Deno 1.0 发布，与 Node 相比的 10 大优缺点

作者：Vanessa

原文链接：https://ld246.com/article/1591867118642

来源网站：链滴

2020-06-11 & 2020-06-12
<h3 id="背景">背景</h3>
动态语言的编写让开发者不必担心内存管理或系统构建之类的细节，可以快速简洁地将复杂的系链接在一起。JavaScript 做为使用最广泛的动态语言，在近几年又通过了 ECMA 标准组织进行了持严谨的改进。我们有理由相信无论在浏览器环境中还是作为独立进程，JavaScript 都将是动态语言中自然选择。
在该领域中，Node.js 被证明是一个非常成功的软件平台。但由于 Node 早在 2009 年就已诞生当时的 JavaScript 还非常混乱。为此，Node 不得不发明一些概念，虽然这些概念后来被标准组织采并以不同的形式添加到语言中，但目前我们依旧可以看到作者 Ryan Dahl 对 Node.js 遗憾的 10 件情
<a href="https://www.youtube.com/watch?v=M3BM9TB-8yA">Design Mistakes in Node/a>
随着 JavaScript 语言的不断发展及 TypeScript 的新特性，构建 Node 项目会变得非常艰巨，再上 NPM 的机制不符合 Web 特性这些诸多原由。我们认为 JavaScript 平台及周围软件基础架构所发的变化足以让我们去寻求一种简单有趣且高效的脚本环境。
<h3 id="简介">简介</h3>
Deno 为 JavaScript 和 TypeScript 提供了一种简单、现代化和运行时安全的环境，他使用 V8 使用 Rust 构建。

默认启用安全。除非明确启用，否则没有文件，网络或环境访问权限。
默认支持 TypeScript。
仅发送一个可执行文件。
具有内置的工具套件，如依赖检查器（deno info）和代码格式化程序（deno fmt）。
拥有一组进过审核的标准模块（deno.land/std）以确保 Deno 以工作

<h3 id="特性">特性</h3>
<h4 id="浏览器端的命令行脚本">浏览器端的命令行脚本</h4>
Deno 是一个新的可以在 web 浏览器以外执行 JavaScript 和 TypeScript 的运行环境。
Deno 提供了一个独立的工具，可以让开发者快速编写功能复杂的脚本。 Deno 始终为一个单独可执行文件。就像 web 浏览器一样，他知道如何获取外部代码。在 Deno 中，单个文件在无需其他何工具的基础上可以定义复杂的行为。如下列代码，不需要任何配置文件和安装即可启动一个服务：
<pre><code class="language-ts highlight-chroma">import {</san> serve } from "https://deno.land/std@0.50.0http/server.ts";

for await (</pan>const req <spn class="highlight-k">of serve({ port: 800 })) {
  req.respond({ body: "Hello World\n" <pan class="highlight-p">});
}
</code></pre>

和浏览器一样，默认情况下，代码在安全的沙箱中执行。 未经允许，脚本无法访问磁盘，打开络连接或进行其他潜在的恶意操作。浏览器提供的用于访问相机和麦克风的 API 需要用户进行授权。eno 在终端中也提供了类似的行为。以上示例如果没有提供 <code>--allow-net</code> 命令行参的话，将会执行失败。
Deno 会遵循浏览器 JavaScript API 的标准。当然，并不是每个浏览器 API 都与 Deno 相关，无论如何，Deno 都不会偏离标准。
<h4 id="TypeScript-支持">TypeScript 支持</h4>
Deno 适用于各种领域：从小型的单行脚本到复杂的服务器端业务逻辑。随着程序复杂度的增加类型检查会变得越来越重要。TypeScript 对 JavaScript 语言做了扩展，他允许用户提供类型信息。
Deno 不需要其他额外的工具即可支持 TypeScript。Deno 的标准模块全部使用 TypeScript 编。
<h4 id="底层-Promises">底层 Promises</h4>
Node 在 JavaScript 有 Promises 或 async/await 概念之前就设计了。Node 与之相对应的是念为 EventEmitter，他基于重要的 API，即 sockets 和 HTTP。除了 async/await 用户友好的优势置外，EventEmitter 模式存在 back-pressure 问题。以 TCP socket 为例，socket 在接受到传入的据包时将提交 "data" 事件。这些 "data" 的回调将以不受限的方式被提交，从而让整个过程中的事件出控制。由于 TCP socket 不存在 back-pressure，远程发送方就无法知道服务器已超负荷将继续发数据，因此 Node 将继续接收新的数据事件。为了缓解这个问题，添加了 <code>pause()</code> 方法。这可以解决问题，但是需要额外的代码。
在 Deno 中，sockets 仍然是异步的，但接收新数据需要用户使用 <code>read()</code>。正构造接收 socket 不需要额外的暂停语义。这对 TCP sockets 来说并不是唯一的。系统中最低的绑定从根本上与 pomises 相关 - 我们称这些绑定为 “ops”。 Deno 中以某种形式出现的所有回调均来自 promises。
Rust 自己有类似于 promise 的抽象，称为 Futures。通过 “op” 抽象，Deno 可以很容易的定 Rust future-based API 到 JavaScript promise 中。
<h4 id="Rust-APIs">Rust APIs</h4>
我们提供的主要组件是 Deno 命令行界面（CLI）。CLI 的版本目前为 1.0。但 Deno 并不是一独立的程序，他以 Rust 架构进行设计，可以允许在不同的层级进行集成。
deno_core 是 Deno 非常核心的版本。他不依于 TypeScript 或 Tokio。他只是提供了我们的 Op 和资源的基础架构。也是说，他提供了一种组织方式，将 Rust 特性绑定到 JavaScript promises 中。CLI 当然也完全建立在deno_core 之上。
rusty_v8 为 V8's C++ API 提供高质量的 Rust 绑。 该 API 会尽可能匹配原生的 C++ API。这是零消耗的绑定 - Rust 中公开的对象与你在 C++ 中操的对象完全相同。rusty_v8 虽然提供了使用 Github Actions CI 构建的二进制文件，但他也允许用户整配置项后自行编译。所有 V8 的源代码都可以进行自我分发。最后，rusty_v8 尝试成为一个安全接。他虽然还不是 100% 的安全，但已经非常接近了。能够以安全的方式与像 V8 这样复杂的 VM 进交互非常令人惊讶，除此外，他还让我们发现了 Deno 中许多困难的错误。
<h4 id="稳定性">稳定性</h4>
我们承诺在 Deno 中维护稳定的 API。Deno 有很多接口和组件，因此对于“稳定”来说就非常要。Deno 内部与操作系统交互的 JavaScript API 都以 “Deno” 做为命名空间（例如 <code>Deno open()</code>）。这些已经过仔细的检查，我们不会对他们进行向后的不兼容修改。
所有没有出现在稳定版中的功能都隐藏在了命令行中 <code>--unstable</code> 的参数下。有不稳定的接口都在 lib.deno.unstable.d.ts 中。在续的版本中，其中一些 API 将会出现在稳定版中。
在全局名称空间中，可以找到所有种类的其他对象（如 <code>setTimeout()</code> 和 <code>fetch()</code>）。我们竭尽全力使这交互与浏览器中的保持一致；如果发现不兼容，会提 issue 行更正。因为定义这些接口的是浏览器标准，而不是 Deno。但我们发布的所有更正均是错误修复，

不是接口修改。如果和浏览器标准的 API 不兼容，将会在主要版本之前进行修复。
Deno 有许多 Rust APIs，即 deno_core 和 rusty_v8。这些 API 都不是 1.0。我们会持续对他进行迭代。
<h4 id="局限性">局限性</h4>
最重要的是你需要知道 Deno 不是 Node 的分支 - 他是一个全新的实现。Deno 从诞生至今仅两年的时间，而 Node 的开发已超过十年。基于对 Deno 的情感，我们相信他会日趋发展和成熟的。/p>
对于一些应用程序而言 Deno 可能是目前不错的选择，但对于另外一些应用而言则为时尚早。这取决于程序的需求。我们会保持这些局限性的透明性，以帮助人们在考虑是否使用 Deno 时做出正确选择。
<h4 id="兼容性">兼容性</h4>
很不幸，目前许多用户发现 Deno 与 JavaScript 工具的兼容性令人沮丧。主要是 Deno 与 Nod（NPM）的软件包不兼容。目前在 https://deno.land/std/node/ 上建立了一个初步的兼容性层，但还远远未完成。
尽管 Deno 采用强硬方法简化了模块系统，但 Deno 和 Node 都是非常相似且有着相同目标的统。随着时间的推移，我们希望 Deno 能够开箱即用地运行越来越多的 Node 程序。
<h4 id="HTTP-服务器性能">HTTP 服务器性能</h4>
我们不断对 Deno HTTP 服务器的性能进行了跟踪。一个 hello-world 的 Deno HTTP 服务器每秒可处理约 25k 个请求，其中最大延迟为 1.3 毫秒。相比 Node 程序每秒处理 34k 个请求来说，其最大延迟介于 2 到 300 毫秒之间。
Deno HTTP 服务器使用 TypeScript 对顶层原生 TCP sockets 进行了实现。Node HTTP 服务使用 C 语言编写，将其高层暴露给 JavaScript 进行绑定。我们一直拒绝将原生的 HTTP 服务器绑定加到 Deno 中，因为我们要优化 TCP socket 层和更通用的 op 接口。
Deno 是一个合适的异步服务器，每秒 25k 的请求足以满足大多数需求。如果不能满足的话，那 JavaScript 可能就不是最佳的选择。此外，由于普遍的使用了 Promise，我们相信 Deno 能表现出好的尾部效应。 综上所述，我们相信该系统还有更多的性能优势可做，我们希望在将来的版本中实现一目标。
<h4 id="TypeScript-编译瓶颈">TypeScript 编译瓶颈</h4>
在内部，Deno 使用 Microsoft 的 TypeScript 编译器来检查类型并生成 JavaScript。这与 V8 析 JavaScript 所花费的时间相比，他非常慢。在项目的早期，我们希望 “V8 快照” 在此能够带来大改进。快照肯定有一定的帮助，但是他依然慢。我们认为可以在现有的 TypeScript 编译器基础上行一些改进，但这需要在 Rust 中实现类型检查。这将是一项艰巨的任务，不会很快执行；但他可以开发人员的关键路径上提供数量级的性能改进。 TSC 必须移植到 Rust 中。如果您有兴趣合作解决此题，请与我们联系。
<h4 id="插件和扩展">插件和扩展</h4>
我们有一个全新的插件系统，可通过自定义操作来扩展运行时的 Deno。 但该接口仍在开发中且标记为不稳定。因此，除了 Deno 提供的功能之外，访问本机系统非常困难。
<h4 id="其他">其他</h4>

可以访问浏览器 API（Fetch, Window）
可以在顶级使用 <code>await</code>，不需要将其包装在一个异步函数中

<h3 id="返回总目录">返回总目录</h3>
每天 30 秒系列之前端资
<h3 id="摘自">摘自</h3>
Deno 1.0