# Diagnose your Node.js app

2018/11/23

by darai0512

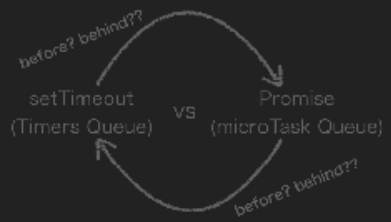
- slide https://darai0512.github.io/nodefest2018/index.html#/
- twitter https://twitter.com/darai_0512

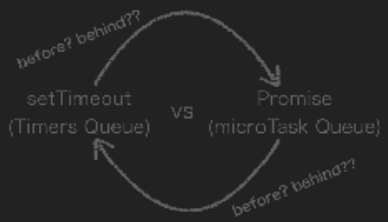
# console.table(me)

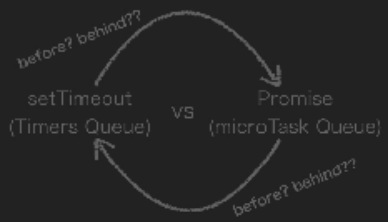
| (index) | Values |
| --- | --- |
| name | darai0512 |
| company | Yahoo! JAPAN |
| work | 広告向けデータ分析システム |
| | 社内言語サポート(Node.js) |
| side jobs | PAY株式会社 |
| | 株式会社ミツモア |

# Node.js初学者の悩み🤔

1. 非同期処理が難しい
   - APIの実行順が直感と違う…
   - 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.catch? 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?



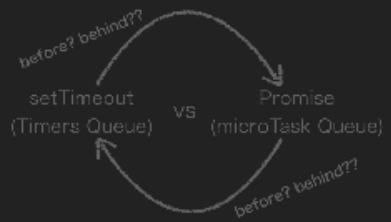


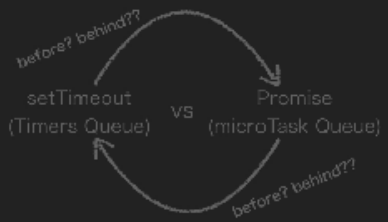


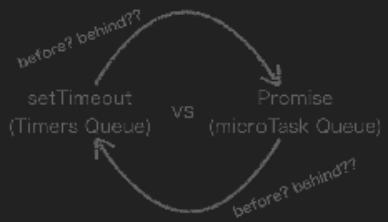

# 非同期処理が難しい

## ex1, 次の出力は?🤔

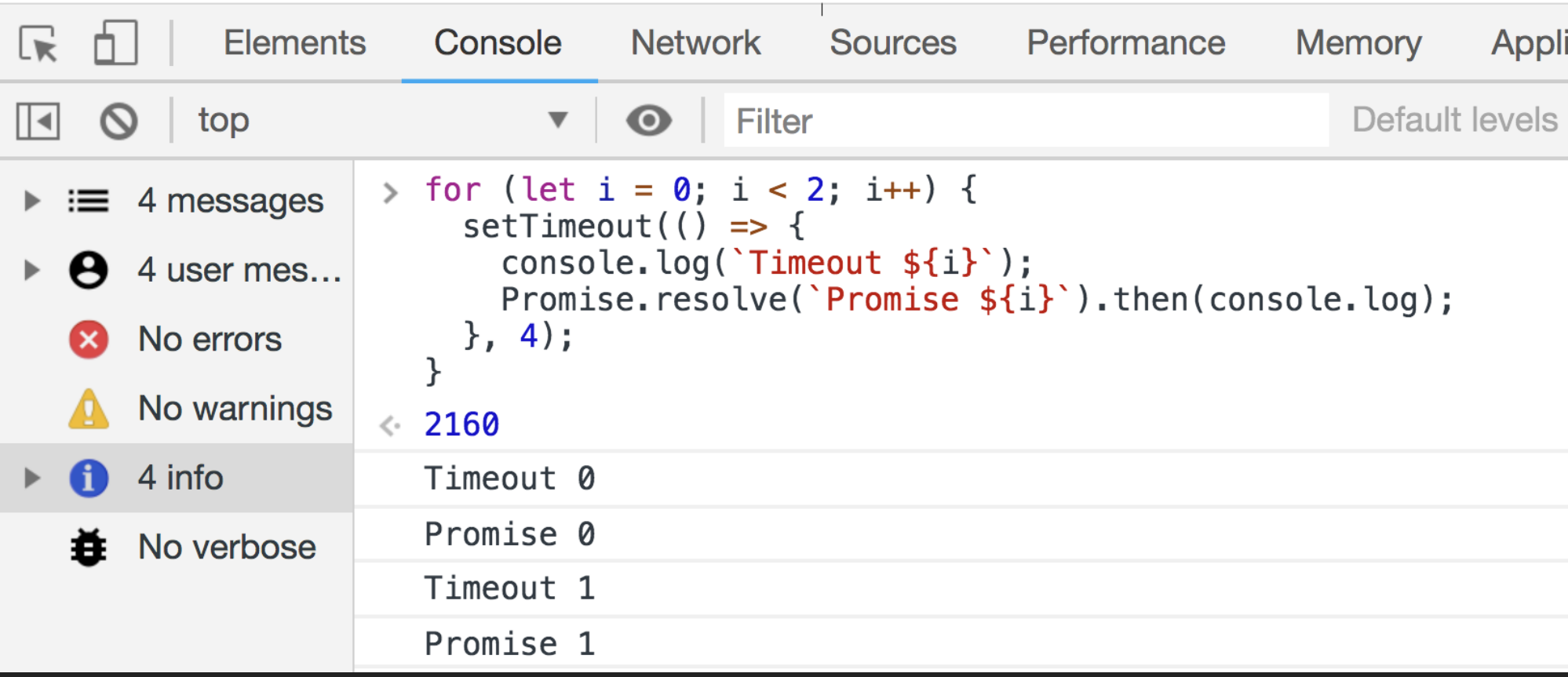
```
for (let i = 0; i < 2; i++) {
  setTimeout(() => {
    console.log(`Timeout ${i}`);
    Promise.resolve(`Promise ${i}`).then(console.log);
  }, 4);
}
```







# ブラウザで実行してみたAさんの答え

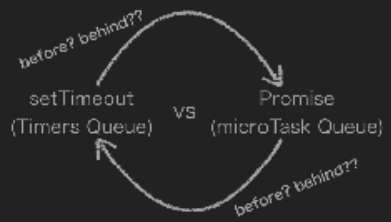


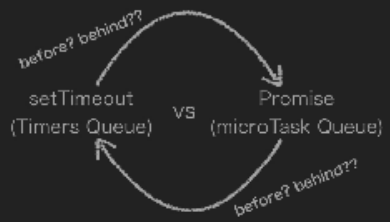
# Node.js v10.xで実行したBさんの答え

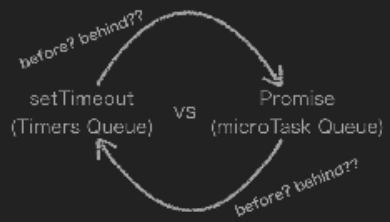
```
[@~/Sites/nodefest]
(^o^)/~$node --version
v10.13.0
[@~/Sites/nodefest]
(^o^)/~$cat demo/async_order.js
for (let i = 0; i < 2; i++) {
  setTimeout(() => {
    console.log(`Timeout ${i}`);
    Promise.resolve(`Promise ${i}`).then(console.log);
  }, 4);
}
[@~/Sites/nodefest]
(^o^)/~$node demo/async_order.js
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
```

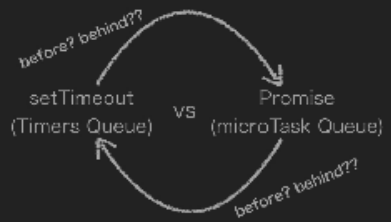


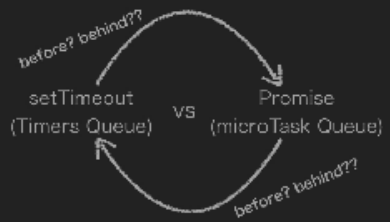


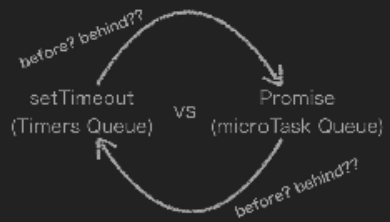



# 非同期処理の実行順は環境やバージョンで異なりうる

- 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}`); // コピ
});
// 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
}
```

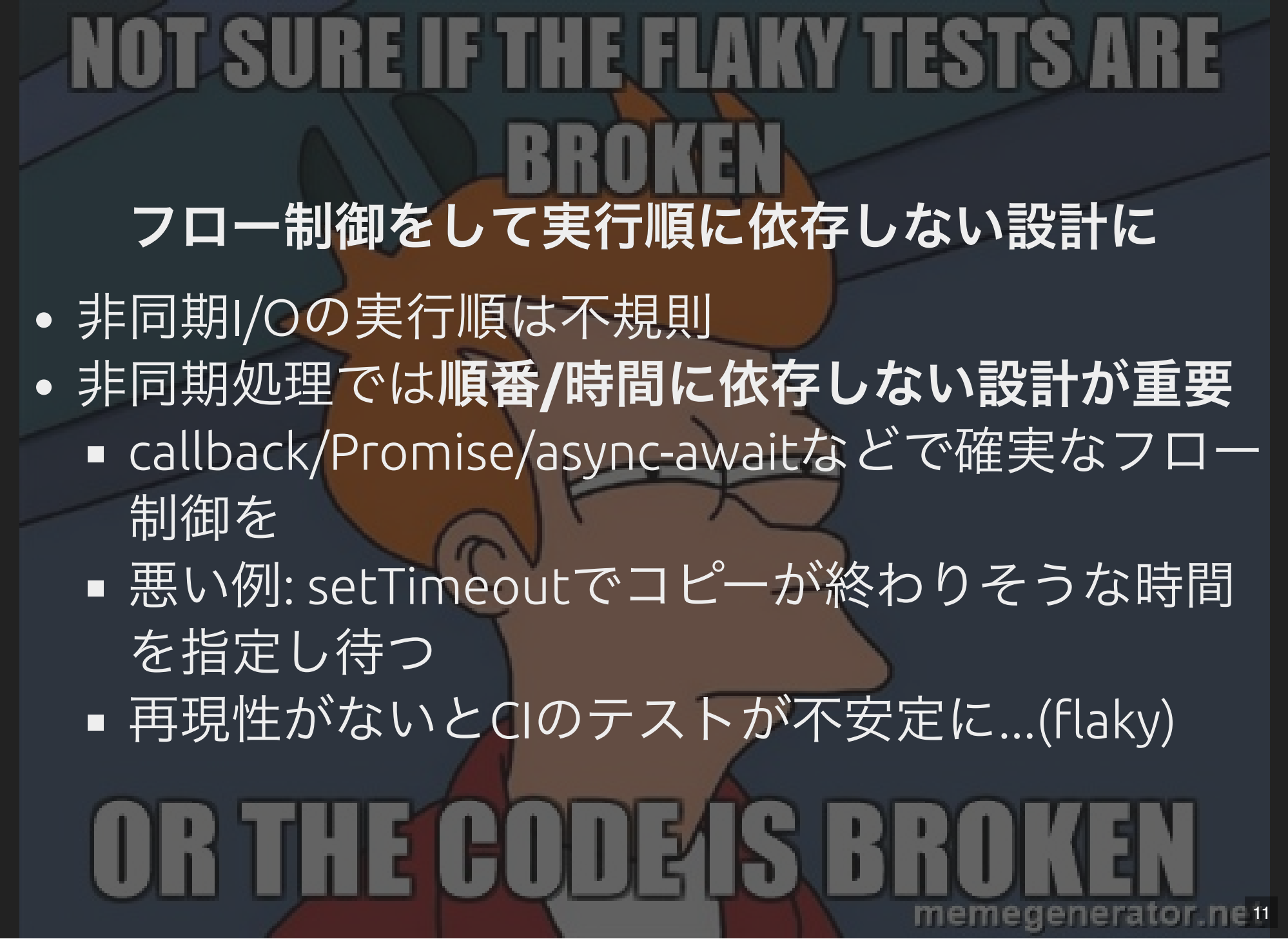
```
$node demo/unknown_order.js 1
(retry ERROR) EEXIST: file already exists, copyfile ... (省略)...
```

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)...
...
```




## 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)

# 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');
```

フロー制御出来てるかはどう確認?🤔

## 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 `Worker`s 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)

- 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()
  - init/destroy: インスタンス生成/破棄時
  - before/after: コールバック前後
  - promiseResolve: resolve時(thenも含む)

```
const ah = require('async_hooks');
const w = (v) => fs.writeSync(process.stdout.fd, v); // 注. ロギング
ah.createHook({
  init(id, type, triggerId, resource) {w(`${type} ${id} created i
  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:

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

## Terminology #

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

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

## Public API #

### Overview #

**ex, 下記コードをトレース**

```
setImmediate(() => {});
console.log();
```

```
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
```

# Async Hooks Tips

- 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

```
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

# エラーハンドリングが難しい🤔

| | type | handling | crash |
|---|---|---|---|
| 非同期 | Callback | if (err) | |
| | EventEmitter | .on('error') | 😱 |
| | Promise | .catch(), .then()の第二引数 | |
| | async/await | .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 |
|---|---|---|---|
| 非同期 | Callback | if (err) | |
| | EventEmitter | .on('error') | 😱 |
| | Promise | .catch(), .then()の第二引数 | |
| | async/await | .catch(), try-catch | |
| 同期 | | 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
  - 潜在的にmemory leakやfd逼迫などの問題を抱えたまま動作(クラッシュ -> 再起動の方がマシかも)
  - core dumpが欲しい時(後述)
- 問題を孕んでいることのサイン
  - unhandledRejection
  - multipleResolves


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

# unhandledRejection

- rejectされたがハンドリングされていないPromiseがあると発生
- APIサーバーなどで放置するとfd制限(ulimit -nの値)の恐れ

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

// reject後にthenもcatchもないので、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);
```

# 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

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

// 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, nex

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

// 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);
```

# unhandledRejectionを巡る議論

- 発生時にクラッシュさせるべきか否か、コミッター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)

# multipleResolves (v10.12.0 ~)

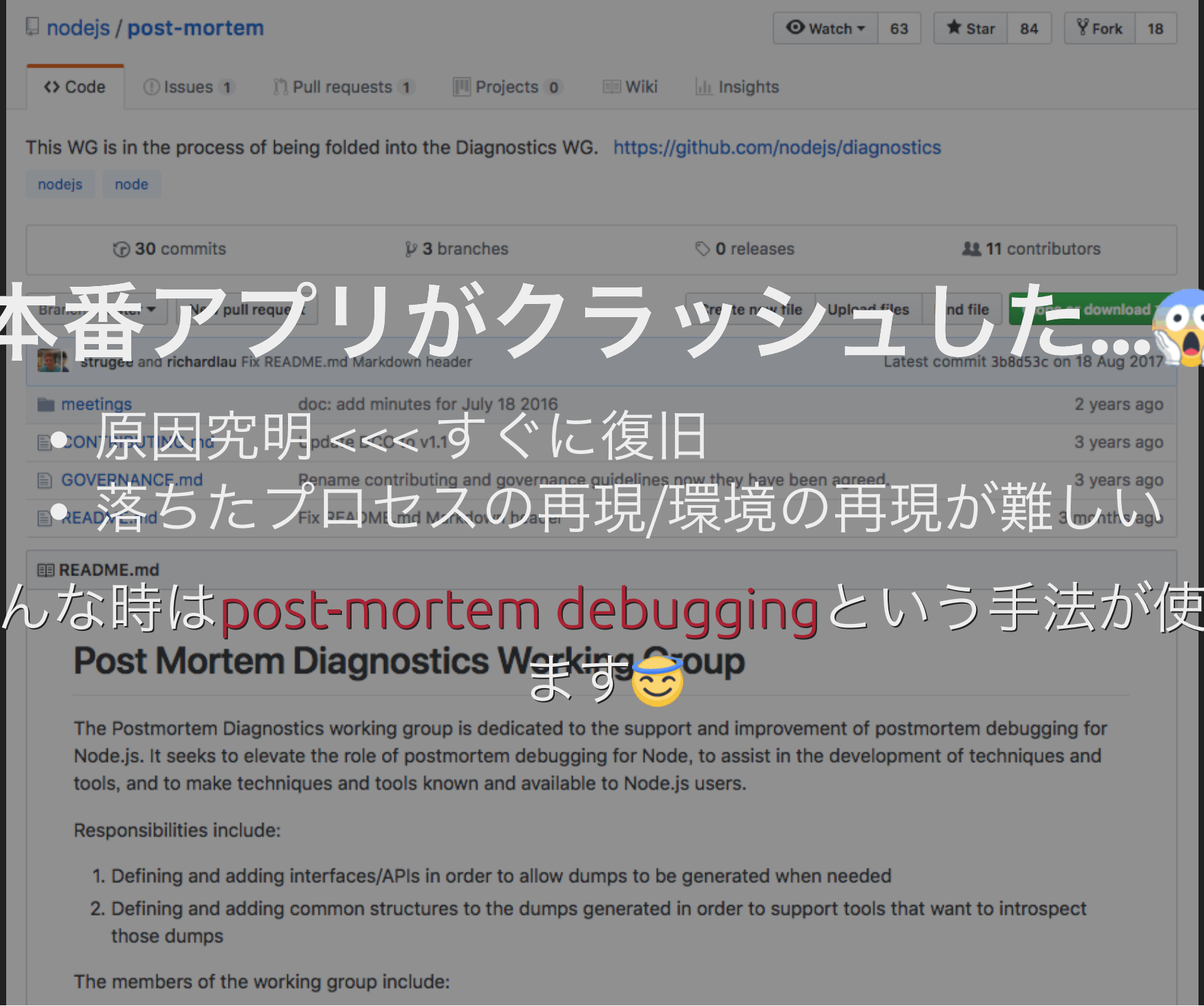
- 複数回resolve/rejectが呼ばれた場合に発火
  - error時のreturn忘れなど
- 議論の流れだとこのケースはクラッシュさせそう

```
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?🤔

- Zero-cost async stack tracesというレポートで示唆
  - 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ったが、継承されるように

```
# 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!!!
```



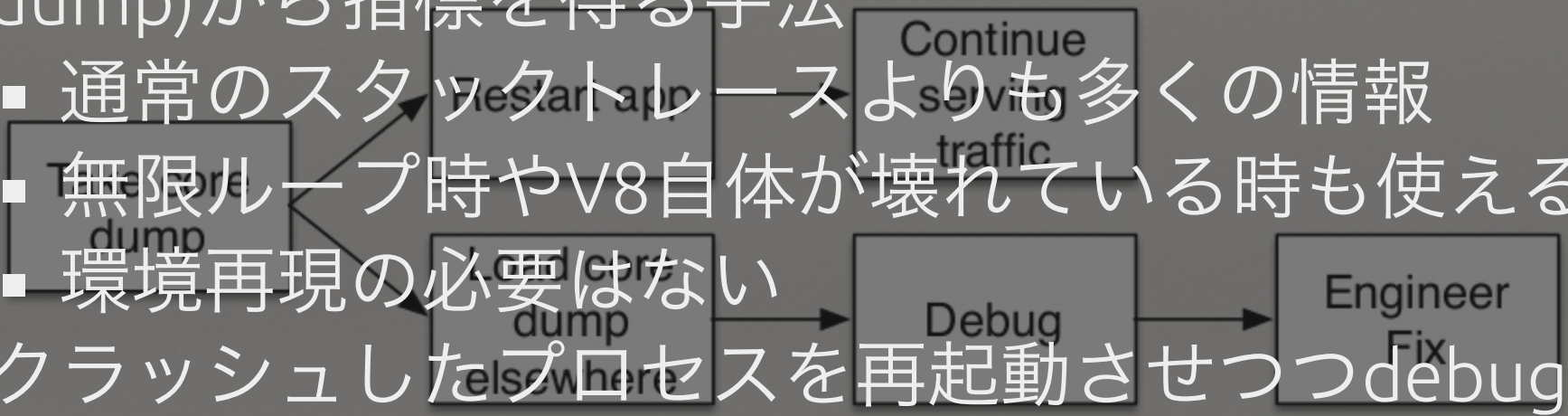

# 本番アプリがクラッシュした…😱

- 原因究明<<<すぐに復旧
- 落ちたプロセスの再現/環境の再現が難しい

そんな時はpost-mortem debuggingという手法が使えます😇

# post-mortem debugging

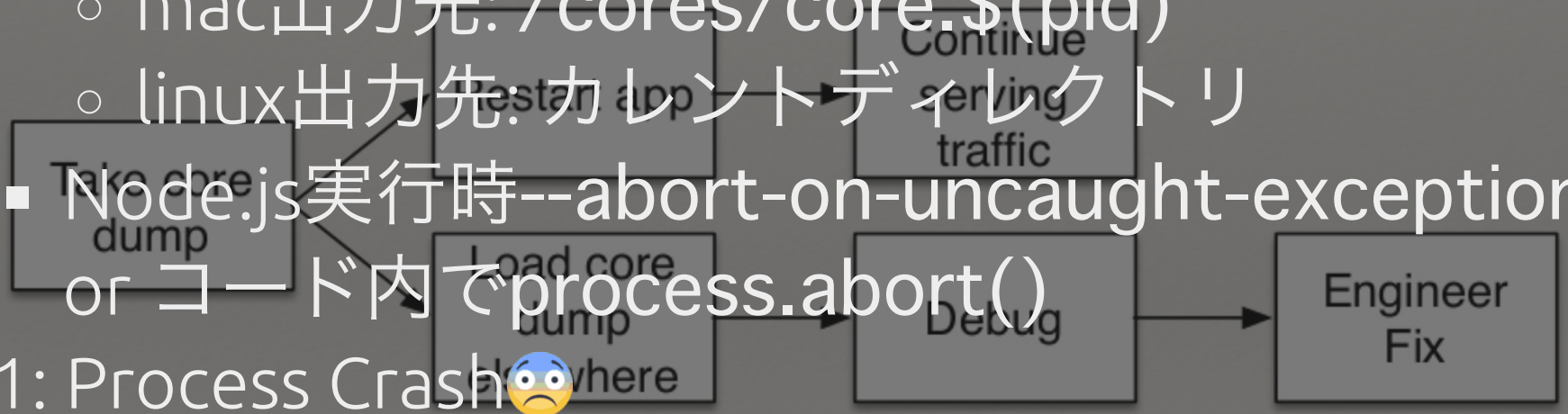
- クラッシュ時のプロセスのメモリーイメージ(=core dump)から指標を得る手法
  - 通常のスタックトレースよりも多くの情報
  - 無限ループ時やV8自体が壊れている時も使える
  - 環境再現の必要はない
- クラッシュしたプロセスを再起動させつつdebugを並行して行える
- gcore(1)/node-reportなどで稼働中プログラムの解析もできる
  - たとえばメモリの使用状況調査




出典: https://www.slideshare.net/yyuu/debugging-node-in-prod (Nov 14, 2015)

# post-mortemの流れと準備

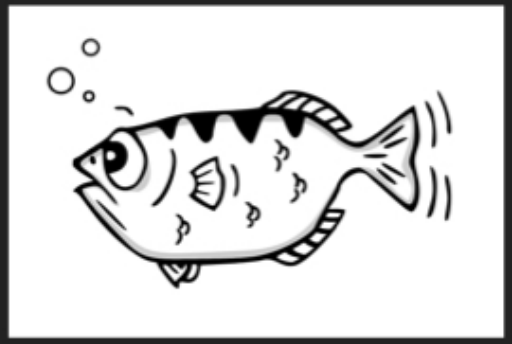
- 0: 準備
  - Core Dump設定: ulimit -c unlimited
    - mac出力先: /cores/core.$(pid)
    - linux出力先: カレントディレクトリ
  - Node.js実行時--abort-on-uncaught-exception or コード内でprocess.abort()
- 1: Process Crash😨
- 2A: Process Restart
  - 本番環境再稼働🎉
- 2B: core dumpの解析
  - core fileをオフラインに移し問題を見つけて本番コード改善🎉




出典: https://www.slideshare.net/yunongx/debugging-node-in-prod (Nov 14, 2014)

The LLDB Debugger

## debugger



| tool | org | desc |
|---|---|---|
| llnode | Node Foundation | Fador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<br>- Postmortem debugging with mdb_v8 |

joyent / mdb_v8

llnode

# ex, llnode

```
$node --abort-on-uncaught-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
```

該当箇所のjsコードやheapの状態、実際に生成されて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();
```

# Summary

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
   - トレースにはAsync Hookが使える
2. エラーハンドリング: Promiseで包めば統一的にハンドリングでき、EventEmitter以外のクラッシュは防げる
   - 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

# Appendix

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も多い。王道なし