TSBM 메이커 회고

Deep Dive into Node.js Internals

2026.01.28

Github @EeeasyCode

Software Engineer of 플레어랩스

TSBM 메이커 회고: Deep Dive into Node.js Internals

발표자 소개

이창민

- 플레어랩스 소프트웨어 엔지니어
- node.js developer (nestjs)
- TSBM 메이커 1기 스터디 운영

TSBM 메이커 회고: Deep Dive into Node.js Internals

TSBM 1기 메이커 활동

node.js 디자인 패턴 바이블 스터디

Introduction

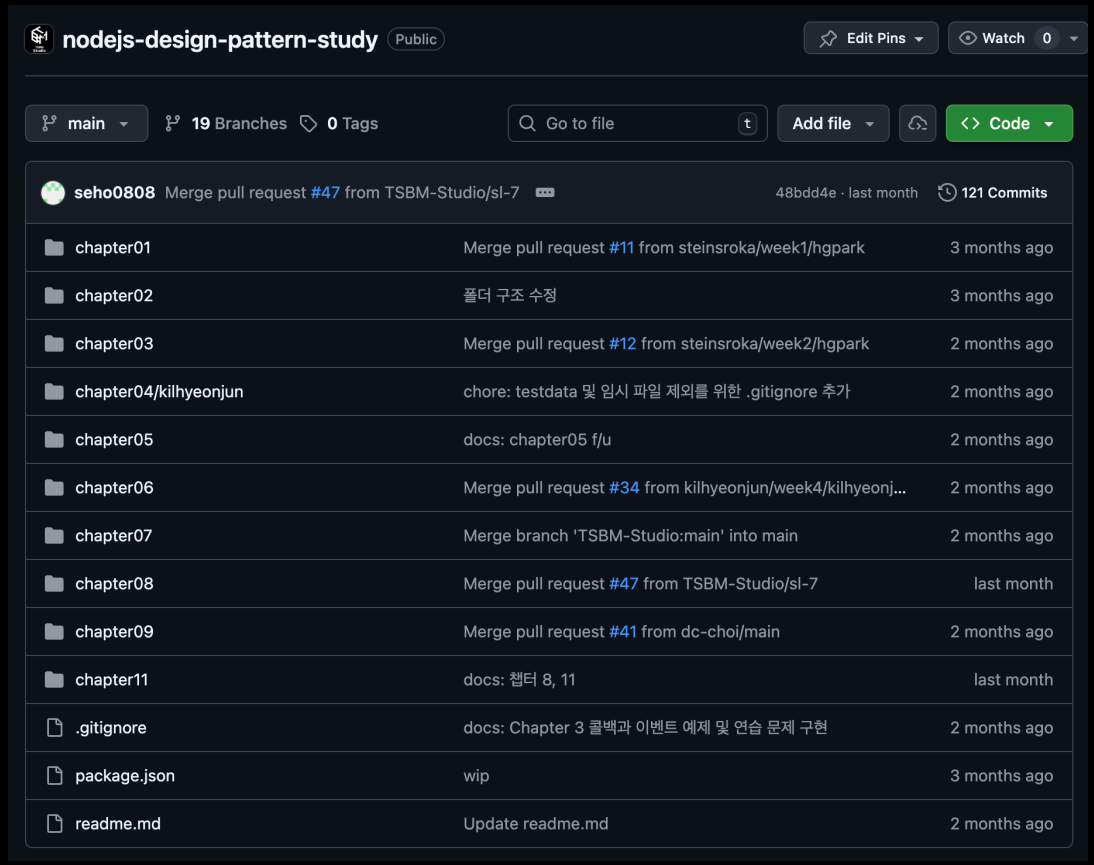
node.js 디자인 패턴 바이블 책을 읽고 배운 내용을 자유롭게 공유하는 스터디입니다.

Who need this study?

- node.js 개발자로서 코어한 부분을 학습하고 싶은 사람
- 남들과 자유롭게 다양한 개발 주제로 이야기하고 싶은 사람
- 다시 기본부터 쌓아가고 싶은 node.js 개발자

How to Study

- 책을 읽고 배운 내용을 요약하여 자료를 준비합니다.
- 책은 이론 위주로 구성되어 있어 실제 코드로 구현해보는 것이 좋습니다.
- 준비한 자료를 바탕으로 발표를 진행합니다.
- 발표 후 질의응답을 진행합니다.



TSBM 메이커 회고: Deep Dive into Node.js Internals

개인적인 목표

- 어쩌면 실무에 당장 도움이 되지는 못할 수도 있지만 엔지니어로서 deep dive를 해보는 것
- 면접이나 누군가에게 node.js 설명을 할 때, 정말 자신있게 설명할 수 있는지 점검
- Why? 라는 질문을 스스로 많이 해보는 것

마주한 문제

- 문서와 여러 블로그 글에 설명되어 있는 추상화된 내용들
- 이론까지만 이해하고 실제 어떻게 동작하는지 연결되지 않는 개념들
- node.js 생태계답게 하나의 개념에도 다양한 해석들이 존재

-> 추상화된 API와 문서만 보고 학습하는 것이 아닌 실제 node.js와 libuv 코드를 따라가보자

TSBM 메이커 회고: Deep Dive into Node.js Internals

공식 문서 읽어보기

핵심 키워드

- 비동기
- 이벤트 기반
- 논블로킹

Node.js®에 대하여

Node.js는 비동기 이벤트 기반의 JavaScript 런타임으로, 확장 가능한 네트워크 애플리케이션을 구축하도록 설계되었습니다. 다음의 "hello world" 예제에서는 많은 연결을 동시에 처리할 수 있습니다. 각 연결마다 콜백이 호출되지만, 할 일이 없으면 Node.js는 대기 상태가 됩니다.

CJS ESM

```javascript
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

JavaScript </> 클립보드에 복사

이는 운영 체제 스레드를 사용하는 오늘날의 더 일반적인 동시성 모델과 대조됩니다. 스레드 기반 네트워킹은 상대적으로 비효율적이며 사용하기도 매우 어렵습니다. 또한, Node.js 사용자는 락(lock)이 없기 때문에 프로세스가 데드락에 걸릴 걱정을 할 필요가 없습니다. Node.js의 거의 모든 함수는 직접 I/O를 수행하지 않으므로, Node.js 표준 라이브러리의 동기 메서드를 사용하여 I/O를 수행하는 경우를 제외하고는 프로세스가 차단되지 않습니다. 이처럼 차단이 발생하지 않기 때문에 Node.js에서는 확장 가능한 시스템을 개발하는 것이 매우 적합합니다.

출처: node.js 공식문서 (https://nodejs.org/ko/about)

예제 코드로 deep dive

```
1  const { createServer } = require("node:http");
2
3  const hostname = "127.0.0.1";
4  const port = 3000;
5
6  const server = createServer((req, res) => {
7    res.statusCode = 200;
8    res.setHeader("Content-Type", "text/plain");
9    res.end("Hello World");
10 });
11
12 server.listen(port, hostname, () => {
13   console.log(`Server running at http://${hostname}:${port}/`);
14 });
15
```

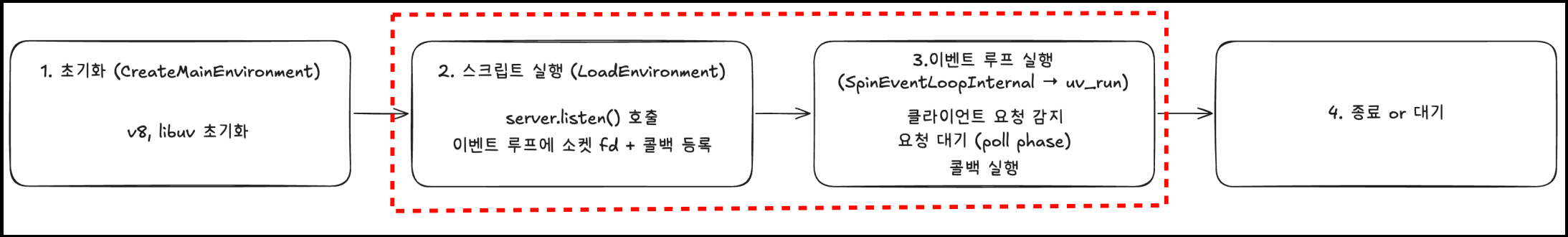
예제 코드를 통해 확인해볼 것

- 공식 문서의 키워드들이 node.js 내부 흐름과 어떻게 이어지는지 확인
- 예제 코드를 따라가며 node.js 각 레이어 연결 지점 파악
- server.listen()부터 handler가 호출되는 지점까지 내부 경로 추적

→ 모든 걸 다루지 않고, 핵심 지점들만 확인해보자.

예제 코드로 deep dive

이번 발표에서는



스크립트 실행 단계

- server.listen() 호출 시 내부에서 무슨 일이 일어나는지
- http → net → C++ binding → libuv → 이벤트 루프에 등록

이벤트 루프 실행 단계

- 클라이언트 요청 → 콜백 실행까지

예제 코드로 deep dive

전달하고자 하는 것

- 이런 방식으로도 공부해볼 수 있구나
- 아 이건 몰랐는데? 오.. 알쓸신잡 지식 +1
- 오? TSBM 메이커 나도 해볼 만하겠는걸?
- 나도 TSBM에서 발표할 수 있겠네

예제 코드로 deep dive

```
1  const { createServer } = require("node:http");
2
3  const hostname = "127.0.0.1";
4  const port = 3000;
5
6  const server = createServer((req, res) => {
7    res.statusCode = 200;
8    res.setHeader("Content-Type", "text/plain");
9    res.end("Hello World");
10 });
11
12 server.listen(port, hostname, () => {
13   console.log(`Server running at http://${hostname}:${port}/`);
14 });
15
```

server.listen()의 실제 구현 코드를 찾아보자.

- 뭔가 node:http의 createServer를 따라가보면 좋을 것 같은데?
- node.js 코드 베이스 클론해서 봐야겠다
- 파다보면 libuv, 커널 레벨까지 연결의 흐름을 확인해볼 수 있겠네

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

require("node:http").createServer()의 구현체 찾기

```
1  // lib/http.js
2  const { Server, ... } = require('_http_server');
3
4  function createServer(...) {
5    return new Server(...);
6  }
```

_htt_server의 Server를 return하고 있네? → 그럼 거기로 가보자

lib/_http_server의 Server 내부에서는 net.Server을 호출하네?

```
1  // lib/_http_server.js
2  function Server(options, requestListener) {
3    net.Server.call(this, { ... });
4  }
5
6  ObjectSetPrototypeOf(Server.prototype, net.Server.prototype);
```

그럼 실제 네트워크 동작은 net.Server에 있으려나? 한번 더 가보자

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

lib/net.js 를 찾다보니, 예제 코드에서 사용한 listen 함수 발견!

```
// lib/net.js
Server.prototype.listen = function(...args) {

  // 1. 인자 정규화
  const normalized = normalizeArgs(args);
  let options = normalized[0];     // { port: 3000 }
  const cb = normalized[1];        // 콜백 (있으면)

  // 2. 콜백 등록
  if (cb !== null) {
    this.once('listening', cb);
  }

  // 3. 포트가 있는 경우 → listenInCluster 호출
  if (typeof options.port === 'number') {

    listenInCluster(this, null, options.port | 0, 4, backlog, ...);

    return this;
  }
}
```

listenInCluster 내부에서는 setupListenHandle를 호출하고 있음

setupListenHandle 은 C++ 바인딩으로 TCP 객체에 뭔가를 하네?

```
// lib/net.js
const { TCP } = internalBinding('tcp_wrap');

function setupListenHandle(address, port, ...) {
  // 1. C++ 바인딩 객체 생성
  this._handle = new TCP(TCPConstants.SERVER);

  // 2. 주소 바인딩
  this._handle.bind(address, port);

  // 3. 리스닝 시작
  this._handle.listen(backlog);
}
```

tcp_wrap의 listen까지 따라가보면 진짜 다온 것 같은데?

예제 코드로 deep dive

어쩌다보니, C++ 코드까지 왔네.. tcp_wrap 찾았다!

```cpp
// src/tcp_wrap.cc
SetProtoMethod(isolate, t, "listen", Listen);

// tcp_wrap의 listen 메서드
void TCPWrap::Listen(const FunctionCallbackInfo<Value>& args) {
  // ..

  // libuv의 uv_listen 함수 호출
  int err = uv_listen(
    reinterpret_cast<uv_stream_t*>(&wrap->handle_),
    backlog,
    OnConnection //<-- 연결이 들어왔을 때 실행될 'C++ 콜백 함수' 등록
  );
}
```

결국에 해당 지점에서는 libuv의 uv_listen 을 호출하기 위함인거네?

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

그럼 libuv의 uv_listen은 뭘까?

```c
// deps/uv/src/unix/tcp.c
int uv__tcp_listen(uv_tcp_t* tcp, int backlog, uv_connection_cb cb) {

  listen(tcp->io_watcher.fd, backlog);

  tcp->connection_cb = cb;

  uv__io_start(tcp->loop, &tcp->io_watcher, POLLIN);
}
```

1. 커널에 소켓의 상태를 listen으로 변경 요청
2. 연결 요청 시 실행할 콜백 등록
3. uv_io_start..? 이건 뭐지?

uv__io_start() → 이벤트 루프 동작 전 최종 단계

```c
// deps/uv/src/unix/core.c
int uv__io_start(uv_loop_t* loop, uv__io_t* w, unsigned int events) {

  w->pevents |= events;

  uv__queue_insert_tail(&loop->watcher_queue, &w->watcher_queue);

  loop->watchers[w->fd] = w;
}
```

1. 이 fd가 어떤 이벤트에 반응할지 기록 (POLLIN - 읽기 가능 이벤트)
2. 이벤트 루프가 순회할 큐에 watcher 추가
3. fd를 인덱스로 배열에 watcher 저장 (이벤트 발생 시 빠르게 찾기 위함)

tcp.c 코드에서 호출한 내용을 기반으로 설명하자면

→ 서버 소켓 fd에 연결 요청(POLLIN)이 오면, connection_cb 호출되도록 등록

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

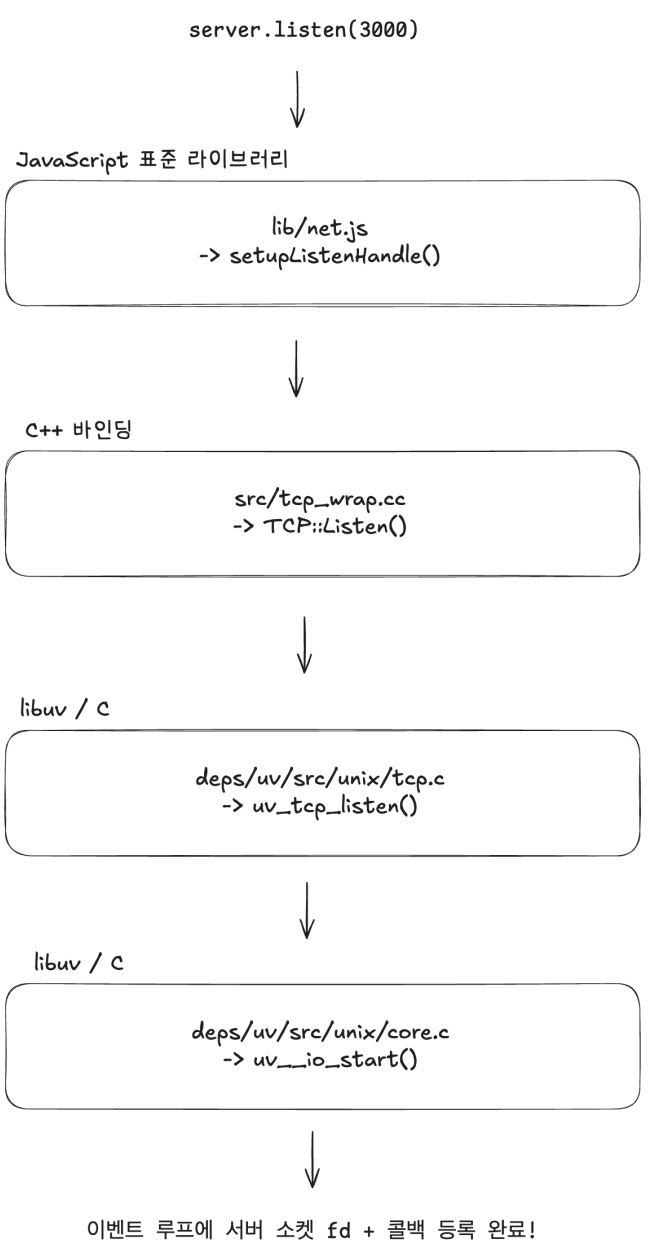
그럼 지금까지 무슨 과정이 있었던거지?

여기까지는 스크립트 실행 단계

→"나중에 이벤트 오면 이렇게 처리해줘"를 등록만 한 것

다음은 이벤트 루프 실행 단계

→ 실제로 이벤트가 오면 어떻게 처리되는지



예제 코드로 deep dive

등록 단계는 끝났고, 이제 이벤트를 기다릴 차례

등록 단계에서 한 일

- io_watcher.cb = uv__server_io (콜백 지정)
- io_watcher.fd = 서버 소켓 fd (감시할 대상)
- loop->watchers[fd] = &io_watcher (조회 목적 저장)
- watcher_queue에 추가 (루프가 순회할 목록)

이제 이벤트 루프가 할 일

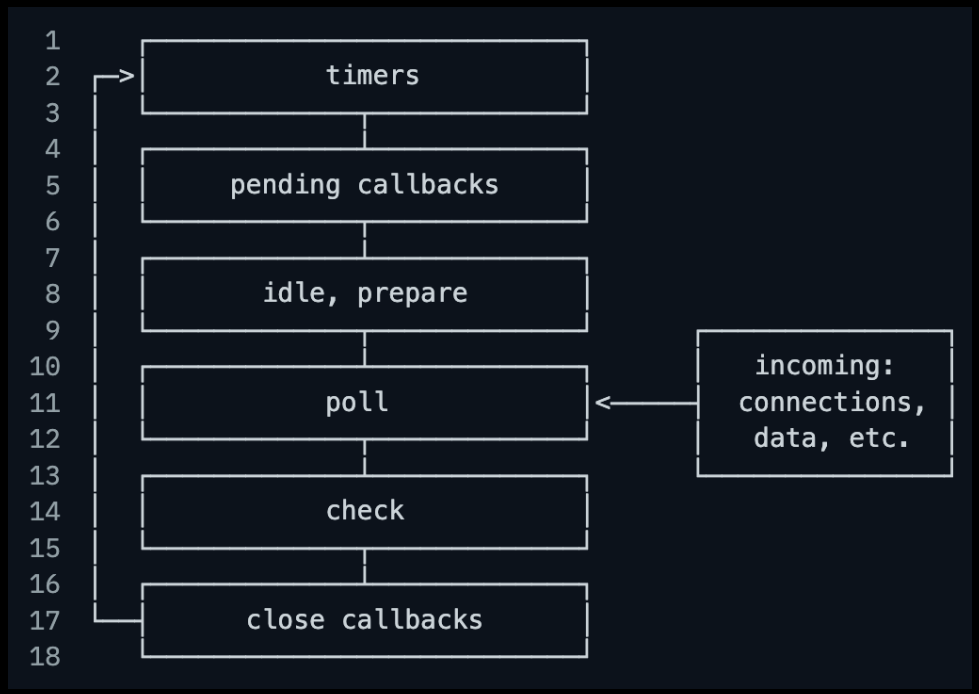
- watcher_queue를 epoll에 등록
- epoll_pwait()로 fd 이벤트 대기
- 이벤트 발생 시 → watchers[fd]로 watcher 찾기
- watcher->cb() 호출 → uv__server_io 실행

"클라이언트 요청이 들어오면, 이벤트 루프는 어떻게 처리할까?"

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

이벤트루프는 어떻게 동작하나?



출처: node.js 공식문서 (https://nodejs.org/ko/learn/asynchronous-work/event-loop-timers-and-nexttick#event-loop-explained)

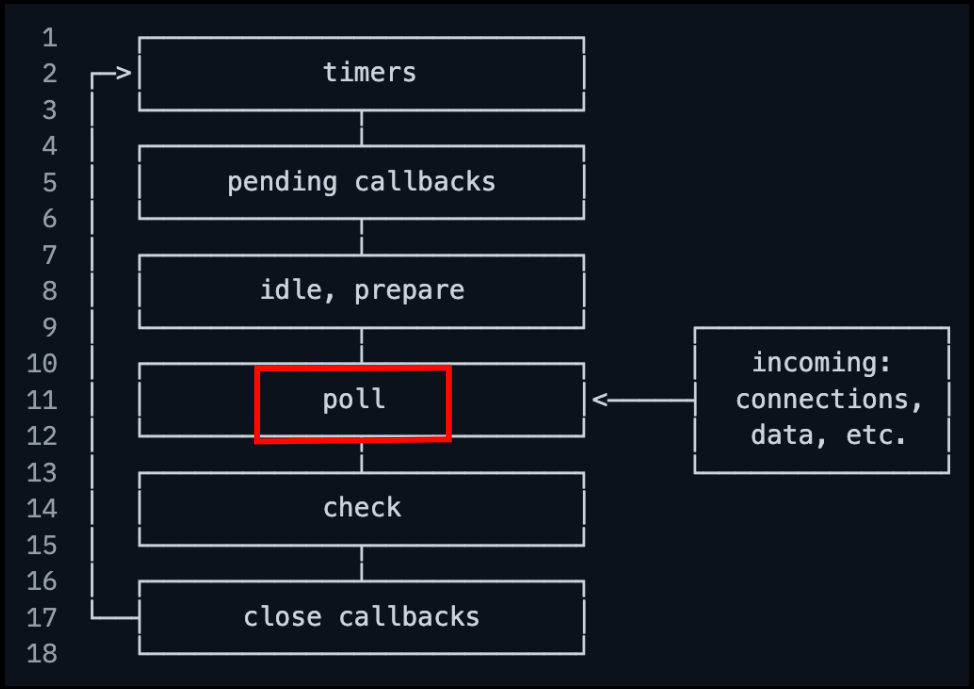
Node.js 프로세스가 살아있는 동안 **uv_run()**이 반복해서 실행됨

```c
// deps/uv/src/unix/core.c - uv_run()
while (loop->stop_flag == 0) {
  uv__run_timers(loop);           // timers
  uv__run_pending(loop);          // pending callbacks
  uv__run_idle(loop);             // idle, prepare
  uv__run_prepare(loop);
  uv__io_poll(loop, timeout);     // poll
  uv__run_check(loop);            // check
  uv__run_closing_handles(loop);  // close callbacks
}
```

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

그럼 우리가 봐야할 loop의 핵심 동작은? → uv_io_poll



출처: node.js 공식문서 (https://nodejs.org/ko/learn/asynchronous-work/event-loop-timers-and-nexttick#event-loop-explained)

왜 uv_io_poll 인가?

- server.listen()은 네트워크 I/O
- 클라이언트 연결 요청은 I/O 이벤트
- I/O 이벤트는 uv_io_poll에서 처리됨

poll 단계에서 하는 일

- 새로 등록된 watcher를 OS에 반영
- 커널에 이벤트 대기 요청
- 이벤트 발생 시 fd로 watcher 찾기
- watcher의 콜백 호출

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

uv__io_poll() 내부 동작 자세히 보자

```c
// deps/uv/src/unix/linux.c (linux 기반이라 가정)
void uv__io_poll(uv_loop_t* loop, int timeout) {

  // 1. 커널에 이벤트 대기 요청
  nfds = epoll_pwait(epollfd, events, ARRAY_SIZE(events), timeout, sigmask);

  // 2. 이벤트 발생한 fd들 순회
  for (i = 0; i < nfds; i++) {
      pe = events + i;
      fd = pe->data.fd;

      // 3. fd로 watcher 찾기
      w = loop->watchers[fd];

      // 4. 콜백 호출
      w->cb(loop, w, pe->events);
  }
}
```

epoll_pwait()

→ 커널에 등록된 fd 중에 이벤트 있는지 확인

→ 이벤트가 없으면 대기, 있으면 즉시 반환

loop → watchers[fd]

→ fd 번호를 인덱스로 watcher를 찾음

w → cb(...)

→ 등록해둔 콜백을 호출함

→ TCP 서버의 경우 **uv__server_io** 를 호출

예제 코드로 deep dive

현재 시점과 클라이언트 요청이 발생하게 된다면?

현재 서버 상태

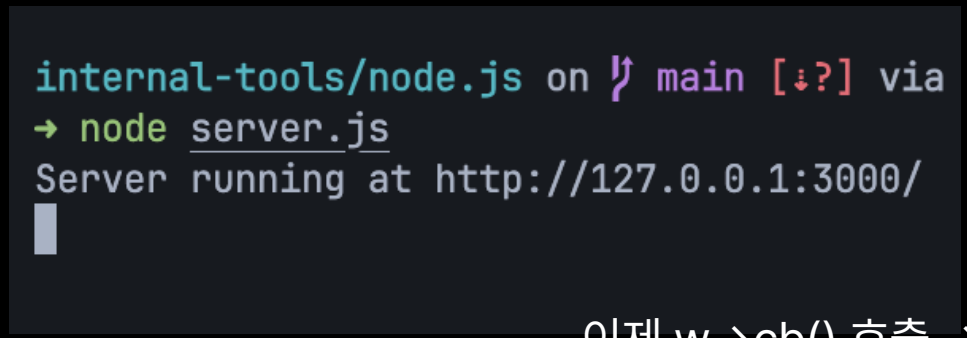
- epoll_wait() 대기

클라이언트 요청 발생

$ curl http://localhost:3000

커널의 동작

1. 서버 소켓 fd에서 EPOLLIN 이벤트 감지 (새 연결 요청 도착)
2. epoll_wait() 대기 해제 → 이벤트 목록 반환
3. for 루프 진입 - w→cb() 호출



이제 w→cb() 호출 → uv__server_io 실행

등록 단계의 uv__tcp_listen에서 io_watcher.cb = uv__server_io로 설정했기 때문

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

이벤트 감지 → uv__server_io 호출

```
1  // deps/uv/src/unix/stream.c
2  void uv__server_io(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
3
4    // watcher로부터 stream 구조체 찾기
5    stream = container_of(w, uv_stream_t, io_watcher);
6
7    // 서버 소켓의 fd 가져오기
8    fd = uv__stream_fd(stream);
9
10   // 1. accept 시스템 콜 - 클라이언트 연결 수락
11   err = uv__accept(fd);
12
13   // 2. 새 클라이언트의 fd 저장
14   stream->accepted_fd = err;
15
16   // 3. connection_cb 호출 - C++의 OnConnection
17   stream->connection_cb(stream, 0);
18 }
```

uv_accept(fd)

→ 커널에 "대기 중인 연결 하나 수락" 요청

→ 성공 시, 새로운 클라이언트 소켓 fd 반환

accepted_fd

→ 새로 생성된 클라이언트 소켓 fd를 임시 저장

→ 다음 레이어에서 처리하기 위해 전달 목적

connection_cb

→ 등록 단계에서 uv_listen(..., OnConnection)으로 전달됨

→ tcp→connection_cb = OnConnection으로 저장됨

→ 여기서 호출 시 C++의 OnConnection 실행

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

다시 C++로 .. → OnConnection()

Instantiate()

→ 새 클라이언트를 위한 TCPWrap 객체 생성

→JS가 직접 syscall 호출 불가해 wrap으로 브릿지

uv_accept()

→ 이전에 받아운 client_fd를 새 TCPWrap에 연결

→ 이제 이 객체로 클라이언트와 통신 가능

MakeCallback()

→JS의 onconnection 함수를 호출함

→ 우리가 등록한 JS 콜백을 실행시키기 위함

→ C++에서 JS로 넘어가는 경계 지점

```cpp
// src/connection_wrap.cc
template <typename WrapType, typename UVType>
void ConnectionWrap<WrapType, UVType>::OnConnection(
    uv_stream_t* handle, int status) {

    WrapType* wrap_data = static_cast<WrapType*>(handle->data);
    Environment* env = wrap_data->env();

    Local<Value> client_handle;

    if (status == 0) {
        // 1. 클라이언트용 JavaScript 객체 생성
        Local<Object> client_obj;
        WrapType::Instantiate(env, wrap_data, WrapType::SOCKET)
            .ToLocal(&client_obj);

        // 2. uv_accept 호출하여 연결 수락 완료
        WrapType* wrap;
        ASSIGN_OR_RETURN_UNWRAP(&wrap, client_obj);
        uv_stream_t* client = reinterpret_cast<uv_stream_t*>(&wrap->handle_);
        uv_accept(handle, client);

        client_handle = client_obj;
    }

    // 3. JavaScript 콜백 호출
    Local<Value> argv[] = { Integer::New(..., status), client_handle };
    wrap_data->MakeCallback(env->onconnection_string(), arraysize(argv), argv);
}
```

TSBM 메이커 회고: Deep Dive into Node.js Internals

예제 코드로 deep dive

다시 JS로 넘어가자

clientHandle

→ C++에서 생성한 TCPWrap 객체

→ 내부에 클라이언트 fd를 가지고 있음

new Socket()

→ 사용자가 사용할 Socket 인스턴스를 생성

→ clientHandle을 내부에 저장

self.emit('connection', socket)

→'connection' 이벤트를 발생시킴

→이 이벤트를 누가 듣고 있을까?

```
1  // lib/net.js
2  function onconnection(err, clientHandle) {
3
4    const handle = this;
5    const self = handle[owner_symbol];  // Server 인스턴스
6
7    if (err) {
8        self.emit('error', new ErrnoException(err, 'accept'));
9        return;
10   }
11
12   // 클라이언트 Socket 객체 생성
13   const socket = new Socket({
14       handle: clientHandle,
15       allowHalfOpen: self.allowHalfOpen,
16       pauseOnCreate: self.pauseOnConnect,
17       readable: true,
18       writable: true,
19   });
20
21   self._connections++;
22   socket.server = self;
23   socket._server = self;
24
25   // 'connection' 이벤트 발생!
26   self.emit('connection', socket);
27 }
```

예제 코드로 deep dive

connection 이벤트를 듣고 있는 곳은?

우리가 등록 단계에서 지나쳤던 lib/_http_server.js 에서 이미 리스너를 등록했다!

```
1  // lib/_http_server.js
2  function Server(options, requestListener) {
3
4    net.Server.call(this, { ... });
5
6    // 클라이언트 연결 이벤트 발생 시 실행
7    this.on('connection', connectionListener);
8
9    // http 파싱이 완료되고 이벤트 발생 -> 실행
10   if (requestListener) {
11       this.on('request', requestListener);
12   }
13 }
```

그럼 connectionListener가 실행된다는 뜻인데, 이건 뭘까?

→ HTTP 파서를 생성해서 클라이언트가 HTTP 요청 데이터를 보내면 파싱

→ 파싱이 완료되면 해당 핸들러 내에서 emit('request', req, res) 발생

예제 코드로 deep dive

connectionListener가 request 이벤트를 발생시키면?

우리가 정의한 콜백 핸들러가 동작하는 최종 지점이다.

```
// lib/_http_server.js
function Server(options, requestListener) {

  net.Server.call(this, { ... });

  // 클라이언트 연결 이벤트 발생 시 실행
  this.on('connection', connectionListener);

  // http 파싱이 완료되고 이벤트 발생 -> 실행
  if (requestListener) {
      this.on('request', requestListener);
  }
}
```

```
const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader("Content-Type", "text/plain");
  res.end("Hello World");
});
```

```
internal-tools/node.js on  main
➜ curl -s http://localhost:3000/
Hello World%
```

TSBM 메이커 회고: Deep Dive into Node.js Internals

증명해본 내용

비동기 (Asynchronous) **논블로킹 (Non-Blocking)** **이벤트 기반 (Event-driven)**

각각의 키워드는 node.js가 이벤트 루프를 통해 제공하는 특성임을 직접 확인해봤다.

1. I/O 요청 후 결과를 기다리지 않고, 등록되었던 콜백/핸들러를 통해 결과를 받는다.
2. OS의 논블로킹 I/O 작업 처리를 통해 메인 스레드가 블로킹되지 않도록 한다.
3. 실행 흐름은 "순서"가 아니라 이벤트 발생 → 콜백 실행으로 이어진다.

추상적이던 개념들을 직접 눈으로 확인해보는 과정으로 검증해보는 것도 나쁘지는 않은 것 같다.

근데 너무 힘들다 ..ㅠ (자주 하는건 비추..)

QnA