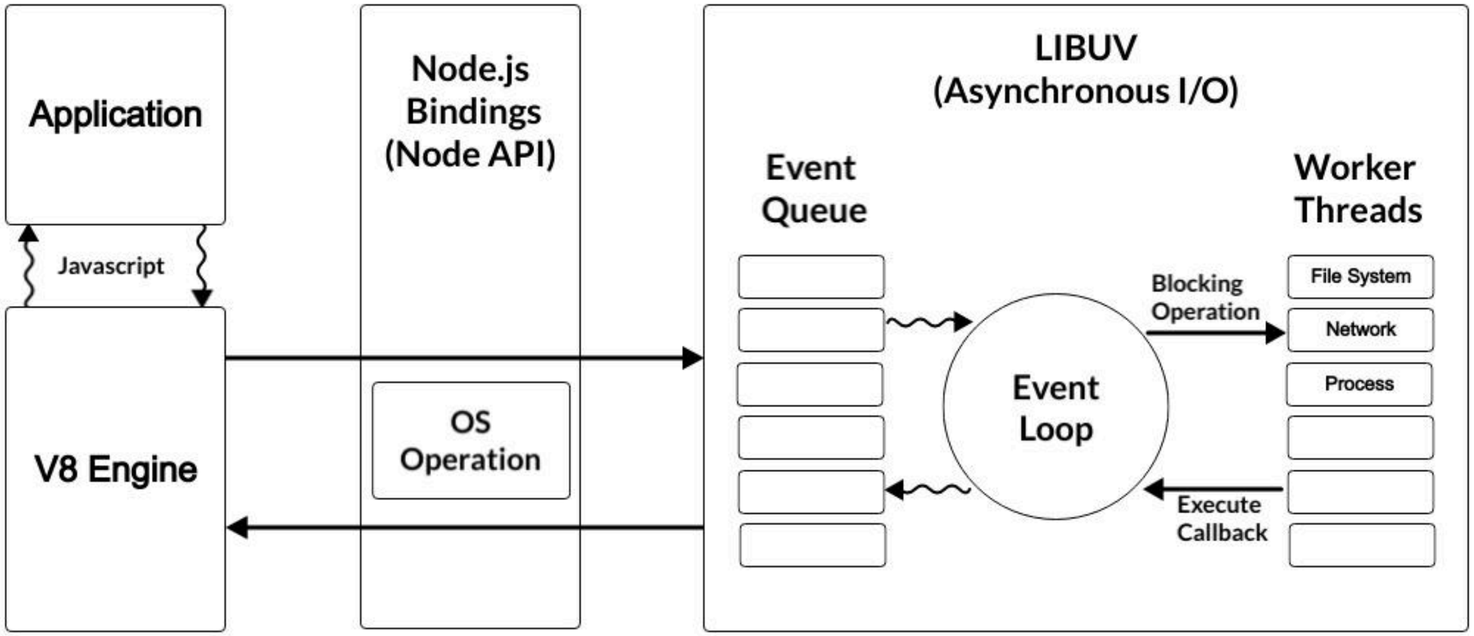
Node.js 멀티스레딩

23.09.17 전애지

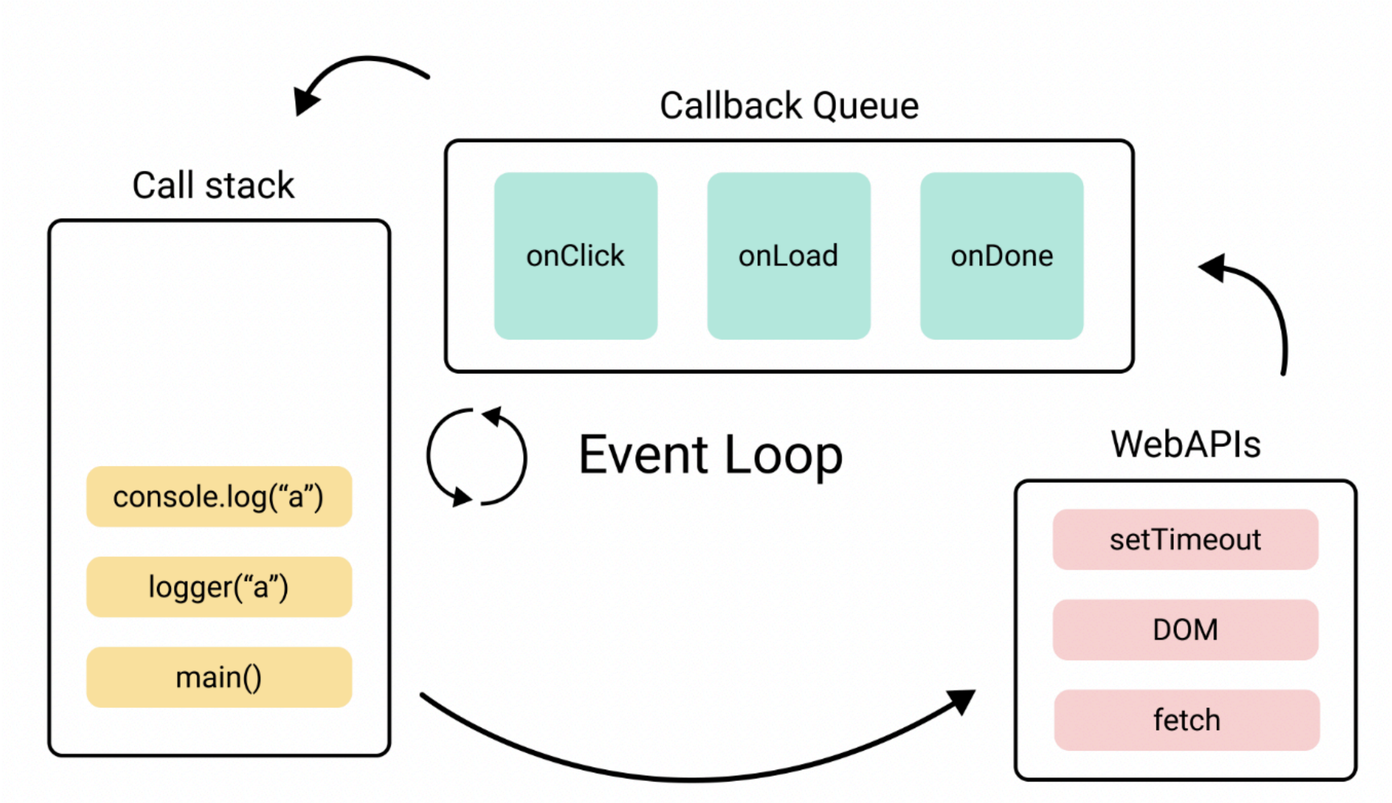
Node.js

- js 실행하기 위한 런타임 환경
- 메인 프로그램은 싱글 스레드로 동작, node.js 자체는 multi-thread

Node.js
아키텍처 및 동작



Node.js
아키텍처 및 동작



Node.js

숨겨진 threads

main thread(1) + libuv threads(4) + v8 엔진 threads(2)

= **7 threads**

Node.js

js에서의 CPU-intensive 작업

i/o작업과 다르게 CPU-intensive 작업은 main 스레드를 blocking

(complex calculations, image resizing, video compression…)

worker threads 모듈을 사용하여 multi-threads를 만들어 병렬로 실행 가능!

js에서의 CPU-intensive 작업

main 스레드 blocking

```
const express = require("express");

const app = express();
const port = process.env.PORT || 3000;

app.get("/non-blocking/", (req, res) => {
  res.status(200).send("This page is non-blocking");
});

app.get("/blocking", (req, res) => {
  let counter = 0;
  for (let i = 0; i < 20_000_000_000; i++) {
    counter++;
  }
  res.status(200).send(`result is ${counter}`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

js에서의 CPU-intensive 작업

main 스레드 blocking

```
const express = require("express");

const app = express();
const port = process.env.PORT || 3000;

app.get("/non-blocking/", (req, res) => {
  res.status(200).send("This page is non-blocking");
});

app.get("/blocking", (req, res) => {
  let counter = 0;
  for (let i = 0; i < 20_000_000_000; i++) {
    counter++;
  }
  res.status(200).send(`result is ${counter}`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

GET /blocking → GET /non-blocking

1. main thread에서 for loop 작업 실행
2. for loop 작업 완료 후 /blocking 요청에 대한 응답을 client에게 전송
3. /non-blocking 요청에 대한 응답을 client에게 전송

js에서의 CPU-intensive 작업

Promise

Promise를 사용해서 해결?

CPU-intensive 작업을 promise로 래핑해도 main 스레드의 blocking 문제는 해결 X

js에서의 CPU-intensive 작업

Promise

```
const express = require("express");

const app = express();
const port = process.env.PORT || 3000;

app.get("/non-blocking/", (req, res) => {
  res.status(200).send("This page is non-blocking");
});

function calculateCount() {
  return new Promise((resolve, reject) => {
    let counter = 0;
    for (let i = 0; i < 20_000_000_000; i++) {
      counter++;
    }
    resolve(counter);
  });
}

app.get("/blocking", async (req, res) => {
  const counter = await calculateCount();
  res.status(200).send(`result is ${counter}`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

js에서의 CPU-intensive 작업

Promise

```
const express = require("express");

const app = express();
const port = process.env.PORT || 3000;

app.get("/non-blocking/", (req, res) => {
  res.status(200).send("This page is non-blocking");
});

function calculateCount() {
  return new Promise((resolve, reject) => {
    let counter = 0;
    for (let i = 0; i < 20_000_000_000; i++) {
      counter++;
    }
    resolve(counter);
  });
}

app.get("/blocking", async (req, res) => {
  const counter = await calculateCount();
  res.status(200).send(`result is ${counter}`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

GET /blocking → GET /non-blocking

1. calculateCount( ) 함수가 호출되면서 for loop 작업에 대한 promise 객체를 생성
2. for loop 작업이 완료되면 promise가 reslove
3. /blocking 요청에 대한 응답이 event queue에 추가
4. /non-blocking route가 응답을 client에게 전송
5. /blocking 요청에 대한 응답이 client에게 전송

js에서의 CPU-intensive 작업

worker thread 사용

```
const { parentPort } = require("worker_threads");

let counter = 0;
for (let i = 0; i < 20_000_000_000; i++) {
  counter++;
}

parentPort.postMessage(counter);
```

worker.js

js에서의 CPU-intensive 작업

worker thread 사용

```
const { parentPort } = require("worker_threads");

let counter = 0;
for (let i = 0; i < 20_000_000_000; i++) {
  counter++;
}

parentPort.postMessage(counter);
```

worker.js

```
const express = require("express");
const { Worker } = require("worker_threads");

const app = express();
const port = process.env.PORT || 3000;

app.get("/non-blocking/", (req, res) => {
  res.status(200).send("This page is non-blocking");
});

app.get("/blocking", async (req, res) => {
  const worker = new Worker("./worker.js");
  worker.on("message", (data) => {
    res.status(200).send(`result is ${data}`);
  });
  worker.on("error", (msg) => {
    res.status(404).send(`An error occurred: ${msg}`);
  });
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

index.js

js에서의 CPU-intensive 작업

4 worker threads 사용하여 최적화

```
const { workerData, parentPort } = require("worker_threads");

let counter = 0;
for (let i = 0; i < 20_000_000_000 / workerData.thread_count; i++) {
  counter++;
}

parentPort.postMessage(counter);
```

worker.js

js에서의 CPU-intensive 작업

4 worker threads 사용하여 최적화

```
const { workerData, parentPort } = require("worker_threads");

let counter = 0;
for (let i = 0; i < 20_000_000_000 / workerData.thread_count; i++) {
  counter++;
}

parentPort.postMessage(counter);
```

worker.js

```
const express = require("express");
const { Worker } = require("worker_threads");

const app = express();
const port = process.env.PORT || 3000;
const THREAD_COUNT = 4;

function createWorker() {
  return new Promise(function (resolve, reject) {
    const worker = new Worker("./four_workers.js", {
      workerData: { thread_count: THREAD_COUNT },
    });
    worker.on("message", (data) => {
      resolve(data);
    });
    worker.on("error", (msg) => {
      reject(`An error ocurred: ${msg}`);
    });
  });
}

app.get("/blocking", async (req, res) => {
  const workerPromises = [];
  for (let i = 0; i < THREAD_COUNT; i++) {
    workerPromises.push(createWorker());
  }
  const thread_results = await Promise.all(workerPromises);
  const total = thread_results.reduce((sum, currValue) => sum + currValue, 0);

  res.status(200).send(`result is ${total}`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

index.js

Node.js 멀티스레딩의 함정

worker thread는 일반적으로 알려진 스레드와 다름

- 일반 멀티-스레드 환경

  state sharing > race-condition 방지하기 위한 관리 필요

Node.js 멀티스레딩의 함정

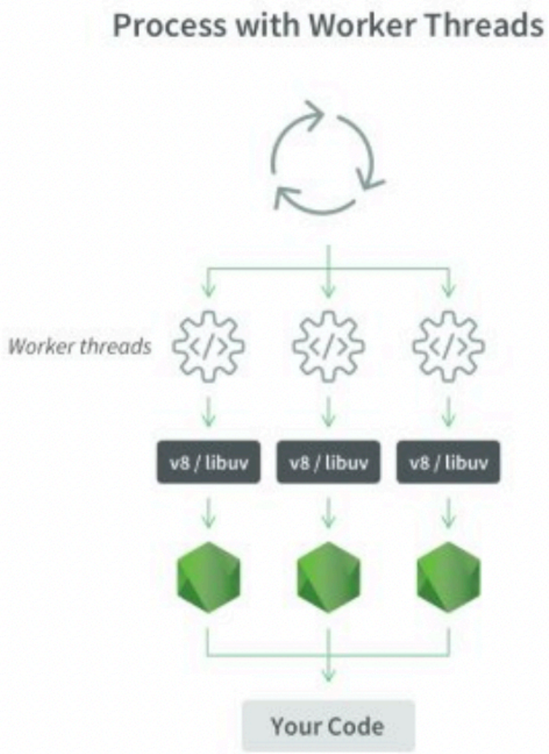
worker thread는 일반적으로 알려진 스레드와 다름

- node.js의 멀티-스레드 환경(worker threads)

  메인 프로세스와 worker thread간 state sharing 일어나지 않음

  데이터 공유는?

  ’event-based messaging system’, ‘share memory 선언’



Node.js 멀티스레딩의 함정

worker thread 생성 비용

- 일반 multi-thread < worker thread < process forking

  state sharing > race-condition 방지하기 위한 관리 필요

- ‘성능 향상 효과’가 ‘worker thread 생성 비용’보다 더 큰 경우에만 사용하는 것이 적합!

  CPU-intensive 작업은 적합, i/o작업은 낭비

END