pino

Very low overhead Node.js logger.

Documentation

- Benchmarks ⇗
- API ⇗
- Browser API ⇗
- Redaction ⇗
- Child Loggers ⇗
- Transports ⇗
- Web Frameworks ⇗
- Pretty Printing ⇗
- Asynchronous Logging ⇗

- Ecosystem ⇗
- Legacy
- Help ⇗
- Long Term Support Policy ⇗

Install

```
$ npm install pino
```

If you would like to install pino v6, refer to https://github.com/pinojs/pino/tree/v6.x.

Usage

```js
const logger = require('pino')()

logger.info('hello world')

const child = logger.child({ a: 'property' })
child.info('hello child!')
```

This produces:

```
{"level":30,"time":1531171074631,"msg":"hello world","pid":657,"hostname":"Davids-MBP-3.fritz.box"}
{"level":30,"time":1531171082399,"msg":"hello child!","pid":657,"hostname":"Davids-MBP-3.fritz.box"
```

For using Pino with a web framework see:

- Pino with Fastify
- Pino with Express
- Pino with Hapi
- Pino with Restify
- Pino with Koa
- Pino with Node core `http`
- Pino with Nest

Essentials

Development Formatting

The `pino-pretty` module can be used to format logs during development:

```
% cat app.log | pino-pretty -c -l
INFO [1459529098958] (94473 on MacBook-Pro-3.home): hello world
ERROR [1459529098959] (94473 on MacBook-Pro-3.home): this is at error level
INFO [1459529098960] (94473 on MacBook-Pro-3.home): the answer is 42
INFO [1459529098960] (94473 on MacBook-Pro-3.home): hello world
    obj: 42
INFO [1459529098960] (94473 on MacBook-Pro-3.home): hello world
    obj: 42
    b: 2
INFO [1459529098960] (94473 on MacBook-Pro-3.home): another
    obj: {
      "aa": "bbb"
    }
ERROR [1459529098961] (94473 on MacBook-Pro-3.home): an error
    Error: an error
        at Object.<anonymous> (/Users/davidclements/z/nearForm/pino/example.js:14:12)
        at Module._compile (module.js:435:26)
        at Object.Module._extensions..js (module.js:442:10)
        at Module.load (module.js:356:32)
        at Function.Module._load (module.js:311:12)
        at Function.Module.runMain (module.js:467:10)
        at startup (node.js:136:18)
        at node.js:963:3
INFO [1459529098962] (94473 on MacBook-Pro-3.home): hello child!
    a: "property"
INFO [1459529098962] (94473 on MacBook-Pro-3.home): hello baby..
    another: "property"
    a: "property"
INFO [1459529098963] (94473 on MacBook-Pro-3.home): after setImmediate
```

Transports & Log Processing

Due to Node's single-threaded event-loop, it's highly recommended that sending, alert triggering, reformatting and all forms of log processing be conducted in a separate process or thread.

In Pino terminology we call all log processors "transports", and recommend that the transports be run in a worker thread using our `pino.transport` API.

For more details see our Transports⇗ document.

Low overhead

Using minimum resources for logging is very important. Log messages tend to get added over time and this can lead to a throttling effect on applications – such as reduced requests per second.

In many cases, Pino is over 5x faster than alternatives.

See the Benchmarks document for comparisons.

Bundling support

Pino supports to being bundled using tools like webpack or esbuild.

See Bundling document for more informations.

The Team

Matteo Collina

https://github.com/pinojs

https://www.npmjs.com/~matteo.collina

https://twitter.com/matteocollina

David Mark Clements

https://github.com/davidmarkclements

https://www.npmjs.com/~davidmarkclements

https://twitter.com/davidmarkclem

James Sumners

https://github.com/jsumners

https://www.npmjs.com/~jsumners

https://twitter.com/jsumners79

Thomas Watson Steen

https://github.com/watson

https://www.npmjs.com/~watson

https://twitter.com/wa7son

Contributing

Pino is an **OPEN Open Source Project**. This means that:

> Individuals making significant and valuable contributions are given commit-access to the project to contribute as they see fit. This project is more like an open wiki than a standard guarded open source project.

See the CONTRIBUTING.md file for more details.

Acknowledgements

This project was kindly sponsored by nearForm.

Logo and identity designed by Cosmic Fox Design: https://www.behance.net/cosmicfox.

License

Licensed under MIT.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

API

`pino([options], [destination]) => logger`

The exported `pino` function takes two optional arguments, `options` and `destination` and returns a logger instance.

`options` (Object)

`name` (String)

Default: `undefined`

The name of the logger. When set adds a `name` field to every JSON line logged.

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

`level` (String)

Default: `'info'`

One of `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` or `'silent'`.

Additional levels can be added to the instance via the `customLevels` option.

- See `customLevels` option

`customLevels` (Object)

Default: `undefined`

Use this option to define additional logging levels. The keys of the object correspond the namespace of the log level, and the values should be the numerical value of the level.

```
const logger = pino({
  customLevels: {
    foo: 35
  }
})
logger.foo('hi')
```

`useOnlyCustomLevels` (Boolean)

Default: `false`

Use this option to only use defined `customLevels` and omit Pino's levels. Logger's default `level` must be changed to a value in `customLevels` in order to use `useOnlyCustomLevels` Warning: this option may not be supported by downstream transports.

```
const logger = pino({
  customLevels: {
    foo: 35
  },
  useOnlyCustomLevels: true,
  level: 'foo'
})
logger.foo('hi')
logger.info('hello') // Will throw an error saying info in not found in logger object
```

`depthLimit` (Number)

Default: `5`

Option to limit stringification at a specific nesting depth when logging circular object.

`edgeLimit` (Number)

Default: `100`

Option to limit stringification of properties/elements when logging a specific object/array with circular references.

`mixin` (Function):

Default: `undefined`

If provided, the `mixin` function is called each time one of the active logging methods is called. The first and only parameter is the value `mergeObject` or an empty object. The function must synchronously return an object. The properties of the returned object will be added to the logged JSON.

```
let n = 0
const logger = pino({
  mixin () {
    return { line: ++n }
  }
})
logger.info('hello')
// {"level":30,"time":1573664685466,"pid":78742,"hostname":"x","line":1,"msg":"hello"}
logger.info('world')
// {"level":30,"time":1573664685469,"pid":78742,"hostname":"x","line":2,"msg":"world"}
```

The result of `mixin()` is supposed to be a *new* object. For performance reason, the object returned by `mixin()` will be mutated by pino. In the following example, passing `mergingObject` argument to the first `info` call will mutate the global `mixin` object by default: (* See `mixinMergeStrategy` option):

```
const mixin = {
    appName: 'My app'
}

const logger = pino({
    mixin() {
        return mixin;
    }
```

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

```
})

logger.info({
    description: 'Ok'
}, 'Message 1')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok" "msg":"Message 1"}
logger.info('Message 2')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","appName":"My app","description":"Ok","msg":"Message 2"}
// Note: the second log contains "description":"Ok" text, even if it was not provided.
```

If the `mixin` feature is being used merely to add static metadata to each log message, then a child logger ⇗ should be used instead.

`mixinMergeStrategy` (Function):

Default: `undefined`

If provided, the `mixinMergeStrategy` function is called each time one of the active logging methods is called. The first parameter is the value `mergeObject` or an empty object, the second parameter is the value resulting from `mixin()` (* See `mixin` option or an empty object. The function must synchronously return an object.

```
// Default strategy, `mergeObject` has priority
const logger = pino({
    mixin() {
        return { tag: 'docker' }
    },
    // mixinMergeStrategy(mergeObject, mixinObject) {
    //     return Object.assign(mixinMeta, mergeObject)
    // }
})

logger.info({
  tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"local","msg":"Message"}
```

```
// Custom mutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign(mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

```
// Custom immutable strategy, `mixin` has priority
const logger = pino({
    mixin() {
        return { tag: 'k8s' }
    },
    mixinMergeStrategy(mergeObject, mixinObject) {
        return Object.assign({}, mergeObject, mixinObject)
    }
})

logger.info({
    tag: 'local'
}, 'Message')
// {"level":30,"time":1591195061437,"pid":16012,"hostname":"x","tag":"k8s","msg":"Message"}
```

`redact` (Array | Object):

Default: `undefined`

As an array, the `redact` option specifies paths that should have their values redacted from any log output.

Each path must be a string using a syntax which corresponds to JavaScript dot and bracket notation.

If an object is supplied, three options can be specified:

- `paths` (array): Required. An array of paths. See redaction - Path Syntax ⇗ for specifics.
- `censor` (String|Function|Undefined): Optional. When supplied as a String the `censor` option will overwrite keys which are to be redacted. When set to `undefined` the key will be removed entirely from the object. The `censor` option may also be a mapping function. The (synchronous) mapping function has the signature `(value, path) => redactedValue` and is called with the unredacted `value` and `path` to the key being redacted, as an array. For example given a redaction path of `a.b.c` the `path` argument would be `['a', 'b', 'c']`. The value returned from the mapping function becomes the applied censor value. Default: `'[Redacted]'` value synchronously. Default: `'[Redacted]'`
- `remove` (Boolean): Optional. Instead of censoring the value, remove both the key and the value. Default: `false`

WARNING: Never allow user input to define redacted paths.

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

- See the redaction ⇗ documentation.
- See fast-redact#caveat ⇗

`hooks` (Object)

An object mapping to hook functions. Hook functions allow for customizing internal logger operations. Hook functions ***must*** be synchronous functions.

`logMethod`

Allows for manipulating the parameters passed to logger methods. The signature for this hook is `logMethod (args, method, level) {}`, where `args` is an array of the arguments that were passed to the log method and `method` is the log method itself, `level` is the log level itself. This hook ***must*** invoke the `method` function by using apply, like so: `method.apply(this, newArgumentsArray)`.

For example, Pino expects a binding object to be the first parameter with an optional string message as the second parameter. Using this hook the parameters can be flipped:

```
const hooks = {
  logMethod (inputArgs, method, level) {
    if (inputArgs.length >= 2) {
      const arg1 = inputArgs.shift()
      const arg2 = inputArgs.shift()
      return method.apply(this, [arg2, arg1, ...inputArgs])
    }
    return method.apply(this, inputArgs)
  }
}
```

`formatters` (Object)

An object containing functions for formatting the shape of the log lines. These functions should return a JSONifiable object and should never throw. These functions allow for full customization of the resulting log lines. For example, they can be used to change the level key name or to enrich the default metadata.

`level`

Changes the shape of the log level. The default shape is `{ level: number }`. The function takes two arguments, the label of the level (e.g. `'info'`) and the numeric value (e.g. `30`).

```
const formatters = {
  level (label, number) {
    return { level: number }
  }
}
```

`bindings`

Changes the shape of the bindings. The default shape is `{ pid, hostname }`. The function takes a single argument, the bindings object. It will be called every time a child logger is created.

```
const formatters = {
  bindings (bindings) {
    return { pid: bindings.pid, hostname: bindings.hostname }
  }
}
```

`log`

Changes the shape of the log object. This function will be called every time one of the log methods (such as `.info`) is called. All arguments passed to the log method, except the message, will be pass to this function. By default it does not change the shape of the log object.

```
const formatters = {
  log (object) {
    return object
  }
}
```

`serializers` (Object)

Default: `{err: pino.stdSerializers.err}`

An object containing functions for custom serialization of objects. These functions should return an JSONifiable object and they should never throw. When logging an object, each top-level property matching the exact key of a serializer will be serialized using the defined serializer.

The serializers are applied when a property in the logged object matches a property in the serializers. The only exception is the `err` serializer as it is also applied in case the object is an instance of `Error`, e.g. `logger.info(new Error('kaboom'))`.

- See pino.stdSerializers

`base` (Object)

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

Default: `{pid: process.pid, hostname: os.hostname}`

Key-value object added as child logger to each log line.

Set to `undefined` to avoid adding `pid`, `hostname` properties to each log.

`enabled` (Boolean)

Default: `true`

Set to `false` to disable logging.

`crlf` (Boolean)

Default: `false`

Set to `true` to logs newline delimited JSON with `\r\n` instead of `\n`.

`timestamp` (Boolean | Function)

Default: `true`

Enables or disables the inclusion of a timestamp in the log message. If a function is supplied, it must synchronously return a partial JSON string representation of the time, e.g. `,"time":1493426328206` (which is the default).

If set to `false`, no timestamp will be included in the output.

See stdTimeFunctions for a set of available functions for passing in as a value for this option.

Example:

```
timestamp: () => `,"time":"${new Date(Date.now()).toISOString()}"`
// which is equivalent to:
// timestamp: stdTimeFunctions.isoTime
```

Caution: attempting to format time in-process will significantly impact logging performance.

`messageKey` (String)

Default: `'msg'`

The string key for the 'message' in the JSON object.

`nestedKey` (String)

Default: `null`

If there's a chance that objects being logged have properties that conflict with those from pino itself (`level`, `timestamp`, `pid`, etc) and duplicate keys in your log records are undesirable, pino can be configured with a `nestedKey` option that causes any `object`s that are logged to be placed under a key whose name is the value of `nestedKey`.

This way, when searching something like Kibana for values, one can consistently search under the configured `nestedKey` value instead of the root log record keys.

For example,

```
const logger = require('pino')({
  nestedKey: 'payload'
})

const thing = { level: 'hi', time: 'never', foo: 'bar'} // has pino-conflicting properties!
logger.info(thing)

// logs the following:
// {"level":30,"time":1578357790020,"pid":91736,"hostname":"x","payload":{"level":"hi","time":"never","foo":"bar"}}
```

In this way, logged objects' properties don't conflict with pino's standard logging properties, and searching for logged objects can start from a consistent path.

`prettyPrint` (Boolean | Object)

Default: `false`

DEPRECATED: look at pino-pretty documentation for alternatives. Using a `transport` is also an option.__

Enables pretty printing log logs. This is intended for non-production configurations. This may be set to a configuration object as outlined in the `pino-pretty` documentation.

The options object may additionally contain a `prettifier` property to define which prettifier module to use. When not present, `prettifier` defaults to `'pino-pretty'`. Regardless of the value, the specified prettifier module must be installed as a separate dependency:

```
npm install pino-pretty
```

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

`browser` (Object)

Browser only, may have `asObject` and `write` keys. This option is separately documented in the Browser API ⇗ documentation.

- See Browser API ⇗

`transport` (Object)

The `transport` option is a shorthand for the pino.transport() function. It supports the same input options:

```
require('pino')({
  transport: {
    target: '/absolute/path/to/my-transport.mjs'
  }
})

// or multiple transports
require('pino')({
  transport: {
    targets: [
      { target: '/absolute/path/to/my-transport.mjs', level: 'error' },
      { target: 'some-file-transport', options: { destination: '/dev/null' }
    ]
  }
})
```

If the transport option is supplied to `pino`, a `destination` parameter may not also be passed as a separate argument to `pino`:

```
pino({ transport: {}}, '/path/to/somewhere') // THIS WILL NOT WORK, DO NOT DO THIS
pino({ transport: {}}, process.stderr) // THIS WILL NOT WORK, DO NOT DO THIS
```

when using the `transport` option. In this case an `Error` will be thrown.

- See pino.transport()

`destination` (SonicBoom | WritableStream | String | Object)

Default: `pino.destination(1)` (STDOUT)

The `destination` parameter, at a minimum must be an object with a `write` method. An ordinary Node.js `stream` can be passed as the destination (such as the result of `fs.createWriteStream`) but for peak log writing performance it is strongly recommended to use `pino.destination` to create the destination stream. Note that the `destination` parameter can be the result of `pino.transport()`.

```
// pino.destination(1) by default
const stdoutLogger = require('pino')()

// destination param may be in first position when no options:
const fileLogger = require('pino')( pino.destination('/log/path'))

// use the stderr file handle to log to stderr:
const opts = {name: 'my-logger'}
const stderrLogger = require('pino')(opts, pino.destination(2))

// automatic wrapping in pino.destination
const fileLogger = require('pino')('/log/path')

// Asynchronous logging
const fileLogger = pino(pino.destination({ dest: '/log/path', sync: false }))
```

However, there are some special instances where `pino.destination` is not used as the default:

- When something, e.g a process manager, has monkey-patched `process.stdout.write`.

In these cases `process.stdout` is used instead.

Note: If the parameter is a string integer, e.g. `'1'`, it will be coerced to a number and used as a file descriptor. If this is not desired, provide a full path, e.g. `/tmp/1`.

- See `pino.destination`

`destination[Symbol.for('pino.metadata')]`

Default: `false`

Using the global symbol `Symbol.for('pino.metadata')` as a key on the `destination` parameter and setting the key it to `true`, indicates that the following properties should be set on the `destination` object after each log line is written:

- the last logging level as `destination.lastLevel`
- the last logging message as `destination.lastMsg`
- the last logging object as `destination.lastObj`
- the last time as `destination.lastTime`, which will be the partial string returned by the time function.
- the last logger instance as `destination.lastLogger` (to support child loggers)

The following is a succinct usage example:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const dest = pino.destination('/dev/null')
dest[Symbol.for('pino.metadata')] = true
const logger = pino(dest)
logger.info({a: 1}, 'hi')
const { lastMsg, lastLevel, lastObj, lastTime} = dest
console.log(
  'Logged message "%s" at level %d with object %o at time %s',
  lastMsg, lastLevel, lastObj, lastTime
) // Logged message "hi" at level 30 with object { a: 1 } at time 1531590545089
```

Logger Instance

The logger instance is the object returned by the main exported `pino` function.

The primary purpose of the logger instance is to provide logging methods.

The default logging methods are `trace`, `debug`, `info`, `warn`, `error`, and `fatal`.

Each logging method has the following signature: `([mergingObject], [message], [...interpolationValues])`.

The parameters are explained below using the `logger.info` method but the same applies to all logging methods.

Logging Method Parameters

`mergingObject` (Object)

An object can optionally be supplied as the first parameter. Each enumerable key and value of the `mergingObject` is copied in to the JSON log line.

```
logger.info({MIX: {IN: true}})
// {"level":30,"time":1531254555820,"pid":55956,"hostname":"x","MIX":{"IN":true}}
```

If the object is of type Error, it is wrapped in an object containing a property err (`{ err: mergingObject }`). This allows for a unified error handling flow.

`message` (String)

A `message` string can optionally be supplied as the first parameter, or as the second parameter after supplying a `mergingObject`.

By default, the contents of the `message` parameter will be merged into the JSON log line under the `msg` key:

```
logger.info('hello world')
// {"level":30,"time":1531257112193,"msg":"hello world","pid":55956,"hostname":"x"}
```

The `message` parameter takes precedence over the `mergingObject`. That is, if a `mergingObject` contains a `msg` property, and a `message` parameter is supplied in addition, the `msg` property in the output log will be the value of the `message` parameter not the value of the `msg` property on the `mergingObject`. See Avoid Message Conflict for information on how to overcome this limitation.

If no `message` parameter is provided, and the `mergingObject` is of type `Error` or it has a property named `err`, the `message` parameter is set to the `message` value of the error.

The `messageKey` option can be used at instantiation time to change the namespace from `msg` to another string as preferred.

The `message` string may contain a printf style string with support for the following placeholders:

- `%s` – string placeholder
- `%d` – digit placeholder
- `%O`, `%o` and `%j` – object placeholder

Values supplied as additional arguments to the logger method will then be interpolated accordingly.

- See `messageKey` pino option
- See `...interpolationValues` log method parameter

`...interpolationValues` (Any)

All arguments supplied after `message` are serialized and interpolated according to any supplied printf-style placeholders (`%s`, `%d`, `%o` | `%O` | `%j`) to form the final output `msg` value for the JSON log line.

```
logger.info('%o hello %s', {worldly: 1}, 'world')
// {"level":30,"time":1531257826880,"msg":"{\"worldly\":1} hello world","pid":55956,"hostname":"x"}
```

Since pino v6, we do not automatically concatenate and cast to string consecutive parameters:

```
logger.info('hello', 'world')
// {"level":30,"time":1531257618044,"msg":"hello","pid":55956,"hostname":"x"}
// world is missing
```

However, it's possible to inject a hook to modify this behavior:

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

```
const pinoOptions = {
  hooks: { logMethod }
}

function logMethod (args, method) {
  if (args.length === 2) {
    args[0] = `${args[0]} %j`
  }
  method.apply(this, args)
}

const logger = pino(pinoOptions)
```

- See `message` log method parameter
- See `logMethod` hook

Errors

Errors can be supplied as either the first parameter or if already using `mergingObject` then as the `err` property on the `mergingObject`.

> **Note**
>
> This section describes the default configuration. The error serializer can be mapped to a different key using the `serializers` option.

```
logger.info(new Error("test"))
// {"level":30,"time":1531257618044,"msg":"test","stack":"...","type":"Error","pid":55956,"hostname":"x"}

logger.info({ err: new Error("test"), otherkey: 123 }, "some text")
// {"level":30,"time":1531257618044,"err":{"msg": "test", "stack":"...","type":"Error"},"msg":"some text","pid":55956,"hos
```

`logger.trace([mergingObject], [message], [...interpolationValues])`

Write a `'trace'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.debug([mergingObject], [message], [...interpolationValues])`

Write a `'debug'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.info([mergingObject], [message], [...interpolationValues])`

Write an `'info'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.warn([mergingObject], [message], [...interpolationValues])`

Write a `'warn'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.error([mergingObject], [message], [...interpolationValues])`

Write a `'error'` level log, if the configured `level` allows for it.

- See `mergingObject` log method parameter
- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.fatal([mergingObject], [message], [...interpolationValues])`

Write a `'fatal'` level log, if the configured `level` allows for it.

Since `'fatal'` level messages are intended to be logged just prior to the process exiting the `fatal` method will always sync flush the destination. Therefore it's important not to misuse `fatal` since it will cause performance overhead if used for any other purpose than writing final log messages before the process crashes or exits.

- See `mergingObject` log method parameter

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

- See `message` log method parameter
- See `...interpolationValues` log method parameter

`logger.silent()`

Noop function.

`logger.child(bindings, [options]) => logger`

The `logger.child` method allows for the creation of stateful loggers, where key-value pairs can be pinned to a logger causing them to be output on every log line.

Child loggers use the same output stream as the parent and inherit the current log level of the parent at the time they are spawned.

The log level of a child is mutable. It can be set independently of the parent either by setting the `level` accessor after creating the child logger or using the `options.level` key.

`bindings` (Object)

An object of key-value pairs to include in every log line output via the returned child logger.

```
const child = logger.child({ MIX: {IN: 'always'} })
child.info('hello')
// {"level":30,"time":1531258616689,"msg":"hello","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
child.info('child!')
// {"level":30,"time":1531258617401,"msg":"child!","pid":64849,"hostname":"x","MIX":{"IN":"always"}}
```

The `bindings` object may contain any key except for reserved configuration keys `level` and `serializers`.

`bindings.serializers` (Object) - DEPRECATED

Use `options.serializers` instead.

`options` (Object)

Options for child logger. These options will override the parent logger options.

`options.level` (String)

The `level` property overrides the log level of the child logger. By default the parent log level is inherited. After the creation of the child logger, it is also accessible using the `logger.level` key.

```
const logger = pino()
logger.debug('nope') // will not log, since default level is info
const child = logger.child({foo: 'bar'}, {level: 'debug'})
child.debug('debug!') // will log as the `level` property set the level to debug
```

`options.redact` (Array | Object)

Setting `options.redact` to an array or object will override the parent `redact` options. To remove `redact` options inherited from the parent logger set this value as an empty array (`[]`).

```
const logger = require('pino')({ redact: ['hello'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794363403,"pid":67930,"hostname":"x","hello":"[Redacted]"}
const child = logger.child({ foo: 'bar' }, { redact: ['foo'] })
logger.info({ hello: 'world' })
// {"level":30,"time":1625794553558,"pid":67930,"hostname":"x","hello":"world", "foo": "[Redacted]" }
```

- See `redact` option

`options.serializers` (Object)

Child loggers inherit the serializers from the parent logger.

Setting the `serializers` key of the `options` object will override any configured parent serializers.

```
const logger = require('pino')()
logger.info({test: 'will appear'})
// {"level":30,"time":1531259759482,"pid":67930,"hostname":"x","test":"will appear"}
const child = logger.child({}, {serializers: {test: () => `child-only serializer`}})
child.info({test: 'will be overwritten'})
// {"level":30,"time":1531259784008,"pid":67930,"hostname":"x","test":"child-only serializer"}
```

- See `serializers` option
- See pino.stdSerializers

`logger.bindings()`

Returns an object containing all the current bindings, cloned from the ones passed in via `logger.child()`.

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
const child = logger.child({ foo: 'bar' })
console.log(child.bindings())
// { foo: 'bar' }
const anotherChild = child.child({ MIX: { IN: 'always' } })
console.log(anotherChild.bindings())
// { foo: 'bar', MIX: { IN: 'always' } }
```

`logger.flush()`

Flushes the content of the buffer when using `pino.destination({ sync: false })`.

This is an asynchronous, fire and forget, operation.

The use case is primarily for asynchronous logging, which may buffer log lines while others are being written. The `logger.flush` method can be used to flush the logs on a long interval, say ten seconds. Such a strategy can provide an optimum balance between extremely efficient logging at high demand periods and safer logging at low demand periods.

- See `destination` parameter
- See Asynchronous Logging ⇗

`logger.level` (String) [Getter/Setter]

Set this property to the desired logging level.

The core levels and their values are as follows:

Level:	trace	debug	info	warn	error	fatal	silent
Value:	10	20	30	40	50	60	Infinity

The logging level is a *minimum* level based on the associated value of that level.

For instance if `logger.level` is `info` *(30)* then `info` *(30)*, `warn` *(40)*, `error` *(50)* and `fatal` *(60)* log methods will be enabled but the `trace` *(10)* and `debug` *(20)* methods, being less than 30, will not.

The `silent` logging level is a specialized level which will disable all logging, the `silent` log method is a noop function.

`logger.isLevelEnabled(level)`

A utility method for determining if a given log level will write to the destination.

`level` (String)

The given level to check against:

```
if (logger.isLevelEnabled('debug')) logger.debug('conditional log')
```

`levelLabel` (String)

Defines the method name of the new level.

- See `logger.level`

`levelValue` (Number)

Defines the associated minimum threshold value for the level, and therefore where it sits in order of priority among other levels.

- See `logger.level`

`logger.levelVal` (Number)

Supplies the integer value for the current logging level.

```
if (logger.levelVal === 30) {
  console.log('logger level is `info`')
}
```

`logger.levels` (Object)

Levels are mapped to values to determine the minimum threshold that a logging method should be enabled at (see `logger.level`).

The `logger.levels` property holds the mappings between levels and values, and vice versa.

```
$ node -p "require('pino')().levels"
```

```
{ labels:
   { '10': 'trace',
     '20': 'debug',
     '30': 'info',
```

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

```
    '40': 'warn',
    '50': 'error',
    '60': 'fatal' },
  values:
   { fatal: 60, error: 50, warn: 40, info: 30, debug: 20, trace: 10 } }
```

- See `logger.level`

logger[Symbol.for('pino.serializers')]

Returns the serializers as applied to the current logger instance. If a child logger did not register it's own serializer upon instantiation the serializers of the parent will be returned.

Event: 'level-change'

The logger instance is also an `EventEmitter ⇗`

A listener function can be attached to a logger via the `level-change` event

The listener is passed four arguments:

- `levelLabel` – the new level string, e.g `trace`
- `levelValue` – the new level number, e.g `10`
- `previousLevelLabel` – the prior level string, e.g `info`
- `previousLevelValue` – the prior level numbebr, e.g `30`

```
const logger = require('pino')()
logger.on('level-change', (lvl, val, prevLvl, prevVal) => {
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.level = 'trace' // trigger event
```

Please note that due to a known bug, every `logger.child()` call will fire a `level-change` event. These events can be ignored by writing an event handler like:

```
const logger = require('pino')()
logger.on('level-change', function (lvl, val, prevLvl, prevVal) {
  if (logger !== this) {
    return
  }
  console.log('%s (%d) was changed to %s (%d)', prevLvl, prevVal, lvl, val)
})
logger.child({}); // trigger an event by creating a child instance, notice no console.log
logger.level = 'trace' // trigger event using actual value change, notice console.log
```

`logger.version` (String)

Exposes the Pino package version. Also available on the exported `pino` function.

- See `pino.version`

Statics

`pino.destination([opts]) => SonicBoom`

Create a Pino Destination instance: a stream-like object with significantly more throughput (over 30%) than a standard Node.js stream.

```
const pino = require('pino')
const logger = pino(pino.destination('./my-file'))
const logger2 = pino(pino.destination())
const logger3 = pino(pino.destination({
  dest: './my-file',
  minLength: 4096, // Buffer before writing
  sync: false // Asynchronous logging
}))
```

The `pino.destination` method may be passed a file path or a numerical file descriptor. By default, `pino.destination` will use `process.stdout.fd` (1) as the file descriptor.

`pino.destination` is implemented on `sonic-boom ⇗`.

A `pino.destination` instance can also be used to reopen closed files (for example, for some log rotation scenarios), see Reopening log files.

- See `destination` parameter
- See `sonic-boom ⇗`
- See Reopening log files
- See Asynchronous Logging ⇗

`pino.transport(options) => ThreadStream`

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

Create a a stream that routes logs to a worker thread that wraps around a Pino Transport.

```
const pino = require('pino')
const transport = pino.transport({
  target: 'some-transport',
  options: { some: 'options for', the: 'transport' }
})
pino(transport)
```

Multiple transports may also be defined, and specific levels can be logged to each transport:

```
const pino = require('pino')
const transport = pino.transport({
  targets: [{
    level: 'info',
    target: 'pino-pretty' // must be installed separately
  }, {
    level: 'trace',
    target: 'pino/file',
    options: { destination: '/path/to/store/logs' }
  }]
})
pino(transport)
```

A pipeline could also be created to transform log lines *before* sending them:

```
const pino = require('pino')
const transport = pino.transport({
  pipeline: [{
    target: 'pino-syslog' // must be installed separately
  }, {
    target: 'pino-socket' // must be installed separately
  }]
})
pino(transport)
```

If `WeakRef`, `WeakMap` and `FinalizationRegistry` are available in the current runtime (v14.5.0+), then the thread will be automatically terminated in case the stream or logger goes out of scope. The `transport()` function adds a listener to `process.on('beforeExit')` and `process.on('exit')` to ensure the worker is flushed and all data synced before the process exits.

Note that calling `process.exit()` on the main thread will stop the event loop on the main thread from turning. As a result, using `console.log` and `process.stdout` after the main thread called `process.exit()` will not produce any output.

If you are embedding/integrating pino within your framework, you will need to make pino aware of the script that is calling it, like so:

```
const pino = require('pino')
const getCaller = require('get-caller-file')

module.exports = function build () {
  const logger = pino({
    transport: {
      caller: getCaller(),
      target: 'transport',
      options: { destination: './destination' }
    }
  })
  return logger
}
```

For more on transports, how they work, and how to create them see the `Transports documentation`.

- See `Transports`
- See `thread-stream` ⇗

Options

- `target`: The transport to pass logs through. This may be an installed module name or an absolute path.
- `options`: An options object which is serialized (see [Structured Clone Algorithm][https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm]), passed to the worker thread, parsed and then passed to the exported transport function.
- `worker`: Worker thread configuration options. Additionally, the `worker` option supports `worker.autoEnd`. If this is set to `false` logs will not be flushed on process exit. It is then up to the developer to call `transport.end()` to flush logs.
- `targets`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options plus a `level` option which will send only logs above a specified level to a transport.
- `pipeline`: May be specified instead of `target`. Must be an array of transport configurations. Transport configurations include the aforementioned `options` and `target` options. All intermediate steps in the pipeline *must* be `Transform` streams and not `Writable`.

`pino.final(logger, [handler]) => Function | FinalLogger`

The use of `pino.final` is discouraged in Node.js v14+ and not required. It will be removed in the next major version.

The `pino.final` method can be used to acquire a final logger instance or create an exit listener function. This is *not* needed in Node.js v14+ as pino automatically can handle those.

The `finalLogger` is a specialist logger that synchronously flushes on every write. This is important to guarantee final log writes, when using `pino.destination({ sync: false })` target.

Since final log writes cannot be guaranteed with normal Node.js streams, if the `destination` parameter of the `logger` supplied to `pino.final` is a Node.js stream `pino.final` will throw.

The use of `pino.final` with `pino.destination` is not needed, as `pino.destination` writes things synchronously.

`pino.final(logger, handler) => Function`

In this case the `pino.final` method supplies an exit listener function that can be supplied to process exit events such as `exit`, `uncaughtException`, `SIGHUP` and so on.

The exit listener function will call the supplied `handler` function with an error object (or else `null`), a `finalLogger` instance followed by any additional arguments the `handler` may be called with.

```
process.on('uncaughtException', pino.final(logger, (err, finalLogger) => {
  finalLogger.error(err, 'uncaughtException')
  process.exit(1)
}))
```

`pino.final(logger) => FinalLogger`

In this case the `pino.final` method returns a finalLogger instance.

```
var finalLogger = pino.final(logger)
finalLogger.info('exiting...')
```

- See `destination` parameter
- See Exit logging help
- See Asynchronous logging ⇗
- See Log loss prevention ⇗

`pino.multistream(streamsArray, opts) => MultiStreamRes`

Create a stream composed by multiple destination streams and returns an object implementing the MultiStreamRes interface.

```
var fs = require('fs')
var pino = require('pino')
var pretty = require('pino-pretty')
var streams = [
  {stream: fs.createWriteStream('/tmp/info.stream.out')},
  {stream: pretty() },
  {level: 'debug', stream: fs.createWriteStream('/tmp/debug.stream.out')},
  {level: 'fatal', stream: fs.createWriteStream('/tmp/fatal.stream.out')}
]

var log = pino({
  level: 'debug' // this MUST be set at the lowest level of the
                 // destinations
}, pino.multistream(streams))

log.debug('this will be written to /tmp/debug.stream.out')
log.info('this will be written to /tmp/debug.stream.out and /tmp/info.stream.out')
log.fatal('this will be written to /tmp/debug.stream.out, /tmp/info.stream.out and /tmp/fatal.stream.out')
```

In order for `multistream` to work, the log level **must** be set to the lowest level used in the streams array.

Options

- `levels`: Pass custom log level definitions to the instance as an object.
- `dedupe`: Set this to `true` to send logs only to the stream with the higher level. Default: `false`

  `dedupe` flag can be useful for example when using `pino.multistream` to redirect `error` logs to `process.stderr` and others to `process.stdout`:

  ```
  var pino = require('pino')
  var multistream = pino.multistream
  var streams = [
    {stream: process.stdout},
    {level: 'error', stream: process.stderr},
  ]

  var opts = {
      levels: {
          silent: Infinity,
          fatal: 60,
          error: 50,
          warn: 50,
  ```

```
        info: 30,
        debug: 20,
        trace: 10
    },
    dedupe: true,
}

var log = pino({
  level: 'debug' // this MUST be set at the lowest level of the
                 // destinations
}, multistream(streams, opts))

log.debug('this will be written ONLY to process.stdout')
log.info('this will be written ONLY to process.stdout')
log.error('this will be written ONLY to process.stderr')
log.fatal('this will be written ONLY to process.stderr')
```

`pino.stdSerializers` (Object)

The `pino.stdSerializers` object provides functions for serializing objects common to many projects. The standard serializers are directly imported from pino-std-serializers.

- See pino-std-serializers ⇗

`pino.stdTimeFunctions` (Object)

The `timestamp` option can accept a function which determines the `timestamp` value in a log line.

The `pino.stdTimeFunctions` object provides a very small set of common functions for generating the `timestamp` property. These consist of the following

- `pino.stdTimeFunctions.epochTime`: Milliseconds since Unix epoch (Default)
- `pino.stdTimeFunctions.unixTime`: Seconds since Unix epoch
- `pino.stdTimeFunctions.nullTime`: Clears timestamp property (Used when `timestamp: false`)
- `pino.stdTimeFunctions.isoTime`: ISO 8601-formatted time in UTC
- See `timestamp` option

`pino.symbols` (Object)

For integration purposes with ecosystem and third party libraries `pino.symbols` exposes the symbols used to hold non-public state and methods on the logger instance.

Access to the symbols allows logger state to be adjusted, and methods to be overridden or proxied for performant integration where necessary.

The `pino.symbols` object is intended for library implementers and shouldn't be utilized for general use.

`pino.version` (String)

Exposes the Pino package version. Also available on the logger instance.

- See `logger.version`

Interfaces

`MultiStreamRes`

Properties:

- `write(data)`
  - `data` Object | string
  - Returns: void

Write `data` onto the streams held by the current instance.

- `add(dest)`
  - `dest` StreamEntry | DestinationStream
  - Returns: MultiStreamRes

Add `dest` stream to the array of streams of the current instance.

- `flushSync()`
  - Returns: `undefined`

Call `flushSync` on each stream held by the current instance.

- `minLevel`
  - number

The minimum level amongst all the streams held by the current instance.

- `streams`
  - Returns: StreamEntry[]

The array of streams currently held by the current instance.

- `clone(level)`
  - `level` Level
  - Returns: MultiStreamRes

Returns a cloned object of the current instance but with the the provided `level`.

`StreamEntry`

Properties:

- `stream`
  - DestinationStream
- `level`
  - Optional: Level

`DestinationStream`

Properties:

- `write(msg)`
  - `msg` string

Types

`Level`

- Values: `"fatal"` | `"error"` | `"warn"` | `"info"` | `"debug"` | `"trace"`