Содержание

5

Организация ввода-вывода данных на платформе Node.js

Через большинство активных веб-приложений и служб проходит немало данных. Эти данные поступают из текстовых файлов, символьных строк формата JSON, буферов и потоков двоичных данных. Именно по этой причине в Node.js встроен целый ряд механизмов, поддерживающих обработку данных, выводимых в систему и вводимых из нее. В этих механизмах очень важно разбираться, чтобы реализовывать эффективные и производительные веб-приложения и службы на платформе Node.js.

Основное внимание в этой главе уделяется вопросам обработки данных в формате JSON, манипулирования буферами двоичных данных, реализации читаемых и записываемых потоков данных, а также уплотнению и разуплотнению данных. В ней также поясняется, как извлечь выгоду из функциональных средств Node.js, чтобы удовлетворить разным требованиям ввода-вывода.

Обработка данных в формате JSON

К числу самых распространенных типов данных, пригодных для реализации веб-приложений и служб на платформе Node.js, относится формат JSON (JavaScript Object Notation — Представление объектов JavaScript). Формат JSON предоставляет упрощенный способ преобразования объектов JavaScript в форму символьных строк, и обратно, т.е. простой способ сериализации объектов данных при их обмене между клиентом и сервером, процессами, потоками данных или их сохранения в базе данных.

Для выбора формата JSON вместо XML в целях сериализации объектов JavaScript имеется несколько причин.

- Формат JSON намного более эффективен, поскольку требует меньше символов.
- Сериализация и десериализация осуществляются в формате JSON быстрее, чем в формате XML, поскольку его синтаксис проще.
- С точки зрения разработчиков, данные в формате JSON проще анализировать, поскольку они очень похожи на синтаксис JavaScript-кода.

Вескими основаниями для выбора формата XML вместо JSON могут стать сложность объектов или выполняемые преобразования XML/XSLT.

Преобразование данных из формата JSON в объекты JavaScript

Символьная строка в формате JSON представляет объект JavaScript в строковой форме. Синтаксис символьных строк похож на JavaScript-код, что упрощает их понимание. Для преобразования символьной строки из формата JSON в корректный объект JavaScript можно воспользоваться методом `JSON.parse(string)`.

Например, в приведенном ниже фрагменте кода определяется переменная `accountStr`, которой присваивается символьная строка в формате JSON, преобразующаяся в объект JavaScript с помощью метода `JSON.parse()`. Свойства этого объекта могут быть доступны посредством записи через точку.

```
var accountStr = '{"name":"Jedi",
                   "members":["Yoda","Obi Wan"],
                   "number":34512,
                   "location": "A galaxy far, far away"}';
var accountObj = JSON.parse(accountStr);
console.log(accountObj.name);
console.log(accountObj.members);
```

Выполнение приведенного выше фрагмента кода дает следующий результат:

```
Jedi
[ 'Yoda', 'Obi Wan' ]
```

Преобразование объектов JavaScript в данные формата JSON

На платформе Node.js имеется также возможность преобразовывать объекты JavaScript в надлежащим образом отформатированную символьную строку в формате JSON. Таким образом, данные в строковой форме могут быть сохранены в файле или базе данных, переданы через сетевое соединение по протоколу HTTP либо записаны в буфер или поток данных. Метод `JSON.stringify(`*`object`*`)` служит для преобразования объекта JavaScript в текстовую строку в формате JSON.

Например, в приведенном ниже фрагменте кода определяется объект JavaScript со свойствами типа строк, чисел и массивов. Весь этот объект преобразуется затем в символьную строку в формате JSON с помощью метода `JSON.stringify()`.

```
var accountObj = {
  name: "Baggins",
  number: 10645,
  members: ["Frodo, Bilbo"],
  location: "Shire"
};
var accountStr = JSON.stringify(accountObj);
console.log(accountStr);
```

Выполнение приведенного выше фрагмента кода дает следующий результат:

```
{"name":"Baggins", "number":10645,
 "members":["Frodo, Bilbo"], "location":"Shire"}
```

Буферизация данных с помощью модуля `Buffer`

Если манипулировать в коде JavaScript символами, представленными в Юникоде, удобно, то двоичными данными — не совсем. Тем не менее двоичные данные приносят определенную пользу при реализации некоторых веб-приложений и служб. Ниже приведены характерные тому примеры.

- Передача уплотненных файлов.
- Формирование динамических изображений.
- Пересылка сериализированных двоичных данных.

Общее представление о буферизированных данных

Буферизированные данные состоят из последовательного ряда октетов в формате с прямым или обратным порядком их следования. Это означает, что они занимают намного меньше места, чем текстовые данные. На платформе Node.js предоставляется модуль `Buffer` с функциональными средствами для создания, чтения, записи и манипулирования двоичными данными в структуре буфера. Модуль `Buffer` является глобальным, и поэтому для доступа к нему не требуется оператор с методом `require()`.

Буферизированные данные хранятся в структуре, аналогичной массиву, хотя это происходит за пределами обычной динамической памяти (так называемой "кучи") интерпретатора V8 в исходно распределенных ячейках неуправляемой памяти. Поэтому изменить размеры буфера нельзя.

Для взаимного преобразования буферизированных данных и символьных строк необходимо явно указать применяемый способ кодирования. Различные способы кодирования, поддерживаемые на платформе Node.js для взаимного преобразования буферизированных данных и символьных строк, перечислены в табл. 5.1.

Таблица 5.1. Способы кодирования буферизированных данных и символьных строк

Способ	Описание
`utf8`	В большинстве документов употребляются в качестве стандартных многобайтные символы в Юникоде
`utf16le`	Двухбайтные или четырехбайтные символы в Юникоде с прямым порядком следования байтов
`ucs2`	То же, что и кодировка **`utf16le`**
`base64`	Кодирование символьных строк по стандарту Base64
`Hex`	**Кодирование каждого байта в виде двух шестнадцатеричных символов**

Прямой и обратный порядок следования байтов

Двоичные данные хранятся в буферах в виде последовательности октетов или восьми нулей и единиц, которые можно выразить шестнадцатеричным значением в пределах от **0x00** да **0xFF**. Их можно прочитать как один байт или слово, состоящее из нескольких байтов. Порядок следования байтов означает расположение значащих бит, образующих слово. При обратном порядке следования байтов (big endian) первым записывается самое старшее значащее слово, а при прямом порядке следования байтов (little endian) — самое младшее значащее слово. Например, последовательность из четырех байтов **0x0A**, **0x0B**, **0x0C** и **0x0D**, составляющих 32-разрядное слово, и сохраненных в буфере, при обратном порядке их следования, будет выглядеть как **[0x0D**, **0x0C**, **0x0B**, **0x0A]**, а при прямом порядке — как **[0x0A**, **0x0B**, **0x0C**, **0x0D]**.

Создание буферов

Объекты типа `Buffer`, по существу, хранятся в исходно распределенных ячейках неуправляемой памяти, и поэтому размеры этих объектов должны быть непременно определены при их создании. Создать объекты типа `Buffer` можно с помощью ключевого слова `new` тремя следующими способами:

```
new Buffer(sizeInBytes)
new Buffer(octetArray)
new Buffer(string, [кодировка])
```

Например, в следующих строках кода определяются буферы с указанием размера в байтах, октетного буфера и символьной строки в кодировке UTF8:

```
var buf256 = new Buffer(256);
var bufOctets = new Buffer(
    [0x6f, 0x63, 0x74, 0x65, 0x74, 0x73]);
var bufUTF8 = new Buffer(
    "Some UTF8 Text \u00b6 \u30c6 \u20ac", 'utf8');
```

Запись данных в буферы

Размер объекта типа `Buffer` нельзя расширить после его создания, но данные можно записать в любое место буфера. В табл. 5.2 приводятся разные способы записи данных в буферы.

Таблица 5.2. Способы записи данных в буферы

Способ	Описание
buffer.write(*string*, [*offset*], [*length*], [*encoding*])	Записывает количество байтов ***length*** из символьной строки ***string***, начиная с позиции в буфере по индексу ***offset*** и применяя указанную кодировку ***encoding***
buffer[*offset*] = value	Заменяет данные по индексу ***offset*** указанным значением **value**
buffer.fill(*value*, [*offset*], [*end*])	Записывает указанное значение в каждый байт буфера, начиная с позиции по индексу ***offset*** и кончая позицией по индексу ***end***

Окончание табл. 5.2

Способ	Описание
writeInt8(*value*, *offset*, [*noAssert*]) **writeInt16LE(*value*, *offset*, [*noAssert*])** **writeInt16BE(*value*, *offset*, [*noAssert*])** ...	Для записи в буферы числовых данных (целых со знаком и без знака, с плавающей точкой одинарной и двойной точности, разного размера, с прямым и обратным следованием байтов) имеются самые разные методы объекта **Buffer**. Параметр ***value*** обозначает записываемое значение, параметр ***offset*** — индекс позиции в буфере, а параметр ***noAssert*** — пропуск проверки достоверности записываемого значения ***value*** и индекса позиции ***offset***. Для параметра ***noAssert*** следует оставить устанавливаемым по умолчанию логическое значение **false**, если нет полной уверенности в правильности остальных параметров

Для наглядной демонстрации описанных выше способов записи данных в листинге 5.1 определяется буфер, который сначала заполняется нулями, затем в его начале записывается некоторый текст методом `write()` в строке кода 3, а далее вводится дополнительный текст методом `write(string, offset, length)` в строке кода 5, изменяя часть уже имеющихся в буфере данных. И, наконец, в строке кода 7 знак `"+"` добавляется в самый конец текстовой строки, находящейся в буфере, путем непосредственного задания его значения по указанному индексу. Результаты выполнения исходного кода из листинга 5.1 приведены в листинге 5.2. Обратите внимание на то, что в операторе `buf256.write("more text", 9, 9)` данные записываются в середину текстовой строки, а в операторе `buf256[18] = 43` изменяется единственный байт в конец строки.

Листинг 5.1. Исходный файл `buffer_write.js`: различные способы записи данных в буфер

```
1 buf256 = new Buffer(256);
2 buf256.fill(0);
3 buf256.write("add some text");
4 console.log(buf256.toString());
5 buf256.write("more text", 9, 9);
6 console.log(buf256.toString());
7 buf256[18] = 43;
8 console.log(buf256.toString());
```

Листинг 5.2. Результаты выполнения исходного кода из файла `buffer_write.js`: вывод данных, записанных в буфер

```
C:\books\node\ch05>node buffer_write.js
add some text
add some more text
add some more text+
```

Чтение данных из буфера

Прочитать данные из буфера можно несколькими способами. Сделать это проще всего, вызвав метод `toString()`, чтобы преобразовать все содержимое буфера или его часть в символьную строку. А непосредственный доступ к содержимому буфера можно получить по индексу или с помощью метода `read()`. На платформе Node.js предоставляется также объект типа `StringDecoder`, у которого есть метод `write(`*`buffer`*`)`, декодирующий и выводящий буферизированные данные в указанной кодировке. Все эти способы чтения данных из буферов перечислены в табл. 5.3.

Таблица 5.3. Способы чтения данных из буфера

Способ	Описание
`buffer.toString([`*`encoding`*`], [`*`start`*`], [`*`end`*`])`	Возвращает символьную строку, содержащую декодированные символы в указанной кодировке ***`encoding`***, начиная с позиции в буфере по индексу ***`start`*** и кончая позицией по индексу ***`end`***. Если начальный и конечный индексы не указаны, то в данном методе по умолчанию выбирается начало и конец буфера
`stringDecoder.write(`*`buffer`*`)`	Возвращает декодированную строковую версию содержимого буфера
`buffer[`*`offset`*`]`	Возвращает из буфера значение октета по указанному индексу ***`offset`***
`readInt8(`*`offset`*`, [`*`noAssert`*`])` **`readInt16LE(`*`offset`*`, [`*`noAssert`*`])`** **`readInt16BE(`*`offset`*`, [`*`noAssert`*`])`** **`...`**	Для чтения из буферов числовых данных (целых со знаком и без знака, с плавающей точкой одинарной и двойной точности, разного размера, с прямым и обратным следованием байтов) используются самые разные методы объекта **`Buffer`**. Этим методам передается параметр ***`offset`*** для чтения данных из конкретного места в буфере и дополнительный параметр ***`noAssert`***, логическое значение которого обозначает, следует ли пропустить проверку достоверности заданного параметра ***`offset`***. Для параметра ***`noAssert`*** следует оставить устанавливаемым по умолчанию логическое значение **`false`**, если нет полной уверенности в правильности заданного параметра ***`offset`***

Для демонстрации описанных выше способов чтения данных из буфера в листинге 5.3 сначала определяется буфер, содержащий символы в кодировке UTF8, затем вызывается метод `toString()` без параметров для чтения всего буфера в целом, а далее — тот же самый метод, но с параметрами `encoding`, `start` и `end` для чтения отдельной части буфера. И, наконец, в строках кода 5–6 создается объект типа `StringDecoder` с заданной кодировкой UTF8. С помощью этого объекта выводится значение октета по индексу `18` и значение 32-разрядного слова с обратным порядком следования байтов, расположенного по этому же индексу. Результаты выполнения исходного кода из листинга 5.3 приведены в листинге 5.4.

Листинг 5.3. Исходный код из файла `buffer_read.js`: различные способы чтения данных из буфера

```
1 bufUTF8 = new Buffer(
2    "Some UTF8 Text \u00b6 \u30c6 \u20ac", 'utf8');
3 console.log(bufUTF8.toString());
4 console.log(bufUTF8.toString('utf8', 5, 9));
5 var StringDecoder = require('string_decoder').StringDecoder;
6 var decoder = new StringDecoder('utf8');
7 console.log(decoder.write(bufUTF8));
8 console.log(bufUTF8[18].toString(16));
9 console.log(bufUTF8.readUInt32BE(18).toString(16));
```

Листинг 5.4. Результаты выполнения исходного кода из файла `buffer_read.js`: вывод данных, читаемых из буфера

```
C:\books\node\ch05>node buffer_read.js
Some UTF8 Text ¶ ƒ Ъ
UTF8
Some UTF8 Text ¶ ƒ Ъ
e3
e3838620
```

Определение длины буфера

В операциях с буферами нередко требуется определить их длину, особенно в тех случаях, когда буфер создается динамически из символьной строки. Чтобы определить длину буфера, следует обратиться к свойству `length` объекта типа `Buffer`. Но это свойство не поможет, если требуется определить длину в байтах, которую занимает символьная строка в буфере. Вместо этого придется вызвать метод `Buffer.byteLength(`*`string`*`, [`*`encoding`*`])`. Следует, однако, иметь в виду, что длина символьной строки отличается от длины буфера. Чтобы наглядно продемонстрировать это отличие, ниже приведен следующий пример кода:

```
"UTF8 text \u00b6".length;
// вычисляется длина 11
Buffer.byteLength("UTF8 text \u00b6", 'utf8');
```

```
// вычисляется длина 12
Buffer("UTF8 text \u00b6").length;
// вычисляется длина 12
```

Как видите, строка состоит из 11 символов, но поскольку она содержит двухбайтный символ, то метод `byteLength()` возвращает ее длину, равную 12 байтам. А в строке кода `Buffer("UTF8 text \u00b6").length` также получается длина, равная 12, поскольку из свойства `length` возвращается длина буфера в байтах.

Копирование буферов

Важной особенностью обращения с буферами является возможность копировать данные из одного буфера в другой. Для этой цели на платформе Node.js предоставляется метод `copy(`*`targetBuffer`*`, [`*`targetStart`*`], [`*`sourceStart`*`], [`*`sourceIndex`*`])`, вызываемый для объектов типа `Buffer`. В качестве параметра *`targetBuffer`* (целевой буфер) указывается другой объект типа `Buffer`, а в качестве параметров *`targetStart`*, *`sourceStart`* и *`sourceIndex`* — индексы соответствующих позиций в целевом и исходном буферах.

> **Примечание**
>
> Чтобы скопировать строковые данные из одного буфера в другой, непременно убедитесь в одинаковой их кодировке в обоих буферах, иначе вы можете получить неожиданные результаты после декодирования содержимого в целевом буфере.

Кроме того, данные можно скопировать из одного буфера в другой, обратившись к ним непосредственно по индексу, как показано ниже.

```
sourceBuffer[index] = destinationBuffer[index]
```

В листинге 5.5 демонстрируются три примера копирования данных из одного буфера в другой. Первым способом в строках кода 4–8 копируется весь буфер в целом. А следующим способом в строках кода 10–14 копируются только 5 байтов из средины буфера. В третьем примере осуществляется циклический обход буфера, откуда байты копируются лишь через одного. Результаты выполнения исходного кода из листинга 5.5 приведены в листинге 5.6.

Листинг 5.5. Исходный код из файла `buffer_copy.js`: различные способы копирования данных из одного буфера в другой

```
01 var alphabet = new Buffer('abcdefghijklmnopqrstuvwxyz');
02 console.log(alphabet.toString());
03 // скопировать весь буфер в целом
04 var blank = new Buffer(26);
05 blank.fill();
06 console.log("Blank: " + blank.toString());
07 alphabet.copy(blank);
08 console.log("Blank: " + blank.toString());
09 // скопировать часть буфера
```

```
10 var dashes = new Buffer(26);
11 dashes.fill('-');
12 console.log("Dashes: " + dashes.toString());
13 alphabet.copy(dashes, 10, 10, 15);
14 console.log("Dashes: " + dashes.toString());
15 // скопировать данные из одного буфера в другой
16 // непосредственно по указанным индексам
17 var dots = new Buffer('-------------------------');
18 dots.fill('.');
19 console.log("dots: " + dots.toString());
20 for (var i=0; i < dots.length; i++){
21 if (i % 2) { dots[i] = alphabet[i]; }
22 }
23 console.log("dots: " + dots.toString());
```

Листинг 5.6. Результаты выполнения исходного кода из файла `buffer_copy.js`: вывод данных, скопированных из одного буфера в другой

```
C:\books\node\ch05>node buffer_copy.js
abcdefghijklmnopqrstuvwxyz
Blank:
Blank: abcdefghijklmnopqrstuvwxyz
Dashes: --------------------------
Dashes: ----------klmno-----------
dots: .........................
dots: .b.d.f.h.j.l.n.p.r.t.v.x.
```

Фрагментация буферов

Еще одной важной особенностью обращения с буферами является их разделение на фрагменты. В качестве *фрагмента* служит часть буфера от начального до конечного индекса. Фрагментация буфера позволяет манипулировать конкретными его частями.

Фрагменты создаются с помощью метода `slice([start], [end])`, возвращающего объект типа `Buffer` с индексом *start*, указывающим на исходный буфер длиной, равной *end* – *start*. Следует, однако, иметь в виду, что фрагмент отличается от копии. Если отредактировать копию, оригинал *не* претерпит никаких изменений. Но если отредактировать фрагмент, то оригинал претерпит изменения.

В листинге 5.7 демонстрируется применение фрагментов буфера. Обратите внимание на то, что при изменении фрагмента в строках кода 5 и 6 соответствующие изменения происходят и в исходном буфере, как следует из результатов, приведенных в листинге 5.8.

Листинг 5.7. Исходный код из файла `buffer_slice.js`: создание фрагментовбуфера и манипулирование ими

```
1 var numbers = new Buffer("123456789");
2 console.log(numbers.toString());
```

```
3 var slice = numbers.slice(3, 6);
4 console.log(slice.toString());
5 slice[0] = '#'.charCodeAt(0);
6 slice[slice.length-1] = '#'.charCodeAt(0);
7 console.log(slice.toString());
8 console.log(numbers.toString());
```

Листинг 5.8. Результаты выполнения исходного кода из файла `buffer_slice.js`: фрагментация и модификация буфера

```
C:\books\node\ch05>node buffer_slice.js
123456789
456
#5#
123#5#789
```

Сцепление буферов

Несколько буферов можно соединить вместе, чтобы сформировать новый буфер. С этой целью методу `concat(`*`list`*`, [`*`totalLength`*`])` передается массив объектов типа `Buffer` в качестве первого параметра, а в качестве второго дополнительного параметра *`totalLength`* — максимальное количество байтов в буфере. Буферы сцепляются в том порядке, в каком они появляются в списке, и в конечном итоге возвращается новый буфер, содержащий данные из исходных буферов в количестве байтов, определяемом параметром `totalLength`. Если не предоставить параметр `totalLength`, его значение будет автоматически определено в методе `concat()`. Но для этого придется обойти список, поэтому эффективнее предоставить значение параметра `totalLength` вручную.

В листинге 5.9 демонстрируется сцепление одного буфера с другим, а результаты этой операции приведены в листинге 5.10.

Листинг 5.9. Исходный код из файла `buffer_concat.js`: сцепление буферов

```
1 var af = new Buffer("African Swallow?");
2 var eu = new Buffer("European Swallow?");
3 var question = new Buffer("Air Speed Velocity of an ");
4 console.log(Buffer.concat([question, af]).toString());
5 console.log(Buffer.concat([question, eu]).toString());
```

Листинг 5.10. Результаты выполнения исходного кода из файла `buffer_concat.js`: сцепление буферов

```
C:\books\node\ch05>node buffer_concat.js
Air Speed Velocity of an African Swallow?
Air Speed Velocity of an European Swallow?
```

Организация потоков данных с помощью модуля `Stream`

Особое место на платформе Node.js принадлежит модулю `Stream`. Потоки данных представляют собой структуры в оперативной памяти, где можно как читать, так и записывать данные. Потоки данных применяются повсюду на платформе Node.js, например для доступа к файлам или чтения данных из HTTP-запросов и ряда других мест. В этом разделе рассматривается применение модуля `Stream` для организации потоков данных, а также чтения и записи в них информации.

Назначение потоков данных состоит в том, чтобы предоставить общий механизм для передачи данных из одного места в другое. Они генерируют также события, например в том случае, когда данные доступны для чтения либо возникают ошибки и т.д. Для обработки данных можно зарегистрировать соответствующие приемники, когда эти данные станут доступными в потоке для чтения или готовыми для записи в него.

Потоки данных чаще всего применяются для обработки HTTP-запросов и файлов. В частности, файл можно открыть как поток читаемых данных или получить доступ к данным из HTTP-запроса как из потока, чтобы прочитать из него байты по мере надобности. Кроме того, можно создать собственные потоки данных. В последующих разделах описывается процесс создания и применения читаемых, записываемых, дуплексных и преобразующих потоков данных.

Читаемые потоки данных

Читаемые (`Readable`) потоки данных предоставляют механизм, упрощающий чтение данных, поступающих в приложение из другого источника. Ниже приведены некоторые примеры читаемых потоков данных.

- HTTP-ответы от сервера клиенту
- HTTP-запросы к серверу
- Читаемые потоки данных из модуля `fs`
- Потоки данных из модуля `zlib`
- Потоки данных из модуля `crypto`
- TCP-сокеты
- Стандартные потоки вывода порожденных процессов `stdout` и `stderr`
- Стандартный поток ввода `process.stdin`

В читаемых потоках данных типа `Readable` предоставляется метод `read([`*`size`*`])` специально для чтения данных, где *`size`* — количество байтов, которое требуется прочитать из потока. Метод `read()` может возвратить объект типа `String` или `Buffer`, или же пустое значение `null`. В читаемых потоках данных типа `Readable` генерируются также перечисленные ниже события.

- **readable.** Возникает в том случае, если фрагмент данных может быть прочитан из потока.
- **data.** То же, что и событие `readable`, за следующим исключением: когда присоединяются приемники событий от данных, поток данных переходит в поточный режим работы, а обработчик событий от данных вызывается непрерывно до тех пор, пока в потоке не будут исчерпаны все данные.
- **end.** Генерируется потоком данных, когда данных больше нет.
- **close.** Возникает, когда закрывается базовый ресурс (например, файл).
- **error.** Генерируется, когда возникает ошибка получения данных.

В читаемых потоках типа `Readable` предоставляется также целый ряд методов, позволяющих читать их содержимое и манипулировать ими. В табл. 5.4 перечислены методы, доступные в объектах потоков данных типа `Readable`.

Таблица 5.4. Методы, доступные в объектах потоков данных типа `Readable`

Метод	Описание
`read([size])`	Читает данные из потока. Эти данные могут быть типа **`String`** или **`Buffer`**, или же пустым значением **`null`**. Последнее означает, что данных в потоке больше не осталось. Если же указан аргумент ***`size`***, то чтение данных из потока ограничивается количеством байтов, определяемым этим аргументом
`setEncoding(encoding)`	Задает кодировку, применяемую при возврате объекта типа **`String`** по запросу из метода **`read()`**
`pause()`	Приостанавливает генерирование событий объектом
`resume()`	Возобновляет генерирование событий объектом
`pipe(destination, [options])`	Направляет данные, выводимые из текущего потока, по конвейеру в записываемый поток данных, определяемый параметром ***`destination`***. А в качестве параметра ***`options`*** указывается объект JavaScript. Так, если указать в качестве этого параметра определение объекта **`{end:true}`**, то целевой записывающий поток завершится по окончании исходного читаемого потока
`unpipe([destination])`	Отсоединяет исходный читаемый поток данных от целевого записывающего потока

Чтобы самостоятельно реализовать специализированный читаемый поток данных, необходимо сначала унаследовать соответствующие функциональные средства из объектов потоков данных типа `Readable`. Сделать это проще всего с помощью метода `inherits()` из модуля `util` следующим образом:

```
var util = require('util');
util.inherits(MyReadableStream, stream.Readable);
```

А затем можно получить экземпляр объекта потока данных с помощью метода `call()`, как показано ниже.

```
stream.Readable.call(this, opt);
```

Необходимо также реализовать метод `_read()`, в котором вызывается метод `push()` для извлечения данных из читаемого потока. В результате вызова метода `push()` должны быть извлечены данные типа `String` или `Buffer` или же выведено пустое значение `null`.

В листинге 5.11 демонстрируются основы организации читаемого потока данных и чтения из него информации. Обратите внимание на то, что объект типа `Answers` сначала наследуется от объекта типа `Readable`, а затем в нем реализуется метод `Answers.prototye._read()` для извлечения данных из читаемого потока. Кроме того, непосредственный вызов метода `read()` в строке кода 18 приводит к чтению лишь первого элемента данных из потока, тогда как остальные данные читаются обработчиком событий `data`, определяемым в строках кода 19–21. Результаты выполнения исходного кода из листинга 5.11 приведены в листинге 5.12.

Листинг 5.11. Исходный код из файла `stream_read.js`: реализация специализированного читаемого потока данных

```
01 var stream = require('stream');
02 var util = require('util');
03 util.inherits(Answers, stream.Readable);
04 function Answers(opt) {
05   stream.Readable.call(this, opt);
06   this.quotes = ["yes", "no", "maybe"];
07   this._index = 0;
08 }
09 Answers.prototype._read = function() {
10   if (this._index > this.quotes.length){
11     this.push(null);
12   } else {
13     this.push(this.quotes[this._index]);
14     this._index += 1;
15   }
16 };
17 var r = new Answers();
18 console.log("Direct read: " + r.read().toString());
19 r.on('data', function(data){
20   console.log("Callback read: " + data.toString());
21 });
22 r.on('end', function(data){
23   console.log("No more answers.");
24 });
```

Листинг 5.12. Результаты выполнения исходного кода из файла `stream_read.js`: реализация специализированного читаемого потока данных

```
C:\books\node\ch05>node stream_read.js
Direct read: yes
Callback read: no
Callback read: maybe
No more answers.
```

Записываемые потоки данных

Записываемые (`Writable`) потоки данных предоставляют механизм, упрощающий запись данных в форме, удобной для употребления в других местах прикладного кода. Ниже приведены некоторые примеры записываемых потоков данных.

- HTTP-запросы со стороны клиента
- HTTP-ответы от сервера
- Записываемые потоки данных из модуля `fs`
- Потоки данных из модуля `zlib`
- Потоки данных из модуля `crypto`
- TCP-сокеты
- Стандартный поток ввода `stdin` порожденного процесса
- Стандартные потоки вывода `process.stdout` и `process.stderr`.

В записываемых потоках данных типа `Writable` предоставляется метод `write(`*`chunk`*`, [`*`encoding`*`], [`*`callback`*`])` специально для записи данных, где *`encoding`* — требующаяся кодировка, а *`callback`* — функция обратного вызова, которая выполняется, как только записываемые данные будут полностью выведены в поток. Метод `write()` возвращает логическое значение `true`, если данные были записаны успешно. В записываемых потоках данных типа `Writable` генерируются также перечисленные ниже события.

- **`drain`.** Это событие возникает, как только из метода `write()` возвратится логическое значение `false`. Оно уведомляет приемники подобных событий, что можно приступить к записи новых данных.
- **`finish`.** Это событие генерируется, когда для объекта типа `Writable` вызывается метод `end()`. Оно уведомляет, что все данные выведены в записываемый поток и больше никаких данных принято не будет.
- **`pipe`.** Это событие возникает, когда для объекта типа `Readable` вызывается метод `pipe()`, чтобы присоединить к исходному читаемому потоку данных текущий записываемый поток данных как целевой.

- **unpipe.** Это событие генерируется, когда для объекта типа `Readable` вызывается метод `unpipe()`, чтобы отсоединить от исходного читаемого потока данных текущий записываемый поток данных как целевой.

В записываемых потоках типа `Writable` предоставляется также целый ряд методов, позволяющих записывать в них данные и манипулировать ими. В табл. 5.5 перечислены методы, доступные в объектах потоков данных типа `Writable`.

Таблица 5.5. Методы, доступные в объектах потоков данных типа `Writable`

Метод	Описание
`Write(`*`chunk`*`, [`*`encoding`*`], [`*`callback`*`])`	Выводит указанный фрагмент данных ***`chunk`*** в записываемый поток. Эти данные могут быть типа **`String`** или **`Buffer`**. Если указана конкретная кодировка ***`encoding`***, она применяется для кодирования выводимых данных. А если задана функция обратного вызова ***`callback`***, то она выполняется по окончании вывода данных в записываемый поток
`end([`*`chunk`*`], [`*`encoding`*`], [`*`callback`*`])`	То же, что и метод **`write()`**, за исключением того, что он переводит записываемый поток данных в такое состояние, в котором он больше не принимает данные, отправляя в ответ событие **`finish`**

Чтобы самостоятельно реализовать специализированный записываемый поток данных, необходимо сначала унаследовать соответствующие функциональные средства из объектов потоков данных типа `Writable`. Сделать это проще всего с помощью метода `inherits()` из модуля `util` следующим образом:

```
var util = require('util');
util.inherits(MyWritableStream, stream.Writable);
```

А затем можно получить экземпляр объекта потока данных с помощью метода `call()`, как показано ниже.

```
stream. Writable.call(this, opt);
```

Необходимо также реализовать метод `_write(`*`data`*`, `*`encoding`*`, `*`callback`*`)`, сохраняющий данные для вывода в записываемый поток. В листинге 5.13 демонстрируются основы организации записываемого потока данных и записи в него информации. Результаты выполнения исходного кода из листинга 5.13 приведены в листинге 5.14.

Листинг 5.13. Исходный код из файла `stream_write.js`: реализация специализированного записываемого потока данных

```
01 var stream = require('stream');
02 var util = require('util');
03 util.inherits(Writer, stream.Writable);
04 function Writer(opt) {
05   stream.Writable.call(this, opt);
```

```
06   this.data = new Array();
07 }
08 Writer.prototype._write =
09        function(data, encoding, callback) {
10   this.data.push(data.toString('utf8'));
11   console.log("Adding: " + data);
12   callback();
13 };
14 var w = new Writer();
15 for (var i=1; i<=5; i++){
16   w.write("Item" + i, 'utf8');
17 }
18 w.end("ItemLast");
19 console.log(w.data);
```

Листинг 5.14. Результаты выполнения исходного кода из файла `stream_write.js`: реализация специализированного записываемого потока данных

```
C:\books\node\ch05>node stream_write.js
Adding: Item1
Adding: Item2
Adding: Item3
Adding: Item4
Adding: Item5
Adding: ItemLast
[ 'Item1', 'Item2', 'Item3', 'Item4',
  'Item5', 'ItemLast' ]
```

Дуплексные потоки данных

Дуплексный (`Duplex`) поток данных сочетает в себе функциональные возможности читаемого и записываемого потоков данных. Характерным примером дуплексного потока данных служит сетевое соединение между TCP-сокетами. Как только данные будут сформированы, их можно читать и записывать через сетевое соединение между сокетами.

Чтобы самостоятельно реализовать специализированный дуплексный поток данных, необходимо сначала унаследовать соответствующие функциональные средства из потоков данных типа `Duplex`. Сделать это проще всего с помощью метода `inherits()` из модуля `util` следующим образом:

```
var util = require('util');
util.inherits(MyDuplexStream, stream.Duplex);
```

А затем можно получить экземпляр объекта потока данных с помощью метода `call()`, как показано ниже.

```
stream. Duplex.call(this, opt);
```

В качестве параметра opt при создании объекта потока данных типа Duplex указывается логическое значение true или false свойства allowHalfOpen этого объекта. Если параметр opt принимает логическое значение true, то читаемая сторона сетевого соединения остается открытой даже по завершении приема данных на записываемой стороне, и наоборот. А если параметр opt принимает логическое значение false, то по завершении приема данных на записываемой стороне завершает передачу данных и читаемая сторона сетевого соединения, и наоборот. Реализуя дуплексный поток данных, необходимо также реализовать оба метода _read(*size*) и _write(*data*, *encoding*, *callback*) при прототипировании класса Duplex.

В листинге 5.15 демонстрируются основы организации дуплексного потока данных и обмена через него информацией. Несмотря на всю простоту приводимого в этом листинге примера, он все же демонстрирует основные принципы организации дуплексных потоков данных. В частности, объект типа Duplexer наследует от объекта потока данных типа Duplex и реализует простейший метод _write(), сохраняющий данные в массиве объекта текущего потока данных. А в методе _read() вызывается метод shift() для получения первого элемента данного массива. Если значение этого элемента равно "stop", то выводится пустое значение null. Если же это другое значение, то извлекается именно оно. А если значение вообще отсутствует, то устанавливается таймер блокировки по времени для обратного вызова метода _read() через 500 мс.

Результаты выполнения исходного кода из листинга 5.15 приведены в листинге 5.16. Обратите внимание на то, что первые две записываемые строки "I think, " и "therefore" читаются вместе, поскольку они были направлены в читаемый поток данных прежде, чем наступило событие data.

Листинг 5.15. Исходный код из файла stream_duplex.js: реализация дуплексного потока данных

```
01 var stream = require('stream');
02 var util = require('util');
03 util.inherits(Duplexer, stream.Duplex);
04 function Duplexer(opt) {
05   stream.Duplex.call(this, opt);
06   this.data = [];
07 }
08 Duplexer.prototype._read = function readItem(size) {
09   var chunk = this.data.shift();
10   if (chunk == "stop") {
11     this.push(null);
12   } else {
13     if(chunk) {
14      this.push(chunk);
15     } else {
16       setTimeout(readItem.bind(this), 500, size);
```

```
17     }
18   }
19 };
20 Duplexer.prototype._write =
21           function(data, encoding, callback) {
22   this.data.push(data);
23   callback();
24 };
25 var d = new Duplexer();
26 d.on('data', function(chunk) {
27   console.log('read: ', chunk.toString());
28 });
29 d.on('end', function() {
30   console.log('Message Complete');
31 });
32 d.write("I think, ");
33 d.write("therefore ");
34 d.write("I am.");
35 d.write("Rene Descartes");
36 d.write("stop");
```

Листинг 5.16. Результаты выполнения исходного кода из файла `stream_duplex.js`: реализация специализированного дуплексного потока данных

```
C:\books\node\ch05>node stream_duplex.js
read: I think,
read: therefore
read: I am.
read: Rene Descartes
Message Complete
```

Преобразующие потоки данных

Еще одной разновидностью потока данных является преобразующий (`Transform`) поток. Этот поток данных расширяет дуплексный поток, но в то же время он модифицирует данные, передаваемые между записываемым и читаемым потоками. Такой поток данных может оказаться удобным в том случае, если требуется преобразовать данные, передаваемые из одной системы в другую. Примеры преобразующих потоков данных таковы:

- потоки данных из модуля `zlib`;
- потоки данных из модуля `crypto`.

Главное отличие дуплексных потоков данных от преобразующих заключается в том, что в последних совсем не обязательно реализовывать прототипные методы `_read()` и `_write()`, поскольку они предоставляются в качестве сквозных функций. Вместо этого придется реализовать методы `_transform(`*`chunk`*`,` *`encoding`*`,` *`callback`*`)` и `_flush(`*`callback`*`)`. В частности, метод `_transform()`

должен принимать данные из запросов, передаваемых методом `write()`, модифицировать их, а затем извлекать модифицированные данные методом `push()`.

В листинге 5.17 демонстрируются основы реализации преобразующего потока данных типа `Transform`. Такой поток данных принимает символьные строки в формате JSON, преобразует их в объекты, а затем генерирует специальное событие `object` для отправки этих объектов любым приемникам подобных событий. Кроме того, объект модифицируется в методе `_transform()`, чтобы включить в него свойство `handled`, а затем отправить его в форме символьной строки. Обратите внимание на то, что строках кода 19–22 реализуется функция обработчика событий `object`, отображающая определенные свойства объекта. А в листинге 5.18 обратите внимание на то, что символьные строки в формате JSON теперь включают в себя свойство `handled`.

Листинг 5.17. Исходный код из файла `stream_transform.js`: реализация преобразующего потока данных

```
01 var stream = require("stream");
02 var util = require("util");
03 util.inherits(JSONObjectStream, stream.Transform);
04 function JSONObjectStream (opt) {
05   stream.Transform.call(this, opt);
06 };
07 JSONObjectStream.prototype._transform =
08           function (data, encoding, callback) {
09   object = data ? JSON.parse(data.toString()) : "";
10   this.emit("object", object);
11   object.handled = true;
12   this.push(JSON.stringify(object));
13   callback();
14 };
15 JSONObjectStream.prototype._flush = function(cb) {
16   cb();
17 };
18 var tc = new JSONObjectStream();
19 tc.on("object", function(object) {
20   console.log("Name: %s", object.name);
21   console.log("Color: %s", object.color);
22 });
23 tc.on("data", function(data) {
24   console.log("Data: %s", data.toString());
25 });
26 tc.write('{"name":"Carolinus", "color": "Green"}');
27 tc.write('{"name":"Solarius", "color": "Blue"}');
28 tc.write('{"name":"Lo Tae Zhao", "color": "Gold"}');
29 tc.write('{"name":"Ommadon", "color": "Red"}');
```

Листинг 5.18. Результаты выполнения исходного кода из файла `stream_transform.js`: реализация специализированного преобразующего потока данных

```
C:\books\node\ch05>node stream_transform.js
Name: Carolinus
Color: Green
Data: {"name":"Carolinus","color":"Green","handled":true}
Name: Solarius
Color: Blue
Data: {"name":"Solarius","color":"Blue","handled":true}
Name: Lo Tae Zhao
Color: Gold
Data: {"name":"Lo Tae Zhao","color":"Gold","handled":true}
Name: Ommadon
Color: Red
Data: {"name":"Ommadon","color":"Red","handled":true}
```

Конвейеризация читаемых и записываемых потоков данных

Самой замечательной особенностью читаемых и записываемых потоков данных является возможность связывать их в цепочку с помощью метода `pipe(writableStream, [options])`. Этот метод направляет выход из читаемого потока данных по конвейеру непосредственно на вход записываемого потока данных. А качестве параметра *options* методу `pipe()` передается объект со свойством `end`, в котором установлено логическое значение `true` или `false`. Если в свойстве `end` данного объекта установлено логическое значение `true`, как в приведенном ниже примере, то записываемый поток данных завершается тогда же, когда и читаемый поток данных, причем такое поведение считается стандартным по умолчанию.

```
readStream.pipe(writeStream, {end:true});
```

С помощью метода `unpipe(destinationStream)` конвейер потоков данных можно разъединить. В листинге 5.19 демонстрируется сначала реализация читаемого и записываемого потоков данных, а затем их связывание в цепочку с помощью метода `pipe()`. А в листинге 5.20 демонстрируется основной процесс, в ходе которого данные, введенные из метода `_write()`, выводятся на консоль.

Листинг 5.19. Исходный код из файла `stream_piped.js`: конвейеризациячитаемых и записываемых потоков данных

```
01 var stream = require('stream');
02 var util = require('util');
03 util.inherits(Reader, stream.Readable);
04 util.inherits(Writer, stream.Writable);
05 function Reader(opt) {
06   stream.Readable.call(this, opt);
07   this._index = 1;
```

```
08 }
09 Reader.prototype._read = function(size) {
10   var i = this._index++;
11   if (i > 10){
12     this.push(null);
13   } else {
14   this.push("Item " + i.toString());
15   }
16 };
17 function Writer(opt) {
18   stream.Writable.call(this, opt);
19   this._index = 1;
20 }
21 Writer.prototype._write =
22        function(data, encoding, callback) {
23   console.log(data.toString());
24   callback();
25 };
26 var r = new Reader();
27 var w = new Writer();
28 r.pipe(w);
```

Листинг 5.20. Результаты выполнения исходного кода из файла `stream_piped.js`: реализация конвейеризации потоков данных

```
C:\books\node\ch05>node stream_piped.js
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10
```

Уплотнение и разуплотнение данных с помощью модуля `Zlib`

Когда приходится иметь дело с крупными системами или перемещать большие массивы данных, то их полезно уплотнять и разуплотнять. Для этой цели на платформе Node.js предоставляется отличная библиотека в модуле `Zlib`, позволяющая легко и эффективно уплотнять и разуплотнять данные в буферах.

Следует, однако, иметь в виду, что уплотнение данных отнимает вычислительные ресурсы ЦП. Поэтому следует непременно убедиться, насколько выгодно уплотнять данные, прежде чем решаться на затраты, которые влечет за собой

уплотнение и разуплотнение данных. Ниже перечислены способы уплотнения и разуплотнения данных, поддерживаемые в модуле `Zlib`.

- **gzip/gunzip.** Стандартный способ уплотнения и разуплотнения данных по алгоритму gzip.
- **deflate/inflate.** Стандартный способ сжатия данных по алгоритму, основанному на кодировании по методу Хаффмана.
- **deflateRaw/inflateRaw.** Способ сжатия в буфере необработанных данных.

Уплотнение и разуплотнение данных в буферах

В модуле `Zlib` предоставляется ряд вспомогательных функций, упрощающих процесс уплотнения и разуплотнения данных в буферах. Все эти функции определяются в одной и той же общей форме `function(`*`buffer`*`, `*`callback`*`)`, где `function` — способ уплотнения и разуплотнения данных, *`buffer`* — конкретный буфер, в котором уплотняются или разуплотняются данные, а *`callback`* — функция обратного вызова, которая выполняется по завершении процесса уплотнения или разуплотнения данных.

Продемонстрировать уплотнение и разуплотнение данных в буферах проще всего на конкретных примерах. Так, в листинге 5.21 представлен ряд примеров такого уплотнения и разуплотнения, а в листинге 5.22 приведен получающийся в каждом примере размер итоговых данных.

Листинг 5.21. Исходный код из файла `zlib_buffers.js`: уплотнение и разуплотнение данных в буферах с помощью модуля `Zlib`

```
01 var zlib = require("zlib");
02 var input = '..............text..............';
03 zlib.deflate(input, function(err, buffer) {
04   if (!err) {
05     console.log("deflate (%s): ", buffer.length,
06                 buffer.toString('base64'));
07     zlib.inflate(buffer, function(err, buffer) {
08       if (!err) {
09         console.log("inflate (%s): ", buffer.length,
10                     buffer.toString());
11       }
12     });
13     zlib.unzip(buffer, function(err, buffer) {
14       if (!err) {
15         console.log("unzip deflate (%s): ",
16                     buffer.length,buffer.toString());
18       }
19     });
20   }
21 });
18
19 zlib.deflateRaw(input, function(err, buffer) {
```

```
20   if (!err) {
21     console.log("deflateRaw (%s): ", buffer.length,
22                 buffer.toString('base64'));
23     zlib.inflateRaw(buffer, function(err, buffer) {
24       if (!err) {
25         console.log("inflateRaw (%s): ",
26                     buffer.length, buffer.toString());
27       }
28     });
29   }
30 });
31
32 zlib.gzip(input, function(err, buffer) {
33   if (!err) {
34     console.log("gzip (%s): ", buffer.length,
35                 buffer.toString('base64'));
36     zlib.gunzip(buffer, function(err, buffer) {
37       if (!err) {
38         console.log("gunzip (%s): ", buffer.length,
39                     buffer.toString());
40       }
41     });
42     zlib.unzip(buffer, function(err, buffer) {
43       if (!err) {
44        console.log("unzip gzip (%s): ", buffer.length,
45                    buffer.toString());
46       }
47     });
48   }
49 });
```

Листинг 5.22. Результаты выполнения исходного кода из файла `zlib_buffers.js`: уплотнение и разуплотнение данных в буферах

```
C:\books\node\ch05>node zlib_buffers.js
deflate (18): eJzT00MBJakVJagiegB9Zgcq
deflateRaw (12): 09NDASWpFSWoInoA
gzip (30): H4sIAAAAAAAAC9PTQwElqRUlqCJ6AIq+x+AiAAAA
inflate (34): ..............text..............
unzip deflate (34): ..............text..............
inflateRaw (34): ..............text..............
gunzip (34): ..............text..............
unzip gzip (34): ..............text..............
```

Уплотнение и разуплотнение данных в потоках

Уплотнение и разуплотнение данных в потоках выполняется средствами модуля `Zlib` несколько иначе, чем в буферах. В этом случае используется метод `pipe()` для передачи данных по конвейеру из одного потока в другой через

объект уплотнения или разуплотнения. Подобным способом можно сжимать данные в любых читаемых и записываемых потоках.

Характерным тому примером служит уплотнение содержимого файла в потоках данных типа `fs.ReadStream` и `fs.WriteStream`. В листинге 5.23 демонстрируется пример уплотнения содержимого файла через объект типа `zlib.Gzip` и обратного их разуплотнения через объект типа `zlib.Gunzip`.

Листинг 5.23. Исходный код из файла `zlib_file.js`: уплотнение и разуплотнение содержимого файла в потоках данных с помощью модуля `Zlib`

```
01 var zlib = require("zlib");
02 var gzip = zlib.createGzip();
03 var fs = require('fs');
04 var inFile = fs.createReadStream('zlib_file.js');
05 var outFile = fs.createWriteStream('zlib_file.gz');
06 inFile.pipe(gzip).pipe(outFile);
07 gzip.flush();
08 outFile.close();
09 var gunzip = zlib.createGunzip();
10 var inFile = fs.createReadStream('zlib_file.gz');
11 var outFile = fs.createWriteStream('zlib_file.unzipped');
12 inFile.pipe(gunzip).pipe(outFile);
```

Резюме

В основу большинства серьезных веб-приложений и служб положен интенсивный обмен потоками данных между системами. В этой главе было показано, как пользоваться функциональными средствами, встроенными в Node.js для обработки данных в формате JSON, манипулирования буферизированными двоичными данными и организации потоков данных. В ней также пояснялись способы уплотнения и разуплотнения буферизированных и потоковых данных.

Продолжение

В следующей главе будет показано, как взаимодействовать с файловой системой из платформы Node.js. В ней будет разъяснено, как организовать чтение и запись данных в файлы, создавать каталоги и получать информацию из файловой системы.