其工作原理：**一旦垃圾回收器准备好释放对象占用的存储空间，将首先调用其finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。所以如果用finalize()就能在垃圾回收时刻做一些重要的清理工作。**

**1. finalize的作用**

* **finalize()是Object的protected方法，子类可以覆盖该方法以实现资源清理工作，GC在回收对象之前调用该方法。**
* finalize()与C++中的析构函数不是对应的。C++中的析构函数调用的时机是确定的（对象离开作用域或delete掉），但**Java中的finalize的调用具有不确定性**
* 不建议用finalize方法完成“非内存资源”的清理工作，但建议用于：① 清理本地对象(通过JNI创建的对象)；② 作为确保某些非内存资源(如Socket、文件等)释放的一个补充：在finalize方法中显式调用其他资源释放方法。其原因可见下文[finalize的问题]

**2. finalize的问题**

* 一些与finalize相关的方法，由于一些致命的缺陷，已经被废弃了，如System.runFinalizersOnExit()方法、Runtime.runFinalizersOnExit()方法
* System.gc()与System.runFinalization()方法**增加了finalize方法执行的机会，但不可盲目依赖它们**
* Java语言规范并不保证finalize方法会被及时地执行、而且根本不会保证它们会被执行
* finalize方法可能会带来性能问题。因为JVM通常在**单独的低优先级线程中完成finalize的执行**
* 对象再生问题：**finalize方法中，可将待回收对象赋值给GC Roots可达的对象引用，从而达到对象再生的目的**
* **finalize方法至多由GC执行一次**(用户当然可以手动调用对象的finalize方法，但并不影响GC对finalize的行为)

**3. finalize的执行过程(生命周期)**

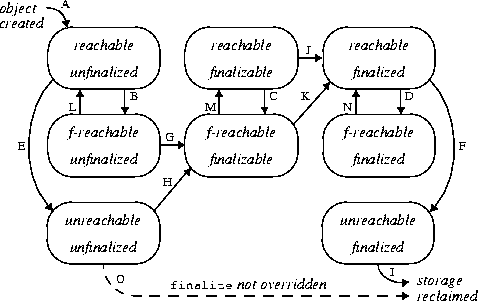
(1) 首先，大致描述一下finalize流程：当**对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖，则直接将其回收。否则，若对象未执行过finalize方法，将其放入F-Queue队列，由一低优先级线程执行该队列中对象的finalize方法。执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”。**

(2) 具体的finalize流程：

对象可由两种状态，涉及到两类状态空间，**一是终结状态空间 F** = {unfinalized, finalizable, finalized}；**二是可达状态空间 R** = {reachable, finalizer-reachable, unreachable}。各状态含义如下：

* unfinalized: **新建对象会先进入此状态，GC并未准备执行其finalize方法，因为该对象是可达的**
* finalizable: **表示GC可对该对象执行finalize方法，GC已检测到该对象不可达。正如前面所述，GC通过F-Queue队列和一专用线程完成finalize的执行**
* finalized: **表示GC已经对该对象执行过finalize方法**
* reachable: **表示GC Roots引用可达**
* finalizer-reachable(f-reachable)：**表示不是reachable，但可通过某个finalizable对象可达**
* unreachable：**对象不可通过上面两种途径可达**

状态变迁图：



变迁说明：

1. 新建对象首先处于[reachable, unfinalized]状态(A)
2. 随着程序的运行，一些引用关系会消失，导致状态变迁，从reachable状态变迁到f-reachable(B, C, D)**(通过某个finalizable对象可达)**或unreachable(E, F) **(不可达)**状态
3. 若JVM检测到处于unfinalized状态的对象变成f-reachable或unreachable，JVM会将其标记为**finalizable状态(G,H)**。若对象原处于[unreachable, unfinalized]状态，则同时将其标记为**f-reachable(H)（塞到F-Queue队列里等待一个线程去调用其**finalize方法**）。**
4. 在某个时刻，JVM取出某个finalizable对象，将其标记为finalized并在某个线程中执行其**finalize方法**。由于是在**活动线程中引用了该对象，该对象将变迁到(reachable, finalized)状态(K或J)。该动作将影响某些其他对象从f-reachable状态重新回到reachable状态(L, M, N)**
5. 处于finalizable状态的对象不能同时是unreahable的，由第4点可知，将对象finalizable对象标记为finalized时会由某个线程执行该对象的finalize方法，致使其变成reachable。这也是图中只有八个状态点的原因
6. 程序员手动调用finalize方法并不会影响到上述内部标记的变化，因此JVM只会至多调用finalize一次，即使该对象“复活”也是如此。程序员手动调用多少次不影响JVM的行为
7. 若JVM检测到finalized状态的对象变成unreachable，回收其内存(I)
8. 若对象并未覆盖finalize方法，JVM会进行优化，直接回收对象（O）
9. 注：System.runFinalizersOnExit()等方法可以使对象即使处于reachable状态，JVM仍对其执行finalize方法

**4. 一些代码示例**

**(1) 对象复活**

1. **public class GC {**
3. **public static GC SAVE\_HOOK = null;**
5. **public static void main(String[] args) throws InterruptedException {**
6. SAVE\_HOOK = **new GC();**
7. SAVE\_HOOK = **null;**
8. System.gc();
9. Thread.sleep(500);
10. **if (null != SAVE\_HOOK) { //此时对象应该处于(reachable, finalized)状态**
11. System.out.println("Yes , I am still alive");
12. } **else {**
13. System.out.println("No , I am dead");
14. }
15. SAVE\_HOOK = **null;**
16. System.gc();
17. Thread.sleep(500);
18. **if (null != SAVE\_HOOK) {**
19. System.out.println("Yes , I am still alive");
20. } **else {**
21. System.out.println("No , I am dead");
22. }
23. }
25. @Override
26. **protected void finalize() throws Throwable {**
27. **super.finalize();**
28. System.out.println("execute method finalize()");
29. SAVE\_HOOK = **this;**
30. }
31. }