jvm系列 (四) ---强、软、弱、虚引用

<http://www.cnblogs.com/-new/p/7491269.html>

# [jvm系列 (四) ---强、软、弱、虚引用](http://www.cnblogs.com/-new/p/7491269.html)

# java引用

## 目录

* [jvm系列（一）：jvm内存区域与溢出](http://www.cnblogs.com/-new/p/7244460.html)
* [jvm系列（二）：垃圾收集器与内存分配策略](http://www.cnblogs.com/-new/p/7284368.html)
* [jvm系列（三）：锁的优化](http://www.cnblogs.com/-new/p/7427036.html)
* [我的博客目录](http://www.cnblogs.com/-new/p/7396648.html)

## 为什么将引用分为不同的强度

* 因为我们需要实现这样一种情况，当内存足够的时候，继续保留，内存空间不够的后则可以回收。

## 强引用

* 只要强引用还在，被引用的对象不会被回收

People jiajun=new People();

## 软引用

* 系统将要发生内存溢出异常之前，会回收软引用的对象，如果回收后还没有足够的内存，抛出内存溢出异常
* 使用SoftReference类，将要软引用的对象最为参数传入,
* 传入ReferenceQuue队列的时候，如果引用的对象被回收，这个引用加入到关联的引用队列

SoftReference(T referent)

创建引用给定对象的新的软引用。

SoftReference(T referent, ReferenceQueue<? super T> q)

创建一个引用给定对象的新的软引用，并向给定队列注册该引用

## 弱引用

* 弱引用的对象只能存活在下一次垃圾回收之前，回收时，不管内存是否足够，都会将弱引用的对象回收
* 使用WeakReference类，将要弱引用的对象作为参数传入
* 传入ReferenceQuue队列的时候，如果引用的对象被回收，这个引用加入到关联的引用队列中

WeakReference(T referent)

创建引用给定对象的新的弱引用。

WeakReference(T referent, ReferenceQueue<? super T> q)

创建引用给定对象的新的弱引用，并向给定队列注册该引用。

## 虚引用

* 虚引用和没有任何引用一样，任何时候都可能被回收
* 为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。用于跟踪对象的回收状态
* 使用PhantoReference类，将要虚引用的对象作为参数传入，此时还需要一个ReferenceQueue 对象
* 对象被回收后，把这个虚引用加入到与之 关联的引用队列中。

PhantomReference(T referent, ReferenceQueue<? super T> q)

创建一个引用给定对象的新的虚引用，并向给定队列注册它。

## ReferenceQueue的作用

* public Reference poll()：从队列中取出一个元素，队列为空则返回null
* public Reference remove()：从队列中出对一个元素，若没有则阻塞至有可出队元素
* Reference对象所引用的对象被GC回收时，该Reference对象将会被加入引用队列中

ReferenceQueue queue = new ReferenceQueue();

SoftReference ref=new SoftReference(object, queue);

object =null;

* 当object对象被回收的时候，此时通过ref.get()方法get出来的是null，说明这个对象已经被回收，但是，此时这个ref对象还是存在的，但是现在已经没作用了。引用队列poll出来的是引用对象。所以可以有下面的操作来回收ref对象

SoftReference ref = null;

while ((ref = (EmployeeRef) q.poll()) != null) {

// 清除ref

}

# 使用场景

## 分析

Student s1=new Student();

HashMap<Student, String> map=new HashMap();

map.put(s1,"jiajun");

s1=null;

* 在这种情况下，我们想要是student对象被回收，虽然s1=null，但是hashmap中还有对student对象的引用，所以并不能被回收
* 因此jdk为我们提供了一个WeakHashMap,通过弱引用来管理entry

private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V>

## 实验

public class Test {

public static void main(String[] args) {

String s1=new String("1");

String s2=new String("2");

String s3=new String("3");

Map map=new WeakHashMap();

map.put(s1, "one");

map.put(s2, "two");

map.put(s3, "three");

s1=null;

System.gc();

Iterator iter = map.entrySet().iterator();

while (iter.hasNext()) {

Map.Entry en = (Map.Entry)iter.next();

System.out.printf("next : %s - %s\n",en.getKey(),en.getValue());

}

}

}

* 输出：next : 2 - two next : 3 - three
* 将WeakHashMap修改为HashMap，可以发现输出结果是有三个，在这里可以发现弱引用的作用了

## 总结

| **比较** | **强引用** | **软引用** | **弱引用** | **虚引用** |
| --- | --- | --- | --- | --- |
| 使用 | People p=new People() | SoftReference s=new SoftReference(p) | WeakReference w=new WeakReference(p) | PhantomReference r=new PhantomReference (p,referenceQueue) |
| 回收情况 | 有强引用的时候不会被回收 | 当要内存溢出内存异常的时候，回收软引用的对象 | 下一次回收一定将他回收 |  |
| 适用情况 |  | 有用但非必须对象 | 非必须对象 |  |