StepFlow Gateway 架构重构计划

🎯 重构目标

将 StepFlow Gateway 重构为支持多种 API 规范的模块化架构，包括 OpenAPI、AsyncAPI 以及未来可能的 GraphQL、gRPC 等。

🏗️ 新的架构设计

1. **核心架构**

```
src/stepflow_gateway/
├── core/                          # 核心模块
│   ├── __init__.py
│   ├── gateway.py                # 主网关类
│   ├── config.py                 # 配置管理
│   ├── registry.py               # API 规范注册器
│   └── base/                     # 基础抽象类
│       ├── __init__.py
│       ├── api_spec.py           # API 规范基类
│       ├── parser.py             # 解析器基类
│       ├── executor.py           # 执行器基类
│       └── protocol.py           # 协议适配器基类
├── plugins/                       # API 规范插件
│   ├── __init__.py
│   ├── openapi/                  # OpenAPI 插件
│   │   ├── __init__.py
│   │   ├── spec.py              # OpenAPI 规范实现
│   │   ├── parser.py            # OpenAPI 解析器
│   │   ├── executor.py          # OpenAPI 执行器
│   │   └── protocols/           # HTTP 协议支持
│   │       ├── __init__.py
│   │       ├── http.py          # HTTP 协议适配器
│   │       └── websocket.py     # WebSocket 协议适配器
│   └── asyncapi/                 # AsyncAPI 插件
│       ├── __init__.py
│       ├── spec.py              # AsyncAPI 规范实现
│       ├── parser.py            # AsyncAPI 解析器
│       ├── executor.py          # AsyncAPI 执行器
│       └── protocols/           # 异步协议支持
│           ├── __init__.py
│           ├── mqtt.py          # MQTT 协议适配器
│           ├── kafka.py         # Kafka 协议适配器
│           ├── amqp.py          # AMQP 协议适配器
│           └── websocket.py     # WebSocket 协议适配器
├── shared/                        # 共享模块
│   ├── __init__.py
│   ├── database/                 # 数据库管理
│   │   ├── __init__.py
│   │   ├── manager.py          # 数据库管理器
│   │   ├── models.py           # 数据模型
│   │   └── migrations/         # 数据库迁移
│   ├── auth/                    # 认证管理
│   │   ├── __init__.py
│   │   ├── manager.py          # 认证管理器
│   │   ├── providers/          # 认证提供者
│   │   └── schemes.py          # 认证方案
│   ├── monitoring/              # 监控和日志
│   │   ├── __init__.py
│   │   ├── logger.py           # 日志管理
│   │   ├── metrics.py          # 指标收集
│   │   └── tracing.py          # 链路追踪
│   └── utils/                   # 工具函数
│       ├── __init__.py
│       ├── validators.py       # 数据验证
│       ├── formatters.py       # 数据格式化
│       └── helpers.py          # 辅助函数
├── web/                         # Web 接口层
│   ├── __init__.py
│   ├── app.py                  # FastAPI 应用
│   ├── routes/                 # 路由定义
│   │   ├── __init__.py
│   │   ├── core.py             # 核心路由
│   │   ├── openapi.py          # OpenAPI 路由
│   │   ├── asyncapi.py         # AsyncAPI 路由
│   │   └── monitoring.py       # 监控路由
│   ├── middleware/             # 中间件
│   │   ├── __init__.py
│   │   ├── cors.py             # CORS 中间件
│   │   ├── auth.py             # 认证中间件
│   │   └── logging.py          # 日志中间件
│   └── models/                 # 请求/响应模型
│       ├── __init__.py
│       ├── common.py           # 通用模型
│       ├── openapi.py          # OpenAPI 模型
│       └── asyncapi.py         # AsyncAPI 模型
└── cli/                         # 命令行工具
    ├── __init__.py
    ├── main.py                 # CLI 主程序
    ├── commands/               # 命令定义
    │   ├── __init__.py
    │   ├── core.py             # 核心命令
    │   ├── openapi.py          # OpenAPI 命令
    │   └── asyncapi.py         # AsyncAPI 命令
    └── utils.py                # CLI 工具函数
```

2. 插件架构设计

核心抽象类

```python
# core/base/api_spec.py
from abc import ABC, abstractmethod
from typing import Dict, Any, List, Optional

class ApiSpecification(ABC):
    """API 规范抽象基类"""

    @property
    @abstractmethod
    def spec_type(self) -> str:
        """规范类型 (openapi, asyncapi, graphql, etc.)"""
        pass

    @property
    @abstractmethod
    def version(self) -> str:
        """规范版本"""
        pass

    @abstractmethod
    def validate(self, content: str) -> bool:
        """验证规范内容"""
        pass

    @abstractmethod
    def parse(self, content: str) -> Dict[str, Any]:
        """解析规范内容"""
        pass

    @abstractmethod
    def extract_endpoints(self, parsed_content: Dict[str, Any]) ->
List[Dict[str, Any]]:
        """提取端点信息"""
        pass

# core/base/parser.py
class BaseParser(ABC):
    """解析器抽象基类"""

    @abstractmethod
    def parse(self, content: str) -> Dict[str, Any]:
        """解析内容"""
        pass

    @abstractmethod
    def validate(self, content: str) -> bool:
        """验证内容"""
        pass

# core/base/executor.py
class BaseExecutor(ABC):
    """执行器抽象基类"""
    @abstractmethod
    async def execute(self, endpoint_id: str, request_data: Dict[str,
Any]) -> Dict[str, Any]:
        """执行 API 调用"""
        pass

    @abstractmethod
    def get_supported_protocols(self) -> List[str]:
        """获取支持的协议"""
        pass

# core/base/protocol.py
class BaseProtocolAdapter(ABC):
    """协议适配器抽象基类"""

    @abstractmethod
    async def connect(self, config: Dict[str, Any]):
        """建立连接"""
        pass

    @abstractmethod
    async def execute(self, operation: str, data: Dict[str, Any]) ->
Dict[str, Any]:
        """执行操作"""
        pass

    @abstractmethod
    async def disconnect(self):
        """断开连接"""
        pass
```

插件注册机制

```python
# core/registry.py
class ApiSpecRegistry:
    """API 规范注册器"""

    def __init__(self):
        self._specs = {}
        self._parsers = {}
        self._executors = {}
        self._protocols = {}

    def register_spec(self, spec_type: str, spec_class: type):
        """注册 API 规范"""
        self._specs[spec_type] = spec_class

    def register_parser(self, spec_type: str, parser_class: type):
        """注册解析器"""
```

```python
            self._parsers[spec_type] = parser_class

    def register_executor(self, spec_type: str, executor_class: type):
        """注册执行器"""
        self._executors[spec_type] = executor_class

    def register_protocol(self, protocol_name: str, protocol_class:
type):
        """注册协议适配器"""
        self._protocols[protocol_name] = protocol_class

    def get_spec(self, spec_type: str) -> Optional[type]:
        """获取规范类"""
        return self._specs.get(spec_type)

    def get_parser(self, spec_type: str) -> Optional[type]:
        """获取解析器类"""
        return self._parsers.get(spec_type)

    def get_executor(self, spec_type: str) -> Optional[type]:
        """获取执行器类"""
        return self._executors.get(spec_type)

    def get_protocol(self, protocol_name: str) -> Optional[type]:
        """获取协议适配器类"""
        return self._protocols.get(protocol_name)
```

3. 数据库设计重构

统一的数据模型

```sql
-- API 规范模板表（统一）
CREATE TABLE api_spec_templates (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    spec_type TEXT NOT NULL,  -- openapi, asyncapi, graphql, etc.
    content TEXT NOT NULL,
    version TEXT,
    status TEXT DEFAULT 'active',
    created_at TEXT NOT NULL,
    updated_at TEXT NOT NULL
);

-- API 文档表（统一）
CREATE TABLE api_documents (
    id TEXT PRIMARY KEY,
    template_id TEXT NOT NULL,
    name TEXT NOT NULL,
    spec_type TEXT NOT NULL,  -- openapi, asyncapi, graphql, etc.
    version TEXT,
    base_url TEXT,
    status TEXT DEFAULT 'active',
    created_at TEXT NOT NULL,
    updated_at TEXT NOT NULL,
    FOREIGN KEY (template_id) REFERENCES api_spec_templates(id)
);

-- 端点表（统一，支持不同类型的端点）
CREATE TABLE api_endpoints (
    id TEXT PRIMARY KEY,
    api_document_id TEXT NOT NULL,
    endpoint_name TEXT NOT NULL,  -- path for OpenAPI, channel for
AsyncAPI
    endpoint_type TEXT NOT NULL,  -- http, mqtt, kafka, websocket, etc.
    method TEXT,                  -- HTTP method for REST APIs
    operation_type TEXT,          -- get, post, publish, subscribe, etc.
    description TEXT,
    parameters TEXT,              -- JSON
    request_schema TEXT,          -- JSON
    response_schema TEXT,         -- JSON
    security TEXT,                -- JSON
    status TEXT DEFAULT 'active',
    created_at TEXT NOT NULL,
    updated_at TEXT NOT NULL,
    FOREIGN KEY (api_document_id) REFERENCES api_documents(id)
);

-- 协议配置表
CREATE TABLE protocol_configs (
    id TEXT PRIMARY KEY,
    api_document_id TEXT NOT NULL,
    protocol_name TEXT NOT NULL,
    protocol_type TEXT NOT NULL,  -- http, mqtt, kafka, amqp, etc.
    config TEXT NOT NULL,         -- JSON
    status TEXT DEFAULT 'active',
    created_at TEXT NOT NULL,
    updated_at TEXT NOT NULL,
    FOREIGN KEY (api_document_id) REFERENCES api_documents(id)
);

-- API 调用日志表（统一）
CREATE TABLE api_call_logs (
    id TEXT PRIMARY KEY,
    endpoint_id TEXT NOT NULL,
    operation_type TEXT NOT NULL,
    request_data TEXT,            -- JSON
    response_data TEXT,           -- JSON
    protocol_type TEXT NOT NULL,
    status TEXT NOT NULL,         -- success/error
    error_message TEXT,
    response_time_ms INTEGER,
    created_at TEXT NOT NULL,
    FOREIGN KEY (endpoint_id) REFERENCES api_endpoints(id)
);
```

4. 插件实现示例

OpenAPI 插件

```
# plugins/openapi/spec.py
from ...core.base.api_spec import ApiSpecification

class OpenApiSpecification(ApiSpecification):
    """OpenAPI 规范实现"""

    @property
    def spec_type(self) -> str:
        return "openapi"

    @property
    def version(self) -> str:
        return "3.0.0"

    def validate(self, content: str) -> bool:
        # OpenAPI 验证逻辑
        pass

    def parse(self, content: str) -> Dict[str, Any]:
        # OpenAPI 解析逻辑
        pass

    def extract_endpoints(self, parsed_content: Dict[str, Any]) ->
List[Dict[str, Any]]:
        # 提取 OpenAPI 端点
        pass

# plugins/openapi/parser.py
from ...core.base.parser import BaseParser

class OpenApiParser(BaseParser):
    """OpenAPI 解析器"""

    def parse(self, content: str) -> Dict[str, Any]:
        # OpenAPI 解析实现
        pass

    def validate(self, content: str) -> bool:
        # OpenAPI 验证实现
        pass

# plugins/openapi/executor.py
from ...core.base.executor import BaseExecutor
```

```python
class OpenApiExecutor(BaseExecutor):
    """OpenAPI 执行器"""

    async def execute(self, endpoint_id: str, request_data: Dict[str,
Any]) -> Dict[str, Any]:
        # OpenAPI 执行实现
        pass

    def get_supported_protocols(self) -> List[str]:
        return ["http", "https", "websocket"]
```

AsyncAPI 插件

```python
# plugins/asyncapi/spec.py
from ...core.base.api_spec import ApiSpecification

class AsyncApiSpecification(ApiSpecification):
    """AsyncAPI 规范实现"""

    @property
    def spec_type(self) -> str:
        return "asyncapi"

    @property
    def version(self) -> str:
        return "2.5.0"

    def validate(self, content: str) -> bool:
        # AsyncAPI 验证逻辑
        pass

    def parse(self, content: str) -> Dict[str, Any]:
        # AsyncAPI 解析逻辑
        pass

    def extract_endpoints(self, parsed_content: Dict[str, Any]) ->
List[Dict[str, Any]]:
        # 提取 AsyncAPI 通道
        pass

# plugins/asyncapi/executor.py
from ...core.base.executor import BaseExecutor

class AsyncApiExecutor(BaseExecutor):
    """AsyncAPI 执行器"""

    async def execute(self, endpoint_id: str, request_data: Dict[str,
Any]) -> Dict[str, Any]:
        # AsyncAPI 执行实现
        pass

    def get_supported_protocols(self) -> List[str]:
        return ["mqtt", "kafka", "amqp", "websocket", "sse"]
```

5. **主网关类重构**

```
# core/gateway.py
class StepFlowGateway:
    """重构后的主网关类"""

    def __init__(self, config: Optional[GatewayConfig] = None):
        self.config = config or load_config()
        self.registry = ApiSpecRegistry()
        self.db_manager = DatabaseManager(self.config.database)
        self.auth_manager = AuthManager(self.db_manager,
self.config.auth)

        # 注册插件
        self._register_plugins()

    def _register_plugins(self):
        """注册所有插件"""
        # 注册 OpenAPI 插件
        from ..plugins.openapi.spec import OpenApiSpecification
        from ..plugins.openapi.parser import OpenApiParser
        from ..plugins.openapi.executor import OpenApiExecutor

        self.registry.register_spec("openapi", OpenApiSpecification)
        self.registry.register_parser("openapi", OpenApiParser)
        self.registry.register_executor("openapi", OpenApiExecutor)

        # 注册 AsyncAPI 插件
        from ..plugins.asyncapi.spec import AsyncApiSpecification
        from ..plugins.asyncapi.parser import AsyncApiParser
        from ..plugins.asyncapi.executor import AsyncApiExecutor

        self.registry.register_spec("asyncapi", AsyncApiSpecification)
        self.registry.register_parser("asyncapi", AsyncApiParser)
        self.registry.register_executor("asyncapi", AsyncApiExecutor)

    def register_api(self, name: str, content: str, spec_type: str,
                     version: str = None, base_url: str = None) ->
Dict[str, Any]:
        """注册 API (统一接口)"""
        try:
            # 获取对应的规范类
            spec_class = self.registry.get_spec(spec_type)
            if not spec_class:
                raise ValueError(f"Unsupported API specification type:
```

```
{spec_type}")

            spec = spec_class()

            # 验证和解析
            if not spec.validate(content):
                raise ValueError(f"Invalid {spec_type} specification")

            parsed_content = spec.parse(content)
            endpoints = spec.extract_endpoints(parsed_content)

            # 保存到数据库
            template_id = self._save_template(name, content, spec_type)
            document_id = self._save_document(template_id, name,
spec_type, version, base_url)
            self._save_endpoints(endpoints, document_id, spec_type)

            return {
                'success': True,
                'template_id': template_id,
                'document_id': document_id,
                'endpoints': endpoints
            }

        except Exception as e:
            return {'success': False, 'error': str(e)}

    async def call_api(self, endpoint_id: str, request_data: Dict[str,
Any]) -> Dict[str, Any]:
        """调用 API (统一接口)"""
        try:
            # 获取端点信息
            endpoint = self.get_endpoint(endpoint_id)
            if not endpoint:
                return {'success': False, 'error': 'Endpoint not found'}

            # 获取对应的执行器
            spec_type = endpoint.get('spec_type', 'openapi')
            executor_class = self.registry.get_executor(spec_type)
            if not executor_class:
                return {'success': False, 'error': f'No executor for
{spec_type}'}

            executor = executor_class(self.db_manager,
self.auth_manager)
            return await executor.execute(endpoint_id, request_data)

        except Exception as e:
            return {'success': False, 'error': str(e)}
```

6. **Web API 重构**

```python
# web/routes/core.py
@router.post("/apis/register")
def register_api(req: ApiRegisterRequest):
    """统一的 API 注册接口"""
    result = gateway.register_api(
        name=req.name,
        content=req.content,
        spec_type=req.spec_type,
        version=req.version,
        base_url=req.base_url
    )
    return result

@router.post("/api/call")
async def call_api(req: ApiCallRequest):
    """统一的 API 调用接口"""
    result = await gateway.call_api(req.endpoint_id, req.request_data)
    return result

# web/routes/openapi.py
@router.get("/openapi/endpoints")
def list_openapi_endpoints():
    """OpenAPI 特定接口"""
    pass

# web/routes/asyncapi.py
@router.get("/asyncapi/channels")
def list_asyncapi_channels():
    """AsyncAPI 特定接口"""
    pass
```

🚀 重构实施计划

阶段 1: 核心架构 (1-2 周)

- [ ] 创建新的目录结构
- [ ] 实现核心抽象类
- [ ] 实现插件注册机制
- [ ] 重构主网关类

阶段 2: 数据库重构 (1 周)

- [ ] 设计统一的数据模型
- [ ] 创建数据库迁移脚本
- [ ] 更新数据库管理器

阶段 3: OpenAPI 插件 (1 周)

- [ ] 将现有 OpenAPI 代码迁移到插件

- [ ] 实现 OpenAPI 规范类
- [ ] 更新 OpenAPI 解析器和执行器

阶段 4: AsyncAPI 插件 (2 周)

- [ ] 实现 AsyncAPI 规范类
- [ ] 实现 AsyncAPI 解析器
- [ ] 实现 AsyncAPI 执行器
- [ ] 实现协议适配器

阶段 5: Web API 重构 (1 周)

- [ ] 重构 Web 路由
- [ ] 实现统一的 API 接口
- [ ] 更新中间件

阶段 6: 测试和优化 (1 周)

- [ ] 编写测试用例
- [ ] 性能优化
- [ ] 文档更新

🎯 重构收益

1. **模块化设计**

- 清晰的职责分离
- 易于扩展新 API 规范
- 插件化架构

2. **统一接口**

- 一致的 API 注册和调用接口
- 统一的数据模型
- 统一的监控和日志

3. **可扩展性**

- 支持未来添加 GraphQL、gRPC 等
- 支持新的协议适配器
- 支持自定义插件

4. **维护性**

- 代码结构清晰
- 减少重复代码
- 易于测试和调试

这个重构计划将为你提供一个强大、灵活、可扩展的 API 网关架构，能够轻松支持各种 API 规范。