系统架构与文件调度算法说明

本文旨在帮助阅读者快速理解 ZooKeeper HA Demo 的整体拓扑，并重点拆解后端文件调度算法的输入、决策
backend/app.

1. 系统拓扑概览

整体拓扑可概括为 “ZooKeeper 集群 + 监控链路 + 日志链路 + Demo 控制平面”四部分，下图为文字版连线说明：

```
[Workload UI] --(REST)--> [FastAPI Backend] --> [Demo Files on /data/uploads]
           |                                \--(Kazoo)-> [ZooKeeper Cluster]
           |                                \--(Prometheus Pull)-> [Prometheus -> Grafana]
           |                                \--(Filebeat)-> [Elasticsearch]
           \--(Static Assets)-> [/ui + /assets + /partials]
```

- **ZooKeeper 集群**：`docker-compose.yml` 启动 3 个 3.9 版本容器（`zk1`/`zk2`/`zk3`），每个节点开放 4lw 命令，提供真实的主从选举状态。
- **监控链路**：每个 ZK 节点旁挂 `zk-exporter`，Prometheus 通过 `prometheus/prometheus.yml` 抓取，再由 Grafana 展示。Exporter 之外，后端也通过 `/metrics` 暴露自定义 Gauge，供同一套 Prometheus 抓取。
- **日志链路**：Filebeat 读取 `./logs` 与 Docker 日志，推送至 Elasticsearch，前端 `logs.html` 可查询。
- **后端 FastAPI 服务**（`backend` 容器）：负责 Demo 文件生成、调度、节点控制、Prometheus 拉取、自身指标和 API。
- **前端静态站点**：`frontend/` 由 Nginx（通过 FastAPI `StaticFiles`）暴露，为工作负载、概览、日志三套页面提供交互。
- **数据目录**：`./data/uploads/<node>` 保存演示文件；`/app/data/demo.db` (SQLite) 存储文件、操作日志、任务记录等。

2. 后端服务职责结构

模块	关键职责
`main.py`	FastAPI 入口、路由、Prometheus 指标、调度计划/执行逻辑、节点管控 API。
`storage.py`	与磁盘交互：保存上传、生成 demo 文件、计算各节点文件数、迁移/删除文件。
`db.py`	SQLite 封装：记录运维操作、文件元数据、任务队列（用于模拟工作负载）。
`zookeeper_utils.py`	卡祖 (Kazoo) 客户端、4lw 命令封装、文件元数据在 ZK 中的登记、注册文件列表。

模块	关键职责
workload.py	Demo workload 生成器，持续制造上传/迁移/任务事件，保证界面有数据。
docker_control.py	直接调用 Docker socket，实现节点容器的 stop/start/restart。

2.1 配置来源

- .env / docker-compose.yml 环境变量 -> config.py -> Settings 数据类。
- 关键项：
  - ZK_NODES: 逗号分隔的 host:port 列表。
  - FILE_STORAGE_PATH: Demo 文件落盘目录（在容器内挂载）。
  - AUTO_SCHEDULER_INTERVAL、SCHEDULER_THRESHOLD：调度算法核心参数。
  - DEMO_WORKLOAD_*: 控制自动 workload 的频率、文件大小、最大数量等。

3. Demo 文件生命周期

1. **生成**：storage.create_demo_file 或 storage.save_upload 在 FILE_STORAGE_PATH 下以节点名分目录写入随机内容。
2. **落库**：db.create_file_record 写入 files 表（包含历史 history 字段，用于记录每次迁移的事件）。
3. **注册 ZK**：zookeeper_utils.register_file_metadata 在 /{ZK_FILE_ROOT}/{uuid} 下保留 JSON 元数据，供前端比对、Grafana 展示。
4. **可观测性**：refresh_metrics() 汇总 Prometheus + DB 数据，更新自定义 Gauges。前端通过 /api/overview、/api/scheduler/diagnostics 轮询。
5. **迁移**：调度器触发后调用 storage.migrate_file，移动到目标节点目录，同步更新 DB & ZooKeeper。

3.1 数据库表速览

表名	用途	重要字段
files	Demo 文件元数据	node、path、size_bytes、history(JSON)
operations	调度/节点操作审计	action (auto_migrate、drain...)，before_metrics / after_metrics
tasks	Demo task workflow	status（queued/running/...）、payload（JSON）

表名	用途	重要字段
node_state（逻辑字段）	通过 db.get_node_states() 读取，持久化 drain 状态（同 SQLite 表）	

4. 调度器设计目标

- **均衡节点文件数**：尽量使每个节点的 Demo 文件数量差值不超阈值 SCHEDULER_THRESHOLD（默认 5）。
- **尊重运维意图**：通过 /api/nodes/{node}/drain 手动摘除节点后，调度器优先将其剩余文件迁走，并且
- **控制迁移节奏**：AUTO_SCHEDULER_INTERVAL（默认 15s）限制自动调度频率；阈值机制避免频繁腾挪。
- **可解释性**：build_scheduler_plan() 输出详细 reason/message，前端可直接展示，便于理解“不迁移”
  below_threshold、no_files、single_target 等）。

5. 调度计划构建流程（build_scheduler_plan）

1. **收集输入**
   - db.get_files()：获取全部文件记录（按创建时间降序，方便挑选最新文件作为候选）。
   - db.get_node_states()：读取节点的 drain 状态。
   - settings.zk_nodes：构建所有节点初始计数。
2. **计算每节点文件数**
   - counts 默认 0，遍历文件累加。
   - 生成 plan["nodeStates"]，理清每个节点的 drain 状态、原因、更新时间，供 UI 显示。
3. **确定源/目标节点**
   - 如果存在“被 drain 且仍有文件”的节点，先在这些节点里找文件最多的作为 sourceNode，确保摘除时优先清空。
   - 否则在所有节点中找 max(count) 作为源。
   - 目标节点首先从 “未 drain 的节点” 中选 min(count)，若全被摘除，则退化为整体最少者（表示虽然
4. **挑选候选文件**
   - 选取 files 列表中第一个 node == sourceNode 的记录，构造 plan["candidate"]（包含文件大小、最近动作、历史长度等）。
   - 若源节点没有记录，则返回 reason = "no_candidate"。
5. **决策逻辑与原因字段**
   - delta = source_count - target_count。

- 只有当 `delta >= scheduler_threshold` 时才允许迁移。
- 若没有可用目标（例如所有节点都 drained）或源=目标，则设置相应 `reason` 并退出。
- 最终可迁移时写入 `shouldMigrate = True` 与友好的 `message`：如“节点 zk1 比 zk2 多 8 个文件，准备迁移 demo-123.bin”。

6. **附加信息**
   - `plan["counts"]`: 每节点文件数。
   - `plan["drainedNodes"]`: 所有 drain 节点列表。
   - `plan["totalFiles"]`、`plan["delta"]`、`plan["threshold"]` 等用于前端绘制图表与文字说明。

5.1 伪代码

```
def build_scheduler_plan():
    files = db.get_files()
    node_states = db.get_node_states()
    counts = init_counts(settings.zk_nodes)
    for record in files:
        counts[record["node"]] += 1

    source_node = pick_source(counts, node_states.drained)
    target_node = pick_target(counts, node_states.drained)
    candidate = find_first(files, node=source_node)

    delta = counts[source_node] - counts[target_node]
    if delta < settings.scheduler_threshold:
        return Plan(reason="below_threshold")
    if target_node is None or source_node == target_node:
        return Plan(reason="no_target")
    return Plan(shouldMigrate=True, candidate=candidate, delta=delta)
```

5.2 示例：三节点场景

假设当前 `counts = {zk1: 12, zk2: 5, zk3: 4}`，阈值 `5`：

1. `source_node = zk1`（最多文件），`target_node = zk3`（最少文件）。
2. `delta = 8 >=` 阈值，允许迁移。
3. 候选文件 `demo-ab12.bin` 在 `zk1`。
4. `plan.message = "  zk1  zk3  8  demo-ab12.bin "`

若此时 `zk3` 被 drain，则 `target_node = zk2`，`delta = 7` 仍满足条件；UI 会提示 `zk3` 已摘除。

6. 迁移执行流程 (maybe_rebalance_files)

1. 再次调用 build_scheduler_plan()，若 shouldMigrate=False 或无 candidate 立即返回。
2. 从 plan 读取源/目标节点并校验（防止同时 drain / 仅一节点）。
3. storage.migrate_file(candidate, target_node)：
   - 实际移动磁盘文件到目标节点目录，同时返回新路径。
   - 如果文件不存在会抛错，外层捕获后下一轮重试。
4. **更新历史**：解析 candidate["history"]，追加 {"action":"auto_migrate","from":...,"to":...} 事件，再写回 db.update_file_record(...)。
5. **同步 ZooKeeper**：调用 zookeeper_utils.register_file_metadata，将新的节点、路径、历史写入对 znode，供其他消费者读取。
6. **记录操作与刷新指标**：db.record_operation(action="auto_migrate", ...) 方便 UI/日志检索；refresh_metrics() 重刷 Prometheus Gauge。
7. 返回 True 供调用方（自动循环或 API）得知本轮是否真的迁移。

6.1 调度回合示例

步骤	描述
1	auto_scheduler_loop tick=0，调用 maybe_rebalance_files()
2	build_scheduler_plan 发现 zk1→zk2 差值 6，触发迁移
3	storage.migrate_file 将 demo-aa22.bin 从 /data/uploads/zk1 移到 /data/uploads/zk2
4	db.update_file_record + zookeeper_utils.register_file_metadata 同步信息
5	db.record_operation 添加一条 auto_migrate 日志，before_metrics 记录迁移前 counts
6	refresh_metrics 更新 Gauges，前端下一次轮询即可看到差值下降

7. 自动与手动调度的触发点

- **自动循环**：auto_scheduler_loop() 在应用启动且 AUTO_SCHEDULER_ENABLED=true 时后台运行。逻辑非常简单：每隔 AUTO_SCHEDULER_INTERVAL 秒调用一次

`maybe_rebalance_files()`。

- **手动触发**：
  - `POST /api/scheduler/run`：立即尝试迁移一次并返回 before/after 计划。
  - `POST /api/files/bulk-generate` 与 `POST /api/demo/stress` 请求都提供 `trigger_scheduler` 开关，便于在制造不均衡后马上尝试调度。
  - 前端工作负载面板会在曲线下方展示“触发调度中…”的进度条，并以 `schedulerInfo` 更新提示。
- **诊断接口**：`GET /api/scheduler/diagnostics` 始终返回当前计划，可用于 CLI `curl` 或 Grafana 面板的 JSON 数据源：

```
curl http://localhost:8080/api/scheduler/diagnostics | jq
```

7.1 相关 API 详细说明

Endpoint	方法	描述	关键字段
`/api/scheduler/diagnostics`	GET	返回最新调度计划	`counts`、`candidate`、`reason`
`/api/scheduler/run`	POST	手动执行一次迁移	响应含 `executed` 布尔值与 before/after
`/api/files/bulk-generate`	POST	大量创建文件，可选触发调度	`mode`(auto/pin)、`trigger_scheduler`
`/api/demo/stress`	POST	在指定节点集中造文件	`node`、`files`、`trigger_scheduler`
`/api/nodes/{node}/drain`	POST	标记节点为 drain	请求体可附 `reason`
`/api/nodes/{node}/undrain`	POST	取消 drain	——

8. 节点 Drain & 保护机制

- `POST /api/nodes/{node}/drain` 将节点标记为不接受新文件（`db.set_node_state`），调度计划里该节 `drainedNodes`。
- 被 drain 的节点仍可作为源，调度逻辑会优先把其文件搬空；但不会作为目标（除非所有节点都被 drain，系统被迫选最少的一个）。
- `POST /api/nodes/{node}/undrain` 使节点重新参与调度池。
- 前端 UI 会把 drain 状态/原因显示在节点卡片上，同时 Prometheus Gauge 也会维持 `drained` 标签，便于告警。

9. 可观测性与调试建议

- **Prometheus 指标**：

  - `zk_demo_files_per_node{node="zk1"}` 反映调度目标；`zk_demo_node_up`/`zk_demo_avg_latency` 反映 ZK 运行状况。
  - `zk_demo_tasks_total{status="queued"}` 来自 Demo workload，可用于展示压力场景。
- **UI / Logs**：`frontend/workload.html` 和 `frontend/overview.html` 会轮询调度计划并可视化 `counts`、`delta`、`threshold`。
  - 如果调度器没有动作，首先查看 `plan.reason` 是否为 `below_threshold` 或 `no_target`。
  - 通过 `frontend/logs.html` 或 `db.list_operations()` 可确认 `auto_migrate` 操作是否被记录。
- **调参路径**：在 `docker-compose.yml` 或环境变量中调整 `AUTO_SCHEDULER_INTERVAL`、`SCHEDULER_THRESH` backend 即可生效。

10. 扩展方向（供实施时参考）

1. **多文件批次迁移**：当前每轮只迁移 1 个文件，可基于 `delta // threshold` 进行批量调度，但需要在 `storage.migrate_file` 前增加容量/磁盘校验。
2. **容量感知**：可在 `counts` 之外引入文件总大小或磁盘使用率，修改 `select_target_node` 排序策略。
3. **可插拔策略**：`plan["reason"]` 与前端展示已经是通用结构，可以在 `build_scheduler_plan` 外层添加策略工厂，方便接入“最短响应时间”“节点权重”等算法。
4. **任务感知调度**：目前只考虑文件数量，可结合 `tasks` 表的运行情况（例如正在 drain 的节点仍有重要任务时延迟迁移）。

11. 运行与调试小贴士

1. **一键启动**：在根目录执行 `docker compose up -d --build`，等待 zk、exporter、prometheus、grafana、backend 容器全部 healthy。
2. **快速确认调度状态**：

   ```
   curl -s http://localhost:8080/api/scheduler/diagnostics | jq '.counts, .reason, .messag
   ```

3. **查看操作日志**：

   ```
   sqlite3 backend/data/demo.db 'select timestamp, action, node, status, details from oper
   ```

4. **人工制造倾斜**：

   ```
   curl -X POST http://localhost:8080/api/demo/stress \
     -H 'Content-Type: application/json' \
     -d '{"node":"zk1","files":12,"trigger_scheduler":true}'
   ```

   该命令会在 `zk1` 上集中造 12 个文件，并在完成后主动触发调度，便于演示。

5. **调试 Kazoo 连接**：如遇 ZooKeeper 连接异常，可 `docker logs zk-demo-backend | grep kazoo` 检查客户端是否成功连接；必要时重启 backend 容器。
6. **Prometheus 速测**：

   ```
   curl -s 'http://localhost:9090/api/v1/query?query=zk_demo_files_per_node' | jq
   ```

   若指标无返回，检查 Prometheus targets (`http://localhost:9090/targets`) 是否显示 `zk-demo-backend`。

以上内容配合前端可视化，可更全面地掌握整个调度闭环。

通过以上结构，你可以快速定位调度相关逻辑（`main.py` 中的 `build_scheduler_plan`、`maybe_rebalance_fi
API/指标对系统状态进行排查。