2023-12-05 阅读组会精读报告

李清伟

论文标题：Collecting Cyclic Garbage across Foreign Function Interfaces: Who Takes the Last Piece of Cake?

发表会议：PLDI 2023

论文作者：Yamazaki, Tetsuro and Nakamaru, Tomoki and Shioya, Ryota and Ugawa, Tomoharu and Chiba, Shigeru

1 术语定义

FFI: Foreign Function Interface, 外部函数接口

Host 侧: 宿主侧，在文中指 Ruby 侧

Guest 侧: 外部语言侧，在文中指 JS 侧

2 研究动机与背景

库的数量以及目标领域的多样性是采用某种编程语言的重要因素。外部函数接口(foreign function interface, FFI)是很多语言提供的一个重要机制，用于复用其他语言编写的库。很多语言都提供了 FFI 以支持 C 库，但是最近在 web 和数据科学上的需求的增长也导致对于 Python，JavaScript 语言库的调用需求。这种新的需求也引发了一个新问题，就是在两个都是使用垃圾回收方式来管理内存的语言 FFI 之上，如何管理跨语言边界的内存，尤其是对于跨语言边界的循环引用的情况的处理。现有的 FFI 机制可以管理部分跨语言边界的情况，但是对于循环引用的情况没有相对轻量、适用性好的解决方案。有些做法需要修改两个语言的垃圾回收器，因此代价较高，对于不同的垃圾回收器需要有不同的实现方式，在商用场景下不好实现。同时由于需要修改外部语言侧的垃圾回收器和运行时，因此可能导致代码的可维护性较差。基于以上提及的现状，希望提出一种适用性强，代价小，可维护性好，能够回收循环引用垃圾的跨语言 FFI 机制。

2.1 FFI 上实现垃圾回收的现状

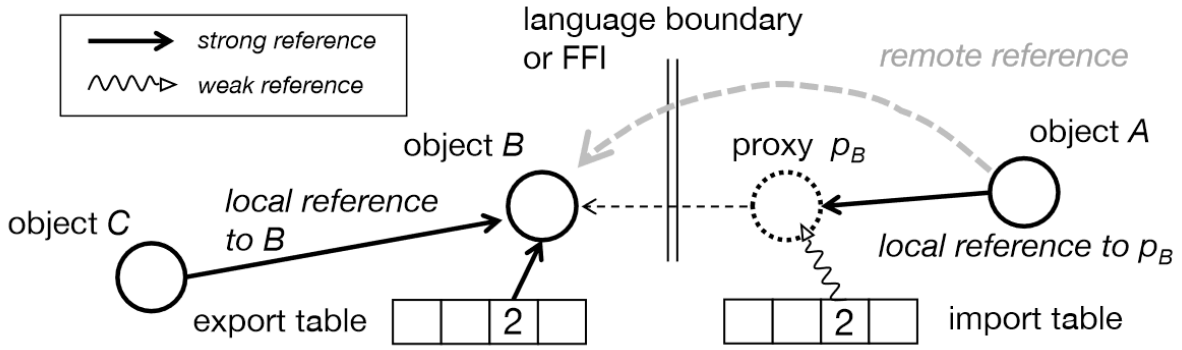

Fig. 1. A remote reference to B and a proxy object ρ_B

Figure 1: remote reference 与 proxy 对象

如 Figure 1 所示，现有 FFI 主要通过 remote reference 和 proxy 对象来实现跨语言边界的内存管理。

当外部语言需要访问对象 B 时，会在逻辑上创建一个从对象 A 到对象 B 的 remote reference。remote reference 的物理实现方式就是外部语言通过访问一个 proxy 对象来达到访问对象 B 以及调

用其方法的操作。为了保证对象 B 在 host 侧的活跃性，需要将对象 B 加入到 export table 中，host 侧的垃圾收集器在扫描时，除了从根集出发扫描之外，还从 export table 出发进行扫描，这样就保证了对象 B 的活跃性。

为了方便回收机制，proxy 对象会在 import table 中通过弱引用 weak reference 的方式注册。当 proxy 对象被外部语言回收后，通过定期扫描 import table 并对 weak reference 进行解引用来检测对象是否被回收，如果对象被回收，就通知 host 侧将宿主侧的对象 B 从 export table 中删除，使得对象 B 从根集不可达。当一个对象从垃圾回收的根集不可达时，该对象会被回收，这样在宿主侧的对象 B 就被回收了。

2.2 FFI 上垃圾回收的问题：循环依赖与性能问题

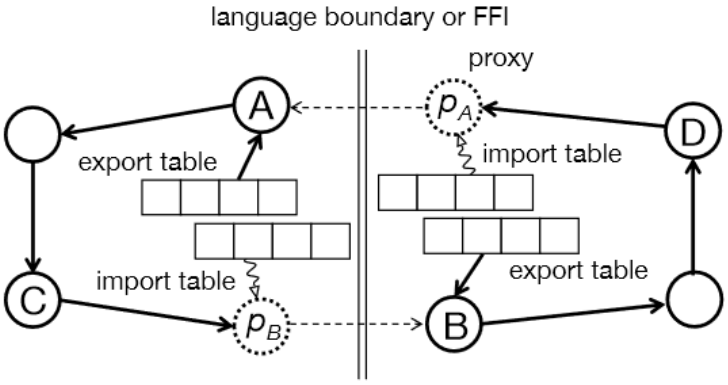

Fig. 2. Cyclic garbage

Figure 2: 循环依赖问题

在现有 FFI 上实现的垃圾回收算法不能解决循环依赖问题，如 Figure 2 所示。即使单语言侧实现了循环依赖对象组的垃圾回收，也不能回收跨语言的循环依赖对象。

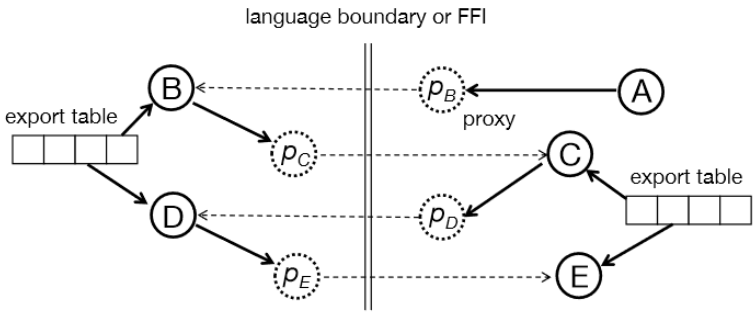

Fig. 3. Zigzag garbage (import tables are omitted)

Figure 3: 之字形问题

即使没有循环依赖，当前 FFI 上的垃圾回收也存在性能差的问题。例如在 Figure 3 问题中。两个语言需要交替进行垃圾回收，这会导致很长的间隔等待时间。

3 解决的问题

提出了 RefGraph-GC 机制，在不修改宿主侧和外部语言侧垃圾回收器的前提下，只简单修改宿主侧和外部语言侧运行时的情况下，实现跨语言边界的循环引用情况的垃圾回收，同时提升其他情况（如 zigzag 形式的垃圾回收）的性能。

4 主要思路

主体思想是不修改宿主侧和外部语言侧的垃圾回收器，通过只修改宿主侧的运行时与外部语言侧的运行时来达到循环引用对象组的回收。具体来说，算法分为 3 个步骤：

1. 在宿主侧运行时中，根据对象引用关系创建一个压缩引用图。图中的边表示通过 remote reference 可访问的外部对象，即 proxy 对象和 host 侧对象（包括根集和 export table 中的对象）之间的可达性关系。
2. 根据压缩引用图，外部语言运行时在外部语言侧的对象引用图中镜像引用边。这使得外部语言侧的垃圾收集器可以追踪 host 侧的活对象
3. 在外部语言侧安装 FFI 级别的发送写屏障。这使得外部语言侧对 host 侧对象的访问以及 host 侧对象中带有的对外部语言侧对象的访问可以重新被外部语言侧垃圾回收器感知，防止外部语言侧的垃圾回收器误回收外部语言侧仍然被 host 侧引用的对象。

4.1 压缩引用图

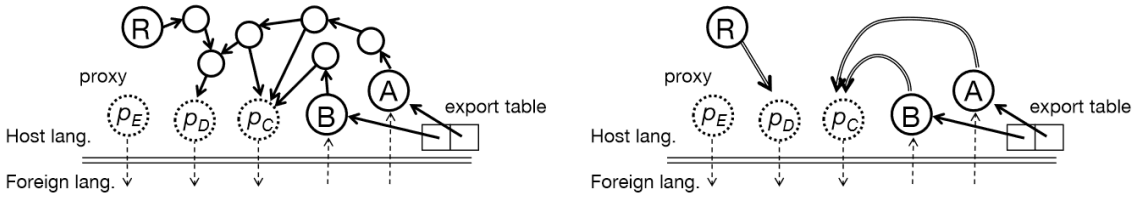

Fig. 4. Original object graph (left) and compressed reference graph (right)

Figure 4: 压缩引用图

压缩引用图将原本 host 侧 GC 所使用的对象引用图进行压缩，只关心根集、proxy 对象、export table 中对象之间的关系。在压缩引用图中，边的目标结点一定是 proxy 对象，源结点是根集或者 export table 中的对象。如果一个 proxy 对象既从 export table 中的对象可达，又从根集可达，那么将会有一条从根集结点到 proxy 对象结点的边，表明这个 proxy 对象仍然在 host 存活。如果 proxy 对象只从 export table 中可达，则说明可能该对象在 host 侧不再从根集可达，一旦 export table 中对象被回收，该对象也可以被回收。从 export table 到 proxy 对象结点的边可能是循环引用的一部分。

4.2 外部语言侧镜像压缩引用图中的引用边

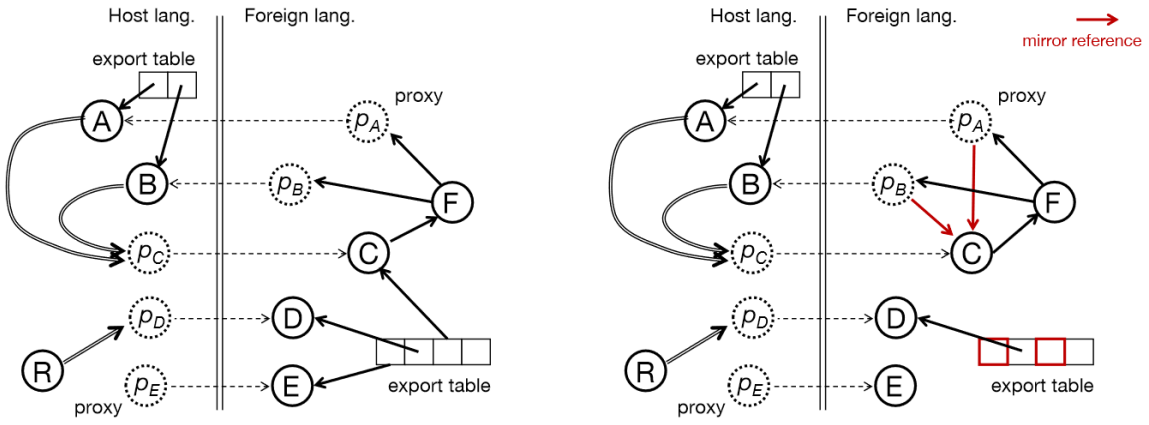

Fig. 5. Mirroring a compressed reference graph (left: before, right: after)

Figure 5: 镜像引用边

由于在外部语言的垃圾收集器视角下并不存在 host 侧的引用关系，因此需要通过镜像方式将这些引用关系让外部语言侧的垃圾收集器感知。

如果在压缩引用图中存在从 export table 到 proxy 对象的边，则在外部语言视角下，存在从 proxy 对象到 export table 中的边，这样的边就称为从压缩引用图中通过镜像方式得到的引用边，如 Figure 5 所示。在添加了这条边之后，相当于把 host 侧的引用信息传递给了外部语言侧，从而使得外部语言侧能够知道引用边的存在，然后决定什么时候回收这些对象。在回收时，当外部语言侧的 proxy 对象被回收时，host 侧的 export table 中的对象会从 export table 中移除，等待 host 侧进行回收。

对于循环引用的情况，在镜像了引用边之后，只需要外部语言侧的垃圾收集器能够支持循环引用的回收，就可以利用上述机制，将跨语言的循环引用回收。

4.3 外部语言侧的屏障

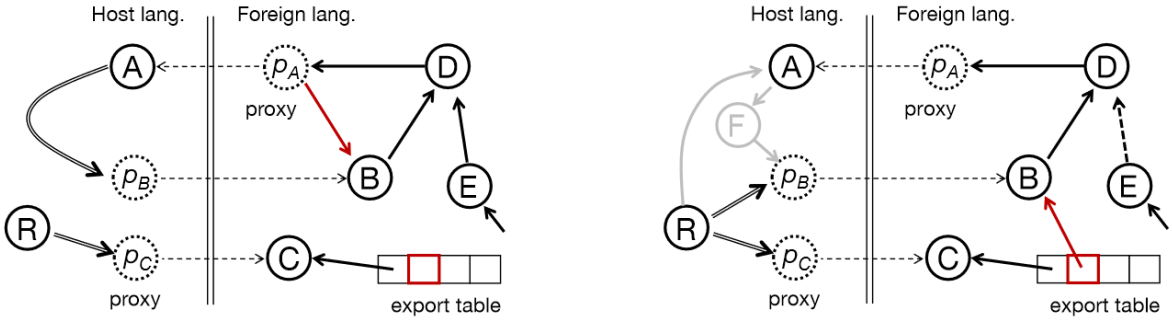

Fig. 6. FFI-level send barrier (left: before, right: after a reference is passed)

Figure 6: 外部语言侧屏障

如 Figure 6 所示，由于在上述镜像过程中会在外部语言侧将 export table 中的对象 B 从 export table 中删去，因此该对象随时可能会被外部语言侧的垃圾收集器回收。但是一旦外部语言侧通过访问 host 侧对象又重新使得这个从 export table 中移除的对象 B 重新在 host 侧活跃，那么就需要将该对象重新恢复到 export table 中。因此该工作通过屏障机制，一旦在外部语言侧检测到对某 host 对象的访问操作，就将 host 侧 export table 中对应的对象所引用的对象重新置为活跃，如果有 proxy

对象(如p_B)被置为活跃，则外部语言侧的 B 对象也会被重新加入到 export table 中，并删除之前的镜像引用边。这样就保证外部语言侧垃圾收集器不会误回收 B 对象。

5 分析/设计/评测方法特色与亮点

1. 在设计方面，作者进行了比较深入的权衡利弊，分析了现有工作的优缺点，发现现有工作要么垃圾回收算法本身不支持循环引用的回收，要么是需要同时修改两个语言的垃圾收集器，可维护性和对其他语言场景的适用性不那么强，对于商业开发场景不适用。即使支持没有循环引用的垃圾回收，也存在性能问题，典型如之字形模式。最终作者决定在 FFI 层级，通过只修改 host 侧语言的运行时以及简单在外部语言侧的运行时中添加函数的方式来实现循环引用垃圾的回收。论文在阐述权衡思路时写的比较好。
2. 在问题分析方面，首先在文章前部提到了该方法的适用范围，即有较多跨语言循环引用的情况。除了分析到了循环引用的场景，还分析到了工作对于之字形等不是循环引用场景也有增益。在评测部分，对于效果不好的案例，也进行了深入分析，包括对测例的用途与模式概括，发现部分用例中实际上并没有较多循环引用的存在，添加 RefGraph-GC 是一个负担而非收益，因此导致从效果上来看反而不如不加 RefGraph-GC 的情况。在评测部分
   - 除了提出新方法之外，该论文还对方法的可靠性和完备性进行了论述
3. 在评测方面总体上来说不仅分析了效果显著的部分，也分析了造成效果较差部分的原因。具体来说
   1. 设置了三类 benchmark：
      1. 真实世界中的应用(通过 Ruby 调用 JS 的 pdfjs 库，在浏览器中渲染 pdf 文件)
      2. 展示效果的小样例，包括循环引用、之字形引用测试代码。
      3. 展示效果的自行编写真实世界程序，通过 Ruby 调用 JS 中的 babel 库解析 JS 程序的 AST，将 def-use 关系存储在 Ruby 表中
      4. 五个传统的性能库，通过 JS 和 Ruby 分别进行实现，将同一个类的的对象全部在一个语言中创建，有些在 JS 中，有些在 Ruby 中，然后让这些对象进行交互，造成循环引用。
   2. 设置了 4 类性能指标方面：
      1. remote reference 的数量，用于测试是否跨语言边界的引用被有效回收
      2. 执行时间，用于说明添加 RefGraph-GC 机制之后是否会导致性能下降
      3. Ruby 侧的堆大小，说明是否能进行有效的垃圾回收。
      4. 压缩引用图的大小，用于说明 RefGraph-GC 引入的内存开销。结果占所有堆空间的大小并不大，最大为 4%。
   3. 总结来说并不是在所有样例上都表现良好，较好的说明了该工作的适用范围，即存在较多跨语言循环引用的情况。在自行构造的 loop 和 chain 上有较好的执行时间、内存大小、循环引用数量。当然有不好的样例，并且对于不好的样例进行分析，将效果不好的原因归结于：
      1. 工作负载本身中没有循环引用导致 RefGraph 机制变成开销而非收益点(如 havlak 样例)
      2. 垃圾回收的随机性（通过 Figure 7 来进行说明）。

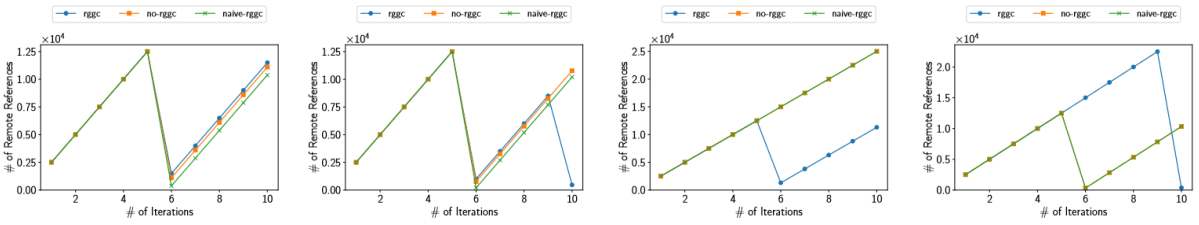

Fig. 11. Number of remote references *nbody* creates

Figure 7: GC 的随机性导致不同次实验的效果很难达到完全一致

6 亮点与不足

亮点：

1. 写作方面文章读起来非常流畅，分析思路比较清晰。而且在阐述步骤时也可以分段讲清楚不同的操作对应的含义是什么，为什么需要这么做。比如在 mirror a graph 中，就分别按段落谈到了当 proxy 对象不在 host 侧中被引用、循环引用的情况。
2. 如上一节所述，文章对已有工作的分析批判非常到位。强调了之前工作对循环引用场景无法适用，即使没有循环引用性能也比较差，对其他语言的适用性不足，可维护性差的问题，为自己的方案的优势做了很多铺垫。

不足（局限性）：

- 如文章自己在总结部分提到的，该工作只能处理跨语言循环引用的情况。对于不存在这样负载的情况，新增的 RefGraph-GC 机制会成为一种负担，导致性能下降。因此需要决定何时、怎样使用这个机制。需要对循环引用垃圾的数量进行估计。同时当前工作只支持单个外部语言，添加对多个外部语言的支持也是未来的工作。（这里多个外部语言应该不是指每次单独处理一个外部语言，而是指同时处理多个外部语言的情况）

7 对自己工作的启发

- 这是一篇很好的利用现有语言技术并通过轻量级的 FFI 改造方式支持跨语言互操作的工作。该工作的思想比较简单，通过代理对象来表示另一侧语言的对象，通过镜像引用来表示另一侧语言的引用关系。在完成镜像之后利用单一语言侧的技术并对另外一侧语言的消息通知达到跨语言互操作的目的。这种范式可以运用于其他跨语言机制的场景，要求两边语言都有类似的机制，通过中间消息传递与镜像方式实现交互。