[**Java/Android引用类型及其使用分析**](http://www.cnblogs.com/lwbqqyumidi/p/4151833.html)

Java/Android中有四种引用类型，分别是：

Strong reference     - 强引用  
Soft Reference        - 软引用  
Weak Reference      - 弱引用  
Phantom Reference - 虚引用

不同的引用类型有着不同的特性，同时也对应着不同的使用场景。

**1.Strong reference - 强引用**

实际编码中最常见的一种引用类型。常见形式如：A a = new A();等。强引用本身存储在栈内存中，其存储指向对内存中对象的地址。一般情况下，当对内存中的对象不再有任何强引用指向它时，垃圾回收机器开始考虑可能要对此内存进行的垃圾回收。如当进行编码：a = null，此时，刚刚在堆中分配地址并新建的a对象没有其他的任何引用，当系统进行垃圾回收时，堆内存将被垃圾回收。

SoftReference、WeakReference、PhantomReference都是类java.lang.ref.Reference的子类。Reference作为抽象基类，定义了其子类对象的基本操作。Reference子类都具有如下特点：  
1.Reference子类不能无参化直接创建，必须至少以强引用对象为构造参数，创建各自的子类对象；  
2.因为1中以强引用对象为构造参数创建对象，因此，使得原本强引用所指向的堆内存中的对象将不再只与强引用本身直接关联，与Reference的子类对象的引用也有一定联系。且此种联系将可能影响到对象的垃圾回收。

根据不同的子类对象对其指示对象（强引用所指向的堆内存中的对象）的垃圾回收不同的影响特点，分别形成了三个子类，即SoftReference、WeakReference和PhantomReference。

**2.Soft Reference - 软引用**

软引用的一般使用形式如下：  
A a = new A();  
SoftReference<A> srA = new SoftReference<A>(a);

通过对象的强引用为参数，创建了一个SoftReference对象，并使栈内存中的wrA指向此对象。

此时，进行如下编码：a = null，对于原本a所指向的A对象的垃圾回收有什么影响呢？

先直接看一下下面一段程序的输出结果：

[复制代码](javascript:void(0);)

1 import java.lang.ref.SoftReference;

2

3 public class ReferenceTest {

4

5 public static void main(String[] args) {

6

7 A a = new A();

8

9 SoftReference<A> srA = new SoftReference<A>(a);

10

11 a = null;

12

13 if (srA.get() == null) {

14 System.out.println("a对象进入垃圾回收流程");

15 } else {

16 System.out.println("a对象尚未被回收" + srA.get());

17 }

18

19 // 垃圾回收

20 System.gc();

21

22 if (srA.get() == null) {

23 System.out.println("a对象进入垃圾回收流程");

24 } else {

25 System.out.println("a对象尚未被回收" + srA.get());

26 }

27

28 }

29 }

30

31 class A {

32

33 }

[复制代码](javascript:void(0);)

##输出结果为：

1 a对象尚未被回收A@4807ccf6

2 a对象尚未被回收A@4807ccf6

当 a = null后，堆内存中的A对象将不再有任何的强引用指向它，但此时尚存在srA引用的对象指向A对象。当第一次调用srA.get()方法返回此指示对象时，由于垃圾回收器很有可能尚未进行垃圾回收，此时get()是有结果的，这个很好理解。当程序执行System.gc();强制垃圾回收后，通过srA.get()，发现依然可以得到所指示的A对象，说明A对象并未被垃圾回收。那么，软引用所指示的对象什么时候才开始被垃圾回收呢？需要满足如下两个条件：

1.当其指示的对象没有任何强引用对象指向它；

2.当虚拟机内存不足时。

因此，SoftReference变相的延长了其指示对象占据堆内存的时间，直到虚拟机内存不足时垃圾回收器才回收此堆内存空间。

**3.Weak Reference - 弱引用**

同样的，软引用的一般使用形式如下：  
A a = new A();  
WeakReference<A> wrA = new WeakReference<A>(a);

当没有任何强引用指向此对象时， 其垃圾回收又具有什么特性呢？

[复制代码](javascript:void(0);)

1 import java.lang.ref.WeakReference;

2

3 public class ReferenceTest {

4

5 public static void main(String[] args) {

6

7 A a = new A();

8

9 WeakReference<A> wrA = new WeakReference<A>(a);

10

11 a = null;

12

13 if (wrA.get() == null) {

14 System.out.println("a对象进入垃圾回收流程");

15 } else {

16 System.out.println("a对象尚未被回收" + wrA.get());

17 }

18

19 // 垃圾回收

20 System.gc();

21

22 if (wrA.get() == null) {

23 System.out.println("a对象进入垃圾回收流程");

24 } else {

25 System.out.println("a对象尚未被回收" + wrA.get());

26 }

27

28 }

29

30 }

31

32 class A {

33

34 }

[复制代码](javascript:void(0);)

##输出结果为：

a对象尚未被回收A@52e5376a

a对象进入垃圾回收流程

输出的第一条结果解释同上。当进行垃圾回收后，wrA.get()将返回null，表明其指示对象进入到了垃圾回收过程中。因此，对弱引用特点总结为：

WeakReference不改变原有强引用对象的垃圾回收时机，一旦其指示对象没有任何强引用对象时，此对象即进入正常的垃圾回收流程。

那么，依据此特点，很可能有疑问：WeakReference存在又有什么意义呢？

其主要使用场景见于：当前已有强引用指向强引用对象，此时由于业务需要，需要增加对此对象的引用，同时又不希望改变此引用的垃圾回收时机，此时WeakReference正好符合需求，常见于一些与生命周期的场景中。

下面给出一个Android中关于WeakReference使用的场景 —— 结合静态内部类和WeakReference来解决Activity中可能存在的Handler内存泄露问题。

Activity中我们需要新建一个线程获取数据，使用handler - sendMessage方式。下面是这一过程的一般性代码：

[复制代码](javascript:void(0);)

1 public class MainActivity extends Activity {

2

3 //...

4 private int page;

5 private Handler handler = new Handler() {

6

7 @Override

8 public void handleMessage(Message msg) {

9 if (msg.what == 1) {

10

11 //...

12

13 page++;

14 } else {

15

16 //...

17

18 }

19

20 };

21 };

22

23 @Override

24 protected void onCreate(Bundle savedInstanceState) {

25 super.onCreate(savedInstanceState);

26 setContentView(R.layout.activity\_main);

27

28 //...

29

30 new Thread(new Runnable() {

31 @Override

32 public void run() {

33 //..

34 Message msg = Message.obtain();

35 msg.what = 1;

36 //msg.obj = xx;

37 handler.sendMessage(msg);

38 }

39 }).start();

40

41 //...

42

43 }

44

45 }

[复制代码](javascript:void(0);)

在Eclispe中Run Link，将会看到警示信息：This Handler class should be static or leaks might occur ...点击查看此信息，其详情中对问题进行了说明并给出了建议性的解决方案。

[复制代码](javascript:void(0);)

Issue: Ensures that Handler classes do not hold on to a reference to an outer class

Id: HandlerLeak

Since this Handler is declared as an inner class, it may prevent the outer class from being garbage collected. If the Handler is using a Looper or MessageQueue for a thread other than the main thread, then there is no issue. If the Handler is using the Looper or MessageQueue of the main thread, you need to fix your Handler declaration, as follows: Declare the Handler as a static class;In the outer class, instantiate a WeakReference to the outer class and pass this object to your Handler when you instantiate the Handler; Make all references to members of the outer class using the WeakReference object.

[复制代码](javascript:void(0);)

大致的意思是建议将Handler定义成内部静态类，并在此静态内部类中定义一个WeakReference的引用，由于指示外部的Activity对象。

问题分析：

Activity具有自身的生命周期，Activity中新开启的线程运行过程中，可能此时用户按下了Back键，或系统内存不足等希望回收此Activity，由于Activity中新起的线程并不会遵循Activity本身的什么周期，也就是说，当Activity执行了onDestroy,由于线程以及Handler 的HandleMessage的存在，使得系统本希望进行此Activity内存回收不能实现，因为非静态内部类中隐性的持有对外部类的引用，导致可能存在的内存泄露问题。

因此，在Activity中使用Handler时，一方面需要将其定义为静态内部类形式，这样可以使其与外部类（Activity）解耦，不再持有外部类的引用，同时由于Handler中的handlerMessage一般都会多少需要访问或修改Activity的属性，此时，需要在Handler内部定义指向此Activity的WeakReference，使其不会影响到Activity的内存回收同时，可以在正常情况下访问到Activity的属性。

 Google官方给出的建议写法为：

[复制代码](javascript:void(0);)

1 public class MainActivity extends Activity {

2

3 //...

4 private int page;

5 private MyHandler mMyHandler = new MyHandler(this);

6

7 private static class MyHandler extends Handler {

8

9 private WeakReference<MainActivity> wrActivity;

10

11 public MyHandler(MainActivity activity) {

12 this.wrActivity = new WeakReference<MainActivity>(activity);

13 }

14

15 @Override

16 public void handleMessage(Message msg) {

17 if (wrActivity.get() == null) {

18 return;

19 }

20 MainActivity mActivity = wrActivity.get();

21 if (msg.what == 1) {

22

23 //...

24 mActivity.page++;

25

26 } else {

27

28 //...

29

30 }

31 }

32

33 }

34

35 @Override

36 protected void onCreate(Bundle savedInstanceState) {

37 super.onCreate(savedInstanceState);

38 setContentView(R.layout.activity\_main);

39

40 //...

41

42 new Thread(new Runnable() {

43 @Override

44 public void run() {

45 //..

46 Message msg = Message.obtain();

47 msg.what = 1;

48 //msg.obj = xx;

49 mMyHandler.sendMessage(msg);

50 }

51 }).start();

52

53 //...

54

55 }

56

57 }

[复制代码](javascript:void(0);)

 对于SoftReference和WeakReference，还有一个构造器参数为ReferenceQueue<T>，当SoftReference或WeakReference所指示的对象确实被垃圾回收后，其引用将被放置于ReferenceQueue中。注意上文中，当SoftReference或WeakReference的get()方法返回null时，仅是表明其指示的对象已经进入垃圾回收流程，此时对象不一定已经被垃圾回收。而只有确认被垃圾回收后，如果ReferenceQueue，其引用才会被放置于ReferenceQueue中。

看下面的一个例子：

[复制代码](javascript:void(0);)

1 public class ReferenceTest {

2

3 public static void main(String[] args) {

4

5 A a = new A();

6

7 WeakReference<A> wrA = new WeakReference<A>(a);

8

9 a = null;

10

11 if (wrA.get() == null) {

12 System.out.println("a对象进入垃圾回收流程");

13 } else {

14 System.out.println("a对象尚未被回收" + wrA.get());

15 }

16

17 // 垃圾回收

18 System.gc();

19

20 if (wrA.get() == null) {

21 System.out.println("a对象进入垃圾回收流程");

22 } else {

23 System.out.println("a对象尚未被回收" + wrA.get());

24 }

25

26 }

27 }

28

29 class A {

30

31 @Override

32 protected void finalize() throws Throwable {

33 super.finalize();

34 System.out.println("in A finalize");

35 }

36

37 }

[复制代码](javascript:void(0);)

##输出结果为：

1 a对象尚未被回收A@46993aaa

2 a对象被回收

3 in A finalize

由此，也验证了上文中的“进入垃圾回收流程”的说法。下面结合ReferenceQueue，看一段代码：

[复制代码](javascript:void(0);)

1 public class ReferenceTest {

2

3 public static void main(String[] args) {

4

5 A a = new A();

6

7 ReferenceQueue<A> rq = new ReferenceQueue<A>();

8 WeakReference<A> wrA = new WeakReference<A>(a, rq);

9

10 a = null;

11

12 if (wrA.get() == null) {

13 System.out.println("a对象进入垃圾回收流程");

14 } else {

15 System.out.println("a对象尚未被回收" + wrA.get());

16 }

17

18 System.out.println("rq item:" + rq.poll());

19

20 // 垃圾回收

21 System.gc();

22

23 if (wrA.get() == null) {

24 System.out.println("a对象进入垃圾回收流程");

25 } else {

26 System.out.println("a对象尚未被回收" + wrA.get());

27 }

28

29 /\*

30 try {

31 Thread.sleep(1000);

32 } catch (InterruptedException e) {

33 e.printStackTrace();

34 }

35 \*/

36

37 System.out.println("rq item:" + rq.poll());

38

39 }

40 }

41

42 class A {

43

44 @Override

45 protected void finalize() throws Throwable {

46 super.finalize();

47 System.out.println("in A finalize");

48 }

49

50 }

[复制代码](javascript:void(0);)

##输出结果为：

1 a对象尚未被回收A@302b2c81

2 rq item:null

3 a对象进入垃圾回收流程

4 rq item:null

5 in A finalize

由此，验证了“仅进入垃圾回收流程的SoftReference或WeakReference引用尚未被加入到ReferenceQueue”。

[复制代码](javascript:void(0);)

1 public class ReferenceTest {

2

3 public static void main(String[] args) {

4

5 A a = new A();

6

7 ReferenceQueue<A> rq = new ReferenceQueue<A>();

8 WeakReference<A> wrA = new WeakReference<A>(a, rq);

9

10 a = null;

11

12 if (wrA.get() == null) {

13 System.out.println("a对象进入垃圾回收流程");

14 } else {

15 System.out.println("a对象尚未被回收" + wrA.get());

16 }

17

18 System.out.println("rq item:" + rq.poll());

19

20 // 垃圾回收

21 System.gc();

22

23 if (wrA.get() == null) {

24 System.out.println("a对象进入垃圾回收流程");

25 } else {

26 System.out.println("a对象尚未被回收" + wrA.get());

27 }

28

29 try {

30 Thread.sleep(1);

31 } catch (InterruptedException e) {

32 e.printStackTrace();

33 }

34

35 System.out.println("rq item:" + rq.poll());

36

37 }

38 }

39

40 class A {

41

42 @Override

43 protected void finalize() throws Throwable {

44 super.finalize();

45 System.out.println("in A finalize");

46 }

47

48 }

[复制代码](javascript:void(0);)

##输出结果为：

1 a对象尚未被回收A@6276e1db

2 rq item:null

3 a对象进入垃圾回收流程

4 in A finalize

5 rq item:java.lang.ref.WeakReference@645064f

由此，证实了上述说法。

**4.PhantomReference**

与SoftReference或WeakReference相比，PhantomReference主要差别体现在如下几点：

1.PhantomReference只有一个构造函数PhantomReference(T referent, ReferenceQueue<? super T> q)，因此，PhantomReference使用必须结合ReferenceQueue；

2.不管有无强引用指向PhantomReference的指示对象，PhantomReference的get()方法返回结果都是null。

[复制代码](javascript:void(0);)

1 public class ReferenceTest {

2

3 public static void main(String[] args) {

4

5 A a = new A();

6

7 ReferenceQueue<A> rq = new ReferenceQueue<A>();

8 PhantomReference<A> prA = new PhantomReference<A>(a, rq);

9

10 System.out.println("prA.get():" + prA.get());

11

12 a = null;

13

14 System.gc();

15

16 try {

17 Thread.sleep(1);

18 } catch (InterruptedException e) {

19 e.printStackTrace();

20 }

21

22 System.out.println("rq item:" + rq.poll());

23

24 }

25 }

26

27 class A {

28

29 }

[复制代码](javascript:void(0);)

##输出结果为：

1 prA.get():null

2 rq item:java.lang.ref.PhantomReference@1da12fc0

代码中的Thread.sleep(1);作用与上例中相同，都是确保垃圾回收线程能够执行。否则，进进入垃圾回收流程而没有真正被垃圾回收的指示对象的虚引用是不会被加入到PhantomReference中的。

与WeakReference相同，PhantomReference并不会改变其指示对象的垃圾回收时机。且可以总结出：ReferenceQueue的作用主要是用于监听SoftReference/WeakReference/PhantomReference的指示对象是否已经被垃圾回收。