RogueWave SOFTWARE | TotalView

Python と C/C++の混合デバッグ

MIXED-LANGUAGE DEBUGGING
WITH PYTHON AND C/C++

(日本語参考訳)

PYPL [*1] を始めとするプログラミング言語の人気ランキングサイトによると、Python の人気は 10 年以上にわたって着実に伸びています。使いやすさ、明確な構文、サードパーティ製の豊富なドキュメント、多数の拡張ライブラリがあるため、Python はディープラーニング、人工知能、株式取引などさまざまな業界で使用されています。

Python の強みの一つは、C や C++コードで簡単に拡張できることです。拡張機能により、Python アプリケーションはレガシーなアルゴリズムにアクセスし、ハードウェアを活用したり特殊な計算を実行することができます。C/C++ による Python の拡張により、開発者はプログラムのさまざまな部分を結合して、混合言語によるアプリケーションを作成することができます。

混在言語アプリケーションにおいて言語障壁の間の相互依存性とデータ交換を理解しデバッグすることは、開発者にとって大きな課題です。この記事では、Python と C/C++の混在したアプリケーションを作成するための技術を簡単にリストアップし、開発者が言語障壁間の相互作用やレイヤ間でのデータのやりとりを理解するためにはどのようなデバッグ技術を使えばよいのかご紹介します。

---

*1 .http://pypl.github.io/PYPL.html

C や C++拡張モジュールを使用して Python を拡張する

Python で開発していると、既存のレガシーコードの活用、特殊なハードウェアリソースへのアクセス、高性能コードの使用など、さまざまな理由で既存の C や C++で書かれたコードにアクセスする必要がしばしば生じます。C/C++コードへのアクセスは、Python 拡張、つまりコンパイルされて共有ライブラリへとリンクされ、実行時に Python インタプリタによって動的にロードされる C/C++のルーチンを使うことで可能になります。

Python は、共有ライブラリ内の関数を呼び出すためのインフラストラクチャと、言語バリア間の C 互換データ型の交換を提供する外部関数ライブラリである ctypes を提供しています。なお、Ctypes は C を呼び出すための唯一の解決策ではありません。Python と C/C++間の呼び出しやデータ交換を容易にするための独自のアプローチが他にもたくさんあります。

Python C/C++グルー技術	説明
ctypes	Python 用の外部関数ライブラリ。
Cython	C 関数の呼び出しと、変数とクラス属性の C 型の宣言をサポートする Python 言語のスーパーセット。
SWIG	C および C++で書かれたプログラムと、Python を含むさまざまな高水準プログラミング言語を結ぶソフトウェア開発ツール。
CFFI	C コードを呼び出す Python の外部関数インタフェース。
PyQt/PySide および SIP	SIP は、C および C++ライブラリ用の Python バインディングの作成を容易にするツールです。
Boost.Python	C++と Python プログラミング言語との間のシームレスな相互運用性を可能にする C++ライブラリ。

この記事の後半では、Python と C/C++アプリケーションのデバッグについて説明しますが、最初に Python と C および C++を組み合わせてデバッグの準備を簡単に行う方法について検討します。簡単な SWIG の例を示します。オンラインの SWIG と Python の例を使用して、Python から C ルーチンを呼び出すことによって、数値の階乗を計算します。C 言語の階乗関数のシグネチャは、ヘッダーファイル example.h で定義されています。

```
int fact(int n);
```

example.h

実装は簡単な再帰ルーチンです。

```
#include "example.h"

int fact(int n) {
  if (n < 0){
    return 0;
  }
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

example.c

Python から C 言語への呼び出しを準備するために、SWIG は言語間にラッパーレイヤーを構築します。これは、C ヘッダーファイルを分析することによって行います。必要に応じて、独自の SWIG インターフェイスファイルを作成することで、きめ細かい制御を実現できます。

```
%module example

%{
#define SWIG_FILE_WITH_INIT
#include "example.h"
%}

int fact(int n);
```

example.i

すべてのファイルが準備できたら、SWIG を使って Python モジュールを作成する必要があります。この例では C を使用していますが、C++を使用している場合は、-c++オプションを SWIG コマンドライン引数に追加します。

```
# Generate the SWIG wrapper files
$ swig -python example.i
```

SWIG で Python モジュールを生成する

SWIG プログラムは、入力ファイル example.i から２つのファイルを生成します：example_wrap.c という名前の C ソースファイルと example.py という名前の Python ソースファイルです。これらのファイルを使って、言語間のグルー(接着剤)を生成します。グルーコードについては、後ほどプログラムをデバッグする時にもう少しみてみましょう。

これまでは、デバッガの下で Python や C/C++コードをデバッグするのに特別な変更は必要ありませんでしたが、Python から呼び出される C/C++コードを効果的にデバッグするには、Python インタプリタのデバッグバージョンをインストールする必要があります。たとえば、Ubuntu ベースのシステムでは、sudo apt-get python-dbg python-dev をインストールします。他の Linux ディストリビューションでは、パッケージマネージャーに同等のパッケージがあるかどうか確認してください。

Python 拡張機能を構築するには、いくつかのアプローチがあります。distutils を使用するか、手動でビルドします。Distutils は、拡張機能を適切に構築するために必要なフラグやその他のシステム依存関係を把握するツールです。簡単にするために、私たちは 2 つの gcc コマンドを使って拡張機能を手作業でコンパイルします。デバッグ用の拡張機能を準備するには、-g 引数を追加して C デバッグ情報を生成し、Python インタプリタヘッダファイルのデバッグバージョンへのインクルードファイル検索パスを指定します。

```
# Compile all files
$ gcc -g -fPIC -c example.c example_wrap.c \
     -I/usr/include/python2.7_d
# Link the files into shared library _example.so
$ gcc -shared example.o example_wrap.o -o _example.so
```

Python の拡張共有ライブラリを生成

Python 拡張モジュールのコンパイルにはいくつか注意すべき点があります。まず、共有ライブラリは、SWIG インターフェイスファイルで定義されたモジュールと同じ名前を付け、さらにアンダースコアを前に付ける必要があります。次に、Python ヘッダのデバッグバージョンでコンパイルされたモジュールは、通常のバージョンの Python インタプリタでは動作しません。これは、完全なデバッグ情報に使用されるシンボルが追加されたためです（[*2] Roger Hu、Python-dbg を使用して Python プログラムを開始できない理由、2013 年 6 月 6 日）。

Python のサンプル拡張をビルドすると、次の単純な Python プログラムを実行して試してみることができます。

```
def getFact():
  import example
  a = 3
  b = 10
  c = a + b
  return example.fact(a)
if __name__ == '__main__':
 result = getFact()
 print result
```

test.py

Python インタプリタのデバッグ版を使用して実行します。

```
ubuntu:~/swigTest> python-dbg test.py
6
```

test.py を実行

課題 – Python と C/C++アプリケーションのデバッグ

Python の拡張を使うと Python と C/C++を簡単に組み合わせることができますが、より複雑なアプリケーションを作成することもできます。開発者はプログラムの実行方法を理解するためにデバッガと動的解析ツールを使用しますが、言語障壁を越えた関数呼び出しやその間のデータの流れを理解しなければならないということは、デバッグに困難な課題をもたらします。ほとんどのデバッガは、両方の言語をシームレスにデバッグするためのソリューションを提供していませんが、いくつかオプションはあります。

Eclipse を使った Python と C/C++のデバッグ

Eclipse や、Eclipse をもとにした PyDev や LiClipse は Python や C/C++のデバッグに強力な環境を提供します。以下の手順では、1 つのセッション内で両方の言語をデバッグする方法のひとつを説明します。

*1 . Roger Hu, Why your Python program can't start when using python-dbg..., June 6, 2013

Eclipse での Python セッションと C/C++デバッグセッションの設定

Python と C/C++のデバッグセッションを設定する方法はいくつかありますが、ここでは Python のデバッグセッションを開始し、C/C++のデバッグセッションを使用して Python インタプリタに接続する方法をご紹介します。

1. Python プロジェクトとデバッグ設定を設定する

   プログラム用の Python プロジェクトを作成し、Python 実行デバッグ設定を設定します。Python 拡張モジュールをデバッグ用にコンパイルしたので、デバッグ時の Python インタプリタを、Python のデバッグ版を使用するように設定する必要があります。Ubuntu 14.04 システムの Python 2.7 の場合、これは python2.7-dbg です。 まずは Python デバッグプロジェクトが問題なく実行されることを確認してください。

2. C/C++ Attach to Application のデバッグ設定を設定する

   Python コードをデバッグ可能にするために、C/C++ Attach to Application デバッグ設定を新しく作成します。新しい設定を作成する以外に特にすべきことはありません。

3. Python デバッグセッションを開始します。

   Python コードで C/C++を呼び出す場所の近くにブレークポイントを設定してから、Python Run デバッグセッションを開始します

4. Python インタプリタにアタッチする

   C/C++ Python 拡張モジュールへの呼び出し近くの Python ブレークポイントがヒットしたら、ステップ 2 で設定した C/C++ Attach to Application デバッグセッションを開始します。するとシステム上のプロセスを示すダイアログが表示されますので、開始した Python プロセスを選択し、[OK]をクリックします。

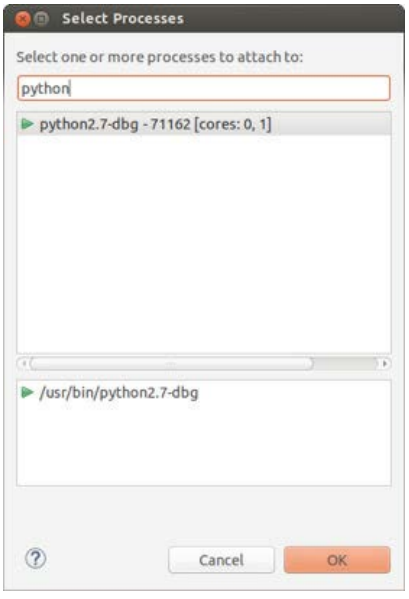

図 1：接続先の Python プロセスを選択する

5. ブレークポイントを C/C++コードに設定し、デバッグを開始

   デバッグしたい C/C++ファイルを開き、止めたい場所にブレークポイントを置いてから、C/C++デバッグセッションを再開します。

Eclipse で Python と C/C++のデバッグを開始する

この時点で、Python デバッガが制御対象となり、ステップ 3 でヒットしたブレークポイントと C/C++拡張モジュールへの呼び出し時点で待機しています。 Step ボタンをクリックすると、C/C++拡張が呼び出され、C/C++ブレークポイントがヒットします。[デバッグ]タブには、Python Run デバッグセッションと C/C++ Attach to Application セッションのスタックトレースが表示されます。Eclipse はブレークポイントに到達したスレッドに自動的には再フォーカスしないので、ユーザーがスレッドをクリックする必要があります。

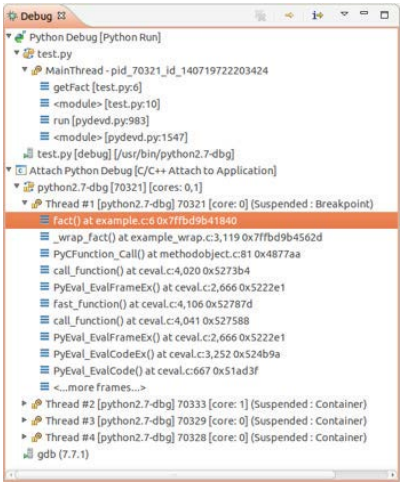

図 2：Python と C/C++呼び出しスタック – 統合されていない

ここまでの手順で、C/C++でのデバッグが可能になりました。変数を調べることができますし、フォーカスを Python スタックフレームに戻せば Python 変数と比較することさえできます。 Python コードのデバッグを続けるには、C/C++ Attach to Application セッションを再開し、Python Run セッションのフレームにフォーカスします。

ローグウェーブの CodeDynamics と TotalView を使った Python と C/C++のデバッグ

CodeDynamics と TotalView は、C/C++拡張機能を使用する Python アプリケーションのデバッグをサポートしています。デバッガはまだブレークポイントの設定や Python コードのステップ実行をサポートしていませんが、デバッグセッションの設定、言語障壁間をまたいだデータチェック、C/C++コードのデバッグが容易に行えます。以下の手順で、CodeDynamics と TotalView を使って Python と C/C++のコードをどのようにすばやくデバッグできるのかを示します。

CodeDynamics または TotalView を使用した Python および C/C++デバッグセッションの設定

CodeDynamics または TotalView を使用して Python および C/C++のデバッグセッションを設定することは、以下の手順に示すように非常に簡単です。

1. 新しいプログラムセッションを設定する

   デバッグセッションを開始するには、Start Page から Program Session を選択し、デバッグ用の Python インタプリタへのパスと、インタープリタへの引数として実行される Python ファイルの名前を入力して、Load Session をクリックします。

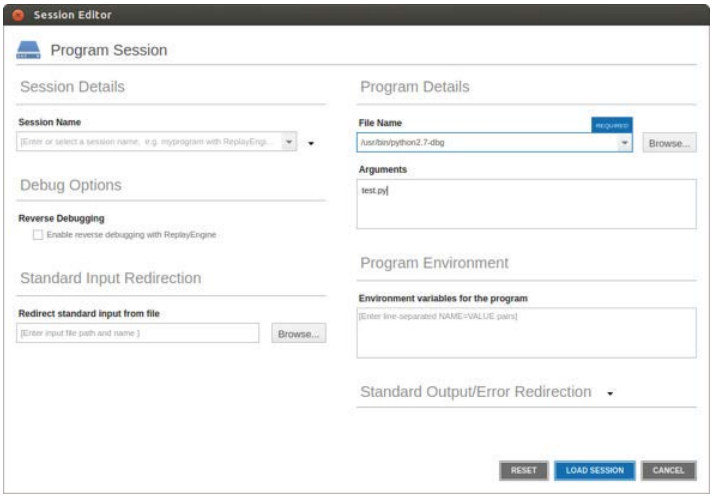

図 3：Python のデバッグセッションを設定する

さらに素早く起動するには、情報を CodeDynamics や TotalView のコマンドライン引数として与えます。

```
codedynamics --args /usr/bin/python2.7-dbg test.py
```

```
totalview -newui --args /usr/bin/python2.7-dbg test.py
```

2. C/C++コードにブレークポイントを設定し、デバッグを開始

C/C++ Python 拡張コードにブレークポイントを設定するには、メニューから Action Points | At Location を選択し、At Location ダイアログを表示します。 「Python 拡張ファイル＃行番号」を入力し、[Create Breakpoint]をクリックすると、指定したファイルと行番号にブレークポイントが作成されます。

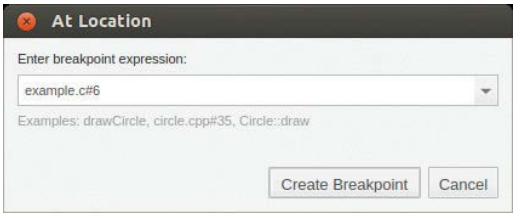

図 4： Python 拡張機能にブレークポイントを作成する

3. Python transformations を有効にする(2017.2 より前のバージョンのみ)

TotalView のバージョン 2017.1 では Python デバッグはデフォルトでは無効になっています。この機能を有効にするには、TotalView CLI から次のコマンドを発行します。 2017.2 以降の TotalView ではこの機能はデフォルトで有効ですので、この手順は不要です。

```
dstacktransform enable
```

CodeDynamics/TotalView で Python と C/C++のデバッグを開始する

Python セッションが設定され、Python 拡張モジュールにブレークポイントが設定されていれば、単に Python インタプリタを起動するだけです。CodeDynamics または TotalView は、ブレークポイントに達すると停止します。

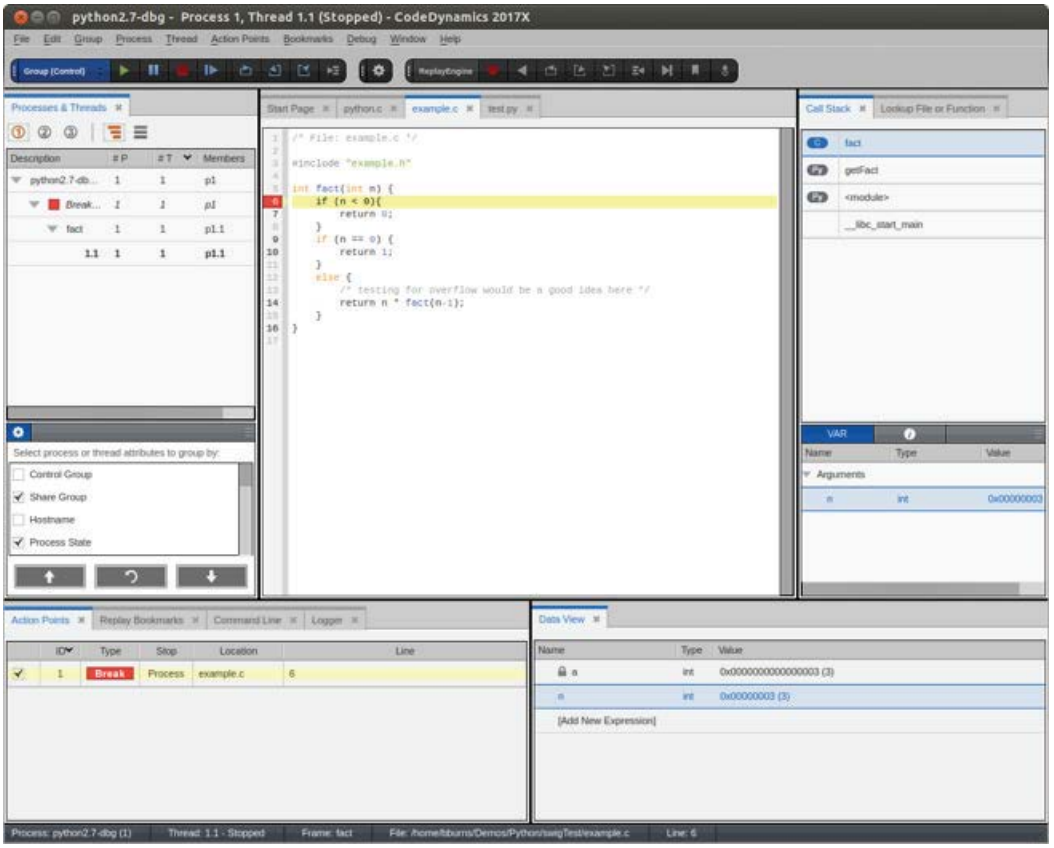

図 5： Python 拡張の関数で停止しコールスタックはすっきりと統合

CodeDynamics や TotalView は優れた機能の 1 つとして、Python および C/C++のすべてのフレームの完全に統合されたコールスタックを提供します。さらに、2 つの言語を結びつける目障りな呼び出しをすべて取り除き、Python から C/C++の呼び出しを開発者にとって見やすくすっきりと表示します。

言語障壁の両側のデータを比較したい場合、C/C++フレームまたは Python フレームをクリックするだけで、VAR パネルでローカル変数の値を調べることができます。変数を Data ビューにドラッグして、値を見たり比較することもできます。

最後に、CodeDynamics と TotalView は、プログラムがどのように実行されたかを理解するためにデバッグセッションの実行を記録し、あとから再現してデバッグできるようにするリバースデバッグエンジン、ReplayEngine などの強力なデバッグ機能を提供します。

Python と C/C++のデバッグ – Eclipse と CodeDynamics または TotalView

Eclipse の Python と C/C++のデバッグは、それぞれ専用の perspective ではうまくいきますが、両方を 1 つのデバッグセッションに組み合わせると、多くのワークフロー上の問題が発生します。Python と C/C++のデバッグセッションを設定するには時間を費やす必要があります。セッションが始まっても、Eclipse は言語障壁を越えて統一されたスタックトレースを提供せず、言語間のグルーコード層がプログラムの流れを曇らせます。開発者は Python と C/C++変数をそれぞれの言語障壁の内側で調べることはできますが、一度に表示されるのはスタックフレームで選択された言語の変数のみであるため、両方を同時に比較してデータの整合性を確かめるのは困難です。

CodeDynamics や TotalView ソリューションは、Python および C/C++のデバッグセッションを確立するための非常に簡単なワークフローを提供します。C/C++コードにブレークポイントを設定するのは簡単で、ヒットするときれいに統合された Python と C/C++のスタックトレースが表示されます。言語障壁間のコードは取り除かれ、開発者が期待するコールスタックだけが表示されます。開発者は Python や C/C++変数を Data ビューにドラッグして値を調べ、データが言語障壁の間で正しくやりとりされていることを簡単に確認できます。

結論

Python は、Deep Learning や人工知能など多くの業界で成長を続けています。 TensorFlow などのアプリケーションは、Python と C/C++の拡張機能を組み合わせた複雑なプログラムの例ですが、Python と C/C++の混在言語のプログラムをデバッグすることは開発者にとって難しい課題です。開発者がこれらの混在した言語プログラムを簡単にデバッグできるようになるためのツールは現れ始めています。Eclipse ソリューションは Python の完全なデバッグを可能にしますが、デバッグ環境の設定が面倒です。ローグウェーブの CodeDynamics と TotalView は、Python や C/C++プログラムをデバッグし言語障壁間のやりとりやデータ交換を理解するための迅速かつ便利な方法を提供します。リバースデバッグなどの先進的な C/C++デバッグ機能と組み合わせ、Python と C/C++の間の実行パスとデータ交換をデバッグするための強力なソリューションとなるのです。Python を利用する業界が拡大していく中、こうした複雑なツールは、開発者が必要とするデバッグ機能を提供するために前進していく必要があります。

CodeDynamics と TotalView が実行時の問題をより迅速に特定して解決します。詳しくは、roguewave.jp をご覧ください。

RogueWave SOFTWARE
Innovate with Confidence