streamlit

构建 Web UI 的核心框架，创建交互组件

Streamlit 基本功能

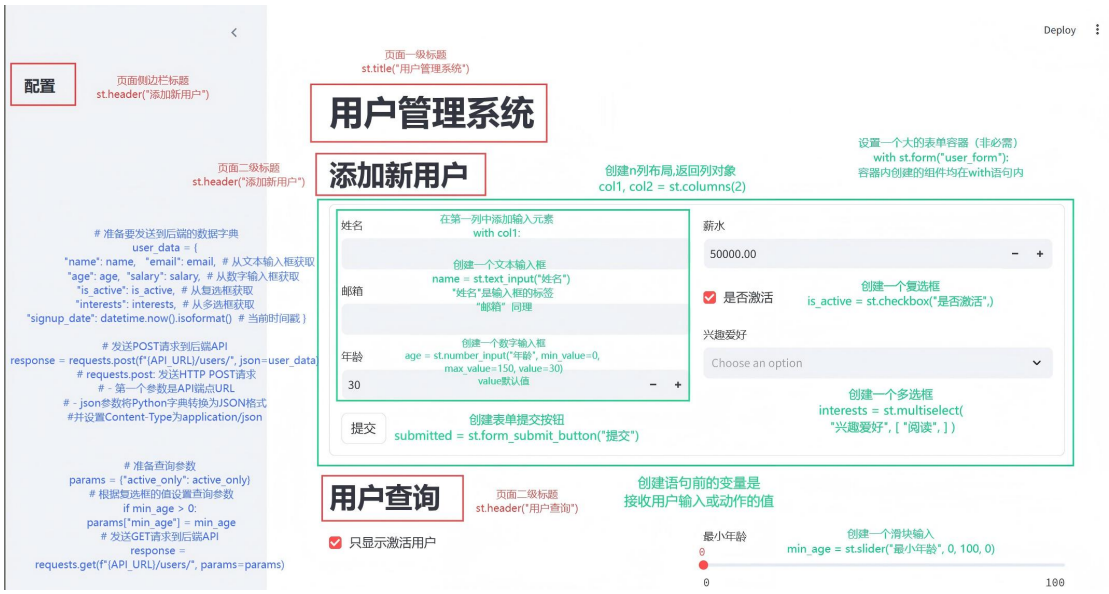
（1）快速构建数据可视化 Web 应用

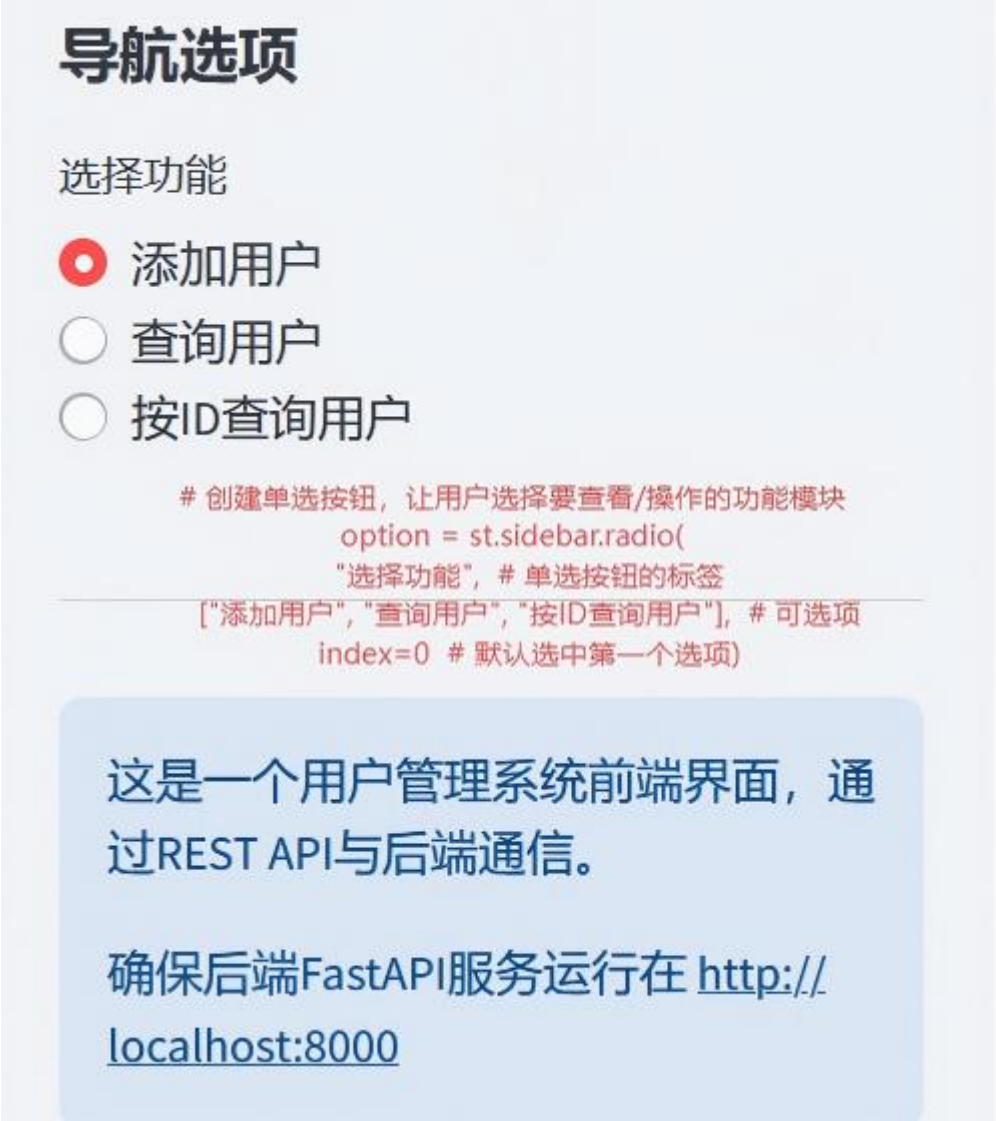
（2）支持交互组件（按钮/输入框/滑块）

（3）实时更新界面（无需手动刷新）

（4）直接集成 Python 数据处理库（Pandas/Matplotlib）

Streamlit 基本组件





页面配置

```
# 页面配置（必须放在所有其他 Streamlit 命令之前）
st.set_page_config(
    page_title="Streamlit 演示应用",
    page_icon="  ",
    layout="wide"
)
```

页面一级标题：st.title("用户管理系统")
页面侧边栏标题：st.header("添加新用户")
页面二级标题：st.header("添加新用户")

输入组件

设置一个大的表单容器（非必需）
with st.form("user_form"):
容器内创建的组件均在 with 语句内

创建 n 列布局,返回列对象
col1, col2 = st.columns(2)

在第一列中添加输入元素
with col1:
创建语句前的变量是接收用户输入或动作的值
创建一个文本输入框，"姓名"是输入框的标签：
name = st.text_input("姓名")
创建一个数字输入框：
age = st.number_input("年龄", min_value=0,max_value=150, value=30)value 默认值
创建一个滑块输入：min_age = st.slider("最小年龄", 0, 100, 0)
st.selectbox: 下拉选择
创建一个多选框：interests = st.multiselect("兴趣爱好", ["阅读",])
创建一个复选框：is_active = st.checkbox("是否激活",)
# 创建单选按钮，让用户选择要查看/操作的功能模块
option = st.sidebar.radio("选择功能", # 单选按钮的标签
["添加用户", "查询用户", "按 ID 查询用户"], # 可选项
index=0 # 默认选中第一个选项)
st.date_input: 日期选择
st.time_input: 时间选择
st.file_uploader: 文件上传

输出组件

st.write: 通用输出

st.dataframe: 数据框显示

st.table: 表格显示

st.json: JSON 格式显示

st.metric: 指标显示

st.progress: 进度条

进阶组件

st.session_state

st.session_state 是 Streamlit 中用于在用户会话中存储和共享状态的核心功能，它解决了 Streamlit 脚本“每次交互都会重新运行”导致的变量状态丢失问题。

核心是设置一个存储器（类似字典）存储设置的变量，用户和应用交互的整个周期，如果存储的变量的值发生了变化，则刷新页面更新值。各个组件也可以获取这些变量值。

核心功能与作用

1. **跨状态存储**：在用户与应用交互的整个会话周期内（从打开页面到关闭页面），保存变量值，避免因脚本重新运行而重置。
2. **跨组件通信**：让不同组件（如按钮、输入框）之间共享数据，例如：
    1. 记录用户的操作历史（如点击次数、筛选条件）
    2. 保存中间计算结果（如数据加载后的处理结果）
    3. 实现多步骤流程（如表单分页、向导式操作）

基本使用方法

1. 初始化状态变量

直接对 st.session_state 赋值即可创建状态变量，通常在应用启动时初始化：

```
# 初始化状态变量（仅在首次运行或变量不存在时执行）
if "count" not in st.session_state:
    st.session_state.count = 0  # 计数器初始值
    if "user_input" not in st.session_state:
     st.session_state.user_input = ""  # 存储用户输入
```

2. 读取和修改状态变量

通过 st.session_state.变量名 或 st.session_state["变量名"] 操作：

```
# 读取状态
st.write(f"当前计数: {st.session_state.count}")
# 修改状态
st.session_state.count += 1  # 计数器+1
st.session_state["user_input"] = "新值"  # 字典式访问
```

3. 状态变化触发刷新

当 st.session_state 中的变量被修改时，Streamlit 会自动重新运行脚本并刷新页面，确保展示最新状态。

Chatbox

```
chat_box = ChatBox(
    assistant_avatar=os.path.join(
        "img",
        "chatchat_icon_blue_square_v2.png"
    )
)
```

聊天组件 chat_message + st.chat_input

核心作用是：

1. **构建对话界面**：模拟即时通讯场景，区分“用户”和“助手”（如 AI）的消息角色。
2. **维护对话历史**：通过 `st.session_state` 存储消息记录，实现对话上下文的保留。
3. **支持交互输入**：提供用户输入框，方便用户发送消息并触发后端逻辑（如调用 AI 接口生成回复）。

- `st.chat_message(role)`：用于显示一条消息，`role` 参数指定角色（如 `"user"` 表示用户消息，`"assistant"` 表示助手消息），支持自定义头像和样式。
- `st.chat_input(placeholder)`：提供一个输入框，用于获取用户输入的消息，返回值为用户输入的字符串（当用户点击发送时触发）。
- `st.session_state`：Streamlit 的会话状态，用于在用户交互过程中保存对话历史（避免页面刷新后消息丢失）。

数据发送请求

Post

```
 # 准备要发送到后端的数据字典
user_data = { "name": name,     "email": email,   # 从文本输入框获取
                    "age": age,   "salary": salary,   # 从数字输入框获取
"is_active": is_active,   # 从复选框获取
"interests": interests,   # 从多选框获取
 "signup_date": datetime.now().isoformat()   # 当前时间戳 }

# 发送 POST 请求到后端 API
response = requests.post(f"{API_URL}/users/", json=user_data)
 # requests.post:  发送 HTTP POST 请求
 # 第一个参数是 API 端点 URL
# json 参数将 Python 字典转换为 JSON 格式，并设置 Content-Type 为 application/json
```

Get

```
# 准备查询参数
params = {"active_only": active_only}
 # 根据复选框的值设置查询参数
if min_age > 0:
 params["min_age"] = min_age
# 发送 GET 请求到后端 API
response =
requests.get(f"{API_URL}/users/", params=params)
```

实例代码

Streamlit 前端代码 (app_frontend.py)

```
import streamlit as st
import requests
import pandas as pd
from datetime import datetime

# 设置页面配置
st.set_page_config(page_title="User Management", layout="wide")
st.title("用户管理系统")

# 后端 API 地址
API_URL = "http://localhost:8000"

# 在侧边栏创建导航选项
st.sidebar.header("导航选项")
# 创建单选按钮，让用户选择要查看/操作的功能模块
option = st.sidebar.radio(
    "选择功能",  # 单选按钮的标签
    ["添加用户", "查询用户", "按 ID 查询用户"],  # 可选项
    index=0  # 默认选中第一个选项
)

# 使用条件语句根据用户选择的导航选项显示相应内容

if option == "添加用户":
    # 添加新用户部分
    st.header("添加新用户")
    # 创建表单容器
    with st.form("user_form"):
        col1, col2 = st.columns(2)

        with col1:
            name = st.text_input("姓名")  # 文本输入框，用于输入姓名
            email = st.text_input("邮箱")  # 文本输入框，用于输入邮箱
            age = st.number_input("年龄", min_value=0, max_value=150,
value=30)  # 数字输入框，用于输入年龄

        with col2:
            salary = st.number_input("薪水", min_value=0.0, value=50000.0)  #
数字输入框，用于输入薪水
            is_active = st.checkbox("是否激活", value=True)  # 复选框，用于选
择是否激活
            interests = st.multiselect(
```

```
                    "兴趣爱好",  # 多选框标签
                    ["编程", "阅读", "运动", "音乐", "旅行", "烹饪", "摄影", "游戏"]
# 多选框选项
                )

        submitted = st.form_submit_button("提交")  # 表单提交按钮

        if submitted:
            if not name or not email:
                st.error("姓名和邮箱是必填项!")  # 错误提示消息
            else:
                user_data = {
                    "name": name,  # 从输入框获取的姓名
                    "email": email,  # 从输入框获取的邮箱
                    "age": age,  # 从输入框获取的年龄
                    "salary": salary,  # 从输入框获取的薪水
                    "is_active": is_active,  # 从复选框获取的激活状态
                    "interests": interests,  # 从多选框获取的兴趣爱好列表
                    "signup_date": datetime.now().isoformat()  # 当前时间戳
                }

                try:
                    # 发送 POST 请求到后端 API 的/users/端点
                    response = requests.post(f"{API_URL}/users/",
json=user_data)
                    if response.status_code == 200:
                        st.success("用户添加成功!")  # 成功提示消息
                    else:
                        st.error(f"添加用户失败: {response.text}")  # 显示后
端返回的错误信息
                except requests.exceptions.ConnectionError:
                    st.error("无法连接到后端 API，请确保 FastAPI 服务正在
运行")  # 连接错误提示

elif option == "查询用户":
    # 用户查询部分
    st.header("用户查询")
    col1, col2 = st.columns(2)
    with col1:
        active_only = st.checkbox("只显示激活用户", value=True)  # 复选框，用
于筛选激活用户
    with col2:
        min_age = st.slider("最小年龄", 0, 100, 0)  # 滑块，用于设置最小年龄
筛选条件
```

```
    if st.button("查询用户"):  # 查询按钮
        try:
            params = {"active_only": active_only}  # 准备查询参数
            if min_age > 0:
                params["min_age"] = min_age  # 如果设置了最小年龄，添加到查询参数

            # 发送 GET 请求到后端 API 的/users/端点
            response = requests.get(f"{API_URL}/users/", params=params)

            if response.status_code == 200:
                users = response.json()  # 解析响应 JSON 数据
                if users:
                    # 转换为 DataFrame 显示
                    df = pd.DataFrame(users)
                    # 格式化日期显示
                    if "signup_date" in df.columns:
                        df["signup_date"] = pd.to_datetime(df["signup_date"]).dt.strftime("%Y-%m-%d %H:%M")

                    st.dataframe(df)  # 显示数据表格

                    # 显示统计信息
                    st.subheader("统计信息")
                    col1, col2, col3 = st.columns(3)
                    col1.metric("总用户数", len(users))  # 显示总用户数指标
                    col2.metric("平均年龄", round(df["age"].mean(), 1))  # 显示平均年龄指标
                    col3.metric("平均薪水", f"${round(df['salary'].mean(), 2)}")  # 显示平均薪水指标
                else:
                    st.info("没有找到符合条件的用户")  # 信息提示消息
            else:
                st.error(f"查询失败: {response.text}")  # 显示查询错误信息
        except requests.exceptions.ConnectionError:
            st.error("无法连接到后端 API，请确保 FastAPI 服务正在运行")  # 连接错误提示

elif option == "按 ID 查询用户":
    # 按 ID 查询用户部分
    st.header("按 ID 查询用户")
    user_id = st.number_input("用户 ID", min_value=0, value=0)  # 数字输入框，用于输入用户 ID
```

```
    if st.button("查询"):  # 查询按钮
        try:
            # 发送 GET 请求到后端 API 的/users/{user_id}端点
            response = requests.get(f"{API_URL}/users/{user_id}")
            if response.status_code == 200:
                user = response.json()  # 解析响应 JSON 数据
                st.json(user)  # 以 JSON 格式显示用户数据
            elif response.status_code == 404:
                st.error("用户不存在")  # 用户不存在错误提示
            else:
                st.error(f"查询失败: {response.text}")  # 其他错误提示
        except requests.exceptions.ConnectionError:
            st.error("无法连接到后端 API，请确保 FastAPI 服务正在运行")  # 连接错误提示

# 在侧边栏底部添加一些辅助信息
st.sidebar.markdown("---")  # 添加分隔线
st.sidebar.info(
    "这是一个用户管理系统前端界面，通过 REST API 与后端通信。\n\n"
    "确保后端 FastAPI 服务运行在 http://localhost:8000"
)
```

st_aggrid

st_aggrid 是 Streamlit 的一个第三方组件，基于 AG Grid 表格库开发，提供了比 Streamlit 自带表格更丰富的交互功能。比如筛选，条件值，排序，编辑等等，像是页面版的 Excel 表格功能。

st_aggrid 基本用法


```
# 1. 导入依赖库
import streamlit as st                    # 导入 Streamlit 库，用于快速构建 Web 应用，简写为 st
import pandas as pd                       # 导入 Pandas 库，用于数据处理（创建/操作 DataFrame）
# 从 st_aggrid 导入核心组件：
# AgGrid：渲染增强型表格；GridOptionsBuilder：构建表格配置；DataReturnMode：设置数据返回规则
from st_aggrid import (
    AgGrid,
    GridOptionsBuilder,
    DataReturnMode
)

# 2. 准备示例数据（构建展示用的结构化数据）
# 创建 Pandas DataFrame，包含员工的各类信息，作为表格的数据源
data = pd.DataFrame({
    "ID": [1001, 1002, 1003, 1004, 1005],          # 员工唯一 ID（数值类型）
    "姓名": ["张三", "李四", "王五", "赵六", "钱七"], # 员工姓名（字符串类型）
    "年龄": [25, 32, 45, 28, 36],                  # 员工年龄（数值类型）
    "部门": ["技术", "产品", "销售", "技术", "人事"], # 所属部门（字符串类型）
    "薪资(月)": [8000, 12000, 15000, 9500, 11000], # 月薪资（数值类型）
    "入职年份": [2020, 2018, 2015, 2021, 2019],    # 入职年份（数值类型）
    "绩效评级": ["A", "B", "A", "B", "A"]          # 绩效等级（字符串类型，用于条件格式化）
})
```

3. 设置页面基础内容
st.title("st_aggrid 真值判断错误修复示例") # 设置应用页面的主标题（大字体，居中）

st 自带表格

st.dataframe(data, height=180)

aggrid 基础配置表格

基础默认配置的 AgGrid 表格
st.subheader("1. 基础表格") # 设置小标题（比 title 小一号，用于划分功能模块）
调用 AgGrid 渲染表格，传入数据源 data，配置基础参数
AgGrid(
    data, # 表格数据源（必填，Pandas DataFrame）
    height=180, # 设置表格高度为 180px，控制垂直占用空间
    fit_columns_on_grid_load=True #表格加载时自动调整列宽，内容完整显示
)
# 添加说明文字（小字体，灰色），说明该表格的基础功能
st.caption("基础功能：列宽调整、标题排序、右键复制")

aggrid 增强交互表格

5. 第二个表格：带完整交互功能的增强表格
st.subheader("2. 带交互功能的表格") # 小标题，标识该模块为交互型表格

# 5.1 创建表格配置构建器（核心步骤，用于自定义表格规则）
# 从 DataFrame 生成基础配置，自动识别列的数据类型
gb = GridOptionsBuilder.from_dataframe(data)

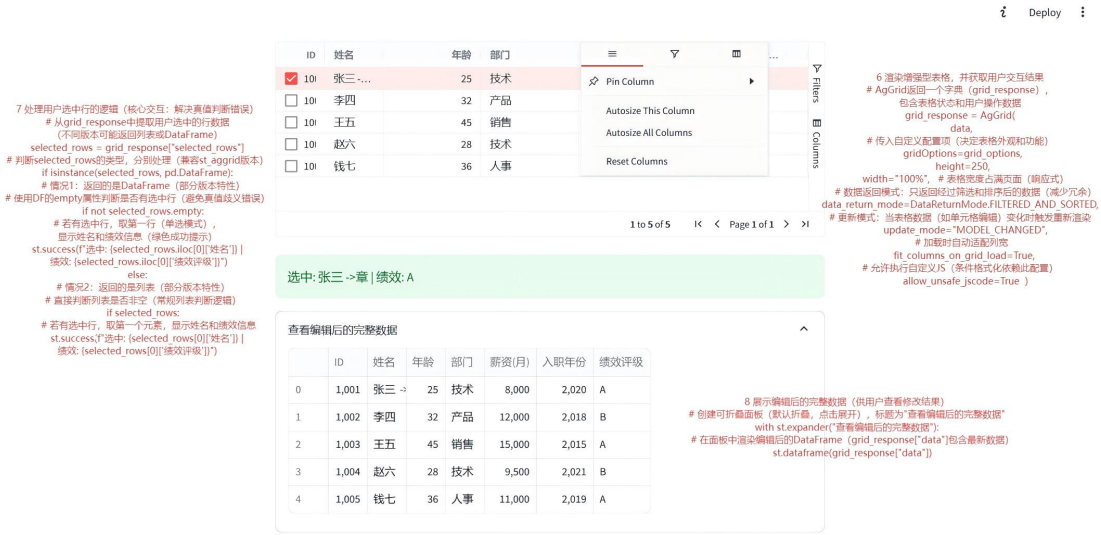
# 5.2 配置列的具体属性（针对每一列单独设置规则）
# 配置"ID"列：固定在左侧（滚动时不消失）、不可编辑、宽度 100px
gb.configure_column("ID", pinned="left", editable=False, width=100)
# 配置"姓名"列：允许单元格编辑（用户可修改）、宽度 120px
gb.configure_column("姓名", editable=True, width=120)
# 配置"薪资(月)"列：标记为数字类型、启用数字筛选器（支持范围筛选）、宽度 150px
gb.configure_column("薪资(月)", type=["numericColumn", "numberColumnFilter"], width=150)
# 配置"绩效评级"列：启用文本筛选器（支持精确匹配）、宽度 120px
gb.configure_column("绩效评级", filter=True, width=120)

# 5.3 配置表格整体功能（全局规则）
# 启用分页功能，自动计算每页显示行数（根据表格高度自适应）
gb.configure_pagination(paginationAutoPageSize=True)

```
# 启用侧边栏（右上角显示菜单按钮，点击可展开筛选、列显示控制面板）
gb.configure_side_bar()
# 配置行选择模式：单选（"single"）、显示复选框（用户通过复选框选择行）
gb.configure_selection("single", use_checkbox=True)

# 5.4 配置条件格式化（根据单元格值设置样式）
# 针对"绩效评级"列，按值设置背景色：A 评级绿色背景，B 评级黄色背景
gb.configure_column(
    "绩效评级",  # 目标列名
    cellStyle={  # 单元格样式配置
        "condition": [  # 条件列表
            # 条件 1：值为"A"时，背景色设为浅绿（#CCFFCC）
            {"condition": "value == 'A'", "style": {"backgroundColor":
"#CCFFCC"}},
            # 条件 2：值为"B"时，背景色设为浅黄（#FFFFCC）
            {"condition": "value == 'B'", "style": {"backgroundColor":
"#FFFFCC"}}
        ]
    }
)
# 5.5 生成最终的表格配置项（将上述所有配置转换为 AgGrid 可识别的格式）
grid_options = gb.build()
```

交互与处理




```
# 5.6 渲染增强型表格，并获取用户交互结果
# AgGrid 返回一个字典（grid_response），包含表格状态和用户操作数据
grid_response = AgGrid(
```

```
    data,                                # 表格数据源
    gridOptions=grid_options,   # 传入自定义配置项（决定表格外观和功能）
    height=250,                          # 表格高度 250px（预留分页和选择功能空间）
    width="100%",                        # 表格宽度占满页面（响应式）
    # 数据返回模式：只返回经过筛选和排序后的数据（减少冗余）
    data_return_mode=DataReturnMode.FILTERED_AND_SORTED,
    # 更新模式：当表格数据（如单元格编辑、筛选）变化时触发重新渲染
    update_mode="MODEL_CHANGED",
    fit_columns_on_grid_load=True,   # 加载时自动适配列宽
    allow_unsafe_jscode=True             # 允许执行自定义 JS（条件格式化依赖此配置）
)

# 5.7 处理用户选中行的逻辑（核心交互：解决真值判断错误）
# 从 grid_response 中提取用户选中的行数据（不同版本可能返回列表或 DataFrame）
selected_rows = grid_response["selected_rows"]

# 判断 selected_rows 的类型，分别处理（兼容不同 st_aggrid 版本）
if isinstance(selected_rows, pd.DataFrame):
    # 情况 1：返回的是 DataFrame（部分版本特性）
    # 使用 DataFrame 的 empty 属性判断是否有选中行（避免真值歧义错误）
    if not selected_rows.empty:
        # 若有选中行，取第一行（单选模式），显示姓名和绩效信息（绿色成功提示）
        st.success(f"选中: {selected_rows.iloc[0]['姓名']} | 绩效: {selected_rows.iloc[0]['绩效评级']}")
else:
    # 情况 2：返回的是列表（部分版本特性）
    # 直接判断列表是否非空（常规列表判断逻辑）
    if selected_rows:
        # 若有选中行，取第一个元素，显示姓名和绩效信息
        st.success(f"选中: {selected_rows[0]['姓名']} | 绩效: {selected_rows[0]['绩效评级']}")

# 5.8 展示编辑后的完整数据（供用户查看修改结果）
# 创建可折叠面板（默认折叠，点击展开），标题为"查看编辑后的完整数据"
with st.expander("查看编辑后的完整数据"):
    # 在面板中渲染编辑后的 DataFrame（grid_response["data"]包含最新数据）
    st.dataframe(grid_response["data"])

# 6. 第三个表格：Streamlit 原生表格（用于对比）
st.subheader("3. 原生表格对比")  # 小标题，标识该模块为原生表格
# 使用 Streamlit 自带的 dataframe 函数渲染表格，高度 180px（与基础 AgGrid 表
```

格一致）
st.dataframe(data, height=180)

核心功能说明

1. **基础渲染：**通过 AgGrid(dataframe) 直接渲染表格，用法与 st.dataframe 类似。
2. **自定义配置：**
    1. 用 GridOptionsBuilder 配置列宽、排序、筛选、固定列等。
    2. 支持分页、侧边栏（筛选面板）、单元格编辑等高级功能。
3. **交互反馈：**通过 grid_response["selected_rows"] 获取用户选中的行数据，实现双向交互。

与 Streamlit 原生表格的联系和区别

维度	Streamlit 原生表格（st.dataframe）	st_aggrid
基础功能	支持表格展示、简单排序、列宽调整	支持表格展示，且功能更丰富
交互能力	有限（仅支持排序、筛选、复制单元格）	强大（支持单选 / 多选、单元格编辑、行拖拽、批量操作等）
自定义程度	低（仅支持少量参数如宽度、高度）	高（可配置列属性、分页、主题、侧边栏等）
性能	适合小数据量（万级以内）	更适合大数据量（支持虚拟滚动，加载更高效）
集成复杂度	简单（无需额外配置）	稍复杂（需学习配置项，但基础用法简单）
适用场景	快速展示数据，无需复杂交互	需用户深度交互的场景（如数据筛选、编辑、选中）

联系：两者都可接收 pandas DataFrame 作为输入，用于在 Streamlit 应用中展示结构化数据，且 st_aggrid 可以看作是原生表格的增强版。

何时使用 st_aggrid？

- 需要用户对表格进行复杂操作（如多选行、编辑单元格、批量处理）时；
- 数据量较大，需要高效加载和虚拟滚动时；
- 需要自定义表格样式、分页、筛选面板等高级功能时。

如果仅需简单展示数据，st.dataframe 更轻量便捷；若需增强交互，st_aggrid 是更好的选择。