LangChain・LangGraph解説書 アウトライン

本ドキュメントは、LLM（大規模言語モデル）を用いたアプリケーション開発フレームワークである「LangChain」および「LangGraph」の解説書のアウトラインです。本書は「入門編」と「実施編」の2つのレベルに分かれています。

入門編

1. はじめに

- 本書の目的と対象読者
- LLMアプリケーション開発における課題
- LangChainとLangGraphが解決する領域

2. LangChainの基本概念と使い方

- LangChainとは何か
- LLMとChatModelsの基本的な扱い方
- プロンプトテンプレートの定義と設計
- 出力パーサーによる構造化データの抽出
- LCEL（LangChain Expression Language）による処理の連結
- メモリ（会話履歴の保持）の仕組み
- RAG（検索拡張生成）の基礎（Vector StoresとRetrievers）

3. LangGraphの基本概念と使い方

- LangGraphとは何か
- なぜLangGraphが必要なのか（循環グラフと状態管理の必要性）
- 状態（State）の定義とデータのマージ方法
- ノード（Nodes）とエッジ（Edges）による処理フローの構築
- 条件付きエッジ（Conditional Edges）による動的な分岐
- グラフのコンパイルと実行フロー

4. LangChainとLangGraphの使い分けと連携

- LangChain単体とLangGraphの選定基準
- 既存のLangChainコンポーネントをLangGraphのノード内で再利用する方法

実施編

5. LangGraphによる高度なエージェント構築

- ReActエージェント（推論と行動のループ）の作成
- Human-in-the-loop（人の承認や介入）の組み込み
- 状態の永続化（Persistence）とチェックポインタの仕組み
- タイムトラベル（過去の状態への巻き戻しと再実行）
- マルチエージェント（連携・監督パターン）の設計

6. 実践的な開発パターンとユースケース

- カスタマーサポート用のチャットボット開発
- 自己修正型RAG（検索結果や生成結果を自己評価して修正するループ）
- コードの自動生成・実行・修正ループ

7. 本番運用とデバッグ

- LangSmithを用いた実行の追跡（トレーシング）と評価
- エラーハンドリングとリトライ処理の設計
- ストリーミング出力によるユーザー体験の向上

8. おわりに

- まとめと今後の展望
- 参考リソースおよび公式ドキュメントの紹介

第1章 はじめに

1.1 本書の目的と対象読者

本書は、ChatGPTなどの大規模言語モデル（LLM）を使って、より実用的で複雑なシステムを作りたいと考えているエンジニアに向けた入門書です。これまでにPythonを使ったプログラミングの経験があり、AIエージェントの仕組みを体系的に学びたい方を対象としています。AIに関する専門的な数学の知識や、難しい統計学の知識は必要ありません。

現在のAI開発では、単にAIとチャットをするだけでなく、社内のデータベースから情報を探させたり、特定の業務ルールに従って自動で作業を進めさせたりするシステム（AIエージェント）の需要が高まっています。しかし、このようなシステムを作るためには、プログラムからAIを正しく制御する方法を学ぶ必要があります。

本書では、AIアプリケーション開発のデファクトスタンダード（業界の標準）となっている「LangChain」と、より複雑な動作を制御するための「LangGraph」という2つのライブラリの使い方をやさしく解説します。本書を読み終えることで、自分で考え、ツールを使いこなしながら仕事を進める賢いAIアシスタントを開発できるようになることを目指します。さらに、本書では単にコードの書き方を説明するだけでなく、「なぜその構成にする必要があるのか」という設計の考え方についても丁寧に説明します。これにより、技術の流行が変わっても応用できる、息の長い開発の基礎力を身につけることができます。

1.2 LLMアプリケーション開発における課題

LLMは非常に優秀ですが、一般的なシステム開発で使われるプログラムとは異なる特徴を持っています。最も大きな違いは、LLMの出力が「確率」に基づいて決定されるため、毎回同じ入力をしても常に同じ出力が得られるとは限らないという点です。このため、従来のプログラムのように「もしこうなら、こう処理する」という厳密なルールだけで制御しようとすると、思い通りの動きをしてくれないことがあります。

また、LLM単体では「過去の会話の内容を覚えておくこと」や「現在のリアルタイムな情報を知ること」ができません。例えば、昨日の天気をLLMに尋ねても、学習データに含まれていない最近の情報には答えることができません。さらに、社内のデータベースにある顧客情報や売上データなどを参照して計算することも、LLMだけでは不可能です。

これらの課題を解決するためには、プログラム側で会話の履歴を管理し、必要に応じてインターネットや社内システムから情報を検索してLLMに手渡す仕組み（RAG：検索拡張生成）を作る必要があります。しかし、これらの仕組みを一からすべて手作業で実装しようとすると、プログラムのコードが複雑になり、開発やメンテナンスが非常に難しくなってしまいます。さらに、エラーが発生したときの処理や、AIの返答が不適切だった場合の軌道修正など、実際の運用で考慮すべき点も多岐にわたります。これらの複雑な周辺機能の実装負荷を減らし、本来の業務ロジックの開発に集中できるようにするための仕組みが強く求められています。

1.3 LangChainとLangGraphが解決する領域

こうしたLLMアプリ開発の課題を解決するために作られたのが、LangChainとLangGraphです。LangChainは、LLMと外部システムを繋ぐための「部品」を豊富に提供するフレームワークです。例えば、LLMに質問する前のプロンプトを整理する部品、LLMからの回答をプログラムで扱いやすいJSON形式などに変換する部品、ベクトルデータベースから必要な情報を検索する部品などが用意されています。これらの部品を「LCEL（ランチェーン・エクスプレッション・ランゲージ）」という共通の記述方法で繋ぎ合わせることで、データの流れをきれいに整理したプログラムを素早く作ることができます。

しかし、LangChain単体では、処理が一方通行になりがちであるという弱点がありました。現実の業務では、「AIに作業を依頼し、その結果にエラーがあれば修正させて、もう一度実行する」というような、何度もぐるぐる回るループ処理が多く存在します。この循環する複雑な処理の流れと、その過程で変化するデータを管理するために登場したのがLangGraphです。

LangGraphを使うことで、処理の分岐やループを「グラフ構造（点と線で表されるネットワーク）」として綺麗に定義でき、AIが自律的に状況を判断して動くシステム（AIエージェント）を安全に開発できるようになります。これにより、開発者は複雑なエラーハンドリングや状態遷移の管理から解放され、AIにどのような仕事を実行させるかという肝心なロジックの開発に集中できるようになります。

第2章 LangChainの基本概念と使い方

2.1 LangChainとは何か

LangChainは、LLM（大規模言語モデル）のパワーを最大限に引き出し、実用的なアプリケーションを効率的に開発するためのオープンソースのフレームワークです。LLMとシステムを接続するための様々なツールや共通のルールを提供しており、PythonやTypeScriptなどの言語で利用することができます。これを使うことで、開発者は低レベルなAPI呼び出しの手間を減らし、アプリケーションのロジック構築に集中できます。

通常、LLMを用いたシステムを構築する場合、APIキーの管理、プロンプトの送受信、応答の解析など、多くの定型的な処理を書く必要があります。LangChainはこれらの基本的な処理をカプセル化（まとめること）し、統一された方法で扱えるようにします。さらに、様々なAIベンダー（OpenAI、Anthropic、Googleなど）の異なるモデルを、コードを大きく変えることなく切り替えて使えるように抽象化する役割も持っています。

また、外部のドキュメント検索システムや電卓などの各種ツールとLLMを結びつけるためのコネクターとしての役割も担っています。これにより、単に言葉を返すだけのチャットボットから、外部データを参照して業務をこなす高度なシステムまで、一貫したアプローチで開発を進めることが可能になります。LangChainは、変化の激しいAI開発において、土台となる強力なフレームワークとして位置づけられています。

2.2 LLMとChatModelsの基本的な扱い方

LangChainでは、LLMを操作するためのインターフェースとして、大きく分けて「LLMs」と「ChatModels」の2つの仕組みを提供しています。「LLMs」は、純粋なテキストを入力として受け取り、その後に続くテキストを予測して出力する古いタイプのインターフェースです。一方、「ChatModels」は、チャットのようなメッセージのやり取り（会話履歴）を入力として受け取り、AIの返答メッセージを出力する現代的なインターフェースです。

現在、多くの主要なAIモデルは会話形式のデータで追加学習されているため、LangChainでもChatModelsの利用が強く推奨されています。ChatModelsでは、発言者を区別するためにいくつかのメッセージタイプを使い分けます。ユーザーからの入力を表す「HumanMessage」、AIからの返答を表す「AIMessage」、そしてAIの振る舞いを指定する前提ルールを設定する「SystemMessage」などがあります。

これらをリスト形式でモデルに渡すことで、AIはこれまでの文脈を正しく理解し、指定された役割に沿った適切な回答を返すことができるようになります。プログラムからこれらのメッセージを生成・送受信する記述方法は統一されているため、裏側で動作するAIモデルをGPT-4からClaude 3などに切り替える際も、メッセージの扱い方を変える必要がありません。この高い柔軟性が、LangChainが提供するインターフェースの大きな強みです。

2.3 プロンプトテンプレートの定義と設計

LLMから期待する回答を得るためには、AIに対する命令文である「プロンプト」を工夫する必要があります。しかし、ユーザーの入力内容ごとにプロンプトを最初から手作業で組み立てるのは非効率的です。そこでLangChainは、プロンプト

の一部を変数として定義し、実行時に動的に値を差し込める「プロンプトテンプレート」という機能を提供しています。

プロンプトテンプレートを使用すると、例えば「以下の文章を{target_language}に翻訳してください：{text}」といった雛形を作ることができます。プログラムの実行時に、`target_language`に「日本語」、`text`に翻訳したい英文を代入することで、完成したプロンプトが自動生成されてLLMに送られます。これにより、プロンプトの再利用性が高まり、コードと命令文の管理が非常に容易になります。また、システムとしての指示（SystemMessage）と、ユーザーの個別入力（HumanMessage）を分離して綺麗にテンプレート化することも容易です。

プロンプトテンプレートは、単なる文字列の組み立てにとどまらず、少数の回答例をプロンプト内に含める「フューショット・プロンプティング」のテンプレート化など、高度なプロンプト設計の手法もサポートしています。これにより、LLMの回答精度を向上させるためのプロンプト調整が簡単に行えるようになります。

2.4 出力パーサーによる構造化データの抽出

LLMは通常、自然言語（話し言葉）で回答を返します。しかし、システム開発においては、AIの回答をそのまま画面に表示するだけでなく、回答の中から「日付」や「金額」などの特定のデータを取り出してデータベースに保存したり、次のプログラムの処理に回したりしたい場合が多々あります。LLMの応答テキストから必要な情報を取り出して、プログラムで扱いやすい形式（JSONやオブジェクトなど）に変換する役割を持つのが「出力パーサー（Output Parser）」です。

出力パーサーを使用すると、まずLLMに対して「回答は以下のJSON形式で出力してください」という指示（スキーマ情報）をプロンプトに自動的に付与します。そして、LLMから返ってきたテキストを自動で解析し、Pythonの辞書型（dict）やPydanticのモデルクラスに変換します。もしLLMが指定された形式を崩して出力してしまった場合には、自動的に「形式が違います。修正して出力してください」とLLMに再送して修正を促すパーサー（RetryParser）なども用意されています。

出力パーサーを組み込むことで、不確実性の高いAIの出力を、信頼性の高い構造化データとしてシステムに組み込むことができるようになります。これは、AIをWebサービスのAPIサーバーやバックエンド処理と連携させる上で欠かせない重要なステップです。

2.5 LCEL（LangChain Expression Language）による処理の連結

LangChainで最も特徴的な機能の一つが、LCEL（LangChain Expression Language）と呼ばれる独自の記述ルールです。これは、プロンプトテンプレート、ChatModel、出力パーサーなど、これまで紹介したさまざまなコンポーネントを一本の処理の鎖（チェーン）として繋ぎ合わせるための書き方です。Unixコマンドのように`|`（パイプ）記号を使って記述します。

例えば、`chain = prompt | model | parser`のように書くことで、プロンプトに値を入力し、それをモデルに渡し、返ってきたテキストをパーサーで解析する、という一連の流れを一行で表現できます。LCELで書かれたチェーンは、自動的にストリーミング出力（文字を少しずつ表示する処理）や、非同期処理（他の処理を止めずに実行する処理）、並列処理に対応するという特徴を持っています。

さらに、エラーが起きた際のリトライ処理や、特定のモデルが使えない場合のバックアップモデルへの切り替えといった複雑なロジックも、パイプラインの一部として簡潔に書き加えることができます。このように、複雑になりがちなAIアプリケ

ーションのコードを、誰が見ても分かりやすく、かつ高機能に保つことができるのがLCELのメリットです。

2.6 メモリ（会話履歴の保持）の仕組み

LLMとのやり取りは、基本的には「一問一答」の形式であり、前後の関係を保持していません。しかし、一般的なチャットアプリのように「さっき言ったことを踏まえて回答してほしい」という場合には、過去の会話履歴をモデルに渡し続ける必要があります。この会話履歴の記憶と管理を自動で行うのが「メモリ（Memory）」の仕組みです。

LangChainでは、会話履歴を一時的に保持する単純なメモリから、データベースに永続化（保存）するメモリまで、用途に合わせたさまざまなメモリコンポーネントが用意されています。最も基本的なメモリは、これまでのメッセージ履歴をすべて単純に蓄積し、次のリクエストの際にプロンプトに含める方式です。しかし、会話が長くなるとメッセージ全体の量が膨大になり、APIの利用料金が高くなったり、LLMが処理できる文字数の限界を超えてしまったりします。

そこで、古い会話を自動で要約してコンパクトにするメモリや、最新の数往復分だけを保持するメモリなどの高度な管理手法も提供されています。メモリを適切に設計することで、ユーザーにとってストレスのない、文脈を捉えた自然な対話システムを作ることができます。これはチャットボットだけでなく、長期にわたるタスクを実行するAIエージェントの構築においても極めて重要です。

2.7 RAG（検索拡張生成）の基礎

LLMは学習した時点の古い情報しか持っておらず、公開されていない社内の機密データなどを知ることはできません。この問題を解決するために、外部のデータベースから関連する情報を検索し、その情報をプロンプトに含めてLLMに質問する手法を「RAG（Retrieval-Augmented Generation：検索拡張生成）」と呼びます。LangChainは、RAGに必要な処理フローを簡単に実装するための部品をすべて備えています。

RAGの一般的な流れは、まずPDFやテキストファイルなどの文書を読み込み、扱いやすいサイズに分割（ドキュメントスプリット）します。次に、それぞれのテキストの意味（概念）を数値に変換（エンベディング）し、ベクトルデータベースと呼ばれる特殊なデータベースに保存（Vector Store）します。ユーザーが質問をすると、システムはその質問の意味と近い内容のテキストをデータベースから検索（Retrieval）して探し出します。

最後に、見つかったドキュメントの内容を「参考資料」としてプロンプトに貼り付け、LLMに回答を生成させます。LangChainを使うことで、これらの複雑な処理パイプラインを数行のコードで繋ぐことができ、自社のデータに特化した高精度な回答を生成するシステムを容易に構築できます。RAGは、現在のエンタープライズ（企業向け）AI開発において、最も広く使われている技術の一つです。

第3章 LangGraphの基本概念と使い方

3.1 LangGraphとは何か

LangGraphは、LangChainの機能をベースにして、複数のステップからなる複雑な処理の流れを構築・管理するためのライブラリです。特に、データや処理の流れが一方通行ではなく、条件によって分岐したり、同じ処理を何度も繰り返したりする「循環（ループ）構造」を持つプログラムを作るのに適しています。これを利用することで、AI自身が状況を判断して行動を選択する「AIエージェント」を構築できます。

一般的なアプリケーション開発において、複雑な状態の遷移やループを管理するのは非常に骨の折れる作業です。LangGraphでは、これらを「グラフ」という数学の概念（点と線で表される図）に当てはめて整理します。グラフを構成するのは、具体的な処理を行う「ノード（点）」と、処理の進む方向を示す「エッジ（線）」です。

開発者は、どのノードがどのエッジで繋がっているかを定義することで、ビジュアル的にも理解しやすいフローをプログラムとして実装することができます。また、LangGraphは並列処理や状態の保存（チェックポイント）といった、本番運用に耐えうる高度な機能を標準で備えています。これにより、ユーザーからの指示に対して単に回答するだけでなく、自ら計画を立ててタスクを実行する自律型エージェントの作成が身近なものとなります。

3.2 なぜLangGraphが必要なのか

従来のLangChainでも、処理を連結して実行する仕組み（チェーン）や、AIがツールを選んで実行する簡単な「エージェント」機能は提供されていました。しかし、従来の仕組みでは、処理の進行が一方向（DAG：有向非巡回グラフ）に制限されており、エラーが起きた場合に前のステップに戻ってやり直すといった、柔軟なループ構造を綺麗に書くことが困難でした。

また、複雑なエージェントを構築する際には、「今、エージェントは何を知っていて、どこまで作業が進んだのか」という現在の「状態（State）」を、ステップ間で破綻させずに正確に引き継いでいく必要があります。従来のコードでこれを実装しようとすると、グローバル変数を使ったり、複雑な引数のやり取りが発生したりして、プログラムの安全性が著しく低下していました。

LangGraphは、グラフ全体で一つの「共有状態（State）」を持ち、各ノードがそれを安全に読み書きする仕組みを導入することで、これらの問題を完全にクリアしました。これにより、「AIがコードを生成し、それをテスト環境で動かしてみて、エラーが出たらコードを自己修正して再テストする」といった、ループを伴う洗練された自律システムを、バグが混入しにくい安全なコードで実装できるようになりました。

3.3 状態（State）の定義とデータのマージ方法

LangGraphの中心となる概念が「状態（State）」です。状態とは、グラフの中を流れるデータの実体であり、実行中の全ノードからアクセスできる共有のメモリ（変数）のようなものです。通常、Pythonのクラスや辞書（dict）として定義され、会話の履歴、現在のタスクリスト、一時的な作業結果などの情報がここに格納されます。

ノードが処理を実行すると、新しいデータを返します。LangGraphは、その返されたデータを自動的に既存の状態（State）にマージ（合流・上書き）します。このマージ方法については、開発者が細かくルールを指定できます。例えば、通常の変数のようには単に「上書き」する一方で、会話履歴（メッセージのリスト）については、古い履歴を消さずに新しいメッセージを末尾に「追加（Append）」していく、といった挙動を簡単に指定できます。

このようにデータごとの更新ルール（Reducer）をあらかじめ状態の定義に組み込んでおくことで、各ノードの処理コード内では複雑なデータ合成のロジックを書く必要がなくなります。各ノードは「状態を受け取って、自分が処理した差分だけを返す」というシンプルな実装になり、コードの可読性と保守性が大幅に向上します。

3.4 ノード（Nodes）とエッジ（Edges）による処理フローの構築

LangGraphでワークフローを作成する際は、処理の要素を「ノード」と「エッジ」に分解して登録していきます。「ノード（Node）」は、具体的に実行されるプログラムの関数やステップです。例えば、「ユーザーの入力を受け取って検索キーワードを作る関数」や「データベースを検索する関数」がノードになります。これらのノードは、現在の状態（State）を引数として受け取り、更新された状態を返します。

「エッジ（Edge）」は、ノードから次のノードへの「道筋（矢印）」です。LangGraphには、無条件で次の処理へ進む通常の「エッジ」と、グラフの開始点を示す「STARTエッジ」、終了点を示す「ENDエッジ」があります。これらを組み立てることで、「START -> ノードA -> ノードB -> END」といった全体の流れをプログラムコードとして宣言的に構築していきます。

このように処理の流れをノードとエッジに分離して定義することで、各処理の依存関係が明確になります。また、処理全体の構造を可視化（画像として出力する機能もLangGraphには標準で備わっています）しやすくなり、チーム開発において設計を共有したりレビューしたりする際にも大きな威力を発揮します。

3.5 条件付きエッジ（Conditional Edges）による動的な分岐

AIエージェントの真骨頂は、状況に応じて次の行動を自律的に判断できる点にあります。LangGraphでこの動的な分岐を実現するのが「条件付きエッジ（Conditional Edges）」です。これは、あるノードの処理終わった後、次にどのノードへ進むかを、プログラムの条件式やLLMの判断によって切り替える仕組みです。

条件付きエッジを設定するには、遷移先を決定するための「判定関数」を用意します。この判定関数は、現在の状態（State）を読み込み、次に進むべきノードの名前（文字列）を返します。例えば、LLMが「Web検索が必要」と判断した場合は「search_node」へ進み、「回答が可能」と判断した場合は「respond_node」へ進む、といった分岐を作ることができます。

これにより、一方通行の固定されたフローではなく、「LLMの出力結果によって、外部ツールの利用を繰り返すか、あるいはユーザーへの回答に移るか」を自動でループ判断する複雑な仕組み（ReActパターンなど）を実装することが可能になります。条件付きエッジは、エージェントを自律的に動かすための最も重要な部品です。

3.6 グラフのコンパイルと実行フロー

ノード、エッジ、条件付きエッジの登録が完了したら、グラフをアプリケーションとして動作させるための「コンパイル（Compile）」という手順を踏みます。コンパイルを行うことで、作成したグラフ構造がLangChainの標準的な「Runnable」オブジェクトへと変換されます。これにより、グラフ全体がLangChainの他の部品と同じように `invoke` や `stream` などのメソッドで呼び出せるようになります。

コンパイルされたグラフに初期入力（Stateの初期値）を与えて実行すると、LangGraphは自動的にSTARTから順にエッジを辿ってノードを実行していきます。実行中、各ノードが状態（State）をどのように更新していったかの履歴はすべて記録されており、実行が終了すると、すべてのノードの処理が完了した後の最終的な状態が返されます。

このコンパイルステップがあることで、グラフを外部のWeb API（FastAPIなど）のエンドポイントに直接組み込むことが非常に容易になります。コンパイルは、静的に定義した処理の流れを、動的に動くシステムへと昇華させるための最終工程と言えます。

第4章 LangChainとLangGraphの使い分けと連携

4.1 LangChain単体とLangGraphの選定基準

新しいAIプロジェクトを立ち上げる際、LangChain単体で開発すべきか、それともLangGraphを導入すべきかは重要な判断基準となります。基本的には、開発したいシステムの複雑さと、処理の流れに「ループ（繰り返しや軌道修正）」が存在するかどうかで判断します。

ユーザーからの入力に対して、いくつかの情報を検索（RAG）し、プロンプトを組み立ててLLMに渡し、その回答を画面に表示する、といった一方通行の処理であれば、LangChain単体が最適です。LCELを用いたシンプルな構成にすることで、コード量も少なく済み、動作のデバッグも容易になります。

一方で、以下のような場合にはLangGraphの導入を強く推奨します。まず、エラーが発生した際に「何がダメだったのか」をAI自身に評価させ、プログラミングコードやクエリを修正して再実行させるようなループ処理がある場合です。また、ユーザーが途中で処理に介入し、中身を確認してから「実行を許可する」といったインタラクティブな動き（Human-in-the-loop）が必要な場合や、複数のAIが役割分担して共同でタスクを進める場合も、LangGraphの強力な状態管理が必須となります。

プロジェクトの初期段階ではLangChain単体で試作（プロトタイピング）を行い、システムの機能要件が複雑化して状態の管理やループ制御が必要になった段階で、LangGraphへとスムーズに移行していくというアプローチが賢明です。

4.2 既存のLangChainコンポーネントをLangGraphのノード内で再利用する方法

LangGraphはLangChainの上に構築されているため、両者は非常に高い親和性を持っています。これは、これまでLangChainで構築した既存のコードやコンポーネントを、捨てることなくそのままLangGraphのパーツとして組み込めることを意味します。

具体的には、LangChainで定義した「LLMとプロンプトをLCELで繋いだチェーン（`prompt | llm`）」や、検索エンジンや計算機などの「ツール（Tools）」は、LangGraphの「ノード」の関数内部で直接呼び出して実行できます。ノードの役割は「現在の状態を受け取って、更新された状態を返す」という非常にシンプルなインターフェースになっているため、関数の内部で既存のLangChainの `invoke` メソッドを実行し、その出力を状態に格納するだけで連携が完了します。

これにより、ライブラリの移行コストを最小限に抑えながら、既存の単純なチェーンを、自律してループ動作する高度なエージェントへと段階的に拡張していくことができます。LangChainで部品を作り、LangGraphでそれらを束ねて複雑な動きを作る、というのが現在のLLMアプリケーション開発におけるベストプラクティスです。

第5章 LangGraphによる高度なエージェント構築

5.1 ReActエージェント（推論と行動のループ）の作成

ReAct（Reasoning and Acting）は、AIが「推論（考えること）」と「行動（ツールを動かすこと）」を交互に繰り返しながら目標を達成する強力な設計パターンです。LLMに単に回答を考えさせるだけでなく、必要に応じて外部の検索エンジンや電卓などの「ツール」を実行し、その結果をもとにさらに考えを進めるというループを作成します。LangGraphでは、このReActエージェントを非常に直感的かつ安全に実装できます。

実装する際は、まずLLMが「次にどのツールを使うべきか（あるいはユーザーに最終回答を返すか）」を判断する思考ノードを作成します。次に、LLMが選んだツールを実際に動かす実行ノードを作成します。思考ノードから実行ノードへは、LLMの判断結果によって進路が変わる「条件付きエッジ」で接続します。ツールが実行されたら、その実行結果を状態（State）に保存した上で、再び思考ノードへと戻るループ（エッジ）を張ります。

以下は、LangGraphを用いた基本的なReActエージェントの実装コードです。

```python
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode

# 1. 状態（State）の定義
# add_messagesを指定することで、古いメッセージを消さずに末尾に追加します
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]

# 2. ツールとLLMの準備
model = ChatOpenAI(model="gpt-4o")
tools = [search_web]  # search_webは事前に定義された外部ツール
model_with_tools = model.bind_tools(tools)

# 3. ノード（処理を行う関数）の定義
def call_model(state: AgentState):
    # LLMを呼び出し、結果をメッセージリストに加えます
    response = model_with_tools.invoke(state["messages"])
    return {"messages": [response]}

# 4. グラフの構築
workflow = StateGraph(AgentState)

# ノードを登録
workflow.add_node("agent", call_model)
workflow.add_node("action", ToolNode(tools))  # ツールの実行を行う標準ノード

# エッジを定義
workflow.add_edge(START, "agent")

# 条件付きエッジで分岐を定義する判定関数
def should_continue(state: AgentState):
    last_message = state["messages"][-1]
    # LLMがツール呼び出しを求めている場合はツール実行ノードへ進む
    if last_message.tool_calls:
        return "action"
    # そうでなければ処理を終了する
    return END
```

```
workflow.add_conditional_edges("agent", should_continue)
workflow.add_edge("action", "agent")  # ツール実行後は再びモデル呼び出しに戻る

# コンパイルして実行可能なオブジェクトにする
app = workflow.compile()
```

このアプローチにより、開発者が事前にすべての手順をプログラムしなくても、AIが自分で考えて最適な手順で問題を解決する柔軟なシステムが構築できます。

5.2 Human-in-the-loop（人の承認や介入）の組み込み

AIエージェントが自律的に動くことは便利ですが、企業の重要なデータを書き換える処理や、外部へメールを送信する処理など、リスクの高い操作をAIだけに任せるのは危険です。そこで、処理の途中で一度プログラムを一時停止し、人間の承認や修正を得てから処理を再開させる設計を「Human-in-the-loop（ヒューマン・イン・ザ・ループ）」と呼びます。LangGraphは、この人間との連携処理を最初からフレームワークの機能としてサポートしています。

具体的には、特定のノード（例えば「メール送信ノード」）を実行する直前で、グラフの処理を自動的に「一時停止（Interrupt）」する設定を追加できます。一時停止したグラフは、その時点までの状態（State）を保持したまま待機状態になります。人間は画面上で「AIが作成した送信メールの下書き」を確認し、問題がなければ「承認」ボタンを押してグラフを再開させます。もし内容に誤りがあれば、人間がテキストを修正して状態（State）を書き換えた上で再開させることも可能です。

以下は、ツールノード実行の直前に一時停止を設定し、承認を挟んで再開させるコード例です。

```
# グラフをコンパイルする際、特定のノードの直前で一時停止する設定を行います
app = workflow.compile(
    interrupt_before=["action"]  # ツールを実行する前に一時停止して人間の承認を待つ
)

# 実行時
config = {"configurable": {"thread_id": "human-check-123"}}
events = app.stream({"messages": [("user", "メール送信のツールを動かして")]}, config)

# 実行すると、actionノードの手前で処理が自動的に一時停止します
for event in events:
    print(event)

# 一時停止中、人間が内容を確認して「OK」と判断したら、Noneを渡してそのまま処理を再開させます
app.invoke(None, config)
```

これにより、AIの自律性と人間による管理（セキュリティや品質管理）を両立させることができます。重大な失敗を防ぐためのこの安全弁のような機能は、実用的なエンタープライズアプリケーションを運用する上で極めて重要です。

5.3 状態の永続化（Persistence）とチェックポインタの仕組み

複雑な業務プロセスを動かすエージェントは、処理が数分から数日間に及ぶことがあります。また、処理の途中でサーバーが再起動したりエラーが発生したりした際に、最初からやり直すのは大きな時間とコストのロスになります。LangGraphでは、処理の各ステップが実行されるたびに、その時点の状態（State）をデータベースに自動で保存する「永続化（Persistence）」と「チェックポインタ」の仕組みが備わっています。

チェックポインタを有効にすると、ノードが一つ実行し終わるたびに、現在の状態のスナップショット（その瞬間の記録）が保存されます。これにより、プログラムが予期せず途中で停止してしまっても、最後に保存されたチェックポイントから何事もなかったかのように処理を再開（レジューム）できます。また、スナップショットは「スレッドID（会話やタスクごとの識別子）」に関連付けて管理されるため、何千人ものユーザーが同時に異なる会話を行っていても、それぞれの状態が混ざることなく正確に記憶されます。

以下は、メモリ上に状態を保存するチェックポインタを使用して、会話の文脈を記憶するコード例です。

```
from langgraph.checkpoint.memory import MemorySaver

# メモリに状態を保存するチェックポインタを準備（本番ではデータベースを指定可能）
memory = MemorySaver()

# コンパイル時にチェックポインタを登録
app = workflow.compile(checkpointer=memory)

# 会話ごとに一意のID（スレッドID）を設定して実行
config = {"configurable": {"thread_id": "user-session-123"}}

# 初回の質問
app.invoke({"messages": [("user", "こんにちは、私の名前は太郎です。")]}, config)

# 次の質問（同じスレッドIDを使うことで、前回の会話「太郎」を自動で覚えています）
response = app.invoke({"messages": [("user", "私の名前が分かりますか？")]}, config)
print(response["messages"][-1].content)  # 「はい、太郎さんですね」と答えます
```

さらに、この仕組みは次に説明する「タイムトラベル」機能の基盤にもなっており、信頼性が高く堅牢なシステムを開発するための要となっています。

5.4 タイムトラベル（過去の状態への巻き戻しと再実行）

LangGraphのチェックポインタによる状態の永続化を応用した非常にユニークな機能が「タイムトラベル（Time Travel）」です。これは、実行中のエージェントの歴史（過去の特定のステップ）に遡って、その時点の状態を確認したり、そこから異なる指示を与えて処理をやり直したりできる機能です。

開発や運用の現場において、「エージェントが10ステップ目で誤ったツールを選択してしまい、結果がおかしくなった」というようなバグに直面することがあります。従来のシステムでは、原因究明のために1ステップ目から再現を実行する必要がありましたが、タイムトラベルを使えば、問題の起きた9ステップ目の状態へ一瞬で巻き戻すことができます。そして、その時点のプロンプトを修正し、そこから別ルートで処理を再スタートさせることが可能です。

以下は、スレッドの過去の状態履歴を取得し、過去の時点から別の入力を与えて処理を分岐させるコード例です。

```
# 1. これまでのすべての実行履歴（チェックポイント）を取得する
history = list(app.get_state_history(config))

# 2. 過去の特定のステップ（例: 2つ前の状態）のIDを取得
checkpoint_id = history[-2].config["configurable"]["checkpoint_id"]

# 3. 過去の時点の構成オブジェクトを作成
historical_config = {
    "configurable": {
        "thread_id": "user-session-123",
        "checkpoint_id": checkpoint_id
    }
}

# 4. 過去の時点（歴史）から別の指示を与えて、新しい世界線で処理を再スタート
app.invoke(
    {"messages": [("user", "今の指示は忘れて、別の話題に変えます。")]},
    historical_config
)
```

また、本番環境のユーザーにとっても、「数ステップ前に戻って会話をやり直す」という便利な操作を提供できるようになります。このデバッグとユーザー体験の両面における強力な柔軟性は、状態の履歴をすべて記録しているLangGraphならではの大きな特徴です。

5.5 マルチエージェント（連携・監督パターン）の設計

一つの巨大なプロンプトや単一のLLMエージェントにすべての業務を任せようとすると、エージェントの「頭脳」が混乱し、指示を忘れたり誤った判断をしたりしやすくなります。そこで、特定の専門分野に特化した複数の小さなエージェントを作り、それらを連携させて大きなタスクを解決させる設計が「マルチエージェント」システムです。

LangGraphは、このマルチエージェントを構築するのに最適なアーキテクチャを持っています。主な設計パターンとして、エージェント同士が対等に話し合いながら作業を渡していく「コラボレーションパターン」や、全体の進行役（スーパーバイザー）となるエージェントが、他の作業者エージェントに仕事を割り振って進捗を管理する「監督（スーパーバイザー）パターン」があります。各エージェントはそれぞれ独自のノードとして定義され、自分が必要な処理を終えると、マージされた状態（State）を介して次のエージェントにバトンを渡します。

以下は、スーパーバイザー（監督者）を配置し、次に実行すべき専門エージェントを振り分ける設計のコード例です。

```python
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

# マルチエージェント用の状態の定義
class RouterState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]
    next_agent: str  # 次に呼び出すエージェントの名前

# 監督者ノードの定義（LLMが次に誰が動くべきかを判断する）
def supervisor(state: RouterState):
    # LLMに会話の履歴を渡し、「研究者」か「ライター」のどちらを動かすか、または終了（FINISH）かを決定さ
せる
    response = supervisor_chain.invoke(state["messages"])
    return {"next_agent": response.next_agent}

# 専門エージェントノードの定義
def researcher_agent(state: RouterState):
    response = researcher_chain.invoke(state["messages"])
    return {"messages": [response]}

def writer_agent(state: RouterState):
    response = writer_chain.invoke(state["messages"])
    return {"messages": [response]}

# グラフの定義
workflow = StateGraph(RouterState)
workflow.add_node("supervisor", supervisor)
workflow.add_node("researcher", researcher_agent)
workflow.add_node("writer", writer_agent)

workflow.add_edge(START, "supervisor")

# 監督者のnext_agentの指示に基づいて遷移先を決定する条件付きエッジ
workflow.add_conditional_edges(
    "supervisor",
    lambda state: state["next_agent"],
    {
        "researcher": "researcher",
        "writer": "writer",
```

```
        "FINISH": END
    }
)

# 各専門エージェントは、作業が終わったら一度監督者に報告（戻る）する
workflow.add_edge("researcher", "supervisor")
workflow.add_edge("writer", "supervisor")

app = workflow.compile()
```

これにより、システムの役割分担が明確になり、各エージェントのプロンプトをシンプルに保つことができるため、全体の精度向上と開発の効率化（モジュール化）が同時に達成できます。

第6章 実践的な開発パターンとユースケース

6.1 カスタマーサポート用のチャットボット開発

実用的なAIアプリケーションの代表例が、問い合わせに対応するカスタマーサポートボットです。単に質問に答えるだけでなく、裏側で動作する顧客システムと安全に連携しながら、ユーザー個別の課題を解決するシステムをLangGraphで設計します。

このチャットボットは、まず顧客の「契約状態」や「過去の履歴」を状態（State）に読み込むことから始めます。ユーザーが「プランを解約したい」と言った場合、AIは社内の解約手続きに関するルール（RAGで検索）を参照し、必要な情報をユーザーから聞き取ります。途中でクレジットカード情報の登録変更などが発生する場合、前述の「Human-in-the-loop」を利用して、ユーザー自身が変更に同意したことを確認してから処理を実行するエッジを作ります。

以下は、サポートボットがユーザー情報の確認と解約などの手続きを安全に進めるための状態遷移コード例です。

```python
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

# カスタマーサポート用の状態定義
class SupportState(TypedDict):
    customer_id: str
    contract_status: str  # "active", "pending_cancel", "canceled" などの状況
    messages: Annotated[Sequence[BaseMessage], add_messages]
    billing_confirmed: bool  # ユーザー同意が得られたか

# 初期情報の取得ノード
def initialize_session(state: SupportState):
    # 顧客IDに基づいてデータベースから現在の契約状況を取得
    customer_info = get_customer_db(state["customer_id"])
    return {"contract_status": customer_info["status"]}

# 解約確認ノード
def request_cancel_confirmation(state: SupportState):
    # ユーザーに対し「本当に解約してもよろしいですか？」というメッセージを生成して返す
    message = ("user", "解約手続きを進めます。契約内容を確認し、同意する場合は承認してください。")
    return {"messages": [message], "billing_confirmed": False}

# 解約処理実行ノード（このノードの手前でHuman-in-the-loopによる一時停止を行います）
def execute_cancellation(state: SupportState):
    # データベースの契約状況を「キャンセル済み」に変更
    update_customer_db(state["customer_id"], "canceled")
    return {"contract_status": "canceled", "messages": [("assistant", "解約手続きが完了しました。")]}

# グラフの構築
workflow = StateGraph(SupportState)
workflow.add_node("initialize", initialize_session)
workflow.add_node("request_confirm", request_cancel_confirmation)
workflow.add_node("execute_cancel", execute_cancellation)

workflow.add_edge(START, "initialize")
workflow.add_edge("initialize", "request_confirm")

# 実行前に人間の確認（同意）を必要とするため、execute_cancelの直前で一時停止する
workflow.add_edge("request_confirm", "execute_cancel")
workflow.add_edge("execute_cancel", END)
```

```
# actionノードの前に一時停止（ユーザーまたは管理者の承認待ち）を挟んでコンパイル
app = workflow.compile(interrupt_before=["execute_cancel"])
```

このように、情報の検索、ユーザーへのヒアリング、安全な手続きの実行、といった一連のワークフローを状態遷移として定義することで、ロボット的ではない、顧客に寄り添った丁寧かつ確実なサポート自動化を実現できます。

6.2 自己修正型RAG（検索結果や生成結果を自己評価して修正するループ）

従来のRAGシステムは、検索したドキュメントの中に質問の答えが含まれていなかったり、LLMが検索内容を無視して誤った回答（ハルシネーション）を生成したりする問題がありました。これらを解決するために、検索と生成のプロセスを自己評価し、自動でリトライや修正を行う「自己修正型RAG（Corrective RAG）」をLangGraphのループ構造で構築します。

この仕組みでは、検索したドキュメントがユーザーの質問に対して「本当に役に立つか」を評価する評価ノードを設けます。もし役に立たないと判断された場合は、検索クエリ（キーワード）をAI自身に書き換えさせ、Web検索などを通じて不足している情報を再取得します。また、生成された回答が「元データに基づいて正しく書かれているか」を検証するノードも作成します。矛盾や事実誤認があれば、生成ノードに戻って書き直させます。

以下は、自己修正型RAGを定義するグラフコード例です。

```python
from typing import List, TypedDict
from langgraph.graph import StateGraph, START, END

class RAGState(TypedDict):
    question: str
    documents: List[str]  # 検索された文書
    generation: str      # 生成された回答
    search_retry_count: int  # 検索クエリ書き換えの再試行回数

# 1. 外部データベースから情報を検索するノード
def retrieve(state: RAGState):
    docs = vector_store.similarity_search(state["question"])
    return {"documents": docs}

# 2. 検索したドキュメントの適合性を評価するノード
def grade_documents(state: RAGState):
    # LLMを用いて「質問の答えがドキュメントに含まれているか」をチェック
    grade = grader_llm.invoke({"question": state["question"], "docs": state["documents"]})
    if grade.is_relevant:
        return {"search_retry_count": state["search_retry_count"]}
    else:
        # 関連性が低いと判断された場合は、再検索フラグとしてカウントを増やす
        return {"search_retry_count": state["search_retry_count"] + 1}

# 3. 再検索のためにクエリを書き換えるノード
def rewrite_query(state: RAGState):
    # LLMを使って新しい検索キーワードを生成
    better_question = query_llm.invoke(state["question"])
    new_docs = vector_store.similarity_search(better_question)
    return {"documents": new_docs, "question": better_question}

# 4. 回答を生成するノード
def generate(state: RAGState):
    answer = generator_llm.invoke({"question": state["question"], "docs": state["documents"]})
    return {"generation": answer}

# グラフの定義
workflow = StateGraph(RAGState)
workflow.add_node("retrieve", retrieve)
workflow.add_node("grade_docs", grade_documents)
workflow.add_node("rewrite_query", rewrite_query)
workflow.add_node("generate", generate)
```

```
workflow.add_edge(START, "retrieve")
workflow.add_edge("retrieve", "grade_docs")

# 適合性評価に基づいて分岐させる条件付きエッジ
def decide_to_generate(state: RAGState):
    # ドキュメントが不適合で、再試行回数が上限に達していなければクエリ書き換えに進む
    if state["search_retry_count"] > 0 and state["search_retry_count"] < 3:
        return "rewrite_query"
    return "generate"

workflow.add_conditional_edges(
    "grade_docs",
    decide_to_generate,
    {
        "rewrite_query": "rewrite_query",
        "generate": "generate"
    }
)
workflow.add_edge("rewrite_query", "grade_docs")  # 再検索後は再度評価を行う
workflow.add_edge("generate", END)

app = workflow.compile()
```

この一連の「検索 -> 評価 -> (失敗なら再クエリ・再検索) -> 生成 -> 検証 -> (失敗なら再生成) -> 完了」というループ処理を実装することで、誤情報の極めて少ない、極めて信頼性の高い知識検索システムを構築することが可能になります。

6.3 コードの自動生成・実行・修正ループ

AIにプログラムのコードを書かせる際、一発で完璧に動作するコードを出力することは稀です。プログラミング言語の細かな仕様変更やライブラリの依存関係により、軽微なエラーが発生することが多いためです。そこで、AIが書いたコードを実際にテスト環境で実行し、エラーが出た場合はそのエラーログをAIにフィードバックして自動修正させる「コーディングループ」を実装します。

まず、AIがユーザーの要望に従ってコードを出力する生成ノードを実行します。次に、そのコードをサンドボックス(安全な実行環境)で実行し、テストを行う実行ノードを作成します。テストが成功すれば処理は終了しますが、エラーが発生した場合は、エラーメッセージとスタックトレース(エラーの詳細な発生場所)を状態(State)に書き込み、生成ノードへと差し戻す条件付きエッジを張ります。差し戻された生成ノードでは、エラーの原因をLLMが分析し、コードの修正版を出力します。

以下は、コード自動生成・実行・修正ループの実装コード例です。

```python
import subprocess
from typing import TypedDict
from langgraph.graph import StateGraph, START, END

class CodingState(TypedDict):
    task: str       # 実装したいプログラムの要件
    code: str       # LLMが生成したPythonコード
    error: str      # 実行した際のエラー内容
    iterations: int # 修正ループの回数

# 1. コード生成（または修正）ノード
def generate_code(state: CodingState):
    # エラーがある場合はエラーを修正するためのプロンプトでLLMを呼び出す
    if state["error"]:
        prompt = f"タスク: {state['task']}\n生成したコード:\n{state['code']}\nエラー内
容:\n{state['error']}\n上記エラーを修正してください。"
    else:
        prompt = f"タスク: {state['task']}\nこの要件を満たすPythonコードを書いてください。"

    code_output = coding_llm.invoke(prompt)
    return {"code": code_output, "iterations": state["iterations"] + 1, "error": ""}

# 2. コードを実行してテストするノード
def execute_test(state: CodingState):
    try:
        # 生成されたコードを一時ファイルに保存して実行する（安全な環境を想定）
        with open("temp_script.py", "w", encoding="utf-8") as f:
            f.write(state["code"])

        # サブプロセスとして実行し、結果を待つ
        result = subprocess.run(["python", "temp_script.py"], capture_output=True, text=True, timeout=5)

        if result.returncode == 0:
            return {"error": ""}  # テスト成功
        else:
            # 実行エラー（例外など）が発生した場合
            return {"error": result.stderr}
    except Exception as e:
        return {"error": str(e)}

# グラフの定義
workflow = StateGraph(CodingState)
```

```
workflow.add_node("generator", generate_code)
workflow.add_node("tester", execute_test)

workflow.add_edge(START, "generator")
workflow.add_edge("generator", "tester")

# テスト結果に基づいてループか終了かを分岐する条件付きエッジ
def check_test_result(state: CodingState):
    if state["error"] and state["iterations"] < 5:
        # エラーがあり、実行回数が上限（5回）未満であれば、再生成（修正）に戻る
        return "generator"
    return "end"

workflow.add_conditional_edges(
    "tester",
    check_test_result,
    {
        "generator": "generator",
        "end": END
    }
)

app = workflow.compile()
```

この修正ループを最大3〜5回などと決めて回すことで、人間の手を煩わせることなく、最終的に「構文エラーがなくテストを通過したプログラム」だけをユーザーに提供する自律的なコード開発ツールを実現できます。

第7章 本番運用とデバッグ

7.1 LangSmithを用いた実行の追跡(トレーシング)と評価

AIアプリケーションの開発において、ブラックボックス(中身が見えない状態)になりがちなLLMの挙動を可視化することは、デバッグや品質向上のために欠かせません。LangChainファミリーが提供する「LangSmith(ラングスミス)」は、本番運用時における実行のすべてを追跡(トレーシング)するためのプラットフォームです。

LangSmithを連携させると、プログラム側のコードを変更することなく、環境変数を設定するだけで、すべてのAPI呼び出し、プロンプトの具体的な値、LLMの応答時間、トークンの消費量などをWebダッシュボード上で詳細に確認できます。特にLangGraphを使った複雑なエージェントでは、「どのノードがどの順序で実行され、状態(State)がどう書き換わったか」をタイムライン形式でビジュアルに追跡できるため、バグの原因特定が劇的に早くなります。

以下は、PythonスクリプトからLangSmithのトレーシングを有効化するための環境変数設定コードです。

```
import os

# LangSmithの連携に必要な環境変数を設定します
#  (APIキーは事前にLangSmithのアカウントから取得しておきます)
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = "lsv2_pt_xxxxxxxxxxxxxxxxxxxxxx"
os.environ["LANGCHAIN_PROJECT"] = "my-langgraph-agent"  # 管理画面でのプロジェクト名

# この状態でLangChainやLangGraphのプログラムを実行するだけで、
# 自動的にすべての呼び出し履歴がLangSmithのクラウド上に記録されます。
```

また、蓄積された実行データをもとにテスト用データセットを作成し、プロンプトやモデルを変更した際にシステムの精度がどう変化したかを自動でテスト・評価(エバリュエーション)する機能も備えており、継続的な改善を支えるインフラとして機能します。

7.2 エラーハンドリングとリトライ処理の設計

本番環境では、APIの通信制限(レートリミット)やネットワークの一時的な瞬断、AIモデルのサーバー障害など、予測できないエラーが頻繁に発生します。信頼性の高いシステムを作るためには、これらの例外的な状況に耐えられる堅牢な「エラーハンドリング」と「リトライ(再試行)処理」を組み込んでおく必要があります。

LangGraphでは、ノードごとにリトライポリシー(再試行のルール)を簡単に設定できます。例えば、一時的なエラーが起きた際には「1秒待って再試行し、それでもダメなら待ち時間を倍(2秒、4秒...)にして最大3回まで繰り返す(指数バックオフ)」といった処理を数行の宣言的なコードで実装可能です。また、どうしてもエラーが解決しない場合には、現在の

状態（State）を安全に保存した上で処理を一時停止し、管理者に通知メールを送って手動での復旧を待つような高度なフローも、グラフの設計段階で組み込むことができます。

以下は、特定のノードに対し、自動リトライポリシーを設定するコード例です。

```
from langgraph.graph import StateGraph
from langgraph.prebuilt import RetryPolicy

# リトライポリシー（再試行ルール）の定義
# エラーが発生した際、間隔を空けながら最大3回まで再実行します
custom_retry_policy = RetryPolicy(
    max_attempts=3,          # 最大3回実行を試みる
    initial_interval=1.0,    # 初回のエラーから再試行までの待機時間（秒）
    backoff_factor=2.0,      # 待機時間を倍々にする（1秒、2秒、4秒…）
    retry_on=Exception       # すべての例外エラーを対象にする
)

workflow = StateGraph(MyState)

# ノードを登録する際、retryオプションとしてポリシーを渡します
workflow.add_node("api_call_node", call_external_api, retry=custom_retry_policy)
```

こうしたエラーへの備えを丁寧に行うことで、夜間にバックグラウンドで大量のバッチ処理を実行するようなケースでも、途中でシステムがクラッシュしてデータが消失するリスクを最小限に抑えることができます。

7.3 ストリーミング出力によるユーザー体験の向上

LLMの応答生成や、LangGraphの複雑なマルチエージェントの処理には、数秒から数分といった長い時間がかかることがあります。ユーザーに対して画面が固まったような状態を見せ続けるのはストレスを与えるため、処理の進捗をリアルタイムに伝える「ストリーミング出力」の実装が必須となります。

LangGraphは、強力なストリーミング機能を標準でサポートしています。ストリーミングには大きく分けて2つのモードがあります。一つは、LLMが言葉を組み立てるそばから1文字ずつリアルタイムに出力していく「メッセージストリーミング（トークンレベル）」です。もう一つは、エージェントの処理が進むごとに「現在、検索ノードが完了しました」「次に評価ノードを実行中です」といった、グラフの各ノードの処理状況（Stateの遷移）を順次出力する「更新ストリーミング（ノードレベル）」です。

以下は、ノードごとの実行進捗と、モデルの出力トークンをリアルタイムで出力するストリーミング実装のコード例です。

```
# 1. ノードレベルのストリーミング（どのノードが処理を終え、何を返したか）
inputs = {"messages": [("user", "LangGraphについて教えて")]}

# streamメソッドを呼び出し、更新状況を随時ループで処理します
for chunk in app.stream(inputs, config):
    for node_name, state_update in chunk.items():
        print(f"\n--- Node: {node_name} が処理を完了しました ---")
        # 更新された状態（メッセージなど）を画面に表示
        if "messages" in state_update:
            print(state_update["messages"][-1].content)


# 2. トークンレベルのストリーミング（LLMが生成中の文字をリアルタイムに表示）
# astream_eventsメソッドを使用して非同期（async）でイベントを取得します
async for event in app.astream_events(inputs, config, version="v2"):
    kind = event["event"]

    # チャットモデルが新しい文字（トークン）を出力したイベントを検知
    if kind == "on_chat_model_stream":
        content = event["data"]["chunk"].content
        if content:
            # 改行なしで文字をリアルタイムに出力
            print(content, end="", flush=True)
```

これら2つのストリーミングをフロントエンド（Web画面）と適切に繋ぎ込み、ローディングアニメーションや途中経過をテンポよく表示することで、ユーザーが体感する待ち時間を大幅に削減し、滑らかで使い心地の良いAIアプリケーションを提供することができます。

第8章 おわりに

8.1 まとめと今後の展望

本書を通じて、LLMアプリケーション開発の基礎であるLangChainから、状態管理とループ処理を軸とした高度なエージェント構築を可能にするLangGraphまでを体系的に学びました。これらのツールを使いこなすことで、従来の直線的なプログラムでは不可能だった「自律的に思考し、行動を修正し、業務を完遂するシステム」を現実のものとすることができます。

AIの進化スピードは非常に早く、モデルの処理能力向上や新しいアルゴリズムの登場が続いています。しかし、どれほどLLM単体の性能が上がろうとも、それを既存のビジネスロジックや業務システムと調和させ、安全かつ確実に制御するための枠組み（オーケストレーション）の重要性は変わりません。本書で学んだ「グラフによる状態遷移の設計」や「Human-in-the-loopによる安全性の確保」といった設計原則は、技術の根底を支える普遍的な知見として、今後のあらゆるAI開発において強力な武器となるはずです。

恐れることなく、まずは身近な小さな業務の自動化からLangGraphを取り入れ、一歩一歩自律型システムの可能性を広げていってください。あなたの開発するAIエージェントが、多くの人々の仕事を支えるパートナーとなることを願っています。

8.2 参考リソースおよび公式ドキュメントの紹介

開発を進めるにあたって、公式の最新情報やコミュニティのサポートを活用することは非常に重要です。ここでは、学習をさらに深め、トラブルシューティングを行うために役立つ信頼性の高い参考リソースを紹介します。

まず最も重要なのが、開発元が提供する公式ドキュメントです。LangChain公式ドキュメント（python.langchain.com）およびLangGraph公式ドキュメント（langchain-ai.github.io/langgraph/）には、常に最新のAPIリファレンスやチュートリアルが掲載されています。特にLangGraphのドキュメントには、数多くのデザインパターン（マルチエージェントやプランニングなど）の具体的な実装例（Cookbook）が豊富に用意されており、実務のコード設計における直接の参考資料となります。

また、LangSmithのダッシュボードや、公式のYouTubeチャンネルで公開されている概念解説の動画、開発元のGitHubリポジトリのDiscussionなども、世界中のエンジニアの知恵が集まる貴重な情報源です。これらのリソースをブックマークし、公式の推奨する最新のベストプラクティスを常に意識しながら開発を進めることをお勧めします。