Rubyアソシエーション開発助成 2019 最終報告書

文責: 近藤宇智朗
メンター: 笹田耕一
作成: 2020年3月13日

本プロジェクトと成果物について

本プロジェクトはLinuxの次世代トレース機能であるeBPFにアクセスするRubygem、「RbBCC」の実装と、それを利用したRubyらしいLinuxの負荷計測やデバッグのためのツールのプロトタイプを作成するところをスコープとする。

成果物として以下の実装が完了したことを報告する。今回、スコープの範囲の実装は一通り終えて公開済みである。

1) RbBCC gemの開発

基本的なeBPFのトレーシング機構が利用可能なRubygemである「RbBCC」について、Python版と同等のトレーシング機能をカバーしたgemが完成した。

プロジェクトの成果物のURLは https://github.com/udzura/rbbcc で、Python版と同様Apache License version 2.0で公開している。

また、開発を開始するためのGetting Startedドキュメントとチュートリアルが完成した。

2) RbBCC を活用したRuby製の実例

RbBCCを活用し、またRubyのエコシステムに貢献するようなツールを2点開発した。

- **BPFQL** - YAMLまたはRuby DSLで宣言的にカーネルのトレースを可能にするコマンド
- **Rack::EBPF** - RailsなどのRackアプリケーションをリクエスト単位でトレースする機構の実現

本稿では続いて、本プロジェクトの成果について詳細を報告する。

最終成果の詳細(1) - RbBCC gemの開発

最初にRbBCCの開発内容の詳細を報告する。

RbBCCのコア機能の移植

中間報告から進んで、「基本的なカーネルのトレーシング機構[1]」のカバーの指標として、Python側の実装のリファレンスドキュメント[2]と対比し、ドキュメント上で23種類ある項目を100%をカバーした。カバーした機能を表1に示す。

[1] 具体的にはk(ret)probe、tracepoint、u(ret)probe、USDT、raw tracepointをサポートした。

[2] https://github.com/iovisor/bcc/blob/v0.10.0/docs/reference_guide.md#bcc-python

表1 コア機能の実装表

Python	Ruby	API種別	実装可否	備考
BPF	RbBCC::BCC	class	OK	
USDT	RbBCC::USDT	class	OK	
BPF.attach_kprobe	BCC#attach_kprobe	event	OK	
BPF.attach_kretprobe	BCC#attach_kretprobe	event	OK	
BPF.attach_tracepoint	BCC#attach_tracepoint	event	OK	
BPF.attach_uprobe	BCC#attach_uprobe	event	OK	
BPF.attach_uretprobe	BCC#attach_uretprobe	event	OK	
USDT.enable_probe	USDT#enable_probe	event	OK	
BPF.attach_raw_tracepoint	BCC#attach_raw_tracepoint	event	OK	
BPF.trace_print	BCC#trace_print	output	OK	
BPF.trace_fields	BCC#trace_fields	output	OK	
BPF.perf_buffer_poll	BCC#perf_buffer_poll	output	OK	
BPF.get_table	BCC#[], BCC#get_table	map	OK	基本的なテーブルは実装済み
table.open_perf_buffers	PerfEventArray#open_perf_buffer	map	OK	
table.items	TableBase#items	map	OK	
table.values	TableBase#values	map	OK	
table.clear	TableBase#clear	map	OK	
table.print_log2_hist	TableBase#print_log2_hist	map	OK	一部オプションは対応中
table.print_linear_hist	TableBase#print_linear_hist	map	OK	一部オプションは対応中
BPF.ksym	BCC.#ksym	helper	OK	一部アドレスタイプ未対応
BPF.ksymname	BCC.#ksymname	helper	OK	
BPF.sym	BCC.#sym	helper	OK	一部アドレスタイプ未対応
BPF.num_open_kprobes	BCC.#num_open_kprobes	helper	OK	

BCC Python Tutorialのカバー

また、同じく「基本的なカーネルのトレーシング機構」のカバーの指標として、Python側のチュートリアルをRubyで完走することが可能であることを確認した（表2）。

表2 チュートリアルの実現済範囲

チュートリアルタイトル	内容	実装済
Lesson 1. Hello World	トレースによるHello world表示	OK
Lesson 2. sys_sync()	sys_sync() カーネル関数のトレース	OK
Lesson 3. hello_fields.rb	kprobeとtrace_fieldsの利用	OK
Lesson 4. sync_timing.rb	sync(2)の発行間隔のトレース	OK
Lesson 5. sync_count.rb	sync(2)の発行回数の表示	OK
Lesson 6. disksnoop.rb	ハードディスクへの操作のレイテンシ表示	OK
Lesson 7. hello_perf_output.rb	BPF_PERF_OUTPUT の利用	OK
Lesson 8. sync_perf_output.rb	sync_timing をBPF_PERF_OUTPUT利用に変更	OK
Lesson 9. bitehist.rb	Disk I/O サイズのヒストグラム表示	OK
Lesson 10. disklatency.rb	Disk I/O レイテンシのヒストグラム表示	OK
Lesson 11. vfsreadlat.rb	vfs_read() のレイテンシを定期的に計測表示	OK
Lesson 12. urandomread.rb	urandomへの操作をtracepoint経由でトレース	OK
Lesson 13. disksnoop.rb fixed	disksnoopをtracepoint経由での実装に	OK
Lesson 14. strlen_count.rb	libcのstrlen()の呼び出しを集計	OK
Lesson 15. nodejs_http_server.rb	node.jsのサーバのUSDTのトレース	OK
Lesson 16. task_switch.c	タスクのコンテクストスイッチ内容を表示	OK

libbcc のバージョン別対応

eBPFの機能へのアクセスとして、libbccというC++によるラッパライブラリを利用し、そのFFIという形で実装している。これはlibbcc側に同梱されるPython版と同じ実装方針である。

libbccのバージョン別の対応状況を表3に示した。結論としては、0.10.0 〜 0.12.0 に関しては動作するように確認をし、なおかつCIのテストターゲットとなるように環境を整備した。

表3 libbccバージョン対応表

libbccバージョン	対応状況	CI	備考
0.10.0	OK	対象	
0.11.0	OK	対象	
0.12.0	OK	対象	
0.13.0	?	対象外	非常に最近のリリースのため未対応

その他ドキュメントの充実

上述したチュートリアルのほか、「Getting Started」ドキュメント[3]を作成した。また、Python版の実装と同等のexamples/tools[4]の移植を行って公開している。

Pythonのバインディング実装との比較

表4で、BCCの公式実装の中で、Python版の実装[5]との比較をする。

表4 Ruby版・Python版の比較

項目	Ruby版	Python版	備考
対応するトレーシング対象	カーネルイベントのみ	カーネルイベント、ネットワークトレーシング	ネットワーク関連の機能については随時実装
ドキュメント	Getting Started、 基本的なチュートリアル、 若干のサンプルコード	Getting Started、 基本的なチュートリアル、 豊富なサンプルコード	
libbccの対応	0.10.0、 0.11.0、 0.12.0	全てのバージョン （libbccと同じリポジトリで開発されているため）	gobpfなどの非同梱実装は同じような対応問題を抱える。
インストール	libbccに依存するため、ライブラリとgemそれぞれのインストールが必要	Pythonのコード含めパッケージあるいはセルフビルド。 pip等からのインストールはできない	
実例	リポジトリのツール実装、 bpfql、rack-ebpf	BCCリポジトリに豊富なツール実装がある	

[3] https://github.com/udzura/rbbcc/blob/master/docs/getting_started.md

[4] https://github.com/udzura/rbbcc/tree/master/examples

[5] https://github.com/iovisor/bcc/tree/master/src/python ほか、Lua、Go言語の実装あり

比較の通り、現状の機能はPython版のサブセットとなっている。ただし後述するBPFQLのようなRubyらしい要素を持った実例の開発をしているところである。

今後の課題と展望

RbBCCについて、何点か今後の課題や展望を示す。
1) チュートリアルの日本語化、Rubyらしいチュートリアル項目の追加を検討している。また、その他リファレンスやドキュメントの充実も必要と考えている。
2) Python版の機能で一部カバーできていないものを実装する。テストケースの充実も含む。

最終成果の詳細(2) - RbBCCを活用したRuby製の実例

続いて、RbBCCを利用したツール例や、実世界での応用例を2点示す。

BPFQL

BPFQL[6]は、eBPFによるトレースを宣言的なRuby/YAML DSLを利用し実現するコマンドラインツールである。

同等のツールとしてはbpftraceが存在しているが、bpftraceはDTrace言語のような独特のフォーマットの外部DSL（図1）を採用しており、文法や定型的表現の習得に若干のコストがかかることが考えられる。

図1 bpftrace言語の例

```
BEGIN
{
        printf("Tracing block device I/O... Hit Ctrl-C to end.\n");
}

kprobe:blk_account_io_start
{
        @start[arg0] = nsecs;
}

kprobe:blk_account_io_done
/@start[arg0]/
{
        @usecs = hist((nsecs - @start[arg0]) / 1000);
        delete(@start[arg0]);
}

END
{
        clear(@start);
}
```

BPFQLでは、YAMLあるいはRubyの内部DSLにより、eBPFの「クエリ」を宣言的に記述できるようにした。文法はSQLを真似たものとなっており、ある程度直感的に利用できるように意図している。また、RubyのDSLにより、繰り返しや環境変数による挙動のスイッチなど、ごく簡単なカスタマイズを入れやすいようにしている。スクリプトの例を図2に示す。

[6] https://github.com/udzura/bpfql

図2 bpfql言語の例

```
BPFQL:
- select: [pid, ts, comm, got_bits]
  from: tracepoint:random:urandom_read
  where: "comm is ruby"
```

```
BPFQL do
  select "ts", "comm", "pid", "__syscall_nr", "ret"
  from "tracepoint:syscalls:sys_exit_read"
  if ENV['TARGET_COMM']
    where "comm", is: ENV['TARGET_COMM']
  elsif ENV['TARGET_PID']
    where "pid", is: ENV['TARGET_PID']
  end
end
```

BPFQLは現在、PoCレベルでtracepointのサポートのみ実装されている。

Rack::EBPF

Rack::EBPF[7]は、Rack Middlewareの内部でUSDT[8]を埋め込むことにより、リクエスト単位でのeBPFによるトレースを実現した実装である。

利用の際は、計測対象とするRuby on RailsなどのRackベースのアプリケーションのライブラリにrack-ebpf gemを含み、その上でRack Middlewareとして設定を行う（図3）。

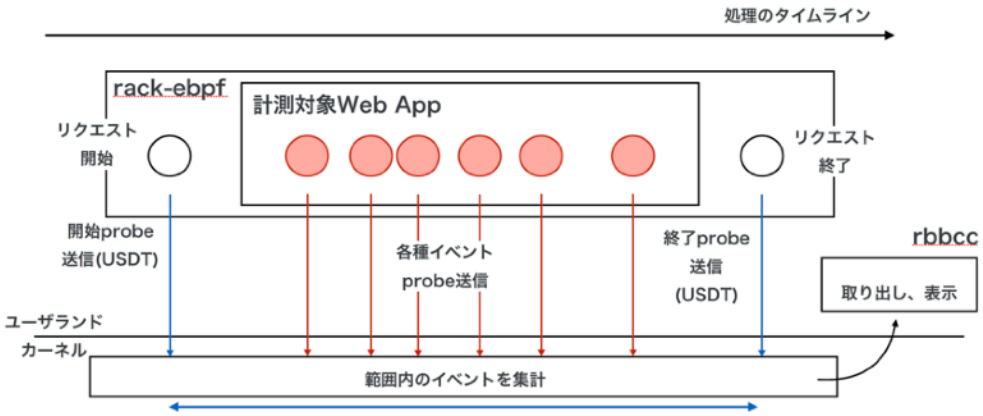
図3 rack-ebpfの設定

```
require 'rack-ebpf'

Rails.application.config.middleware.insert_before Rack::Sendfile, Rack::EBPF
```

設定後、Rack Middlewareの特性を生かし、単一のリクエストの開始時点からレスポンスを生成し終わった終了時点までのイベントをスレッド単位でeBPFにて集計、表示する（図4）。

図4 rack-ebpfの計測モデル



[7] https://github.com/udzura/rack-ebpf

[8] User Statically Defined Tracing。ユーザランドのプログラムの任意の箇所に、任意の内容のprobeを埋め込む技術。eBPFからも容易にトレースできる。

計測の前提としては、単一のスレッドでは同時に単一のリクエストしか取り扱わないようなモデルのサーバでの計測をサポートする。複数のリクエストに対してはマルチプロセスあるはマルチスレッドで対応するようなものである。これはpuma、unicornなどのメジャーな実装が相当する。一方今後node.jsのような、非同期IOをベースに同一スレッドのイベントループ上でリクエストを切り替えるモデルのサーバが出てきた場合、計測方法を再考する必要があるだろう。

現在の実装例としてはシステムコールの数え上げというサンプル的な内容にとどまる。これについて一定の負荷の元でも正しく動くことを確認している（図5）[9]。ランダムな回数read(2)を発行するアクションに対するApache Benchによるテスト[10]では、probe有効時（計測中）と無効時（非計測中）とでパフォーマンスに差はほとんど見られなかった（表5）。

図5 リクエストごとシステムコール発行数の分布の表示

```
# bundle exec rack-ebpf-run 5222
Found fnc: on_usdt_fired
Found fnc: tracepoint__syscalls__sys_enter_read
Attach: syscalls:sys_enter_read
Attach: p__proc_5222_root…usdt_marker_so_0x799_5222
Tracing... Hit Ctrl-C to end.
^C
     count of read       : count     distribution
         0 -> 1          : 274      |*****                                   |
         2 -> 3          : 253      |*****                                   |
         4 -> 7          : 494      |**********                              |
         8 -> 15         : 1020     |********************                    |
        16 -> 31         : 1973     |****************************************|
        32 -> 63         : 986      |*******************                     |
```

表5 計測有効時、無効時の比較

項目	失敗リクエスト数	平均リクエスト時間	80/90/95/99パーセンタイル
probe有効	0	222.16 (ms)	236/252/267/311 (ms)
probe無効	0	218.74 (ms)	233/247/263/308 (ms)

今後の課題と展望

RbBCCの2点の実例についての今後の課題と展望を述べる。

1) BPFQLについては、probeとしてtracepointのみサポートするPoCに近い実装である。今後、kprobeなどの他のprobeや、集計関数のサポートを予定している。

2) Rack::EBPFの利用例に関しては、毎リクエストでの特定システムコールにかかった時間の割合の可視化や、毎リクエストで作られるRubyオブジェクトの個数やメソッド呼び出し回数の可視化などを実装し、その他利用方法をドキュメント化する予定である。

---

[9] https://github.com/udzura/ragrant-2019-demos/tree/master/tracee サンプルでは、特定のアクションでランダムなFile.readを発行しそこに対しベンチをかけている。

[10] 両方ともパラメータは `ab -c 50 -n 5000` としている。

その他今後の展望・課題

スコープの範囲での課題についてはそれぞれの詳細で述べた。その他周辺の内容についての今後の開発の展望や課題について記す。

Red Data Tools との連携

2018年度Rubyアソシエーション開発助成事例の「Charty[11]」には、任意の外部のデータ型やクラスに対応するためのアダプタ層が存在する。それをRbBCCのデータ構造であるeBPF mapに対応するテーブルクラスに対応させれば、可視化を低コストかつ柔軟に行える。

Chartyは例えばunicode_plot[12]などの、もともとのBCCから利用できないような可視化ドライバに対応している。またRubyのエコシステム（RubyGems/Bundler）を活用し相互連携や環境構築が非常に容易であると考えられる。これらの点は他の言語でのeBPFラッパー実装と比べて優位な点となるのではないかと考え、引き続き対応を進める。

ネットワーク関連のトレースAPIのサポート

eBPFの非常に重要な用途としてパケットフィルタやキャプチャなど、ネットワーク関連の機能が存在している。現在のところRbBCCではそれらの機能のサポートを行っていないため、引き続き開発を続ける必要がある。

参考までにPython版BCCでの実装のサンプルを掲示する。図6は、HTTPサーバの授受する平文のリクエストライン・ステータスラインを外部からトレースするものである。

図6 Python BCC におけるHTTP snoop

```
# python examples/networking/http_filter/http-parse-simple.py -i docker0
binding socket to 'docker0'
GET / HTTP/1.1
MHTTP/1.1 200 OK
GET /favicon.ico HTTP/1.1
HTTP/1.1 404 Not Found
```

謝辞

本プロジェクトでの開発にあたり、メンターの笹田耕一さんには継続的に様々な面でアドバイスやアイデアをいただいた。

また、さくらインターネットの松本亮介さん、九州大学 情報基盤研究開発センターの嶋吉隆夫先生、笠原義晃先生には開発に関するアドバイスと、BPFQLの元となるアイデアを頂戴した。この場を借りてお礼を申し上げたい。

以上

[11] https://github.com/red-data-tools/charty

[12] https://github.com/red-data-tools/unicode_plot.rb