Introduction

Deno is a JavaScript/TypeScript runtime with secure defaults and a great developer experience.

It's built on V8, Rust, and Tokio.

Feature Highlights

- Secure by default. No file, network, or environment access (unless explicitly enabled).
- Supports TypeScript out of the box.
- Ships a single executable (`deno`).
- Has built-in utilities like a dependency inspector (`deno info`) and a code formatter (`deno fmt`).
- Has a set of reviewed (audited) standard modules that are guaranteed to work with Deno.
- Scripts can be bundled into a single JavaScript file.

Philosophy

Deno aims to be a productive and secure scripting environment for the modern programmer.

Deno will always be distributed as a single executable. Given a URL to a Deno program, it is runnable with nothing more than the ~15 megabyte zipped executable. Deno explicitly takes on the role of both runtime and package manager. It uses a standard browser-compatible protocol for loading modules: URLs.

Among other things, Deno is a great replacement for utility scripts that may have been historically written with bash or python.

Goals

- Only ship a single executable (`deno`).
- Provide Secure Defaults
  - Unless specifically allowed, scripts can't access files, the environment, or the network.
- Browser compatible: The subset of Deno programs which are written completely in JavaScript and do not use the global `Deno` namespace (or feature test for it), ought to also be able to be run in a modern web browser without change.
- Provide built-in tooling like unit testing, code formatting, and linting to improve developer experience.
- Does not leak V8 concepts into user land.
- Be able to serve HTTP efficiently

Comparison to Node.js

- Deno does not use `npm`

  - It uses modules referenced as URLs or file paths

- Deno does not use `package.json` in its module resolution algorithm.

- All async actions in Deno return a promise. Thus Deno provides different APIs than Node.

- Deno requires explicit permissions for file, network, and environment access.

- Deno always dies on uncaught errors.
- Uses "ES Modules" and does not support `require()`. Third party modules are imported via URLs:

```
import * as log from "https://deno.land/
  ↪ std@$STD_VERSION/log/mod.ts";
```

Other key behaviors

- Remote code is fetched and cached on first execution, and never updated until the code is run with the `--reload` flag. (So, this will still work on an airplane.)
- Modules/files loaded from remote URLs are intended to be immutable and cacheable.

Getting Started

In this chapter we'll discuss:

- Installing Deno
- Setting up your environment
- Running a `Hello World` script
- Writing our own script
- Command line interface
- Understanding permissions
- Using Deno with TypeScript
- Using WebAssembly

Installation

Deno works on macOS, Linux, and Windows. Deno is a single binary executable. It has no external dependencies.

Download and install

deno_install provides convenience scripts to download and install the binary.

Using Shell (macOS and Linux):

```
curl -fsSL https://deno.land/x/install/install.sh | sh
```

Using PowerShell (Windows):

```
iwr https://deno.land/x/install/install.ps1 -useb | iex
```

Using Scoop (Windows):

```
scoop install deno
```

Using Chocolatey (Windows):

```
choco install deno
```

Using Homebrew (macOS):

```
brew install deno
```

Using Cargo (Windows, macOS, Linux):

```
cargo install deno
```

Deno binaries can also be installed manually, by downloading a zip file at github.com/denoland/deno/releases. These packages contain just a single executable file. You will have to set the executable bit on macOS and Linux.

Testing your installation

To test your installation, run `deno --version`. If this prints the Deno version to the console the installation was successful.

Use `deno help` to see help text documenting Deno's flags and usage. Get a detailed guide on the CLI here.

Updating

To update a previously installed version of Deno, you can run:

```
deno upgrade
```

This will fetch the latest release from github.com/denoland/deno/releases, unzip it, and replace your current executable with it.

You can also use this utility to install a specific version of Deno:

```
deno upgrade --version 1.0.1
```

Building from source

Information about how to build from source can be found in the `Contributing` chapter.

Set up your environment

To productively get going with Deno you should set up your environment. This means setting up shell autocomplete, environmental variables and your editor or IDE of choice.

Environmental variables

There are several env vars that control how Deno behaves:

`DENO_DIR` defaults to `$HOME/.cache/deno` but can be set to any path to control where generated and cached source code is written and read to.

`NO_COLOR` will turn off color output if set. See https://no-color.org/. User code can test if `NO_COLOR` was set without having `--allow-env` by using the boolean constant `Deno.noColor`.

Shell autocomplete

You can generate completion script for your shell using the `deno ↪ completions <shell>` command. The command outputs to stdout so you should redirect it to an appropriate file.

The supported shells are:

- zsh
- bash
- fish
- powershell
- elvish

Example (bash):

```
deno completions bash > /usr/local/etc/bash_completion.d
  ↪ /deno.bash
source /usr/local/etc/bash_completion.d/deno.bash
```

Example (zsh without framework):

```
mkdir ~/.zsh # create a folder to save your completions.
  ↪  it can be anywhere
```

```
deno completions zsh > ~/.zsh/_deno
```

then add this to your `.zshrc`

```
fpath=(~/.zsh $fpath)
autoload -Uz compinit
compinit -u
```

and restart your terminal. note that if completions are still not loading, you may need to run `rm ~/.zcompdump/` to remove previously generated completions and then `compinit` to generate them again.

Example (zsh + oh-my-zsh) [recommended for zsh users] :

```
mkdir ~/.oh-my-zsh/custom/plugins/deno
deno completions zsh > ~/.oh-my-zsh/custom/plugins/deno/
  ↪ _deno
```

After this add deno plugin under plugins tag in `~/.zshrc` file. for tools like `antigen` path will be `~/.antigen/bundles/robbyrussell/oh-my-zsh/`
↪ `plugins` and command will be `antigen bundle deno` and so on.

Editors and IDEs

Because Deno requires the use of file extensions for module imports and allows http imports, and most editors and language servers do not natively support this at the moment, many editors will throw errors about being unable to find files or imports having unnecessary file extensions.

The community has developed extensions for some editors to solve these issues:

VS Code The beta version of vscode_deno is published on the Visual Studio Marketplace. Please report any issues.

JetBrains IDEs Support for JetBrains IDEs is available through the Deno plugin.

For more information on how to set-up your JetBrains IDE for Deno, read this comment on YouTrack.

Vim and NeoVim Vim works fairly well for Deno/TypeScript if you install CoC (intellisense engine and language server protocol).

After CoC is installed, from inside Vim, run:`CocInstall coc-tsserver` and `:CocInstall coc-deno`. To get autocompletion working for Deno type definitions run `:CocCommand deno.types`. Optionally restart the CoC server `:CocRestart`. From now on, things like `gd` (go to definition) and `gr` (goto/find references) should work.

Emacs Emacs works pretty well for a TypeScript project targeted to Deno by using a combination of tide which is the canonical way of using TypeScript within Emacs and typescript-deno-plugin which is what is used by the official VSCode extension for Deno.

To use it, first make sure that `tide` is setup for your instance of Emacs. Next, as instructed on the typescript-deno-plugin page, first `npm install --save-dev typescript-deno-plugin typescript` in your project (`npm init -y` as necessary), then add the following block to your `tsconfig.json` and you are off to the races!

```
{
  "compilerOptions": {
    "plugins": [
      {
        "name": "typescript-deno-plugin",
        "enable": true, // default is `true`
        "importmap": "import_map.json"
      }
    ]
  }
}
```

If you don't see your favorite IDE on this list, maybe you can develop an extension. Our community Discord group can give you some pointers on where to get started.

First steps

This page contains some examples to teach you about the fundamentals of Deno.

This document assumes that you have some prior knowledge of JavaScript, especially about `async`/`await`. If you have no prior knowledge of JavaScript, you might want to follow a guide on the basics of JavaScript before attempting to start with Deno.

Hello World

Deno is a runtime for JavaScript/TypeScript which tries to be web compatible and use modern features wherever possible.

Browser compatibility means a `Hello World` program in Deno is the same as the one you can run in the browser:

```
console.log("Welcome to Deno  ");
```

Try the program:

```
deno run https://deno.land/std@$STD_VERSION/examples/
  ↪ welcome.ts
```

Making an HTTP request

Many programs use HTTP requests to fetch data from a webserver. Let's write a small program that fetches a file and prints its contents out to the terminal.

Just like in the browser you can use the web standard `fetch` API to make HTTP calls:

```
const url = Deno.args[0];
const res = await fetch(url);

const body = new Uint8Array(await res.arrayBuffer());
await Deno.stdout.write(body);
```

Let's walk through what this application does:

1. We get the first argument passed to the application, and store it in the `url` constant.
2. We make a request to the url specified, await the response, and store it in the `res` constant.
3. We parse the response body as an `ArrayBuffer`, await the response, and convert it into a `Uint8Array` to store in the `body` constant.
4. We write the contents of the `body` constant to `stdout`.

Try it out:

```
deno run https://deno.land/std@$STD_VERSION/examples/
  ↪ curl.ts https://example.com
```

You will see this program returns an error regarding network access, so what did we do wrong? You might remember from the introduction that Deno is a runtime which is secure by default. This means you need to explicitly give programs the permission to do certain ‘privileged’ actions, such as access the network.

Try it out again with the correct permission flag:

```
deno run --allow-net=example.com https://deno.land/
  ↪ std@$STD_VERSION/examples/curl.ts https://example.
  ↪ com
```

Reading a file

Deno also provides APIs which do not come from the web. These are all contained in the `Deno` global. You can find documentation for these APIs on doc.deno.land.

Filesystem APIs for example do not have a web standard form, so Deno provides its own API.

In this program each command-line argument is assumed to be a filename, the file is opened, and printed to stdout.

```
const filenames = Deno.args;
for (const filename of filenames) {
  const file = await Deno.open(filename);
  await Deno.copy(file, Deno.stdout);
  file.close();
}
```

The `copy()` function here actually makes no more than the necessary kernel→userspace→kernel copies. That is, the same memory from which data is read from the file, is written to stdout. This illustrates a general design goal for I/O streams in Deno.

Try the program:

```
deno run --allow-read https://deno.land/std@$STD_VERSION
  ↪ /examples/cat.ts /etc/passwd
```

TCP server

This is an example of a server which accepts connections on port 8080, and returns to the client anything it sends.

```
const hostname = "0.0.0.0";
const port = 8080;
const listener = Deno.listen({ hostname, port });
console.log(`Listening on ${hostname}:${port}`);
for await (const conn of listener) {
  Deno.copy(conn, conn);
}
```

For security reasons, Deno does not allow programs to access the network without explicit permission. To allow accessing the network, use a command-line flag:

```
deno run --allow-net https://deno.land/std@$STD_VERSION/
  ↪ examples/echo_server.ts
```

To test it, try sending data to it with netcat:

```
$ nc localhost 8080
hello world
hello world
```

Like the `cat.ts` example, the `copy()` function here also does not make unnecessary memory copies. It receives a packet from the kernel and sends it back, without further complexity.

More examples

You can find more examples, like an HTTP file server, in the `Examples` chapter.

Command line interface

Deno is a command line program. You should be familiar with some simple commands having followed the examples thus far and already understand the basics of shell usage.

There are multiple ways of viewing the main help text:

```
# Using the subcommand.
deno help

# Using the short flag -- outputs the same as above.
deno -h

# Using the long flag -- outputs more detailed help text
  ↪  where available.
deno --help
```

Deno's CLI is subcommand-based. The above commands should show you a list of those supported, such as `deno bundle`. To see subcommand-specific help for `bundle`, you can similarly run one of:

```
deno help bundle
deno bundle -h
deno bundle --help
```

Detailed guides to each subcommand can be found here.

Script source

Deno can grab the scripts from multiple sources, a filename, a url, and '-' to read the file from stdin. The last is useful for integration with other applications.

```
deno run main.ts
deno run https://mydomain.com/main.ts
cat main.ts | deno run -
```

Script arguments

Separately from the Deno runtime flags, you can pass user-space arguments to the script you are running by specifying them after the script name:

```
deno run main.ts a b -c --quiet
```

```
// main.ts
console.log(Deno.args); // [ "a", "b", "-c", "--quiet" ]
```

Note that anything passed after the script name will be passed as a script argument and not consumed as a Deno runtime flag. This leads to the following pitfall:

```
# Good. We grant net permission to net_client.ts.
deno run --allow-net net_client.ts

# Bad! --allow-net was passed to Deno.args, throws a net
  ↪  permission error.
deno run net_client.ts --allow-net
```

Some see it as unconventional that:

> a non-positional flag is parsed differently depending on its position.

However:

1. This is the most logical way of distinguishing between runtime flags and script arguments.
2. This is the most ergonomic way of distinguishing between runtime flags and script arguments.
3. This is, in fact, the same behaviour as that of any other popular runtime.
   - Try `node -c index.js` and `node index.js -c`. The first will only do a syntax check on `index.js` as per Node's `-c` flag. The second will *execute* `index.js` with `-c` passed to `require("process").argv`.

There exist logical groups of flags that are shared between related subcommands. We discuss these below.

Integrity flags

Affect commands which can download resources to the cache: `deno cache`, `deno run` and `deno test`.

```
--lock <FILE>    Check the specified lock file
--lock-write     Write lock file. Use with --lock.
```

Find out more about these here.

Cache and compilation flags

Affect commands which can populate the cache: `deno cache`, `deno run` and `deno test`. As well as the flags above this includes those which affect module resolution, compilation configuration etc.

```
--config <FILE>                 Load tsconfig.json
  ↪ configuration file
--importmap <FILE>              UNSTABLE: Load import map
  ↪ file
--no-remote                     Do not resolve remote
  ↪ modules
--reload=<CACHE_BLOCKLIST>      Reload source code cache (
  ↪ recompile TypeScript)
--unstable                      Enable unstable APIs
```

Runtime flags

Affect commands which execute user code: `deno run` and `deno test`. These include all of the above as well as the following.

Permission flags These are listed here.

Other runtime flags More flags which affect the execution environment.

```
--cached-only                  Require that remote
  ↪ dependencies are already cached
--inspect=<HOST:PORT>          activate inspector on host:
  ↪ port ...
--inspect-brk=<HOST:PORT>      activate inspector on host:
  ↪ port and break at ...
--seed <NUMBER>                Seed Math.random()
--v8-flags=<v8-flags>          Set V8 command line options
  ↪ . For help: ...
```

Permissions

Deno is secure by default. Therefore, unless you specifically enable it, a deno module has no file, network, or environment access for example. Access to security-sensitive areas or functions requires the use of permissions to be granted to a deno process on the command line.

For the following example, `mod.ts` has been granted read-only access to the file system. It cannot write to it, or perform any other security-sensitive functions.

```
deno run --allow-read mod.ts
```

Permissions list

The following permissions are available:

- **-A, –allow-all** Allow all permissions. This disables all security.
- **–allow-env** Allow environment access for things like getting and setting of environment variables.
- **–allow-hrtime** Allow high-resolution time measurement. High-resolution time can be used in timing attacks and fingerprinting.
- **–allow-net=<allow-net>** Allow network access. You can specify an optional, comma-separated list of domains to provide an allow-list of allowed domains.
- **–allow-plugin** Allow loading plugins. Please note that –allow-plugin is an unstable feature.
- **–allow-read=<allow-read>** Allow file system read access. You can specify an optional, comma-separated list of directories or files to provide a allow-list of allowed file system access.
- **–allow-run** Allow running subprocesses. Be aware that subprocesses are not run in a sandbox and therefore do not have the

same security restrictions as the deno process. Therefore, use with caution.

- **–allow-write=<allow-write>** Allow file system write access. You can specify an optional, comma-separated list of directories or files to provide a allow-list of allowed file system access.

Permissions allow-list

Deno also allows you to control the granularity of some permissions with allow-lists.

This example restricts file system access by allow-listing only the `/usr` directory, however the execution fails as the process was attempting to access a file in the `/etc` directory:

```
$ deno run --allow-read=/usr https://deno.land/
  ↪ std@$STD_VERSION/examples/cat.ts /etc/passwd
error: Uncaught PermissionDenied: read access to "/etc/
  ↪ passwd", run again with the --allow-read flag
 $deno$/dispatch_json.ts:40:11
    at DenoError ($deno$/errors.ts:20:5)
    ...
```

Try it out again with the correct permissions by allow-listing `/etc` instead:

```
deno run --allow-read=/etc https://deno.land/
  ↪ std@$STD_VERSION/examples/cat.ts /etc/passwd
```

`--allow-write` works the same as `--allow-read`.

Network access:

fetch.ts:

```
const result = await fetch("https://deno.land/");
```

This is an example of how to allow-list hosts/urls:

```
deno run --allow-net=github.com,deno.land fetch.ts
```

If `fetch.ts` tries to establish network connections to any other domain, the process will fail.

Allow net calls to any host/url:

```
deno run --allow-net fetch.ts
```

Using TypeScript

Deno supports both JavaScript and TypeScript as first class languages at runtime. This means it requires fully qualified module names, including the extension (or a server providing the correct media type). In addition, Deno has no “magical” module resolution. Instead, imported modules are specified as files (including extensions) or fully qualified URL imports. Typescript modules can be directly imported. E.g.

```
import { Response } from "https://deno.land/
  ↪ std@$STD_VERSION/http/server.ts";
import { queue } from "./collections.ts";
```

`--no-check` option

When using `deno run`, `deno test`, `deno cache`, `deno info`, or `deno bundle` you can specify the `--no-check` flag to disable TypeScript type checking. This can significantly reduce the time that program startup takes. This can be very useful when type checking is provided by your editor and you want startup time to be as fast as possible (for example when restarting the program automatically with a file watcher).

Because `--no-check` does not do TypeScript type checking we can not automatically remove type only imports and exports as this would require type information. For this purpose TypeScript provides the `import ↪ type` and `export type` syntax. To export a type in a different file use `export type { AnInterface } from "./mod.ts";`. To import a type use `import type { AnInterface } from "./mod.ts";`. You can check that you are using `import type` and `export type` where necessary by setting the `isolatedModules` TypeScript compiler option to `true`. You can see an example `tsconfig.json` with this option in the standard library.

Because there is no type information when using `--no-check`, `const enum` is not supported because it is type-directed. `--no-check` also does not support the legacy `import =` and `export =` syntax.

Using external type definitions

The out of the box TypeScript compiler though relies on both extension-less modules and the Node.js module resolution logic to apply types to JavaScript modules.

In order to bridge this gap, Deno supports three ways of referencing type definition files without having to resort to "magic" resolution.

Compiler hint If you are importing a JavaScript module, and you know where the type definition for that module is located, you can specify the type definition at import. This takes the form of a compiler hint. Compiler hints inform Deno the location of `.d.ts` files and the JavaScript code that is imported that they relate to. The hint is `@deno ↪ -types` and when specified the value will be used in the compiler instead of the JavaScript module. For example, if you had `foo.js`, but you know that alongside of it was `foo.d.ts` which was the types for the file, the code would look like this:

```
// @deno-types="./foo.d.ts"
import * as foo from "./foo.js";
```

The value follows the same resolution logic as importing a module, meaning the file needs to have an extension and is relative to the current module. Remote specifiers are also allowed.

The hint affects the next `import` statement (or `export ... from` statement) where the value of the `@deno-types` will be substituted at compile time instead of the specified module. Like in the above example, the Deno compiler will load `./foo.d.ts` instead of `./foo.js`. Deno will still load `./foo.js` when it runs the program.

Triple-slash reference directive in JavaScript files If you are hosting modules which you want to be consumed by Deno, and you want to inform Deno about the location of the type definitions, you can utilize a triple-slash directive in the actual code. For example, if you have a JavaScript module and you would like to provide Deno with the location of the type definition which happens to be alongside that file, your JavaScript module named `foo.js` might look like this:

```
/// <reference types="./foo.d.ts" />
export const foo = "foo";
```

Deno will see this, and the compiler will use `foo.d.ts` when type checking the file, though `foo.js` will be loaded at runtime. The resolution of the value of the directive follows the same resolution logic as importing a module, meaning the file needs to have an extension and is relative to the current file. Remote specifiers are also allowed.

X-TypeScript-Types custom header If you are hosting modules which you want to be consumed by Deno, and you want to inform Deno the location of the type definitions, you can use a custom HTTP header of `X-TypeScript-Types` to inform Deno of the location of that file.

The header works in the same way as the triple-slash reference mentioned above, it just means that the content of the JavaScript file itself does not need to be modified, and the location of the type definitions can be determined by the server itself.

Not all type definitions are supported.

Deno will use the compiler hint to load the indicated `.d.ts` files, but some `.d.ts` files contain unsupported features. Specifically, some `.d.ts` files expect to be able to load or reference type definitions from other packages using the module resolution logic. For example a type reference directive to include `node`, expecting to resolve to some path like `./`
`↪ node_modules/@types/node/index.d.ts`. Since this depends on non-relative "magical" resolution, Deno cannot resolve this.

Why not use the triple-slash type reference in TypeScript files?

The TypeScript compiler supports triple-slash directives, including a type reference directive. If Deno used this, it would interfere with the behavior of the TypeScript compiler. Deno only looks for the directive in JavaScript (and JSX) files.

Custom TypeScript Compiler Options

In the Deno ecosystem, all strict flags are enabled in order to comply with TypeScript's ideal of being `strict` by default. However, in order to provide a way to support customization a configuration file such as `tsconfig.json` might be provided to Deno on program execution.

You need to explicitly tell Deno where to look for this configuration by setting the `-c` (or `--config`) argument when executing your application.

```
deno run -c tsconfig.json mod.ts
```

Following are the currently allowed settings and their default values in Deno:

```
{
  "compilerOptions": {
    "allowJs": false,
    "allowUmdGlobalAccess": false,
    "allowUnreachableCode": false,
    "allowUnusedLabels": false,
    "alwaysStrict": true,
    "assumeChangesOnlyAffectDirectDependencies": false,
    "checkJs": false,
    "disableSizeLimit": false,
    "generateCpuProfile": "profile.cpuprofile",
    "jsx": "react",
    "jsxFactory": "React.createElement",
    "lib": [],
    "noFallthroughCasesInSwitch": false,
    "noImplicitAny": true,
    "noImplicitReturns": true,
    "noImplicitThis": true,
    "noImplicitUseStrict": false,
    "noStrictGenericChecks": false,
    "noUnusedLocals": false,
    "noUnusedParameters": false,
    "preserveConstEnums": false,
    "removeComments": false,
    "resolveJsonModule": true,
    "strict": true,
    "strictBindCallApply": true,
    "strictFunctionTypes": true,
    "strictNullChecks": true,
    "strictPropertyInitialization": true,
    "suppressExcessPropertyErrors": false,
    "suppressImplicitAnyIndexErrors": false,
    "useDefineForClassFields": false
  }
}
```

For documentation on allowed values and use cases please visit the typescript docs.

Note: Any options not listed above are either not supported by Deno or are listed as deprecated/experimental in the TypeScript documentation.

WebAssembly support

Deno can execute WebAssembly binaries.

```
const wasmCode = new Uint8Array([
  0, 97, 115, 109, 1, 0, 0, 0, 1, 133, 128, 128, 128, 0,
    ↪  1, 96, 0, 1, 127,
  3, 130, 128, 128, 128, 0, 1, 0, 4, 132, 128, 128, 128,
    ↪  0, 1, 112, 0, 0,
  5, 131, 128, 128, 128, 0, 1, 0, 1, 6, 129, 128, 128,
    ↪ 128, 0, 0, 7, 145,
  128, 128, 128, 0, 2, 6, 109, 101, 109, 111, 114, 121,
    ↪ 2, 0, 4, 109, 97,
  105, 110, 0, 0, 10, 138, 128, 128, 128, 0, 1, 132,
    ↪ 128, 128, 128, 0, 0,
  65, 42, 11
]);
const wasmModule = new WebAssembly.Module(wasmCode);
const wasmInstance = new WebAssembly.Instance(wasmModule
  ↪ );
console.log(wasmInstance.exports.main().toString());
```

Runtime

Documentation for all runtime functions (Web APIs + `Deno` global) can be found on `doc.deno.land`.

Web APIs

For APIs where a web standard already exists, like `fetch` for HTTP requests, Deno uses these rather than inventing a new proprietary API.

The detailed documentation for implemented Web APIs can be found on doc.deno.land. Additionally, a full list of the Web APIs which Deno implements is also available in the repository.

The TypeScript definitions for the implemented web APIs can be found in the `lib.deno.shared_globals.d.ts` and `lib.deno.window.d.ts` files.

Definitions that are specific to workers can be found in the `lib.deno.` ↪ `worker.d.ts` file.

`Deno` global

All APIs that are not web standard are contained in the global `Deno` ↪ namespace. It has the APIs for reading from files, opening TCP sockets, and executing subprocesses, etc.

The TypeScript definitions for the Deno namespaces can be found in the `lib.deno.ns.d.ts` file.

The documentation for all of the Deno specific APIs can be found on doc.deno.land.

Stability

As of Deno 1.0.0, the `Deno` namespace APIs are stable. That means we will strive to make code working under 1.0.0 continue to work in future versions.

However, not all of Deno's features are ready for production yet. Features which are not ready, because they are still in draft phase, are locked behind the `--unstable` command line flag.

```
deno run --unstable mod_which_uses_unstable_stuff.ts
```

Passing this flag does a few things:

- It enables the use of unstable APIs during runtime.
- It adds the `lib.deno.unstable.d.ts` file to the list of TypeScript definitions that are used for type checking. This includes the output of `deno types`.

You should be aware that many unstable APIs have **not undergone a security review**, are likely to have **breaking API changes** in the future, and are **not ready for production**.

Standard modules

Deno's standard modules (https://deno.land/std/) are not yet stable. We currently version the standard modules differently from the CLI to reflect this. Note that unlike the `Deno` namespace, the use of the standard modules do not require the `--unstable` flag (unless the standard module itself makes use of an unstable Deno feature).

Program lifecycle

Deno supports browser compatible lifecycle events: `load` and `unload`. You can use these events to provide setup and cleanup code in your program.

Listeners for `load` events can be asynchronous and will be awaited. Listeners for `unload` events need to be synchronous. Both events cannot be cancelled.

Example:

main.ts

```
import "./imported.ts";

const handler = (e: Event): void => {
  console.log(`got ${e.type} event in event handler (
    ↪ main)`);
};

window.addEventListener("load", handler);

window.addEventListener("unload", handler);

window.onload = (e: Event): void => {
  console.log(`got ${e.type} event in onload function (
    ↪ main)`);
};
```

```
window.onunload = (e: Event): void => {
  console.log(`got ${e.type} event in onunload function
    ↪ (main)`);
};

console.log("log from main script");
```

imported.ts

```
const handler = (e: Event): void => {
  console.log(`got ${e.type} event in event handler (
    ↪ imported)`);
};

window.addEventListener("load", handler);
window.addEventListener("unload", handler);

window.onload = (e: Event): void => {
  console.log(`got ${e.type} event in onload function (
    ↪ imported)`);
};

window.onunload = (e: Event): void => {
  console.log(`got ${e.type} event in onunload function
    ↪ (imported)`);
};

console.log("log from imported script");
```

Note that you can use both `window.addEventListener` and `window.onload` ↪ /`window.onunload` to define handlers for events. There is a major difference between them, let's run the example:

```
$ deno run main.ts
log from imported script
log from main script
got load event in onload function (main)
got load event in event handler (imported)
got load event in event handler (main)
got unload event in onunload function (main)
got unload event in event handler (imported)
got unload event in event handler (main)
```

All listeners added using `window.addEventListener` were run, but `window`
↪ `.onload` and `window.onunload` defined in `main.ts` overrode handlers defined in `imported.ts`.

In other words, you can register multiple `window.addEventListener "`
↪ `load"` or `"unload"` events, but only the last loaded `window.onload` or `window.onunload` events will be executed.

Permission APIs

This API is unstable. Learn more about unstable features.

Permissions are granted from the CLI when running the `deno` command. User code will often assume its own set of required permissions, but there is no guarantee during execution that the set of *granted* permissions will align with this.

In some cases, ensuring a fault-tolerant program requires a way to interact with the permission system at runtime.

Permission descriptors

On the CLI, read permission for `/foo/bar` is represented as `--allow-`
↪ `read=/foo/bar`. In runtime JS, it is represented as the following:

```
const desc = { name: "read", path: "/foo/bar" };
```

Other examples:

```
// Global write permission.
const desc1 = { name: "write" };

// Write permission to `$PWD/foo/bar`.
const desc2 = { name: "write", path: "foo/bar" };

// Global net permission.
const desc3 = { name: "net" };

// Net permission to 127.0.0.1:8000.
const desc4 = { name: "net", url: "127.0.0.1:8000" };

// High-resolution time permission.
const desc5 = { name: "hrtime" };
```

Query permissions

Check, by descriptor, if a permission is granted or not.

```
// deno run --unstable --allow-read=/foo main.ts

const desc1 = { name: "read", path: "/foo" };
console.log(await Deno.permissions.query(desc1));
// PermissionStatus { state: "granted" }

const desc2 = { name: "read", path: "/foo/bar" };
console.log(await Deno.permissions.query(desc2));
// PermissionStatus { state: "granted" }

const desc3 = { name: "read", path: "/bar" };
```

```
console.log(await Deno.permissions.query(desc3));
// PermissionStatus { state: "prompt" }
```

Permission states

A permission state can be either "granted", "prompt" or "denied". Permissions which have been granted from the CLI will query to `{ state: ↪ "granted" }`. Those which have not been granted query to `{ state: ↪ "prompt" }` by default, while `{ state: "denied" }` reserved for those which have been explicitly refused. This will come up in Request permissions.

Permission strength

The intuitive understanding behind the result of the second query in Query permissions is that read access was granted to `/foo` and `/foo/bar` is within `/foo` so `/foo/bar` is allowed to be read.

We can also say that `desc1` is *stronger than* `desc2`. This means that for any set of CLI-granted permissions:

1. If `desc1` queries to `{ state: "granted" }` then so must `desc2`.
2. If `desc2` queries to `{ state: "denied" }` then so must `desc1`.

More examples:

```
const desc1 = { name: "write" };
// is stronger than
const desc2 = { name: "write", path: "/foo" };

const desc3 = { name: "net" };
// is stronger than
const desc4 = { name: "net", url: "127.0.0.1:8000" };
```

Request permissions

Request an ungranted permission from the user via CLI prompt.

```
// deno run --unstable main.ts

const desc1 = { name: "read", path: "/foo" };
const status1 = await Deno.permissions.request(desc1);
//   Deno requests read access to "/foo". Grant? [g/d (g
  ↪ = grant, d = deny)] g
console.log(status1);
// PermissionStatus { state: "granted" }

const desc2 = { name: "read", path: "/bar" };
const status2 = await Deno.permissions.request(desc2);
//   Deno requests read access to "/bar". Grant? [g/d (g
  ↪ = grant, d = deny)] d
console.log(status2);
// PermissionStatus { state: "denied" }
```

If the current permission state is "prompt", a prompt will appear on the user's terminal asking them if they would like to grant the request. The request for `desc1` was granted so its new status is returned and execution will continue as if `--allow-read=/foo` was specified on the CLI. The request for `desc2` was denied so its permission state is downgraded from "prompt" to "denied".

If the current permission state is already either "granted" or "denied", the request will behave like a query and just return the current status. This prevents prompts both for already granted permissions and previously denied requests.

Revoke permissions

Downgrade a permission from “granted” to “prompt”.

```
// deno run --unstable --allow-read=/foo main.ts

const desc = { name: "read", path: "/foo" };
console.log(await Deno.permissions.revoke(desc));
// PermissionStatus { state: "prompt" }
```

However, what happens when you try to revoke a permission which is *partial* to one granted on the CLI?

```
// deno run --unstable --allow-read=/foo main.ts

const desc = { name: "read", path: "/foo/bar" };
console.log(await Deno.permissions.revoke(desc));
// PermissionStatus { state: "granted" }
```

It was not revoked.

To understand this behaviour, imagine that Deno stores an internal set of *explicitly granted permission descriptors*. Specifying `--allow-read` ↪ `=/foo,/bar` on the CLI initializes this set to:

```
[
  { name: "read", path: "/foo" },
  { name: "read", path: "/bar" },
];
```

Granting a runtime request for `{ name: "write", path: "/foo" }` updates the set to:

```
[
  { name: "read", path: "/foo" },
  { name: "read", path: "/bar" },
  { name: "write", path: "/foo" },
];
```

Deno's permission revocation algorithm works by removing every element from this set which the argument permission descriptor is *stronger than*. So to ensure `desc` is not longer granted, pass an argument descriptor *stronger than* whichever *explicitly granted permission descriptor* is *stronger than* `desc`.

```
// deno run --unstable --allow-read=/foo main.ts

const desc = { name: "read", path: "/foo/bar" };
console.log(await Deno.permissions.revoke(desc)); //
  ↪ Insufficient.
// PermissionStatus { state: "granted" }

const strongDesc = { name: "read", path: "/foo" };
await Deno.permissions.revoke(strongDesc); // Good.

console.log(await Deno.permissions.query(desc));
// PermissionStatus { state: "prompt" }
```

Compiler APIs

This API is unstable. Learn more about unstable features.

Deno supports runtime access to the built-in TypeScript compiler. There are three methods in the `Deno` namespace that provide this access.

`Deno.compile()`

This works similar to `deno cache` in that it can fetch and cache the code, compile it, but not run it. It takes up to three arguments, the `rootName`, optionally `sources`, and optionally `options`. The `rootName` is the root module which will be used to generate the resulting program. This is like the module name you would pass on the command line in `deno run` ↪ `--reload example.ts`. The `sources` is a hash where the key is the fully qualified module name, and the value is the text source of the module. If `sources` is passed, Deno will resolve all the modules from within that hash and not attempt to resolve them outside of Deno. If `sources` are not provided, Deno will resolve modules as if the root module had been passed on the command line. Deno will also cache any of these resources. All resolved resources are treated as dynamic imports and require read or net permissions depending on if they're local or remote. The `options` argument is a set of options of type `Deno.CompilerOptions`, which is a subset of the TypeScript compiler options containing the ones supported by Deno.

The method resolves with a tuple. The first argument contains any diagnostics (syntax or type errors) related to the code. The second argument is a map where the keys are the output filenames and the values are the content.

An example of providing sources:

```
const [diagnostics, emitMap] = await Deno.compile("/foo.
  ↪ ts", {
  "/foo.ts": `import * as bar from "./bar.ts";\nconsole.
    ↪ log(bar);\n`,
  "/bar.ts": `export const bar = "bar";\n`,
});
```

```
assert(diagnostics == null); // ensuring no diagnostics
  ↪ are returned
console.log(emitMap);
```

We would expect map to contain 4 "files", named `/foo.js.map`, `/foo.js`, `/bar.js.map`, and `/bar.js`.

When not supplying resources, you can use local or remote modules, just like you could do on the command line. So you could do something like this:

```
const [diagnostics, emitMap] = await Deno.compile(
  "https://deno.land/std@$STD_VERSION/examples/welcome.
    ↪ ts",
);
```

In this case `emitMap` will contain a `console.log()` statement.

`Deno.bundle()`

This works a lot like `deno bundle` does on the command line. It is also like `Deno.compile()`, except instead of returning a map of files, it returns a single string, which is a self-contained JavaScript ES module which will include all of the code that was provided or resolved as well as exports of all the exports of the root module that was provided. It takes up to three arguments, the `rootName`, optionally `sources`, and optionally `options`. The `rootName` is the root module which will be used to generate the resulting program. This is like module name you would pass on the command line in `deno bundle example.ts`. The `sources` is a hash where the key is the fully qualified module name, and the value is the text source of the module. If `sources` is passed, Deno will resolve all the modules from within that hash and not attempt to resolve them outside of Deno. If `sources` are not provided, Deno will resolve modules as if the root module had been passed on the command line. All resolved resources are treated as dynamic imports and require read or net permissions depending if they're local or remote. Deno will also cache any of these resources. The `options` argument is a set of options of type `Deno.CompilerOptions`, which is a subset of the TypeScript compiler options containing the ones supported by Deno.

An example of providing sources:

```
const [diagnostics, emit] = await Deno.bundle("/foo.ts",
  ↪ {
  "/foo.ts": `import * as bar from "./bar.ts";\nconsole.
    ↪ log(bar);\n`,
  "/bar.ts": `export const bar = "bar";\n`,
});

assert(diagnostics == null); // ensuring no diagnostics
  ↪ are returned
console.log(emit);
```

We would expect `emit` to be the text for an ES module, which would contain the output sources for both modules.

When not supplying resources, you can use local or remote modules, just like you could do on the command line. So you could do something like this:

```
const [diagnostics, emit] = await Deno.bundle(
  "https://deno.land/std@$STD_VERSION/http/server.ts",
);
```

In this case `emit` will be a self contained JavaScript ES module with all of its dependencies resolved and exporting the same exports as the source module.

`Deno.transpileOnly()`

This is based off of the TypeScript function `transpileModule()`. All this does is “erase” any types from the modules and emit JavaScript. There is no type checking and no resolution of dependencies. It accepts up to two arguments, the first is a hash where the key is the module name and the value is the content. The only purpose of the module name is when putting information into a source map, of what the source file name was. The second argument contains optional `options` of the type `Deno.CompilerOptions`. The function resolves with a map where the key is the source module name supplied, and the value is an object with a property of `source` and optionally `map`. The first is the output contents of the module. The `map` property is the source map. Source maps are provided by default, but can be turned off via the `options` argument.

An example:

```
const result = await Deno.transpileOnly({
  "/foo.ts": `enum Foo { Foo, Bar, Baz };\n`,
});

console.log(result["/foo.ts"].source);
console.log(result["/foo.ts"].map);
```

We would expect the `enum` would be rewritten to an IIFE which constructs the enumerable, and the map to be defined.

Referencing TypeScript library files

When you use `deno run`, or other Deno commands which type check TypeScript, that code is evaluated against custom libraries which describe the environment that Deno supports. By default, the compiler runtime APIs which type check TypeScript also use these libraries (`Deno.compile()` and `Deno.bundle()`).

But if you want to compile or bundle TypeScript for some other runtime, you may want to override the default libraries. To do this, the runtime APIs support the `lib` property in the compiler options. For example, if you had TypeScript code that is destined for the browser, you would want to use the TypeScript `"dom"` library:

```
const [errors, emitted] = await Deno.compile(
  "main.ts",
  {
    "main.ts": `document.getElementById("foo");\n`,
  },
  {
    lib: ["dom", "esnext"],
  },
);
```

For a list of all the libraries that TypeScript supports, see the `lib` compiler option documentation.

Don't forget to include the JavaScript library

Just like `tsc`, when you supply a `lib` compiler option, it overrides the default ones, which means that the basic JavaScript library won't be included and you should include the one that best represents your target runtime (e.g. `es5`, `es2015`, `es2016`, `es2017`, `es2018`, `es2019`, `es2020` or esnext).

Including the `Deno` namespace In addition to the libraries that are provided by TypeScript, there are four libraries that are built into Deno that can be referenced:

- `deno.ns` - Provides the `Deno` namespace.
- `deno.shared_globals` - Provides global interfaces and variables which Deno supports at runtime that are then exposed by the final runtime library.
- `deno.window` - Exposes the global variables plus the Deno namespace that are available in the Deno main worker and is the default for the runtime compiler APIs.
- `deno.worker` - Exposes the global variables that are available in workers under Deno.

So to add the Deno namespace to a compilation, you would include the `deno.ns` lib in the array. For example:

```
const [errors, emitted] = await Deno.compile(
  "main.ts",
  {
    "main.ts": `document.getElementById("foo");\n`,
  },
  {
    lib: ["dom", "esnext", "deno.ns"],
  },
);
```

Note that the Deno namespace expects a runtime environment that is at least ES2018 or later. This means if you use a lib "lower" than ES2018 you will get errors logged as part of the compilation.

Using the triple slash reference You do not have to specify the `lib` in the compiler options. Deno also supports the triple-slash reference to a lib which can be embedded in the contents of the file. For example, if you have a `main.ts` like:

```
/// <reference lib="dom" />

document.getElementById("foo");
```

It would compile without errors like this:

```
const [errors, emitted] = await Deno.compile("./main.ts
  ↪ ", undefined, {
  lib: ["esnext"],
});
```

Note that the `dom` library conflicts with some of the default globals that are defined in the default type library for Deno. To avoid this, you need to specify a `lib` option in the compiler options to the runtime compiler APIs.

Workers

Deno supports `Web Worker API`.

Workers can be used to run code on multiple threads. Each instance of `Worker` is run on a separate thread, dedicated only to that worker.

Currently Deno supports only `module` type workers; thus it's essential to pass the `type: "module"` option when creating a new worker.

Relative module specifiers are not supported at the moment. You can instead use the `URL` contructor and `import.meta.url` to easily create a specifier for some nearby script.

```
// Good
new Worker(new URL("worker.js", import.meta.url).href, {
  ↪  type: "module" });

// Bad
new Worker(new URL("worker.js", import.meta.url).href);
new Worker(new URL("worker.js", import.meta.url).href, {
  ↪  type: "classic" });
new Worker("./worker.js", { type: "module" });
```

Permissions

Creating a new `Worker` instance is similar to a dynamic import; therefore Deno requires appropriate permission for this action.

For workers using local modules; `--allow-read` permission is required:

main.ts

```
new Worker(new URL("worker.ts", import.meta.url).href, {
  ↪  type: "module" });
```

worker.ts

```
console.log("hello world");
self.close();
```

```
$ deno run main.ts
error: Uncaught PermissionDenied: read access to "./
  ↪ worker.ts", run again with the --allow-read flag

$ deno run --allow-read main.ts
hello world
```

For workers using remote modules; `--allow-net` permission is required:

main.ts

```
new Worker("https://example.com/worker.ts", { type: "
  ↪ module" });
```

worker.ts (at https://example.com/worker.ts)

```
console.log("hello world");
self.close();
```

```
$ deno run main.ts
error: Uncaught PermissionDenied: net access to "https
  ↪ ://example.com/worker.ts", run again with the --
  ↪ allow-net flag

$ deno run --allow-net main.ts
hello world
```

Using Deno in worker

> This is an unstable Deno feature. Learn more about unstable features.

By default the `Deno` namespace is not available in worker scope.

To add the `Deno` namespace pass `deno: true` option when creating new worker:

main.js

```
const worker = new Worker(new URL("worker.js", import.
  ↪ meta.url).href, {
  type: "module",
  deno: true,
});
worker.postMessage({ filename: "./log.txt" });
```

worker.js

```
self.onmessage = async (e) => {
  const { filename } = e.data;
  const text = await Deno.readTextFile(filename);
  console.log(text);
  self.close();
};
```

log.txt

```
hello world
```

```
$ deno run --allow-read --unstable main.js
hello world
```

When the `Deno` namespace is available in worker scope, the worker inherits its parent process' permissions (the ones specified using `--allow-*` flags).

We intend to make permissions configurable for workers.

Linking to third party code

In the Getting Started section, we saw Deno could execute scripts from URLs. Like browser JavaScript, Deno can import libraries directly from URLs. This example uses a URL to import an assertion library:

test.ts

```
import { assertEquals } from "https://deno.land/
  ↪ std@$STD_VERSION/testing/asserts.ts";

assertEquals("hello", "hello");
assertEquals("world", "world");
```

```
console.log("Asserted! ");
```

Try running this:

```
$ deno run test.ts
Compile file:///mnt/f9/Projects/github.com/denoland/deno
  ↪ /docs/test.ts
Download https://deno.land/std@$STD_VERSION/testing/
  ↪ asserts.ts
Download https://deno.land/std@$STD_VERSION/fmt/colors.
  ↪ ts
Download https://deno.land/std@$STD_VERSION/testing/diff
  ↪ .ts
Asserted!
```

Note that we did not have to provide the `--allow-net` flag for this program, and yet it accessed the network. The runtime has special access to download imports and cache them to disk.

Deno caches remote imports in a special directory specified by the `DENO_DIR` environment variable. It defaults to the system's cache directory if `DENO_DIR` is not specified. The next time you run the program, no downloads will be made. If the program hasn't changed, it won't be recompiled either. The default directory is:

- On Linux/Redox: `$XDG_CACHE_HOME/deno` or `$HOME/.cache/deno`
- On Windows: `%LOCALAPPDATA%/deno` (`%LOCALAPPDATA%` = `FOLDERID_LocalAppData`)
- On macOS: `$HOME/Library/Caches/deno`
- If something fails, it falls back to `$HOME/.deno`

FAQ

How do I import a specific version of a module?

Specify the version in the URL. For example, this URL fully specifies the code being run: `https://unpkg.com/liltest@0.0.5/dist/liltest.js` ↪ .

It seems unwieldy to import URLs everywhere.

> What if one of the URLs links to a subtly different version of a library?
>
> Isn't it error prone to maintain URLs everywhere in a large project?

The solution is to import and re-export your external libraries in a central `deps.ts` file (which serves the same purpose as Node's `package.` ↪ `json` file). For example, let's say you were using the above assertion library across a large project. Rather than importing `"https://deno.` ↪ `land/std@$STD_VERSION/testing/asserts.ts"` everywhere, you could create a `deps.ts` file that exports the third-party code:

deps.ts

```
export {
  assert,
  assertEquals,
  assertStrContains,
} from "https://deno.land/std@$STD_VERSION/testing/
  ↪ asserts.ts";
```

And throughout the same project, you can import from the `deps.ts` and avoid having many references to the same URL:

```
import { assertEquals, runTests, test } from "./deps.ts
  ↪ ";
```

This design circumvents a plethora of complexity spawned by package management software, centralized code repositories, and superfluous file formats.

How can I trust a URL that may change?

By using a lock file (with the `--lock` command line flag), you can ensure that the code pulled from a URL is the same as it was during initial development. You can learn more about this here.

But what if the host of the URL goes down? The source won't be available.

This, like the above, is a problem faced by *any* remote dependency system. Relying on external servers is convenient for development but brittle in production. Production software should always vendor its dependencies. In Node this is done by checking `node_modules` into source control. In Deno this is done by pointing `$DENO_DIR` to some project-local directory at runtime, and similarly checking that into source control:

```
# Download the dependencies.
DENO_DIR=./deno_dir deno cache src/deps.ts

# Make sure the variable is set for any command which
  ↪ invokes the cache.
DENO_DIR=./deno_dir deno test src

# Check the directory into source control.
git add -u deno_dir
git commit
```

Reloading modules

By default, a module in the cache will be reused without fetching or re-compiling it. Sometimes this is not desirable and you can force deno to refetch and recompile modules into the cache. You can invalidate your local `DENO_DIR` cache using the `--reload` flag of the `deno cache` subcommand. It's usage is described below:

To reload everything

```
deno cache --reload my_module.ts
```

To reload specific modules

Sometimes we want to upgrade only some modules. You can control it by passing an argument to a `--reload` flag.

To reload all $STD_VERSION standard modules

```
deno cache --reload=https://deno.land/std@$STD_VERSION
  ↪ my_module.ts
```

To reload specific modules (in this example - colors and file system copy) use a comma to separate URLs

```
deno cache --reload=https://deno.land/std@$STD_VERSION/
  ↪ fs/copy.ts,https://deno.land/std@$STD_VERSION/fmt/
  ↪ colors.ts my_module.ts
```

Integrity checking & lock files

Introduction

Let's say your module depends on remote module `https://some.url`
↪ `/a.ts`. When you compile your module for the first time `a.ts` is retrieved, compiled and cached. It will remain this way until you run your module on a new machine (say in production) or reload the cache (through `deno cache --reload` for example). But what happens if the content in the remote url `https://some.url/a.ts` is changed? This could lead to your production module running with different dependency code than your local module. Deno's solution to avoid this is to use integrity checking and lock files.

Caching and lock files

Deno can store and check subresource integrity for modules using a small JSON file. Use the `--lock=lock.json` to enable and specify lock file checking. To update or create a lock use `--lock=lock.json --lock-`
↪ `write`. The `--lock=lock.json` tells Deno what the lock file to use is, while the `--lock-write` is used to output dependency hashes to the lock file (`--lock-write` must be used in conjunction with `--lock`).

A `lock.json` might look like this, storing a hash of the file against the dependency:

```
{
  "https://deno.land/std@$STD_VERSION/textproto/mod.ts":
    ↪  "3118
    ↪ d7a42c03c242c5a49c2ad91c8396110e14acca1324e7aaefd31a
    ↪ ",
  "https://deno.land/std@$STD_VERSION/io/util.ts": "
    ↪ ae133d310a0fdcf298cea7bc09a599c49acb616d34e148e263bc
    ↪ ",
  "https://deno.land/std@$STD_VERSION/async/delay.ts":
    ↪ "35957
    ↪ d585a6e3dd87706858fb1d6b551cb278271b03f52c5a2cb70e65
    ↪ ",
   ...
}
```

A typical workflow will look like this:

src/deps.ts

```
// Add a new dependency to "src/deps.ts", used somewhere
  ↪  else.
export { xyz } from "https://unpkg.com/xyz-lib@v0.9.0/
  ↪ lib.ts";
```

Then:

```
# Create/update the lock file "lock.json".
deno cache --lock=lock.json --lock-write src/deps.ts

# Include it when committing to source control.
git add -u lock.json
git commit -m "feat: Add support for xyz using xyz-lib"
git push
```

Collaborator on another machine – in a freshly cloned project tree:

```
# Download the project's dependencies into the machine's
  ↪  cache, integrity
# checking each resource.
deno cache --reload --lock=lock.json src/deps.ts

# Done! You can proceed safely.
deno test --allow-read src
```

Runtime verification

Like caching above, you can also use the `--lock=lock.json` option during use of the `deno run` sub command, validating the integrity of any locked modules during the run. Remember that this only validates against dependencies previously added to the `lock.json` file. New dependencies will be cached but not validated.

You can take this a step further as well by using the `--cached-only` flag to require that remote dependencies are already cached.

```
deno run --lock=lock.json --cached-only mod.ts
```

This will fail if there are any dependencies in the dependency tree for mod.ts which are not yet cached.

Proxies

Deno supports proxies for module downloads and the Web standard `fetch` API.

Proxy configuration is read from environmental variables: `HTTP_PROXY` and `HTTPS_PROXY`.

In case of Windows, if environment variables are not found Deno falls back to reading proxies from registry.

Import maps

> This is an unstable feature. Learn more about unstable features.

Deno supports import maps.

You can use import maps with the `--importmap=<FILE>` CLI flag.

Current limitations:

- single import map
- no fallback URLs
- Deno does not support `std:` namespace
- supports only `file:`, `http:` and `https:` schemes

Example:

import_map.json

```
{
   "imports": {
      "fmt/": "https://deno.land/std@$STD_VERSION/fmt/"
   }
}
```

color.ts

```
import { red } from "fmt/colors.ts";

console.log(red("hello world"));
```

Then:

```
$ deno run --importmap=import_map.json --unstable color.
  ↪ ts
```

To use starting directory for absolute imports:

```
// import_map.json

{
  "imports": {
    "/": "./"
  }
}
```

```
// main.ts

import { MyUtil } from "/util.ts";
```

You may map a different directory: (eg. src)

```
// import_map.json

{
  "imports": {
    "/": "./src"
  }
}
```

Standard library

Deno provides a set of standard modules that are audited by the core team and are guaranteed to work with Deno.

Standard library is available at: https://deno.land/std/

Versioning and stability

Standard library is not yet stable and therefore it is versioned differently than Deno. For latest release consult https://deno.land/std/ or https://deno.land/std/version.ts. The standard library is released each time Deno is released.

We strongly suggest to always use imports with pinned version of standard library to avoid unintended changes. For example, rather than linking to the master branch of code, which may change at any time, potentially causing compilation errors or unexpected behavior:

```
// imports from master, this should be avoided
import { copy } from "https://deno.land/std/fs/copy.ts";
```

instead, used a version of the std library which is immutable and will not change:

```
// imports from v0.50.0 of std, never changes
import { copy } from "https://deno.land/std@$STD_VERSION
  ↪ /fs/copy.ts";
```

Troubleshooting

Some of the modules provided in standard library use unstable Deno APIs.

Trying to run such modules without `--unstable` CLI flag ends up with a lot of TypeScript errors suggesting that some APIs in the `Deno` namespace do not exist:

```
// main.ts
import { copy } from "https://deno.land/std@$STD_VERSION
  ↪ /fs/copy.ts";

copy("log.txt", "log-old.txt");
```

```
$ deno run --allow-read --allow-write main.ts
Compile file:///dev/deno/main.ts
Download https://deno.land/std@$STD_VERSION/fs/copy.ts
Download https://deno.land/std@$STD_VERSION/fs/
  ↪ ensure_dir.ts
Download https://deno.land/std@$STD_VERSION/fs/_util.ts
```

```
error: TS2339 [ERROR]: Property 'utime' does not exist
  ↪ on type 'typeof Deno'.
    await Deno.utime(dest, statInfo.atime, statInfo.
      ↪ mtime);
                ~~~~~
    at https://deno.land/std@$STD_VERSION/fs/copy.ts
      ↪ :90:16

TS2339 [ERROR]: Property 'utimeSync' does not exist on
  ↪ type 'typeof Deno'.
    Deno.utimeSync(dest, statInfo.atime, statInfo.mtime)
      ↪ ;
          ~~~~~~~~~
    at https://deno.land/std@$STD_VERSION/fs/copy.ts
      ↪ :101:10
```

Solution to that problem requires adding `--unstable` flag:

```
deno run --allow-read --allow-write --unstable main.ts
```

To make sure that API producing error is unstable check `lib.deno.`
`↪ unstable.d.ts` declaration.

This problem should be fixed in the near future. Feel free to omit the flag if the particular modules you depend on compile successfully without it.

Testing

Deno has a built-in test runner that you can use for testing JavaScript or TypeScript code.

Writing tests

To define a test you need to call `Deno.test` with a name and function to be tested. There are two styles you can use.

```
// Simple name and function, compact form, but not
  ↪ configurable
Deno.test("hello world #1", () => {
  const x = 1 + 2;
  assertEquals(x, 3);
});

// Fully fledged test definition, longer form, but
  ↪ configurable (see below)
Deno.test({
  name: "hello world #2",
  fn: () => {
    const x = 1 + 2;
    assertEquals(x, 3);
  },
});
```

Assertions

There are some useful assertion utilities at https://deno.land/std@$STD_
to make testing easier:

```
import {
  assertEquals,
  assertArrayContains,
} from "https://deno.land/std@$STD_VERSION/testing/
  ↪ asserts.ts";

Deno.test("hello world", () => {
  const x = 1 + 2;
  assertEquals(x, 3);
  assertArrayContains([1, 2, 3, 4, 5, 6], [3], "Expected
    ↪  3 to be in the array");
});
```

Async functions

You can also test asynchronous code by passing a test function that returns a promise. For this you can use the `async` keyword when defining a function:

```
import { delay } from "https://deno.land/
  ↪ std@$STD_VERSION/async/delay.ts";

Deno.test("async hello world", async () => {
  const x = 1 + 2;

  // await some async task
  await delay(100);

  if (x !== 3) {
    throw Error("x should be equal to 3");
  }
});
```

Resource and async op sanitizers

Certain actions in Deno create resources in the resource table (learn more here). These resources should be closed after you are done using them.

For each test definition, the test runner checks that all resources created in this test have been closed. This is to prevent resource ‘leaks’. This is enabled by default for all tests, but can be disabled by setting the `sanitizeResources` boolean to false in the test definition.

The same is true for async operation like interacting with the filesystem. The test runner checks that each operation you start in the test is completed before the end of the test. This is enabled by default for all tests, but can be disabled by setting the `sanitizeOps` boolean to false in the test definition.

```
Deno.test({
  name: "leaky test",
  fn() {
    Deno.open("hello.txt");
  },
  sanitizeResources: false,
  sanitizeOps: false,
});
```

Running tests

To run the test, call `deno test` with the file that contains your test function. You can also omit the file name, in which case all tests in the current directory (recursively) that match the glob `{*_,*.,}test.{js,` ↪ `mjs,ts,jsx,tsx}` will be run. If you pass a directory, all files in the directory that match this glob will be run.

```
# Run all tests in the current directory and all sub-
  ↪ directories
deno test

# Run all tests in the util directory
deno test util/
```

```
# Run just my_test.ts
deno test my_test.ts
```

`deno test` uses the same permission model as `deno run` and therefore will require, for example, `--allow-write` to write to the file system during testing.

To see all runtime options with `deno test`, you can reference the command line help:

```
deno help test
```

Filtering

There are a number of options to filter the tests you are running.

Command line filtering

Tests can be run individually or in groups using the command line `--`
↪ `filter` option.

The filter flags accept a string or a pattern as value.

Assuming the following tests:

```
Deno.test({ name: "my-test", fn: myTest });
Deno.test({ name: "test-1", fn: test1 });
Deno.test({ name: "test2", fn: test2 });
```

This command will run all of these tests because they all contain the word "test".

```
deno test --filter "test" tests/
```

On the flip side, the following command uses a pattern and will run the second and third tests.

```
deno test --filter "/test-*\d/" tests/
```

To let Deno know that you want to use a pattern, wrap your filter with forward-slashes like the JavaScript syntactic sugar for a REGEX.

Test definition filtering

Within the tests themselves, you have two options for filtering.

Filtering out (Ignoring these tests) Sometimes you want to ignore tests based on some sort of condition (for example you only want a test to run on Windows). For this you can use the `ignore` boolean in the test definition. If it is set to true the test will be skipped.

```
Deno.test({
  name: "do macOS feature",
  ignore: Deno.build.os !== "darwin",
  fn() {
    doMacOSFeature();
  },
});
```

Filtering in (Only run these tests) Sometimes you may be in the middle of a problem within a large test class and you would like to focus on just that test and ignore the rest for now. For this you can use the `only` option to tell the test framework to only run tests with this set to true. Multiple tests can set this option. While the test run will report on the success or failure of each test, the overall test run will always fail if any test is flagged with `only`, as this is a temporary measure only which disables nearly all of your tests.

```
Deno.test({
  name: "Focus on this test only",
  only: true,
  fn() {
    testComplicatedStuff();
  },
});
```

Failing fast

If you have a long running test suite and wish for it to stop on the first failure, you can specify the `--failfast` flag when running the suite.

```
deno test --failfast
```

Assertions

To help developers write tests the Deno standard library comes with a built in assertions module which can be imported from `https://deno.`
`↪ land/std@$STD_VERSION/testing/asserts.ts`.

```
import { assert } from "https://deno.land/
  ↪ std@$STD_VERSION/testing/asserts.ts";

Deno.test("Hello Test", () => {
  assert("Hello");
});
```

The assertions module provides nine assertions:

- `assert(expr: unknown, msg = ""): asserts expr`

- `assertEquals(actual: unknown, expected: unknown, msg?:`
  `↪ string): void`
- `assertNotEquals(actual: unknown, expected: unknown, msg?:`
  `↪ string): void`
- `assertStrictEquals(actual: unknown, expected: unknown, msg?:`
  `↪  string): void`
- `assertStringContains(actual: string, expected: string, msg?:`
  `↪  string): void`
- `assertArrayContains(actual: unknown[], expected: unknown[],`
  `↪ msg?: string): void`
- `assertMatch(actual: string, expected: RegExp, msg?: string):`
  `↪  void`
- `assertThrows(fn: ()=> void, ErrorClass?: Constructor,`
  `↪ msgIncludes = "", msg?: string): Error`
- `assertThrowsAsync(fn: ()=> Promise<void>, ErrorClass?:`
  `↪ Constructor, msgIncludes = "", msg?: string): Promise<`
  `↪ Error>`

Assert

The assert method is a simple 'truthy' assertion and can be used to assert any value which can be inferred as true.

```
Deno.test("Test Assert", () => {
  assert(1);
  assert("Hello");
  assert(true);
});
```

Equality

There are three equality assertions available, `assertEquals()`, `assertNotEquals()` and `assertStrictEquals()`.

The `assertEquals()` and `assertNotEquals()` methods provide a general equality check and are capable of asserting equality between primitive types and objects.

```
Deno.test("Test Assert Equals", () => {
  assertEquals(1, 1);
  assertEquals("Hello", "Hello");
  assertEquals(true, true);
  assertEquals(undefined, undefined);
  assertEquals(null, null);
  assertEquals(new Date(), new Date());
  assertEquals(new RegExp("abc"), new RegExp("abc"));

  class Foo {}
  const foo1 = new Foo();
  const foo2 = new Foo();

  assertEquals(foo1, foo2);
});

Deno.test("Test Assert Not Equals", () => {
  assertNotEquals(1, 2);
  assertNotEquals("Hello", "World");
  assertNotEquals(true, false);
  assertNotEquals(undefined, "");
  assertNotEquals(new Date(), Date.now());
  assertNotEquals(new RegExp("abc"), new RegExp("def"));
});
```

By contrast `assertStrictEquals()` provides a simpler, stricter equality check based on the `===` operator. As a result it will not assert two instances of identical objects as they won't be referentially the same.

```
Deno.test("Test Assert Strict Equals", () => {
  assertStrictEquals(1, 1);
  assertStrictEquals("Hello", "Hello");
  assertStrictEquals(true, true);
  assertStrictEquals(undefined, undefined);
});
```

The `assertStrictEquals()` assertion is best used when you wish to make a precise check against two primitive types.

Contains

There are two methods available to assert a value contains a value, `assertStringContains()` and `assertArrayContains()`.

The `assertStringContains()` assertion does a simple includes check on a string to see if it contains the expected string.

```
Deno.test("Test Assert String Contains", () => {
  assertStringContains("Hello World", "Hello");
});
```

The `assertArrayContains()` assertion is slightly more advanced and can find both a value within an array and an array of values within an array.

```
Deno.test("Test Assert Array Contains", () => {
  assertArrayContains([1, 2, 3], [1]);
  assertArrayContains([1, 2, 3], [1, 2]);
  assertArrayContains(Array.from("Hello World"), Array.
    ↪ from("Hello"));
});
```

Regex

You can assert regular expressions via the `assertMatch()` assertion.

```
Deno.test("Test Assert Match", () => {
  assertMatch("abcdefghi", new RegExp("def"));

  const basicUrl = new RegExp("^https?://[a-z.]+.com$");
  assertMatch("https://www.google.com", basicUrl);
  assertMatch("http://facebook.com", basicUrl);
});
```

Throws

There are two ways to assert whether something throws an error in Deno, `assertThrows()` and `assertAsyncThrows()`. Both assertions allow you to check an Error has been thrown, the type of error thrown and what the message was.

The difference between the two assertions is `assertThrows()` accepts a standard function and `assertAsyncThrows()` accepts a function which returns a Promise.

The `assertThrows()` assertion will check an error has been thrown, and optionally will check the thrown error is of the correct type, and assert the error message is as expected.

```
Deno.test("Test Assert Throws", () => {
  assertThrows(
    () => {
      throw new Error("Panic!");
    },
    Error,
    "Panic!",
  );
});
```

The `assertAsyncThrows()` assertion is a little more complicated, mainly because it deals with Promises. But basically it will catch thrown errors or rejections in Promises. You can also optionally check for the error type and error message.

```
Deno.test("Test Assert Throws Async", () => {
  assertThrowsAsync(
    () => {
      return new Promise(() => {
        throw new Error("Panic! Threw Error");
      });
    },
    Error,
    "Panic! Threw Error",
  );

  assertThrowsAsync(
    () => {
      return Promise.reject(new Error("Panic! Reject
        ↪ Error"));
    },
    Error,
    "Panic! Reject Error",
  );
});
```

Custom Messages

Each of Deno's built in assertions allow you to overwrite the standard CLI error message if you wish. For instance this example will output

"Values Don't Match!" rather than the standard CLI error message.

```
Deno.test("Test Assert Equal Fail Custom Message", () =>
  ↪  {
  assertEquals(1, 2, "Values Don't Match!");
});
```

Built-in tooling

Deno provides some built in tooling that is useful when working with JavaScript and TypeScript:

- bundler (`deno bundle`)
- debugger (`--inspect, --inspect-brk`)
- dependency inspector (`deno info`)
- documentation generator (`deno doc`)
- formatter (`deno fmt`)
- test runner (`deno test`)
- linter (`deno lint`)

Debugger

Deno supports the V8 Inspector Protocol.

It's possible to debug Deno programs using Chrome Devtools or other clients that support the protocol (eg. VSCode).

To activate debugging capabilities run Deno with the `--inspect` or `--`
↪ `inspect-brk` flags.

The `--inspect` flag allows attaching the debugger at any point in time, while `--inspect-brk` will wait for the debugger to attach and will pause execution on the first line of code.

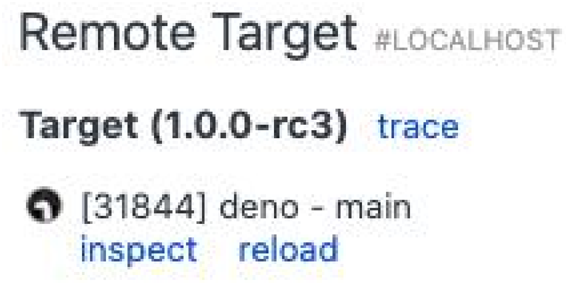
Chrome Devtools

Let's try debugging a program using Chrome Devtools. For this, we'll use file_server.ts from `std`, a static file server.

Use the `--inspect-brk` flag to break execution on the first line:

```
$ deno run --inspect-brk --allow-read --allow-net https
  ↪ ://deno.land/std@$STD_VERSION/http/file_server.ts
Debugger listening on ws://127.0.0.1:9229/ws/1e82c406-85
  ↪ a9-44ab-86b6-7341583480b1
Download https://deno.land/std@$STD_VERSION/http/
  ↪ file_server.ts
Compile https://deno.land/std@$STD_VERSION/http/
  ↪ file_server.ts
...
```

Open `chrome://inspect` and click `Inspect` next to target:


Remote Target #LOCALHOST

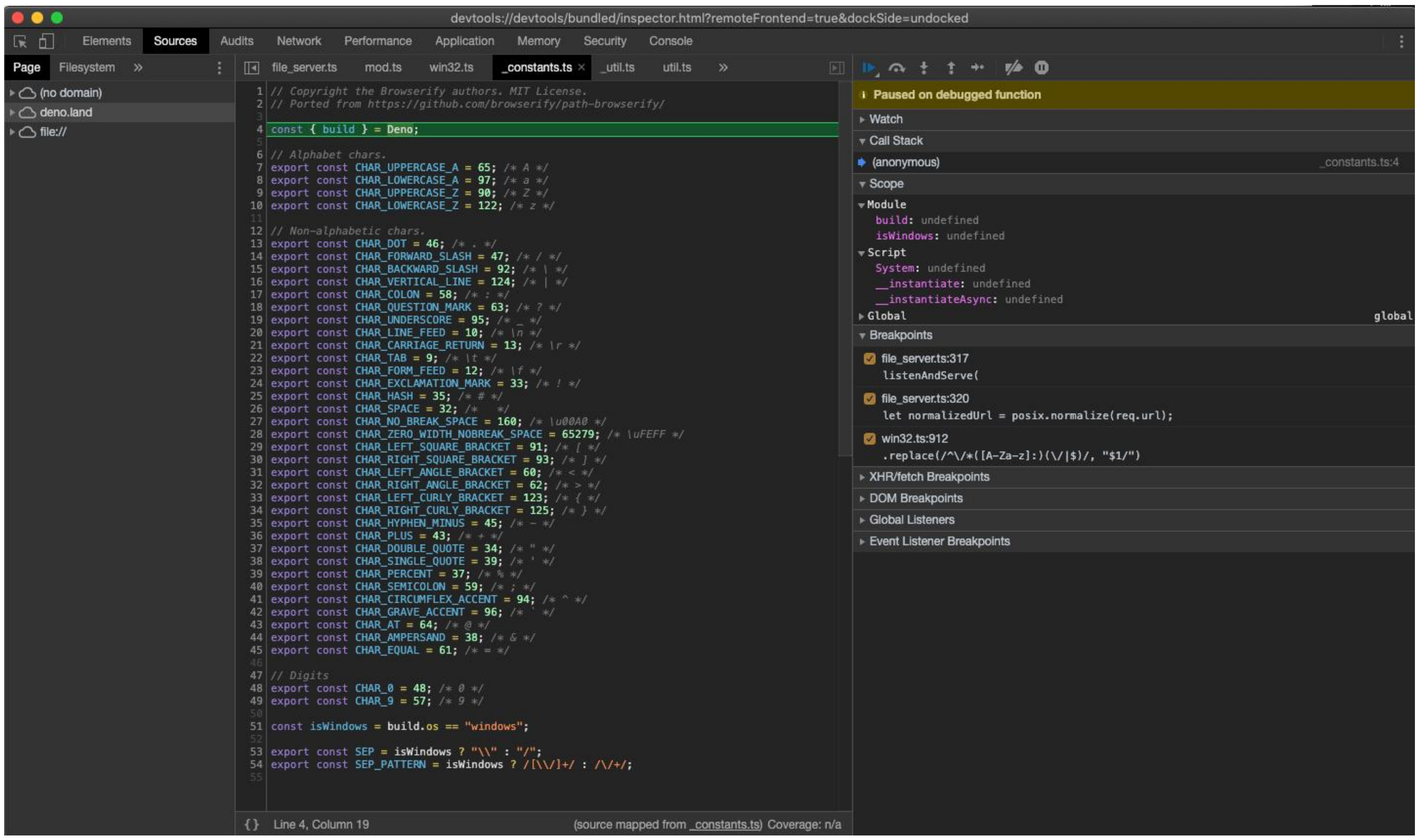
Target (1.0.0-rc3) trace

[31844] deno - main
inspect reload

Figure 1: chrome://inspect

It might take a few seconds after opening the Devtools to load all modules.

You might notice that Devtools paused execution on the first line of `_constants.ts` instead of `file_server.ts`. This is expected behavior



```
// Copyright the Browserify authors. MIT License.
// Ported from https://github.com/browserify/path-browserify/

const { build } = Deno;

// Alphabet chars.
export const CHAR_UPPERCASE_A = 65; /* A */
export const CHAR_LOWERCASE_A = 97; /* a */
export const CHAR_UPPERCASE_Z = 90; /* Z */
export const CHAR_LOWERCASE_Z = 122; /* z */

// Non-alphabetic chars.
export const CHAR_DOT = 46; /* . */
export const CHAR_FORWARD_SLASH = 47; /* / */
export const CHAR_BACKWARD_SLASH = 92; /* \ */
export const CHAR_VERTICAL_LINE = 124; /* | */
export const CHAR_COLON = 58; /* : */
export const CHAR_QUESTION_MARK = 63; /* ? */
export const CHAR_UNDERSCORE = 95; /* _ */
export const CHAR_LINE_FEED = 10; /* \n */
export const CHAR_CARRIAGE_RETURN = 13; /* \r */
export const CHAR_TAB = 9; /* \t */
export const CHAR_FORM_FEED = 12; /* \f */
export const CHAR_EXCLAMATION_MARK = 33; /* ! */
export const CHAR_HASH = 35; /* # */
export const CHAR_SPACE = 32; /*   */
export const CHAR_NO_BREAK_SPACE = 160; /* \u00A0 */
export const CHAR_ZERO_WIDTH_NOBREAK_SPACE = 65279; /* \uFEFF */
export const CHAR_LEFT_SQUARE_BRACKET = 91; /* [ */
export const CHAR_RIGHT_SQUARE_BRACKET = 93; /* ] */
export const CHAR_LEFT_ANGLE_BRACKET = 60; /* < */
export const CHAR_RIGHT_ANGLE_BRACKET = 62; /* > */
export const CHAR_LEFT_CURLY_BRACKET = 123; /* { */
export const CHAR_RIGHT_CURLY_BRACKET = 125; /* } */
export const CHAR_HYPHEN_MINUS = 45; /* - */
export const CHAR_PLUS = 43; /* + */
export const CHAR_DOUBLE_QUOTE = 34; /* " */
export const CHAR_SINGLE_QUOTE = 39; /* ' */
export const CHAR_PERCENT = 37; /* % */
export const CHAR_SEMICOLON = 59; /* ; */
export const CHAR_CIRCUMFLEX_ACCENT = 94; /* ^ */
export const CHAR_GRAVE_ACCENT = 96; /* ` */
export const CHAR_AT = 64; /* @ */
export const CHAR_AMPERSAND = 38; /* & */
export const CHAR_EQUAL = 61; /* = */

// Digits
export const CHAR_0 = 48; /* 0 */
export const CHAR_9 = 57; /* 9 */

const isWindows = build.os == "windows";

export const SEP = isWindows ? "\\" : "/";
export const SEP_PATTERN = isWindows ? /[\\/]+/ : /\/+/;
```

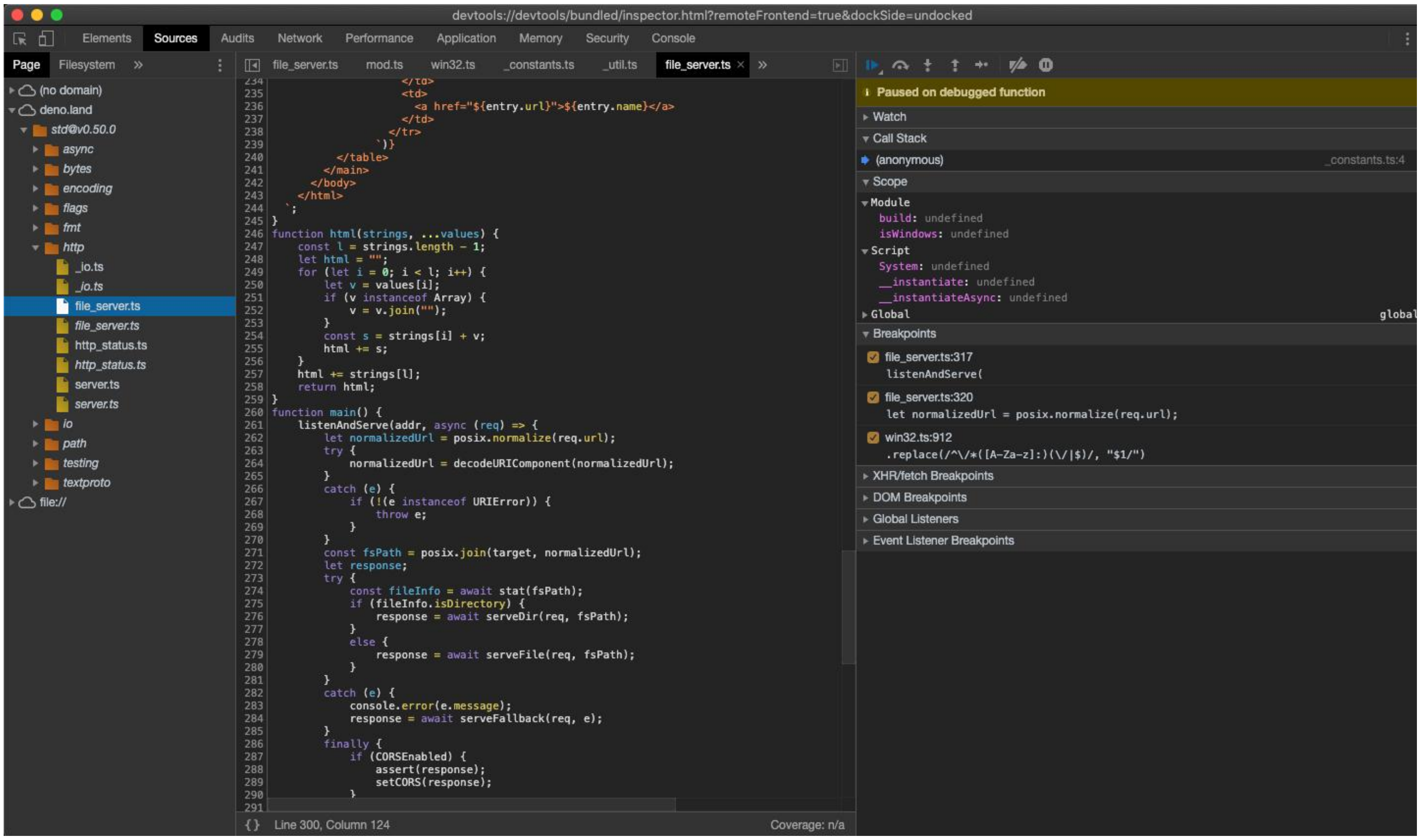
```
{} Line 4, Column 19                    (source mapped from _constants.ts) Coverage: n/a
```

```
Paused on debugged function
▸ Watch
▾ Call Stack
(anonymous)                                   _constants.ts:4
▾ Scope
▾Module
  build: undefined
  isWindows: undefined
▾Script
  System: undefined
  __instantiate: undefined
  __instantiateAsync: undefined
▸Global                                               global
▾ Breakpoints
☑ file_server.ts:317
  listenAndServe(
☑ file_server.ts:320
  let normalizedUrl = posix.normalize(req.url);
☑ win32.ts:912
  .replace(/^\/*([A-Za-z]:)(\/|$)/, "$1/")
▸ XHR/fetch Breakpoints
▸ DOM Breakpoints
▸ Global Listeners
▸ Event Listener Breakpoints
```

Figure 2: Devtools opened

and is caused by the way ES modules are evaluated by V8 (`_constants` ↪ `.ts` is left-most, bottom-most dependency of `file_server.ts` so it is evaluated first).

At this point all source code is available in the Devtools, so let's open up `file_server.ts` and add a breakpoint there; go to "Sources" pane and expand the tree:



```
                </td>
                <td>
                  <a href="${entry.url}">${entry.name}</a>
                </td>
              </tr>
            `)}
        </table>
      </main>
    </body>
  </html>
  `;
}
function html(strings, ...values) {
    const l = strings.length - 1;
    let html = "";
    for (let i = 0; i < l; i++) {
        let v = values[i];
        if (v instanceof Array) {
            v = v.join("");
        }
        const s = strings[i] + v;
        html += s;
    }
    html += strings[l];
    return html;
}
function main() {
    listenAndServe(addr, async (req) => {
        let normalizedUrl = posix.normalize(req.url);
        try {
            normalizedUrl = decodeURIComponent(normalizedUrl);
        }
        catch (e) {
            if (!(e instanceof URIError)) {
                throw e;
            }
        }
        const fsPath = posix.join(target, normalizedUrl);
        let response;
        try {
            const fileInfo = await stat(fsPath);
            if (fileInfo.isDirectory) {
                response = await serveDir(req, fsPath);
            }
            else {
                response = await serveFile(req, fsPath);
            }
        }
        catch (e) {
            console.error(e.message);
            response = await serveFallback(req, e);
        }
        finally {
            if (CORSEnabled) {
                assert(response);
                setCORS(response);
            }
```

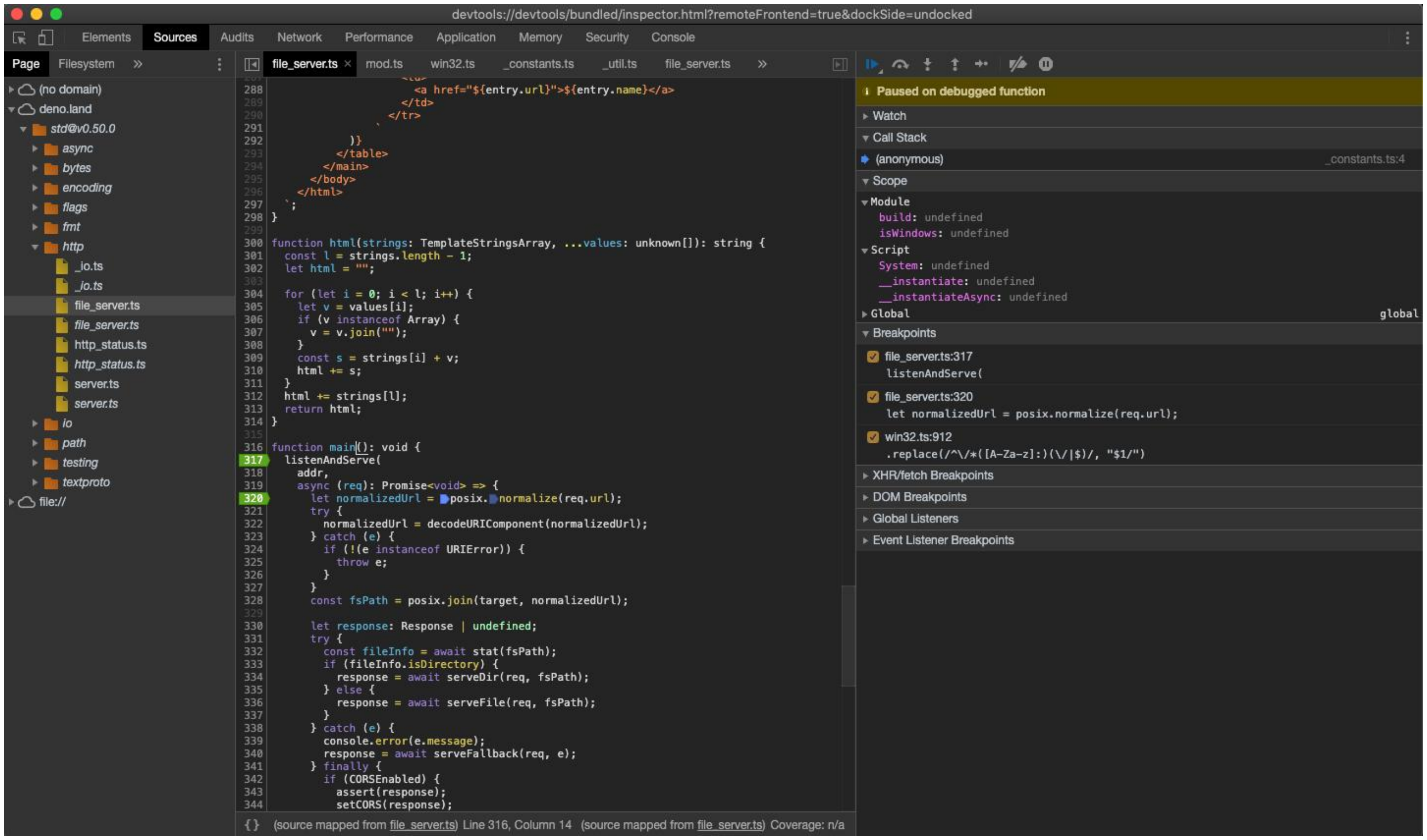
```
▾Module
   build: undefined
   isWindows: undefined
▾Script
   System: undefined
   __instantiate: undefined
   __instantiateAsync: undefined
▸Global                                  global
```

```
file_server.ts:317
listenAndServe(
file_server.ts:320
let normalizedUrl = posix.normalize(req.url);
win32.ts:912
.replace(/^\/*([A-Za-z]:)(\/|$)/, "$1/")
```

Figure 3: Open file_server.ts

Looking closely you'll find duplicate entries for each file; one written regularly and one in italics. The former is compiled source file (so in the case of `.ts` *files it will be emitted JavaScript source), while the latter is a source map for the file.*

Next, add a breakpoint in the `listenAndServe` method:



```
288                 <a href="${entry.url}">${entry.name}</a>
289               </td>
290             </tr>
291           `
292         )}
293       </table>
294     </main>
295   </body>
296 </html>
297   `;
298 }
299
300 function html(strings: TemplateStringsArray, ...values: unknown[]): string {
301   const l = strings.length - 1;
302   let html = "";
303
304   for (let i = 0; i < l; i++) {
305     let v = values[i];
306     if (v instanceof Array) {
307       v = v.join("");
308     }
309     const s = strings[i] + v;
310     html += s;
311   }
312   html += strings[l];
313   return html;
314 }
315
316 function main(): void {
317   listenAndServe(
318     addr,
319     async (req): Promise<void> => {
320       let normalizedUrl = posix.normalize(req.url);
321       try {
322         normalizedUrl = decodeURIComponent(normalizedUrl);
323       } catch (e) {
324         if (!(e instanceof URIError)) {
325           throw e;
326         }
327       }
328       const fsPath = posix.join(target, normalizedUrl);
329
330       let response: Response | undefined;
331       try {
332         const fileInfo = await stat(fsPath);
333         if (fileInfo.isDirectory) {
334           response = await serveDir(req, fsPath);
335         } else {
336           response = await serveFile(req, fsPath);
337         }
338       } catch (e) {
339         console.error(e.message);
340         response = await serveFallback(req, e);
341       } finally {
342         if (CORSEnabled) {
343           assert(response);
344           setCORS(response);
```

{} (source mapped from file_server.ts) Line 316, Column 14 (source mapped from file_server.ts) Coverage: n/a

```
Module
  build: undefined
  isWindows: undefined
Script
  System: undefined
  __instantiate: undefined
  __instantiateAsync: undefined
Global                                   global
```

Breakpoints

- ✓ file_server.ts:317
  `listenAndServe(`
- ✓ file_server.ts:320
  `let normalizedUrl = posix.normalize(req.url);`
- ✓ win32.ts:912
  `.replace(/^\/*([A-Za-z]:)(\/|$)/, "$1/")`

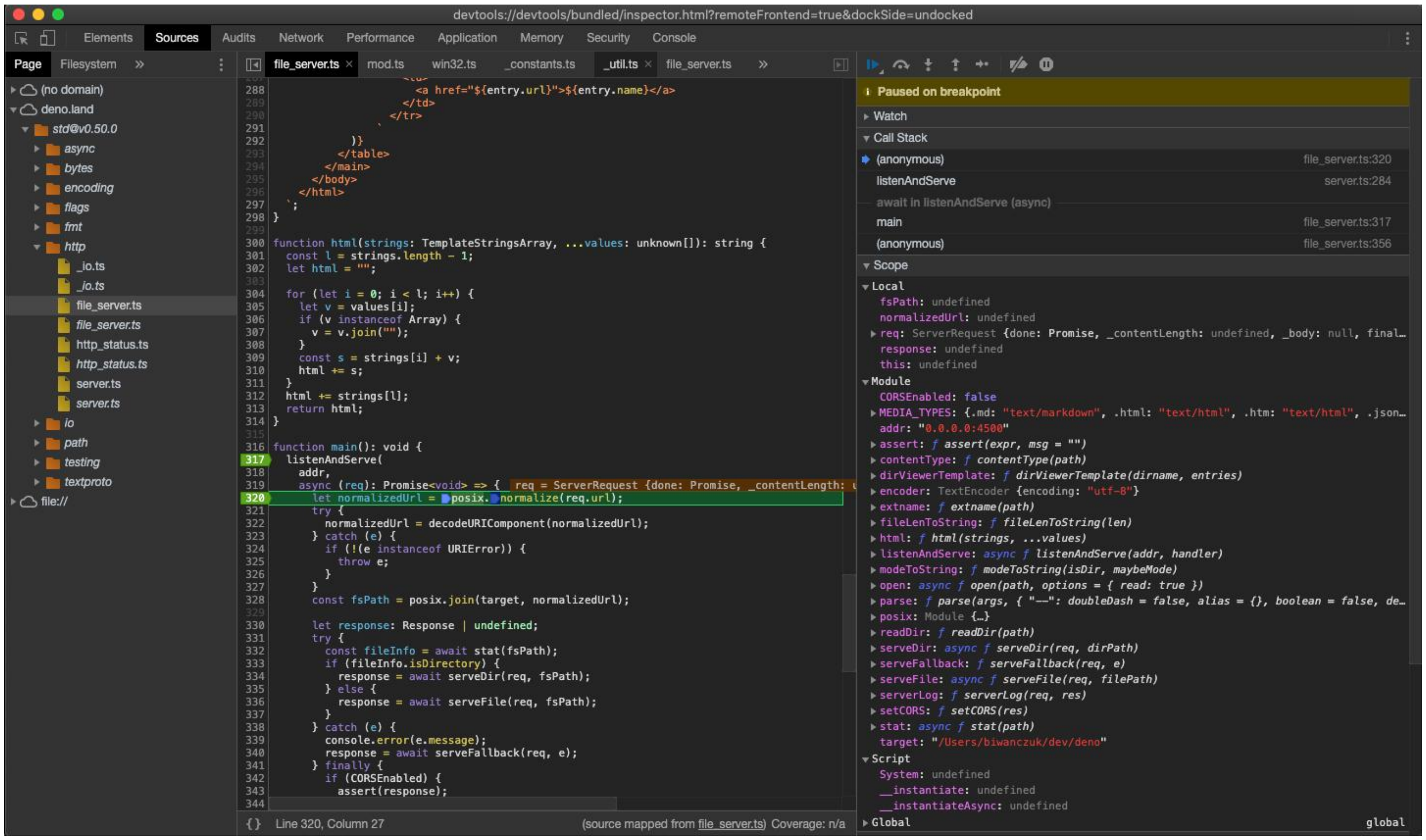
Figure 4: Break in file_server.ts

As soon as we've added the breakpoint Devtools automatically opened up the source map file, which allows us step through the actual source code that includes types.

Now that we have our breakpoints set, we can resume the execution of our script so that we might inspect an incoming request. Hit the Resume script execution button to do so. You might even need to hit it twice!

Once our script is running again, let's send a request and inspect it in Devtools:

```
$ curl http://0.0.0.0:4500/
```



```
              <a href="${entry.url}">${entry.name}</a>
            </td>
          </tr>
        `
      )}
    </table>
  </main>
</body>
</html>
`;
}

function html(strings: TemplateStringsArray, ...values: unknown[]): string {
  const l = strings.length - 1;
  let html = "";

  for (let i = 0; i < l; i++) {
    let v = values[i];
    if (v instanceof Array) {
      v = v.join("");
    }
    const s = strings[i] + v;
    html += s;
  }
  html += strings[l];
  return html;
}

function main(): void {
  listenAndServe(
    addr,
    async (req): Promise<void> => {  req = ServerRequest {done: Promise, _contentLength: u
      let normalizedUrl = posix.normalize(req.url);
      try {
        normalizedUrl = decodeURIComponent(normalizedUrl);
      } catch (e) {
        if (!(e instanceof URIError)) {
          throw e;
        }
      }
      const fsPath = posix.join(target, normalizedUrl);

      let response: Response | undefined;
      try {
        const fileInfo = await stat(fsPath);
        if (fileInfo.isDirectory) {
          response = await serveDir(req, fsPath);
        } else {
          response = await serveFile(req, fsPath);
        }
      } catch (e) {
        console.error(e.message);
        response = await serveFallback(req, e);
      } finally {
        if (CORSEnabled) {
          assert(response);
```

```
{}  Line 320, Column 27                    (source mapped from file_server.ts) Coverage: n/a
```

```
(anonymous)                                  file_server.ts:320
listenAndServe                               server.ts:284
await in listenAndServe (async)
main                                         file_server.ts:317
(anonymous)                                  file_server.ts:356
```

```
▼ Local
    fsPath: undefined
    normalizedUrl: undefined
  ▶ req: ServerRequest {done: Promise, _contentLength: undefined, _body: null, final…
    response: undefined
    this: undefined
▼ Module
    CORSEnabled: false
  ▶ MEDIA_TYPES: {.md: "text/markdown", .html: "text/html", .htm: "text/html", .json…
    addr: "0.0.0.0:4500"
  ▶ assert: ƒ assert(expr, msg = "")
  ▶ contentType: ƒ contentType(path)
  ▶ dirViewerTemplate: ƒ dirViewerTemplate(dirname, entries)
  ▶ encoder: TextEncoder {encoding: "utf-8"}
  ▶ extname: ƒ extname(path)
  ▶ fileLenToString: ƒ fileLenToString(len)
  ▶ html: ƒ html(strings, ...values)
  ▶ listenAndServe: async ƒ listenAndServe(addr, handler)
  ▶ modeToString: ƒ modeToString(isDir, maybeMode)
  ▶ open: async ƒ open(path, options = { read: true })
  ▶ parse: ƒ parse(args, { "--": doubleDash = false, alias = {}, boolean = false, de…
  ▶ posix: Module {…}
  ▶ readDir: ƒ readDir(path)
  ▶ serveDir: async ƒ serveDir(req, dirPath)
  ▶ serveFallback: ƒ serveFallback(req, e)
  ▶ serveFile: async ƒ serveFile(req, filePath)
  ▶ serverLog: ƒ serverLog(req, res)
  ▶ setCORS: ƒ setCORS(res)
  ▶ stat: async ƒ stat(path)
    target: "/Users/biwanczuk/dev/deno"
▼ Script
    System: undefined
    __instantiate: undefined
    __instantiateAsync: undefined
▶ Global                                                                   global
```

Figure 5: Break in request handling

At this point we can introspect the contents of the request and go step-by-step to debug the code.

VSCode

Deno can be debugged using VSCode.

Official support via the plugin is being worked on - https://github.com/de

We can still attach the debugger by manually providing a `launch.json` config:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Deno",
      "type": "pwa-node",
      "request": "launch",
      "cwd": "${workspaceFolder}",
      "runtimeExecutable": "deno",
      "runtimeArgs": ["run", "--inspect-brk", "-A", "${
        ↪ file}"],
      "attachSimplePort": 9229
    }
  ]
}
```

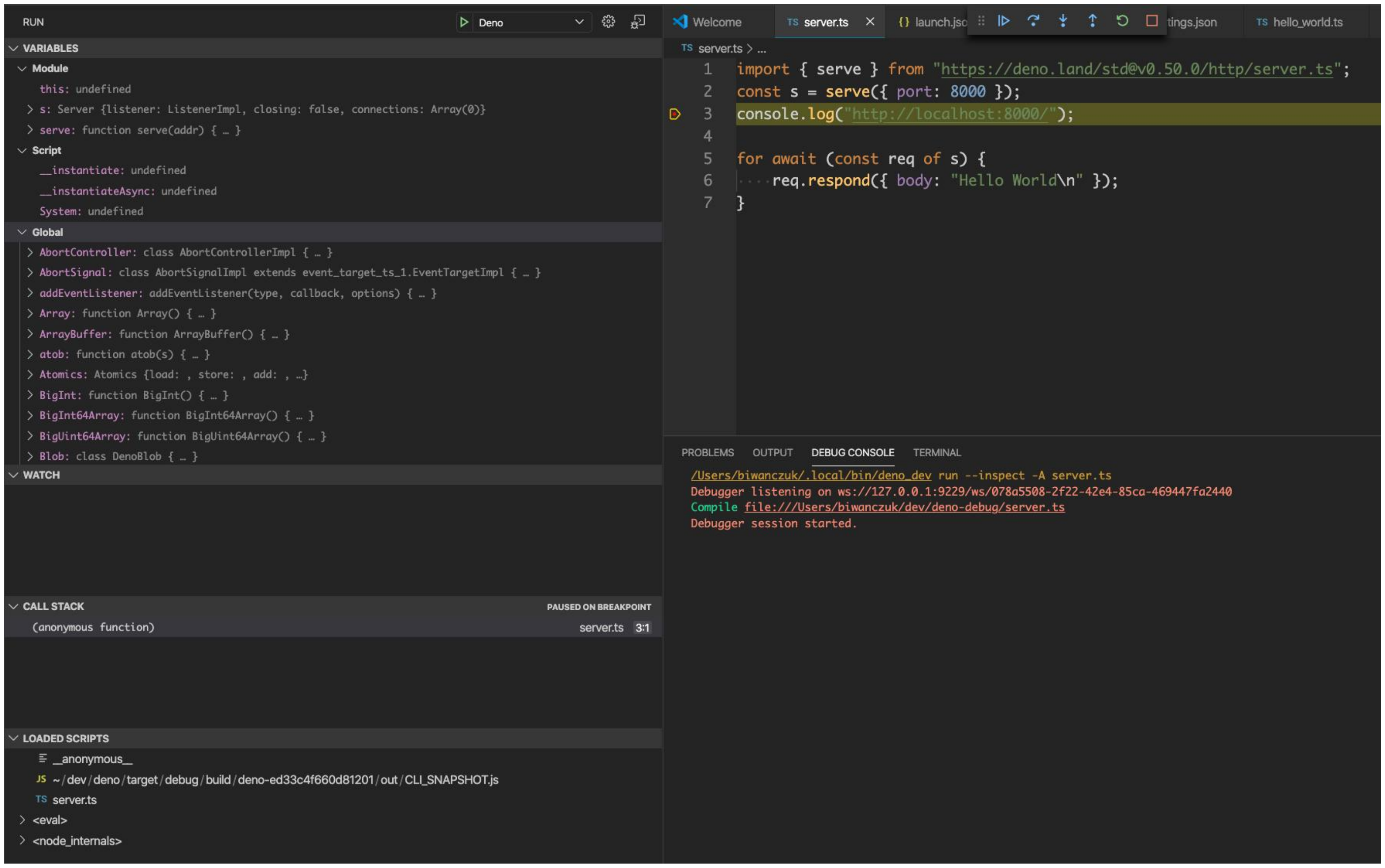
NOTE: This uses the file you have open as the entry point; replace `${file}` with a script name if you want a fixed entry point.

Let's try out debugging a local source file. Create `server.ts`:

```
import { serve } from "https://deno.land/
  ↪ std@$STD_VERSION/http/server.ts";
const server = serve({ port: 8000 });
console.log("http://localhost:8000/");

for await (const req of server) {
  req.respond({ body: "Hello World\n" });
}
```

Then we can set a breakpoint, and run the created configuration:



```
this: undefined
> s: Server {listener: ListenerImpl, closing: false, connections: Array(0)}
> serve: function serve(addr) { … }
__instantiate: undefined
__instantiateAsync: undefined
System: undefined
> AbortController: class AbortControllerImpl { … }
> AbortSignal: class AbortSignalImpl extends event_target_ts_1.EventTargetImpl { … }
> addEventListener: addEventListener(type, callback, options) { … }
> Array: function Array() { … }
> ArrayBuffer: function ArrayBuffer() { … }
> atob: function atob(s) { … }
> Atomics: Atomics {load: , store: , add: , …}
> BigInt: function BigInt() { … }
> BigInt64Array: function BigInt64Array() { … }
> BigUint64Array: function BigUint64Array() { … }
> Blob: class DenoBlob { … }
```

```
(anonymous function)                server.ts 3:1
```

```
__anonymous__
~/dev/deno/target/debug/build/deno-ed33c4f660d81201/out/CLI_SNAPSHOT.js
server.ts
> <eval>
> <node_internals>
```

```
import { serve } from "https://deno.land/std@v0.50.0/http/server.ts";
const s = serve({ port: 8000 });
console.log("http://localhost:8000/");

for await (const req of s) {
    req.respond({ body: "Hello World\n" });
}
```

```
/Users/biwanczuk/.local/bin/deno_dev run --inspect -A server.ts
Debugger listening on ws://127.0.0.1:9229/ws/078a5508-2f22-42e4-85ca-469447fa2440
Compile file:///Users/biwanczuk/dev/deno-debug/server.ts
Debugger session started.
```

Figure 6: VSCode debugger

JetBrains IDEs

You can debug Deno using your JetBrains IDE by right-clicking the file you want to debug and selecting the `Debug 'Deno: <file name>'` option. This will create a run/debug configuration with no permission flags set. To configure these flags edit the run/debug configuration and modify the `Arguments` field with the required flags.

Other

Any client that implements the Devtools protocol should be able to connect to a Deno process.

Limitations

Devtools support is still immature. There is some functionality that is known to be missing or buggy:

- autocomplete in Devtools' console causes the Deno process to exit
- profiling and memory dumps might not work correctly

Script installer

Deno provides `deno install` to easily install and distribute executable code.

`deno install [OPTIONS...] [URL] [SCRIPT_ARGS...]` will install the script available at `URL` under the name `EXE_NAME`.

This command creates a thin, executable shell script which invokes `deno` ↪ using the specified CLI flags and main module. It is placed in the installation root's `bin` directory.

Example:

```
$ deno install --allow-net --allow-read https://deno.
  ↪ land/std@$STD_VERSION/http/file_server.ts
[1/1] Compiling https://deno.land/std@$STD_VERSION/http/
  ↪ file_server.ts

 Successfully installed file_server.
/Users/deno/.deno/bin/file_server
```

To change the executable name, use `-n/--name`:

```
deno install --allow-net --allow-read -n serve https://
  ↪ deno.land/std@$STD_VERSION/http/file_server.ts
```

The executable name is inferred by default:

- Attempt to take the file stem of the URL path. The above example would become 'file_server'.
- If the file stem is something generic like 'main', 'mod', 'index' or 'cli', and the path has no parent, take the file name of the parent path. Otherwise settle with the generic name.
- If the resulting name has an '@...' suffix, strip it.

To change the installation root, use `--root`:

```
deno install --allow-net --allow-read --root /usr/local
  ↪ https://deno.land/std@$STD_VERSION/http/file_server
  ↪ .ts
```

The installation root is determined, in order of precedence:

- `--root` option
- `DENO_INSTALL_ROOT` environment variable
- `$HOME/.deno`

These must be added to the path manually if required.

```
echo 'export PATH="$HOME/.deno/bin:$PATH"' >> ~/.bashrc
```

You must specify permissions that will be used to run the script at installation time.

```
deno install --allow-net --allow-read https://deno.land/
  ↪ std@$STD_VERSION/http/file_server.ts -p 8080
```

The above command creates an executable called `file_server` that runs with network and read permissions and binds to port 8080.

For good practice, use the `import.meta.main` idiom to specify the entry point in an executable script.

Example:

```
// https://example.com/awesome/cli.ts
async function myAwesomeCli(): Promise<void> {
  -- snip --
}

if (import.meta.main) {
  myAwesomeCli();
}
```

When you create an executable script make sure to let users know by adding an example installation command to your repository:

```
# Install using deno install

$ deno install -n awesome_cli https://example.com/
  ↪ awesome/cli.ts
```

Code formatter

Deno ships with a built in code formatter that auto-formats TypeScript and JavaScript code.

```
# format all JS/TS files in the current directory and
  ↪ subdirectories
deno fmt
# format specific files
deno fmt myfile1.ts myfile2.ts
```

```
# check if all the JS/TS files in the current directory
  ↪ and subdirectories are formatted
deno fmt --check
# format stdin and write to stdout
cat file.ts | deno fmt -
```

Ignore formatting code by preceding it with a `// deno-fmt-ignore` comment:

```
// deno-fmt-ignore
export const identity = [
    1, 0, 0,
    0, 1, 0,
    0, 0, 1,
];
```

Or ignore an entire file by adding a `// deno-fmt-ignore-file` comment at the top of the file.

Bundling

`deno bundle [URL]` will output a single JavaScript file, which includes all dependencies of the specified input. For example:

```
> deno bundle https://deno.land/std@$STD_VERSION/
  ↪ examples/colors.ts colors.bundle.js
Bundling "colors.bundle.js"
Emitting bundle to "colors.bundle.js"
9.2 kB emitted.
```

If you omit the out file, the bundle will be sent to `stdout`.

The bundle can just be run as any other module in Deno would:

```
deno run colors.bundle.js
```

The output is a self contained ES Module, where any exports from the main module supplied on the command line will be available. For example, if the main module looked something like this:

```
export { foo } from "./foo.js";

export const bar = "bar";
```

It could be imported like this:

```
import { foo, bar } from "./lib.bundle.js";
```

Bundles can also be loaded in the web browser. The bundle is a self-contained ES module, and so the attribute of `type` must be set to `"`
↪ `module"`. For example:

```
<script type="module" src="website.bundle.js"></script>
```

Or you could import it into another ES module to consume:

```
<script type="module">
  import * as website from "website.bundle.js";
</script>
```

Documentation Generator

`deno doc` followed by a list of one or more source files will print the JSDoc documentation for each of the module's **exported** members. Currently, only exports in the style `export <declaration>` and `export`
↪ `... from ...` are supported.

For example, given a file `add.ts` with the contents:

```
/**
 * Adds x and y.
 * @param {number} x
 * @param {number} y
 * @returns {number} Sum of x and y
 */
export function add(x: number, y: number): number {
  return x + y;
}
```

Running the Deno `doc` command, prints the function's JSDoc comment to `stdout`:

```
deno doc add.ts
function add(x: number, y: number): number
  Adds x and y. @param {number} x @param {number} y
    ↪ @returns {number} Sum of x and y
```

Use the `--json` flag to output the documentation in JSON format. This JSON format is consumed by the deno doc website and used to generate module documentation.

Dependency Inspector

`deno info [URL]` will inspect ES module and all of its dependencies.

```
deno info https://deno.land/std@0.52.0/http/file_server.
  ↪ ts
Download https://deno.land/std@0.52.0/http/file_server.
  ↪ ts
...
local: /Users/deno/Library/Caches/deno/deps/https/deno.
  ↪ land/5
  ↪ bd138988e9d20db1a436666628ffb3f7586934e0a2a9fe2a7b7bf4
  ↪
type: TypeScript
```

```
compiled: /Users/deno/Library/Caches/deno/gen/https/deno
  ↪ .land/std@0.52.0/http/file_server.ts.js
map: /Users/deno/Library/Caches/deno/gen/https/deno.land
  ↪ /std@0.52.0/http/file_server.ts.js.map
deps:
https://deno.land/std@0.52.0/http/file_server.ts
   https://deno.land/std@0.52.0/path/mod.ts
    https://deno.land/std@0.52.0/path/win32.ts
     https://deno.land/std@0.52.0/path/_constants.ts
     https://deno.land/std@0.52.0/path/_util.ts
      https://deno.land/std@0.52.0/path/_constants.ts
     https://deno.land/std@0.52.0/testing/asserts.ts
       https://deno.land/std@0.52.0/fmt/colors.ts
       https://deno.land/std@0.52.0/testing/diff.ts
    https://deno.land/std@0.52.0/path/posix.ts
     https://deno.land/std@0.52.0/path/_constants.ts
     https://deno.land/std@0.52.0/path/_util.ts
    https://deno.land/std@0.52.0/path/common.ts
     https://deno.land/std@0.52.0/path/separator.ts
    https://deno.land/std@0.52.0/path/separator.ts
    https://deno.land/std@0.52.0/path/interface.ts
    https://deno.land/std@0.52.0/path/glob.ts
      https://deno.land/std@0.52.0/path/separator.ts
      https://deno.land/std@0.52.0/path/_globrex.ts
      https://deno.land/std@0.52.0/path/mod.ts
      https://deno.land/std@0.52.0/testing/asserts.ts
   https://deno.land/std@0.52.0/http/server.ts
    https://deno.land/std@0.52.0/encoding/utf8.ts
    https://deno.land/std@0.52.0/io/bufio.ts
     https://deno.land/std@0.52.0/io/util.ts
      https://deno.land/std@0.52.0/path/mod.ts
      https://deno.land/std@0.52.0/encoding/utf8.ts
     https://deno.land/std@0.52.0/testing/asserts.ts
    https://deno.land/std@0.52.0/testing/asserts.ts
    https://deno.land/std@0.52.0/async/mod.ts
     https://deno.land/std@0.52.0/async/deferred.ts
     https://deno.land/std@0.52.0/async/delay.ts
     https://deno.land/std@0.52.0/async/
       ↪ mux_async_iterator.ts
       https://deno.land/std@0.52.0/async/deferred.ts
    https://deno.land/std@0.52.0/http/_io.ts
      https://deno.land/std@0.52.0/io/bufio.ts
      https://deno.land/std@0.52.0/textproto/mod.ts
       https://deno.land/std@0.52.0/io/util.ts
       https://deno.land/std@0.52.0/bytes/mod.ts
        https://deno.land/std@0.52.0/io/util.ts
       https://deno.land/std@0.52.0/encoding/utf8.ts
      https://deno.land/std@0.52.0/testing/asserts.ts
      https://deno.land/std@0.52.0/encoding/utf8.ts
      https://deno.land/std@0.52.0/http/server.ts
      https://deno.land/std@0.52.0/http/http_status.ts
   https://deno.land/std@0.52.0/flags/mod.ts
    https://deno.land/std@0.52.0/testing/asserts.ts
   https://deno.land/std@0.52.0/testing/asserts.ts
```

Dependency inspector works with any local or remote ES modules.

Cache location

`deno info` can be used to display information about cache location:

```
deno info
DENO_DIR location: "/Users/deno/Library/Caches/deno"
Remote modules cache: "/Users/deno/Library/Caches/deno/
  ↪ deps"
TypeScript compiler cache: "/Users/deno/Library/Caches/
  ↪ deno/gen"
```

Linter

Deno ships with a built in code linter for JavaScript and TypeScript.

Note: linter is a new feature and still unstable thus it requires `--unstable` flag

```
# lint all JS/TS files in the current directory and
  ↪ subdirectories
deno lint --unstable
# lint specific files
deno lint --unstable myfile1.ts myfile2.ts
# print result as JSON
deno lint --unstable --json
# read from stdin
cat file.ts | deno lint --unstable -
```

For more detail, run `deno lint --help`.

Available rules

- `adjacent-overload-signatures`
- `ban-ts-comment`
- `ban-types`
- `ban-untagged-ignore`
- `constructor-super`
- `for-direction`
- `getter-return`
- `no-array-constructor`
- `no-async-promise-executor`
- `no-case-declarations`
- `no-class-assign`
- `no-compare-neg-zero`

- `no-cond-assign`
- `no-constant-condition`
- `no-control-regex`
- `no-debugger`
- `no-delete-var`
- `no-dupe-args`
- `no-dupe-class-members`
- `no-dupe-else-if`
- `no-dupe-keys`
- `no-duplicate-case`
- `no-empty`
- `no-empty-character-class`
- `no-empty-interface`
- `no-empty-pattern`
- `no-ex-assign`
- `no-explicit-any`
- `no-extra-boolean-cast`
- `no-extra-non-null-assertion`
- `no-extra-semi`
- `no-func-assign`
- `no-inferrable-types`
- `no-invalid-regexp`
- `no-irregular-whitespace`
- `no-misused-new`
- `no-mixed-spaces-and-tabs`
- `no-namespace`
- `no-new-symbol`
- `no-obj-call`
- `no-octal`

- `no-prototype-builtins`
- `no-regex-spaces`
- `no-self-assign`
- `no-setter-return`
- `no-shadow-restricted-names`
- `no-this-alias`
- `no-this-before-super`
- `no-unexpected-multiline`
- `no-unsafe-finally`
- `no-unsafe-negation`
- `no-unused-labels`
- `no-with`
- `prefer-as-const`
- `prefer-namespace-keyword`
- `require-yield`
- `triple-slash-reference`
- `use-isnan`
- `valid-typeof`

Ignore directives

Files To ignore whole file `// deno-lint-ignore-file` directive should placed at the top of the file:

```
// deno-lint-ignore-file

function foo(): any {
  // ...
}
```

Ignore directive must be placed before first stament or declaration:

```
// Copyright 2020 the Deno authors. All rights reserved.
  ↪  MIT license.

/**
 * Some JS doc
 **/

// deno-lint-ignore-file

import { bar } from "./bar.js";

function foo(): any {
  // ...
}
```

Diagnostics To ignore certain diagnostic `// deno-lint-ignore ↪ <codes...>` directive should be placed before offending line. Specifying ignored rule name is required:

```
// deno-lint-ignore no-explicit-any
function foo(): any {
  // ...
}

// deno-lint-ignore no-explicit-any explicit-function-
  ↪ return-type
function bar(a: any) {
  // ...
}
```

To provide some compatibility with ESLint `deno lint` also supports `// ↪ eslint-disable-next-line` directive. Just like with `// deno-lint- ↪ ignore`, it's required to specify the ignored rule name:

```
// eslint-disable-next-line no-empty
while (true) {}

// eslint-disable-next-line @typescript-eslint/no-
  ↪ explicit-any
function bar(a: any) {
  // ...
}
```

Embedding Deno

Deno consists of multiple parts, one of which is `deno_core`. This is a rust crate that can be used to embed a JavaScript runtime into your rust application. Deno is built on top of `deno_core`.

The Deno crate is hosted on crates.io.

You can view the API on docs.rs.

Contributing

- Read the style guide.
- Please don't make the benchmarks worse.
- Ask for help in the community chat room.
- If you are going to work on an issue, mention so in the issue comments *before* you start working on the issue.
- Please be professional in the forums. We follow Rust's code of conduct (CoC). Have a problem? Email ry@tinyclouds.org.

Development

Instructions on how to build from source can be found here.

Submitting a Pull Request

Before submitting, please make sure the following is done:

1. That there is a related issue and it is referenced in the PR text.
2. There are tests that cover the changes.
3. Ensure `cargo test` passes.
4. Format your code with `./tools/format.py`
5. Make sure `./tools/lint.py` passes.

Changes to `third_party`

`deno_third_party` contains most of the external code that Deno depends on, so that we know exactly what we are executing at any given time. It is carefully maintained with a mixture of manual labor and private scripts. It's likely you will need help from @ry or @piscisaureus to make changes.

Adding Ops (aka bindings)

We are very concerned about making mistakes when adding new APIs. When adding an Op to Deno, the counterpart interfaces on other platforms should be researched. Please list how this functionality is done in Go, Node, Rust, and Python.

As an example, see how `Deno.rename()` was proposed and added in PR #671.

Releases

Summary of the changes from previous releases can be found here.

Documenting APIs

It is important to document public APIs and we want to do that inline with the code. This helps ensure that code and documentation are tightly coupled together.

Utilize JSDoc

All publicly exposed APIs and types, both via the `deno` module as well as the global/`window` namespace should have JSDoc documentation. This documentation is parsed and available to the TypeScript compiler, and therefore easy to provide further downstream. JSDoc blocks come just prior to the statement they apply to and are denoted by a leading `/**` before terminating with a `*/`. For example:

```
/** A simple JSDoc comment */
export const FOO = "foo";
```

Find more at https://jsdoc.app/

Building from source

Below are instructions on how to build Deno from source. If you just want to use Deno you can download a prebuilt executable (more information in the `Getting Started` chapter).

Cloning the Repository

Clone on Linux or Mac:

```
git clone --recurse-submodules https://github.com/
  ↪ denoland/deno.git
```

Extra steps for Windows users:

1. Enable “Developer Mode” (otherwise symlinks would require administrator privileges).

2. Make sure you are using git version 2.19.2.windows.1 or newer.

3. Set `core.symlinks=true` before the checkout:

   ```
   git config --global core.symlinks true
   git clone --recurse-submodules https://github.com/
     ↪ denoland/deno.git
   ```

Prerequisites

> Deno requires the progressively latest stable release of Rust.
> Deno does not support the Rust nightlies.

Update or Install Rust. Check that Rust installed/updated correctly:

```
rustc -V
cargo -V
```

Setup rust targets and components

```
rustup target add wasm32-unknown-unknown
rustup target add wasm32-wasi
```

Building Deno

The easiest way to build Deno is by using a precompiled version of V8:

```
cargo build -vv
```

However if you want to build Deno and V8 from source code:

```
V8_FROM_SOURCE=1 cargo build -vv
```

When building V8 from source, there are more dependencies:

Python 2. Ensure that a suffix-less `python`/`python.exe` exists in your `PATH` and it refers to Python 2, not 3.

For Linux users glib-2.0 development files must also be installed. (On Ubuntu, run `apt install libglib2.0-dev`.)

Mac users must have Command Line Tools installed. (XCode already includes CLT. Run `xcode-select --install` to install it without XCode.)

For Windows users:

1. Get VS Community 2019 with "Desktop development with C++" toolkit and make sure to select the following required tools listed below along with all C++ tools.

    - Visual C++ tools for CMake
    - Windows 10 SDK (10.0.17763.0)
    - Testing tools core features - Build Tools
    - Visual C++ ATL for x86 and x64
    - Visual C++ MFC for x86 and x64
    - C++/CLI support
    - VC++ 2015.3 v14.00 (v140) toolset for desktop

2. Enable “Debugging Tools for Windows”. Go to “Control Panel” → “Programs” → “Programs and Features” → Select “Windows Software Development Kit - Windows 10” → “Change” → “Change” → Check “Debugging Tools For Windows” → “Change” → “Finish”. Or use: Debugging Tools for Windows (Notice: it will download the files, you should install `X64 Debuggers And Tools-x64_en-us.msi` file manually.)

See rusty_v8’s README for more details about the V8 build.

Building

Build with Cargo:

```
# Build:
cargo build -vv

# Build errors?  Ensure you have latest master and try
  ↪ building again, or if that doesn't work try:
cargo clean && cargo build -vv

# Run:
./target/debug/deno run cli/tests/002_hello.ts
```

Testing and Tools

Tests

Test `deno`:

```
# Run the whole suite:
cargo test

# Only test cli/js/:
```

```
cargo test js_unit_tests
```

Test `std/`:

```
cargo test std_tests
```

Lint and format

Lint the code:

```
./tools/lint.py
```

Format the code:

```
./tools/format.py
```

Profiling

To start profiling:

```
# Make sure we're only building release.
# Build deno and V8's d8.
ninja -C target/release d8

# Start the program we want to benchmark with --prof
./target/release/deno run tests/http_bench.ts --allow-
  ↪ net --v8-flags=--prof &

# Exercise it.
third_party/wrk/linux/wrk http://localhost:4500/
kill `pgrep deno`
```

V8 will write a file in the current directory that looks like this: `isolate`
↪ `-0x7fad98242400-v8.log`. To examine this file:

```
D8_PATH=target/release/ ./third_party/v8/tools/linux-
  ↪ tick-processor
isolate-0x7fad98242400-v8.log > prof.log
# on macOS, use ./third_party/v8/tools/mac-tick-
  ↪ processor instead
```

`prof.log` will contain information about tick distribution of different calls.

To view the log with Web UI, generate JSON file of the log:

```
D8_PATH=target/release/ ./third_party/v8/tools/linux-
  ↪ tick-processor
isolate-0x7fad98242400-v8.log --preprocess > prof.json
```

Open `third_party/v8/tools/profview/index.html` in your browser, and select `prof.json` to view the distribution graphically.

Useful V8 flags during profiling:

- –prof
- –log-internal-timer-events
- –log-timer-events
- –track-gc
- –log-source-code
- –track-gc-object-stats

To learn more about `d8` and profiling, check out the following links:

- https://v8.dev/docs/d8
- https://v8.dev/docs/profile

Debugging with LLDB

To debug the deno binary, we can use `rust-lldb`. It should come with `rustc` and is a wrapper around LLDB.

```
$ rust-lldb -- ./target/debug/deno run --allow-net tests
  ↪ /http_bench.ts
# On macOS, you might get warnings like
# `ImportError: cannot import name _remove_dead_weakref`
# In that case, use system python by setting PATH, e.g.
# PATH=/System/Library/Frameworks/Python.framework/
  ↪ Versions/2.7/bin:$PATH
(lldb) command script import "/Users/kevinqian/.rustup/
  ↪ toolchains/1.36.0-x86_64-apple-darwin/lib/rustlib/
  ↪ etc/lldb_rust_formatters.py"
(lldb) type summary add --no-value --python-function
  ↪ lldb_rust_formatters.print_val -x ".*" --category
  ↪ Rust
(lldb) type category enable Rust
(lldb) target create "../deno/target/debug/deno"
Current executable set to '../deno/target/debug/deno' (
  ↪ x86_64).
(lldb) settings set -- target.run-args  "tests/
  ↪ http_bench.ts" "--allow-net"
(lldb) b op_start
(lldb) r
```

V8 flags

V8 has many many internal command-line flags:

```
$ deno run --v8-flags=--help _
SSE3=1 SSSE3=1 SSE4_1=1 SSE4_2=1 SAHF=1 AVX=1 FMA3=1
  ↪ BMI1=1 BMI2=1 LZCNT=1 POPCNT=1 ATOM=0
Synopsis:
  shell [options] [--shell] [<file>...]
  d8 [options] [-e <string>] [--shell] [[--module] <file
    ↪ >...]

  -e        execute a string in V8
  --shell   run an interactive JavaScript shell
  --module  execute a file as a JavaScript module

Note: the --module option is implicitly enabled for *.
  ↪ mjs files.

The following syntax for options is accepted (both '-'
  ↪ and '--' are ok):
  --flag         (bool flags only)
  --no-flag      (bool flags only)
  --flag=value   (non-bool flags only, no spaces around
    ↪ '=')
  --flag value   (non-bool flags only)
  --             (captures all remaining args in
    ↪ JavaScript)

Options:
  --use-strict (enforce strict mode)
        type: bool  default: false
  --es-staging (enable test-worthy harmony features (for
    ↪  internal use only))
        type: bool  default: false
  --harmony (enable all completed harmony features)
        type: bool  default: false
  --harmony-shipping (enable all shipped harmony
    ↪ features)
        type: bool  default: true
```

```
--harmony-regexp-sequence (enable "RegExp Unicode
  ↪ sequence properties" (in progress))
      type: bool  default: false
--harmony-weak-refs-with-cleanup-some (enable "harmony
  ↪  weak references with FinalizationRegistry.
  ↪ prototype.cleanupSome" (in progress))
      type: bool  default: false
--harmony-regexp-match-indices (enable "harmony regexp
  ↪  match indices" (in progress))
      type: bool  default: false
--harmony-top-level-await (enable "harmony top level
  ↪ await")
      type: bool  default: false
--harmony-namespace-exports (enable "harmony namespace
  ↪  exports (export * as foo from 'bar')")
      type: bool  default: true
--harmony-sharedarraybuffer (enable "harmony
  ↪ sharedarraybuffer")
      type: bool  default: true
--harmony-import-meta (enable "harmony import.meta
  ↪ property")
      type: bool  default: true
--harmony-dynamic-import (enable "harmony dynamic
  ↪ import")
      type: bool  default: true
--harmony-promise-all-settled (enable "harmony Promise
  ↪ .allSettled")
      type: bool  default: true
--harmony-promise-any (enable "harmony Promise.any")
      type: bool  default: true
--harmony-private-methods (enable "harmony private
  ↪ methods in class literals")
      type: bool  default: true
```

```
--harmony-weak-refs (enable "harmony weak references")
      type: bool  default: true
--harmony-string-replaceall (enable "harmony String.
  ↪ prototype.replaceAll")
      type: bool  default: true
--harmony-logical-assignment (enable "harmony logical
  ↪ assignment")
      type: bool  default: true
--lite-mode (enables trade-off of performance for
  ↪ memory savings)
      type: bool  default: false
--future (Implies all staged features that we want to
  ↪ ship in the not-too-far future)
      type: bool  default: false
--assert-types (generate runtime type assertions to
  ↪ test the typer)
      type: bool  default: false
--allocation-site-pretenuring (pretenure with
  ↪ allocation sites)
      type: bool  default: true
--page-promotion (promote pages based on utilization)
      type: bool  default: true
--always-promote-young-mc (always promote young
  ↪ objects during mark-compact)
      type: bool  default: true
--page-promotion-threshold (min percentage of live
  ↪ bytes on a page to enable fast evacuation)
      type: int  default: 70
--trace-pretenuring (trace pretenuring decisions of
  ↪ HAllocate instructions)
      type: bool  default: false
--trace-pretenuring-statistics (trace allocation site
  ↪ pretenuring statistics)
```

```
      type: bool  default: false
--track-fields (track fields with only smi values)
      type: bool  default: true
--track-double-fields (track fields with double values
  ↪ )
      type: bool  default: true
--track-heap-object-fields (track fields with heap
  ↪ values)
      type: bool  default: true
--track-computed-fields (track computed boilerplate
  ↪ fields)
      type: bool  default: true
--track-field-types (track field types)
      type: bool  default: true
--trace-block-coverage (trace collected block coverage
  ↪  information)
      type: bool  default: false
--trace-protector-invalidation (trace protector cell
  ↪ invalidations)
      type: bool  default: false
--feedback-normalization (feed back normalization to
  ↪ constructors)
      type: bool  default: false
--enable-one-shot-optimization (Enable size
  ↪ optimizations for the code that will only be
  ↪ executed once)
      type: bool  default: false
--unbox-double-arrays (automatically unbox arrays of
  ↪ doubles)
      type: bool  default: true
--interrupt-budget (interrupt budget which should be
  ↪ used for the profiler counter)
      type: int  default: 147456
```

```
--jitless (Disable runtime allocation of executable
  ↪ memory.)
      type: bool  default: false
--use-ic (use inline caching)
      type: bool  default: true
--budget-for-feedback-vector-allocation (The budget in
  ↪  amount of bytecode executed by a function before
  ↪  we decide to allocate feedback vectors)
      type: int  default: 1024
--lazy-feedback-allocation (Allocate feedback vectors
  ↪ lazily)
      type: bool  default: true
--ignition-elide-noneffectful-bytecodes (elide
  ↪ bytecodes which won't have any external effect)
      type: bool  default: true
--ignition-reo (use ignition register equivalence
  ↪ optimizer)
      type: bool  default: true
--ignition-filter-expression-positions (filter
  ↪ expression positions before the bytecode pipeline
  ↪ )
      type: bool  default: true
--ignition-share-named-property-feedback (share
  ↪ feedback slots when loading the same named
  ↪ property from the same object)
      type: bool  default: true
--print-bytecode (print bytecode generated by ignition
  ↪  interpreter)
      type: bool  default: false
--enable-lazy-source-positions (skip generating source
  ↪  positions during initial compile but regenerate
  ↪ when actually required)
      type: bool  default: true
```

```
--stress-lazy-source-positions (collect lazy source
  ↪ positions immediately after lazy compile)
      type: bool  default: false
--print-bytecode-filter (filter for selecting which
  ↪ functions to print bytecode)
      type: string  default: *
--trace-ignition-codegen (trace the codegen of
  ↪ ignition interpreter bytecode handlers)
      type: bool  default: false
--trace-ignition-dispatches (traces the dispatches to
  ↪ bytecode handlers by the ignition interpreter)
      type: bool  default: false
--trace-ignition-dispatches-output-file (the file to
  ↪ which the bytecode handler dispatch table is
  ↪ written (by default, the table is not written to
  ↪ a file))
      type: string  default: nullptr
--fast-math (faster (but maybe less accurate) math
  ↪ functions)
      type: bool  default: true
--trace-track-allocation-sites (trace the tracking of
  ↪ allocation sites)
      type: bool  default: false
--trace-migration (trace object migration)
      type: bool  default: false
--trace-generalization (trace map generalization)
      type: bool  default: false
--turboprop (enable experimental turboprop mid-tier
  ↪ compiler.)
      type: bool  default: false
--concurrent-recompilation (optimizing hot functions
  ↪ asynchronously on a separate thread)
      type: bool  default: true
```

```
--trace-concurrent-recompilation (track concurrent
  ↪ recompilation)
      type: bool  default: false
--concurrent-recompilation-queue-length (the length of
  ↪  the concurrent compilation queue)
      type: int  default: 8
--concurrent-recompilation-delay (artificial
  ↪ compilation delay in ms)
      type: int  default: 0
--block-concurrent-recompilation (block queued jobs
  ↪ until released)
      type: bool  default: false
--concurrent-inlining (run optimizing compiler's
  ↪ inlining phase on a separate thread)
      type: bool  default: false
--max-serializer-nesting (maximum levels for nesting
  ↪ child serializers)
      type: int  default: 25
--trace-heap-broker-verbose (trace the heap broker
  ↪ verbosely (all reports))
      type: bool  default: false
--trace-heap-broker-memory (trace the heap broker
  ↪ memory (refs analysis and zone numbers))
      type: bool  default: false
--trace-heap-broker (trace the heap broker (reports on
  ↪  missing data only))
      type: bool  default: false
--stress-runs (number of stress runs)
      type: int  default: 0
--deopt-every-n-times (deoptimize every n times a
  ↪ deopt point is passed)
      type: int  default: 0
```

```
--print-deopt-stress (print number of possible deopt
  ↪ points)
      type: bool  default: false
--opt (use adaptive optimizations)
      type: bool  default: true
--turbo-sp-frame-access (use stack pointer-relative
  ↪ access to frame wherever possible)
      type: bool  default: false
--turbo-control-flow-aware-allocation (consider
  ↪ control flow while allocating registers)
      type: bool  default: true
--turbo-filter (optimization filter for TurboFan
  ↪ compiler)
      type: string  default: *
--trace-turbo (trace generated TurboFan IR)
      type: bool  default: false
--trace-turbo-path (directory to dump generated
  ↪ TurboFan IR to)
      type: string  default: nullptr
--trace-turbo-filter (filter for tracing turbofan
  ↪ compilation)
      type: string  default: *
--trace-turbo-graph (trace generated TurboFan graphs)
      type: bool  default: false
--trace-turbo-scheduled (trace TurboFan IR with
  ↪ schedule)
      type: bool  default: false
--trace-turbo-cfg-file (trace turbo cfg graph (for C1
  ↪ visualizer) to a given file name)
      type: string  default: nullptr
--trace-turbo-types (trace TurboFan's types)
      type: bool  default: true
--trace-turbo-scheduler (trace TurboFan's scheduler)
```

```
      type: bool  default: false
--trace-turbo-reduction (trace TurboFan's various
  ↪ reducers)
      type: bool  default: false
--trace-turbo-trimming (trace TurboFan's graph trimmer
  ↪ )
      type: bool  default: false
--trace-turbo-jt (trace TurboFan's jump threading)
      type: bool  default: false
--trace-turbo-ceq (trace TurboFan's control
  ↪ equivalence)
      type: bool  default: false
--trace-turbo-loop (trace TurboFan's loop
  ↪ optimizations)
      type: bool  default: false
--trace-turbo-alloc (trace TurboFan's register
  ↪ allocator)
      type: bool  default: false
--trace-all-uses (trace all use positions)
      type: bool  default: false
--trace-representation (trace representation types)
      type: bool  default: false
--turbo-verify (verify TurboFan graphs at each phase)
      type: bool  default: false
--turbo-verify-machine-graph (verify TurboFan machine
  ↪ graph before instruction selection)
      type: string  default: nullptr
--trace-verify-csa (trace code stubs verification)
      type: bool  default: false
--csa-trap-on-node (trigger break point when a node
  ↪ with given id is created in given stub. The
  ↪ format is: StubName,NodeId)
      type: string  default: nullptr
```

```
--turbo-stats (print TurboFan statistics)
      type: bool  default: false
--turbo-stats-nvp (print TurboFan statistics in
  ↪ machine-readable format)
      type: bool  default: false
--turbo-stats-wasm (print TurboFan statistics of wasm
  ↪ compilations)
      type: bool  default: false
--turbo-splitting (split nodes during scheduling in
  ↪ TurboFan)
      type: bool  default: true
--function-context-specialization (enable function
  ↪ context specialization in TurboFan)
      type: bool  default: false
--turbo-inlining (enable inlining in TurboFan)
      type: bool  default: true
--max-inlined-bytecode-size (maximum size of bytecode
  ↪ for a single inlining)
      type: int  default: 500
--max-inlined-bytecode-size-cumulative (maximum
  ↪ cumulative size of bytecode considered for
  ↪ inlining)
      type: int  default: 1000
--max-inlined-bytecode-size-absolute (maximum
  ↪ cumulative size of bytecode considered for
  ↪ inlining)
      type: int  default: 5000
--reserve-inline-budget-scale-factor (maximum
  ↪ cumulative size of bytecode considered for
  ↪ inlining)
      type: float  default: 1.2
--max-inlined-bytecode-size-small (maximum size of
  ↪ bytecode considered for small function inlining)
```

```
      type: int  default: 30
--max-optimized-bytecode-size (maximum bytecode size
  ↪ to be considered for optimization; too high
  ↪ values may cause the compiler to hit (release)
  ↪ assertions)
      type: int  default: 61440
--min-inlining-frequency (minimum frequency for
  ↪ inlining)
      type: float  default: 0.15
--polymorphic-inlining (polymorphic inlining)
      type: bool  default: true
--stress-inline (set high thresholds for inlining to
  ↪ inline as much as possible)
      type: bool  default: false
--trace-turbo-inlining (trace TurboFan inlining)
      type: bool  default: false
--turbo-inline-array-builtins (inline array builtins
  ↪ in TurboFan code)
      type: bool  default: true
--use-osr (use on-stack replacement)
      type: bool  default: true
--trace-osr (trace on-stack replacement)
      type: bool  default: false
--analyze-environment-liveness (analyze liveness of
  ↪ environment slots and zap dead values)
      type: bool  default: true
--trace-environment-liveness (trace liveness of local
  ↪ variable slots)
      type: bool  default: false
--turbo-load-elimination (enable load elimination in
  ↪ TurboFan)
      type: bool  default: true
```

```
--trace-turbo-load-elimination (trace TurboFan load
  ↪ elimination)
      type: bool  default: false
--turbo-profiling (enable basic block profiling in
  ↪ TurboFan)
      type: bool  default: false
--turbo-profiling-verbose (enable basic block
  ↪ profiling in TurboFan, and include each function'
  ↪ s schedule and disassembly in the output)
      type: bool  default: false
--turbo-verify-allocation (verify register allocation
  ↪ in TurboFan)
      type: bool  default: false
--turbo-move-optimization (optimize gap moves in
  ↪ TurboFan)
      type: bool  default: true
--turbo-jt (enable jump threading in TurboFan)
      type: bool  default: true
--turbo-loop-peeling (Turbofan loop peeling)
      type: bool  default: true
--turbo-loop-variable (Turbofan loop variable
  ↪ optimization)
      type: bool  default: true
--turbo-loop-rotation (Turbofan loop rotation)
      type: bool  default: true
--turbo-cf-optimization (optimize control flow in
  ↪ TurboFan)
      type: bool  default: true
--turbo-escape (enable escape analysis)
      type: bool  default: true
--turbo-allocation-folding (Turbofan allocation
  ↪ folding)
      type: bool  default: true
```

```
--turbo-instruction-scheduling (enable instruction
  ↪ scheduling in TurboFan)
      type: bool  default: false
--turbo-stress-instruction-scheduling (randomly
  ↪ schedule instructions to stress dependency
  ↪ tracking)
      type: bool  default: false
--turbo-store-elimination (enable store-store
  ↪ elimination in TurboFan)
      type: bool  default: true
--trace-store-elimination (trace store elimination)
      type: bool  default: false
--turbo-rewrite-far-jumps (rewrite far to near jumps (
  ↪ ia32,x64))
      type: bool  default: true
--stress-gc-during-compilation (simulate GC/compiler
  ↪ thread race related to https://crbug.com/v8/8520)
      type: bool  default: false
--turbo-fast-api-calls (enable fast API calls from
  ↪ TurboFan)
      type: bool  default: false
--reuse-opt-code-count (don't discard optimized code
  ↪ for the specified number of deopts.)
      type: int  default: 0
--turbo-nci (enable experimental native context
  ↪ independent code.)
      type: bool  default: false
--turbo-nci-as-highest-tier (replace default TF with
  ↪ NCI code as the highest tier for testing purposes
  ↪ .)
      type: bool  default: false
--print-nci-code (print native context independent
  ↪ code.)
```

```
      type: bool  default: false
--trace-turbo-nci (trace native context independent
  ↪ code.)
      type: bool  default: false
--turbo-collect-feedback-in-generic-lowering (enable
  ↪ experimental feedback collection in generic
  ↪ lowering.)
      type: bool  default: false
--optimize-for-size (Enables optimizations which favor
  ↪  memory size over execution speed)
      type: bool  default: false
--untrusted-code-mitigations (Enable mitigations for
  ↪ executing untrusted code)
      type: bool  default: false
--expose-wasm (expose wasm interface to JavaScript)
      type: bool  default: true
--assume-asmjs-origin (force wasm decoder to assume
  ↪ input is internal asm-wasm format)
      type: bool  default: false
--wasm-num-compilation-tasks (maximum number of
  ↪ parallel compilation tasks for wasm)
      type: int  default: 128
--wasm-write-protect-code-memory (write protect code
  ↪ memory on the wasm native heap)
      type: bool  default: false
--wasm-async-compilation (enable actual asynchronous
  ↪ compilation for WebAssembly.compile)
      type: bool  default: true
--wasm-test-streaming (use streaming compilation
  ↪ instead of async compilation for tests)
      type: bool  default: false
--wasm-max-mem-pages (maximum initial number of 64KiB
  ↪ memory pages of a wasm instance)
```

```
      type: uint  default: 32767
--wasm-max-mem-pages-growth (maximum number of 64KiB
  ↪ pages a Wasm memory can grow to)
      type: uint  default: 65536
--wasm-max-table-size (maximum table size of a wasm
  ↪ instance)
      type: uint  default: 10000000
--wasm-max-code-space (maximum committed code space
  ↪ for wasm (in MB))
      type: uint  default: 1024
--wasm-tier-up (enable tier up to the optimizing
  ↪ compiler (requires --liftoff to have an effect))
      type: bool  default: true
--trace-wasm-ast-start (start function for wasm AST
  ↪ trace (inclusive))
      type: int  default: 0
--trace-wasm-ast-end (end function for wasm AST trace
  ↪ (exclusive))
      type: int  default: 0
--liftoff (enable Liftoff, the baseline compiler for
  ↪ WebAssembly)
      type: bool  default: true
--trace-wasm-memory (print all memory updates
  ↪ performed in wasm code)
      type: bool  default: false
--wasm-tier-mask-for-testing (bitmask of functions to
  ↪ compile with TurboFan instead of Liftoff)
      type: int  default: 0
--wasm-expose-debug-eval (Expose wasm evaluator
  ↪ support on the CDP)
      type: bool  default: false
--validate-asm (validate asm.js modules before
  ↪ compiling)
```

```
      type: bool  default: true
--suppress-asm-messages (don't emit asm.js related
  ↪ messages (for golden file testing))
      type: bool  default: false
--trace-asm-time (log asm.js timing info to the
  ↪ console)
      type: bool  default: false
--trace-asm-scanner (log tokens encountered by asm.js
  ↪ scanner)
      type: bool  default: false
--trace-asm-parser (verbose logging of asm.js parse
  ↪ failures)
      type: bool  default: false
--stress-validate-asm (try to validate everything as
  ↪ asm.js)
      type: bool  default: false
--dump-wasm-module-path (directory to dump wasm
  ↪ modules to)
      type: string  default: nullptr
--experimental-wasm-eh (enable prototype exception
  ↪ handling opcodes for wasm)
      type: bool  default: false
--experimental-wasm-simd (enable prototype SIMD
  ↪ opcodes for wasm)
      type: bool  default: false
--experimental-wasm-return-call (enable prototype
  ↪ return call opcodes for wasm)
      type: bool  default: false
--experimental-wasm-compilation-hints (enable
  ↪ prototype compilation hints section for wasm)
      type: bool  default: false
--experimental-wasm-gc (enable prototype garbage
  ↪ collection for wasm)
```

```
      type: bool  default: false
--experimental-wasm-typed-funcref (enable prototype
  ↪ typed function references for wasm)
      type: bool  default: false
--experimental-wasm-reftypes (enable prototype
  ↪ reference type opcodes for wasm)
      type: bool  default: false
--experimental-wasm-threads (enable prototype thread
  ↪ opcodes for wasm)
      type: bool  default: false
--experimental-wasm-type-reflection (enable prototype
  ↪ wasm type reflection in JS for wasm)
      type: bool  default: false
--experimental-wasm-bigint (enable prototype JS BigInt
  ↪  support for wasm)
      type: bool  default: true
--experimental-wasm-bulk-memory (enable prototype bulk
  ↪  memory opcodes for wasm)
      type: bool  default: true
--experimental-wasm-mv (enable prototype multi-value
  ↪ support for wasm)
      type: bool  default: true
--wasm-staging (enable staged wasm features)
      type: bool  default: false
--wasm-opt (enable wasm optimization)
      type: bool  default: false
--wasm-bounds-checks (enable bounds checks (disable
  ↪ for performance testing only))
      type: bool  default: true
--wasm-stack-checks (enable stack checks (disable for
  ↪ performance testing only))
      type: bool  default: true
```

```
--wasm-math-intrinsics (intrinsify some Math imports
  ↪ into wasm)
      type: bool  default: true
--wasm-trap-handler (use signal handlers to catch out
  ↪ of bounds memory access in wasm (currently Linux
  ↪ x86_64 only))
      type: bool  default: true
--wasm-fuzzer-gen-test (generate a test case when
  ↪ running a wasm fuzzer)
      type: bool  default: false
--print-wasm-code (Print WebAssembly code)
      type: bool  default: false
--print-wasm-stub-code (Print WebAssembly stub code)
      type: bool  default: false
--asm-wasm-lazy-compilation (enable lazy compilation
  ↪ for asm-wasm modules)
      type: bool  default: false
--wasm-lazy-compilation (enable lazy compilation for
  ↪ all wasm modules)
      type: bool  default: false
--wasm-lazy-validation (enable lazy validation for
  ↪ lazily compiled wasm functions)
      type: bool  default: false
--wasm-atomics-on-non-shared-memory (allow atomic
  ↪ operations on non-shared WebAssembly memory)
      type: bool  default: true
--wasm-grow-shared-memory (allow growing shared
  ↪ WebAssembly memory objects)
      type: bool  default: true
--wasm-simd-post-mvp (allow experimental SIMD
  ↪ operations for prototyping that are not included
  ↪ in the current proposal)
      type: bool  default: false
```

```
--wasm-code-gc (enable garbage collection of wasm code
  ↪ )
      type: bool  default: true
--trace-wasm-code-gc (trace garbage collection of wasm
  ↪  code)
      type: bool  default: false
--stress-wasm-code-gc (stress test garbage collection
  ↪ of wasm code)
      type: bool  default: false
--wasm-max-initial-code-space-reservation (maximum
  ↪ size of the initial wasm code space reservation (
  ↪ in MB))
      type: int  default: 0
--frame-count (number of stack frames inspected by the
  ↪  profiler)
      type: int  default: 1
--stress-sampling-allocation-profiler (Enables
  ↪ sampling allocation profiler with X as a sample
  ↪ interval)
      type: int  default: 0
--lazy-new-space-shrinking (Enables the lazy new space
  ↪  shrinking strategy)
      type: bool  default: false
--min-semi-space-size (min size of a semi-space (in
  ↪ MBytes), the new space consists of two semi-
  ↪ spaces)
      type: size_t  default: 0
--max-semi-space-size (max size of a semi-space (in
  ↪ MBytes), the new space consists of two semi-
  ↪ spaces)
      type: size_t  default: 0
--semi-space-growth-factor (factor by which to grow
  ↪ the new space)
```

```
      type: int  default: 2
--max-old-space-size (max size of the old space (in
  ↪ Mbytes))
      type: size_t  default: 0
--max-heap-size (max size of the heap (in Mbytes) both
  ↪  max_semi_space_size and max_old_space_size take
  ↪ precedence. All three flags cannot be specified
  ↪ at the same time.)
      type: size_t  default: 0
--initial-heap-size (initial size of the heap (in
  ↪ Mbytes))
      type: size_t  default: 0
--huge-max-old-generation-size (Increase max size of
  ↪ the old space to 4 GB for x64 systems withthe
  ↪ physical memory bigger than 16 GB)
      type: bool  default: true
--initial-old-space-size (initial old space size (in
  ↪ Mbytes))
      type: size_t  default: 0
--global-gc-scheduling (enable GC scheduling based on
  ↪ global memory)
      type: bool  default: true
--gc-global (always perform global GCs)
      type: bool  default: false
--random-gc-interval (Collect garbage after random(0,
  ↪ X) allocations. It overrides gc_interval.)
      type: int  default: 0
--gc-interval (garbage collect after <n> allocations)
      type: int  default: -1
--retain-maps-for-n-gc (keeps maps alive for <n> old
  ↪ space garbage collections)
      type: int  default: 2
```

```
--trace-gc (print one trace line following each
  ↪ garbage collection)
      type: bool  default: false
--trace-gc-nvp (print one detailed trace line in name=
  ↪ value format after each garbage collection)
      type: bool  default: false
--trace-gc-ignore-scavenger (do not print trace line
  ↪ after scavenger collection)
      type: bool  default: false
--trace-idle-notification (print one trace line
  ↪ following each idle notification)
      type: bool  default: false
--trace-idle-notification-verbose (prints the heap
  ↪ state used by the idle notification)
      type: bool  default: false
--trace-gc-verbose (print more details following each
  ↪ garbage collection)
      type: bool  default: false
--trace-gc-freelists (prints details of each freelist
  ↪ before and after each major garbage collection)
      type: bool  default: false
--trace-gc-freelists-verbose (prints details of
  ↪ freelists of each page before and after each
  ↪ major garbage collection)
      type: bool  default: false
--trace-evacuation-candidates (Show statistics about
  ↪ the pages evacuation by the compaction)
      type: bool  default: false
--trace-allocations-origins (Show statistics about the
  ↪  origins of allocations. Combine with --no-inline
  ↪ -new to track allocations from generated code)
      type: bool  default: false
```

```
--trace-allocation-stack-interval (print stack trace
  ↪ after <n> free-list allocations)
      type: int  default: -1
--trace-duplicate-threshold-kb (print duplicate
  ↪ objects in the heap if their size is more than
  ↪ given threshold)
      type: int  default: 0
--trace-fragmentation (report fragmentation for old
  ↪ space)
      type: bool  default: false
--trace-fragmentation-verbose (report fragmentation
  ↪ for old space (detailed))
      type: bool  default: false
--minor-mc-trace-fragmentation (trace fragmentation
  ↪ after marking)
      type: bool  default: false
--trace-evacuation (report evacuation statistics)
      type: bool  default: false
--trace-mutator-utilization (print mutator utilization
  ↪ , allocation speed, gc speed)
      type: bool  default: false
--incremental-marking (use incremental marking)
      type: bool  default: true
--incremental-marking-wrappers (use incremental
  ↪ marking for marking wrappers)
      type: bool  default: true
--incremental-marking-task (use tasks for incremental
  ↪ marking)
      type: bool  default: true
--incremental-marking-soft-trigger (threshold for
  ↪ starting incremental marking via a task in
  ↪ percent of available space: limit - size)
      type: int  default: 0
```

```
--incremental-marking-hard-trigger (threshold for
  ↪ starting incremental marking immediately in
  ↪ percent of available space: limit - size)
      type: int  default: 0
--trace-unmapper (Trace the unmapping)
      type: bool  default: false
--parallel-scavenge (parallel scavenge)
      type: bool  default: true
--scavenge-task (schedule scavenge tasks)
      type: bool  default: true
--scavenge-task-trigger (scavenge task trigger in
  ↪ percent of the current heap limit)
      type: int  default: 80
--scavenge-separate-stack-scanning (use a separate
  ↪ phase for stack scanning in scavenge)
      type: bool  default: false
--trace-parallel-scavenge (trace parallel scavenge)
      type: bool  default: false
--write-protect-code-memory (write protect code memory
  ↪ )
      type: bool  default: true
--concurrent-marking (use concurrent marking)
      type: bool  default: true
--concurrent-array-buffer-sweeping (concurrently sweep
  ↪  array buffers)
      type: bool  default: true
--concurrent-allocation (concurrently allocate in old
  ↪ space)
      type: bool  default: false
--local-heaps (allow heap access from background tasks
  ↪ )
      type: bool  default: false
```

```
--stress-concurrent-allocation (start background
  ↪ threads that allocate memory)
      type: bool  default: false
--parallel-marking (use parallel marking in atomic
  ↪ pause)
      type: bool  default: true
--ephemeron-fixpoint-iterations (number of fixpoint
  ↪ iterations it takes to switch to linear ephemeron
  ↪  algorithm)
      type: int  default: 10
--trace-concurrent-marking (trace concurrent marking)
      type: bool  default: false
--concurrent-store-buffer (use concurrent store buffer
  ↪  processing)
      type: bool  default: true
--concurrent-sweeping (use concurrent sweeping)
      type: bool  default: true
--parallel-compaction (use parallel compaction)
      type: bool  default: true
--parallel-pointer-update (use parallel pointer update
  ↪  during compaction)
      type: bool  default: true
--detect-ineffective-gcs-near-heap-limit (trigger out-
  ↪ of-memory failure to avoid GC storm near heap
  ↪ limit)
      type: bool  default: true
--trace-incremental-marking (trace progress of the
  ↪ incremental marking)
      type: bool  default: false
--trace-stress-marking (trace stress marking progress)
      type: bool  default: false
--trace-stress-scavenge (trace stress scavenge
  ↪ progress)
```

```
      type: bool  default: false
--track-gc-object-stats (track object counts and
  ↪ memory usage)
      type: bool  default: false
--trace-gc-object-stats (trace object counts and
  ↪ memory usage)
      type: bool  default: false
--trace-zone-stats (trace zone memory usage)
      type: bool  default: false
--zone-stats-tolerance (report a tick only when
  ↪ allocated zone memory changes by this amount)
      type: size_t  default: 1048576
--track-retaining-path (enable support for tracking
  ↪ retaining path)
      type: bool  default: false
--concurrent-array-buffer-freeing (free array buffer
  ↪ allocations on a background thread)
      type: bool  default: true
--gc-stats (Used by tracing internally to enable gc
  ↪ statistics)
      type: int  default: 0
--track-detached-contexts (track native contexts that
  ↪ are expected to be garbage collected)
      type: bool  default: true
--trace-detached-contexts (trace native contexts that
  ↪ are expected to be garbage collected)
      type: bool  default: false
--move-object-start (enable moving of object starts)
      type: bool  default: true
--memory-reducer (use memory reducer)
      type: bool  default: true
--memory-reducer-for-small-heaps (use memory reducer
  ↪ for small heaps)
```

```
      type: bool  default: true
--heap-growing-percent (specifies heap growing factor
  ↪ as (1 + heap_growing_percent/100))
      type: int  default: 0
--v8-os-page-size (override OS page size (in KBytes))
      type: int  default: 0
--always-compact (Perform compaction on every full GC)
      type: bool  default: false
--never-compact (Never perform compaction on full GC -
  ↪  testing only)
      type: bool  default: false
--compact-code-space (Compact code space on full
  ↪ collections)
      type: bool  default: true
--flush-bytecode (flush of bytecode when it has not
  ↪ been executed recently)
      type: bool  default: true
--stress-flush-bytecode (stress bytecode flushing)
      type: bool  default: false
--use-marking-progress-bar (Use a progress bar to scan
  ↪  large objects in increments when incremental
  ↪ marking is active.)
      type: bool  default: true
--stress-per-context-marking-worklist (Use per-context
  ↪  worklist for marking)
      type: bool  default: false
--force-marking-deque-overflows (force overflows of
  ↪ marking deque by reducing it's size to 64 words)
      type: bool  default: false
--stress-compaction (stress the GC compactor to flush
  ↪ out bugs (implies --force_marking_deque_overflows
  ↪ ))
      type: bool  default: false
```

```
--stress-compaction-random (Stress GC compaction by
  ↪ selecting random percent of pages as evacuation
  ↪ candidates. It overrides stress_compaction.)
      type: bool  default: false
--stress-incremental-marking (force incremental
  ↪ marking for small heaps and run it more often)
      type: bool  default: false
--fuzzer-gc-analysis (prints number of allocations and
  ↪  enables analysis mode for gc fuzz testing, e.g.
  ↪ --stress-marking, --stress-scavenge)
      type: bool  default: false
--stress-marking (force marking at random points
  ↪ between 0 and X (inclusive) percent of the
  ↪ regular marking start limit)
      type: int  default: 0
--stress-scavenge (force scavenge at random points
  ↪ between 0 and X (inclusive) percent of the new
  ↪ space capacity)
      type: int  default: 0
--gc-experiment-background-schedule (new background GC
  ↪  schedule heuristics)
      type: bool  default: false
--gc-experiment-less-compaction (less compaction in
  ↪ non-memory reducing mode)
      type: bool  default: false
--disable-abortjs (disables AbortJS runtime function)
      type: bool  default: false
--randomize-all-allocations (randomize virtual memory
  ↪ reservations by ignoring any hints passed when
  ↪ allocating pages)
      type: bool  default: false
--manual-evacuation-candidates-selection (Test mode
  ↪ only flag. It allows an unit test to select
  ↪ evacuation candidates pages (requires --
  ↪ stress_compaction).)
      type: bool  default: false
--fast-promotion-new-space (fast promote new space on
  ↪ high survival rates)
      type: bool  default: false
--clear-free-memory (initialize free memory with 0)
      type: bool  default: false
--young-generation-large-objects (allocates large
  ↪ objects by default in the young generation large
  ↪ object space)
      type: bool  default: true
--debug-code (generate extra code (assertions) for
  ↪ debugging)
      type: bool  default: false
--code-comments (emit comments in code disassembly;
  ↪ for more readable source positions you should add
  ↪  --no-concurrent_recompilation)
      type: bool  default: false
--enable-sse3 (enable use of SSE3 instructions if
  ↪ available)
      type: bool  default: true
--enable-ssse3 (enable use of SSSE3 instructions if
  ↪ available)
      type: bool  default: true
--enable-sse4-1 (enable use of SSE4.1 instructions if
  ↪ available)
      type: bool  default: true
--enable-sse4-2 (enable use of SSE4.2 instructions if
  ↪ available)
      type: bool  default: true
--enable-sahf (enable use of SAHF instruction if
  ↪ available (X64 only))
```

```
      type: bool  default: true
--enable-avx (enable use of AVX instructions if
  ↪ available)
      type: bool  default: true
--enable-fma3 (enable use of FMA3 instructions if
  ↪ available)
      type: bool  default: true
--enable-bmi1 (enable use of BMI1 instructions if
  ↪ available)
      type: bool  default: true
--enable-bmi2 (enable use of BMI2 instructions if
  ↪ available)
      type: bool  default: true
--enable-lzcnt (enable use of LZCNT instruction if
  ↪ available)
      type: bool  default: true
--enable-popcnt (enable use of POPCNT instruction if
  ↪ available)
      type: bool  default: true
--arm-arch (generate instructions for the selected ARM
  ↪  architecture if available: armv6, armv7, armv7+
  ↪ sudiv or armv8)
      type: string  default: armv8
--force-long-branches (force all emitted branches to
  ↪ be in long mode (MIPS/PPC only))
      type: bool  default: false
--mcpu (enable optimization for specific cpu)
      type: string  default: auto
--partial-constant-pool (enable use of partial
  ↪ constant pools (X64 only))
      type: bool  default: true
--sim-arm64-optional-features (enable optional
  ↪ features on the simulator for testing: none or
  ↪ all)
      type: string  default: none
--enable-source-at-csa-bind (Include source
  ↪ information in the binary at CSA bind locations.)
      type: bool  default: false
--enable-armv7 (deprecated (use --arm_arch instead))
      type: maybe_bool  default: unset
--enable-vfp3 (deprecated (use --arm_arch instead))
      type: maybe_bool  default: unset
--enable-32dregs (deprecated (use --arm_arch instead))
      type: maybe_bool  default: unset
--enable-neon (deprecated (use --arm_arch instead))
      type: maybe_bool  default: unset
--enable-sudiv (deprecated (use --arm_arch instead))
      type: maybe_bool  default: unset
--enable-armv8 (deprecated (use --arm_arch instead))
      type: maybe_bool  default: unset
--enable-regexp-unaligned-accesses (enable unaligned
  ↪ accesses for the regexp engine)
      type: bool  default: true
--script-streaming (enable parsing on background)
      type: bool  default: true
--stress-background-compile (stress test parsing on
  ↪ background)
      type: bool  default: false
--finalize-streaming-on-background (perform the script
  ↪  streaming finalization on the background thread)
      type: bool  default: false
--disable-old-api-accessors (Disable old-style API
  ↪ accessors whose setters trigger through the
  ↪ prototype chain)
      type: bool  default: false
--expose-gc (expose gc extension)
```

```
      type: bool  default: false
--expose-gc-as (expose gc extension under the
  ↪ specified name)
      type: string  default: nullptr
--expose-externalize-string (expose externalize string
  ↪  extension)
      type: bool  default: false
--expose-trigger-failure (expose trigger-failure
  ↪ extension)
      type: bool  default: false
--stack-trace-limit (number of stack frames to capture
  ↪ )
      type: int  default: 10
--builtins-in-stack-traces (show built-in functions in
  ↪  stack traces)
      type: bool  default: false
--experimental-stack-trace-frames (enable experimental
  ↪  frames (API/Builtins) and stack trace layout)
      type: bool  default: false
--disallow-code-generation-from-strings (disallow eval
  ↪  and friends)
      type: bool  default: false
--expose-async-hooks (expose async_hooks object)
      type: bool  default: false
--expose-cputracemark-as (expose cputracemark
  ↪ extension under the specified name)
      type: string  default: nullptr
--allow-unsafe-function-constructor (allow invoking
  ↪ the function constructor without security checks)
      type: bool  default: false
--force-slow-path (always take the slow path for
  ↪ builtins)
      type: bool  default: false
```

```
--test-small-max-function-context-stub-size (enable
  ↪ testing the function context size overflow path
  ↪ by making the maximum size smaller)
      type: bool  default: false
--inline-new (use fast inline allocation)
      type: bool  default: true
--trace (trace javascript function calls)
      type: bool  default: false
--trace-wasm (trace wasm function calls)
      type: bool  default: false
--lazy (use lazy compilation)
      type: bool  default: true
--max-lazy (ignore eager compilation hints)
      type: bool  default: false
--trace-opt (trace lazy optimization)
      type: bool  default: false
--trace-opt-verbose (extra verbose compilation tracing
  ↪ )
      type: bool  default: false
--trace-opt-stats (trace lazy optimization statistics)
      type: bool  default: false
--trace-deopt (trace optimize function deoptimization)
      type: bool  default: false
--trace-file-names (include file names in trace-opt/
  ↪ trace-deopt output)
      type: bool  default: false
--always-opt (always try to optimize functions)
      type: bool  default: false
--always-osr (always try to OSR functions)
      type: bool  default: false
--prepare-always-opt (prepare for turning on always
  ↪ opt)
      type: bool  default: false
```

```
--trace-serializer (print code serializer trace)
      type: bool  default: false
--compilation-cache (enable compilation cache)
      type: bool  default: true
--cache-prototype-transitions (cache prototype
  ↪ transitions)
      type: bool  default: true
--parallel-compile-tasks (enable parallel compile
  ↪ tasks)
      type: bool  default: false
--compiler-dispatcher (enable compiler dispatcher)
      type: bool  default: false
--trace-compiler-dispatcher (trace compiler dispatcher
  ↪  activity)
      type: bool  default: false
--cpu-profiler-sampling-interval (CPU profiler
  ↪ sampling interval in microseconds)
      type: int  default: 1000
--trace-side-effect-free-debug-evaluate (print debug
  ↪ messages for side-effect-free debug-evaluate for
  ↪ testing)
      type: bool  default: false
--hard-abort (abort by crashing)
      type: bool  default: true
--expose-inspector-scripts (expose injected-script-
  ↪ source.js for debugging)
      type: bool  default: false
--stack-size (default size of stack region v8 is
  ↪ allowed to use (in kBytes))
      type: int  default: 984
--max-stack-trace-source-length (maximum length of
  ↪ function source code printed in a stack trace.)
      type: int  default: 300
```

```
--clear-exceptions-on-js-entry (clear pending
  ↪ exceptions when entering JavaScript)
      type: bool  default: false
--histogram-interval (time interval in ms for
  ↪ aggregating memory histograms)
      type: int  default: 600000
--heap-profiler-trace-objects (Dump heap object
  ↪ allocations/movements/size_updates)
      type: bool  default: false
--heap-profiler-use-embedder-graph (Use the new
  ↪ EmbedderGraph API to get embedder nodes)
      type: bool  default: true
--heap-snapshot-string-limit (truncate strings to this
  ↪  length in the heap snapshot)
      type: int  default: 1024
--sampling-heap-profiler-suppress-randomness (Use
  ↪ constant sample intervals to eliminate test
  ↪ flakiness)
      type: bool  default: false
--use-idle-notification (Use idle notification to
  ↪ reduce memory footprint.)
      type: bool  default: true
--trace-ic (trace inline cache state transitions for
  ↪ tools/ic-processor)
      type: bool  default: false
--modify-field-representation-inplace (enable in-place
  ↪  field representation updates)
      type: bool  default: true
--max-polymorphic-map-count (maximum number of maps to
  ↪  track in POLYMORPHIC state)
      type: int  default: 4
--native-code-counters (generate extra code for
  ↪ manipulating stats counters)
```

```
      type: bool  default: false
--thin-strings (Enable ThinString support)
      type: bool  default: true
--trace-prototype-users (Trace updates to prototype
  ↪ user tracking)
      type: bool  default: false
--trace-for-in-enumerate (Trace for-in enumerate slow-
  ↪ paths)
      type: bool  default: false
--trace-maps (trace map creation)
      type: bool  default: false
--trace-maps-details (also log map details)
      type: bool  default: true
--allow-natives-syntax (allow natives syntax)
      type: bool  default: false
--allow-natives-for-differential-fuzzing (allow only
  ↪ natives explicitly allowlisted for differential
  ↪ fuzzers)
      type: bool  default: false
--parse-only (only parse the sources)
      type: bool  default: false
--trace-sim (Trace simulator execution)
      type: bool  default: false
--debug-sim (Enable debugging the simulator)
      type: bool  default: false
--check-icache (Check icache flushes in ARM and MIPS
  ↪ simulator)
      type: bool  default: false
--stop-sim-at (Simulator stop after x number of
  ↪ instructions)
      type: int  default: 0
--sim-stack-alignment (Stack alingment in bytes in
  ↪ simulator (4 or 8, 8 is default))
```

```
      type: int  default: 8
--sim-stack-size (Stack size of the ARM64, MIPS64 and
  ↪ PPC64 simulator in kBytes (default is 2 MB))
      type: int  default: 2048
--log-colour (When logging, try to use coloured output
  ↪ .)
      type: bool  default: true
--trace-sim-messages (Trace simulator debug messages.
  ↪ Implied by --trace-sim.)
      type: bool  default: false
--async-stack-traces (include async stack traces in
  ↪ Error.stack)
      type: bool  default: true
--stack-trace-on-illegal (print stack trace when an
  ↪ illegal exception is thrown)
      type: bool  default: false
--abort-on-uncaught-exception (abort program (dump
  ↪ core) when an uncaught exception is thrown)
      type: bool  default: false
--correctness-fuzzer-suppressions (Suppress certain
  ↪ unspecified behaviors to ease correctness fuzzing
  ↪ : Abort program when the stack overflows or a
  ↪ string exceeds maximum length (as opposed to
  ↪ throwing RangeError). Use a fixed suppression
  ↪ string for error messages.)
      type: bool  default: false
--randomize-hashes (randomize hashes to avoid
  ↪ predictable hash collisions (with snapshots this
  ↪ option cannot override the baked-in seed))
      type: bool  default: true
--rehash-snapshot (rehash strings from the snapshot to
  ↪  override the baked-in seed)
      type: bool  default: true
```

```
--hash-seed (Fixed seed to use to hash property keys
  ↪ (0 means random)(with snapshots this option
  ↪ cannot override the baked-in seed))
      type: uint64  default: 0
--random-seed (Default seed for initializing random
  ↪ generator (0, the default, means to use system
  ↪ random).)
      type: int  default: 0
--fuzzer-random-seed (Default seed for initializing
  ↪ fuzzer random generator (0, the default, means to
  ↪  use v8's random number generator seed).)
      type: int  default: 0
--trace-rail (trace RAIL mode)
      type: bool  default: false
--print-all-exceptions (print exception object and
  ↪ stack trace on each thrown exception)
      type: bool  default: false
--detailed-error-stack-trace (includes arguments for
  ↪ each function call in the error stack frames
  ↪ array)
      type: bool  default: false
--adjust-os-scheduling-parameters (adjust OS specific
  ↪ scheduling params for the isolate)
      type: bool  default: true
--runtime-call-stats (report runtime call counts and
  ↪ times)
      type: bool  default: false
--rcs (report runtime call counts and times)
      type: bool  default: false
--rcs-cpu-time (report runtime times in cpu time (the
  ↪ default is wall time))
      type: bool  default: false
```

```
--profile-deserialization (Print the time it takes to
  ↪ deserialize the snapshot.)
      type: bool  default: false
--serialization-statistics (Collect statistics on
  ↪ serialized objects.)
      type: bool  default: false
--serialization-chunk-size (Custom size for
  ↪ serialization chunks)
      type: uint  default: 4096
--regexp-optimization (generate optimized regexp code)
      type: bool  default: true
--regexp-mode-modifiers (enable inline flags in regexp
  ↪ .)
      type: bool  default: false
--regexp-interpret-all (interpret all regexp code)
      type: bool  default: false
--regexp-tier-up (enable regexp interpreter and tier
  ↪ up to the compiler after the number of executions
  ↪  set by the tier up ticks flag)
      type: bool  default: true
--regexp-tier-up-ticks (set the number of executions
  ↪ for the regexp interpreter before tiering-up to
  ↪ the compiler)
      type: int  default: 1
--regexp-peephole-optimization (enable peephole
  ↪ optimization for regexp bytecode)
      type: bool  default: true
--trace-regexp-peephole-optimization (trace regexp
  ↪ bytecode peephole optimization)
      type: bool  default: false
--trace-regexp-bytecodes (trace regexp bytecode
  ↪ execution)
      type: bool  default: false
```

```
--trace-regexp-assembler (trace regexp macro assembler
  ↪  calls.)
      type: bool  default: false
--trace-regexp-parser (trace regexp parsing)
      type: bool  default: false
--trace-regexp-tier-up (trace regexp tiering up
  ↪ execution)
      type: bool  default: false
--testing-bool-flag (testing_bool_flag)
      type: bool  default: true
--testing-maybe-bool-flag (testing_maybe_bool_flag)
      type: maybe_bool  default: unset
--testing-int-flag (testing_int_flag)
      type: int  default: 13
--testing-float-flag (float-flag)
      type: float  default: 2.5
--testing-string-flag (string-flag)
      type: string  default: Hello, world!
--testing-prng-seed (Seed used for threading test
  ↪ randomness)
      type: int  default: 42
--testing-d8-test-runner (test runner turns on this
  ↪ flag to enable a check that the function was
  ↪ prepared for optimization before marking it for
  ↪ optimization)
      type: bool  default: false
--fuzzing (Fuzzers use this flag to signal that they
  ↪ are ... fuzzing. This causes intrinsics to fail
  ↪ silently (e.g. return undefined) on invalid usage
  ↪ .)
      type: bool  default: false
--embedded-src (Path for the generated embedded data
  ↪ file. (mksnapshot only))
```

```
      type: string  default: nullptr
--embedded-variant (Label to disambiguate symbols in
  ↪ embedded data file. (mksnapshot only))
      type: string  default: nullptr
--startup-src (Write V8 startup as C++ src. (
  ↪ mksnapshot only))
      type: string  default: nullptr
--startup-blob (Write V8 startup blob file. (
  ↪ mksnapshot only))
      type: string  default: nullptr
--target-arch (The mksnapshot target arch. (mksnapshot
  ↪  only))
      type: string  default: nullptr
--target-os (The mksnapshot target os. (mksnapshot
  ↪ only))
      type: string  default: nullptr
--target-is-simulator (Instruct mksnapshot that the
  ↪ target is meant to run in the simulator and it
  ↪ can generate simulator-specific instructions. (
  ↪ mksnapshot only))
      type: bool  default: false
--minor-mc-parallel-marking (use parallel marking for
  ↪ the young generation)
      type: bool  default: true
--trace-minor-mc-parallel-marking (trace parallel
  ↪ marking for the young generation)
      type: bool  default: false
--minor-mc (perform young generation mark compact GCs)
      type: bool  default: false
--help (Print usage message, including flags, on
  ↪ console)
      type: bool  default: true
--dump-counters (Dump counters on exit)
```

```
      type: bool  default: false
--dump-counters-nvp (Dump counters as name-value pairs
  ↪  on exit)
      type: bool  default: false
--use-external-strings (Use external strings for
  ↪ source code)
      type: bool  default: false
--map-counters (Map counters to a file)
      type: string  default:
--mock-arraybuffer-allocator (Use a mock ArrayBuffer
  ↪ allocator for testing.)
      type: bool  default: false
--mock-arraybuffer-allocator-limit (Memory limit for
  ↪ mock ArrayBuffer allocator used to simulate OOM
  ↪ for testing.)
      type: size_t  default: 0
--gdbjit (enable GDBJIT interface)
      type: bool  default: false
--gdbjit-full (enable GDBJIT interface for all code
  ↪ objects)
      type: bool  default: false
--gdbjit-dump (dump elf objects with debug info to
  ↪ disk)
      type: bool  default: false
--gdbjit-dump-filter (dump only objects containing
  ↪ this substring)
      type: string  default:
--log (Minimal logging (no API, code, GC, suspect, or
  ↪ handles samples).)
      type: bool  default: false
--log-all (Log all events to the log file.)
      type: bool  default: false
--log-api (Log API events to the log file.)
```

```
      type: bool  default: false
--log-code (Log code events to the log file without
  ↪ profiling.)
      type: bool  default: false
--log-handles (Log global handle events.)
      type: bool  default: false
--log-suspect (Log suspect operations.)
      type: bool  default: false
--log-source-code (Log source code.)
      type: bool  default: false
--log-function-events (Log function events (parse,
  ↪ compile, execute) separately.)
      type: bool  default: false
--prof (Log statistical profiling information (implies
  ↪  --log-code).)
      type: bool  default: false
--detailed-line-info (Always generate detailed line
  ↪ information for CPU profiling.)
      type: bool  default: false
--prof-sampling-interval (Interval for --prof samples
  ↪ (in microseconds).)
      type: int  default: 1000
--prof-cpp (Like --prof, but ignore generated code.)
      type: bool  default: false
--prof-browser-mode (Used with --prof, turns on
  ↪ browser-compatible mode for profiling.)
      type: bool  default: true
--logfile (Specify the name of the log file.)
      type: string  default: v8.log
--logfile-per-isolate (Separate log files for each
  ↪ isolate.)
      type: bool  default: true
--ll-prof (Enable low-level linux profiler.)
```

```
      type: bool  default: false
--gc-fake-mmap (Specify the name of the file for fake
  ↪ gc mmap used in ll_prof)
      type: string  default: /tmp/__v8_gc__
--log-internal-timer-events (Time internal events.)
      type: bool  default: false
--redirect-code-traces (output deopt information and
  ↪ disassembly into file code-<pid>-<isolate id>.asm
  ↪ )
      type: bool  default: false
--redirect-code-traces-to (output deopt information
  ↪ and disassembly into the given file)
      type: string  default: nullptr
--print-opt-source (print source code of optimized and
  ↪  inlined functions)
      type: bool  default: false
--vtune-prof-annotate-wasm (Used when
  ↪ v8_enable_vtunejit is enabled, load wasm source
  ↪ map and provide annotate support (experimental).)
      type: bool  default: false
--win64-unwinding-info (Enable unwinding info for
  ↪ Windows/x64)
      type: bool  default: true
--interpreted-frames-native-stack (Show interpreted
  ↪ frames on the native stack (useful for external
  ↪ profilers).)
      type: bool  default: false
--predictable (enable predictable mode)
      type: bool  default: false
--predictable-gc-schedule (Predictable garbage
  ↪ collection schedule. Fixes heap growing, idle,
  ↪ and memory reducing behavior.)
      type: bool  default: false
--single-threaded (disable the use of background tasks
  ↪ )
      type: bool  default: false
--single-threaded-gc (disable the use of background gc
  ↪  tasks)
      type: bool  default: false
```

Particularly useful ones:

```
--async-stack-trace
```

Continuous Benchmarks

See our benchmarks over here

The benchmark chart supposes https://github.com/denoland/benchmark pages/data.json has the type `BenchmarkData[]` where `BenchmarkData` is defined like the below:

```
interface ExecTimeData {
  mean: number;
  stddev: number;
  user: number;
  system: number;
  min: number;
  max: number;
}

interface BenchmarkData {
  created_at: string;
  sha1: string;
  benchmark: {
    [key: string]: ExecTimeData;
  };
  binarySizeData: {
    [key: string]: number;
  };
  threadCountData: {
    [key: string]: number;
  };
  syscallCountData: {
    [key: string]: number;
  };
}
```

Deno Style Guide

Copyright Headers

Most modules in the repository should have the following copyright header:

```
// Copyright 2018-2020 the Deno authors. All rights
  ↪ reserved. MIT license.
```

If the code originates elsewhere, ensure that the file has the proper copyright headers. We only allow MIT, BSD, and Apache licensed code.

Use underscores, not dashes in filenames.

Example: Use `file_server.ts` instead of `file-server.ts`.

Add tests for new features.

Each module should contain or be accompanied by tests for its public functionality.

TODO Comments

TODO comments should usually include an issue or the author's github username in parentheses. Example:

```
// TODO(ry): Add tests.
// TODO(#123): Support Windows.
// FIXME(#349): Sometimes panics.
```

Meta-programming is discouraged. Including the use of Proxy.

Be explicit even when it means more code.

There are some situations where it may make sense to use such techniques, but in the vast majority of cases it does not.

Inclusive code

Please follow the guidelines for inclusive code outlined at https://chromiur

Rust

Follow Rust conventions and be consistent with existing code.

TypeScript

The TypeScript portions of the codebase include `cli/js` for the built-ins and the standard library `std`.

Use TypeScript instead of JavaScript.

Use the term "module" instead of "library" or "package".

For clarity and consistency avoid the terms "library" and "package". Instead use "module" to refer to a single JS or TS file and also to refer to a directory of TS/JS code.

Do not use the filename `index.ts`/`index.js`.

Deno does not treat "index.js" or "index.ts" in a special way. By using these filenames, it suggests that they can be left out of the module specifier when they cannot. This is confusing.

If a directory of code needs a default entry point, use the filename `mod` ↪ `.ts`. The filename `mod.ts` follows Rust's convention, is shorter than `index.ts`, and doesn't come with any preconceived notions about how it might work.

Exported functions: max 2 args, put the rest into an options object.

When designing function interfaces, stick to the following rules.

1. A function that is part of the public API takes 0-2 required arguments, plus (if necessary) an options object (so max 3 total).
2. Optional parameters should generally go into the options object.

An optional parameter that's not in an options object might be acceptable if there is only one, and it seems inconceivable that we would add more optional parameters in the future.

3. The 'options' argument is the only argument that is a regular 'Object'.

   Other arguments can be objects, but they must be distinguishable from a 'plain' Object runtime, by having either:

   - a distinguishing prototype (e.g. `Array`, `Map`, `Date`, `class ↪ MyThing`).
   - a well-known symbol property (e.g. an iterable with `Symbol ↪ .iterator`).

   This allows the API to evolve in a backwards compatible way, even when the position of the options object changes.

```
// BAD: optional parameters not part of options object.
  ↪ (#2)
export function resolve(
  hostname: string,
  family?: "ipv4" | "ipv6",
  timeout?: number,
): IPAddress[] {}

// GOOD.
export interface ResolveOptions {
  family?: "ipv4" | "ipv6";
  timeout?: number;
}
export function resolve(
  hostname: string,
  options: ResolveOptions = {},
): IPAddress[] {}
```

```
export interface Environment {
  [key: string]: string;
}

// BAD: `env` could be a regular Object and is therefore
  ↪  indistinguishable
// from an options object. (#3)
export function runShellWithEnv(cmdline: string, env:
  ↪ Environment): string {}

// GOOD.
export interface RunShellOptions {
  env: Environment;
}
export function runShellWithEnv(
  cmdline: string,
  options: RunShellOptions,
): string {}
```

```
// BAD: more than 3 arguments (#1), multiple optional
  ↪ parameters (#2).
export function renameSync(
  oldname: string,
  newname: string,
  replaceExisting?: boolean,
  followLinks?: boolean,
) {}

// GOOD.
interface RenameOptions {
  replaceExisting?: boolean;
  followLinks?: boolean;
}
export function renameSync(
  oldname: string,
  newname: string,
  options: RenameOptions = {},
) {}
```

```
// BAD: too many arguments. (#1)
export function pwrite(
  fd: number,
  buffer: TypedArray,
  offset: number,
  length: number,
  position: number,
) {}

// BETTER.
export interface PWrite {
  fd: number;
  buffer: TypedArray;
  offset: number;
  length: number;
  position: number;
}
export function pwrite(options: PWrite) {}
```

Minimize dependencies; do not make circular imports.

Although `cli/js` and `std` have no external dependencies, we must still be careful to keep internal dependencies simple and manageable. In particular, be careful not to introduce circular imports.

If a filename starts with an underscore: `_foo.ts`, do not link to it.

Sometimes there may be situations where an internal module is necessary but its API is not meant to be stable or linked to. In this case prefix it with an underscore. By convention, only files in its own directory should import it.

Use JSDoc for exported symbols.

We strive for complete documentation. Every exported symbol ideally should have a documentation line.

If possible, use a single line for the JSDoc. Example:

```
/** foo does bar. */
export function foo() {
  // ...
}
```

It is important that documentation is easily human readable, but there is also a need to provide additional styling information to ensure generated documentation is more rich text. Therefore JSDoc should generally follow markdown markup to enrich the text.

While markdown supports HTML tags, it is forbidden in JSDoc blocks.

Code string literals should be braced with the back-tick (‘) instead of quotes. For example:

```
/** Import something from the `deno` module. */
```

Do not document function arguments unless they are non-obvious of their intent (though if they are non-obvious intent, the API should be considered anyways). Therefore `@param` should generally not be used. If `@param` is used, it should not include the `type` as TypeScript is already strongly typed.

```
/**
 * Function with non obvious param.
 * @param foo Description of non obvious parameter.
 */
```

Vertical spacing should be minimized whenever possible. Therefore single line comments should be written as:

```
/** This is a good single line JSDoc. */
```

And not:

```
/**
 * This is a bad single line JSDoc.
 */
```

Code examples should not utilise the triple-back tick ("') notation or tags. They should just be marked by indentation, which requires a break before the block and 6 additional spaces for each line of the example. This is 4 more than the first column of the comment. For example:

```
/** A straight forward comment and an example:
 *
 *       import { foo } from "deno";
 *       foo("bar");
 */
```

Code examples should not contain additional comments. It is already inside a comment. If it needs further comments it is not a good example.

Each module should come with a test module.

Every module with public functionality `foo.ts` should come with a test module `foo_test.ts`. A test for a `cli/js` module should go in `cli/js/`
↪ `tests` due to their different contexts, otherwise it should just be a sibling to the tested module.

Unit Tests should be explicit.

For a better understanding of the tests, function should be correctly named as its prompted throughout the test command. Like:

```
test myTestFunction ... ok
```

Example of test:

```
import { assertEquals } from "https://deno.land/
  ↪ std@$STD_VERSION/testing/asserts.ts";
import { foo } from "./mod.ts";

Deno.test("myTestFunction", function () {
  assertEquals(foo(), { bar: "bar" });
});
```

Top level functions should not use arrow syntax.

Top level functions should use the `function` keyword. Arrow syntax should be limited to closures.

Bad:

```
export const foo = (): string => {
  return "bar";
};
```

Good:

```
export function foo(): string {
  return "bar";
}
```

std

Do not depend on external code. `https://deno.land/std/` is intended to be baseline functionality that all Deno programs can rely on. We want to guarantee to users that this code does not include potentially unreviewed third party code.

Document and maintain browser compatiblity. If a module is browser compatible, include the following in the JSDoc at the top of the module:

```
/** This module is browser compatible. */
```

Maintain browser compatibility for such a module by either not using the global `Deno` namespace or feature-testing for it. Make sure any new dependencies are also browser compatible.

Internal details

Deno and Linux analogy

Linux	Deno
Processes	Web Workers
Syscalls	Ops
File descriptors (fd)	Resource ids (rid)
Scheduler	Tokio

	Linux	Deno
Userland:	libc++ / glib / boost	https://deno.land/std/
	/proc/$$/stat	Deno.metrics()
	man pages	deno types

Resources Resources (AKA `rid`) are Deno's version of file descriptors. They are integer values used to refer to open files, sockets, and other concepts. For testing it would be good to be able to query the system for how many open resources there are.

```
console.log(Deno.resources());
// { 0: "stdin", 1: "stdout", 2: "stderr" }
Deno.close(0);
console.log(Deno.resources());
// { 1: "stdout", 2: "stderr" }
```

Metrics Metrics is Deno's internal counter for various statistics.

```
> console.table(Deno.metrics())

     (index)        Values

  opsDispatched       9
   opsCompleted       9
 bytesSentControl    504
  bytesSentData       0
  bytesReceived      856
```

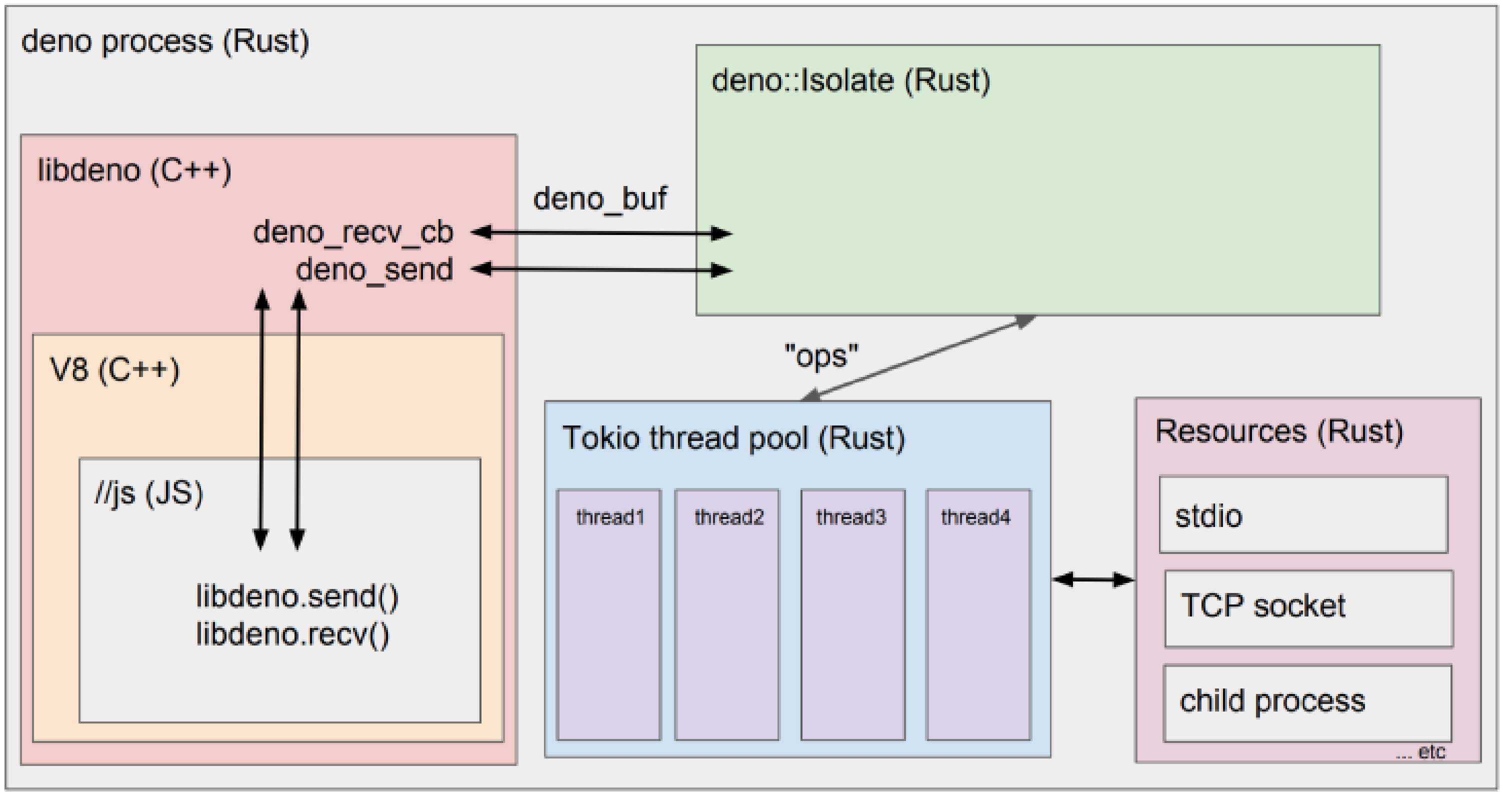


Figure 7: architectural schematic

Schematic diagram

Conference

Ryan Dahl - An interesting case with Deno

Examples

In this chapter you can find some example programs that you can use to learn more about the runtime.

Basic

- Hello World
- Import and Export Modules

- How to Manage Dependencies
- Fetch Data
- Read and Write Files

Advanced

- Unix Cat
- File Server
- TCP Echo
- Subprocess
- Permissions
- OS Signals
- File System Events
- Testing If Main

Hello World

Deno is a secure runtime for both JavaScript and TypeScript. As the hello world examples below highlight the same functionality can be created in JavaScript or TypeScript, and Deno will execute both.

JavaScript

In this JavaScript example the message `Hello [name]` is printed to the console and the code ensures the name provided is capitalized.

Command: `deno run hello-world.js`

```
function capitalize(word) {
  return word.charAt(0).toUpperCase() + word.slice(1);
}
```

```
function hello(name) {
  return "Hello " + capitalize(name);
}

console.log(hello("john"));
console.log(hello("Sarah"));
console.log(hello("kai"));

/**
 * Output:
 *
 * Hello John
 * Hello Sarah
 * Hello Kai
**/
```

TypeScript

This TypeScript example is exactly the same as the JavaScript example above, the code just has the additional type information which TypeScript supports.

The `deno run` command is exactly the same, it just references a `*.ts` file rather than a `*.js` file.

Command: `deno run hello-world.ts`

```
function capitalize(word: string): string {
  return word.charAt(0).toUpperCase() + word.slice(1);
}

function hello(name: string): string {
  return "Hello " + capitalize(name);
}

console.log(hello("john"));
console.log(hello("Sarah"));
console.log(hello("kai"));

/**
 * Output:
 *
 * Hello John
 * Hello Sarah
 * Hello Kai
**/
```

Import and Export Modules

Deno by default standardizes the way modules are imported in both JavaScript and TypeScript. It follows the ECMAScript 6 `import/export` standard with one caveat, the file type must be included at the end of import statement.

```
import {
  add,
  multiply,
} from "./arithmetic.ts";
```

Dependencies are also imported directly, there is no package management overhead. Local modules are imported in exactly the same way as remote modules. As the examples show below, the same functionality can be produced in the same way with local or remote modules.

Local Import

In this example the `add` and `multiply` functions are imported from a local `arithmetic.ts` module.

Command: `deno run local.ts`

```
import { add, multiply } from "./arithmetic.ts";

function totalCost(outbound: number, inbound: number,
  ↪ tax: number): number {
  return multiply(add(outbound, inbound), tax);
}

console.log(totalCost(19, 31, 1.2));
console.log(totalCost(45, 27, 1.15));

/**
 * Output
 *
 * 60
 * 82.8
 */
```

Export

In the example above the `add` and `multiply` functions are imported from a locally stored arithmetic module. To make this possible the functions stored in the arithmetic module must be exported.

To do this just add the keyword `export` to the beginning of the function signature as is shown below.

```
export function add(a: number, b: number): number {
  return a + b;
}

export function multiply(a: number, b: number): number {
  return a * b;
}
```

All functions, classes, constants and variables which need to be accessible inside external modules must be exported. Either by prepending them with the `export` keyword or including them in an export statement at the bottom of the file.

To find out more on ECMAScript Export functionality please read the MDN Documentation.

Remote Import

In the local import example above an `add` and `multiply` method are imported from a locally stored arithmetic module. The same functionality can be created by importing `add` and `multiply` methods from a remote module too.

In this case the Ramda module is referenced, including the version number. Also note a JavaScript module is imported directly into a TypeScript module, Deno has no problem handling this.

Command: `deno run ./remote.ts`

```
import {
  add,
  multiply,
} from "https://x.nest.land/ramda@0.27.0/source/index.js
  ↪ ";
```

```
function totalCost(outbound: number, inbound: number,
  ↪ tax: number): number {
  return multiply(add(outbound, inbound), tax);
}

console.log(totalCost(19, 31, 1.2));
console.log(totalCost(45, 27, 1.15));

/**
 * Output
 *
 * 60
 * 82.8
 */
```

Managing Dependencies

In Deno there is no concept of a package manager as external modules are imported directly into local modules. This raises the question of how to manage remote dependencies without a package manager. In big projects with many dependencies it will become cumbersome and time consuming to update modules if they are all imported individually into individual modules.

The standard practice for solving this problem in Deno is to create a `deps.ts` file. All required remote dependencies are referenced in this file and the required methods and classes are re-exported. The dependent local modules then reference the `deps.ts` rather than the remote dependencies.

This enables easy updates to modules across a large codebase and solves the 'package manager problem', if it ever existed. Dev dependencies can also be managed in a separate `dev_deps.ts` file.

deps.ts example

```
/**
 * deps.ts re-exports the required methods from the
   ↪ remote Ramda module.
 **/
export {
  add,
  multiply,
} from "https://x.nest.land/ramda@0.27.0/source/index.js
  ↪ ";
```

In this example the same functionality is created as is the case in the local and remote import examples. But in this case instead of the Ramda module being referenced directly it is referenced by proxy using a local `deps.ts` module.

Command: `deno run dependencies.ts`

```
import {
  add,
  multiply,
} from "./deps.ts";

function totalCost(outbound: number, inbound: number,
  ↪ tax: number): number {
  return multiply(add(outbound, inbound), tax);
}

console.log(totalCost(19, 31, 1.2));
console.log(totalCost(45, 27, 1.15));
```

```
/**
 * Output
 *
 * 60
 * 82.8
 */
```

Fetch Data

When building any sort of web application developers will usually need to retrieve data from somewhere else on the web. This works no differently in Deno than in any other JavaScript application, just call the the `fetch()` method. For more information on fetch read the MDN documentation.

The exception with Deno occurs when running a script which makes a call over the web. Deno is secure by default which means access to IO (Input / Output) is prohibited. To make a call over the web Deno must be explicitly told it is ok to do so. This is achieved by adding the `--allow-net` flag to the `deno run` command.

Command: `deno run --allow-net fetch.ts`

```
/**
 * Output: JSON Data
 */
const json = fetch("https://api.github.com/users/
  ↪ denoland");

json.then((response) => {
  return response.json();
}).then((jsonData) => {
  console.log(jsonData);
});

/**
 * Output: HTML Data
 */
const text = fetch("https://deno.land/");

text.then((response) => {
  return response.text();
}).then((textData) => {
  console.log(textData);
});

/**
 * Output: Error Message
 */
const error = fetch("https://does.not.exist/");

error.catch((error) => console.log(error.message));
```

Read and Write Files

Interacting with the filesystem to read and write files is a basic requirement of most development projects. Deno provides a number of ways to do this via the standard library and the Deno runtime API.

As highlighted in the Fetch Data example Deno restricts access to Input / Output by default for security reasons. So when interacting with the filesystem the `--allow-read` and `--allow-write` flags must be used with the `deno run` command.

Read

The Deno runtime API makes it possible to read text files via the `readTextFile()` method, it just requires a path string or URL object. The method returns a promise which provides access to the file's text data.

Command: `deno run --allow-read read.ts`

```
async function readFile(path: string): Promise<string> {
  return await Deno.readTextFile(new URL(path, import.
    ↪ meta.url));
}

const text = readFile("./people.json");

text.then((response) => console.log(response));

/**
 * Output:
 *
 * [
 *   {"id": 1, "name": "John", "age": 23},
 *   {"id": 2, "name": "Sandra", "age": 51},
 *   {"id": 5, "name": "Devika", "age": 11}
 * ]
 */
```

The Deno standard library enables more advanced interaction with the filesystem and provides methods to read and parse files. The `readJson` ↪ `()` and `readJsonSync()` methods allow developers to read and parse files containing JSON. All these methods require is a valid file path string which can be generated using the `fromFileUrl()` method.

In the example below the `readJsonSync()` method is used, for asynchronus execution use the `readJson()` method.

Currently some of this functionality is marked as unstable so the `--`
↪ `unstable` flag is required along with the `deno run` command.

Command: `deno run --unstable --allow-read read.ts`

```
import { readJsonSync } from "https://deno.land/
  ↪ std@$STD_VERSION/fs/mod.ts";
import { fromFileUrl } from "https://deno.land/
  ↪ std@$STD_VERSION/path/mod.ts";

function readJson(path: string): object {
  const file = fromFileUrl(new URL(path, import.meta.url
    ↪ ));
  return readJsonSync(file) as object;
}

console.log(readJson("./people.json"));

/**
 * Output:
 *
 * [
 *   {"id": 1, "name": "John", "age": 23},
 *   {"id": 2, "name": "Sandra", "age": 51},
 *   {"id": 5, "name": "Devika", "age": 11}
 * ]
 */
```

Write

The Deno runtime API allows developers to write text to files via the `writeTextFile()` method. It just requires a file path and text string. The method returns a promise which resolves when the file was successfully written.

To run the command the `--allow-write` flag must be supplied to the `deno run` command.

Command: `deno run --allow-write write.ts`

```
async function writeFile(path: string, text: string):
  ↪ Promise<void> {
  return await Deno.writeTextFile(path, text);
}

const write = writeFile("./hello.txt", "Hello World!");

write.then(() => console.log("File written to."));

/**
 * Output: File written to.
 */
```

The Deno standard library makes available more advanced features to write to the filesystem. For instance it is possible to write an object literal to a JSON file.

This requires a combination of the `ensureFile()`, `ensureFileSync()`, `writeJson()` and `writeJsonSync()` methods. In the example below the `ensureFileSync()` and the `writeJsonSync()` methods are used. The former checks for the existence of a file, and if it doesn't exist creates it. The latter method then writes the object to the file as

JSON. If asynchronus execution is required use the `ensureFile()` and `writeJson()` methods.

To execute the code the `deno run` command needs the unstable flag and both the write and read flags.

Command: `deno run --allow-write --allow-read --unstable write`
`↪ .ts`

```
import {
  ensureFileSync,
  writeJsonSync,
} from "https://deno.land/std@$STD_VERSION/fs/mod.ts";

function writeJson(path: string, data: object): string {
  try {
    ensureFileSync(path);
    writeJsonSync(path, data);

    return "Written to " + path;
  } catch (e) {
    return e.message;
  }
}

console.log(writeJson("./data.json", { hello: "World" })
  ↪ );

/**
 * Output: Written to ./data.json
 */
```

An implementation of the unix “cat” program

In this program each command-line argument is assumed to be a filename, the file is opened, and printed to stdout.

```
for (let i = 0; i < Deno.args.length; i++) {
  const filename = Deno.args[i];
  const file = await Deno.open(filename);
  await Deno.copy(file, Deno.stdout);
  file.close();
}
```

The `copy()` function here actually makes no more than the necessary kernel -> userspace -> kernel copies. That is, the same memory from which data is read from the file, is written to stdout. This illustrates a general design goal for I/O streams in Deno.

Try the program:

```
deno run --allow-read https://deno.land/std@$STD_VERSION
  ↪ /examples/cat.ts /etc/passwd
```

File server

This one serves a local directory in HTTP.

```
deno install --allow-net --allow-read https://deno.land/
  ↪ std@$STD_VERSION/http/file_server.ts
```

Run it:

```
$ file_server .
Downloading https://deno.land/std@$STD_VERSION/http/
  ↪ file_server.ts...
[...]
HTTP server listening on http://0.0.0.0:4500/
```

And if you ever want to upgrade to the latest published version:

```
file_server --reload
```

TCP echo server

This is an example of a server which accepts connections on port 8080, and returns to the client anything it sends.

```
const listener = Deno.listen({ port: 8080 });
console.log("listening on 0.0.0.0:8080");
for await (const conn of listener) {
  Deno.copy(conn, conn);
}
```

When this program is started, it throws PermissionDenied error.

```
$ deno run https://deno.land/std@$STD_VERSION/examples/
  ↪ echo_server.ts
error: Uncaught PermissionDenied: network access to
  ↪ "0.0.0.0:8080", run again with the --allow-net flag
 $deno$/dispatch_json.ts:40:11
    at DenoError ($deno$/errors.ts:20:5)
    ...
```

For security reasons, Deno does not allow programs to access the network without explicit permission. To allow accessing the network, use a command-line flag:

```
deno run --allow-net https://deno.land/std@$STD_VERSION/
  ↪ examples/echo_server.ts
```

To test it, try sending data to it with netcat:

```
$ nc localhost 8080
hello world
hello world
```

Like the `cat.ts` example, the `copy()` function here also does not make unnecessary memory copies. It receives a packet from the kernel and sends back, without further complexity.

Run subprocess

API Reference

Example:

```
// create subprocess
const p = Deno.run({
  cmd: ["echo", "hello"],
});

// await its completion
await p.status();
```

Run it:

```
$ deno run --allow-run ./subprocess_simple.ts
hello
```

Here a function is assigned to `window.onload`. This function is called after the main script is loaded. This is the same as onload of the browsers, and it can be used as the main entrypoint.

By default when you use `Deno.run()` subprocess inherits `stdin`, `stdout` and `stderr` of parent process. If you want to communicate with started subprocess you can use `"piped"` option.

```
const fileNames = Deno.args;

const p = Deno.run({
  cmd: [
    "deno",
    "run",
    "--allow-read",
    "https://deno.land/std@$STD_VERSION/examples/cat.ts
      ↪ ",
    ...fileNames,
  ],
  stdout: "piped",
  stderr: "piped",
});

const { code } = await p.status();

if (code === 0) {
  const rawOutput = await p.output();
  await Deno.stdout.write(rawOutput);
} else {
  const rawError = await p.stderrOutput();
  const errorString = new TextDecoder().decode(rawError)
    ↪ ;
  console.log(errorString);
}

Deno.exit(code);
```

When you run it:

```
$ deno run --allow-run ./subprocess.ts <somefile>
[file content]

$ deno run --allow-run ./subprocess.ts non_existent_file
  ↪ .md
```

```
Uncaught NotFound: No such file or directory (os error
  ↪ 2)
    at DenoError (deno/js/errors.ts:22:5)
    at maybeError (deno/js/errors.ts:41:12)
    at handleAsyncMsgFromRust (deno/js/dispatch.ts
      ↪ :27:17)
```

Handle OS Signals

> This program makes use of an unstable Deno feature. Learn more about unstable features.

API Reference

You can use `Deno.signal()` function for handling OS signals:

```
for await (const _ of Deno.signal(Deno.Signal.SIGINT)) {
  console.log("interrupted!");
}
```

`Deno.signal()` also works as a promise:

```
await Deno.signal(Deno.Signal.SIGINT);
console.log("interrupted!");
```

If you want to stop watching the signal, you can use `dispose()` method of the signal object:

```
const sig = Deno.signal(Deno.Signal.SIGINT);
setTimeout(() => {
  sig.dispose();
}, 5000);

for await (const _ of sig) {
  console.log("interrupted");
}
```

The above for-await loop exits after 5 seconds when `sig.dispose()` is called.

File system events

To poll for file system events:

```
const watcher = Deno.watchFs("/");
for await (const event of watcher) {
  console.log(">>>> event", event);
  // { kind: "create", paths: [ "/foo.txt" ] }
}
```

Note that the exact ordering of the events can vary between operating systems. This feature uses different syscalls depending on the platform:

- Linux: inotify
- macOS: FSEvents
- Windows: ReadDirectoryChangesW

Testing if current file is the main program

To test if the current script has been executed as the main input to the program check `import.meta.main`.

```
if (import.meta.main) {
  console.log("main");
}
```