Writable поток ответа res, метод pipe

Нашим следующим шагом будет использование потоков для работы с сетевыми соединениями. И начнем мы с отдачи посетителей файлов. Если помните, у нас была такая задача: если посетитель запросит следующий *url*, то отдать ему файл.

1. **if** (req.url == '/big.html') {
2.
3. fs.readFile('big.html', **function**(err, content) {
4. **if** (err) {
5. res.statusCode = 500;
6. res.end('Server error');
7. } **else** {
8. res.setHeader("Content-type", "text/html; charset=utf-8");
9. res.end(content);
10. }
11. });

Пример решения этой задачи без потоков может быть таким: читаем файл и вызываем *callback.* дальше при ошибке сообщаем о ней, а если все хорошо, то ставим заголовок, чтобы указать, какой это файл. И записываем содержимое файла в ответ вызовом res.end(content);  который отдает контент и завершает соединение. Это решение в принципе работает, но его проблема – это пожирание памяти, потому что если файл большой, то *readFile* его сначала считает, а потом вызовет *callback.* В результате получится, что если клиент медленный, то весь этот считанный контент зависнет в памяти до того, как клиент его получит.

А что если у нас таких медленных клиентов много? А если файл очень большой? Получается, что сервер почти мгновенно может занять всю доступную память, что совершенно неприемлемо. Чтобы такого не происходило, мы заменим код отдачи файла на принципиально другой, использующий потоки. Мы уже умеем читать s-файлы при помощи *ReadStream:*

1. **var** file = **new** fs.ReadStream('big.html');

Это будет входным потоком данных. А выходным потоком будет объект ответа *res*, который является объектом класса http.ServerResponse на следующим от stream.Writable.

Общий алгоритм использования потоков для записи сильно отличатся от того, что мы рассматривали ранее. Он выглядит так. Вначале мы создаем объект потока. Если у нас *http* сервер, то этот объект уже создан. Это *res.* Дальше мы хотим отправить что-то клиенту. Это можно сделать вызовом *res.write* и передать там наши данные. Обычно это либо буфер, либо строка. Наши данные при этом добавляются к специальному свойству потока, которое называют его буфером. Если пока этот буфер не очень большой, то данные прибавляются к нему, и *write* возвращает *true*, что означает, что мы можем писать еще. При этом обязательство по отсылке данных берет на себя уже сам поток. Как правило, эта отсылка происходит асинхронно.

Возможен и другой вариант. Например, если мы передали сразу очень много данных, или если буфер уже был чем-то занят, то метод *write* может вернуть *false*, который означает, что внутренний буфер потока переполнен и прямо сейчас запись, конечно, можно сделать, но это будет нецелесообразно, потому что в буфере все будет копиться, копиться… Поэтому при получении *false* обычно запись не продолжают, а ждут специального события *drain*, которое будет сгенерировано потоком, когда он все отошлет, то есть, когда его внутренний буфер опустеет.

Таким образом можем вызывать *write* много раз, и когда мы понимаем, что все данные записаны, то мы должны вызвать метод *end*. Тут тоже можно передать с первым аргументом данные. В этом случае он просто *write* вызовет. Самая главная задача *end –* это закончить запись. Поток это делает, при необходимости вызывает внутренние операции закрытия ресурсов (файлов), соединений и т.д. И потом генерирует события *finish*, который означает, что запись полностью завершена.

Обращаю ваше внимание, что аналогичное событие у *stream.Readable* называется *end.* Это различие неслучайно, потому что есть потоки *duplex*, которые умеют и читать, и писать. Соответственно, они могут генерировать как одно событие, так и другое.

Поток в любой момент можно разрушить вызовом метода *destroy*. При вызове этого метода работа потока прекращается, и все ассоциированные с ним ресурсы будут освобождены. Конечно, событие *finish* уже никогда не состоится, потому что это уже успешное окончание работы потока, успешная отдача всех данных.

Реализуем правильную отдачу файла, используя схему справок как шпаргалку. Я буду делать то в отдельной функции, которая будет называться *sendFile*. Она будет принимать один поток для файла и второй поток для ответа. Первое, что мы будем делать с такой функцией – это ждать данных:

1. **function** sendFile(file, res) {
2.
3. file.on('readable', write);

Затем, когда они получены, то внутри обработчика *Readable* читать их и отправлять в ответ:

1. **function** write() {
2. **var** fileContent = file.read(); // считать

Конечно же, она не выдерживает никакой критики, поскольку в том случае, если клиент пока не может получить эти данные (например, потому что у него медленная скорость соединения), то они зависнут в буфере объекта *res*:

1. res.write(fileContent)) { // отправить

Таким образом, если файл быстро считан, но пока не отправлен, то он займет большое количество памяти, а этого мы как раз хотели избежать. В этом небольшом коде изложен пример универсального решения этой задачи:

1. **var** fileContent = file.read(); // считать
2.
3. **if** (fileContent && !res.write(fileContent)) { // отправить
4.
5. file.removeListener('readable', write);
6.
7. res.once('drain', **function**() { // подождать
8. file.on('readable', write);
9. write();
10. });
11. }

Мы тоже читаем содержимое из файла на событие *Readable,* но мы не просто отправляем его вызовом *res.write,* а еще и анализируем, что этот вызов вернет. Если *res* принимает данные очень быстро, то *res.write* будет возвращать *true.* Это означает, что ветка *if* никогда не выполнится

1. **if** (fileContent && !res.write(fileContent)) {...}

Соответственно, мы получим *read-write, read-write* и т.д.

Более интересный случай, когда *res.write* вернул *false.* То есть, когда буфер переполнен, в этом случае мы временно отказываемся обрабатывать события *Readable* на файле.

1. file.removeListener('readable', write);

Само по себе такое снятие обработчика не означает, что файловый поток перестанет читать данные. Нет, он будет читать данные, но дочитает их до определенного уровня, заполнит свой внутренний буфер объекта файла, и затем, так как никто *read* не вызывает, то этот внутренний буфер останется заполнен на определенном уровне, то есть, файловый поток что-то считает и там застопорится. Далее мы дождемся события *drain*

1. res.once('drain', **function**() { // подождать
2. file.on('readable', write);
3. write();
4. });

То есть, когда данные будут успешно отданы в ответ (означает, что мы можем принять что-то еще из файла), мы вновь показываем свой интерес в событиях *Readable* и вызываем метод *write* сразу. Зачем? Просто потому что, пока мы ждали этого *drain*, новые данные вполне могли придти. Это означает, что имеет смысл их тут же прочитать. Вызов *read* вернет ??? в том случае, если данных нет. А если есть, то они просто будут обработаны тем же способом, о котором мы говорили раньше.

1. **if** (fileContent && !res.write(fileContent)) { // отправить
2.
3. file.removeListener('readable', write);
4.
5. res.once('drain', **function**() { // подождать
6. file.on('readable', write);
7. write();
8. });
9. }

Получается такая вот своеобразная рекурсивная функция: считать, отправить то, что считано, при необходимости подождать *drain;* считать, отправить и подождать, и т.д. по циклу, пока файл не закончится.

По окончании файла наступит событие *end,* в обработчике которого мы завершим ответ вызовом *res.end*. Таким образом, будет закрыто исходящее соединение, потому что файл полностью отослан. Получившийся код является весьма универсальным:

1. **var** http = require('http');
2. **var** fs = require('fs');
3.
4. **new** http.Server(**function**(req, res) {
5. // res instanceof http.ServerResponse < stream.Writable
6.
7. **if** (req.url == '/big.html') {
8.
9. **var** file = **new** fs.ReadStream('big.html');
10. sendFile(file, res);
11.
12. }
13. }).listen(3000);
14.
15. **function** sendFile(file, res) {
16.
17. file.on('readable', write);
18.
19. **function** write() {
20. **var** fileContent = file.read(); // считать
21.
22. **if** (fileContent && !res.write(fileContent)) { // отправить
23.
24. file.removeListener('readable', write);
25.
26. res.once('drain', **function**() { // подождать
27. file.on('readable', write);
28. write();
29. });
30. }
31. }
32.
33. file.on('end', **function**() {
34. res.end();
35. });
36.
37. }

Он реализует достаточно общий алгоритм отправки данных их одного потока в другой, используя самые стандартные методы потоков *Readable* и *Writeable.* Об этом, конечно же, подумали и разработчики ??? Node.js и добавили его несколько оптимизированную реализацию в стандартную библиотеку потоков.

Соответствующий метод называется *pipe*. Он есть у всех *Readable* потоков и работает так: *Readable.pipe Writeable* - куда писать. Кроме того, что это всего лишь одна строка, то есть еще один бонус, например, можно один и тот же входной поток пайпить в несколько выходных:

1. file.pipe(res);
2. file.pipe(res);

Например, кроме ответа клиенту будем выводить его еще стандартный вывод процесса:

1. file.pipe(res);
2. file.pipe(process.stdout);

Итак, запускаю. Вывелось одновременно и там, и там. Готов ли этот замечательный код к промышленной эксплуатации?

1. **var** http = require('http');
2. **var** fs = require('fs');
3.
4. **new** http.Server(**function**(req, res) {
5. // res instanceof http.ServerResponse < stream.Writable
6.
7. **if** (req.url == '/big.html') {
8.
9. **var** file = **new** fs.ReadStream('big.html');
10. sendFile(file, res);
11.
12. }
13. }).listen(3000);
14.
15. **function** sendFile(file, res) {
16.
17. file.pipe(res);
18.
19. }

Есть ли еще какие-то нюансы, которые нужно учесть? Первым делом в глаза должно броситься отсутствие работы с ошибками. Если вдруг файл не найден или что-то с ним еще не так, тогда вообще упадет весь сервер. Это не то, что нам нужно. Поэтому добавим, например, такой обработчик

1. file.on('error',**function**(err) {
2. res.statusCode = 500;
3. res.end('Server Error');
4. console.error(err);
5. })

 Теперь мы немножко ближе к реальной жизни, и в ряде руководств такой код выдается вполне нормальный, но на самом деле это не так. Ставить такой код на живой сервер ни в коем случае нельзя. В чем же дело? Для того, чтобы продемонстрировать проблему, я сейчас добавлю дополнительные обработчики на события *open* и *close* для файла. Стартую. Обновляю браузер и страницу. Обновляю, обновляю… Видите, файл перезагружается и совершенно нормально то, что файл открывается, потом он целиком отдается и закрывается.

А теперь открою консоль и запущу утилиту *curl*, которая будет скачивать <http://localhost:3000/big.html> с ограничением скорости 1кб/сек. Если вы работаете под Windows, то эту утилиту можно легко нагуглить и поставить. Запускаю. Открывается файл и начинается получение. С виду все хорошо. Жмем Ctrl+С, прекращаю загрузку. Обратите внимание, никакого *close* нету. Давайте еще раз. Получается, что если клиент открыл соединение, но закрыл его до того, как загрузка файла была завершена, то файл останется повисшим. А если файл остался открытым, то, во-первых, все ассоциированные с ним структуры остались тоже в памяти, во-вторых, операционные системы зачастую, если лимитное количество одновременно открытых файлов, а в-третьих, вместе с файлом навечно зависает в памяти и соответствующий объект потока. А вместе с ним и все замыкание, в котором он находится:

1. file.pipe(res);
2.
3. file.on('error', **function**(err) {
4. res.statusCode = 500;
5. res.end("Server Error");
6. console.error(err);
7. });
8.
9. file
10. .on('open',**function**() {
11. console.log("open");
12. })
13. .on('close', **function**() {
14. console.log("close");
15. });
16.
17. res.on('close', **function**() {
18. file.destroy();
19. });

Чтобы избежать этой проблемы и ее последствий, достаточно всего лишь отловить момент, когда соединение закрыто, и при этом удостовериться, что файл тоже будет закрыт.

Событие, которое нас интересует, называется *res.on('close').* Это событие отсутствует в обычном *stream.Writeable,* то есть, это именно расширение стандартного интерфейса потоков также, как у файлов, есть *close*, так и у объекта ответа сервера *res* тоже есть *close.* Но смысл второго *close* сильно отличается от смысла первого. Это очень важно, потому что на файловом потоке *close –* это нормальное завершение (файл закрывается всегда в конце), а для объекта ответа *close* – это сигнал, что соединение было оборвано. При нормальном завершении происходит не *close,* а *finish.* Итак, если соединение было оборвано, то нам нужно закрыть файл и освободить его ресурсы, поскольку файл нам больше передавать некому. Для этого мы вызываем метод потоков *file.destroy*:

1. res.on('close', **function**() {
2. file.destroy();
3. });

Теперь все будет хорошо. Давайте еще раз проверим. Запускаю. Теперь наш код можно пускать на живой сервер.