Git Write-Tree Command Implementation Explained

Overview

This file implements a JavaScript version of Git's `write-tree` command, which creates tree objects representing the entire directory structure of a repository. Tree objects are Git's way of storing directory hierarchies and file references, forming the foundation of Git's snapshot-based version control system.

Architecture Overview

The implementation consists of two main components:

1. **writeFileBlob()** - Creates blob objects for individual files
2. **WriteTreeCommand class** - Orchestrates the recursive tree creation process

Helper Function: writeFileBlob()

Purpose

Creates Git blob objects for individual files, which store the actual file content in Git's object database.

Implementation Breakdown

1. Reading File Content

```
const contents = fs.readFileSync(currentPath);
const len = contents.length;
```

- Reads the entire file content into memory
- Determines the byte length for the Git object header

2. Creating Git Blob Object

```
const header = `blob ${len}\0`;
const blob = Buffer.concat([Buffer.from(header), contents]);
```

- Creates the standard Git object format: `blob <size>\0<content>`
- The null byte (`\0`) separates the header from the actual file content
- This format is identical to how Git internally stores file content

3. Generating Content Hash

```
const hash = crypto.createHash("sha1").update(blob).digest("hex");
```

- Computes SHA-1 hash of the complete blob object (header + content)
- This hash becomes the unique identifier for this specific file content
- Same content always produces the same hash (content-addressable storage)

4. Storing the Blob

```
const folder = hash.slice(0, 2);
const file = hash.slice(2);
const completeFolderPath = path.join(process.cwd(), '.git', 'objects', folder);

// Create directory if needed
if (!fs.existsSync(completeFolderPath)) {
    fs.mkdirSync(completeFolderPath, { recursive: true });
}

// Compress and store
const compressData = zlib.deflateSync(blob)
fs.writeFileSync(path.join(completeFolderPath, file), compressData);
```

- Uses Git's standard storage pattern: first 2 hash characters as directory, remaining 38 as filename
- Creates directories recursively if they don't exist
- Compresses the blob using zlib deflate (same as Git)
- Writes the compressed blob to the calculated path

Main Class: WriteTreeCommand

Core Method: execute()

The main logic is contained within a nested recursive function `recursiveCreateTree()`.

Recursive Tree Creation Algorithm

1. Directory Traversal

```
function recursiveCreateTree(basePath) {
```

```
    const dirContents = fs.readdirSync(basePath);
    const result = [];

    for (const dirContent of dirContents) {
        if (dirContent.includes(".git")) continue;
        // ... process each item
    }
}
```

- Reads all items in the current directory
- Skips `.git` directory to avoid infinite recursion and irrelevant files
- Maintains a `result` array to collect tree entries

2. Item Processing Logic

```
const currentPath = path.join(basePath, dirContent);
const stat = fs.statSync(currentPath);

if (stat.isDirectory()) {
    const sha = recursiveCreateTree(currentPath);
    if (sha) {
        result.push({
            mode: "40000",        // Directory mode
            basename: path.basename(currentPath),
            sha,
        });
    }
} else if (stat.isFile()) {
    const sha = writeFileBlob(currentPath);
    result.push({
        mode: "100644",         // Regular file mode
        basename: path.basename(currentPath),
        sha,
    });
}
```

Directory Processing:

- Recursively processes subdirectories
- Uses mode "40000" (Git's standard directory mode)
- Only adds directory entry if it contains valid content (non-empty)

File Processing:

- Creates blob objects for files using `writeFileBlob()`
- Uses mode "100644" (Git's standard file mode for regular files)
- Always adds file entries to the tree

3. File Modes in Git

- **40000**: Directory
- **100644**: Regular file
- **100755**: Executable file
- **120000**: Symbolic link

4. Tree Object Construction

```
const treeData = result.reduce((acc, current) => {
   const { mode, basename, sha } = current;
   return Buffer.concat([
      acc,
      Buffer.from(`${mode} ${basename}\0`),
      Buffer.from(sha, "hex")
   ]);
}, Buffer.alloc(0));
```

Tree Entry Format: Each entry follows Git's binary format:

- `<mode> <filename>\0<20-byte-sha>`
- Mode and filename are text
- Null byte separates filename from SHA
- SHA is stored as 20 raw bytes (not hex string)

5. Final Tree Object Creation

```
const tree = Buffer.concat([Buffer.from(`tree ${treeData.length}\0`), treeData]);
const hash = crypto.createHash('sha1').update(tree).digest('hex');
```

- Creates complete Git tree object with header: `tree <size>\0<tree_data>`
- Computes SHA-1 hash of the entire tree object
- This hash uniquely identifies this specific directory structure

6. Storage and Compression

```
const folder = hash.slice(0, 2);
const file = hash.slice(2);
const treeFolderPath = path.join(process.cwd(), '.git', 'objects', folder)

if (!fs.existsSync(treeFolderPath)) {
   fs.mkdirSync(treeFolderPath);
```

```
}

const compressed = zlib.deflateSync(tree)
fs.writeFileSync(path.join(treeFolderPath, file), compressed);
```

- Follows same storage pattern as blobs
- Compresses and stores the tree object
- Returns the tree hash for parent directory reference

Key Technical Concepts

Content-Addressable Storage

- Objects are identified by their content hash, not location
- Same directory structure always produces same tree hash
- Enables efficient deduplication and integrity checking

Recursive Tree Structure

- Each directory becomes a tree object
- Tree objects reference other trees (subdirectories) and blobs (files)
- Creates a complete snapshot of the directory hierarchy

Git Object Types Integration

- **Blobs**: Store file content
- **Trees**: Store directory structure and file metadata
- **Commits**: Reference trees and add commit metadata
- **Tags**: Mark specific commits

Binary Format Efficiency

- Tree entries use binary format for space efficiency
- SHA hashes stored as 20 bytes instead of 40-character hex strings
- Null bytes used as separators instead of newlines

Usage Context

This command creates the tree structure that commits reference:

1. `write-tree` creates tree objects for entire repository
2. `commit-tree` creates commit pointing to root tree

3. Branches and tags reference specific commits

Potential Improvements

1. **Gitignore support**: Currently ignores only `.git` directory
2. **Symbolic link handling**: No special handling for symlinks
3. **File permissions**: Only handles regular files, not executables
4. **Error handling**: Limited validation and error recovery
5. **Performance**: Could optimize for large directories

This implementation demonstrates Git's elegant approach to storing directory structures as immutable, content-addressed objects that can be efficiently compared, shared, and reconstructed.