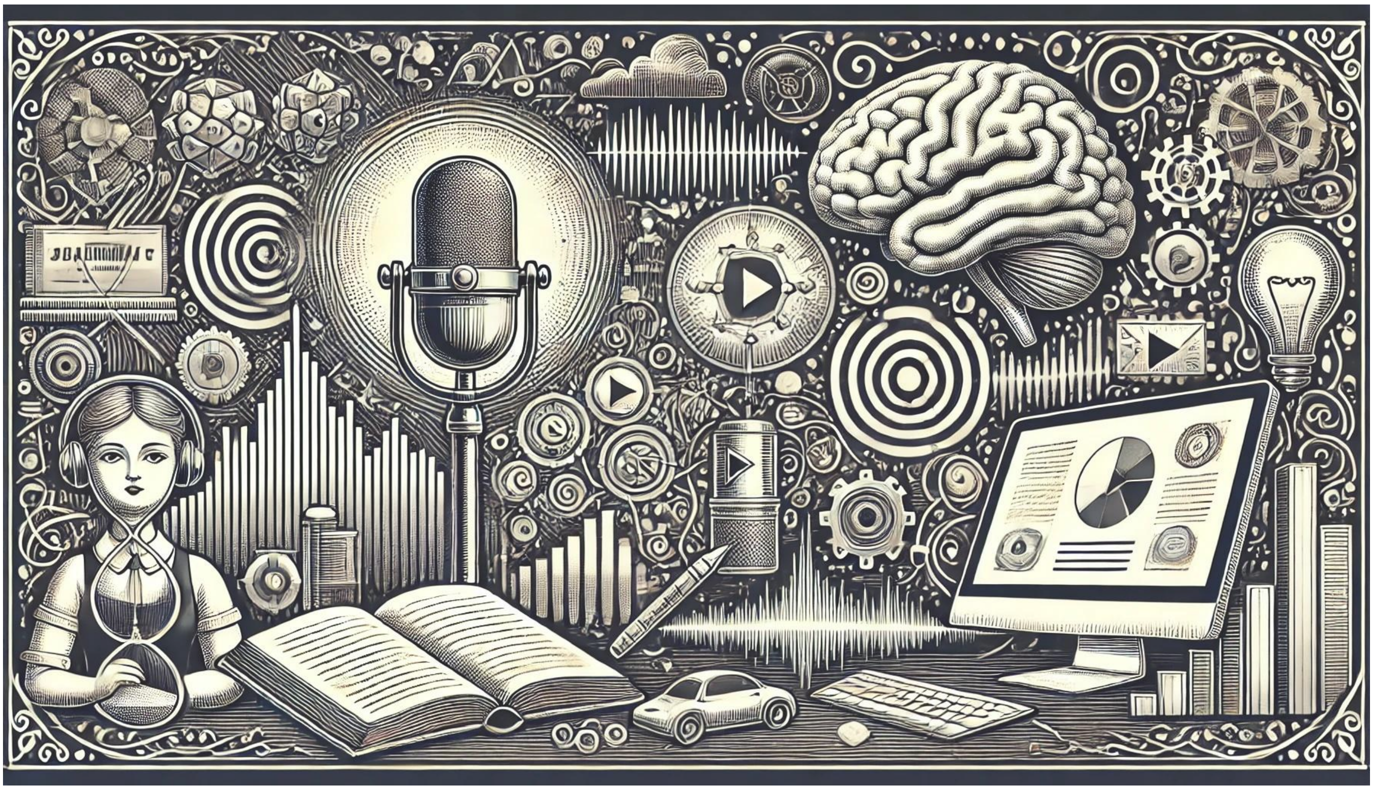
使用 Flutter 打造語音助理

使用 Flutter 所提供的套件以及 Gemini 打造自己的語音助理

楊維中 *a.k.a zonble*

目錄

前言



這幾年語音助理應用以各種方式走入了每個人的生活。蘋果在 iPhone 4 之後加入了 Siri 功能，也應用在 Apple TV、HomePod 等產品上，Amazon 打造了 Alexa 智慧喇叭，Google 則在 Android 手機與 Google Nest 等裝置上加入了 Google Assistant 功能，之後又有更多的智慧喇叭以及車載語音產品。這些語音助理的功能不僅僅是回答問題，還能夠控制家電、播放音樂、設定鬧鐘等等。

我們可以注意到，這些語音助理有一個共同的特色，就是他們都有自己的生態系統。這些語音助理首先串接的是自家的服務，而幾乎每家都有自己的音樂、地圖導航等。蘋果就有自家的 Apple Music、Amazon 也有自己的 Prime Music、Google 則有 YouTube Music；地圖方面，蘋果一開始與 Google 合作使用 Google Maps，後來也開發了自家的 Apple Maps。或著，就是與一些最大的音樂、氣象、新聞等服務合作，像是 Spotify、Weather.com 等等。

為了擴大生態圈，第三方開發者也可以為這些語音助理開發自己的服務，像是 iOS App 的開發者可以透過SiriKit，為 Siri 增加功能，讓 Siri 可以串接到 App 的功能。Amazon 則提供了

Amazon 則使用 Alexa Skills，Google 則使用 Actions on Google 來串接各種第三方服務。而由於這些語音助理都是大眾導向的產品，大多建立在這些平台上的服務，也往往是大眾導向的服務，我們還是較少能看到在一些內部、或是專屬的系統上，使用語音助理技術或功能。

這幾年也陸續看到一些店家開始在專屬的服務上提供語音助理功能，像是餐飲店也開始提供語音點餐等，客戶在電家門口的 Kiosk 機台，就可以用語音完成點餐等。但即使如此，我們還是可以期待，語音助理可以應用在更多行業中、更多系統上。

語音助理最適合的場合大概是，當使用者不方便挪出雙手操作，然後又有明確目標，用語音比使用手指或鍵盤滑鼠慢慢選擇有效率時，而且通常會用在比較私密、個人的場合，像我們大概就不會在公共場合大聲叫 Siri 幫我們查詢一些私人資訊。所以我們往往會在駕車時請語音助理協助導航，在家中使用智慧喇叭，以及在運動、慢跑時使用耳機上的語音助理；另一方面，又對周圍環境有一定的要求，周圍如果太過吵雜，往往影響錄音以及語音辨識的效果。

我們或許可以想像，在醫院、病房當中，病床床頭可能擺了台平板，醫護人員或是病人可以快速使用語音命令，完成一些工作或是查詢資料，而不用放開手邊的工作，或是離開病床。剛生完寶寶的母親可以用語音命令，快速查看寶寶的影像。但前提是，語音助理要能夠連接醫院的系統，但目前都還相當欠缺這類型的應用。

以下就是一個使用 Flutter Web 技術開發，運作在 Chrome 瀏覽器當中的語音掛號的功能展示：

在 PDF 檔案中，無法呈現影片，請下載 MP4 影片。

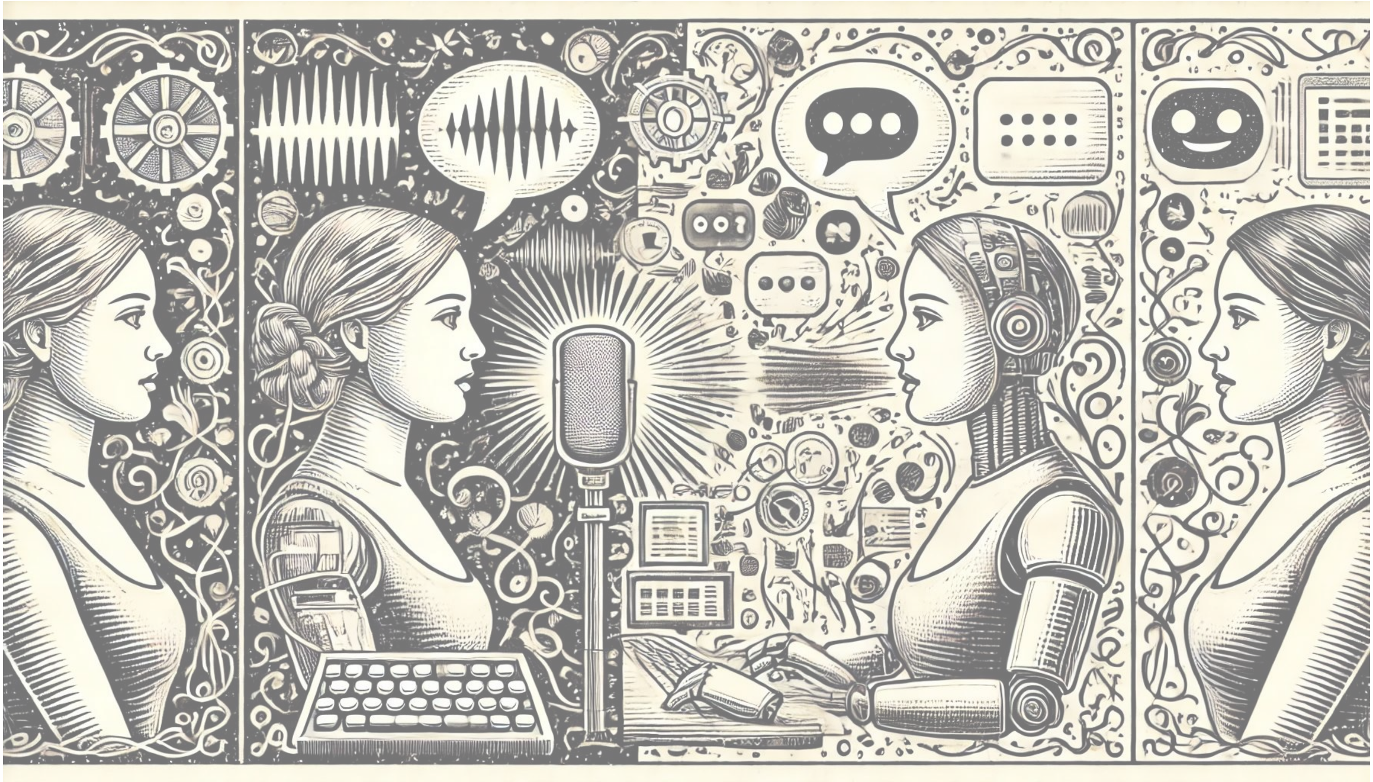
隨著這幾年 Flutter 應用程式的開發框架愈來愈成熟，加上大型語言模型（LLM）也快速普及，結合這兩者，其實就可以只寫大約幾百行的 Dart 程式，就能夠快速打造跨平台的語音助理。在這本小冊子中，會提到：

- 打造語音助理需要哪些元件
- 這些元件有哪些可用的 Flutter 套件
- 如何串接這些元件，成為完整的語音助理
- 如何設計語音互動流程

我預期您在閱讀這本小冊子之前，具備一定的 Flutter 應用程式的開發基礎與經驗。這本小冊子放在 Github 上，如果您發現錯漏，可以直接在 Github 上發 PR 修正，或是在 Github 上開 Issue 討論。因為內容也是以 Git 管理，因此不額外提供版本歷史。

這本小冊子當中的程式碼，也可以從 flutter_dialog 處取得。

聊天機器人與語音助理



隨著 ChatGPT、Gemini 等服務問世，大家應該已經熟悉了所謂的聊天機器人（Chatbot）以及生成式 AI。聊天機器人與虛擬助理其實出自於相同的概念與技術，但是在實際應用上有些許的不同。

根據維基百科，所謂的聊天機器人是「一種設計用來模仿栩栩如生的人類文字或語音互動的軟體或是網頁介面」[1]，至於虛擬助理，則是「根據使用者的命令或是問題等輸入，包括語言輸入，而執行一系列工作或是服務的軟體。 」 此外，「虛擬助理的技術通常會用到聊天機器人的能力，用以模擬人際對話。」[2]，根據這樣的定義，我們可以知道兩者之間都具備與使用者對話的能力，但是虛擬助理會在聊天機器人的基礎上，多出可以執行工作，可以呼叫語音系統以外的程式。像是，如果是導航用的助理，就需要能夠向外呼叫地圖或導航的 SDK。

要能夠順利完成這些工作，虛擬助理—無論是文字助理或語音助理—在了解到使用者想進行某項工作的意圖時，就得進入一套系統化的流程，一步一步引導使用者完成任務，而不能像是聊天機器人一樣，由於只需要進行對話，所以就可以天南地北地閒聊。

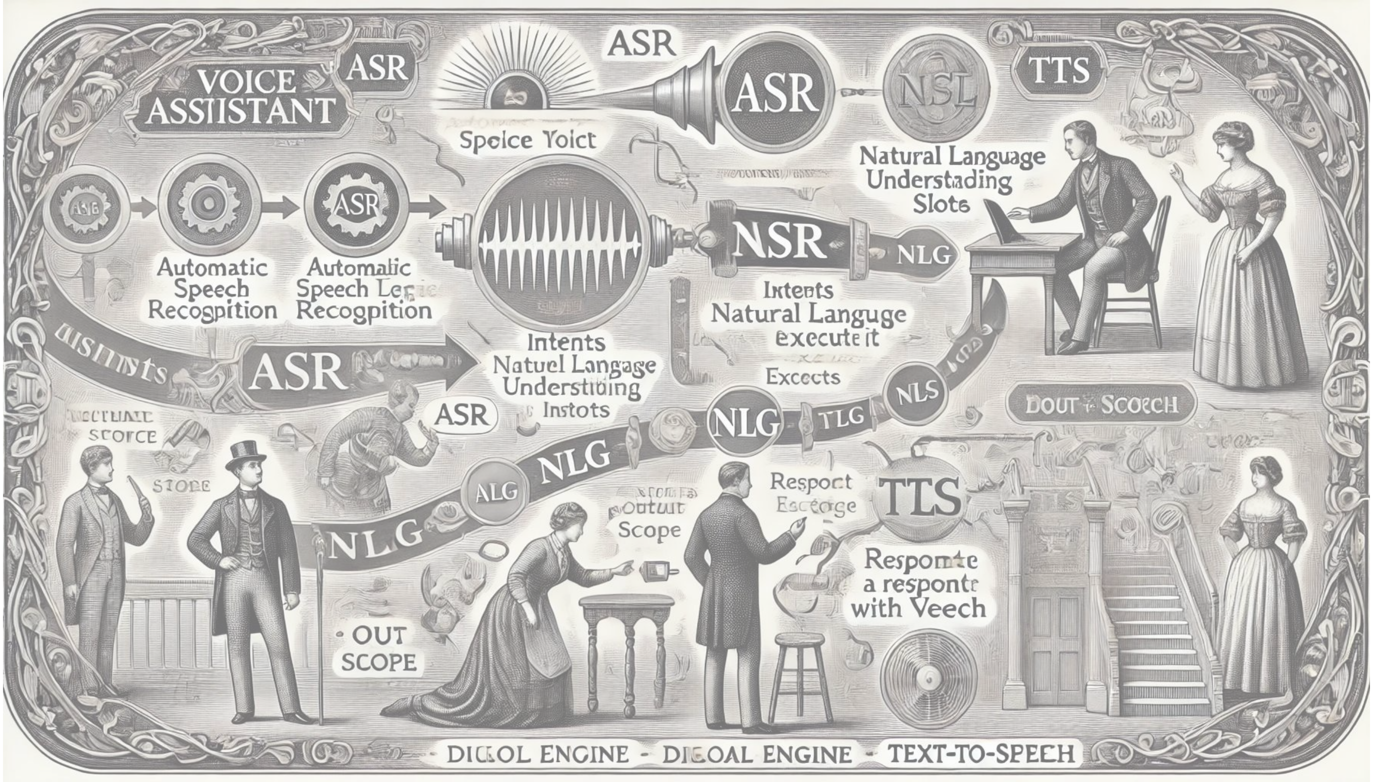
由於 ChatGPT 與 Gemini 等服務都已經開放了 API，在這些服務的範例中，我們可以輕鬆開發出一套聊天機器人，而我們現在還要更進一步，把語音助理可以可以執行的任務區分出來，至於這些任務以外的部分，我們仍然可以交給 LLM 處理，讓 LLM 產生各種回答。

而怎樣「了解、區分出這是一件可以讓語音助理執行的工作」，我們其實也可以用 LLM 的能力達成—過去，人們已經用了一些好幾代之前的技術，打造了許多的語音助理，應用了新的 LLM，我們相信能夠打造出更好的語音助理。

---

1. https://en.wikipedia.org/wiki/Chatbot ↩
2. https://en.wikipedia.org/wiki/Virtual_assistant ↩

語音助理的基本組成


VOICE ASSISTANT
ASR
ASR
Speice Yoict
ASR
NSL
TTS
Natural Language Understading Slots
ASG
ASR
Automatic Speech Recognition
Automalic Speech Legic Recognition
NSR
NLG
Intents Natural Langugе execute it
Excects
ASR
Intents Natuel Langage Understiding Instots
ASR
NLG
TLG
NLS
DOUT SCORCH
SCORCE
STORE
ALG
NLG
NLG
Outuit Scope
Respoct Escetoge
TTS
Responte a respontr with Veech
OUT SCOPE
DICOL ENGINE
DICOAL ENGINE
TEXT-TO-SPEECH

在這邊需要介紹一些專有名詞。一套語音助理系統的組成，通常會包括—ASR、NLU、NLG、TTS 等，在一個完整的語音互動中，扮演各自的角色。

- ASR：Automatic Speech Recognition，自動語音識別，又稱為 STR （Speech-to-Text），負責將語音訊號轉換為文字。對於行動開發者來說，應該清楚蘋果提供了 Sppech 框架，在 Android 系統中也有 SpeechRecognizer 可用，在瀏覽器中也一樣有 SpeechRecognition。除此之外，許多雲端服務，像是 Google Cloud 與 Azure，也具備將上傳的語音資料轉換成文字的服務。
- NLU：Natural Language Understanding，自然語言理解，負責從文字中抽取出意圖，像是知道使用者想要做什麼事情，想要問什麼問題，也可以分析使用者所說的話當中的情緒（悲傷、高興、憤怒），或是口吻是正式還是輕鬆等。在行動開發的框架中，通常沒有這部份的框架，但是所有的交談型 LLM 都一定包含 NLU 的能力。此外，比較有名的 NLU 引擎包括 Google Dialogflow、Microsoft LUIS、IBM Watson、Nuance Recognizer 等等。
- NLG：Natural Language Generation，自然語言生成，負責產生對特定文字的回覆。其實各種 LLM，就是在擔任 NLG 引擎的工作。
- TTS：Text-to-Speech。負責將文字轉換為語音。在行動開發者的框架中，蘋果提供了 AVSpeechSynthesizer、Google 提供了 TextToSpeech。在瀏覽器中，也有 SpeechSynthesis 可用。雲端服務中，像是 Google Cloud 與 Azure 也提供了將文字轉換成語音的服務。

各種組件的分工

在執行一段語音助理的工作流程中，各個組件的分工如下：

c8a7afe0-1f60-486c-be1b-2ff740745f0d

在上圖中，多了一個叫做「對話引擎」的角色，也就是我們需要自行開發的部分。對話引擎需要負責管理一份對應表格（mapping），將各種從 NLU 引擎抽取出的意圖，對應到我們的語音助手可以執行的工作上。如果不能處理這項工作（像使用者只說了「你好」，但是我們沒辦法用「你好」這句話決定可以做什麼事），我們可以有幾個選擇，或是就直接用 TTS 告訴使用者我們不理解你的意圖，或是就由 NLU 引擎產生一個回答，這樣會讓使用者感到比較親切。

至於是可以執行的工作，就會走入一套我們設計好的對話流程。我們在後面再詳細討論。

狀態

從以上的流程圖中，我們也可以發現，對話引擎擁有以下幾種狀態：

- 閒置狀態：也是語音助理的初始狀態。
- 聆聽/辨識狀態：使用者從 App 啓動了麥克風，對系統輸入語音資料，同時 ASR 引擎也在嘗試辨識語音。通常等到使用者停止講話一陣子，我們架設使用者已經把想講的話說完，我們就會進入下一個狀態。而如果在一段時間內，使用者什麼話都沒說，或是沒有任何 ASR 辨識結果，我們也會離開這個狀態，回到閒置狀態，我們也可能用 TTS 提示使用者「我不能理解你的意思」。
- 處理狀態：我們將 ASR 辨識結果送到 NLU 引擎後，NLU 引擎往往需要一段時間分析，我們也往往需要獲得一些其他的資訊，才知道對話應該如何繼續進行。這段時間，我們通常會在 App 上顯示一個等待的畫面，或是用 TTS 提示使用者「我正在處理你的要求」。
- TTS 播報狀態：當我們用 TTS 回應使用者時，我們會進入這個狀態。這個狀態通常會持續一段時間，直到 TTS 播報完畢，我們才會回到閒置狀態。或是，如果我們需要繼續追問使用者，我們會回到聆聽/辨識狀態。

f541293a-fe80-49f5-b20b-55d56da6204d

這些狀態還可以繼續細分，像是前面提到，處理階段中，也可以繼續拆成 NLU 處理中，或是在獲取額外資訊等階段。而 App 應該要監聽這些狀態，並且在不同的狀態下，顯示不同的畫面，或是提供不同的操作，以符合使用者的預期。

```
@immutable
abstract class DialogEngineState {}

/// Idling state
class DialogEngineIdling extends DialogEngineState {}

/// Listening state
class DialogEngineListening extends DialogEngineState {
  final String asrResult;
  DialogEngineListening({required this.asrResult});
}

/// Processing state
class DialogEngineCompleteListening extends DialogEngineState {
  final String asrResult;
  DialogEngineCompleteListening({required this.asrResult});
}

/// TTS playing state
```

```
class DialogEnginePlayingTts extends DialogEngineState {
  final String prompt;
  DialogEnginePlayingTts({required this.prompt});
}
```

介面設計

我們可以使用的各種 ASR、NLU、NLG、TTS 的服務非常多，除了各種已有的選擇之外，我們甚至可能會開發自己的服務，像是在我們自己的伺服器上放置我們自己調整過的模型。從開發對話引擎的角度來看，我們便應該專注於介面，而非個別服務的實作，我們只需要知道 ASR、NLU...每個引擎所具備的能力，之後可以隨時抽換實作。我們的對話引擎大概會像這樣：

- 有 ASR、NLU、NLG、TTS 四個引擎的實例
- 有可以讓外部監聽的狀態
- 有可以讓外部設定的對話流程

```
class DialogEngine implements VuiFlowDelegate {
  final AsrEngine asrEngine;
  final TtsEngine ttsEngine;
  final NluEngine nluEngine;
  final NlgEngine nlgEngine;

  final StreamController<DialogEngineState> _stateStream = StreamController();
  DialogEngineState _state = DialogEngineIdling();
  DialogEngineState get state => _state;
  Stream<DialogEngineState> get stateStream => _stateStream.stream;

  Map<String, VuiFlow> _vuiFlowMap = {};
}
```

ASR

我們定義的 ASR 引擎像這樣：

```
enum AsrEngineState {
  listening,
  notListening,
  done,
}
```

```
abstract class AsrEngine {
  Future<bool> init();
  Future<bool> startRecognition();
  Future<bool> stopRecognition();
  Future<void> setLanguage(String language);

  Function(String, bool)? onResult;
  Function(AsrEngineState)? onStatusChange;
  Function(dynamic)? onError;
  bool get isInitialized;
}
```

在 ASR 引擎的介面上，最主要的 method 就是初始化、開始辨識、停止辨識，以及設定語言。我們也提供了一些 callback，讓外部可以監聽 ASR 引擎的狀態。因為在開啓 ASR 引擎之前，可能需要做一些權限相關的設定，因此設計了一個初始化的 method，而在啓動 ASR 引擎之後，對話引擎就會進入 Listening 狀態，這時候就會開始接收語音輸入，透過 onResult 接收目前辨識出的文字。當使用者停止講話，或是 ASR 引擎辨識出一段語音，我們就會進入 Done 狀態，這時候就可以將辨識結果送到 NLU 引擎。

NLU

```
class NluIntent {
  final String intent;
  final Map slots;

  NluIntent({
    required this.intent,
    required this.slots,
  });

  factory NluIntent.fromMap(Map json) {
    return NluIntent(
      intent: json['intent'] ?? '',
      slots: json['slots'] ?? [],
    );
  }
}

abstract class NluEngine {
  Set<String> availableIntents = <String>{};
  Set<String> availableSlots = <String>{};

  Future<NluIntent> extractIntent(
```

```
    String utterance, {
    String? currentIntent,
    String? additionalRequirement,
  });
}
```

如前所述，NLU 引擎的角色就是從文字抽取出意圖。所以我們在這邊定義了 `NluIntent` 物件，在這個物件中，包含被抽取出的意圖的代號，以及與這個意圖相關的 Slot—所謂的 Slot 就是意圖中的參數。例如「導航到動物園」這句話中，「導航」是使用者想要執行的意圖，而「動物園」就是這個意圖的 Slot。

在 `NluEngine` 介面中，我們定義了一個 method `extractIntent`，這個 method 會接收一段文字，並且回傳一個 `NluIntent` 物件。在這個 method 中，我們也可以設定一些參數，像是目前的意圖、或是一些額外的需求，讓 NLU 引擎可以更好地抽取出意圖。像是，我們告訴 NLU 引擎我們想要哪些 NLU 意圖以及 Slot，就可以幫助他盡可能歸類到我們限制的分類中。

NLG

```
abstract class NlgEngine {
  Future<String?> generateResponse(
    String utterance, {
    bool useDefaultPrompt = true,
    bool? preventMeaningLessMessage,
  });
}
```

NLG 的介面非常簡單，基本上就是將一段文字交給 NLG 引擎，並且回傳一段回應。在這個 method 中，我們也可以設定一些參數，像是是否使用預設的提示，或是是否要避免回傳無意義的訊息。

TTS

TTS 引擎所需要的介面，就是播放某個句子以及與停止播放。此外，我們通常需要能夠設定 TTS 引擎的語言、語速、音量、音高，以及聲音（男聲、女聲等）。我們另外設計了一些 callback，讓外部可以監聽 TTS 引擎的狀態。

```
import 'dart:async';
```

```
abstract class TtsEngine {
  Future<void> playPrompt(String prompt);
  Future<void> stopPlaying();
  Future<void> setLanguage(String language);
  Future<void> setSpeechRate(double rate);
  Future<void> setVolume(double volume);
  Future<void> setPitch(double pitch);
  Future<void> setVoice(Map<String, String> voice);

  Function()? onStart;
  Function()? onComplete;
  Function(String text, int startOffset, int endOffset, String word)?
      onProgress;
  Function(String msg)? onError;
  Function()? onCancel;
  Function()? onPause;
  Function()? onContinue;
}
```

可用的 Flutter 套件

在進入實作 ASR、NLU、NLG、TTS 的套件之前，我們先來看看 Flutter 有哪些套件可以用來實作這些功能。基本上，我們使用 speech、TTS、等關鍵字，就可以在 Dart Pub 上找到許多套件。

ASR 相關套件

- speech_to_text：支援 iOS、Android 以及 Web 的語音辨識套件。這個套件主要用平台原生的語音辨識功能。在使用這個方案時，開發者不用支付額外的服務費用，這也是目前在 Dart Pub 上評分最高、最流行的語音辨識套件。
- google_speech：使用 Google 雲端服務的語音辨識套件。
- azure_speech_recognition_null_safety：使用 Azure 雲端服務的語音辨識套件。
- openai_api：當中包含了 OpenAI 的語音辨識服務。
- dart_openai：另外一套 OpenAI API 的實作。
- ifly_speech_recognition：科大訊飛是一家對岸的語音服務公司，也提供了連接他們的服務的 Flutter 套件。

在使用這些套件時，往往還需要一些額外的權限與隱私設定。像是 iOS 上可能需要在 Info.plist 中設定 `NSMicrophoneUsageDescription` 與 `NSSpeechRecognitionUsageDescription`，告知使用者為什麼要使用麥克風與語音辨識權限。而在 Android 上，可能需要在 AndroidManifest.xml 中設定 `RECORD_AUDIO` 權限，而如果我們想要用藍芽裝置錄音，那可能還要額外設定 `BLUETOOTH` 權限。使用時請先看這些套件的說明。

TTS 相關套件

- flutter_tts：支援 iOS、Android、Web、macOS 與 Windows 文字轉語音套件。一樣使用平台原生的語音合成功能，因此開發者不用支付額外的服務費用，這也是目前在 Dart Pub 上評分最高、最流行的文字轉語音套件。

- text_to_speech：支援 iOS、Android、Web 與 macOS。
- galli_text_to_speech：另外一套支援 iOS、Android、Web 與 macOS 的文字轉語音套件。
- cloud_text_to_speech：介接了 Google、微軟以及 Amazon 的文字轉語音雲端服務的套件。
- flutter_azure_tts：使用 Azure 雲端服務的文字轉語音套件。
- openai_api：當中包含了 OpenAI 的文字轉聲音服務。

在 Dart Pub 上，我們還可以看到有人介接了騰訊、百度、科大訊飛等公司的文字轉語音服務，不過看起來一陣子沒有維護。

NLU/NLG 相關套件

如前所述，我們計畫使用目前流行的 LLM 來實作 NLU 與 NLG。

Gemini

- google_generative_ai：Google 的官方 Gemini 套件，支援各種平台。
- flutter_gemini：另外一套 Gemini 實作
- gemini_flutter：另外一套 Gemini 實作

Gemma

- flutter_gemma：可以讓 Flutter App 使用 Gemma 本地模型的套件。

ChatGPT

當然也有不少與 ChatGPT 有關的套件。

- flutter_chatgpt_api
- chat_gpt_api
- chat_gpt_flutter

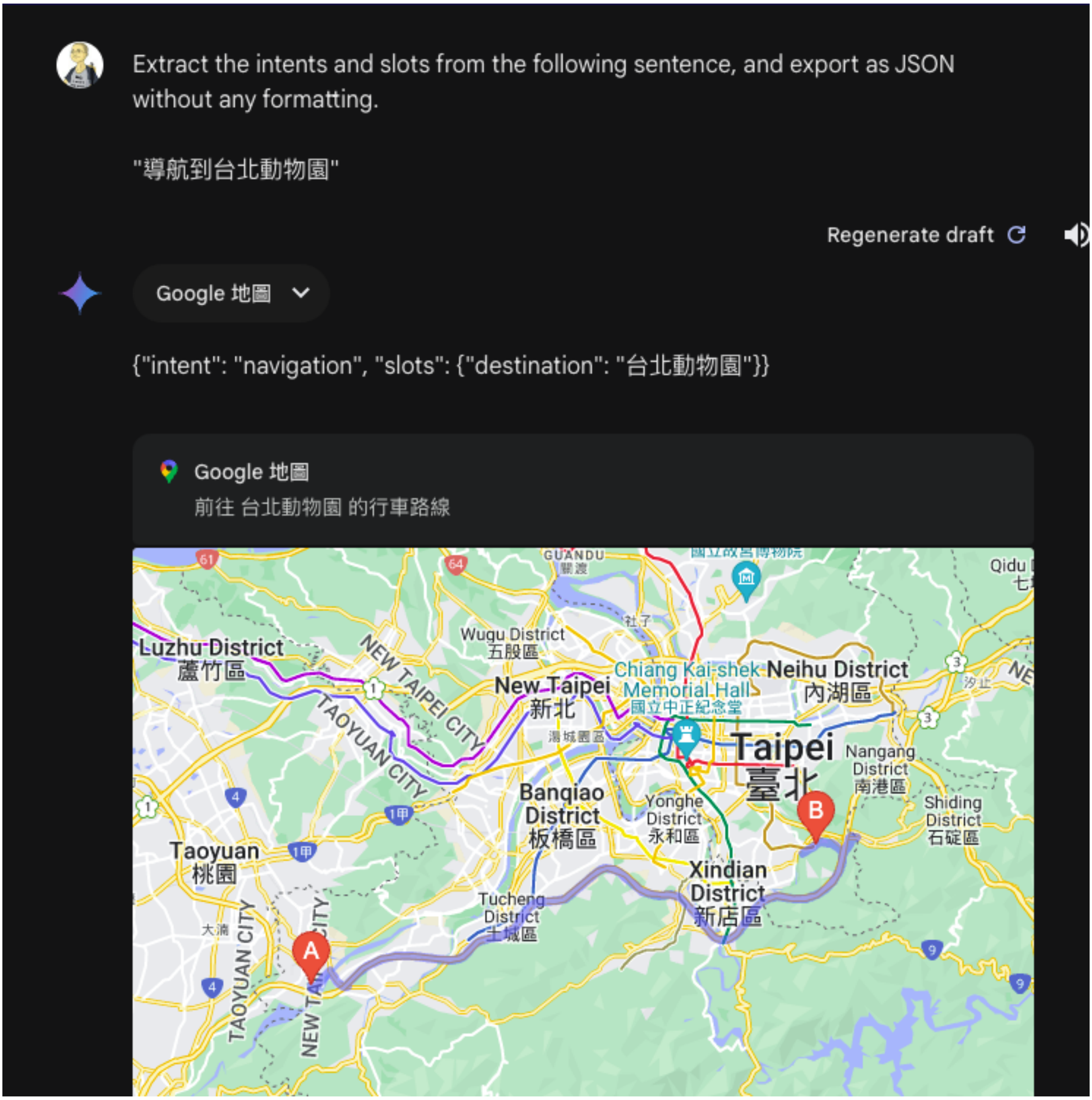
讓 Gemini 成為 NLU 與 NLG 引擎

這一節中，會說明如何實作我們的 NLU 與 NLG 引擎。這裡使用 Gemini 示範，也可以使用其他的 LLM。

抽取 Intent 與 Slots 的 Prompt

讓交談式的 LLM 成為 NLU 引擎其實十分簡單，就是下達與以下類似的 prompt：「請幫我從以下這句話當中抽取出 Intent 與 Slots，然後輸出成 JSON 格式，不要其他的輸出內容」。

我們可以在 Gemini 或是 Google AI Studio 的網頁介面中，就可以試試看我們的 prompt。下圖中，我們嘗試輸入「導航到台北動物園」這句話。



Extract the intents and slots from the following sentence, and export as JSON without any formatting.

"導航到台北動物園"

```
{"intent": "navigation", "slots": {"destination": "台北動物園"}}
```

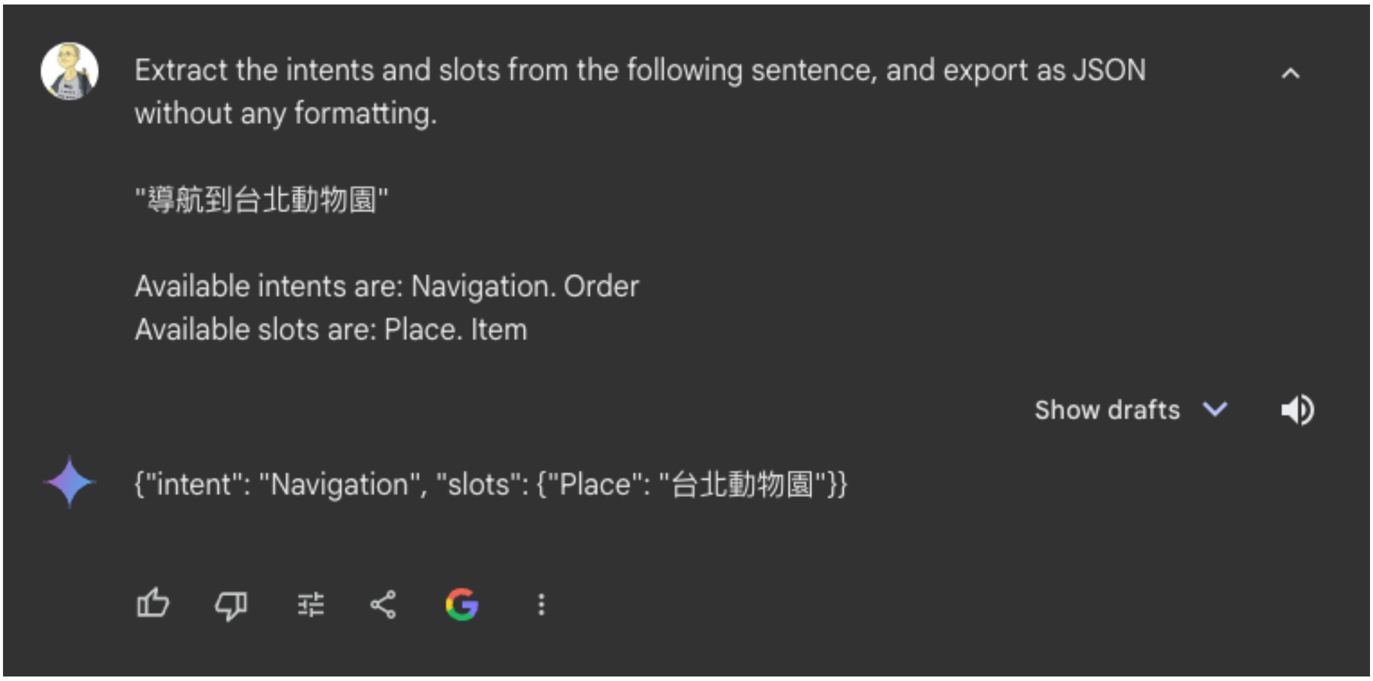
Google 地圖
前往 台北動物園 的行車路線

Gemini 幫我們成功抽取出「navigation」這個意圖，同時也抽取了目的地是「台北動物園」。不過，輸出結果比我們預期得還多，多出了 JSON 以外的地圖部分，但這沒關係，我們可以用用一個 Regular Expression 捕捉輸出結果中的 JSON 部分，然後用 Dart 本身的 JSON parser，轉換成我們需要的 Dart Map 物件。

限制 Intent 的名稱

在上面的範例中，我們雖然讓 Gemini 抽取出了 Intent，但是 Intent 是什麼名稱，完全讓 Gemini 決定，很有可能每次 Gemini 輸出的結果就會不太一樣，像是可能大小寫不一，可能 navigation 變成大寫的 Navigation 或 NAVIGATION。所以，我們可以多做一些補充，讓 Gemini 盡可能產生符合我們預期的結果。

比方說，我們希望導航的意圖固定是大寫的，然後導航的地點從 destination 這個字變成 Place，我們就可以在上面的 prompt 當中，



Extract the intents and slots from the following sentence, and export as JSON without any formatting.

"導航到台北動物園"

Available intents are: Navigation. Order
Available slots are: Place. Item

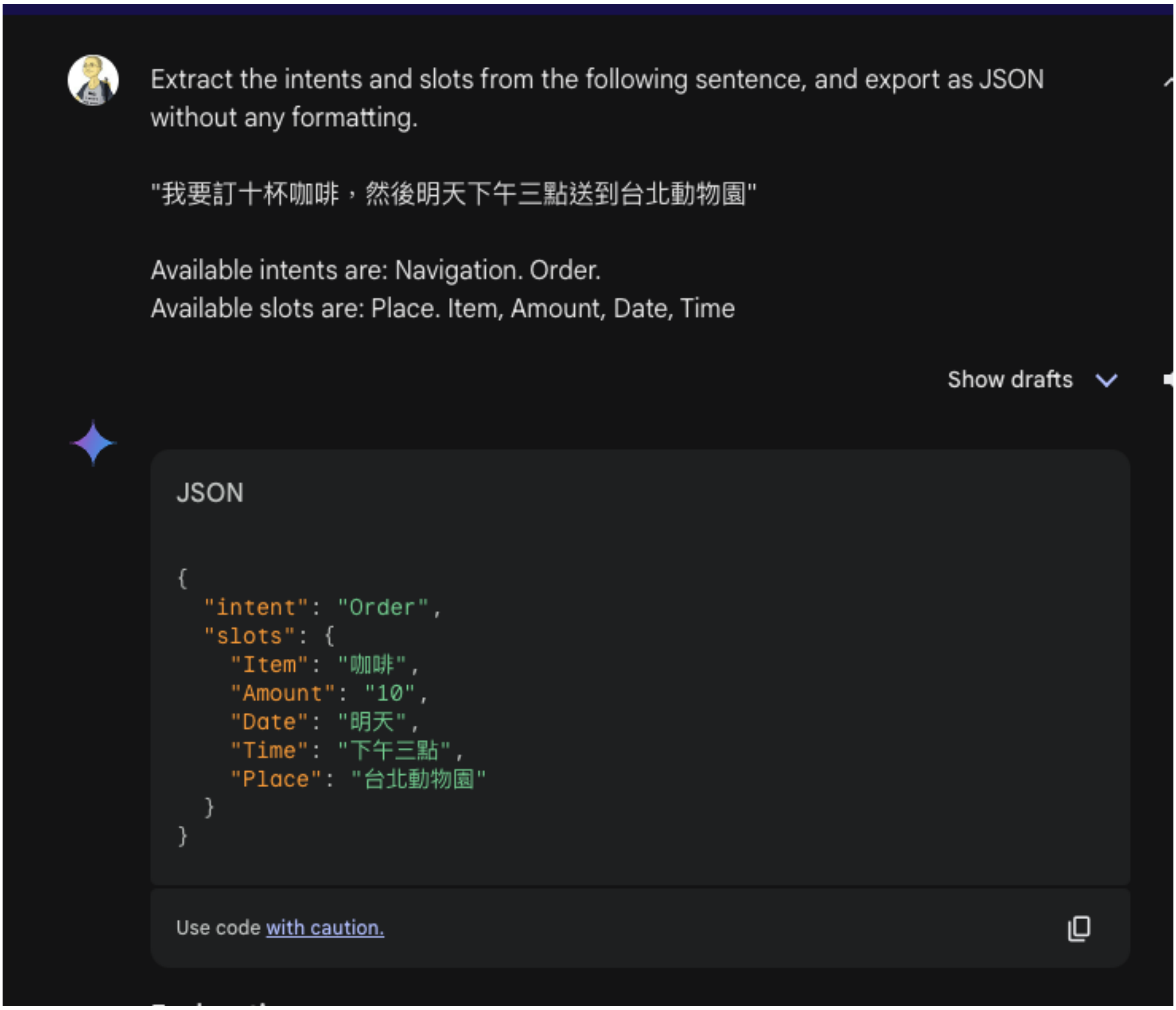
{"intent": "Navigation", "slots": {"Place": "台北動物園"}}

在 Gemini 的輸出結果中，就成為我們所預期指定的 Intent 與 Slot 名稱了。

處理省略指代

在我們的日常對話中，我們經常會用到指代，像是「他」、「她」、「它」等等，代表某個具體的人或是物品，或是用「今天」、「昨天」、「明天」，代替某個具體的日期。如果只是用上面的 prompt，Gemini 仍然可以幫助我們抽取出這些 Slot，

比方說，我們想要處理「我要訂十杯咖啡，然後明天下午三點送到台北動物園」這句話，Gemini 給我們的結果如下：



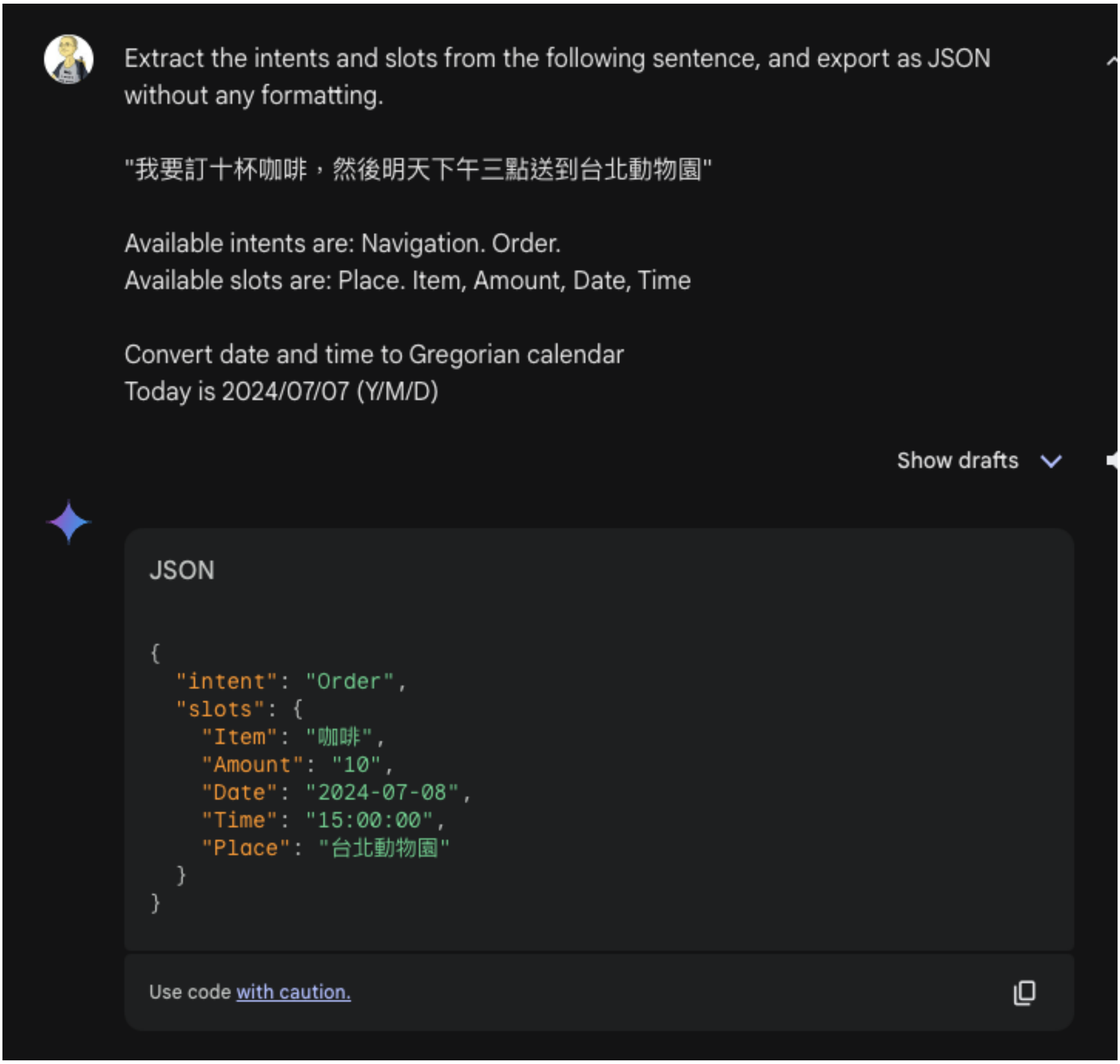
Extract the intents and slots from the following sentence, and export as JSON without any formatting.

"我要訂十杯咖啡，然後明天下午三點送到台北動物園"

Available intents are: Navigation. Order.
Available slots are: Place. Item, Amount, Date, Time

```
{
  "intent": "Order",
  "slots": {
    "Item": "咖啡",
    "Amount": "10",
    "Date": "明天",
    "Time": "下午三點",
    "Place": "台北動物園"
  }
}
```

「下午三點」與「明天」並不太方便我們做後續的處理，除了要把「下午三點」從中文轉換成一般程式的時間格式外，還需要從「明天」中推算出確實的日期。但，我們可以在 prompt 當中多提供一些線索，Gemini 就可以幫我們預先處理好。像是，我們就可以告訴 Gemini 今天是哪一天，然後要求把日期都轉換成西曆。



Extract the intents and slots from the following sentence, and export as JSON without any formatting.

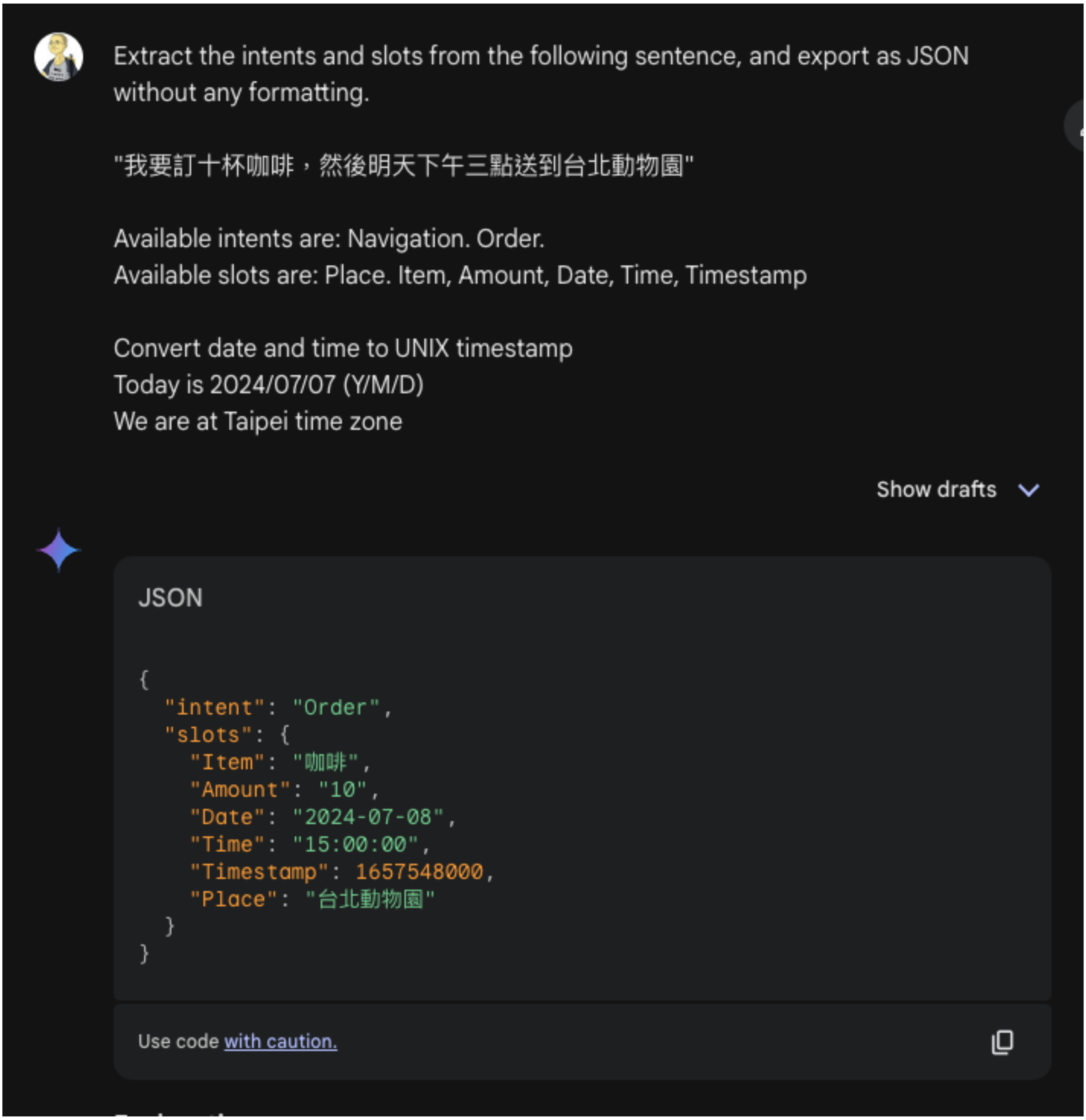
"我要訂十杯咖啡，然後明天下午三點送到台北動物園"

Available intents are: Navigation. Order.
Available slots are: Place. Item, Amount, Date, Time

Convert date and time to Gregorian calendar
Today is 2024/07/07 (Y/M/D)

```
{
  "intent": "Order",
  "slots": {
    "Item": "咖啡",
    "Amount": "10",
    "Date": "2024-07-08",
    "Time": "15:00:00",
    "Place": "台北動物園"
  }
}
```

但很多時候，我們不能夠信賴 Gemini 或是各種 LLM 的能力，比方說，我們現在貪心一點，還想將日期轉換轉換成 UNIX timestamp，結果產生了 `1657548000`，但這其實是一個 2022 年的日期，這就不是我們所預期的結果，而且會造成後面系統的錯亂。



Extract the intents and slots from the following sentence, and export as JSON without any formatting.

"我要訂十杯咖啡，然後明天下午三點送到台北動物園"

Available intents are: Navigation. Order.
Available slots are: Place. Item, Amount, Date, Time, Timestamp

Convert date and time to UNIX timestamp
Today is 2024/07/07 (Y/M/D)
We are at Taipei time zone

JSON

```
{
  "intent": "Order",
  "slots": {
    "Item": "咖啡",
    "Amount": "10",
    "Date": "2024-07-08",
    "Time": "15:00:00",
    "Timestamp": 1657548000,
    "Place": "台北動物園"
  }
}
```

Use code with caution.

要增加哪些額外的 prompt，其實與我們計畫讓我們的語音助理有多少能力有關。在前面的例子中，如果我們的語音助理只需要負責導航相關的工作，那麼在處理訂單相關的這些跟日期有關的處理，其實都是不需要的，所以，我們的 NLU 引擎，應該具備動態根據需求，調整使用不同的 prompt 能力。

實作基於 Gemini 的 NLU 引擎

一個基於 Gemini 的 NLU 引擎的實作如下。由於使用 Gemini 需要一把 API key，我們設計了可以傳入 API key 的介面。

```dart
import 'dart:convert';

import 'package:google_generative_ai/google_generative_ai.dart';
import 'package:meta/meta.dart';

import '../interface/nlu_engine.dart';

@immutable
class GeminiNluEngineError implements Exception {
  final String message;

  const GeminiNluEngineError(this.message);

  @override
  String toString() {
    return '${super.toString()} $message';
  }
}

class GeminiNluEngine extends NluEngine {
  final String apiKey;
  GeminiNluEngine({
    required this.apiKey,
  });

  @override
  Future<NluIntent> extractIntent(
    String utterance, {
    String? currentIntent,
    String? additionalRequirement,
  }) async {
    final model = GenerativeModel(
      model: 'gemini-1.5-flash-latest',
      apiKey: apiKey,
    );

    var prompt =
        'Extract the intent and the slots for the sentence:\n\n$utterance\n\n';
    prompt += "The output is in JSON without any formatting.";
    prompt += "Valid intents are including: ${availableIntents.join(',')}.";
    if (currentIntent != null) {
      prompt += "The current indent is $currentIntent.";
```

```
    }
    prompt += "Valid slots are including: ${availableSlots.join(',')}.";
    prompt += "If the intent is not in the valid intents, return null.";
    if (additionalRequirement != null) {
      prompt += additionalRequirement;
    }

    final content = [Content.text(prompt)];
    final response = await model.generateContent(content);
    final string = response.text;
    if (string == null) {
      throw const GeminiNluEngineError('Failed to generate response.');
    }
    final RegExp regex = RegExp(r'\{.*\}');
    final match = regex.firstMatch(string);
    if (match == null) {
      throw GeminiNluEngineError(
          'Failed to extract response from string $string.');
    }
    final jsonString = match.group(0);
    if (jsonString == null) {
      throw GeminiNluEngineError(
          'Failed to extract response from jsonString $jsonString.');
    }
    final map = json.decode(jsonString);
    final intent = NluIntent.fromMap(map);
    return intent;
  }
}
```

有了這個引擎之後，我們可以寫點單元測試，試試看是否可以產生符合我們預期的意圖：

```
import 'package:flutter_dialog/flutter_dialog.dart';
import 'package:flutter_test/flutter_test.dart';

const key = 'YOUR GEMINI API KEY';

void main() {
  test('Test intent for LeaveApplication 1', () async {
    final engine = GeminiNluEngine(apiKey: key);
    engine.availableIntents = {'LeaveApplication'};
    engine.availableSlots = {'Reason', 'Date'};
    final result = await engine.extractIntent('我生病了，今天必須要請假');
    expect(result.intent, 'LeaveApplication');
  });
}
```

實作基於 Gemini 的 NLG 引擎

我們會在幾種狀況下用到 NLG 引擎。或是我們在無法取得可以用來執行任務的意圖時，幫使用者產生一個回應，或是，在一個對話過程中，需要藉由 NLG 引擎的能力，產生符合當時情景的回答，像是點餐系統在結束時祝福使用者用餐愉快、或是掛號系統祝福早日康復等。所以，我們的輸入就會有兩種路徑。一個基於 Gemion 的 NLG 引擎的實作如下：

```dart
import 'package:google_generative_ai/google_generative_ai.dart';

import '../interface/nlg_engine.dart';

/// The Gemini based NLG (Natural Language Generating) engine.
class GeminiNlgEngine extends NlgEngine {
  /// The API key.
  final String apiKey;

  /// Creates a new instance.
  GeminiNlgEngine({
    required this.apiKey,
  });

  @override
  Future<String?> generateResponse(
    String utterance, {
    bool useDefaultPrompt = true,
    bool? preventMeaningLessMessage,
  }) async {
    final model = GenerativeModel(
      model: 'gemini-1.5-flash-latest',
      apiKey: apiKey,
    );

    var prompt = utterance;

    if (useDefaultPrompt) {
      prompt = 'Create a response for the sentence:\n\n$utterance\n\n';
      prompt += 'The response should be less than 30 words.\n';
      prompt += 'The response should not be another question.\n';
      prompt += 'The response should not contain emoji.\n';
      if (preventMeaningLessMessage == true) {
        prompt += 'If the sentence is lacking of context.'
            ' Just say you do not understand'
            ' using the same language as the incoming sentence.';
      }
    }
```

```
    final content = [Content.text(prompt)];
    final response = await model.generateContent(content);
    return response.text;
  }
}
```

我們通常需要告訴 Gemini 不要在回應中使用顏文字，不然 TTS 會把這些顏文字一起念出來，效果會很奇怪。

進一步的 NLU 分析與應用

對一個基本的語音助理來說，我們只需要 Intent 與 Slots，不過，像 Gemini 這樣的語言模型，其實可以幫我們分析出心情、語氣等資訊，以下是一個簡單的例子：

請分析以下句子 Intent 與 Slots，並分析當中的語氣與心情，然後產生 JSON 格式的回應

"我生病了，我想要請假"

我們要分析

- 禮貌程度
- 積極程度
- 快樂程度
- 憤怒程度
- 敵意程度

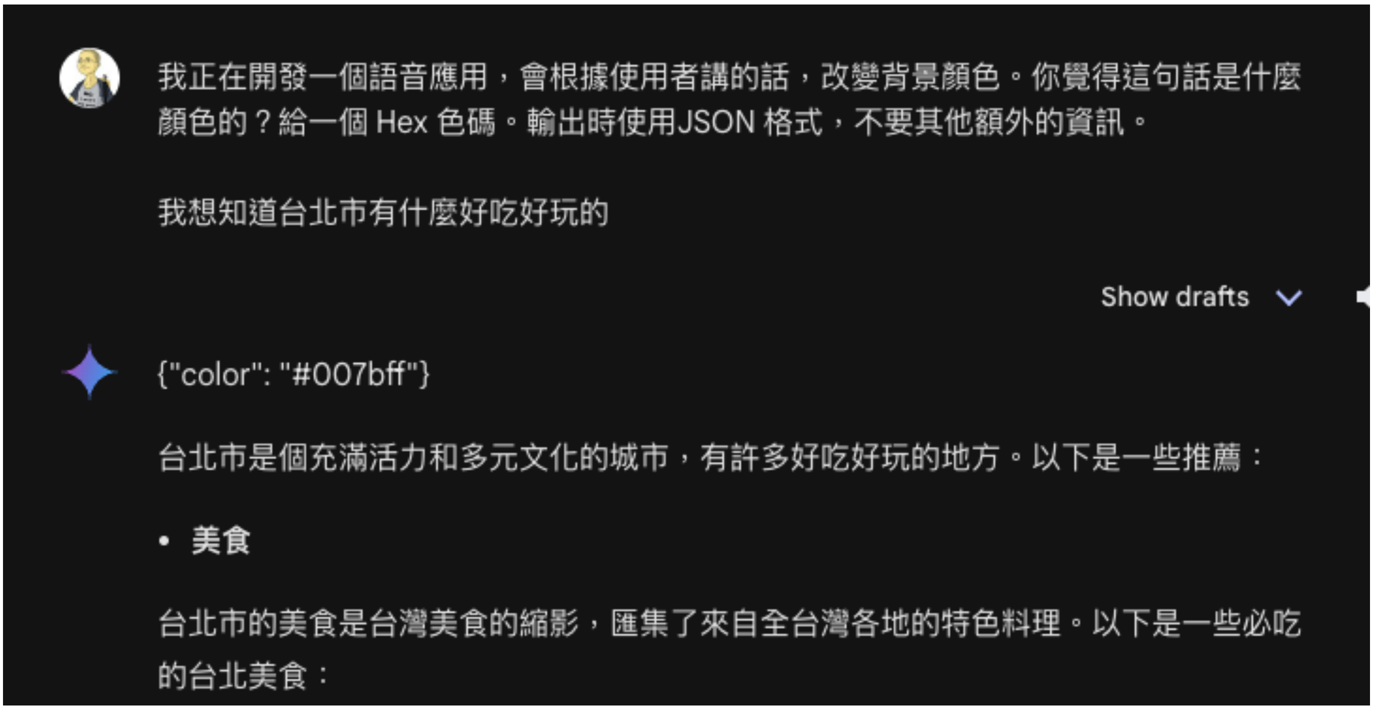
心情欄位叫做 emotion，當中的 key 都是英文，value 介於 0 - 100

Sure, here is the JSON analysis of the sentence "我生病了，我想要請假":

```
{
  "intent": "請假",
  "slots": {
    "reason": "生病"
  },
  "emotion": {
    "politeness": 0.8,
    "positivity": 0.2,
    "happiness": 0.2,
    "anger": 0.2,
    "hostility": 0.2
  }
```

如果我們希望我們的語音助理更擬人，也可以根據使用者的心情，給予不同的回應，像是發現使用者講的話實在太過沒禮貌，或許可以選擇不回應，或是告訴使用者不該這樣講話。這些都是我們在產品設計上可以用到的資訊。

甚至，我們可以根據使用者講的不同內容，請 Gemini 提供 UI 配色的建議。



我正在開發一個語音應用，會根據使用者講的話，改變背景顏色。你覺得這句話是什麼顏色的？給一個 Hex 色碼。輸出時使用JSON 格式，不要其他額外的資訊。

我想知道台北市有什麼好吃好玩的

{"color": "#007bff"}

台北市是個充滿活力和多元文化的城市，有許多好吃好玩的地方。以下是一些推薦：

- **美食**

台北市的美食是台灣美食的縮影，匯集了來自全台灣各地的特色料理。以下是一些必吃的台北美食：

ChatGPT

我們也可以使用 ChatGPT，實作 NLU 與 NLG 引擎。以下是以 ChatGPT 實作 NLG 引擎的範例：

```
import 'dart:convert';
import 'package:dart_openai/dart_openai.dart';
import '../interface/nlu_engine.dart';

class ChatgptNluEngine extends NluEngine {
  final String chatGptModel;

  /// Creates a new instance.
  ChatgptNluEngine({
    this.chatGptModel = "gpt-3.5-turbo-1106",
  });

  @override
  Future<bool> init() async => true;

  @override
  Future<NluIntent> extractIntent(String utterance,
      {String? currentIntent, String? additionalRequirement}) async {
    var systemPrompt = "return any message you are given as JSON.";
    if (currentIntent != null) {
```

```
      systemPrompt += "The current indent is $currentIntent.";
    }
    systemPrompt +=
        "Valid intents are including: ${availableIntents.join(',')}.";
    systemPrompt += "Valid slots are including: ${availableSlots.join(',')}.";
    systemPrompt += "Intents are in Pascal case.";
    systemPrompt += "If the intent is not in the valid intents, return null.";
    if (additionalRequirement != null) {
      systemPrompt += additionalRequirement;
    }

    final systemMessage = OpenAIChatCompletionChoiceMessageModel(content: [
      OpenAIChatCompletionChoiceMessageContentItemModel.text(systemPrompt),
    ], role: OpenAIChatMessageRole.assistant);

    var prompt =
        'Extract the intent and the slots for the sentence:\n\n$utterance\n\n';

    final userMessage = OpenAIChatCompletionChoiceMessageModel(content: [
      OpenAIChatCompletionChoiceMessageContentItemModel.text(prompt)
    ], role: OpenAIChatMessageRole.user);

    final requestMessages = [
      systemMessage,
      userMessage,
    ];

    final chatCompletion = await OpenAI.instance.chat.create(
      model: chatGptModel,
      responseFormat: {"type": "json_object"},
      seed: 6,
      messages: requestMessages,
      temperature: 0.2,
      maxTokens: 500,
    );

    final responseText =
        chatCompletion.choices.first.message.content?.first.text;
    if (responseText is! String) {
      throw Exception('Failed to extract message from chat completion.');
    }
    final map = json.decode(responseText);
    final intent = NluIntent.fromMap(map);
    return intent;
  }

  @override
  bool get isInitialized => true;
}
```

實作 ASR 與 TTS 引擎

ASR 引擎

我們選擇用 speech_to_text 實作我們的 ASR 引擎。可以看到，其實我們就只是簡單地把 speech_to_text 包裝一層。而使用 speech_to_text 時要注意一點，就是 `cancel()` 與 `stop()` 的差異，兩者雖然都會停止辨識，但是 `stop()` 會把目前辨識中的結果強制送到 `onResult` 這個 callback 中。

```
import 'package:speech_to_text/speech_to_text.dart' as stt;
import '../interface/asr_engine.dart';

const _pauseForSeconds = 3;

class PlatformAsrEngine extends AsrEngine {
  /// The default locale ID.
  var _localeId = 'en_US';
  var _isInitialized = false;

  @override
  Future<bool> init() async {
    if (_isInitialized) {
      return _isInitialized;
    }

    final speech = stt.SpeechToText();

    _isInitialized = await speech.initialize(
        onStatus: (status) {
          final map = {
            stt.SpeechToText.listeningStatus: AsrEngineState.listening,
            stt.SpeechToText.notListeningStatus: AsrEngineState.notListening,
            stt.SpeechToText.doneStatus: AsrEngineState.done,
          };
          var state = map[status] ?? AsrEngineState.notListening;
          onStatusChange?.call(state);
        },
        onError: (error) => onError?.call(error));
    if (!_isInitialized) {
      print("The user has denied the use of speech recognition.");
    }
    return _isInitialized;
```

```
  }

  @override
  Future<bool> startRecognition() async {
    if (!_isInitialized) {
      return false;
    }
    final speech = stt.SpeechToText();
    speech.listen(
        pauseFor: const Duration(seconds: _pauseForSeconds),
        localeId: _localeId,
        listenOptions: stt.SpeechListenOptions(
          listenMode: stt.ListenMode.search,
          partialResults: true,
          onDevice: false,
          cancelOnError: true,
        ),
        onResult: (result) {
          final words = result.recognizedWords;
          final isFinal = result.finalResult;
          onResult?.call(words, isFinal);
        });
    return true;
  }

  @override
  Future<bool> stopRecognition() async {
    if (!_isInitialized) {
      return false;
    }
    final speech = stt.SpeechToText();
    await speech.cancel();
    return true;
  }

  @override
  Future<void> setLanguage(String language) async {
    _localeId = language;
  }

  @override
  bool get isInitialized => _isInitialized;
}
```

TTS 引擎

至於 TTS，我們也只是把 flutter_tts 簡單包裝一層。在這邊，我們希望 `await playPrompt(prompt)` 的時候，可以等整句話都播放完畢再執行下一行，但 flutter_tts 的行為並不是這樣，所以額外增加了一個 Completer 等待。

```dart
import 'dart:async';
import 'package:flutter_tts/flutter_tts.dart';
import '../interface/tts_engine.dart';

/// Please refer to https://pub.dev/packages/flutter_tts to update your Android
/// manifest and iOS configuration.
class PlatformTtsEngine extends TtsEngine {
  final flutterTTs = FlutterTts();
  Completer? _ttsCompleter;

  PlatformTtsEngine() {
    flutterTTs.setStartHandler(() {
      onStart?.call();
    });
    flutterTTs.setCompletionHandler(() {
      _ttsCompleter?.complete();
      _ttsCompleter = null;
      onComplete?.call();
    });
    flutterTTs.setProgressHandler((text, startOffset, endOffset, word) {
      onProgress?.call(text, startOffset, endOffset, word);
    });
    flutterTTs.setErrorHandler((msg) {
      _ttsCompleter?.complete();
      _ttsCompleter = null;
      onError?.call(msg);
    });
    flutterTTs.setCancelHandler(() {
      _ttsCompleter?.complete();
      _ttsCompleter = null;
      onCancel?.call();
    });
    flutterTTs.setPauseHandler(() {
      print('TTS setPauseHandler');
      onPause?.call();
    });
    flutterTTs.setContinueHandler(() {
      print('TTS setContinueHandler');
      onContinue?.call();
    });
  }
```

```
  @override
  Future<void> playPrompt(String prompt) async {
    await flutterTTs.speak(prompt);
    var ttsCompleter = Completer();
    _ttsCompleter = ttsCompleter;
    await ttsCompleter.future;
  }

  @override
  Future<void> stopPlaying() async {
    await flutterTTs.stop();
    _ttsCompleter?.complete();
    _ttsCompleter = null;
  }

  @override
  Future<void> setLanguage(String language) async {
    await flutterTTs.setLanguage(language);
  }

  @override
  Future<void> setPitch(double pitch) async {
    await flutterTTs.setPitch(pitch);
  }

  @override
  Future<void> setSpeechRate(double rate) async {
    await flutterTTs.setSpeechRate(rate);
  }

  @override
  Future<void> setVolume(double volume) async {
    await flutterTTs.setVolume(volume);
  }

  @override
  Future<void> setVoice(Map<String, String> voice) async {
    flutterTTs.setVoice(voice);
  }
}
```

Mock

上面的 ASR 與 NLU 引擎，由於用到了平台的功能，所以沒辦法在單元測試使用。為了測試方面，我們可以實作 Mock 的 ASR 與 NLU 引擎。基本上就是基於 `AsrEngine` 與 `TtsEngine` 的介面，但是不做任何的實作。

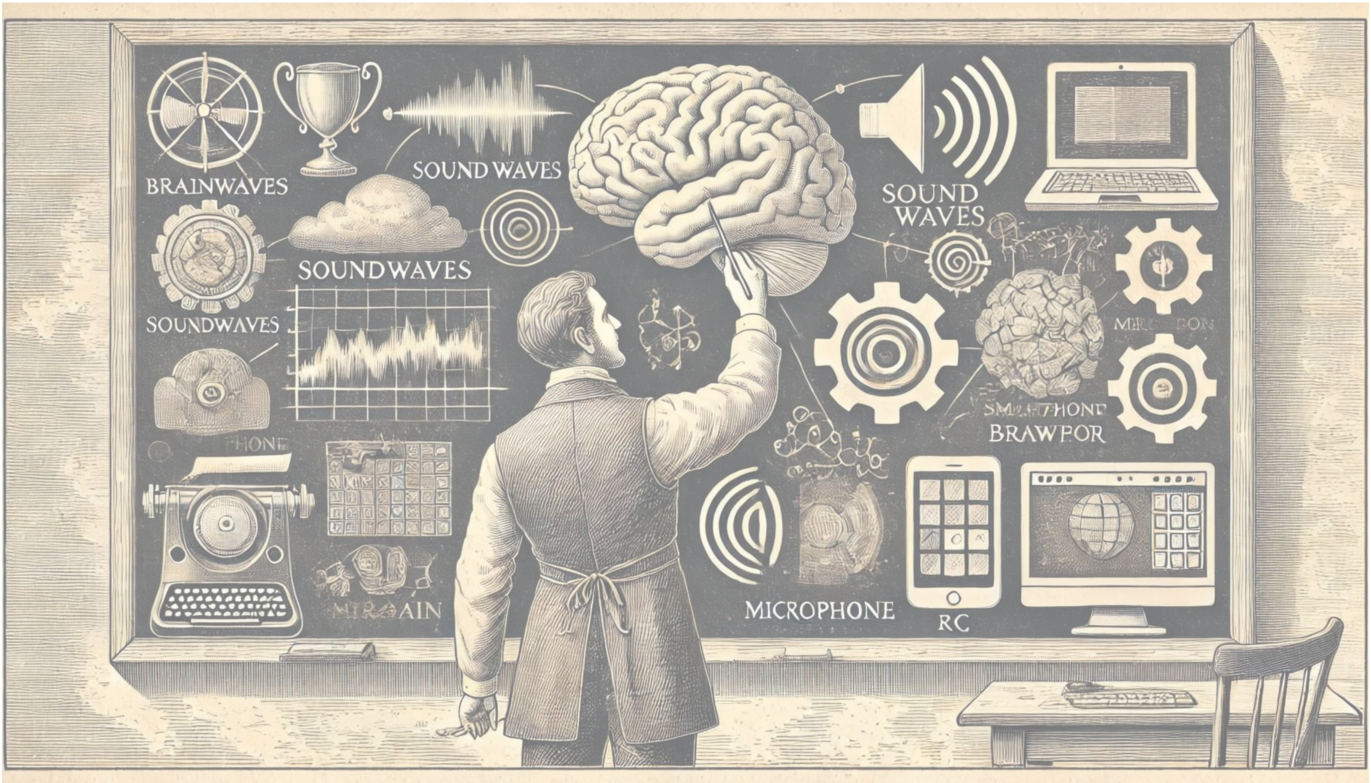
```dart
import '../interface/asr_engine.dart';

/// A mock ASR (Automatic Speech Recognition) engine for testing.
class MockAsrEngine extends AsrEngine {
  @override
  Future<bool> init() async {
    return true;
  }

  @override
  bool get isInitialized => true;

  @override
  Future<bool> startRecognition() async {
    return true;
  }

  @override
  Future<bool> stopRecognition() async {
    return true;
  }

  @override
  Future<void> setLanguage(String language) async {}
}
```

```dart
import '../interface/tts_engine.dart';

/// A mock TTS (Text to Speech) engine for testing.
class MockTtsEngine extends TtsEngine {
  @override
  Future<void> playPrompt(String prompt) async {
    print('MockTtsEngine play $prompt');
    await Future.delayed(const Duration(milliseconds: 200));
    onComplete?.call();
  }

  @override
  Future<void> setLanguage(String language) async {}

  @override
```

```
  Future<void> setPitch(double pitch) async {}

  @override
  Future<void> setSpeechRate(double rate) async {}

  @override
  Future<void> setVoice(Map<String, String> voice) async {}

  @override
  Future<void> setVolume(double volume) async {}

  @override
  Future<void> stopPlaying() async {
    onCancel?.call();
  }
}
```

在後續章節〈實作完整的對話引擎〉中，會說明如何使用這些 Mock 的 ASR 與 TTS 引擎。

設計語音互動流程



在透過 NLU 引擎了解使用者的意圖之後，我們就可以開始嘗試達成對應的任務。我們的對話引擎平時維持一份意圖與流程之間的表格，收到意圖之後，就從表格中取出這段流程執行。

語音流程的輸入與輸出

一段語音互動的流程，可能會被稱做 Dialog，在英文中，Dialog 與 Conversation 的意義不太一樣，雖然在中文中都叫做對話，但 Dialog 更具備一種事先安排好，是在劇本當中安排好的兩個角色之間的談話，比起 Conversation，更貼近語音助理要完成任務的過程。也可能稱作 VUI flow，VUI 是 Voice User Interface 的縮寫。以下我們統稱 VUI flow。

一段 VUI flow 會以一個意圖開始，在設計上，我們就不太能夠讓多段不同的 VUI flow 對應到同一個意圖上，而是一個 VUI flow，會根據傳入的 Slots 以及當前的系統狀態，走進同一個 VUI flow 的不同分支。而要完成任務，往往會需要一些必須的 Slot，像是導航就會需要知道目的地，訂餐就會需要品項與數量等。

同時，也可能會有一些非必要的 Slot，但會決定我們如何執行任務，像是導航的語音命令中，可能會多出一個途經點或是迴避點的選項，使用者可能會說「導航到台北動物園，中途經過加油站，但是想辦法避開塞車路段，不要走國道」這樣複雜的句子。遇到必要的 Slot，我們大概就要進入多輪對話追問使用者，一段 VUI flow，可說就是在蒐集必要的 Slot 的過程。

之後，這段流程往往需要知道一些外部狀態，像是執行前述的導航流程時，我們可能就會想知道目前系統是否已經在導航中，向使用者確認是否需要停止目前的導航工作，如果確定要終止目前的導航，我們才會執行新的導航。

這種向外部詢問狀態，或是要求外部執行工作的呼叫，通常叫做 System Call。這些 System Call 通常也是語音開發者與 App 開發者之間溝通的介面，語音開發者專注在 VUI flow 的設計與開發，不用關心實際上導航這部份的細節，而 App 開發者也只要專心在這些 System Call 的實作。最後，無論成功或失敗，我們可以回應使用者一段 TTS，最後結束對話。TTS 可以是一段固定的文字，也可以交給 NLG 引擎產生，另外，也可能用一小段音效—像是嗶嗶聲—代替。

所以，我們可以分析出，一段語音互動流程，有著以下的輸入與輸出：

輸入

- Intent
- Slots (必要與非必要)
- 獲取外部狀態的 System Call

輸出

- 執行工作的 System Call
- TTS 回應，或是用一些替代的音效
- 該結束對話，還是該進入下一輪對話，而下一輪對話該是哪個 VUI flow

繪製 VUI flow 流程圖

根據上述的輸入與輸出，我們通朝用 UML 當中的 Activity Diagram，表現對話過程中會有哪些分支。以下以上述的導航流程為例：

`c94ea9fa-d80e-42cb-a5eb-c5a302d04c1f`

在設計一個 VUI flow 時，我們通常也會附上如何啓動這個流程、滿足可以車取出意圖的例句（Utterance），方便日後的測試以及功能驗證。像是：

- 開始導航
- 我想要去台北動物園
- 帶我去動物園，不要上高架橋…

在上圖中，可以注意到，其實這張圖是多個 VUI flow 組成的，中間有一段確認是否要終止導航的流程，就是以「確認」、「取消」等意圖，開啓一段新的 VUI flow。

這個時候，通常會預期只會收到在限制範圍內的意圖，但使用者的行為可能千奇百怪—舉另外一個例子，如果是一套點餐系統，當使用者說「我要點餐」時，我們可能會回應「您想點什麼」，並且預期收到「我要一碗牛肉麵」這樣的回答，但使用者可能會說「有什麼可以點的」，這時候我們可能選擇回覆「我們不理解你的意圖」，或是必須再設計一條分支，念出一些熱門的餐點，然後再問一次想點什麼餐點。流程應該怎麼設計，最後還是要回歸使用者到底怎麼使用，以及團隊的使用者研究的結果。

VUI flow 的介面

在實際進入撰寫 VUI flow 之前，我們先設計一個屬於所有 VUI flow 的介面。

```
abstract class VuiFlow {
  Future<void> handle(NluIntent intent);

  Future<void> cancel() async {
    delegate = null;
  }

  VuiFlowDelegate? delegate;
  String get intent => '';
  List<String> get slots => [];
  String? get additionalNluPrompt => null;
}
```

這個介面分成三個部分：

- 這個 VUI flow 實際上的工作：當我們要使用這個 VUI flow 時，會去呼叫 `handle` 這個 method
- 這個 VUI flow 所需要的外部相依：集中在 delegate 中
- 其他的 metadata：包括什麼意圖會進入這個 VUI flow，期待收到哪些 Slot，以及是否需要額外的 LLM Prompt 協助抽取 Slot

在 delegate 中，我們定義了一個 VUI flow 怎麼在中途使用 TTS 引擎與 NLG 引擎，而如果他需要進入另外一輪的對話，可以用 `onSettingCurrentVuiFlow` 指定下一個 VUI flow 是什麼，並且用 `onStartingAsr` 重新開始 ASR 識別。另外，就是一個用來結束對話的 `onEndingConversation` 。

```
abstract class VuiFlowDelegate {
  Future<void> onPlayingPrompt(String prompt);
  Future<void> onSettingCurrentVuiFlow(VuiFlow? vuiFlow);
  Future<void> onStartingAsr();
  Future<void> onEndingConversation();
  Future<String?> onGeneratingResponse(String utterance,
      {bool useDefaultPrompt = true});
}
```

另外，使用者可能隨時取消目前正在進行中的 VUI flow，像是在執行到一半的時候，就按下退出語音助理的按鈕，不但直接關閉 ASR 識別與 TTS 語音，也需要取消正在 `handle` method 中執行的工作。如果要做完整的取消的話，那每個 VUI flow 中，都需要一個叫做 cancelled 的變數，當這個變數為 true 時，`handle` 裡頭每一行，都要檢查 cancelled 是不是已經變成 true，如果是，就要退出。

在這邊先用了一個偷懶的辦法：只要把 delegate 設成 null 之後，這個 VUI flow 就不會有對外的聯繫。也不會繼續有 ASR 識別與 TTS 語音，不過 `handle` 其實還是會繼續往下執行。

實作語音互動流程

我們現在來設計一個個人用的語音助理，可以在我生病在床，一點都不想爬起床的時候，只要用簡單的語音指令，就寫出一篇洋洋灑灑的假單，然後用 Teams 傳給老闆。我們先簡單設計這個助理的流程：

517d5a47-e7f9-465a-89c0-50a176faedf8

定義 Metadata

我們首先繼承 VuiFlow，寫一個自己的 `LeaveApplicationVuiFlow`。由於我們希望 NLU 引擎幫我們抽取出 `LeaveApplication` 這個請假意圖，以及 `Reason` 與 `Date` 兩個 Slots，所以我們要實作 `intent` 與 `slots` 這兩個屬性。

```
class LeaveApplicationVuiFlow extends VuiFlow {
  @override
  String get intent => 'LeaveApplication';

  @override
  List<String> get slots => ['Reason', 'Date'];
}
```

保留必要的 Slot

我們已經預期需要透過多輪對話，取得日期與事由，所以我們用 date 與 reason 這兩個變數來保留這兩個 Slot 的值。然後，我們希望這個多輪對話有個上限，當對話進行了好幾次，都沒辦法取得日期與事由時，就結束對話，因此，我們加了一個叫做 errorCount 的變數，保存失敗的次數。

```
  String? date;
  String? reason;
  var errorCount = 0;
```

實作取得必要的 Slot 的流程

我們現在要來開始實作 `handle` method：

```
@override
Future<void> handle(NluIntent intent) async {
  Future<void> handleMaxError() async {
    await delegate?.onSettingCurrentVuiFlow(null);
    await delegate?.onPlayingPrompt('很抱歉，錯誤次數太多');
    await delegate?.onEndingConversation();
  }

  var date = intent.slots['Date'] ?? this.date;
  var reason = intent.slots['Reason'] ?? this.reason;
  var errorCount = this.errorCount;
  if (date == null || reason == null) {
    final prompt = () {
      if (date == null && reason == null) {
        return '請問您要請假的日期與事由？';
      }
      if (date == null) {
        return '請問您要請假的日期？';
      }
      if (reason == null) {
        return '請問您要請假的事由？';
      }
      return '';
    }();

    errorCount += 1;
    if (errorCount >= _maxErrorCount) {
      await handleMaxError();
      return;
    }
    await delegate?.onPlayingPrompt(prompt);
    await Future.delayed(const Duration(microseconds: 500));
    final vuiFlow = LeaveApplicationVuiFlow(
        onMakingLeaveApplication: onMakingLeaveApplication)
      ..date = date
      ..reason = reason
      ..errorCount = errorCount;
    await delegate?.onSettingCurrentVuiFlow(vuiFlow);
    await delegate?.onStartingAsr();
    return;
  }
  .....
}
```

在這一部分，我們首先確認，是不是在 intent 中，以及在 LeaveApplication 物件上，已經有了 date 與 errorCount 加一，數量到了一定程度後，就走入 `handleMaxError`。不然，就把 LeaveApplicationVuiFlow 拷貝一份（先忽略 onMakingLeaveApplication），然後把現在的 date、reason 與 errorCount 複製上去，然後告訴 delegate，這就是下一輪對話使用的 VUI flow，收到任何意圖就要傳到這個 VUI flow，然後重新開始 ASR 識別。

使用 NLG 引擎

```
    final promptForApplication =
        '幫我寫一份關於我要請假的短文，語氣嚴謹而且禮貌，大量使用成語，並且引用唐詩，內容大約兩百中文字，不分段'
        '，請假日期是 $date，事由是 $reason。不要問我其他額外的問題，像是姓名職位等，也不要出現emoji。';
    final text = await delegate?.onGeneratingResponse(
          promptForApplication,
          useDefaultPrompt: false,
        ) ??
        '';
    final promptForReply =
        "如果一個人發生了 $reason 的狀況，你會怎麼祝他順利呢？只要一句話就好了，不要出現 emoji。";
    final greet = await delegate?.onGeneratingResponse(
          promptForReply,
          useDefaultPrompt: false,
        ) ??
        '';
    await delegate?.onSettingCurrentVuiFlow(null);
    final result = await onMakingLeaveApplication(reason, date, text);
    if (result) {
      await delegate?.onPlayingPrompt('好的，正在幫您請假！$greet');
    } else {
      await delegate?.onPlayingPrompt('很抱歉，無法幫你請假！但我想告訴你，$greet');
    }
    await Future.delayed(const Duration(microseconds: 500));
    await delegate?.onEndingConversation();
```

設計 System Call

前面所出現的 `onMakingLeaveApplication`，就是這個 VUI flow 用來呼叫外部的 System Call，所以我們只要再這個 class 當中定義一個匿名函式

```
class LeaveApplicationVuiFlow extends VuiFlow {
  Future<bool> Function(
    String reason,
    String date,
    String text,
  ) onMakingLeaveApplication;

  LeaveApplicationVuiFlow({required this.onMakingLeaveApplication});
  ...
}
```

單元測試

要對前面的實作進行單元測試，首先我們 mock 一個 `VuiFlowDelegate`，在這個 mock 中設置好一些特定的條件，像是我們假設 NLG 引擎會有哪些回覆，以及會保存最後一次 TTS 的內容，或是會保存最後一次的 VuiFlow 是什麼。

```
class LeaveApplicationMockDelegate extends VuiFlowDelegate {
  final completer = Completer();

  @override
  Future<void> onEndingConversation() async {
    completer.complete();
  }

  @override
  Future<String?> onGeneratingResponse(String utterance,
      {bool useDefaultPrompt = true}) async {
    if (utterance.startsWith('幫我寫一份關於我要請假的短文')) {
      return '我要請假';
    } else if (utterance.startsWith('如果一個人發生了')) {
      return '祝你早日康復';
    }
    return '';
  }

  String? lastPrompt;

  @override
  Future<void> onPlayingPrompt(String prompt) async {
    lastPrompt = prompt;
  }

  VuiFlow? currentVuiFlow;
```

```
  @override
  Future<void> onSettingCurrentVuiFlow(VuiFlow? vuiFlow) async {
    currentVuiFlow = vuiFlow;
  }

  @override
  Future<void> onStartingAsr() async {}
}
```

接著就可以寫測試了：

```
  test('Test Leave Application VUI flow - with slots', () async {
    final delegate = LeaveApplicationMockDelegate();
    var systemCallCalled = false;
    final vuiFlow = LeaveApplicationVuiFlow(
        onMakingLeaveApplication: (reason, date, text) async {
      expect(reason, '生病');
      expect(date, '今天');
      expect(text, '我要請假');
      systemCallCalled = true;
      return true;
    })
      ..delegate = delegate;
    final intent = NluIntent(
        intent: 'LeaveApplication', slots: {'Reason': '生病', 'Date': '今天'});
    vuiFlow.handle(intent);
    await delegate.completer.future;
    expect(systemCallCalled, isTrue);
    expect(delegate.lastPrompt, isNotNull);
    expect(delegate.lastPrompt?.contains('祝你早日康復'), isTrue);
  });
```

這個 test case 中，我們只測試了一種狀況，就是使用者一開始就說清楚了請假的事由與日期，然後看看最後的 TTS 對不對，以及最後請假用的 System Call 是否被呼叫了。實務上，這樣的測試只考慮的一個情境，沒有考慮少了某個 slot ，或是一開始就已經在上一輪對話中得到了 slot 的狀況，還需要更多的 case 才能完全覆蓋——在這些 case 中，則需要 assert 是否提供了用做下一輪對話的 VuiFlow 物件，但是因為篇幅的關係，不在此贅述。

實際使用 System Call

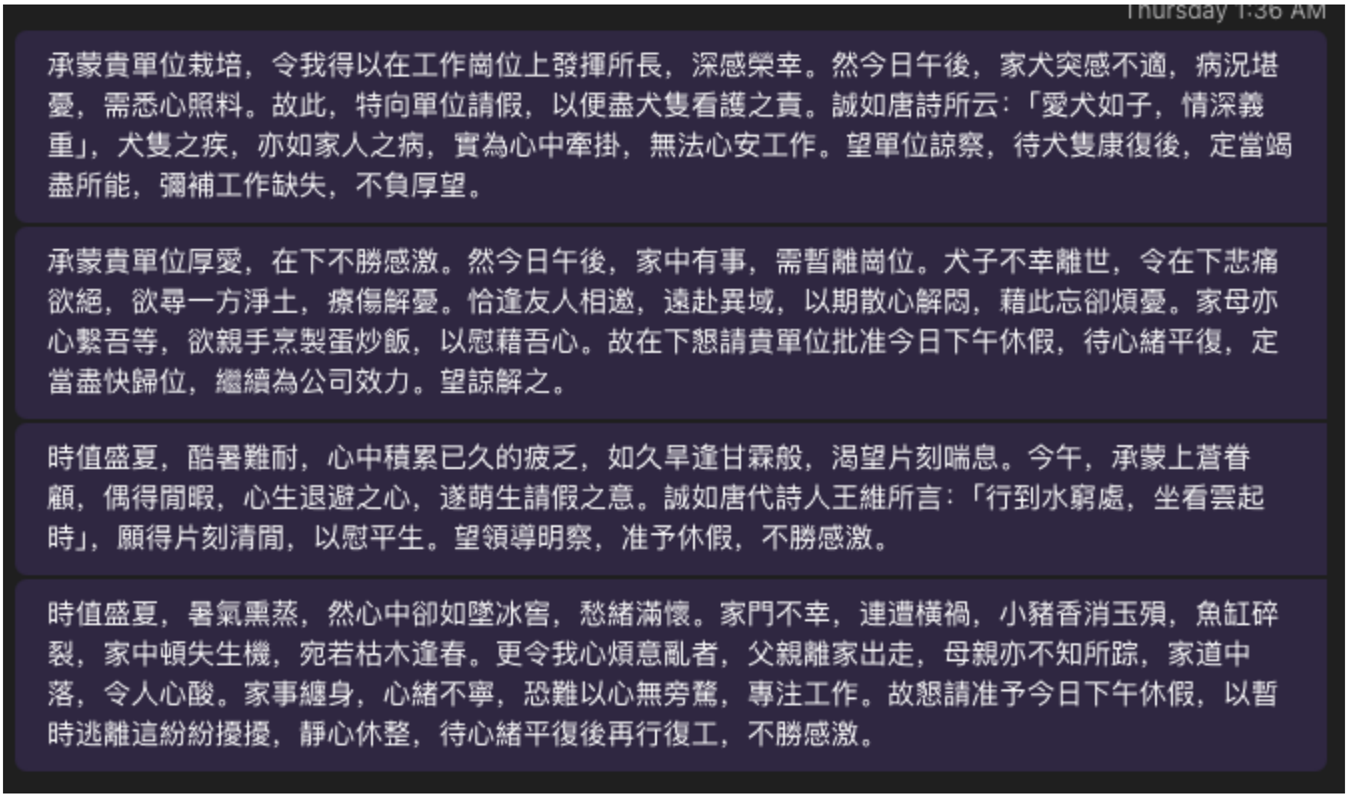
前面提到，我們想把這份文情並茂的假單，送到老闆的 Teams 上，我們去查了微軟的文件，發現只要透過特定的 URL[1]，就可以開啓新的對話，所以我們也可以小段用來開啓這個 URL 的程式：

```
class TeamsHelper {
  /// See https://learn.microsoft.com/en-us/microsoftteams/platform/concepts/build-and-test/deep-
link-teams#deep-link-to-start-a-new-chatœ
  static sendMessage({
    required String user,
    required String topicName,
    required String message,
  }) {
    final encodedUser = Uri.encodeComponent(user);
    final encodedTopicName = Uri.encodeComponent(topicName);
    final encodedMessage = Uri.encodeComponent(message);

    final url = Uri.parse("https://teams.microsoft.com/l/chat/0/0?"
        "users=$encodedUser&"
        "topicName=$encodedTopicName&"
        "message=$encodedMessage");
    launchUrl(url,
        // mode: LaunchMode.externalApplication
    );
  }
}
```

然後在建立 LeaveApplicationVuiFlow 的時候，傳入我們的實作

```
  LeaveApplicationVuiFlow(
        onMakingLeaveApplication: (reason, date, text) async {
      TeamsHelper.sendMessage(
        user: "myboss@mycompany.com",
        topicName: "Leave Application",
        message: text,
      );

      return true;
    })
```

我們可以來看看 Gemini 幫我們產生了哪些精彩的請假事由：



Thursday 1:36 AM

承蒙貴單位栽培，令我得以在工作崗位上發揮所長，深感榮幸。然今日午後，家犬突感不適，病況堪憂，需悉心照料。故此，特向單位請假，以便盡犬隻看護之責。誠如唐詩所云：「愛犬如子，情深義重」，犬隻之疾，亦如家人之病，實為心中牽掛，無法心安工作。望單位諒察，待犬隻康復後，定當竭盡所能，彌補工作缺失，不負厚望。

承蒙貴單位厚愛，在下不勝感激。然今日午後，家中有事，需暫離崗位。犬子不幸離世，令在下悲痛欲絕，欲尋一方淨土，療傷解憂。恰逢友人相邀，遠赴異域，以期散心解悶，藉此忘卻煩憂。家母亦心繫吾等，欲親手烹製蛋炒飯，以慰藉吾心。故在下懇請貴單位批准今日下午休假，待心緒平復，定當盡快歸位，繼續為公司效力。望諒解之。

時值盛夏，酷暑難耐，心中積累已久的疲乏，如久旱逢甘霖般，渴望片刻喘息。今午，承蒙上蒼眷顧，偶得閒暇，心生退避之心，遂萌生請假之意。誠如唐代詩人王維所言：「行到水窮處，坐看雲起時」，願得片刻清閒，以慰平生。望領導明察，准予休假，不勝感激。

時值盛夏，暑氣熏蒸，然心中卻如墜冰窖，愁緒滿懷。家門不幸，連遭橫禍，小豬香消玉殞，魚缸碎裂，家中頓失生機，宛若枯木逢春。更令我心煩意亂者，父親離家出走，母親亦不知所踪，家道中落，令人心酸。家事纏身，心緒不寧，恐難以心無旁騖，專注工作。故懇請准予今日下午休假，以暫時逃離這紛紛擾擾，靜心休整，待心緒平復後再行復工，不勝感激。

---

1. https://learn.microsoft.com/en-us/microsoftteams/platform/concepts/build-and-test/deep-link-teams#deep-link-to-start-a-new-chat ↩

實作完整的對話引擎

我們已經逐一實作了 ASR、NLU、NLG、TTS 引擎以及對話流程，現在，就可以根據前章〈語音助理的基本組成〉所述，將各個元件串接起來，形成一個完整的對話引擎。

初始化

在可以使用這個對話引擎之前，需要先初始化 ASR 引擎。我們於是在對話引擎中加入了一個 `init` method，另外，寫了一個 `_emit` method，用來發送對話引擎的狀態。

```
class DialogEngine implements VuiFlowDelegate {
  void _emit(DialogEngineState state) {
    _state = state;
    _stateStream.add(state);
  }

  Future<bool> init() async => await asrEngine.init();
}
```

建立意圖與對話流程的對應表格

對話引擎內部會維護一份叫做 _vuiFlowMap 的表格，之後，當對話引擎收到某個意圖時，就可以去這個表格中尋找是否有對應的對話流程。而我們也設計了一個叫做 `registerFlows` 的 method，可以將新的對話流程加到這個表格中。

```
class DialogEngine implements VuiFlowDelegate {
...

  Map<String, VuiFlow> _vuiFlowMap = {};
  VuiFlow? _currentVuiFlow;
...

  /// Resets the list of [VuiFlow].
  void resetFlows() {
    _vuiFlowMap = {};
    _updateIntents();
```

```
  }

  /// Register a list of [VuiFlow].
  void registerFlows(List<VuiFlow> flows) {
    for (final flow in flows) {
      _vuiFlowMap[flow.intent] = flow;
      flow.delegate = this;
    }
    _updateIntents();
  }

  void _updateIntents() {
    var intents = <String>{
      'Confirm',
      'Acknowledge',
      'Agree',
      'Cancel',
      'Reject',
      'Disagree',
      'Deny',
    };
    var slots = <String>{};

    for (final key in _vuiFlowMap.keys) {
      intents.add(key);
      final flow = _vuiFlowMap[key];
      if (flow == null) {
        continue;
      }
      slots.addAll(flow.slots);
    }
    nluEngine.availableIntents = intents;
    nluEngine.availableSlots = slots;
  }

...
}
```

當有新的對話流程被加入到對話引擎時，我們會呼叫 `_updateIntents` method，這個 method 會將所有的意圖與 Slot 都更新到 NLU 引擎中，這樣，NLU 引擎就可以知道目前有哪些意圖與 Slot 可以被辨識。

實作 VUI flow Delegate

每個 VUI flow 的 delegate，其實就是我們的對話引擎。所以我們需要實作 `VuiFlowDelegate` 這個 interface。

```
class DialogEngine implements VuiFlowDelegate {
...
  @override
  Future<void> onEndingConversation() async {
    await stop();
  }

  @override
  Future<String?> onGeneratingResponse(
    String utterance, {
    bool useDefaultPrompt = true,
  }) async {
    return await nlgEngine.generateResponse(
      utterance,
      useDefaultPrompt: useDefaultPrompt,
    );
  }

  @override
  Future<void> onPlayingPrompt(String prompt) async {
    await ttsEngine.stopPlaying();
    _emit(DialogEnginePlayingTts(prompt: prompt));
    await ttsEngine.playPrompt(prompt);
  }

  @override
  Future<void> onSettingCurrentVuiFlow(VuiFlow? vuiFlow) async {
    vuiFlow?.delegate = this;
    _currentVuiFlow = vuiFlow;
  }

  @override
  Future<void> onStartingAsr() async {
    await start(clearCurrentVuiFlow: false);
  }
}
```

然後 `start` 與 `stop` 的實作如下。大概就是開始以及停止 ASR/NLU 引擎的相關工作。可以想像如果 NLU 或 NLG 引擎正在運作中，也應該停止，不過這邊就先偷懶不寫。

```
Future<bool> start({
  clearCurrentVuiFlow = true,
}) async {
  if (!asrEngine.isInitialized) {
    return false;
  }

  if (clearCurrentVuiFlow) {
    _currentVuiFlow?.cancel();
    _currentVuiFlow = null;
  }
  await ttsEngine.stopPlaying();
  await asrEngine.stopRecognition();
  await asrEngine.startRecognition();
  _emit(DialogEngineListening(asrResult: ''));
  return true;
}

Future<bool> stop() async {
  if (!asrEngine.isInitialized) {
    return false;
  }

  _currentVuiFlow?.cancel();
  _currentVuiFlow = null;
  await ttsEngine.stopPlaying();
  await asrEngine.stopRecognition();
  _emit(DialogEngineIdling());
  return true;
}
```

連接 ASR 與 NLU 引擎

在開始建立這個對話引擎的時候，我們就開始監聽 ASR 引擎的狀態。如果 ASR 引擎還在識別當中，我們就透過更新狀態，反應目前的辨識結果，如果辨識完成，就開始讓 NLU 引擎分析意圖（也就是 `handleInput` 這段）。這邊有一小段邏輯，在於處理使用者完全不說話的狀態—如果完全沒有識別結果，而且不在對話流程中，就會直接進入閒置狀態。

```
DialogEngine({
  required this.asrEngine,
  required this.ttsEngine,
  required this.nluEngine,
  required this.nlgEngine,
}) {
```

```
  asrEngine.onResult = (result, isFinal) async {
    if (!isFinal) {
      _emit(DialogEngineListening(asrResult: result));
    } else {
      await handleInput(result);
    }
  };
  asrEngine.onError = (error) async {
    await ttsEngine.stopPlaying();
    await asrEngine.stopRecognition();
    _emit(DialogEngineIdling());
  };
  asrEngine.onStatusChange = (state) async {
    if (state == AsrEngineState.listening) {
      return;
    }
    final current = this.state;
    if (current is DialogEngineCompleteListening) {
      return;
    }
    if (current is DialogEngineListening) {
      if (current.asrResult != '') {
        return;
      }
    }

    if (_currentVuiFlow != null) {
      final intent = NluIntent(intent: '', slots: {});
      await _currentVuiFlow
          ?.handle(intent)
          .timeout(const Duration(seconds: 60));
      return;
    }

    if (current is DialogEngineListening) {
      _emit(DialogEngineIdling());
    }
  };
}
```

至於 `handleInput` 的內容如下。我們平常會暴露 `handleInput`，因此，外部可以不用真的透過 ASR 語音錄音，而是直接對這個 method 傳入一段文字，就可以啓動 NLU 引擎以及對話流程，進行整合測試。

```
String fallbackErrorMessage = 'Sorry, I do not understand for now.';

Future handleInput(String input) async {
  _emit(DialogEngineCompleteListening(asrResult: input));
```

```
    await ttsEngine.stopPlaying();
    await asrEngine.stopRecognition();

    try {
      final additionalPrompt = _collectionAdditionalNluPrompt();
      final intent = await nluEngine.extractIntent(
        input,
        currentIntent: _currentVuiFlow?.intent,
        additionalRequirement: additionalPrompt,
      );
      if (_currentVuiFlow != null) {
        await _currentVuiFlow
            ?.handle(intent)
            .timeout(const Duration(seconds: 60));
        return;
      }
      final flow = _vuiFlowMap[intent.intent];
      if (flow != null) {
        await flow.handle(intent).timeout(const Duration(seconds: 60));
        return;
      }
      final prompt =
          await nlgEngine.generateResponse(input) ?? fallbackErrorMessage;
      await onPlayingPrompt(prompt);
      await stop();
    } catch (e) {
      _currentVuiFlow = null;
      await onPlayingPrompt(fallbackErrorMessage);
      await stop();
    }
```

到這裡，我們已經完成了我們的對話引擎。

整合測試

在我們急著把這個對話引擎整合到我們的應用程式之前，我們可以先寫一些整合測試，來確保這個對話引擎的功能是正確的。這邊我們可以使用 `flutter_test` 這個套件，來寫一些測試。測試的內容是，我們先用「我想請假」，啟動 `LeaveApplication` 意圖，但由於欠缺日期與事由，所以會進入多輪對話，然後我們再輸入「明天下午我想出去玩」，這樣就可以完成這個對話流程，最後驗證是否正確到了 System Call。為了避免因為網路斷線等問題，造成測試卡住，我們預期二十秒之內要完成測試。

```
test('Test Engine with Leave Application', () async {
  final engine = DialogEngine(
    asrEngine: MockAsrEngine(),
    ttsEngine: MockTtsEngine(),
    nluEngine: GeminiNluEngine(apiKey: key),
    nlgEngine: GeminiNlgEngine(apiKey: key),
  );

  final completer = Completer();
  var systemCallCalled = false;
  engine.registerFlows([
    LeaveApplicationVuiFlow(
        onMakingLeaveApplication: (reason, date, text) async {
      expect(date, '明天下午');
      expect(reason, '出去玩');
      systemCallCalled = true;
      completer.complete();
      return true;
    })
  ]);
  await engine.init();
  await engine.handleInput('我想請假');
  await Future.delayed(const Duration(seconds: 5));
  await engine.handleInput('明天下午我想出去玩');
  await completer.future.timeout(const Duration(seconds: 20));
  expect(systemCallCalled, isTrue);
});
```

在這個整合測試當中，我們也示範了怎樣從外部建立我們的對話引擎的作法。

- 首先建立對話引擎的 instance，當中選擇了我們要使用的 ASR、NLU、NLG、TTS 引擎
- 對引擎註冊我們要使用的 VUI flow。
- 初始化對話引擎

之後，我們透過 `handleInput` 測試，在實際的 app 中，我們則用 `start` method 來啓動對話引擎。

確認與取消

會進入多輪對話的場合，除了想辦法蒐集完成一項任務所需要的所有 Slot 外（做法則向前面提到的，重複複製同一個 VuiFlow 子類別，循環使用），就是在要進行工作之前，與使用者再次確認。直到使用者說了「對」、「好」、「沒錯」等句子，才會實際呼叫 System Call。

確認

因為向使用者確認也是一段對話，我們也可以把這段對話包裝在 `VuiFlow` 中。

```
const _maxErrorCount = 5;

class ConfirmVuiFlow extends VuiFlow {
  final VuiFlow positiveFlow;
  final VuiFlow negativeFlow;
  var errorCount = 0;

  ConfirmVuiFlow({
    required this.positiveFlow,
    required this.negativeFlow,
  });

  @override
  Future<void> handle(NluIntent intent) async {
    if (['Confirm', 'Acknowledge', 'Agree'].contains(intent.intent)) {
      positiveFlow.delegate = delegate;
      positiveFlow.handle(NluIntent(intent: '', slots: {}));
    } else if (['Cancel', 'Reject', 'Disagree', 'Deny']
        .contains(intent.intent)) {
      negativeFlow.delegate = delegate;
      negativeFlow.handle(NluIntent(intent: '', slots: {}));
    } else {
      errorCount += 1;
      if (errorCount >= _maxErrorCount) {
        await delegate?.onPlayingPrompt('很抱歉，錯誤次數過多');
        await delegate?.onEndingConversation();
        return;
      }
      await delegate?.onPlayingPrompt('很抱歉，我不懂你的意思，你確定嗎？');
      final newFlow =
          ConfirmVuiFlow(positiveFlow: positiveFlow, negativeFlow: negativeFlow)
```

```
            ..errorCount = errorCount;
      await delegate?.onSettingCurrentVuiFlow(newFlow);
      await delegate?.onStartingAsr();
    }
  }

  @override
  String get intent => '';

  @override
  List<String> get slots => <String>[];
}
```

`ConfirmVuiFlow` 有兩個成員變數：`positiveFlow` 與 `negativeFlow`，分別代表使用者回答「對」與「不對」時，要進入的下一個 VuiFlow。在 `handle` 中，我們檢查使用者的回答，如果是「對」，就進入 `positiveFlow`，如果是「不對」，就進入 `negativeFlow`。如果使用者的回答不是我們預期的，我們會再次詢問使用者，直到錯誤次數過多，就結束對話。

取消

我們需要特別注意，使用者不一定在確認的流程中說出「取消」，而很有可能在各種其他的狀況下說出來。像是，在點餐系統中，使用者才剛說完「我想點餐」，我們用「您想要點什麼」回應之後，使用者突然反悔，可能是突然想去別家店，也可能是看不到喜歡的餐點，而說了「取消」，想要結束對話。

我們這時候可以有兩種選擇：一種就是在原本「你想要點什麼」的 VuiFlow 中，多判斷傳入的意圖是不是取消，如果是，那就進入結束對話的流程分支。這麼做在我們有很多的 VuiFlow，語音助理的功能很多時，就很容易產生大量重複的程式碼，因為等於每個詢問，都需要處理取消的意圖。我們也可以實作一個全域的判斷，在任何時刻，只要使用者說出「取消」，就結束對話。修改的地方就在對話引擎的 `handleInput` 這段：

```
  Future handleInput(String input) async {
    _emit(DialogEngineCompleteListening(asrResult: input));
    await ttsEngine.stopPlaying();
    await asrEngine.stopRecognition();

    try {
      final additionalPrompt = _collectionAdditionalNluPrompt();
      final intent = await nluEngine.extractIntent(
        input,
        currentIntent: _currentVuiFlow?.intent,
```

```
      additionalRequirement: additionalPrompt,
    );

    if (intent.intent == 'Cancel') {
      // 播放 TTS
      stop();
      return;
    }
    ...
```

兩種作法之間並沒有哪種比較好，而是根據實際的產品需求決定。

開發語音應用的挑戰

在打造語音應用的時候，最常見的挑戰，來自於模型的能力，環境噪音，還有延遲的問題。

模型的能力

在我們前面的示範中，我們大量使用系統所提供的 ASR、TTS 的能力，在 NLU 與 NLG 方面，則大量使用 LLM 的能力，在這裡使用的是 Gemini。而我們的語音助理最終的使用體驗，就大量取決於這些元件、模型的表現—如果辨識錯誤，或是無法辨識，使用者體驗就會大打折扣。而通常最大的問題是 ASR。

對於使用者來說，NLU 引擎辨識意圖的過程，其實是不可見的，所以，就算辨識出對開發者而言錯誤的意圖，我們還是有辦法可以對應到指定的對話流程上。如果我們發現某套 TTS 引擎，會把某個句子中的某個詞唸錯，我們也可以把這個句子換掉，換成某個其實寫起來不對，但是發音正確的句子到 TTS 引擎上，然後在 GUI 畫面與 TTS 引擎其實拿到不同的句子。這些問題大概都可以用障眼法解決。但如果是 ASR 辨識錯誤，那就沒有什麼障眼法可以用了。

假設我們有一套手搖飲料的的語音點餐系統，裡頭有一項叫做「好的噗咩茶」這樣的產品，這並不是常見的詞彙，各平台上的 ASR 引擎大概就不認得，可能會辨識成「普洱茶」。我們可以做個選擇，當 ASR 引擎完整辨識出了「普洱茶」時，我們假設這是「噗咩茶」的錯誤結果，把要訂購的產品强制指定到噗咩茶上，但，如果店家同時有噗咩茶與普洱茶，這樣就很有問題。還有一種可能是，ASR 引擎根本沒辦法把「噗咩茶」辨識出來，

一個方向是，這個模型需要加强分辨「噗咩茶」或「普洱茶」的訓練，但使用既有模型的我們並沒有這個能力，我們或許要求助有自家 ASR 模型的公司，在我的職業生涯中，也看到一些公司曾經嘗試透過 Open Source 的元件打造語音應用，但最後還是因為 ASR 的能力，而去尋找可以客製 ASR 模型的公司。

或著，可以選擇在產品設計上避開，比方說，我們的店家是否應該要有「噗咩茶」這個產品，我們是否可以把「噗咩茶」換成「檸檬茶」，這樣就可以避開這個問題。這邊能做的選擇，就會依賴商業上的判斷，像是使用「噗咩茶」這個名稱能帶來的收益，與找人訂製 ASR 模型的成本相比，是否更值得。

而在一些時候，我們還會遇到完全無法適用 ASR 功能的情境，比方說，我們想透過語音功能，傳訊息給指定的聯絡人，但他的名稱可能充滿了罕見字，像是「龘䶛䨻䆉𠁥蚮𨎮𪙊𪚊韇𩙧𩯨𤛓」[1]，或是充滿了一堆想唸都唸不出來的符號，像是「▆▄▂z★nble▁▂▄▆」。那就完全無能為力了。

環境噪音

如果使用者處在一個吵雜的環境中，那麼環境的噪音一定大大影響 ASR 的辨識結果。我們可以在產品設計上提出一些對使用者的要求，例如要使用指定型號的耳機或麥克風，甚至是特定的指向性麥克風陣列。

或著，在一些系統中，我們可以透過設定一些系統消除噪音，以 iOS 為例，我們可以對 AVAudioSession 設定不同的 mode，iOS 會在設定不同的 mode 之後，套用不同的消噪的演算法，而所謂的 mode 就是不同的情境，比方說，如果選擇了 `videoRecording`，用手機拍片的模式，那就會變成盡可能要把什麼聲音（包括噪音）都錄進來，顯然就不是我們想要的錄音模式。比較適合像是 `voiceChat` 或是 `videoChat` 這種假設使用者是手持手機，跟使用者保持一個距離的錄音模式，而在 iOS 12 之後，蘋果特定針對語音識別的情境，加上了 voicePrompt mode。

而如果需要更進接的消噪，那大概得具有消噪技術的供應商了。

延遲

在我們這次的示範中，使用的 ASR 引擎會用到網路識別，像是，speech_to_text 在 iOS 上用到了蘋果的 API，而蘋果的實作是用到了他們的網路服務，而 NLU、NLG 引擎，則是用到了 Gemini API。只要是透過網路傳輸資料，自然就會有延遲（Latency），這個延遲會影響到使用者的體驗—使用者會因為等待而煩躁，像是使用者只需要說一聲「對」或「好」，之後卻需要好幾秒的等待，才會進入下一步。

針對一些短句，我們或許需要做一些最佳化，消除使用者的等待時間。比方說，在收到 ASR 結果之後，我們先用一個簡單的字串表格，與 ASR 結果比對一輪，如果發現是一些明確的短句，那也就不用去使用線上的 NLU 分析功能。另外，ASR 引擎通常會在使用者講完話之後，大概等待幾秒鐘，確定使用者真的講完了，但是，如果我們一開始就知道，語音引擎現在期待的就是講一些「好」、「是」、「確定」或「取消」等短句，我們可以不用做這樣的等待，只要收到了符合的辨識結果，就馬上進入下一步對話流程。

另外，您也可能會有在完全不連網的狀況下，需要一套語音助理。那麼，這裡所打造出的方案，就完全不適用了。

其他

其他挑戰像是：我們需要開發特定語言的語音應用，但是我們自己並不會講這種語言，身邊也沒有會講這種語言的人，一來不知道怎麼不知道怎麼發音，讓我們有辦法測試開發出的系統，也不確定在特定文化中，我們的流程是否符合當地習慣，甚至會不會冒犯使用者。我們或許要找更多的合作夥伴，不是單純可以從技術的方向解決。

而說到國際化的問題，想要讓語音助理同時處理多國語文，也是一項挑戰。在 NLU 的部分，基於文字的語言模型，通常比較能夠分辨出一段文字屬於哪一種語言，而且根據傳入的語言給予相同語言的回應，比較大的問題還是出在 ASR 上，目前常見的 ASR 引擎通常還是以一個語言為主，突然講另外一個語言，大概都會得到預期以外的回應。但目前 ASR 技術也在飛速進步，我們期待不久就能夠方便使用可以同時正確辨識多國語文的 ASR 引擎。

---

1. 出自「龘齧𩇔穲」這首歌，https://zh-yue.wikipedia.org/wiki/龘齧𩇔穲 ↩

下一步

我們完成了一個簡單的對話引擎，不過，距離完整的語音助理，還有很多的工作要做。

與 GUI 的整合

接下來自然是要將這個對話引擎，實際應用在一個完整的 Flutter 應用當中，我們可以考慮把我們的語音引擎放在一個 Provider 或是 BLoC 中，放在 Widget Tree 比較上層的位置，讓整個 App 的其他部分可以開始或是結束對話，以及透過監聽語音助理的狀態變化，更新 GUI。

在實務上，我們除了在 GUI 上反應語音助理的狀態，也常常會有複雜的 GUI/VUI 混合的互動流程，比方說，在點餐系統中，當使用者講了「我想點餐」，我們除了用 VUI 反問使用者「您想點些什麼」，進入第二輪對話之外，也可能用一個 System Call，在畫面中跳出一份菜單，除了可以用語音回答之外，也可以用點擊的方式選擇，我們或許也可以設計像「前一個」、「下一個」、「前一頁」、「下一頁」等命令，而這些命令，也都會讓 GUI 因此改動。

在做 GUI/VUI 混合的產品設計時，除了 VUI flow 本身的 Activity Diagram 之外，也需要設計 GUI 的 Wire Frame 或是 Mockup，兩者互相搭配。至於 GUI 開發，則與大多的 Flutter 應用開發過程無異，在這裡不贅述。

更多不同的實作

我們現在使用一些 Flutter 本身提供的 ASR 與 TTS 套件，加上 Gemini API 提供 NLU/NLG 功能，不過，我們不見得想使用這些套件或服務。由於在這個架構中， ASR、TTS、NLU 與 NLG，都是 interface，因此，只要按照這些 interface 實作，都可以替換掉原本的套件。

喚醒（Wake-up）

我們在這裡並沒有提供喚醒功能。在喚醒系統的時候，會需要一套喚醒詞（Wake-up Word），常見的喚醒時就是「Hey Siri」、「OK Google」、「Alexa」等，語音助理平常處於一種等待狀態，雖然在錄音，但是並不做複雜的識別，只偵測在路到的聲音當中是否符合上述這些喚醒詞，如果有，才進入 ASR 識別狀態，開始嘗試識別完整的文字。雖然在很多應用中，並不需要用這種方式啓動語音助理，但或許您會有這種需求。

在 Flutter 提供的套件中，其實並沒有特別是合開發喚醒功能的套件。喚醒功能的要求是省電，而且避免連接網路，降低使用者的設備的耗電以及網路費用，但是像 iOS/Android 系統所提供的系統語音識別，背後通常都連接到特定的網路服務上，而且每一次識別都有時間上限，不適合長時間處於閒置狀態，而且每天每個 app 可以開啓識別的次數也有限制，大約一天 2000 次左右（這個次數可能隨時會有變化）。

因此，如果您的 app 有使用喚醒詞的需求，大概還是要倚賴有喚醒技術的供應商。