WP2-1 対話理解モジュール 技術文書

1. 概要

1.1 モジュールの目的

WP2-1 対話理解モジュールは、Google Gemini API を利用した法律専門の対話 AI 機能を提供する FastAPI ベースの Web サービスです。ユーザーからの自然言語入力を受け取り、法律に関する専門的な回答を生成します。

1.2 主要機能

- **自然言語対話**: ユーザーの質問を理解し、適切な回答を生成
- **法律専門性**: 日本の法律に関する専門知識を活用した回答
- **統一 API**: 他のモジュール(Emotion/Legal)との連携を考慮した設計
- **エラーハンドリング**: タイムアウトや API 制限に対する堅牢な処理
- **構造化ログ**: 監視・デバッグのための詳細なログ出力

---

2. ファイル構成と動作

2.1 ディレクトリ構造

```
WP2-1 対話理解モジュール/
├── app/                          # メインアプリケーション
│   ├── __init__.py               # Python パッケージ初期化
│   ├── main.py                   # FastAPI 起動・ルーティング
│   ├── config.py                 # 環境変数・設定管理
│   ├── logger.py                 # 構造化ログ設定
│   ├── schemas.py                # データモデル定義
│   ├── clients/                  # 外部 API 連携
│   │   ├── __init__.py
│   │   └── gemini_client.py      # Gemini API クライアント
│   ├── prompts/                  # プロンプト設計
│   │   ├── __init__.py
│   │   └── prompt_builder.py     # プロンプト構築ロジック
│   ├── services/                 # ビジネスロジック
```

```
│   │   ├── __init__.py
│   │   └── chat_service.py       # チャット処理サービス
│   └── utils/                    # ユーティリティ
│       ├── __init__.py
│       └── error_mapping.py      # エラーハンドリング
├── .venv/                        # Python 仮想環境
├── .env                          # 環境変数（API キー等）
├── env.example                   # 環境変数テンプレート
├── requirements.txt              # Python 依存関係
└── README.md                     # セットアップ手順
```

2.2 各ファイルの役割

app/main.py – FastAPI アプリケーション
- **役割**: Web サーバーの起動と API エンドポイントの定義
- **主要機能**:
  - `/` – ヘルスチェック
  - `/health` – 詳細ヘルスチェック
  - `/v1/chat` – チャット処理（メイン機能）
- **動作**: HTTP リクエストを受け取り、適切なサービスに処理を委譲

app/config.py – 設定管理
- **役割**: 環境変数から設定値を読み込み、アプリケーション全体で使用
- **管理項目**:
  - Gemini API キー
  - モデル名
  - タイムアウト設定
  - ログレベル
- **動作**: 起動時に`.env`ファイルを読み込み、設定オブジェクトを生成

app/schemas.py – データモデル
- **役割**: リクエスト・レスポンスのデータ構造を定義
- **主要モデル**:
  - `ChatRequest` – チャットリクエスト
  - `ApiResponse` – 統一レスポンス形式

```
  - `Message` - 会話メッセージ
- **動作**: Pydantic による自動バリデーション

#### **app/clients/gemini_client.py** - Gemini API 連携
- **役割**: Google Gemini API との通信を担当
- **主要機能**:
  - HTTP リクエスト送信
  - レスポンス解析
  - エラーハンドリング
  - タイムアウト処理
- **動作**: REST API 経由で Gemini にリクエストを送信し、生成されたテキストを取得

#### **app/prompts/prompt_builder.py** - プロンプト設計
- **役割**: ユーザー入力から Gemini API 用のプロンプトを構築
- **主要機能**:
  - system/user/assistant ロールの変換
  - 基底プロンプトの適用
  - 会話履歴の正規化
- **動作**: 会話履歴を Gemini API の形式に変換

#### **app/services/chat_service.py** - ビジネスロジック
- **役割**: チャット処理の全体制御
- **主要機能**:
  - プロンプト構築
  - Gemini API 呼び出し
  - レスポンス整形
- **動作**: 各コンポーネントを組み合わせてチャット処理を実行

#### **app/utils/error_mapping.py** - エラーハンドリング
- **役割**: アプリケーション例外を HTTP 例外に変換
- **主要機能**:
  - エラーコードの分類
  - HTTP ステータスコードの割り当て
- **動作**: 内部エラーを適切な HTTP レスポンスに変換

---
```

3. 機能概要

3.1 アーキテクチャ概要

```
[ユーザー] → [FastAPI] → [ChatService] → [PromptBuilder] → [GeminiClient] → [Gemini API]
                ↓              ↓              ↓              ↓
          [Logger]      [ErrorHandler]  [Config]      [ResponseParser]
```

3.2 処理フロー

1. **リクエスト受信**: FastAPI が HTTP リクエストを受信
2. **バリデーション**: Pydantic でリクエスト形式を検証
3. **プロンプト構築**: 会話履歴を Gemini API 形式に変換
4. **API 呼び出し**: Gemini API にリクエスト送信
5. **レスポンス解析**: API レスポンスからテキストを抽出
6. **レスポンス整形**: 統一 JSON 形式でレスポンス生成
7. **ログ出力**: 処理結果を構造化ログで記録

3.3 エラーハンドリング

- **ネットワークエラー**: タイムアウト、接続失敗
- **API エラー**: Gemini API の制限、無効なリクエスト
- **バリデーションエラー**: 不正な入力形式
- **内部エラー**: 予期しない例外

---

4. 入出力仕様

4.1 入力(リクエスト)

エンドポイント: `POST /v1/chat`

リクエスト形式:

```json
```

```
{
  "messages": [
    {
      "role": "system",
      "content": "あなたは法律に詳しいアシスタントです。"
    },
    {
      "role": "user",
      "content": "行政手続法の趣旨を初心者向けに説明して。"
    }
  ],
  "max_output_tokens": 512,
  "temperature": 0.7
}
```

パラメータ詳細:

パラメータ	型	必須	デフォルト	範囲	説明
`messages`	`List[Message]`	✅	-	-	会話履歴
`max_output_tokens`	`int`	❌	1024	1–8192	最大出力トークン数
`temperature`	`float`	❌	0.7	0.0–2.0	創造性パラメータ

Message 構造:

```json
{
  "role": "system|user|assistant",
  "content": "メッセージ内容（文字列）"
}
```

- **system**: アシスタントの行動指針を定義
- **user**: ユーザーの質問や入力
- **assistant**: 過去の応答履歴

4.2 出力（レスポンス）

成功時:

```json
{
  "success": true,
  "data": {
    "assistant": {
      "text": "行政手続法は、行政が国民に対して行う処分や行政指導、そして国民からの申請について、その手続きを公正にし、透明性を高めることを目的とした法律です...",
      "reasoning": null
    },
    "usage": {
      "prompt_tokens": null,
      "completion_tokens": null,
      "total_tokens": null
    },
    "meta": {
      "model": "gemini-2.0-flash",
      "latency_ms": 3025
    }
  },
  "error": null
}
```

失敗時:

```json
{
  "success": false,
  "data": null,
  "error": {
    "code": "GEMINI_TIMEOUT",
    "message": "Upstream request timed out",
    "details": {
      "error": "Request timeout"
```

```
    }
  }
}
```

レスポンス項目詳細:

項目	型	説明
`success`	`boolean`	処理成功フラグ
`data.assistant.text`	`string`	生成された回答テキスト
`data.assistant.reasoning`	`string¥|null`	推論過程（将来拡張用）
`data.usage.prompt_tokens`	`int¥|null`	入力トークン数
`data.usage.completion_tokens`	`int¥|null`	出力トークン数
`data.meta.model`	`string`	使用した Gemini モデル名
`data.meta.latency_ms`	`int`	処理時間（ミリ秒）
`error.code`	`string`	エラーコード
`error.message`	`string`	エラーメッセージ
`error.details`	`object¥|null`	エラー詳細情報

---

5. 主要関数

5.1 ChatService.chat()

```python
async def chat(self, req: ChatRequest) -> SuccessData
```

- **目的**: チャット処理のメイン関数
- **入力**: `ChatRequest` – チャットリクエスト
- **出力**: `SuccessData` – 成功レスポンスデータ
- **処理内容**:
  1. プロンプト構築
  2. Gemini API 呼び出し
  3. レスポンス整形

````
### 5.2 PromptBuilder.build_prompt()
```python
@staticmethod
def build_prompt(messages: List[Message]) -> List[dict]
```
- **目的**: 会話履歴を Gemini API 形式に変換
- **入力**: `List[Message]` – 会話履歴
- **出力**: `List[dict]` – Gemini API 用 contents 形式
- **処理内容**:
  1. system メッセージを最初の user メッセージに統合
  2. assistant ロールを model ロールに変換
  3. Gemini API 形式に正規化

### 5.3 GeminiClient.generate()
```python
async def generate(self, contents: list[dict], max_tokens: int | None,
temperature: float | None) -> Tuple[str, Dict[str, Any]]
```
- **目的**: Gemini API との通信
- **入力**:
  - `contents` – Gemini API 用 contents 形式
  - `max_tokens` – 最大出力トークン数
  - `temperature` – 温度パラメータ
- **出力**: `(生成テキスト, 使用量情報)`
- **処理内容**:
  1. HTTP リクエスト送信
  2. レスポンス解析
  3. エラーハンドリング

### 5.4 to_http_exception()
```python
def to_http_exception(err: AppError) -> HTTPException
```
- **目的**: アプリケーション例外を HTTP 例外に変換
- **入力**: `AppError` – アプリケーション例外
- **出力**: `HTTPException` – HTTP 例外
````

```
- **処理内容**: エラーコードに応じて HTTP ステータスコードを決定

---

## 6. 依存関係

### 6.1 Python 依存関係(requirements.txt)
```
fastapi==0.115.0   # Web フレームワーク
uvicorn[standard]==0.30.6   # ASGI サーバー
pydantic==2.9.2   # データバリデーション
pydantic-settings==2.11.0   # 設定管理(Pydantic v2 対応)
httpx==0.27.2   # HTTP クライアント
python-dotenv==1.0.0   # 環境変数読み込み
structlog==24.1.0   # 構造化ログ(オプション)
```

### 6.2 外部サービス依存関係
- **Google Gemini API**: テキスト生成機能
  - エンドポイント:
`https://generativelanguage.googleapis.com/v1beta/models/{model}:generateConte
nt`
  - 認証: API キー(環境変数`GEMINI_API_KEY`)
  - モデル: `gemini-2.0-flash`(推奨)

### 6.3 システム要件
- **Python**: 3.8 以上(推奨: 3.12)
- **OS**: Windows, macOS, Linux
- **メモリ**: 512MB 以上
- **ネットワーク**: HTTPS 接続可能

---

## 7. 実行環境

### 7.1 開発環境セットアップ
```

````
#### **1. 仮想環境作成**
```bash
cd "WP2-1 対話理解モジュール"
python3 -m venv .venv
source .venv/bin/activate  # Windows: .venv¥Scripts¥activate
```

#### **2. 依存関係インストール**
```bash
pip install -r requirements.txt
```

#### **3. 環境変数設定**
```bash
cp env.example .env
# .env ファイルを編集して GEMINI_API_KEY を設定
```

#### **4. サーバー起動**
```bash
uvicorn app.main:app --host 0.0.0.0 --port 8081 --reload
```

### 7.2 本番環境デプロイ

#### **Docker 使用例**
```dockerfile
FROM python:3.12-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY app/ ./app/
COPY .env .
````
````

```
EXPOSE 8081
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8081"]
```

環境変数(本番)
```bash
GEMINI_API_KEY=your_production_api_key
GEMINI_MODEL=gemini-2.0-flash
REQUEST_TIMEOUT_SEC=30
CONNECT_TIMEOUT_SEC=10
LOG_LEVEL=WARNING
ENVIRONMENT=production
```

7.3 監視・ログ

ログ形式
```json
{
  "level": "INFO",
  "name": "app.services.chat_service",
  "message": "Chat processing completed successfully",
  "ts": "2025-10-23 10:24:29,385"
}
```

ヘルスチェック
- **エンドポイント**: `GET /health`
- **レスポンス**: サーバー状態とモデル情報

---

8. 結合時の注意点

8.1 他のモジュールとの連携

Emotion モジュール連携

```python
# 感情分析結果をプロンプトに追加
emotion_result = await emotion_service.analyze(user_input)
enhanced_prompt = prompt_builder.add_emotion_context(base_prompt,
emotion_result)
```

Legal モジュール連携

```python
# 法令検索結果をプロンプトに追加
legal_context = await legal_service.search_relevant_laws(user_input)
enhanced_prompt = prompt_builder.add_external_knowledge(base_prompt,
legal_context)
```

8.2 API 設計の注意点

統一レスポンス形式

- すべてのモジュールで同じ`ApiResponse`形式を使用
- `success`フラグで処理結果を明確に区別
- `error`オブジェクトでエラー情報を構造化

エラーハンドリング

- 各モジュールで独自のエラーコード体系を定義
- HTTP ステータスコードとの適切なマッピング
- エラー詳細情報の保持

8.3 パフォーマンス考慮事項

タイムアウト設定

- **接続タイムアウト**: 5 秒（推奨）
- **リクエストタイムアウト**: 20-30 秒（推奨）
- **Gemini API 制限**: レート制限に注意

````
#### **キャッシュ戦略**
```python
# 将来実装予定
@lru_cache(maxsize=1000)
def build_cached_prompt(messages_hash: str) -> List[dict]:
    # 同じプロンプトのキャッシュ
    pass
```

#### **並行処理**
- FastAPI の非同期処理を活用
- 複数リクエストの同時処理に対応
- リソース使用量の監視

### 8.4 セキュリティ考慮事項

#### **API キー管理**
- 環境変数での管理(`.env`ファイル)
- 本番環境での暗号化
- ログへの出力禁止

#### **入力検証**
- Pydantic による自動バリデーション
- SQL インジェクション対策
- XSS 対策

#### **レート制限**
```python
# 将来実装予定
from slowapi import Limiter

limiter = Limiter(key_func=get_remote_address)

@app.post("/v1/chat")
@limiter.limit("10/minute")  # 1 分間に 10 リクエスト
async def chat(request: Request, req: ChatRequest):
````
````

```
    pass
```

8.5 拡張性考慮事項

プラグインアーキテクチャ
```python
# 将来実装予定
class PromptEnhancer:
    def enhance(self, prompt: str, context: dict) -> str:
        pass

class EmotionEnhancer(PromptEnhancer):
    def enhance(self, prompt: str, context: dict) -> str:
        # 感情分析結果をプロンプトに追加
        pass
```

設定の外部化
- プロンプトテンプレートの外部ファイル化
- モデル設定の動的変更
- A/B テスト対応

---

9. トラブルシューティング

9.1 よくある問題と解決方法

問題 1: PydanticImportError
```
pydantic.errors.PydanticImportError: `BaseSettings` has been moved to the
`pydantic-settings` package
```
解決方法:
```bash
```

```
pip install pydantic-settings
```

問題 2: Gemini API 404 エラー

```
models/gemini-1.5-pro is not found for API version v1beta
```

解決方法:

- 利用可能なモデル名を確認
- `gemini-2.0-flash`を使用

問題 3: タイムアウトエラー

```
GEMINI_TIMEOUT: Upstream request timed out
```

解決方法:

- `REQUEST_TIMEOUT_SEC`を増加
- ネットワーク接続を確認

9.2 ログ分析

重要なログメッセージ

- `ChatService initialized` - サービス起動確認
- `GeminiClient initialized` - API 接続確認
- `Generated text length: X chars` - 応答生成確認
- `Chat processing completed successfully` - 処理完了確認

エラーログの例

```json
{
  "level": "ERROR",
  "name": "app.clients.gemini_client",
  "message": "Gemini API error response: 400",
  "ts": "2025-10-23 10:23:15,806"
}
```

```
---

## 10. 将来の拡張計画

### 10.1 短期計画(1-3ヶ月)
- **会話履歴の永続化**: データベース連携
- **レスポンスキャッシュ**: Redis連携
- **レート制限**: SlowAPI導入

### 10.2 中期計画(3-6ヶ月)
- **外部知識ベース連携**: 法令データベース
- **感情分析モジュール統合**: Emotion API連携
- **多言語対応**: 英語・中国語対応

### 10.3 長期計画(6ヶ月以上)
- **音声入力対応**: Whisper API連携
- **画像解析**: 文書画像の読み取り
- **機械学習最適化**: プロンプト自動調整

---

## 11. 参考資料

### 11.1 公式ドキュメント
- [FastAPI Documentation](https://fastapi.tiangolo.com/)
- [Google Gemini API Documentation](https://ai.google.dev/docs)
- [Pydantic Documentation](https://docs.pydantic.dev/)

### 11.2 関連プロジェクト
- WP1-1: 要件定義書
- WP1-2: ワークパッケージ
- WP1-3: システム構成設計

### 11.3 連絡先
- 開発者: [開発チーム連絡先]
```

```
- 技術サポート: [サポート連絡先]
- 緊急時: [緊急連絡先]

---

**文書バージョン**: 1.0
**最終更新日**: 2025 年 10 月 23 日
**次回更新予定**: 機能追加時
```