# foreground-child

Run a child as if it's the foreground process. Give it stdio. Exit when it exits.

Mostly this module is here to support some use cases around wrapping child processes for test coverage and such. But it's also generally useful any time you want one program to execute another as if it's the "main" process, for example, if a program takes a --cmd argument to execute in some way.

## USAGE

import { foregroundChild } from 'foreground-child'

// hybrid module, this also works:

// const { foregroundChild } = require('foreground-child')

// cats out this file

const child = foregroundChild('cat', [\_\_filename])

// At this point, it's best to just do nothing else.

// return or whatever.

// If the child gets a signal, or just exits, then this

// parent process will exit in the same way.

You can provide custom spawn options by passing an object after the program and arguments:

const child = foregroundChild(`cat ${\_\_filename}`, { shell: true })

A callback can optionally be provided, if you want to perform an action before your foreground-child exits:

const child = foregroundChild('cat', [\_\_filename], spawnOptions, () => {

doSomeActions()

})

The callback can return a Promise in order to perform asynchronous actions. If the callback does not return a promise, then it must complete its actions within a single JavaScript tick.

const child = foregroundChild('cat', [\_\_filename], async () => {

await doSomeAsyncActions()

})

If the callback throws or rejects, then it will be unhandled, and node will exit in error.

If the callback returns a string value, then that will be used as the signal to exit the parent process. If it returns a number, then that number will be used as the parent exit status code. If it returns boolean false, then the parent process will not be terminated. If it returns undefined, then it will exit with the same signal/code as the child process.

## Caveats

The "normal" standard IO file descriptors (0, 1, and 2 for stdin, stdout, and stderr respectively) are shared with the child process. Additionally, if there is an IPC channel set up in the parent, then messages are proxied to the child on file descriptor 3.

In Node, it's possible to also map arbitrary file descriptors into a child process. In these cases, foreground-child will not map the file descriptors into the child. If file descriptors 0, 1, or 2 are used for the IPC channel, then strange behavior may happen (like printing IPC messages to stderr, for example).

Note that a SIGKILL will always kill the parent process, but will not proxy the signal to the child process, because SIGKILL cannot be caught. In order to address this, a special "watchdog" child process is spawned which will send a SIGKILL to the child process if it does not terminate within half a second after the watchdog receives a SIGHUP due to its parent terminating.

On Windows, issuing a process.kill(process.pid, signal) with a fatal termination signal may cause the process to exit with a 1 status code rather than reporting the signal properly. This module tries to do the right thing, but on Windows systems, you may see that incorrect result. There is as far as I'm aware no workaround for this.

## util: foreground-child/proxy-signals

If you just want to proxy the signals to a child process that the main process receives, you can use the proxy-signals export from this package.

import { proxySignals } from 'foreground-child/proxy-signals'

const childProcess = spawn('command', ['some', 'args'])

proxySignals(childProcess)

Now, any fatal signal received by the current process will be proxied to the child process.

It doesn't go in the other direction; ie, signals sent to the child process will not affect the parent. For that, listen to the child exit or close events, and handle them appropriately.

## util: foreground-child/watchdog

If you are spawning a child process, and want to ensure that it isn't left dangling if the parent process exits, you can use the watchdog utility exported by this module.

import { watchdog } from 'foreground-child/watchdog'

const childProcess = spawn('command', ['some', 'args'])

const watchdogProcess = watchdog(childProcess)

// watchdogProcess is a reference to the process monitoring the

// parent and child. There's usually no reason to do anything

// with it, as it's silent and will terminate

// automatically when it's no longer needed.