概述

存在与内存中，但是没有被使用的对象。这些对象如果不清除，就会一直保留到程序结束。如果这些对象占据着内存空间，而其他对象需要那么就会出现内存溢出，垃圾回收就是要清理这些没有被使用的对象，释放内存空间。

在c/c++等语言中对象的释放是通过程序员手动释放的。
在Java语言中对象的释放是由垃圾回收机制自动的释放的，不需要程序员来参与。

垃圾回收有三个需要完成的过程：

- 哪些对象需要回收
- 什么时候回收
- 怎么回收

虽然Java虚拟机中垃圾回收是自动化的处理，但是有些时候一样会出现内存溢出，或者垃圾回收成为系统瓶颈，这个时候就需要知道垃圾回
行调节来解决问题

如何判断对象是否需要被回收

引用计数算法

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值加1；当引用失效时，计数器减1；任何时刻计数器都为0的对象就是不可

引用计数算法(Reference Counting)的实现简单，判断效率也很高，在大部分情况下它都是一个不错的算法。但是Java语言中没有选用引用计
个原因是它很难解决对象之间相互循环引用的问题。

例如：在testGC()方法中，对象objA和objB都有字段instance，赋值令objA.instance=objB及objB.instance=objA，除此之外这两个对象再无
不能再被访问，但是它们因为相互引用着对象方，异常它们的引用计数都不为0，于是引用计数算法无法通知GC收集器回收它们

根搜索算法

在主流的商用程序语言中(Java和C#)，都是使用根搜索算法(GC Roots Tracing)判断对象是否存活的。这个算法的基本思路就是通过一系列名
这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是

在Java语言里，可作为GC Roots对象的包括如下几种：

- 虚拟机栈(栈桢中的本地变量表)中的引用的对象
- 方法区中的类静态属性引用的对象
- 方法区中的常量引用的对象
- 本地方法栈中JNI的引用的对象

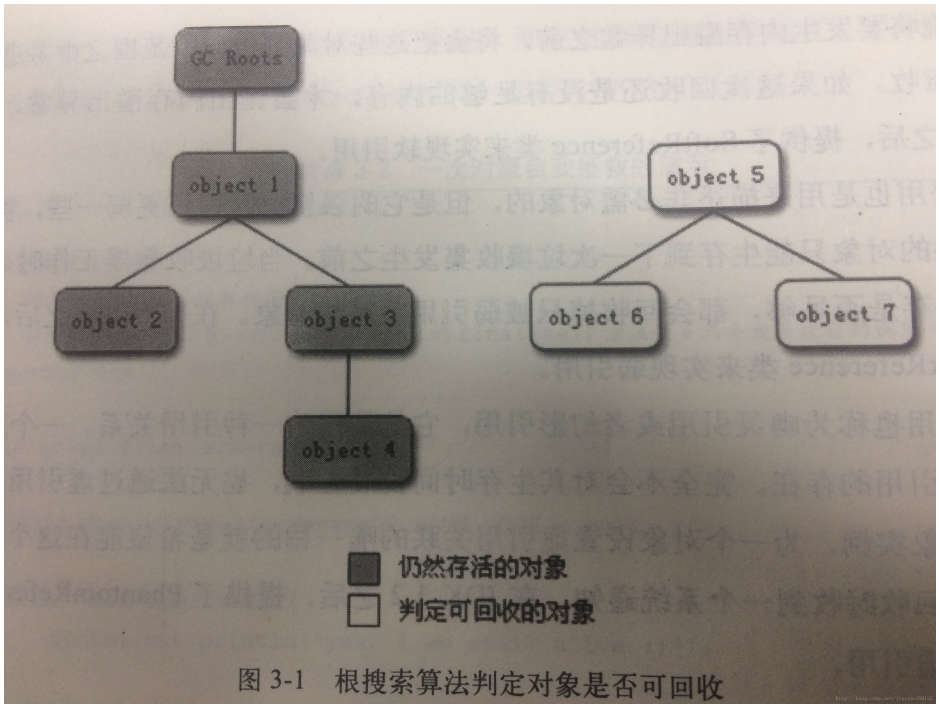


图 3-1　根搜索算法判定对象是否可回收

真正的垃圾：判断可触及性

一般来说如果从根节点无法访问到对象了，那就说么这个对象可以被回收了。但是在特定的情况下，这个对象有可能“复活”自己。那么这样

可触及性包含三种状态：

可触及的：从根节点开始，可以达到这个对象。

可复活的：对象的所有引用都被释放，但是可能在finalize()函数中复活。
不可触及的：对象在finalize()函数被调用，并且没有复活，那么就会进入不可触及状态。

以上三种状态：只有在不可触及的时候才能进行回收

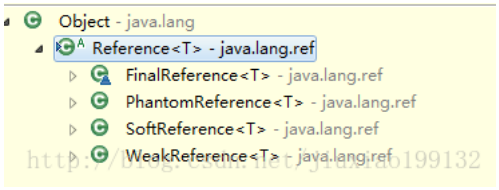
举例说明

```
public class CanReliveObj {
    public static CanReliveObj obj;

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("canreliveObj finalize called");
        obj = this;
    }

     @Override
     public String toString() {
         return "I an CanReliveObj";
     }

     public static void main(String[] args) throws InterruptedException {
         obj = new CanReliveObj();
         obj = null;
         System.out.println("第一次GC");
         System.gc();
         Thread.sleep(1000);
         if(obj == null){
             System.out.println("obj is null");
         }else{
             System.out.println("obj 可用");
         }
         System.out.println("第二次GC");
         obj = null;
         System.gc();
         Thread.sleep(1000);
         if(obj == null){
             System.out.println("obj is null");
         }else{
             System.out.println("obj 可用");
         }
     }
}

/*输出
    第一次GC
    canreliveObj finalize called
    obj 可用
    第二次GC
    obj is null
*/

/*说明
```

```
48 第一次gc的时候在finalize中复活了自己，所以obj没有被回收。
49 第二次gc的时候由于finalize只调用一次，所以obj被回收
50 */
```

引用和可触及性的强度

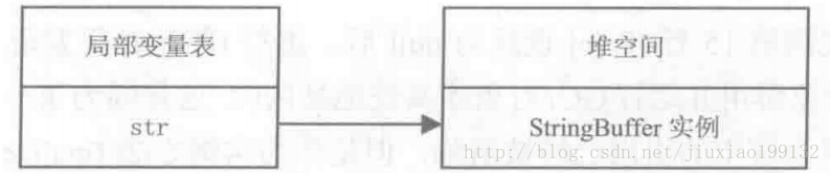
Java中提供了四个级别的引用：强引用，软引用，弱引用，虚引用，除了强引用之外其他三种都可用在Java.lang.ref包中看到。如图



强引用

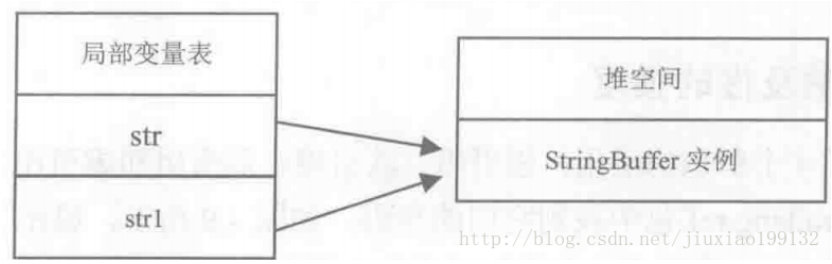
强引用是可触及的不会被回收。假设下面的代码片段在方法内。

```
1 StringBuffer str = new StringBuffer("hello world");
```



str分配在栈空间，Stringbuffer被分配到堆空间。str就是StringBuffer的强引用。
如果继续执行一段代码

```
1 str1 = str;
```



任何时候强引用都不会被回收，虚拟机宁愿抛出OutOfMemoryError异常也不会回收。

强引用可以直接操作对象。

强引用不会被回收，所以可能导致内存

软引用

软引用是被强类型弱一点的引用，当堆内存不足的时候，就会被回收。通过java.lang.ref.SoftReference实现

```
1 import java.lang.ref.SoftReference;
2
3 public class SoftReferenceTest {
4     //-Xmx15M 启动
5     public static void main(String[] args) {
6         User u = new User(1, "张三");
7         SoftReference<User> softRef = new SoftReference<User>(u);
8         u = null;
```

```
        System.out.println(softRef.get());
        System.gc();
        System.out.println("after gc");
        System.out.println(softRef.get());
        byte[] bt = new byte[8*1024*1024];
        System.gc();
        System.out.println(softRef.get());
    }

    static class User{
        private int id;
        private String name;
        private byte[] bt = new byte[5*1024*1024];
        public int getId() {
            return id;
        }
        public void setId(int id) {
            this.id = id;
        }
        public byte[] getBt() {
            return bt;
        }
        public void setBt(byte[] bt) {
            this.bt = bt;
        }
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public User(int id,String name){
            this.id = id;
            this.name = name;
        }
        @Override
        public String toString() {
            return "id : " + id + " , name : " + name;
        }

    }
}

/*输出
id : 1 , name : 张三
after gc
id : 1 , name : 张三
null
*/

/*说明
这里设置15M最大堆内存，实例化一个User的强引用，然后通过强引用建立一个软引用，去掉强引用。
设置一个内存为8M字节数组之后，总内存15M，字节数组8M，加上软引用中有5M，GC发现内存比较紧张，那么回收软引用。
*/
```

弱引用

弱引用是比软引用还要低一点的引用，垃圾回收机制发现就会回收它，但是通常垃圾回收机制线程优先级比较低，所以弱引用可以存活一段存活的。通过java.lang.ref.WeakReference使用。

虚引用

最弱的一种引用，和没有引用时一样的，通过虚引用的get获取强引用时总是失败。虚引用一般用于垃圾回收跟踪，通过java.lang.ref.Phant

垃圾回收算法

标记-清除算法

标记清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。

标记阶段：通过根节点（GC Roots）开始，区分对象是否可用回收进行标记。
清除阶段：标记完毕之后，统一进行回收

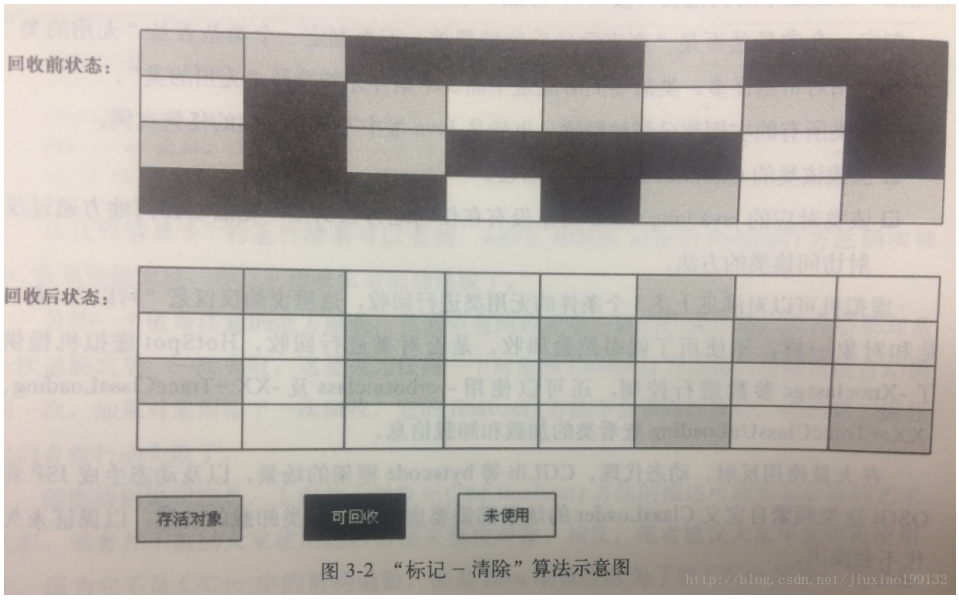


图 3-2 “标记－清除”算法示意图

如图所示：回收完成之后的内存空间是不连续的，当有大对象需要存储的时候，不连续的内存空间效率远远低于连续的内存空间，这是该算

复制算法

复制算法的是将原有的内存空间分成两块，每次只是用其中一块，在垃圾回收的时候，将正在使用的内存中存活的对象复制到未使用的内存所有对象，在交换两个内存的角色，完成垃圾回收

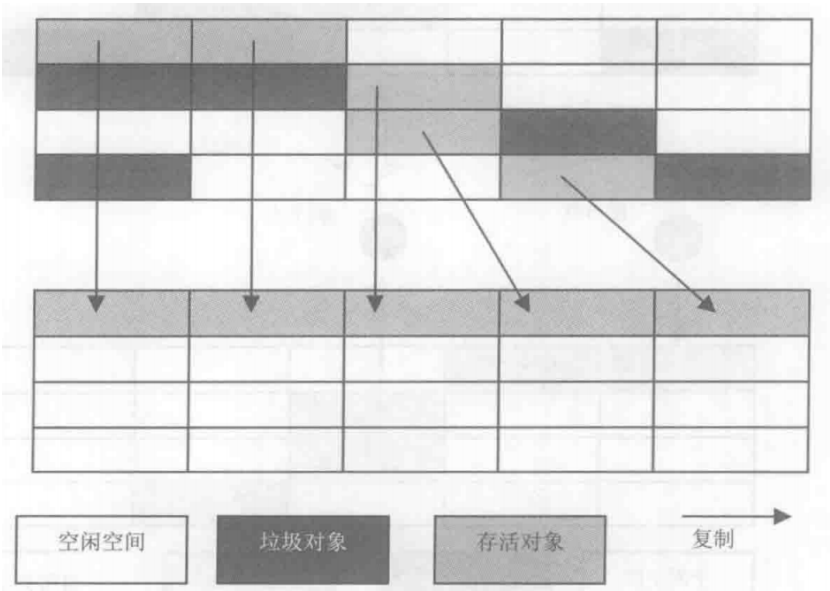
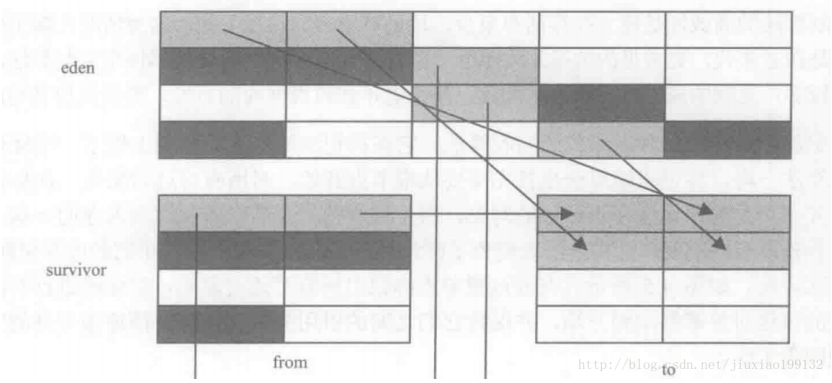


图 4.3 复制算法工作示意图

如图所示：将内存分为A和B两块，先使用A，在垃圾回收的时候把存活的对象复制到B，最后情况A内存。把B内存设置为使用中的。这种算
较少，这样复制比较快。然后进行清理效率很高。

在Java新生代串行化垃圾回收器中使用了复制的思想，新生代分为eden，from，to三个空间。其中from和to空间大小相等进行相互复制，因
象



标记压缩算法

复制算法适用于新生代，因为新生代比较多垃圾对象。
而标记压缩算法适用于老年代，标记压缩算法在标记清除算法的基础上做了一些优化，它不仅仅是清理了垃圾对象，还把存活的对象进行了

分代算法

将内存按照对象存活周期的不同，划分为几块。
新生代和老年代：新生代由于存活对象少，使用复制算法，老年代由于对象存活率高，无额外空间担保，使用“标记——清除”或“标记——整

分区算法

分代算法是按照对象的生命周期进行划分。
分区算法是把堆内存空间划分成多个连续的小空间，每一个小空间都是一个区域，每一个小区域单独回收。

堆内存空间越大，进行一次回收会消耗很长的时间。这样就会产生长时间的垃圾回收停顿。
而将堆内存空间分成多个小区域，每次回收不同的小区域，就会减少一次垃圾回收停顿

垃圾回收停顿现象–stop the world

垃圾回收的任务是识别和回收垃圾对象。为了让垃圾回收正确并且高效的执行，大部分情况下会要求程序进入一个停顿的状态。停顿的状态
圾对象，保证程序的一致性。这个停顿期间程序会卡死，无响应。只允许gc线程执行（标记、清除等），直到gc完毕才会恢复。这个停顿就

代码

```
import java.util.HashMap;
import java.util.Map;

/**
 * vm args : -Xmx4g -Xms4g -XX:+UseSerialGC -Xloggc:gc.log -XX:+PrintGCDetails
 */
public class StopTheWorld {

    static class PutThread implements Runnable {
        private Map<Long, Object> map = new HashMap<>();

        @Override
        public void run() {
            try {
                while (true){
                    if (map.size() >= 7000 * 1024){     // 防止内存溢出，设置的8G内存 下面的for循环一次1M
                        map.clear();
                        System.out.println("map clear");
                    }
                    for (int i = 0; i < 1024; i++) {    // 一次循环下来 占用 1M
                        map.put(System.nanoTime(), new byte[1024]); // 1KB
                    }
                    Thread.sleep(1);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }


    static class PrintThread implements Runnable {

        private static final long stratTime = System.currentTimeMillis();

        @Override
        public void run() {
            try {
                while (true){
                    long t = System.currentTimeMillis() - stratTime;
                    String v = String.valueOf(t%1000);
                    if (v.length() == 1){
                        v = "00" + v;
```

```
                    }else if (v.length() == 2){
                        v = "0" + v;
                    }
                    System.out.println(t/1000 + "." + v);
                    Thread.sleep(100);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {

        Thread putThread = new Thread(new PutThread());
        putThread.setName("putThread");
        Thread printThread = new Thread(new PrintThread());
        printThread.setName("printThread");
        putThread.start();
        printThread.start();

    }
}
```

输出

```
D:\jdk\bin\java.exe -Xmx8g -Xms8g -XX:+UseSerialGC -Xloggc:gc.log -XX:+PrintGCDetails -javaagent:D:\
0.001
0.101
0.202
0.303
0.403
0.504
0.605
0.706
 0.806
0.907
1.008
1.109
1.209
1.310
1.411
1.512
1.612
1.713
1.814
1.914
2.015
2.116
2.217
2.317
2.418
2.519
2.620
```

```
2.720
2.821
2.922
3.023
3.123
3.224
3.325
4.276
4.377
4.478
4.578
4.679
4.780
4.881
4.981
5.082
5.183
5.283
5.384
5.485
5.586
5.686
5.787
5.888
5.989
6.089
6.190
6.291
6.392
6.492
6.593
6.694
6.795
6.895
6.995
7.096
7.197
7.298
7.398
7.499
7.600
7.701
7.801
7.902
8.003
9.364
9.465
9.566
9.666
9.767
9.868
9.969
10.069
10.170
```

```
10.271
10.371
10.472
10.573
10.674
10.774
10.875
10.976
11.077
11.177
11.278
11.379
11.480
11.580
11.681
11.782
11.883
11.983
12.084
12.185
12.285
12.386
12.487
12.588
12.688
12.789
12.890
12.990
13.091
13.192
14.424
14.524
14.625
14.726
14.827
14.927
15.028
15.129
15.229
15.330
15.431
15.532
15.632
15.733
15.834
15.935
16.035
16.136
16.237
16.338
16.438
16.539
16.640
```

```
16.741
16.841
16.942
17.043
17.143
17.244
17.345
17.446
17.546
17.647
17.748
17.848
17.949
18.050
18.151
19.299
19.399
19.500
19.601
19.702
19.802
19.903
20.004
20.105
20.205
20.306
20.407
20.507
20.614
20.716
20.817
20.917
21.018
21.119
21.220
21.320
21.421
21.522
21.622
21.723
21.824
21.925
22.025
22.126
22.227
22.328
22.428
22.529
22.630
22.731
22.831
22.932
25.821
25.922
```

```
26.023
26.124
26.224
26.325
26.426
26.527
26.627
26.728
26.829
26.929
27.030
27.131
27.232
27.332
27.433
27.534
27.635
27.735
27.836
27.937
28.038
30.837
map clear
30.938
31.039
31.139
31.240
31.341
32.096
32.196
32.297
32.398
32.499
32.599
32.700
32.801
32.901
33.002
33.103
33.204
33.304
33.405
33.506
33.607
33.708
33.808
33.909
34.010
34.110
34.211
34.312
34.413
34.513
```

```
34.614
34.715
34.816
34.916
35.017
35.118
35.219
35.319
35.420
35.521
35.622
35.722
36.326
36.426
36.527
36.628
36.729
36.829
36.930
37.031
37.131
37.232
37.333
37.434
37.534
37.635
37.736
37.837
37.937
38.038
38.139
38.240
```

gc日志

```
Java HotSpot(TM) 64-Bit Server VM (25.181-b13) for windows-amd64 JRE (1.8.0_181-b13), built on Jul
Memory: 4k page, physical 16667856k(11312548k free), swap 19158224k(11781788k free)
CommandLine flags: -XX:InitialHeapSize=8589934592 -XX:MaxHeapSize=8589934592 -XX:+PrintGC -XX:+Print
3.498: [GC (Allocation Failure) 3.498: [DefNew: 2236928K->279616K(2516544K), 0.8836646 secs] 223692
8.144: [GC (Allocation Failure) 8.144: [DefNew: 2516544K->279615K(2516544K), 1.3247999 secs] 348828
13.298: [GC (Allocation Failure) 13.298: [DefNew: 2516543K->279615K(2516544K), 1.2303646 secs] 5015
18.269: [GC (Allocation Failure) 18.269: [DefNew: 2516543K->279616K(2516544K), 1.1351922 secs] 6417
23.084: [GC (Allocation Failure) 23.084: [DefNew: 2510103K->2510103K(2516544K), 0.0000146 secs]23.08
28.197: [Full GC (Allocation Failure) 28.197: [Tenured: 5592447K->5592447K(5592448K), 2.7452655 secs
31.495: [Full GC (Allocation Failure) 31.495: [Tenured: 5592447K->343457K(5592448K), 0.7060099 secs
35.903: [GC (Allocation Failure) 3 5.903: [DefNew: 2236928K->279615K(2516544K), 0.5274803 secs] 258
```

说明

正常情况下 打印线程会 每零点一秒 就打印一次控制台。但是发现实际上会有停顿。 第一次停顿的时间大约在 3.325 秒，接着 后面的3.498秒 发生了一次 gc，持续时间 0.8秒左右，这说明 打印线程也受到了 GC的影响，停止了工作，这就是 Stop-The-world

另外观察gc日志，在28秒多的时候 gc 时间是 2秒多。 但是 31秒开始 时间竟然下降了， 再看控制台，这个时候 map被清空了。

危害

- 长时间服务停止，没有响应
- 遇到HA系统，可能引起主备切换，严重危害生产环境。
- 新生代的gc时间比较短（），危害小。
- 老年代的gc有时候时间短，但是有时候比较长几秒甚至100秒--几十分钟都有。
- 堆越大花的时间越长