

# § 14 PoC 4.5 — per-opcode tax の根本原因究明(static libpython vs shared .so 確定)

項目	値
対象 spec	spec_section_14_draft.md v0.1, § 14.9 PoC 計画 PoC 4.5(PoC 4 残課題)
実施日	2026-06-04
環境	WSL2 Ubuntu 24.04 / Rust 1.93.1 / CPython 3.12.3 / PyO3 0.22.6 / x86_64 / glibc 2.39
目標	PoC 4 で発見した「埋め込み CPython の定常 per-opcode が python3 CLI より ~18-20% 重い」の根本原因を特定。perf 不可環境でも、初期化経路を Py_RunMain 相当に寄せた比較ランナーと構造証拠で切り分ける
結果	☑根本原因を確定: /usr/bin/python3 は libpython を静的に埋め込む(非 PIC・直接コール)、我々の Isolate は共有 libpython3.12.so を動的リンク(PIC・GOT/PLT 間接化)。これは「--enable-shared Python は静的ビルドより ~10-30% 遅い」という既知現象そのもの。しかも修正可能(libpython を静的リンクすれば消える見込み)

## 0. 結論

PoC 4 は「同一 libpython3.12.so ・バイト同一コードなのに ~18% 遅い」という反直観的状态で原因を perf に委ねていた。PoC 4.5 でこの 前提が誤りだったことが判明し、原因が確定した:

- /usr/bin/python3.12 (8MB 実行ファイル)は libpython.so に依存していない — libpython を静的に埋め込んでいる。eval ループ \_PyEval\_EvalFrameDefault は実行ファイル本体の中 (0x5d6900)。
- 我々の Isolate( cli\_isolate / poc3\_isolate )は共有 libpython3.12.so を動的リンク。eval ループは .so 内( 0x119500 )。
- つまり両者は 別物の eval ループ機械語を走らせている。python3 側は静的・非 PIC(内部関数呼び出しが直接コール、global データアクセスが直アドレス、クロス関数最適化が効く)。我々側は 共有 .so ・PIC( -fPIC で内部参照が GOT/PLT 経由の間接化)。
- これは「--enable-shared (共有 libpython)ビルドは静的ビルドより ~10-30% 遅い」という CPython の有名な既知現象。実測の ~18-20% はその range にドンピシャ。

最重要: この tax はアーキテクチャ的に修正可能。Isolate runner が libpython を静的リンク(CPython を --disable-shared --enable-optimizations でビルドし libpython3.12.a をリンク)すれば PIC オーバーヘッドが消え、PoC 4 のクロスオーバー(重い計算で Isolate が遅くなる現象)は解消される見込み。

# 1. 切り分けの設計と実行

## 1.1 三者比較ランナー(初期化経路の排除)

PoC 4 §2.1 の有力候補「PyO3 Py\_InitializeFromConfig vs python3 Py\_RunMain の初期化差」を直接潰すため、**Py\_BytesMain** (= python3 CLI が叩く実エントリ)を Rust から呼ぶランナー **cli\_isolate** を新設( src/bin/cli\_isolate.rs )。これは python3 と完全に同一の CPython 起動・スクリプト実行経路を通る Rust バイナリ。

ランナー	起動経路	libpython
/usr/bin/python3	Py_BytesMain (CLI)	静的埋め込み
<b>cli_isolate</b> (新)	<b>Py_BytesMain</b> (CLI と同一)	共有 .so (動的)
poc3_isolate	PyO3 with_gil + run_bound	共有 .so (動的)

## 1.2 三者タイミング(純粋整数ループ、M=3,000,000)

```
=== THREE-WAY TIMING, pure int loop, M=3,000,000 (default pymalloc) ===
python3 (static libpython) : 175.7 ms (1.000)
cli_isolate (Py_BytesMain) : 210.0 ms (1.195) <- 完全同一の CLI 初期化経路
poc3_isolate (PyO3)       : 206.8 ms (1.177) <- PyO3 経路
cli vs poc3               : 0.985
```

読み取り:

1. **cli\_isolate** は python3 と完全同一の **Py\_BytesMain** 経路なのに、なお ~19.5% 遅い。→ 初期化経路・スクリプト実行経路は原因ではない(PoC 4 候補 #2 を反証)。
2. **cli\_isolate** ≈ **poc3\_isolate** (0.985)。→ PyO3 の **with\_gil** / **run\_bound** 経路も原因ではない。
3. 残る構造差は「静的埋め込み libpython(python3)vs 共有 .so (両 Rust バイナリ)」の 1 点のみ。

## 1.3 構造証拠(static vs shared)

```
-- /usr/bin/python3 libpython dependency:
  (none) -> libpython STATICALLY embedded in the 8MB executable
-- our Isolate libpython dependency:
  libpython3.12.so.1.0 => /lib/x86_64-linux-gnu/libpython3.12.so.1.0
-- eval-loop symbol location:
00000000005d6900 T _PyEval_EvalFrameDefault ^ in python3 executable (static)
0000000000119500 T _PyEval_EvalFrameDefault ^ in shared .so (what we link)
```

file /usr/bin/python3.12 は 8,020,928 バイトの実行ファイルで **ldd** に libpython 行なし —— 静的埋め込みを裏付ける。我々のバイナリは .so 依存。eval ループは別アドレス・別ビルドの別コード。

## 1.4 二次寄与: pymalloc アリーナ

```
=== same three-way but PYTHONMALLOC=malloc (disable pymalloc arenas) ===
python3      : 189.8 ms  (1.000)
cli_isolate  : 213.1 ms  (1.123)  <- 1.195 -> 1.123 にギャップ縮小
poc3_isolate : 206.0 ms  (1.086)  <- 1.177 -> 1.086 にギャップ縮小
```

pymalloc を切ると tax が ~4 割縮む(1.195→1.123)。ヒープ/アリーナ配置も二次的に寄与するが、システム malloc でも ~9-12% 残る。主因は static/shared(PIC)で、pymalloc アリーナレイアウトは副次。

## 2. 原因のメカニズム(static vs shared / PIC)

- 共有 **.so** は **-fPIC** で全コードが位置独立。libpython 内部の関数呼び出し・global 変数/定数テーブルアクセスが GOT/PLT 経由の間接参照になる。eval ループのように 1 opcode あたり数回の内部参照が走る hot path では、この間接化が積み上がる。
- 静的埋め込みの **python3** 実行ファイルは、libpython のオブジェクトを実行ファイル本体にリンクするため、内部呼び出しが **PC 相対の直接コール**になり、リンク時にクロス関数最適化(より良いインライン/レイアウト)も効く。**-fno-semantic-interposition** 効果も最大化。
- 結果、まったく同じ C ソースの eval ループでも、静的ビルドの方が ~10-30% 速い。これは CPython コミュニティで繰り返し報告されている **--enable-shared** のコスト。本実測 ~18-20% は典型値。
- 我々が PoC 4 で置いた「共有 .so なので両者バイト同一」という前提は、**python3 がそもそも .so を使っていなかった**ため成立しなかった。

## 3. 含意と修正方針(PoC 5 アクション)

### 3.1 § 14 への朗報

PoC 4 は「Isolate は sustained compute で遅くなる( $M=10^6$  で  $1.375\times$ )」というクロスオーバーを発見し、これが Hybrid Isolate モデルの構造的限界に見えた。PoC 4.5 でそれは **構造的限界ではなく、libpython のリンク方式という調整可能なビルド選択**だと判明。

### 3.2 修正方針

Isolate runner を静的 libpython でビルドする:

1. CPython を **--disable-shared --enable-optimizations (PGO)+ 可能なら --with-lto** でビルドし **libpython3.12.a** を得る。
2. PyO3 を静的 libpython に向ける(**PYO3\_PYTHON** をその静的ビルドの python に向ける / **PYO3\_CONFIG\_FILE** で **lib\_dir** と静的指定)。
3. 再度 **run\_amortize.sh** / **probe\_rootcause.sh** を実行し、tax が ~1.0× 付近に縮むか検証。

予測: PIC タックスが消えれば、PoC 4 の **per-opcode tax は概ね解消** → Isolate は短命実行で速く (launch 経路が軽い)、sustained でも python3 と同等になり、**クロスオーバーが消える**。

### 3.3 副次

- `bench/amort_workload.py` の crossover 点(現状  $M \approx 10^4 \sim 10^5$ )は静的リンクで右方向へ大きく移動 or 消滅するはず。
- WASI/aarch64(PoC 5)で provisioning する CPython も、どうせ別ビルドするなら **静的・PGO 指定** で用意すれば一石二鳥。

## 4. 重要な観察

1. PoC 4 の前提誤りを実証で訂正できた: 「同一 .so」は誤り。 `/usr/bin/python3` は静的埋め込み。  
`ldd` 1 行で覆る前提を、 `Py_BytesMain` ランナーで初期化経路を排除した上で構造証拠まで取って確定。
2. `Py_BytesMain` ランナーが **decisive** だった: `python3` と完全同一の CLI 経路を Rust から再現してなお ~19.5% 遅い —— これ無しには「PyO3 のせい / 初期化のせい」を消せなかった。比較対象を 1 つ精密に増やすことで原因空間を一気に縮小。
3. 原因未確定 → 確定への昇格: PoC 4 は perf 不可で「仮説リスト + 延期」だった。PoC 4.5 は perf 無しでも **構造証拠(static vs shared)** で確定に持ち込めた。計測器が無くても設計実験で root-cause は取れる好例。
4. **honest finding** が修正可能性に転じた: 「Isolate は重い計算で遅い」という不利な finding が、根本原因究明により「ビルド方式で消せる」という**実装可能な改善 backlog** になった。
5. **bit-exact** は全工程で不変: 三者(`python3` / `cli_isolate` / `poc3_isolate`)とも同一出力。速度特性の調査は正しさを一切揺るがさない。

## 5. 配布物(PoC 3 クレートに追加)

```
engine/poc3_realworld/
├─ src/bin/cli_isolate.rs      Py_BytesMain で CLI 完全同一経路を通す Rust ランナー(切り分け用)
├─ probe_rootcause.sh         三者タイミング + static/shared 構造証拠 + pymalloc トグル(再現)
└─ (既存) src/main.rs(poc3_isolate) / samples / run_bit_exact.sh / run_speed.sh / run_amortize.sh / bench/
```

再現:

```
cd engine/poc3_realworld
bash probe_rootcause.sh          # 構造証拠 + 三者 M=3,000,000
```

## 6. § 14.9 PoC ロードマップ 進捗

Phase	内容	状態
☒PoC 0-3	最小実証 / Any / 動的構文 / 巷の実コード 20 + 速度	完了
☒PoC 4	startup 償却(クロスオーバー発見) + cross-platform 評価	完了
☒PoC 4.5	per-opcode tax 根本原因 = static libpython vs shared .so(PIC) 確定	完了(本書)
PoC 5	静的 libpython( <b>--disable-shared --enable-optimizations</b> )で Isolate を再ビルドし tax 解消を実証 / cross sysroot・WASI CPython provisioning / RustPython(Light tier)差分	次

PoC 4.5 完了。PoC 4 の per-opcode tax(~18-20%)の根本原因を確定: `/usr/bin/python3` は libpython を静的埋め込み(非 PIC・直接コール)、我々の Isolate は共有 `libpython3.12.so` を動的リンク(PIC・GOT/PLT 間接化)。Py\_BytesMain で CLI 完全同一経路を再現した `cli_isolate` も同じく ~19.5% 遅く、初期化経路・PyO3 経路を排除。「同一 .so バイト同一」という PoC 4 の前提が誤り(python3 は .so 非依存の 8MB 静的実行ファイル)。pymalloc アリーナは二次寄与(~4 割)。これは既知の `--enable-shared` コストで、修正可能——静的 libpython リンクで tax 解消・クロスオーバー消失を PoC 5 で実証予定。bit-exact は全工程不変。