6J-08

Pythonインタプリタへのマークスイープ型ガベージコレクタによる参照カウントの除去

付 高和 †　芝浦工業大学
福田 浩章 ‡　芝浦工業大学

1. はじめに

Python 言語は, 学習コストの低さや生産性の高さ, 著名ライブラリやドキュメント等エコシステムの強力さにより近年人気を伸ばしている [1]. Python は, ソースコードから各計算機に対応したバイナリへのコンパイルを必要とせず, インタプリタがソースコードを直接実行する動的言語である. Python の処理系には複数の実装が存在するが, C 言語で実装された CPython が広く一般に用いられている.

アプリケーションは, OS の提供するスレッド機構を用いて, 複数のタスクを並行処理させることで実行効率を高めることができる. CPython は, OS のスレッド機構を使用した並列処理機構を提供する一方, 複数スレッド間でメモリのデータ競合を防ぐ戦略としてグローバルインタプリタロック (GIL) による同時実行スレッド数の制限を行っている. よって, 計算機のマルチコア化が進むなか, CPython は計算資源を十分に活用できず, 実行効率の向上が難しい.

[2] では, トランザクショナルメモリ (TM)[3] の競合検出・ロールバック機能を用いてメモリ保護を行うことで, GIL を廃した複数スレッドの安全な同時実行が可能であることが示された. 一方で, CPython が用いる参照カウント方式のガベージコレクタは, TM における高頻度なトランザクションの失敗により, リトライや排他制御へのフォールバックによる実行効率の悪化を招くため, これを軽減する必要がある.

TM 使用時のトランザクション失敗を軽減するには, 既存の参照カウント方式のガベージコレクタを除去する必要があるが, これを行うと, アプリケーションが一度確保したメモリを OS に返却できなくなり, メモリリークが発生する.

そこで本研究では, メモリリークを防ぐ手段として, CPython へマークスイープ型ガベージコレクタ (GC) を実装し, 参照カウントを除去する手法を提案する.

Eliminating reference counting in Python interpreter by introducing mark and sweep garbage collector

†Takakazu Fu, Shibaura Institute of Technology
‡Hiroaki Fukuda, Shibaura Institute of Technology

2. アプローチと実装

以下のステップで動作するマークスイープ型 GC を提案する.

1. マーク: インタプリタに存在し, 他のオブジェクトより参照されている, もしくは参照される可能性のあるオブジェクトにフラグを設定する.
2. スイープ: インタプリタの全てのオブジェクトの集合を走査し, マークされていない (フラグが設定されていない) オブジェクトを削除し, メモリを解放する.

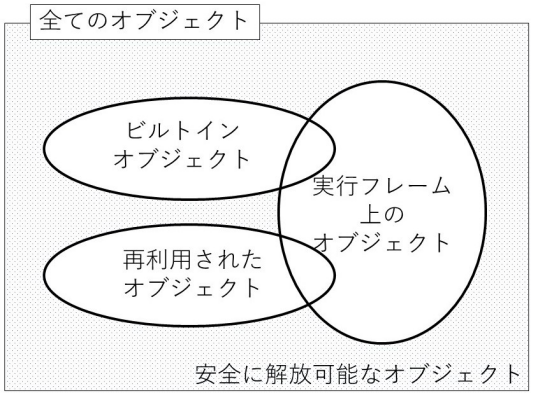

図 1: ガベージコレクタが扱うオブジェクト

図 1 は, GC が扱うオブジェクトを示している. 実行時に参照されているオブジェクト (実行フレーム上のオブジェクト) や, 実行時に参照される可能性のあるオブジェクト (ビルトインオブジェクト・再利用されたオブジェクト) をマークした後に, 全てのオブジェクトの集合に対してスイープを行うことで, 安全に解放可能なオブジェクトのみを処理することができる.

2.1 マーク

2.1.1 実行コンテキストの取得

図 2 は, CPython におけるインタプリタ・スレッド状態の管理構造を示している. CPython では, コード実

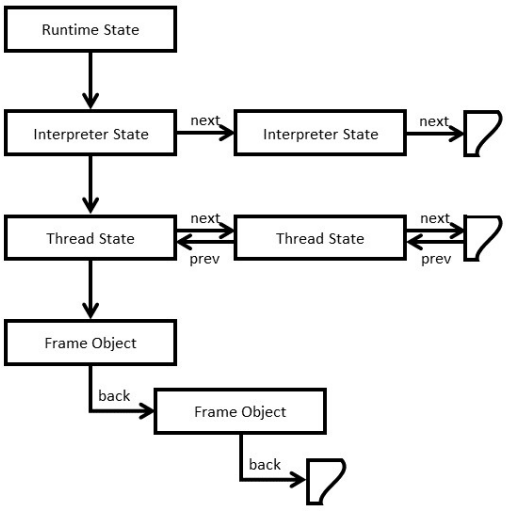

図 2: Runtime と PyFrameObject の関係

行中のスタックや変数等のコンテキストをフレームオブジェクト (PyFrameObject) に格納している. グローバルなランタイム構造体 (Runtime State) より, 全てのインタプリタ構造体 (Interpreter State), スレッド構造体 (Thread State) が持つフレームオブジェクトを取得することで, Python コードの実行中に参照されている全てのオブジェクトをマークすることができる.

2.1.2 参照される可能性のあるオブジェクト

コードの実行時に必ずしも参照されるとは限らないが, 常に参照される可能性のあるオブジェクトとして, ビルトインオブジェクトや, 再利用されたオブジェクトが挙げられる.

ビルトインオブジェクトは, int 型や str 型などの, インタプリタに埋め込まれたオブジェクトである. これらは一般的に静的に確保されたオブジェクトであるが, 全てのオブジェクトの集合に含まれているため, 誤って解放することのないようにマークする必要がある.

また, CPython では, 動作の最適化のために, 0 や 1 といった int 型 の小さな整数など, 頻繁に使用されることが予想されるオブジェクトをインタプリタ起動時にあらかじめ生成し, 特定の配列に格納して再利用している. これらのオブジェクトもあらかじめマークを行い, 実行コンテキストから参照されなくなった後も解放してはならない.

2.1.3 構造体メンバの巡回

オブジェクトの中には, 構造体のフィールドとして他のオブジェクトへの参照を持つものがあり, これをマークする必要がある. CPython では循環参照の検出のため, 型オブジェクトは自身の持つフィールドを巡回する tp_traverse 関数を実装することができるが, これは list や dict 型など, 循環参照を引き起こす可能性のある"コンテナ型" オブジェクトに対しての巡回のみを必須としており, 全てのフィールドを巡回することが保証されない. ただし, tp_traverse 関数においては, コンテナ型以外のオブジェクトを巡回しても副作用がないことより, 本研究においては, インタプリタの各 tp_traverse 関数に対して, 全てのフィールドを巡回できるよう追加の実装を行う.

2.2 スイープ

2.2.1 全てのオブジェクトの集合

スイープによって解放可能なオブジェクトを検出するためには, インタプリタが確保した全てのオブジェクトの集合が必要である. CPython では, インタプリタを --with-trace-refs フラグでビルドすると, 全ての Python オブジェクト構造体の先頭に Listings 1 に示すヘッダが追加され, 全てのオブジェクトが双方向連結リストを構成する. このリストを走査することで, インタプリタの持つ全てのオブジェクトに到達可能である.

Listing 1: Include/object.h

```
#define _PyObject_HEAD_EXTRA            \
    struct _object *_ob_next;           \
    struct _object *_ob_prev;
```

3. 研究状況と今後の予定

現在は, スイープ機構については実装を完了しており, マーク機構については汎用的な機構の実装を完了したところである.

マーク機構については, 構造体メンバの巡回について, 対象の型によって個別の実装が必要であり, 工数の多い実装であることから, 今後も継続して行う予定である.

参考文献

[1] Stack Overflow Developer Survey 2020: https://insights.stackoverflow.com/survey/2020

[2] Fuad Tabba: Adding concurrency in python using a commercial processor's hardware transactional memory support, In *ACM SIGARCH Computer Architecture News*, 2010, Volume 38, Issue 5

[3] Maurice *et al.*: Transactional memory: architectural support for lock-free data structures, In *ACM SIGARCH Computer Architecture News*, 1993, Volume 21, Issue 2

[4] Python 3.8.7 documentation: https://docs.python.org/3.8/

[5] Python Developer's Guide: https://devguide.python.org/