

Node 的设计错误

Ryan Dahl
JS Conf 柏林
2018.06

背景:

1. 基于最初的开发，我创建了并管理 Node。
2. 我的主要关注目标是事件驱动的 HTTP server。
3. 这一主要目标对当时服务器端的 JavaScript 起着关键作用。即使在当时这一点不那么明显，但是服务器端 JS 的成功需要事件循环的助力。

背景:

2012 年我离开 Node 项目的时候，当时我觉得 Node（或多或少）实现了我的预定目标：创建一个用户友好的非阻塞框架，具体内容如下

- 1 核心支持许多协议：HTTP，SSL...
2. 在 Windows（使用 IOCP）Linux（EPOLL）和MAC（KQuey）上跨平台工作。
3. 一个比较稳定的相对较小的核心API。
4. 通过 NPM 增加外部模块的生态系统。

但我错了...因为还有许多问题仍待解决.....

使得 Node 保持增长的几项关键工作

1. NPM (Isaac 开发) 将核心节点库解耦并允许生态系统的分布。
2. N-API 是设计精美的绑定API。
3. Ben Noordhuis 和 Bert Belder 构建了 libuv。
4. Mikeal Rogers 组织了管理活动和社区。
5. Fedor Indutny 在代码基础上产生了巨大的影响，尤其是在加密中。
6. 还有许多为 Node 增长作出贡献的人: TJ Fontaine, Rod Vagg, Myles Borins, Nathan Rajlich, Dave Pacheco, Robert Mustacchi, Bryan Cantrill, Igor Zinkovsky, Aria Stewart, Paul Querna, Felix Geisendörfer, Tim Caswell, Guillermo Rauch, Charlie Robbins, Matt Ranney, Rich Trott, Michael Dawson, James Snell

六个月前，我才再次开始使用 Node。

在这期间，我关注的目标发生了变化。

动态语言是科学计算的正确工具，通常你会用它进行快速的一次性计算。

而 JavaScript 就是最好的动态语言。

但是相反，现在我将会抱怨 Node 的所有缺点。

当你是某个项目的负责人时，你总是很难发现其中的错误。

有时 Node 对我来说就像是板上钉钉的事。

它本来可以更好。

遗憾：不遵守“诺言”

- 我在 2009 年 6 月向 Node 添加了 Promise，但在 2010 年 2 月愚蠢地删除了它们。
- Promise 是异步/等待的必要抽象。
- 在 Node 中统一使用 Promise 可能会加快最终标准化和 async/await 的交付。
- 今天的 Node 的许多异步 API 因此而严重老化

遗憾： 安全性

- V8 本身是一个非常好的安全沙箱。
- 如果我对如何维护某些应用程序进行更多的思考，Node 可能会拥有一些其他任何语言都无法获得的安全保证。
- 示例：您的 linter 不应该完全访问您的计算机和网络。

遗憾：构建系统(GYP)

- 构建系统非常困难且非常重要。
- V8（通过Chrome）开始使用GYP，并且我切换了 Node 使之结合。
- 后来，Chrome 放弃了 GN 的 GYP。让 Node 成为唯一的 GYP 用户。
- GYP 不是一个丑陋的内部界面，它暴露给任何试图绑定到 V8 的人。
- 这对用户来说是一个可怕的经历。这是一种披着 Python 外衣的假 JSON。

遗憾：构建系统(GYP)

- GYP 的持续使用可能是 Node 核心最大的故障。
- 我应该提供一个核心的外部功能接口（FFI），而不是指导用户编写 C++ 来绑定到 V8
- 许多人很早就建议搬到FFI（也就是 Cantrill），遗憾的是我忽视了他们的建议。
- （我对 libuv 采用了autotools 表示极度不满）。

遗憾： **package.json**

- Isaac 在 NPM 中发明了 package.json（大部分）。
- 但是我通过允许 Node 的 require() 来检查 package.json 文件的“main”。
- 最后，我在 Node 发布中包含了NPM，这使得它成为事实上的标准。
- 不幸的是，有一个模块化的（私有控制的）存储库。

require("somemodule")不是明确的。

定义的地方太多了。

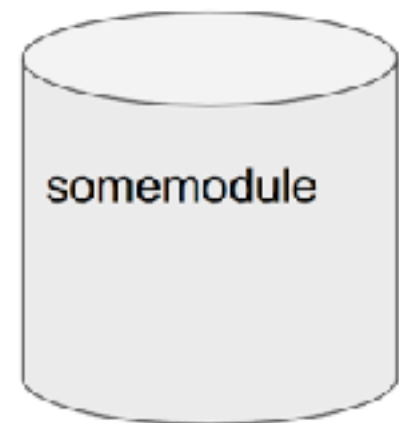
your javascript program

```
require("somemodule")  
  
// Code that uses  
// somemodule
```

package.json

```
{  
  ....  
  "dependencies": {  
    "somemodule": "^0.0.1"  
  }  
  ...  
}
```

NPM's database



Local node_modules folder



遗憾： **package.json**

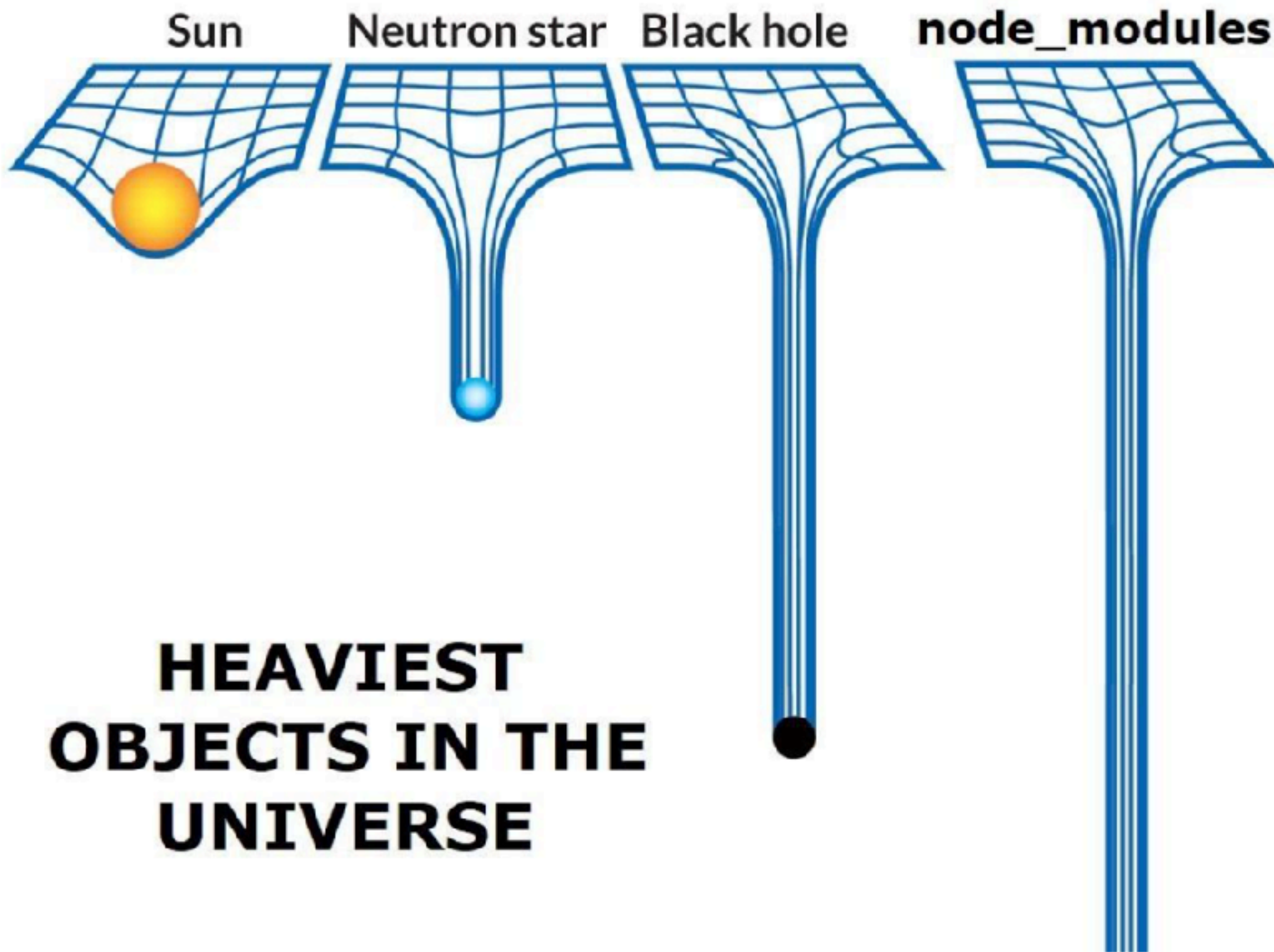
package.json 提出了一个“module”作为文件目录的概念。

这不是一个严格必要的抽象，也不是网络上存在的抽象。

package.json 现在包含了各种不必要的信息。许可证？ 仓库？ 描述？

这是陈词滥调的。

如果在导入时只使用相对文件和 URL，则路径将会定义版本。不需要列出依赖项。



遗憾： `node_modules`

它极大地复杂化了模块分辨率算法。

默认情况下有良好的意图，但在实践中，只使用 `$Node_PATH` 不会排除这一点。

很大程度上偏离了浏览器语义。

这是我的错，我很抱歉。不幸的是现在不可能撤消。

遗憾： `require("module")` 没有扩展名 `".js"`

- 不必要的不明确。
- 不是浏览器 JavaScript 工作方式。不能在脚本标记 `src` 属性中省略`".js"`。
- 模块加载器必须在多个位置查询文件系统，试图猜测用户的意图。

遗憾： `index.js`

我认为它很可爱，因为有 `index.html`。

它不需要复杂的模块加载系统。

在 `require` 支持了 `package.json` 后，它变得特别不必要。

我的 Node 问题几乎完全围绕它如何管理用户代码。

与早期关注的 I/O 相比，模块系统本质上是一种事后考虑。

考虑到这一点，我长期以来一直在思考如何做得更好。

免责声明：我只是提出了一个初具雏形的原型。

除非你急于卷起袖子跳进 `lldb`，否则不要费劲去尝试建造它。

即便如此...

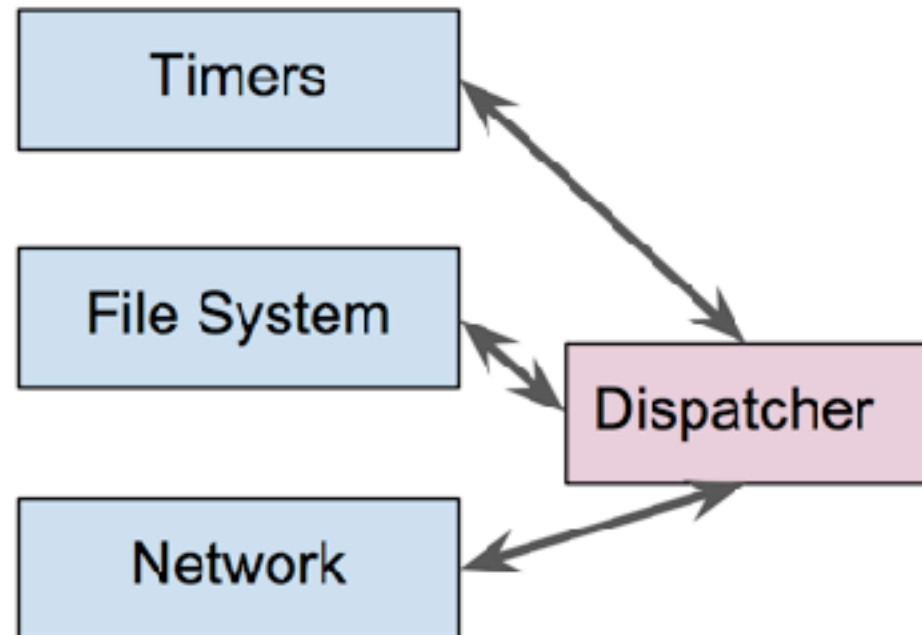
Deno <https://github.com/ry/deno>

V8上一个安全的 TypeScript 运行时

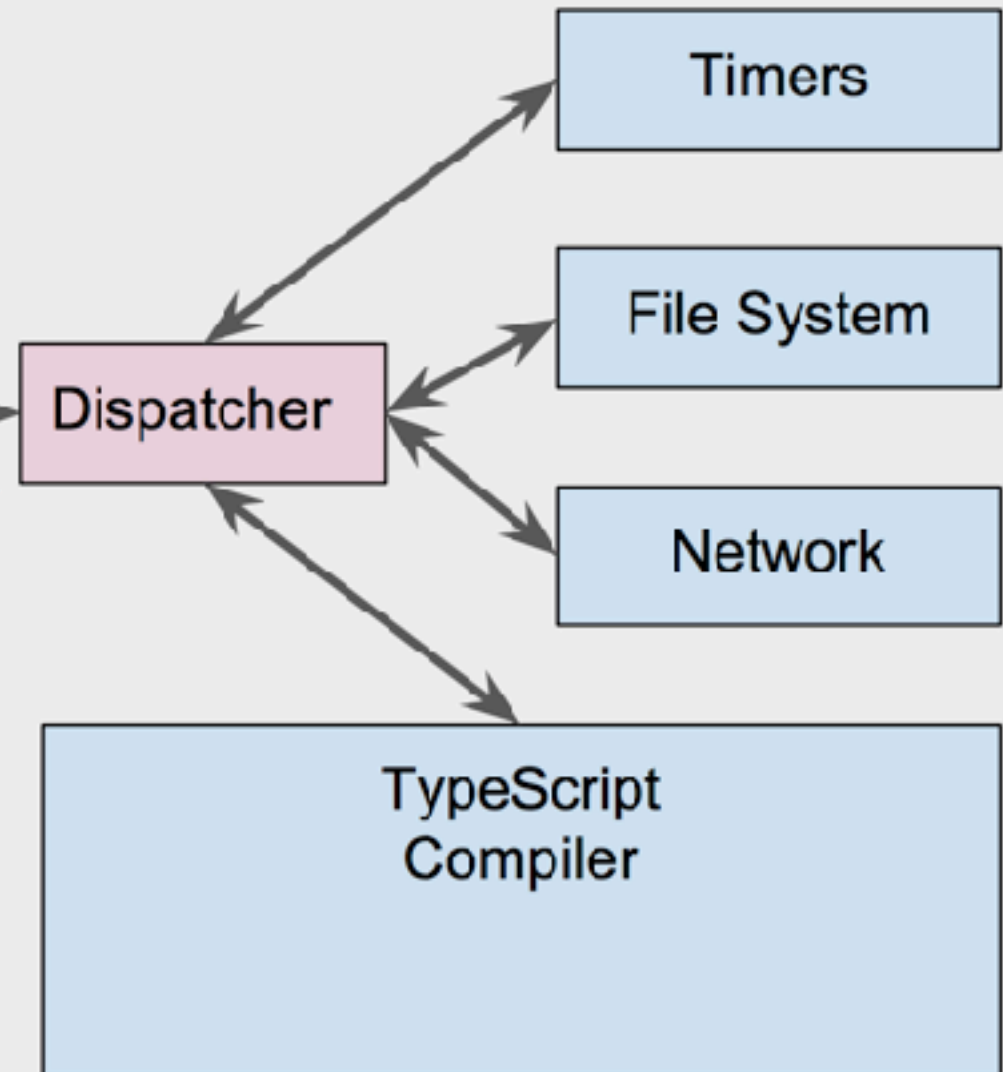
Deno 目标：安全

- 利用 JavaScript 是一个安全沙箱的事实。
 - 默认情况下，脚本应该在没有任何网络或文件系统写访问的情况下运行。
 - 用户可以选择通过标志访问：--allow-net --allow-write
 - 这允许用户运行不可信的实用程序（例如就像一个 linter）
- 不允许任意本地函数绑定到V8中
 - 所有的系统调用都是通过消息传递完成的（原BoFF序列化）
 - 有两个本地函数: send 和 rev。
 - 这既简化了设计，又使系统更易于审核。

Deno Process (Privileged)



V8 VM (Unprivileged)



Protobuf

Deno 目标：简化模块系统

- 没有试图与现有 Node 模块兼容。
- 导入的只是相对的或绝对的URL。（参见语义化版本）

```
import { test } from "https://unpkg.com/deno_testing@0.0.5/testing.ts"  
import { log } from "./util.ts"
```

- 导入时必须提供扩展名。
- 远程 URL 在第一次加载后被无限期地提取和缓存。只有提供了--reload 标志，才能再次获取资源。
- Vendoring 可以通过指定非默认缓存目录来完成。

Deno 目标：内置于可执行文件中的 TS 编译器

- TS 是绝对出色的。
 - 它终于交付了实用的可选语言。
 - 允许代码无缝增长--从快速入侵到大型的，结构良好的机器。
- Deno 挂接到 TS 编译器以执行构建工件的模块解析和增量缓存。
- 未修改的 TS 文件不应重新编译。
- 普通的 JS 也应该可以运行（但这很简单，因为TS是JS的超集）
- 应该使用 V8 快照来快速启动（尚未在原型中）

Deno 目标：以最少的链接发送一个可执行文件

```
> ls -lh deno
```

```
-rwxrwxr-x 1 ryan ryan 55M May 28 23:46 deno
```

```
> ldd deno
```

```
linux-vdso.so.1 => (0x00007ffc6797a000)
```

```
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f104fa47000)
```

```
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f104f6c5000)
```

```
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f104f3bc000)
```

```
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f104f1a6000)
```

```
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f104eddc000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x00007f104fc64000)
```

Deno 目标：利用 2018 年

- 通过将带有 Parcel 的 Node 模块编译为捆绑包来引导运行时（这是对Node必须做的大量简化）。
- 卓越的基础架构现在以本机代码存在：
 - EG 不需要担心 HTTP。其他人已经使其工作（Node中的情况并非如此.. web 服务器100%手动滚动）。
 - 目前 Deno 的非 JS 部分正在使用 Go，但我并未完全销售和研究，由于现在已经创造了原型。
 - Rust 可能是一个不错的选择。
 - 如果允许其他人针对 Go 或 Rust构建他们自己的 Denos，C ++可能仍然是一个不错的选择？

Deno 目标：杂项

- 发生未捕捉 Promise 错误时立刻自动终止运行（疯狂的是在 Node 中并非如此）
- 支持 top-level 的 await（尚未在原型中）
- 兼容浏览器（功能重叠时）

Deno : <https://github.com/ry/deno>

它诞生只有一个月的时间。 它的实用性还不是很强。

但我对迄今为止的设计感到满意。

评论？ 问题？ 关注？ 请联系：

ry@tinyclouds.org

幻灯片: <http://tinyclouds.org/jsconf2018.pdf>