# 整合 GDB 進行動態架構視覺化的深度研究報告：方法論、工具鏈與實踐

## 執行摘要

在現代軟體工程中，理解複雜系統（特別是使用 C、C++、Rust 等系統語言編寫的應用程式）的架構，往往面臨著靜態代碼定義與動態執行行為之間的巨大鴻溝。開發者通常依賴靜態分析工具來獲取類別圖或呼叫圖，但這些工具無法捕捉執行時期的多型派發（Polymorphic Dispatch）、動態記憶體配置模式、執行緒互動以及競態條件（Race Conditions）。GNU Debugger (GDB) 傳統上被視為一種微觀的除錯工具，用於隔離錯誤和檢查狀態。然而，透過其強大的 Python API 擴充機制與機器介面（Machine Interface, MI），GDB 具備轉型為「動態架構恢復引擎」的潛力。

本報告旨在詳盡探討如何將 GDB 結合現有的圖表表示法（如 UML 循序圖、Graphviz DOT 有向圖、火焰圖等），以實現程式碼架構的快速理解。分析範圍涵蓋從底層的 GDB Python 自動化腳本，到中層的日誌解析與轉換工具，再到上層的整合式視覺化前端。報告將深入剖析如何利用這些工具生成動態呼叫圖（Dynamic Call Graphs）、物件關聯圖（Object Relationship Diagrams）以及時序互動圖（Sequence Diagrams），並評估各種方法在效能開銷、資訊保真度與操作複雜度上的權衡。

## 1. 動態架構恢復的理論框架與挑戰

軟體架構的理解不僅僅是閱讀源代碼的目錄結構或類別定義，更關鍵的是掌握組件在執行時期的互動方式。這種「動態架構」包含了實際執行的控制流、記憶體中的資料結構形態，以及物件之間的訊息傳遞時序。

### 1.1 靜態分析的侷限性與動態分析的必要性

靜態分析工具（如 Doxygen、clang-uml）擅長建立程式碼的靜態索引，能夠生成標準的 UML 類別圖。然而，靜態分析存在「抽象的不精確性」。例如，在 C++ 中，靜態分析器可以看到對虛擬函式 Base::doWork() 的呼叫，但在缺乏複雜指標分析的情況下，無法確定執行時期究竟是 DerivedA::doWork() 還是 DerivedB::doWork() 被呼叫。此外，靜態工具無法視覺化系統的「狀態」：一個包含三個節點的連結串列與一個包含百萬個節點的連結串列，在靜態類別圖中看起來是一樣的，但其對系統效能與架構的影響卻截然不同 1。

動態分析則通過觀察程式的實際執行來填補這一空白。它能夠捕捉真實的執行路徑、具體的變數數值以及精確的時序關係。GDB 作為介於二進位執行檔與作業系統之間的橋樑，擁有對程式執行狀態、記憶體佈局與處理器暫存器的完全存取權限。將 GDB 從「修復錯誤」的工具轉變為「探索架構」的工具，是實現動態架構恢復的核心策略 3。

### 1.2 GDB 在架構視覺化中的角色演變

傳統上，GDB 的使用者介面是基於文字的命令列，這對於巨觀架構的理解構成了認知負載。開發者需要在大腦中將線性的文字追蹤（Trace）重組為二維或三維的架構模型。隨著 GDB 7.x 版本引入 Python API，這一局勢發生了根本性的轉變。Python API 允許開發者編寫腳本來自動化 GDB 的操作，遍歷內部的 C++ 資料結構，並將這些資訊匯出為結構化的格式（如 JSON、XML 或 Graphviz DOT），從而使得外部視覺化引擎能夠渲染出人類可讀的圖表 4。

這種轉變催生了「圖表即代碼」（Diagrams as Code）在除錯領域的應用。開發者不再手繪架構圖，而是由 GDB 在執行過程中自動生成反映真實行為的圖表。這種方法不僅提高了理解速度，還保證了架構文件與實際代碼的一致性 6。

## 2. GDB 可程式化層與資料提取機制

要理解如何利用 GDB 生成圖表，必須深入了解其內部的可程式化機制。GDB 並非僅是一個黑盒工具，它提供了一系列介面供外部工具提取資料。

### 2.1 GDB Python API 的架構洞察力

GDB 的 Python API 是連接底層除錯資訊與上層視覺化邏輯的關鍵介面。它提供了幾個核心物件，使得架構資料的提取成為可能：

* **gdb.Value 物件**：這是最基礎的抽象，代表了受除錯程式中的一個變數。透過這個物件，Python 腳本可以像存取 Python 物件屬性一樣存取 C/C++ 的結構成員（例如 node['next']），而無需手動解析記憶體位址或處理指標運算。這對於生成「物件關聯圖」至關重要 8。
* **gdb.Type 物件**：允許腳本檢查變數的型別資訊，包括繼承關係、模板參數與欄位定義。這使得腳本能夠動態地識別多型物件的真實型別，解決靜態分析無法處理的動態繫結問題 5。
* **gdb.events 事件監聽**：腳本可以註冊回呼函式（Callback）來監聽特定事件，如「暫停」（Stop）、「繼續」（Continue）或「斷點命中」。這是生成「循序圖」的基礎，因為它允許在每個關鍵執行點自動記錄日誌，而無需人工干預 9。

### 2.2 GDB 機器介面 (MI) 與前端整合

除了 Python API，GDB 還提供了機器介面（Machine Interface, MI），這是一種專為軟體互動設計的基於文字的通訊協議。許多圖形化前端（如 gdbgui、VisualGDB）都是透過 GDB/MI 來驅動 GDB。MI 的輸出是結構化的（類似 JSON 的鍵值對），這使得前端工具能夠即時解析堆疊幀（Stack Frames）、執行緒列表與變數數值，並將其渲染為 GUI 元件 10。

然而，對於自定義的架構視覺化，直接使用 Python API 通常比解析 MI 輸出更為靈活且高效，因為 Python 腳本是在 GDB 的行程空間內執行，減少了行程間通訊（IPC）的開銷。

## 3. 控制流視覺化：動態呼叫圖 (Dynamic Call Graphs)

動態呼叫圖是理解程式「行為架構」的基石。它回答了「誰呼叫了誰」以及「呼叫頻率如何」的問題。與靜態呼叫圖展示所有「可能」的呼叫不同，動態呼叫圖展示的是「實際」發生的呼叫，這對於識別熱點路徑（Hot Paths）與無用代碼（Dead Code）至關重要。

### 3.1 方法論一：基於日誌的堆疊追蹤 (Backtrace Logging)

這是最通用且無需修改源代碼的方法。其核心思想是利用 GDB 自動化地在函式入口處收集堆疊資訊。

#### 3.1.1 實作機制

利用 GDB 的 rbreak（正規表達式斷點）功能，可以在所有感興趣的函式上設置斷點。例如，rbreak file.c:. 會在 file.c 中的所有函式設置斷點。接著，透過 commands 指令定義斷點命中時的自動化行為：

程式碼片段

set pagination off  
set logging file trace.log  
set logging on  
rbreak.  
commands  
 silent  
 backtrace 1  
 continue  
end  
run

上述腳本指示 GDB 在每次函式被呼叫時，靜默地列印當前堆疊幀（呼叫者與被呼叫者），然後立即繼續執行。這會生成一個包含函式呼叫序列的線性日誌檔 11。

#### 3.1.2 圖表生成與 gdb\_graphs

生成的日誌檔包含原始的文字資料，需要進一步處理才能轉化為圖表。工具 **gdb\_graphs** 12 正是為此設計。它包含一個 Python 腳本 gen\_graph.py，負責解析 GDB 的堆疊日誌，識別函式節點（Nodes）與呼叫邊（Edges）。

* **資料轉換**：腳本將日誌中的 Frame #1（呼叫者）與 Frame #0（被呼叫者）轉換為有向圖中的邊 Caller -> Callee。
* **視覺化渲染**：利用 Graphviz 函式庫生成 DOT 檔案，最終輸出為 SVG 或 PNG 圖片。
* **互動性**：生成的 SVG 圖表通常支援互動功能，使用者點擊某個節點即可高亮顯示其完整的呼叫鏈（父節點與子節點），這對於在龐大的呼叫圖中導航極為有用。

**架構洞察**：這種方法能清晰地揭示軟體的「分層架構」，讓開發者看到高層業務邏輯如何一步步穿透至底層系統呼叫。然而，由於每個斷點都會觸發 GDB 與被除錯程式之間的上下文切換（Context Switch），其執行效能開銷極大，僅適用於小規模追蹤或非時間敏感的應用。

### 3.2 方法論二：動態追蹤點 dprintf 與高效能記錄

為了緩解斷點帶來的效能衝擊，GDB 引入了動態 printf (dprintf) 13。dprintf 是一種特殊的追蹤點，它在命中時會執行格式化輸出並自動繼續，通常不需要將控制權交還給 GDB 前端，從而大幅降低開銷。

* **指令範例**：dprintf my\_function, "Entry: %s called by %s\n", $\_func, $\_caller
* **架構應用**：透過在關鍵的架構邊界（如模組介面、API 入口）設置 dprintf，開發者可以生成高層次的「訊息流日誌」。後續的腳本可以解析這些日誌，生成 PlantUML 格式的圖表（詳見第 4 章），從而實現低干擾的架構監控。

### 3.3 方法論三：取樣分析與 gprof2dot

如果目標是理解系統的「效能架構」而非精確的邏輯流，取樣（Sampling）是一種更高效的策略。雖然 gprof2dot 15 通常與專門的分析器（如 perf, gprof, Valgrind Callgrind）配合使用，但 GDB 也可以作為一個簡易的取樣器。

* **GDB 取樣腳本**：編寫一個外部 Shell 腳本，迴圈呼叫 GDB 對執行中的行程進行 bt（Backtrace）操作，然後休眠極短時間。
* **堆疊摺疊 (Stack Collapsing)**：將收集到的多個堆疊追蹤合併，計算每個呼叫路徑出現的頻率。
* **視覺化**：gprof2dot 將這些統計資料轉換為加權的有向圖，其中節點顏色（熱圖）代表執行時間佔比。這能幫助架構師快速識別系統的效能瓶頸與熱點模組。

**表 1：GDB 動態呼叫圖生成方法比較**

| **特性** | **斷點與堆疊日誌 (Backtrace Logging)** | **動態追蹤點 (dprintf)** | **取樣分析 (Sampling / gprof2dot)** |
| --- | --- | --- | --- |
| **精確度** | 極高（捕捉每一次呼叫） | 高（取決於插樁位置） | 統計近似值 |
| **效能開銷** | 極高（每次呼叫皆切換上下文） | 中/低（取決於實作） | 低 |
| **適用場景** | 理解詳細的邏輯流與依賴關係 | 監控訊息傳遞與介面互動 | 識別效能瓶頸與熱點架構 |
| **主要工具** | gdb\_graphs, 自定義腳本 | PlantUML 解析器 | gprof2dot, flamegraph |
| **輸出格式** | Graphviz DOT, SVG | PlantUML, Mermaid | DOT, PNG, 火焰圖 |

## 4. 互動視覺化：循序圖與時序分析 (Sequence Diagrams)

循序圖（Sequence Diagrams）是描述分散式系統、通訊協定與多執行緒互動的標準語言。它強調的是「時間」維度上的訊息交換。GDB 可以透過記錄事件發生的順序，自動生成這種圖表，這對於理解併發架構與非同步流程至關重要。

### 4.1 從執行追蹤到循序圖的映射

將 GDB 的執行資料映射到 UML 循序圖的核心在於將程式實體對應到圖表元素：

* **生命線 (Lifelines)**：對應於執行緒（Threads）、類別實例（Objects）或模組（Modules）。
* **訊息 (Messages)**：對應於函式呼叫（Synchronous Calls）或訊息佇列的寫入（Asynchronous Messages）。
* **活化條 (Activation Bars)**：對應於函式在堆疊上的存活時間（從 Entry 到 Return）。

### 4.2 工具鏈：trace-to-sequence-diagram 與日誌解析

**trace-to-sequence-diagram** 17 是一個典型的解決方案，它提供了一套 Python 腳本框架，能夠將任意格式的文字日誌轉換為 **EventStudio** 或 **PlantUML** 的循序圖。

#### 4.2.1 整合 GDB 與 dprintf

為了使用此工具，開發者需要配置 GDB 的 dprintf 以輸出符合特定正規表達式（Regex）的日誌格式。

* **配置策略**：  
  在 GDB 中定義輸出典板：  
  (gdb) dprintf my\_func, " Caller=%s -> Callee=%s\n", $\_caller, $\_func
* **解析邏輯**：  
  在 trace-to-sequence 的配置檔 config.py 中定義 Regex：  
  traceRegex = r'\ Caller=(?P<caller>.\*) -> Callee=(?P<callee>.\*)'
* **生成圖表**：  
  工具會解析日誌，根據時間順序生成 PlantUML 代碼：  
  Caller -> Callee : my\_func  
  最終由 PlantUML 渲染為圖片或 PDF。

這種方法的優勢在於其高度的可自定義性。開發者可以決定「生命線」的粒度——是顯示每個物件的互動，還是將多個物件聚合為一個「組件」（Component），從而生成高層次的組件互動圖（Component Level Interaction Diagram）17。

### 4.3 協定層級的視覺化：SIPSequencer 案例

**SIPSequencer** 18 展示了針對特定領域（如 VoIP SIP 協定）的視覺化策略。雖然它是為 SIP 日誌設計的，但其架構原理完全適用於 GDB。

* **原理**：該工具讀取包含 Call-ID 的日誌，利用 awk 腳本過濾出特定通話的所有訊息，並使用 mscgen（Message Sequence Chart Generator）生成圖表。
* **GDB 應用**：架構師可以參考此模式，利用 GDB 腳本追蹤特定「交易 ID」（Transaction ID）在系統中的流轉，過濾掉不相關的背景雜訊，生成專注於單一業務流程的循序圖。

### 4.4 自動化 PlantUML 生成腳本

對於希望「快速理解」的開發者，編寫一個輕量級的 GDB Python 腳本來直接輸出 PlantUML 語法是最直接的路徑。

Python

# GDB Python 腳本概念驗證：自動生成 PlantUML  
import gdb  
  
class SequenceTracer:  
 def \_\_init\_\_(self, filename):  
 self.file = open(filename, "w")  
 self.file.write("@startuml\n")  
  
 def on\_stop(self, event):  
 # 獲取當前堆疊幀  
 frame = gdb.newest\_frame()  
 func\_name = frame.name()  
 # 假設我們追蹤 caller -> callee  
 # 這裡需要更複雜的邏輯來判斷是 Call 還是 Return  
 self.file.write(f'User -> System : {func\_name}\n')  
  
 def close(self):  
 self.file.write("@enduml\n")  
 self.file.close()  
  
# 註冊事件監聽  
tracer = SequenceTracer("trace.puml")  
gdb.events.stop.connect(tracer.on\_stop)

這種腳本可以擴展以包含執行緒 ID 作為參與者（Participant），從而視覺化多執行緒之間的競爭與同步行為 19。

## 5. 資料結構與記憶體視覺化：物件關聯圖 (Object Graphs)

對於 C++ 架構師而言，架構不僅是行為，更是資料的組織方式。物件之間的指標關係、容器的巢狀結構以及記憶體的佈局，構成了系統的靜態骨架。GDB 透過直接存取記憶體，能夠繪製出真實的「物件關聯圖」。

### 5.1 記憶體遍歷與 Graphviz

靜態類別圖只能顯示 Class A *擁有* 一個指向 Class B 的指標，但無法顯示執行時期 Class A 實際上串接了多少個 Class B 實例，或者是否存在循環參照。GDB 的 Python API 允許開發者編寫「爬蟲」腳本來遍歷這些結構。

#### 5.1.1 VDB (Visual Debugger) 工具

**vdb** 21 是一個強大的 GDB 擴充套件，專注於資料結構的視覺化。

* **ftree 模組**：該模組允許使用者指定一個根指標（例如二元樹的根節點），然後遞迴地遍歷指標，生成 Graphviz DOT 檔案。它能夠視覺化樹狀結構或有向圖，直接反映記憶體中的資料鏈結。
* **指標鏈結 (Pointer Chaining)**：vdb 支援追蹤指標的深度，並根據記憶體區域（堆疊、堆積、程式碼段）對節點進行著色。這提供了極具價值的架構洞察，例如發現本應指向堆積的指標卻指向了堆疊（可能導致懸空指標錯誤）。

### 5.2 堆積記憶體視覺化 (Heap Visualization)

對於底層系統程式設計，了解記憶體分配器的行為是架構分析的一部分。

* **vHeap 與 gdb-heap**：這些工具 22 專注於 glibc 堆積分配器的視覺化。它們不展示高層物件，而是展示記憶體區塊（Chunks）、Arena 與 Free Lists 的佈局。
* **架構意義**：這有助於理解記憶體碎裂化（Fragmentation）的模式，以及識別潛在的記憶體洩漏架構（例如某個模組持續分配但未釋放特定大小的區塊）。

### 5.3 領域特定視覺化：OpenImageDebugger

**OpenImageDebugger** 24 展示了針對特定資料類型的視覺化能力。它專門用於視覺化記憶體中的影像緩衝區（如 OpenCV cv::Mat 或 Eigen 矩陣）。這提醒我們，GDB 的視覺化不僅限於圖表，還可以是影像或熱圖，這對於訊號處理或圖形應用的架構理解至關重要。

## 6. 整合式前端與「圖表即代碼」趨勢

雖然腳本提供了無限的靈活性，但整合式前端提供了更低的使用門檻。此外，「圖表即代碼」的興起為 GDB 的輸出提供了標準化的渲染目標。

### 6.1 瀏覽器與 GUI 前端

* **gdbgui** 10：提供基於瀏覽器的介面，雖然主要關注於傳統除錯（斷點、堆疊），但其視覺化的堆疊追蹤與執行緒視圖提供了基礎的架構概覽。
* **Seer** 25：基於 Qt 的前端，提供強大的陣列視覺化與記憶體視覺化功能。它支援 GDB 的「反向除錯」（Reverse Debugging），允許開發者在時間軸上前後移動，觀察架構狀態的演變。
* **VisualGDB** 26：這是一個商業工具，它在架構視覺化方面走得更遠。其「Code Explorer」功能能夠自動生成呼叫樹（Call Trees）與關係圖，並結合即時追蹤來顯示哪些路徑被實際執行。這代表了 IDE 整合方向的極致。

### 6.2 安全領域的視覺化工具：GEF 與 Pwndbg

在逆向工程與安全研究領域，架構的可視化更為關鍵。

* **GEF (GDB Enhanced Features)** 27：提供架構無關的命令介面，並透過 heap-view 等指令提供記憶體佈局的文字圖形化顯示。
* **Pwndbg** 28：類似於 GEF，但更專注於低階開發。它能視覺化暫存器狀態、堆疊幀與堆積結構。雖然主要是文字介面，但其資訊密度極高，能幫助架構師快速建立對底層系統行為的心智模型。

### 6.3 混合方法：靜態與動態的結合 (clang-uml)

最全面的架構理解往往來自靜態與動態的結合。

* **clang-uml** 1：這是一個基於 Clang 的靜態分析工具，能生成類別圖、循序圖與套件圖。
* **混合工作流**：架構師可以使用 clang-uml 生成「預期」的架構圖，然後使用 GDB 動態生成的圖表來驗證「實際」的行為。兩者的差異（Drift）往往揭示了未經授權的依賴、廢棄的代碼路徑或隱藏的耦合。

## 7. 進階工作流與未來展望：文字化追蹤 (Literate Tracing)

隨著「文檔即代碼」（Docs-as-Code）理念的普及，除錯領域也出現了 **「文字化追蹤」（Literate Tracing）** 的新趨勢。

### 7.1 TReX 與可執行文件

**TReX** 29 是一個前沿的研究系統，旨在將程式執行的追蹤轉化為可讀的文件。它的理念不是生成一個巨大的日誌檔，而是生成一份包含互動式圖表、代碼片段與狀態說明的 HTML 或 PDF 文件。GDB 在此過程中扮演資料來源的角色，透過腳本提取關鍵狀態，然後由 TReX 渲染為敘事性的文件。這對於複雜演算法的解釋與教學具有革命性意義。

### 7.2 時間旅行除錯 (Time Travel Debugging) 與 rr

**rr** 4 支援確定性的錄製與重播（Record and Replay）。結合 GDB，這改變了架構分析的時間維度。架構師不再需要擔心「海森堡錯誤」（Heisenbugs），而是可以錄製一次執行，然後在重播過程中任意次數地生成不同視角（Call Graph, Object Graph）的圖表，直到完全理解系統行為為止。

## 8. 結論與策略建議

回答使用者的提問：**現有的圖表表示法絕對可以結合 GDB 來快速理解程式碼架構。** 這不是單一工具的功能，而是一個由 GDB Python API 驅動的工具鏈生態系統。

### 8.1 建議的實踐路徑

根據不同的架構理解需求，建議採用以下工作流：

**表 2：針對不同架構視角的 GDB 視覺化策略**

| **架構理解目標** | **推薦圖表表示法** | **關鍵工具/腳本** | **核心 GDB 機制** | **產出格式** |
| --- | --- | --- | --- | --- |
| **控制流與依賴** (誰呼叫誰？) | **有向圖 (Directed Graph)** | gdb\_graphs 12, gprof2dot | rbreak, backtrace | Graphviz (DOT/SVG) |
| **邏輯與時序互動** (發生了什麼？) | **UML 循序圖 (Sequence Diagram)** | trace-to-sequence 17, SIPSequencer | dprintf, Regex Parsing | PlantUML, Mermaid |
| **資料結構與狀態** (記憶體長怎樣？) | **物件關聯圖 (Object Graph)** | vdb (ftree) 21, 自定義 Python 腳本 | gdb.Value, 指標遍歷 | Graphviz (DOT) |
| **記憶體佈局** (配置與碎裂化) | **堆積視覺化 (Heap Layout)** | vHeap 22, GEF | glibc 結構分析 | HTML, 文字圖形 |
| **綜合架構驗證** (預期 vs 實際) | **混合圖表** | clang-uml (靜態) + GDB (動態) | Clang AST + GDB Trace | PlantUML |

### 8.2 結語

GDB 已不再僅僅是一個命令列除錯器。透過適當的腳本與外部視覺化工具的結合，它成為了一個強大的動態分析平台。對於追求「快速理解」的開發者來說，掌握 gdb\_graphs 進行概覽、利用 trace-to-sequence 進行邏輯梳理，並使用 vdb 進行資料探勘，將是提升架構掌握能力的最有效途徑。未來，隨著 AI 與文字化追蹤技術的成熟，GDB 的輸出將更加語意化與圖形化，進一步縮短從二進位代碼到架構洞察的認知距離。

#### 引用的著作

1. bkryza/clang-uml: Customizable automatic UML diagram ... - GitHub, 檢索日期：2月 6, 2026， <https://github.com/bkryza/clang-uml>
2. Are there any software to generate UML Diagrams from analyzing code? - Stack Overflow, 檢索日期：2月 6, 2026， <https://stackoverflow.com/questions/4841312/are-there-any-software-to-generate-uml-diagrams-from-analyzing-code>
3. The Architecture of Open Source Applications (Volume 2)GDB, 檢索日期：2月 6, 2026， <https://aosabook.org/en/v2/gdb.html>
4. bgirard/gdb-rr-dashboard: Modular visual interface for GDB ... - GitHub, 檢索日期：2月 6, 2026， <https://github.com/bgirard/gdb-rr-dashboard>
5. The GDB Python API - Red Hat Developer, 檢索日期：2月 6, 2026， <https://developers.redhat.com/blog/2017/11/10/gdb-python-api>
6. Code Visualization: 4 Types of Diagrams and 5 Useful Tools - CodeSee, 檢索日期：2月 6, 2026， <https://www.codesee.io/learning-center/code-visualization>
7. Diagram as Code - Decoding Architectural Diagrams - DevConf.CZ 2025 - YouTube, 檢索日期：2月 6, 2026， <https://www.youtube.com/watch?v=Jsrs0OuW0_M>
8. Automate Debugging with GDB Python API - Memfault Interrupt, 檢索日期：2月 6, 2026， <https://interrupt.memfault.com/blog/automate-debugging-with-gdb-python-api>
9. Python scripts for GDB. Gdb is a very powerful tool, pairing it… | by ..., 檢索日期：2月 6, 2026， <https://medium.com/@tarun27sh/python-scripts-for-gdb-9b17ca090ac5>
10. gdbgui, 檢索日期：2月 6, 2026， <https://www.gdbgui.com/>
11. Call graph generation - clu2's notes, 檢索日期：2月 6, 2026， <http://publicclu2.blogspot.com/2013/05/call-graph-generation.html>
12. tarun27sh/gdb\_graphs: To visualize function call flow for a ... - GitHub, 檢索日期：2月 6, 2026， <https://github.com/tarun27sh/gdb_graphs>
13. Hacking Tips — Firefox Source Docs documentation - Mozilla, 檢索日期：2月 6, 2026， <https://firefox-source-docs.mozilla.org/js/hacking_tips.html>
14. apropos page on DragonFly - Polarhome, 檢索日期：2月 6, 2026， <http://www.polarhome.com/service/man/?of=DragonFly>
15. Gprof2Dot: a Python script to convert the output of many profilers into a dot graph. - Reddit, 檢索日期：2月 6, 2026， <https://www.reddit.com/r/Python/comments/ln85u/gprof2dot_a_python_script_to_convert_the_output/>
16. jrfonseca/gprof2dot: Converts profiling output to a dot graph. - GitHub, 檢索日期：2月 6, 2026， <https://github.com/jrfonseca/gprof2dot>
17. eventhelix/trace-to-sequence-diagram: Python scripts for ... - GitHub, 檢索日期：2月 6, 2026， <https://github.com/eventhelix/trace-to-sequence-diagram>
18. nickrobinson/SIPSequencer: Library used to generate SIP ... - GitHub, 檢索日期：2月 6, 2026， <https://github.com/nickrobinson/SIPSequencer>
19. It is possible to generate sequence diagram from python code? - Stack Overflow, 檢索日期：2月 6, 2026， <https://stackoverflow.com/questions/45238329/it-is-possible-to-generate-sequence-diagram-from-python-code>
20. PlantUML sequence diagrams, 檢索日期：2月 6, 2026， <https://plantuml.com/sequence-diagram>
21. PlasmaHH/vdb: A set of python visual enhancements for gdb. - GitHub, 檢索日期：2月 6, 2026， <https://github.com/PlasmaHH/vdb>
22. wes4m/vheap: Extendable Visualization & Exploitation tool ... - GitHub, 檢索日期：2月 6, 2026， <https://github.com/wes4m/vheap>
23. rogerhu/gdb-heap: Heap Analyzer for Python - GitHub, 檢索日期：2月 6, 2026， <https://github.com/rogerhu/gdb-heap>
24. OpenImageDebugger/OpenImageDebugger: An advanced in-memory image visualization plugin for GDB and LLDB on Linux, with experimental support for MacOS and Windows. Previously known as gdb-imagewatch. - GitHub, 檢索日期：2月 6, 2026， <https://github.com/OpenImageDebugger/OpenImageDebugger>
25. epasveer/seer: Seer - a gui frontend to gdb - GitHub, 檢索日期：2月 6, 2026， <https://github.com/epasveer/seer>
26. Using Code Explorer to Understand Complex Call Graphs ..., 檢索日期：2月 6, 2026， <https://visualgdb.com/tutorials/intellisense/codeexplorer/calls/>
27. hugsy/gef: GEF (GDB Enhanced Features) - a modern ... - GitHub, 檢索日期：2月 6, 2026， <https://github.com/hugsy/gef>
28. Pwndbg: Annotate disassembly code - Archive Project Details | Google Summer of Code, 檢索日期：2月 6, 2026， <https://summerofcode.withgoogle.com/archive/2024/projects/Ar1tOf4w>
29. Literate Tracing - arXiv, 檢索日期：2月 6, 2026， <https://arxiv.org/html/2510.09073v1>
30. cpplinks/debugging.tracing.md at master · MattPD/cpplinks · GitHub, 檢索日期：2月 6, 2026， <https://github.com/MattPD/cpplinks/blob/master/debugging.tracing.md>
31. rr: lightweight recording & deterministic debugging, 檢索日期：2月 6, 2026， <https://rr-project.org/>