数据库系统大作业开发说明书

🤗连接数据库

我们使用由TiDB Cloud提供的云端数据库服务，避免大家进行开发时各自本地建库。

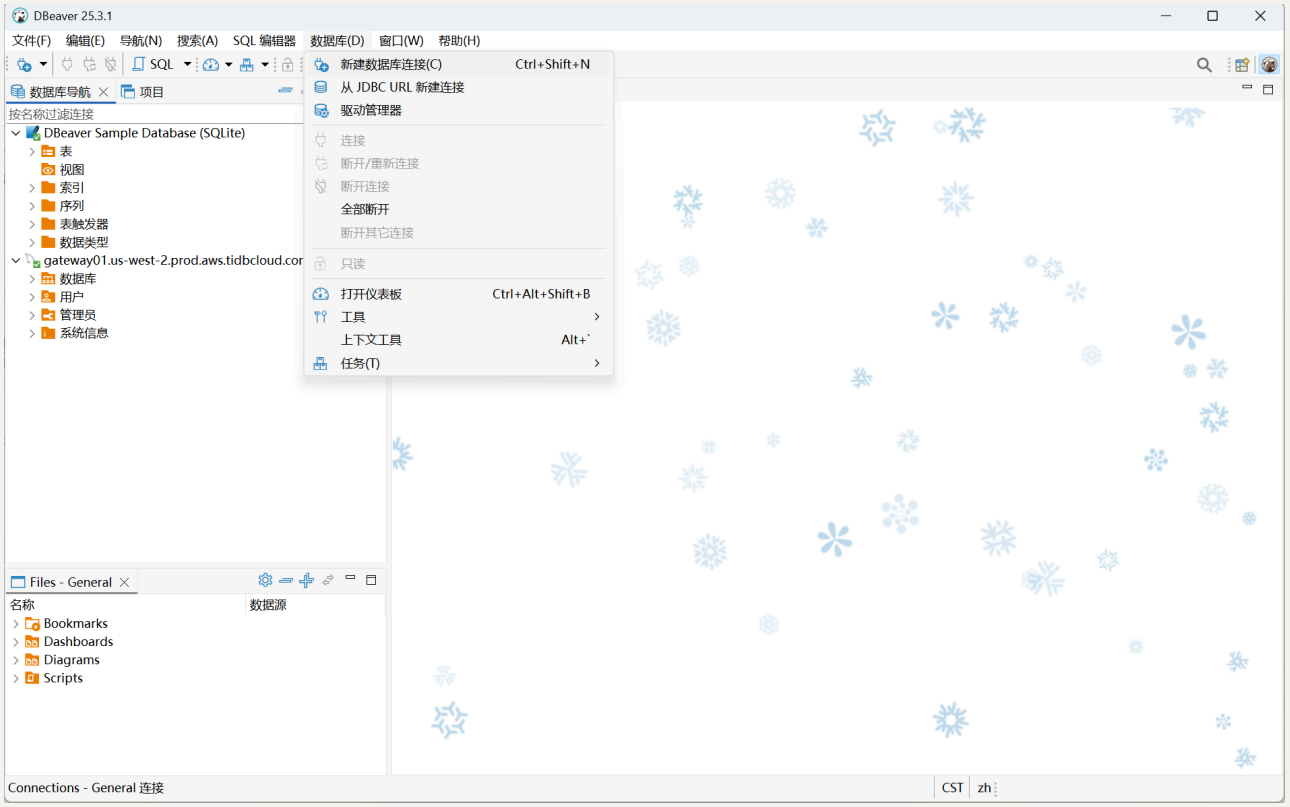
连接数据库的方法分为两种，第一种是直接配置项目的env文件，具体方法在后面会说。第二种则是下载DBeaver。

下载DBeaver

Download | DBeaver Community

使用DBeaver连接数据库

新建数据库连接

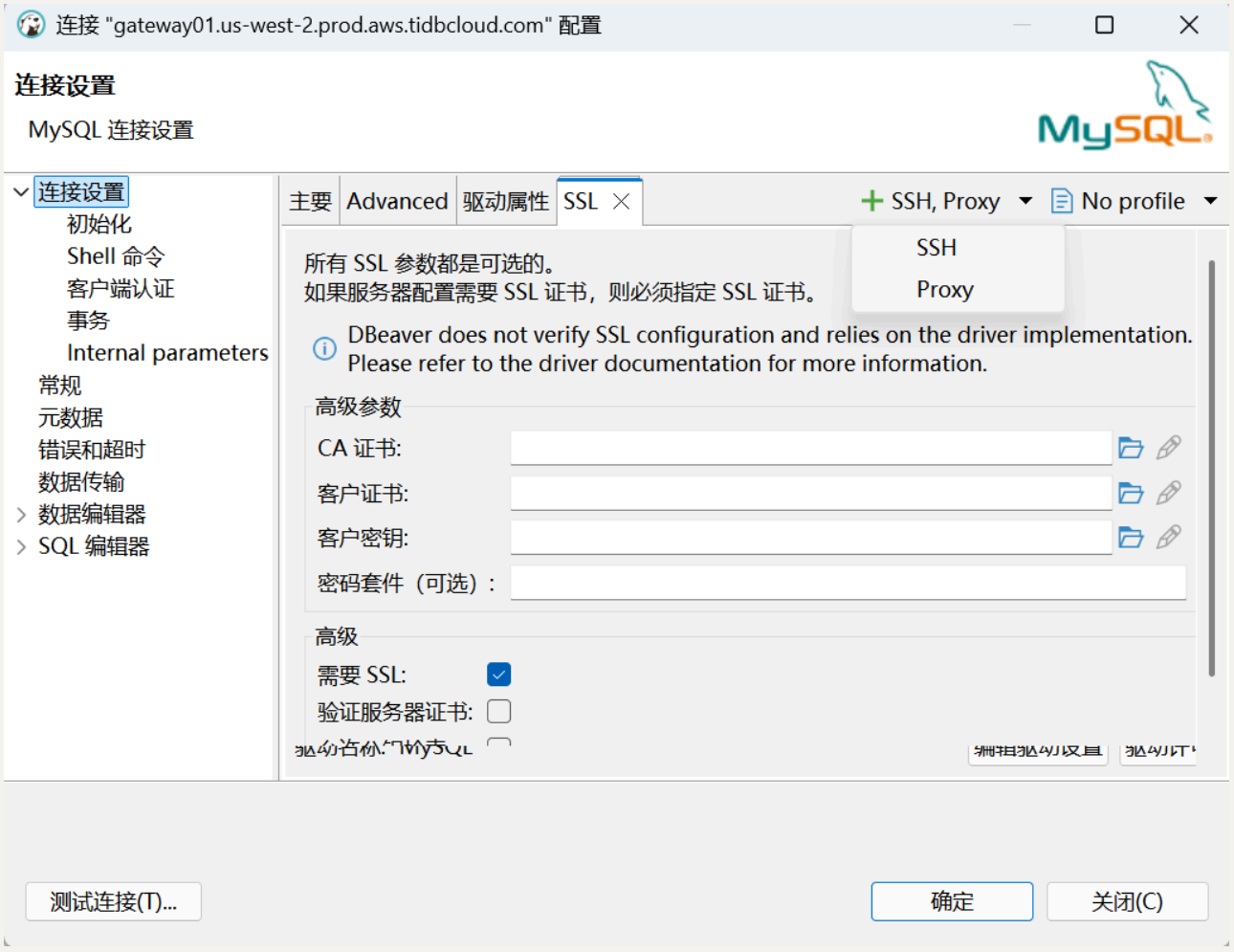


选mysql，服务器地址：gateway01.us-west-2.prod.aws.tidbcloud.com，端口为4000

用户名：2A7oCHJDJFAcs2V.root

密码：PFEbwsm0CfrBHlWe

记得添加ssl，只勾选需要ssl这一选项。



点击测试连接，若成功则点击确定，至此就成功建立了连接，可以通过DBeaver对数据库进行操作。

🎉开发环境配置

后端环境搭建

1.克隆项目

浏览器打开网址Editing Course_db_Project/README.md at main · pofvvvv/Course_db_Project点击`fork`生成自己的项目代码仓库`YOUR_GITHUB/PROJECT`，将fork仓库克隆到本地

```
git clone https://github.com/YOUR_GITHUB/PROJECT.git
cd PROJECT
git remote add upstream https://github.com/OWNER/PROJECT.git   # 加主仓库别名 upstream，这里的地址应该是我的仓库地址
https://github.com/pofvvvv/Course_db_Project.git
```

至此可以先告一段落，详细提交流程在后面详细说。

2.安装环境依赖

如果担心自己的python环境被污染可以创建虚拟环境。否则在项目根目录打开终端输入

```
pip install -r requirements.txt
```

3.配置环境变量

```
# 复制环境变量示例文件
copy env.example .env  # Windows
cp env.example .env    # Linux/Mac

# 编辑 .env 文件，填写数据库连接信息
```

这里`env.example`文件的

```
# TiDB Cloud 数据库配置
DB_USER=root
DB_PASSWORD=your-password
DB_HOST=your-tidb-host
DB_PORT=4000
DB_NAME=instrument_booking
```

改为

```
# TiDB Cloud 数据库配置
DB_USER=2A7oCHJDJFAcs2V.root
DB_PASSWORD=PFEbwsmOCfrBHlWe
DB_HOST=gateway01.us-west-2.prod.aws.tidbcloud.com
DB_PORT=4000
DB_NAME=test
```

启动后端服务

```
python run.py
# 或
flask run
```

后端服务将在 `http://localhost:5000` 运行

前端环境搭建

3.1 安装 Node.js

确保已安装 Node.js >= 16.0.0

```
node --version
npm --version
```

3.2 安装依赖

```
cd frontend
npm install
```

3.3 启动开发服务器

```
npm run dev
```

前端服务将在 `http://localhost:3000` 启动

验证环境

- 访问后端 API 文档：`http://localhost:5000/apidocs`
- 访问前端应用：`http://localhost:3000`
- 测试健康检查接口：`http://localhost:5000/api/v1/health`

📁项目结构说明

后端结构

```
app/
├── models/                 # 数据库模型
|   ├── __init__.py      # 模型统一导出
|   ├── mixins.py        # 模型混入类（ToDictMixin）
|   └── *.py             # 各个表的模型定义
|
├── api/v1/               # API v1 版本
|   ├── __init__.py      # 蓝图注册
|   ├── schemas/         # DTO/Schema
|   |   ├── __init__.py
|   |   └── *_schema.py # 各模块的 Schema
|   └── *.py             # 各业务模块的 API 路由
|
```

```
├── services/           # 业务逻辑层
│   ├── __init__.py
│   └── *_service.py    # 各模块的服务层
│
└── utils/              # 工具类
    ├── response.py     # success() / fail() 响应函数
    ├── exceptions.py   # 异常类和全局异常处理
    └── schemas.py      # Schema 基类
```

前端结构

```
frontend/src/
├── api/                # API 请求封装
│   ├── request.js      # Axios 实例和拦截器
│   └── *.js            # 各模块的 API 函数
│
├── views/              # 页面组件
│   └── *.vue           # 各页面组件
│
├── components/         # 公共组件
│   └── *.vue
│
├── router/             # 路由配置
│   └── index.js
│
└── assets/             # 静态资源
    └── styles/         # 全局样式
        └── global.css
```

📐开发规范

1.命名规范

后端

- 模型文件: `laboratory.py`, `student.py` (小写，单数)
- **Service** 文件: `lab_service.py`, `user_service.py` (小写 + 下划线)
- **API** 文件: `laboratory.py`, `user.py` (小写，单数)
- **Schema** 文件: `lab_schema.py`, `user_schema.py`

前端

- 组件文件: `LaboratoryList.vue`, `UserCard.vue` (PascalCase)
- **API** 文件: `laboratory.js`, `user.js` (小写)
- 工具文件: `request.js`, `utils.js` (小写)

2.注释规范

Python

```python
def create_lab(data):
    """
    创建实验室

    Args:
        data: 实验室数据字典，包含 name 和 location

    Returns:
        Laboratory: 创建的实验室对象

    Raises:
        ValidationError: 数据验证失败
    """
    pass
```

Vue/JavaScript

```
/**
 * 获取实验室列表
 * @returns {Promise} 返回实验室列表数据
 */
export function getLabList() {
  return request({
    url: '/laboratories/',
    method: 'get'
  })
}
```

3. 提交信息规范

使用约定式提交（Conventional Commits）：

```
<type>(<scope>): <subject>

<body>

<footer>
```

Type 类型：

- `feat`: 新功能
- `fix`: 修复 bug
- `docs`: 文档更新
- `style`: 代码格式（不影响功能）
- `refactor`: 重构
- `test`: 测试相关
- `chore`: 构建/工具相关

示例：

```
feat(lab): 添加实验室删除功能

- 实现删除实验室 API
- 添加删除确认对话框
- 更新前端删除按钮样式

Closes #123
```

🔌 API 开发指南

1. 创建新 API 模块（以 Equipment 为例）

步骤 1: 创建 Schema

`app/api/v1/schemas/equipment_schema.py`

```python
from marshmallow import fields, validate
from app.utils.schemas import BaseCreateSchema, BaseUpdateSchema,
BaseSchema

class EquipmentSchema(BaseSchema):
    """设备 Schema（用于响应）"""
    id = fields.Integer(dump_only=True)
    name = fields.String(required=True)
    # ... 其他字段

class EquipmentCreateSchema(BaseCreateSchema):
    """创建设备 Schema"""
    name = fields.String(required=True,
validate=validate.Length(min=1, max=100))
    # ... 其他字段
```

步骤 2: 创建 Service

`app/services/equipment_service.py`

```python
from app import db
from app.models.equipment import Equipment
from app.utils.exceptions import NotFoundError, ValidationError

def get_equipment_list():
    """获取设备列表"""
    return Equipment.query.all()

def create_equipment(data):
    """创建设备"""
    equipment = Equipment(**data)
    db.session.add(equipment)
    db.session.commit()
    return equipment
```

步骤 3: 创建 API 路由

`app/api/v1/equipment.py`

```python
from flask import Blueprint, request
from flasgger import swag_from
from app.services import equipment_service
from app.api.v1.schemas.equipment_schema import EquipmentSchema,
EquipmentCreateSchema
from app.utils.response import success, fail

equipment_bp = Blueprint('equipment', __name__)
equipment_schema = EquipmentSchema()
equipment_create_schema = EquipmentCreateSchema()

@equipment_bp.route('/', methods=['GET'])
@swag_from({...})  # Swagger 文档配置
def get_equipments():
    """获取设备列表"""
    try:
```

```
        equipments = equipment_service.get_equipment_list()
        data = equipment_schema.dump(equipments, many=True)
        return success(data=data, msg='查询成功')
    except Exception as e:
        return fail(code=500, msg=f'查询失败: {str(e)}')

@equipment_bp.route('/', methods=['POST'])
def create_equipment():
    """创建设备"""
    try:
        json_data = request.get_json()
        errors = equipment_create_schema.validate(json_data)
        if errors:
            return fail(code=422, msg='数据验证失败', data=errors)

        validated_data = equipment_create_schema.load(json_data)
        equipment =
equipment_service.create_equipment(validated_data)
        data = equipment_schema.dump(equipment)
        return success(data=data, msg='创建成功')
    except Exception as e:
        return fail(code=500, msg=f'创建失败: {str(e)}')
```

步骤 **4:** 注册蓝图

在 `app/api/v1/__init__.py` 中：

```
from app.api.v1 import equipment

api_v1.register_blueprint(equipment.equipment_bp,
url_prefix='/equipment')
```

2. API 开发规范

统一响应格式

使用 `app/utils/response` 中的函数：

```
from app.utils.response import success, fail

# 成功响应
return success(data={...}, msg='操作成功')

# 失败响应
return fail(code=400, msg='操作失败', data={...})
```

异常处理

使用自定义异常类：

```
from app.utils.exceptions import NotFoundError, ValidationError

if not resource:
    raise NotFoundError('资源不存在')

if not valid:
    raise ValidationError('数据验证失败', payload={...})
```

数据验证

使用 Marshmallow Schema：

```
# 验证请求数据
errors = schema.validate(json_data)
if errors:
    return fail(code=422, msg='数据验证失败', data=errors)

# 加载并验证数据
validated_data = schema.load(json_data)
```

🎨 前端开发指南

1. 创建新页面（以 Equipment 为例）

步骤 **1:** 创建 **API** 请求

`frontend/src/api/equipment.js`

```
import request from './request'

export function getEquipmentList() {
  return request({
    url: '/equipment/',
    method: 'get'
  })
}

export function createEquipment(data) {
  return request({
    url: '/equipment/',
    method: 'post',
    data
  })
}
```

步骤 **2:** 创建页面组件

`frontend/src/views/Equipment.vue`

```
<template>
  <div class="equipment-page">
    <!-- 页面内容 -->
  </div>
</template>
```

```vue
<script setup>
import { ref, onMounted } from 'vue'
import { getEquipmentList } from '@/api/equipment'

const equipmentList = ref([])

const fetchList = async () => {
  const res = await getEquipmentList()
  equipmentList.value = res.data || []
}

onMounted(() => {
  fetchList()
})
</script>

<style scoped lang="scss">
// 样式
</style>
```

步骤 **3:** 添加路由

`frontend/src/router/index.js`

```
{
  path: '/equipment',
  name: 'Equipment',
  component: () => import('@/views/Equipment.vue'),
  meta: {
    title: '设备管理'
  }
}
```

2. 前端开发规范

组件结构

```
<template>
  <!-- 模板 -->
</template>

<script setup>
// 逻辑
</script>

<style scoped lang="scss">
// 样式
</style>
```

使用统一的样式类

- `glass-card`: 毛玻璃卡片
- `cute-button`: 可爱按钮样式
- `gradient-text`: 渐变文字
- `float`: 浮动动画
- `animate__animated animate__fadeInUp`: Animate.css 动画

API 调用

```
import { getLabList } from '@/api/laboratory'

try {
  const res = await getLabList()
  // res 已经是处理后的数据，格式为 { code, msg, data }
  if (res.code === 200) {
    // 处理数据
  }
} catch (error) {
  // 错误已通过 ElMessage 提示，这里可以处理额外逻辑
}
```

🗄️ 数据库开发指南

1. 创建新模型

步骤 **1:** 创建模型文件

`app/models/equipment.py`

```python
from app import db
from app.models.mixins import ToDictMixin

class Equipment(db.Model, ToDictMixin):
    """设备表"""
    __tablename__ = 'equipment'

    id = db.Column(db.BigInteger, primary_key=True, comment='设备ID')
    name = db.Column(db.String(100), nullable=False, comment='设备名称')
    # ... 其他字段

    def __repr__(self):
        return f'<Equipment {self.id}: {self.name}>'
```

步骤 **2:** 在 **init.py** 中导入

`app/models/__init__.py`

```python
from app.models.equipment import Equipment
```

步骤 **3:** 创建迁移

```bash
flask db migrate -m "Add equipment table"
flask db upgrade
```

2. 数据库操作规范

- 使用 SQLAlchemy ORM，避免直接写 SQL
- 所有数据库操作都在 Service 层
- 使用事务处理（`db.session.commit()`）
- 异常时回滚（`db.session.rollback()`）

🌲 Git 工作流

把自己的工作完成后，通过git上传至github并发起pr，我审阅后合并（话说我真的会看吗）。

1. 分支管理

- `main/master`: 主分支，稳定版本
- `feat/*`: 功能分支
- `fix/*`: 修复分支
- `docs/*`: 文档分支

2. 工作流程

开始新功能前，先同步主仓库最新代码

```
git fetch upstream
git checkout -b feat/xxx upstream/main   # 基于上游 main 创建本地分支
```

开发、提交

```
... 写代码 ...
git add .
git commit -m "feat: 描述"
```

浏览器进入你 Fork 的页面，会看到一个 **Compare & pull request** 按钮，点它即可向 `OWNER/PROJECT` 发起 PR。
以后每次同步主仓库：

```
git fetch upstream
git rebase upstream/main            # 或 merge，保持分支最新
```

🧪 测试指南

1. 后端测试

单元测试

```python
import unittest
from app import create_app, db
from app.models import Laboratory

class LaboratoryTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_create_lab(self):
        lab = Laboratory(name='测试实验室', location='测试位置')
        db.session.add(lab)
        db.session.commit()
        self.assertIsNotNone(lab.id)
```

API 测试

使用 Postman 或 curl 测试 API：

```
# 获取列表
curl http://localhost:5000/api/v1/laboratories/

# 创建
curl -X POST http://localhost:5000/api/v1/laboratories/ \
  -H "Content-Type: application/json" \
  -d '{"name":"测试实验室","location":"测试位置"}'
```

2. 前端测试

- 在浏览器中手动测试各个功能
- 检查控制台是否有错误
- 测试不同屏幕尺寸的响应式布局

📚 参考资源

文档（真的有人会去看吗）

- Flask 官方文档
- Vue 3 官方文档
- Element Plus 文档
- SQLAlchemy 文档

代码示例

参考已有代码：

- 后端：`app/api/v1/laboratory.py`（完整的 API 实现示例）
- 前端：`frontend/src/views/LaboratoryList.vue`（完整的前端页面示例）

最后更新时间：**2026-1-1**，新年快乐各位组员。