Lecture 15. Streams in Node js

13 April 2025 19:02

Node.js Streams are a powerful feature that allows you to handle data efficiently, especially when dealing with large amounts of data. Streams enable you to read, write, or process data piece-by-piece (or chunk-by-chunk) rather than loading the entire data into memory at once. This approach minimizes memory consumption and improves performance, particularly for large files or real-time applications.

Types of Streams in Node.js

There are four main types of streams in Node.js:

1. **Readable Streams**: These streams allow you to read data from a source (e.g., files, HTTP requests, etc.).
2. **Writable Streams**: These streams allow you to write data to a destination (e.g., files, HTTP responses, etc.).
3. **Duplex Streams**: These streams are both readable and writable, enabling bidirectional communication (e.g., a TCP socket).
4. **Transform Streams**: A subtype of Duplex streams that can modify the data as it is being read or written (e.g., compressing or encrypting data).

How Streams Work

Streams work by reading or writing data in **chunks** instead of handling the entire data at once. This improves efficiency when dealing with large amounts of data, as you don't need to load the entire file into memory before processing it.

Stream Methods and Events

Each stream has different methods and events for interacting with it. Here's a breakdown of key methods and events for each type of stream.

1. Readable Streams

- **Methods:**
  - stream.read(size): Reads the specified number of bytes from the stream.
  - stream.pipe(destination): Pipes the readable stream into a writable stream.
- **Events:**
  - data: Emitted when data is available to be read from the stream.
  - end: Emitted when there is no more data to read from the stream.
  - error: Emitted when there is an error in reading data.

Example: Reading from a file with a readable stream:

javascript
Copy code

```
const fs = require('fs');
// Create a readable stream from a file
const readStream = fs.createReadStream('example.txt', { encoding: 'utf8' });
// Event: 'data' - when a chunk of data is available to read
readStream.on('data', (chunk) => {
  console.log('Data received: ', chunk);
});
// Event: 'end' - when there is no more data to read
readStream.on('end', () => {
  console.log('File reading completed.');
});
// Event: 'error' - when an error occurs
readStream.on('error', (err) => {
  console.error('Error reading the file:', err);
});
```

2. Writable Streams

- **Methods:**
  - stream.write(data): Writes data to the stream.
  - stream.end(): Signals the end of the stream (optional, but required for most writable streams).
- **Events:**
  - drain: Emitted when the internal buffer of the writable stream is emptied and can accept more data.
  - finish: Emitted when all data has been flushed to the destination.

Example: Writing to a file with a writable stream:

javascript
Copy code

```
const fs = require('fs');
// Create a writable stream to a file
const writeStream = fs.createWriteStream('output.txt');
// Write data to the stream
writeStream.write('Hello, World!\n');
writeStream.write('This is a test.\n');
// End the writable stream
writeStream.end(() => {
  console.log('Data has been written to the file.');
});
```

3. Duplex Streams

These streams are both readable and writable, meaning they can handle both input and output. They are useful for applications like network connections where you can send and receive data.

Example: TCP socket (duplex stream):

javascript
Copy code

```
const net = require('net');
// Create a TCP server
const server = net.createServer((socket) => {
  console.log('Client connected');
// Send data to the client
  socket.write('Hello, client!');
// Handle data from the client
  socket.on('data', (data) => {
    console.log('Received from client:', data.toString());
  });
// Handle client disconnection
  socket.on('end', () => {
    console.log('Client disconnected');
  });
});
// Start the server
server.listen(8080, () => {
  console.log('Server listening on port 8080');
});
```

4. Transform Streams

Transform streams are a special type of Duplex stream that allow you to modify or transform the data as it's being read or written. A common example is compressing or encrypting data.

Example: Using a transform stream to uppercase text:

javascript
Copy code

```
const { Transform } = require('stream');
// Create a transform stream that converts data to uppercase
```

```
const upperCaseStream = new Transform({
  transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
});
// Create a readable stream to read data
const readableStream = require('fs').createReadStream('example.txt');
// Pipe the readable stream through the transform stream and into a writable stream
readableStream.pipe(upperCaseStream).pipe(process.stdout);
```

Using Streams Together

Streams can be chained together using the pipe() method to process data in a sequence, which is very efficient for tasks like reading from a file, transforming data, and writing it to another file.

Example: Chaining Streams (Read -> Transform -> Write):

javascript
Copy code

```
const fs = require('fs');
const { Transform } = require('stream');
// Create a transform stream that converts data to uppercase
const upperCaseStream = new Transform({
  transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
});
// Create a readable stream to read data from the file
const readStream = fs.createReadStream('input.txt');
// Create a writable stream to write the modified data
const writeStream = fs.createWriteStream('output.txt');
// Pipe the streams together
readStream.pipe(upperCaseStream).pipe(writeStream);
```

Benefits of Streams in Node.js

- **Efficient Memory Usage**: Data is processed in chunks, so you don't have to load the entire data into memory.
- **Better Performance**: Especially useful when dealing with large files or streams of data like HTTP requests, real-time communications, or database queries.
- **Non-blocking**: Streams allow Node.js to process data asynchronously, improving responsiveness in I/O-bound tasks.