PERFORMANCE, DEBUGGING UND ASYNCHRONE ARCHITEKTUREN IN NODE.JS

HALLO, ICH BIN DER BASTI 👋

- Sebastian Springer
- React & Node.js
- München

AGENDA

- Event Loop und Asynchronität
- Kindprozesse und Threads
- Profiling und Debugging
- Streams

ASYNCHRONITÄT IN NODE.JS

EVENT LOOP

- Single-Threaded non blocking: Node.js ist single threaded, delegiert aber I/O-Aufgaben an das Betriebssystem.
- Phasenbasiertes Modell: Der Event Loop besteht aus mehreren Phasen, die nacheinander abgearbeitet werden.
- Pro Phase gibt es eine FIFO Queue
- Callbacks und asynchrone Verarbeitung: Node.js verwaltet mit dem Event Loop asynchrone Funktionen sehr effizient.
- Microtasks vs. Macrotasks: Promises (Microtasks) haben eine höhere Priorität als Makrotasks (z. B. setTimeout) und werden zwischen den Event-Loop-Phasen ausgeführt.
- I/O-Handling: Asynchrone I/O-Operationen (z. B. Datei- und Netzwerkzugriffe) werden so verwaltet, dass der Hauptthread nicht blockiert wird.

DIE PHASEN DES EVENT LOOPS

1. timers: Callbacks von setTimeout und setInterval
2. pending callbacks: Callbacks für einige System-Operationen z.B. TCP Errors
3. idle, prepare: for internal use only
4. poll: Überprüfen, ob es neue I/O-Events und wie lange gewartet wird, Events in der queue verarbeiten
5. check: callbacks von setImmediate
6. close callbacks: z.B. socket.on('close',...)

DIE PHASEN DES EVENT LOOPS

- In einem I/O Cycle wird set Immediate immer vor setTimeout ausgeführt
- Process.nextTick wird sofort in der aktuellen Phase gefeuert

MICROTASKS VS. MACROTASKS

- Microtasks sind kleiner und haben eine höhere Priorität
- Process.nextTick hat eine Sonderrolle und eine höhere Priorität innerhalb der Microtasks
- Microtasks: Promises, process.nextTick
- Macrotasks: setTimeout, setInterval, setImmediate, I/O, UI rendering

PROMISES VS. EVENTS

EVENTS

- Observer Pattern: basiert auf EventEmitter und Listenern
- Mehrfache Auslösung: Ein Event kann beliebig oft ausgelöst werden
- Callback basiert: Listener werden als Callbacks registriert
- Kontinuierlich: Listener werden bei jedem Event aufgerufen
- Anwendungsfall: wiederkehrende Ereignisse wie Datenströme und Interaktionen
- Ähnliche Priorität wie process.nextTick

PROMISES VS. EVENTS

PROMISES

- Promises repräsentieren einen einzelnen zukünftigen Wert
- Einmalige Auslösung: nur einmal resolved oder rejected
- Verkettung: Verkettung mit .then und .catch
- Status: drei mögliche Zustände: pending, fulfilled, rejected
- Anwendungsfall: einzelne asynchrone Operationen wie HTTP-Requests oder DB Queries
- Microtask im Event Loop

PROMISES VS. EVENTS

- Häufigkeit: Events können mehrfach ausgelöst werden. Promises nur einmal
- Behandlung: Evelts mit Listeners, Promises mit then/catch oder await
- Verwendugn: Events für wiederholende Ereignisse, Promises für einmalige Operationen
- Zustandsmanagement: Promises haben explizite Zustände, Events nicht
- Fehlerbehandlung: Promises mit catch, Events mit separaten Error Events

PROMISES IN NODE.JS

- fs: Promise-basierte Methoden für Dateioperationen wie Lesen, Schreiben und Bearbeiten
- dns: DNS lookup und resolve
- stream: pipeline, finished
- timers: setTimeout, setInterval, setImmediate
- crypto: nahezu alle Methoden arbeiten mit Promises
- util.promisify: wandelt eine Callback-basierte Funktion in eine Promise-basierte um

P-LIMIT

- Ermöglicht die Begrenzung der Anzahl gleichzeitig ausgeführter Promises
- Verhindert Überlastung durch zu viele parallele Anfragen
- Einfache Integration in bestehende Promise-basierte Anwendungen
- Hilft bei der Einhaltung von API-Rate limits
- Verbessert die Leistung bei ressourcenintensiven Operationen
- Einfache Lösung für Concurrency-Management

POTENZIELL BLOCKIERENDES AWAIT

- await pausiert die aktuelle async-Funktion, aber blockiert nicht den gesamten Event-Loop
- Wenn CPU-intensive Operationen im Main-Thread mit await versehen werden, kann dies die Event-Loop blockieren
- Lange await-Aufrufe in einem API-Endpunkt verzögern die Antwort
- Parallele Ausführung mehrerer await-Operationen mit Promise.all() kann die Blockierung reduzieren im Vergleich zu sequentiellen await-Aufrufen
- Das Verschieben von rechenintensiven Aufgaben in Worker-Threads oder Microservices kann helfen, Blockierungen durch await zu vermeiden

PARALLEL PROCESSING

SINGLE THREADED

- Node läuft auf einem einzigen Thread, was bedeutet, dass JavaScript-Code sequentiell in einem Prozess ausgeführt wird.
- Einfaches Programmiermodell - keine komplexen Synchronisationsprobleme wie Race Conditions oder Deadlocks.
- Ressourceneffizienz - weniger Arbeitsspeicherverbrauch als multi-threaded Systeme.
- CPU-intensive Aufgaben blockieren den Prozess und verlangsamen die Anwendung.
- Node selbst kann nicht automatisch alle CPU-Kerne nutzen, was zu Unterauslastung führen kann.
- Worker Threads und Child Processes können genutzt werden, um CPU-intensive Aufgaben auszulagern und Multi-Core-Systeme effektiver zu nutzen.

BLOCKING CODE

- Blockierender Code kann sinnvoll sein, weil er einfacher zu verstehen und zu debuggen ist.
- Bei geringer Last, kleinen Dateien oder beim Startup können synchrone Operationen in Ordnung sein.
- In Startup Phasen können synchrone Operationen sinnvoll sein, weil die Voraussetzungen erforderlich sind.
- Finden keine parallelen Abläufe statt, können synchrone Operationen auch sinnvoll sein.

PARALLEL PROCESSING

- Process-Modul: Separate Prozesse mit IPC für CPU-intensive Aufgaben.
- Cluster-Modul: Mehrere Node-Prozesse (Worker) teilen sich einen Server-Port und verteilen HTTP-Anfragen auf mehrere CPU-Kerne.
- Worker Threads: Leichtgewichtige Alternative zu Child Processes, teilen Memory und kommunizieren effizienter.
- Child Processes und Cluster starten separate V8-Engines, Worker Threads teilen sich eine V8-Engine.
- Kommunikation: Child Processes und Cluster nutzen Serialisierung, Worker Threads nutzen SharedArrayBuffer.
- I/O-gebundene Anwendungen: Child Process, CPU-intensive Berechnungen: Worker Threads.

CHILD PROCESS

- Erstellt neue Prozesse für rechenintensive Aufgaben.
- Methoden:
  - exec: Shell-Kommando ausführen
  - execFile: Datei direkt ausführen
  - fork: Neuen Node-Prozess erzeugen
  - spawn: Neuen Prozess erzeugen
- Kommunikation über Standard-Streams oder IPC-Kanal bei fork().
- Synchrone und asynchrone Varianten verfügbar.
- Systemnahe Befehle ausführen, andere Sprachen einbinden, Aufgaben verteilen.

FORK

- Startet eigene V8 Instanz
- Kommunikation über `child.on` und `child.send` bzw. `process.on` und `process.send`
- Übertragung nur von einfachen und serialisierbaren Daten
- Events: `error` - Fehler beim Start oder während der Ausführung, `exit` - Beendigung des Prozesses
- Optimierung: Child Process Pooling

CLUSTER

- Ein Hauptprozess, der mehrere Worker-Prozesse erzeugen und verwalten kann
- Prozesse laufen auf verschiedenen Kernen unabhängig voneinander
- Das Cluster-Modul übernimmt die automatische Lastverteilung
- Der Hauptprozess kann bei Absturz eines Workers einen neuen Prozess starten
- Die Kommunikation funktioniert über `on` und `send`

WORKER THREADS

- Keine eigene V8-Instanz - dadurch leichtgewichtiger als Kindprozesse
- Kommunikation über Nachrichten, ähnlich wie Kindprozesse
- Zugriff auf gemeinsam genutzten Speicher über `SharedArrayBuffer`
- Gut geeignet für CPU-intensive Operationen

SHARED MEMORY

- Shared Memory mit SharedArrayBuffer: Gemeinsame Nutzung von Speicher zwischen worker_threads ohne Kopieren.
- Atomare Operationen mit Atomics: Methoden wie Atomics.add() oder Atomics.wait() vermeiden Race Conditions.
- Direkter Zugriff auf binäre Daten mit TypedArray: Int32Array, Float64Array und andere Varianten teilen effizient große Datenmengen.
- Parallele Verarbeitung ohne Message Passing: Worker greifen direkt auf denselben Speicherbereich zu, was Overhead verringert.
- Effizient für Hochleistungsanwendungen: Ideal für numerische Simulationen, Machine Learning und Echtzeit-Datenverarbeitung.
- Richtige Synchronisation ist entscheidend: Atomics.notify() und Atomics.wait() helfen, Threads sicher zu synchronisieren.

ATOMICS

- Atomics stellt atomare Operationen für SharedArrayBuffer bereit.
- Sie ermöglichen sichere, synchronisierte Zugriffe zwischen worker_threads.
- Verhindert Race Conditions durch garantierte Konsistenz der Speicherzugriffe.
- Atomics.load(typedArray, index) – Sicheren Wert auslesen
- Atomics.wait(typedArray, index, expectedValue, timeout?) – Warten auf Änderung
- Atomics.notify(typedArray, index, count) – Andere Threads aufwecken
- Atomics.store(typedArray, index, value) – Sicheren Wert setzen

ATOMICS

1. Worker 1 wartet mit Atomics.wait(sharedArray, 0, 0), falls Wert 0 bleibt.
2. Worker 2 setzt den Wert mit Atomics.store() und ruft Atomics.notify() auf.
3. Worker 1 wird aufgeweckt und liest den neuen Wert mit Atomics.load().

FEHLERSUCHE UND PROFILING

DEBUGGER

- Schrittweise Untersuche des Quellcodes, setzen von Breakpoints, Inspizieren von Variablen, Verfolgen des Callstacks
- Integrierter Debugger auf Basis der V8-Engine. Unterstützt CLI-Debugging und grafische Tools
- Möglichkeit der schrittweisen Codeausführung (Step Over, Step Into, Step Out
- Logging als ergänzende Maßnahme

DEBUGGER MIT `inspect`

- Einfacher integrierter Debugger für schnelle Fehlranalyse, gut für minimalistische Setups oder Debugging über SSH
- Interaktiver Debugger auf der Konsole
- `node inspect index.js`
- Debugging über Kommandos wie c (continue), s (step in), repl
- Breakpoints setzen mit sb

DEBUGGING MIT DEN CHROME DEVTOOLS

- Debugger über das Chrome DevTools-Protokoll
- Debugging über externe Applikationen wie die Chrome DevTools oder eine IDE
- Verbindung über eine WebSocket-Verbindung
- Bietet im Gegensatz zu `inspect` eine grafische Oberfläche und erweiterte Features
- `--inspect`: startet das Debugging, aber das Script läuft sofort weiter
- `--inspect-brk`: Pausiert direkt in der ersten Zeile, bevor der Code ausgeführt wird

DEBUGGING MIT DER IDE

- VSCode
- WebStorm
- Entweder aus der IDE heraus starten oder mit `--inspect` starten und dann verbinden

DEBUGGING MIT TYPESCRIPT

- Applikation aufsezten und konfigurieren
- Source Map Support aktivieren (tsconfig.json > "sourceMap": true)
- Task + Launch Config erstellen

```
{
	"version": "2.0.0",
	"tasks": [
		{
			"type": "typescript",
			"tsconfig": "tsconfig.json",
			"option": "watch",
			"problemMatcher": [
				"$tsc-watch"
			],
			"group": "build",
			"label": "tsc build"
		}
	]
}
```

```
{
	"version": "0.2.0",
	"configurations": [
		{
			"type": "node",
			"request": "launch",
			"name": "Node TS debug",
			"skipFiles": [
				"<node_internals>/**"
			],
			"program": "${workspaceFolder}/02_node_perf/examples/03-debug-profile/02-typescript/src/inde
			"preLaunchTask": "tsc build",
			"outFiles": [
				"${workspaceFolder}/**/*.js"
			]
		}
	]
}
```

ZEITMESSUNG IN EINER APPLIKATION

- Performance-Optimierung: langsame Codeabschnitte identifizieren, Engpässe reduzieren und Gesamtleistung verbesser
- Effizienz bewerten: Verschiedene Implementierungen vergleichen
- Skalierbarkeit prüfen: Messen, wie sich die Applikation unter Last verhält
- Asynchrone Prozesse analysieren: Promises oder Callbacks messen

ZEITMESSUNG IN NODE

- `console.time` & `console.timeEnd` - einfache (aber eher ungenaue) Zeitmessung
- `performance.now()`: hochgenauer Zeitstempel seit dem Start des Prozesses
- `performance.mark` & `performance.measure`: hochgenaue Messung zwischen zwei Punkten
- `performance.timerify`: wrappt um eine Funktion für Zeitmessung
- PerformanceObserver kann Garbage Collection, Event Loop und andere Metriken messen

WEITERE METRIKEN MIT DEM PERFORMANCEOBSERVER

- dns - Misst DNS-Lookup-Zeiten für Netzwerkanfragen. Beispiel: Zeit von dns.lookup()
- function - Erfasst Laufzeit von überwachten Funktionen (z. B. perf_hooks.monitorEventLoopDelay()).
- gc (Garbage Collection) - Misst die Dauer und den Speicherverbrauch von GC-Läufen.
- http / http2 - Misst HTTP- und HTTP/2-Anfragen, inklusive TTFB (Time to First Byte).
- mark - Setzt eine Markierung an einem bestimmten Punkt im Code.
- measure - Misst die Zeit zwischen zwei mark()-Punkten oder vom Start der Anwendung bis zu einem Punkt.
- net - Misst Netzwerkereignisse wie Verbindungen und Verbindungszeiten.
- resource - Erfasst Ressourcenladezeiten (z. B. für externe API-Aufrufe oder Dateien).

CPU PROFILING

- Erfassung der CPU-Auslastung: Analysiert, welche Funktionen die meiste CPU-Zeit beanspruchen.
- Identifikation von Performance-Engpässen: Zeigt blockierende oder ineffiziente Funktionen in der Anwendung.
- Messung der Funktionslaufzeiten: Bestimmt, wie lange einzelne Funktionen ausgeführt werden.
- Optimierung der Event Loop: Hilft, Verzögerungen und unnötige Blockierungen im Event Loop zu reduzieren.
- Generierung von CPU-Profilen: Erstellt detaillierte Logs, die mit Chrome DevTools oder anderen Tools analysiert werden können.
- Weitere Tools: clinic.js, 0x

CPU PROFILING

1. Profile aufzeichnen: `node --prof index.js`
2. Interaktion mit dem Prozess
3. Profile analysieren: `node --prof-process *.log`

CPU PROFILING-VISUALISIERUNG

1. `npm install speedscope -D`
2. `node --prof --logfile=cpu-profile.log index.js`
3. `node --prof-process --preprocess cpu-profile.log > cpu-profile.json`
4. `speedscope cpu-profile.json`

CPU USAGE MESSEN

- `process.cpuUsage()` misst die CPU-Zeit für einen Block user (JavaScript) und system (OS)
- `os.loadavg` durchschnittliche CPU-Last des Systems über 1, 5 und 15 Minuten (nicht Windows)
- `os.cpus()` gibt Infos über das CPU-Timing aus, dient als Berechnungsgrundlage
- `pidusage` NPM-Paket mit Informationen zu CPU- und RAM-Nutzugn eines Node Prozesses

HEAP SNAPSHOTS

- Speicheranalyse-Tool: Momentaufnahme der Speicherbelegung von Objekten.
- Objektverweise und Retaining Paths: Zeigt, welche Objekte im Speicher gehalten werden und warum sie nicht freigegeben werden.
- Erzeugung mit v8-Modul oder --inspect: Kann über das v8-Modul in Node.js oder Chrome DevTools erstellt werden.
- Visualisierung in Chrome DevTools: Die .heapsnapshot-Datei kann in den Chrome DevTools unter dem “Memory”-Tab analysiert werden.
- Performance-Optimierung: Hilft, übermäßige Speicherzuweisungen und unnötige Objekterstellungen zu identifizieren.
- Speicherlecks aufspüren: Ermöglicht das Identifizieren von Memory Leaks, indem man den Speicherverbrauch über mehrere Snapshots vergleicht.

HEAP SNAPSHOTS

- `npm add v8` v8 Modul installieren
- `v8.getHeapSnapshot()` Snapshot aufzeichnen
- Snapshot in den Chrome DevTools importieren

STREAMS

STREAMS IN NODE

- Warum Streams?
  - Verarbeitung großer Datenmengen ohne vollständiges Laden in den Speicher
  - Effiziente I/O-Operationen durch schrittweise Verarbeitung
- Vergleich mit klassischen I/O-Operationen
  - Ohne Streams: Datei wird komplett in den Speicher geladen → hoher RAM-Verbrauch
  - Mit Streams: Daten werden in kleinen Chunks verarbeitet → besser skalierbar
- Streams in Node.js
  - Kernbestandteil der Plattform
  - Grundlage für viele Module (fs, http, net, zlib)

ARTEN VON STREAMS

- Readable Streams (Lesend)
  - Datenquelle, aus der gelesen werden kann (z. B. fs.createReadStream)
  - Events: data, end, error, readable
- Writable Streams (Schreibend)
  - Ziel für Daten (z. B. fs.createWriteStream)
  - Events: drain, finish, error
- Duplex Streams (Lesen & Schreiben)
  - Kombination aus Readable & Writable (z. B. net.Socket)
  - Ermöglicht bidirektionale Datenübertragung
- Transform Streams (Datenverarbeitung)
  - Spezialform eines Duplex Streams mit Datenmanipulation
  - Beispiel: zlib.createGzip() zur Komprimierung von Daten

ERSTELLUNG VON STREAMS

- Streams aus vorhandenen Modulen nutzen
  - fs.createReadStream('file.txt') – Datei zeilenweise lesen
  - fs.createWriteStream('output.txt') – Datei schrittweise schreiben
- Eigene Readable & Writable Streams erstellen
  - stream.Readable durch Implementierung der _read()-Methode
  - stream.Writable durch Implementierung der _write()-Methode
- Transform Streams für Datenmanipulation
  - Ableitung von stream.Transform
  - Implementierung der _transform()-Methode
  - Beispiel: Daten in Großbuchstaben umwandeln
- Optionen für Stream-Erstellung
  - highWaterMark zur Steuerung der Puffergröße
  - objectMode: true für die Verarbeitung von Nicht-Buffer-Daten

VERBINDUNG VON STREAMS

- Streams kombinieren mit .pipe()
  - Einfachste Methode zur Weiterleitung von Daten
  - Beispiel: Datei einlesen, transformieren und schreiben
  - readableStream.pipe(transformStream).pipe(writableStream);
- Fehlerbehandlung in Pipelines
  - .pipe() leitet Fehler nicht automatisch weiter
  - Besser: pipeline() aus dem stream-Modul
- Mehrere Streams verketten (Chaining)
- Best Practices
  - pipeline() bevorzugen, da es automatisch Fehler behandelt und korrekt aufräumt
  - Puffergrößen (highWaterMark) beachten, um Back Pressure zu vermeiden
  - Fehler immer abfangen, z. B. mit .on('error', callback)

OBJECT MODE

- Was ist der Object Mode?
  - Standardmäßig arbeiten Streams mit Binärdaten (Buffer) oder Strings
  - Object Mode erlaubt beliebige JavaScript-Objekte (z. B. JSON, Arrays, Zahlen)
- Warum Object Mode?
  - Verarbeitung strukturierter Daten ohne manuelles Serialisieren
  - Ideal für Streams mit JSON-Daten oder Event-Handling
- Object Mode aktivieren
  - Beim Erstellen eines Streams mit { objectMode: true }
- Best Practices
  - Nur nutzen, wenn Objekte notwendig sind → höhere Speicherlast
  - Konsistenz sicherstellen: Kein Mischen von Buffer- und Objekt-Daten
  - Back Pressure beachten, da Objekte größer als Buffer sein können

STREAM MODES

- Flowing Mode (Daten fließen automatisch)
  - Stream gibt Daten direkt aus (data-Event)
  - Wird automatisch aktiv, wenn ein data-Listener vorhanden ist
- Paused Mode (Manuelles Lesen der Daten)
  - Daten werden nur mit .read() abgerufen
  - Wechsel in den Paused Mode mit .pause()
- Umschalten zwischen den Modi
  - Flowing → Paused: readableStream.pause()
  - Paused → Flowing: readableStream.resume() oder data-Listener hinzufügen
- Best Practices
  - Flowing Mode für kontinuierliche Verarbeitung nutzen
  - Paused Mode für mehr Kontrolle, z. B. wenn Back Pressure auftritt
  - Nicht beide Modi mischen, um unerwartetes Verhalten zu vermeiden

BACK PRESSURE

- Was ist Back Pressure?
  - Entsteht, wenn ein Readable Stream schneller liefert, als ein Writable Stream verarbeiten kann
  - Führt zu hohem Speicherverbrauch und möglichen Abstürzen
- Wie äußert sich Back Pressure?
  - highWaterMark-Limit wird überschritten
  - Schreib-Operationen (.write()) geben false zurück
  - Writable Stream pausiert, bis Daten verarbeitet wurden
- Lösungen
  - Puffergröße anpassen (highWaterMark)
  - drain-Event nutzen
  - pipeline statt pipe nutzen (Errorhandling)

WEB STREAMS API

GEMEINSAMKEITEN

- Beide ermöglichen stückweise Verarbeitung von Daten
- Unterstützen Readable & Writable Streams
- Ermöglichen effizientes Arbeiten mit großen Datenmengen

UNTERSCHIEDE

Eigenschaft	Node.js Streams	Web Streams API
Standard	Node.js-spezifisch (stream-Modul)	Offizieller Web-Standard (browserkompatibel)
Chunk-Typ	Buffer (Standard) oder objectMode	Uint8Array oder beliebige Objekte
Piping	.pipe() und pipeline()	.pipeThrough() und .pipeTo()
Back Pressure	Automatisch mit pipeline()	Automatisch integriert
Einsatzgebiet	Serverseitige Verarbeitung (Dateisystem, Netzwerke)	Browser-Streaming (z. B. fetch().body)