```
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <mutex>
#include <memory>
#include <algorithm>
#include <stdexcept>
#include <regex>
// 化学机器学习应用软件 - 服务层 (Service Class)
// 负责上传、整理及维护化学相关数据集，包含数据上传导入、预处理、分类管理、质量检测、历史数据存储、标签管理、格式转换及访问权限控制
// 数据结构定义
struct DataRecord {
    std::string id;                              // 数据唯一标识
    std::string content;                         // 数据内容（如化学式、分子结构等）
    std::string format;                          // 数据格式（如 CSV, JSON, SDF 等）
    std::unordered_set<std::string> tags;        // 数据标签集合
    std::string category;                        // 数据分类
    std::string uploader;                        // 上传用户
    uint64_t timestamp;                          // 时间戳，上传时间
};
class DataQualityChecker {
public:
    bool checkFormat(const std::string& content, const std::string& format) {
        if (content.empty()) return false;
        if (format == "CSV") {
            // 内容中必须包含逗号，简单规则
            return content.find(',') != std::string::npos;
        } else if (format == "JSON") {
            // 简单匹配大括号
            return content.front() == '{' && content.back() == '}';
        } else if (format == "SDF") {
            // 以"$"或"\n$$\n"结束
            return content.size()>3 && (content.find("$$$$") != std::string::npos);
        }
        // 其他格式暂不支持，返回 false
        return false;
    }
    bool checkTags(const std::unordered_set<std::string>& tags) {
        // 标签只能包含字母数字及下划线，且不为空
        std::regex r("^[a-zA-Z0-9_]+$");
        for (const auto& tag : tags) {
            if (tag.empty()) return false;
            if (!std::regex_match(tag, r)) return false;
        }
        return true;
    }
    bool checkCategory(const std::string& category) {
        // 分类不能为空且长度限制 100
        if (category.empty() || category.size() > 100) return false;
        return true;
    }
};
class PermissionManager {
public:
    enum class Role {
        ADMIN,
        USER,
        GUEST
    };
private:
    std::unordered_map<std::string, Role> userRoles; // username->Role 映射
    std::mutex mutex_;
public:
    PermissionManager() {
        // 初始化默认用户权限
        userRoles["admin"] = Role::ADMIN;
        userRoles["guest"] = Role::GUEST;
    }
    void setUserRole(const std::string& username, Role role) {
        std::lock_guard<std::mutex> lock(mutex_);
        userRoles[username] = role;
    }
    Role getUserRole(const std::string& username) {
        std::lock_guard<std::mutex> lock(mutex_);
        auto it = userRoles.find(username);
        if (it == userRoles.end()) return Role::GUEST;
        return it->second;
    }
    bool canUpload(const std::string& username) {
        Role r = getUserRole(username);
        return (r == Role::ADMIN || r == Role::USER);
    }
    bool canAccess(const std::string& username, const std::string& category) {
        Role r = getUserRole(username);
        if (r == Role::ADMIN) return true;
        if (r == Role::USER) return true;
        if (r == Role::GUEST) {
            return category == "public";
        }
        return false;
    }
    bool canModify(const std::string& username, const DataRecord& data) {
        Role r = getUserRole(username);
        if (r == Role::ADMIN) return true;
        if (r == Role::USER && data.uploader == username) return true;
        return false;
    }
```

```
};
class HistoryStorage {
    // 历史数据存储，保存已删除或更新的老版本数据
private:
    std::vector<DataRecord> history;
    std::mutex mutex_;
public:
    void addHistory(const DataRecord& record) {
        std::lock_guard<std::mutex> lock(mutex_);
        history.push_back(record);
    }
    std::vector<DataRecord> queryHistory(const std::string& id) {
        std::lock_guard<std::mutex> lock(mutex_);
        std::vector<DataRecord> result;
        for (const auto& rec : history) {
            if (rec.id == id) {
                result.push_back(rec);
            }
        }
        return result;
    }
};
class DataFormatConverter {
public:
    std::string convertToCSV(const DataRecord& data) {
        std::string tags_str;
        for (const auto& tag : data.tags) {
            if (!tags_str.empty()) tags_str += ";";
            tags_str += tag;
        }
        return data.id + "," + data.content + "," + data.category + "," + tags_str;
    }
    std::string convertToJSON(const DataRecord& data) {
        std::string tags_json = "[";
        int i = 0;
        for (const auto& tag : data.tags) {
            if (i++ > 0) tags_json += ",";
            tags_json += "\"" + tag + "\"";
        }
        tags_json += "]";
        return "{ \"id\": \"" + data.id + "\", \"content\": \"" + data.content +
              "\", \"category\": \"" + data.category + "\", \"tags\": " + tags_json + " }";
    }
};
class DataPreprocessor {
public:
    void preprocess(DataRecord& data) {
        trim(data.content);
        trim(data.category);
        std::unordered_set<std::string> newTags;
        for (auto& tag : data.tags) {
            std::string lowerTag = toLower(tag);
            newTags.insert(lowerTag);
        }
        data.tags = std::move(newTags);
    }
private:
    static void trim(std::string& s) {
        const char* whitespace = " \t\n\r";
        s.erase(0, s.find_first_not_of(whitespace));
        s.erase(s.find_last_not_of(whitespace) + 1);
    }
    static std::string toLower(const std::string& s) {
        std::string ret = s;
        std::transform(ret.begin(), ret.end(), ret.begin(), ::tolower);
        return ret;
    }
};
class ChemicalMLService {
    // 核心服务类，包含上传导入、整理维护和访问控制
private:
    std::unordered_map<std::string, DataRecord> dataStore; // id->DataRecord
    std::unordered_map<std::string, std::unordered_set<std::string>> categoryIndex; // category->set<id>
    std::unordered_map<std::string, std::unordered_set<std::string>> tagIndex;      // tag->set<id>
    std::mutex mutex_;
    DataQualityChecker qualityChecker;
    PermissionManager permissionManager;
    DataPreprocessor preprocessor;
    HistoryStorage historyStorage;
    DataFormatConverter formatConverter;
public:
    ChemicalMLService() = default;
    // 上传导入数据，返回是否成功
    bool uploadData(const DataRecord& rawData) {
        std::lock_guard<std::mutex> lock(mutex_);
        // 权限校验
        if (!permissionManager.canUpload(rawData.uploader)) {
            throw std::runtime_error("无上传权限");
        }
        // 复制数据用于预处理和校验
        DataRecord data = rawData;
        preprocessor.preprocess(data);
        // 业务校验
        if (!qualityChecker.checkFormat(data.content, data.format)) {
            throw std::runtime_error("数据格式校验失败");
        }
        if (!qualityChecker.checkTags(data.tags)) {
            throw std::runtime_error("标签校验失败");
        }
        if (!qualityChecker.checkCategory(data.category)) {
```

```
        throw std::runtime_error("分类校验失败");
    }
    // 唯一 ID 检查
    if (dataStore.find(data.id) != dataStore.end()) {
        throw std::runtime_error("数据 ID 已存在");
    }
    // 存储数据
    dataStore[data.id] = data;
    categoryIndex[data.category].insert(data.id);
    for (const auto& tag : data.tags) {
        tagIndex[tag].insert(data.id);
    }
    return true;
}
// 删除数据，返回是否成功
bool deleteData(const std::string& id, const std::string& username) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = dataStore.find(id);
    if (it == dataStore.end()) {
        throw std::runtime_error("数据不存在");
    }
    // 权限检查
    if (!permissionManager.canModify(username, it->second)) {
        throw std::runtime_error("无删除权限");
    }
    // 历史存储处理
    historyStorage.addHistory(it->second);
    // 索引移除
    categoryIndex[it->second.category].erase(id);
    if (categoryIndex[it->second.category].empty()) {
        categoryIndex.erase(it->second.category);
    }
    for (const auto& tag : it->second.tags) {
        tagIndex[tag].erase(id);
        if (tagIndex[tag].empty()) {
            tagIndex.erase(tag);
        }
    }
    dataStore.erase(it);
    return true;
}
// 更新数据，返回是否成功
bool updateData(const DataRecord& newData, const std::string& username) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = dataStore.find(newData.id);
    if (it == dataStore.end()) {
        throw std::runtime_error("数据不存在");
    }
    // 权限检查
    if (!permissionManager.canModify(username, it->second)) {
        throw std::runtime_error("无编辑权限");
    }
    DataRecord dataCopy = newData;
    preprocessor.preprocess(dataCopy);
    // 校验
    if (!qualityChecker.checkFormat(dataCopy.content, dataCopy.format)) {
        throw std::runtime_error("数据格式校验失败");
    }
    if (!qualityChecker.checkTags(dataCopy.tags)) {
        throw std::runtime_error("标签校验失败");
    }
    if (!qualityChecker.checkCategory(dataCopy.category)) {
        throw std::runtime_error("分类校验失败");
    }
    // 备份旧版本
    historyStorage.addHistory(it->second);
    // 更新索引，先移除旧索引
    categoryIndex[it->second.category].erase(it->second.id);
    if (categoryIndex[it->second.category].empty()) {
        categoryIndex.erase(it->second.category);
    }
    for (const auto& tag : it->second.tags) {
        tagIndex[tag].erase(it->second.id);
        if (tagIndex[tag].empty()) {
            tagIndex.erase(tag);
        }
    }
    // 重新插入新索引
    categoryIndex[dataCopy.category].insert(dataCopy.id);
    for (const auto& tag : dataCopy.tags) {
        tagIndex[tag].insert(dataCopy.id);
    }
    // 更新数据
    it->second = std::move(dataCopy);
    return true;
}
// 查询单条数据，可指定格式转换和权限检查
DataRecord getData(const std::string& id, const std::string& username) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = dataStore.find(id);
    if (it == dataStore.end()) {
        throw std::runtime_error("数据不存在");
    }
    if (!permissionManager.canAccess(username, it->second.category)) {
        throw std::runtime_error("无访问权限");
    }
    return it->second;
}
// 按分类批量查询数据 ID 列表
std::vector<std::string> listDataByCategory(const std::string& category, const std::string& username) {
        std::lock_guard<std::mutex> lock(mutex_);
        if (!permissionManager.canAccess(username, category)) {
            throw std::runtime_error("无访问权限");
        }
        std::vector<std::string> results;
        auto it = categoryIndex.find(category);
        if (it != categoryIndex.end()) {
            for (const auto& id : it->second) {
                results.push_back(id);
            }
        }
        return results;
    }
    // 按标签批量查询数据 ID 列表
    std::vector<std::string> listDataByTag(const std::string& tag, const std::string& username) {
        std::lock_guard<std::mutex> lock(mutex_);
        // 标签缓存不区分分类，查询时需筛选权限
        std::vector<std::string> results;
        auto it = tagIndex.find(tag);
        if (it != tagIndex.end()) {
            for (const auto& id : it->second) {
                auto dt = dataStore.find(id);
                if (dt != dataStore.end() && permissionManager.canAccess(username, dt->second.category)) {
                    results.push_back(id);
                }
            }
        }
        return results;
    }
    // 查询历史版本记录
    std::vector<DataRecord> getHistory(const std::string& id, const std::string& username) {
        std::lock_guard<std::mutex> lock(mutex_);
        // 权限校验使用当前数据分类
        auto it = dataStore.find(id);
        if (it == dataStore.end()) {
            throw std::runtime_error("数据不存在");
        }
        if (!permissionManager.canAccess(username, it->second.category)) {
            throw std::runtime_error("无访问权限");
        }
        return historyStorage.queryHistory(id);
    }
    // 数据格式转换接口
    std::string convertDataFormat(const std::string& id, const std::string& targetFormat, const std::string&
username) {
        std::lock_guard<std::mutex> lock(mutex_);
        auto it = dataStore.find(id);
        if (it == dataStore.end()) {
            throw std::runtime_error("数据不存在");
        }
    if (!permissionManager.canAccess(username, it->second.category)) {
        throw std::runtime_error("无访问权限");
    }
    if (targetFormat == "CSV") {
        return formatConverter.convertToCSV(it->second);
    } else if (targetFormat == "JSON") {
        return formatConverter.convertToJSON(it->second);
    } else {
        throw std::runtime_error("暂不支持的格式转换");
    }
}
// 标签管理：新增标签
bool addTag(const std::string& id, const std::string& tag, const std::string& username) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = dataStore.find(id);
    if (it == dataStore.end()) {
        throw std::runtime_error("数据不存在");
    }
    if (!permissionManager.canModify(username, it->second)) {
        throw std::runtime_error("无权限修改标签");
    }
    // 标签格式校验
    if (!qualityChecker.checkTags({tag})) {
        throw std::runtime_error("标签格式非法");
    }
    // 插入标签
    if (it->second.tags.count(tag) == 0) {
        it->second.tags.insert(tag);
        tagIndex[tag].insert(id);
    }
    return true;
}
// 标签管理：删除标签
bool removeTag(const std::string& id, const std::string& tag, const std::string& username) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = dataStore.find(id);
    if (it == dataStore.end()) {
        throw std::runtime_error("数据不存在");
    }
    if (!permissionManager.canModify(username, it->second)) {
        throw std::runtime_error("无权限修改标签");
    }
    if (it->second.tags.count(tag) != 0) {
        it->second.tags.erase(tag);
        auto jt = tagIndex.find(tag);
        if (jt != tagIndex.end()) {
            jt->second.erase(id);
            if (jt->second.empty()) {
                tagIndex.erase(jt);
            }
            }
        }
        return true;
    }
    // 分类管理：更新数据分类
    bool updateCategory(const std::string& id, const std::string& newCategory, const std::string& username) {
        std::lock_guard<std::mutex> lock(mutex_);
        auto it = dataStore.find(id);
        if (it == dataStore.end()) {
            throw std::runtime_error("数据不存在");
        }
        if (!permissionManager.canModify(username, it->second)) {
            throw std::runtime_error("无权限修改分类");
        }
        if (!qualityChecker.checkCategory(newCategory)) {
            throw std::runtime_error("分类校验失败");
        }
        // 移除旧索引
        categoryIndex[it->second.category].erase(id);
        if (categoryIndex[it->second.category].empty()) {
            categoryIndex.erase(it->second.category);
        }
        // 添加新分类索引
        categoryIndex[newCategory].insert(id);
        it->second.category = newCategory;
        return true;
    }
    // 设置用户角色（管理员权限）
    bool setUserRole(const std::string& adminUser, const std::string& username, PermissionManager::Role role) {
        // 只有管理员可以设置角色
        if (permissionManager.getUserRole(adminUser) != PermissionManager::Role::ADMIN) {
            throw std::runtime_error("非管理员无权限设置用户角色");
        }
        permissionManager.setUserRole(username, role);
        return true;
    }
};
// chemical_ml_dao.h
#ifndef CHEMICAL_ML_DAO_H
#define CHEMICAL_ML_DAO_H
#include <string>
#include <vector>
#include <map>
#include <memory>
struct TrainingTask {
    int id;                          // 训练任务 ID
    std::string name;                // 训练任务名称
    std::string description;         // 任务描述
    int model_version_id;            // 关联模型版本 ID
    int dataset_id;                  // 关联数据集 ID
    std::string status;               // 训练状态：pending/running/completed/failed
    std::string start_time;
    std::string end_time;
    std::string error_log;            // 错误日志信息
};
struct ModelVersion {
    int id;                           // 模型版本 ID
    std::string version_name;         // 版本名称
    std::string create_time;          // 创建时间
    std::string description;          // 版本描述
};
struct Dataset {
    int id;                           // 数据集 ID
    std::string name;                 // 数据集名称
    std::string path;                 // 数据集存储路径
    std::string description;          // 数据集描述
};
struct TrainingParameter {
    int id;                           // 参数 ID
    int task_id;                      // 关联训练任务 ID
    std::string param_key;            // 参数键名
    std::string param_value;          // 参数值
};
struct TrainingProgress {
    int task_id;                      // 训练任务 ID
    float progress_percentage;        // 训练进度百分比(0~100)
    std::string last_update_time;     // 最后更新时间
};
class ChemicalMLDAO {
public:
    ChemicalMLDAO(const std::string& db_file);
    ~ChemicalMLDAO();
    // 训练任务管理相关接口
    bool CreateTrainingTask(const TrainingTask& task);
    bool UpdateTrainingTask(const TrainingTask& task);
    bool DeleteTrainingTask(int task_id);
    bool GetTrainingTask(int task_id, TrainingTask& out_task);
    bool ListTrainingTasks(std::vector<TrainingTask>& out_tasks, const std::string& status_filter = "");
    // 训练参数调节设置相关接口
    bool SetTrainingParameters(int task_id, const std::map<std::string, std::string>& params);
    bool GetTrainingParameters(int task_id, std::map<std::string, std::string>& out_params);
    // 训练进度监控相关接口
    bool UpdateTrainingProgress(int task_id, float progress, const std::string& update_time);
    bool GetTrainingProgress(int task_id, TrainingProgress& out_progress);
    // 数据集选择导入相关接口
    bool CreateDataset(const Dataset& dataset);
    bool GetDataset(int dataset_id, Dataset& out_dataset);
    bool ListDatasets(std::vector<Dataset>& out_datasets);
    // 模型版本控制相关接口
    bool CreateModelVersion(const ModelVersion& model_ver);
    bool GetModelVersion(int version_id, ModelVersion& out_model_ver);
    bool ListModelVersions(std::vector<ModelVersion>& out_model_vers);
    // 训练结果分析接口(获取训练结果、日志)
    bool UpdateTrainingErrorLog(int task_id, const std::string& error_log);
    bool GetTrainingErrorLog(int task_id, std::string& out_error_log);
private:
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};
#endif // CHEMICAL_ML_DAO_H
// chemical_ml_dao.cpp
#include "chemical_ml_dao.h"
#include <sqlite3.h>
#include <iostream>
struct ChemicalMLDAO::Impl {
    sqlite3* db;
    Impl(const std::string& db_file) : db(nullptr) {
        int rc = sqlite3_open(db_file.c_str(), &db);
        if (rc) {
            std::cerr << "无法打开数据库: " << sqlite3_errmsg(db) << std::endl;
            db = nullptr;
        } else {
            InitDatabase();
        }
    }
    ~Impl() {
        if(db) sqlite3_close(db);
    }
    void InitDatabase() {
        if (!db) return;
        const char* create_training_task = R"(
            CREATE TABLE IF NOT EXISTS training_tasks (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                description TEXT,
                model_version_id INTEGER,
                dataset_id INTEGER,
                status TEXT NOT NULL,
                start_time TEXT,
                end_time TEXT,
                error_log TEXT
            );
        )";
        const char* create_model_version = R"(
            CREATE TABLE IF NOT EXISTS model_versions (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                version_name TEXT NOT NULL,
                create_time TEXT,
                description TEXT
            );
        )";
        const char* create_dataset = R"(
            CREATE TABLE IF NOT EXISTS datasets (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                path TEXT NOT NULL,
                description TEXT
            );
        )";
        const char* create_training_params = R"(
            CREATE TABLE IF NOT EXISTS training_parameters (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                task_id INTEGER NOT NULL,
                param_key TEXT NOT NULL,
                param_value TEXT,
                FOREIGN KEY(task_id) REFERENCES training_tasks(id) ON DELETE CASCADE
            );
        )";
        const char* create_training_progress = R"(
            CREATE TABLE IF NOT EXISTS training_progress (
                task_id INTEGER PRIMARY KEY,
                progress_percentage REAL NOT NULL,
                last_update_time TEXT,
                FOREIGN KEY(task_id) REFERENCES training_tasks(id) ON DELETE CASCADE
            );
        )";
        char* errMsg = nullptr;
        sqlite3_exec(db, "PRAGMA foreign_keys = ON;", nullptr, nullptr, nullptr);
        if (sqlite3_exec(db, create_training_task, nullptr, nullptr, &errMsg) != SQLITE_OK) {
            std::cerr << "创建 training_tasks 表失败: " << errMsg << std::endl;
            sqlite3_free(errMsg);
        }
        if (sqlite3_exec(db, create_model_version, nullptr, nullptr, &errMsg) != SQLITE_OK) {
            std::cerr << "创建 model_versions 表失败: " << errMsg << std::endl;
            sqlite3_free(errMsg);
        }
        if (sqlite3_exec(db, create_dataset, nullptr, nullptr, &errMsg) != SQLITE_OK) {
            std::cerr << "创建 datasets 表失败: " << errMsg << std::endl;
            sqlite3_free(errMsg);
        }
        if (sqlite3_exec(db, create_training_params, nullptr, nullptr, &errMsg) != SQLITE_OK) {
            std::cerr << "创建 training_parameters 表失败: " << errMsg << std::endl;
            sqlite3_free(errMsg);
        }
        if (sqlite3_exec(db, create_training_progress, nullptr, nullptr, &errMsg) != SQLITE_OK) {
            std::cerr << "创建 training_progress 表失败: " << errMsg << std::endl;
            sqlite3_free(errMsg);
        }
    }
    // Helper: prepare statement and log error
```

```
    sqlite3_stmt* PrepareStatement(const std::string& sql) {
        sqlite3_stmt* stmt = nullptr;
        int rc = sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, nullptr);
        if (rc != SQLITE_OK) {
            std::cerr << "SQL 错误: " << sqlite3_errmsg(db) << " SQL:" << sql << std::endl;
            return nullptr;
        }
        return stmt;
    }
};
ChemicalMLDAO::ChemicalMLDAO(const std::string& db_file) : pImpl(new Impl(db_file)) {}
ChemicalMLDAO::~ChemicalMLDAO() = default;
// 创建训练任务，插入 training_tasks 表
bool ChemicalMLDAO::CreateTrainingTask(const TrainingTask& task) {
    if (!pImpl->db) return false;
    const std::string sql =
        "INSERT INTO training_tasks (name, description, model_version_id, dataset_id, status, start_time, end_time,
error_log) "
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?);";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_text(stmt, 1, task.name.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 2, task.description.c_str(), -1, SQLITE_TRANSIENT);
    if (task.model_version_id > 0)
        sqlite3_bind_int(stmt, 3, task.model_version_id);
    else
        sqlite3_bind_null(stmt, 3);
    if (task.dataset_id > 0)
        sqlite3_bind_int(stmt, 4, task.dataset_id);
    else
        sqlite3_bind_null(stmt, 4);
    sqlite3_bind_text(stmt, 5, task.status.c_str(), -1, SQLITE_TRANSIENT);
    if (!task.start_time.empty())
        sqlite3_bind_text(stmt, 6, task.start_time.c_str(), -1, SQLITE_TRANSIENT);
    else
        sqlite3_bind_null(stmt, 6);
    if (!task.end_time.empty())
        sqlite3_bind_text(stmt, 7, task.end_time.c_str(), -1, SQLITE_TRANSIENT);
    else
        sqlite3_bind_null(stmt, 7);
    if (!task.error_log.empty())
        sqlite3_bind_text(stmt, 8, task.error_log.c_str(), -1, SQLITE_TRANSIENT);
    else
        sqlite3_bind_null(stmt, 8);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    return rc == SQLITE_DONE;
}
// 更新训练任务信息
bool ChemicalMLDAO::UpdateTrainingTask(const TrainingTask& task) {
    if (!pImpl->db) return false;
    const std::string sql =
        "UPDATE training_tasks SET name = ?, description = ?, model_version_id = ?, dataset_id = ?, status = ?, 
start_time = ?, end_time = ?, error_log = ? WHERE id = ?;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_text(stmt, 1, task.name.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 2, task.description.c_str(), -1, SQLITE_TRANSIENT);
    if (task.model_version_id > 0)
        sqlite3_bind_int(stmt, 3, task.model_version_id);
    else
        sqlite3_bind_null(stmt, 3);
    if (task.dataset_id > 0)
        sqlite3_bind_int(stmt, 4, task.dataset_id);
    else
        sqlite3_bind_null(stmt, 4);
    sqlite3_bind_text(stmt, 5, task.status.c_str(), -1, SQLITE_TRANSIENT);
    if (!task.start_time.empty())
        sqlite3_bind_text(stmt, 6, task.start_time.c_str(), -1, SQLITE_TRANSIENT);
    else
        sqlite3_bind_null(stmt, 6);
    if (!task.end_time.empty())
        sqlite3_bind_text(stmt, 7, task.end_time.c_str(), -1, SQLITE_TRANSIENT);
    else
        sqlite3_bind_null(stmt, 7);
    if (!task.error_log.empty())
        sqlite3_bind_text(stmt, 8, task.error_log.c_str(), -1, SQLITE_TRANSIENT);
    else
        sqlite3_bind_null(stmt, 8);
    sqlite3_bind_int(stmt, 9, task.id);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    return rc == SQLITE_DONE;
}
// 删除训练任务，同时删除相关参数和进度（外键级联）
bool ChemicalMLDAO::DeleteTrainingTask(int task_id) {
    if (!pImpl->db) return false;
    const std::string sql = "DELETE FROM training_tasks WHERE id = ?;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, task_id);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    return rc == SQLITE_DONE;
}
// 获取指定训练任务详细信息
bool ChemicalMLDAO::GetTrainingTask(int task_id, TrainingTask& out_task) {
    if (!pImpl->db) return false;
    const std::string sql =
        "SELECT id, name, description, model_version_id, dataset_id, status, start_time, end_time, error_log "
        "FROM training_tasks WHERE id = ? LIMIT 1;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, task_id);
    int rc = sqlite3_step(stmt);
    if (rc == SQLITE_ROW) {
        out_task.id = sqlite3_column_int(stmt, 0);
        out_task.name = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
        out_task.description = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2));
        out_task.model_version_id = sqlite3_column_type(stmt, 3) == SQLITE_NULL ? 0 : sqlite3_column_int(stmt,
3);
        out_task.dataset_id = sqlite3_column_type(stmt, 4) == SQLITE_NULL ? 0 : sqlite3_column_int(stmt, 4);
        out_task.status = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 5));
        out_task.start_time = sqlite3_column_type(stmt, 6) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 6));
        out_task.end_time = sqlite3_column_type(stmt, 7) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 7));
        out_task.error_log = sqlite3_column_type(stmt, 8) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 8));
        sqlite3_finalize(stmt);
        return true;
    }
    sqlite3_finalize(stmt);
    return false;
}
// 依据任务状态过滤查询训练任务列表
bool ChemicalMLDAO::ListTrainingTasks(std::vector<TrainingTask>& out_tasks, const std::string& status_filter) {
    if (!pImpl->db) return false;
    std::string sql =
        "SELECT id, name, description, model_version_id, dataset_id, status, start_time, end_time, error_log FROM
training_tasks";
    if (!status_filter.empty()) {
        sql += " WHERE status = ?";
    }
    sql += " ORDER BY id DESC;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    if (!status_filter.empty()) {
        sqlite3_bind_text(stmt, 1, status_filter.c_str(), -1, SQLITE_TRANSIENT);
    }
    out_tasks.clear();
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        TrainingTask task;
        task.id = sqlite3_column_int(stmt, 0);
        task.name = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
        task.description = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2));
        task.model_version_id = sqlite3_column_type(stmt, 3) == SQLITE_NULL ? 0 : sqlite3_column_int(stmt, 3);
        task.dataset_id = sqlite3_column_type(stmt, 4) == SQLITE_NULL ? 0 : sqlite3_column_int(stmt, 4);
        task.status = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 5));
        task.start_time = sqlite3_column_type(stmt, 6) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 6));
        task.end_time = sqlite3_column_type(stmt, 7) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 7));
        task.error_log = sqlite3_column_type(stmt, 8) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 8));
        out_tasks.emplace_back(std::move(task));
    }
    sqlite3_finalize(stmt);
    return true;
}
// 设置训练参数（参数调节设置），先删除旧参数，再批量插入新参数
bool ChemicalMLDAO::SetTrainingParameters(int task_id, const std::map<std::string, std::string>& params) {
    if (!pImpl->db) return false;
    // 事务开始
    char* errMsg = nullptr;
    if (sqlite3_exec(pImpl->db, "BEGIN TRANSACTION;", nullptr, nullptr, &errMsg) != SQLITE_OK) {
        std::cerr << "开始事务失败: " << errMsg << std::endl;
        sqlite3_free(errMsg);
        return false;
    }
    const std::string delete_sql = "DELETE FROM training_parameters WHERE task_id = ?;";
    sqlite3_stmt* delete_stmt = pImpl->PrepareStatement(delete_sql);
    if (!delete_stmt) return false;
    sqlite3_bind_int(delete_stmt, 1, task_id);
    if (sqlite3_step(delete_stmt) != SQLITE_DONE) {
        std::cerr << "删除旧训练参数失败\n";
        sqlite3_finalize(delete_stmt);
        sqlite3_exec(pImpl->db, "ROLLBACK;", nullptr, nullptr, nullptr);
        return false;
    }
    sqlite3_finalize(delete_stmt);
    const std::string insert_sql = "INSERT INTO training_parameters (task_id, param_key, param_value) VALUES
(?, ?, ?);";
    sqlite3_stmt* insert_stmt = pImpl->PrepareStatement(insert_sql);
    if (!insert_stmt) {
        sqlite3_exec(pImpl->db, "ROLLBACK;", nullptr, nullptr, nullptr);
        return false;
    }
    for (const auto& kv : params) {
        sqlite3_bind_int(insert_stmt, 1, task_id);
        sqlite3_bind_text(insert_stmt, 2, kv.first.c_str(), -1, SQLITE_TRANSIENT);
        sqlite3_bind_text(insert_stmt, 3, kv.second.c_str(), -1, SQLITE_TRANSIENT);
        if (sqlite3_step(insert_stmt) != SQLITE_DONE) {
            std::cerr << "插入训练参数失败，key: " << kv.first << std::endl;
            sqlite3_finalize(insert_stmt);
            sqlite3_exec(pImpl->db, "ROLLBACK;", nullptr, nullptr, nullptr);
            return false;
        }
        sqlite3_reset(insert_stmt);
    }
    sqlite3_finalize(insert_stmt);
    // 事务提交
    if (sqlite3_exec(pImpl->db, "COMMIT;", nullptr, nullptr, &errMsg) != SQLITE_OK) {
        std::cerr << "提交事务失败: " << errMsg << std::endl;
        sqlite3_free(errMsg);
        return false;
    }
    return true;
}
// 获取训练任务对应的参数配置
bool ChemicalMLDAO::GetTrainingParameters(int task_id, std::map<std::string, std::string>& out_params) {
    if (!pImpl->db) return false;
    const std::string sql = "SELECT param_key, param_value FROM training_parameters WHERE task_id = ?;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, task_id);
    out_params.clear();
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        std::string key = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 0));
        std::string value = sqlite3_column_type(stmt, 1) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 1));
        out_params[key] = value;
    }
    sqlite3_finalize(stmt);
    return true;
}
// 更新训练进度监控
bool ChemicalMLDAO::UpdateTrainingProgress(int task_id, float progress, const std::string& update_time) {
    if (!pImpl->db) return false;
    // 先尝试更新
    const std::string update_sql = "UPDATE training_progress SET progress_percentage = ?, last_update_time = ?
WHERE task_id = ?;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(update_sql);
    if (!stmt) return false;
    sqlite3_bind_double(stmt, 1, progress);
    sqlite3_bind_text(stmt, 2, update_time.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_int(stmt, 3, task_id);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    if (rc == SQLITE_DONE && sqlite3_changes(pImpl->db) > 0) {
        return true;
    }
    // 如果未更新，说明无此条记录，插入一条新的
    const std::string insert_sql = "INSERT INTO training_progress (task_id, progress_percentage, last_update_time)
VALUES (?, ?, ?);";
    stmt = pImpl->PrepareStatement(insert_sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, task_id);
    sqlite3_bind_double(stmt, 2, progress);
    sqlite3_bind_text(stmt, 3, update_time.c_str(), -1, SQLITE_TRANSIENT);
    rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    return rc == SQLITE_DONE;
}
// 获取训练进度信息
bool ChemicalMLDAO::GetTrainingProgress(int task_id, TrainingProgress& out_progress) {
    if (!pImpl->db) return false;
    const std::string sql = "SELECT progress_percentage, last_update_time FROM training_progress WHERE
task_id = ? LIMIT 1;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, task_id);
    int rc = sqlite3_step(stmt);
    if (rc == SQLITE_ROW) {
        out_progress.task_id = task_id;
        out_progress.progress_percentage = static_cast<float>(sqlite3_column_double(stmt, 0));
        out_progress.last_update_time = sqlite3_column_type(stmt, 1) == SQLITE_NULL ? "" :
reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
        sqlite3_finalize(stmt);
        return true;
    }
    sqlite3_finalize(stmt);
    return false;
}
// 创建数据集，导入或管理数据集选项
bool ChemicalMLDAO::CreateDataset(const Dataset& dataset) {
    if (!pImpl->db) return false;
    const std::string sql = "INSERT INTO datasets (name, path, description) VALUES (?, ?, ?);";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_text(stmt, 1, dataset.name.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 2, dataset.path.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 3, dataset.description.c_str(), -1, SQLITE_TRANSIENT);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    return rc == SQLITE_DONE;
}
// 查询指定数据集详细信息
bool ChemicalMLDAO::GetDataset(int dataset_id, Dataset& out_dataset) {
    if (!pImpl->db) return false;
    const std::string sql = "SELECT id, name, path, description FROM datasets WHERE id = ? LIMIT 1;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, dataset_id);
    int rc = sqlite3_step(stmt);
    if (rc == SQLITE_ROW) {
        out_dataset.id = sqlite3_column_int(stmt, 0);
        out_dataset.name = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
        out_dataset.path = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2));
        out_dataset.description = sqlite3_column_type(stmt, 3) == SQLITE_NULL ? "" : reinterpret_cast<const
```

```
char*>(sqlite3_column_text(stmt, 3));
        sqlite3_finalize(stmt);
        return true;
    }
    sqlite3_finalize(stmt);
    return false;
}
// 列出所有管理的数据集
bool ChemicalMLDAO::ListDatasets(std::vector<Dataset>& out_datasets) {
    if (!pImpl->db) return false;
    const std::string sql = "SELECT id, name, path, description FROM datasets ORDER BY id DESC;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    out_datasets.clear();
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        Dataset ds;
        ds.id = sqlite3_column_int(stmt, 0);
        ds.name = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
        ds.path = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2));
        ds.description = sqlite3_column_type(stmt, 3) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 3));
        out_datasets.emplace_back(std::move(ds));
    }
    sqlite3_finalize(stmt);
    return true;
}
// 创建模型版本，模型版本控制
bool ChemicalMLDAO::CreateModelVersion(const ModelVersion& model_ver) {
    if (!pImpl->db) return false;
    const std::string sql = "INSERT INTO model_versions (version_name, create_time, description) VALUES
(?, ?, ?);";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_text(stmt, 1, model_ver.version_name.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 2, model_ver.create_time.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 3, model_ver.description.c_str(), -1, SQLITE_TRANSIENT);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    return rc == SQLITE_DONE;
}
// 获取指定模型版本信息
bool ChemicalMLDAO::GetModelVersion(int version_id, ModelVersion& out_model_ver) {
    if (!pImpl->db) return false;
    const std::string sql = "SELECT id, version_name, create_time, description FROM model_versions WHERE id = ?
LIMIT 1;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, version_id);
    int rc = sqlite3_step(stmt);
    if (rc == SQLITE_ROW) {
        out_model_ver.id = sqlite3_column_int(stmt, 0);
        out_model_ver.version_name = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
        out_model_ver.create_time = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2));
        out_model_ver.description = sqlite3_column_type(stmt, 3) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 3));
        sqlite3_finalize(stmt);
        return true;
    }
    sqlite3_finalize(stmt);
    return false;
}
// 列出所有历史模型版本
bool ChemicalMLDAO::ListModelVersions(std::vector<ModelVersion>& out_model_vers) {
    if (!pImpl->db) return false;
    const std::string sql = "SELECT id, version_name, create_time, description FROM model_versions ORDER BY id
DESC;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    out_model_vers.clear();
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        ModelVersion mv;
        mv.id = sqlite3_column_int(stmt, 0);
        mv.version_name = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1));
        mv.create_time = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2));
        mv.description = sqlite3_column_type(stmt, 3) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 3));
        out_model_vers.emplace_back(std::move(mv));
    }
    sqlite3_finalize(stmt);
    return true;
}
// 更新训练任务中的错误日志信息
bool ChemicalMLDAO::UpdateTrainingErrorLog(int task_id, const std::string& error_log) {
    if (!pImpl->db) return false;
    const std::string sql = "UPDATE training_tasks SET error_log = ? WHERE id = ?;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_text(stmt, 1, error_log.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_int(stmt, 2, task_id);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    return rc == SQLITE_DONE;
}
// 获取训练任务的错误日志信息
bool ChemicalMLDAO::GetTrainingErrorLog(int task_id, std::string& out_error_log) {
    if (!pImpl->db) return false;
    const std::string sql = "SELECT error_log FROM training_tasks WHERE id = ? LIMIT 1;";
    sqlite3_stmt* stmt = pImpl->PrepareStatement(sql);
    if (!stmt) return false;
    sqlite3_bind_int(stmt, 1, task_id);
    int rc = sqlite3_step(stmt);
    if (rc == SQLITE_ROW) {
        out_error_log = sqlite3_column_type(stmt, 0) == SQLITE_NULL ? "" : reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 0));
        sqlite3_finalize(stmt);
        return true;
    }
    sqlite3_finalize(stmt);
    return false;
}
/**
 * 项目名称：化学机器学习应用软件
 * 功能描述：已注册用户输入账号密码完成身份验证登录系统，
 * 实现功能包括账号密码验证、多因素认证、忘记密码重置、登录状态保持、
 * 安全验证码输入、自动登录功能、账号锁定机制、用户身份识别等。
 * 使用技术栈：React 18 + Ant Design 5 + React Router 6 + axios
 */
import React, { useState, useEffect, useRef } from 'react';
import {
  Form,
  Input,
  Button,
  Checkbox,
  Modal,
  message,
  Typography,
  Space,
  Alert,
  Row,
  Col,
  Divider,
  Avatar,
  Tooltip,
} from 'antd';
import {
  UserOutlined,
  LockOutlined,
  SafetyCertificateOutlined,
  ReloadOutlined,
  UnlockOutlined,
  QuestionCircleOutlined,
  SmileOutlined,
} from '@ant-design/icons';
import axios from 'axios';
const API_BASE = 'https://api.chemmlapp.com/v1';
// 本地缓存自动登录与 TOKEN 等 KEY
const STORAGE_KEYS = {
  TOKEN: 'chemml_token',
  USERNAME: 'chemml_username',
  AUTO_LOGIN: 'chemml_auto_login',
  MFA_REQUIRED: 'chemml_mfa_required',
  LOCKOUT_INFO: 'chemml_lockout_info',
};
// 登录失败次数最大限制阈值
const MAX_FAIL_COUNT = 5;
// 登录表单布局配置
const formItemLayout = {
  labelCol: { span: 6 },
  wrapperCol: { span: 16 },
};
const tailFormItemLayout = {
  wrapperCol: { offset: 6, span: 16 },
};
function ChemMLLogin() {
  /** 登录状态 **/
  const [loading, setLoading] = useState(false); // 登录请求加载状态
  const [failCount, setFailCount] = useState(0); // 登录失败计数
  const [locked, setLocked] = useState(false); // 账号是否锁定状态
  const [lockExpire, setLockExpire] = useState(null); // 账号锁定的解锁时间
  const [captchaUrl, setCaptchaUrl] = useState(''); // 验证码图片链接
  const [showCaptcha, setShowCaptcha] = useState(false); // 是否显示验证码输入
  const [autoLogin, setAutoLogin] = useState(false); // 自动登录勾选状态
  /** 多因素认证状态 **/
  const [mfaRequired, setMfaRequired] = useState(false); // 是否需要 MFA
  const [mfaMethod, setMfaMethod] = useState(''); // MFA 类型（短信、邮箱、App OTP 等）
  const [mfaSending, setMfaSending] = useState(false); // MFA 验证码发送中
  const [mfaCode, setMfaCode] = useState(''); // MFA 验证码输入
  /** 忘记密码模态框状态 **/
  const [resetVisible, setResetVisible] = useState(false); // 忘记密码弹窗
  const [resetLoading, setResetLoading] = useState(false); // 忘记密码请求加载
  const [resetStep, setResetStep] = useState(1); // 忘记密码操作步骤
  const [resetEmailOrPhone, setResetEmailOrPhone] = useState(''); // 忘记密码输入的邮箱或手机号
  const [resetVerifyCode, setResetVerifyCode] = useState(''); // 忘记密码验证码
  const [resetNewPassword, setResetNewPassword] = useState(''); // 新密码输入
  const [resetConfirmPassword, setResetConfirmPassword] = useState(''); // 确认新密码
  /** 当前登录用户信息 **/
  const [userInfo, setUserInfo] = useState(null); // 登录成功后保存用户信息
  /** 表单实例 **/
  const [form] = Form.useForm();
  const [mfaForm] = Form.useForm();
  /** 定时更新验证码图片（避免缓存） **/
  const refreshCaptcha = () => {
    setCaptchaUrl(`${API_BASE}/auth/captcha?${Date.now()}`);
  };
  /** 初始化，尝试读取本地缓存进行自动登录 **/
  useEffect(() => {
    refreshCaptcha();
    const storedToken = localStorage.getItem(STORAGE_KEYS.TOKEN);
    const storedUsername = localStorage.getItem(STORAGE_KEYS.USERNAME);
    const storedAutoLogin = localStorage.getItem(STORAGE_KEYS.AUTO_LOGIN) === 'true';
    if (storedToken && storedUsername && storedAutoLogin) {
      setAutoLogin(true);
      verifyToken(storedToken, storedUsername);
    }
    // 账号锁定检查
    checkLockoutInfo();
  }, []);
  /** 检查账号锁定缓存 **/
  const checkLockoutInfo = () => {
    const lockInfoStr = localStorage.getItem(STORAGE_KEYS.LOCKOUT_INFO);
    if (!lockInfoStr) return;
    try {
      const lockInfo = JSON.parse(lockInfoStr);
      if (lockInfo && lockInfo.locked && lockInfo.expireAt) {
        const expireTime = new Date(lockInfo.expireAt).getTime();
        if (expireTime > Date.now()) {
          setLocked(true);
          setLockExpire(lockInfo.expireAt);
          setFailCount(MAX_FAIL_COUNT);
        } else {
          // 锁定时间过期，清除锁定信息
          localStorage.removeItem(STORAGE_KEYS.LOCKOUT_INFO);
          setLocked(false);
          setLockExpire(null);
          setFailCount(0);
        }
      }
    } catch (e) {
      // do nothing
    }
  };
  /** 验证 token 有效性，自动登录 **/
  const verifyToken = async (token, username) => {
    setLoading(true);
    try {
      const resp = await axios.get(`${API_BASE}/auth/verify-token`, {
        headers: { Authorization: `Bearer ${token}` },
      });
      if (resp.data && resp.data.success) {
        // 自动登录成功，设置用户信息
        setUserInfo({ username });
        message.success(`欢迎回来，${username}！自动登录成功`);
      } else {
        // token 无效，清除存储
        localStorage.removeItem(STORAGE_KEYS.TOKEN);
        localStorage.removeItem(STORAGE_KEYS.USERNAME);
        localStorage.removeItem(STORAGE_KEYS.AUTO_LOGIN);
        message.warning('自动登录已过期，请重新登录');
      }
    } catch (error) {
      localStorage.removeItem(STORAGE_KEYS.TOKEN);
      localStorage.removeItem(STORAGE_KEYS.USERNAME);
      localStorage.removeItem(STORAGE_KEYS.AUTO_LOGIN);
      message.error('自动登录失败，请手动登录');
    } finally {
      setLoading(false);
    }
  };
  /** 登录请求处理 **/
  const onFinish = async (values) => {
    if (locked) {
      message.error('账号已被锁定，请稍后再试或联系客服');
      return;
    }
    setLoading(true);
    try {
      // 构造登录参数
      const loginParams = {
        username: values.username.trim(),
        password: values.password,
        captcha: values.captcha,
      };
      const resp = await axios.post(`${API_BASE}/auth/login`, loginParams);
      // 登录成功，判断是否需要 MFA
      if (resp.data) {
        if (resp.data.mfaRequired) {
          // MFA 多因素认证开始
          setMfaRequired(true);
          setMfaMethod(resp.data.mfaMethod || '短信验证码');
          setFailCount(0);
          setShowCaptcha(false);
          setUserInfo({ username: values.username.trim() });
          // 自动登录功能留到登录完全成功后保存
          message.info('需要输入多因素认证验证码');
          sendMfaCode(values.username.trim());
        } else if (resp.data.token) {
          // 不需要 MFA，登录成功，保存 token 和用户
          handleSuccessfulLogin(resp.data.token, values.username.trim());
        } else if (resp.data.accountLocked) {
          // 账号锁定返回
          handleLockout(resp.data.lockDurationMinutes || 30);
        } else {
          // 其他异常情况
          message.error('未知登录错误，请重试');
        }
      } else {
        message.error('响应异常，登录失败');
      }
    } catch (error) {
      handleLoginError(error);
    } finally {
      setLoading(false);
    }
  };
  /** 登陆失败处理 **/
  const handleLoginError = (error) => {
    let msg = '登录失败，请检查账号密码和验证码';
    if (error.response && error.response.data && error.response.data.message) {
      msg = error.response.data.message;
    }
    message.error(msg);
    // 累计失败次数
    const newFailCount = failCount + 1;
    setFailCount(newFailCount);
    if (newFailCount >= MAX_FAIL_COUNT) {
      // 触发账号锁定机制
      handleLockout(30);
      return;
    }
    // 超过 3 次错误，显示验证码输入
    if (newFailCount >= 3) {
      setShowCaptcha(true);
      refreshCaptcha();
    }
  };
  /** 账号锁定处理 **/
  const handleLockout = (durationMinutes) => {
    setLocked(true);
    const expireAt = new Date(Date.now() + durationMinutes * 60 * 1000).toISOString();
    setLockExpire(expireAt);
    localStorage.setItem(
      STORAGE_KEYS.LOCKOUT_INFO,
      JSON.stringify({ locked: true, expireAt })
    );
    message.error(`账号被锁定，${durationMinutes}分钟后自动解锁`);
  };
  /** 登录成功后处理 **/
  const handleSuccessfulLogin = (token, username) => {
    setUserInfo({ username });
    localStorage.setItem(STORAGE_KEYS.TOKEN, token);
    localStorage.setItem(STORAGE_KEYS.USERNAME, username);
    if (autoLogin) {
      localStorage.setItem(STORAGE_KEYS.AUTO_LOGIN, 'true');
    } else {
      localStorage.removeItem(STORAGE_KEYS.AUTO_LOGIN);
    }
    // 清除锁定信息和失败计数
    localStorage.removeItem(STORAGE_KEYS.LOCKOUT_INFO);
    setFailCount(0);
    setLocked(false);
    setLockExpire(null);
    message.success('登录成功，欢迎！');
  };
  /** 发送 MFA 验证码 **/
  const sendMfaCode = async (username) => {
    setMfaSending(true);
    try {
      await axios.post(`${API_BASE}/auth/send-mfa-code`, { username, method: mfaMethod });
      message.success(`多因素认证验证码已发送至您的${mfaMethod}`);
    } catch (error) {
      message.error('MFA 验证码发送失败，请稍后重试');
    } finally {
      setMfaSending(false);
    }
  };
  /** MFA 验证码提交 **/
  const onMfaFinish = async (values) => {
    setLoading(true);
    try {
      const resp = await axios.post(`${API_BASE}/auth/verify-mfa-code`, {
        username: userInfo.username,
        code: values.mfaCode,
      });
      if (resp.data && resp.data.token) {
        // MFA 成功，登录完成
        handleSuccessfulLogin(resp.data.token, userInfo.username);
        setMfaRequired(false);
        mfaForm.resetFields();
      } else if (resp.data && resp.data.error === 'invalid_code') {
        message.error('多因素认证码错误，请重新输入');
      } else if (resp.data && resp.data.accountLocked) {
        handleLockout(resp.data.lockDurationMinutes || 30);
        setMfaRequired(false);
      } else {
        message.error('验证失败，请重试');
      }
    } catch (error) {
      message.error('验证失败，请检查网络或稍后再试');
    } finally {
      setLoading(false);
    }
  };
  /** 退出登录 **/
  const logout = () => {
    setUserInfo(null);
    localStorage.removeItem(STORAGE_KEYS.TOKEN);
    localStorage.removeItem(STORAGE_KEYS.USERNAME);
    localStorage.removeItem(STORAGE_KEYS.AUTO_LOGIN);
    message.info('已退出登录');
    form.resetFields();
    mfaForm.resetFields();
    setMfaRequired(false);
    setShowCaptcha(false);
    setFailCount(0);
    setLocked(false);
    setLockExpire(null);
  };
  /** 自动登录切换 **/
  const onAutoLoginChange = (e) => {
    setAutoLogin(e.target.checked);
  };
  /** 忘记密码操作 **/
  // 发送重置密码验证码
  const sendResetVerifyCode = async () => {
    if (!resetEmailOrPhone || resetEmailOrPhone.trim().length === 0) {
      message.warning('请输入注册的邮箱或手机号');
      return;
    }
    setResetLoading(true);
    try {
      await axios.post(`${API_BASE}/auth/send-reset-code`, {
        contact: resetEmailOrPhone.trim(),
      });
      message.success('验证验证码已发送，请注意查收');
      setResetStep(2);
    } catch (error) {
      message.error('发送验证码失败，请稍后重试');
    } finally {
      setResetLoading(false);
    }
  };
  // 提交重置密码请求
  const submitResetPassword = async () => {
    if (!resetVerifyCode || resetVerifyCode.trim().length === 0) {
      message.warning('请输入收到的验证码');
      return;
    }
    if (!resetNewPassword || resetNewPassword.length < 6) {
      message.warning('请输入至少 6 位的新密码');
      return;
    }
    if (resetNewPassword !== resetConfirmPassword) {
      message.warning('新密码与确认密码不一致');
      return;
    }
    setResetLoading(true);
    try {
      await axios.post(`${API_BASE}/auth/reset-password`, {
        contact: resetEmailOrPhone.trim(),
        verifyCode: resetVerifyCode.trim(),
        newPassword: resetNewPassword,
      });
      message.success('密码重置成功，请使用新密码登录');
      setResetVisible(false);
      // 重置弹窗状态
      setResetStep(1);
      setResetEmailOrPhone('');
      setResetVerifyCode('');
      setResetNewPassword('');
      setResetConfirmPassword('');
    } catch (error) {
      message.error('密码重置失败，请确认验证码或稍后重试');
    } finally {
      setResetLoading(false);
    }
  };
  /** 关闭忘记密码弹窗时重置状态 **/
  const onResetCancel = () => {
    setResetVisible(false);
    setResetStep(1);
    setResetEmailOrPhone('');
    setResetVerifyCode('');
    setResetNewPassword('');
    setResetConfirmPassword('');
    setResetLoading(false);
  };
  /** 锁定倒计时计算 **/
  const [lockCountdown, setLockCountdown] = useState(0);
  useEffect(() => {
    if (locked && lockExpire) {
      const interval = setInterval(() => {
        const expireTime = new Date(lockExpire).getTime();
        const left = Math.max(0, Math.floor((expireTime - Date.now()) / 1000));
        setLockCountdown(left);
        if (left <= 0) {
          setLocked(false);
          setLockExpire(null);
          setFailCount(0);
          localStorage.removeItem(STORAGE_KEYS.LOCKOUT_INFO);
          clearInterval(interval);
          message.info('账号锁定解除，可以重新登录');
          refreshCaptcha();
        }
      }, 1000);
      return () => clearInterval(interval);
    }
  }, [locked, lockExpire]);
  return (
    <Row justify="center" align="middle" style={{ height: '100vh', background: '#f0f2f5' }}>
      <Col xs={22} sm={16} md={12} lg={10} xl={8} xxl={6} style={{ padding: 24, background: '#fff', borderRadius:
```

```
8, boxShadow: '0 4px 12px rgba(0,0,0,0.15)' }}>
        <Typography.Title level={2} style={{ textAlign: 'center', marginBottom: 24 }}>
          化学机器学习应用软件 登录
        </Typography.Title>
        {userInfo ? (
          /** 登录后信息展示区域 **/
          <React.Fragment>
            <Space direction="vertical" style={{ width: '100%' }} size="large" align="center">
              <Avatar size={64} icon={<UserOutlined />} style={{ backgroundColor: '#1890ff' }} />
              <Typography.Text strong style={{ fontSize: 18 }}>
                欢迎您，{userInfo.username}
              </Typography.Text>
              <Typography.Text type="secondary">您已成功登录化学机器学习应用软件系统</Typography.Text>
              <Button type="primary" danger block shape="round" icon={<UnlockOutlined />} onClick={logout}>
                退出登录
              </Button>
            </Space>
          </React.Fragment>
        ) : locked ? (
          /** 锁定提示 **/
          <div style={{ textAlign: 'center' }}>
            <Alert
              type="error"
              showIcon
              icon={<SafetyCertificateOutlined />}
              message="账号已被锁定"
              description={
                <div>
                  由于连续登录失败超过限制，账号被锁定。请在&nbsp;
                  <Typography.Text code>{Math.floor(lockCountdown / 60)}分{lockCountdown % 60}秒
</Typography.Text>&nbsp;后重试，或联系客服解除。
                </div>
              }
              style={{ marginBottom: 24 }}
            />
            <Button type="primary" block onClick={refreshCaptcha} icon={<ReloadOutlined />}>
              刷新验证码
            </Button>
          </div>
        ) : mfaRequired ? (
          /** 多因素认证表单 **/
          <Form
            form={mfaForm}
            name="mfa_login"
            layout="vertical"
            onFinish={onMfaFinish}
            requiredMark={false}
            style={{ maxWidth: '100%' }}
          >
            <Typography.Paragraph type="secondary" style={{ marginBottom: 16, textAlign: 'center' }}>
```

```
      多因素认证：请输入通过<span style={{ color: '#1890ff' }}>{mfaMethod}</span>收到的验证码
    </Typography.Paragraph>
    <Form.Item
      label="认证码"
      name="mfaCode"
      rules={[
        { required: true, message: '请输入多因素认证码' },
        { len: 6, message: '认证码长度应为 6 位' },
      ]}
    >
      <Input
        maxLength={6}
        placeholder="请输入 6 位验证码"
        prefix={<SafetyCertificateOutlined />}
        onChange={(e) => setMfaCode(e.target.value.trim())}
        autoFocus
      />
    </Form.Item>
    <Form.Item {...tailFormItemLayout}>
      <Space>
        <Button
          type="primary"
          htmlType="submit"
          loading={loading}
          disabled={!mfaCode || mfaCode.length !== 6}
          shape="round"
        >
          确认
        </Button>
        <Button
          htmlType="button"
          onClick={() => sendMfaCode(userInfo.username)}
          loading={mfaSending}
          shape="round"
        >
          重新发送验证码
        </Button>
        <Button
          type="link"
          onClick={() => {
            setMfaRequired(false);
            mfaForm.resetFields();
          }}
        >
          取消
        </Button>
      </Space>
    </Form.Item>
  </Form>
) : (
  /** 登录表单 **/
  <Form
    form={form}
    name="login"
    {...formItemLayout}
    onFinish={onFinish}
    initialValues={{ remember: autoLogin }}
    style={{ maxWidth: '100%' }}
    scrollToFirstError
    requiredMark={false}
  >
    <Form.Item
      label="账号"
      name="username"
      rules={[
        { required: true, message: '请输入您的账号' },
        { min: 4, message: '账号长度至少 4 位' },
        { max: 20, message: '账号最长 20 位' },
      ]}
      hasFeedback
    >
      <Input
        placeholder="请输入账号"
        prefix={<UserOutlined />}
        disabled={loading}
        autoComplete="username"
      />
    </Form.Item>
    <Form.Item
      label="密码"
      name="password"
      rules={[
        { required: true, message: '请输入您的密码' },
        { min: 6, message: '密码长度至少 6 位' },
        { max: 32, message: '密码最长 32 位' },
      ]}
      hasFeedback
    >
      <Input.Password
        placeholder="请输入密码"
        prefix={<LockOutlined />}
        disabled={loading}
        autoComplete="current-password"
      />
    </Form.Item>
    {showCaptcha && (
      <Form.Item
        label="验证码"
        name="captcha"
        extra={
           <Tooltip title="看不清楚？点击验证码图片可刷新">
             <img
               src={captchaUrl}
               alt="验证码"
               style={{ cursor: 'pointer', marginTop: 8, borderRadius: 4, border: '1px solid #d9d9d9' }}
               onClick={refreshCaptcha}
               draggable={false}
             />
           </Tooltip>
         }
         rules={[
           { required: true, message: '请输入验证码' },
           { len: 4, message: '验证码长度 4 位' },
         ]}
         hasFeedback
       >
         <Input
           maxLength={4}
           placeholder="请输入验证码"
           disabled={loading}
           prefix={<SafetyCertificateOutlined />}
           autoComplete="off"
         />
       </Form.Item>
      )}
      <Form.Item {...tailFormItemLayout} name="remember" valuePropName="checked">
       <Checkbox checked={autoLogin} onChange={onAutoLoginChange}>
         自动登录
       </Checkbox>
      </Form.Item>
      <Form.Item {...tailFormItemLayout}>
       <Space direction="vertical" style={{ width: '100%' }}>
         <Button
           type="primary"
           htmlType="submit"
           loading={loading}
           block
           shape="round"
         >
           登录
         </Button>
         <Button
           type="link"
           block
           onClick={() => setResetVisible(true)}
           icon={<QuestionCircleOutlined />}
         >
           忘记密码?
         </Button>
       </Space>
```

```
          </Form.Item>
        </Form>
      )}
      <Divider>© 2024 化学机器学习应用软件</Divider>
      {/* 忘记密码弹窗 */}
      <Modal
        title="重置密码"
        open={resetVisible}
        onCancel={onResetCancel}
        destroyOnClose
        centered
      >
        {resetStep === 1 && (
          <Space direction="vertical" style={{ width: '100%' }}>
            <Typography.Paragraph>请输入您注册时绑定的邮箱或手机号，我们将发送验证码帮助您重置密码。
</Typography.Paragraph>
            <Input
              placeholder="邮箱或手机号"
              value={resetEmailOrPhone}
              onChange={(e) => setResetEmailOrPhone(e.target.value.trim())}
              disabled={resetLoading}
              allowClear
              autoFocus
            />
            <Button
              type="primary"
              block
              loading={resetLoading}
              onClick={sendResetVerifyCode}
              disabled={!resetEmailOrPhone}
              shape="round"
            >
              发送验证码
            </Button>
          </Space>
        )}
        {resetStep === 2 && (
          <Form layout="vertical" onFinish={submitResetPassword}>
            <Form.Item
              label="验证码"
              name="verifyCode"
              rules={[
                { required: true, message: '请输入收到的验证码' },
                { len: 6, message: '验证码长度应为 6 位' },
              ]}
              hasFeedback
            >
              <Input
                maxLength={6}
                placeholder="请输入验证码"
```

```
    value={resetVerifyCode}
    onChange={(e) => setResetVerifyCode(e.target.value.trim())}
    disabled={resetLoading}
  />
</Form.Item>
<Form.Item
  label="新密码"
  name="newPassword"
  rules={[
    { required: true, message: '请输入新密码' },
    { min: 6, message: '密码长度至少 6 位' },
    { max: 32, message: '密码最长 32 位' },
  ]}
  hasFeedback
>
  <Input.Password
    placeholder="请输入新密码"
    value={resetNewPassword}
    onChange={(e) => setResetNewPassword(e.target.value)}
    disabled={resetLoading}
  />
</Form.Item>
<Form.Item
  label="确认新密码"
  name="confirmPassword"
  dependencies={['newPassword']}
  hasFeedback
  rules={[
    { required: true, message: '请确认新密码' },
    ({ getFieldValue }) => ({
      validator(_, value) {
        if (!value || getFieldValue('newPassword') === value) {
          return Promise.resolve();
        }
        return Promise.reject(new Error('两次输入的密码不匹配'));
      },
    }),
  ]}
>
  <Input.Password
    placeholder="请确认新密码"
    value={resetConfirmPassword}
    onChange={(e) => setResetConfirmPassword(e.target.value)}
    disabled={resetLoading}
  />
</Form.Item>
<Form.Item>
  <Space>
    <Button
      type="primary"
```

```
                    htmlType="submit"
                    loading={resetLoading}
                    disabled={
                      !resetVerifyCode ||
                      !resetNewPassword ||
                      !resetConfirmPassword ||
                      resetNewPassword !== resetConfirmPassword
                    }
                    shape="round"
                  >
                    重置密码
                  </Button>
                  <Button onClick={() => setResetStep(1)} disabled={resetLoading}>
                    上一步
                  </Button>
                </Space>
              </Form.Item>
            </Form>
          )}
        </Modal>
      </Col>
    </Row>
  );
}
// 主页面组件
export default function ChemicalMLApp() {
  // 侧边栏菜单选中状态
  const [selectedMenu, setSelectedMenu] = useState("tasks");
  return (
    <Layout style={{ minHeight: "100vh" }}>
      {/* 顶部 Header */}
      <Header
        style={{
          color: "#fff",
          fontSize: 20,
          fontWeight: "bold",
          textAlign: "center",
          userSelect: "none",
        }}
      >
        化学机器学习应用软件
      </Header>
      <Layout>
        {/* 左侧菜单 */}
        <Sider width={220} theme="dark">
          <Menu
            theme="dark"
            mode="inline"
            selectedKeys={[selectedMenu]}
            onClick={({ key }) => setSelectedMenu(key)}
            items={[
              {
                key: "tasks",
                icon: <PlayCircleOutlined />,
                label: "训练任务管理",
              },
              {
                key: "config",
                icon: <SettingOutlined />,
                label: "模型训练配置",
              },
              {
                key: "analysis",
                icon: <FileDoneOutlined />,
                label: "训练结果分析",
              },
              {
                key: "logs",
                icon: <FileExclamationOutlined />,
                label: "训练错误日志",
              },
              {
                key: "versions",
                icon: <HistoryOutlined />,
                label: "模型版本控制",
              },
              {
                key: "data",
                icon: <CloudUploadOutlined />,
                label: "数据集选择导入",
              },
            ]}
          />
        </Sider>
        {/* 内容区域，切换不同页面 */}
        <Layout style={{ padding: 24, backgroundColor: "#fff" }}>
          <Content>
            {selectedMenu === "tasks" && <TasksPage />}
            {selectedMenu === "config" && <ConfigPage />}
            {selectedMenu === "analysis" && <AnalysisPage />}
            {selectedMenu === "logs" && <ErrorLogsPage />}
            {selectedMenu === "versions" && <ModelVersionsPage />}
            {selectedMenu === "data" && <DatasetImportPage />}
          </Content>
        </Layout>
      </Layout>
    </Layout>
  );
}
// ============================ 训练任务管理页面 ============================
function TasksPage() {
  // 任务列表数据和加载状态
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(false);
  // 新建任务对话框显示状态
  const [createModalVisible, setCreateModalVisible] = useState(false);
  // 当前操作任务 ID
  const [currentTaskId, setCurrentTaskId] = useState(null);
  // 刷新任务列表
  const fetchTasks = useCallback(() => {
    setLoading(true);
    api
      .fetchTasks()
      .then((data) => {
        setTasks(data.tasks || []);
      })
      .finally(() => setLoading(false));
  }, []);
  useEffect(() => {
    fetchTasks();
  }, [fetchTasks]);
  // 删除任务确认
  const onDeleteTask = (taskId, taskName) => {
    confirm({
      title: `确认删除训练任务 "${taskName}" 吗？ `,
      icon: <ExclamationCircleOutlined />,
      okText: "删除",
      okType: "danger",
      cancelText: "取消",
      onOk() {
        return api
          .deleteTask(taskId)
          .then(() => {
            message.success(`任务 "${taskName}" 删除成功`);
            fetchTasks();
          })
          .catch(() => message.error("删除失败"));
      },
    });
  };
  // 启动训练任务
  const onStartTraining = (taskId) => {
    message.loading({content:'启动中...', key:'startTrain'})
    api
      .startTraining(taskId)
      .then(() => {
        message.success({ content:'启动成功', key:'startTrain', duration:2 });
        fetchTasks();
      })
      .catch(() => {
```

```
      message.error({content:'启动失败', key:'startTrain', duration:2});
    });
  };
  // 表格列配置
  const columns = [
    {
      title: "任务名称",
      dataIndex: "name",
      key: "name",
      ellipsis: true,
    },
    {
      title: "状态",
      dataIndex: "status",
      key: "status",
      width: 110,
      render: (status) => {
        let color = "default";
        let text = "未知";
        switch (status) {
          case "pending":
            color = "orange";
            text = "待启动";
            break;
          case "running":
            color = "blue";
            text = "训练中";
            break;
          case "completed":
            color = "green";
            text = "已完成";
            break;
          case "failed":
            color = "red";
            text = "失败";
            break;
          default:
            color = "default";
            text = status;
            break;
        }
        return <Tag color={color}>{text}</Tag>;
      },
    },
    {
      title: "创建时间",
      dataIndex: "createdAt",
      key: "createdAt",
      width: 180,
      sorter: (a, b) =>
        new Date(a.createdAt).getTime() - new Date(b.createdAt).getTime(),
      render: (text) => {
        if (!text) return "-";
        const d = new Date(text);
        return d.toLocaleString();
      },
    },
    {
      title: "操作",
      key: "actions",
      width: 240,
      fixed: "right",
      render: (_, record) => (
        <Space size="middle">
          <Button
            type="primary"
            onClick={() => setCurrentTaskId(record.id)}
            size="small"
          >
            编辑
          </Button>
          <Button
            type="default"
            onClick={() => onStartTraining(record.id)}
            disabled={record.status === "running"}
            size="small"
            icon={<PlayCircleOutlined />}
          >
            启动训练
          </Button>
          <Button
            type="danger"
            onClick={() => onDeleteTask(record.id, record.name)}
            size="small"
            icon={<DeleteOutlined />}
          >
            删除
          </Button>
        </Space>
      ),
    },
  ];
  return (
    <Card
      title="训练任务管理"
      extra={
        <Button
          type="primary"
          icon={<PlusOutlined />}
          onClick={() => setCreateModalVisible(true)}
        >
          新建任务
        </Button>
      }
    >
      <Table
        rowKey="id"
        loading={loading}
        pagination={{ pageSize: 8 }}
        columns={columns}
        dataSource={tasks}
        scroll={{ x: 800 }}
        locale={{ emptyText: "暂无训练任务" }}
      />
      {/* 新建/编辑任务模态框 */}
      {(createModalVisible || currentTaskId) && (
        <TaskFormModal
          visible={createModalVisible || Boolean(currentTaskId)}
          taskId={currentTaskId}
          onCancel={() => {
            setCreateModalVisible(false);
            setCurrentTaskId(null);
            fetchTasks();
          }}
          onSuccess={() => {
            setCreateModalVisible(false);
            setCurrentTaskId(null);
            fetchTasks();
          }}
        />
      )}
    </Card>
  );
}
// 任务表单模态框（新建/编辑）
function TaskFormModal({ visible, onCancel, onSuccess, taskId }) {
  const [form] = Form.useForm();
  const [loading, setLoading] = useState(false);
  const [datasets, setDatasets] = useState([]);
  const [loadingDatasets, setLoadingDatasets] = useState(false);
  // 任务详情（编辑时）
  const [taskDetail, setTaskDetail] = useState(null);
  // 加载数据集列表
  useEffect(() => {
    setLoadingDatasets(true);
    api
      .fetchDatasets()
      .then((data) => {
        setDatasets(data.datasets || []);
      })
      .finally(() => setLoadingDatasets(false));
  }, []);
  // 编辑时获取任务详情
  useEffect(() => {
    if (!taskId) {
      setTaskDetail(null);
      form.resetFields();
      return;
    }
    setLoading(true);
    api
      .fetchTaskDetail(taskId)
      .then((data) => {
        setTaskDetail(data.task);
        // 填充表单
        form.setFieldsValue({
          name: data.task.name,
          description: data.task.description,
          datasetId: data.task.datasetId,
          parameters: data.task.parameters || {},
        });
      })
      .finally(() => setLoading(false));
  }, [taskId, form]);
  // 提交保存表单
  const onFinish = (values) => {
    setLoading(true);
    const payload = {
      name: values.name,
      description: values.description,
      datasetId: values.datasetId,
      parameters: values.parameters,
    };
    const request = taskId
      ? api.updateTask(taskId, payload)
      : api.createTask(payload);
    request
      .then(() => {
        message.success(`任务${taskId ? "更新" : "创建"}成功`);
        onSuccess();
      })
      .catch(() => {
        message.error(`任务${taskId ? "更新" : "创建"}失败`);
      })
      .finally(() => setLoading(false));
  };
  // 监听表单参数，方便调试展示
  const params = Form.useWatch("parameters", form) || {};
  return (
    <Modal
```

```
    open={visible}
    title={taskId ? "编辑训练任务" : "新建训练任务"}
    onCancel={onCancel}
    width={640}
    maskClosable={false}
    destroyOnClose
  >
    <Form
      form={form}
      layout="vertical"
      onFinish={onFinish}
      initialValues={{
        parameters: {
          learning_rate: 0.01,
          batch_size: 32,
          epochs: 10,
        },
      }}
    >
      {/* 任务名称 */}
      <Form.Item
        label="任务名称"
        name="name"
        rules={[
          { required: true, message: "请输入任务名称" },
          { max: 50, message: "最多输入 50 个字符" },
        ]}
      >
        <Input placeholder="请输入任务名称" maxLength={50} />
      </Form.Item>
      {/* 任务描述 */}
      <Form.Item
        label="任务描述"
        name="description"
        rules={[{ max: 200, message: "最多输入 200 个字符" }]}
      >
        <Input.TextArea
          placeholder="任务描述（选填）"
          rows={3}
          maxLength={200}
          showCount
        />
      </Form.Item>
      {/* 数据集选择 */}
      <Form.Item
        label="选择数据集"
        name="datasetId"
        rules={[{ required: true, message: "请选择数据集" }]}
      >
        <Select
```

```
          loading={loadingDatasets}
          showSearch
          placeholder="选择训练用的数据集"
          optionFilterProp="children"
          filterOption={(input, option) =>
            (option?.label ?? "")
              .toLowerCase()
              .includes(input.toLowerCase())
          }
          options={datasets.map((ds) => ({
            label: `${ds.name} (${ds.size}条)`,
            value: ds.id,
          }))}
        />
      </Form.Item>
      {/* 训练参数 */}
      <Divider orientation="left">训练参数配置</Divider>
      {/* 训练参数动态表单 */}
      <ParameterFields form={form} name="parameters" />
      <Divider />
      {/* 参数预览 */}
      <div>
        <Text strong>当前参数预览:</Text>
        <pre
          style={{
            backgroundColor: "#f7f7f7",
            borderRadius: 4,
            padding: 12,
            maxHeight: 140,
            overflowY: "auto",
          }}
        >
          {JSON.stringify(params, null, 2)}
        </pre>
      </div>
      {/* 提交按钮 */}
      <Form.Item style={{ marginTop: 24, textAlign: "right" }}>
        <Button onClick={onCancel} style={{ marginRight: 8 }}>
          取消
        </Button>
        <Button type="primary" htmlType="submit" loading={loading}>
          保存
        </Button>
      </Form.Item>
    </Form>
  </Modal>
 );
}
// 参数输入字段组件，一般 ML 训练的参数较多，支持增加、删除与调节
// 这里用 Form.List 实现动态键值对配置，支持数字和字符串混合
```

```
function ParameterFields({ form, name }) {
  return (
    <Form.List name={name}>
      {(fields, { add, remove }) => (
        <>
          {fields.map(({ key, name: paramName, ...restField }) => (
            <Row
              gutter={8}
              key={key}
              style={{
                paddingBottom: 8,
                alignItems: "center",
              }}
            >
              {/* 参数名 */}
              <Col span={10}>
                <Form.Item
                  {...restField}
                  name={[paramName, "key"]}
                  rules={[
                    { required: true, message: "请输入参数名称" },
                    {
                      pattern: /^[a-zA-Z0-9_\-]+$/,
                      message: "参数名只能由字母数字下划线和-组成",
                    },
                  ]}
                >
                  <Input placeholder="参数名" maxLength={30} />
                </Form.Item>
              </Col>
              {/* 参数值 */}
              <Col span={10}>
                <Form.Item
                  {...restField}
                  name={[paramName, "value"]}
                  rules={[{ required: true, message: "请输入参数值" }]}
                >
                  <Input placeholder="参数值" />
                </Form.Item>
              </Col>
              {/* 删除按钮 */}
              <Col span={4}>
                <Button
                  danger
                  onClick={() => remove(paramName)}
                  type="primary"
                >
                  删除
                </Button>
              </Col>
          </Row>
        ))}
        {/* 添加新参数 */}
        <Form.Item>
          <Button
            type="dashed"
            onClick={() => add({ key: "", value: "" })}
            block
            icon={<PlusOutlined />}
          >
            添加参数
          </Button>
        </Form.Item>
      </>
    )}
  </Form.List>
 );
}
// ========================== 模型训练配置页面 ==========================
// 该页面集中展示当前任务所选参数的高阶配置，可视化调节和参数模板管理
function ConfigPage() {
 const [selectedTask, setSelectedTask] = useState(null);
 const [tasks, setTasks] = useState([]);
 const [loadingTasks, setLoadingTasks] = useState(false);
 useEffect(() => {
  loadTasks();
 }, []);
 // 加载任务列表
 const loadTasks = () => {
  setLoadingTasks(true);
  api
   .fetchTasks()
   .then((data) => setTasks(data.tasks || []))
   .finally(() => setLoadingTasks(false));
 };
 return (
  <Card title="模型训练配置管理">
   <Row gutter={16} align="middle" style={{ marginBottom: 20 }}>
    <Col flex="320px">
     <Select
      style={{ width: "100%" }}
      allowClear
      showSearch
      placeholder="请选择训练任务"
      loading={loadingTasks}
      optionFilterProp="children"
      onChange={(val) => {
       const task = tasks.find((t) => t.id === val);
       setSelectedTask(task || null);
      }}
```

```
              filterOption={(input, option) =>
                (option?.children ?? "")
                  .toLowerCase()
                  .includes(input.toLowerCase())
              }
              value={selectedTask?.id || undefined}
            >
              {tasks.map((t) => (
                <Option key={t.id} value={t.id}>
                  {t.name}
                </Option>
              ))}
            </Select>
          </Col>
        </Row>
        {selectedTask ? (
          <TrainingConfigEditor task={selectedTask} onUpdate={loadTasks} />
        ) : (
          <Text type="secondary">请选择左侧或上方任务进行配置编辑</Text>
        )}
      </Card>
  );
}
// 高级训练参数配置编辑器：提供滑动条、开关、数值输入等富交互界面
function TrainingConfigEditor({ task, onUpdate }) {
  const [form] = Form.useForm();
  const [saving, setSaving] = useState(false);
  // 初始加载任务参数
  useEffect(() => {
    api.fetchTaskDetail(task.id).then(({ task: detail }) => {
      // 解析参数，转成 key-value 形式
      const parameters = detail.parameters || {};
      // 如果参数是数组 KV，则转为对象
      // 这里让参数保持扁平 Map 结构
      const normalizedParams = {};
      if (Array.isArray(parameters)) {
        parameters.forEach((p) => {
          normalizedParams[p.key] = p.value;
        });
      } else {
        Object.entries(parameters).forEach(([k, v]) => {
          normalizedParams[k] = v;
        });
      }
      form.setFieldsValue({
        ...normalizedParams,
      });
    });
  }, [task, form]);
  // 提交保存训练配置
```

```
const onFinish = (values) => {
  setSaving(true);
  // 封装为 {parameters: {...}} 格式提交
  api
    .updateTask(task.id, { parameters: values })
    .then(() => {
      message.success("训练参数配置保存成功");
      onUpdate();
    })
    .catch(() => {
      message.error("训练参数配置保存失败");
    })
    .finally(() => setSaving(false));
};
return (
  <Form
    form={form}
    layout="vertical"
    onFinish={onFinish}
    style={{ maxWidth: 600 }}
    initialValues={{
      learning_rate: 0.01,
      batch_size: 32,
      epochs: 10,
      dropout: 0.5,
      weight_decay: 0,
      momentum: 0.9,
      optimizer: "adam",
      use_augmentation: false,
    }}
  >
    <Row gutter={16}>
      {/* 学习率 */}
      <Col span={12}>
        <Form.Item
          label="学习率 (learning_rate)"
          name="learning_rate"
          rules={[
            { required: true, message: "请输入学习率" },
            {
              type: "number",
              min: 0,
              max: 1,
              transform: (value) => Number(value),
              message: "请输入 0-1 之间的数字",
            },
          ]}
        >
          <Input type="number" step={0.0001} min={0} max={1} />
        </Form.Item>
```

```
</Col>
{/* 批大小 */}
<Col span={12}>
  <Form.Item
    label="批大小 (batch_size)"
    name="batch_size"
    rules={[
      { required: true, message: "请输入批大小" },
      {
        type: "number",
        min: 1,
        max: 1024,
        transform: (value) => Number(value),
        message: "请输入 1-1024 之间的整数",
      },
    ]}
  >
    <Input type="number" step={1} min={1} max={1024} />
  </Form.Item>
</Col>
{/* 迭代次数 */}
<Col span={12}>
  <Form.Item
    label="迭代次数 (epochs)"
    name="epochs"
    rules={[
      { required: true, message: "请输入迭代次数" },
      {
        type: "number",
        min: 1,
        max: 10000,
        transform: (value) => Number(value),
        message: "请输入 1-10000 之间的整数",
      },
    ]}
  >
    <Input type="number" step={1} min={1} max={10000} />
  </Form.Item>
</Col>
{/* dropout 概率 */}
<Col span={12}>
  <Form.Item
    label="Dropout 概率 (dropout)"
    name="dropout"
    rules={[
      { required: true, message: "请输入 Dropout 概率" },
      {
        type: "number",
        min: 0,
        max: 1,
        transform: (value) => Number(value),
        message: "请输入 0-1 之间的数字",
      },
    ]}
  >
    <Input type="number" step={0.01} min={0} max={1} />
  </Form.Item>
</Col>
{/* 权重衰减 */}
<Col span={12}>
  <Form.Item
    label="权重衰减 (weight_decay)"
    name="weight_decay"
    rules={[
      {
        type: "number",
        min: 0,
        transform: (value) => Number(value),
        message: "请输入 0 及以上的数字",
      },
    ]}
  >
    <Input type="number" step={0.0001} min={0} />
  </Form.Item>
</Col>
{/* 动量 */}
<Col span={12}>
  <Form.Item
    label="动量 (momentum)"
    name="momentum"
    rules={[
      {
        type: "number",
        min: 0,
        max: 1,
        transform: (value) => Number(value),
        message: "请输入 0-1 之间的数字",
      },
    ]}
  >
    <Input type="number" step={0.01} min={0} max={1} />
  </Form.Item>
</Col>
{/* 优化器选择 */}
<Col span={12}>
  <Form.Item label="优化器 (optimizer)" name="optimizer">
    <Select>
      <Option value="adam">Adam</Option>
      <Option value="sgd">SGD</Option>
      <Option value="rmsprop">RMSprop</Option>
```

```
            <Option value="adagrad">Adagrad</Option>
          </Select>
        </Form.Item>
      </Col>
      {/* 是否使用数据增强 */}
      <Col span={12}>
        <Form.Item
          label="是否使用数据增强"
          name="use_augmentation"
          valuePropName="checked"
        >
          <Select>
            <Option value={true}>是</Option>
            <Option value={false}>否</Option>
          </Select>
        </Form.Item>
      </Col>
    </Row>
    <Form.Item style={{ marginTop: 24, textAlign: "right" }}>
      <Button
        type="primary"
        htmlType="submit"
        loading={saving}
        disabled={saving}
      >
        保存配置
      </Button>
    </Form.Item>
  </Form>
 );
}
// ============================ 训练结果分析页面 ============================
// 该页面展示训练过程的指标趋势图、结果分布、评估报告等
// 这里用简单表格和 Progress 条和卡片展示，真实可引入图表库如 echarts/recharts
function AnalysisPage() {
 const [tasks, setTasks] = useState([]);
 const [selectedTaskId, setSelectedTaskId] = useState(null);
 const [results, setResults] = useState(null);
 const [loadingTasks, setLoadingTasks] = useState(false);
 const [loadingResults, setLoadingResults] = useState(false);
 useEffect(() => {
   loadTasks();
 }, []);
 useEffect(() => {
   if (selectedTaskId) {
     loadResults(selectedTaskId);
   } else {
     setResults(null);
   }
 }, [selectedTaskId]);
```

```
// 加载任务列表
const loadTasks = () => {
  setLoadingTasks(true);
  api
    .fetchTasks()
    .then((data) => setTasks(data.tasks || []))
    .finally(() => setLoadingTasks(false));
};
// 加载训练结果详情
const loadResults = (taskId) => {
  setLoadingResults(true);
  api
    .fetchTrainingResults(taskId)
    .then((data) => setResults(data.results || null))
    .finally(() => setLoadingResults(false));
};
return (
  <Card title="训练结果分析" style={{ minHeight: 480 }}>
    <Row>
      <Col span={6}>
        {/* 任务选择 */}
        <Select
          placeholder="请选择训练任务"
          loading={loadingTasks}
          style={{ width: "100%" }}
          onChange={setSelectedTaskId}
          allowClear
          optionFilterProp="children"
          filterOption={(input, option) =>
            (option?.children ?? "")
              .toLowerCase()
              .includes(input.toLowerCase())
          }
          value={selectedTaskId || undefined}
        >
          {tasks.map((t) => (
            <Option key={t.id} value={t.id}>
              {t.name}
            </Option>
          ))}
        </Select>
      </Col>
    </Row>
    <Divider />
    {selectedTaskId && (
      <>
        {loadingResults ? (
          <div style={{ textAlign: "center", padding: 40 }}>
            <SyncOutlined spin style={{ fontSize: 24 }} />
            <div>加载训练结果中...</div>
            </div>
          ) : results ? (
            <ResultOverview results={results} />
          ) : (
            <Text type="secondary" style={{ padding: 24, display: "block" }}>
              暂无训练结果数据
            </Text>
          )}
        </>
      )}
    </Card>
  );
}
function ResultOverview({ results }) {
  // 典型指标：准确率，损失值，训练用时
  // results = {
  //   accuracy: 0.85,
  //   loss: 0.35,
  //   epochs: 20,
  //   history: [{epoch:1, acc:0.5, loss:0.9}, ...],
  //   metrics: {precision: 0.8, recall: 0.78, f1: 0.79}
  // }
  const { accuracy, loss, epochs, history = [], metrics = {} } = results;
  return (
    <>
      <Row gutter={24}>
        {/* 准确率 */}
        <Col span={6}>
          <Card bordered={false} style={{ textAlign: "center" }}>
            <Text strong>准确率 (Accuracy)</Text>
            <Progress
              percent={Number((accuracy * 100).toFixed(2))}
              status="active"
              strokeColor="#52c41a"
              format={(percent) => `${percent}%`}
            />
          </Card>
        </Col>
        {/* 损失值 */}
        <Col span={6}>
          <Card bordered={false} style={{ textAlign: "center" }}>
            <Text strong>损失值 (Loss)</Text>
            <Progress
              percent={Number(
                Math.max(0, Math.min(100, 100 - loss * 100)).toFixed(0)
              )}
              status="exception"
              strokeColor="#eb2f96"
              format={() => loss.toFixed(4)}
            />
          </Card>
        </Col>
        {/* 迭代次数 */}
        <Col span={6}>
          <Card bordered={false} style={{ textAlign: "center" }}>
            <Text strong>迭代次数 (Epochs)</Text>
            <div style={{ fontSize: 32, fontWeight: "bold", marginTop: 12 }}>
              {epochs}
            </div>
          </Card>
        </Col>
        {/* 精确率/召回率/F1 分数 */}
        <Col span={6}>
          <Card bordered={false} style={{ textAlign: "center" }}>
            <Text strong>评估指标</Text>
            <div style={{ marginTop: 12, fontSize: 14 }}>
              精确率:{" "}
              <Text code>
                {metrics.precision !== undefined
                  ? metrics.precision.toFixed(3)
                  : "-"}
              </Text>
              <br />
              召回率:{" "}
              <Text code>
                {metrics.recall !== undefined ? metrics.recall.toFixed(3) : "-"}
              </Text>
              <br />
              F1 分数:{" "}
              <Text code>
                {metrics.f1 !== undefined ? metrics.f1.toFixed(3) : "-"}
              </Text>
            </div>
          </Card>
        </Col>
      </Row>
      <Divider />
      <Card title="训练趋势曲线（示意图）" style={{ minHeight: 240 }}>
        <TrendChart history={history} />
      </Card>
    </>
  );
}
function TrendChart({ history }) {
  // history 格式: [{epoch, acc, loss}, ...]
  if (!history || history.length === 0)
    return <Text type="secondary">无训练记录数据</Text>;
  // 取最近 20 个 epoch 数据
  const showData = history.slice(-20);
  // x 轴标签
```

```
const epochs = showData.map((h) => h.epoch);
// 准确率数据
const accs = showData.map((h) => h.acc);
// 损失数据
const losses = showData.map((h) => h.loss);
// 计算最大值与最小值（简单比例缩放）
const maxAcc = Math.max(...accs, 1);
const minLoss = Math.min(...losses, 0);
const maxLoss = Math.max(...losses, 1);
// 简单折线图用 SVG 绘制，横轴为 epoch，纵轴为数值比例
const width = 600;
const height = 180;
const margin = { top: 20, bottom: 30, left: 40, right: 20 };
// 转换坐标点
function toPoint(index, value, min, max) {
  const x =
    margin.left + ((width - margin.left - margin.right) / (epochs.length - 1)) * index;
  const y = margin.top + ((max - value) / (max - min)) * (height - margin.top - margin.bottom);
  return `${x},${y}`;
}
const accPoints = accs
  .map((acc, idx) => toPoint(idx, acc, 0, maxAcc))
  .join(" ");
const lossPoints = losses
  .map((loss, idx) => toPoint(idx, loss, minLoss, maxLoss))
  .join(" ");
return (
  <svg width={width} height={height} style={{ width: "100%" }}>
    {/* 背景 */}
    <rect width={width} height={height} fill="#fafafa" />
    {/* 轴线 */}
    <line
      x1={margin.left}
      y1={height - margin.bottom}
      x2={width - margin.right}
      y2={height - margin.bottom}
      stroke="#ccc"
    />
    <line
      x1={margin.left}
      y1={margin.top}
      x2={margin.left}
      y2={height - margin.bottom}
      stroke="#ccc"
    />
    {/* 准确率折线 */}
    <polyline
      points={accPoints}
      fill="none"
      stroke="#52c41a"
```

```
        strokeWidth={2}
        style={{ transition: "all 0.3s ease" }}
      />
      {/* 损失折线 */}
      <polyline
        points={lossPoints}
        fill="none"
        stroke="#eb2f96"
        strokeWidth={2}
        style={{ transition: "all 0.3s ease" }}
      />
      {/* 点标记 */}
      {accs.map((acc, idx) => {
        const [x, y] = accPoints.split(" ")[idx].split(",");
        return (
          <circle key={"acc-" + idx} cx={x} cy={y} r={3} fill="#52c41a" />
        );
      })}
      {losses.map((loss, idx) => {
        const [x, y] = lossPoints.split(" ")[idx].split(",");
        return (
          <circle key={"loss-" + idx} cx={x} cy={y} r={3} fill="#eb2f96" />
        );
      })}
      {/* 图例 */}
      <text x={width - margin.right - 140} y={margin.top + 20} fill="#52c41a" fontSize={14}>
        准确率 (Accuracy)
      </text>
      <text x={width - margin.right - 140} y={margin.top + 40} fill="#eb2f96" fontSize={14}>
        损失值 (Loss)
      </text>
    </svg>
  );
}
// =========================== 训练错误日志页面 ============================
// 显示选中任务的训练错误日志，支持过滤和分页
function ErrorLogsPage() {
  const [tasks, setTasks] = useState([]);
  const [selectedTaskId, setSelectedTaskId] = useState(null);
  const [logs, setLogs] = useState([]);
  const [loadingLogs, setLoadingLogs] = useState(false);
  useEffect(() => {
    loadTasks();
  }, []);
  useEffect(() => {
    if (selectedTaskId) {
      loadLogs(selectedTaskId);
    } else {
      setLogs([]);
    }
  }, [selectedTaskId]);
  // 任务列表加载
  const loadTasks = () => {
    api.fetchTasks().then((data) => setTasks(data.tasks || []));
  };
  // 日志加载
  const loadLogs = (taskId) => {
    setLoadingLogs(true);
    api
      .fetchErrorLogs(taskId)
      .then((data) => setLogs(data.logs || []))
      .finally(() => setLoadingLogs(false));
  };
  // 表格列
  const columns = [
    {
      title: "时间",
      dataIndex: "timestamp",
      key: "timestamp",
      width: 180
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <mutex>
#include <shared_mutex>
#include <regex>
#include <memory>
// 化学机器学习应用后端控制层代码
// 包含数据上传导入、预处理、分类管理、质量检测、标签管理、权限管理等功能
struct ChemicalData {
    std::string id;                    // 数据唯一标识
    std::string rawData;               // 原始数据字符串，可能是多种格式
    std::string format;                // 数据格式，如 csv,json,sdf 等
    std::string category;              // 数据分类，如有机,无机等
    std::unordered_set<std::string> tags; // 数据标签集合
    bool qualityCheckPassed;           // 质量检测结果
    ChemicalData() : qualityCheckPassed(false) {}
};
// 枚举错误码
enum class ErrorCode {
    SUCCESS = 0,
    INVALID_DATA_FORMAT,
    UPLOAD_FAILED,
    NOT_FOUND,
    PERMISSION_DENIED,
    INVALID_PARAMETER,
    QUALITY_CHECK_FAILED,
    DUPLICATE_DATA,
    UNKNOWN_ERROR
};
// 权限类型
enum class AccessLevel {
    NONE = 0,
    READ = 1,
    WRITE = 2,
    ADMIN = 3
};
// 用户权限管理（简化）
class PermissionManager {
private:
    std::unordered_map<std::string, AccessLevel> userPermissions; // 用户 ID -> 权限级别
    std::shared_mutex permMutex;
public:
    PermissionManager() {
        // 初始化系统管理员权限
        userPermissions["admin"] = AccessLevel::ADMIN;
    }
    void setPermission(const std::string& userId, AccessLevel level) {
        std::unique_lock lock(permMutex);
        userPermissions[userId] = level;
    }
    AccessLevel getPermission(const std::string& userId) {
        std::shared_lock lock(permMutex);
        auto it = userPermissions.find(userId);
        if (it != userPermissions.end()) return it->second;
        return AccessLevel::NONE;
    }
    bool hasWritePermission(const std::string& userId) {
        AccessLevel level = getPermission(userId);
        return level == AccessLevel::WRITE || level == AccessLevel::ADMIN;
    }
    bool hasReadPermission(const std::string& userId) {
        AccessLevel level = getPermission(userId);
        return level == AccessLevel::READ || level == AccessLevel::WRITE || level == AccessLevel::ADMIN;
    }
    bool hasAdminPermission(const std::string& userId) {
        return getPermission(userId) == AccessLevel::ADMIN;
    }
};
// 数据格式转换工具类
class DataFormatConverter {
public:
    static std::string convert(const std::string& sourceData, const std::string& sourceFormat, const std::string&
targetFormat) {
        if (sourceFormat == targetFormat) return sourceData;
        return "[converted:" + targetFormat + "]" + sourceData;
    }
};
```

```
// 数据质量检测工具类
class DataQualityChecker {
public:
    // 简单质量检测: 检查数据不为空, 格式符合预期, 数据有效性简单验证
    static bool checkQuality(const ChemicalData& data) {
        if (data.rawData.empty()) return false;
        if (!isValidFormat(data.format)) return false;
        if (data.rawData.length() < 10) return false;
        if (!std::regex_match(data.rawData, std::regex("[\\w\\s\\-\\,\\.\\;\\:]+"))) return false;
        return true;
    }
private:
    static bool isValidFormat(const std::string& format) {
        static std::unordered_set<std::string> validFormats = {
            "csv", "json", "sdf", "mol", "txt"
        };
        return validFormats.count(format) > 0;
    }
};
// 数据标签管理类
class DataTagManager {
public:
    void addTag(ChemicalData& data, const std::string& tag) {
        data.tags.insert(tag);
    }
    void removeTag(ChemicalData& data, const std::string& tag) {
        data.tags.erase(tag);
    }
    bool hasTag(const ChemicalData& data, const std::string& tag) {
        return data.tags.find(tag) != data.tags.end();
    }
};
// 数据分类管理类
class DataCategoryManager {
private:
    std::unordered_set<std::string> validCategories;
public:
    DataCategoryManager() {
        validCategories = {"有机", "无机", "高分子", "催化剂", "纳米材料", "药物化学"};
    }
    bool isValidCategory(const std::string& category) {
        return validCategories.count(category) > 0;
    }
    std::unordered_set<std::string> getCategories() {
        return validCategories;
    }
};
// 主数据仓库类（线程安全）
class ChemicalDataRepository {
private:
    std::unordered_map<std::string, ChemicalData> dataMap; // id -> data
    mutable std::shared_mutex dataMutex;
public:
    bool exists(const std::string& id) const {
        std::shared_lock lock(dataMutex);
        return dataMap.find(id) != dataMap.end();
    }
    bool addData(const ChemicalData& data) {
        std::unique_lock lock(dataMutex);
        if (dataMap.find(data.id) != dataMap.end()) return false;
        dataMap[data.id] = data;
        return true;
    }
    bool updateData(const ChemicalData& data) {
        std::unique_lock lock(dataMutex);
        auto it = dataMap.find(data.id);
        if (it == dataMap.end()) return false;
        it->second = data;
        return true;
    }
    bool removeData(const std::string& id) {
        std::unique_lock lock(dataMutex);
        return dataMap.erase(id) > 0;
    }
    ChemicalData getData(const std::string& id) const {
        std::shared_lock lock(dataMutex);
        auto it = dataMap.find(id);
        if (it == dataMap.end()) return ChemicalData{};
        return it->second;
    }
    std::vector<ChemicalData> listDataByCategory(const std::string& category) const {
        std::vector<ChemicalData> result;
        std::shared_lock lock(dataMutex);
        for (const auto& [id, data] : dataMap) {
            if (data.category == category) result.push_back(data);
        }
        return result;
    }
};
// 数据校验工具类
class DataValidator {
public:
    static bool validateId(const std::string& id) {
        if (id.empty()) return false;
        // id 要求仅字母数字和下划线,长度 1-32
        return std::regex_match(id, std::regex("[a-zA-Z0-9_]{1,32}"));
    }
    static bool validateFormat(const std::string& format) {
        static std::unordered_set<std::string> validFormats = {"csv","json","sdf","mol","txt"};
        return validFormats.count(format) > 0;
    }
    static bool validateCategory(const std::string& category) {
        static std::unordered_set<std::string> validCategories = {"有机","无机","高分子","催化剂","纳米材料","药物
化学"};
        return validCategories.count(category) > 0;
    }
    static bool validateTags(const std::unordered_set<std::string>& tags) {
        for (const auto& tag : tags) {
            if (tag.empty() || tag.length() > 16) return false;
            if (!std::regex_match(tag, std::regex("[\\w\\-]+"))) return false;
        }
        return true;
    }
    static bool validateRawData(const std::string& data) {
        return !data.empty();
    }
};
// 控制层主类，提供外部接口
class ChemicalMLController {
private:
    ChemicalDataRepository repository;
    PermissionManager permManager;
    DataCategoryManager categoryManager;
    DataTagManager tagManager;
public:
    ChemicalMLController() {}
    // 上传数据接口
    ErrorCode uploadData(const std::string& userId,
                         const std::string& dataId,
                         const std::string& rawData,
                         const std::string& dataFormat,
                         const std::string& category,
                         const std::unordered_set<std::string>& tags)
    {
        // 权限检查
        if (!permManager.hasWritePermission(userId)) return ErrorCode::PERMISSION_DENIED;
        // 校验参数
        if (!DataValidator::validateId(dataId)) return ErrorCode::INVALID_PARAMETER;
        if (!DataValidator::validateFormat(dataFormat)) return ErrorCode::INVALID_DATA_FORMAT;
        if (!categoryManager.isValidCategory(category)) return ErrorCode::INVALID_PARAMETER;
        if (!DataValidator::validateTags(tags)) return ErrorCode::INVALID_PARAMETER;
        if (!DataValidator::validateRawData(rawData)) return ErrorCode::INVALID_PARAMETER;
        if (repository.exists(dataId)) return ErrorCode::DUPLICATE_DATA;
        // 构造数据对象
        ChemicalData data;
        data.id = dataId;
        data.rawData = rawData;
        data.format = dataFormat;
        data.category = category;
        data.tags = tags;
        // 质量检测
        data.qualityCheckPassed = DataQualityChecker::checkQuality(data);
        if (!data.qualityCheckPassed) return ErrorCode::QUALITY_CHECK_FAILED;
        // 存储数据
        if (!repository.addData(data)) return ErrorCode::UPLOAD_FAILED;
        return ErrorCode::SUCCESS;
    }
    // 获取数据接口
    std::pair<ErrorCode, ChemicalData> getData(const std::string& userId, const std::string& dataId) {
        if (!permManager.hasReadPermission(userId)) return {ErrorCode::PERMISSION_DENIED, ChemicalData{}};
        if (!repository.exists(dataId)) return {ErrorCode::NOT_FOUND, ChemicalData{}};
        ChemicalData data = repository.getData(dataId);
        return {ErrorCode::SUCCESS, data};
    }
    // 更新数据接口 - 允许修改 rawData/category/tags,自动重新验证质量
    ErrorCode updateData(const std::string& userId,
                         const std::string& dataId,
                         const std::string& newRawData,
                         const std::string& newFormat,
                         const std::string& newCategory,
                         const std::unordered_set<std::string>& newTags)
    {
        if (!permManager.hasWritePermission(userId)) return ErrorCode::PERMISSION_DENIED;
        if (!repository.exists(dataId)) return ErrorCode::NOT_FOUND;
        if (!DataValidator::validateFormat(newFormat)) return ErrorCode::INVALID_DATA_FORMAT;
        if (!categoryManager.isValidCategory(newCategory)) return ErrorCode::INVALID_PARAMETER;
        if (!DataValidator::validateTags(newTags)) return ErrorCode::INVALID_PARAMETER;
        if (!DataValidator::validateRawData(newRawData)) return ErrorCode::INVALID_PARAMETER;
        ChemicalData data = repository.getData(dataId);
        data.rawData = newRawData;
        data.format = newFormat;
        data.category = newCategory;
        data.tags = newTags;
        data.qualityCheckPassed = DataQualityChecker::checkQuality(data);
        if (!data.qualityCheckPassed) return ErrorCode::QUALITY_CHECK_FAILED;
        if (!repository.updateData(data)) return ErrorCode::UNKNOWN_ERROR;
        return ErrorCode::SUCCESS;
    }
    // 删除数据接口
    ErrorCode deleteData(const std::string& userId, const std::string& dataId) {
        if (!permManager.hasAdminPermission(userId)) return ErrorCode::PERMISSION_DENIED;
        if (!repository.exists(dataId)) return ErrorCode::NOT_FOUND;
        if (!repository.removeData(dataId)) return ErrorCode::UNKNOWN_ERROR;
        return ErrorCode::SUCCESS;
    }
    // 数据格式转换接口
    std::pair<ErrorCode, std::string> convertDataFormat(const std::string& userId,
                                                        const std::string& dataId,
                                                        const std::string& targetFormat)
    {
        if (!permManager.hasReadPermission(userId)) return {ErrorCode::PERMISSION_DENIED, ""};
        if (!repository.exists(dataId)) return {ErrorCode::NOT_FOUND, ""};
        if (!DataValidator::validateFormat(targetFormat)) return {ErrorCode::INVALID_DATA_FORMAT, ""};
        ChemicalData data = repository.getData(dataId);
        std::string converted = DataFormatConverter::convert(data.rawData, data.format, targetFormat);
        return {ErrorCode::SUCCESS, converted};
    }
    // 数据分类列表查询接口
    std::vector<std::string> listCategories() {
        std::vector<std::string> cats;
        for (const auto& c : categoryManager.getCategories())
            cats.push_back(c);
        return cats;
    }
    // 列出指定分类下的数据 ID 列表（仅返回 id 列表）
    std::vector<std::string> listDataIdsByCategory(const std::string& userId, const std::string& category) {
        std::vector<std::string> result;
        if (!permManager.hasReadPermission(userId)) return result;
        if (!categoryManager.isValidCategory(category)) return result;
        auto datas = repository.listDataByCategory(category);
        for (const auto& data : datas) {
            result.push_back(data.id);
        }
        return result;
    }
    // 为指定数据项添加标签
    ErrorCode addTagToData(const std::string& userId, const std::string& dataId, const std::string& tag) {
        if (!permManager.hasWritePermission(userId)) return ErrorCode::PERMISSION_DENIED;
        if (!repository.exists(dataId)) return ErrorCode::NOT_FOUND;
        if (tag.empty() || tag.length() > 16 || !std::regex_match(tag, std::regex("[\\w\\-]+"))) return
ErrorCode::INVALID_PARAMETER;
        ChemicalData data = repository.getData(dataId);
        tagManager.addTag(data, tag);
        if (!repository.updateData(data)) return ErrorCode::UNKNOWN_ERROR;
        return ErrorCode::SUCCESS;
    }
    // 从指定数据项移除标签
    ErrorCode removeTagFromData(const std::string& userId, const std::string& dataId, const std::string& tag) {
        if (!permManager.hasWritePermission(userId)) return ErrorCode::PERMISSION_DENIED;
        if (!repository.exists(dataId)) return ErrorCode::NOT_FOUND;
        ChemicalData data = repository.getData(dataId);
        if (!tagManager.hasTag(data, tag)) return ErrorCode::INVALID_PARAMETER;
        tagManager.removeTag(data, tag);
        if (!repository.updateData(data)) return ErrorCode::UNKNOWN_ERROR;
        return ErrorCode::SUCCESS;
    }
    // 设置用户权限（仅管理员操作）
    ErrorCode setUserPermission(const std::string& adminUserId, const std::string& targetUserId, AccessLevel
level) {
        if (!permManager.hasAdminPermission(adminUserId)) return ErrorCode::PERMISSION_DENIED;
        permManager.setPermission(targetUserId, level);
        return ErrorCode::SUCCESS;
    }
};
// int main() {
//     ChemicalMLController controller;
//     ErrorCode ec;
//     std::cout << "上传状态: " << static_cast<int>(ec) << std::endl;
//     auto [err, data] = controller.getData("admin", "data001");
//     if (err == ErrorCode::SUCCESS) {
//         std::cout << "数据内容: " << data.rawData << std::endl;
//     }
//     return 0;
// }
```