Chokidar

Minimal and efficient cross-platform file watching library

Why?

Node.js `fs.watch`:

Doesn't report filenames on MacOS.

Doesn't report events at all when using editors like Sublime on MacOS.

Often reports events twice.

Emits most changes as `rename`.

Does not provide an easy way to recursively watch file trees.

Does not support recursive watching on Linux.

Node.js `fs.watchFile`:

Almost as bad at event handling.

Also does not provide any recursive watching.

Results in high CPU utilization.

Chokidar resolves these problems.

Initially made for **Brunch** (an ultra-swift web app build tool), it is now used in Microsoft's Visual Studio Code, gulp, karma, PM2, browserify, webpack, BrowserSync, and many others. It has proven itself in production environments.

Version 3 is out! Check out our blog post about it: Chokidar 3: How to save 32TB of traffic every week

How?

Chokidar does still rely on the Node.js core `fs` module, but when using `fs.watch` and `fs.watchFile` for watching, it normalizes the events it receives, often checking for truth by getting file stats and/or dir contents.

On MacOS, chokidar by default uses a native extension exposing the Darwin `FSEvents` API. This provides very efficient recursive watching compared with implementations like `kqueue` available on most *nix platforms. Chokidar still does have to do some work to normalize the events received that way as well.

On most other platforms, the `fs.watch`-based implementation is the default, which avoids polling and keeps CPU usage down. Be advised that chokidar will initiate watchers recursively for everything within scope of the paths that have been specified, so be judicious about not wasting system resources by watching much more than needed.

Getting started

Install with npm:

```
npm install chokidar
```

Then `require` and use it in your code:

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

// One-liner for current directory
chokidar.watch('.').on('all', (event,
        path) => {
  console.log(event, path);
});
```

API

```
// Example of a more typical
        implementation structure

// Initialize watcher.
const watcher = chokidar.watch('file,
        dir, glob, or array', {
  ignored: /(^|[\/\\])\../, // ignore
        dotfiles
  persistent: true
});
```

```
// Something to use when events are
        received.
const log = console.log.bind(console);
// Add event listeners.
watcher
  .on('add', path => log(`File ${path}
        has been added`))
  .on('change', path => log(`File $
        {path} has been changed`))
  .on('unlink', path => log(`File $
        {path} has been removed`));

// More possible events.
watcher
  .on('addDir', path => log(`Directory $
        {path} has been added`))
  .on('unlinkDir', path =>
        log(`Directory ${path} has been
        removed`))
  .on('error', error => log(`Watcher
        error: ${error}`))
  .on('ready', () => log('Initial scan
        complete. Ready for changes'))
  .on('raw', (event, path, details) =>
        { // internal
    log('Raw event info:', event, path,
        details);
  });

// 'add', 'addDir' and 'change' events
        also receive stat() results as
        second
```


only; without windows support. Tons of bugfixes. - **v1 (Apr 7, 2015):** Glob support, symlink support, tons of bugfixes. Node 0.8+ is supported - **v0.1 (Apr 20, 2012):** Initial release, extracted from Brunch

Also

Why was chokidar named this way? What's the meaning behind it?

> *Chowkidar is a transliteration of a Hindi word meaning 'watchman, gatekeeper', चौकीदार. This ultimately comes from Sanskrit _ चतुष्क_ (crossway, quadrangle, consisting-of-four). This word is also used in other languages like Urdu as (چوکیدار) which is widely used in Pakistan and India.*

License

- Chokidar is producing `ENOSP` error on Linux, like this:
  - `bash: cannot set terminal process group (-1): Inappropriate ioctl for device bash: no job control in this shell Error: watch / home/ ENOSPC`
  - This means Chokidar ran out of file handles and you'll need to increase their count by executing the following command in Terminal: `echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p`

Changelog

For more detailed changelog, see `full_changelog.md`. - **v3.5 (Jan 6, 2021):** Support for ARM Macs with Apple Silicon. Fixes for deleted symlinks. - **v3.4 (Apr 26, 2020):** Support for directory-based symlinks. Fixes for macos file replacement. - **v3.3 (Nov 2, 2019):** `FSWatcher#close()` method became async. That fixes IO race conditions related to close method. - **v3.2 (Oct 1, 2019):** Improve Linux RAM usage by 50%. Race condition fixes. Windows glob fixes. Improve stability by using tight range of dependency versions. - **v3.1 (Sep 16, 2019):** dotfiles are no longer filtered out by default. Use `ignored` option if needed. Improve initial Linux scan time by 50%. - **v3 (Apr 30, 2019):** massive CPU & RAM consumption improvements; reduces deps / package size by a factor of 17x and bumps Node.js requirement to v8.16 and higher. - **v2 (Dec 29, 2017):** Globs are now posix-style-


```
// argument when available: https://
        nodejs.org/api/
        fs.html#fs_class_fs_stats
watcher.on('change', (path, stats) => {
  if (stats) console.log(`File ${path}
        changed size to ${stats.size}`);
});

// Watch new files.
watcher.add('new-file');
watcher.add(['new-file-2', 'new-
        file-3', '**/other-file*']);

// Get list of actual paths being
        watched on the filesystem
var watchedPaths = watcher.getWatched();

// Un-watch some files.
await watcher.unwatch('new-file*');

// Stop watching.
// The method is async!
watcher.close().then(() =>
        console.log('closed'));

// Full list of options. See below for
        descriptions.
// Do not use this example!
chokidar.watch('file', {
  persistent: true,

  ignored: '*.txt',
  ignoreInitial: false,
  followSymlinks: true,
```

```
  cwd: '.',
  disableGlobbing: false,

  usePolling: false,
  interval: 100,
  binaryInterval: 300,
  alwaysStat: false,
  depth: 99,
  awaitWriteFinish: {
    stabilityThreshold: 2000,
    pollInterval: 100
  },

  ignorePermissionErrors: false,
  atomic:
        true // or a custom 'atomicity
        delay', in milliseconds (default
        100)
});
```

`chokidar.watch(paths, [options])`

- `paths` (string or array of strings). Paths to files, dirs to be watched recursively, or glob patterns.
  - Note: globs must not contain windows separators (`\`), because that's how they work by the standard — you'll need to replace them with forward slashes (`/`).
  - Note 2: for additional glob documentation, check out low-level library: picomatch.
- `options` (object) Options object as defined below:

- `.getWatched()`: Returns an object representing all the paths on the file system being watched by this `FSWatcher` instance. The object's keys are all the directories (using absolute paths unless the `cwd` option was used), and the values are arrays of the names of the items contained in each directory.

CLI

If you need a CLI interface for your file watching, check out chokidar-cli, allowing you to execute a command on each change, or get a stdio stream of change events.

Install Troubleshooting

- `npm WARN optional dep failed, continuing fsevents@n.n.n`
  - This message is normal part of how `npm` handles optional dependencies and is not indicative of a problem. Even if accompanied by other related error messages, Chokidar should function properly.
- `TypeError: fsevents is not a constructor`
  - Update chokidar by doing `rm -rf node_modules package-lock.json yarn.lock && npm install`, or update your dependency that uses chokidar.

- `atomic` (default: `true` if `useFsEvents` and `usePolling` are `false`). Automatically filters out artifacts that occur when using editors that use “atomic writes” instead of writing directly to the source file. If a file is re-added within 100 ms of being deleted, Chokidar emits a `change` event rather than `unlink` then `add`. If the default of 100 ms does not work well for you, you can override it by setting `atomic` to a custom value, in milliseconds.

Methods & Events

`chokidar.watch()` produces an instance of `FSWatcher`. Methods of `FSWatcher`:

- `.add(path / paths)`: Add files, directories, or glob patterns for tracking. Takes an array of strings or just one string.
- `.on(event, callback)`: Listen for an FS event. Available events: `add`, `addDir`, `change`, `unlink`, `unlinkDir`, `ready`, `raw`, `error`. Additionally `all` is available which gets emitted with the underlying event name and path for every event other than `ready`, `raw`, and `error`. `raw` is internal, use it carefully.
- `.unwatch(path / paths)`: Stop watching files, directories, or glob patterns. Takes an array of strings or just one string.
- `.close()`: **async** Removes all listeners from watched files. Asynchronous, returns Promise. Use with `await` to ensure bugs don’t happen.

Persistence

- `persistent` (default: `true`). Indicates whether the process should continue to run as long as files are being watched. If set to `false` when using `fsevents` to watch, no more events will be emitted after `ready`, even if the process continues to run.

Path filtering

- `ignored` (anymatch-compatible definition) Defines files/paths to be ignored. The whole relative or absolute path is tested, not just filename. If a function with two arguments is provided, it gets called twice per path - once with a single argument (the path), second time with two arguments (the path and the `fs.Stats` object of that path).
- `ignoreInitial` (default: `false`). If set to `false` then `add`/`addDir` events are also emitted for matching paths while instantiating the watching as chokidar discovers these file paths (before the `ready` event).
- `followSymlinks` (default: `true`). When `false`, only the symlinks themselves will be watched for changes instead of following the link references and bubbling events through the link’s path.
- `cwd` (no default). The base directory from which watch `paths` are to be derived. Paths emitted with events will be relative to this.
- `disableGlobbing` (default: `false`). If set to `true` then the strings passed to `.watch()` and `.add()` are treated as literal path names, even if they look like globs.

Performance

- `usePolling` (default: `false`). Whether to use fs.watchFile (backed by polling), or fs.watch. If polling leads to high CPU utilization, consider setting this to `false`. It is typically necessary to **set this to `true` to successfully watch files over a network**, and it may be necessary to successfully watch files in other non-standard situations. Setting to `true` explicitly on MacOS overrides the `useFsEvents` default. You may also set the CHOKIDAR_USEPOLLING env variable to true (1) or false (0) in order to override this option.
- *Polling-specific settings* (effective when `usePolling: true`)
  - `interval` (default: `100`). Interval of file system polling, in milliseconds. You may also set the CHOKIDAR_INTERVAL env variable to override this option.
  - `binaryInterval` (default: `300`). Interval of file system polling for binary files. (see list of binary extensions)
- `useFsEvents` (default: `true` on MacOS). Whether to use the `fsevents` watching interface if available. When set to `true` explicitly and `fsevents` is available this supercedes the `usePolling` setting. When set to `false` on MacOS, `usePolling: true` becomes the default.
- `alwaysStat` (default: `false`). If relying upon the `fs.Stats` object that may get passed with `add`, `addDir`, and `change` events, set this to `true` to ensure it is provided even in cases where it wasn't already available from the underlying watch events.

- `depth` (default: `undefined`). If set, limits how many levels of subdirectories will be traversed.
- `awaitWriteFinish` (default: `false`). By default, the `add` event will fire when a file first appears on disk, before the entire file has been written. Furthermore, in some cases some `change` events will be emitted while the file is being written. In some cases, especially when watching for large files there will be a need to wait for the write operation to finish before responding to a file creation or modification. Setting `awaitWriteFinish` to `true` (or a truthy value) will poll file size, holding its `add` and `change` events until the size does not change for a configurable amount of time. The appropriate duration setting is heavily dependent on the OS and hardware. For accurate detection this parameter should be relatively high, making file watching much less responsive. Use with caution.
  - `options.awaitWriteFinish` *can be set to an object in order to adjust timing params:*
  - `awaitWriteFinish.stabilityThreshold` (default: 2000). Amount of time in milliseconds for a file size to remain constant before emitting its event.
  - `awaitWriteFinish.pollInterval` (default: 100). File size polling interval, in milliseconds.

Errors

- `ignorePermissionErrors` (default: `false`). Indicates whether to watch files that don't have read permissions if possible. If watching fails due to `EPERM` or `EACCES` with this set to `true`, the errors will be suppressed silently.