调试内存泄漏

在Scrapy中，诸如请求，响应和项目之类的对象具有有限的生命周期：它们被创建，使用一段时间，最后被销毁。

从所有这些对象中，Request可能是具有最长生命周期的对象，因为它在Scheduler队列中保持等待，直到处理它为止。欲了解更多信息请参阅架构概述。

由于这些Scrapy对象具有（相当长的）生命周期，因此总是存在将它们累积在内存中而不正确释放它们的风险，从而导致所谓的“内存泄漏”。

为了帮助调试内存泄漏，Scrapy提供了一种用于跟踪名为trackref的对象引用的内置机制，您还可以使用名为Guppy的第三方库来进行更高级的内存调试（有关详细信息，请参阅下文）。必须从Telnet控制台使用这两种机制。

内存泄漏的常见原因

它经常发生（有时是偶然的，有时是故意的）Scrapy开发人员传递Requests中引用的对象（例如，使用 `meta` 属性或请求回调函数），并且有效地将这些引用对象的生命周期限制为生命周期的生命周期。请求。到目前为止，这是Scrapy项目中内存泄漏的最常见原因，而且对于新手来说是一个非常困难的调试。

在大型项目中，蜘蛛通常由不同的人编写，其中一些蜘蛛可能会“泄漏”，从而影响其他（编写良好的）蜘蛛的其余部分，当它们同时运行时，这反过来会影响蜘蛛整个爬行过程。

如果您没有正确释放（先前分配的）资源，泄漏也可能来自您编写的自定义中间件，管道或扩展。例如，如果每个进程运行多个蜘蛛，则分配资源 `spider_opened` 但不释放它们 `spider_closed` 可能会导致问题。

请求太多了？

默认情况下，Scrapy将请求队列保留在内存中; 它包括 `Request` 对象和Request属性中引用的所有对象（例如in `meta` ）。虽然不一定是泄漏，但这可能会占用大量内存。启用 持久作业队列可以帮助控制内存使用。

调试内存泄漏 `trackref`

`trackref` 是Scrapy提供的一个模块，用于调试最常见的内存泄漏情况。它基本上跟踪对所有实时请求，响应，项目和选择器对象的引用。

您可以使用 `prefs()` 函数作为函数的别名来进入telnet控制台并检查当前有多少个对象（上面提到的类）的存活 `print_live_refs()` ：

```
telnet localhost 6023

>>> prefs()
Live References

ExampleSpider                       1   oldest: 15s ago
HtmlResponse                       10   oldest: 1s ago
Selector                            2   oldest: 0s ago
FormRequest                       878   oldest: 7s ago
```

如您所见，该报告还显示了每个类中最旧对象的“年龄”。如果您在每个进程中运行多个蜘蛛，则可以通过查看最早的请求或响应来确定哪个蜘蛛正在泄漏。您可以使用该 `get_oldest()` 函数（来自telnet控制台）获取每个类的最旧对象 。

跟踪哪些对象？

跟踪的对象 `trackrefs` 都来自这些类（及其所有子类）：

- `scrapy.http.Request`
- `scrapy.http.Response`
- `scrapy.item.Item`
- `scrapy.selector.Selector`
- `scrapy.spiders.Spider`

一个真实的例子

让我们看一个假设的内存泄漏案例的具体例子。假设我们有一些蜘蛛的线条与此类似：

```
return Request("http://www.somenastyspider.com/product.php?pid=%d" % product_id,
    callback=self.parse, meta={referer: response})
```

该行在请求中传递响应引用，该响应引用有效地将响应生存期与请求生命周期联系起来，这肯定会导致内存泄漏。

让我们看看我们如何通过使用该 `trackref` 工具发现原因（当然不知道它是先验的）。

爬虫运行几分钟后我们注意到它的内存使用量增长了很多，我们可以进入它的telnet控制台并检查实时引用：

```
>>> prefs()
Live References

SomenastySpider                     1   oldest: 15s ago
HtmlResponse                     3890   oldest: 265s ago
Selector                            2   oldest: 0s ago
Request                          3878   oldest: 250s ago
```

事实上有很多现场回复（并且它们已经很老了）这一事实肯定是可疑的，因为与请求相比，回复应该具有相对较短的生命周期。响应的数量与请求的数量相似，因此看起来它们在某种程度上是相互关联的。我们现在可以去查看蜘蛛的代码，以发现产生泄漏的令人讨厌的行（在请求中传递响应引用）。

有时，有关实时对象的额外信息会很有帮助。我们来看看最老的回复：

```
>>> from scrapy.utils.trackref import get_oldest
>>> r = get_oldest('HtmlResponse')
>>> r.url
'http://www.somenastyspider.com/product.php?pid=123'
```

如果要迭代所有对象，而不是获取最旧的对象，则可以使用以下`scrapy.utils.trackref.iter_all()`函数：

```
>>> from scrapy.utils.trackref import iter_all
>>> [r.url for r in iter_all('HtmlResponse')]
['http://www.somenastyspider.com/product.php?pid=123',
 'http://www.somenastyspider.com/product.php?pid=584',
...
```

蜘蛛太多了？

如果您的项目有太多并行执行的蜘蛛，则输出`prefs()`可能难以阅读。出于这个原因，该函数有一个`ignore`参数，可用于忽略特定的类（及其所有子类）。例如，这不会显示对蜘蛛的任何实时引用：

```
>>> from scrapy.spiders import Spider
>>> prefs(ignore=Spider)
```

scrapy.utils.trackref模块

以下是`trackref`模块中可用的功能。

类 scrapy.utils.trackref.object_ref

如果要使用`trackref`模块跟踪实时实例，请从此类（而不是对象）继承。

scrapy.utils.trackref.print_live_refs (*class_name* , *ignore = NoneType*)

打印按类名分组的实时参考报告。

参数： ignore（*类或类元组*）- 如果给定，将忽略指定类（*或类的元组*）中的所有对象。

scrapy.utils.trackref.get_oldest (*class_name*)

使用给定的类名返回最旧的对象，或者 `None` 如果找不到则返回。 `print_live_refs()` 首先使用每个类名获取所有跟踪的活动对象的列表。

scrapy.utils.trackref.iter_all (*class_name*)

在具有给定类名的所有活动对象上返回迭代器，或者 `None` 如果找不到则返回迭代器。 `print_live_refs()` 首先使用每个类名获取所有跟踪的活动对象的列表。

使用Guppy调试内存泄漏

`trackref` 提供了一种非常方便的机制来跟踪内存泄漏，但它只跟踪更有可能导致内存泄漏的对象（请求，响应，项目和选择器）。但是，在其他情况下，内存泄漏可能来自其他（或多或少模糊）对象。如果这是你的情况，并且你无法找到你的泄漏 `trackref` ，你仍然有另一个资源：Guppy库。如果您使用的是Python3，请参阅使用muppy 调试内存泄漏。

如果使用 `pip` ，可以使用以下命令安装Guppy：

```
pip install guppy
```

telnet控制台还带有一个 `hpy` 用于访问Guppy堆对象的内置快捷方式（ ）。这是一个使用Guppy查看堆中可用的所有Python对象的示例：

```
>>> x = hpy.heap()
>>> x.bytype
Partition of a set of 297033 objects. Total size = 52587824 bytes.
 Index  Count   %     Size   % Cumulative  % Type
     0  22307   8 16423880  31  16423880  31 dict
     1 122285  41 12441544  24  28865424  55 str
     2  68346  23  5966696  11  34832120  66 tuple
     3    227   0  5836528  11  40668648  77 unicode
     4   2461   1  2222272   4  42890920  82 type
     5  16870   6  2024400   4  44915320  85 function
     6  13949   5  1673880   3  46589200  89 types.CodeType
     7  13422   5  1653104   3  48242304  92 list
     8   3735   1  1173680   2  49415984  94 _sre.SRE_Pattern
     9   1209   0   456936   1  49872920  95 scrapy.http.headers.Headers
<1676 more rows. Type e.g. '_.more' to view.>
```

你可以看到dicts使用了大部分空间。然后，如果要查看引用这些dicts的属性，可以执行以下操作：

```
>>> x.bytype[0].byvia
Partition of a set of 22307 objects. Total size = 16423880 bytes.
 Index  Count   %     Size   % Cumulative  % Referred Via:
     0  10982  49  9416336  57   9416336  57 '.__dict__'
     1   1820   8  2681504  16  12097840  74 '.__dict__', '.func_globals'
     2   3097  14  1122904   7  13220744  80
     3    990   4   277200   2  13497944  82 "['cookies']"
     4    987   4   276360   2  13774304  84 "['cache']"
     5    985   4   275800   2  14050104  86 "['meta']"
     6    897   4   251160   2  14301264  87 '[2]'
     7      1   0   196888   1  14498152  88 "['moduleDict']", "['modules']"
     8    672   3   188160   1  14686312  89 "['cb_kwargs']"
     9     27   0   155016   1  14841328  90 '[1]'
<333 more rows. Type e.g. '_.more' to view.>
```

正如您所看到的，Guppy模块非常强大，但也需要一些关于Python内部的深入知识。有关Guppy的更多信息，请参阅 Guppy文档。

使用muppy调试内存泄漏

如果您使用的是Python 3，则可以使用Pympler中的muppy。

如果使用`pip`，可以使用以下命令安装muppy：

```
pip install Pympler
```

这是一个使用muppy查看堆中可用的所有Python对象的示例：

```
>>> from pympler import muppy
>>> all_objects = muppy.get_objects()
>>> len(all_objects)
28667
>>> from pympler import summary
>>> suml = summary.summarize(all_objects)
>>> summary.print_(suml)
                                types |   # objects |   total size
===================================== | =========== | ============
                           <class 'str |        9822 |      1.10 MB
                          <class 'dict |        1658 |    856.62 KB
                          <class 'type |         436 |    443.60 KB
                          <class 'code |        2974 |    419.56 KB
          <class '_io.BufferedWriter'> |           2 |    256.34 KB
                           <class 'set |         420 |    159.88 KB
          <class '_io.BufferedReader'> |           1 |    128.17 KB
          <class 'wrapper_descriptor'> |        1130 |     88.28 KB
                         <class 'tuple |        1304 |     86.57 KB
                        <class 'weakref |        1013 |     79.14 KB
  <class 'builtin_function_or_method'> |         958 |     67.36 KB
           <class 'method_descriptor'> |         865 |     60.82 KB
                 <class 'abc.ABCMeta'> |          62 |     59.96 KB
                          <class 'list |         446 |     58.52 KB
                           <class 'int |        1425 |     43.20 KB
```

有关muppy的更多信息，请参阅muppy文档。

没有泄漏的泄漏

有时，您可能会注意到Scrapy过程的内存使用量只会增加，但永远不会减少。不幸的是，即使Scrapy和你的项目都没有泄漏内存，也可能发生这种情况。这是由于（不太好）已知的Python问题，在某些情况下可能无法将释放的内存返回给操作系统。有关此问题的详细信息，请参阅：

- Python内存管理
- Python内存管理第2部分
- Python内存管理第3部分

Evan Jones提出的改进（在本文中详细介绍）已在Python 2.5中合并，但这只能减少问题，并不能完全解决问题。引用论文：

> *不幸的是，如果不再有分配对象的话，这个补丁只能释放一个竞技场。这意味着碎片化是一个大问题。一个应用程序可以拥有许多兆字节的可用内存，分散在所有竞技场中，但它将无法释放任何内存。这是所有内存分配器遇到的问题。解决它的唯一方法是转移到压缩垃圾收集器，它可以在内存中移动对象。这需要对Python解释器进行重大更改。*

为了使内存消耗合理，您可以将作业拆分为几个较小的作业，或者 不时启用持久作业队列和停止/启动蜘蛛。