

Diagnose your Node.js

app

2018/11/23

by darai0512

- slide <https://darai0512.github.io/nodefest2018/index.html#/>
- twitter https://twitter.com/darai_0512

23-24TH NOVEMBER

東京Node学園祭2018

console.table(me)

(index)

Values

name darai0512

company Yahoo! JAPAN

work 広告向けデータ分析システム

社内言語サポート(Node.js)

side jobs PAY株式会社

株式会社ミツモア

Node.js初学者の悩み



1. 非同期処理が難しい

- APIの実行順が直感が違う
- エラーが再現しない...

2. エラーハンドリングが難しい

- try-catch? Promise? callbackやon('error')?

3. debugが難しい

- 非同期中に例外が起きた時、スタックトレースが途切れるんだけど...
- 本番アプリが落ちた...どうdebugすればいい?

Node Summit Code of Conduct

Node Summit and its owner, Asynch Media, are dedicated to providing a harassment-free conference experience for everyone, regardless of gender identity, age, sexual orientation, disability, physical appearance, experience and religious or other beliefs.

We do not tolerate harassment of participants in any form. Conference participants violating these rules may be sanctioned or expelled from the conference without a refund at the discretion of the conference organizers.

Details

Part of the success of the Node.js community is due to the fact many early members have a strong bias towards respect. As a participant of a Node Summit event, you are helping support the goal of a strong, vibrant, inclusive Node.js community where everyone feels welcome and safe.

We are committed to providing a harassment free event experience and this is only possible if participants are committed to this objective.

Participants include everyone involved in a Node Summit event; organizers, attendees, speakers, sponsors, staff, volunteers and gate crashers. Harassment includes but is not limited to offensive verbal comments related to gender identity, age, sexual orientation, disability, physical appearance, body size, race, experience and religious or other beliefs. It also includes displaying sexual images in public spaces, deliberate intimidation, stalking, following, harassing photography or recording, sustained disruption of sessions, inappropriate physical contact and unwelcome attention.

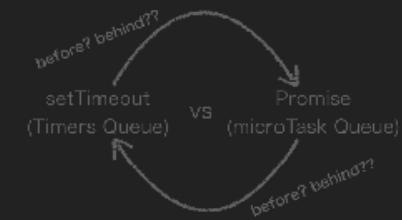
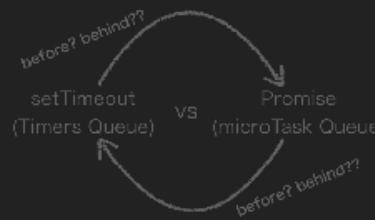
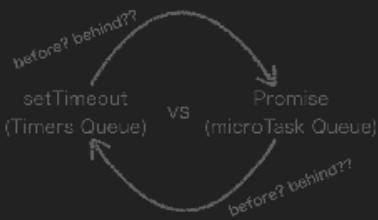
Participants asked to stop any harassing behavior are expected to comply immediately. Failure to do so will result in removal from future Node Summit events.

If a participant engages in harassing behavior, the conference organizers will take appropriate action, including but not limited to warning the offender or expulsion from the event.

If you are being harassed, notice that someone else is being harassed, or otherwise assist those experiencing harassment, including but not limited to warning the offender or expulsion from the event.

FIND: Any member of the conference staff, including but not limited to the conference organizers, who are available to help.

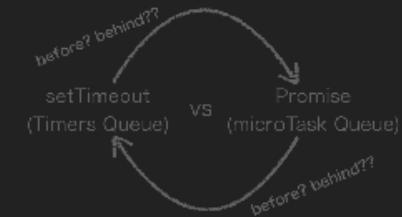
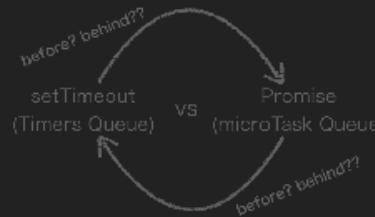
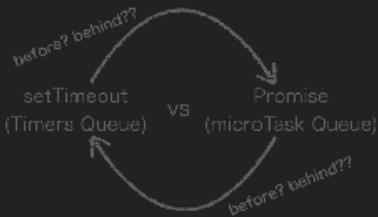
CONTACT: If you are unsure who to approach, contact the conference organizers.



非同期処理が難しい

ex1, 次の出力は? 🤔

```
(T) for (let i = 0; i < 2; i++) {  
    setTimeout(() => {  
        console.log(`Timeout ${i}`);  
        Promise.resolve(`Promise ${i}`).then(console.log);  
    }, 4);  
}
```



ブラウザで実行してみたAさんの答え

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. On the left, there's a sidebar with various log levels: 4 messages, 4 user messages, No errors, No warnings, 4 info, and No verbose. The main area displays the following JavaScript code and its execution results:

```
for (let i = 0; i < 2; i++) {
  setTimeout(() => {
    console.log(`Timeout ${i}`);
    Promise.resolve(`Promise ${i}`).then(console.log);
  }, 4);
}

2160
```

The output in the console is:

- Timeout 0
- Promise 0
- Timeout 1
- Promise 1

Node.js v10.xで実行したBさんの答え

```
[@~/Sites/nodefest]
```

```
(^o^)/~$node --version
```

```
v10.13.0
```

```
[@~/Sites/nodefest]
```

```
(^o^)/~$cat demo/async_order.js
```

```
for (let i = 0; i < 2; i++) {
```

```
    setTimeout(() => {
```

```
        console.log(`Timeout ${i}`);
```

```
        Promise.resolve(`Promise ${i}`).then(console.log);
```

```
    }, 4);
```

```
}
```

```
[@~/Sites/nodefest]
```

```
(^o^)/~$node demo/async_order.js
```

```
Timeout 0
```

```
Timeout 1
```

```
Promise 0
```

```
Promise 1
```

Node.js v11.xで実行したCさんの答え

```
[@~/Sites/nodefest]
```

```
(^o^)/~$node --version
```

```
v11.1.0
```

```
[@~/Sites/nodefest]
```

```
(^o^)/~$cat demo/async_order.js
```

```
for (let i = 0; i < 2; i++) {
```

```
    setTimeout(() => {
```

```
        console.log(`Timeout ${i}`);
```

```
        Promise.resolve(`Promise ${i}`).then(console.log);
```

```
    }, 4);
```

```
}
```

```
[@~/Sites/nodefest]
```

```
(^o^)/~$node demo/async_order.js
```

```
Timeout 0
```

```
Promise 0
```

```
Timeout 1
```

```
Promise 1
```

非同期処理の実行順は環境やバージョンで異なりうる

- PromiseとsetTimeout/setIntervalの実行順がNode.js v11.xから変わった
 - <https://github.com/nodejs/node/pull/22842>
 - semver-majorで各種ブラウザの挙動に合わせた
- 非同期APIの実行順の互換性は非保証

ex2, 直感的ではない挙動(実際にあった例)

```
// 自身のコピーをとり、失敗したらリトライするスクリプト
const fs = require('fs');
const backup = `/tmp/backup` + process.argv[2];
const flag = fs.constants.COPYFILE_EXCL; // copy先ファイルが存在するとE
const cp = (m) => fs.copyFile(__filename, backup, flag, (e) => {
  if (e) return console.error(`(${m} ERROR) ${e.message}`); // コヒ
});
// メイン処理
cp('1st'); // コピー開始
if (fs.existsSync(backup)) { // コピー先ファイルが出来てたら
  console.log('copied'); // ログを出して終了
} else { // コピーが出来なければ
  cp('retry'); // リトライ
}
```

```
$node demo/unknown_order.js 1
(retry ERROR) EEXIST: file already exists, copyfile ... (省略)...
```

retry(=コピー先fileがない)なのにfile already exists?

```

const fs = require('fs');
const backup = `/tmp/backup` + process.argv[2];
const flag = fs.constants.COPYFILE_EXCL;
// copyFileの結果の取得(コールバック発火)は非同期だがシステムコールは即実行
const cp = (m) => fs.copyFile(__filename, backup, flag, (e) => {
  if (e) return console.error(`(${m} ERROR) ${e.message}`);
});
// メイン処理
cp('1st'); // existsSyncによるbackupへのaccessとの競走
if (fs.existsSync(backup)) { // copyFileによるbackupのopenとの競走
  console.log('copied');
} else {
  cp('retry'); // 既に1stでbackupのopenは終わっているためfile already e
}

```

但し再現性はなく並列実行時は一部copied(成功)

```

$for i in `seq 1 30`;do node ./demo/unknown_order.js ${i} & done
(retry ERROR) EEXIST: file already exists, copyfile ... (省略)...
(retry ERROR) EEXIST: file already exists, copyfile ... (省略)...
copied
(retry ERROR) EEXIST: file already exists, copyfile ... (省略)...
...

```

NOT SURE IF THE FLAKY TESTS ARE BROKEN

フロー制御をして実行順に依存しない設計に

- 非同期I/Oの実行順は不規則
- 非同期処理では順番/時間に依存しない設計が重要
 - callback/Promise/async-awaitなどで確実なフロー制御を
 - 悪い例: setTimeoutでコピーが終わりそうな時間を指定し待つ
 - 再現性がないとCIのテストが不安定に...(flaky)

OR THE CODE IS BROKEN

Async Hooks

#

Stability: 1 - Experimental

The `async_hooks` module provides an API to register callbacks tracking the lifetime of asynchronous resources created inside a Node.js application. It can be accessed using:

```
const async_hooks = require('async_hooks');
```

フロー制御出来るかはどう確認?

Terminology

#

Node core APIならAsync Hooks

An asynchronous resource represents an object with an associated callback. This callback may be called multiple times, for example, the `'connection'` event in `net.createServer()`, or just a single time like in `fs.open()`. A resource can also be closed before the callback is called. `AsyncHook` does not explicitly distinguish between these different cases but will represent them as the abstract concept that is a resource.

If `Workers` are used, each thread has an independent `async_hooks` interface, and each thread will use a new set of async IDs.

Public API

#

Overview

#

Async Hooks (v8.0.0~, Experimental)

Async Hooks

#

- 非同期I/O, タイマー, Promiseなどトレースできる
- 使い方: `createHook()`に非同期のライフタイム毎にさせたい処理を渡して`enable()`

The `async_hooks` module provides an API to register callbacks tracking the lifetime of asynchronous resources created inside a Node.js application. It can be accessed using:

- `init/destroy`: インスタンス生成/破棄時
- `before/after`: コールバック前後

Termination

#

- `promiseResolve`: `resolve`時(`then`も含む)

```
const ah = require('async_hooks');
const w = (v) => fs.writeFileSync(process.stdout.fd, v); // 注. ロギング
ah.createHook({
  init(id, type, triggerId, resource) {w(`[${type}] ${id} created`)},
  before(id) {w(`before ${id} callback in ${ah.executionAsyncId()}`)},
  after(id) {w(`after ${id} callback in ${ah.executionAsyncId()}`)},
  destroy(id) {w(`[${id}] destroy in ${ah.executionAsyncId()}`)},
  promiseResolve(id) {w(`PROMISE ${id} resolved`)}
}).enable();
// your app code
```

Async Hooks

#

Stability: 1 - Experimental

The `async_hooks` module provides an API to register callbacks tracking the lifetime of asynchronous resources created inside a Node.js application. It can be accessed using:

```
setImmediate(() => {});  
const console = require('console');
```

```
Immediate 5 created in 1 # 1はユーザーランドを意味  
TickObject 6 created in 1 # = console.log()  
before 6 callback in 6 # 記述順に生成はされたが、  
after 6 callback in 6    # 実行は後のconsole.logの方が早い  
6 destroy in 0  
before 5 callback in 5 # console.logの後で、  
after 5 callback in 5   # setImmediateのcallbackが呼ばれた  
5 destroy in 0 # 0はJavaScript stackではなくC++から処理されたことを意味
```

Terr

#

An asynchronous example before the abstract

closed
as

If Worker

s.

Public API

#

Overview

#

Performance Timing API Async Hooks Tips

Stability: 1 - Experimental

- type(コンストラクタ名:setIntervalならTimeout)が
わかりにくい
 - ex, スタックトレースで行番号/列番号を取得

Measuring the duration of async operations

```
Error.stackTraceLimit = 20; // console.logの表示にはこれくらい必要
const init = (id, type, triggerId, resource) => {
  const e = {};
  Error.captureStackTrace(e); // -> 'at AsyncHook init'
  e.stack.split(require('os').EOL).filter(v => // ユーザーランドのスタッ
    v.includes(' /') && !v.includes('at AsyncHook.init'));
// TickObject 6 at Object.<anonymous> (/tmp/demo.js:2:1)
};
```

- Performance Timing API(v8.5.0~,Experimental)と組
み合わせて間隔計測可能
 - 正確なパフォーマンス指標(cf. process.hrtime())
 - 参考: Measuring the duration of async operations

```
const set = new Set();
const hook = async_hooks.createHook(({
  init(id, type) {
    if (type === 'interval') {
      performance_hooks.setInterval(id);
      set.add(id);
    }
  },
  destroy(id) {
    if (set.has(id)) {
      set.delete(id);
    }
  }
}))
```

エラーハンドリングが難しい

type	handling	crash
非同期	Callback EventEmitter Promise async/await	if (err) .on('error') .catch(), .then()の第二引数 .catch(), try-catch
		
同期		try-catch, Promise, async/await
		

crash: 例外処理の記述が漏れた場合にプロセスがクラッシュ

- Callback型はPromise化できる(ex, require('util').promisify)
- (EventEmitterを除けば)Promiseでのハンドリングに統一可能

	type	handling	crash		type	handling	crash
非同期	Callback	if (err)		非同期	Callback	if (err)	
	EventEmitter	.on('error')	!		EventEmitter	.on('error')	!
	Promise	.catch(), .then()の第二引数			Promise	.catch(), .then()の第二引数	
	async/await	.catch(), try-catch try-catch, Promise, async/await	!		async/await	.catch(), try-catch try-catch, Promise, async/await	!

Promise(async/await)で統一すれば安全?

- EventEmitterのエラーハンドリングは必要
- クラッシュさせた方がいいケースもある

	type	handling	crash		type	handling	crash
非同期	Callback	if (err)		非同期	Callback	if (err)	
	EventEmitter	.on('error')	!		EventEmitter	.on('error')	!
	Promise	.catch(), .then()の第二引数			Promise	.catch(), .then()の第二引数	
	async/await	.catch(), try-catch try-catch, Promise, async/await	!		async/await	.catch(), try-catch try-catch, Promise, async/await	!

- core dumpが欲しい時(後述)

- 問題を孕んでいることのサイン
 - unhandledRejection

	type	handling	crash		type	handling	crash
非同期	Callback	if (err)		非同期	Callback	if (err)	
	EventEmitter	.on('error')	!		EventEmitter	.on('error')	!
	Promise	.catch(), .then()の第二引数			Promise	.catch(), .then()の第二引数	
	async/await	.catch(), try-catch try-catch, Promise, async/await	!		async/await	.catch(), try-catch try-catch, Promise, async/await	!

Internationalization
Modules
Net
OS
Path
Performance Hooks
Process
Punycode
Query String
Readline
REPL
Stream
String Decoder
Timers
TLS/SSL
Trace Events
TTY
UDP/Datagram
URL
Utilities
V8
VM
Worker Thread
ZLIB
GitHub Repo & Issue Tracker

Event: 'unhandledRejection'

▶ History

unhandledRejection

The 'unhandledRejection' event is emitted whenever a Promise is rejected and no error handler is attached to the promise within a turn of the event loop. When programming with Promises, exceptions are encapsulated as "rejected promises". Rejections can be caught and handled using `promise.catch()` and are propagated through a Promise chain. The 'unhandledRejection' event is useful for detecting and keeping track of promises that were rejected whose rejections have not yet been handled.

- rejectされたがハンドリングされていないPromiseがあると発生

The listener function is called with the following arguments:

- reason <Error> | <any> The object with which the promise was rejected (typically an `Error` object).
- p the Promise that was rejected.

- APIサーバーなどで放置するとfd制限(ulimit -nの値)の恐れ

```
process.on('unhandledRejection', (reason, p) => {
  console.log(`Unhandled rejection at: ${p} reason: ${reason}`);
  // application specific logging, throwing an error or other logic here
});

somePromise.then((res) => {
  return reportToUser(JSON.parse(res)); // note the typo ('pasre')
}); // no `catch()` or `then()`
```

```
const app = require('express')();
const controller = require('./contoroller'); // もしもrejectが起きたら

// reject後にthenもcatchもないで、接続するとunhandledRejection発生
// クライアント側がtimeoutなどしない限り、リソースを離さずやがてfd制限に
app.get('/', async (req, res, next) => res.send(await controller(
  app.listen(8080);
```

In this example case, it is possible to track the rejection as a developer error as would typically be the case for other 'unhandledRejection' events. To address such failures, a non-operational `.catch(() => {})` handler may be attached to `resource.loaded`, which would prevent the 'unhandledRejection' event from being emitted. Alternatively, the 'rejectionHandled' event may be used.

Internationalization
Modules
Net
OS
Path
Performance Hooks
Process
Punycode
Query String
Readline
REPL
Stream
String Decod
Timers
TLS/SSL
Trace Events
TTY
UDP/Datagra
URL
Utilities
V8
VM
Worker Thre
ZLIB

Event: 'unhandledRejection'

#

expressならエンドポイント毎にcatch漏れを防ぐ wrap関数を挟む事を推奨

The 'unhandledRejection' event is emitted whenever a Promise is rejected and no error handler is attached to the promise within a turn of the event loop. When programming with Promises, exceptions are usually caught either by their own catch clause and handled using try/catch, or are propagated through a Promise chain. The 'unhandledRejection' event is useful for catching cases of promises that were rejected but were never handled.

The listener function is called with the following arguments:

```
const app = require('express')();
const controller = require('./controller');

// 各エンドポイントのハンドラ関数(Promiseで返る前提)にcatchを足す
const wrap = (fn) => {return (req, res, next) => fn(req, res, next).catch((err) => {
    console.error(err);
    res.sendStatus(500);
})};

app.get('/', wrap(async (req, res, next) => res.send(await controller.get(req))));
```

// wrap関数でcatchされた場合の共通エラー処理
// 個別のエラー処理は各ハンドラ関数内でcatch

```
app((err, req, res, next) => {
    console.error(err); // 基本的に意図しないエラーなので、ロギングと
    res.sendStatus(500); // Internal Server Errorだけ返す
});
```

```
app.listen(8080);
```

a non-
operational .catch(() => { }) handler may be attached to resource.loaded, which would prevent the 'unhandledRejection' event from being emitted. Alternatively, the 'rejectionHandled' event may be used.

Internationalization
Modules
Net
OS
Path
Performance Hooks
Process
Punycode
Query Strings
Readline
REPL
Stream
String Decoder
Timers
TLS/SSL
Trace Events
TTY
UDP/Datagram
URL
Utilities
V8
VM
Worker Threads
ZLIB
GitHub Repo & Issue Tracker

Event: 'unhandledRejection'

▶ History

The 'unhandledRejection' event is emitted whenever a `Promise` is rejected and no error handler is attached to the promise within a turn of the event loop. When programming with Promises, exceptions are encapsulated as "rejected promises". Rejections can be caught and handled using `promise.catch()` and are propagated through a `Promise` chain. The 'unhandledRejection' event is useful for detecting and keeping track of promises that were rejected whose rejections have not yet been handled.

The listener function is called with the following arguments:

- `reason`: Error | string | object with which the promise was rejected (typically an error object)
- `p`: the `Promise` that was rejected.

unhandledRejectionを巡る議論

- 発生時にクラッシュさせるべきか否か、コミッタ一間でも意見が割れている
- <https://github.com/nodejs/node/issues/22822>
- 将来的にはクラッシュする可能性(今はflagで選べる案に傾いてそう)

```
process.on('unhandledRejection', (reason, p) => {
  console.log('Unhandled Rejection at: ', p, reason);
  // application specific logging, throwing an error, or other logic here
});

// Example
somePromise.then(res => {
  return reportToUser(JSON.pasre(res)); // note the typo ('pasre')
}); // no .catch() or .then()

// Another example
function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}

const resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

In this example case, it is possible to track the rejection as a developer error as would typically be the case for other 'unhandledRejection' events. To address such failures, a non-operational `.catch(() => {})` handler may be attached to `resource.loaded`, which would prevent the 'unhandledRejection' event from being emitted. Alternatively, the 'rejectionHandled' event may be used.

multipleResolves (v10.12.0 ~)

Event: multipleResolves

Added in: v10.12.0

- type <string> The error type. One of 'resolve' or 'reject'.
 - promise <Promise> The promise that resolved or rejected more than once.
 - value <any> The value with which the promise was either resolved or rejected after the original resolve.

The 'multipleResolves' event is emitted whenever a promise has been either:

- Resolved more than once.
- Rejected more than once.
- Rejected after resolve.
- Resolved after reject.

This is useful for tracking errors in an application while using the promise constructor. Otherwise such mistakes are silently swallowed due to being in a dead zone.

```
const fs = require('fs');
process.on('multipleResolves', (promise, reason) => {
  console.error(promise, reason);
});

new Promise((resolve, reject) => {
  return fs.readFile('nothing', (err, data) => {
    if (err) {
      reject(err);
    }
    return resolve(data);
  });
});
```

Node12系からasyncスタックトレースが強化?!



概要

- Zero-cost [async stack traces](#)というレポートで示唆

Zero-cost [async stack traces](#)

Zero-cost [async stack traces](#)
Attention: Shared Google-externally

Author: bmeurer@, yangguo@

Last Updated: 2018/11/09

Motivation

Solution

Prerequisites

Overview

Alternative Solution

Arbitrary Promise chains

Await Optimization

How to do it

Implementation

How to get it

Async Generation

Error.stack

Error.prepare

Promise.

Shipping

■ bit.ly/v8-zero-cost-async-stack-traces

This document describes a new mechanism to provide (limit!) [async stack traces](#) without adding any additional (runtime) overhead to the v8 engine, which only works as long as the user sticks to [async functions](#) (i.e. it won't help with [Promise](#) only code).

Short link: bit.ly/v8-zero-cost-async-stack-traces

- 7.2系のv8を含むNode+[--async-stack-traces](#)

■ <https://nodejs.org/download/v8-canary/>

- 非同期処理が開始されるとそれまでのスタックトレースは継承されなかつたが、継承されるように

Bug ID: Q:7532 nodejs/node#11365

Motivation
When programming with [async functions](#), which have recently been added to JavaScript with the ES2017 revision, developers are currently facing the problem that the

現在は例外を出した非同期関数の呼び出し元の関数名は表示されない

```
$node app.js
```

Error: 非同期コールバック中に例外発生

```
at module.exports (/tmp/wrongModule.js:6:9)
```

```
./node-v12.0.0-v8-canaryXXX/bin/node --async-stack-traces app.js
```

Error: 非同期コールバック中に例外発生

```
at module.exports (/tmp/wrongModule.js:6:9)
```

```
at async myFn (/tmp/app.js:7:10) # <- 呼び出し元も出るように!!!
```

Error: Let's have a look...

[Code](#)[Issues 1](#)[Pull requests 1](#)[Projects 0](#)[Wiki](#)[Insights](#)

This WG is in the process of being folded into the Diagnostics WG. <https://github.com/nodejs/diagnostics>

[nodejs](#) [node](#)

30 commits

3 branches

0 releases

11 contributors



そんな時は **post-mortem debugging** という手法が使えます 😊

Post Mortem Diagnostics Working Group

The Postmortem Diagnostics working group is dedicated to the support and improvement of postmortem debugging for Node.js. It seeks to elevate the role of postmortem debugging for Node, to assist in the development of techniques and tools, and to make techniques and tools known and available to Node.js users.

Responsibilities include:

1. Defining and adding interfaces/APIs in order to allow dumps to be generated when needed
2. Defining and adding common structures to the dumps generated in order to support tools that want to introspect those dumps

The members of the working group include:

post-mortem debugging

- クラッシュ時のプロセスのメモリーイメージ(=core dump)から指標を得る手法
 - 通常のスタックトレースよりも多くの情報
 - 無限ループ時やV8自体が壊れている時も使える
 - 環境再現の必要はない
- クラッシュしたプロセスを再起動させつつdebugを並行して行える
- gcore(1)/node-reportなどで稼働中プログラムの解析もできる
 - たとえばメモリの使用状況調査

出典: <https://www.slideshare.net/yurikoma/post-mortem-debugging-node-in-prod> (Nov 14, 2015)

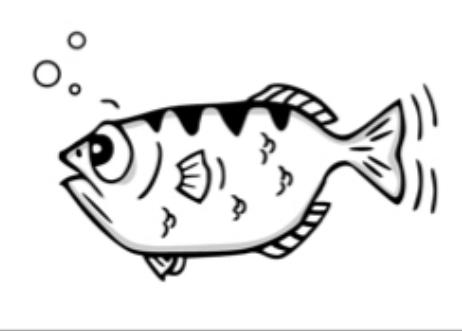
post-mortemの流れと準備

- 0: 準備
 - Core Dump設定: ulimit -c unlimited
 - mac出力先: /cores/core.\$(pid)
 - linux出力先: カレントディレクトリ
 - Node.js実行時--abort-on-uncaught-exception or コード内でprocess.abort()
- 1: Process Crash
- 2A: Process Restart
 - 本番環境再稼働
- 2B: core dumpの解析
 - core fileをオンラインに移し問題を見つけて本番コード改善

出典: <https://www.slideshare.net/juniorongx/Debugging-node-in-prod> (Nov 14, 2014)

The LLDB Debugger

debugger



tool	org	desc
llnode	Node Foundation	Fedor氏作のlldb plugin, npm
node-report	Node Foundation	簡単に軽量なサマリーを取得可能, npm (※前処理不要)
gdb	GNU	
mdb	joyent	- Joyent Production Practices - Postmortem debugging



joyent / mdb_v8





llnode

```
$node --abort-on-uncatched-exception demo.js
$llnode `which node` -c /cores/core.PID # 約1.8GB
(llnode) v8 bt
  * thread #1: tid = 0x0000, 0x0000000100b177e2 \
node`v8::base::OS::Abort() + 18, stop reason = signal SIGSTOP
  * frame #0: 0x0000000100b177e2 node`v8::base::OS::Abort() + 18
...
frame #3: 0x00002629f23042fd <exit> # ここからjs (上はC++領域)
frame #4: 0x00002629f2407a7a <stub>
frame #5: 0x00002629f23bd1d6 cb(this=0x000006e23f5a3da1:\
<Object: Timeout>) at /tmp/demo.js:3:1 fn=0x000006e23f5a3cb9
frame #6: 0x00002629f23bd1d6 ontimeout(this=0x000006e2fa1822d
...
frame #10: 0x00002629f2304101 <entry> # ここまでjs (以下はC++領域)
```

We have nightly test runs against all Node.js active release lines. We also test against Node.js master and Node.js v8-nightly builds to help us identify breaking changes on Node.js and V8 before they land on an active release line.

該当箇所のjsコードやheapの状態、実際に生成された
いたオブジェクトの一覧なども見れます

該当箇所のjsコードを表示

```
(llnode) v8 inspect --source 0x000006e23f5a3cb9
undefinedFunction();
```

• To get started with the llnode commands see the [Commands](#) section.

Node.js Summit San Francisco

1. 非同期処理: 順番に依存しない作りにする
 - 非同期処理の順番は保証されないし不規則
 - トレスにはAsync Hookが使える
2. エラーハンドリング: Promiseで包れば統一的にハンドリングでき、EventEmitter以外のクラスは防げる
 - 但しunhandledRejection/multipleResolvesのdebugは必ずすべき
3. debug: async スタックトレースが強化される見込み
 - 本番環境のdebugにはpost mortemという手法もあるよ

[Code](#)[Issues 56](#)[Pull requests 5](#)[Projects 0](#)[Wiki](#)[Insights](#)

Appendix

Node.js Diagnostics Working Group

nodejs node

CFPに書いたが今回話さなかった話題

- 冒頭のsetTimeout vs Promiseの内部理解
 - Tasks, microtasks, queues and schedules
 - Node.jsではlibuv Eventloop/Node.js microtasks/nexttickだが「タスク間(Eventloopのフェーズ間)でマイクロタスクキューが空になるまで処理」は同じ
 - 拙著 😊 : 内部実装から読み解くNode.js(v11.0.0) Eventloop
- v8 スタックプロファイラ(--prof/--prof-process)を使ったパフォーマンスチューニング
 - 下記公式の解説推奨(v4で5倍up,v10でも2倍up)
 - <https://nodejs.org/en/docs/guides/simple-profiling/>
 - 但し現場ではうまくいかないことが多い。王道なし

Diagnostics Working Group