

§ 14 PoC 8 — Light tier を wasm32-wasip1 にクロスコンパイルし実機実行(PoC 4/5 の WASI ブロックを突破)

項目	値
対象 spec	spec_section_14_draft.md v0.1, § 14.9 PoC 計画 PoC 8(移植性の実機確認)
実施日	2026-06-04
環境	WSL2 Ubuntu 24.04 / RustPython 0.5.0 / wasmtime 40.0.0 / Rust 1.93.1 / target = wasm32-wasip1
目標	PoC 7 の「Light tier(RustPython)は純 Rust ゆえ移植可能」という主張を実機で証明。RustPython を wasm32-wasip1 にクロスコンパイル し、wasmtime 上で PoC 3 の 20 サンプルを実行、(1) ネイティブ RustPython と挙動一致、(2) CPython との忠実度、(3) 速度を計測
結果	<input checked="" type="checkbox"/> wasm == ネイティブ RustPython 20/20 (クロスコンパイルは完全に挙動保存)/ wasm == CPython 19/20 (PoC 7 と同一、差は abc 例外文面のみ)/ 速度は事前コンパイルで CPython の ~5.7×。 PoC 4/5 で CPython 埋め込みが詰まったまさに WASI ターゲットで Light tier が完動

0. 結論

PoC 4/5 で CPython 埋め込みは **WASI で全滅**だった(`pyo3-ffi` が `PY03_CROSS_PYTHON_VERSION` を要求、ターゲット `libpython` 無し)。PoC 8 は、§ 14 Light tier の RustPython を **wasm32-wasip1 に実際にクロスコンパイル**し、その同じ WASI ターゲットで 20 サンプルを wasmtime 上で実行した:

- ・ **クロスコンパイルは挙動を完全保存**: wasm 出力は 20/20 でネイティブ x86_64 RustPython と stdout バイト一致。アーキテクチャを跨いでも 1 件もズレない。
- ・ **CPython との忠実度は不変**: 19/20 **bit-exact**(PoC 7 と同一、唯一の差は #04 abc の例外メッセージ文面のみ、意味論は一致)。
- ・ **WASI ブロックの突破**: `freeze-stdlib` で Python stdlib を `.wasm` に埋め込むため **libpython 不要**。CPython 埋め込みが詰まった所を Light tier がそのまま埋める。
- ・ **速度**: 事前コンパイル(`wasmtime compile`)で CPython の ~5.7×(ネイティブ RustPython の ~2×)。コールドの毎回 JIT は 17.8× だが、これは wasmtime が 22MB モジュールを毎起動コンパイルするコストで、`.cwasm` 事前コンパイル or 常駐ランタイムで償却できる。

§ 14 ティアリングの実機確定: Full tier(CPython 静的 Isolate, PoC 6)= x86_64 で 20/20 ・ python3 速度。Light tier(RustPython→WASM, PoC 8)= **どこでも 1 つの .wasm で動く** ・ 意味論 20/20 ・ 速度は数倍。移植性の主張が、CPython が動かない WASI 上で実証された。

1. クロスコンパイル構成

```
engine/poc8_wasm/ (rustpython-vm を freeze-stdlib で wasm32-wasip1 にビルド)
Cargo.toml: rustpython = { default-features=false,
    features=["freeze-stdlib","stdlib","stdio","importlib","host_env"] }
src/main.rs: InterpreterBuilder::new().init_stdlib().interpreter()
    → interp.run(|vm| { compile(Exec) → run_code_obj }) # finalize で stdio
flush
ビルド: cargo build --target wasm32-wasip1 --release → poc8_wasm.wasm (22 MB)
実行: wasmtime run --dir=<dir>:/s poc8_wasm.wasm /s/<sample>.py
```

成果物は単一 22MB **.wasm**、ホスト **libpython** 非依存。wasmtime (任意の WASI ランタイム / ブラウザ) で動く。

1.1 到達までの正直な経緯(WASI 向け feature 確定)

3 つの落とし穴を実機で潰した (再現者向けに記録) :

1. **InterpreterConfig** は誤り → **umbrella** は **InterpreterBuilder** + **InterpreterBuilderExt::init_stdlib()**。
2. **process::exit** で **stdout** が消える → **Interpreter::enter** でなく **Interpreter::run** (finalize で **sys.stdout/stderr** を flush)。
3. **feature** 不足: **default-features=false** で **stdio** が落ち **sys.stdout=None**(flush で **AttributeError**)。 **stdio** を足すと今度は **host_env** 無しで **_io** に **FileIO** が無く **dataclasses** → **io import** が失敗。 → **stdio** + **host_env** 両方必須(WASI は fd を持つので **host_env** の fd ベース **_io** が動く)。

最終 feature: **freeze-stdlib** + **stdlib** + **stdio** + **importlib** + **host_env** (**threading** / **ssl** は wasm 不適で除外)。

2. 差分計測(20 サンプル、wasmtime 実行)

各サンプルにつき wasm 出力を (a) ネイティブ RustPython、(b) CPython と SHA-256 比較。

sample	wasm==native	wasm==CPython
-----	-----	-----
01_metaclass_registry	same	PASS
02_init_subclass_registry	same	PASS
03_type_dynamic_class	same	PASS
04_abc_abstractmethod	same	DIFF (例外文面のみ、PoC 7 と同一)
05_slots_repr	same	PASS
06_property_validation	same	PASS
07_descriptor_set_name	same	PASS
08_getattr_lazy_proxy	same	PASS
09_setattr_frozen	same	PASS
10_callable_memoizer	same	PASS
11_functools_partial	same	PASS
12 singledispatch	same	PASS

13_wraps_counter	same	PASS
14_namedtuple	same	PASS
15_dataclass_postinit	same	PASS
16_enum_functional	same	PASS
17_defaultdict_counter	same	PASS
18_generator_yield_from	same	PASS
19_contextmanager	same	PASS
20_eval_exec_namespace	same	PASS

=== PoC 8 wasm summary ===

wasm == native rustpython : 20 / 20 (クロスコンパイルは挙動保存)
wasm == CPython python3 : 19 / 20 (Light-tier 忠実度、WASI 上で不変)

wasm == native 20/20 が肝: x86_64 から wasm へ跨いでも、RustPython の出力は 1 byte もズレない。クロスコンパイルが挙動を変えないことの実証。CPython との 19/20 も PoC 7 のネイティブ計測と完全一致(差は同じ abc 例外文面の 1 件)。

3. 速度(WASM Light tier のコスト)

/tmp/pure.py (整数ループ)で python3 / ネイティブ RustPython / wasm(wasmtime)を比較。

```
=== compute, pure M=100,000 ===
python3          18.2 ms   1.00x
native rustpython 49.2 ms   2.7x
wasm @ wasmtime(JIT) 324.2 ms 17.8x  <- 毎回 22MB モジュールを JIT コンパイル
wasm @ precompiled 108.8 ms  5.7x  <- wasmtime compile で .cwasm 化

=== startup (M=1) ===
python3 12.4ms | native rpy 30.4ms
wasm JIT 364.8ms -> wasm precompiled 103.0ms (JIT 除去で ~3.5x 改善)
```

- ・コードの 17.8× / 364ms は wasmtime が 22MB wasm を毎起動コンパイルするコストが支配的。
- ・wasmtime compile で .cwasm 事前コンパイルすると compute ~5.7×(ネイティブ RustPython の ~2×)、起動 103ms。常駐ランタイムや module cache を使えばさらに償却。
- ・wasm 段の追加コストはネイティブ RustPython 比 ~2×(wasm バイトコード vs ネイティブ機械語)。Light tier の価値は速度でなく普遍移植性(1 つの .wasm がブラウザ / エッジ / 任意 WASI で動く)。

4. ティア×プラットフォーム 到達マトリクス(PoC 4~8 総括)

	x86_64/glibc	aarch64	WASI/WASM
Full tier(CPython 静的 Isolate)	☑20/20・python3 同等以上(PoC 6)	☑ターゲット libpython 要(未)	☑PoC 4/5 で PYO3_CROSS ブロック

	x86_64/glibc	aarch64	WASI/WASM
Light tier(RustPython)	☒19/20・~3×(PoC 7)	▶ 純 Rust ゆえ可(同型、未実行)	☒19/20・実機実行(PoC 8)

- Full tier は速いが arch 縛り、Light tier は数倍遅いが arch フリー。WASI のように CPython が出せない所こそ Light tier の独擅場。
- aarch64 は Light tier なら `cargo build --target aarch64-...` で同型に出せる(本 PoC では WASM を実行対象に選択)。

5. 重要な観察

1. 移植性主張を「CPython が動かない場所」で実証: PoC 7 の「純 Rust だから運べる」を、PoC 4/5 が詰まった WASI でそのまま動かして証明。机上でなく実機(wasmtime)で 19/20。
2. クロスコンパイルは挙動を変えない(20/20 wasm==native): アーキテクチャ非依存に同一バイト出力。Light tier の「どこでも同じ結果」が arch 跨ぎでも成立。
3. WASM の速度コストは JIT コンパイルが主で償却可能: コールド 17.8× → 事前コンパイル 5.7×。短命大量起動なら `.cwasm` / 常駐必須、という運用知見。
4. WASI 向け RustPython の feature レシピを確定: `stdio` (stdout 接続)+ `host_env(_io.FileIO)` が必須、`Interpreter::run` (flush) 必須。再現可能な最小構成を文書化。
5. 正しさ計測は PoC 0~8 通して一貫: stdout SHA-256 一本で、Full=20/20、Light(native/wasm)=19/20(意味論 20/20)を横並び評価。

6. 配布物

```
engine/poc8_wasm/
├─ Cargo.toml          rustpython freeze-stdlib + stdio + host_env、wasm32-wasip1 向け
├─ src/main.rs         InterpreterBuilder→run(compile/run_code_obj)、finalize で flush
├─ run_wasm_diff.sh    20 sample を wasmtime 実行、wasm vs native vs CPython 分類
└─ target/wasm32-wasip1/release/poc8_wasm.wasm (22 MB、単一・libpython 非依存)
```

再現:

```
cd engine/poc8_wasm
cargo build --target wasm32-wasip1 --release
bash run_wasm_diff.sh
# 事前コンパイル: wasmtime compile poc8_wasm.wasm -o poc8.cwasm
#                  wasmtime run --allow-precompiled --dir=DIR:/s poc8.cwasm /s/
script.py
```

7. § 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 / 4.5 / 5 / 6	償却・根本原因・静的解消・ソース正規ビルド(Full tier 確立)	完了
☒PoC 7	Light tier(RustPython)忠実度 19/20・移植性主張	完了
☒PoC 8	Light tier を wasm32-wasip1 にクロスコンパイル・wasmtime 実機実行(WASI 突破)	完了(本書)
PoC 9	Full↔Light 自動振り分け(例外文字列依存 / hot-loop 判定で tier 選択)/ aarch64 Light tier の実機(gemu)/ ブラウザ WASM デモ	次

PoC 8 完了。§ 14 Light tier の RustPython を wasm32-wasip1 にクロスコンパイル(**freeze-stdlib + stdio + host_env**、22MB 単一 **.wasm**、ホスト libpython 非依存)し、wasmtime 上で PoC 3 の 20 サンプルを実行——wasm 出力はネイティブ RustPython と 20/20 byte 一致(クロスコンパイルが挙動保存)、CPython とは 19/20(PoC 7 と同一、差は abc 例外文面のみ)。PoC 4/5 で CPython 埋め込みが PYO3_CROSS で詰まった WASI ターゲットを、Light tier がそのまま完動して突破。速度は事前コンパイルで CPython の ~5.7×(ネイティブ RustPython の ~2×)、コードの 17.8× は wasmtime の毎回 JIT が主因で **.cwasm** 化で償却可。§ 14 二層構成(Full=速い・arch 縛り / Light=数倍遅い・arch フリー)が、CPython の出せない WASI 上で実機実証された。