ローカルで動かす対話AI

スマホのローカル LLM で、会話の感情をスコアにする

ぱそぴ屋　2026

目次

はじめに　動機／出発点「感情はベクトルである」／検証の道筋

第1章　感情ベクトルとは何か
1.1 論文が示したこと／1.2 感情を「地図の上の点」として持つ

第2章　感情をどう取り出し、何を信号にするか
2.1 三つの取り出し方／2.2 自己申告 valence の弁別／2.3 ①活性 と ②自己申告 の信頼度（実測）／2.4 なぜ「快・不快」一本にまとめるのか／2.5 好感度の積み方（m と p）／2.6 スコアは文脈に戻る（だから採点は決定論的に）

第3章　アプリの設計
3.1 ルール／3.2 目的は「会話の続き方」に置く／3.3 性格で感情の出方を変える／3.4 裏づけ：HRI と Pepper／3.5 長期の一貫性：3分という区切り

第4章　どの LLM・ランタイムで動かすか
4.1 推論ランタイム（llama.cpp か LiteRT-LM か）／4.2 モデルはテキスト専用版にする

第5章　モデルを学習して載せる
5.1 LoRA を学習して、捨てた／5.2 テキスト専用モデルを LiteRT-LM に載せる／5.3 詰まった箇所：サンプラーの制御／5.4 実測結果：メモリの実態／5.5 計画：メモリ管理の先にある一手

第6章　スマホで速く動かす
6.1 第一声が遅い原因と、効いた一手／6.2 GPU オフロードの実装／6.3 TTS の競合／6.4 どの端末で要るか（RAM より SoC）

第7章　音声まわりの選択肢（STT・TTS）
7.1 音声認識（STT）／7.2 音声合成（TTS）

おわりに　／　参照

はじめに

本書は、頒布版の増補改訂版である。配布時にいただいた質問を踏まえて増補した。配布版は当時のアプリで取得したデータをもとにしており、増補版で差し替えた内容もある。その点はあらかじめ断っておきたい。

本書は、スマートフォンの中だけで動くローカル LLM を使い、会話の「感情」を数値スコアとして取り出して、それを中心に据えた会話ゲームを作った記録である。今回作成したアプリ（スマホで動く 3 分間の会話ゲーム）を題材に、モデルの学習・応答の高速化・メモリ管理・選べる技術スタックを並べる。目的は、**同じ問題に取り組む人が、設計判断と落とし穴を追えるようにすること**である。各章は「どう進めるか」を本文に置き、避けるべきことは末尾の **注意点** にまとめた。

なお、本書でいう「本アプリ」は、実機で STT → LLM → TTS の会話ループが動く試作を指す。本文では、**実装済み・計画・外部の公表値**をなるべく分けて書いた（たとえば LoRA への焼き込みやキャラ別 LoRA、MediaPipe 移行は計画、ベンダの速度は公表値である）。読むときの目安になれば幸いである。

動機

筆者の目的は、**AI を、できるだけ多くの人が実際に使う道具にする**ことである。クラウドの API に課金し続ける道具は、料金が要る時点で「じゃあ使わない」となりやすく、わざわざ使おうと思う人は限られる。この**料金という壁**をなくしたい。スマホは誰もが持っているデバイスだから、スマホの中のローカル LLM で、通信も課金もなしに対話を続けられる状態を作れば、そこで止まっていた人にも届く。本アプリがオフライン・オンデバイスにこだわるのは、この理由による。

出発点 ── 「感情はベクトルである」

直接のきっかけは、生成 AI なんでも展示会 vol.5 で、ある展示を見たことだった。入力した文章を、ローカル LLM の内部表現から感情として取り出してリアルタイムに可視化する展示で、これに興味を持った。その後、この展示を解説した milktea 氏の note 記事「感情ベクトルと戯れる」がよくまとまっていたので、参考にした[1]。展示の下敷きは、Anthropic の論文「Emotion Concepts and their Function in a Large Language Model」（2026）である。

ここで得た気づきは、まず **AI は感情のような機能を持つ**こと──人のように「感じている」わけではないが、感情に相当する内部の働きがあって出力を左右する（第1章で詳しく見る）。そして、**その感情はベクトルとして表せる**こと。ベクトルなら向き（傾き）と大きさ（強さ）を持つ。だから数値化でき、会話の中で蓄積でき、キャラクターの感情をリアルタイムに動かせる。つまり、あらかじめ用意した分岐をたどるのではなく、**こちらの話し方しだいで相手の感情が連続的に動くゲーム**が作れる。これが出発点である。

検証の道筋

本書はおおむねこの順で進む。

1. **取り出せるか**：LM Studio で、手元の LLM から感情に相当する反応が取り出せそうかを確認する。
2. **論文と照合する**：感情が内部表現（活性）として線形に存在し、valence×arousal で整理できる枠組みを確認する（第1章）。
3. **ベクトルだと確かめ、読み方を決める**：llama.cpp ベースの API で内部表現を抜いて向きと強さを確かめ、オンデバイス制約から①活性ではなく②自己申告を信号に採る。①と②の信頼度も実測で突き合わせる（第2章）。

4. **ゲームにする**：キャラごとの性格を persona として渡し、同じ働きかけでも感情の出方がキャラで変わるようにする。性格で感情反応が割れること、3 分という区切りの妥当性も、実測と文献で押さえる（第3章）。
5. **スマホで動かす**：LLM とランタイムを選び（第4章）、学習で何をやり何を捨てたかを残し（第5章）、オンデバイスで現実的な速度・メモリに収め、どの端末で動くかまで詰める（第6章）。
6. **声を与える**：オンデバイスの音声認識・合成（STT・TTS）の選択肢を記録する（第7章）。

> [1] 出典：milktea「生成 AI なんでも展示会 vol.5 展示解説／感情ベクトルと戯れる」note 記事。

第1章　感情ベクトルとは何か

この章では、出発点になった Anthropic の論文（2026）の中身のうち、**本アプリで使う点だけ**を拾う。本書で以降「論文」と書けばこれを指す（ほかの研究は著者名を添える）。感情ベクトルそのものの作り方（活性の抽出・平均差分）は論文に譲り、ここでは「だからキャラクターをこう扱える」という側に寄せて書く。

1.1　論文が示したこと

LLM は文章を処理する途中で、各層に **残差ストリーム（residual stream）** と呼ばれる高次元の内部表現を持つ。論文が示したのは、**モデルが感情を含む文章を読んだときの内部表現の中に、感情が線形に表れる**ことだ。

ただし、これは **LLM 自身が感情を感じている**という話ではない。感情ベクトルは、ある感情を持つ登場人物の物語を大量に読ませ、そのときの活性を平均して作る。つまり取り出しているのは、**文章（話者）の感情を読み取って内部に表す、モデルの能力**であって、受け手である LLM の「気分」ではない。論文はこれを functional emotions（機能的感情）と呼ぶ。主観的な体験ではなく、出力に影響する内部表現、という意味である。

整理すると、「AI は感情を持つか」は 3 つの層に分けると正確になる。

層	中身	AI は？
感じている（主観）	人間のように内側で本当に感じている	無い
内部にある（機能）	感情に対応する内側の傾きがあり、返答を左右する	有る
出てくる（表出）	結果として、感情があるように振る舞う	有る

だから本アプリが読み取るのは、表の**真ん中の段（機能）**である。AI の気持ちそのもの（いちばん上の段）を測っているのではなく、**この後の返答が良い**

方に出るか悪い方に出るかを、言葉になる前に先読みしている——これが本アプリでいう「感情を読む」の中身だ。

そのうえで、本アプリで使う性質は次のとおり。

- **感情は細かく分類できる（171 概念）。** 論文は happy・afraid・angry・desperate など 171 種類の感情概念を扱う。感情は「喜び／怒り」の二択ではなく、細かい構造を持つ。
- **その構造は 2 軸に畳める。** 171 概念の方向ベクトルに主成分分析（PCA）をかけると、心理学で知られる **快/不快（valence）** と **興奮/沈静（arousal）** に近い 2 軸が現れる。つまり感情は、おおよそ 1 枚の平面の上に配置できる。
- **平面の上では円環状に広がる。** valence×arousal の平面に感情を置くと、原点を囲んで放射状に分布する。喜びは右上、怒りは左上、安らぎは右下、というように、向きで感情の種類が、原点からの距離で強さが表れる。
- **valence には符号がある。** 横軸の正負がそのまま快/不快に対応するので、ある発話が「快寄りか不快寄りか」を符号で読める。スコアにしやすい。
- **読み取った表現は、出力にも因果的に効く。** 論文は、ある感情（たとえば絶望）を含む文章を読ませたときの活性を平均して「その感情の方向」を取り出し、それを推論の最中の内部表現に**直接足す**（活性注入＝steering。言葉で「絶望して」と頼むプロンプトとは別物で、内部のダイヤルを直接ひねる操作である）。すると振る舞いが実際に変わった——「絶望」を足すと不誠実な行動が増え、「落ち着き」を足すと減る、というように注入と行動が対応した。少なくとも論文の条件では、感情の方向は観察対象にとどまらず、応答を変える操作対象にもなりうる。だから将来は、測るだけでなく **操作**（難易度調整など）にも使える余地がある。
- **感情はその場限りで、持続しない。** 論文の感情はその時点の操作的なもので、何ターンも保持される「気分」ではない。だから**気分の蓄積はモデ**

ルに任せず、コード側で状態として持つ——という設計が、ここから直接決まる（第2章）。

なお、感情や快・不快が内部表現の**ある方向**として線形に取り出せること自体は、Anthropic（2026）より前から積み上がっている。たとえば 2024 年には、文章の好き嫌い（sentiment）が隠れた内部表現の中の**一本の線**として表れることが報告されている（Tigges ら, 2024）。Anthropic の論文は、その上に一歩踏み込んだ。先行研究が「感情の方向が内部に**ある**」ことを示したのに対し、Anthropic は、その方向を内部に足すと**出てくる言葉や振る舞いが実際に変わる**こと——つまり感情の方向が出力を生む原因になっていることまで確かめた、という位置づけになる。

図1　valence × arousal の感情平面。本書が信号に使うのは横軸 valence 一本

1.2　ここからの踏み込み —— 感情を「地図の上の点」として持つ

ここからは論文そのものではなく、論文の枠組みから本アプリが踏み込んだ設計の出発点である。

第1章の図1のように、感情は１枚の地図の上に広がっている。すると、どの感情も「**地図の上のどのあたりにあるか**」という一点で表せる。右へ行くほど快、左へ行くほど不快、というように、置かれた場所がその感情の中身になる。

この見方の利点は、**キャラクターの「いまの気分」を、地図の上の一点として持てる**ことだ。会話のたびに生まれる感情を地図の上で少しずつ積み重ねていけば、キャラの気分の点が動く。あらかじめ用意した分岐をたどるのではなく、会話の流れでキャラの気分そのものが動く——という当初の狙いは、この見方から具体化した。

そして、キャラごとに地図の上の「いつもの居場所（くせ）」が違うと考えれば、同じ働きかけでも気分の動き方が変わる。これが「性格で感情の出方を変える」発想につながる（第3章）。

なお、ここまでの性質は大型モデル（論文の対象は Claude Sonnet 4.5）で示されたものである。本アプリではそもそも、スマホで使う推論ランタイム（LiteRT-LM）が活性を取り出せない（第4章）。だから感情は活性抽出（①）ではなく、モデル自身に答えさせる **自己申告（②、第2章）** で読む——②に逃げたというより、①が使えないので②で読む、が正確である。加えて、仮に活性が取れる環境でも、小型・量子化モデルでは①の精度が落ちやすい（量子化で内部表現がドリフトする）という事情もあり、これは第2章で実測して確かめる。

第2章　感情をどう取り出し、何を信号にするか

2.1　三つの取り出し方

感情の出所は三方式に整理できる。どれを使えるかは、後の LLM・ランタイム選定（第4章）に縛られる。

方式	何をするか	必要なもの
① 活性抽出	推論中の残差ストリームにフックを刺し、valence 方向へ射影	活性にアクセスできる推論エンジン
② 自己申告	LLM 自身に「今の valence はいくつか」を答えさせる	構造化出力プロンプト＋パーサ
③ 入力分類	ユーザー発話を別モデルで感情分類	分類器（補助的）

①は第1章の手法そのもので、llama.cpp 系なら推論コールバック（cb_eval）で活性を抜ける。理屈に最も忠実だが、活性を返すエンジンでしか使えない。②はモデルに直接 valence を申告させる方式で、**(a)** 単一ラベルではなく valence 一本の連続値にし、**(b)** 返答と valence を構造化出力で同時に吐かせてパースする、の二点で実用になる。①が使えない環境では②を採る。

実装では、返答本文とは別に、**感情信号を構造化されたフィールドとして受け取る**。本質は JSON かどうかではなく、話す内容とスコアに積む信号を混ぜないことである。受け取り方を具体で書いておく。

②自己申告（本アプリ） は、systemPrompt の末尾に「1 行目に valence、2 行目から返事」と指示し、返ってきた構造化応答をパースする：

```
（指示）1 行目に、いまの気持ちの良し悪しを valence=<-1.0〜1.0> だけで書く。
        2 行目から、相手への返事を書く。
```

```
（LLM の出力）
valence=0.6
へえ、それいいね！もっと聞かせてよ。
```

1 行目を正規表現で抜けば、その回の valence（感情信号を快・不快軸へ落とした値）が取れる——これを今回の好感度 m として p に積む（m と p の扱いは §2.5、valence の大きさが性格でどう違うかは §3.3）。残りの行が、発話する返事だ。`valence=0.6` の 0.6 は形式を示すための値で、特定の意味はない。なお小型モデルでは、トークン名や言語を変える（日本語で書かせるなど）と申告が安定することがあるので、実機で確かめる。

①活性抽出 なら、llama.cpp の `cb_eval` で受け取るのは活性ベクトル（次元＝隠れ層の幅の float 配列）だ。これを、あらかじめ作っておいた valence 方向の単位ベクトルと内積して、スカラーの valence にする。

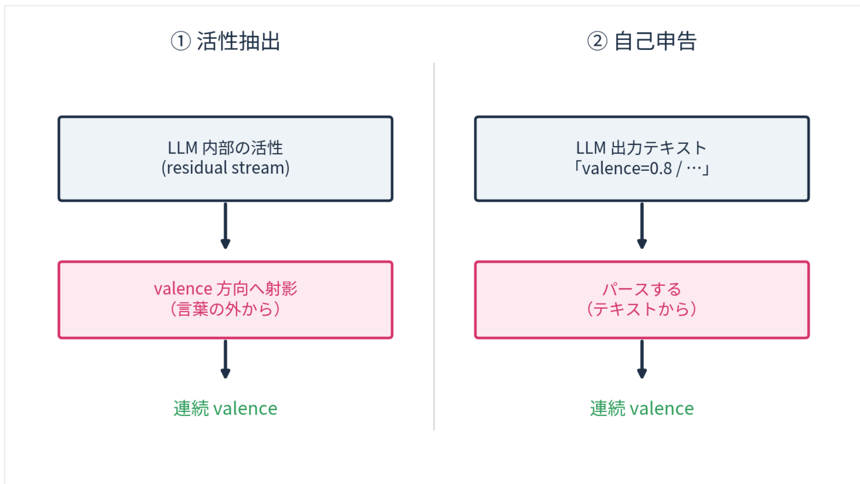

図2　①活性抽出 と ②自己申告 の対比

2.2　自己申告 valence の弁別

②が実際に感情を弁別するかを確認した。会話用に調整したモデルにいくつかの感情シーンを与え、返答に添えて valence を申告させた（temp=0.2）。

```
JOY     valence= 0.7〜0.9    (平均 +0.8)
ANGER   valence=-0.8〜-1.0   (平均 -0.93)
NEUTRAL valence= 0.3
```

joy／anger／neutral を単調に弁別する。なお値は非対称で、ANGER は端（−1.0）に張り付き、JOY は +1.0 まで届かず、NEUTRAL もゼロでなく +0.3 とやや正側にいる。今回の条件では、会話を気持ちよく続ける方向に調整された instruction-tuned モデルらしく、ベースラインがやや正側に寄る傾向が見えた。本アプリはこれを別途ならす補正はしない——性格差は感情ベクトルの大きさに最初から入っており（クールは小さく、明るいは大きく。§3.3）、その大きさをそのまま好感度へ積むからである（§2.5）。

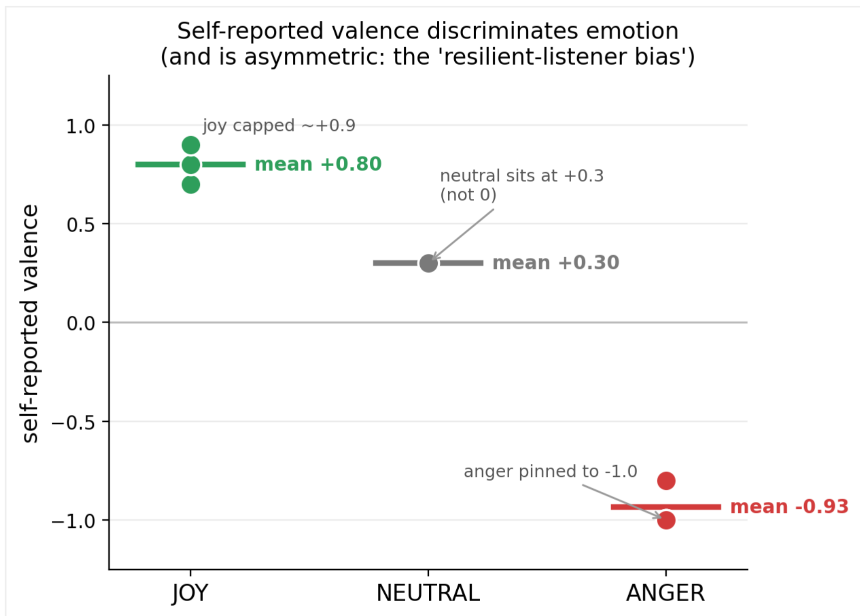

図3　自己申告 valence の実測。JOY/ANGER/NEUTRAL を弁別しつつ負へ振れにくい

2.3　①活性 と ②自己申告 の信頼度（実測）

①（活性抽出）と②（自己申告）が、同じ入力に対してどれだけ一致するかを実測した。快・不快 × 活性 の 4 象限に分けた **N=1152**（各象限 288 文）の各文に、①（内部の活性を valence／arousal 方向へ射影）と②（モデルに valence／arousal を自己申告させる）の両方を当てて突き合わせた。

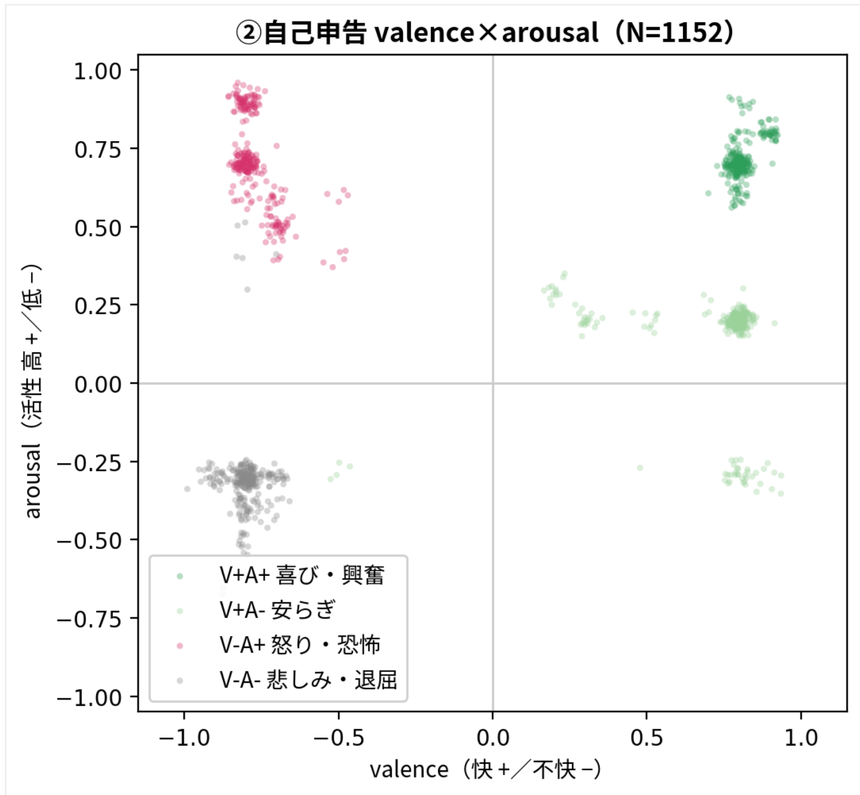

図4　自己申告 valence × arousal の4象限

②自己申告は、快・不快（横軸）を明確に左右へ分けた（V＋群 +0.7〜+0.9、V−群 −0.8 前後）。活性（縦軸）の分離はやや弱く、モデルは負の活性をあまり申告しない――2.2 で見た聞き役バイアス（正側への偏り）と同じ傾向である。

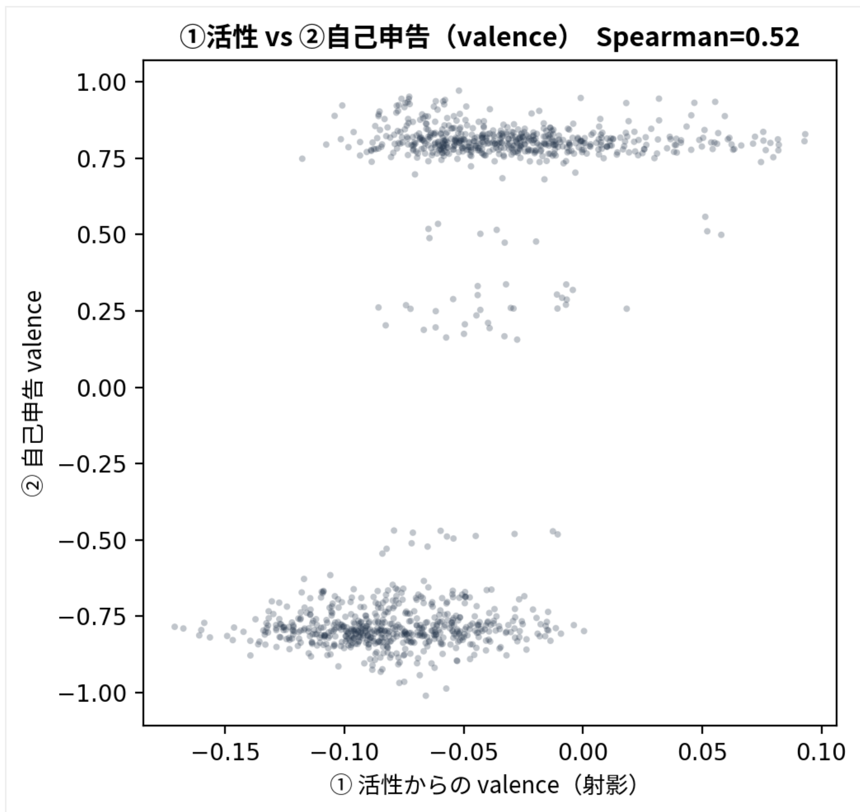

図5　①活性射影 と ②自己申告 の相関

①と②の相関は、valence で **Spearman 0.52／Pearson 0.59**、arousal で 0.56。弱くはないが重なりが大きく、①は②ほどきれいには分かれない。ただしこれは「①が一般に弱い」のではなく、ここで使った①が**最終層・平均プーリングという手軽な取り方**だからである。本プロジェクトのより丁寧な活性抽出では、自己申告 valence と活性 valence の Spearman は **0.70** に達していた。なお正解ラベル（設計上の極性）との一致は、自己申告が **0.876**、活性が **0.605** で、自己申告のほうが正解に近い。②を本番に採ったのは、スマホで動かす量子化済みモデル（Gemma-4 E2B Q4_K_M）では、手軽に取れる①（手軽抽出は量子化で 0.52）よりも②のほうが安定して信頼できる、という

比較に基づく（より丁寧な①なら 0.70 まで上がるが、本番でそこまでの抽出コストはかけない）。

なお本アプリは、活性注入（①の方向を内部に直接足す操作）は**していない**。やっているのは②で、その場の感情の傾きを**読み取って**スコアにすることだけである。活性注入は、論文が「傾きが出力を動かす」と証明するために行った実験室の操作で、ゲームの仕組みとは別物である。

2.4　なぜ感情を「快・不快」一本にまとめるのか

感情は 171 種類もあるが、ゲームが毎ターン必要とするのは「いま機嫌が良いか・悪いか、どのくらいか」という **一本の数字** だけだ。だから、たくさんある感情を、最後はこの一本に変換する。その中身を書く。

使うのは、第1章の図1の「感情の地図」だ。地図の **左右** が「不快（左）⇔快（右）」を表す（上下は『落ち着き⇔興奮』だが、ゲームでは使わない）。どんな感情でも、**地図の上でどれくらい右（快）か左（不快）か**、その左右の位置だけを読む。喜びはずっと右（プラス）、怒りはずっと左（マイナス）、どちらでもない話題は真ん中あたり（ほんの少しプラス）になる。こうして 171 種類の感情も、ゲーム上は「**怒り寄り（−）〜喜び寄り（+）**」という一本の目盛りへ近似して扱える。これが本書でいう valence（快・不快）で、ゲームが使う唯一の信号だ。

2.5　好感度の積み方（m と p）

スコアは二つの値で持つ——**m（今回のターンの好感度）** と **p（会話全体の蓄積好感度）** だ。

毎ターン、LLM が感情ベクトル（快・不快の Valence 一本）を出す。これは性格を映した、その場限りの値で、**クールなキャラほど小さく、明るいキャラほど大きく**振れる（性格差は、この大きさに最初から入っている）。このターンの Valence から **今回の好感度 m** を取り出し、それを **蓄積好感度 p** に積

む。p は 3 分のあいだ積み上げて**稼ぐ**値で、良い会話なら伸び、まずい会話なら削れる。

第1章のとおり、感情ベクトルはその場限りで揮発する（モデルは持ち越さない）。だから積み上げは p の形で**コード側に持つ**——決定論的に積め、New Game で巻き戻せる。性格差はすでに感情ベクトルの大きさに入っているので、コード側で別途ならしたり係数を掛けたりはしない。**感情ベクトル → m → p、の素直な積み上げだけ**で、キャラ別の補正層や追加係数は置かない。

> **注意点**
> ・本アプリがゲーム信号として使うのは Valence 一本。個別の感情（喜び・怒り…）を別々のゲージにしない。
> ・性格差は感情ベクトルの大きさに入っている。コード側で均一化したり係数で消したりしない。
> ・蓄積をモデルに期待しない（感情は揮発する）。p の形でコード側に持ち、決定論的に積む。

2.6　スコアは文脈に戻る（だから採点は決定論的に）

好感度 p は、毎ターン LLM に渡される——「上限に対するスコア（たとえば 60/100）」として前置し、モデルはいまどのくらい好かれているかを読み取って、態度に自然ににじませる（口調を命令で指定するのではなく、好感度という情報を渡すだけだ）。会話ロジックとスコアロジックは、ここで**ループ**になっている——**p → 会話 → 感情ベクトル → m → p**。

このループがあるから、採点はプロンプトの揺れに左右されず、**決定論的に**出す——採点が揺れると、キャラに渡る値も揺れるからだ。なお、長く続けたときにキャラが筋を失う問題（ペルソナ・ドリフト）は、採点とは別の層として§3.5 で扱う。

> **注意点**
> ・好感度は状態として持つだけでなく、毎ターン文脈に戻る（p → 会話 → m → p のループ）。

> ・だから採点は決定論的に。プロンプトの揺れで点が動くと、会話に戻る値もぶれる。
> ・キャラが筋を失う「ドリフト」は採点とは別の層。§3.5 で扱う。

第3章　アプリの設計

3.1　ルール

- 8 人の女性キャラから一人を選び、3 分間、会話を続ける。
- やり取りは音声のみ。こちらが話すと相手が声で返す――これを交互に繰り返す（ターン制）。
- キャラの機嫌が会話で上下し、好感度が動く。
- 3 分間、会話を続けながら好感度を積み上げる（マイナスで終えなければ上々）。

3.2　目的は「会話の続き方」に置く

このゲームで達成させたいのは、**相手と自然に会話を続けられること**――会話のラリーを、気持ちよく長くつなぐ体験である。だから評価が向くのは、**どれだけ自然に会話を続けられたか**――つないだターン数やコンボといった"会話の上手さ"である（スコアの具体的な構成はまだ設計中で、方向としてこれを置く）。

好感度とは、相手の感情を快・不快の一本に落とした値で、会話のなかで正にも負にも揺れる（取り出し方は第2章）。p はゲームを通じて積み上げる値だ。ゲームが見るのは、好感度を無理に稼ぐことではなく、**会話を自然に続けた結果として p がどう積み上がったか**――その"会話の上手さ"である（成功/失敗の二値ではない）。

「好感度を最大値まで上げること」を主指標に据えると、ゲームは攻略法を覚えて特定の到達点を引く作業になり、肝心の会話が形骸化する。だから好感度は**会話の質の結果**として扱い、目的には据えない。

会話の主役は、こちら（プレイヤー）側に置く。 自然に会話を続ける練習である以上、話すのは人間の側で、キャラは**聞き役**にまわる――相手の言葉を受け、共感し、気持ちで応える側である。だからキャラからは話題を振らない

し、自分の話に持っていって受け流すこともしない（キャラが自分語りを始めると、相手の言葉をかわすことになる）。一回の発話は、ひと呼吸で言い切れる短さにして、すぐ相手に番を返す——長く喋りすぎると、会話の「量の格率」（グライス）に反して不自然になる。

キャラ側が「もっと聞きたい」かどうかは、好感度で切り替える。 聞き役に徹すると、ともすれば「うん」「そうだね」で会話が止まってしまう。そこで、相手の話に一歩踏み込む**助け舟**——「それで、どうなったの？」のように、相手がいま言ったことを短くたずねて、もっと話してもらう問いかけ——を、**好感度が高いときほど出やすく**する。好意が芽生えるほど「もっと聞きたい」と前のめりになり、まだ打ち解けていなければ受けるだけにとどめる、という切り替えだ。相手の話を掘る問いかけは聞き手の関心の表れで、相手の好意を引き出す（フォローアップをよく投げる人ほど好かれる、という対人研究 Huang ら 2017 とも一致する）。ただし問いを重ねすぎると詰問になるので、本当に気になったときだけにする（§3.4）。

3.3　性格で感情の出方を変える（考え方）

本アプリの考え方は、**全キャラで同じ LLM を使い、キャラごとの性格は persona として持たせる**ことである。感情の測り方（valence 一本）は全キャラ共通だが、**性格を持たせると、同じ働きかけへの感情の出方が変わる**。性格はあらかじめ数値（トレイト）として持たせ、persona 文に展開して LLM に渡す。トレイトの軸立ては HRI（人とロボット対話）で使われる Big Five 系の性格次元に倣う（§3.4）。各キャラの persona には「何に心が動き、何に白けるか」を一行添える。

まず、性格は「感情の出方の大きさ」の違いとして出る（これは今ある差）。 同じ働きかけでも、明るいキャラは大きく喜び、クールなキャラは控えめにしか応えない。クールが「冷たい」のではなく、**喜びを表に出す幅が狭い**だけだ。この大きさがそのまま今回の好感度 m になるので、クールは伸びにくく、明るいは伸びやすい——**それがそのまま難易度になる**（コード側にキャラ別の係数は無い。§2.5）。

そのうえで、関係が深まるほど伸び方を変える「カーブ」を、後から味付けとして足したい（これはまだ確定していない設計の方向）。 たとえば——

- **明るいキャラ**：序盤からぐんぐん上がる（易しい）。
- **ツンデレなキャラ**：最初は上がりにくいが、一定の好感度を超えると急に稼ぎやすくなる（閾値で跳ねる）。
- **クールなキャラ**：全体に稼ぎにくいが、会話のターンを長く伸ばせる。

この「関係段階で変わるカーブ」を、実装上は **gain_c**——好感度の獲得カーブを決める関数——として持つ想定である。まだ確定しておらず、ゲームとして気持ちよくなるようプレイテストで詰める後段の課題だ。

性格で感情反応が本当に割れることは、データで確かめた。 8 キャラに persona を渡し、会話の打ち手（質問・共感・自己開示・称賛・ノリ・理知）を当てて自己申告の良し悪しを測った（各キャラ 252 回、計 **N=2016**）。反応はキャラごとにはっきり割れ（Kruskal-Wallis **H=733**、p≪.001）、並び順も設計どおり——明るい・甘えを受け止める系が高く、クール・気位の高い系が低い。"刺さりどころ" も persona の一行で出る：称賛にはヒナが大きく動き（+0.90）レイは控えめ（+0.36）、**レイは理知的な打ち手に最も持ち上がり（+0.55）、クゥは称賛・共感で開く**（図6）。なかでも**称賛と共感は、どのキャラでも比較的プラス**に出た——関係を温めやすい普遍的な手筋である（§3.4）。

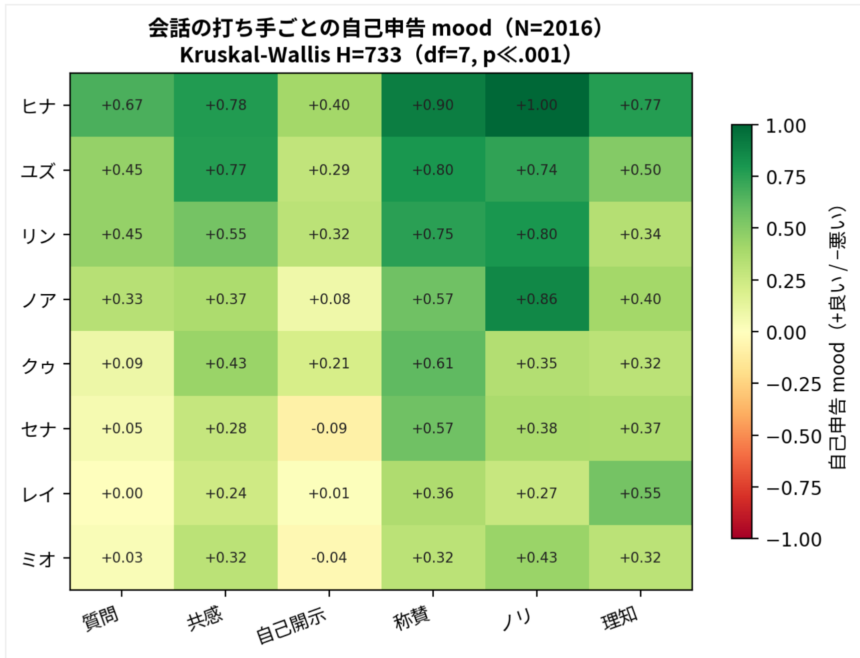

図6　同じ刺激への自己申告が、性格でキャラごとにはっきり割れる

性格は LLM 側に効き、採点規則はコード側で性格に依らない。 性格は persona 経由で **LLM の感情の出方（自己申告する valence）** を変える——クールなキャラは valence を控えめに申告する。コードがやるのは、その valence を好感度へ積む変換で、**どのキャラでも同じ規則**で決定論的に行う（効きは難易度・親密度・コンボで変わるが、キャラ別の係数は持たない）。だから性格の差は LLM の valence に表れ、採点規則そのものは性格を知らない。性格を LLM に効かせる経路は三つある。

- **prompt**（本アプリで採用）：トレイトを persona 文に展開して渡す。学習不要で、上の性格差は prompt だけで出た。
- **LoRA**：性格を**重みに焼く**。会話が伸びても薄れず、persona をプロンプトから外して文脈も軽くできる（ドリフトは §3.5、計画は §5.5）。本番ランタイム（LiteRT-LM）では base にマージして使う。

- **control vector**：内部状態（活性）に性格方向を直接足す。理想的だが、LiteRT-LM は内部フックを公開しておらず、現状は llama.cpp 系にとどまる。

どの渡し方でも、valence を好感度へ落とす採点規則はコードが決定論的に持つ。ここで LLM に委ねるのは valence の観測までで、好感度への変換規則はコード側に固定する。

3.4　裏づけ：HRI と Pepper

感情をスコアにすること、キャラに性格を持たせることは、HRI（人とロボットの対話）研究と実機ロボットの蓄積の上にある。

2014 年の Pepper は、感情を認識するだけでなく自分で感情を生成した[2]。cocoro SB の感情生成エンジンは人間のホルモン分泌をモデルにし、センサ入力を **内分泌型多層ニューラルネットワーク** に通して感情を自律生成する。本アプリも、入力から内部状態を更新して応答に反映する、という**目的のレベルでは近い**。ただし中身は別物で、手設計の内分泌型 NN を再実装しているわけではなく、**LLM が学習で獲得した感情表現** を使う。感情エンジンの中身が学習済み LLM に置き換わった、と見るのが近い。

キャラの性格そのものも、HRI ではロボットに **Big Five（外向性・協調性・神経症傾向…の5次元）的な性格次元を与える**試みが知られる。本アプリのトレイト軸（内向⇔外向、落ち着き⇔不安 など）もこれに倣い、性格を**数値パラメータのベクトル**として持つ。§3.3 の「性格で感情の出方を変える」は、この Big Five 由来のパラメータを persona に展開して LLM に渡す操作にあたる。実装上は、この数値ベクトルを persona 文へ展開して LLM へ渡し（喋りは LLM が出す）、好感度を積む決定論的な採点だけをコードが担う。

会話を成立させる設計原則も HRI から借りている。

- **ラポール（親密さ）は頻度ではなく随伴性で決まる。** タイミングの合った反応が信頼を生む（Gratch 2007）。だから相づちの位置と極性を最優先する。

- **信頼の破綻は回復しにくい。** 3 回の違反後はどんな戦略でも完全には回復しない（Three Strikes 2023）。
- **LLM は聞き返し・明確化を出さなすぎる**（人間の約 1/3、フォローアップは約 1/16。Shaikh 2025）。だから明示的に持たせる。
- **評価はセッション単位で見る。** 会話品質はターン単位では人間評価と一致しにくいが、セッション単位では強く一致する（ADEM で turn r=0.41 → system r=0.95）。主指標を「完走率＋もう一度やりたい度」に置くのはこのためで、BLEU/ROUGE のような単語重なり系は会話品質の評価に向きにくいので使わない。

実装としては、キャラの毎ターンを `[極性一致の受け取り] + [本応答] + [次へのフック]` の 3 部構造にし、平叙の言い切りで終えない。proactive にすると対話継続が伸びる（PaRT 2025）が、中程度が最良で仕切りすぎは逆効果なので、質問は連続 3 つ以上しない（反映:質問 ≒ 2:1）。

> [2] SoftBank/Aldebaran/cocoro SB, Pepper（2014）。
> **注意点**
> ・ネガティブな発話に「いいね」と返すような、極性のずれた相づちを出さない（信頼を壊す）。
> ・質問を連続で打ち返さない。仕切りすぎない。
> ・1 ターンの良し悪しで評価しない。セッション単位で見る。

3.5　長期の一貫性：3分という区切り

はじめに、ここでいう「一貫性」をはっきりさせておく。**開幕と終了で同じ態度でいること**ではない。好感度が上がって、ツンからデレへ態度が変わる——そういう変化はむしろ**設計どおりで、一貫している**。このゲームは好感度を稼いでキャラの態様を変えるのが主眼で、好感度に応じて会話の表出が伸縮するのは正しい動きだ。崩れて困るのは、**性格そのものが筋を失い、同じキャラに見えなくなる**こと。それがこの節でいうペルソナ・ドリフトである。

ロールプレイ中の LLM は、ターンを重ねるほど与えた性格から逸れていく（会話が伸びると、冒頭の persona への注意が相対的に薄まる。ペルソナ・ドリフト）[3]。一貫性が大きく落ちるのは 8〜12 ターンあたりという報告があり、本アプリでも会話が約 9 ターンで崩れ始めるのを実機で観測した。

本アプリを **3 分・有限ターン**に区切っているのは——後付けの整理ではあるが——会話を**崩れる前**に収めてキャラの一貫性を保つ、という設計上の理由づけになる。開放型の長い対話なら崩れていくが、短く区切ればドリフトは軽微に留まる。

その先（もっと長い対話）でも一貫性を保つなら、二段で組むことになる。**固定の性格は LoRA に焼いて**注意の希釈に左右されないようにし、**変動する会話の状態は記憶（毎ターンの再グラウンディング）で補う**[4]。両者でブレ幅を抑えられる。これは第5.5節の LoRA 計画とも地続きである。

> [3] ペルソナ・ドリフト：Measuring and Controlling Persona Drift in Language Model Dialogs (arXiv 2402.10962)／Examining Identity Drift in Conversations of LLM Agents (arXiv 2412.00804) ほか。
> [4] 役割の再グラウンディング・記憶検索：Dynamic Context Adaptation for Consistent Role-Playing Agents with RAG (arXiv 2508.02016)。

第4章　どの LLM・ランタイムで動かすか

ここからが実装である。まず、スマホで動かす LLM とランタイムをどう選んだかを書く。これは感情の取り出し方（第2章）にも直結する。全体の構成は次のとおりで、会話 1 ターンは STT → LLM → 文分割 → TTS → 再生という一方向のカスケードになっている。重い LLM だけを GPU に逃がし、STT・TTS は CPU で回す——この役割分担が本章と第6章の主題である。

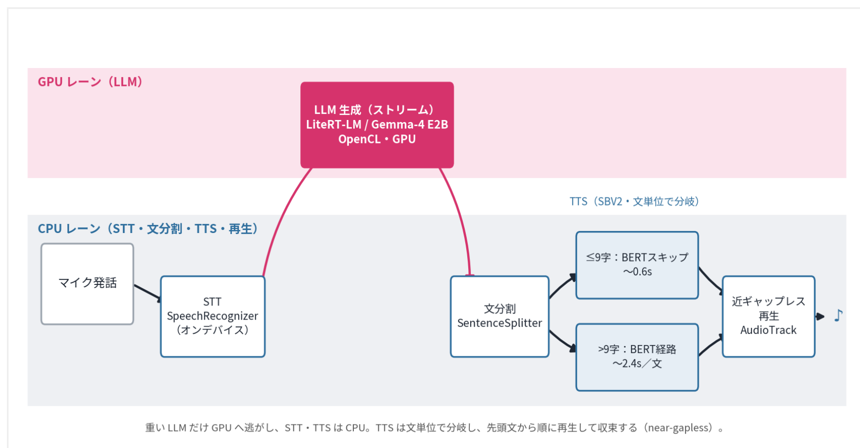

図7　会話1ターンの硬いカスケードと GPU/CPU の役割分担

4.1　推論ランタイム —— llama.cpp か LiteRT-LM か

案	ランタイム	感情の出所	速度	footprint	工数
現状	llama.cpp CPU	①（計画）	2.37 t/s	2.7GB	—
B	llama.cpp + Adreno OpenCL	① 既存	GPU（要実測）	2.7GB	OpenCL ビルド

案	ランタイム	感情の出所	速度	footprint	工数
C	LiteRT-LM フォーク＋活性フック	① 新規	高速	<1.5GB	Bazel 大工事
D	LiteRT-LM（素）	② 自己申告	高速（実機OK）	約 2GB PSS	prompt＋parse

要点は、**LiteRT-LM の高レベル API は活性を返さない**（text in / text out のみ）ことである。素の LiteRT-LM（案 D）に乗ると①活性抽出はできず、感情は②自己申告で取ることになる。②は第2章で弁別を確認済みなので、これで埋まる。GPU が実機で動き（第6章）、投機デコード（MTP）も LiteRT-LM が本家なので、現時点の採用は **案 D** である。速度が未達なら案 B（llama.cpp + Adreno OpenCL。Adreno には Qualcomm 公式の OpenCL backend があり、Gemma-4 E2B は乗る）へ分岐するゲートを引いてあったが、実機で GPU が通ったので分岐は不要になっている。

4.2　モデルはテキスト専用版にする

本アプリは音声をテキストに直してから LLM に渡すカスケード構成なので、画像・音声の入力機能は要らない。だから **テキスト専用版の Gemma-4 E2B** を使う。利点は二つある。重み自体が小さく済み（重み footprint で約 0.8GB まで）、メモリに優しい。そして、後の LoRA 学習・マージも、テキスト専用版のほうが順調に通る。

> **注意点**
> ・活性が要る感情方式（①）を前提に、ランタイムを決めない。LiteRT-LM の API は活性を返さないので、先に「①か②か」を決めてからランタイムを選ぶ。
> ・カスケード構成なら、最初からテキスト専用版を選ぶ。軽く、学習・マージも順調。

第5章　モデルを学習して載せる

実機側の構成と、学習まわりで実際にやったこと（と、捨てたこと）を書く。学習は GPU と高 RAM が要るので Google Colab で行う。

5.1　LoRA を学習して、捨てた

固定情報を重みに移す——その入口として、まず質問癖の抑制を LoRA でやってみた。結論から言うと、**学習までしたが、最終的に載せずに捨てた**。経緯を方法論として残す。

前提として、**LLM は「会話を続けたがる」傾向を持つ**。対話を続けることが学習で報われてきたので、放っておくと文末にたいてい質問を付けて相手にボールを返す（「〜なんだね。あなたはどう思う?」）。人間どうしの自然な会話を目指すなら、この質問癖は抑えたい。

抑え方には段階があった。まず、**否定指示（「質問するな」）は効かない**（RLHF で焼き付いた挙動は、プロンプトの否定では上書きできない）。一方、**キャラごとの few-shot 例で「こう返す」と示すと効いた**——これだけで実機の質問率が 100% から 33% に下がった。残りを学習層で消そうと、「質問しない感情受容」の教師データ 96 例で LoRA を焼いた。学習の要点だけ残しておく。

1. Gemma-4 E2B のテキスト専用版を base に QLoRA。チャットテンプレートは `apply_chat_template(..., tokenize=False)` で文字列化してからトークナイズする（先頭 BOS の二重付与を避ける）。loss が 3.5 → 1.0 へ順調に下がれば学習は良好。
2. 学習後、peft で base にマージし、`llama-quantize` で Q4_K_M。埋め込みの型は `--tensor-type "per_layer_token_embd.weight=q5_K"` で固定する（指定しないと Q6_K に昇格して肥大し、生成が約 4 倍遅くなる）。

ところが——**質問率は LoRA を入れても下がらなかった（33% → 37%）**。few-shot がすでに抑えていて、LoRA が押し下げる余地が小さかったのだ。質的な改善（敬語キャラのナレーション漏れ修正など）も、温度・シードを変えた再検証では安定して再現しなかった。

そして決定打として、後述のテキスト専用モデルへの移行と **メモリ管理を優先した結果、この学習済み LoRA は本番に載せず、いったん捨てた**。得た教訓は二つ。**(1) まず few-shot を試す**（否定指示は効かない、few-shot は効く）。**(2) LoRA は重い**——学習・マージ・量子化・再 export の一式が要り、効果が小さければ割に合わない。今回はそう判断して手放した。

> **注意点**
> ・質問癖を「質問するな」で直そうとしない（効かない）。few-shot で「こう返す」と示すと効く。
> ・LoRA を焼く前に、プロンプト（few-shot）で足りないかを先に確かめる。重いので、効果が小さければ割に合わない。
> ・量子化も Colab の高 RAM で行う。マージ済みモデルの量子化は、巨大な埋め込み（per_layer_token_embd）を f32 展開する瞬間に約 9.4GB を要求し、11GB の WSL は落ちる。

5.2　テキスト専用モデルを LiteRT-LM に載せる

実機には、マージ済み（または素の）テキスト専用モデルを `.litertlm` 形式に export して載せる。注意点として、LiteRT-LM の C++ ランタイムは LoRA に対応しているが、公式配布の `.litertlm` は LoRA 用の placeholder 無しで export されているため、**実行時に別の LoRA を動的注入することは今はできない**（公開 API 未整備＝issue #1188。対応は進行中）。だから LoRA を使う場合は base にマージしてから export する。なお Gemma は gated なので、再 DL には HF トークンとライセンス承諾が要る。

5.3　詰まった箇所：サンプラーの制御

実機で動かす段で詰まったのは、トークンを選ぶ **サンプラー**（次トークンの選択器）の部分だった。LiteRT-LM の配布物（Maven AAR）では、本体の `libLiteRt.so` が機能を絞った ABI で配られており、GPU サンプラーが必要とする多数のシンボルを一つも公開していない。このため **サンプリングだけが CPU にフォールバック**する（本体の計算は GPU で走る）。ここで時間を溶かさないために書いておくと、**別の `.so` を後から足しても構造的に直らない**――直すにはモノリシックな `.so` への総入れ替えか、ソースからのビルドが要る。GPU に寄せようと貪欲法（topK=1）にすると応答の多様性が死んで会話ゲームにならないので、**サンプリングは CPU のまま据え置く**。計算は GPU なので速度の主役は確保できており、既知の制約として受け入れている（サンプリングが CPU に残るぶん速度・発熱には響くが、現状のボトルネックとしては許容範囲だった）。

5.4　実測結果：メモリの実態

モデル選択でいちばん効くのがメモリなので、実測した数字を残す。

- **アプリ全体のメモリ（PSS）は約 2GB で、これ以上は下げにくい。** 公称の「テキスト専用なら 1GB 未満」は、CPU 側の活性ヒープだけを数えた **weight footprint** の話で、巨大な埋め込み（約 1.1GB）の mmap を除いている。Android の総 PSS はその mmap を計上するので約 2GB になる。**この埋め込みのぶんは必ず常駐する**ので、テキスト専用に替えても総 PSS は変わらない（マルチモーダル版も、テキスト利用時は画像・音声を on-demand で非ロードにするので同じ）。
- **だからメモリ最小化は「容量」ではなく「速度」のレバーとして効く。** メモリを圧迫すると重みが evict され（常駐 957MB → 128MB に崩落）、読み戻しのフォルトで decode が約 10.5% 遅くなることを A/B で確認した。decode の重みはページング可能なメモリ側にあり、GPU に完全 pin されているわけではない。低 RAM 機を出荷ターゲットに含めるなら、テ

キスト専用などでフットプリントを小さく保つことが、圧迫時の速度を守ることになる。

> **注意点**
> ・総 PSS が「1GB 未満」になると期待しない。埋め込みの mmap が必ず常駐するので、約 2GB がおおよその下限。削れるのは KV キャッシュ（maxNumTokens を絞る）程度。
> ・新規の特殊トークンを足さない。`.litertlm` への変換時に vocab 不一致でクラッシュしやすい。キャラ選択タグなども既存トークンの範囲で工夫する。

5.5　計画：メモリ管理の先にある一手

メモリ管理を詰めていくと、次の一手が見えてくる。――ここでまた LoRA が出てくるが、5.1 で手放したものとは使い道が違う。あちらは「質問癖を抑える」ための LoRA で、few-shot と効果が重なったから捨てた。今度は挙動を学習で変えるのではなく、**固定情報の"容器"** として LoRA を使う。すなわち、固定情報――キャラの性格・キャラごとの few-shot 例・日本語で会話させる指示・出力フォーマット――を毎ターンのプロンプトに積むのをやめ、**最初から LoRA に焼いてモデルに持たせる**、という方向だ。固定情報をプロンプトから外せば、第一声の prefill が軽くなり、KV キャッシュの圧迫も減るはずである（プロンプトに固定情報を積むと、第6章で見る prefill 遅延と KV 飽和の両方を悪化させる）。可変な情報――好感度（いまどのくらい好かれているか）・直近の文脈――だけをプロンプトで渡す。好感度は素の数値のまま渡せば、小型モデルもその目盛りを読み取って態度ににじませる（口調を命令で指定する必要はなく、情報として渡すだけでよい）。

これは **まだ実装していない計画段階**である。前例はある（固定ペルソナを学習で焼き、可変情報を実行時に渡す Fixed-Persona SLMs, arXiv 2511.10277, 2025）。ただし、8 キャラを 1 つの LoRA に詰めると性格が混線しうること、第5.2節のとおり LiteRT-LM では今のところキャラ別 LoRA の動的差し

替えができず 1 つのマージ済みモデルに畳む必要があること、といった制約がある。効果は、実装して実測してから判断する。

キャラを増やす方向では、**キャラ＝小さな LoRA** という持ち方が、本アプリの「1 セッション 1 キャラ」という構造によく合う。同時に 1 キャラ分しか要らないので、キャラ選択時にそのキャラの LoRA を効かせればよく、ベースを増やさずキャラを足していける（Doc-To-LoRA）。実装の候補としては、MediaPipe の LLM Inference API がオンデバイス（GPU）で Gemma の LoRA をサポートしており、キャラ選択時にその LoRA で初期化する形が考えられる。ただし未検証で、attention 層のみの LoRA で性格をどこまで焼けるか、テキスト専用モデルに LoRA を重ねたときの出力安定性は要確認である——という計画段階に留まる。

第6章　スマホで速く動かす

最初は実機で動かそうとして、**まともな速度では動かせなかった**。CPU だけで 2.7GB の Gemma-4 E2B（llama.cpp の量子化重みのファイルサイズ。第5.4節の PSS 約 2GB・重み footprint 0.8GB とは別の数え方）を回すと、会話に入って第一声が出るまで数十秒かかる（計測条件で 33〜42 秒。内訳はこの章で示す）。これを実用速度に持っていくまでの実装を書く。

6.1　第一声が遅い原因と、効いた一手

原因を切り分けると、prefill 中に CPU が 380〜696% に張り付く **compute 律速**だった。同じ Q4_K_M を x86（8コア）で測ると **長文 prefill が約 68 t/s**（本環境実測）出るので、ミドルレンジ機の CPU の演算能力が単純に足りていない（decode はマシン依存で、本環境では約 13 t/s）。効いた手は次の三つである。

1. **opening prefill を短縮する。** 開幕の長い systemPrompt（809 文字）の prefill が律速だったので、persona を user ロールに前置して投機的に先行 pin した（SENTINEL ＋ token-LCP）。開幕 prefill 23.2 秒 → 1.6 秒、入室から第一声 33 秒 → 11 秒（いずれも CPU・入室から発話まで）。
2. **固定情報を LoRA に移す（第5.5節の計画。未実装）。** 実現すれば prefill 対象そのものが薄くなり、さらに効くはずである（この削減はまだ実測していない）。
3. **演算を GPU へ逃がす。** 根本解はこれで、実機で動かして検証済みである。詳細は次節。

6.2　GPU オフロードの実装（実機で動作確認済み）

ミドルレンジ機の GPU（Adreno）で LiteRT-LM を動かすと、第一声（TTFT＝発話完了から最初の音が出るまで）が **CPU の 42 秒から GPU で**

1.6 秒になった。compute 律速は GPU で解ける、というのを実機で確認できた。MTP（投機デコード）が使える版（2026 年 5 月以降の `.litertlm`。第7章の注記）も載せたが、TTFT 改善の主因は GPU 化で、MTP 単体の寄与は別途 A/B で切り分ける必要がある。

GPU を有効にするには、AndroidManifest の `<application>` 直下に次の 2 行が要る。

```
<uses-native-library android:name="libvndksupport.so"
android:required="false" />
<uses-native-library android:name="libOpenCL.so" android:required="false" />
```

これが無いと、ベンダの OpenCL（`libOpenCL.so`）を dlopen できず、Engine 初期化が `INTERNAL` エラーで失敗する。targetSdk でも同梱 `.so` でもモデルでもなく、この宣言の有無が原因だった。

6.3　TTS の競合

補足として、音声合成（SBV2）は CPU で回り、LLM と decode を食い合う（TTS 有効時 2.37 t/s）。文単位で順に合成・再生してさばいている。SBV2 内部では BERT（DeBERTa）の処理が文の長さによらず約 860ms の固定の重さになっており、短文ではこれをスキップすると第一声を縮められる（1.62 秒 → 0.75 秒）。根本的には CPU 競合なので、TTS は軽いものに差し替える余地がある（第7章）。

> **注意点**
> ・`n_ubatch` を上げない。compute 律速の環境ではバッチを上げると遅くなる（64 → 256 で 34 秒 → 44 秒）。
> ・KleidiAI／i8mm／Flash Attention の有効化で速くなると期待しない（Q4_K_M では DOTPROD カーネルが選ばれ i8mm は不発、37 秒で不変）。
> ・GPU を使うなら AndroidManifest の `<uses-native-library>` 宣言を忘れない。これが GPU 初期化の絶対条件。

> ・Per-Layer Embedding（per_layer_token_embd）を GGUF から剥がない。forward で使うので、剥ぐと loader が弾くか出力がゴミ化する。

6.4　どの端末で要るか —— RAM より SoC

会場で「メモリは何 GB あればいいですか」と何度か聞かれた。答えは、**RAM には下限があるが、動くかどうかを決めるのは RAM ではなく SoC** ——とくに LLM を加速器（GPU／NPU）に載せられるかどうか——である。

RAM の下限。 本アプリのランタイムは PSS 約 2GB（埋め込みの mmap が常駐する。第5.4節）で、これに OS と他アプリが乗る。**6GB 以上が無難**で、4GB 機はメモリ圧迫で重みが evict され、読み戻しのフォルトで decode が落ちる（実測 −10.5%、第5.4節）。ただし RAM は「足りていれば速さは決めない」床であって、**床を超えたあとの速さを決めるのは SoC** である。

SoC で一桁変わる。 律速は演算（prefill／decode のスループット）で、LLM を CPU だけで回すか、GPU／NPU に逃がせるかで体感が一桁違う。本プロジェクトで実測した両極はこうだった。

- **Snapdragon（Adreno GPU）**：OpenCL で GPU オフロードが効き、第一声 1.6 秒・decode 52 t/s で快適（実測）。llama.cpp も LiteRT-LM も OpenCL バックエンドは Adreno 向けに調整されている。
- **Dimensity 6300（中位 MediaTek）**：使える加速器パスが無く CPU 一本になり、decode ～2 t/s・帯域天井 ～7.5 t/s で会話には厳しい（実測）。

ここから一般化すると、見るべきは世代の数字ではなく**加速パスの有無**である。

- **Snapdragon は「Adreno＋OpenCL が通るか」で見る。** OpenCL は **Adreno 710（Snapdragon 6 Gen 3・7s Gen 2 など）でも通る**ので、GPU パス自体はミドルレンジから使える。実用の下限はおおむね **6 Gen**

3 クラス、それ以上（7 Gen／8 Gen と上位 Adreno）は余裕、という目安になる（6 Gen 3 自体は未実測。OpenCL 対応はスペック確認）。

- **MediaTek は二極化している。** 中位以下（6300 など 6000 番台）は加速器パスが無く CPU 律速で厳しい。一方、**Dimensity 7300／8300／9000／9200／9300／9400 は Google の LiteRT NeuroPilot で NPU が第一級ターゲットになり**（2025 年 12 月）、Dimensity 9500 級では Gemma 3n E2B が prefill 1600 t/s・decode 28 t/s に達すると公表されている（ベンダ公表値・別モデル・本アプリでの自前計測ではない）。つまり「Dimensity だからダメ」ではなく「**6000 番台の中位 Dimensity がダメ**」で、7300 以上なら NPU パスが開きうる（本アプリ＝LiteRT-LM では未検証で、端末ごとの確認が要る）。

SoC の区分	加速パス	体感	根拠
Snapdragon 8／7 Gen・上位 Adreno	Adreno OpenCL	快適	実測 decode 52 t/s
Snapdragon 6 Gen 3〜7s（Adreno 710）	Adreno OpenCL（対応）	実用下限の目安	OpenCL 対応＝スペック・未実測
Dimensity 7300〜9400（NeuroPilot）	NPU（LiteRT）	速い・将来パス	公表 decode 28 t/s @D9500
Dimensity 6000 番台ほか中位以下	無し → CPU 律速	会話には厳しい	実測 decode ～2 t/s

注意点

・「メモリを増やせば速くなる」は誤り。RAM は 6GB 前後で床に達し、それ以上は速さを決めない。床を超えたら、速さを決めるのは SoC の加速パス。

・Snapdragon は世代名より「Adreno＋OpenCL が通るか」、MediaTek は「LiteRT NeuroPilot 対応（Dimensity 7300 以上）か」で見る。

・同じ "ミドルレンジ" でも、Adreno 機（GPU パスあり）と中位

> Dimensity 機（CPU 律速）では体感が一桁違う。RAM ではなくここで選ぶ。

第7章　音声まわりの選択肢（STT・TTS）

LLM の周辺、音声認識（STT）と音声合成（TTS）の選択肢を記録する。「どれが優れているか」ではなく、「どういう選択肢があり、どんな理由で何を選んだ／何を試したいか」を残す。

7.1　音声認識（STT）—— Android 標準の音声認識を使う

現状は Android 標準の音声認識 API（`SpeechRecognizer`）を使っている。端末側の音声認識サービスがオンデバイス処理に対応していれば、追加モデルの配布なしでオフラインに動く——ただし実際にオフラインになるかは端末・言語パック・サービス実装に依存するので、出荷ターゲット端末では確認が要る。選んだ理由は Android ネイティブで追加配布なしに使えるからで、オンデバイス前提に最も摩擦が少ない。

一点だけ注意がある。オンデバイス専用の API（`createOnDeviceSpeechRecognizer`）は `AMBIENT_ONESHOT` で初期化され、明示的に区切らないと結果（`onResults`）が返らない。だから最初から通常の `createSpeechRecognizer`（端末の Google 音声認識サービスにバインドされる）を使う。

ただしこれは据え置きの第一候補にすぎない。筆者は以前から **ReazonSpeech** を使ってきた経緯があり、時間があれば複数の ASR を差し替えて比較したい。候補は次のとおり。

- **Parakeet 0.6B JP**：0.62B と軽量ながら、日本語 CER 14.74%（ひらがな CER 5.86%）と報告され、軽量ながら 2B 級に匹敵するとされる（公表値・自前計測ではない）。ミドルレンジ機でも現実的。
- **ReazonSpeech-k2-v2**：これまで使ってきたモデル（公表 CER 17.55%）。

いずれも、確信度の低い箇所を **ひらがなのまま LLM に渡す** と、漢字変換ミス由来の感情スコア誤爆が減る場面があった（後段の Gemma が文脈で補完する。筆者の観測）。

7.2　音声合成（TTS）

オンデバイスの本番 TTS は **Style-Bert-VITS2（SBV2）** を使っている。品質は高いが重く、Android の GPU/NPU には乗らず（バックエンドが CUDA/CoreML/DirectML のみ）CPU で LLM と競合する。より軽い選択肢としては、CPU 向けに軽量な **Piper Plus**（ONNX）があり、SBV2 が重すぎる端末向けのフォールバックとして見ている。

本番では実機ビルドした Rust 製の合成核で SBV2 を動かし、fp16＋mmap で常駐メモリを約 0.46GB に収めている。SBV2 を採ったのは**表現力**のためで、この水準の合成をオンデバイスで動かす手段が現状ほかに見当たらなかった――第6.3節の 860ms BERT という重さを承知のうえでの選択である。

なお声づくりの段階では、**Irodori-TTS** を Google Colab で使い、テキスト入力からサンプル音声を生成して各キャラの声の方向性を探った。これは声の設計用で、オンデバイスでは動かないため本番 TTS の差し替え候補ではない。

キャラの声は SBV2 を自前で学習して作る。学習ステップを増やすほど品質は上がるが、ある所で頭打ちになった。より軽い iSTFT デコーダ版を蒸留した実験では、s12k 付近まで回しても金属質な癖が残り、それ以上は改善しなかった（モデル側の天井）。効いたのはステップ数ではなく別のレバー――iSTFT の n_fft を上げる・学習データを足す・warm-start（学習済みの状態から始める）――だった。回せば良くなるのは途中までで、頭打ちの先は手を変える必要がある。

> **注意点**
> ・2026 年 5 月以前にダウンロードした `.litertlm` は投機デコードが無効な旧版。MTP を使うなら再ダウンロードする（Gemma は gated）。

> ・高精度な ASR でも、静音下で数十回に一度は誤認識しうる。感情スコア側は、その程度の誤認識に強い設計にする。

おわりに

出発点は「感情はベクトルである。ならば数値化でき、蓄積でき、分岐に頼らないゲームが作れるはずだ」という一つの見立てだった。実際に作ってみて、感情ベクトルという概念は会話ゲームの信号源として機能し、機嫌の上下がこちらの直感とおおむね合う形で返ってくることを確認できた。スマホ単体・オフラインで、GPU に演算を逃がして実用的な速度まで漕ぎ着けられた（具体値と端末差は第6章）。

振り返ると、このゲームの目的は「会話の続き方」であって、キャラを揃えること自体ではない。だとすれば、8 キャラを用意するより、1 キャラの性格設定を切り替えられる形でも目的には足りていた。キャラ＝小さな LoRA という持ち方（第5.5節）は、その意味でも理にかなっている——という反省も、記録として残しておく。

最後に、本書をたどって同じものを作るときの手順を一枚にまとめておく。

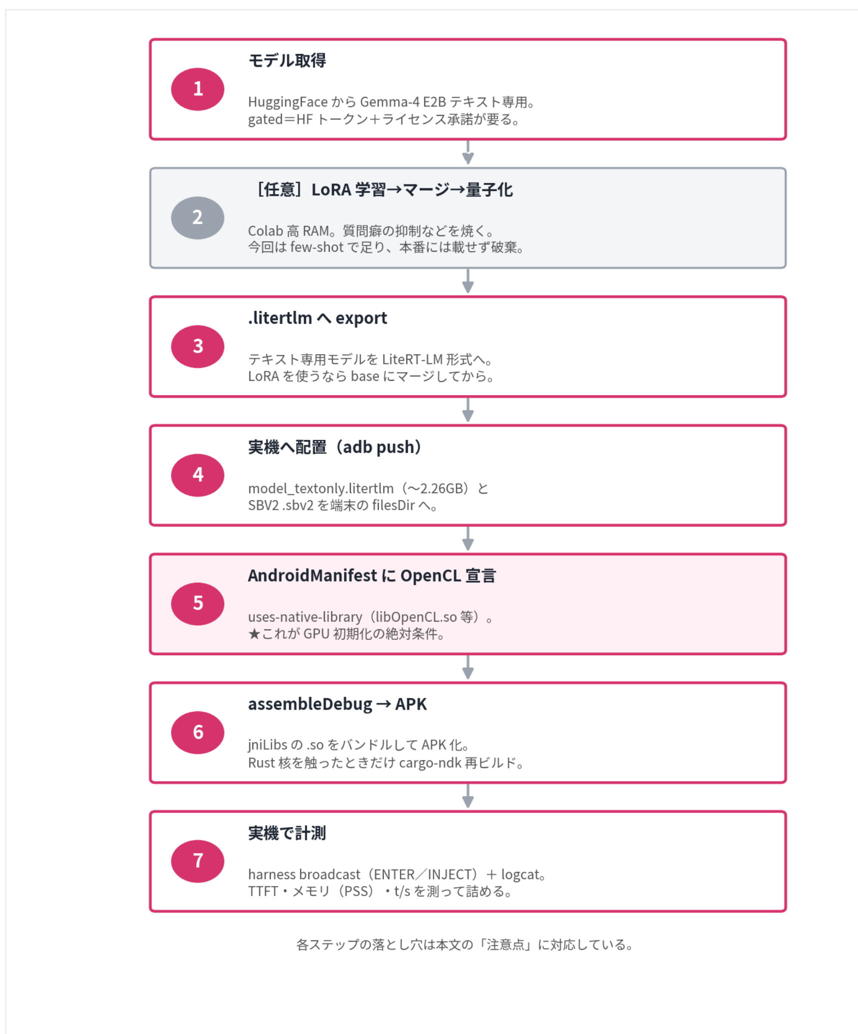

図8　再現手順の全体像

本書で並べた手順と注意点が、ローカル LLM の内部に踏み込んで何かを作るときの参照になれば幸いである。

参照

- milktea「生成 AI なんでも展示会 vol.5 展示解説／感情ベクトルと戯れる」note 記事
- Anthropic「Emotion Concepts and their Function in a Large Language Model」(2026)
- Tigges ら「Linear Representations of Sentiment in Large Language Models」(2024)
- Measuring and Controlling Persona Drift in Language Model Dialogs（arXiv 2402.10962）
- Examining Identity Drift in Conversations of LLM Agents（arXiv 2412.00804）
- Dynamic Context Adaptation for Consistent Role-Playing Agents with RAG（arXiv 2508.02016）
- States and Traits: Theories, Models, and Assessment（European Journal of Psychological Assessment 33(4), 2017）
- Kuppens & Verduyn「Emotional inertia: A key concept in the dynamics of emotions」(2017) ／ Kuppens ら「Emotional Inertia and Psychological Maladjustment」(Psychological Science, 2010)
- MediaPipe LLM Inference API（オンデバイス GPU で Gemma の LoRA に対応）
- Fixed-Persona SLMs with Modular Memory（arXiv 2511.10277, 2025）
- ゆうすけ『オープンソースで作る音声対話AI —— 2026 年版 オープンソース技術紹介と使いこなし』（初版 2026 年 4 月／技術書典頒布／X: @yusuke_kizuna）
- Pepper: SoftBank/Aldebaran/cocoro SB (2014)
- Gratch ら「Creating Rapport with Virtual Agents」(IVA 2007, Springer LNCS 4722, pp.125–138)

- Esterwood & Robert「Three Strikes and you are out!: 複数回の信頼違反と修復がロボットの信頼性に与える影響」(Computers in Human Behavior 142:107658, 2023)
- Shaikh ら「Navigating Rifts in Human-LLM Grounding: Study and Benchmark」(ACL 2025, arXiv:2503.13975)
- 「PaRT: Enhancing Proactive Social Chatbots with Personalized Real-Time Retrieval」(2025, arXiv:2504.20624)
- Lowe ら「Towards an Automatic Turing Test: Learning to Evaluate Dialogue Responses（ADEM）」(ACL 2017, arXiv:1708.07149)
- Huang ら「It Doesn't Hurt to Ask: Question-Asking Increases Liking」(JPSP 113(3):430–452, 2017)
- Grice「Logic and Conversation」（会話の協調原理・量の格率, 1975）