链滴

# Java 源码剖析——彻底搞懂 Reference 和 ReferenceQueue

作者：jesministrator

原文链接：https://ld246.com/article/1513083921948

来源网站：链滴

之前博主的一篇读书笔记——《深入理解Java虚拟机》系列之回收对象算法与四种引用类型博客中为家介绍了Java中的四种引用类型，很多同学都希望能够对引用，还有不同类型引用的原理进行更深入了解。因此博主查看了抽象父类Reference和负责注册引用对象的引用队列ReferenceQueue的源码在此和大家一起分享，并做了一些分析，感兴趣的同学可以一起学习。

## Reference源码分析

首先我们先看一下Reference类的注释：

```
/**
 * Abstract base class for reference objects.  This class defines the
 * operations common to all reference objects.  Because reference objects are
 * implemented in close cooperation with the garbage collector, this class may
 * not be subclassed directly.
 引用对象的抽象基类。此类定义了常用于所有引用对象的操作。因为引用对象是通过与垃圾回收器的切合作来实现的，所以不能直接为此类创建子类。
 */
```

该类提供了两个构造函数：

```
Reference(T referent) {
    this(referent, null);
}

Reference(T referent, ReferenceQueue<? super T> queue) {
        this.referent = referent;
        this.queue = (queue == null) ? ReferenceQueue.NULL : queue;
}
```

一个构造函数带需要注册到的引用队列，一个不带。<font color="red">带queue的意义在于我们以吃从外部通过对queue的操作来了解到引用实例所指向的实际对象是否被回收了，同时我们也可以过queue对引用实例进行一些额外的操作；但如果我们的引用实例在创建时没有指定一个引用队列，我们要想知道实际对象是否被回收，就只能够不停地轮询引用实例的get()方法是否为空了。值得注意是虚引用PhantomReference，由于它的get()方法永远返回null，因此它的构造函数必须指定一个引队列。这两种查询实际对象是否被回收的方法都有应用，如weakHashMap中就选择去查询queue的据，来判定是否有对象将被回收；而ThreadLocalMap，则采用判断get()是否为null来作处理。</fon>

接下来是它的主要成员：

```
private T referent;         /* Treated specially by GC */
```

在这里我们首先明确一些名词，Reference类也被称为引用类，它的实例 Reference Instance就是引实例，但是由于它是一个抽象类，它的实例只能是子类软(soft)引用,弱(weak)引用,虚(phantom)引用的某个，至于引用实例所引用的对象我们称之为实际对象(也就是我们上面所写出的referent)。

```
volatile ReferenceQueue<? super T> queue;   /* 引用对象队列*/
```

queue是当前引用实例所注册的引用队列，一旦实际对象的可达性发生适当的变化后，此引用实例将被添加到queue中。

```
/* When active:   NULL
 *     pending:   this
 *    Enqueued:   next reference in queue (or this if last)
```

```
 *    Inactive:   this
 */
@SuppressWarnings("rawtypes")
Reference next;
```

next用来表示当前引用实例的下一个需要被处理的引用实例，我们在注释中看到的四个状态，是引用例的内部状态，不可以被外部查看或是直接修改：

- Active：新创建的引用实例处于Active状态，但当GC检测到该实例引用的实际对象的可达性发生某适当的改变(实际对象对于GC roots不可达)后，它的状态将会根据此实例是否注册在引用队列中而变Pending或是Inactive。
- Pending：当引用实例被放置在pending-Reference list中时，它处于Pending状态。此时，该实在等待一个叫Reference-handler的线程将此实例进行enqueue操作。如果某个引用实例没有注册在个引用队列中，该实例将永远不会进入Pending状态。
- Enqueued： 当引用实例被添加到它注册在的引用队列中时，该实例处于Enqueued状态。当某个用实例被从引用队列中删除后，该实例将从Enqueued状态变为Inactive状态。如果某个引用实例没注册在一个引用队列中，该实例将永远不会进入Enqueued状态。
- Inactive：一旦某个引用实例处于Inactive状态，它的状态将不再会发生改变，同时说明该引用实例指向的实际对象一定会被GC所回收。

事实上Reference类并没有显示地定义内部状态值，JVM仅需要通过成员queue和next的值就可以判当前引用实例处于哪个状态：

- Active：queue为创建引用实例时传入的ReferenceQueue的实例或是ReferenceQueue.NULL；ne t为null
- Pending：queue为创建引用实例时传入的ReferenceQueue的实例；next为this
- Enqueued：queue为ReferenceQueue.ENQUEUED；next为队列中下一个需要被处理的实例或是his如果该实例为队列中的最后一个
- Inactive：queue为ReferenceQueue.NULL；next为this

```
/* List of References waiting to be enqueued.  The collector adds
 * References to this list, while the Reference-handler thread removes
 * them.  This list is protected by the above lock object. The
 * list uses the discovered field to link its elements.
 */
private static Reference<Object> pending = null;

/* When active:   next element in a discovered reference list maintained by GC (or this if last)
 *     pending:   next element in the pending list (or null if last)
 *   otherwise:   NULL
 */
transient private Reference<T> discovered;  /* used by VM */
```

看到注释的同学们有可能会有一些疑惑，明明pending是一个Reference类型的对象，为什么注释说是一个list呢？其实是因为GC检测到某个引用实例指向的实际对象不可达后，会将该pending指向该用实例，discovered字段则是用来表示下一个需要被处理的实例，因此我们只要不断地在处理完当前ending之后，将discovered指向的实例赋予给pending即可。所以这个static字段pending其实就是个链表。

```
private static class ReferenceHandler extends Thread {
  ......
```

```
  public void run() {
      while (true) {
          tryHandlePending(true);
      }
  }
}
```

ReferenceHandler是一个优先级最高的线程，它执行的工作就是将pending list中的引用实例添加到用队列中，并将pending指向下一个引用实例。

```
 static boolean tryHandlePending(boolean waitForNotify) {
     ......
     synchronized (lock) {
       if (pending != null) {
           r = pending;
           // 'instanceof' might throw OutOfMemoryError sometimes
           // so do this before un-linking 'r' from the 'pending' chain...
           c = r instanceof Cleaner ? (Cleaner) r : null;
           // unlink 'r' from 'pending' chain
           pending = r.discovered;
           r.discovered = null;
       }
    }
    ......
    ReferenceQueue<? super Object> q = r.queue;
       if (q != ReferenceQueue.NULL) q.enqueue(r);
       return true;
 }
```

Reference对外提供的方法就比较简单了：

```
public T get() {
   return this.referent;
}
```

get()方法就是简单的返回引用实例所引用的实际对象，如果该对象被回收了或者该引用实例被clear则返回null

```
public void clear() {
  this.referent = null;
}
```

调用此方法不会导致此对象入队。此方法仅由Java代码调用；当垃圾收集器清除引用时，它直接执行而不调用此方法。

clear的方法本质上就是将referent置为null，清除引用实例所引用的实际对象，这样通过get()方法就能再访问到实际对象了。

```
public boolean isEnqueued() {
  return (this.queue == ReferenceQueue.ENQUEUED);
}
```

判断此引用实例是否已经被放入队列中是通过引用队列实例是否等于ReferenceQueue.ENQUEUED得知的。

```
public boolean enqueue() {
  return this.queue.enqueue(this);
}
```

enqueue()方法能够手动将引用实例加入到引用队列当中去。

## ReferenceQueue源码分析

同样我们先看一下ReferenceQueue的注释：

```
/**
 * Reference queues, to which registered reference objects are appended by the
 * garbage collector after the appropriate reachability changes are detected.
 * 引用队列，在检测到适当的可到达性更改后，垃圾回收器将已注册的引用对象添加到该队列中
 */
```

ReferenceQueue实现了队列的入队(enqueue)和出队(poll)，其中的内部元素就是我们上文中提到的eference对象。队列元素的存储结构是单链式存储，依靠每个reference对象的next域去找下一个元。

主要成员有：

```
private  volatile Reference extends T> head = null;
```

用来存储当前需要被处理的节点

```
static ReferenceQueue NULL = new Null<>();
static ReferenceQueue ENQUEUED = new Null<>();
```

static变量NUlL和ENQUEUED分别用来表示没有提供默认引用队列的空队列和已经执行过enqueue作的队列。

引用实例入队的逻辑很简单：

```
synchronized (lock) {
    // 检查reference是否已经执行过入队操作
    ReferenceQueue<?> queue = r.queue;
    if ((queue == NULL) || (queue == ENQUEUED)) {
        return false;
    }
    //将引用实例的成员queue置为ENQUEUED
    r.queue = ENQUEUED;
    //若头节点为空，说明该引用实例为队列中的第一个元素，将它的next实例等于this
    //若头节点不为空，将它的next实例指向头节点指向的元素
    r.next = (head == null) ? r : head;
    //头节点指向当前引用实例
    head = r;
    //length + 1
    queueLength++;

    lock.notifyAll();
    return true;
}
```

简单来说，入队操作就是将每次需要入队的引用实例放在头节点的位置，并将它的next域指向旧的头点元素。因此整个ReferenceQueue是一个后进先出的数据结构。

出队的逻辑为：

```
r指向头节点元素
Reference<? extends T> r = head;
if (r != null) {
    //头节点指向null，如果队列中只有一个元素；否则指向r.next
    head = (r.next == r) ? null : r.next;
    //头节点元素的queue指向ReferenceQueue.NULL
    r.queue = NULL;
    //将r.next指向this
    r.next = r;
    //length-1
    queueLength--;

    return r;
}
```

**总体来看，ReferenceQueue的作用就是JAVA GC与Reference引用对象之间的中间层，我们可以外部通过ReferenceQueue及时地根据所监听的对象的可达性状态变化而采取处理操作。**