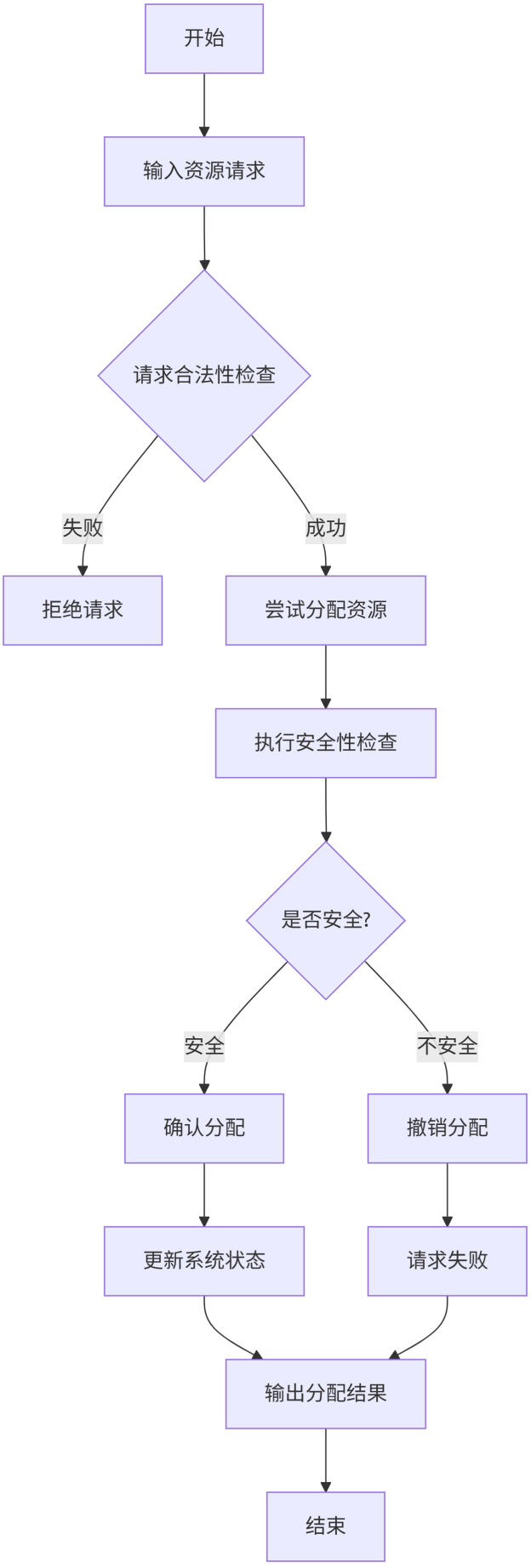
银行家算法

一、实验环境

使用go语言实现，版本为go1.24.3 windows/amd64，在windows11环境下进行实验。

二、实验原理

银行家算法实现的流程图如下：


开始
输入资源请求
请求合法性检查
失败
成功
拒绝请求
尝试分配资源
执行安全性检查
是否安全?
安全
不安全
确认分配
撤销分配
更新系统状态
请求失败
输出分配结果
结束

三、项目结构

1.系统数据结构

```
// System 系统资源状态
type System struct {
    processes  int     // 进程数量
    resources  int     // 资源种类数量
    available  []int   // 可用资源向量
    max        [][]int // 最大需求矩阵
    allocation [][]int // 分配矩阵
    need       [][]int // 需求矩阵
}
```

- **processes**: 系统中的进程数量
- **resources**: 资源种类数量
- **available**: 系统中每种资源的可用数量
- **max**: 每个进程对每种资源的最大需求量
- **allocation**: 每个进程当前已分配的各种资源数量
- **need**: 每个进程还需要的各种资源数量(`need[i][j]= max[i][j] - allocation[i][j]`)

2.初始化系统

```
// NewSystem 初始化系统
func NewSystem(processes, resources int, available []int, max, allocation [][]int) *System {
    system := &System{
        processes:  processes,
        resources:  resources,
        available:  make([]int, resources),
        max:        make([][]int, processes),
        allocation: make([][]int, processes),
        need:       make([][]int, processes),
    }

    copy(system.available, available)

    for i := 0; i < processes; i++ {
        system.max[i] = make([]int, resources)
        system.allocation[i] = make([]int, resources)
        system.need[i] = make([]int, resources)
        copy(system.max[i], max[i])
        copy(system.allocation[i], allocation[i])

        // 计算需求矩阵
        for j := 0; j < resources; j++ {
            system.need[i][j] = max[i][j] - allocation[i][j]
        }
    }
```

```
    return system
}
```

3.请求资源

```
// RequestResource 请求资源
func (s *System) RequestResource(pid int, request []int) bool {
    fmt.Printf("\n===== 进程 P%d 请求资源: %v =====\n", pid, request)

    // 步骤1: 检查请求是否超过需求
    for j := 0; j < s.resources; j++ {
        if request[j] > s.need[pid][j] {
            fmt.Printf("错误: 请求超过需求 (资源%d: 请求%d > 需求%d)\n", j, request[j],
s.need[pid][j])
            return false
        }
    }

    // 步骤2: 检查请求是否超过可用资源
    for j := 0; j < s.resources; j++ {
        if request[j] > s.available[j] {
            fmt.Printf("错误: 请求超过可用资源 (资源%d: 请求%d > 可用%d)\n", j, request[j],
s.available[j])
            return false
        }
    }

    // 步骤3: 尝试分配资源
    fmt.Println("尝试分配资源...")
    backupAvailable := make([]int, s.resources)
    backupAllocation := make([]int, s.resources)
    backupNeed := make([]int, s.resources)

    copy(backupAvailable, s.available)
    copy(backupAllocation, s.allocation[pid])
    copy(backupNeed, s.need[pid])

    for j := 0; j < s.resources; j++ {
        s.available[j] -= request[j]
        s.allocation[pid][j] += request[j]
        s.need[pid][j] -= request[j]
    }

    // 步骤4: 安全性检查
    if s.IsSafe() {
        fmt.Printf("资源分配成功! 系统处于安全状态\n")
        return true
    } else {
        // 恢复原始状态
        fmt.Println("资源分配导致不安全状态，撤销分配")
        copy(s.available, backupAvailable)
        copy(s.allocation[pid], backupAllocation)
```

```
            copy(s.need[pid], backupNeed)
            return false
        }
    }
```

4.测试样例

在main函数中mock了三个测试样例，分别模拟安全状态、不安全状态以及超过需求三种情况：

```
func main() {
    // 测试用例1: 安全分配
    fmt.Println("===== 测试用例 1: 安全分配 =====")
    system1 := NewSystem(
        5,               // 进程数
        3,               // 资源种类
        []int{3, 3, 2}, // 可用资源
        // 最大需求矩阵
        [][]int{
            {7, 5, 3},
            {3, 2, 2},
            {9, 0, 2},
            {2, 2, 2},
            {4, 3, 3},
        },
        // 已分配矩阵
        [][]int{
            {0, 1, 0},
            {2, 0, 0},
            {3, 0, 2},
            {2, 1, 1},
            {0, 0, 2},
        },
    )
    system1.PrintState()
    system1.IsSafe()

    // P1请求资源 (1,0,2)
    request1 := []int{1, 0, 2}
    system1.RequestResource(1, request1)
    system1.PrintState()

    fmt.Println("\n" + strings.Repeat("=", 50))

    // 测试用例2: 不安全请求
    fmt.Println("\n===== 测试用例 2: 不安全分配 =====")
    system2 := NewSystem(
        5,
        3,
        []int{3, 3, 2},
        [][]int{
            {7, 5, 3},
            {3, 2, 2},
```

```
            {9, 0, 2},
            {2, 2, 2},
            {4, 3, 3},
        },
        [][]int{
            {0, 1, 0},
            {2, 0, 0},
            {3, 0, 2},
            {2, 1, 1},
            {0, 0, 2},
        },
    )
    system2.PrintState()

    // P0请求资源 (0,3,0) - 会导致不安全状态
    request2 := []int{0, 3, 0}
    system2.RequestResource(0, request2)
    system2.PrintState()

    fmt.Println("\n" + strings.Repeat("=", 50))

    // 测试用例3: 超过需求
    fmt.Println("\n===== 测试用例 3: 超过最大需求 =====")
    system3 := NewSystem(
        5,
        3,
        []int{3, 3, 2},
        [][]int{
            {7, 5, 3},
            {3, 2, 2},
            {9, 0, 2},
            {2, 2, 2},
            {4, 3, 3},
        },
        [][]int{
            {0, 1, 0},
            {2, 0, 0},
            {3, 0, 2},
            {2, 1, 1},
            {0, 0, 2},
        },
    )
    system3.PrintState()

    // P1请求资源 (2,1,1) - 超过最大需求
    request3 := []int{2, 1, 1}
    system3.RequestResource(1, request3)
    system3.PrintState()
}
```

四、项目运行

切换到 lab4/src 目录下，执行`go run .`来启动项目：

```
(base) D:\CODE\THUEE-OS-lab\lab4\src [master +7 ~2 -0 !]> go run .
===== 测试用例 1: 安全分配 =====

当前系统状态:
可用资源: [3 3 2]

进程    最大需求        已分配  需求
P0      [7 5 3] [0 1 0] [7 4 3]
P1      [3 2 2] [2 0 0] [1 2 2]
P2      [9 0 2] [3 0 2] [6 0 0]
P3      [2 2 2] [2 1 1] [0 1 1]
P4      [4 3 3] [0 0 2] [4 3 1]

开始安全性检查...
初始工作向量: [3 3 2]
  - 进程 P1 可执行 (需求: [1 2 2] <= 工作向量: [3 3 2])
    进程 P1 完成, 释放资源: [2 0 0]
    更新工作向量: [5 3 2]
  - 进程 P3 可执行 (需求: [0 1 1] <= 工作向量: [5 3 2])
    进程 P3 完成, 释放资源: [2 1 1]
    更新工作向量: [7 4 3]
  - 进程 P4 可执行 (需求: [4 3 1] <= 工作向量: [7 4 3])
    进程 P4 完成, 释放资源: [0 0 2]
    更新工作向量: [7 4 5]
  - 进程 P0 可执行 (需求: [7 4 3] <= 工作向量: [7 4 5])
    进程 P0 完成, 释放资源: [0 1 0]
    更新工作向量: [7 5 5]
  - 进程 P2 可执行 (需求: [6 0 0] <= 工作向量: [7 5 5])
    进程 P2 完成, 释放资源: [3 0 2]
    更新工作向量: [10 5 7]
安全序列: P1 -> P3 -> P4 -> P0 -> P2
系统处于安全状态
```

```
===== 进程 P1 请求资源: [1 0 2] =====
尝试分配资源...

开始安全性检查...
初始工作向量: [2 3 0]
  - 进程 P1 可执行 (需求: [0 2 0] <= 工作向量: [2 3 0])
    进程 P1 完成, 释放资源: [3 0 2]
    更新工作向量: [5 3 2]
  - 进程 P3 可执行 (需求: [0 1 1] <= 工作向量: [5 3 2])
    进程 P3 完成, 释放资源: [2 1 1]
    更新工作向量: [7 4 3]
  - 进程 P4 可执行 (需求: [4 3 1] <= 工作向量: [7 4 3])
    进程 P4 完成, 释放资源: [0 0 2]
    更新工作向量: [7 4 5]
  - 进程 P0 可执行 (需求: [7 4 3] <= 工作向量: [7 4 5])
    进程 P0 完成, 释放资源: [0 1 0]
    更新工作向量: [7 5 5]
  - 进程 P2 可执行 (需求: [6 0 0] <= 工作向量: [7 5 5])
    进程 P2 完成, 释放资源: [3 0 2]
    更新工作向量: [10 5 7]
安全序列: P1 -> P3 -> P4 -> P0 -> P2
系统处于安全状态
资源分配成功! 系统处于安全状态

当前系统状态:
可用资源: [2 3 0]

进程    最大需求        已分配  需求
P0      [7 5 3] [0 1 0] [7 4 3]
P1      [3 2 2] [3 0 2] [0 2 0]
P2      [9 0 2] [3 0 2] [6 0 0]
P3      [2 2 2] [2 1 1] [0 1 1]
P4      [4 3 3] [0 0 2] [4 3 1]

================================================
```

```
===== 测试用例 2: 不安全分配 =====

当前系统状态:
可用资源: [3 3 2]

进程     最大需求        已分配   需求
P0       [7 5 3] [0 1 0] [7 4 3]
P1       [3 2 2] [2 0 0] [1 2 2]
P2       [9 0 2] [3 0 2] [6 0 0]
P3       [2 2 2] [2 1 1] [0 1 1]
P4       [4 3 3] [0 0 2] [4 3 1]

===== 进程 P0 请求资源: [0 3 0] =====
尝试分配资源...

开始安全性检查...
初始工作向量: [3 0 2]
未找到安全序列! 系统处于不安全状态
资源分配导致不安全状态, 撤销分配

当前系统状态:
可用资源: [3 3 2]

进程     最大需求        已分配   需求
P0       [7 5 3] [0 1 0] [7 4 3]
P1       [3 2 2] [2 0 0] [1 2 2]
P2       [9 0 2] [3 0 2] [6 0 0]
P3       [2 2 2] [2 1 1] [0 1 1]
P4       [4 3 3] [0 0 2] [4 3 1]

==================================================
```

```
===== 测试用例 3: 超过最大需求 =====

当前系统状态:
可用资源: [3 3 2]

进程     最大需求        已分配   需求
P0       [7 5 3] [0 1 0] [7 4 3]
P1       [3 2 2] [2 0 0] [1 2 2]
P2       [9 0 2] [3 0 2] [6 0 0]
P3       [2 2 2] [2 1 1] [0 1 1]
P4       [4 3 3] [0 0 2] [4 3 1]

===== 进程 P1 请求资源: [2 1 1] =====
错误: 请求超过需求 (资源0: 请求2 > 需求1)

当前系统状态:
可用资源: [3 3 2]

进程     最大需求        已分配   需求
P0       [7 5 3] [0 1 0] [7 4 3]
P1       [3 2 2] [2 0 0] [1 2 2]
P2       [9 0 2] [3 0 2] [6 0 0]
P3       [2 2 2] [2 1 1] [0 1 1]
P4       [4 3 3] [0 0 2] [4 3 1]
```

结果与理论结果吻合，可以验证算法正确性。

五、鲁棒性及算法效率分析

1.鲁棒性分析

本实现在鲁棒性方面有以下特点：

1. **输入验证**:
   - 检查资源请求是否超过进程声明的最大需求
   - 检查请求是否超过系统可用资源
   - 这些检查避免了非法输入导致的系统状态错误
2. **状态保护**:
   - 在尝试分配前备份系统状态
   - 如果分配导致不安全状态，能够回滚到之前的安全状态
   - 确保系统始终处于一致的安全状态
3. **边界情况处理**:
   - 能够处理资源耗尽的情况

- 能够处理进程需求为零的情况
- 能够处理特殊的进程组合和资源分配模式

2.效率分析

银行家算法的时间复杂度分析：

1. **安全性检查**:
   - 最坏情况下需要扫描所有进程 n 次，每次扫描需检查 m 种资源
   - 时间复杂度为 $O(n^2m)$，其中 n 是进程数，m 是资源种类数
2. **资源请求处理**:
   - 检查请求合法性: O(m)
   - 尝试分配资源: O(m)
   - 安全性检查: $O(n^2m)$
   - 总体时间复杂度仍为 $O(n^2m)$
3. **空间复杂度**:
   - 需存储系统状态矩阵: O(nm + m)
   - 安全性检查中的临时数组: O(n + m)
   - 总体空间复杂度为 O(nm)

六、思考题解答

1.银行家算法在实现过程中需注意资源分配的哪些事项才能避免死锁?

- **最大需求声明**：要求进程在运行前预先声明其对每类资源的最大需求量
- **完整性检查**：确保系统记录了所有进程的最大需求，不允许未声明的进程请求资源
- **合法性检查**：确保请求不超过进程声明的最大需求
- **可用性检查**：确保系统有足够的可用资源满足请求
- **安全性检查**：只在确保系统保持安全状态的前提下分配资源