The Journal of Systems and Software 96 (2014) 139–151

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Memory leak detection in Java: Taxonomy and classification of approaches

Vladimir Šor[a,b,*], Satish Narayana Srirama[c]

[a] Plumbr OÜ, Ülikooli 2, Tartu, Estonia
[b] Software Technology and Applications Competence Center, Ülikooli 2, Tartu, Estonia
[c] University of Tartu, Institute of Computer Science, Liivi 2,Tartu, Estonia

ARTICLE INFO

Article history:
Received 3 December 2013
Received in revised form 2 June 2014
Accepted 3 June 2014
Available online 11 June 2014

Keywords:
Java
Memory leak detection
Garbage collection

ABSTRACT

Memory leaks are usually not associated with runtime environments with automatic garbage collection; however, memory leaks do happen in such environments and present a challenge to detect and find a root cause. Currently in the industry manual heap dump analysis is the most popular way of finding memory leaks, regardless of the number of automated methods proposed by scientists over the years. However, heap dump analysis alone cannot answer all questions needed to fix the leak effectively. The current paper reviews memory leak detection approaches proposed over the years and classifies them from the point of view of assessed metrics, performance overhead and intrusiveness. In addition, we classify the methods into online, offline and hybrid groups based on their features.

1. Introduction

There used to be two major potential sources of bugs associated with manual memory management: dangling pointers and memory leaks. While garbage collection completely solves the problem of dangling pointers, the problem of memory leaks is solved only partially, as garbage collector cannot reclaim objects which are still referenced while being unused. Unused references mean that some object (or object subgraph) is not needed for the application anymore, but there is a reference somewhere, which prevents the object from being garbage collected. In languages with manual memory management, static source code analysis can be used to check whether an allocated block of memory was freed later on; however, in the garbage collected environment, like Java virtual machine, memory leaks are part of software aging and require either runtime analysis or heap dump analysis. Most troublesome memory leaks are so-called *slow leaks*, which accumulate over time and allow the program to run for a long time before `java.lang.OutOfMemoryError` is thrown and the application crashes. The problem with diagnosing such slow leaks is that they need time to accumulate, thus making their reproduction very hard. Slow leaks can be caused by some rarely used function, some non-trivial sequence of user actions is performed or the data in the production system (much richer than data in Q/A or development environments) creates a combination causing the leak. In any case it is very hard to find the actual sequence of actions to simulate the leak in the development environment. Thus, a tool, which can be used in production environments to track down leaks that cannot be reproduced in development, is more valuable.

Heap dump analysis is an effective way to analyze the heap and reveal what is keeping the objects from being garbage collected, but it misses some important features. For example, heap dumps do not contain any temporal information about *when* a particular object was created, nor they contain information about an allocation site, or *where* an object was created. Heap dumps will be described in more detail in Section 4.

Several studies have addressed the online memory leak detection problem. Some of them focus at online analysis of either growth (Chen and Chen, 2007; Jump and McKinley, 2007, 2006; Šor and Srirama, 2011; Šor et al., 2011) or staleness detection (e.g., Bond and McKinley, 2006; Rayside et al., 2006; Rayside and Mendel, 2007; Xu and Rountev, 2008, 2013) and others on offline analysis by either automatically performing dump analysis (Mitchell and Sevitsky, 2003; Maxwell et al., 2010) or doing visualization (De Pauw and Sevitsky, 1999; Reiss, 2009) of the heap dump to simplify the leak detection task for the programmer.

However, we discovered that approaches that provide good insight and analysis of the leak are implemented as a modification of the JVM or garbage collector, which is too intrusive for actual

* Corresponding author at: Plumbr OÜ, Ülikooli 2, Tartu, Estonia. Tel.: +372 53401651.
E-mail address: volli@ut.ee (V. Šor).

http://dx.doi.org/10.1016/j.jss.2014.06.005
0164-1212/

use. Analysis of such solutions presented so far show little adoption outside the research community. Tools which can be simply attached to an application either give too much performance overhead to be used anywhere besides the development environment or the application has to be annotated for the tool to work or only some specific types of memory leaks are handled. Looking back over more than ten years of research shows that JVM vendors have not adopted any of the proposed solutions. The details are provided in the following sections.

The current publication is organized as follows. The classification of memory leak detection techniques is provided in Section 2.1, along with a review of the most commonly used terminology in the field for clarity. In Section 3 we describe methods focusing on online detection, further separating methods measuring object staleness, size growth and self-healing methods. In Section 4 offline methods are reviewed, including methods performing analysis of heap or reference dumps. Visualization methods are also reviewed in Section 4. The remaining methods that do not fall into any other group are reviewed in Section 5. Sections 6 and 6.3 cover aspects such as performance overhead, leak detection performance and intrusiveness, respectively. In Section 6.4 we discuss the observed qualities of the methods and see which tradeoffs have to be made while selecting one or another method for practical use. The paper is concluded with Section 7.

2. Memory leak detection techniques: classification and terminology

After first appearing in 1995, Java programming language, and most importantly Java Virtual Machine (JVM), made enormous progress. According to the TIOBE Programming Community Index (Tiobe Software BV, 2013) Java was the most popular programming language since the beginning of year 2000, with the exception of years 2005 and 2013, where C took over. Java progressed not only as a language but most importantly as a cross-platform runtime environment with garbage collection. In recent years we can observe an increase in popularity of languages such as Scala, Clojure, Groovy, etc., which utilize Java Virtual Machine and its bytecode to produce platform-independent applications.

An online search for the terms "memory leak java" or "OutOfMemoryError" finds thousands of blog posts, forum, and mailing list discussions, which means that memory leaks in JVM languages are not just a theoretical problem. Memory leak detection has been studied over the years and several solutions have been proposed; however, practically none of them have reached the industry. In this work we review memory leak approaches considering their implementation complexity, measured metrics, and intrusiveness. As a result, we propose classification of memory leak detection from analyzed standpoints.

The state-of-the-art approaches can be classified as methods implementing:

1. Online detection, further separating methods into
   (a) measuring staleness and
   (b) detecting growth.
2. Offline detection, including methods:
   (a) analyzing heap dumps and other kinds of captured state,
   (b) using visualization to aid manual leak detection, and
   (c) static analysis of the source code.
3. Hybrid methods, combining features from both online and offline approaches.

Before proceeding further with the classification, following subsection briefly describes the most commonly used terminology in the field, which is necessary to understand the classification and the following discussion.

2.1. Terminology

Garbage collection roots – special references which are directly accessible by the application threads and the garbage collector. Roots include references and variables on stack, static variables, threads, and Java Native Interface (JNI) references (reference from the native code to a Java object). It is hereafter, it is referred to as *GC root*.

Strong reference – a reference from one object to another via direct field or variable reference.

Weak/soft reference – a reference from one object to another made by using proxy objects of type `java.lang.ref.WeakReference`, or `java.lang.ref.SoftReference`. These reference classes are of special meaning to the garbage collector. If an object is reachable only via weak reference object then the object is called *weakly reachable*. If an object is reachable only via a soft reference then the object is called *softly reachable*. If an object is weakly reachable, then it is eligible for finalization (a special method to be called before the object can be disposed) and garbage collection, and thus will be reclaimed. The reference object will be notified that the object it was referring was collected. If an object is softly reachable, then the garbage collector can choose not to collect the object as soon it becomes softly reachable but it can leave the object on the heap until memory pressure arises. Softly reachable objects are guaranteed to be collected before `java.lang.OutOfMemoryError` will be thrown (see Oracle Corp., 2013).

Reachability, reachable object – an object is called reachable when there exists a path to it from a GC root. An object is strongly reachable if it can be reached from the GC roots via strong references only (without traversing any reference objects). If an object is strongly reachable, it cannot be garbage collected.

Mark-sweep garbage collection – the garbage collection algorithm, modifications of which are used in modern JVMs. On a very high level, the algorithm works in two steps. Starting from special references, called garbage collection roots, traversing all objects that can be reached by traversing intermediate references. During this traversal, objects are marked as reachable. The next step removes all unmarked objects from the heap, as they cannot be reached, and thus are unused. The main benefit of the method is that it can also handle circular references between objects. Depending on which priorities are set for the garbage collector (throughput or latency), different optimizations are applied to the basic mark-sweep algorithm (concurrent mark-sweep, parallel mark-sweep, mark-sweep-compact, etc.).

The *liveness* of the object is defined as an object being actively used in addition to being just reachable, i.e., if reachability is a property, which prevents the garbage collector from freeing the object, then liveness of the object shows whether the object is still needed for the application. Liveness can be measured only during runtime and is not available, for example, in a heap dump.

Staleness of the object indicates whether an object has not been used for a while. It is not a quantifiable measure, and the longer an object is not used, the more stale it becomes. Staleness of an object is a good indicator of an unused object; however, it can be expensive to calculate and obtain. Staleness can be measured only during runtime and is not available, for example, in a heap dump.

Dangling pointer/reference – in programming languages with manual memory management a dangling pointer emerges when the object, to which the pointer was pointing is freed, but the pointer itself is not nullified. Thanks to garbage collection dangling pointers do not occur in managed languages; however, the term is often encountered in respective literature.