银行家案例

安全性算法

安全性算法的作用范围是整个系统，它可以检查系统中是否存在安全序列，从而判断系统是否安全。

```
//安全性算法部分{
// 检查是否存在安全序列
public boolean checkSafe() {
    boolean[] finish = new boolean[n];
    int[] work = Arrays.copyOf(available, m);//把可用资源复制到work数组中
    List<Integer> safeSequence = new ArrayList<>();//建立一个安全序列链表，用于存储解法顺序
    boolean flag = true;
    while (flag) {
        flag = false;
        for (int i = 0; i < n; i++) {
            if (!finish[i] && checkNeedLess(i, work)) {//当一个进程没有使用被分配资源且当前的可用资源大于所需资源时
                for (int j = 0; j < m; j++) {
                    work[j] += allocation[i][j];//分配完后回收资源
                }
                finish[i] = true;
                safeSequence.add(i);
                flag = true;
            }
        }
    }
    return safeSequence.size() == n;//返回链表大小是否等于当前进程
}

//}
```

银行家算法

银行家算法的作用范围是单个进程，它可以检查某个进程是否能够获得所需的资源而不会发生死锁。如果进程能够获得所需的资源，则可以分配资源；否则，需要等待其他进程释放资源或者增加可用资源。

```
import java.util.*;
public class Banker {
    // 进程数量
    private int n;
    // 资源数量
    private int m;
    // 可用资源向量
```

```
    private int[] available;
    // 最大需求矩阵
    private int[][] max;
    // 已分配矩阵
    private int[][] allocation;
    // 需求矩阵
    private int[][] need;

    // 初始化银行家算法
    public Banker(int n, int m, int[] available, int[][] max, int[][]
allocation) {
        this.n = n;
        this.m = m;
        this.available = available;
        this.max = max;
        this.allocation = allocation;
        this.need = new int[n][m];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                need[i][j] = max[i][j] - allocation[i][j];
            }
        }
    }

    //安全性算法部分{
    // 检查需求是否小于等于可用资源
    private boolean checkNeedLess(int i, int[] work) {//查看当前的可用资源是否大于或等
于所需资源
        for (int j = 0; j < m; j++) {
            if (need[i][j] > work[j]) {
                return false;
            }
        }
        return true;
    }

    // 检查是否存在安全序列
    public boolean checkSafe() {
        boolean[] finish = new boolean[n];
        int[] work = Arrays.copyOf(available, m);//把可用资源复制到work数组中
        List<Integer> safeSequence = new ArrayList<>();//建立一个安全序列链表，用于存
储解法顺序
        boolean flag = true;
        while (flag) {
            flag = false;
            for (int i = 0; i < n; i++) {
                if (!finish[i] && checkNeedLess(i, work)) {//当一个进程没有使用被分配
资源且当前的可用资源大于所需资源时
                    for (int j = 0; j < m; j++) {
                        work[j] += allocation[i][j];//分配完后回收资源
                    }
                    finish[i] = true;
                    safeSequence.add(i);
                    flag = true;
                }
            }
        }
        return safeSequence.size() == n;//返回链表大小是否等于当前进程
```

```
    }

    //}
// 检查请求是否小于等于需求
    private boolean checkRequestLess(int i, int[] request) {
        for (int j = 0; j < m; j++) {
            if (request[j] > need[i][j]) {
                return false;
            }
        }
        return true;
    }

    // 检查请求是否小于等于可用资源
    private boolean checkRequestLess(int[] request) {
        for (int j = 0; j < m; j++) {
            if (request[j] > available[j]) {
                return false;
            }
        }
        return true;
    }

    // 分配资源
    public boolean allocate(int i, int[] request) {
        //如果当前请求的数量小于等于需求且当前可用资源大于或等于请求
        if (checkRequestLess(i, request) && checkRequestLess(request)) {
            for (int j = 0; j < m; j++) {
                available[j] -= request[j];//减少当前可用资源
                allocation[i][j] += request[j];//增加请求进程的当前使用资源
                need[i][j] -= request[j];//减少当前进程需要的资源
            }
            return true;
        } else {
            return false;
        }
    }


    // 释放资源
    public void release(int i, int[] release) {
        for (int j = 0; j < m; j++) {
            available[j] += release[j];
            allocation[i][j] -= release[j];
            need[i][j] += release[j];
        }
    }

    // 打印当前状态
    public void printState() {
        System.out.println("available:");
        System.out.println(Arrays.toString(available));
        System.out.println("allocation:");
        for (int i = 0; i < n; i++) {
            System.out.println(Arrays.toString(allocation[i]));
        }
        System.out.println("need:");
        for (int i = 0; i < n; i++) {
```

```
            System.out.println(Arrays.toString(need[i]));
        }
    }

    public static void main(String[] args) {
        int n = 5;
        int m = 3;
        int[] available = {3, 3, 2};
        int[][] max = {
                {6, 5, 3},
                {3, 2, 2},
                {6, 0, 4},
                {2, 2, 2},
                {4, 3, 3}
        };
        int[][] allocation = {
                {0, 1, 0},
                {2, 0, 0},
                {2, 0, 2},
                {2, 1, 1},
                {0, 0, 2}
        };
        Banker bankerAlgorithm = new Banker(n, m, available, max, allocation);
        bankerAlgorithm.printState();
        //查看是否有安全序列
        if (bankerAlgorithm.checkSafe()) {
            System.out.println("Existence safety sequence");
        } else {
            System.out.println("There is no safe sequence");
        }
        //征用资源
        int[] request = {1, 0, 2};
        int i = 2;
        System.out.println("request:"+i+ Arrays.toString(request));
        if (bankerAlgorithm.allocate(i, request)) {
            System.out.println("Allocation");
            bankerAlgorithm.printState();
        } else {
            System.out.println("no Allocation");
        }
        //释放资源
        int[] release = {3, 0, 0};
        System.out.println("Release:"+i+Arrays.toString(release));
        System.out.println("Release");
        bankerAlgorithm.printState();
    }
}
```

银行家算法中，整合了安全性算法，来首先判断是否有可能性成功解决所有进程。再进行分配给特定进程和释放特定进程的资源。

测试阶段:

这里的测试使用了在代码中就设立的max组，allocation组，和设置了request请求资源和release释放资源来测试。如果只是要解决所有的进程防止死锁，只需要跟着安全性算法中得出的序列就可以解决。

```
available:
[3, 3, 2]
allocation:
[0, 1, 0]
[2, 0, 0]
[2, 0, 2]
[2, 1, 1]
[0, 0, 2]
need:
[6, 4, 3]
[1, 2, 2]
[4, 0, 2]
[0, 1, 1]
[4, 3, 1]
Existence safety sequence
request:2[1, 0, 2]
Allocation
available:
[2, 3, 0]
allocation:
[0, 1, 0]
[2, 0, 0]
[3, 0, 4]
[2, 1, 1]
[0, 0, 2]
need:
[6, 4, 3]
[1, 2, 2]
[3, 0, 0]
[0, 1, 1]
[4, 3, 1]
Release: 2[3, 0, 0]
available:
[5, 3, 0]
allocation:
[0, 1, 0]
[2, 0, 0]
[0, 0, 4]
[2, 1, 1]
[0, 0, 2]
need:
[6, 4, 3]
[1, 2, 2]
[6, 0, 0]
[0, 1, 1]
[4, 3, 1]

进程已结束,退出代码0
```

新算法：超时机制

原理：

超时机制是避免死锁的方法，通过设置超时时间来避免进程等待资源的时间过长而导致死锁的发生。如果进程在等待资源的过程中超过了设定的超时时间，就会放弃等待并释放已经占用的资源，从而避免死锁的发生。

```
import java.util.Scanner;
import java.util.concurrent.*;

public class timeout {
    public static void main(String[] args) {
        Scanner scanner=new Scanner(System.in);
        System.out.print("choice time:");//设置一个进行时间
        int time=scanner.nextInt();
        int n =time*1000;
        ExecutorService executor = Executors.newSingleThreadExecutor();//使用了Java的Executor框架来创建一个线程池
        Future<String> future = executor.submit(new Callable<String>() {//提交了一个Callable任务
            public String call() throws Exception {
                // 模拟需要执行的任务
                Thread.sleep(n);//模拟进行n秒的操作
                return "Task completed";
            }
        });

        try {
            String result = future.get(3, TimeUnit.SECONDS);//使用Future的get方法来获取任务的执行结果，设置了超时时间为3秒
            System.out.println(result);
        } catch (TimeoutException e) {
            System.out.println("Task timed out");//如果超时
        } catch (InterruptedException e) {
            System.out.println("Task interrupted");//如果任务被中断，就会输出"Task interrupted"
        } catch (ExecutionException e) {
            System.out.println("Task failed");//任务执行失败，就会输出"Task failed
        } finally {
            future.cancel(true);//取消任务的执行，并关闭线程池
            executor.shutdown();
        }
    }
}
```

使用Java的Executor框架创建一个线程池，并提交一个Callable任务。这个任务模拟一个需要执行n秒的操作。然后，使用Future的get方法来获取任务的执行结果，同时设置了超时时间为m秒。如果任务在m秒内没有执行完成，就会抛出Timeout Exception异常。如果任务执行成功，就会输出"Task completed"；如果任务被中断，就会输出"Task interrupted"；如果任务执行失败，就会输出"Task failed"。最后，我们使用future. cancel方法来取消任务的执行，并关闭线程池。这样就可以避免任务一直等待而导致资源的浪费。

测试阶段:

`choice time:1`

`Task completed`

```
choice time:4
Task timed out

.....
```

性能对比

银行家算法和超时机制的性能差距主要来自与应用场景和实现方法。

银行家算法性能：

银行家算法是一种基于资源分配的避免死锁的方法。

优点：可以保证系统的安全性，避免死锁的发生。

缺点：银行家算法需要实时监测系统中的资源分配情况，对系统的性能牺牲较大，并且需要对资源的分配和释放进行严格的控制，可能会导致资源的浪费和效率低下。

超时机制：

超时机制是一种简单有效的设置限制来避免死锁的方法。

优点：实现简单，对系统的性能影响较小。

缺点：如果超时时间设置不合适，可能会导致任务被取消或者过早完成而浪费资源，导致对系统的性能影响不明确，可能浪费资源或者浪费系统响应时间。

PS：算法的实现代码部分参考了网上的多份资料，主要来自StackOverflow帖子。