Разделы документации:

Документация Gulp на русском

> Представленная документация на русском языке является полным адаптированным переводом официальной документации Gulp. Информация регулярно обновляется и поддерживается в актуальном состоянии.

Быстрый старт

Если ранее вы устанавливали Gulp глобально, запустите `npm rm --global gulp` перед тем, как выполнять какие-либо инструкции документации.

Проверьте наличие и версии node, npm и npx

```
node --version
```

```
npm --version
```

```
npx --version
```

Если они не установлены, следуйте инструкциям на сайте Node.js® для их установки.

Установите утилиту командной строки Gulp

```
npm install --global gulp-cli
```

Создайте каталог проекта и перейдите в него

```
npx mkdirp my-project
```

```
cd my-project
```

Создайте файл package.json в каталоге вашего проекта

```
npm init
```

Это поможет вам задать имя, версию, описание проекта и другую информацию.

Установите пакет gulp в ваши зависимости devDependencies

```
npm install --save-dev gulp
```

Проверьте версию Gulp

```
gulp --version
```

Убедитесь, что вывод соответствует или похож на тот, что показан ниже. Если это не так, повторите шаги, описанные в этом руководстве ранее.

```
~/gulp/docs
$ gulp --version
[14:45:59] CLI version 2.0.1
[14:45:59] Local version 4.0.0
```

Создайте gulpfile

Используя редактор кода, создайте файл с именем gulpfile.js в корневом каталоге вашего проекта со следующим содержимым:

```
function defaultTask(cb) {
  // Разместите здесь код дефолтного таска
  cb();
}

exports.default = defaultTask
```

Тест

Запустите команду gulp в каталоге вашего проекта:

```
gulp
```

Для запуска нескольких задач вы можете использовать `gulp <task> <othertask>`.

Результат

Дефолтный таск запущен:

```
~/gulp/docs
$ gulp
[14:45:59] Using gulpfile ~/gulp/docs/gulpfile.js
[14:45:59] Starting 'default'
[14:45:59] Finished 'default' after 1.22 ms
```

JavaScript и Gulpfiles

Gulp позволяет использовать ваши знания JavaScript для написания gulpfiles. Для упрощения работы с файловой системой и командной строкой предусмотрено несколько утилит, однако все остальное, что вы пишете - это чистый JavaScript.

Основы Gulpfile

Gulpfile - это файл в каталоге вашего проекта с именем gulpfile.js (или с заглавной буквы - Gulpfile.js), который автоматически загружается при запуске команды gulp. Чаще всего в данном файле используется API Gulp, например, src(), dest(), series() или parallel(), однако можно использовать любые модули JavaScript или Node. Все функции можно экспортировантъ и зарегистрировать в системе задач gulp.

Транспиляция

Вы можете писать gulpfile, используя язык, который требует транспиляции, такой как TypeScript или Babel. Измените расширение вашего gulpfile.js, чтобы указать язык и установите соответствующий модуль транспилятора.

> Транспиляция — преобразование программы, при котором используется исходный код программы, написанной на одном языке программирования в качестве исходных данных, и производится эквивалентный исходный код на другом языке программирования. Википедия.

- Для TypeScript переименуйте его в `gulpfile.ts` и установите модуль ts-node.
- Для Babel, переименуйте в `gulpfile.babel.js` и установите модуль @babel/register.

Большинство новых версий node поддерживают большинство функций, предоставляемых TypeScript или Babel, за исключением синтаксиса импорта/экспорта. Если требуется только этот синтаксис, переименуйте его в `gulpfile.esm.js` и установите модуль esm.

Разделение gulpfile

Многие пишут всю логику в gulpfile. Если он станет слишком большим, он может быть разбит на отдельные файлы.

Каждый таск можно вынести в отдельный файл, а затем импортировать в свой gulpfile. Это не только позволяет упорядочить проект, но и позволит вам тестировать каждую задачу отдельно, в зависимости от того, что требуется в каждом конкретном случае.

Также можно заменить файл `gulpfile.js` на каталог с именем `gulpfile.js`, который содержит файл `index.js`. Он будет обрабатываться как `gulpfile.js`. Этот каталог может содержать ваши отдельные модули для тасков. Если вы используете транспиллер, назовите папку и файл соответствующим образом.

Создание задач (Тасков)

Каждая задача gulp представляет собой асинхронную функцию JavaScript, которая принимает колбэк с первым аргументом-ошибкой или возвращает поток, промис, источник событий, дочерний процесс или наблюдаемый тип (подробнее об этом позже). Из-за некоторых ограничений платформы синхронные задачи не поддерживаются, однако есть довольно изящная альтернатива (см. Использование async/await).

Экспорт

Задачи (таски) бывают **общедоступными** или **частными**.

- **Общедоступные задачи** экспортируются из вашего gulpfile, что позволяет запускать их командой `gulp`.
- **Частные задачи** создаются для внутреннего использования, как часть `series()` или `parallel()`.

Частная задача выглядит и действует как любая другая задача, но конечный пользователь не может выполнить ее отдельно. Чтобы сделать задачу публичной, экспортируйте ее из своего файла gulpfile.

```
const { series } = require('gulp');

// Функция `clean` не экспортируется, поэтому ее можно считать частной задачей.
// Она все еще может быть использована в составе `series()`.

function clean(cb) {
  // тело функции
  cb();
}

// Функция `build` экспортирована, поэтому она общедоступна и может быть запуще
// Она также может быть использована в составе `series()`.

function build(cb) {
  // тело функции
  cb();
}

exports.build = build;
exports.default = series(clean, build);
```

```
~/gulp/docs
$ gulp --tasks
[11:14:59] Tasks for ~/gulp/docs/gulpfile.js
[11:14:59] ├── build
[11:14:59] └─┬ default
[11:14:59]   └─┬ <series>
[11:14:59]     ├── clean
[11:14:59]     └── build
```

> Раньше функция `task()` использовалась для регистрации ваших функций в качестве тасков. Хоть этот API все еще доступен, экспорт должен быть основным механизмом регистрации, за исключением крайних случаев, когда экспорт не будет работать.

Составление тасков

Gulp предоставляет два мощных метода для создания тасков - `series()` и `parallel()`, позволяющие объединять отдельные задачи в более крупные операции. Оба метода принимают любое количество функций или составных операций. `series()` и `parallel()` могут быть вложены сами в себя или друг в друга на любую глубину.

Чтобы ваши задачи выполнялись по порядку, используйте метод `series()`.

```
const { series } = require('gulp');

function transpile(cb) {
  // тело функции
  cb();
}

function bundle(cb) {
  // тело функции
  cb();
}

exports.build = series(transpile, bundle);
```

Чтобы задачи выполнялись параллельно, объедините их методом `parallel()`.

```
const { parallel } = require('gulp');

function javascript(cb) {
  // тело функции
  cb();
}

function css(cb) {
  // тело функции
  cb();
}

exports.build = parallel(javascript, css);
```

Таски выполняются сразу при вызове series() или parallel(). Это позволяет варьировать состав тасков, в зависимости от условий.

```
const { series } = require('gulp');

function minify(cb) {
  // тело функции
  cb();
}


function transpile(cb) {
```

```
  // тело функции
  cb();
}

function livereload(cb) {
  // тело функции
  cb();
}

if (process.env.NODE_ENV === 'production') {
  exports.build = series(transpile, minify);
} else {
  exports.build = series(transpile, livereload);
}
```

series() и parallel() могут быть вложены на любую глубину.

```
const { series, parallel } = require('gulp');

function clean(cb) {
  // тело функции
  cb();
}

function cssTranspile(cb) {
  // тело функции
  cb();
}

function cssMinify(cb) {
  // тело функции
  cb();
}

function jsTranspile(cb) {
  // тело функции
  cb();
}

function jsBundle(cb) {
  // тело функции
  cb();
}

function jsMinify(cb) {
  // тело функции
```

```
  cb();
}

function publish(cb) {
  // тело функции
  cb();
}

exports.build = series(
  clean,
  parallel(
    cssTranspile,
    series(jsTranspile, jsBundle)
  ),
  parallel(cssMinify, jsMinify),
  publish
);
```

Когда выполняется составная операция, каждая задача будет выполняться каждый раз, когда на нее ссылаются. Например, таск `clean`, на который ссылаются в двух разных задачах, будет запущен дважды и приведет к нежелательным результатам. Вместо этого проведите рефакторинг таска `clean`, который будет указан в окончательном составе.

Если у вас такой код:

```
// Так делать НЕЛЬЗЯ

const { series, parallel } = require('gulp');

const clean = function(cb) {
  // тело функции
  cb();
};

const css = series(clean, function(cb) {
  // тело функции
  cb();
});

const javascript = series(clean, function(cb) {
  // тело функции
  cb();
});

exports.build = parallel(css, javascript);
```

Лучше сделать так:

```
const { series, parallel } = require('gulp');

function clean(cb) {
  // тело функции
  cb();
}

function css(cb) {
  // тело функции
  cb();
}

function javascript(cb) {
  // тело функции
  cb();
}

exports.build = series(clean, parallel(css, javascript));
```

Асинхронное выполнение

Библиотеки Node работают с асинхронностью различными способами. Самым распространенным является колбэк с первым аргументом-ошибкой, где первый аргумент зарезервирован для ошибки. Вы также можете столкнуться с потоками, промисами, источниками событий, дочерними процессами или наблюдаемыми типами. Gulp-таски нормализуют все эти типы асинхронности.

Сигнал выполнения задачи

Когда поток, промис, источник событий, дочерний процесс или наблюдаемый тип возвращают результат выполнения таска, об успехе или ошибке они информируют Gulp для дальнейшего продолжения или завершения. Если задание выполнено с ошибкой, Gulp немедленно прекратит работу и покажет эту ошибку.

При составлении задач с помощью **series()** ошибка завершит всю структуру, и дальнейшие задачи выполняться не будут. При компоновке задач с использованием функции **parallel()** ошибка приведет к завершению текущей компоновки, но другие параллельные задачи могут завершиться (или не завершиться, в зависимости от условий).

Возвращение потока

```
const { src, dest } = require('gulp');

function streamTask() {
  return src('*.js')
    .pipe(dest('output'));
}

exports.default = streamTask;
```

Возвращение промиса

```
unction promiseTask() {
  return Promise.resolve('значение игнорируется');
}

exports.default = promiseTask;
```

Возвращение источника событий

```
const { EventEmitter } = require('events');

function eventEmitterTask() {
  const emitter = new EventEmitter();
  // Эмит должен выполняться асинхронно, иначе Gulp его не сможет прослушать
  setTimeout(() => emitter.emit('finish'), 250);
  return emitter;
}

exports.default = eventEmitterTask;
```

Возвращение дочернего процесса

```
const { exec } = require('child_process');

function childProcessTask() {
  return exec('date');
}

exports.default = childProcessTask;
```

Возвращение наблюдаемого типа

```
const { Observable } = require('rxjs');

function observableTask() {
  return Observable.of(1, 2, 3);
}

exports.default = observableTask;
```

Использование первого агрумента с ошибкой

Если из вашей задачи ничего не возвращается, вы должны использовать колбек с ошибкой, чтобы сообщить о завершении. Колбек будет передан вашей задаче в качестве единственного аргумента с именем `cb()`.

```
function callbackTask(cb) {
  // `cb()` вызывается как пример выполнения какой-то работы
  cb();
}

exports.default = callbackTask;
```

Чтобы указать Gulp, что ошибка произошла в задаче, использующей колбек первого агрумента с ошибкой, вызовите ее с `Error` в качестве единственного аргумента.

```
function callbackError(cb) {
  // `cb()` вызывается как пример выполнения какой-то работы
  cb(new Error('kaboom'));
}

exports.default = callbackError;
```

Однако, чаще всего вы будете передавать этот колбек другому API, а не вызывать его самостоятельно.

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

function passingCallback(cb) {
  fs.access('gulpfile.js', cb);
}

exports.default = passingCallback;
```

Скажем «НЕТ» синхронным задачам

Синхронные задачи больше не поддерживаются. Они часто приводили к тонким ошибкам, которые трудно было отладить, например, могли забыть возвратить потоки из задачи.

Если вы видите предупреждение «Did you forget to signal async completion?», значит ни один из методов, упомянутых выше не использовался. Вам нужно будет использовать колбек первого агрумента с ошибкой или вернуть поток, промис, источник событий, дочерний процесс или наблюдаемый тип для решения проблемы.

Использование async/await

Если вы не используете ни одну из упомянутых опций, вы можете определить свою задачу как `asyn` функцию, которая оборачивает вашу задачу в промис. Это позволяет вам работать с промисами синхронно, используя await и использовать другой синхронный код.

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

async function asyncAwaitTask() {
  const { version } = fs.readFileSync('package.json');
  console.log(version);
  await Promise.resolve('some result');
}

exports.default = asyncAwaitTask;
```

Работа с файлами

Методы `src()` и `dest()` используются для взаимодействия с файлами на вашем компьютере.

`src()` использует glob для чтения файловой системы и создания потока Node. Он находит все подходящие файлы и считывает их в память для прохождения через поток.

> Glob - Шаблон поиска с использованием метасимволов, например `**/*.js`.

Поток, созданный `src()`, должен быть возвращен из задачи, чтобы сигнализировать об асинхронном завершении, как указано в разделе Создание задач.

```
const { src, dest } = require('gulp');
```

```
2
3 exports.default = function() {
4   return src('src/*.js')
5     .pipe(dest('output/'));
6 }
```

Основным API потока является метод `.pipe()` для объединения потоков Transform или Writable.

```
1 const { src, dest } = require('gulp');
2 const babel = require('gulp-babel');
3
4 exports.default = function() {
5   return src('src/*.js')
6     .pipe(babel())
7     .pipe(dest('output/'));
8 }
```

`dest()` получает строку пути выходного каталога, а также создает поток Node. Когда метод получает файл, переданный через конвейер, он записывает содержимое и другие детали в файловую систему в данном каталоге. Метод `symlink()` также доступен и работает как `dest()`, но создает ссылки вместо файлов (подробности см. в symlink() API).

Чаще всего плагины размещаются между `src()` и `dest()` с помощью метода `.pipe()` и преобразуют файлы в потоке.

Добавление файлов в поток

`src()` также может быть размещен в середине конвейера для добавления файлов в поток на основе Globs (Шаблонов поиска). Дополнительные файлы будут доступны только для преобразований в потоке позже. Если Globs перекрываются, файлы будут добавлены снова.

> Два или более globs, которые намеренно или не намеренно совпадают с одним и тем же файлом, считаются перекрывающимися. Когда перекрывающиеся Globs используются внутри одного `src()`, gulp делает все возможное, чтобы удалить дубликаты.

Это может быть полезно для переноса некоторых файлов перед добавлением простых файлов JavaScript в конвейер или удалением всего.

```
1 const { src, dest } = require('gulp');
```

```
const babel = require('gulp-babel');
const uglify = require('gulp-uglify');

exports.default = function() {
  return src('src/*.js')
    .pipe(babel())
    .pipe(src('vendor/*.js'))
    .pipe(uglify())
    .pipe(dest('output/'));
}
```

Выход по фазам (поэтапный)

`dest()` может использоваться в середине конвейера для записи промежуточных состояний в файловую систему. Когда файл получен, текущее состояние записывается в файловую систему, путь обновляется для представления нового местоположения выходного файла, затем этот файл продолжает движение по конвейеру.

Эта функция может быть полезна, например, для создания минимизированных и неминимизированных файлов с одним и тем же конвейером.

```
const { src, dest } = require('gulp');
const babel = require('gulp-babel');
const uglify = require('gulp-uglify');
const rename = require('gulp-rename');

exports.default = function() {
  return src('src/*.js')
    .pipe(babel())
    .pipe(src('vendor/*.js'))
    .pipe(dest('output/'))
    .pipe(uglify())
    .pipe(rename({ extname: '.min.js' }))
    .pipe(dest('output/'));
}
```

Режимы: потоковый, буферизованный и пустой

`src()` может работать в трех режимах: буферизация, поток и пустой (empty). Это настраивается опциями `bufer` и `read` в `src()`.

- Режим буферизации используется по умолчанию и загружает содержимое файла в память. Плагины обычно работают в режиме буферизации. Многие из них не

поддерживают потоковый режим.

- Режим потоковой передачи предназначен в основном для работы с большими файлами, которые не помещаются в памяти, такими как гигантские изображения или фильмы. Содержимое передается из файловой системы небольшими порциями, а не загружается сразу. Если вам нужно использовать потоковый режим, найдите плагин, который его поддерживает или напишите свой собственный.
- Режим empty полезен при работе только с метаданными файла. Он ничего не содержит.

Шаблоны Globs

Глобирование (Globbing) - это поиск файлов в файловой системе с использованием одного или нескольких шаблонов поиска.

Метод `src()` принимает glob строку или массив для определения файлов, с которыми будет работать конвейер. Для вашего glob(s) должно быть найдено хотя бы одно совпадение, иначе `src()` выдаст ошибку. Когда используется массив globs, элементы обрабатываются по порядку, что особенно полезно для определения негативных globs.

Сегменты и разделители

Сегмент - это все, что находится между разделителями. Разделителем в glob всегда является символ `/`, независимо от операционной системы, даже в Windows, где разделителем пути является `\\`. В glob `\\` зарезервирован как символ экранирования.

Здесь `*` экранируется, поэтому он рассматривается как литерал, а не как символ шаблона.

```
'glob_with_uncommon_\\*_character.js'
```

Избегайте использования методов `path` Node, таких, как `path.join`, для создания globs. В Windows он создает недопустимый glob потому, что Node использует `\\` в качестве разделителя. Также избегайте использования `__dirname` global, `__filename` global или `process.cwd()` по тем же причинам.

```
const invalidGlob = path.join(__dirname, 'src/*.js');
```

Специальный символ: * (одиночная звезда)

Соответствует любому количеству (включая ни одного) символов в пределах одного сегмента. Полезно для выборки файлов в одном каталоге.

Этот glob будет соответствовать файлам вроде `index.js`, но не файлам в подкаталоге `scripts/index.js` или `scripts/nested/index.js`

```
'*.js'
```

Специальный символ: ** (двойная звезда)

Соответствует любому количеству (включая ни одного) символов в сегментах. Полезно для выборки файлов во вложенных каталогах. Убедитесь, что вы правильно ограничили двойные звездочки, чтобы избежать ненужного сопоставления больших каталогов.

Здесь glob ограничен каталогом `scripts/`. Будут выбраны все .js файлы во всех вложенных каталогах, такие, как `scripts/index.js`, `scripts/nested/index.js` и `scripts/nested/twice/index.js`.

```
'scripts/**/*.js'
```

> В данном примере, если бы не было префикса `scripts/`, все зависимости в node_modules или других каталогах также были бы включены в обработку.

Специальный символ: ! (негативный, отрицательный)

Если globs перечисляются в порядке массива, отрицательный glob должен следовать по крайней мере за одним неотрицательным glob в массиве. Первый находит набор совпадений, затем отрицательный glob удаляет часть этих результатов. При исключении всех файлов в подкаталоге необходимо добавить `/**` после имени каталога.

```
['scripts/**/*.js', '!scripts/vendor/**']
```

Если за отрицательным glob следуют неотрицательные globs, из более позднего набора совпадений ничего не будет удалено.

```
['scripts/**/*.js', '!scripts/vendor/**', 'scripts/vendor/react.js']
```

Отрицательные globs могут быть использованы в качестве альтернативы для ограничения globs с двойными звездами.

```
['**/*.js', '!node_modules/**']
```

> В данном примере, если бы отрицательный glob был с расширением .js `!node_modules/**/*.js`, globbing-библиотека не оптимизировала бы отрицание, и каждое совпадение пришлось бы сравнивать с отрицательным glob, который работал бы чрезвычайно медленно. Чтобы игнорировать все файлы в каталоге, лучше добавить только `/**` после имени каталога.

Перекрывающиеся globs

Два или более globs, которые намеренно или не намеренно совпадают с одним и тем же файлом, считаются перекрывающимися. Когда перекрывающиеся Globs используются внутри одного `src()`, gulp делает все возможное, чтобы удалить дубликаты.

Дополнительные ресурсы

Большая часть того, что вам нужно для работы с globs в gulp, описана в дополнительных ресурсах данного раздела. Если вы хотите получить более подробную информацию, вот несколько ресурсов:

- Документация Micromatch
- node-glob's Glob Primer
- Globbing - Базовая документация
- Страница Glob на Википедии

Использование плагинов

Плагины Gulp - это преобразующие Node потоки, которые инкапсулируют обычное поведение для преобразования файлов в конвейер. В большинстве случаев размещаются между `src()` и `dest()` с помощью метода `.pipe()`. Они могут изменить имя файла, метаданные или содержимое каждого файла, который проходит через поток.

Плагины из npm, с использованием ключевых слов "gulpplugin" и "gulpfriendly" можно просматривать и искать на странице поиска плагинов.

Каждый плагин обычно выполняет какую-то небольшую часть работы, их следует использовать как строительные блоки. Возможно, вам придется использовать несколько плагинов, чтобы получить желаемый результат.

```
const { src, dest } = require('gulp');
const uglify = require('gulp-uglify');
const rename = require('gulp-rename');

exports.default = function() {
  return src('src/*.js')
    // Плагин gulp-uglify не будет обновлять имя файла
    .pipe(uglify())
    // Поэтому используйте gulp-rename, чтобы изменить его
    .pipe(rename({ extname: '.min.js' }))
    .pipe(dest('output/'));
}
```

Нужен ли вам плагин?

Не всегда в gulp нужно использовать плагины. Это, конечно достаточно быстрый способ начать работу, однако многие операции выглядят лучше благодаря использованию модуля или библиотеки.

```
const { rollup } = require('rollup');

// Rollup's промис отлично работает в асинхронных задачах
exports.default = async function() {
  const bundle = await rollup.rollup({
    input: 'src/index.js'
  });

  return bundle.write({
    file: 'output/bundle.js',
    format: 'iife'
  });
}
```

Плагины всегда преобразовывают файлы. Используйте (не подключаемый) Node-модуль или библиотеку для любых других операций.

```
const del = require('delete');

exports.default = function(cb) {
  // Используйте модуль `delete` напрямую, вместо использования плагина gulp-rm
  del(['output/*.js'], cb);
}
```

Условные плагины

Так как операции с плагином не учитывают тип файла, вам может понадобиться плагин типа gulp-if для преобразования подмножеств файлов.

```
const { src, dest } = require('gulp');
const gulpif = require('gulp-if');
const uglify = require('gulp-uglify');

function isJavaScript(file) {
  // Проверка, является ли расширение файла '.js'
  return file.extname === '.js';
}

exports.default = function() {
  // Включить файлы JavaScript и CSS в один конвейер
  return src(['src/*.js', 'src/*.css'])
    // Применить плагин gulp-uglify только к файлам JavaScript
    .pipe(gulpif(isJavaScript, uglify()))
    .pipe(dest('output/'));
}
```

Встроенные плагины

Встроенные плагины - это одноразовые Transform потоки, которые вы определяете внутри своего gulpfile, записывая желаемое поведение.

Есть две ситуации, в которых полезно создание встроенного плагина:

- Вместо того, чтобы создавать и поддерживать свой собственный плагин.
- Вместо того, чтобы создавать плагин, предназначенный для того, чтобы добавить желаемую функцию.

```
const { src, dest } = require('gulp');
const uglify = require('uglify-js');
const through2 = require('through2');

exports.default = function() {
  return src('src/*.js')
    // Вместо использования gulp-uglify вы можете создать встроенный плагин
    .pipe(through2.obj(function(file, _, cb) {
      if (file.isBuffer()) {
        const code = uglify.minify(file.contents.toString())
        file.contents = Buffer.from(code)
      }
      cb(null, file);
    }))
    .pipe(dest('output/'));
}
```

Наблюдение за файлами

API `watch()` связывает globs с тасками используя вотчер файловой системы. Он отслеживает изменения в файлах, соответствующих шаблону или шаблонам Glob и выполняет задачу, если происходит изменение. Если задача не сигнализирует об асинхронном выполнении, она никогда не будет запущена второй раз.

Этот API-интерфейс обеспечивает встроенную задержку и организацию очереди на основе наиболее распространенных значений по умолчанию.

```
const { watch, series } = require('gulp');

function clean(cb) {
  // тело функции
  cb();
}

function javascript(cb) {
  // тело функции
  cb();
}

function css(cb) {
  // тело функции
  cb();
}
```

```
17
18 exports.default = function() {
19   // Вы можете использовать одну задачу
20   watch('src/*.css', css);
21   // Или композицию тасков
22   watch('src/*.js', series(clean, javascript));
23 };
```

Предупреждение: избегайте синхронности

Таск вотчера не может быть синхронным. Если вы создаете синхронную задачу, выполнение не будет корректным, задача не запустится снова, так как предполагается, что она все еще выполняется.

Вы не получите сообщения об ошибке или предупреждения, потому что средство отслеживания файлов поддерживает ваш процесс Node. Поскольку процесс не завершается, невозможно определить, выполнена ли задача или для ее выполнения просто требуется много времени.

Наблюдаемые события

По умолчанию вотчер выполняет задачи всякий раз, когда файл создается, изменяется или удаляется. Если вам нужно использовать разные события, вы можете использовать опцию `events` при вызове `watch()`. Доступны следующие события: `'add'`, `'addDir'`, `'change'`, `'unlink'`, `'unlinkDir'`, `'ready'`, `'error'`. Дополнительно доступно `'all'`, которое представляет все события, кроме `'ready'` и `'error'`.

```
1 const { watch } = require('gulp');
2
3 exports.default = function() {
4   // Все события будут отслеживаться
5   watch('src/*.js', { events: 'all' }, function(cb) {
6     // тело функции
7     cb();
8   });
9 };
```

Начало исполнения

После вызова `watch()` задачи не будут выполнены, вместо этого они будут ждать первого изменения файла.

Чтобы выполнить задачи перед первым изменением файла, установите для параметра `ignoreInitial` значение `false`.

```
const { watch } = require('gulp');

exports.default = function() {
  // Задача будет выполнена при запуске
  watch('src/*.js', { ignoreInitial: false }, function(cb) {
    // тело функции
    cb();
  });
};
```

Очередь

Каждый `watch()` гарантирует, что выполняемая в данный момент задача не будет выполнена снова. Если изменение файла происходит во время выполнения задачи вотчера, другое выполнение будет поставлено в очередь и запущено после завершения задачи. Только один запуск может быть поставлен в очередь за раз.

Чтобы отключить очередь, установите для параметра `queue` значение `false`.

```
const { watch } = require('gulp');

exports.default = function() {
  // Задача будет выполняться (одновременно) для каждого внесенного изменения
  watch('src/*.js', { queue: false }, function(cb) {
    // тело функции
    cb();
  });
};
```

Задержка

После изменения файла задача наблюдателя не будет запущена, пока не истечет задержка в 200 мс. Это сделано для того, чтобы избежать слишком раннего запуска задачи, когда множество файлов изменяется одновременно. Например, поиск и замена.

Чтобы настроить длительность задержки, установите для параметра задержки положительное целое число.

```
const { watch } = require('gulp');

```

```
exports.default = function() {
  // Задача не будет запущена, пока не пройдет 500 мс. с момента первого измене
  watch('src/*.js', { delay: 500 }, function(cb) {
    // тело функции
    cb();
  });
};
```

Использование экземпляра вотчера

Скорее всего, вы не будете использовать эту функцию. Если вам нужен полный контроль над измененными файлами - например, доступ к путям или метаданным, используйте экземпляр chokidar, возвращенный из `watch()`.

Будьте внимательны: возвращенный экземпляр chokidar не имеет функций очереди, задержки или асинхронного завершения.

Необязательная зависимость

Gulp имеет необязательную зависимость, называемую fsevents, которая является средством вотчинга файлов на Mac. Если вы видите предупреждение об установке для fsevents - «npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents» - это не проблема. Если установка fsevents пропущена, будет использован резервный вотчер и любые ошибки, возникающие в вашем gulpfile, не будут связаны с этим предупреждением.

Концепции API

Следующие понятия являются базовыми для понимания документации API.

Vinyl

Vinyl - это объект метаданных, который описывает файл. Основными свойствами экземпляра Vinyl являются `path` и `contents` - основные аспекты файла в вашей файловой системе. Объекты Vinyl могут использоваться для описания файлов из многих источников - в локальной файловой системе или в любом другом удаленном хранилище.

Адаптеры Vinyl

Хотя Vinyl предоставляет способ описания файла, необходим способ доступа к этим файлам. Доступ к каждому источнику файлов осуществляется с помощью адаптера Vinyl.

Адаптер предоставляет:

- Метод с ключом `src(globs, [options])` и возвращает поток, который создает объекты Vinyl.
- Метод с ключом `dest(папка, [options])` и возвращает поток, который использует объекты Vinyl.
- Любые дополнительные методы, специфичные для их среды ввода/вывода, такие как метод `symlink`, обеспечивающий vinyl-fs. Они всегда должны возвращать потоки, которые производят и/или потребляют объекты Vinyl.

Задачи (Tasks)

Каждая задача gulp представляет собой асинхронную функцию JavaScript, которая либо принимает колбек с ошибкой, либо возвращает поток, промис, источник событий, дочерний процесс или наблюдаемый тип. Из-за некоторых ограничений платформы синхронные задачи не поддерживаются.

Для более подробного объяснения см. Создание задач.

Globs

Glob - это строка литеральных и/или шаблонных символов, таких как `*`, `**` или `!`, используемая для определения путей к файлам. Globbing - это поиск файлов в файловой системе с использованием одного или нескольких шаблонов поиска.

Если у вас нет опыта работы с globs, см. Шаблоны Globs.

Основа Glob

Основа glob - это сегмент пути перед любыми специальными символами в строке glob. Таким образом, основа glob `/src/js/**.js` - это `/src/js/`. Все пути, которые соответствуют glob, гарантированно используют основу glob - этот сегмент пути не может быть переменным.

Экземпляры Vinyl, сгенерированные `src()`, создаются с базовым свойством `base`. При записи в файловую систему с помощью `dest()`, `base` будет удален из выходного пути для сохранения структуры каталогов.

Для получения более подробной информации см. репозиторий glob-parent.

Статистика файловой системы

Файловые метаданные предоставляются в качестве экземпляра Node `fs.Stats`. Он доступен как свойство `stat` в ваших экземплярах Vinyl и используется внутри, чтобы определить, представляет ли объект Vinyl каталог или символическую ссылку. При записи в файловую систему разрешения и значения времени синхронизируются из свойства `stat` объекта Vinyl.

Режимы файловой системы

Режимы файловой системы определяют, какие разрешения существуют для файла. Большинство файлов и каталогов в вашей файловой системе будут иметь доступный режим, позволяющий gulp читать/записывать/обновлять файлы от вашего имени. По умолчанию gulp создает файлы с теми же разрешениями, что и у запущенного процесса, но вы можете настроить режимы с помощью параметров в `src()`, `dest()` и т. д. Если у вас возникают проблемы с разрешениями (EPERM), проверьте текущие разрешения ваших файлов.

Модули

Gulp состоит из множества небольших модулей, которые объединены для совместной работы. Используя semver в небольших модулях, мы можем выпускать исправления ошибок и функции без публикации новых версий gulp. Часто, когда вы не видите прогресса в основном хранилище, работа выполняется в одном из этих модулей.

Если у вас возникли проблемы, убедитесь, что ваши текущие модули обновлены с помощью команды `npm update`. Если проблема не устранена, изучите проблему в репозитории отдельных проектов.

- undertaker - система регистрации задач
- vinyl - виртуальные файловые объекты
- vinyl-fs - адаптер для вашей локальной файловой системы
- glob-watcher - вотчер файлов
- bach - оркестровка задач с использованием `series()` и `parallel()`
- last-run - отслеживает время последнего выполнения задачи
- vinyl-sourcemap - встроенная поддержка sourcemap
- gulp-cli - интерфейс командной строки для взаимодействия с gulp

src()

Создает поток для чтения объектов Vinyl из файловой системы.

Примечание. BOMs (метки порядка байтов) не имеют смысла в UTF-8 и будут удалены из файлов UTF-8, считываемых с помощью `src()`, если только это не отключено с помощью опции `removeBOM`.

Использование

```
const { src, dest } = require('gulp');

function copy() {
  return src('input/*.js')
    .pipe(dest('output/'));
}

exports.copy = copy;
```

Ключ

```
src(globs, [options])
```

Параметры

Параметр	Тип	Примечание
globs	строка, массив	Globs для просмотра в файловой системе.
option	объект	Подробнее в опциях ниже.

Возвращает

Поток, который можно использовать в начале или в середине конвейера для добавления файлов на основе заданных globs.

Ошибки

Если аргумент `globs` соответствует только одному файлу (например, `foo/bar.js`) и совпадение не найдено, выдается сообщение об ошибке «File not found with singular glob». Чтобы исправить эту ошибку, установите для параметра `allowEmpty` значение `true`.

Когда в `globs` указан недопустимый glob, выдается сообщение об ошибке «Invalid glob argument».

Опции

Для опций, которые принимают функцию, переданная функция будет вызываться с каждым объектом Vinyl и должна возвращать значение другого из перечисленных типов.

Опция	Тип	Дефолт.	Примечание
buffer	boolean, function	true	Если true, содержимое файла буферизуется в памяти. Если false, свойство `contents` объекта Vinyl будет приостановлено потоком. Опция може не позволить буферизовать содержимое больших файлов. Примечание: в плагинах может быть не реализована поддержка потокового содержимого.
read	boolean, function	true	Если false, файлы не будут читаться, а их объекты Vinyl не будут записываться на диск через `.dest()`.
since	date, timestamp, function		Если установлено, создаются объекты Vinyl только для файлов, измененных с указанного времени.
removeBOM	boolean, function	true	Если true, BOM из файлов в кодировке UTF-8 будет удален. Если false, BOM игнорируется.
sourcemaps	boolean, function	false	Если true, поддержка sourcemaps на созданных объектах Vinyl будет включена. Загружает внутренние sourcemaps и разрешает внешние ссылки на sourcemaps.
resolveSymlinks	boolean, function	true	При значении true рекурсивно разрешаются символические ссылки. Если false, символические ссылки сохраняются и устанавливается свойство `symlink` объекта Vinyl к пути файла.

Опция	Тип	Дефолт.	Примечание
cwd	string	process.cwd()	Директория, которая должна быть объединена с любым относительным путем, чтобы сформировать абсолютный путь. Игнорируется для абсолютных путей. Используйте, чтобы избежать объединения `globs` с `path.join()`. *Эта опция передается напрямую в glob-stream.*
base	string		Явно устанавливает свойство `base` для созданных объектов Vinyl. Подробно в API Concepts. Эта опция передается напрямую в glob-stream.
cwdbase	boolean	false	Если true, опции `cwd` и `base` должны быть выровнены. Эта опция передается напрямую в glob-stream.
root	string		Корневой путь, для которого разрешены globs. Эта опция передается напрямую в glob-stream.
allowEmpty	boolean	false	Если false, `globs`, которые соответствуют только одному файлу (например, `foo/bar.js`), вызывают ошибку, если они не находят соответствия. Если true, соответствующие ошибки globs игнорируются. Эта опция передается напрямую в glob-stream.
uniqueBy	string, function	'path'	Удаляет дубликаты из потока, сравнивая имя свойства строки или результат функции. Примечание. При использовании функции, функция получает потоковые данные (объекты, содержащие свойства `cwd`, `base`, `path`).
dot	boolean	false	Если true, сравнивает globs с .dot файлами, например .gitignore. Эта опция передается напрямую в node-glob.
silent	boolean	true	Если true, подавляет предупреждения от `stderr`. Примечание. Эта опция передается напрямую в node-glob, но по умолчанию она имеет значение true, а не false.
mark	boolean	false	Если true, символ `/` будет добавлен к совпадениям директории. Обычно не требуется, потому что пути нормализуются внутри конвейера. Эта опция передается напрямую в node-glob.

Опция	Тип	Дефолт.	Примечание
nosort	boolean	false	Если true, отключает сортировку результатов globs. Эта опция передается напрямую в node-glob.
stat	boolean	false	Если true, `fs.stat()` вызывается для всех результатов. Это добавляет дополнительную нагрузку и, как правило, не должно использоваться. Эта опция передается напрямую в node-glob.
strict	boolean	false	Если true, будет выдано сообщение об ошибке, если при попытке чтения каталога возникнет непредвиденная проблема. Эта опция передается напрямую в node-glob.
nounique	boolean	false	При false предотвращает дублирование файлов в наборе результатов. Эта опция передается напрямую в node-glob.
debug	boolean	false	Если true, отладочная информация будет отражена в командной строке. Эта опция передается напрямую в node-glob.
nobrace	boolean	false	Если true, избегаются наборы скобок - например, {a, b} или {1..3}. Эта опция передается напрямую в node-glob.
noglobstar	boolean	false	Если true, обрабатывает двухзвездный glob как однозвездный. Эта опция передается напрямую в node-glob.
noext	boolean	false	Если true, игнорируются сопоставления шаблонов extglob - например, `+(Ab)`. Эта опция передается напрямую в node-glob.
nocase	boolean	false	Если true, выполняет сопоставление без учета регистра. Примечание. В файловых системах без учета регистра шаблоны, не относящиеся к magic patterns, будут совпадать по умолчанию. Эта опция передается напрямую в node-glob.

Опция	Тип	Дефолт.	Примечание
matchBase	boolean	false	Если true и globs не содержат символов `/`, перебираются все каталоги и совпадения с этим glob - например, `*.js` будет рассматриваться как эквивалент `**/*.js`. Эта опция передается напрямую в node-glob.
nodir	boolean	false	Если true, обрабатываются только соответствия файлам, а не каталогам. Примечание: чтобы обрабатывать только соответствия каталогам, завершите ваш glob знаком `/`. Эта опция передается напрямую в node-glob.
ignore	string, array		Исключение globs из совпадений. Эта опция сочетается с отрицательными `globs`. Примечание. Эти globs всегда сопоставляются с файлами .dot, независимо от других настроек. Эта опция передается напрямую в node-glob.
follow	boolean	false	Если true, каталоги с символьными ссылками будут перемещаться при расширении `**` globs. Примечание. Это может вызвать проблемы с циклическими ссылками. Эта опция передается напрямую в node-glob.
realpath	boolean	false	Если true, `fs.realpath()` вызывается для всех результатов. Это может привести к висячим указателям. Эта опция передается напрямую в node-glob.
cache	object		Ранее сгенерированный объект кэша - игнорирует некоторые вызовы файловой системы. Эта опция передается напрямую в node-glob.
statCache	object		Ранее сгенерированный кеш результатов `fs.Stat` - игнорирует некоторые вызовы файловой системы. Эта опция передается напрямую в node-glob.
symlinks	object		Ранее сгенерированный кеш символических ссылок - игнорирует некоторые вызововы файловой системы. Эта опция передается напрямую в node-glob.

Опция	Тип	Дефолт.	Примечание
nocomment	boolean	false	Если false, символ `#` в начале glob трактуется как комментарий. Эта опция передается напрямую в node-glob.

Sourcemaps

Поддержка Sourcemap встроена непосредственно в `src()` и `dest()`, но по умолчанию отключена. Включите поддержку для создания внутренних или внешних sourcemaps.

Внутренние sourcemaps:

```
const { src, dest } = require('gulp');
const uglify = require('gulp-uglify');

src('input/**/*.js', { sourcemaps: true })
  .pipe(uglify())
  .pipe(dest('output/', { sourcemaps: true }));
```

External sourcemaps:

```
const { src, dest } = require('gulp');
const uglify = require('gulp-uglify');

src('input/**/*.js', { sourcemaps: true })
  .pipe(uglify())
  .pipe(dest('output/', { sourcemaps: '.' }));
```

---

dest()

Создает поток для записи объектов Vinyl в файловую систему.

Использование

```
const { src, dest } = require('gulp');

function copy() {
  return src('input/*.js')
    .pipe(dest('output/'));
```

```
6 }
7
8 exports.copy = copy;
```

Ключ

```
dest(directory, [options])
```

Параметры

Параметр	Тип	Примечание
directory (обязательный)	string, function	Путь к выходному каталогу, в который будут записываться файлы. Если используется функция, функция будет вызываться с каждым объектом Vinyl и должна возвращать строку пути к каталогу.
options	object	Подробнее в опциях ниже.

Возвращает

Поток, который можно использовать в середине или в конце конвейера для создания файлов в файловой системе. Всякий раз, когда объект Vinyl передается через поток, он записывает содержимое и другие детали в файловую систему в указанном каталоге. Если объект Vinyl обладает свойством `symlink`, вместо записи содержимого будет создана символическая ссылка. После создания файла его метаданные будут обновлены, чтобы соответствовать объекту Vinyl.

Всякий раз, когда файл создается в файловой системе, объект Vinyl будет изменен.

- Свойства `cwd`, `base` и `path` будут обновлены в соответствии с созданным файлом.
- Свойство `stat` будет обновлено в соответствии с файлом в файловой системе.
- Если свойство `contents` является потоком, оно будет сброшено для того, чтобы его можно было снова прочитать.

Ошибки

Если `directory` представляет собой пустую строку, выдается сообщение об ошибке «Invalid dest() folder argument. Please specify a non-empty string or a function.».

Если `directory` не является строкой или функцией, выдается сообщение об ошибке «Invalid dest() folder argument. Please specify a non-empty string or a function.».

Если `directory` является функцией, которая возвращает пустую строку или не определена, выдается ошибка с сообщением «Invalid output folder».

Опции

Для опций, которые принимают функцию, переданная функция будет вызываться с каждым объектом Vinyl и должна возвращать значение другого из перечисленных типов.

Опция	Тип	Дефолт.	Примечание
cwd	string, function	`process.cwd()`	Каталог будет объединен с любым относительным путем для того, чтобы сформировать абсолютный путь. Игнорируется для абсолютных путей. Используйте, чтобы избежать объединения `directory` с `path.join()`.
mode	number, function	`stat.mode` объекта Vinyl	Режим, используемый при создании файлов. Если не установлено и `stat.mode` отсутствует, вместо него будет использован режим процесса.
dirMode	number, function		Режим, используемый при создании каталогов. Если не установлен, будет использован режим процесса.
overwrite	boolean, function	true	Если true, перезаписывает существующие файлы с тем же путем.
append	boolean, function	false	Если true, добавляет содержимое в конец файла вместо замены существующего содержимого.
sourcemaps	boolean, string, function	false	Если true, записывает внутренние sourcemaps в выходной файл. При указании пути к строке будут записаны внешние sourcemaps по заданному пути.
relativeSymlinks	boolean, function	false	Если false, любые созданные символические ссылки будут абсолютными. Примечание. Игнорируется, если создается соединение, поскольку оно должно быть абсолютным.

Опция	Тип	Дефолт.	Примечание
useJunctions	boolean, function	true	Эта опция актуальна только в Windows и игнорируется в других местах. При значении true создается символьная ссылка на каталог как соединение. Подробнее в Символических ссылках на Windows ниже.

Обновления метаданных

Когда поток `dest()` создает файл, объекты Vinyl `mode` `mtime` и `atime` сравниваются с созданным файлом. Если они различаются, созданный файл будет обновлен в соответствии с метаданными объекта Vinyl. Если эти свойства одинаковы или gulp не имеет прав для внесения изменений, попытка пропускается без уведомления.

Эта функциональность отключена в Windows или других операционных системах, которые не поддерживают методы Node's `process.getuid()` или `process.geteuid()`. Это связано с тем, что Windows дает неожиданные результаты при использовании `fs.fchmod()` и `fs.futimes()`.

Примечание. Метод `fs.futimes()` внутренне преобразует метки времени `mtime` и `atime` в секунды. Это деление на 1000 может привести к некоторой потере точности в 32-битных операционных системах.

Sourcemaps

Поддержка sourcemaps встроена непосредственно в `src()` и `dest()`, но по умолчанию она отключена. Включите ее для создания внутренних или внешних sourcemaps.

Внутренние sourcemaps:

```
const { src, dest } = require('gulp');
const uglify = require('gulp-uglify');

src('input/**/*.js', { sourcemaps: true })
  .pipe(uglify())
  .pipe(dest('output/', { sourcemaps: true }));
```

Внешние sourcemaps:

```
const { src, dest } = require('gulp');
const uglify = require('gulp-uglify');
```

```
3
4 src('input/**/*.js', { sourcemaps: true })
5   .pipe(uglify())
6   .pipe(dest('output/', { sourcemaps: '.' }));
```

Символические ссылки в Windows

При создании символических ссылок в Windows аргумент `type` передается методу Node's `fs.symlink()`, который указывает тип цели. Тип ссылки установлен на:

- `'file'`, когда целью является обычный файл
- `'junction'`, когда целью является каталог
- `'dir'`, когда целью является каталог и пользователь отключает опцию useJunctions

Если вы попытаетесь создать висячий (указывающий на несуществующую цель) указатель, тип указателя не может быть определен автоматически. В таких случаях поведение будет зависеть от того, создается ли такой указатель с помощью `symlink()` или `dest()`.

Для висячих указателей, созданных с помощью `symlink()`, входящий объект Vinyl представляет цель, поэтому его статистика будет определять желаемый тип указателя. Если `isDirectory()` возвращает false, то создается указатель `'file'`, в противном случае создается указатель `'junction'` или `'dir'`, в зависимости от параметра `useJunctions`.

Для висячих указателей, созданных с помощью `dest()`, входящий объект Vinyl представляет указатель, обычно загружаемый с диска через `src(..., {resolSymlinks: false})`. В этом случае тип указателя не может быть адекватно определен и по умолчанию используется «файл». Это может вызвать неожиданное поведение, если вы создаете висячий указатель на каталог. Избегайте этого сценария.

---

symlink()

Создает поток для связи объектов Vinyl с файловой системой.

Использование

```
1 const { src, symlink } = require('gulp');
2
3 function link() {
```

```
  return src('input/*.js')
    .pipe(symlink('output/'));
}

exports.link = link;
```

Ключ

```
symlink(directory, [options])
```

Параметры

Параметр	Тип	Примечание
directory (обязательный)	string, function	Путь к выходному каталогу, в котором будут созданы символические ссылки. Если используется функция, функция будет вызываться с каждым объектом Vinyl и должна возвращать строку пути к каталогу.
options	object	Подробно в опциях ниже.

Возвращает

Поток, который можно использовать в середине или в конце конвейера для создания символических ссылок в файловой системе. Всякий раз, когда объект Vinyl пропускается через поток, он создает символическую ссылку на исходный файл в файловой системе в текущем каталоге.

Всякий раз, когда в файловой системе создается символическая ссылка, объект Vinyl будет изменен.

- Свойства `cwd`, `base` и `path` будут обновлены в соответствии с созданной символической ссылкой.
- Свойство `stat` будет обновлено, чтобы соответствовать символической ссылке в файловой системе.
- Свойство `contents` будет установлено как `null`.
- Свойство `symlink` будет добавлено или заменено на исходный путь.

Примечание. В Windows ссылки на каталоги создаются с использованием соединений по умолчанию. Опция `useJunctions` отключает это поведение.

Ошибки

Если `directory` является пустой строкой, вы получите сообщение об ошибке «Invalid symlink() folder argument. Please specify a non-empty string or a function.».

Если `directory` не является строкой или функцией, выдается сообщение об ошибке «Invalid symlink() folder argument. Please specify a non-empty string or a function.».

Когда `directory` является функцией, которая возвращает пустую строку или `undefined`, выдает ошибку с сообщением «Invalid output folder».

Опции

Для опций, которые принимают функцию, переданная функция будет вызываться с каждым объектом Vinyl и должна возвращать значение другого из перечисленных типов.

Опция	Тип	Дефолт.	Примечание
cwd	string, function	`process.cwd()`	Каталог будет объединен с любым относительным путем для того, чтобы сформировать абсолютный путь. Игнорируется для абсолютных путей. Используйте, чтобы избежать объединения `directory` с `path.join()`.
dirMode	number, function		Режим, используемый при создании каталогов. Если не установлен, будет использован режим процесса.
overwrite	boolean, function	true	При значении true перезаписывает существующие файлы с тем же путем.
relativeSymlinks	boolean, function	false	Если false, любые созданные символические ссылки будут абсолютными. Примечание. Игнорируется, если создается соединение, поскольку оно должно быть абсолютным.

Опция	Тип	Дефолт.	Примечание
useJunctions	boolean, function	true	Эта опция актуальна только в Windows и игнорируется в других ОС. При значении true создает символьную ссылку на каталог как соединение. Подробнее в «Символических ссылках на Windows» ниже.

Символические ссылки в Windows

При создании символических ссылок в Windows аргумент `type` передается методу Node's `fs.symlink()`, который указывает тип цели. Тип ссылки установлен на:

- `'file'`, если целью является обычный файл
- `'junction'`, когда целью является каталог
- `'dir'`, если целью является каталог и пользователь отключает опцию useJunctions

Если вы попытаетесь создать висячую (указывающую на несуществующую цель) ссылку (или висячий указатель), тип ссылки не может быть определен автоматически. В этих случаях поведение будет зависеть от того, создается ли висячая ссылка с помощью `symlink()` или `dest()`.

Для висячих ссылок (указателей), созданных с помощью `symlink()`, входящий объект Vinyl представляет цель, поэтому его статистика будет определять желаемый тип ссылки. Если `isDirectory()` возвращает false, то создается ссылка `'file'`, в противном случае создается ссылка `'junction'` или `'dir'`, в зависимости от значения опции `useJunctions`.

Для висячих ссылок (указателей), созданных с помощью `dest()`, входящий объект Vinyl представляет ссылку, обычно загружаемую с диска через `src(..., {resolSymlinks: false})`. В таком случае тип ссылки не может быть адекватно определен и по умолчанию используется `'file'`. Это может вызвать неожиданное поведение при создании висячей ссылки на каталог. Избегайте этого сценария.

---

lastRun()

Получает отметку времени последнего раза, когда задача была успешно завершена во время текущего рабочего процесса. Наиболее полезно при выполнении последующих задач

во время работы вотчера.

В сочетании с `src()` включает инкрементные сборки для ускорения выполнения, пропуская файлы, которые не изменились с момента последнего успешного завершения задачи.

Использование

```
const { src, dest, lastRun, watch } = require('gulp');
const imagemin = require('gulp-imagemin');

function images() {
  return src('src/images/**/*.jpg', { since: lastRun(images) })
    .pipe(imagemin())
    .pipe(dest('build/img/'));
}

exports.default = function() {
  watch('src/images/**/*.jpg', images);
};
```

Ключ

```
lastRun(task, [precision])
```

Параметры

Параметр	Тип	Примечание
task (обязательный)	function string	Функция таска или псевдоним строки зарегистрированной задачи.
precision	number	По умолчанию: 1000 на Node v0.10 и 0 на Node v0.12+. Подробно в разделе «Точность временной метки» ниже.

Возвращает

Отметку времени (в миллисекундах), соответствующую времени последнего выполнения задачи. Если задача не была выполнена или потерпела неудачу, возвращает `undefined`.

Чтобы избежать кеширования недопустимого состояния, возвращаемое значение должно быть `undefined`, если задача будет выполнена с ошибкой.

Ошибки

При вызове со значением, отличным от строки или функции, выдается ошибка с сообщением «Only functions can check lastRun».

Если вызывается для нерасширяемой функции, а в Node отсутствует WeakMap, выдается ошибка с сообщением: «Only extensible functions can check lastRun».

Точность временной метки

Хотя существуют точные значения по умолчанию для точности меток времени, они могут быть округлены с использованием параметра точности. Полезно, если ваша файловая система или версия Node имеют точность с потерями в атрибутах времени файла.

- `lastRun(someTask)` вернет 1426000001111
- `lastRun(someTask, 100)` вернет 1426000001100
- `lastRun(someTask, 1000)` вернет 1426000001000

Точность `mtime stat` файла может варьироваться в зависимости от версии Node и/или используемой файловой системы.

Платформа	Точность
Node v0.10	1000ms
Node v0.12+	1ms
Файловая система FAT32	2000ms
Файловая система HFS+ или Ext3	1000ms
NTFS с использованием Node v0.10	1s
NTFS с использованием Node 0.12+	100ms
Ext4 с использованием Node v0.10	1000ms
Ext4 с использованием Node 0.12+	1ms

series()

Объединяет функции задач и/или составные операции в более крупные, которые выполняются последовательно. Нет никаких ограничений на глубину вложенности составных операций с использованием `series()` и `parallel()`.

Использование

```
const { series } = require('gulp');

function javascript(cb) {
  // тело функции
  cb();
}

function css(cb) {
  // тело функции
  cb();
}

exports.build = series(javascript, css);
```

Ключ

```
series(...tasks)
```

Параметры

Параметр	Тип	Примечание
tasks (обязательный)	function string	В качестве отдельных аргументов может быть передано любое количество функций задач. Строки можно использовать, если вы уже зарегистрировали задачи, но так делать не рекомендуется.

Возвращает

Составную операцию, которая должна быть зарегистрирована как задача или вложена в другие композиции `series` и/или `parallel`.

Когда выполняется составная операция, все задачи (таски) будут выполняться последовательно. Если ошибка возникает в одной задаче, никакие последующие задачи запущены не будут.

Ошибки

Если никакие задачи не передаются, выдается ошибка с сообщением «One or more tasks should be combined using series or parallel».

Если передаются недопустимые или незарегистрированные задачи, выдается сообщение об ошибке «Task never defined».

Прямые ссылки

Прямая ссылка - это когда вы составляете задачи, используя строковые ссылки, которые еще не зарегистрированы. Это было обычной практикой в старых версиях, но данная функция была удалена для ускорения выполнения задач.

В более новых версиях вы получите сообщение об ошибке «Task never defined», если попытаетесь использовать прямые ссылки. Это может возникнуть при попытке использовать `exports` для регистрации задач и составления задач по строке. В данной ситуации используйте именованные функции вместо строковых ссылок.

Во время миграции вам может понадобиться использовать реестр прямых ссылок. Однако это значительно замедлит вашу сборку. Данный фикс не будет работать всегда, это лишь временное решение.

Избегайте дублирования задач

При выполнении составной операции каждая задача будет выполняться каждый раз, когда до нее доходит дело.

Таск `clean`, указанный в двух разных композициях, будет выполнен дважды и приведет к нежелательным результатам. Вместо этого проведите рефакторинг таска `clean`, который будет указан в окончательном составе.

Если у вас есть такой код:

```
1  // Пример НЕКОРРЕКТНЫЙ
2
3  const { series, parallel } = require('gulp');
4
5  const clean = function(cb) {
6    // тело функции
```

```
  cb();
};

const css = series(clean, function(cb) {
  // тело функции
  cb();
});

const javascript = series(clean, function(cb) {
  // тело функции
  cb();
});

exports.build = parallel(css, javascript);
```

Лучше сделать так:

```
const { series, parallel } = require('gulp');

function clean(cb) {
  // тело функции
  cb();
}

function css(cb) {
  // тело функции
  cb();
}

function javascript(cb) {
  // тело функции
  cb();
}

exports.build = series(clean, parallel(css, javascript));
```

---

parallel()

Объединяет функции задач и/или составные операции в более крупные, которые выполняются одновременно (параллельно). Нет никаких ограничений на глубину вложенности составных операций с использованием `series()` и `parallel()`.

Использование

```
const { parallel } = require('gulp');

function javascript(cb) {
  // тело функции
  cb();
}

function css(cb) {
  // тело функции
  cb();
}

exports.build = parallel(javascript, css);
```

Ключ

```
parallel(...tasks)
```

Параметры

Параметр	Тип	Примечание
tasks (обязательный)	function string	В качестве отдельных аргументов может быть передано любое количество функций задач. Строки можно использовать, если вы уже зарегистрировали задачи, но так делать не рекомендуется.

Возвращает

Составную операцию, которая должна быть зарегистрирована как задача или вложена в другие композиции `series` и/или `parallel`.

Когда выполняется составная операция, все задачи (таски) будут выполняться параллельно. Если ошибка возникает в одной задаче, другие задачи могут завершиться недетерминированно или не завершиться.

Ошибки

Когда никакие задачи не передаются, выдается ошибка с сообщением «One or more tasks should be combined using series or parallel».

Когда передаются недопустимые или незарегистрированные задачи, выдается сообщение об ошибке «Task never defined».

Прямые ссылки

Прямая ссылка - это когда вы составляете задачи, используя строковые ссылки, которые еще не зарегистрированы. Это было обычной практикой в старых версиях, но данная функция была удалена для ускорения выполнения задач.

В более новых версиях вы получите сообщение об ошибке «Task never defined», если попытаетесь использовать прямые ссылки. Это может возникнуть при попытке использовать `exports` для регистрации задач и составления задач по строке. В данной ситуации используйте именованные функции вместо строковых ссылок.

Во время миграции вам может понадобиться использовать реестр прямых ссылок. Однако это значительно замедлит вашу сборку. Данный фикс не будет работать всегда, это лишь временное решение.

Избегайте дублирования задач

При выполнении составной операции каждая задача будет выполняться каждый раз, когда до нее доходит дело.

Таск `clean`, указанный в двух разных композициях, будет выполнен дважды и приведет к нежелательным результатам. Вместо этого проведите рефакторинг таска `clean`, который будет указан в окончательном составе.

Если у вас есть такой код:

```
// Пример НЕКОРРЕКТНЫЙ

const { series, parallel } = require('gulp');

const clean = function(cb) {
  // тело функции
  cb();
};

const css = series(clean, function(cb) {
  // тело функции
  cb();
});

const javascript = series(clean, function(cb) {
```

```
  // тело функции
  cb();
});

exports.build = parallel(css, javascript);
```

Лучше сделать так:

```
const { series, parallel } = require('gulp');

function clean(cb) {
  // тело функции
  cb();
}

function css(cb) {
  // тело функции
  cb();
}

function javascript(cb) {
  // тело функции
  cb();
}

exports.build = series(clean, parallel(css, javascript));
```

---

watch()

Позволяет наблюдать за globs и запускать задачи при изменении. Задачи обрабатываются как обычно с остальной частью системы задач.

Использование

```
const { watch } = require('gulp');

watch(['input/*.js', '!input/something.js'], function(cb) {
  // тело функции
  cb();
});
```

Ключ

```
watch(globs, [options], [task])
```

Параметры

Параметр	Тип	Примечание
globs (обязательный)	string array	Globs для вотчинга в файловой системе.
options	object	Подробно в опциях ниже.
task	function string	Функция задачи или составная задача - генерируется с помощью `series()` и `parallel()`.

Возвращает

Экземпляр `chokidar` для детального контроля над настройками вотчинга.

Ошибки

Если non-string или массив с какими-либо non-strings's передается в виде globs, выдается ошибка с сообщением «Non-string provided as watch path».

Если строка или массив передаются в качестве `task`, выдается сообщение об ошибке: «watch task has to be a function (optionally generated by using gulp.parallel or gulp.series)».

Опции

Опция	Тип	Дефолт.	Примечание
ignoreInitial	boolean	true	Если false, задача вызывается во время создания экземпляра, когда обнаруживаются пути к файлам. Используется для запуска задачи непосредственно во время запуска. Примечание. Эта опция передается в chokidar, но по умолчанию имеет значение `true`, а не `false`.

Опция	Тип	Дефолт.	Примечание
delay	number	200	Задержка в миллисекундах между изменением файла и выполнением задачи. Позволяет ожидать множества изменений перед выполнением задачи, например, поиск и замена в большом количестве файлов.
queue	boolean	true	Если true и задача уже запущена, любые изменения файла будут поставлены в очередь выполнения одной задачи. Позволяет избежать наложения долго-выполняющихся задач.
events	string array	['add', 'change', 'unlink']	Наблюдаемые события запускают выполнение задачи. Это могут быть `'add'`, `'addDir'`, `'change'`, `'unlink'`, `'unlinkDir'`, `'ready'`, и/или `'error'`. Дополнительно доступно `'all'`, которое представляет все события, кроме `'ready'` и `'error'`. Данная опция передается напрямую в chokidar.
persistent	boolean	true	Если false, вотчер не будет продолжать процесс Node. Отключение этой опции не рекомендуется. Данная опция передается напрямую в chokidar.
ignored	array string RegExp function		Определяет globs, которые будут игнорироваться. Если предоставляется функция, она будет вызываться дважды для каждого пути - один раз только с путем, затем с путем и объектом `fs.Stats` этого файла. Данная опция передается напрямую в chokidar.
followSymlinks	boolean	true	Если true, изменения как символических ссылок, так и связанных файлов инициируют события. Если false, только изменения в символических ссылок инициируют события. Данная опция передается напрямую в chokidar.
cwd	string		Каталог будет объединен с любым относительным путем, чтобы сформировать абсолютный путь. Игнорируется для абсолютных путей. Используйте, чтобы избежать объединения globs с `path.join()`. Данная опция передается напрямую в chokidar.

Опция	Тип	Дефолт.	Примечание
disableGlobbing	boolean	false	Если true, все globs обрабатываются как буквенные имена путей, даже если они имеют специальные символы. Данная опция передается напрямую в chokidar.
usePolling	boolean	false	Если false, вотчер будет использовать для просмотра функцию `fs.watch()` (или fsevents на Mac). Если true, используется `fs.watchFile()`. Это необходимо для успешного просмотра файлов по сети или в других нестандартных ситуациях. Переопределяет значение по умолчанию `useFsEvents`. Данная опция передается напрямую в chokidar.
interval	number	100	В сочетании с `usePolling: true`. Интервал опроса файловой системы. Данная опция передается напрямую в chokidar.
binaryInterval	number	300	В сочетании с `usePolling: true`. Интервал опроса файловой системы для двоичных файлов. Данная опция передается напрямую в chokidar.
useFsEvents	boolean	true	Если true, использует fsevents для просмотра, если доступно. Если явно установлено значение true, заменяет параметр `usePolling`. Если установлено значение false, автоматически устанавливает `usePolling` значение в true. Данная опция передается напрямую в chokidar.
alwaysStat	boolean	false	Если true, всегда вызывает `fs.stat()` для измененных файлов. Это замедлит вотчинг файлов. Объект `fs.Stat` доступен, только если вы используете экземпляр chokidar напрямую. Данная опция передается напрямую в chokidar.
depth	number		Указывает, сколько вложенных уровней каталогов будет просмотрено. Данная опция передается напрямую в chokidar.
awaitWriteFinish	boolean	false	Не используйте эту опцию, вместо этого используйте `delay`. Данная опция передается напрямую в chokidar.

Опция	Тип	Дефолт.	Примечание
ignorePermissionErrors	boolean	false	Установите значение в true, чтобы просматривать файлы, не имеющие разрешений на чтение. Если просмотр не удастся из-за ошибок EPERM или EACCES, они будут пропущены без уведомления. Данная опция передается напрямую в chokidar.
atomic	number	100	Работает, только если `useFsEvents` и `usePolling` имеют значение false. Автоматически отфильтровывает артефакты, возникающие при «атомарной записи» некоторыми редакторами. Если файл повторно добавляется в течение указанных миллисекунд после удаления, будет отправлено событие change вместо unlink.

Экземпляр chokidar

Метод `watch()` возвращает базовый экземпляр chokidar, предоставляя полный контроль над настройкой вотчинга. Чаще всего используется для регистрации отдельных обработчиков событий, которые предоставляют `path` или `stats` измененных файлов.

При непосредственном использовании экземпляра chokidar у вас не будет доступа к интеграции системы задач, включая асинхронное завершение, организацию очереди и задержку.

```
const { watch } = require('gulp');

const watcher = watch(['input/*.js']);

watcher.on('change', function(path, stats) {
  console.log(`Файл ${path} был изменен`);
});

watcher.on('add', function(path, stats) {
  console.log(`Файл ${path} был добавлен`);
});

watcher.on('unlink', function(path, stats) {
  console.log(`Файл ${path} был удален`);
});

watcher.close();
```

`watcher.on(eventName, eventHandler)`

Регистрируются функции `eventHandler`, вызываемые при возникновении указанного события.

Параметр	Тип	Примечание
eventName	string	Можно наблюдать следующие события: `'add'`, `'addDir'`, `'change'`, `'unlink'`, `'unlinkDir'`, `'ready'`, `'error'`, или `'all'`.
eventHandler	function	Функция, вызываемая при возникновении указанного события. Аргументы подробно изложены в таблице ниже.

Аргумент	Тип	Примечание
path	string	Путь к файлу, который изменился. Если была установлена опция `cwd`, путь будет относительным путем удаления `cwd`.
stats	object	Объект `fs.Stat` (агрумент может быть неопределен). Если для параметра `alwaysStat` установлено значение `true`, `stats` будет предоставлен всегда.

`watcher.close()`

Выключает средство просмотра файлов. После выключения больше событий не будет.

`watcher.add(globs)`

Добавляет дополнительные globs к уже запущенному экземпляру вотчера.

Параметр	Тип	Примечание
globs	string array	Дополнительные globs для просмотра.

`watcher.unwatch(globs)`

Удаляет наблюдаемые globs, пока вотчер продолжает работу с оставшимися путями.

Параметр	Тип	Примечание
globs	string array	Globs к удалению.

task()

> Внимание: использование данного API не рекомендуется в настоящее время. Экспортируйте ваши задачи.

Определяет задачу в системе задач. Затем к задаче можно получить доступ из командной строки или API-интерфейсов `series()`, `parallel()` и `lastRun()`.

Использование

Зарегистрируйте именованную функцию как задачу:

```
const { task } = require('gulp');

function build(cb) {
  // тело функции
  cb();
}

task(build);
```

Зарегистрируйте анонимную функцию как задачу:

```
const { task } = require('gulp');

task('build', function(cb) {
  // тело функции
  cb();
});
```

Получить задачу, которая была зарегистрирована ранее:

```
const { task } = require('gulp');

task('build', function(cb) {
  // тело функции
  cb();
});

const build = task('build');
```

Ключ

```
task([taskName], taskFunction)
```

Параметры

Если `taskName` не указано, будет использовано свойство `name` именованной функции или пользовательское свойство `displayName`. Параметр `taskName` должен использоваться для анонимных функций, в которых отсутствует свойство `displayName`.

Поскольку любая зарегистрированная задача может быть запущена из командной строки, избегайте использования пробелов при именовании задач.

Параметр	Тип	Примечание
taskName	string	Псевдоним для функции задачи в системе задач. Не требуется при использовании именованных функций для `taskFunction`.
taskFunction (required)	function	Функция задачи (таска) или составная задача - генерируется с помощью `series()` и `parallel()`. В идеале, именованная функция. Метаданные задачи могут быть прикреплены для предоставления дополнительной информации в командной строке.

Возвращает

При регистрации задачи ничего не возвращается.

При получении задачи будет возвращена упакованная задача (не оригинальная функция), зарегистрированная как `taskName`. Обернутая задача имеет метод `unwrap()`, который возвращает исходную функцию.

Ошибки

При регистрации задачи, в которой `TaskName` отсутствует и функция `TaskFunction` является анонимной, выдается сообщение об ошибке «Task name must be specified».

Метаданные задачи

Свойство	Тип	Примечание

Свойство	Тип	Примечание
name	string	Особое свойство именованных функций. Используется для регистрации задачи. Примечание: `name` не доступно для записи, его нельзя установить или изменить.
displayName	string	При присоединении к `taskFunction` создается псевдоним задачи. В случаях, когда требуется использовать символы, которые не разрешены в именах функций, используйте это свойство.
description	string	При присоединении к `taskFunction` предоставляет описание, которое будет напечатано в командной строке при выводе списка задач.
flags	object	При присоединении к функции `taskFunction` предоставляет флаги, которые должны быть напечатаны в командной строке при перечислении задач. Ключи объекта представляют флаги, а значения - их описания.

```
const { task } = require('gulp');

const clean = function(cb) {
  // тело функции
  cb();
};
clean.displayName = 'clean:all';

task(clean);

function build(cb) {
  // тело функции
  cb();
}
build.description = 'Билд проекта';
build.flags = { '-e': 'Пример флага' };

task(build);
```

registry()

Позволяет подключать пользовательские реестры к системе задач, которые предоставляют общие задачи или расширенную функциональность.

Примечание. В пользовательский реестр будут включены только задачи, зарегистрированные с помощью `task()`. Функции задач, переданные непосредственно в `series()` или `parallel()` предоставляться не будут. Если вам нужно настроить поведение реестра, создавайте задачи со строковыми ссылками.

При назначении нового реестра каждая задача из текущего реестра будет перенесена, а текущий реестр будет заменен новым. Это позволяет добавлять несколько пользовательских реестров в последовательном порядке.

Использование

```
const { registry, task, series } = require('gulp');
const FwdRef = require('undertaker-forward-reference');

registry(FwdRef());

task('default', series('forward-ref'));

task('forward-ref', function(cb) {
  // тело функции
  cb();
});
```

Ключ

```
registry([registryInstance])
```

Параметры

Параметр	Тип	Примечание
registryInstance	object	Экземпляр (не класс) пользовательского реестра.

Возвращает

Если передан параметр `instanceInstance`, ничего не будет возвращено. Если аргументы не переданы, возвращает текущий экземпляр реестра.

Ошибки

Когда конструктор (вместо экземпляра) передается как `registryInstance`, выдает ошибку с сообщением: «Custom registries must be instantiated, but it looks like you passed a constructor».

Когда реестр без метода `get` передается как `registryInstance`, выдает ошибку с сообщением «Custom registry must have `get` function».

Когда реестр без метода `set` передается как `registryInstance`, выдает ошибку с сообщением «Custom registry must have `set` function».

Когда реестр без метода `init` передается как `registryInstance`, выдает ошибку с сообщением «Custom registry must have `init` function».

Когда реестр без метода `tasks` передается как `registryInstance`, выдает ошибку с сообщением «Custom registry must have `tasks` function».

---

tree()

Выбирает текущее дерево зависимостей задач - в редких случаях, когда это необходимо.

Скорее всего, вы не будете использовать `tree()`, однако не стоит забывать, что такая возможность есть и может пригодиться, когда необходимо в CLI отобразить дерево зависимости задач, определенных в gulpfile.

Использование

Пример gulpfile:

```
1  const { series, parallel } = require('gulp');
2
3  function one(cb) {
4    // тело функции
5    cb();
6  }
7
8  function two(cb) {
9    // тело функции
10   cb();
11 }
12
13 function three(cb) {
14   // тело функции
15   cb();
16 }
17
```

```
const four = series(one, two);

const five = series(four,
  parallel(three, function(cb) {
    // тело функции
    cb();
  })
);

module.exports = { one, two, three, four, five };
```

Вывод для `tree()`:

```
{
  label: 'Tasks',
  nodes: [ 'one', 'two', 'three', 'four', 'five' ]
}
```

Вывод для `tree({ deep: true })`:

```
{
  label: "Tasks",
  nodes: [
    {
      label: "one",
      type: "task",
      nodes: []
    },
    {
      label: "two",
      type: "task",
      nodes: []
    },
    {
      label: "three",
      type: "task",
      nodes: []
    },
    {
      label: "four",
      type: "task",
      nodes: [
        {
          label: "<series>",
          type: "function",
```

```
        branch: true,
        nodes: [
          {
            label: "one",
            type: "function",
            nodes: []
          },
          {
            label: "two",
            type: "function",
            nodes: []
          }
        ]
      }
    ]
  },
  {
    label: "five",
    type: "task",
    nodes: [
      {
        label: "<series>",
        type: "function",
        branch: true,
        nodes: [
          {
            label: "<series>",
            type: "function",
            branch: true,
            nodes: [
              {
                label: "one",
                type: "function",
                nodes: []
              },
              {
                label: "two",
                type: "function",
                nodes: []
              }
            ]
          },
          {
            label: "<parallel>",
            type: "function",
            branch: true,
            nodes: [
```

```
            {
              label: "three",
              type: "function",
              nodes: []
            },
            {
              label: "<anonymous>",
              type: "function",
              nodes: []
            }
          ]
        }
      ]
    }
  ]
}
```

Ключ

```
tree([options])
```

Параметры

Параметр	Тип	Примечание
options	object	Подробно в опциях ниже.

Возвращает

Объект, детализирующий дерево зарегистрированных задач, содержащий вложенные объекты со свойствами `'label'` и `'nodes'` (совместимые по archy).

Каждый объект может иметь свойство `type`, которое можно использовать для определения того, является ли node `task` или `function`.

Каждый объект может иметь свойство `branch`, которое, если оно `true`, указывает на то, что node был создан с помощью `series()` или `parallel()`.

Опции

Опция	Тип	Дефолт.	Примечание
deep	boolean	false	Если true, будет возвращено все дерево. Если false, будут возвращены только задачи верхнего уровня.

Vinyl

Виртуальный формат файла. Когда файл читается с помощью `src()`, генерируется объект Vinyl для его представления, включая путь, содержимое и другие метаданные.

Объекты Vinyl могут быть преобразованы с помощью плагинов. Они также могут быть сохранены в файловой системе с помощью `dest()`.

При создании ваших собственных объектов Vinyl, вместо генерации с помощью `src()` используйте внешний модуль `vinyl`, как показано в разделе «Использование» ниже.

Использование

```
const Vinyl = require('vinyl');

const file = new Vinyl({
  cwd: '/',
  base: '/test/',
  path: '/test/file.js',
  contents: new Buffer('var x = 123')
});

file.relative === 'file.js';

file.dirname === '/test';
file.dirname = '/specs';
file.path === '/specs/file.js';

file.basename === 'file.js';
file.basename = 'file.txt';
file.path === '/specs/file.txt';

file.stem === 'file';
file.stem = 'foo';
file.path === '/specs/foo.txt';
file.extname === '.txt';
file.extname = '.js';
```

```
25  file.path === '/specs/file.js';
```

Ключ

```
new Vinyl([options])
```

Параметры

Параметр	Тип	Примечание
options	object	Подробнее в опциях ниже.

Возвращает

Экземпляр класса Vinyl, представляющий один виртуальный файл.

Ошибки

Если какие-либо переданные параметры не соответствуют определениям свойств экземпляра (например, если `path` задан как число), происходит выброс, как определено в таблице.

Опции

Опция	Тип	Дефолт.	Примечание
cwd	string	`process.cwd()`	Каталог, из которого будут получены относительные пути. Будет нормализовано и удалены конечные разделители.
base	string		Используется для вычисления свойства `relative` экземпляра. Фолбек к значению `cwd`, если не установлено. Обычно устанавливается на основание glob. Будет нормализовано и удалены конечные разделители.
path	string		Полный, абсолютный путь к файлу. Будет нормализовано и удалены конечные разделители.

Опция	Тип	Дефолт.	Примечание
history	array	`[ ]`	Массив путей для предварительного заполнения `history` экземпляра Vinyl. Обычно происходит при получении нового объекта Vinyl от предыдущего объекта Vinyl. Если `path` и `history` пройдены, `path` добавляется к `history`. Каждый элемент будет нормализован, разделители будут удалены.
stat	object		Экземпляр `fs.Stats`, обычно является результатом вызова `fs.stat()` файла. Используется для определения того, представляет ли объект Vinyl каталог или символическую ссылку.
contents	ReadableStream Buffer `null`	`null`	Содержимое файла. Если `contents` является `ReadableStream`, оно помещается в клонируемо-читаемый поток.

Любые другие свойства `options` будут напрямую назначены экземпляру Vinyl.

```
const Vinyl = require('vinyl');

const file = new Vinyl({ foo: 'bar' });
file.foo === 'bar';
```

Экземпляр Vinyl

Каждый экземпляр объекта Vinyl будет иметь свойства и методы для доступа и/или изменения информации о виртуальном файле.

Свойства экземпляра

Все пути с внутренним управлением (любое свойство экземпляра, за исключением `contents` и `stat`) нормализуются и удаляются конечные разделители. См. «Нормализация и объединение» ниже, для получения дополнительной информации.

Свойство	Тип	Описание	Сброс

Свойство	Тип	Описание	Сброс
contents	ReadableStream Buffer `null`	Получает и задает содержимое виртуального файла. Если установлено значение ReadableStream, оно помещается в клонируемо-читаемый поток.	Если установлено любое значение, отличное от ReadableStream, Buffer или `null`.
stat	object	Получает и задает экземпляр `fs.Stats`. Используется при определении того, представляет ли объект Vinyl каталог или символическую ссылку.	
cwd	string	Получает и устанавливает текущий рабочий каталог. Используется для определения относительных путей.	Если задана пустая строка или любое нестроковое значение.
base	string	Получает и устанавливает базовый каталог. Используется для вычисления свойства `relative` экземпляра. На объекте Vinyl, сгенерированном с помощью `src()`, будет установлено glob base. Если установлено значение `null` или `undefined`, сработает фолбек к значению свойства экземпляра `cwd`.	Если задана пустая строка или любое значение non-string (кроме `null` или `undefined`).
path	string	Получает и задает полный абсолютный путь к файлу. Установка значения, отличного от текущего `path`, добавляет новый путь к свойству экземпляра `history`.	Если установлено любое non-string значение.
history	array	Массив всех значений `path`, которым был назначен объект Vinyl. Первый элемент - это исходный путь, а последний элемент - текущий путь. Данное свойство и его элементы должны рассматриваться как доступные только для чтения и изменяться только косвенно, путем установки свойства экземпляра `path`.	

Свойство	Тип	Описание	Сброс
relative	string	Получает относительный сегмент пути между свойствами экземпляра `base` и `path`.	Если установлено любое значение. Если есть доступ, когда `path` недоступен.
dirname	string	Получает и задает каталог свойства экземпляра `path`.	Если есть доступ, когда `path` недоступен.
stem	string	Получает и задает основу (имя файла без расширения) свойства экземпляра ``.	Если есть доступ, когда `path` недоступен.
extname	string	Получает и задает расширение свойства экземпляра `path`.	Если есть доступ, когда `path` недоступен.
basename	string	Получает и задает имя файла (`stem` + `extname`) свойства экземпляра `path`.	Если есть доступ, когда `path` недоступен.
symlink	string	Получает и задает референсный путь символической ссылки.	Если установлено любое non-string значение.

Методы экземпляра

Метод	Возвращаемый тип	Возвращает
`isBuffer()`	boolean	Если свойством экземпляра `contents` является Buffer, возвращает true.

Метод	Возвращаемый тип	Возвращает
`isStream()`	boolean	Если свойством экземпляра `contents` является Stream, возвращает true.
`isNull()`	boolean	Если свойством экземпляра `contents` является `null`, возвращает true.
`isDirectory()`	boolean	Если экземпляр представляет каталог, возвращает true. Экземпляр считается каталогом, когда `isNull()` возвращает true, свойство экземпляра `stat` является объектом и `stat.isDirectory()` возвращает true. Это предполагает, что объект Vinyl был создан с допустимым (или правильно сказать смоделированным) объектом `fs.Stats`.
`isSymbolic()`	boolean	Если экземпляр представляет собой символическую ссылку, возвращает true. Экземпляр считается символическим, когда `isNull()` возвращает true, свойство экземпляра `stat` является объектом и `stat.isSymbolicLink()` возвращает true. Это предполагает, что объект Vinyl был создан с допустимым (или правильно сказать смоделированным) объектом `fs.Stats`.
`clone([options])`	object	Новый объект Vinyl со всеми клонированными свойствами. По умолчанию все пользовательские свойства клонированы. Если параметр `deep` имеет значение false, пользовательские атрибуты будут клонированы не глубоко. Если для параметра `contents` установлено значение false, а свойством экземпляра содержимого является Buffer, то Buffer будет использоваться повторно, а не клонироваться.
`inspect()`	string	Возвращает отформатированную интерпретацию объекта Vinyl. Автоматически вызывается по Node's console.log.

Нормализация и конкатенация

Все свойства пути нормализуются их сеттерами. Объедините пути с помощью `/`, вместо использования `path.join()` и нормализация будет происходить правильно на всех платформах. Никогда не стройте пути с `\` - это допустимый символ имени файла в системе POSIX.

```
const file = new File();
file.path = '/' + 'test' + '/' + 'foo.bar';

console.log(file.path);
// posix => /test/foo.bar
// win32 => \\test\\foo.bar
```

Vinyl.isVinyl()

Определяет, является ли объект экземпляром Vinyl. Используйте этот метод вместо `instanceof`.

Примечание. В этом методе используется внутреннее свойство, которое не отображалось в некоторых старых версиях Vinyl, что приводит к ложному отрицанию (!) при использовании устаревшей версии.

Использование

```
const Vinyl = require('vinyl');

const file = new Vinyl();
const notAFile = {};

Vinyl.isVinyl(file) === true;
Vinyl.isVinyl(notAFile) === false;
```

Ключ

```
Vinyl.isVinyl(file);
```

Параметры

Параметр	Тип	Примечание
file	object	Объект для проверки.

Возвращает

true, если объект файла является экземпляром Vinyl.

Vinyl.isCustomProp()

Определяет, является ли свойство управляемым с помощью Vinyl. Используется экземпляром Vinyl при установке значений внутри конструктора или при копировании свойств в методе экземпляра `clone()`.

Этот метод полезен при расширении класса Vinyl. Подробно в «Расширение Vinyl» ниже.

Использование

```
1 const Vinyl = require('vinyl');
2
3 Vinyl.isCustomProp('sourceMap') === true;
4 Vinyl.isCustomProp('path') === false;
```

Ключ

```
Vinyl.isCustomProp(property)
```

Параметры

Параметр	Тип	Примечание
property	string	Наименование свойства для проверки.

Возвращает

true, если свойство не управляется изнутри.

Расширение Vinyl

Когда пользовательские свойства управляются внутри, статический метод `isCustomProp` должен быть расширен и возвращать значение false, если запрашивается одно из пользовательских свойств.

```
1 const Vinyl = require('vinyl');
```

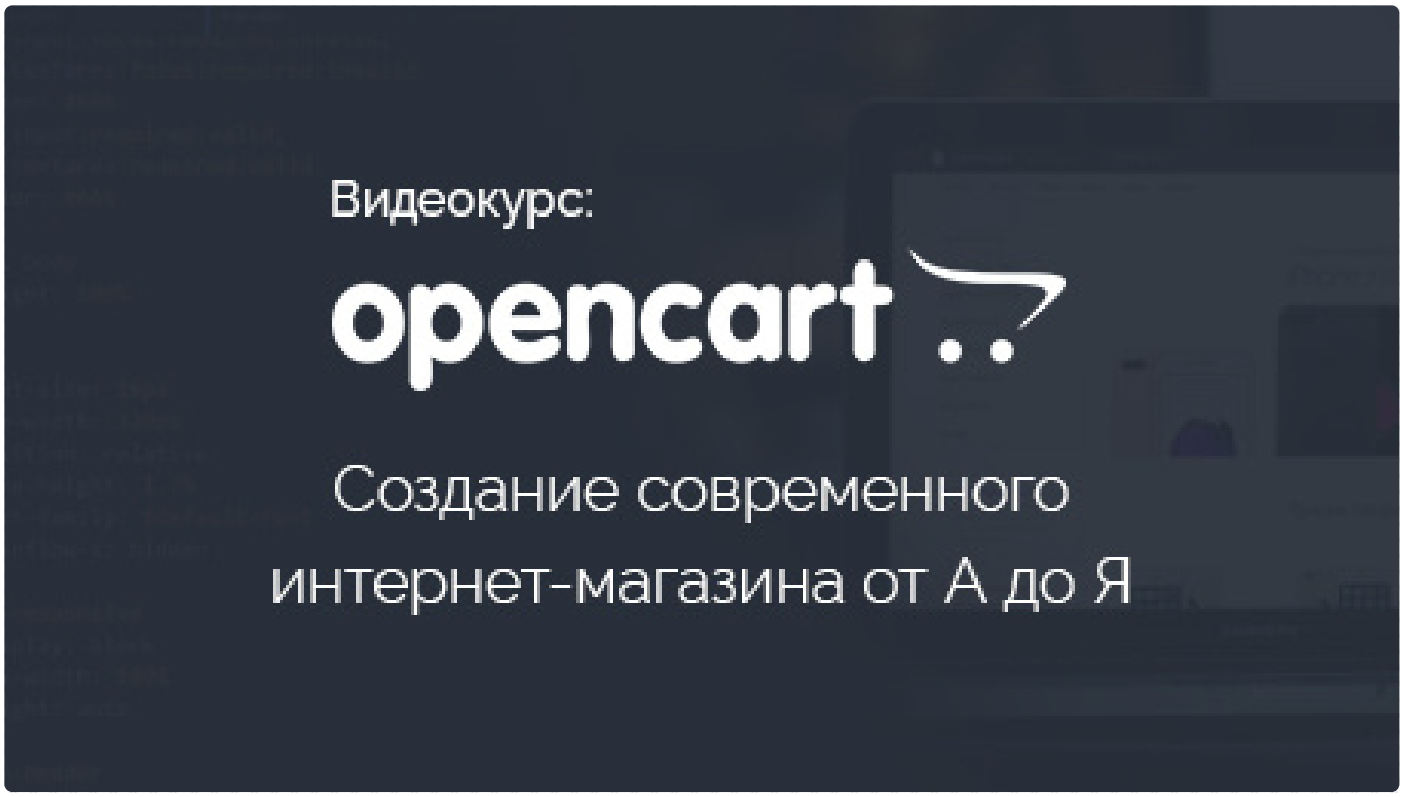
```
2
3  const builtInProps = ['foo', '_foo'];
4
5  class SuperFile extends Vinyl {
6    constructor(options) {
7      super(options);
8      this._foo = 'пример внутреннего значения только для чтения';
9    }
10
11   get foo() {
12     return this._foo;
13   }
14
15   static isCustomProp(name) {
16     return super.isCustomProp(name) && builtInProps.indexOf(name) === -1;
17   }
18 }
```

В приведенном выше примере `foo` и `_foo` не будут назначены новому объекту при клонировании или переданы через `options` в `new SuperFile(options)`.

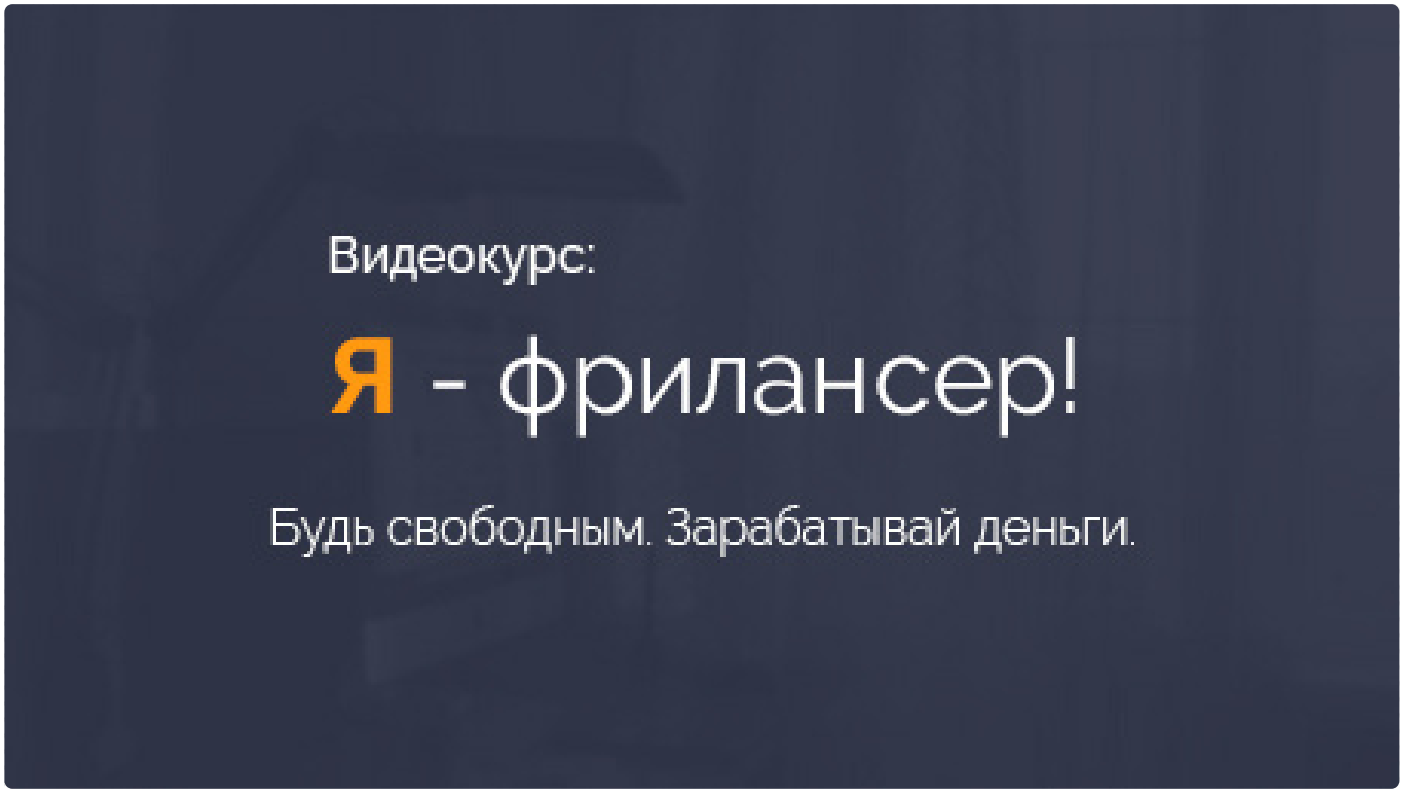
Если ваши пользовательские свойства или логика требуют особой обработки во время клонирования, переопределите метод клонирования при расширении Vinyl.

Премиум уроки от WebDesign Master

Создание сайта от А до Я. Комплексный курс



Создание современного интернет-магазина от А до Я



Я - фрилансер! Базовый курс для начинающих