v3_done_代码解释_AGENT-BASED SIMULATION OF CONSTRUCTION WORKFLOW AND INFORMATION FLOW

基本介绍

在建筑行业数字化转型的浪潮中，仿真技术通过模拟工作流与信息流为项目管理效率提升、资源优化配置及风险预测提供了关键支持。本内容以“基于代理的建筑工作流与信息流仿真”为核心，通过SQL脚本构建了一套覆盖项目全生命周期的精细化管理系统，从环境初始化、基础表结构设计（如工序表`process`、依赖关系表`dependency`）、扰动数据生成（任务量±10%波动、10%无效日概率）到动态任务跟踪（`task_progress`表及`process_work_day`函数），系统实现了对建筑项目的全流程数字化管控。通过定义工序依赖逻辑（如“碎石基层”完成后才能启动“地板管道”）、楼层顺序约束（低楼层未完成则高楼层任务阻塞）及工种资源动态分配（`trade_resource`状态切换），系统确保工序按预设逻辑高效推进，同时引入扰动因子模拟真实施工场景的不确定性。每日自动模拟通过`daily_task_record`记录实际效率（基于`initial_production_rate`计算）、返工标记及完成状态，结合`project_progress_details`视图提供实时进度概览与累计工作量分析，最终为管理者提供从风险预警到资源调度的数据驱动决策支持，推动建筑项目从规划到交付的智能化转型，显著提升工程管理的精准性与可预测性。

步骤1：清理环境（每次重新运行前必须执行）

这段代码是一个SQL脚本，用于清理数据库中的表和视图。下面是对代码的逐步解释：

步骤1：清理环境

```SQL
/*================================================================
 步骤1：清理环境（每次重新运行前必须执行）
================================================================*/
```

- 这是一个注释块，说明了接下来的代码块的目的是清理数据库环境，确保在重新运行某些操作之前，数据库处于一个干净的状态。

删除表

```SQL
-- 删除所有相关表和视图，CASCADE 确保级联删除依赖对象
DROP TABLE IF EXISTS daily_task_record, daily_work_history,
task_progress, trade_resource, space, task, dependency, process
CASCADE;
```

- `DROP TABLE`：用于删除数据库中的表。
- `IF EXISTS`：是一个可选的条件子句，用于在删除表之前检查表是否存在。如果表不存在，不会报错。
- 列出的表名：`daily_task_record`，`daily_work_history`，`task_progress`，`trade_resource`，`space`，`task`，`dependency`，`process` 是要被删除的表。
- `CASCADE`：是一个可选的关键字，用于指定级联删除。这意味着如果存在依赖于这些表的其他数据库对象（如视图、外键约束等），这些依赖对象也会被自动删除。

删除视图

```SQL
DROP VIEW IF EXISTS current_project_status,
project_progress_details CASCADE;
```

- `DROP VIEW`：用于删除数据库中的视图。
- `IF EXISTS`：同样用于检查视图是否存在，如果不存在则不会报错。
- 列出的视图名：`current_project_status`，`project_progress_details` 是要被删除的视图。
- `CASCADE`：同样用于级联删除，确保依赖于这些视图的其他对象也被删除。

总结

这段SQL脚本的作用是清理数据库中指定的表和视图，确保在重新运行某些操作（如重新创建表和视图）之前，数据库中没有残留的旧数据或结构。`CASCADE`选项确保了即使存在依赖关系，也能顺利删除相关对象，避免因依赖关系导致的删除失败。

步骤2：创建基础表结构（所有数值字段使用整数）

这段SQL脚本的作用是创建数据库中的基础表结构，用于存储与项目管理相关的数据。下面是对代码的逐步解释：

步骤2：创建基础表结构

```SQL
/*================================================================
=
 步骤2：创建基础表结构（所有数值字段使用整数）
================================================================*
/
```

- 这是一个注释块，说明接下来的代码块的目的是创建数据库的基础表结构，并且所有数值字段都使用整数类型。

工序表

```SQL
-- 工序表：核心基准数据，记录工序名称、工种和初始生产率
CREATE TABLE process (
    process TEXT PRIMARY KEY,              -- 工序名称（主键）
    trade TEXT NOT NULL,                   -- 所属工种（如
Plumbing、Electricity）
    initial_production_rate INT NOT NULL CHECK
(initial_production_rate > 0)  -- 初始生产率（单位：工作量/人/天）
);
```

- `CREATE TABLE process`：创建一个名为 `process` 的表，用于存储工序的相关信息。
- `process TEXT PRIMARY KEY`：定义 `process` 字段为主键，存储工序名称，类型为文本。
- `trade TEXT NOT NULL`：定义 `trade` 字段，存储工序所属的工种，类型为文本，且不能为空。
- `initial_production_rate INT NOT NULL CHECK (initial_production_rate > 0)`：定义 `initial_production_rate` 字段，存储工序的初始生产率，类型为整数，且不能为空，同时通过 `CHECK` 约束确保值大于0。

依赖关系表

```SQL
-- 依赖关系表：记录工序间的依赖关系（前置工序 -> 后续工序）
CREATE TABLE dependency (
    predecessor_process TEXT REFERENCES process(process),  -- 前置工序
    successor_process TEXT REFERENCES process(process),    -- 后续工序
    PRIMARY KEY (predecessor_process, successor_process)   -- 复合主键
);
```

- `CREATE TABLE dependency`：创建一个名为 `dependency` 的表，用于存储工序之间的依赖关系。
- `predecessor_process TEXT REFERENCES process(process)`：定义 `predecessor_process` 字段，存储前置工序的名称，类型为文本，并通过 `REFERENCES` 约束引用 `process` 表的 `process` 字段。
- `successor_process TEXT REFERENCES process(process)`：定义 `successor_process` 字段，存储后续工序的名称，类型为文本，并通过 `REFERENCES` 约束引用 `process` 表的 `process` 字段。
- `PRIMARY KEY (predecessor_process, successor_process)`：定义复合主键，由 `predecessor_process` 和 `successor_process` 组成，确保每条依赖关系唯一。

任务表

```SQL
-- 任务表：带±10%随机扰动的初始任务量（结果取整）
CREATE TABLE task (
    process TEXT PRIMARY KEY REFERENCES process(process),  -- 工序名称（外键）
    initial_quantity INT NOT NULL CHECK (initial_quantity > 0)  -- 扰动后的初始任务量
);
```

- `CREATE TABLE task`：创建一个名为 `task` 的表，用于存储任务的相关信息。
- `process TEXT PRIMARY KEY REFERENCES process(process)`：定义 `process` 字段为主键，存储工序名称，类型为文本，并通过 `REFERENCES` 约束引用 `process` 表的 `process` 字段。
- `initial_quantity INT NOT NULL CHECK (initial_quantity > 0)`：定义 `initial_quantity` 字段，存储初始任务量，类型为整数，且不能为空，同时通过

`CHECK` 约束确保值大于0。

空间表

```SQL
-- 空间表：每个楼层的工作量与任务量相同
CREATE TABLE space (
    process TEXT REFERENCES task(process),  -- 工序名称（外键）
    floor INT CHECK (floor BETWEEN 1 AND 5),  -- 楼层（1-5）
    quantity INT NOT NULL,                    -- 该楼层的工作量（=任务初始量）
    PRIMARY KEY (process, floor)              -- 复合主键
);
```

- `CREATE TABLE space`：创建一个名为 `space` 的表，用于存储每个楼层的工作量信息。
- `process TEXT REFERENCES task(process)`：定义 `process` 字段，存储工序名称，类型为文本，并通过 `REFERENCES` 约束引用 `task` 表的 `process` 字段。
- `floor INT CHECK (floor BETWEEN 1 AND 5)`：定义 `floor` 字段，存储楼层信息，类型为整数，并通过 `CHECK` 约束确保值在1到5之间。
- `quantity INT NOT NULL`：定义 `quantity` 字段，存储该楼层的工作量，类型为整数，且不能为空。
- `PRIMARY KEY (process, floor)`：定义复合主键，由 `process` 和 `floor` 组成，确保每个楼层的工序工作量唯一。

总结

这段SQL脚本创建了四个基础表：`process`、`dependency`、`task` 和 `space`，用于存储项目管理中的核心数据。每个表都有明确的字段定义和约束，确保数据的完整性和一致性。通过外键和复合主键的使用，建立了表之间的关联，形成了一个结构化的数据模型。

步骤3：插入基础数据（所有数值保持整数）

这段SQL脚本的作用是向之前创建的基础表中插入基础数据。下面是对代码的逐步解释：

步骤3：插入基础数据

```SQL
/*=============================================================
 步骤3：插入基础数据（所有数值保持整数）
=============================================================*/
```

- 这是一个注释块，说明接下来的代码块的目的是向表中插入基础数据，并且所有数值都保持为整数类型。

插入工序数据

```SQL
-- 插入工序数据（工序名称、工种、初始生产率）
INSERT INTO process VALUES
('Gravel base layer',          'Gravel',    34),  -- 碎石基层
('Pipes in the floor',         'Plumbing',  69),  -- 地板管道
('Electric conduits in the floor', 'Electricity', 51),  -- 地板电缆管道
('Floor tiling',               'Tiling',    62),  -- 地板铺砖
('Partition phase 1',          'Partition', 55),  -- 隔断阶段1
('Pipes in the wall',          'Plumbing',  37),  -- 墙面管道
('Partition phase 2',          'Partition', 51),  -- 隔断阶段2
('Electric conduits in the wall', 'Electricity', 41),  -- 墙面电缆管道
('Partition phase 3',          'Partition', 32),  -- 隔断阶段3
('Wall tiling',                'Tiling',    27);  -- 墙面铺砖
```

- `INSERT INTO process VALUES`：向 `process` 表中插入多条记录。
- 每条记录包含三个字段：`process`（工序名称）、`trade`（所属工种）、`initial_production_rate`（初始生产率）。
- 这些数据是项目中各个工序的基本信息，例如碎石基层、地板管道等。

插入带扰动的任务数据

```SQL
-- 插入带扰动的任务数据：基于原始数据 ±10% 随机扰动并取整
INSERT INTO task
SELECT
    process,
    ROUND(initial_quantity * (0.9 + 0.2 * random()))::INT  -- 扰动公式
FROM (VALUES  -- 原始数据
    ('Gravel base layer',          170),
    ('Pipes in the floor',         100),
    ('Electric conduits in the floor', 80),
    ('Floor tiling',               720),
    ('Partition phase 1',          750),
    ('Pipes in the wall',          190),
    ('Partition phase 2',           20),
    ('Electric conduits in the wall', 180),
    ('Partition phase 3',          200),
    ('Wall tiling',                290)
) AS raw_data(process, initial_quantity);
```

- `INSERT INTO task`：向 `task` 表中插入数据。
- 使用 `SELECT` 语句从一个子查询中获取数据。
- 子查询中的原始数据是一个值列表，包含每个工序的初始任务量。
- `ROUND(initial_quantity * (0.9 + 0.2 * random()))::INT`：对原始任务量进行 ±10% 的随机扰动，并取整。
  - `0.9 + 0.2 * random()`：生成一个在 0.9 到 1.1 之间的随机数。
  - `ROUND(...)`：对计算结果进行四舍五入。
  - `::INT`：将结果转换为整数类型。

插入依赖关系

```SQL
-- 插入依赖关系：定义工序执行顺序约束
INSERT INTO dependency VALUES
('Gravel base layer', 'Pipes in the floor'),          -- 碎石基层完成才能开始地板管道
('Gravel base layer', 'Electric conduits in the floor'),
('Pipes in the floor', 'Floor tiling'),              -- 地板管道完成才能开始铺砖
('Electric conduits in the floor', 'Floor tiling'),
('Partition phase 1', 'Pipes in the wall'),           -- 隔断阶段1完成才能开始墙面管道
('Pipes in the wall', 'Partition phase 2'),
('Partition phase 2', 'Electric conduits in the wall'),
('Electric conduits in the wall', 'Partition phase 3'),
('Partition phase 3', 'Wall tiling');
```

- `INSERT INTO dependency VALUES`：向 `dependency` 表中插入多条记录。
- 每条记录定义了两个工序之间的依赖关系，表示前置工序必须在后续工序之前完成。
- 例如，`('Gravel base layer', 'Pipes in the floor')` 表示碎石基层必须在地板管道之前完成。

生成空间数据

```SQL
-- 生成空间数据：每个工序分配到1-5层，工作量=任务总量
INSERT INTO space
SELECT
    t.process,
    f.floor,
    t.initial_quantity
FROM task t
CROSS JOIN generate_series(1, 5) AS f(floor);  -- 使用交叉连接生成5层数据
```

- `INSERT INTO space`：向 `space` 表中插入数据。
- 使用 `SELECT` 语句从 `task` 表中获取数据，并通过 `CROSS JOIN` 与 `generate_series(1, 5)` 生成的楼层数据进行交叉连接。
- `generate_series(1, 5)`：生成从1到5的序列，表示楼层。

- `CROSS JOIN`：将 `task` 表中的每条记录与生成的楼层数据进行笛卡尔积，确保每个工序在每个楼层都有记录。
- 插入的字段包括：`process`（工序名称）、`floor`（楼层）、`quantity`（工作量，等于任务的初始量）。

总结

这段SQL脚本向数据库中的基础表插入了项目管理所需的基础数据，包括工序信息、任务数据、工序依赖关系以及空间数据。通过随机扰动任务数据和生成楼层数据，确保了数据的多样性和完整性，为后续的项目进度管理和分析提供了基础。

步骤4：创建资源与任务进度跟踪表

这段SQL脚本的作用是创建资源与任务进度跟踪表，并初始化相关数据。下面是对代码的逐步解释：

步骤4：创建资源与任务进度跟踪表

```SQL
/*==========================================================================
=
 步骤4：创建资源与任务进度跟踪表
==========================================================================*
/
```

- 这是一个注释块，说明接下来的代码块的目的是创建用于跟踪资源和任务进度的表。

工种资源表

```SQL
-- 工种资源表：记录各工种可用工人数和状态
CREATE TABLE trade_resource (
    trade TEXT PRIMARY KEY,                    -- 工种名称
    available_workers INT DEFAULT 1 CHECK (available_workers >= 1),
-- 可用工人数（默认1）
    status TEXT DEFAULT 'available' CHECK (status IN
('available','busy'))  -- 资源状态
);
```

- `CREATE TABLE trade_resource`：创建一个名为 `trade_resource` 的表，用于记录各工种的资源信息。
- `trade TEXT PRIMARY KEY`：定义 `trade` 字段为主键，存储工种名称，类型为文本。

- `available_workers INT DEFAULT 1 CHECK (available_workers >= 1)`：定义`available_workers`字段，存储可用工人数，类型为整数，默认值为1，并通过`CHECK`约束确保值大于等于1。
- `status TEXT DEFAULT 'available' CHECK (status IN ('available','busy'))`：定义`status`字段，存储资源状态，类型为文本，默认值为'available'，并通过`CHECK`约束确保值只能是'available'或'busy'。

任务进度表

```SQL
-- 任务进度表：跟踪每个工序在各楼层的进度
CREATE TABLE task_progress (
    process TEXT REFERENCES process(process),    -- 工序名称
    floor INT,                                   -- 楼层
    remaining_quantity INT CHECK (remaining_quantity >= 0),  -- 剩余工作量
    status TEXT DEFAULT 'pending' CHECK (status IN ('pending','in_progress','completed')),  -- 任务状态
    start_day INT,                               -- 任务开始日
    last_update_day INT,                         -- 最后更新日
    assigned_trade TEXT REFERENCES trade_resource(trade),  -- 分配的工种
    planned_remaining INT,                       -- 初始计划剩余量
    daily_productivity INT,                       -- 实际日效率
    PRIMARY KEY (process, floor)                  -- 复合主键
);
```

- `CREATE TABLE task_progress`：创建一个名为`task_progress`的表，用于跟踪每个工序在各楼层的进度。
- `process TEXT REFERENCES process(process)`：定义`process`字段，存储工序名称，类型为文本，并通过`REFERENCES`约束引用`process`表的`process`字段。
- `floor INT`：定义`floor`字段，存储楼层信息，类型为整数。
- `remaining_quantity INT CHECK (remaining_quantity >= 0)`：定义`remaining_quantity`字段，存储剩余工作量，类型为整数，并通过`CHECK`约束确保值大于等于0。
- `status TEXT DEFAULT 'pending' CHECK (status IN ('pending','in_progress','completed'))`：定义`status`字段，存储任务状态，类型为文本，默认值为'pending'，并通过`CHECK`约束确保值只能是'pending'、'in_progress'或'completed'。
- `start_day INT`：定义`start_day`字段，存储任务开始日，类型为整数。
- `last_update_day INT`：定义`last_update_day`字段，存储最后更新日，类型为整数。

- `assigned_trade TEXT REFERENCES trade_resource(trade)`：定义 `assigned_trade` 字段，存储分配的工种，类型为文本，并通过 `REFERENCES` 约束引用 `trade_resource` 表的 `trade` 字段。
- `planned_remaining INT`：定义 `planned_remaining` 字段，存储初始计划剩余量，类型为整数。
- `daily_productivity INT`：定义 `daily_productivity` 字段，存储实际日效率，类型为整数。
- `PRIMARY KEY (process, floor)`：定义复合主键，由 `process` 和 `floor` 组成，确保每个工序在每个楼层的进度记录唯一。

初始化工种资源

```sql
-- 初始化工种资源：插入所有工种，默认可用工人数为1
INSERT INTO trade_resource (trade) VALUES
('Gravel'), ('Plumbing'), ('Electricity'), ('Tiling'),
('Partition');
```

- `INSERT INTO trade_resource (trade) VALUES`：向 `trade_resource` 表中插入工种数据。
- 每条记录只包含 `trade` 字段，其他字段使用默认值。插入的工种包括'Gravel'、'Plumbing'、'Electricity'、'Tiling'和'Partition'。

初始化任务进度

```sql
-- 初始化任务进度：为每个工序的每个楼层创建初始记录
INSERT INTO task_progress (
    process, floor, remaining_quantity,
    assigned_trade, planned_remaining
)
SELECT
    s.process,
    s.floor,
    s.quantity,        -- 初始剩余量=总工作量
    p.trade,           -- 从工序表获取工种
    s.quantity         -- 初始计划剩余量
FROM space s
JOIN process p USING (process);
```

- `INSERT INTO task_progress`：向 `task_progress` 表中插入数据。
- 使用 `SELECT` 语句从 `space` 表和 `process` 表中获取数据。

- `s.process`：从 `space` 表中获取工序名称。
- `s.floor`：从 `space` 表中获取楼层。
- `s.quantity`：从 `space` 表中获取工作量，作为初始剩余量和初始计划剩余量。
- `p.trade`：从 `process` 表中获取工种。
- `JOIN process p USING (process)`：通过 `process` 字段将 `space` 表和 `process` 表连接起来。

总结

这段SQL脚本创建了两个表：`trade_resource` 和 `task_progress`，用于跟踪工种资源和任务进度。通过插入初始数据，确保了资源和任务进度的初始状态正确设置，为后续的项目进度管理和资源调度提供了基础。

步骤5：创建更新任务进度的函数（增加楼层先后顺序判断）

这段SQL脚本定义了一个PL/pgSQL函数 `update_task_progress`，用于更新任务进度，启动符合条件的新任务。下面是对代码的逐步解释：

步骤5：创建更新任务进度的函数

```SQL
/*=============================================================
=
 步骤5：创建更新任务进度的函数（增加楼层先后顺序判断）
=============================================================*
/
```

- 这是一个注释块，说明接下来的代码块的目的是创建一个用于更新任务进度的函数，并且增加了对楼层先后顺序的判断。

函数定义

```SQL
-- 函数功能：启动符合条件的新任务（依赖完成、低楼层完成、工种可用）
CREATE OR REPLACE FUNCTION update_task_progress(current_day INT)
RETURNS void AS $$
DECLARE
    task_record RECORD;
    assigned_trades TEXT[] := ARRAY[]::TEXT[];  -- 记录已分配工种，避免重复占用
BEGIN
```

- `CREATE OR REPLACE FUNCTION update_task_progress(current_day INT)`：创建或替换一个名为 `update_task_progress` 的函数，接受一个整数参数 `current_day`，表示当前日期。
- `RETURNS void`：函数返回类型为 `void`，表示没有返回值。
- `DECLARE`：声明函数内部使用的变量。
  - `task_record RECORD`：声明一个记录变量，用于存储查询结果。
  - `assigned_trades TEXT[] := ARRAY[]::TEXT[]`：声明一个文本数组变量，用于记录已分配的工种，避免重复占用。

查询符合条件的待启动任务

SQL

```
    -- 查询符合条件的待启动任务（按楼层和工序排序）
    FOR task_record IN (
        SELECT
            tp.process,
            tp.floor,
            tp.assigned_trade
        FROM task_progress tp
        WHERE tp.status = 'pending'
          -- 检查依赖是否完成（同一楼层）
          AND NOT EXISTS (
              SELECT 1
              FROM dependency d
              JOIN task_progress tp2 ON d.predecessor_process =
tp2.process
              WHERE d.successor_process = tp.process
                AND tp2.floor = tp.floor
                AND tp2.status != 'completed'
          )
          -- 检查低楼层是否完成（同一工序）
          AND NOT EXISTS (
              SELECT 1
              FROM task_progress lower_tp
              WHERE lower_tp.process = tp.process
                AND lower_tp.floor < tp.floor
                AND lower_tp.status != 'completed'
          )
          -- 检查工种是否可用
          AND EXISTS (
              SELECT 1
              FROM trade_resource tr
              WHERE tr.trade = tp.assigned_trade
                AND tr.status = 'available'
          )
        ORDER BY tp.floor, tp.process  -- 按楼层和工序顺序启动
    )
```

- `FOR task_record IN (SELECT ...)`：使用游标循环查询符合条件的待启动任务。
- `SELECT`：从 `task_progress` 表中选择符合条件的记录。
  - `tp.process`：工序名称。

    - `tp.floor`：楼层。
    - `tp.assigned_trade`：分配的工种。
- `WHERE`：筛选条件：
    - `tp.status = 'pending'`：任务状态为待启动。
    - `NOT EXISTS (SELECT 1 FROM dependency d JOIN task_progress tp2 ...)`：检查依赖关系是否完成（同一楼层）。
    - `NOT EXISTS (SELECT 1 FROM task_progress lower_tp ...)`：检查低楼层是否完成（同一工序）。
    - `EXISTS (SELECT 1 FROM trade_resource tr ...)`：检查工种是否可用。
- `ORDER BY tp.floor, tp.process`：按楼层和工序顺序排序，确保按顺序启动任务。

循环处理任务

```SQL
    LOOP
        -- 确保同一工种不同时分配多个任务
        IF NOT task_record.assigned_trade = ANY(assigned_trades)
THEN
            -- 更新任务状态为进行中，记录开始时间
            UPDATE task_progress
            SET
                status = 'in_progress',
                start_day = current_day,
                last_update_day = current_day
            WHERE process = task_record.process
              AND floor = task_record.floor;

            -- 标记工种为忙碌状态
            UPDATE trade_resource
            SET status = 'busy'
            WHERE trade = task_record.assigned_trade;

            assigned_trades := array_append(assigned_trades,
task_record.assigned_trade);
        END IF;
    END LOOP;
```

- `LOOP`：循环处理每个符合条件的任务记录。
- `IF NOT task_record.assigned_trade = ANY(assigned_trades)`：检查当前工种是否已经分配给其他任务，避免重复占用。
- `UPDATE task_progress`：更新任务状态为进行中，并记录开始时间和最后更新时间。
- `UPDATE trade_resource`：将分配的工种状态标记为忙碌。

- `assigned_trades := array_append(assigned_trades, task_record.assigned_trade)`：将当前工种添加到已分配工种列表中。

函数结束

```SQL
END;
$$ LANGUAGE plpgsql;
```

- `END`：结束函数定义。
- `$$ LANGUAGE plpgsql`：指定函数的实现语言为PL/pgSQL。

总结

这段SQL脚本定义了一个PL/pgSQL函数 `update_task_progress`，用于根据当前日期更新任务进度，启动符合条件的新任务。函数通过查询和更新操作，确保任务的启动符合依赖关系、楼层顺序和工种可用性等条件，为项目的自动化进度管理提供了核心功能。

步骤6：创建每日记录表

这段SQL脚本的作用是创建两个表，用于记录项目管理中的每日工作状态和任务详细信息。下面是对代码的逐步解释：

步骤6：创建每日记录表

```SQL
/*==========================================================================
=
 步骤6：创建每日记录表
==========================================================================*
/
```

- 这是一个注释块，说明接下来的代码块的目的是创建用于记录每日工作状态和任务详细信息的表。

每日工作状态历史记录表

```sql
-- 每日工作状态历史记录（原始记录）
CREATE TABLE daily_work_history (
    day_number INT,          -- 模拟天数
    floor INT,               -- 楼层
    process TEXT,            -- 工序
    trade TEXT,              -- 工种
    planned_remaining INT,   -- 计划剩余量
    actual_done INT,         -- 当日实际完成量
    is_valid BOOLEAN,        -- 是否有效日（无效日完成量为0）
    recorded_at TIMESTAMPTZ DEFAULT NOW(),  -- 记录时间
    PRIMARY KEY (day_number, floor, process)
);
```

- `CREATE TABLE daily_work_history`：创建一个名为 `daily_work_history` 的表，用于记录每日工作状态的原始数据。
- `day_number INT`：记录模拟的天数，类型为整数。
- `floor INT`：记录楼层，类型为整数。
- `process TEXT`：记录工序名称，类型为文本。
- `trade TEXT`：记录工种，类型为文本。
- `planned_remaining INT`：记录计划剩余量，类型为整数。
- `actual_done INT`：记录当日实际完成量，类型为整数。
- `is_valid BOOLEAN`：记录是否为有效日（无效日完成量为o），类型为布尔值。
- `recorded_at TIMESTAMPTZ DEFAULT NOW()`：记录时间，类型为带时区的时间戳，默认值为当前时间。
- `PRIMARY KEY (day_number, floor, process)`：定义复合主键，由 `day_number`、`floor` 和 `process` 组成，确保每天每层每个工序的记录唯一。

每日任务详细记录表

```sql
-- 每日任务详细记录（含扰动、返工、效率等详细信息）
CREATE TABLE daily_task_record (
    day_number INT,
    process TEXT,
    floor INT,
    trade TEXT,
    is_invalid BOOLEAN,        -- 是否为无效日（10%概率）
    is_rework BOOLEAN,         -- 是否为返工（任务重启）
    daily_work_done INT,       -- 当日完成工作量
    efficiency TEXT,           -- 效率百分比（对比理论最大值）
    start_day INT,             -- 任务启动日
    complete_day INT,          -- 任务完成日（未完成则为NULL）
    recorded_at TIMESTAMPTZ DEFAULT NOW(),
    PRIMARY KEY (day_number, process, floor)
);
```

- `CREATE TABLE daily_task_record`：创建一个名为 `daily_task_record` 的表，用于记录每日任务的详细信息，包括扰动、返工、效率等。
- `day_number INT`：记录模拟的天数，类型为整数。
- `process TEXT`：记录工序名称，类型为文本。
- `floor INT`：记录楼层，类型为整数。
- `trade TEXT`：记录工种，类型为文本。
- `is_invalid BOOLEAN`：记录是否为无效日（10%概率），类型为布尔值。
- `is_rework BOOLEAN`：记录是否为返工（任务重启），类型为布尔值。
- `daily_work_done INT`：记录当日完成工作量，类型为整数。
- `efficiency TEXT`：记录效率百分比（对比理论最大值），类型为文本。
- `start_day INT`：记录任务启动日，类型为整数。
- `complete_day INT`：记录任务完成日（未完成则为NULL），类型为整数。
- `recorded_at TIMESTAMPTZ DEFAULT NOW()`：记录时间，类型为带时区的时间戳，默认值为当前时间。
- `PRIMARY KEY (day_number, process, floor)`：定义复合主键，由 `day_number`、`process` 和 `floor` 组成，确保每天每层每个工序的记录唯一。

总结

这段SQL脚本创建了两个表：`daily_work_history` 和 `daily_task_record`，用于记录项目管理中的每日工作状态和任务详细信息。通过这些表，可以跟踪每天的工作进展、任务完成情况以及相关效率和状态信息，为项目的进度管理和分析提供了详细的数据支持。

步骤7：扩展处理每日工作的函数（记录每日详细信息）

这段SQL脚本定义了一个PL/pgSQL函数 `process_work_day`，用于处理每日工作任务，更新任务进度，并记录每日详细信息。下面是对代码的逐步解释：

步骤7：扩展处理每日工作的函数

```SQL
/*=====================================================================
=
 步骤7：扩展处理每日工作的函数（记录每日详细信息）
=====================================================================*
/
```

- 这是一个注释块，说明接下来的代码块的目的是扩展处理每日工作的函数，并记录每日详细信息。

函数定义

```SQL
CREATE OR REPLACE FUNCTION process_work_day(current_day INT)
RETURNS void AS $$
DECLARE
    task_rec RECORD;
    work_done INT;          -- 当日实际完成量
    is_invalid BOOLEAN;     -- 是否无效日
    new_remaining INT;      -- 更新后剩余量
    eff NUMERIC(10,2);      -- 效率计算值
    is_rework BOOLEAN;      -- 是否返工
BEGIN
```

- `CREATE OR REPLACE FUNCTION process_work_day(current_day INT)`：创建或替换一个名为 `process_work_day` 的函数，接受一个整数参数 `current_day`，表示当前日期。
- `RETURNS void`：函数返回类型为 `void`，表示没有返回值。
- `DECLARE`：声明函数内部使用的变量。
  - `task_rec RECORD`：声明一个记录变量，用于存储查询结果。
  - `work_done INT`：声明一个整数变量，用于存储当日实际完成量。
  - `is_invalid BOOLEAN`：声明一个布尔变量，用于标记是否为无效日。
  - `new_remaining INT`：声明一个整数变量，用于存储更新后的剩余工作量。
  - `eff NUMERIC(10,2)`：声明一个数值变量，用于存储效率计算值，保留两位小数。

- `is_rework BOOLEAN`：声明一个布尔变量，用于标记是否为返工。

遍历所有进行中的任务

```SQL
-- 遍历所有进行中的任务
FOR task_rec IN (
    SELECT
        tp.*,
        p.initial_production_rate,
        tr.available_workers
    FROM task_progress tp
    JOIN process p ON tp.process = p.process
    JOIN trade_resource tr ON tp.assigned_trade = tr.trade
    WHERE tp.status = 'in_progress'
)
LOOP
```

- `FOR task_rec IN (SELECT ...)`：使用游标循环查询所有进行中的任务。
- `SELECT`：从 `task_progress` 表中选择进行中的任务，并关联 `process` 表和 `trade_resource` 表，获取初始生产率和可用工人数。
  - `tp.*`：选择 `task_progress` 表中的所有字段。
  - `p.initial_production_rate`：从 `process` 表中获取初始生产率。
  - `tr.available_workers`：从 `trade_resource` 表中获取可用工人数。
- `WHERE tp.status = 'in_progress'`：筛选条件，只选择状态为进行中的任务。

处理每个任务

```SQL
        -- 生成10%无效日概率
        is_invalid := (random() < 0.1);

        -- 计算实际完成量：无效日为0，否则应用扰动公式
        work_done := CASE
            WHEN is_invalid THEN 0
            ELSE GREATEST(1, ROUND(task_rec.initial_production_rate
* (0.9 + 0.2 * random())) * task_rec.available_workers)
        END;

        -- 更新剩余工作量
        new_remaining := CASE
            WHEN task_rec.remaining_quantity <= work_done THEN 0
            ELSE task_rec.remaining_quantity - work_done
        END;

        -- 更新任务进度表
        UPDATE task_progress
        SET
            remaining_quantity = new_remaining,
            last_update_day = current_day,
            daily_productivity = work_done,
            status = CASE
                        WHEN new_remaining = 0 THEN 'completed'
                        ELSE 'in_progress'
                    END
        WHERE process = task_rec.process AND floor =
task_rec.floor;

        -- 任务完成时释放工种资源
        IF new_remaining = 0 THEN
            UPDATE trade_resource
            SET status = 'available'
            WHERE trade = task_rec.assigned_trade;
        END IF;

        -- 判断返工：任务启动日早于当前日
        is_rework := (task_rec.start_day IS NOT NULL AND
task_rec.start_day < current_day);
```

```
        -- 计算效率百分比
        IF (task_rec.initial_production_rate *
task_rec.available_workers) > 0 THEN
            eff := ROUND((work_done::NUMERIC /
(task_rec.initial_production_rate * task_rec.available_workers)) *
100, 2);
        ELSE
            eff := 0;
        END IF;

        -- 插入每日详细记录
        INSERT INTO daily_task_record (
            day_number, process, floor, trade,
            is_invalid, is_rework, daily_work_done, efficiency,
            start_day, complete_day
        ) VALUES (
            current_day,
            task_rec.process,
            task_rec.floor,
            task_rec.assigned_trade,
            is_invalid,
            is_rework,
            work_done,
            eff::TEXT || '%',
            task_rec.start_day,
            CASE WHEN new_remaining = 0 THEN current_day ELSE NULL
END
        );
    END LOOP;
```

- `is_invalid := (random() < 0.1)`：生成一个随机数，判断是否为无效日（10%的概率）。
- `work_done := CASE ...`：计算当日实际完成量，无效日为0，否则应用扰动公式。
- `new_remaining := CASE ...`：更新剩余工作量，如果剩余工作量小于等于完成量，则置为0，否则减去完成量。
- `UPDATE task_progress`：更新任务进度表，设置新的剩余工作量、最后更新日、实际日效率和任务状态。
- `IF new_remaining = 0 THEN ...`：如果任务完成，更新工种资源表，将工种状态标记为可用。
- `is_rework := (task_rec.start_day IS NOT NULL AND task_rec.start_day < current_day)`：判断是否为返工，即任务启动日早于当前日。

- `eff := ROUND(...)`：计算效率百分比，对比理论最大值。
- `INSERT INTO daily_task_record`：将每日详细记录插入到 `daily_task_record` 表中，包括当日完成量、效率、是否无效日、是否返工等信息。

启动新任务

```SQL
    -- 启动新任务
    PERFORM update_task_progress(current_day);
END;
$$ LANGUAGE plpgsql;
```

- `PERFORM update_task_progress(current_day)`：调用 `update_task_progress` 函数，启动符合条件的新任务。
- `END`：结束函数定义。
- `$$ LANGUAGE plpgsql`：指定函数的实现语言为PL/pgSQL。

总结

这段SQL脚本定义了一个PL/pgSQL函数 `process_work_day`，用于处理每日工作任务，更新任务进度，并记录每日详细信息。函数通过遍历进行中的任务，模拟每日工作情况，更新任务状态和资源状态，并将详细信息记录到 `daily_task_record` 表中，为项目的进度管理和分析提供了全面的数据支持。

步骤8：执行模拟

这段SQL脚本使用匿名代码块来模拟每日工作，直到所有任务完成或达到365天。下面是对代码的逐步解释：

步骤8：执行模拟

```SQL
/*=================================================================
 步骤8：执行模拟
=================================================================*/
```

- 这是一个注释块，说明接下来的代码块的目的是执行模拟。

匿名代码块定义

```SQL
-- 使用匿名代码块模拟每日工作，直到所有任务完成或达到365天
DO $$
DECLARE
    current_day INTEGER := 0;
    max_days INTEGER := 365;  -- 最大模拟天数
BEGIN
```

- `DO $$ ... $$`：定义一个匿名代码块。
- `DECLARE`：声明代码块内部使用的变量。
  - `current_day INTEGER := 0`：声明一个整数变量 `current_day`，初始化为0，用于记录当前模拟天数。
  - `max_days INTEGER := 365`：声明一个整数变量 `max_days`，初始化为365，用于记录最大模拟天数。

清空历史记录表

```SQL
    -- 清空历史记录表
    TRUNCATE TABLE daily_work_history;
    TRUNCATE TABLE daily_task_record;
```

- `TRUNCATE TABLE daily_work_history`：清空 `daily_work_history` 表中的所有数据。
- `TRUNCATE TABLE daily_task_record`：清空 `daily_task_record` 表中的所有数据。

按天循环处理

```SQL
    -- 按天循环处理
    WHILE current_day < max_days LOOP
```

- `WHILE current_day < max_days LOOP`：使用 `WHILE` 循环，按天处理，直到 `current_day` 达到 `max_days`。

记录当日状态到历史表

```SQL
-- 记录当日状态到历史表
INSERT INTO daily_work_history (
    day_number, floor, process, trade,
    planned_remaining, actual_done, is_valid
)
SELECT
    current_day,
    tp.floor,
    tp.process,
    tp.assigned_trade,
    tp.planned_remaining,
    COALESCE(tp.daily_productivity, 0),
    (COALESCE(tp.daily_productivity,0) > 0)
FROM task_progress tp;
```

- `INSERT INTO daily_work_history`：将当日任务状态记录到 `daily_work_history` 表中。
- `SELECT`：从 `task_progress` 表中选择数据。
  - `current_day`：当前模拟天数。
  - `tp.floor`：楼层。
  - `tp.process`：工序名称。
  - `tp.assigned_trade`：分配的工种。
  - `tp.planned_remaining`：计划剩余量。
  - `COALESCE(tp.daily_productivity, 0)`：实际完成量，如果为 `NULL` 则默认为0。
  - `(COALESCE(tp.daily_productivity,0) > 0)`：判断是否为有效日（实际完成量大于0）。

执行当日工作

```SQL
-- 执行当日工作
PERFORM process_work_day(current_day);
```

- `PERFORM process_work_day(current_day)`：调用 `process_work_day` 函数，处理当日工作任务。

检查是否全部任务完成

```SQL
        -- 检查是否全部任务完成
        EXIT WHEN NOT EXISTS (
            SELECT 1 FROM task_progress WHERE status != 'completed'
        );
        current_day := current_day + 1;
    END LOOP;
```

- `EXIT WHEN NOT EXISTS (SELECT 1 FROM task_progress WHERE status != 'completed')`：检查是否所有任务都已完成，如果已完成则退出循环。
- `current_day := current_day + 1`：增加当前天数。

输出模拟结果

```SQL
    RAISE NOTICE 'Simulation completed in % days', current_day;  -- 输出模拟结果
END $$;
```

- `RAISE NOTICE`：输出模拟完成时的天数。

总结

这段SQL脚本使用匿名代码块模拟每日工作，直到所有任务完成或达到365天。通过循环调用 `process_work_day` 函数，处理每日工作任务，并将任务状态记录到 `daily_work_history` 表中。模拟完成后，输出完成所需的天数。

步骤9：查看结果

这段SQL脚本的作用是创建一个视图来汇总项目进度详情，并查询任务进度和每日详细记录。下面是对代码的逐步解释：

步骤9：查看结果

```SQL
/*=============================================================
 步骤9：查看结果
=============================================================*/
```

- 这是一个注释块，说明接下来的代码块的目的是查看模拟结果。

创建视图：汇总项目进度详情

```sql
-- 创建视图：汇总项目进度详情（含工种状态、累计完成量等）
CREATE OR REPLACE VIEW project_progress_details AS
SELECT
    dwh.day_number AS "Day",
    dwh.floor AS "Floor",
    dwh.process AS "Process",
    p.trade AS "Process Trade",
    r.status AS "Trade Status",
    t.initial_quantity AS "Total Work",
    dwh.planned_remaining AS "Plan Remaining",
    dwh.actual_done AS "Daily Done",
    (t.initial_quantity - dwh.planned_remaining) AS "Cumulative
Done",
    CASE WHEN dwh.is_valid THEN 'Valid' ELSE 'Invalid' END AS
"Validity",
    ROUND(dwh.actual_done::NUMERIC / p.initial_production_rate *
100) || '%' AS "Efficiency",
    tp.status AS "Task Status"
FROM daily_work_history dwh
JOIN task t USING (process)
JOIN process p USING (process)
JOIN task_progress tp ON dwh.process = tp.process AND dwh.floor =
tp.floor
JOIN trade_resource r ON tp.assigned_trade = r.trade;
```

- `CREATE OR REPLACE VIEW project_progress_details`：创建或替换一个名为 `project_progress_details` 的视图，用于汇总项目进度详情。
- `SELECT`：从多个表中选择数据并进行连接。
  - `dwh.day_number AS "Day"`：从 `daily_work_history` 表中获取模拟天数。
  - `dwh.floor AS "Floor"`：从 `daily_work_history` 表中获取楼层。
  - `dwh.process AS "Process"`：从 `daily_work_history` 表中获取工序名称。
  - `p.trade AS "Process Trade"`：从 `process` 表中获取工序所属的工种。
  - `r.status AS "Trade Status"`：从 `trade_resource` 表中获取工种状态。
  - `t.initial_quantity AS "Total Work"`：从 `task` 表中获取总工作量。
  - `dwh.planned_remaining AS "Plan Remaining"`：从 `daily_work_history` 表中获取计划剩余量。

    - `dwh.actual_done AS "Daily Done"`：从 `daily_work_history` 表中获取当日实际完成量。
    - `(t.initial_quantity - dwh.planned_remaining) AS "Cumulative Done"`：计算累计完成量。
    - `CASE WHEN dwh.is_valid THEN 'Valid' ELSE 'Invalid' END AS "Validity"`：判断当日是否为有效日。
    - `ROUND NUMERIC / p.initial_production_rate * 100  || '%' AS "Efficiency"`：计算并格式化效率百分比。
    - `tp.status AS "Task Status"`：从 `task_progress` 表中获取任务状态。
- `FROM daily_work_history dwh`：从 `daily_work_history` 表开始连接。
- `JOIN task t USING (process)`：连接 `task` 表，使用 `process` 字段。
- `JOIN process p USING (process)`：连接 `process` 表，使用 `process` 字段。
- `JOIN task_progress tp ON dwh.process = tp.process AND dwh.floor = tp.floor`：连接 `task_progress` 表，使用 `process` 和 `floor` 字段。
- `JOIN trade_resource r ON tp.assigned_trade = r.trade`：连接 `trade_resource` 表，使用 `trade` 字段。

查询最终任务进度

```sql
-- 查询最终任务进度
SELECT * FROM task_progress;
```

- `SELECT * FROM task_progress`：查询 `task_progress` 表中的所有记录，查看最终的任务进度。

查询每日详细记录

```sql
-- 查询每日详细记录（按天、工序、楼层排序）
SELECT * FROM daily_task_record ORDER BY day_number, process,
floor;
```

- `SELECT * FROM daily_task_record ORDER BY day_number, process, floor`：查询 `daily_task_record` 表中的所有记录，并按天数、工序和楼层排序，查看每日的详细记录。

总结

这段SQL脚本创建了一个视图 `project_progress_details`，用于汇总项目进度的详细信息，包括工种状态、累计完成量等。同时，通过查询 `task_progress` 和

`daily_task_record` 表，可以查看最终的任务进度和每日的详细记录，为项目的进度管理和分析提供了全面的数据支持。

章节总结

以上内容是一个完整的SQL脚本，用于模拟和管理建筑项目的每日工作进度。以下是对其内容的总结：

环境清理

- **删除表和视图**：脚本开始时清理了所有相关的表和视图，确保在重新运行脚本前数据库处于干净状态。

基础表结构创建

- **工序表**：存储工序的基本信息，包括工序名称、所属工种和初始生产率。
- **依赖关系表**：记录工序间的依赖关系，即哪些工序必须在其他工序之前完成。
- **任务表**：存储每个工序的初始任务量，带有一定的随机扰动。
- **空间表**：分配每个工序到不同的楼层，并记录每个楼层的工作量。

插入基础数据

- **工序数据**：插入了多个工序的详细信息，包括名称、工种和生产率。
- **任务数据**：基于原始数据插入了带随机扰动的任务数据。
- **依赖关系数据**：定义了工序之间的执行顺序。
- **空间数据**：为每个工序在1到5层生成了相应的空间数据。

资源与任务进度跟踪表

- **工种资源表**：记录各工种的可用工人数和状态。
- **任务进度表**：跟踪每个工序在各楼层的详细进度，包括剩余工作量、状态、开始日等信息。
- **初始化数据**：为工种资源和任务进度表插入了初始数据。

更新任务进度的函数

- **功能**：根据当前日期更新任务进度，启动符合条件的新任务，考虑了依赖关系、楼层顺序和工种可用性。

每日记录表

- **每日工作状态历史记录**：记录每天每层每个工序的工作状态，包括计划剩余量、实际完成量等。
- **每日任务详细记录**：记录每日任务的详细信息，如是否为无效日、返工情况、效率等。

扩展处理每日工作的函数

- **功能**：处理每日工作任务，更新任务进度，记录每日详细信息，包括实际完成量、效率计算等。

执行模拟

- **匿名代码块**：模拟每日工作，直到所有任务完成或达到365天，记录每日状态并执行工作任务。

查看结果

- **视图创建**：创建视图汇总项目进度详情，包括工种状态、累计完成量等。
- **查询结果**：查询并展示最终的任务进度和每日详细记录。

总结

该SQL脚本构建了一个基于代理的建筑项目仿真系统，通过环境初始化、表结构设计（`process`定义工序基准、`dependency`管理工序逻辑）、扰动数据生成（任务量±10%波动、10%无效日概率）及动态资源分配（`trade_resource`状态切换），实现了对施工全流程的数字化模拟。系统以工序依赖（如“碎石基层”完成方可启动“地板管道”）、楼层顺序约束（低楼层未完成则高楼层任务阻塞）为核心规则，结合`task_progress`跟踪剩余工作量与状态，并通过`process_work_day`函数模拟每日施工效率（基于`initial_production_rate`计算实际产出），动态记录扰动影响、返工标记及任务完成情况。自动化模拟流程生成`daily_task_record`详细日志与`project_progress_details`可视化视图，实时反馈累计完成量、工种状态及效率偏差，为管理者提供风险预警、资源调度及进度优化依据。该系统通过数据驱动的精细化管控，有效提升施工效率与可预测性，推动建筑行业从传统管理向智能化、高精度决策转型，为数字化转型提供了可落地的技术范本。