http://blog.csdn.net/lyfi01/article/details/6415726
http://lionheartyd.iteye.com/blog/1902002

Java.lang.ref 是 Java 类库中比较特殊的一个包，它提供了与 Java 垃圾回收器密切相关的引用类。这些引用类对象可以指向其它对象，但它们不同于一般的引用，因为它们的存在并不防碍 Java 垃圾回收器对它们所指向的对象进行回收。其好处就在于使者可以保持对使用对象的引用，同时 JVM 依然可以在内存不够用的时候对使用对象进行回收。因此这个包在用来实现与缓存相关的应用时特别有用。同时该包也提供了在对象的“可达”性发生改变时，进行提醒的机制。

java.lang.ref 这个包的结构，如图 1 所示

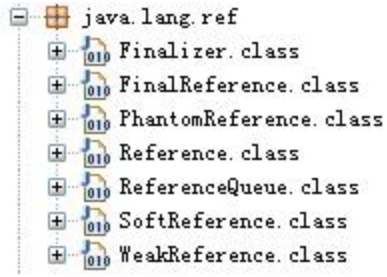
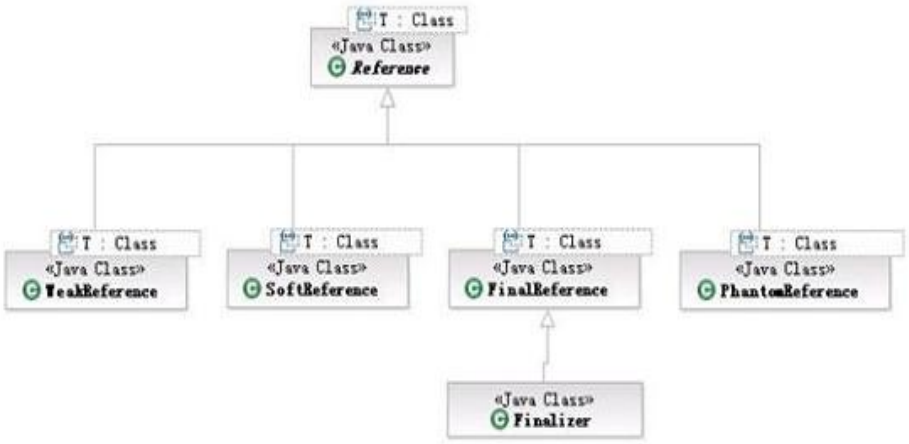
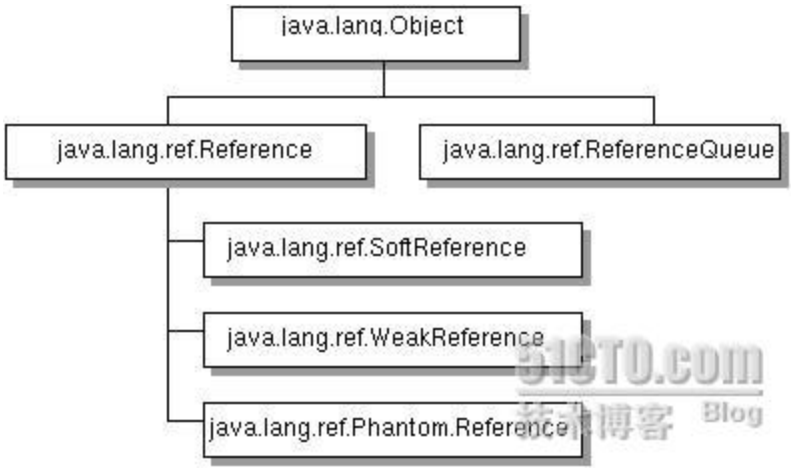


图 2. java.lang.ref 包中类的继承关系：





StrongReference： 强引用

我们都知道 JVM 中对象是被分配在堆（heap）上的，当程序行动中不再有引用指向这个对象时，这个对象就可以被垃圾回收器所回收。这里所说的引用也就是我们一般意义上申明的对象类型的变量（如 String, Object, ArrayList 等），区别于原始数据类型的变量（如 int, short, long 等）也称为强引用。

```
String tag = new String("T");
```

此处的 tag 引用就称之为强引用。而强引用有以下特征：

1. **强引用可以直接访问目标对象。**
2. **强引用所指向的对象在任何时候都不会被系统回收。**
3. **强引用可能导致内存泄漏。**

SoftReference： 软引用

SoftReference 在“弱引用”中属于最强的引用。SoftReference 所指向的对象，当没有强引用指向它时，会在内存中停留一段的时间，垃圾回收器会根据 JVM 内存的使用情况（内存的紧缺程度）以及 SoftReference 的 get() 方法的调用情况来决定是否对其进行回收。（后面章节会用几个实验进行阐述）

具体使用一般是通过 SoftReference 的构造方法，将需要用弱引用来指向的对象包装起来。当需要使用的时候，调用 SoftReference 的 get() 方法来获取。当对象未被回收时 SoftReference 的 get() 方法会返回该对象的强引用。如下：

```
SoftReference<Bean> sr = new SoftReference<Bean>(new Bean("name", 10));
```

```
System.out.println(sr.get());  // “name:10”
```

软引用有以下特征：

1. **软引用使用 get() 方法取得对象的强引用从而访问目标对象。**
2. **软引用所指向的对象按照 JVM 的使用情况（Heap 内存是否临近阈值）来决定是否回收。**
3. **软引用可以避免 Heap 内存不足所导致的异常。**

当垃圾回收器决定对其回收时，会先清空它的 SoftReference，也就是说 SoftReference 的 get() 方法将会返回 null，然后再调用对象的 finalize() 方法，并在下一轮 GC 中对其真正进行回收。

WeakReference： 弱引用

WeakReference 是弱于 SoftReference 的引用类型。弱引用的特性和基本与软引用相似，区别就在于弱引用所指向的对象只要进行系统垃圾回收，不管内存使用情况如何，永远对其进行回收（get() 方法返回 null）。

完全可以通过和 SoftReference 一样的方式来操作 WeakReference，这里就不再复述。
弱引用有以下特征：

1. **弱引用使用 get() 方法取得对象的强引用从而访问目标对象。**
2. **一旦系统内存回收，无论内存是否紧张，弱引用指向的对象都会被回收。**
3. **弱引用也可以避免 Heap 内存不足所导致的异常。**

PhantomReference： 虚引用

PhantomReference 是所有“弱引用”中最弱的引用类型。不同于软引用和弱引用，虚引用无法通过 get() 方法来取得目标对象的强引用从而使用目标对象，观察源码可以发现 get() 被重写为永远返回 null。

那虚引用到底有什么作用？**其实虚引用主要被用来跟踪对象被垃圾回收的状态，通过查看引用队列中是否包含对象所对应的虚引用来判断它是否即将被垃圾回收，从而采取行动。**它并不被期待用来取得目标对象的引用，而目标对象被回收前，它的引用会被放入一个 ReferenceQueue 对象中，从而达到跟踪对象垃圾回收的作用。

所以具体用法和之前两个有所不同，**它必须传入一个 ReferenceQueue 对象。当虚引用所引用对象被垃圾回收后，虚引用会被添加到这个队列中。**如：

```
public static void main(String[] args) {
      ReferenceQueue<String> refQueue = new ReferenceQueue<String>();
      PhantomReference<String> referent = new PhantomReference<String>(new
      String("T"), refQueue);
      System.out.println(referent.get());// null
      System.gc();
      System.runFinalization();
      System.out.println(refQueue.poll() == referent); //true
}
```

虚引用有以下特征：

1. 虚引用永远无法使用 get() 方法取得对象的强引用从而访问目标对象。
2. 虚引用所指向的对象在被系统内存回收前，虚引用自身会被放入 ReferenceQueue 对象中从而跟踪对象垃圾回收。
3. 虚引用不会根据内存情况自动回收目标对象。

另外值得注意的是，其实 **SoftReference, WeakReference 以及 PhantomReference 的构造函数都可以接收一个 ReferenceQueue 对象。**

— 当 SoftReference 以及 WeakReference 被清空的同时，也就是 Java 垃圾回收器准备对它们所指向的对象进行回收时，调用对象的 finalize() 方法之前，它们自身会被加入到这个 ReferenceQueue 对象中，此时可以通过 ReferenceQueue 的 poll() 方法取到它们。

— 而 PhantomReference 只有当 Java 垃圾回收器对其所指向的对象真正进行回收时，会将其加入到这个 ReferenceQueue 对象中，这样就可以追综对象的销毁情况。

```
// Demonstrates Reference objects
import java.lang.ref.*;
import java.util.*;

class VeryBig {
    private static final int SIZE = 10000;
    private long[] la = new long[SIZE];
    private String ident;

    public VeryBig(String id) {
        ident = id;
    }
```

```
    public String toString() {
        return ident;
    }
    protected void finalize() {
        System.out.println("Finalizing " + ident);
    }
}

public class References {
    private static ReferenceQueue<VeryBig> rq = new ReferenceQueue<VeryBig>();

    public static Reference<? extends VeryBig> checkQueue() {
        Reference<? extends VeryBig> inq = rq.poll();
        System.out.println("inq : =" + inq);
        if (inq != null)
            System.out.println("In queue: " + inq.get());
        return inq;
    }

    public static void main(String[] args) throws InterruptedException {
        int size = 10;
        // Or, choose size via the command line:
        if (args.length > 0)
            size = new Integer(args[0]);
        /* 下面这个循环中的对象被包装成SoftReference，而且程序中没有这些对象的强引用
         * 那么在JVM还没有out of memory的时候就不会回收，这些VeryBig对象，也就不会执行finalize()方法。
         */
        LinkedList<SoftReference<VeryBig>> sa = new LinkedList<SoftReference<VeryBig>>();
        for (int i = 0; i < size; i++) {
            sa.add(new SoftReference<VeryBig>(new VeryBig("Soft " + i), rq));
            System.out.println("Just created: " + sa.getLast());
            checkQueue();
        }
       /*下面这些对象被包装成WeakReference，在没有强引用的时候，gc会将它们都进行标记
        * 表示可以被回收，在被回收之前，对象会执行finalize()方法。
        */
        LinkedList<WeakReference<VeryBig>> wa = new LinkedList<WeakReference<VeryBig>>();
        for (int i = 0; i < size; i++) {
            wa.add(new WeakReference<VeryBig>(new VeryBig("Weak " + i), rq));
            System.out.println("Just created: " + wa.getLast());
            checkQueue();
        }
        SoftReference<VeryBig> s = new SoftReference<VeryBig>(new VeryBig("Soft"));
```

```
    WeakReference<VeryBig> w = new WeakReference<VeryBig>(new VeryBig("Weak"));
    PhantomReference<VeryBig> z = new PhantomReference<VeryBig>(new VeryBig("Weak"), rq);

    LinkedList<PhantomReference<VeryBig>> pa = new
LinkedList<PhantomReference<VeryBig>>();
    for (int i = 0; i < size; i++) {
        pa.add(new PhantomReference<VeryBig>(new VeryBig("Phantom " + i),rq));
        System.out.println("Just created: " + pa.getLast());
        checkQueue();
    }
 }
} /* (Execute to see output) */
```