0 WebAPP系统数据新增开发指南

系统采用 **Dash 前端 + SQLAlchemy (SQLite) 后端**，且使用了 `ui_state` **(JSON 字段)** 来存储大量非结构化界面状态，增加新栏（Field）通常不需要修改数据库表结构，只需要在前端代码中修改 4 个关键环节。

---

1. 系统架构与数据流机制

系统的**双层存储**和**四向同步**机制。

数据分类

你的系统中有两类数据，增加字段的处理方式完全不同：

1. **UI State (界面状态数据):**
   - **定义:** 用户的草稿、非核心配置、页面临时输入（如：输入框里的文字、下拉框选中的项、图表的视角）。
   - **存储位置:** 数据库 `ProjectState` 表中的 `ui_state` 列（JSON 类型）。
   - **特点: 无需修改数据库结构**，即插即用，最为灵活。
2. **Core Data (核心业务数据):**
   - **定义:** 参与计算、生成报告的关键数据（如：`Mission` 定义、`Components` 列表、`DVM` 矩阵）。
   - **存储位置:** 独立的数据库表（如 `missions`，`components` 表）。
   - **特点: 需要修改 ORM 模型（models.py）**，通常涉及数据库迁移。

四向同步机制

在 `phase1.py` 等页面中，数据流转如下：

1. **Auto-Save (自动保存):** 用户修改输入 -> 触发 `@callback` -> 更新 `ui_state` 字典 -> 写入数据库。
2. **Manual-Save (手动保存):** 点击保存按钮 -> 读取所有 `State` -> 强制写入数据库（核心表 + UI表）。
3. **Auto-Load (自动加载):** URL 变更（`/phase1`）-> 触发加载回调 -> 从数据库读取 -> 填充 `Output`。
4. **Manual-Load (手动加载):** 点击加载按钮 -> 触发同一加载回调 -> 填充 `Output`。

---

2. 实战场景 A：新增 UI State 字段（推荐，无需改库）

场景举例： 在 **Phase 1** 页面增加一个 **"分析师姓名 (Analyst Name)"** 输入框。这是一个备注信息，不参与复杂计算。

步骤 1：修改布局 (Layout)

文件： phase1.py

操作： 在 layout 中添加组件，务必设置唯一的 id。

Python

```
# [Phase 1 Layout]
dbc.Row([
    dbc.Col([
        dbc.Label("分析师姓名"),
        dbc.Input(
            id="input-analyst-name",  # <--- 关键 ID
            type="text",
            placeholder="请输入您的姓名..."
        )
    ])
])
```

步骤 2：修改自动保存回调 (Auto-Save)

文件： phase1.py

函数： auto_save_phase1_ui

原理： 将新字段加入监听列表，发生变化时将其存入 ui_state 字典。

Python

```
@callback(
    [Output('phase1-save-status', 'children', allow_duplicate=True),
     Output('phase1-ui-state', 'data', allow_duplicate=True)],
    [Input('input-mission-title', 'value'),
     Input('input-mission-desc', 'value'),
     Input('input-threshold', 'value'),
     # [新增 1] 监听新组件的变化
     Input('input-analyst-name', 'value')],
    prevent_initial_call=True
)
def auto_save_phase1_ui(title, desc, threshold, analyst_name): # [新增 2] 添加参数
    from dash import ctx
```

```
    if not ctx.triggered: return no_update, no_update

    state = get_state_manager()

    # 读取旧状态
    current_ui = state.load('phase1', 'ui_state') or {}

    updates = {}
    # 原有逻辑...
    if title is not None: updates['mission_title'] = title

    # [新增 3] 将新数据存入字典
    if analyst_name is not None:
        updates['analyst_name'] = analyst_name

    if not updates:
        return no_update, no_update

    current_ui.update(updates)
    state.save('phase1', 'ui_state', current_ui)

    return html.Span("已保存", className="text-success small"), current_ui
```

步骤 3：修改加载回调 (Load)

文件： phase1.py

函数： 通常是最后的 load_phase1_data (由 URL 触发)。

原理： 从 ui_state 字典取出数据，填回输入框。

Python

```
@callback(
    [Output('input-mission-title', 'value'),
     Output('input-mission-desc', 'value'),
     Output('input-threshold', 'value'),
     # [新增 1] 增加输出目标
     Output('input-analyst-name', 'value')],
    [Input('url', 'pathname'),
     Input('btn-load-phase1', 'n_clicks')],
    prevent_initial_call=True
)
def load_phase1_data(pathname, n_clicks):
    if pathname != '/phase1': return no_update
```

```
    # ... 获取数据的逻辑 ...
    state = get_state_manager()
    ui_state = state.load('phase1', 'ui_state') or {}

    # [新增 2] 从字典中取值，必须给默认值防止报错
    restored_analyst = ui_state.get('analyst_name', "")

    # ... 其他原有数据获取 ...

    # [新增 3] 返回值顺序必须严格对应 Output 列表
    return (
        mission_title,
        mission_desc,
        threshold,
        restored_analyst
    )
```

步骤 4：修改手动保存回调 (Manual Save)

文件： phase1.py

函数： save_phase1_manual

原理： 点击保存按钮时，不仅保存核心数据，也要强制刷新一遍 UI State。

Python

```
@callback(
    Output('phase1-manual-save-status', 'children'),
    [Input('btn-save-phase1', 'n_clicks')],
    [State('input-mission-title', 'value'),
     # ...
     # [新增 1] 获取当前输入框的值
     State('input-analyst-name', 'value'),
     State('phase1-ui-state', 'data')],
    prevent_initial_call=True
)
def save_phase1_manual(n_clicks, title, ..., analyst_name, ui_state_store): #
[新增 2] 参数
    if not n_clicks: return no_update

    state = get_state_manager()

    # 更新 UI State
    current_ui = ui_state_store or {}
    if analyst_name is not None:
```

```
        current_ui['analyst_name'] = analyst_name # [新增 3] 写入字典

    state.save('phase1', 'ui_state', current_ui)

    # ... 保存其他核心数据 ...

    return dbc.Alert("保存成功", color="success")
```

---

3. 实战场景 B：新增 Core Data 字段（需修改数据库）

场景举例： 在 **Phase 1** 的 Mission 定义中增加 **"预算上限 (Budget Cap)"**。这是一个核心约束，可能会影响后续算法。

步骤 0：修改数据库模型 (Backend)

文件： database/models.py (你未上传此文件，但必须修改它)

注意： 修改模型后，必须删除旧的 .db 文件让系统重新建表，或者使用 Alembic 进行迁移。

Python

```
# 在 database/models.py 中找到 Mission 类
class Mission(Base):
    # ... 原有字段 ...
    # [新增]
    budget_cap = Column(Float, default=0.0)
```

步骤 1：修改布局 (Layout)

文件： phase1.py

同上，增加 id="input-budget-cap" 的组件。

步骤 2：修改自动保存回调 (Auto-Save)

文件： phase1.py

函数： auto_save_phase1_ui

特别注意： 这里需要同时更新 ui_state（为了回显）和 mission 对象（为了业务存储）。

Python

```
def auto_save_phase1_ui(..., budget_cap):
    # ... ui_state 保存逻辑同上 ...
```

```
    # [新增核心数据保存]
    # 注意：auto_save 通常只存 UI State，但如果希望核心数据也实时保存：
    if budget_cap is not None:
        # 读取当前 mission
        current_mission = state.load('phase1', 'mission') or {}
        # 更新字典 (StateManager 会处理对象映射)
        current_mission['budget_cap'] = float(budget_cap)
        state.save('phase1', 'mission', current_mission)
```

注：通常建议核心数据仅在"手动保存"时提交，或者在 Auto-save 中更新 UI State，在 Manual Save 时才写入 Mission 表。

步骤 3：修改加载回调 (Load)

文件： phase1.py

函数： load_phase1_data

Python

```
def load_phase1_data(pathname, n_clicks):
    # ...
    # 读取核心数据
    mission_data = state.load('phase1', 'mission') or {}

    # [新增] 从核心对象中取值
    restored_budget = mission_data.get('budget_cap', 0.0)

    return (..., restored_budget)
```

步骤 4：修改手动保存回调 (Manual Save)

文件： phase1.py

函数： save_phase1_manual

Python

```
def save_phase1_manual(..., budget_cap, ...):
    state = get_state_manager()

    # [新增] 构造核心数据对象
    mission_data = {
        'title': title,
```

```
        'description': desc,
        'budget_cap': budget_cap  # <--- 存入 Core Data
    }

    state.save('phase1', 'mission', mission_data)
    # ...
```

4. 避坑检查清单 (Checklist)

检查项	说明	常见错误后果
Output 顺序	`load_phase1_data` 返回值的顺序必须与 `@callback` 中的 `Output` 列表完全一致。	数据填错位置（例如：预算填到了姓名框）。
None 值判断	在保存前检查 `if value is not None`。	初始化时将数据库里的有效值覆盖为空。
UI State 键名	确保字典 Key (`updates['key']`) 在整个 Phase 中唯一。	数据互相覆盖。
Input ID	确保 `id="..."` 在整个应用中唯一。	Dash 报错 `DuplicateIdError`。
StateManager	核心数据保存依赖 `state.save` 的实现。	如果 `state_manager_v2.py` 不支持保存该 Key，数据会丢失。

5. 关于 `state_manager_v2.py` 的适配

如果你新增的是 **Core Data** 且不是标准的 Mission/Components，你可能需要修改 `state_manager_v2.py` 中的 `save` 方法。

例如，如果你想在 Phase 1 存一个全新的表 `ProjectRisks`，你需要在 `StateManagerV2.save` 中添加逻辑：

Python

```
# state_manager_v2.py -> save 方法
def save(self, phase: str, key: str, value: Any) -> bool:
    with get_db_session() as session:
        # ...
        if phase == 'phase1' and key == 'project_risks':
            # [新增] 处理新类型的核心数据存储逻辑
            self._save_project_risks(session, value)
        # ...
```

总结： 对于 90% 的情况（增加参数、配置、备注），请使用 **场景 A (UI State)**，它最安全、最快，且不需要触碰后端数据库代码。