

gatsby-worker

Utility to execute tasks in forked processes. Highly inspired by `jest-worker`.

Example

File `worker.ts`:

```
export async function heavyTask(param: string): Promise<string> {  
  // using workers is ideal for CPU intensive tasks  
  return await heavyProcessing(param)  
}  
  
export async function setupStep(param: string): Promise<void> {  
  await heavySetup(param)  
}
```

File `parent.ts`:

```
import { WorkerPool } from "gatsby-worker"  
  
const workerPool = new WorkerPool<typeof import("./worker")>(  
  require.resolve(`./worker`),  
  {  
    numWorkers: 5,  
    env: {  
      CUSTOM_ENV_VAR_TO_SET_IN_WORKER: `foo`,  
    },  
    silent: false,  
  }  
)  
  
// queue a task on all workers  
const arrayOfPromises = workerPool.all.setupStep(`bar`)  
  
// queue a task on single worker  
const singlePromise = workerPool.single.heavyTask(`baz`)
```

API

Constructor

```
// TypeOfWorkerModule allows to type exposed functions ensuring type safety.  
// It will convert sync methods to async and discard/disallow usage of exports  
// that are not functions. Recommended to use with `<typeof import("path_to_worker_module")`.  
const workerPool = new WorkerPool<TypeOfWorkerModule>(  
  // Absolute path to worker module. Recommended to use with `require.resolve`  
  workerPath: string,
```

```

    // Not required options
    options?: {
        // Number of workers to spawn. Defaults to `1` if not defined.
        numWorkers?: number
        // Additional env vars to set in worker. Worker will inherit env vars of parent process
        // as well as additional `GATSBY_WORKER_ID` env var (starting with "1" for first worker, etc)
        env?: Record<string, string>,
        // Whether or not the output from forked process should be ignored. Defaults to `false` if not defined.
        silent?: boolean,
    }
)

.single

// Exports of the worker module become available under `.single` property of `WorkerPool` instance
// Calling those will either start executing immediately if there are any idle workers or queue
// to be executed once a worker becomes idle.
const singlePromise = workerPool.single.heavyTask(`baz`)

.all

// Exports of the worker module become available under `.all` property of `WorkerPool` instance
// Calling those will ensure a function is executed on all workers. Best usage for this is for
// setup/bootstrap of workers.
const arrayOfPromises = workerPool.all.setupStep(`baz`)

.end

// Used to shutdown `WorkerPool`. If there are any in progress or queued tasks, promises for those
const arrayOfPromises = workerPool.end()

isWorker

// Determine if current context is executed in worker context. Useful for conditional handling
import { isWorker } from "gatsby-worker"

if (isWorker) {
    // this is executed in worker context
} else {
    // this is NOT executed in worker context
}

```

Messaging

`gatsby-worker` allows sending messages from worker to main and from main to worker at any time.

Sending messages from worker File message-types.ts:

```
// `gatsby-worker` supports message types. Creating common module that centralize possible /  
// that is shared by worker and parent will ensure messages type safety.
```

```
interface IPingMessage {  
  type: `PING`  
}
```

```
interface IAnotherMessageFromChild {  
  type: `OTHER_MESSAGE_FROM_CHILD`  
  payload: {  
    foo: string  
  }  
}
```

```
export type MessagesFromChild = IPingMessage | IAnotherMessageFromChild
```

```
interface IPongMessage {  
  type: `PONG`  
}
```

```
interface IAnotherMessageFromParent {  
  type: `OTHER_MESSAGE_FROM_PARENT`  
  payload: {  
    foo: string  
  }  
}
```

```
export type MessagesFromParent = IPongMessage | IAnotherMessageFromParent
```

File worker.ts:

```
import { getMessenger } from "gatsby-worker"
```

```
import { MessagesFromParent, MessagesFromChild } from "../message-types"
```

```
const messenger = getMessenger<MessagesFromParent, MessagesFromChild>()  
// messenger might be `undefined` if `getMessenger`  
// is called NOT in worker context  
if (messenger) {  
  // send a message to a parent  
  messenger.send({ type: `PING` })  
  messenger.send({  
    type: `OTHER_MESSAGE_FROM_CHILD`,  
    payload: {  
      foo: `bar`,  
    },  
  },
```

```

    })

    // following would cause type error as message like that is
    // not part of MessagesFromChild type union
    // messenger.send({ type: `NOT_PART_OF_TYPES` })

    // start listening to messages from parent
    messenger.onMessage(msg => {
      switch (msg.type) {
        case `PONG`: {
          // handle PONG message
          break
        }
        case `OTHER_MESSAGE_FROM_PARENT`: {
          // msg.payload.foo will be typed as `string` here
          // handle
          break
        }
      }

      // following would cause type error as there is no msg with
      // given type as part of MessagesFromParent type union
      // case `NOT_PART_OF_TYPES`: {}
    })
  }
}

```

File parent.ts:

```

import { getMessenger } from "gatsby-worker"

import { MessagesFromParent, MessagesFromChild } from "../message-types"

const workerPool = new WorkerPool<
  typeof import("../worker"),
  MessagesFromParent,
  MessagesFromChild
>(
  workerPath: require.resolve("../worker")
)

// `sendMessage` on WorkerPool instance requires second parameter
// `workerId` to specify to which worker to send message to
// (`workerId` starts at 1 for first worker).
workerPool.sendMessage(
  {
    type: `OTHER_MESSAGE_FROM_PARENT`,
    payload: {

```

```

        foo: `baz`
      }
    },
    1
  )

  // start listening to messages from child
  // `onMessage` callback will be called with message sent from worker
  // and `workerId` (to identify which worker send this message)
  workerPool.onMessage((msg: MessagesFromChild, workerId: number): void => {
    switch(msg.type) {
      case `PING`: {
        // send message back making sure we send it back to same worker
        // that sent `PING` message
        workerPool.sendMessage({ type: `PONG` }, workerId)
        break
      }
    }
  })
})

```

Usage with unit tests

If you are working with source files that need transpilation, you will need to make it possible to load untranspiled modules in child processes. This can be done with `@babel/register` (or similar depending on your build toolchain). Example setup:

```

const testWorkerPool = new WorkerPool<WorkerModuleType>(workerModule, {
  numWorkers,
  env: {
    NODE_OPTIONS: `--require ${require.resolve(`./ts-register`)}`,
  },
})

```

This will execute additional module before allowing adding runtime support for new JavaScript syntax or support for TypeScript. Example `ts-register.js`:

```

// spawned process won't use jest config (or other testing framework equivalent) to support
require(`@babel/register`)({
  extensions: [`.js`, `*.ts`],
  configFile: require.resolve(relativePathToYourBabelConfig),
  ignore: [/node_modules/],
})

```