Gemini 3 Pro Image 示例与实现说明

Codex CLI 文档草案

2025 年 12 月 5 日

摘要

本手册只保留与「例子」相关的内容，通过三类典型对话场景说明 `gemini-3-pro-image-preview` 在 Codex CLI 中是如何被实际调用的。每个场景分为「用户视角」与「实现视角」，帮助有兴趣的开发者把 TUI/exec 中看到的行为和内部的数据结构、请求构造过程对应起来。

目录

1 模型与整体调用链路概览 **2**

1.1 图像模型预设：`gemini-3-pro-image-preview` 2

1.2 调用链路一览 2

2 图片在 Codex 中的实际流转（当前实现） **3**

2.1 用户图片输入：`UserInput::LocalImage` → `InputImage` 3

2.2 发送请求：`ContentItem::InputImage` → inlineData 4

2.3 模型输出图片：inlineData → `InputImage` → 本地文件 4

2.4 TUI 中的 `/open-image` 与「最近一张图」 5

3 参考图集合 RefSet 的抽象（设计方向） **6**

3.1 情况 1：RefSet 为空——纯文本模式 6

3.2 情况 2：RefSet 含 1 张图——单图参考 7

3.3 情况 3：RefSet 含多张图——多图组合 9

4 当前实现与 RefSet 抽象的关系 **12**

1 RefSet：一组有序的参考图

在后文所有示例中，我们都使用同一个抽象来描述「这一轮真正送给模型的图片集合」：

RefSet 在某一轮请求中，作为 inlineData 发送给 Gemini 的有序图片列表：

$$\text{RefSet} = [I_1, I_2, \ldots, I_k].$$

这里的 I_i 不关心具体是「本地文件」、「上一轮生成的图片」还是「从历史消息中抽取出来的图片」——它们在真正发给 Gemini 之前都会被转换成 data URL，进一步编码为 `GeminiInlineData`。

为了把例子说清楚，下面我们约定一个最小实现结构。真实项目的字段会更多，但核心思路基本一致：

Listing 1: 示意性的 RefSet 相关结构

```
#[derive(Clone)]
struct RefImage {
    id: String, // 供日志和调试使用的人类可读 ID
    path: std::path::PathBuf,
    mime_type: String, // 例如 "image/png"
    base64: String, // 将文件读入内存后编码出的 Base64
}

struct ConversationImages {
    /// 当前轮要用的有序参考图集合
    ref_set: Vec<RefImage>,
    /// 最近一次模型输出的图片（方便实现“隐式使用上一张图”）
    last_generated: Option<RefImage>,
}
```

在 TUI/exec 侧，每当用户执行与图片相关的操作时（例如 `/ref-image`、`/clear-ref`，或者模型生成了新图片），就会更新 `ConversationImages` 中的状态。真正构造请求时，只关心当前轮的 `ref_set`。

在所有示例里，我们使用同一个辅助函数来把 `RefSet` 编码进请求：

Listing 2: 根据 RefSet 构造当前轮的用户消息

```
fn build_user_message_with_images(
    user_text: &str,
    ref_set: &[RefImage],
) -> GeminiContentRequest {
    let mut parts = Vec::new();

    if !user_text.is_empty() {
        parts.push(GeminiPartRequest {
            text: Some(user_text.to_string()),
```

```
            inline_data: None,
            ..Default::default()
        });
    }

    for image in ref_set {
        parts.push(GeminiPartRequest {
            text: None,
            inline_data: Some(GeminiInlineData {
                mime_type: image.mime_type.clone(),
                data: image.base64.clone(),
            }),
            ..Default::default()
        });
    }

    GeminiContentRequest {
        role: Some("user".to_string()),
        parts,
    }
}
```

剩下的工作就是：针对不同场景，决定这一轮的 `ref_set` 应该是什么。下面用三个例子展开说明。

2 示例一：RefSet 为空——纯文本模式

2.1 用户视角：历史有图，但这一轮只聊文本

先看一个最常见的场景：

1. 第 1 轮：
   - 用户：画一只戴蓝色项圈的狗，背景是草地；
   - 模型：生成了一张图 `dog_blue.png`，并被 Codex 落盘；
   - TUI 显示「Generated image saved ...」。
2. 第 2 轮：
   - 用户：写一段介绍这只狗性格的文案；
   - 模型：返回一段文字描述。

3. 第 3 轮（当前轮）：

   - 用户：如果我要把这只狗写进儿童绘本，有哪些情节建议?
   - 用户没有执行 `/ref-image` 或其它图片相关命令。

从用户视角看，第三轮只是「继续聊文本」，即使历史里有图，也不希望模型再读一遍那张图。

2.2 实现视角：当前轮的 ref_set 为空

在这一轮开始时，ConversationImages 可能长这样：

```
ConversationImages {
    ref_set: vec![], // 当前轮没有主动指定要用的参考图
    last_generated: Some(dog_blue), // 历史上最近生成的一张图
}
```

关键点在于：**我们不做任何隐式图像附加操作**，这一轮只使用文本。因此构造请求时，直接传入空的 ref_set：

Listing 3: RefSet 为空时构造请求

```
let user_text =
    "如果我要把这只狗写进儿童绘本，有哪些情节建议? ";

let message = build_user_message_with_images(user_text, &[]);
let contents = vec![message];
```

构造出的 contents 概念上类似：

```
contents = [
    GeminiContentRequest {
        role: Some("user".to_string()),
        parts: vec![
            GeminiPartRequest {
                text: Some(
                    "如果我要把这只狗写进儿童绘本，有哪些情节建议? "
                        .to_string()
                ),
                inline_data: None,
                ..
            }
        ],
    }
];
```

可以看到，这一轮请求中不再包含任何 inlineData 图片。模型只能依靠之前的文本与思考链来「记住」那只狗的形象。

3 示例二：RefSet 只有一张图——单图参考

第二类场景是「当前轮只用一张图做参考」。这张图可能来自上一轮生成结果，也可能来自本地文件。

3.1 示例 2-A：基于上一轮生成的图继续改图

用户视角

1. 第 1 轮：
   - 用户：画一只戴蓝色项圈的狗，背景是草地；
   - 模型：生成 `dog_blue.png`；
   - TUI 落盘并提示：

     ```
     Generated image saved
     images/…/000000.png · run /open-image to open it
     ```

2. 第 2 轮（当前轮）：
   - 用户直接输入：把刚才这张图里的项圈改成红色。
   - 用户没有执行 `/ref-image`。

从用户角度看，这是「基于上一张输出继续改图」。理想体验是：即使只发送文本，模型也能看到上一张图。

实现视角：隐式使用 `last_generated`

在第 1 轮结束时，我们已经从 SSE 流中解析出模型返回的 inlineData 图片，解码为本地文件，并更新对话状态：

```
fn on_model_generated_image(conv: &mut ConversationImages, image: RefImage) {
    conv.last_generated = Some(image);
    // 是否自动把它放进 ref_set 取决于设计，这里选择“仅记录，不自动使用”
}
```

第二轮开始时，TUI 收到用户的纯文本输入。此时可以有一段「隐式附图」逻辑：

Listing 4: 示意性的隐式“上一张图”规则

```
fn maybe_attach_last_image(
    conv: &mut ConversationImages,
    user_text: &str,
    has_explicit_ref_command: bool,
    has_inline_images: bool,
) {
    if has_explicit_ref_command {
        return; // 用户已经显式指定参考图，尊重显式选择
    }

    if has_inline_images {
        return; // 当前轮输入里已经带了图片，不再额外附加
    }

    if !conv.ref_set.is_empty() {
        return; // 会话里已经有手动设定的 RefSet
    }

    if let Some(image) = &conv.last_generated {
        conv.ref_set.push(image.clone());
    }
}
```

经过这一步后，当前轮的状态大致是：

```
ConversationImages {
    ref_set: vec![dog_blue.clone()], // 唯一参考图
    last_generated: Some(dog_blue),
}
```

构造请求时，传入这个 `ref_set` 即可：

Listing 5: 单图参考时构造请求

```
let user_text = "把刚才这张图里的项圈改成红色。";
maybe_attach_last_image(&mut conv, user_text, false, false);

let message = build_user_message_with_images(user_text, &conv.ref_set);
let contents = vec![message];
```

对应到具体的 `contents` 形状大致如下：

```
contents = [
```

```
    GeminiContentRequest {
        role: Some("user".to_string()),
        parts: vec![
            GeminiPartRequest {
                text: Some("把刚才这张图里的项圈改成红色。".to_string()),
                inline_data: None,
                ..
            },
            GeminiPartRequest {
                text: None,
                inline_data: Some(GeminiInlineData {
                    mime_type: "image/png".to_string(),
                    data: "<dog_blue_base64>".to_string(),
                }),
                ..
            },
        ],
    },
];
```

这样，模型既能看到用户的自然语言指令，也能看到上一张输出图的像素内容，完成「基于上一张图的改图」。

3.2 示例 2-B：使用本地 dog.png 作为唯一参考图

用户视角

假设用户本地有一张 ~/Pictures/dog.png，希望以它为参考生成 Q 版风格的插画。在 TUI 中可以这样操作：

```
/ref-image  /Pictures/dog.png
帮我把这只狗画成 Q 版风格，背景是蓝天草地。
```

从用户视角看：/ref-image 这条命令指定了「这一轮只用这张图做参考」。

实现视角：/ref-image 设置新的 RefSet

当 TUI 解析到 /ref-image 时，可以调用一个类似下面的处理函数：

Listing 6: 示意性的 /ref-image 处理逻辑

```
fn handle_ref_image_command(
    conv: &mut ConversationImages,
```

```
    paths: Vec<std::path::PathBuf>,
) -> anyhow::Result<()> {
    conv.ref_set.clear();

    for (index, path) in paths.into_iter().enumerate() {
        let bytes = std::fs::read(&path)?;
        let base64 = base64::engine::general_purpose::STANDARD.encode(&bytes);

        let mime_type = infer::get(&bytes)
            .map(|t| t.mime_type().to_string())
            .unwrap_or_else(|| "application/octet-stream".to_string());

        conv.ref_set.push(RefImage {
            id: format!("ref-{index}"),
            path,
            mime_type,
            base64,
        });
    }

    Ok(())
}
```

这里有几个实现要点：

- **读取文件并缓存 Base64**：避免每一轮都重新读盘、重复编码；
- **mime 类型推断**：使用类似 `infer` 这样的库，根据文件头推断 `image/png` / `image/jpeg` 等，推断失败时回退为 `application/octet-stream`；
- **重置 RefSet**：每次执行 `/ref-image` 都替换当前 RefSet，确保「本轮只用这些图」的语义清晰。

之后，用户输入的自然语言会作为 `user_text`，和刚刚设置好的 `ref_set` 一起传入 `build_user_message_with_images`：

```
let user_text =
    "帮我把这只狗画成 Q 版风格，背景是蓝天草地。";

let message = build_user_message_with_images(user_text, &conv.ref_set);
let contents = vec![message];
```

此时构造出的请求与上一小节类似，只不过 inlineData 中的图片来自本地文件 `dog.png` 而不是上一轮生成结果。

4 示例三：RefSet 含多张图——多图组合

第三类场景是「一次性给模型多张图，让它按约定组合使用」。

4.1 示例 3-A：用 A 当背景，用 B 当素材

用户视角

用户本地有两张图：

- bg.png：一张房间背景图；
- dog.png：一只站立狗的照片。

希望让模型「把狗贴到背景里」。在 TUI 中可以输入：

```
/ref-image bg.png dog.png
請把第二張圖里的狗抠出來，放到第一張圖房間的中央，光影保持一致。
```

用户约定「第一张 / 第二张」的说法与命令里的顺序一致。

实现视角：RefSet 的顺序对应提示词中的“第几张图”

当解析 /ref-image bg.png dog.png 时，仍然可以复用 handle_ref_image_command，此时得到的状态类似：

```
ConversationImages {
    ref_set: vec![bg_ref, dog_ref], // I1 = 背景，I2 = 狗
    last_generated: Some(...), // 可有可无，与本轮无关
}
```

构造请求时：

```
let user_text =
    "請把第二張圖里的狗抠出來，放到第一張圖房間的中央，光影保持一致。";

let message = build_user_message_with_images(user_text, &conv.ref_set);
let contents = vec![message];
```

展开后，parts 中的顺序就是：

1. 一段文本，描述「第二张图里的狗」「第一张图房间的中央」等关系；
2. 第一张 inlineData：背景 bg.png；
3. 第二张 inlineData：狗 dog.png。

模型可以据此理解「第一张」「第二张」分别是哪张图，从而完成组合操作。

4.2 示例 3-B：多轮使用同一个 RefSet

用户视角

很多工作流会希望「设定一组参考图后，连续几轮都以此为基础」。例如：

1. 第 1 轮：
   - 用户执行：`/ref-image x01.png x02.png`；
   - 说明：用第一张做背景，第二张做角色，先生成一张基准图。
   - 模型输出 `result1.png`。
2. 第 2 轮：
   - 用户直接输入：再稍微调亮一点，让狗的毛发更有质感。
   - 用户没有执行新的 `/ref-image` 或 `/clear-ref`。

直觉上的期望是：`x01.png` 和 `x02.png` 仍然是当前轮的参考图，而上一轮生成的 `result1.png` 只是「历史事实」，不会自动加入 RefSet。

实现视角：RefSet 是会话级状态

要实现这种「多轮复用同一个 RefSet」的行为，一个简单做法是：

- `ConversationImages.ref_set` 是会话级状态；
- `/ref-image ...` 会**覆盖**这个状态；
- `/clear-ref` 会**清空**这个状态；
- 普通文本输入不会改变 `ref_set`。

对应的逻辑非常直接：

Listing 7: 清空当前 RefSet 的示意实现

```
fn handle_clear_ref_command(conv: &mut ConversationImages) {
    conv.ref_set.clear();
}
```

在没有执行 `/clear-ref` 的情况下，第二轮开始时的状态仍然是：

```
ConversationImages {
    ref_set: vec![x01_ref, x02_ref],
    last_generated: Some(result1_ref),
}
```

因此构造请求时，只需要继续使用同一个 `ref_set`：

```
let user_text =
    "再稍微调亮一点，让狗的毛发更有质感。";

let message = build_user_message_with_images(user_text, &conv.ref_set);
let contents = vec![message];
```

这样，模型在每一轮都能看到同一组参考图，只是用户不断调整提示词，而无需每轮重复指定图片。

5 如何把示例嵌入现有 Codex 调用链路

前面的示例只关注「这一轮如何决定 RefSet，并据此构造 `GeminiContentRequest`」。在真实项目中，它们需要嵌入到 Codex CLI 现有调用链路中，通常可以按以下方式划分职责：

- **TUI/exec 前端**：负责解析用户输入（包括 `/ref-image`、`/clear-ref` 等命令），维护 `ConversationImages` 状态，并在每一轮调用核心逻辑前，把「本轮用户文本」和「当前 `ref_set`」打包好；
- **protocol 层**：维持统一的中间表示，例如 `ContentItem::InputText` / `InputImage`，使得不同提供方（OpenAI / Gemini / 本地模型）共用一套结构；
- **Gemini 客户端 (core 层)**：在构造请求时，针对 Gemini 的 API 需要的 `contents` 结构，把文本和图片转换成 `GeminiContentRequest` 与 `GeminiInlineData`。

本手册中的代码片段刻意只展示与例子直接相关的那一小段实现思路，方便感兴趣的开发者「顺着例子」把 TUI/exec 中的行为和内部逻辑对上号。在此基础上，可以进一步结合实际代码库中的类型和模块划分，对照实现或调整细节。