// TDM任务依赖矩阵算法：通过邻接矩阵建模任务依赖关系，支持拓扑排序、关键路径计算和动态资源调度，提供可视化输出和DAG导出功能import java.util.\*;import java.util.stream.Collectors;public class TaskDependencyMatrix { private Map<String, Integer> taskIdMap = new HashMap<>(); //任务名称到ID映射 private List<String> taskNames = new ArrayList<>(); //任务名称列表 private int[][] dependencyMatrix; //依赖矩阵（存储最小间隔时间） private int[][] transitiveClosure; //传递闭包矩阵 private List<Integer>[] adjacencyList; //邻接表 private int taskCount =0; public TaskDependencyMatrix() { //初始化默认容量 resizeMatrix(10); } private void resizeMatrix(int newSize) { int[][] newMatrix = new int[newSize][newSize]; int[][] newClosure = new int[newSize][newSize]; //复制旧数据 if(dependencyMatrix !=null){ for(int i=0;i<Math.min(taskCount,newSize);i++){ System.arraycopy(dependencyMatrix[i],0,newMatrix[i],0,Math.min(taskCount,newSize)); System.arraycopy(transitiveClosure[i],0,newClosure[i],0,Math.min(taskCount,newSize)); } } dependencyMatrix = newMatrix; transitiveClosure = newClosure; adjacencyList = new ArrayList[newSize]; for(int i=0;i<newSize;i++){ adjacencyList[i] = new ArrayList<>(); } } public void addTask(String taskName) { if(taskIdMap.containsKey(taskName)) return; if(taskCount >= dependencyMatrix.length){ resizeMatrix(dependencyMatrix.length\*2); } taskIdMap.put(taskName, taskCount); taskNames.add(taskName); taskCount++; } public void addDependency(String fromTask, String toTask, int minInterval) { Integer fromId = taskIdMap.get(fromTask); Integer toId = taskIdMap.get(toTask); if(fromId ==null||toId ==null) throw new IllegalArgumentException("任务不存在"); dependencyMatrix[fromId][toId] = minInterval; adjacencyList[fromId].add(toId); updateTransitiveClosure(fromId, toId); } private void updateTransitiveClosure(int from, int to) { transitiveClosure[from][to] =1; //更新传递闭包 for(int i=0;i<taskCount;i++){ if(transitiveClosure[i][from]==1){ for(int j=0;j<taskCount;j++){ if(transitiveClosure[to][j]==1){ transitiveClosure[i][j] =1; } } } } } public boolean hasCircularDependency() { int[] visited = new int[taskCount]; //0:未访问 1:访问中 2:已访问 for(int i=0;i<taskCount;i++){ if(visited[i]==0 && dfsCheckCycle(i, visited)){ return true; } } return false; } private boolean dfsCheckCycle(int node, int[] visited) { visited[node] =1; for(int neighbor : adjacencyList[node]){ if(visited[neighbor]==1) return true; if(visited[neighbor]==0 && dfsCheckCycle(neighbor, visited)){ return true; } } visited[node] =2; return false; } public List<String> topologicalSort() throws IllegalStateException { if(hasCircularDependency()){ throw new IllegalStateException("存在循环依赖，无法进行拓扑排序"); } Stack<Integer> stack = new Stack<>(); boolean[] visited = new boolean[taskCount]; for(int i=0;i<taskCount;i++){ if(!visited[i]){ topologicalSortUtil(i, visited, stack); } } List<String> result = new ArrayList<>(); while(!stack.isEmpty()){ result.add(taskNames.get(stack.pop())); } return result; } private void topologicalSortUtil(int v, boolean[] visited, Stack<Integer> stack) { visited[v] =true; for(int neighbor : adjacencyList[v]){ if(!visited[neighbor]){ topologicalSortUtil(neighbor, visited, stack); } } stack.push(v); } public CriticalPathResult calculateCriticalPath() { List<String> topoOrder; try { topoOrder = topologicalSort(); } catch (IllegalStateException e) { return null; //TODO 这里应该处理循环依赖情况 } int[] earliestStart = new int[taskCount]; int[] latestStart = new int[taskCount]; Arrays.fill(latestStart, Integer.MAX\_VALUE); //正向计算最早开始时间 for(String task : topoOrder){ int taskId = taskIdMap.get(task); for(int neighbor : adjacencyList[taskId]){ if(earliestStart[neighbor] < earliestStart[taskId] + dependencyMatrix[taskId][neighbor]){ earliestStart[neighbor] = earliestStart[taskId] + dependencyMatrix[taskId][neighbor]; } } } //反向计算最晚开始时间 int totalDuration = Arrays.stream(earliestStart).max().getAsInt(); latestStart[taskIdMap.get(topoOrder.get(topoOrder.size()-1))] = totalDuration; for(int i=topoOrder.size()-1;i>=0;i--){ int taskId = taskIdMap.get(topoOrder.get(i)); for(int neighbor : adjacencyList[taskId]){ if(latestStart[taskId] > latestStart[neighbor] - dependencyMatrix[taskId][neighbor]){ latestStart[taskId] = latestStart[neighbor] - dependencyMatrix[taskId][neighbor]; } } } //识别关键路径 List<String> criticalPath = new ArrayList<>(); for(String task : topoOrder){ int taskId = taskIdMap.get(task); if(earliestStart[taskId] == latestStart[taskId]){ criticalPath.add(task); } } return new CriticalPathResult(criticalPath, totalDuration); } public void optimizeResourceAllocation() { CriticalPathResult cp = calculateCriticalPath(); if(cp ==null) return; Set<String> criticalTasks = new HashSet<>(cp.getCriticalPath()); //TODO 这里可以添加更复杂的资源优化算法 for(int i=0;i<taskCount;i++){ String task = taskNames.get(i); if(!criticalTasks.contains(task)){ //对非关键任务进行资源调整 for(int j=0;j<taskCount;j++){ if(dependencyMatrix[i][j] >0){ dependencyMatrix[i][j] = (int)(dependencyMatrix[i][j] \*0.8); //减少20%时间 } } } } } public String generateDAGDescription() { StringBuilder sb = new StringBuilder(); sb.append("digraph G {\n"); for(int i=0;i<taskCount;i++){ sb.append(" ").append(i).append(" [label=\"").append(taskNames.get(i)).append("\"];\n"); } for(int i=0;i<taskCount;i++){ for(int j=0;j<taskCount;j++){ if(dependencyMatrix[i][j] >0){ sb.append(" ").append(i).append(" -> ").append(j) .append(" [label=\"").append(dependencyMatrix[i][j]).append("\"];\n"); } } } sb.append("}"); return sb.toString(); } public void visualizeDependencyGraph() { String dot = generateDAGDescription(); //TODO 调用外部工具生成可视化图形 System.out.println("生成DOT格式的依赖图:\n"+dot); } public static class CriticalPathResult { private List<String> criticalPath; private int totalDuration; public CriticalPathResult(List<String> path, int duration) { this.criticalPath = path; this.totalDuration = duration; } public List<String> getCriticalPath() { return criticalPath; } public int getTotalDuration() { return totalDuration; } } //测试用例 public static void main(String[] args) { TaskDependencyMatrix tdm = new TaskDependencyMatrix(); tdm.addTask("需求分析"); tdm.addTask("系统设计"); tdm.addTask("数据库设计"); tdm.addTask("前端开发"); tdm.addTask("后端开发"); tdm.addTask("系统测试"); tdm.addTask("部署上线"); tdm.addDependency("需求分析", "系统设计", 2); tdm.addDependency("系统设计", "数据库设计", 1); tdm.addDependency("系统设计", "前端开发", 1); tdm.addDependency("系统设计", "后端开发", 1); tdm.addDependency("数据库设计", "后端开发", 2); tdm.addDependency("前端开发", "系统测试", 3); tdm.addDependency("后端开发", "系统测试", 4); tdm.addDependency("系统测试", "部署上线", 1); System.out.println("拓扑排序结果: "+tdm.topologicalSort()); CriticalPathResult cp = tdm.calculateCriticalPath(); System.out.println("关键路径: "+cp.getCriticalPath()+" 总时长: "+cp.getTotalDuration()); tdm.optimizeResourceAllocation(); System.out.println("优化后的DAG描述:"); tdm.visualizeDependencyGraph(); }}// DRA动态资源分配算法实现：通过实时监控资源状态和任务队列，动态计算权重并生成评分矩阵，实现智能资源分配import java.util.\*;import java.util.concurrent.\*;import java.util.stream.Collectors;public class DynamicResourceAllocator { private static final int WINDOW\_SIZE = 5; //滑动窗口大小 private volatile boolean isRunning = true; private final PriorityBlockingQueue<Task> taskQueue = new PriorityBlockingQueue<>(100, new TaskComparator()); private final Map<String, ResourceNode> resourceNodes = new ConcurrentHashMap<>(); private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2); private final Deque<AllocationRecord> historyRecords = new ConcurrentLinkedDeque<>(); //TODO 这里权重计算方式需要优化，当前版本对突发负载处理不够平滑 public void start() { scheduler.scheduleAtFixedRate(this::collectResourceMetrics, 0, 1, TimeUnit.SECONDS); scheduler.scheduleAtFixedRate(this::processAllocation, 2, 3, TimeUnit.SECONDS); } private void collectResourceMetrics() { resourceNodes.forEach((nodeId, node) -> { double cpuUsage = ResourceMonitor.getCpuUsage(nodeId); double memUsage = ResourceMonitor.getMemUsage(nodeId); node.updateMetrics(cpuUsage, memUsage); if (cpuUsage > 90 || memUsage > 85) { triggerEmergencyRelease(nodeId); //紧急释放资源 } }); } private void processAllocation() { if (taskQueue.isEmpty()) return; Map<String, Double> scoreMatrix = buildScoreMatrix(); List<Task> allocatedTasks = new ArrayList<>(); while (!taskQueue.isEmpty()) { Task currentTask = taskQueue.peek(); Optional<Map.Entry<String, Double>> optimalNode = scoreMatrix.entrySet().stream() .filter(e -> resourceNodes.get(e.getKey()).canAllocate(currentTask)) .max(Map.Entry.comparingByValue()); if (optimalNode.isPresent()) { ResourceNode node = resourceNodes.get(optimalNode.get().getKey()); Task task = taskQueue.poll(); if (task != null) { node.allocateTask(task); allocatedTasks.add(task); recordAllocation(task, node); } } else { adjustTaskPriority(currentTask); //没有合适节点时调整优先级 break; } } notifyTaskStatus(allocatedTasks); } private Map<String, Double> buildScoreMatrix() { Map<String, Double> scores = new HashMap<>(); resourceNodes.forEach((nodeId, node) -> { double healthScore = node.calculateHealthScore(); double trendScore = calculateTrendScore(node); double strategyWeight = getStrategyWeight(node); scores.put(nodeId, healthScore \* 0.6 + trendScore \* 0.3 + strategyWeight \* 0.1); }); return scores; } private double calculateTrendScore(ResourceNode node) { List<Double> cpuTrend = node.getCpuUsageHistory(); if (cpuTrend.size() < WINDOW\_SIZE) return 0.5; double sum = 0; for (int i = 1; i < WINDOW\_SIZE; i++) { sum += cpuTrend.get(i) - cpuTrend.get(i-1); } double trend = sum / (WINDOW\_SIZE-1); return trend > 0 ? 0.3 : 0.7; //下降趋势给更高分 } private void triggerEmergencyRelease(String nodeId) { ResourceNode node = resourceNodes.get(nodeId); List<Task> released = node.releaseTasks(20); //释放20%资源 released.forEach(t -> { t.increasePriority(); //提高优先级重新入队 taskQueue.offer(t); }); } private void adjustTaskPriority(Task task) { task.decreasePriority(); if (task.getRetryCount() > 3) { taskQueue.remove(task); logFailedTask(task); } } private void recordAllocation(Task task, ResourceNode node) { AllocationRecord record = new AllocationRecord( task.getId(), node.getId(), System.currentTimeMillis(), node.getCpuUsage(), node.getMemUsage(), task.getPriority() ); historyRecords.addFirst(record); if (historyRecords.size() > 1000) { historyRecords.removeLast(); } } private static class TaskComparator implements Comparator<Task> { @Override public int compare(Task t1, Task t2) { int priorityCompare = Integer.compare(t2.getPriority(), t1.getPriority()); if (priorityCompare != 0) return priorityCompare; return Long.compare(t1.getCreateTime(), t2.getCreateTime()); } } public void addResourceNode(ResourceNode node) { resourceNodes.put(node.getId(), node); } public void submitTask(Task task) { taskQueue.offer(task); } public void shutdown() { isRunning = false; scheduler.shutdown(); } //以下为内部类定义 private static class ResourceNode { private final String id; private final int totalCpu; private final long totalMem; private volatile int usedCpu = 0; private volatile long usedMem = 0; private final List<Task> runningTasks = new CopyOnWriteArrayList<>(); private final Deque<Double> cpuUsageHistory = new ConcurrentLinkedDeque<>(); private final Deque<Double> memUsageHistory = new ConcurrentLinkedDeque<>(); public ResourceNode(String id, int totalCpu, long totalMem) { this.id = id; this.totalCpu = totalCpu; this.totalMem = totalMem; } public void updateMetrics(double cpuUsage, double memUsage) { while (cpuUsageHistory.size() >= WINDOW\_SIZE) { cpuUsageHistory.removeLast(); } cpuUsageHistory.addFirst(cpuUsage); while (memUsageHistory.size() >= WINDOW\_SIZE) { memUsageHistory.removeLast(); } memUsageHistory.addFirst(memUsage); } public boolean canAllocate(Task task) { return (usedCpu + task.getCpuReq()) <= totalCpu \* 0.9 && (usedMem + task.getMemReq()) <= totalMem \* 0.9; } public void allocateTask(Task task) { runningTasks.add(task); usedCpu += task.getCpuReq(); usedMem += task.getMemReq(); } public List<Task> releaseTasks(double percent) { int releaseCount = (int) (runningTasks.size() \* percent / 100); List<Task> released = runningTasks.stream() .sorted(Comparator.comparingInt(Task::getPriority)) .limit(releaseCount) .collect(Collectors.toList()); released.forEach(t -> { runningTasks.remove(t); usedCpu -= t.getCpuReq(); usedMem -= t.getMemReq(); }); return released; } public double calculateHealthScore() { double cpuScore = 1 - (usedCpu / (double) totalCpu); double memScore = 1 - (usedMem / (double) totalMem); return (cpuScore \* 0.7 + memScore \* 0.3) \* 100; } public List<Double> getCpuUsageHistory() { return new ArrayList<>(cpuUsageHistory); } public String getId() { return id; } public int getTotalCpu() { return totalCpu; } public long getTotalMem() { return totalMem; } } private static class Task { private final String id; private final int cpuReq; private final long memReq; private final long createTime; private volatile int priority; private volatile int retryCount = 0; public Task(String id, int cpuReq, long memReq, int priority) { this.id = id; this.cpuReq = cpuReq; this.memReq = memReq; this.priority = priority; this.createTime = System.currentTimeMillis(); } public void increasePriority() { priority = Math.min(priority + 1, 10); } public void decreasePriority() { priority = Math.max(priority - 1, 1); retryCount++; } public String getId() { return id; } public int getCpuReq() { return cpuReq; } public long getMemReq() { return memReq; } public long getCreateTime() { return createTime; } public int getPriority() { return priority; } public int getRetryCount() { return retryCount; } } private static class AllocationRecord { private final String taskId; private final String nodeId; private final long timestamp; private final double cpuUsage; private final double memUsage; private final int taskPriority; public AllocationRecord(String taskId, String nodeId, long timestamp, double cpuUsage, double memUsage, int taskPriority) { this.taskId = taskId; this.nodeId = nodeId; this.timestamp = timestamp; this.cpuUsage = cpuUsage; this.memUsage = memUsage; this.taskPriority = taskPriority; } } private double getStrategyWeight(ResourceNode node) { //根据节点类型返回不同策略权重 if (node.getId().startsWith("GPU")) return 0.8; if (node.getId().startsWith("HIGHMEM")) return 0.6; return 0.5; } private void notifyTaskStatus(List<Task> tasks) { tasks.forEach(t -> System.out.println("Task allocated: " + t.getId())); } private void logFailedTask(Task task) { System.err.println("Task failed after retries: " + task.getId()); }}class ResourceMonitor { public static native double getCpuUsage(String nodeId); public static native double getMemUsage(String nodeId);}import com.fasterxml.jackson.databind.ObjectMapper;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.stereotype.Service;import org.springframework.transaction.annotation.Transactional;import org.springframework.web.client.RestTemplate;import java.util.\*;import java.util.stream.Collectors;@Servicepublic class WorkbenchService { @Autowired private RestTemplate restTemplate; private ObjectMapper objectMapper = new ObjectMapper(); //TODO 这里需要优化缓存策略，目前直接调接口性能较差 public List<WorkbenchVO> getWorkbenchList() { String url = "/api/workbench/list"; String response = restTemplate.getForObject(url, String.class); try { List<WorkbenchDTO> dtos = objectMapper.readValue(response, objectMapper.getTypeFactory().constructCollectionType(List.class, WorkbenchDTO.class)); return dtos.stream().map(this::convertToVO).collect(Collectors.toList()); } catch (Exception e) { throw new RuntimeException("获取工作台列表失败", e); } } @Transactional public void addWorkbench(WorkbenchCreateParam param) { validateCreateParam(param); String url = "/api/workbench/add"; try { String jsonParam = objectMapper.writeValueAsString(param); restTemplate.postForObject(url, jsonParam, Void.class); } catch (Exception e) { throw new RuntimeException("创建工作台失败", e); } } @Transactional public void updateWorkbench(WorkbenchUpdateParam param) { if(param.getId() ==null){ throw new IllegalArgumentException("工作台ID不能为空"); } String url = "/api/workbench/update"; try { String jsonParam = objectMapper.writeValueAsString(param); restTemplate.postForObject(url, jsonParam, Void.class); } catch (Exception e) { throw new RuntimeException("更新工作台失败", e); } } @Transactional public void deleteWorkbench(Long id) { if(id ==null){ throw new IllegalArgumentException("工作台ID不能为空"); } String url = "/api/workbench/delete?id=" + id; try { restTemplate.delete(url); } catch (Exception e) { throw new RuntimeException("删除工作台失败", e); } } public WorkbenchDetailVO getWorkbenchDetail(Long id) { String url = "/api/workbench/detail?id=" + id; String response = restTemplate.getForObject(url, String.class); try { WorkbenchDTO dto = objectMapper.readValue(response, WorkbenchDTO.class); return convertToDetailVO(dto); } catch (Exception e) { throw new RuntimeException("获取工作台详情失败", e); } } private void validateCreateParam(WorkbenchCreateParam param) { if(param.getName() ==null||param.getName().trim().isEmpty()){ throw new IllegalArgumentException("工作台名称不能为空"); } if(param.getGpuClusters() ==null||param.getGpuClusters().isEmpty()){ throw new IllegalArgumentException("GPU集群不能为空"); } if(param.getTaskQueueTypes() ==null||param.getTaskQueueTypes().isEmpty()){ throw new IllegalArgumentException("任务队列类型不能为空"); } } private WorkbenchVO convertToVO(WorkbenchDTO dto) { WorkbenchVO vo = new WorkbenchVO(); vo.setId(dto.getId()); vo.setName(dto.getName()); vo.setCreateTime(dto.getCreateTime()); vo.setStatus(dto.getStatus()); vo.setGpuClusterCount(dto.getGpuClusters().size()); vo.setTaskQueueTypeCount(dto.getTaskQueueTypes().size()); return vo; } private WorkbenchDetailVO convertToDetailVO(WorkbenchDTO dto) { WorkbenchDetailVO vo = new WorkbenchDetailVO(); vo.setId(dto.getId()); vo.setName(dto.getName()); vo.setDescription(dto.getDescription()); vo.setCreateTime(dto.getCreateTime()); vo.setUpdateTime(dto.getUpdateTime()); vo.setStatus(dto.getStatus()); vo.setGpuClusters(dto.getGpuClusters()); vo.setTaskQueueTypes(dto.getTaskQueueTypes()); vo.setMetrics(dto.getMetrics()); vo.setAlertRules(dto.getAlertRules()); vo.setRefreshInterval(dto.getRefreshInterval()); return vo; } // 获取工作台监控数据 public WorkbenchMonitorData getMonitorData(Long workbenchId) { String url = "/api/workbench/monitor-data?workbenchId=" + workbenchId; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, WorkbenchMonitorData.class); } catch (Exception e) { throw new RuntimeException("获取监控数据失败", e); } } // 获取GPU使用率数据 public List<GpuUsageData> getGpuUsageData(Long workbenchId, Date startTime, Date endTime) { String url = String.format("/api/workbench/gpu-usage?workbenchId=%d&startTime=%d&endTime=%d", workbenchId, startTime.getTime(), endTime.getTime()); String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, objectMapper.getTypeFactory().constructCollectionType(List.class, GpuUsageData.class)); } catch (Exception e) { throw new RuntimeException("获取GPU使用率数据失败", e); } } // 获取任务队列状态数据 public List<TaskQueueStatus> getTaskQueueStatus(Long workbenchId) { String url = "/api/workbench/task-queue-status?workbenchId=" + workbenchId; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, objectMapper.getTypeFactory().constructCollectionType(List.class, TaskQueueStatus.class)); } catch (Exception e) { throw new RuntimeException("获取任务队列状态失败", e); } } // 获取资源健康状态 public ResourceHealthStatus getResourceHealthStatus(Long workbenchId) { String url = "/api/workbench/resource-health?workbenchId=" + workbenchId; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, ResourceHealthStatus.class); } catch (Exception e) { throw new RuntimeException("获取资源健康状态失败", e); } } // 检查工作台名称是否已存在 public boolean checkNameExists(String name) { String url = "/api/workbench/check-name?name=" + name; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, Boolean.class); } catch (Exception e) { throw new RuntimeException("检查名称是否存在失败", e); } } // 获取所有GPU集群选项 public List<GpuClusterOption> getAllGpuClusters() { String url = "/api/workbench/all-gpu-clusters"; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, objectMapper.getTypeFactory().constructCollectionType(List.class, GpuClusterOption.class)); } catch (Exception e) { throw new RuntimeException("获取GPU集群选项失败", e); } } // 获取所有任务队列类型选项 public List<TaskQueueTypeOption> getAllTaskQueueTypes() { String url = "/api/workbench/all-task-queue-types"; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, objectMapper.getTypeFactory().constructCollectionType(List.class, TaskQueueTypeOption.class)); } catch (Exception e) { throw new RuntimeException("获取任务队列类型选项失败", e); } } // 获取所有监控指标选项 public List<MetricOption> getAllMetrics() { String url = "/api/workbench/all-metrics"; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, objectMapper.getTypeFactory().constructCollectionType(List.class, MetricOption.class)); } catch (Exception e) { throw new RuntimeException("获取监控指标选项失败", e); } } // 获取告警规则模板 public List<AlertRuleTemplate> getAlertRuleTemplates() { String url = "/api/workbench/alert-rule-templates"; String response = restTemplate.getForObject(url, String.class); try { return objectMapper.readValue(response, objectMapper.getTypeFactory().constructCollectionType(List.class, AlertRuleTemplate.class)); } catch (Exception e) { throw new RuntimeException("获取告警规则模板失败", e); } } // 获取刷新间隔选项 public List<RefreshIntervalOption> getRefreshIntervalOptions() { return Arrays.asList( new RefreshIntervalOption(5, "5秒"), new RefreshIntervalOption(10, "10秒"), new RefreshIntervalOption(30, "30秒"), new RefreshIntervalOption(60, "1分钟"), new RefreshIntervalOption(300, "5分钟"), new RefreshIntervalOption(600, "10分钟") ); }}class WorkbenchDTO { private Long id; private String name; private String description; private Date createTime; private Date updateTime; private Integer status; private List<String> gpuClusters; private List<String> taskQueueTypes; private List<String> metrics; private List<AlertRule> alertRules; private Integer refreshInterval; // getters and setters}class WorkbenchVO { private Long id; private String name; private Date createTime; private Integer status; private Integer gpuClusterCount; private Integer taskQueueTypeCount; // getters and setters}class WorkbenchDetailVO { private Long id; private String name; private String description; private Date createTime; private Date updateTime; private Integer status; private List<String> gpuClusters; private List<String> taskQueueTypes; private List<String> metrics; private List<AlertRule> alertRules; private Integer refreshInterval; // getters and setters}class WorkbenchCreateParam { private String name; private String description; private List<String> gpuClusters; private List<String> taskQueueTypes; private List<String> metrics; private List<AlertRule> alertRules; private Integer refreshInterval; // getters and setters}class WorkbenchUpdateParam { private Long id; private String name; private String description; private List<String> gpuClusters; private List<String> taskQueueTypes; private List<String> metrics; private List<AlertRule> alertRules; private Integer refreshInterval; // getters and setters}class WorkbenchMonitorData { private Long workbenchId; private Date timestamp; private List<GpuUsage> gpuUsages; private List<TaskQueueStatus> taskQueueStatuses; private ResourceHealthStatus resourceHealthStatus; // getters and setters}class GpuUsageData { private String cluster; private String node; private Date timestamp; private Double usage; // getters and setters}class TaskQueueStatus { private String queueType; private Integer waitingCount; private Integer runningCount; private Integer failedCount; private Integer completedCount; private Double avgWaitTime; // getters and setters}class ResourceHealthStatus { private Integer status; private List<HealthAlert> alerts; // getters and setters}class HealthAlert { private String resourceType; private String resourceId; private String metric; private Double value; private Double threshold; private String message; // getters and setters}class GpuClusterOption { private String id; private String name; // getters and setters}class TaskQueueTypeOption { private String id; private String name; // getters and setters}class MetricOption { private String id; private String name; private String description; // getters and setters}class AlertRuleTemplate { private String id; private String name; private String description; private String metric; private String condition; private Double threshold; private String severity; // getters and setters}class AlertRule { private String id; private String metric; private String condition; private Double threshold; private String severity; private Boolean enabled; // getters and setters}class RefreshIntervalOption { private Integer value; private String label; public RefreshIntervalOption(Integer value, String label) { this.value = value; this.label = label; } // getters and setters}import org.springframework.stereotype.Service;import org.springframework.transaction.annotation.Transactional;import org.springframework.web.multipart.MultipartFile;import javax.annotation.Resource;import java.io.IOException;import java.util.\*;import java.util.stream.Collectors;@Servicepublic class AiResourceMgmtService { @Resource private ResourceDao resourceDao; //新增计算资源 @Transactional(rollbackFor = Exception.class) public void addComputeResource(ComputeResourceDto dto) { if(dto ==null){ throw new BizException("参数不能为空"); } if(StringUtils.isBlank(dto.getResourceName())){ throw new BizException("资源名称不能为空"); } ComputeResource entity=new ComputeResource(); entity.setResourceName(dto.getResourceName().trim()); entity.setResourceType(dto.getResourceType()); entity.setSpecification(dto.getSpecification()); entity.setClusterId(dto.getClusterId()); entity.setAllocationStrategy(dto.getAllocationStrategy()); entity.setStatus(ResourceStatusEnum.IDLE.getCode()); entity.setCreateTime(new Date()); entity.setUpdateTime(new Date()); resourceDao.insert(entity); } //编辑计算资源 @Transactional(rollbackFor = Exception.class) public void updateComputeResource(ComputeResourceDto dto) { if(dto.getId()==null){ throw new BizException("ID不能为空"); } ComputeResource old=resourceDao.selectById(dto.getId()); if(old==null){ throw new BizException("资源不存在"); } if(!ResourceStatusEnum.IDLE.getCode().equals(old.getStatus()) &&!ResourceStatusEnum.MAINTENANCE.getCode().equals(old.getStatus())){ throw new BizException("只有空闲或维护中的资源才能编辑"); } old.setResourceType(dto.getResourceType()); old.setSpecification(dto.getSpecification()); old.setClusterId(dto.getClusterId()); old.setAllocationStrategy(dto.getAllocationStrategy()); old.setStatus(dto.getStatus()); old.setUpdateTime(new Date()); resourceDao.updateById(old); } //删除计算资源 @Transactional(rollbackFor = Exception.class) public void deleteComputeResource(Long id) { if(id==null){ throw new BizException("ID不能为空"); } ComputeResource resource=resourceDao.selectById(id); if(resource==null){ throw new BizException("资源不存在"); } if(ResourceStatusEnum.OCCUPIED.getCode().equals(resource.getStatus())){ throw new BizException("占用中的资源不能删除"); } resourceDao.deleteById(id); } //批量导入 @Transactional(rollbackFor = Exception.class) public void batchImport(MultipartFile file) throws IOException { if(file==null||file.isEmpty()){ throw new BizException("文件不能为空"); } List<ComputeResource> resources=new ArrayList<>(); try(InputStream is=file.getInputStream()){ Workbook workbook=WorkbookFactory.create(is); Sheet sheet=workbook.getSheetAt(0); for(int i=1;i<=sheet.getLastRowNum();i++){ Row row=sheet.getRow(i); if(row==null) continue; ComputeResource resource=new ComputeResource(); resource.setResourceName(getCellValue(row.getCell(0))); resource.setResourceType(getCellValue(row.getCell(1))); resource.setSpecification(getCellValue(row.getCell(2))); resource.setClusterId(Long.parseLong(getCellValue(row.getCell(3)))); resource.setAllocationStrategy(getCellValue(row.getCell(4))); resource.setStatus(ResourceStatusEnum.IDLE.getCode()); resource.setCreateTime(new Date()); resource.setUpdateTime(new Date()); resources.add(resource); } } if(!resources.isEmpty()){ resourceDao.batchInsert(resources); } } //批量导出 public byte[] batchExport(List<Long> ids) throws IOException { List<ComputeResource> resources; if(ids==null||ids.isEmpty()){ resources=resourceDao.selectList(null); }else{ resources=resourceDao.selectBatchIds(ids); } try(ByteArrayOutputStream out=new ByteArrayOutputStream(); Workbook workbook=new XSSFWorkbook()){ Sheet sheet=workbook.createSheet("计算资源"); //创建表头 Row header=sheet.createRow(0); header.createCell(0).setCellValue("资源名称"); header.createCell(1).setCellValue("资源类型"); header.createCell(2).setCellValue("算力规格"); header.createCell(3).setCellValue("所属集群"); header.createCell(4).setCellValue("分配策略"); header.createCell(5).setCellValue("当前状态"); //填充数据 for(int i=0;i<resources.size();i++){ ComputeResource resource=resources.get(i); Row row=sheet.createRow(i+1); row.createCell(0).setCellValue(resource.getResourceName()); row.createCell(1).setCellValue(resource.getResourceType()); row.createCell(2).setCellValue(resource.getSpecification()); row.createCell(3).setCellValue(resource.getClusterId().toString()); row.createCell(4).setCellValue(resource.getAllocationStrategy()); row.createCell(5).setCellValue(ResourceStatusEnum.getDescByCode(resource.getStatus())); } workbook.write(out); return out.toByteArray(); } } //获取单元格值 private String getCellValue(Cell cell){ if(cell==null) return ""; switch(cell.getCellType()){ case STRING:return cell.getStringCellValue().trim(); case NUMERIC: if(DateUtil.isCellDateFormatted(cell)){ return new SimpleDateFormat("yyyy-MM-dd").format(cell.getDateCellValue()); }else{ return String.valueOf((long)cell.getNumericCellValue()); } case BOOLEAN:return String.valueOf(cell.getBooleanCellValue()); case FORMULA:return cell.getCellFormula(); default:return ""; } } //TODO 这里的性能优化需要再考虑下，大数据量时可能会有问题 public List<ComputeResourceVo> queryResources(ResourceQueryDto query) { LambdaQueryWrapper<ComputeResource> wrapper=new LambdaQueryWrapper<>(); if(StringUtils.isNotBlank(query.getResourceName())){ wrapper.like(ComputeResource::getResourceName,query.getResourceName()); } if(StringUtils.isNotBlank(query.getResourceType())){ wrapper.eq(ComputeResource::getResourceType,query.getResourceType()); } if(query.getClusterId()!=null){ wrapper.eq(ComputeResource::getClusterId,query.getClusterId()); } if(StringUtils.isNotBlank(query.getStatus())){ wrapper.eq(ComputeResource::getStatus,query.getStatus()); } wrapper.orderByDesc(ComputeResource::getUpdateTime); List<ComputeResource> resources=resourceDao.selectList(wrapper); return resources.stream().map(r->{ ComputeResourceVo vo=new ComputeResourceVo(); vo.setId(r.getId()); vo.setResourceName(r.getResourceName()); vo.setResourceType(r.getResourceType()); vo.setSpecification(r.getSpecification()); vo.setClusterId(r.getClusterId()); vo.setAllocationStrategy(r.getAllocationStrategy()); vo.setStatus(r.getStatus()); vo.setStatusDesc(ResourceStatusEnum.getDescByCode(r.getStatus())); vo.setCreateTime(r.getCreateTime()); vo.setUpdateTime(r.getUpdateTime()); return vo; }).collect(Collectors.toList()); }}//枚举类@Getterenum ResourceStatusEnum { IDLE("0","空闲"), OCCUPIED("1","占用"), MAINTENANCE("2","维护中"); private String code; private String desc; ResourceStatusEnum(String code,String desc){ this.code=code; this.desc=desc; } public static String getDescByCode(String code){ for(ResourceStatusEnum e:values()){ if(e.getCode().equals(code)){ return e.getDesc(); } } return ""; }}//DTO类@Dataclass ComputeResourceDto { private Long id; private String resourceName; private String resourceType; private String specification; private Long clusterId; private String allocationStrategy; private String status;}@Dataclass ResourceQueryDto { private String resourceName; private String resourceType; private Long clusterId; private String status;}//VO类@Dataclass ComputeResourceVo { private Long id; private String resourceName; private String resourceType; private String specification; private Long clusterId; private String allocationStrategy; private String status; private String statusDesc; private Date createTime; private Date updateTime;}//实体类@Data@TableName("t\_compute\_resource")class ComputeResource { @TableId(type = IdType.AUTO) private Long id; private String resourceName; private String resourceType; private String specification; private Long clusterId; private String allocationStrategy; private String status; private Date createTime; private Date updateTime;}import java.util.\*;import java.time.LocalDateTime;import com.fasterxml.jackson.databind.ObjectMapper;import org.springframework.http.\*;import org.springframework.web.client.RestTemplate;//资源健康监控服务类public class ResourceHealthMonitorService { private RestTemplate restTemplate = new RestTemplate(); private ObjectMapper objectMapper = new ObjectMapper(); //新增资源健康监控数据 public void addResourceHealthData(ResourceHealthData data) throws Exception { //校验数据有效性 if(data == null || data.getResourceType() == null || data.getCheckTime() == null) { throw new IllegalArgumentException("资源类型和检测时间不能为空"); } //设置默认阈值 if(data.getCpuThreshold() == 0) data.setCpuThreshold(90); if(data.getMemThreshold() == 0) data.setMemThreshold(85); if(data.getGpuTempThreshold() == 0) data.setGpuTempThreshold(85); if(data.getNetworkThreshold() == 0) data.setNetworkThreshold(80); //计算健康状态 calculateHealthStatus(data); //调用API保存数据 HttpHeaders headers = new HttpHeaders(); headers.setContentType(MediaType.APPLICATION\_JSON); String jsonData = objectMapper.writeValueAsString(data); HttpEntity<String> request = new HttpEntity<>(jsonData, headers); ResponseEntity<String> response = restTemplate.postForEntity( "/api/resourceHealth/add", request, String.class); if(response.getStatusCode() != HttpStatus.OK) { throw new RuntimeException("保存资源健康数据失败: " + response.getBody()); } } //修改资源健康监控配置 public void updateResourceConfig(ResourceHealthData data) throws Exception { if(data == null || data.getId() == null) { throw new IllegalArgumentException("ID不能为空"); } //重新计算健康状态 calculateHealthStatus(data); //调用API更新数据 HttpHeaders headers = new HttpHeaders(); headers.setContentType(MediaType.APPLICATION\_JSON); String jsonData = objectMapper.writeValueAsString(data); HttpEntity<String> request = new HttpEntity<>(jsonData, headers); ResponseEntity<String> response = restTemplate.exchange( "/api/resourceHealth/update", HttpMethod.PUT, request, String.class); if(response.getStatusCode() != HttpStatus.OK) { throw new RuntimeException("更新资源健康配置失败: " + response.getBody()); } } //删除资源健康监控记录 public void deleteResourceRecord(Long id) throws Exception { if(id == null) { throw new IllegalArgumentException("ID不能为空"); } //调用API删除数据 ResponseEntity<String> response = restTemplate.exchange( "/api/resourceHealth/delete?id=" + id, HttpMethod.DELETE, null, String.class); if(response.getStatusCode() != HttpStatus.OK) { throw new RuntimeException("删除资源健康记录失败: " + response.getBody()); } } //计算资源健康状态 private void calculateHealthStatus(ResourceHealthData data) { boolean isHealthy = true; List<String> abnormalItems = new ArrayList<>(); if(data.getCpuUsage() > data.getCpuThreshold()) { isHealthy = false; abnormalItems.add("CPU使用率过高"); } if(data.getMemUsage() > data.getMemThreshold()) { isHealthy = false; abnormalItems.add("内存使用率过高"); } if(data.getGpuTemp() > data.getGpuTempThreshold()) { isHealthy = false; abnormalItems.add("GPU温度过高"); } if(data.getNetworkUsage() > data.getNetworkThreshold()) { isHealthy = false; abnormalItems.add("网络带宽占用过高"); } data.setHealthy(isHealthy); data.setAbnormalItems(abnormalItems); data.setLastCheckTime(LocalDateTime.now()); } //获取资源健康历史数据 public List<ResourceHealthData> getHistoryData(String resourceType, Date startTime, Date endTime) throws Exception { if(resourceType == null) { throw new IllegalArgumentException("资源类型不能为空"); } //构建查询参数 String url = "/api/resourceHealth/history?resourceType=" + resourceType; if(startTime != null) { url += "&startTime=" + startTime.getTime(); } if(endTime != null) { url += "&endTime=" + endTime.getTime(); } //调用API获取数据 ResponseEntity<String> response = restTemplate.getForEntity(url, String.class); if(response.getStatusCode() != HttpStatus.OK) { throw new RuntimeException("获取历史数据失败: " + response.getBody()); } return objectMapper.readValue(response.getBody(), objectMapper.getTypeFactory().constructCollectionType(List.class, ResourceHealthData.class)); } //获取告警阈值配置 public ThresholdConfig getThresholdConfig(String resourceType) throws Exception { if(resourceType == null) { throw new IllegalArgumentException("资源类型不能为空"); } //调用API获取配置 ResponseEntity<String> response = restTemplate.getForEntity( "/api/resourceHealth/threshold?resourceType=" + resourceType, String.class); if(response.getStatusCode() != HttpStatus.OK) { throw new RuntimeException("获取阈值配置失败: " + response.getBody()); } return objectMapper.readValue(response.getBody(), ThresholdConfig.class); } //资源健康数据实体类 public static class ResourceHealthData { private Long id; private String resourceType; //资源类型：CPU/MEM/GPU/NETWORK private LocalDateTime checkTime; //检测时间 private Double cpuUsage; //CPU使用率 private Integer cpuThreshold; //CPU告警阈值 private Double memUsage; //内存使用率 private Integer memThreshold; //内存告警阈值 private Double gpuTemp; //GPU温度 private Integer gpuTempThreshold; //GPU温度阈值 private Double networkUsage; //网络带宽使用率 private Integer networkThreshold; //网络带宽阈值 private Boolean healthy; //是否健康 private List<String> abnormalItems; //异常项 private String remark; //备注 private LocalDateTime lastCheckTime; //最后检测时间 //getters and setters public Long getId() { return id; } public void setId(Long id) { this.id = id; } public String getResourceType() { return resourceType; } public void setResourceType(String resourceType) { this.resourceType = resourceType; } public LocalDateTime getCheckTime() { return checkTime; } public void setCheckTime(LocalDateTime checkTime) { this.checkTime = checkTime; } public Double getCpuUsage() { return cpuUsage; } public void setCpuUsage(Double cpuUsage) { this.cpuUsage = cpuUsage; } public Integer getCpuThreshold() { return cpuThreshold; } public void setCpuThreshold(Integer cpuThreshold) { this.cpuThreshold = cpuThreshold; } public Double getMemUsage() { return memUsage; } public void setMemUsage(Double memUsage) { this.memUsage = memUsage; } public Integer getMemThreshold() { return memThreshold; } public void setMemThreshold(Integer memThreshold) { this.memThreshold = memThreshold; } public Double getGpuTemp() { return gpuTemp; } public void setGpuTemp(Double gpuTemp) { this.gpuTemp = gpuTemp; } public Integer getGpuTempThreshold() { return gpuTempThreshold; } public void setGpuTempThreshold(Integer gpuTempThreshold) { this.gpuTempThreshold = gpuTempThreshold; } public Double getNetworkUsage() { return networkUsage; } public void setNetworkUsage(Double networkUsage) { this.networkUsage = networkUsage; } public Integer getNetworkThreshold() { return networkThreshold; } public void setNetworkThreshold(Integer networkThreshold) { this.networkThreshold = networkThreshold; } public Boolean getHealthy() { return healthy; } public void setHealthy(Boolean healthy) { this.healthy = healthy; } public List<String> getAbnormalItems() { return abnormalItems; } public void setAbnormalItems(List<String> abnormalItems) { this.abnormalItems = abnormalItems; } public String getRemark() { return remark; } public void setRemark(String remark) { this.remark = remark; } public LocalDateTime getLastCheckTime() { return lastCheckTime; } public void setLastCheckTime(LocalDateTime lastCheckTime) { this.lastCheckTime = lastCheckTime; } } //阈值配置实体类 public static class ThresholdConfig { private String resourceType; private Integer cpuThreshold; private Integer memThreshold; private Integer gpuTempThreshold; private Integer networkThreshold; private LocalDateTime updateTime; //getters and setters public String getResourceType() { return resourceType; } public void setResourceType(String resourceType) { this.resourceType = resourceType; } public Integer getCpuThreshold() { return cpuThreshold; } public void setCpuThreshold(Integer cpuThreshold) { this.cpuThreshold = cpuThreshold; } public Integer getMemThreshold() { return memThreshold; } public void setMemThreshold(Integer memThreshold) { this.memThreshold = memThreshold; } public Integer getGpuTempThreshold() { return gpuTempThreshold; } public void setGpuTempThreshold(Integer gpuTempThreshold) { this.gpuTempThreshold = gpuTempThreshold; } public Integer getNetworkThreshold() { return networkThreshold; } public void setNetworkThreshold(Integer networkThreshold) { this.networkThreshold = networkThreshold; } public LocalDateTime getUpdateTime() { return updateTime; } public void setUpdateTime(LocalDateTime updateTime) { this.updateTime = updateTime; } }}import java.util.\*;import java.util.concurrent.\*;import java.util.stream.Collectors;import org.springframework.stereotype.Service;import org.springframework.web.client.RestTemplate;import com.fasterxml.jackson.databind.ObjectMapper;@Servicepublic class TaskQueueService { // 任务队列，按优先级排序 private PriorityBlockingQueue<AiTask> taskQueue = new PriorityBlockingQueue<>(100, Comparator.comparingInt(AiTask::getPriority).reversed() .thenComparing(AiTask::getSubmitTime)); // 资源规格缓存 private Map<String, ResourceSpec> resourceSpecCache = new ConcurrentHashMap<>(); //TODO 这里需要优化资源释放逻辑，当前实现可能有并发问题 private final RestTemplate restTemplate = new RestTemplate(); private final ObjectMapper objectMapper = new ObjectMapper(); /\*\* \* 新增AI任务 \* @param taskType 任务类型 \* @param priority 优先级 \* @param specId 资源规格ID \* @return 任务ID \*/ public String addAiTask(String taskType, int priority, String specId) { if (priority < 1 || priority > 10) { throw new IllegalArgumentException("优先级必须在1-10之间"); } // 获取资源规格 ResourceSpec spec = fetchResourceSpec(specId); if (spec == null) { throw new RuntimeException("无效的资源规格ID"); } String taskId = generateTaskId(taskType); AiTask task = new AiTask(taskId, taskType, priority, spec, new Date()); // 加入队列 taskQueue.put(task); // 异步通知调度器 CompletableFuture.runAsync(() -> notifyScheduler(task)); return taskId; } private ResourceSpec fetchResourceSpec(String specId) { // 先从缓存获取 if (resourceSpecCache.containsKey(specId)) { return resourceSpecCache.get(specId); } // 调用API获取 try { String response = restTemplate.getForObject("/api/resource/spec/" + specId, String.class); ResourceSpec spec = objectMapper.readValue(response, ResourceSpec.class); resourceSpecCache.put(specId, spec); return spec; } catch (Exception e) { throw new RuntimeException("获取资源规格失败", e); } } private String generateTaskId(String taskType) { long timestamp = System.currentTimeMillis(); int random =new Random().nextInt(900)+ 100; return taskType.substring(0, 3).toUpperCase() + timestamp + random; } private void notifyScheduler(AiTask task) { try { restTemplate.postForObject("/api/scheduler/notify", task, Void.class); } catch (Exception e) { // 记录日志但继续执行 System.err.println("通知调度器失败: " + e.getMessage()); } } /\*\* \* 调整任务优先级 \* @param taskId 任务ID \* @param newPriority 新优先级 \*/ public void adjustPriority(String taskId, int newPriority) { if (newPriority < 1 || newPriority > 10) { throw new IllegalArgumentException("优先级必须在1-10之间"); } // 查找任务 AiTask task = findTaskById(taskId); if (task == null) { throw new RuntimeException("任务不存在"); } // 创建新任务对象并更新优先级 AiTask newTask = new AiTask( task.getTaskId(), task.getTaskType(), newPriority, task.getResourceSpec(), task.getSubmitTime() ); // 先移除旧任务 taskQueue.remove(task); // 添加新任务 taskQueue.put(newTask); // 更新数据库 updateTaskInDB(newTask); } private AiTask findTaskById(String taskId) { return taskQueue.stream() .filter(t -> t.getTaskId().equals(taskId)) .findFirst() .orElse(null); } private void updateTaskInDB(AiTask task) { try { restTemplate.put("/api/tasks/" + task.getTaskId(), task); } catch (Exception e) { throw new RuntimeException("更新任务失败", e); } } /\*\* \* 删除任务 \* @param taskId 任务ID \*/ public void deleteTask(String taskId) { AiTask task = findTaskById(taskId); if (task == null) { throw new RuntimeException("任务不存在"); } // 从队列移除 taskQueue.remove(task); // 释放资源 releaseResources(task); // 删除数据库记录 deleteTaskFromDB(taskId); } private void releaseResources(AiTask task) { try { restTemplate.postForObject( "/api/resources/release", task.getResourceSpec(), Void.class); } catch (Exception e) { throw new RuntimeException("释放资源失败", e); } } private void deleteTaskFromDB(String taskId) { try { restTemplate.delete("/api/tasks/" + taskId); } catch (Exception e) { throw new RuntimeException("删除任务记录失败", e); } } /\*\* \* 获取当前队列状态 \* @return 按优先级排序的任务列表 \*/ public List<AiTask> getQueueStatus() { return new ArrayList<>(taskQueue); } // 内部类定义 public static class AiTask { private String taskId; private String taskType; private int priority; private ResourceSpec resourceSpec; private Date submitTime; public AiTask(String taskId, String taskType, int priority, ResourceSpec resourceSpec, Date submitTime) { this.taskId = taskId; this.taskType = taskType; this.priority = priority; this.resourceSpec = resourceSpec; this.submitTime = submitTime; } // getters public String getTaskId() { return taskId; } public String getTaskType() { return taskType; } public int getPriority() { return priority; } public ResourceSpec getResourceSpec() { return resourceSpec; } public Date getSubmitTime() { return submitTime; } } public static class ResourceSpec { private String specId; private String gpuModel; private int cpuCores; private int memoryGB; public ResourceSpec(String specId, String gpuModel, int cpuCores, int memoryGB) { this.specId = specId; this.gpuModel = gpuModel; this.cpuCores = cpuCores; this.memoryGB = memoryGB; } // getters public String getSpecId() { return specId; } public String getGpuModel() { return gpuModel; } public int getCpuCores() { return cpuCores; } public int getMemoryGB() { return memoryGB; } }}import com.fasterxml.jackson.databind.ObjectMapper;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.stereotype.Service;import org.springframework.transaction.annotation.Transactional;import org.springframework.util.StringUtils;import javax.servlet.http.HttpServletRequest;import java.util.\*;import java.util.stream.Collectors;@Servicepublic class TaskDependencyService { @Autowired private ObjectMapper objectMapper; //TODO 这里循环依赖检测算法需要优化，当数据量大时性能较差 public Map<String, Object> addTaskDependency(HttpServletRequest request) throws Exception { Map<String, String> params = parseRequestParams(request); validateAddParams(params); // 检查是否已存在相同依赖 Map<String, Object> existCheck = checkExistDependency(params.get("preTaskId"), params.get("nextTaskId")); if ((boolean) existCheck.get("exists")) { throw new RuntimeException("该依赖关系已存在"); } // 检查循环依赖 if (checkCircularDependency(params.get("preTaskId"), params.get("nextTaskId"))) { throw new RuntimeException("检测到循环依赖，请检查任务依赖关系"); color: #1890ff;}.ant-btn-link:hover { text-decoration: underline;}.ant-btn-danger { color: #ff4d4f;}import React, { useState, useEffect } from 'react';import './ResourceAllocation.css';const ResourceAllocation = () => { const [statsData, setStatsData] = useState({ todayAllocations: 0, avgDuration: 0, peakUsage: 0, completionRate: 0, yesterdayComparison: { allocations: 0, duration: 0, usage: 0, completion: 0 } }); const [searchParams, setSearchParams] = useState({ allocationId: '', resourceId: '' }); const [records, setRecords] = useState([]); const [loading, setLoading] = useState(true); const [currentPage, setCurrentPage] = useState(1); // 获取概览数据 const fetchOverviewData = async () => { try { const response = await fetch('/api/resource/overview'); const data = await response.json(); setStatsData({ todayAllocations: data.todayAllocations, avgDuration: data.avgDuration, peakUsage: data.peakUsage, completionRate: data.completionRate, yesterdayComparison: data.yesterdayComparison }); } catch (error) { console.error('获取概览数据失败:', error); } }; // 获取记录数据 const fetchRecords = async () => { setLoading(true); try { const query = new URLSearchParams({ ...searchParams, page: currentPage }).toString(); const response = await fetch(`/api/resource/records?${query}`); const data = await response.json(); setRecords(data.records); } catch (error) { console.error('获取记录失败:', error); } finally { setLoading(false); } }; useEffect(() => { fetchOverviewData(); fetchRecords(); }, [currentPage]); const handleSearch = () => { setCurrentPage(1); fetchRecords(); }; const handleReset = () => { setSearchParams({ allocationId: '', resourceId: '' }); setCurrentPage(1); }; const handleInputChange = (e) => { const { name, value } = e.target; setSearchParams(prev => ({ ...prev, [name]: value })); }; const handleDelete = async (id) => { try { await fetch(`/api/resource/${id}`, { method: 'DELETE' }); fetchRecords(); } catch (error) { console.error('删除失败:', error); } }; const formatDuration = (minutes) => { const hours = Math.floor(minutes / 60); const mins = minutes % 60; return hours > 0 ? `${hours}小时${mins}分钟` : `${mins}分钟`; }; const getComparisonText = (current, yesterday, isPercentage = false) => { const diff = current - yesterday; const percentage = (diff / yesterday \* 100).toFixed(1); const sign = diff >= 0 ? '+' : ''; const value = isPercentage ? `${sign}${percentage}%` : `${sign}${diff}`; return `同比昨日 ${value}`; }; return ( <div className="resource-container"> {/\* 数据概览区 \*/} <div className="stats-cards"> <div className="card pink-bg"> <div className="card-icon"> <i className="icon-bell"></i> </div> <div className="card-value">{statsData.todayAllocations}</div> <div className="card-label">今日资源分配次数</div> <div className="card-comparison"> {getComparisonText(statsData.todayAllocations, statsData.yesterdayComparison.allocations)} </div> </div> <div className="card orange-bg"> <div className="card-icon"> <i className="icon-hourglass"></i> </div> <div className="card-value">{statsData.avgDuration.toFixed(1)}小时</div> <div className="card-label">平均资源使用时长</div> <div className="card-comparison"> {getComparisonText(statsData.avgDuration, statsData.yesterdayComparison.duration)} </div> </div> <div className="card green-bg"> <div className="card-icon"> <i className="icon-server"></i> </div> <div className="card-value">{statsData.peakUsage}%</div> <div className="card-label">资源使用率峰值</div> <div className="card-comparison"> {getComparisonText(statsData.peakUsage, statsData.yesterdayComparison.usage, true)} </div> </div> <div className="card purple-bg"> <div className="card-icon"> <i className="icon-circle"></i> </div> <div className="card-value">{statsData.completionRate}%</div> <div className="card-label">任务完成率</div> <div className="card-comparison"> {getComparisonText(statsData.completionRate, statsData.yesterdayComparison.completion, true)} </div> </div> </div> {/\* 搜索筛选区 \*/} <div className="search-area"> <button className="btn btn-primary">新增</button> <div className="search-inputs"> <input type="text" name="allocationId" value={searchParams.allocationId} onChange={handleInputChange} placeholder="分配ID：请输入" /> <input type="text" name="resourceId" value={searchParams.resourceId} onChange={handleInputChange} placeholder="资源ID：请输入" /> </div> <div className="search-buttons"> <button className="btn btn-primary" onClick={handleSearch}>搜索</button> <button className="btn btn-reset" onClick={handleReset}>重置</button> </div> </div> {/\* 记录表格 \*/} <div className="records-table"> {loading ? ( <div className="loading">加载中...</div> ) : ( <> <table> <thead> <tr> <th>分配ID</th> <th>资源ID</th> <th>任务ID</th> <th>分配开始时间</th> <th>释放时间</th> <th>使用时长</th> <th>使用率</th> <th>分配状态</th> <th>资源类型</th> <th>任务类型</th> <th>操作</th> </tr> </thead> <tbody> {records.map(record => ( <tr key={record.allocationId}> <td>{record.allocationId}</td> <td>{record.resourceId}</td> <td>{record.taskId}</td> <td>{record.startTime}</td> <td>{record.endTime || '-'}</td> <td>{formatDuration(record.duration)}</td> <td>{record.usageRate}%</td> <td>{record.status}</td> <td>{record.resourceType}</td> <td>{record.taskType}</td> <td> <button className="btn-edit">修改</button> <button className="btn-delete" onClick={() => handleDelete(record.allocationId)} > 删除 </button> </td> </tr> ))} </tbody> </table> <div className="table-footer"> <div className="total-count">共{records.length}项</div> <div className="pagination"> <span className={currentPage === 1 ? 'active' : ''}>1</span> </div> </div> </> )} </div> </div> );};export default ResourceAllocation;/\* ResourceAllocation.css \*/.resource-container { padding: 20px; font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;}.stats-cards { display: flex; gap: 15px; margin-bottom: 20px;}.card { flex: 1; padding: 15px; border-radius: 8px; box-shadow: 0 2px 8px rgba(0,0,0,0.1); position: relative; overflow: hidden;}.card-icon { font-size: 24px; margin-bottom: 10px;}.card-value { font-size: 28px; font-weight: bold; margin: 5px 0;}.card-label { font-size: 14px; color: #666; margin-bottom: 5px;}.card-comparison { font-size: 12px; color: #888;}.pink-bg { background: linear-gradient(135deg, rgba(255,182,193,0.8), rgba(255,192,203,0.6));}.pink-bg .card-icon { color: #ff4757;}.orange-bg { background: linear-gradient(135deg, rgba(255,165,0,0.8), rgba(255,140,0,0.6));}.orange-bg .card-icon { color: #ff7f50;}.green-bg { background: linear-gradient(135deg, rgba(144,238,144,0.8), rgba(152,251,152,0.6));}.green-bg .card-icon { color: #2ed573;}.purple-bg { background: linear-gradient(135deg, rgba(147,112,219,0.8), rgba(138,43,226,0.6));}.purple-bg .card-icon { color: #7d5fff;}.search-area { display: flex; gap: 10px; margin-bottom: 20px; align-items: center;}.search-inputs { display: flex; gap: 10px;}.search-inputs input { padding: 8px 12px; border: 1px solid #ddd; border-radius: 4px; width: 200px;}.search-buttons { display: flex; gap: 5px;}.btn { padding: 8px 16px; border-radius: 4px; cursor: pointer; border: none; font-weight: bold;}.btn-primary { background-color: #1E90FF; color: white;}.btn-reset { background-color: white; border: 1px solid #ddd; color: #666;}.records-table { background: white; border-radius: 8px; box-shadow: 0 2px 8px rgba(0,0,0,0.1); overflow: hidden;}table { width: 100%; border-collapse: collapse;}th, td { padding: 12px 15px; text-align: left; border-bottom: 1px solid #eee;}th { font-weight: bold; color: #333; background: white;}tbody tr:hover { background-color: #f5f5f5;}.btn-edit { color: #1E90FF; background: none; border: none; cursor: pointer; margin-right: 5px;}.btn-delete { color: #ff4757; background: none; border: none; cursor: pointer;}.table-footer { display: flex; justify-content: space-between; padding: 15px; align-items: center;}.total-count { color: #666; font-size: 14px;}.pagination { display: flex; gap: 5px;}.pagination span { padding: 5px 10px; border-radius: 4px; cursor: pointer;}.pagination .active { border: 1px solid #1E90FF; color: #1E90FF;}.loading { padding: 20px; text-align: center; color: #666;}import React, { useState, useEffect } from 'react';import { Table, Input, Button, Space, Tag, message, Popconfirm } from 'antd';import { SearchOutlined, PlusOutlined, EditOutlined, DeleteOutlined, FilterOutlined, ExportOutlined, SyncOutlined, UnorderedListOutlined, EllipsisOutlined } from '@ant-design/icons';import './ResourcePolicy.css'; // 样式文件const ResourcePolicy = () => { const [data, setData] = useState([]); const [loading, setLoading] = useState(false); const [selectedRowKeys, setSelectedRowKeys] = useState([]); const [searchText, setSearchText] = useState(''); const [pagination, setPagination] = useState({ current: 1, pageSize: 10, total: 0, }); // 获取策略数据 const fetchData = async (params = {}) => { setLoading(true); try { const response = await fetch('/api/resource-policy/list', { method: 'POST', headers: { 'Content-Type': 'application/json', }, body: JSON.stringify({ page: params.current || pagination.current, pageSize: params.pageSize || pagination.pageSize, search: searchText, ...params, }), }); const result = await response.json(); if (result.code === 200) { setData(result.data.list.map((item, index) => ({ ...item, key: item.id, no: index + 1, }))); setPagination({ ...pagination, total: result.data.total, current: params.current || pagination.current, pageSize: params.pageSize || pagination.pageSize, }); } else { message.error(result.message); } } catch (error) { message.error('获取数据失败'); } finally { setLoading(false); } }; useEffect(() => { fetchData(); }, [searchText]); // 处理表格变化 const handleTableChange = (pagination, filters, sorter) => { fetchData({ current: pagination.current, pageSize: pagination.pageSize, sortField: sorter.field, sortOrder: sorter.order, ...filters, }); }; // 删除策略 const handleDelete = async () => { if (selectedRowKeys.length === 0) { message.warning('请选择要删除的策略'); return; } try { const response = await fetch('/api/resource-policy/delete', { method: 'POST', headers: { 'Content-Type': 'application/json', }, body: JSON.stringify({ ids: selectedRowKeys, }), }); const result = await response.json(); if (result.code === 200) { message.success('删除成功'); fetchData(); setSelectedRowKeys([]); } else { message.error(result.message); } } catch (error) { message.error('删除失败'); } }; // 表格列配置 const columns = [ { title: 'No.', dataIndex: 'no', width: 40, align: 'center', }, { title: '策略ID', dataIndex: 'policyId', width: 100, sorter: true, }, { title: '策略名称', dataIndex: 'policyName', width: 150, }, { title: '策略描述', dataIndex: 'policyDesc', width: 180, }, { title: '适用资源类型', dataIndex: 'resourceType', width: 120, render: (text) => { const types = text.split('/'); return types.map((type) => ( <Tag color={type === 'GPU' ? 'blue' : 'orange'} key={type}> {type} </Tag> )); }, }, { title: '优先级规则', dataIndex: 'priorityRule', width: 120, }, { title: '资源回收规则', dataIndex: 'recycleRule', width: 140, }, { title: '策略状态', dataIndex: 'status', width: 80, render: (text) => ( <Tag color={text === '启用' ? 'blue' : 'default'}>{text}</Tag> ), }, { title: '创建时间', dataIndex: 'createTime', width: 140, }, { title: '更新时间', dataIndex: 'updateTime', width: 140, }, ]; // 表格行选择 const rowSelection = { selectedRowKeys, onChange: (selectedKeys) => { setSelectedRowKeys(selectedKeys); }, }; return ( <div className="resource-policy-container"> <div className="toolbar"> <Space> <Input placeholder="查询" prefix={<SearchOutlined />} value={searchText} onChange={(e) => setSearchText(e.target.value)} style={{ width: 200 }} /> <Button type="primary" icon={<PlusOutlined />} onClick={() => { // TODO: 打开新增策略弹窗 }} > 新增 </Button> <Button icon={<EditOutlined />} disabled={selectedRowKeys.length !== 1} onClick={() => { // TODO: 打开编辑策略弹窗 }} > 修改 </Button> <Popconfirm title="确定要删除选中的策略吗？" onConfirm={handleDelete} okText="确定" cancelText="取消" > <Button icon={<DeleteOutlined />} disabled={selectedRowKeys.length === 0} > 删除 </Button> </Popconfirm> </Space> <Space className="toolbar-right"> <Button icon={<FilterOutlined />} /> <Button icon={<ExportOutlined />} /> <Button icon={<SyncOutlined />} onClick={() => fetchData()} /> <Button icon={<UnorderedListOutlined />} /> <Button icon={<EllipsisOutlined />} /> </Space> </div> <Table columns={columns} dataSource={data} rowSelection={rowSelection} loading={loading} pagination={pagination} onChange={handleTableChange} scroll={{ x: 1200 }} bordered /> </div> );};export default ResourcePolicy;/\* ResourcePolicy.css \*/.resource-policy-container { padding: 16px; background: #fff;}.toolbar { display: flex; justify-content: space-between; margin-bottom: 16px;}.toolbar-right { margin-left: auto;}.ant-table-thead > tr > th { background-color: #f0f7ff !important; font-weight: bold; color: #333;}.ant-table-tbody > tr > td { border-bottom: 1px solid #f0f0f0;}.ant-table-pagination { margin: 16px 0; float: right;}/\* 按钮样式覆盖 \*/.ant-btn-primary { background-color: #1890ff; border-color: #1890ff;}.ant-btn-default { border-color: #d9d9d9; color: rgba(0, 0, 0, 0.65);}.ant-btn[disabled] { color: rgba(0, 0, 0, 0.25); background: #f5f5f5; border-color: #d9d9d9;}/\* 表格行高 \*/.ant-table-tbody > tr > td { padding: 12px 8px; height: 36px;}/\* 状态标签样式 \*/.ant-tag { margin-right: 0;}/\* 工具栏按钮间距 \*/.toolbar .ant-space-item { margin-right: 8px;}import React, { useState, useEffect } from 'react';import { Table, Input, Button, Card, Calendar, Select, Space, Divider } from 'antd';import { FilterOutlined, DownloadOutlined, SyncOutlined, UnorderedListOutlined, DownOutlined, MoreOutlined } from '@ant-design/icons';import './ResourceReservation.css'; // 样式文件const { Search } = Input;const { Option } = Select;// 统计卡片组件const StatCard = ({ color, value, label, icon }) => { return ( <Card style={{ backgroundColor: color, borderRadius: 8, color: 'white', flex: 1, marginRight: 16, minWidth: 180 }} > <div style={{ display: 'flex', justifyContent: 'space-between' }}> <div> <div style={{ fontSize: 24, fontWeight: 'bold' }}>{value}</div> <div style={{ fontSize: 12, color: 'rgba(255,255,255,0.7)' }}>{label}</div> </div> <div style={{ fontSize: 24 }}>{icon}</div> </div> </Card> );};// 资源预留管理主组件const ResourceReservation = () => { const [searchKey, setSearchKey] = useState(''); const [currentPage, setCurrentPage] = useState(1); const [selectedRowKeys, setSelectedRowKeys] = useState([]); const [calendarMode, setCalendarMode] = useState('month'); const [currentDate, setCurrentDate] = useState(new Date()); const [reservations, setReservations] = useState([]); const [stats, setStats] = useState({ applied: 0, approved: 0, released: 0, conflicts: 0 }); // 获取统计数据 const fetchStats = async () => { try { const response = await fetch('/api/reservation/stats'); const data = await response.json(); setStats({ applied: data.applied, approved: data.approved, released: data.released, conflicts: data.conflicts }); } catch (error) { console.error('获取统计数据失败:', error); } }; // 获取资源预留列表 const fetchReservations = async () => { try { const response = await fetch(`/api/reservation/list?page=${currentPage}&search=${searchKey}`); const data = await response.json(); setReservations(data.list); } catch (error) { console.error('获取资源预留列表失败:', error); } }; // 初始化数据 useEffect(() => { fetchStats(); fetchReservations(); }, [currentPage, searchKey]); // 表格列定义 const columns = [ { title: 'No.', dataIndex: 'index', key: 'index', width: 60, render: (\_, \_\_, index) => (currentPage - 1) \* 15 + index + 1 }, { title: '预留ID', dataIndex: 'reservationId', key: 'reservationId', sorter: (a, b) => a.reservationId.localeCompare(b.reservationId) }, { title: '资源ID', dataIndex: 'resourceId', key: 'resourceId', sorter: (a, b) => a.resourceId.localeCompare(b.resourceId) }, { title: '预留用途', dataIndex: 'purpose', key: 'purpose', sorter: (a, b) => a.purpose.localeCompare(b.purpose) }, { title: '预留开始时间', dataIndex: 'startTime', key: 'startTime', render: text => new Date(text).toLocaleString() }, { title: '预留结束时间', dataIndex: 'endTime', key: 'endTime', render: text => new Date(text).toLocaleString() } ]; // 处理表格选择 const onSelectChange = selectedKeys => { setSelectedRowKeys(selectedKeys); }; // 处理搜索 const handleSearch = value => { setSearchKey(value); setCurrentPage(1); }; // 处理日历视图切换 const handleCalendarModeChange = value => { setCalendarMode(value); }; // 处理日历导航 const handleCalendarNavigate = type => { const newDate = new Date(currentDate); if (type === 'prev') { newDate.setMonth(newDate.getMonth() - 1); } else if (type === 'next') { newDate.setMonth(newDate.getMonth() + 1); } else { newDate = new Date(); } setCurrentDate(newDate); }; // 渲染日历单元格 const dateCellRender = value => { const dateStr = value.toISOString().split('T')[0]; const dayReservations = reservations.filter( item => new Date(item.startTime).toISOString().split('T')[0] === dateStr ); if (dayReservations.length > 0) { return ( <div className="calendar-cell"> {dayReservations.length > 3 ? ( <div className="more-tip">{dayReservations.length - 3} more</div> ) : null} </div> ); } return null; }; // 处理表格分页 const handleTableChange = (pagination, filters, sorter) => { setCurrentPage(pagination.current); }; return ( <div className="resource-reservation-container"> {/\* 顶部统计卡片区域 \*/} <div className="stats-container"> <StatCard color="#1E88E5" value={stats.applied} label="已申请预留" icon={<UnorderedListOutlined />} /> <StatCard color="#4CAF50" value={stats.approved} label="已批准预留" icon={<FilterOutlined />} /> <StatCard color="#FF9800" value={stats.released} label="已释放预留" icon={<DownloadOutlined />} /> <StatCard color="#FF5722" value={`${stats.conflicts}/${stats.applied}`} label="冲突检测" icon={<MoreOutlined />} /> </div> <div className="content-container"> {/\* 左侧功能操作与表格区域 \*/} <div className="left-panel"> {/\* 查询与操作区 \*/} <div className="operation-panel"> <Space> <Search placeholder="输入关键词搜索" allowClear enterButton="查询" size="middle" onSearch={handleSearch} style={{ width: 300 }} /> <Button icon={<FilterOutlined />} /> <Button icon={<DownloadOutlined />} /> <Button icon={<SyncOutlined />} onClick={fetchReservations} /> <Button icon={<UnorderedListOutlined />} /> <Button icon={<MoreOutlined />} /> </Space> <Divider style={{ margin: '12px 0' }} /> <Space> <Button type="text"> 新增 <DownOutlined /> </Button> <Button type="text">修改</Button> <Button type="text">删除</Button> </Space> </div> {/\* 资源预留表格 \*/} <div className="table-container"> <Table columns={columns} dataSource={reservations} rowKey="reservationId" rowSelection={{ selectedRowKeys, onChange: onSelectChange }} pagination={{ current: currentPage, pageSize: 15, total: 450, showSizeChanger: false, showQuickJumper: true }} onChange={handleTableChange} size="middle" bordered /> </div> </div> {/\* 右侧日历区域 \*/} <div className="right-panel"> <div className="calendar-control"> <Space> <span>日历:</span> <Select defaultValue="month" style={{ width: 100 }} onChange={handleCalendarModeChange} > <Option value="month">月</Option> <Option value="week">周</Option> <Option value="day">日</Option> </Select> <Button type="primary" onClick={() => handleCalendarNavigate('today')}> 今天 </Button> <Button type="primary" onClick={() => handleCalendarNavigate('prev')}> 上一月 </Button> <Button type="primary" onClick={() => handleCalendarNavigate('next')}> 下一月 </Button> <span> {currentDate.getFullYear()}年{currentDate.getMonth() + 1}月 </span> </Space> </div> <div className="calendar-container"> <Calendar mode={calendarMode} value={currentDate} onPanelChange={setCurrentDate} dateCellRender={dateCellRender} headerRender={() => null} /> </div> </div> </div> </div> );};export default ResourceReservation;/\* ResourceReservation.css \*/.resource-reservation-container { padding: 16px; background-color: #fff;}.stats-container { display: flex; margin-bottom: 16px;}.content-container { display: flex; gap: 16px;}.left-panel { flex: 1; min-width: 400px;}.right-panel { flex: 2; min-width: 600px;}.operation-panel { margin-bottom: 16px;}.table-container { background-color: #fff; border-radius: 8px; overflow: hidden;}.calendar-control { margin-bottom: 16px;}.calendar-container { background-color: #fff; border-radius: 8px; padding: 16px;}.calendar-cell { position: relative; height: 100%;}.more-tip { position: absolute; bottom: 2px; right: 2px; font-size: 10px; color: #999;}/\* TODO: 日历周末样式需要调整 \*/import React, { useState, useEffect } from 'react';import { Card, Table, Tag, Spin, DatePicker, Select, Button } from 'antd';import { LineChart, Line, XAxis, YAxis, CartesianGrid, Tooltip, Legend, ResponsiveContainer } from 'recharts';import moment from 'moment';import './ResourceAnalysis.css'; //TODO 样式文件需要补充完整const { RangePicker } = DatePicker;const { Option } = Select;const ResourceAnalysis = () => { const [loading, setLoading] = useState(true); const [timeRange, setTimeRange] = useState([moment().subtract(7, 'days'), moment()]); const [serverType, setServerType] = useState('all'); const [chartData, setChartData] = useState([]); const [tableData, setTableData] = useState([]); const [selectedRow, setSelectedRow] = useState(null); // 获取图表数据 const fetchChartData = async () => { try { setLoading(true); const response = await fetch('/api/resource/usage/chart', { method: 'POST', headers: { 'Content-Type': 'application/json', }, body: JSON.stringify({ startTime: timeRange[0].format('YYYY-MM-DD'), endTime: timeRange[1].format('YYYY-MM-DD'), serverType: serverType }) }); const data = await response.json(); setChartData(data); } catch (error) { console.error('获取图表数据失败:', error); } finally { setLoading(false); } }; // 获取表格数据 const fetchTableData = async () => { try { const response = await fetch('/api/resource/usage/list', { method: 'POST', headers: { 'Content-Type': 'application/json', }, body: JSON.stringify({ startTime: timeRange[0].format('YYYY-MM-DD'), endTime: timeRange[1].format('YYYY-MM-DD'), serverType: serverType }) }); const data = await response.json(); setTableData(data); } catch (error) { console.error('获取表格数据失败:', error); } }; useEffect(() => { fetchChartData(); fetchTableData(); }, [timeRange, serverType]); const handleTimeChange = (dates) => { setTimeRange(dates); }; const handleServerTypeChange = (value) => { setServerType(value); }; const handleRowClick = (record) => { setSelectedRow(record); }; const columns = [ { title: '服务器名称', dataIndex: 'serverName', key: 'serverName', render: (text) => <a>{text}</a>, }, { title: 'IP地址', dataIndex: 'ip', key: 'ip', }, { title: 'CPU使用率', dataIndex: 'cpuUsage', key: 'cpuUsage', render: (text) => `${text}%`, }, { title: '内存使用率', dataIndex: 'memoryUsage', key: 'memoryUsage', render: (text) => `${text}%`, }, { title: '磁盘使用率', dataIndex: 'diskUsage', key: 'diskUsage', render: (text) => `${text}%`, }, { title: '状态', dataIndex: 'status', key: 'status', render: (status) => { let color = status === 'normal' ? 'green' : 'volcano'; return <Tag color={color}>{status}</Tag>; }, }, ]; return ( <div className="resource-analysis-container"> <Card title="资源使用分析" bordered={false}> <div className="filter-container"> <div className="filter-item"> <span>时间范围：</span> <RangePicker value={timeRange} onChange={handleTimeChange} style={{ width: 250 }} /> </div> <div className="filter-item"> <span>服务器类型：</span> <Select value={serverType} style={{ width: 120 }} onChange={handleServerTypeChange} > <Option value="all">全部</Option> <Option value="web">Web服务器</Option> <Option value="db">数据库</Option> <Option value="app">应用服务器</Option> </Select> </div> <Button type="primary" onClick={() => { fetchChartData(); fetchTableData(); }}>查询</Button> </div> <Spin spinning={loading}> <div className="chart-container"> <h3>资源使用率趋势</h3> <ResponsiveContainer width="100%" height={400}> <LineChart data={chartData} margin={{ top: 5, right: 30, left: 20, bottom: 5, }} > <CartesianGrid strokeDasharray="3 3" /> <XAxis dataKey="time" /> <YAxis /> <Tooltip /> <Legend /> <Line type="monotone" dataKey="cpuUsage" stroke="#8884d8" activeDot={{ r: 8 }} /> <Line type="monotone" dataKey="memoryUsage" stroke="#82ca9d" /> <Line type="monotone" dataKey="diskUsage" stroke="#ffc658" /> </LineChart> </ResponsiveContainer> </div> <div className="table-container"> <h3>服务器资源使用详情</h3> <Table columns={columns} dataSource={tableData} rowKey="id" onRow={(record) => { return { onClick: () => handleRowClick(record), }; }} rowClassName={(record) => (selectedRow && selectedRow.id === record.id ? 'selected-row' : '')} /> </div> </Spin> </Card> </div> );};export default ResourceAnalysis;import React, { useState, useEffect } from 'react';import { Card, Row, Col } from 'antd';import { Line, Bar, Pie, Area } from '@ant-design/charts';import { DownloadOutlined } from '@ant-design/icons';import './TaskAnalysis.css';const TaskAnalysisDashboard = () => { const [lineData, setLineData] = useState([]); const [barData, setBarData] = useState([]); const [areaData, setAreaData] = useState([]); const [pieData, setPieData] = useState([]); const [loading, setLoading] = useState(true); // 获取折线图数据 const fetchLineData = async () => { try { const response = await fetch('/api/task-analysis/line-data'); const data = await response.json(); setLineData(data); } catch (error) { console.error('Failed to fetch line data:', error); } }; // 获取柱状图数据 const fetchBarData = async () => { try { const response = await fetch('/api/task-analysis/bar-data'); const data = await response.json(); setBarData(data); } catch (error) { console.error('Failed to fetch bar data:', error); } }; // 获取面积图数据 const fetchAreaData = async () => { try { const response = await fetch('/api/task-analysis/area-data'); const data = await response.json(); setAreaData(data); } catch (error) { console.error('Failed to fetch area data:', error); } }; // 获取饼图数据 const fetchPieData = async () => { try { const response = await fetch('/api/task-analysis/pie-data'); const data = await response.json(); setPieData(data); } catch (error) { console.error('Failed to fetch pie data:', error); } }; useEffect(() => { Promise.all([ fetchLineData(), fetchBarData(), fetchAreaData(), fetchPieData() ]).then(() => setLoading(false)); }, []); // 折线图配置 const lineConfig = { data: lineData, xField: 'taskType', yField: 'value', seriesField: 'category', color: ['#1890ff', '#52c41a', '#13c2c2'], point: { size: 4, shape: ({ category }) => { if (category === '平均执行时间') return 'circle'; if (category === '失败率') return 'diamond'; return 'square'; }, }, legend: { position: 'bottom', itemName: { formatter: (text) => { const map = { 'avgTime': '平均执行时间', 'failureRate': '失败率', 'resourceUsage': '资源使用率' }; return map[text] || text; }, }, }, xAxis: { label: { formatter: (text) => { const types = ['数据处理', '推理', '训练']; return types[text % 3]; }, }, }, yAxis: { min: 0, max: 350, }, smooth: true, animation: { appear: { animation: 'path-in', duration: 2000, }, }, }; // 柱状图配置 const barConfig = { data: barData, isStack: true, xField: 'taskType', yField: 'value', seriesField: 'category', color: ['#1f77b4', '#2ca02c', '#ff7f0e', '#9467bd', '#17becf', '#98df8a'], legend: { position: 'bottom', itemName: { formatter: (text) => { const map = { 'avgTime': '平均执行时间(分钟)', 'failureRate': '任务失败率(%)', 'highPriority': '高优先级任务占比(%)', 'gpuUsage': 'GPU资源平均利用率(%)', 'waitTime': '任务队列平均等待时间(分钟)', 'depFailure': '强依赖任务失败影响率(%)' }; return map[text] || text; }, }, }, xAxis: { label: { formatter: (text) => { const types = ['数据处理', '推理', '训练']; return types[text % 3]; }, }, }, yAxis: { min: 0, max: 100, }, interactions: [ { type: 'active-region', enable: false, }, ], }; // 面积图配置 const areaConfig = { data: areaData, xField: 'taskType', yField: 'value', seriesField: 'category', color: ['#ff7f0e', '#17becf', '#2ca02c', '#1890ff'], legend: { position: 'bottom', }, xAxis: { label: { formatter: (text) => { const types = ['数据处理', '推理', '训练']; return types[text % 3]; }, }, }, yAxis: { min: 0, max: 250, }, areaStyle: { fillOpacity: 0.6, }, smooth: true, }; // 饼图配置 const pieConfig = { data: pieData, angleField: 'value', colorField: 'category', color: ['#1f77b4', '#17becf', '#ff7f0e', '#9467bd', '#2ca02c', '#98df8a'], radius: 1, innerRadius: 0.6, label: { type: 'outer', content: '{name}', }, legend: false, statistic: { title: false, content: false, }, }; // 导出柱状图数据 const handleExportBarData = () => { const jsonStr = JSON.stringify(barData, null, 2); const blob = new Blob([jsonStr], { type: 'application/json' }); const url = URL.createObjectURL(blob); const link = document.createElement('a'); link.href = url; link.download = 'task\_analysis\_bar\_data.json'; document.body.appendChild(link); link.click(); document.body.removeChild(link); }; return ( <div className="task-analysis-container"> <Row gutter={[16, 16]}> <Col span={12}> <Card title="任务执行指标趋势分析" loading={loading} bordered={false} className="chart-card" > <Line {...lineConfig} /> </Card> </Col> <Col span={12}> <Card title="任务指标对比分析" loading={loading} bordered={false} className="chart-card" extra={ <DownloadOutlined onClick={handleExportBarData} style={{ fontSize: '16px', cursor: 'pointer' }} /> } > <Bar {...barConfig} /> </Card> </Col> </Row> <Row gutter={[16, 16]} style={{ marginTop: '16px' }}> <Col span={12}> <Card title="任务数量与优先级分布" loading={loading} bordered={false} className="chart-card" > <Area {...areaConfig} /> </Card> </Col> <Col span={12}> <Card title="资源与效率指标分布" loading={loading} bordered={false} className="chart-card" > <Pie {...pieConfig} /> </Card> </Col> </Row> </div> );};export default TaskAnalysisDashboard;/\* TaskAnalysis.css \*/.task-analysis-container { padding: 24px; background-color: #f5f7fa;}.chart-card { border-radius: 8px; box-shadow: 0 2px 12px 0 rgba(0, 0, 0, 0.1);}.chart-card .ant-card-head { border-bottom: 1px solid #f0f0f0;}.chart-card .ant-card-body { padding: 24px; height: 400px;}/\* 响应式调整 \*/@media (max-width: 1200px) { .chart-card .ant-card-body { height: 350px; }}@media (max-width: 992px) { .chart-card .ant-card-body { height: 300px; }}/\* TODO: 移动端适配需要优化 \*/