在鸿蒙上200行代码实现基于工作流的启动编排设计

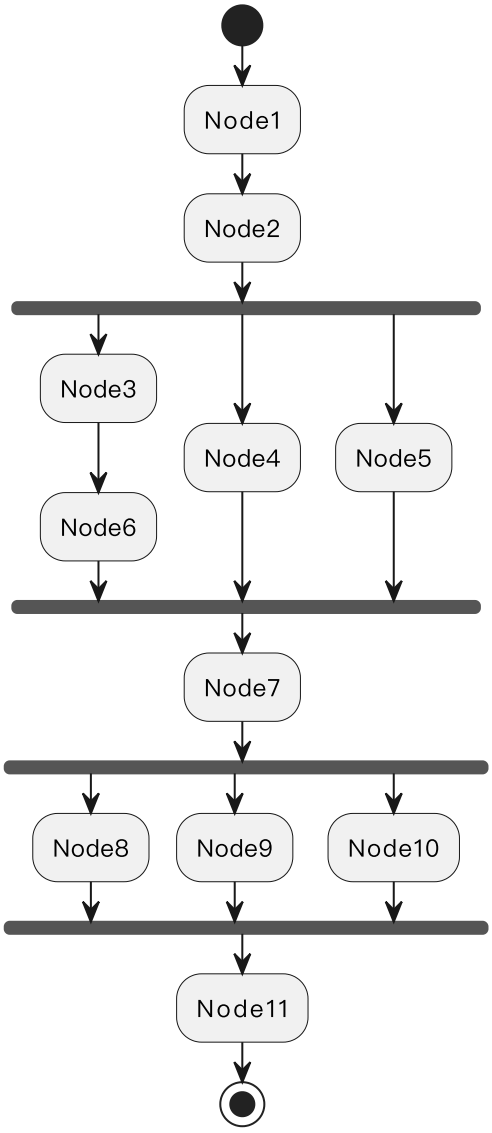
在现代软件系统中，工作流的设计与实现是一个常见且重要的需求。如何用简洁、高效的代码来实现一个支持并发执行、依赖管理的工作流引擎？

本文将带领大家在鸿蒙上一步一步实现一个基于工作流的启动编排引擎，整个实现不到200行代码。

背景与需求

假设我们有一个启动流程，需要按照一定的顺序和依赖关系来执行多个节点。其中部分节点可以并行执行，部分节点需要等待其他节点完成后才能开始。

启动流程图示例如下：



我们用以下的配置文件来描述这个启动流程：

```
{
 "nodes": [
  {
   "id": "Node1",
   "params": { "xx": "xx" },
   "next": ["Node2"]
  },
  {
   "id": "Node2",
   "next": ["Node3", "Node4", "Node5"]
  },
  {
   "id": "Node3",
   "next": ["Node6"]
  },
  {
   "id": "Node4",
   "next": ["Node7"]
  },
  {
   "id": "Node5",
   "next": ["Node7"]
  },
  {
   "id": "Node6",
   "next": ["Node7"]
  },
  {
   "id": "Node7",
   "next": ["Node8", "Node9", "Node10"]
  },
  {
   "id": "Node8",
   "next": ["Node11"]
  },
  {
   "id": "Node9",
   "next": ["Node11"]
  },
  {
   "id": "Node10",
   "next": ["Node11"]
  },
  {
```

```
    "id": "Node11",
    "next": []
  }
 ]
}
```

这个配置文件定义了节点之间的依赖关系，例如：

- `Node1` 执行完成后，触发 `Node2` 的执行。
- `Node2` 执行完成后，同时触发 `Node3`、`Node4`、`Node5` 的执行。
- `Node3` 执行完成后，触发 `Node6` 的执行。
- `Node4`、`Node5`、`Node6` 都执行完成后，触发 `Node7` 的执行。
- 以此类推。

我们的目标是设计一个工作流引擎，能够按照上述依赖关系，正确地调度和执行各个节点，并且在可能的情况下支持节点的并行执行。

设计思路

为了解决上述需求，我们需要解决以下几个关键问题：

1. **节点的依赖管理**：如何表示节点之间的依赖关系，确定一个节点何时可以开始执行？
2. **并发执行**：如何在满足依赖关系的前提下，尽可能地并行执行节点，提升执行效率？
3. **执行控制**：如何监控节点的执行状态，确保所有节点都被正确地执行？

针对以上问题，我们的设计思路如下：

- **数据结构设计**：
  - 使用 `nodeMap` 存储节点的映射关系，方便快速根据节点 ID 获取节点信息。
  - 使用 `predecessors` 存储每个节点的前驱节点集合，便于判断一个节点的所有依赖是否已经满足。
  - 使用 `executingNodes` 和 `completedNodes` 分别记录正在执行和已经完成的节点，防止重复执行。
- **执行流程**：
  - 初始化时，找出所有没有前驱节点的节点（即 `predecessors` 为空的节点），这些节点可以立即开始执行。
  - 当一个节点执行完成后，更新其后继节点的前驱集合，移除已完成的节点。
  - 如果后继节点的前驱集合为空，表示其所有依赖都已满足，可以开始执行。
- **异步执行与事件驱动**：
  - 每个节点的执行都是异步的，执行完成后立即触发后继节点的检查。
  - 不使用批处理的方式，而是采用事件驱动的模型，使节点可以尽早地开始执行。

代码实现

下面是完整的代码实现，并添加了丰富的注释，帮助理解每个部分的作用。

定义节点和配置接口

```
// 定义节点接口，表示工作流中的一个节点
export interface Node {
  id: string; // 节点的唯一标识符
  params?: object;// 可选的参数
  next: string[]; // 后继节点的 ID 列表
}

// 定义配置接口，包含所有节点和起始节点
export interface Config {
  nodes: Node[]; // 节点列表
}
```

工作流引擎的实现

```
import { Config } from './Config';
import { Node } from './Node';

/**
 * @author zhanglulu
 * @description 工作流引擎类，实现节点的调度和执行
 */
export class WorkflowEngine {
  // 配置信息
  private config: Config;
  // 节点映射，将节点 ID 映射到 Node 对象，便于快速查找
  private nodeMap: Map<string, Node> = new Map();
  // 前驱节点映射，每个节点对应一个前驱节点的集合，用于判断节点是否可以执行
  private predecessors: Map<string, Set<string>> = new Map();
  // 已完成的节点集合，用于记录哪些节点已经执行完成
  private completedNodes: Set<string> = new Set();
  // 正在执行的节点集合，防止重复执行
  private executingNodes: Set<string> = new Set();
  // 用于等待所有节点执行完成的 Promise
  private executionPromise: Promise<void>;
  // 用于在所有节点执行完成时触发的 resolve 函数
  private resolveExecution?: () => void;
  // Logger，用于打印日志
  private loggerEntity: ((msg: string) => void) | undefined = undefined;
```

```
// 节点执行器，用于执行实际的节点逻辑
private nodeExecutor: ((node: Node) => Promise<void>) | undefined = undefined;

/**
 * 构造函数，接受配置对象并初始化
 * @param config
 */
constructor(config: Config) {
  // 配置信息
  this.config = config;
  // 构建节点映射
  this.buildNodeMap();
  // 构建前驱节点映射
  this.buildPredecessors();

  // 在构造函数中初始化 executionPromise，并获取 resolve 函数
  this.executionPromise = new Promise<void>(resolve => {
    this.resolveExecution = resolve;
  });
}

/**
 * 设置 Logger
 * @param logger
 */
public setLogger(logger: (msg: string) => void) {
  this.loggerEntity = logger;
}

/**
 * 设置节点执行函数，用于执行实际的节点逻辑
 * @param nodeExecutor
 */
public setNodeExecutor(nodeExecutor: (node: Node) => Promise<void>) {
  this.nodeExecutor = nodeExecutor;
}

/**
 * 日志打印
 * @param msg
 */
private log(msg: string) {
  if (this.loggerEntity) {
    this.loggerEntity(msg);
  } else {
    console.log(msg);
```

```
  }
}

/**
 * 构建节点映射，方便通过节点 ID 快速获取节点信息
 */
private buildNodeMap() {
  this.config.nodes.forEach(node => {
    this.nodeMap.set(node.id, node);
  });
}

/**
 * 构建前驱节点映射，初始化每个节点的前驱节点集合
 */
private buildPredecessors() {
  // 初始化所有节点的前驱集合为空集合
  this.config.nodes.forEach(node => {
    if (!this.predecessors.has(node.id)) {
      this.predecessors.set(node.id, new Set());
    }
  });

  // 遍历所有节点，填充后继节点的前驱集合
  this.config.nodes.forEach(node => {
    node.next.forEach(nextNodeId => {
      if (!this.predecessors.has(nextNodeId)) {
        this.predecessors.set(nextNodeId, new Set());
      }
      // 将当前节点添加到后继节点的前驱集合中
      this.predecessors.get(nextNodeId)?.add(node.id);
    });
  });
}

/**
 * 执行工作流
 */
public async execute() {
  // 存放准备好的节点
  const readyNodes = new Set<string>();

  // 找出所有没有前驱节点的节点，即可立即执行的节点
  for (let entriesElement of this.predecessors.entries()) {
    // 遍历所有前驱节点集合
    let nodeId = entriesElement[0];
```

```
    let preds = entriesElement[1];
    // 如果前驱集合为空，说明没有前驱节点，即可立即执行
    if (preds.size === 0) {
      readyNodes.add(nodeId);
    }
  }

  // 开始执行所有准备好的节点
  readyNodes.forEach(nodeId => {
    // 异步执行节点
    this.executeNode(nodeId);
  });

  // 等待所有节点执行完成
  await this.executionPromise;
}

/**
 * 执行单个节点
 * @param nodeId
 */
private async executeNode(nodeId: string) {
  // 如果节点正在执行或已完成，直接返回，防止重复执行
  if (this.executingNodes.has(nodeId) || this.completedNodes.has(nodeId)) {
    return;
  }
  // 标记节点为正在执行
  this.executingNodes.add(nodeId);
  // 获取节点对象
  const node = this.nodeMap.get(nodeId);
  if (!node) {
    return;
  }
  this.log(`正在执行节点 ${node.id}`);
  if (!this.nodeExecutor) {
    throw new Error("NodeExecutor is not set!");
  }
  // 真正的执行逻辑
  await this.nodeExecutor(node);
  // 将节点标记为已完成
  this.completedNodes.add(nodeId);
  // 从正在执行的集合中移除
  this.executingNodes.delete(nodeId);

  // 检查是否所有节点都已完成
  if (this.completedNodes.size === this.nodeMap.size) {
```

```
      // 触发流程执行完成的 Promise
      this.resolveExecution?.();
    }

    // 处理后继节点，检查它们是否可以执行
    node.next.forEach(nextNodeId => {
      // 获取后继节点的前驱集合
      const preds = this.predecessors.get(nextNodeId);
      // 从前驱集合中移除当前已完成的节点
      preds?.delete(nodeId);
      if (preds?.size === 0) {
        // 如果前驱集合为空，说明后继节点的所有前驱都已完成，可以开始执行
        this.executeNode(nextNodeId);
      }
    });
  }
}
```

使用示例

```
import { WorkflowEngine } from './WorkflowEngine';
import { Config } from './Config';

// 读取并解析配置文件
const configJson = `...`; // 省略，使用前面的配置文件内容
const config: Config = JSON.parse(configJson);

// 创建工作流引擎实例
const engine = new WorkflowEngine(config);

// 设置日志函数
engine.setLogger((msg: string) => {
  console.log(msg);
});

// 设置节点执行器，模拟节点的执行逻辑
engine.setNodeExecutor(async (node) => {
  // 模拟执行时间
  const delay = node.id === "Node4" ? 5000 : Math.random() * 1000;
  await new Promise(resolve => setTimeout(resolve, delay));
  console.log(`节点 ${node.id} 执行完成`);
});

// 执行工作流
(async () => {
```

```
  await engine.execute();
  console.log('流程执行完成。');
})();
```

运行结果

执行上述代码，可以看到输出结果：

```
正在执行节点 Node1
节点 Node1 执行完成
正在执行节点 Node2
节点 Node2 执行完成
正在执行节点 Node3
正在执行节点 Node4
正在执行节点 Node5
节点 Node3 执行完成
正在执行节点 Node6
节点 Node6 执行完成
节点 Node5 执行完成
节点 Node4 执行完成
正在执行节点 Node7
节点 Node7 执行完成
正在执行节点 Node8
正在执行节点 Node9
正在执行节点 Node10
节点 Node8 执行完成
节点 Node9 执行完成
节点 Node10 执行完成
正在执行节点 Node11
节点 Node11 执行完成
流程执行完成。
```

从结果可以看出：

- `Node1` 执行完成后，`Node2` 开始执行。
- `Node2` 执行完成后，`Node3`、`Node4`、`Node5` 并行执行。
- `Node3` 执行较快，完成后立即开始执行 `Node6`，无需等待 `Node4` 和 `Node5`。
- `Node4` 执行较慢（延迟了5秒），但不影响 `Node6` 的执行。
- `Node4`、`Node5`、`Node6` 都执行完成后，`Node7` 开始执行。
- 后续节点按依赖关系继续执行，直到流程完成。

代码解读

节点执行器与日志器

在 `WorkflowEngine` 中，我们提供了 `setNodeExecutor` 和 `setLogger` 方法，允许用户自定义节点的执行逻辑和日志输出方式。这使得引擎具有更好的灵活性和可扩展性。

```
// 设置节点执行函数，用于执行实际的节点逻辑
public setNodeExecutor(nodeExecutor: (node: Node) => Promise<void>) {
 this.nodeExecutor = nodeExecutor;
}

// 设置 Logger
public setLogger(logger: (msg: string) => void) {
 this.loggerEntity = logger;
}
```

执行流程

整个执行流程基于事件驱动模型，实现了以下逻辑：

- **初始执行**：在 `execute` 方法中，找到所有没有前驱节点的节点，立即开始执行。
- **节点执行完成后处理**：在 `executeNode` 方法中，节点执行完成后，更新后继节点的前驱集合，检查是否可以开始执行后继节点。
- **流程完成检测**：在每个节点执行完成后，检查是否所有节点都已完成，若是，则触发 `resolveExecution`，结束流程。

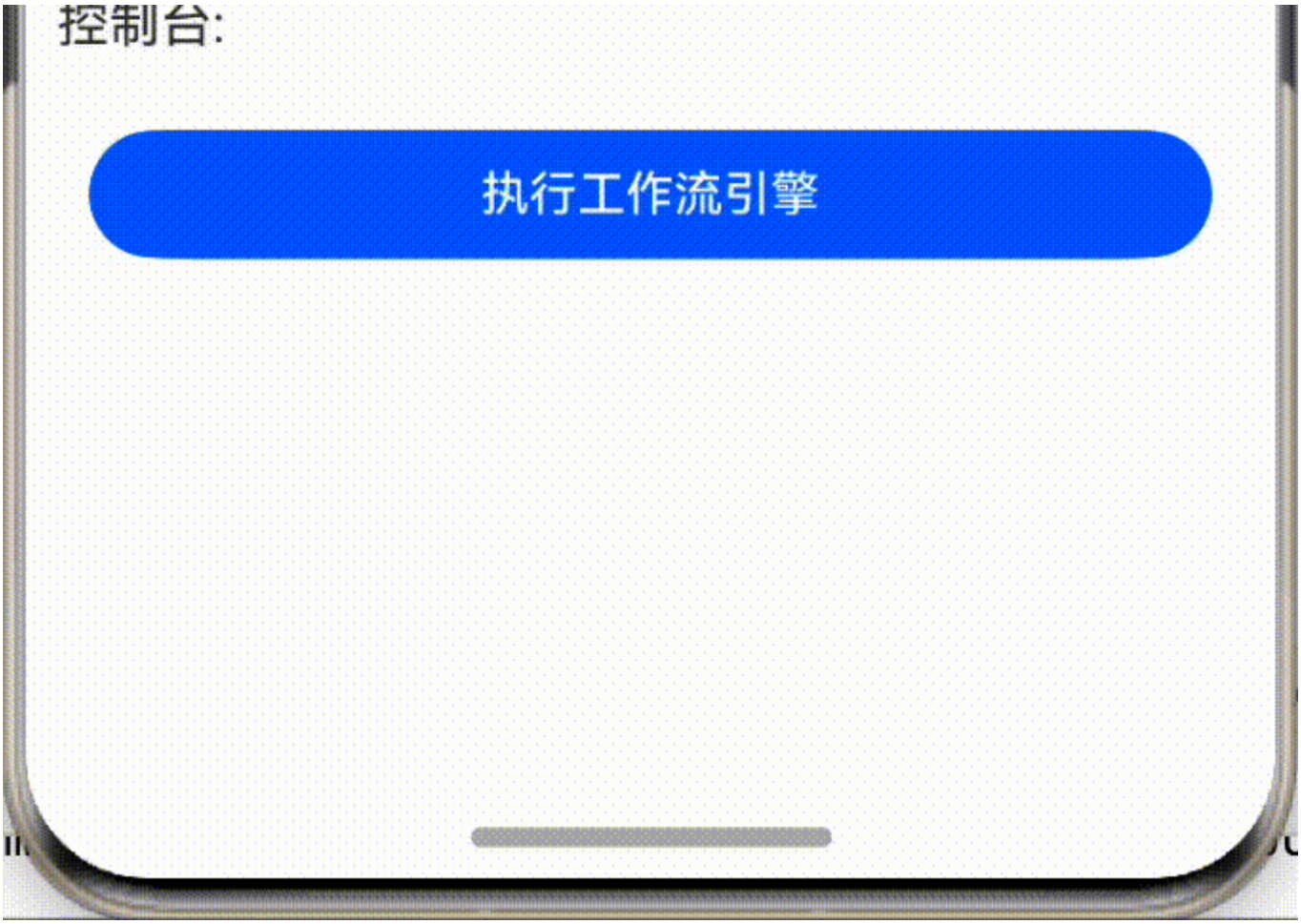
并发执行

由于每个节点的执行都是异步的，并且在依赖关系允许的情况下立即开始执行，整个流程中节点可以并行执行，最大程度地提升了执行效率。

DEMO

GitHub: https://github.com/changer0/HarmonyOSWorkflowEngine


10:25
100
工作流引擎DEMO
输出:
[欢迎使用工作流引擎]



总结

本文展示了如何用不到200行的 ArkTS 代码，实现一个支持并发执行、依赖管理的工作流引擎。通过合理的数据结构设计和事件驱动的执行模型，我们解决了节点的依赖管理和并发执行问题。

这种设计思路不仅适用于启动流程的编排，也可以扩展到其他需要依赖管理和并发执行的场景。希望本文的内容对您有所帮助！

参考

- HarmonyOS Developer：https://developer.huawei.com/consumer/cn/develop/
- TypeScript 官方文档：https://www.typescriptlang.org/docs/