Cat-File Command Implementation

This file implements Git's `cat-file` command in Node.js. The `cat-file` command is used to examine the contents of Git objects stored in the repository's object database.

Purpose

The `cat-file` command retrieves and displays the content of Git objects (blobs, trees, commits, or tags) by their SHA-1 hash. It's primarily used for debugging and inspecting Git's internal storage.

Class Structure

```
class CatFileCommand {
    constructor(flag, commitSHA) {
        this.flag = flag;
        this.commitSHA = commitSHA;
    }
}
```

The class takes two parameters:

- `flag`: Command-line flag that determines the output format (currently only supports `-p`)
- `commitSHA`: The SHA-1 hash of the Git object to examine (despite the name, it works with any Git object, not just commits)

Step-by-Step Execution

1. Parameter Extraction

```
const flag = this.flag;
const commitSHA = this.commitSHA;
```

Extracts the flag and SHA from the instance variables for local use.

2. Flag Processing

```
switch(flag) {
    case '-p': {
```

```
        // Implementation for -p flag
    }
    break;
}
```

Uses a switch statement to handle different flags. Currently only implements the `-p` flag, which means "pretty-print" the object content.

3. Object Path Construction

```
const folder = commitSHA.slice(0,2);
const file = commitSHA.slice(2);

const completePath = path.join(process.cwd(),'.git', 'objects', folder, file);
```

Follows Git's standard object storage pattern:

- First 2 characters of SHA become the directory name
- Remaining characters become the filename
- Full path: `.git/objects/ab/c123def456...`

4. File Validation

```
if(!fs.existsSync(completePath)){
    throw new Error(`Not a valid object name ${commitSHA}`);
}
```

Checks if the object file exists in the Git object database, throwing an error if not found.

5. Reading and Decompressing

```
const fileContents = fs.readFileSync(completePath)
const outputBuffer = zlib.inflateSync(fileContents)
```

- Reads the compressed object file from disk
- Decompresses it using zlib (all Git objects are compressed)

6. Content Extraction

```
const output = outputBuffer.toString().split('\x00')[1];
```

This is the key parsing step:

- Converts the decompressed buffer to a string
- Splits on null character (`\x00`)
- Takes the second part (index 1), which is the actual content

Why this works: Git objects have the format `{type} {size}\0{content}`, so splitting on null and taking the second part gives us just the content.

7. Output

process.stdout.write(output);

Writes the extracted content directly to stdout without any additional formatting.

Example Usage Scenarios

For a blob object (file content):

node cat-file.js -p abc123def456
Output: Hello World

For a commit object:

node cat-file.js -p def456abc123
Output:
tree 1234567890abcdef
parent 0987654321fedcba
author John Doe <john@example.com> 1234567890 +0000
committer John Doe <john@example.com> 1234567890 +0000
#
Initial commit

For a tree object:

node cat-file.js -p 567890abcdef
Output:
100644 blob abc123 README.md
040000 tree def456 src

Git Object Format Reminder

All Git objects follow this format:

{type} {size}\0{content}

Where:

- `{type}`: blob, tree, commit, or tag
- `{size}`: size of content in bytes
- `\0`: null character separator
- `{content}`: the actual object data

Limitations

- Only implements the `-p` flag (pretty-print)
- Missing other common flags like `-t` (show type), `-s` (show size)
- Error handling is basic
- The parameter is named `commitSHA` but works with any Git object type

What the `-p` Flag Does

The `-p` flag tells `cat-file` to "pretty-print" the object content:

- For blobs: Shows the raw file content
- For trees: Shows the formatted directory listing
- For commits: Shows the commit message and metadata
- For tags: Shows the tag information

This implementation provides the core functionality of Git's `cat-file -p` command, which is essential for examining Git's internal object storage.