内存管理模拟系统

2350939 卜天

项目介绍

本项目是一个动态分区分配方式模拟系统，用于展示和比较不同的内存分配算法。系统通过可视化的方式展示了四种经典的内存分配算法在相同任务序列下的工作过程和效果，帮助用户理解各种算法的特点和适用场景。

重要声明：本项目借鉴了 GitHub 开源项目（作者：MinmusLin）的 UI 设计，但项目中算法部分包括首次适应算法（First Fit）和最佳适应算法（Best Fit）完全由作者本人重写，并自主新增了最坏适应算法（Worst Fit）和临近适应算法（Next Fit）的实现。同时新增了内存条的可视化展示，便于更直观地理解内存分配的过程。

本项目已被部署到作者本人的 Ubuntu 服务器，可通过http://124.223.93.75:6055直接访问体验。

功能特点

- 模拟四种经典内存分配算法
- 可视化展示内存分配过程
- 实时显示内存块状态和分配情况
- 支持重置操作，方便比较不同算法

算法详解

1. 首次适应算法 (First Fit)

首次适应算法按照内存块的地址顺序查找，找到第一个能满足需求的空闲块就进行分配。

算法步骤：

1. 按照内存块的顺序遍历所有空闲块
2. 找到第一个大小大于或等于请求大小的空闲块

3. 将该空闲块分配给请求进程
4. 如果分配后还有剩余空间 ’创建一个新的空闲块

特点 ：

- 实现简单 ’开销小
- 倾向于使用低地址的空闲块 ’高地址的大空闲块得以保留
- 可能导致低地址处产生大量小的空闲块 ’增加查找时间

```
// 首次适应算法
firstFit(size: number): void {
    for (let block of this.memoryBlocks) {
        if (block.isFree && block.size >= size) {
            this.allocateMemory(block, size);
            return;
        }
    }
}
```

2. 最佳适应算法 (Best Fit)

最佳适应算法查找最小的能满足需求的空闲块进行分配 ’目的是减少碎片 。

算法步骤 ：

1. 遍历所有空闲块
2. 找到大小大于或等于请求大小的最小空闲块
3. 将该空闲块分配给请求进程
4. 如果分配后还有剩余空间 ’创建一个新的空闲块

特点 ：

- 能最大限度地减少空间浪费
- 会产生更多的小碎片
- 查找过程需要遍历整个空闲块列表 ’开销较大

```
// 最佳适应算法
bestFit(size: number): void {
    let bestBlock: MemoryBlock | null = null;
    for (let block of this.memoryBlocks) {
        if (block.isFree && block.size >= size) {
            if (!bestBlock || block.size < bestBlock.size) {
                bestBlock = block;
            }
        }
    }
    if (bestBlock) {
        this.allocateMemory(bestBlock, size);
    }
}
```

3. 最坏适应算法 (Worst Fit)

最坏适应算法查找最大的空闲块进行分配 ’目的是避免产生太多小碎片 °

算法步骤 ：

1. 遍历所有空闲块
2. 找到大小大于或等于请求大小的最大空闲块
3. 将该空闲块分配给请求进程
4. 如果分配后还有剩余空间 ’创建一个新的空闲块

特点 ：

- 每次分配后剩余的空闲块尽可能大 ’更有可能被再次使用
- 大空闲块容易被迅速用完 ’可能导致大进程无法分配
- 查找过程需要遍历整个空闲块列表 ’开销较大

```
// 最坏适应算法
worstFit(size: number): void {
    let worstBlock: MemoryBlock | null = null;
    for (let block of this.memoryBlocks) {
        if (block.isFree && block.size >= size) {
            if (!worstBlock || block.size > worstBlock.size) {
                worstBlock = block;
            }
        }
    }
    if (worstBlock) {
        this.allocateMemory(worstBlock, size);
    }
}
```

4. 临近适应算法 (Next Fit)

临近适应算法是首次适应算法的变种 ’从上次分配的位置开始查找 ’而不是每次都从头开始 。

算法步骤 ：

1. 从上次分配结束的位置开始查找
2. 找到第一个大小大于或等于请求大小的空闲块
3. 将该空闲块分配给请求进程
4. 如果分配后还有剩余空间 ’创建一个新的空闲块
5. 如果查找到列表末尾仍未找到合适的块 ’则从列表头部继续查找

特点 ：

- 分配更均匀 ’避免低地址空间的过度使用
- 查找时间比首次适应算法短
- 可能会错过前面的合适空闲块

```
// 临近适应算法
nearFit(size: number): void {
    // 从上一次分配的内存块lastId开始查找
    let startIndex = this.memoryBlocks.findIndex(
        (block) => block.id > this.lastId
    );
    if (startIndex === -1) startIndex = 0; // 如果没找到比lastId大的块，从头开始

    // 先从startIndex到数组末尾查找
    for (let i = startIndex; i < this.memoryBlocks.length; i++) {
        if (
            this.memoryBlocks[i].isFree &&
            this.memoryBlocks[i].size >= size
        ) {
            this.allocateMemory(this.memoryBlocks[i], size);
            return;
        }
    }

    // 如果没找到，再从数组开头到startIndex查找
    for (let i = 0; i < startIndex; i++) {
        if (
            this.memoryBlocks[i].isFree &&
            this.memoryBlocks[i].size >= size
        ) {
            this.allocateMemory(this.memoryBlocks[i], size);
            return;
        }
    }
}
```

内存管理核心功能

除了四种分配算法外，系统还实现了以下核心功能：

内存分配

当找到合适的空闲块后，系统会将其分配给请求进程，并根据需要分割空闲块：

```
// 分配内存块
allocateMemory(block: MemoryBlock, size: number): void {
    if (block.size < size || !block.isFree) {
        console.log("内存分配失败");
        return;
    }
    if (block.size === size) {
        block.isFree = false;
        return;
    }
    const remainingSize = block.size - size;
    block.size = size;
    block.isFree = false;
    // 剩下的内存为一个新的空闲块
    this.memoryBlocks.push({
        id: ++this.lastId,
        size: remainingSize,
        isFree: true,
    });
}
```

内存释放与合并

当进程释放内存时 ’系统会将对应的内存块标记为空闲 ’并尝试合并相邻的空闲块以减少碎片 ：

```
releaseMemory(id: number): void {
    const blockIndex = this.memoryBlocks.findIndex(
        (block) => block.id === id && !block.isFree
    );
    if (blockIndex !== -1) {
        this.memoryBlocks[blockIndex].isFree = true;
        // 全局合并
        this.mergeMemory();
    }
}

// 全局合并
mergeMemory(): void {
    this.memoryBlocks.sort((a, b) => a.id - b.id);
    let blockIndex = 0;
    while (blockIndex < this.memoryBlocks.length - 1) {
        // 连续两个空闲块
        if (
            this.memoryBlocks[blockIndex].isFree &&
            this.memoryBlocks[blockIndex + 1].isFree
        ) {
            this.memoryBlocks[blockIndex].size +=
                this.memoryBlocks[blockIndex + 1].size;
            this.memoryBlocks.splice(blockIndex + 1, 1);
            continue;
        }
        blockIndex++;
    }
}
```

使用方法

1. 下载项目到本地
2. 安装依赖：`yarn install` 或 `npm install`
3. 启动开发服务器：`yarn dev` 或 `npm run dev`
4. 在浏览器中访问 `http://localhost:5173`

操作说明

1. 点击对应算法按钮开始模拟
2. 系统会按照预设的任务序列执行内存分配和释放操作
3. 实时观察内存块状态变化和内存条可视化效果
4. 点击"重置"按钮可以重新开始模拟

技术栈

- Vue 3 ：前端框架
- TypeScript ：类型安全的 JavaScript 超集
- Element Plus ：UI 组件库
- Vite ：构建工具

项目结构

```
├── src/
│   ├── App.vue          # 主应用组件
│   ├── classes.ts       # 内存管理核心类和算法
│   └── main.ts          # 应用入口
├── index.html           # HTML 模板
├── package.json         # 项目依赖和脚本
└── vite.config.ts       # Vite 配置
```