师资培训课件（ node 题目）

1. 十四届省赛 Markdown 文档解析

```
class Parser {
  constructor() {
    // 用于匹配不同Markdown语法的正则表达式
    this.heading = /^(#{1,6}\s+)/; // 匹配标题
    this.blockQuote = /^(\>\s+)/; // 匹配引用区块
    this.unorderedList = /^((\*|-){1}\s+)/; // 匹配无序列表
    this.image = /\!\[(.*?)\]\((.*?)\)/g; // 匹配图片
    this.strongText = /\*{2}(.*?)\*{2}/g; // 匹配粗体文本
    this.codeLine = /\`{1}(.*?)\`{1}/g; // 匹配行内代码
    // TODO: 添加水平分隔线的正则表达式
    this.hr = /^(\*{3,}|-{3,})/; // 匹配水平分隔线
  }

  // 设置要解析的行文本
  parseLineText(lineText) {
    this.lineText = lineText;
  }

  // 检查行是否为空
  isEmptyLine() {
    return this.lineText === "";
  }

  // 解析空行
  parseEmptyLine() {
```

```
    return "<br/>";
  }

  // 检查行是否为标题
  isHeading() {
    return this.heading.test(this.lineText);
  }

  // 解析标题
  parseHeading() {
    const temp = this.lineText.split(" ");
    const headingLevel = temp[0].length;
    const title = temp[1].trim();
    return `<h${headingLevel}>${title}</h${headingLevel}>`;
  }

  // TODO: 实现解析水平分隔线、引用区块、无序列表、图片、粗体文本和行内代码的方法

  // 检查行是否为无序列表项
  isUnorderedList() {
    return this.unorderedList.test(this.lineText);
  }

  // 解析无序列表项
  parseUnorderedList() {
    const tempStr = this.lineText.replace(this.unorderedList, "");
    return "<li>" + tempStr + "</li>";
  }

  // 检查行是否为水平分隔线
  isHr() {
    return this.hr.test(this.lineText);
  }

  // 解析水平分隔线
  parseHr() {
    return "<hr>";
  }

  // 检查行是否为引用区块
  isBlockQuote() {
    return this.blockQuote.test(this.lineText);
  }

  // 解析引用区块
  parseBlockQuote() {
    const tempStr = this.lineText.replace(this.blockQuote, "");
    return "<p>" + tempStr + "</p>";
  }

  // 检查行是否包含图片
  isImage() {
    return this.image.test(this.lineText);
  }

  // 解析图片
  parseImage(str) {
```

```
    return str.replace(this.image, (result, str1, str2) => {
      return '<img src="' + str2 + '" alt="' + str1 + '">';
    });
  }

  // 检查行是否包含粗体文本
  isStrongText() {
    return this.strongText.test(this.lineText);
  }

  // 解析粗体文本
  parseStrongText(str) {
    return str.replace(this.strongText, (result, str1) => {
      return "<b>" + str1 + "</b>";
    });
  }

  // 检查行是否包含行内代码
  isCodeLine() {
    return this.codeLine.test(this.lineText);
  }

  // 解析行内代码
  parseCodeLine(str) {
    return str.replace(this.codeLine, (result, str1) => {
      return "<code>" + str1 + "</code>";
    });
  }

  // 解析行内元素，如行内代码、图片和粗体文本
  inlineParse() {
    let str = this.lineText;

    // 解析行内代码
    if (this.isCodeLine()) {
      str = this.parseCodeLine(str);
    }
    // 解析图片
    if (this.isImage()) {
      str = this.parseImage(str);
    }
    // 解析粗体文本
    if (this.isStrongText()) {
      str = this.parseStrongText(str);
    }
    return str;
  }
}

class Reader {
  constructor(text) {
    // 存储原始文本
    this.text = text;
    // 将文本拆分为行
    this.lines = this.getLines();
    // 创建Parser的新实例
    this.parser = new Parser();
```

```
  }

  // 运行解析器以解析Markdown内容
  runParser() {
    let currentLine = 0;
    let hasParsed = [];

    while (!this.reachToEndLine(currentLine)) {
      // 设置当前行文本到解析器
      this.parser.parseLineText(this.getLineText(currentLine));

      // 检查空行
      if (this.parser.isEmptyLine()) {
        hasParsed.push(this.parser.parseEmptyLine());
        currentLine++;
        continue;
      }

      // 解析标题
      if (this.parser.isHeading()) {
        hasParsed.push(this.parser.parseHeading());
        currentLine++;
        continue;
      }

      // 检查水平分隔线
      if (this.parser.isHr()) {
        hasParsed.push(this.parser.parseHr());
        currentLine++;
        continue;
      }

      // 检查引用区块
      if (this.parser.isBlockQuote()) {
        const tempParsed = ["<blockquote>"];
        while (currentLine < this.lines.length &&
this.parser.isBlockQuote()) {
          tempParsed.push(this.parser.parseBlockQuote());
          currentLine++;
          if (currentLine < this.lines.length) {
            this.parser.parseLineText(this.getLineText(currentLine));
          }
        }
        tempParsed.push("</blockquote>");
        hasParsed.push(...tempParsed);
        continue;
      }

      // 检查无序列表
      if (this.parser.isUnorderedList()) {
        const tempParsed = ["<ul>"];
        while (currentLine < this.lines.length &&
this.parser.isUnorderedList()) {
          tempParsed.push(this.parser.parseUnorderedList());
          currentLine++;
          if (currentLine < this.lines.length) {
            this.parser.parseLineText(this.getLineText(currentLine));
```

```
          }
        }
        tempParsed.push("</ul>");
        hasParsed.push(...tempParsed);
        continue;
      }

      // 解析行内元素
      const tempStr = this.parser.inlineParse();
      hasParsed.push(tempStr);
      currentLine++;
    }
    return hasParsed.join("");
  }

  // 获取指定行的文本
  getLineText(lineNum) {
    return this.lines[lineNum];
  }

  getLines() {
    this.lines = this.text.split("\n");
    return this.lines;
  }

  reachToEndLine(line) {
    return line >= this.lines.length;
  }
}

module.exports = function parseMarkdown(markdownContent) {
  return new Reader(markdownContent).runParser();
};
```

2. 找到未引用的图片

```
const findUnlinkImages = async function () {
  const unlinkImages = []; // 未被任何 md 文件引用的图片的数组
  // TODO 请通过 Node.js 在此处继续完成代码编写
  const articles = await traversalDir(articlesPath);
  for (let i = 0; i < articles.length; i++) {
    const filename = articles[i];
    const filedir = path.join(articlesPath, filename);
    const stats = await fs.statSync(filedir); // 判断目标是否为文件
    if (stats.isFile()) {
      const md = fs.readFileSync(filedir, "utf8");
      searchImage(md); // 检索出文章内的图片链接
    }
  }
  useImgs = [...new Set(useImgs)]; // 去重，非必要
  const allNames = await traversalDir(imagesPath);
```

```
  const allImgs = await getAllImages(allNames); // 获取全部图片
  let diff = allImgs
    .concat(useImgs)
    .filter((x) => !allImgs.includes(x) || !useImgs.includes(x)); // 取差集，
最好是用 ES7 的 includes；ES6 可以用 Array.from结合Set；ES5 可以用 indexOf
  for (let i = 0; i < diff.length; i++) {
    // fs.unlinkSync(path.join(imagesPath, diff[i])) // 删除文件
    unlinkImages.push(diff[i]);
  }
  // console.log(`找到了 ${diff.length} 张无效图片`);
  // 获取全部图片的相对地址
  async function getAllImages(res) {
    return res.map((x) => "../images/" + x);
  }
  // TODO-END
  return unlinkImages; // 此处应返回一个数组，如不明白，请仔细阅读题目
};
```

node 题目属于专属大学组的考点，考点难度为中等，分数占比 15 -20 分。必考题，考题数量 1，高频高点集中在 `http` 与 `fs` 模块。 node 题目通常会结合别的考点如：数组操作，字符串操作等作为一道综合题出现。

备考建议：node 属于大学组的专属考点，通常属于逻辑题，考点设置不会特别难，建议有条件的同学积极复习 fs 和 http 模块中的常见知识点，建议拿到此题的满分或者部分分数。

高频考点讲解：

fs模块：

Node.js中的 `fs` 模块是文件系统模块，用于对文件进行读写操作。以下是`fs`模块中一些常见的方法：

1. **fs.readFile(path[, options], callback)**: 用于异步读取文件的内容。参数`path`是文件路径，`options`是一个可选的对象，用于指定编码等选项，`callback`是回调函

数，它的参数是读取到的文件内容。

```
const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

`fs.readFileSync()`是Node.js中`fs`模块提供的一个同步方法，用于同步地读取文件的内容。与`fs.readFile()`不同，`fs.readFileSync()`会阻塞代码执行，直到文件读取完成才会继续执行后续代码。

以下是`fs.readFileSync()`方法的基本用法示例：

```
const fs = require('fs');

const filePath = 'example.txt';

try {
  const data = fs.readFileSync(filePath, 'utf8');
  console.log('File content:', data);
} catch (err) {
  console.error('Error reading file:', err);
}
```

在上面的示例中，`fs.readFileSync()`接受两个参数：要读取的文件的路径和文件的编码（可选，默认为`'utf8'`）。该方法会返回文件的内容。如果文件读取过程中发生错误，会抛出一个异常，我们可以通过`try...catch`语句来捕获并处理这个异常。

需要注意的是，由于`fs.readFileSync()`是同步的，它会阻塞代码的执行，直到文件读取完成。因此，在读取大型文件或在需要高响应性的应用程序中，最好使用`fs.readFile()`等异步方法，以避免阻塞主线程。

2. **fs.writeFile(file, data[, options], callback)**: 用于异步写入文件。参数`file`是文件路径，`data`是要写入的数据，`options`是一个可选的对象，用于指定编码等选项，`callback`是写入完成后的回调函数。

```
const fs = require('fs');

fs.writeFile('example.txt', 'Hello, world!', 'utf8', (err) => {
  if (err) throw err;
```

```
  console.log('File written successfully.');
});
```

`fs.writeFileSync()`是Node.js中`fs`模块提供的一个同步方法，用于同步地将数据写入文件中。与`fs.writeFile()`不同，`fs.writeFileSync()`会阻塞代码执行，直到文件写入完成才会继续执行后续代码。

以下是`fs.writeFileSync()`方法的基本用法示例：

```
const fs = require('fs');

const filePath = 'example.txt';
const content = 'This is the content to write to the file.';

try {
  fs.writeFileSync(filePath, content, 'utf8');
  console.log('File written successfully');
} catch (err) {
  console.error('Error writing file:', err);
}
```

在上面的示例中，`fs.writeFileSync()`接受三个参数：要写入的文件的路径、要写入的内容以及文件的编码（可选，默认为`'utf8'`）。该方法会将指定内容写入到文件中。如果文件写入过程中发生错误，会抛出一个异常，我们可以通过`try...catch`语句来捕获并处理这个异常。

需要注意的是，由于`fs.writeFileSync()`是同步的，它会阻塞代码的执行，直到文件写入完成。因此，在写入大量数据或在需要高响应性的应用程序中，最好使用`fs.writeFile()`等异步方法，以避免阻塞主线程。

3. **fs.existsSync(path)**: 判断文件或目录是否存在。如果存在，则返回`true`，否则返回`false`。

```
const fs = require('fs');

if (fs.existsSync('example.txt')) {
  console.log('File exists.');
} else {
  console.log('File does not exist.');
}
```

4. **fs.unlink(path, callback)**: 用于异步删除文件。

```
const fs = require('fs');

fs.unlink('example.txt', (err) => {
  if (err) throw err;
  console.log('File deleted successfully.');
});
```

5. `fs.stat()`

`fs.stat()`是Node.js中`fs`模块提供的一个方法，用于获取文件或目录的状态信息。该方法可以用于检查文件或目录的存在性、类型、大小、权限等属性。

下面是`fs.stat()`方法的基本用法示例：

```
const fs = require('fs');

const path = 'example.txt';

fs.stat(path, (err, stats) => {
  if (err) {
    console.error(err);
    return;
  }

  console.log('Stats:', stats);
  console.log('File size in bytes:', stats.size);
  console.log('Is it a file?', stats.isFile());
  console.log('Is it a directory?', stats.isDirectory());
  console.log('Is it a symbolic link?', stats.isSymbolicLink());
});
```

在上面的示例中，`fs.stat()`函数接受两个参数：要查询的文件或目录的路径和一个回调函数。回调函数有两个参数，第一个参数是错误对象，如果查询过程中出现错误，该参数将不为null。第二个参数是表示文件或目录状态信息的`stats`对象。

`stats`对象包含了很多有用的信息，比如：

- `stats.size`：文件大小（以字节为单位）。
- `stats.isFile()`：如果是一个文件，则返回true，否则返回false。
- `stats.isDirectory()`：如果是一个目录，则返回true，否则返回false。
- `stats.isSymbolicLink()`：如果是一个符号链接（软链接），则返回true，否则返回false。
- 等等。

通过使用`fs.stat()`方法，你可以获取文件或目录的各种属性信息，从而根据需要进行相应的操作，比如检查文件类型、大小等。

http模块：

Node.js中的`http`模块是用于创建HTTP服务器和客户端的模块，可以用来搭建Web服务器。以下是`http`模块中一些常见的方法：

1. **http.createServer([options][, requestListener])**: 创建一个HTTP服务器。参数`options`是一个可选的对象，用于指定服务器的配置，`requestListener`是一个回调函数，用于处理请求并返回响应。

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, world!\n');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

```
const http = require('http');

const server = http.createServer((req, res) => {
  // 根据请求的路径进行处理
  if (req.url === '/hello') {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello, there!\n');
  } else {
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.end('404 Not Found\n');
  }
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

2. **http.get(options[, callback])**: 发起一个HTTP GET请求。参数`options`是一个对象，包含请求的URL和其他选项，`callback`是一个回调函数，用于处理响应。

```
const http = require('http');

http.get('http://www.example.com', (res) => {
  let data = '';
  res.on('data', (chunk) => {
    data += chunk;
  });
  res.on('end', () => {
    console.log(data);
  });
}).on('error', (err) => {
  console.error(err);
});
```