FinalReference 与 Finalizer 详解

2018-12-29 · 编程

FinalReference 与 Finalizer 详解

说明

？？？ 说好只有四种引用呢，怎么又跑出来一个FinalReference？还有一个奇奇怪怪的Finalizer？

别别别，把枪放下，事情不是你想的那样。 

FinalReference虽然也是继承自Reference类，但是并不能直接使用它，因为它是包可见的。

```
class FinalReference<T> extends Reference<T> {
    public FinalReference(T referent, ReferenceQueue<? super T> q) {
        super(referent, q);
    }
}
```

也很简单明了，就这一个构造函数。既然是包可见，自然是为了来继承的，不直接提供给外部使用。

FinalReference由JVM来实例化，JVM会对那些实现了Object中finalize()方法的类对象实例化一个对应的FinalReference。 而事实上，JVM实际操作的是其子类——Finalizer，那么Finalizer是如何工作的呢?

Finalizer标记

类其实除了语法层面的显示标记（如final，abstract，public等等）之外，在JVM中其实还会给类标记其他一些符号，比如finalizer类，如果一个类覆盖了Object类的finalize方法，并且方法体非空，则这个类就是finalizer类，JVM会给它做一个标记，以下简称“f类”，GC在处理这种类的对象的时候会做一些特殊的处理，如在这个对象被回收之前会先调用其finalize方法。

Finalizer源码解析

在java.lang.ref包下，还有最后一个没有说到类，也就是FinalReference的子类——Finalizer，一听就是个专门给人善后的家伙。来看看它长什么样。 

```
final class Finalizer extends FinalReference<Object> {
    ...
}
```

TOC

能再被继承。

这个类是专门留给JVM去使用的，所以可以才如此设计，防止被篡改。

当加载一个类时，如果该类覆盖了finalize方法，并且方法体非空，那么这个类就会被JVM做上标记，每次实例化该类对象时，就会为其生成一个Finalizer对象，JVM会调用Finalizer.register()将这个对象注册到Finalizer的内部队列中。

成员变量

接下来看看Finalizer的成员变量：

```
1  private Finalizer
2          next = null,
3          prev = null;
```

Finalizer是类似双链表的结构，next指向其后一个节点，prev指向其前一个节点。

```
1  private static final Object lock = new Object();
```

这里也有一个lock对象用来做锁。

```
1  private static Finalizer unfinalized = null;
```

unfinalized用来链接所有f类对象，以下称其为“f类对象链表”。这是一个静态变量，目的是防止f类对象在执行finalize方法之前被GC回收掉。

```
1  private static ReferenceQueue<Object> queue = new ReferenceQueue<>();
```

queue是静态队列（单链表结构），JVM在回收对象时，如果发现它是F类对象，则将其从f类对象链表中取出，将它放入引用队列queue中，并通知FinalizerThread去消费。也就是说，发生GC时并不会直接回收该对象占用的内存，而是将其移入队列中，等到之后的一次或者几次GC时才真正回收其占用的内存。

构造函数

```
1  private Finalizer(Object finalizee) {
2      super(finalizee, queue);
3      add();
4  }
```

构造函数也是私有的，意味着无法在该类之外构建这类对象，在构造函数中调用add方法，将当前Finalizer插入到f类对象链表中。

内部方法

TOC

```
static void register(Object finalizee) {
    new Finalizer(finalizee);
}
```

没错，它也是给JVM调用的，那么问题来了，虚拟机会在什么时候调用这个函数呢?

也许你已经猜到了，在创建对象的时候，JVM会将当前对象传递给Finalizer.register方法，给它创建一个Finalizer并且添加到f类对象链表中。

另外，如果我们是通过clone的方式来复制对象时，如果被复制的对象是一个f类对象，那么在clone完成的时候也会调用Finalizer.register方法进行注册。

```
private void add() {
    synchronized (lock) {
        if (unfinalized != null) {
            this.next = unfinalized;
            unfinalized.prev = this;
        }
        unfinalized = this;
    }
}
```

add方法中，使用lock对象锁进行加锁操作，然后将当前对象注册到f类对象链表的头部节点。

```
private void remove() {
    synchronized (lock) {
        if (unfinalized == this) {
            if (this.next != null) {
                unfinalized = this.next;
            } else {
                unfinalized = this.prev;
            }
        }
        if (this.next != null) {
            this.next.prev = this.prev;
        }
        if (this.prev != null) {
            this.prev.next = this.next;
        }
        this.next = this;
        this.prev = this;
    }
}
```

remove方法中则同样以lock对象锁进行加锁后，将当前对象从f类对象链表中移除。并将next和prev均指向自身，这也用来判断f类对象是否已经被执行过finalize方法。

```
private boolean hasBeenFinalized() {
    return (next == this);
}
```

TOC

FinalizerThread线程

在Finalizer类的最后，有一段静态代码块，用来初始化FinalizerThread线程。

```
static {
    ThreadGroup tg = Thread.currentThread().getThreadGroup();
    for (ThreadGroup tgn = tg;
         tgn != null;
         tg = tgn, tgn = tg.getParent());
    Thread finalizer = new FinalizerThread(tg);
    finalizer.setPriority(Thread.MAX_PRIORITY - 2);
    finalizer.setDaemon(true);
    finalizer.start();
}
```

这跟之前说过的ReferenceHandler线程十分相似，但是很重要的一点区别是，这里设置的线程优先级并不是最高优先级，而是：

```
finalizer.setPriority(Thread.MAX_PRIORITY - 2);
```

所以，这意味着在CPU比较紧张的情况下，这条线程被调度的优先级可能会受到影响。

```
private static class FinalizerThread extends Thread {
    // 用来判断该线程是否已经启动的标志
    private volatile boolean running;
    FinalizerThread(ThreadGroup g) {
        super(g, "Finalizer");
    }
    public void run() {
        // 如果发生了递归调用则直接返回
        if (running)
            return;

        // Finalizer线程在 System.initializeSystemClass 被调用前启动
        // 需要等到JVM已经初始化完成才能执行
        while (!VM.isBooted()) {
            try {
                VM.awaitBooted();
            } catch (InterruptedException x) {
            }
        }
        final JavaLangAccess jla = SharedSecrets.getJavaLangAccess();
        running = true;
        for (;;) {
            try {
                // 将节点从队列中移除
                Finalizer f = (Finalizer)queue.remove();
                // 调用其runFinalizer方法
                f.runFinalizer(jla);
            } catch (InterruptedException x) {
                // 出错直接忽略
            }
        }
```

TOC

这个线程的逻辑并不复杂，等待JVM初始化完成后，便开启死循环模式，从引用队列中阻塞式获取元素，并执行其runFinalizer方法。注意这里的try...catch语句，捕获到异常都是忽略处理，所以**如果在类的finalize方法中如果抛出异常，你是得不到任何错误信息的**。

```
private void runFinalizer(JavaLangAccess jla) {
    synchronized (this) {
        // 先判断其是否已经被执行过finalize方法
        if (hasBeenFinalized()) return;
        remove();
    }
    try {
        // 取出其引用的对象
        Object finalizee = this.get();
        // 如果不为null且不是Enum对象
        if (finalizee != null && !(finalizee instanceof java.lang.Enum)) {
            // 执行其finalize方法
            jla.invokeFinalize(finalizee);

            // 清空包含该变量的堆栈，以减少被保守型GC保留的可能性
            finalizee = null;
        }
    } catch (Throwable x) { }
    // 调用Reference的clear方法
    super.clear();
}
```

这里的同步代码块只有最前面的一小段，先判断是否已经执行过finalize方法，如果已经执行过，则直接返回。所以**一个对象finalize方法最多只会被执行一次**。所以如果在f类对象的finalize方法中，重新使用全局变量给它关联一个强引用，使其变成一个强可达对象，当这个对象再次变成不可达的对象的时候，就不会再执行它的finalize方法了。这一点在《深入理解JVM虚拟机》一书中有讲到。

该方法在判断完之后，取出Finalizer的内部引用对象，执行其finalize方法，并将其置为null。

SecondaryFinalizer线程

emmm....除了上面那条线程之外，还有两条辅助线程，在runFinalization方法和runAllFinalizers方法中调用。前一个方法将依次取出queue中的Finalizer并执行其runFinalizer方法，后一个方法则会依次对f类对象链表中的对象执行runFinalizer方法。

```
static void runFinalization() {
    if (!VM.isBooted()) {
        return;
    }

    forkSecondaryFinalizer(new Runnable() {
        private volatile boolean running;
        public void run() {
            // 如果是递归调用，则直接返回
            if (running)
                return;
            final JavaLangAccess jla = SharedSecrets.getJavaLangAccess();
            running = true;
            for (;;) {
                Finalizer f = (Finalizer)queue.poll();
```

TOC

```
            }
        }
    });
}
```

runFinalization方法对比一下上面的FinalizerThread的run方法便发现其实几乎一样。这是提供给其他类调用的，但Finalizer是包访问权限，所以其他类（如Runtime、Shutdown）并不是直接调用，而是通过JVM间接调用。

例如，调用System.runFinalization方法时，便会调用Runtime.runFinalization方法，最终通过虚拟机，调用Finalizer.runFinalization方法。

再来看看runAllFinalizers方法。

```
static void runAllFinalizers() {
    if (!VM.isBooted()) {
        return;
    }

    forkSecondaryFinalizer(new Runnable() {
        private volatile boolean running;
        public void run() {
            // 如果是递归调用，则直接返回
            if (running)
                return;
            final JavaLangAccess jla = SharedSecrets.getJavaLangAccess();
            running = true;
            for (;;) {
                Finalizer f;
                synchronized (lock) {
                    f = unfinalized;
                    if (f == null) break;
                    unfinalized = f.next;
                }
                f.runFinalizer(jla);
            }}});
}
```

这里的处理与上面也很相似，只是将queue换成了unfinalized链表。

在java.lang.ShutDown类中的sequence方法中，会调用runAllFinalizer方法：

```
if (rfoe) runAllFinalizers();
```

而这个方法其实是一个本地方法，由JVM间接调用Finalizer的runAllFinalizer方法。

```
/* Wormhole for invoking java.lang.ref.Finalizer.runAllFinalizers */
private static native void runAllFinalizers();
```

这两个方法中都用到了同一个模板方法——forkSecondaryFinalizer：

```
private static void forkSecondaryFinalizer(final Runnable proc) {
```

TOC

```
            public void run() {
                ThreadGroup tg = Thread.currentThread().getThreadGroup();
                for (ThreadGroup tgn = tg;
                     tgn != null;
                     tg = tgn, tgn = tg.getParent());
                Thread sft = new Thread(tg, proc, "Secondary finalizer");
                sft.start();
                try {
                    sft.join();
                } catch (InterruptedException x) {
                    Thread.currentThread().interrupt();
                }
                return null;
            }});
    }
```

这里调用了AccessController.doPrivileged方法，这个方法的作用是使其内部的代码段获得更大的权限，可以在里面访问更多的资源。这个涉及到另一个话题，如果想要了解的话可以参考这篇文章——Java安全模型。

这里你只需要关注run方法即可，run方法里只是启动一个线程的模板代码。

Finalizer与内存泄漏

利用finalize来释放资源，听起来好像挺不错的，但是事实上却并没有想象中那么好，很容易会导致内存泄漏。

通常而言，你不会知道垃圾回收器何时进行垃圾回收，也不知道何时回收某个特定的对象。但你可能会关心对象的finalize方法是否被执行。Java规范中对于Finalizer有以下规定;

> 在回收一个有Finalizer关联的对象的内存之前，垃圾回收器会先调用其finalizer中的方法（即执行对象的finalize方法）。

但是由于你并不知道对象何时被垃圾回收器收集，你只知道对象的finalize方法最终会被执行。所以必须清楚的一点是，你不会知道一个对象的finalize方法何时被执行。所以不要设计一个需要依赖程序finalize及时执行的程序。

使用finalize一个经典的用法便是在构造器中打开文件，然后在finalize方法中关闭文件。这个设计看似很整洁完美，实际上隐藏一个隐秘的bug，Java中文件句柄数量是有限的，如果所有的句柄都用完了，那么程序将会无法打开任何文件。

这样使用finalize方法，在某些经常执行finalization以确保有足够多可用句柄的JVM中可能工作良好，但是在另一些JVM中可能无法正常工作，因为那些垃圾回收器并不会经常执行finalization来确保有足够的句柄可用。

此外FinalizeThread线程的优先级并不是最高的，所有当CPU资源紧张时，可能会有相当长一段时间不会执行Finalizer队列中的f对象的finalize方法，从而导致内存泄漏的发生。

对于这些即将被回收掉的f对象，并不会在最近的一次GC中马上被回收释放掉，而是会延迟到下一个或几个GC时才会被真正回收。finalize方法无法在GC过程中执行，第一次GC只会讲其放入队列中去，由FinalizerThread去轮询执行。

所以，不要在运行期间不断创建f对象，否则内存泄漏将常伴你左右。 

TOC

入，又有对象移出时，你无法知道这些Finalizer对象的具体执行顺序，所以不要设计依赖Finalizer执行顺序的程序。

当然，如果你不得不使用finalize方法，并且需要确保其被执行，可以在代码中显式调用System.runFinalization方法。

Finalizer 应用场景

好嘛，叽叽歪歪介绍了这么一大堆，结果都在说Finalizer怎么怎么不好，怎么怎么会出错。那要它何用？ 

嗯，自有妙用。Finalizer一个比较适合的场景便是释放nativa方法中申请的内存，如果一个对象调用了本地方法，并且申请了内存（例如C中的malloc方法），那么可以在这个对象的finalize方法中调用native方法进行内存释放（如free方法），因为在这种情况下，本地方法申请的内存不会被垃圾回收器自动回收。

另一个更常见的用法是为释放非内存资源（如：文件句柄、sockets）提供一个反馈机制。之前提到，你不应该依赖Finalizer来释放这些有限的资源。你应该提供一个释放这些资源的方法。但是你仍希望有一个Finalizer来检查这些资源是否已经被释放，如果没有则将其释放。相当于做一个防护措施，因为当你的代码被其他程序员调用时，也许他会粗心大意的忘记调用释放资源的方法。

小结

终于讲完了，现在来小结一下。

- FinalReference是为处理对象的finalize方法而设计的
- 如果一个类或者其父类覆盖了Object类的finalize方法，那么这个类就叫做f类，会被JVM特殊标记
- f类对象在创建时会顺便注册一个与其关联的Finalizer对象
- f类对象在其不可达时会在GC中被放入引用队列
- f类对象的finalize方法执行时间并不确定，f对象至少要经历两次GC才能被回收，有可能执行finalize期间已经经历了多次GC
- Finalizer对象的处理是在GC时进行的，如果没有触发GC就不会触发对Finalizer对象的处理，unfinalized队列中的对象也就不会被放入队列，其finalize方法也不会被执行
- 依赖f类对象的finalize执行顺序和执行时间的程序很可能会出现内存泄漏
- 因为f对象的finalize方法迟迟没有执行，有可能会导致大部分f对象进入到old分代，此时容易导致老年代的GC，甚至Full GC，会使GC暂停时间明显变长

最后更新时间：2019-01-08 01:02:40
本文作者： 清风
本文链接：https://www.mfrank.cn/programming/java/reference/final-reference-code-detail.html

Frank

赏

TOC

Prev
强引用、软引用、弱引用和虚引用深入探讨

Next
Java 多线程篇

TOC