内容：强引用与弱引用

在Java里, 当一个对象o被创建时, 它被放在Heap里. 当GC运行的时候, 如果发现没有任何引用指向o, o就会被回收以腾出内存空间. 或者换句话说, 一个对象被回收, 必须满足两个条件: 1)没有任何引用指向它 2)GC被运行.

在现实情况写代码的时候, 我们往往通过把所有指向某个对象的referece置空来保证这个对象在下次GC运行的时候被回收 (可以用java -verbose:gc来观察gc的行为)

Object c = **new** Car();

c=**null**;

但是, 手动置空对象对于程序员来说, 是一件繁琐且违背自动回收的理念的.

对于简单的情况, 手动置空是不需要程序员来做的, 因为在java中, 对于简单对象, 当调用它的方法执行完毕后, 指向它的引用会被从stack中popup, 所以他就能在下一次GC执行时被回收了.

但是, 也有特殊例外. 当使用cache的时候, 由于cache的对象正是程序运行需要的, 那么只要程序正在运行, cache中的引用就不会被GC给(或者说, cache中的reference拥有了和主程序一样的life cycle). 那么随着cache中的reference越来越多, GC无法回收的object也越来越多, 无法被自动回收. 当这些object需要被回收时, 回收这些object的任务只有交给程序编写者了. 然而这却违背了GC的本质(自动回收可以回收的objects).

所以, java中引入了weak reference. 相对于前面举例中的strong reference:

Object c = **new** Car(); *//只要c还指向car object, car object就不会被回收*

 当一个对象仅仅被weak reference指向, 而没有任何其他strong reference指向的时候, 如果GC运行, 那么这个对象就会被回收. weak reference的语法是:

WeakReference<Car> weakCar = new WeakReference(**Car**)(**car**)*;*

 当要获得weak reference引用的object时, 首先需要判断它是否已经被回收:

**weakCar**.get();

 如果此方法为空, 那么说明weakCar指向的对象已经被回收了.

下面来看一个例子:

|  |
| --- |
| package weakreference;  /\*\*  \* @author wison  \*/  public class Car {  private double price;  private String colour;    public Car(double price, String colour){  this.price = price;  this.colour = colour;  }    public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  public String getColour() {  return colour;  }  public void setColour(String colour) {  this.colour = colour;  }    public String toString(){  return colour +"car costs $"+price;  }    } |

package weakreference;

import java.lang.**ref**.WeakReference;

*/\*\**

*\* @author wison*

*\*/***public** **class** **TestWeakReference** {

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

Car car = **new** Car(22000,"silver");

WeakReference<Car> weakCar = **new** WeakReference<Car>(car);

**int** i=0;

**while**(true){

**if**(weakCar.**get**()!=null){

i++;

System.**out**.println("Object is alive for "+i+" loops - "+weakCar);

}**else**{

System.**out**.println("Object has been collected.");

**break**;

}

}

}

}

在上例中, 程序运行一段时间后, 程序打印出"Object has been collected." 说明, weak reference指向的对象的被回收了.

值得注意的一点 , 即使有 *car* 引用指向对象, 且 *car* 是一个strong reference, weak reference weakCar指向的对象仍然被回收了. 这是因为java的编译器在发现进入while循环之后, *car* 已经没有被使用了, 所以进行了优化(将其置空?). 当把TestWeakReference.java修改为:

package weakreference;

import java.lang.**ref**.WeakReference;

*/\*\**

*\* @author wison*

*\*/***public** **class** **TestWeakReference** {

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

Car car = **new** Car(22000,"silver");

WeakReference<Car> weakCar = **new** WeakReference<Car>(car);

**int** i=0;

**while**(true){

System.**out**.println("here is the strong reference 'car' "+car);

**if**(weakCar.**get**()!=null){

i++;

System.**out**.println("Object is alive for "+i+" loops - "+weakCar);

}**else**{

System.**out**.println("Object has been collected.");

**break**;

}

}

}

}

weak reference指向的object就不会被回收了. 因为还有一个strong reference *car* 指向它.

\* WeakReference的一个特点是它何时被回收是不可确定的, 因为这是由GC运行的不确定性所确定的. 所以, 一般用weak reference引用的对象是有价值被cache, 而且很容易被重新被构建, 且很消耗内存的对象.

## **ReferenceQueue**

在weak reference指向的对象被回收后, weak reference本身其实也就没有用了. java提供了一个ReferenceQueue来保存这些所指向的对象已经被回收的reference. 用法是在定义WeakReference的时候将一个ReferenceQueue的对象作为参数传入构造函数.

## **其他类型的references**

-SoftReference

soft reference和weak reference一样, 但被GC回收的时候需要多一个条件: 当系统内存不足时(GC是如何判定系统内存不足? 是否有参数可以配置这个threshold?), soft reference指向的object才会被回收. 正因为有这个特性, soft reference比weak reference更加适合做cache objects的reference. 因为它可以尽可能的retain cached objects, 减少重建他们所需的时间和消耗.

java语言中为对象的引用分为了四个级别，分别为 强引用 、软引用、弱引用、虚引用。

本文只针对java中的弱引用进行一些分析，如有出入还请多指正。

在分析弱引用之前，先阐述一个概念：什么是对象可到达和对象不可到达状态。

其实很简单，我举个例子：

现在有如下两个类class A class B，在JVM上生成他们两个类的实例分别为 instance a  instance b

有如下表达式:

A a = new A();

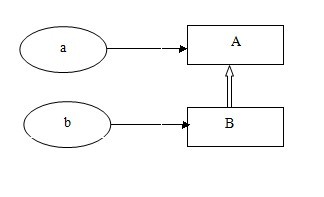
B b = new B();

两个强引用对象就生成了，好吧，那么这个时候我做一下修改：

A a = new A();

B b = new B(a);

B的默认构造函数上是需要一个A的实例作为参数的，那么这个时候 A和B就产生了依赖，也可以说a和b产生了依赖，我们再用一个接近内存结构的图来表达：



a是对象A的引用，b是对象B的引用，对象B同时还依赖对象A，那么这个时候我们认为从对象B是可以到达对象A的。

于是我又修改了一下代码

A a = new A();

B b = new B(a);

a = null;

A对象的引用a置空了，a不再指向对象A的地址，我们都知道当一个对象不再被其他对象引用的时候，是会被GC回收的，很显然及时a=null，那么A对象也是不可能被回收的，因为B依然依赖与A，在这个时候，造成了内存泄漏！

那么如何避免上面的例子中内存泄漏呢？

很简单：

A a = new A();

B b = new B(a);

a = null;

b = null;

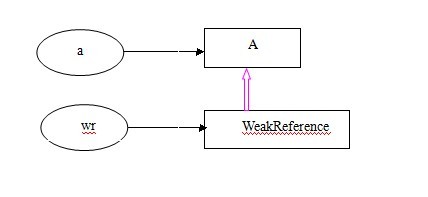
这个时候B对象再也没有被任何引用，A对象只被B对象引用，尽管这样，GC也是可以同时回收他们俩的，因为他们处于不可到达区域。

弱引用来了！

A a = new A();

WeakReference wr = new WeakReference(a);

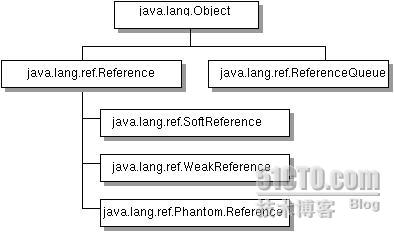
//B b = new B(a);



当 a=null ，这个时候A只被弱引用依赖，那么GC会立刻回收A这个对象，这就是弱引用的好处！他可以在你对对象结构和拓扑不是很清晰的情况下，帮助你合理的释放对象，造成不必要的内存泄漏！！

**1．对象的强、软、弱和虚引用**

在JDK 1.2以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及（reachable）状态，程序才能使用它。从JDK 1.2版本开始，把对象的引用分为4种级别，从而使程序能更加灵活地控制对象的生命周期。这4种级别由高到低依次为：强引用、软引用、弱引用和虚引用。图1为对象应用类层次。



⑴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。

⑵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（下文给出示例）。

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。

⑶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。

问题：引用队列（ReferenceQueue）引用队列的使用

软引用与弱引用的区别，怎么选择

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。

|  |
| --- |
| ReferenceQueue queue = **new** ReferenceQueue ();  PhantomReference pr = **new** PhantomReference (object, queue); |

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

//=============================================================================

//异步任务

private static class InitialTask extends AsyncTask<Void, Void, String> {

/引用，防止内存泄漏

private WeakReference<SplashActvt> \_host;/引用宿主host外部类

//构造函数，传入外部类宿主host

public InitialTask(SplashActvt h){

\_host = new WeakReference<SplashActvt>(h);

}

//

@Override

protected String doInBackground(Void... params) {

String result = null;

long startTime = System.currentTimeMillis();

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

//do something

//

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

//延时

AsyncTaskExt.sleep(this,SHOW\_MIN\_TIME\_MS-(System.currentTimeMillis() - startTime));

//返回

return result;

}

@Override

protected void onPostExecute(String result) {

SplashActvt t = \_host.get();

if(t != null){

t.startActivity(); //跳转页面

}

}

}