第12回

LLM活用 - Gemini APIとチャットボット開発

出席認証コード: **4802**

📋 本日のロードマップ

🎯 最終目標：忍者キャラクターチャットボット

Part	内容	成果物
Part 1	**Gemini API取得と設定**	API環境構築・セキュリティ
Part 2	**基本チャットボット作成**	シンプルなAIチャット
Part 3	**キャラクターチャット実装**	個性豊かなキャラクターとの会話

Part 1: Gemini API取得と設定

LLM（大規模言語モデル）とは？

- **Large Language Model**の略
- 大量のテキストデータで学習された人工知能
- 自然な会話や文章生成が可能
- 例：ChatGPT、Gemini、Claude など

Google Gemini の特徴

- Googleが開発した最新のLLM
- 多言語対応（日本語も自然）
- マルチモーダル対応（テキスト + 画像）
- 無料枠あり（月1000回まで）

Google AI Studio でのAPI取得

手順

こちらを参考にして取得してください

⚠️ セキュリティの注意点

- APIキーは**秘密情報**として扱う
- コードに直接書かない
- GitHubなどに公開しない
- 使用量制限を設定する

APIキーの安全な管理方法

1. Streamlit Secretsを使用（推奨）

```
# .streamlit/secrets.toml
GEMINI_API_KEY = "your_api_key_here"
```

2. アプリケーションでの読み込み

```
import streamlit as st

# Streamlit Secretsから読み込み
api_key = st.secrets["GEMINI_API_KEY"]
```

3. .streamlit/secrets.tomlファイルの作成手順

```
# プロジェクトフォルダ内で
mkdir .streamlit
echo 'GEMINI_API_KEY = "your_api_key_here"' > .streamlit/secrets.toml
```

Part 2: 基本チャットボットの作成

必要なライブラリのインストール

```
pip install google-generativeai streamlit pillow
```

基本的な使い方

```
import google.generativeai as genai

# API設定
genai.configure(api_key="YOUR_API_KEY")

# モデルの初期化
model = genai.GenerativeModel('gemini-2.0-flash-light')

# テキスト生成
response = model.generate_content("こんにちは！元気ですか？")
print(response.text)
```

基本チャットボットの作り方

Step 1: 必要なライブラリの準備

```
import streamlit as st
import google.generativeai as genai

# ページ設定
st.set_page_config(page_title="基本チャットボット", page_icon="🤖")
st.title("🤖 Gemini 基本チャットボット")
```

🔍 Step 1 の詳細解説

各行の役割：

- `import streamlit as st`: Webアプリを作るためのライブラリ
- `import google.generativeai as genai`: Gemini APIを使うためのライブラリ
- `st.set_page_config()`: ブラウザのタブに表示される設定
  - `page_title`: タブのタイトル
  - `page_icon`: タブのアイコン
- `st.title()`: ページの一番上に表示される大きなタイトル

Step 2: APIキーの設定とモデル初期化

```
# Streamlit SecretsからAPIキーを取得
api_key = st.secrets["GEMINI_API_KEY"]

# Gemini APIの設定
genai.configure(api_key=api_key)
model = genai.GenerativeModel('gemini-2.0-flash-lite')
```

🔍 Step 2 の詳細解説

各行の役割：

- `st.secrets["GEMINI_API_KEY"]`：`.streamlit/secrets.toml` ファイルからAPIキーを安全に読み込み
- `genai.configure(api_key=api_key)`：Google AI にAPIキーを設定してログイン
- `genai.GenerativeModel()`：AIモデルを初期化
  - `'gemini-2.0-flash-lite'`：高速で軽量なGeminiモデルを指定

なぜこの順番？

1. APIキーを取得 → 2. API設定 → 3. モデル準備 の順序で初期化

Streamlitチャット機能の解説

`st.chat_message()` - チャットメッセージの表示

```
# role = "user" または "assistant"
with st.chat_message("user"):       # ユーザーメッセージ（右側に表示）
    st.markdown("こんにちは！")

with st.chat_message("assistant"): # AIメッセージ（左側に表示、アイコン付き）
    st.markdown("こんにちは！何をお手伝いしましょうか？")
```

`st.chat_input()` - チャット入力フィールド

```
# チャット専用の入力フィールド（下部に固定表示）
prompt = st.chat_input("ここにメッセージを入力...")

# 通常のテキスト入力との違い
# st.text_input() → 通常の入力フィールド
# st.chat_input() → チャット専用、送信ボタン付き、下部固定
```

Step 3: 基本的なユーザー入力とAI応答

```
# ユーザー入力（画面下部に固定表示される入力フィールド）
prompt = st.chat_input("メッセージを入力してください...")
if prompt:  # ユーザーが何か入力してEnterを押した場合

    # 1. ユーザーメッセージを表示（チャット画面の右側に表示）
    with st.chat_message("user"):
        st.markdown(prompt)

    # 2. AI応答を生成・表示（チャット画面の左側、アイコン付きで表示）
    with st.chat_message("assistant"):
        response = model.generate_content(prompt)
        st.markdown(response.text)
```

🔍 Step 3 の詳細解説

コードの流れ：

1. `st.chat_input()`: ユーザーからの入力を待つ（画面下部固定）
2. `if prompt:`: 何か入力があった場合のみ処理を実行
3. `st.chat_message("user")`: ユーザーメッセージを右側に表示
4. `model.generate_content()`: Gemini APIでAI応答を生成
5. `st.chat_message("assistant")`: AI応答を左側に表示

注意点：

- この段階では履歴は保存されない（一回限りの会話）
- ページを更新すると会話が消える

Step 4: メッセージ履歴の管理（会話の記録）

```
# メッセージ履歴の初期化
if "messages" not in st.session_state:
    st.session_state.messages = []

# 過去のメッセージを表示
for message in st.session_state.messages:
    # message["role"] は "user" または "assistant"
    # ユーザーメッセージは右側、AIメッセージは左側に表示される
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# ユーザー入力
prompt = st.chat_input("メッセージを入力してください...")
if prompt:
    # 1. ユーザーメッセージを表示
    with st.chat_message("user"):
        st.markdown(prompt)

    # 2. ユーザーメッセージを履歴に追加（次回表示時のため）
    st.session_state.messages.append({"role": "user", "content": prompt})

    # 3. AI応答を生成・表示
    with st.chat_message("assistant"):
        response = model.generate_content(prompt)
        st.markdown(response.text)

    # 4. AI応答を履歴に追加（次回表示時のため）
    st.session_state.messages.append({
        "role": "assistant",
        "content": response.text
    })
```

🔍 Step 4 の詳細解説

メッセージ履歴の仕組み：

- `st.session_state.messages`: **辞書のリスト**形式でメッセージを保存
- 各メッセージ = `{"role": "user/assistant", "content": "メッセージ内容"}`

履歴管理の流れ：

1. **初期化**: 空のリストを作成（初回アクセス時のみ）
2. **過去表示**: 保存された全メッセージを順番に表示
3. **新規追加**: ユーザー・AI応答を両方とも履歴に追加

なぜ辞書のリスト？

- 役割（user/assistant）と内容を分けて管理
- 表示時に適切な位置・スタイルを適用可能
- Streamlitの `st.chat_message()` に直接対応

セッション状態の重要性：

- ページ更新しても会話が消えない

Part 3: テンプレートベースでチャット機能追加

🎯 実習方針：プロンプトエンジニアリングにフォーカス

目標： 準備済みテンプレートの`create_character_prompt`関数を完成させる

📝 実習の流れ

1. **テンプレートファイルを使用**（`character_chat_template.py`）
2. **コアな部分のみ実装**（プロンプト作成関数）
3. **その他は準備済み**（チャット機能、状態管理など）

実習準備：テンプレートファイルの確認

事前準備されている機能

✅ **準備済み：**

- Gemini API設定とエラーハンドリング
- キャラクター紹介機能（既存機能）
- チャットボタンとUI
- メッセージ履歴管理
- 会話コンテキストの管理
- エラーハンドリング

🎯 **実装が必要：**

- `create_character_prompt` 関数の中身（**10~15行程度**）

📋 課題：プロンプト作成関数の実装

実装する関数

```
def create_character_prompt(char):
    """
    キャラクター固有のプロンプトを作成する関数

    Args:
        char (dict): キャラクター情報の辞書
            - name: キャラクター名
            - type: 性格のリスト
            - appearance: 見た目のリスト
            - first_person: 一人称のリスト
            - desc: 詳細説明

    Returns:
        str: Gemini APIに送信するプロンプト文字列
    """
    # 🎯 ここに実装してください！
```

📝 実装の手順とヒント

Step 1: リストを文字列に変換

```
# ヒント1: リストを文字列に変換
personality = ', '.join(char['type'])
appearance = ', '.join(char['appearance'])
first_person = ', '.join(char['first_person'])
```

Step 2: プロンプトの構造を考える

必要な要素：

1. キャラクター名での会話指示
2. キャラクター設定（性格、見た目、一人称、詳細）
3. 会話のルール（5つ程度）
4. 会話開始の指示

Step 3: f文字列でプロンプト作成

プロンプトの基本構造

```
prompt = f"""
あなたは「{char['name']}」として会話してください。

【キャラクター設定】
- 名前: {char['name']}
- 性格: {personality}
- 見た目の特徴: {appearance}
- 一人称: {first_person}
- 詳細設定: {char['desc']}

【会話のルール】
1. このキャラクター設定を一貫して保つ
2. キャラクターの性格や口調を反映した自然な会話をする
3. 忍術学園の世界観を活かした話をする
4. ユーザーに親しみやすく接する
5. 簡潔で分かりやすい回答をする

それでは、{char['name']}として会話を始めましょう。
"""
```

🔧 実装済み機能の解説

1. API設定（準備済み）

```
# エラーハンドリング付きAPI設定
try:
    api_key = st.secrets["GEMINI_API_KEY"]
    genai.configure(api_key=api_key)
    model = genai.GenerativeModel('gemini-2.0-flash-lite')
    chat_available = True
except:
    chat_available = False
    st.warning("⚠️ Gemini APIが設定されていません")
```

2. チャット状態管理（準備済み）

```
# キャラクター別状態管理
chat_key = f"chat_active_{char['name']}"
messages_key = f"messages_{char['name']}"
```

🔧 会話の流れ（準備済み）

プロンプト使用箇所

```
# あなたが作成した関数がここで使用されます
character_prompt = create_character_prompt(char)

# 会話履歴と組み合わせてAIに送信
full_prompt = character_prompt + "\n\n【現在の会話】\n"
for msg in st.session_state[messages_key][-5:]:
    role = "ユーザー" if msg["role"] == "user" else char['name']
    full_prompt += f"{role}: {msg['content']}\n"
full_prompt += f"\n{char['name']}:"

response = model.generate_content(full_prompt)
```

🏃 実習の進め方

実習手順

1. `character_chat_template.py` を開く
2. `create_character_prompt` 関数を見つける
3. TODOコメント部分を実装
4. アプリを実行してテスト
5. キャラクターとチャットして動作確認

💡 プロンプトエンジニアリングのコツ

良いプロンプトの特徴

1. **明確な役割指定**
    - 「〜として会話してください」
2. **具体的な設定**
    - 性格、口調、背景などの詳細
3. **行動指針**
    - 会話のルールや注意事項
4. **一貫性の確保**
    - キャラクター設定を保つ指示

避けるべき要素

- 曖昧な指示、矛盾した設定、長すぎるプロンプト

デモ: 完成版のキャラクターチャット

実際に動かしてみよう！

1. **キャラクター紹介機能**
    - 好みに合ったキャラクター推薦
    - または、ランダムキャラクター表示
2. **個別チャット機能**
    - 各キャラクターに「チャットを開始する」ボタン
    - ボタンを押すとその場でチャット開始
    - キャラクター設定に基づく自然な会話
3. **複数キャラクターとの会話**
    - 異なるキャラクターと個別に会話可能
    - それぞれの履歴が独立して保存

今日学んだ技術のまとめ

1. Gemini API

- Google AI Studioでの取得
- `google-generativeai` ライブラリの使用
- APIキーの安全な管理

2. Streamlitチャット機能

- `st.chat_message()` でのメッセージ表示
- `st.chat_input()` でのユーザー入力
- `st.session_state` での履歴管理

3. プロンプトエンジニアリング

- キャラクター設定の定義
- システムプロンプトの作成
- 会話コンテキストの管理

質疑応答 & まとめ

今日のポイント

1. **LLM API**の基本的な使い方を習得
2. **チャットボット**の実装パターンを理解
3. **プロンプトエンジニアリング**の重要性を体験

次回への準備

- 今回作成したコードを復習
- 可能であれば発展課題に挑戦
- データ分析機能との連携アイデアを考える

お疲れさまでした！ 🎉