basado en eventos* significa que los objetos no se comunican directamente entre sí a través de llamadas de función; en cambio, se utilizan eventos para esta comunicación. Por lo tanto, la programación basada en eventos se utiliza principalmente para controlar el flujo del programa. A diferencia del enfoque clásico, donde el código fuente se ejecuta de forma lineal, aquí las funciones se ejecutan cuando ocurren ciertos eventos. Un pequeño ejemplo en el listado 2 ilustra este enfoque.

Listado 2. Desarrollo basado en eventos en Node.js.

```
myObj.on('myEvent', (data) => {
    console.log(data);
});
myObj.emit('myEvent', 'Hello World');
```

Puedes utilizar el método on de un objeto que derive de events.EventEmitter, un componente de la plataforma Node.js, para definir qué función deseas utilizar para responder a cada evento. Este patrón se conoce como patrón de publicación-suscripción. De esta forma, los objetos pueden registrarse en un emisor de eventos y luego recibir una notificación cuando se produce el evento.

El primer argumento del método on es el nombre del evento en forma de cadena a la que se debe responder. El segundo argumento consiste en una función de devolución de llamada que se implementa como una función de flecha en este caso, que se ejecuta una vez que se produce el evento. Por lo tanto, la llamada de función del método on no hace nada más que registrar la función de devolución de llamada la primera vez que se ejecuta. Más adelante en el script, se llama al método emit en myObj. Esto garantiza que se ejecuten todas las funciones de devolución de llamada registradas por el método on.

Lo que funciona en este ejemplo con un objeto personalizado lo utiliza Node.js para realizar una variedad de tareas asíncronas.

Sin embargo, las funciones de devolución de llamada no se ejecutan en paralelo, sino de forma secuencial. El enfoque de un solo subproceso de Node.js crea el problema de que solo se puede ejecutar una operación a la vez. Las operaciones de lectura o escritura que consumen mucho tiempo, en particular, bloquearían toda la ejecución de la aplicación. Por este motivo, todas las operaciones de lectura y escritura se subcontratan mediante el ciclo de eventos. Esto permite que el código de la aplicación aproveche el hilo disponible.

Una vez que se realiza una solicitud a un recurso externo en el código fuente, se pasa al ciclo de eventos. Se registra una devolución de llamada para la solicitud que reenvía la solicitud al sistema operativo; Node.js luego recupera el control y puede continuar ejecutando la aplicación. Una vez que se completa la operación externa, el resultado se devuelve al ciclo de eventos. Se produce un evento y el ciclo de eventos garantiza que se ejecuten las funciones de devolución de llamada asociadas. La figura 4 muestra cómo funciona el ciclo de eventos.

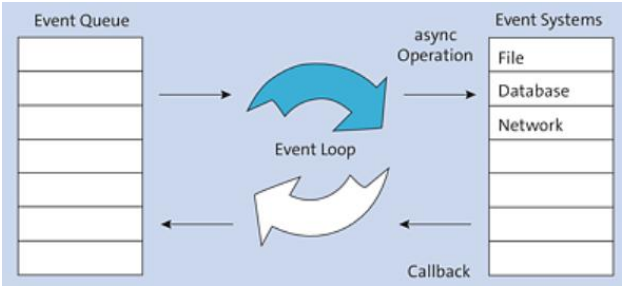


Figura 4: Ciclo de eventos.

El ciclo de eventos original utilizado en Node.js se basa en libev, una biblioteca escrita en C que representa un alto rendimiento y una amplia gama de funciones. libev se basa en los enfoques de libevent pero tiene un índice de rendimiento superior, como lo demuestran varios puntos de referencia.

Incluso una versión mejorada de libevent - libevent2 - no iguala el rendimiento de libev. Sin embargo, por razones de compatibilidad, el ciclo de eventos se abstrajo para lograr una mejor portabilidad a otras plataformas.

Entrada y salida

El ciclo de eventos solo en combinación con el motor V8 permite la ejecución de JavaScript, pero aún no hay posibilidad de interactuar con el sistema operativo directamente en forma de operaciones de lectura o escritura en el sistema de archivos. En la implementación de aplicaciones del lado del servidor, los accesos al sistema de archivos juegan un papel importante. Por ejemplo, la configuración de una aplicación a menudo se externaliza a un archivo de configuración separado. Esta configuración debe ser leída por la aplicación desde

el sistema de archivos. Sin embargo, las plantillas, que se rellenan dinámicamente con valores y luego se envían al cliente, también suelen estar disponibles como archivos separados.
Tanto la lectura como la escritura de información en archivos suelen ser un requisito para una aplicación JavaScript del lado del servidor.

El registro dentro de una aplicación es otra área común de uso de los accesos de escritura al sistema de archivos. Aquí, se registran diferentes tipos de eventos dentro de la aplicación en un archivo de registro. Dependiendo de dónde se ejecute la aplicación, solo se escriben errores fatales, advertencias o incluso información de tiempo de ejecución. Los accesos de escritura también se utilizan para conservar información. Durante el tiempo de ejecución de una aplicación, generalmente a través de la interacción de los usuarios y varios cálculos, se genera información que debe capturarse para su posterior reutilización.

Node.js utiliza la biblioteca C libeio para estas tareas. Garantiza que las operaciones de escritura y lectura puedan realizarse de forma asíncrona y, por lo tanto, la biblioteca trabaja muy de cerca con el ciclo de eventos. Sin embargo, las características de libeio no se limitan al acceso de escritura y lectura al sistema de archivos; más bien, ofrecen muchas más posibilidades para interactuar con el sistema de archivos.

Estas opciones van desde la lectura de información de archivos (por ejemplo, tamaño, fecha de creación o fecha de acceso) hasta la gestión de directorios (es decir, su creación o eliminación) y la modificación de los derechos de acceso. De manera similar al ciclo de eventos, durante el transcurso de su desarrollo, esta biblioteca se separó de la aplicación real mediante una capa de abstracción.

Para acceder al sistema de archivos, Node.js proporciona su propio módulo, el módulo del sistema de archivos. Este módulo le permite acceder a las interfaces de libeio y, por lo tanto, representa un contenedor muy liviano alrededor de libeio.

libuv

Las dos bibliotecas que has visto hasta ahora están relacionadas con Linux. Sin embargo, Node.js se suponía que iba a convertirse en una plataforma independiente del sistema operativo. Por este motivo, la biblioteca libuv se introdujo en la versión 0.6 de Node.js. Esta biblioteca se utiliza principalmente para abstraer las diferencias entre los distintos sistemas operativos.

En consecuencia, el uso de libuv hace posible que Node.js funcione también en sistemas Windows. La estructura sin libuv, tal como se utilizaba en Node.js hasta la versión 0.6, se ve así: el núcleo es el motor V8; se complementa con libev y libeio con el ciclo de eventos y el

acceso asíncrono al sistema de archivos. Con libuv, estas dos bibliotecas ya no están integradas directamente en la plataforma, sino que se abstraen.

Para que Node.js funcione en Windows, es necesario proporcionar los componentes básicos para las plataformas Windows.
El motor V8 no es un problema aquí; ha estado funcionando en el navegador Chrome durante muchos años en Windows sin ningún problema. Sin embargo, la cosa se complica con el ciclo de eventos y las operaciones asíncronas del sistema de archivos.

Algunos componentes de libev necesitarían ser reescritos al ejecutarse en Windows. Además, libev se basa en implementaciones nativas del sistema operativo de la función select, pero, en Windows, está disponible una variante optimizada para el sistema operativo en forma de IOCP. Para evitar tener que crear diferentes versiones de Node.js para los diferentes sistemas operativos, los desarrolladores decidieron incluir una capa de abstracción con libuv que permite utilizar libev para sistemas Linux e IOCP para Windows.

Con libuv, se han adaptado algunos conceptos básicos de Node.js. Por ejemplo, ya no hablamos de eventos, sino de operaciones. Una operación se pasa al componente libuv; dentro de libuv, la operación se pasa a la infraestructura subyacente, es decir, libev o IOCP, respectivamente. De este modo, la interfaz de Node.js permanece inalterada independientemente del sistema operativo utilizado.

libuv es responsable de gestionar todas las operaciones de E/S asíncronas. Esto significa que todo acceso al sistema de archivos ya sea de lectura o escritura, se realiza a través de las interfaces de libuv. Para este propósito, libuv proporciona las funciones uv_fs_functions, así como temporizadores, es decir, llamadas dependientes del tiempo, y conexiones asíncronas del Protocolo de Control de Transmisión (TCP) y el Protocolo de Datagramas de Usuario (UDP) que se ejecutan a través de libuv.

Además de estas funcionalidades básicas, libuv administra características complejas como la creación y el lanzamiento de procesos secundarios y la programación de grupos de subprocesos, una abstracción que permite que las tareas se completen en subprocesos separados y que las devoluciones de llamadas se vinculen a ellos.
El uso de una capa de abstracción como libuv es un componente importante para la adopción más amplia de Node.js y hace que la plataforma sea un poco menos dependiente del sistema.

Sistema de nombres de dominio

Las raíces de Node.js se encuentran en Internet. Cuando estás en Internet, te encontrarás rápidamente con el problema de la resolución de nombres. En realidad, todos los servidores de Internet se identifican por su dirección IP. En el Protocolo de Internet versión 4 (IPv4), la

dirección es un número de 32 bits representado en cuatro bloques de 8 bits cada uno. En IPv6, las direcciones tienen un tamaño de 128 bits y se dividen en ocho bloques de números hexadecimales. Rara vez quieres trabajar directamente con estas direcciones crípticas, especialmente si se agrega una asignación dinámica a través del protocolo de configuración dinámica de host (DHCP).

La solución a esto es el sistema de nombres de dominio (DNS). El DNS es un servicio para la resolución de nombres en la web que garantiza que los nombres de dominio se conviertan en direcciones IP. También existe la posibilidad de resolución inversa, donde una dirección IP se traduce en un nombre de dominio. Si quieres conectar un servicio web o leer una página web en tu aplicación Node.js, aquí también se utiliza el DNS.

Internamente, Node.js no se encarga de la resolución de nombres, sino que pasa las solicitudes correspondientes a la biblioteca C-Ares. Esto se aplica a todos los métodos del módulo dns, excepto a dns.lookup, que utiliza la función getaddrinfo del propio sistema operativo. Esta excepción se debe a que getaddrinfo es más constante en sus respuestas que la biblioteca C-Ares, que, por sí sola, es mucho más eficiente que getaddrinfo.

Criptografía

El componente criptográfico de la plataforma Node.js te ofrece varias opciones de cifrado para fines de desarrollo. Este componente se basa en OpenSSL. Esto significa que este software debe estar instalado en tu sistema si deseas cifrar datos. El módulo criptográfico te permite cifrar datos con diferentes algoritmos, así como crear firmas digitales dentro de tu aplicación. Todo el sistema se basa en llaves privadas y públicas.

La llave privada, como su nombre lo indica, es solo para ti y tu aplicación. La llave pública está disponible para tus socios de comunicación. Si se debe cifrar contenido, esto se hace con la llave pública. Los datos solo se pueden descifrar con tu llave privada. Lo mismo se aplica a la firma digital de los datos. En este caso, se utiliza tu llave privada para generar dicha firma. El destinatario de un mensaje puede utilizar la firma y su llave pública para determinar si el mensaje proviene de ti y no ha sido modificado.

Zlib

Al crear aplicaciones web, como desarrollador, debes tener en cuenta los recursos de tus usuarios y su propio entorno de servidor. Por ejemplo, el ancho de banda disponible o la memoria libre para los datos pueden ser una limitación. Para abordar estos casos, la plataforma Node.js contiene el componente zlib. Con su ayuda, puedes comprimir datos y descomprimirlos nuevamente cuando desees procesarlos. Para la compresión de datos,

puedes utilizar dos algoritmos, Deflate y Gzip. Node.js trata los datos que sirven como entrada para los algoritmos como flujos.

Node.js no implementa los algoritmos de compresión en sí, sino que utiliza el zlib establecido y transmite las solicitudes en cada caso. El módulo zlib de Node.js simplemente proporciona un contenedor liviano para los algoritmos subyacentes Gzip, Deflate/Inflate y Brotli y garantiza que los flujos de E/S se gestionen correctamente.

Analizador (parser) HTTP

Como plataforma para aplicaciones web, Node.js debe ser capaz de manejar no solo transmisiones, datos comprimidos y cifrado, sino también HTTP. Debido a que el análisis de HTTP es un procedimiento laborioso, el analizador HTTP que maneja esta tarea se ha subcontratado a un proyecto independiente y ahora está incluido en la plataforma Node.js.

Al igual que las otras bibliotecas externas, el analizador HTTP está escrito en C y sirve como una herramienta de alto rendimiento que lee tanto las solicitudes como las respuestas HTTP. Como desarrollador, esto significa que puedes usar el analizador HTTP para leer, por ejemplo, la información variada en el encabezado HTTP o el texto del mensaje en sí.

El objetivo principal del desarrollo de Node.js es proporcionar una plataforma de alto rendimiento para aplicaciones web. Para cumplir con este requisito, Node.js se basa en una estructura modular. Esto permite la inclusión de bibliotecas externas como la libuv descrita anteriormente o el analizador HTTP. El enfoque modular continúa a través de los módulos internos de la plataforma Node.js y se extiende a las extensiones que creas para tu propia aplicación.

A lo largo de este curso, aprenderás sobre las diferentes capacidades y tecnologías que ofrece la plataforma Node.js para desarrollar tus propias aplicaciones. Comenzaremos con una introducción al sistema de módulos de Node.js.

Resumen

Desde hace muchos años, Node.js ha sido una parte integral del desarrollo web. En este contexto, Node.js no solo se usa para crear aplicaciones de servidor, sino que también es la base de una amplia gama de herramientas, desde el paquete webpack hasta herramientas como Babel y el compilador para preprocesadores CSS.

El éxito de la plataforma se basa en varios conceptos muy simples. La plataforma se basa en una colección de bibliotecas establecidas, que juntas crean un entorno de trabajo muy flexible. A lo largo de los años, el núcleo de la plataforma siempre se ha mantenido compacto, ofreciendo solo un conjunto de funcionalidades básicas. Para todos los demás requisitos, puedes usar npm para integrar una amplia variedad de paquetes en tu aplicación.

Aunque Node.js ya se ha probado en la práctica durante varios años, es posible que aún escuches con frecuencia la siguiente pregunta: ¿Puedo usar Node.js de manera segura para mi aplicación? En las versiones anteriores a la 0.6, no se podía responder afirmativamente a esta pregunta con la conciencia tranquila, ya que las interfaces de la plataforma estaban sujetas a cambios frecuentes. Sin embargo, hoy en día Node.js es mucho más maduro.

Los desarrolladores mantienen estables las interfaces. La versión LTS se creó para su uso en empresas. Se trata de una versión de Node.js que cuenta con soporte de actualizaciones durante un total de 30 meses. Esto aumenta la fiabilidad de la plataforma y alivia la presión de las empresas de actualizar siempre a la última versión.

Un capítulo muy emocionante en la historia del desarrollo fue la separación de io.js, ya que el desarrollo de Node.js había perdido impulso y no se introdujeron innovaciones en la plataforma durante mucho tiempo. Este evento fue un punto de inflexión crucial para el desarrollo de Node.js. Se formó la Fundación Node.js y la responsabilidad del desarrollo se transfirió de un individuo a un grupo. Como resultado, se estandarizaron los ciclos de lanzamiento y las versiones, lo que indicaba confiabilidad y un desarrollo continuo para los usuarios de la plataforma.

Si decides profundizar en Node.js, estarás en buena compañía con numerosas empresas grandes y pequeñas en todo el mundo que ahora utilizan Node.js estratégicamente para el desarrollo de aplicaciones.