Java引用总结--StrongReference、SoftReference、WeakReference、PhantomReference

Java引用总结--StrongReference、SoftReference、WeakReference、PhantomReference

1 Java引用介绍

Java从1.2版本开始引入了4种引用，这4种引用的级别由高到低依次为：

强引用 > 软引用 > 弱引用 > 虚引用

(1)强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

(2)软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

(3)弱引用（WeakReference）

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周

期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

⑷虚引用（PhantomReference）

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

由于引用和内存回收关系紧密。下面，先通过实例对内存回收有个认识；然后，进一步通过引用实例加深对引用的了解。

2 内存回收

创建公共类MyDate，它的作用是覆盖finalize()函数：在finalize()中输出打印信息，方便追踪。

说明：finalize()函数是在JVM回收内存时执行的，但JVM并不保证在回收内存时一定会调用finalize()。

MyDate代码如下：

```
package com.skywang.java;
```

```
import java.util.Date;

public class MyDate extends Date {

    /** Creates a new instance of MyDate */
    public MyDate() {
    }
    // 覆盖finalize()方法
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("obj [Date: " + this.getTime() + "] is gc");
    }

    public String toString() {
        return "Date: " + this.getTime();
    }
}
```

在这个类中，对java.util.Date类进行了扩展，并重写了finalize()和toString()方法。

创建公共类ReferenceTest，它的作用是定义一个方法drainMemory()：消耗大量内存，以此来引发JVM回收内存。

ReferenceTest代码如下：

```
package com.skywang.java;

public class ReferenceTest {
    /** Creates a new instance of ReferenceTest */
    public ReferenceTest() {
    }

    // 消耗大量内存
    public static void drainMemory() {
        String[] array = new String[1024 * 10];
        for(int i = 0; i < 1024 * 10; i++) {
            for(int j = 'a'; j <= 'z'; j++) {
                array[i] += (char)j;
            }
        }
    }
}
```

在这个类中定义了一个静态方法drainMemory()，此方法旨在消耗大量的内存，促使JVM运行垃圾回收。

有了上面两个公共类之后，我们即可测试JVM什么时候进行垃圾回收。下面分3种情况进行测试：

情况1：清除对象

实现代码：

```
package com.skywang.java;

public class NoGarbageRetrieve {

    public static void main(String[] args) {
        MyDate date = new MyDate();
        date = null;
    }
}
```

运行结果：

<无任何输出>

结果分析：date虽然设为null，但由于JVM没有执行垃圾回收操作，MyDate的finalize()方法没有被运行。

情况2：显式调用垃圾回收

实现代码：

```
package com.skywang.java;

public class ExplicitGarbageRetrieve {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
```

```
        MyDate date = new MyDate();
        date = null;
        System.gc();
    }

}
```

运行结果：

obj [Date: 1372137067328] is gc

结果分析：调用了System.gc()，使JVM运行垃圾回收，MyDate的finalize()方法被运行。

情况3：隐式调用垃圾回收

实现代码：

```
package com.skywang.java;

public class ImplicitGarbageRetrieve {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyDate date = new MyDate();
        date = null;
        ReferenceTest.drainMemory();
    }

}
```

运行结果：

obj [Date: 1372137171965] is gc

结果分析：虽然没有显式调用垃圾回收方法System.gc()，但是由于运行了耗费大量内存的方法，触发JVM进行垃圾回收。

总结：JVM的垃圾回收机制，在内存充足的情况下，除非你显式调用

System.gc()，否则它不会进行垃圾回收；在内存不足的情况下，垃圾回收将自动运行

3、Java对引用的分类

3.1 强引用

实例代码：

```
package com.skywang.java;

public class StrongReferenceTest {

    public static void main(String[] args) {
        MyDate date = new MyDate();
        System.gc();
    }
}
```

运行结果：

<无任何输出>

结果说明：即使显式调用了垃圾回收，但是用于date是强引用，date没有被回收。

3.2 软引用

实例代码：

```
package com.skywang.java;

import java.lang.ref.SoftReference;

public class SoftReferenceTest {

    public static void main(String[] args) {
        SoftReference ref = new SoftReference(new MyDate());
```

```
        ReferenceTest.drainMemory();
    }
}
```

运行结果：

<无任何输出>

结果说明：在内存不足时，软引用被终止。软引用被禁止时，

SoftReference ref = new SoftReference(new MyDate());
ReferenceTest.drainMemory();

等价于

MyDate date = new MyDate();

// 由JVM决定运行
If(JVM.内存不足()) {
date = null;
System.gc();
}

3.3 弱引用

示例代码：

```
package com.skywang.java;

import java.lang.ref.WeakReference;

public class WeakReferenceTest {

    public static void main(String[] args) {
        WeakReference ref = new WeakReference(new MyDate());
        System.gc();
    }
}
```

运行结果：

obj [Date: 1372142034360] is gc

结果说明：在JVM垃圾回收运行时，弱引用被终止.

```
WeakReference ref = new WeakReference(new MyDate());
System.gc();
```

等同于：

```
MyDate date = new MyDate();

// 垃圾回收
If(JVM.内存不足()) {
date = null;
System.gc();
}
```

3. 4 假象引用

示例代码：

```
package com.skywang.java;

import java.lang.ref.ReferenceQueue;
import java.lang.ref.PhantomReference;

public class PhantomReferenceTest {

    public static void main(String[] args) {
        ReferenceQueue queue = new ReferenceQueue();
        PhantomReference ref = new PhantomReference(new MyDate(), queue);
        System.gc();
    }
}
```

运行结果：

obj [Date: 1372142282558] is gc

结果说明：假象引用，在实例化后，就被终止了。

```
ReferenceQueue queue = new ReferenceQueue();
PhantomReference ref = new PhantomReference(new MyDate(), queue);
System.gc();
```

等同于：

```
MyDate date = new MyDate();
date = null;
```

可以用以下表格总结上面的内容：

级别	什么时候被垃圾回收	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	在内存不足时	对象简单？缓存	内存不足时终止
弱引用	在垃圾回收时	对象缓存	gc运行后终止
虚引用	Unknown	Unknown	Unknown

点击下载：源代码