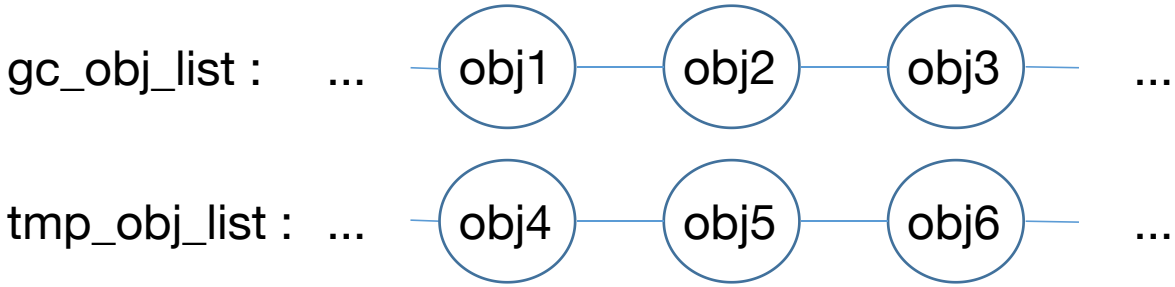
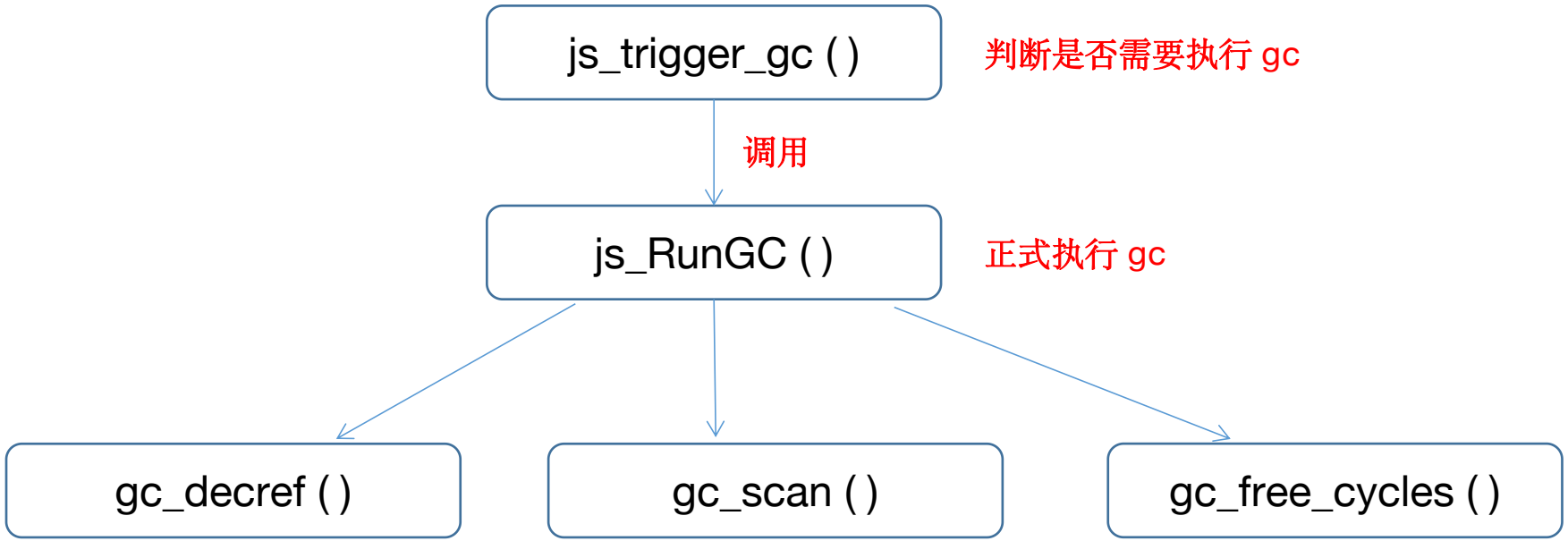
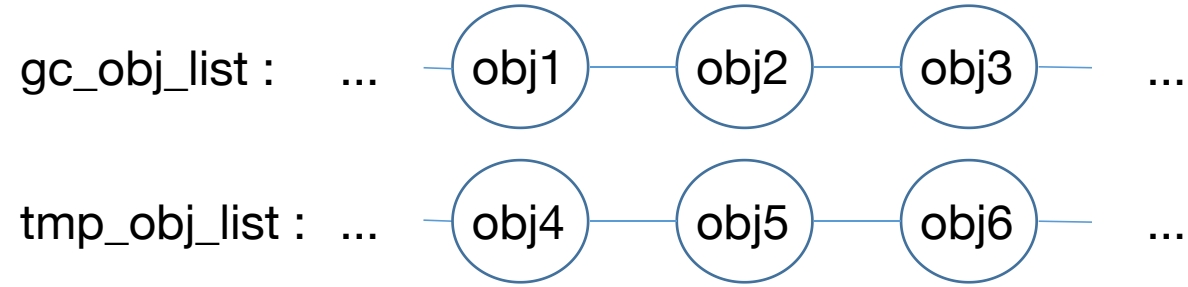
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}

JSGCObjectHeader {
    ...
    mark;
    ref_count;
}

JSObject {
    ...
    JSGCObjectHeader *p;
}
```




js_trigger_gc ( )
判断是否需要执行 gc
调用
js_RunGC ( )
正式执行 gc
gc_decref ( )
gc_scan ( )
gc_free_cycles ( )
遍历 gc_obj_list 中的对象，将每个对象的属性引用计数减 1，减完之后如果计数为 0，则存放到 tmp_obj_list 中
再次遍历 gc_obj_list 中的对象，将每个对象的属性引用计数加1，如果属性的引用计数等于1，则从tmp_obj_list 中移除，添加回 gc_obj_list
释放还存在于 tmp_obj_list 的对象

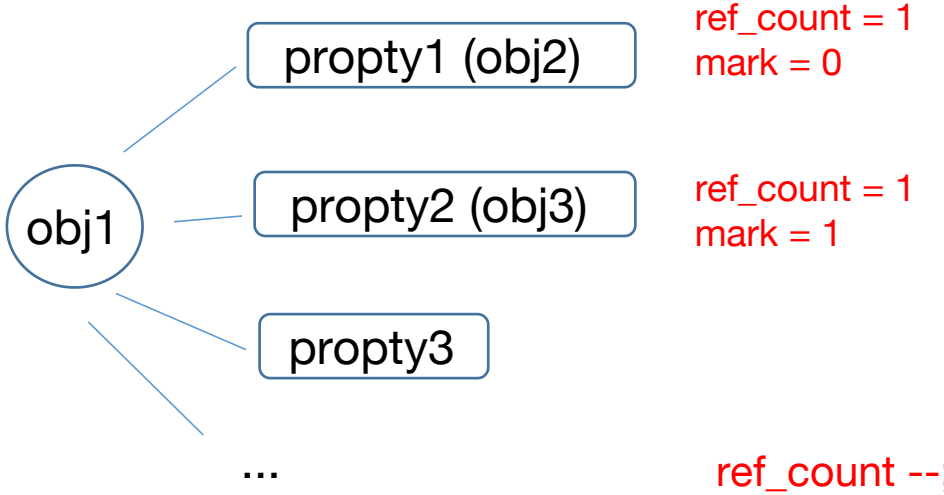
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

ref_count = 1
mark = 0

gc_decref :



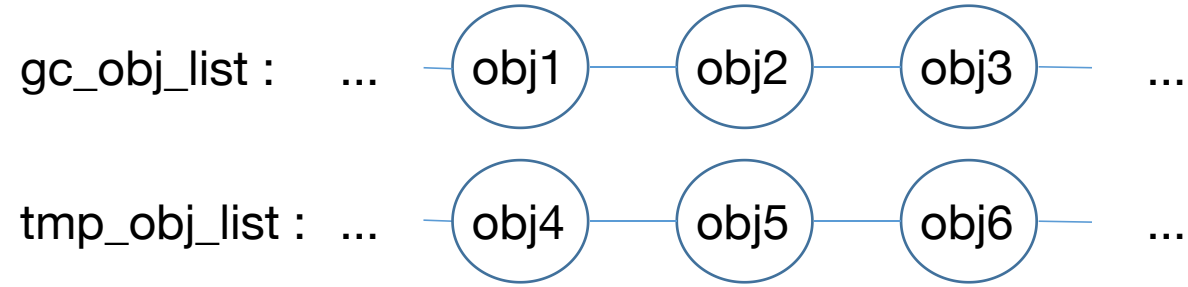
ref_count = 1
mark = 0

ref_count = 1
mark = 1

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

```
ref_count --;
if (ref_count == 1 && mark == 1)
    put propty into tmp_obj_list
```

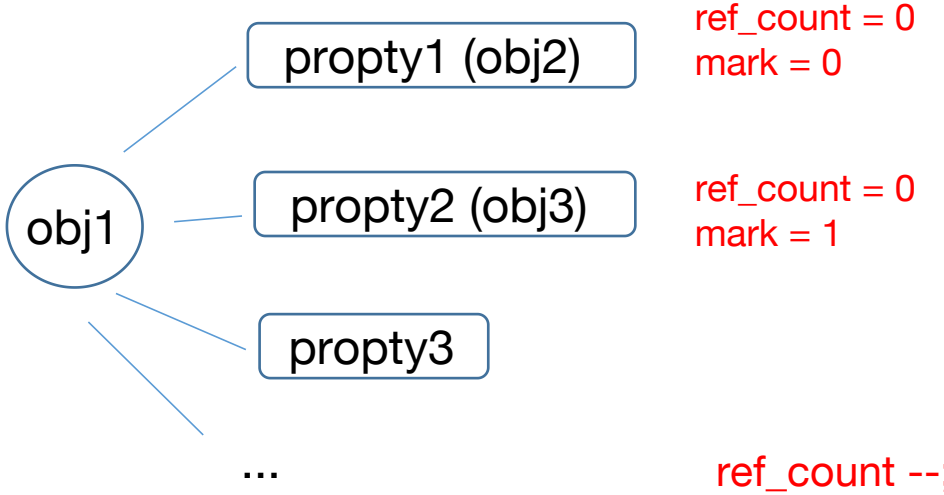
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

ref_count = 1
mark = 0

gc_decref :



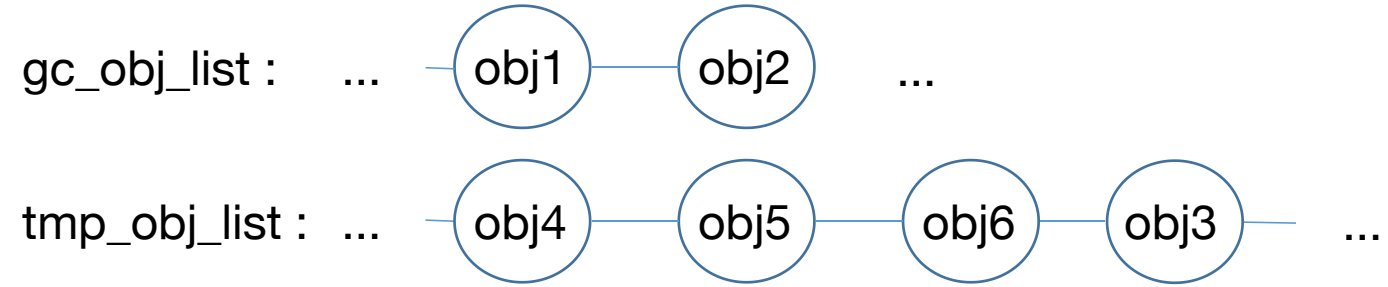
ref_count = 0
mark = 0

ref_count = 0
mark = 1

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

```
ref_count --;
if (ref_count == 1 && mark == 1)
    put propty into tmp_obj_list
```

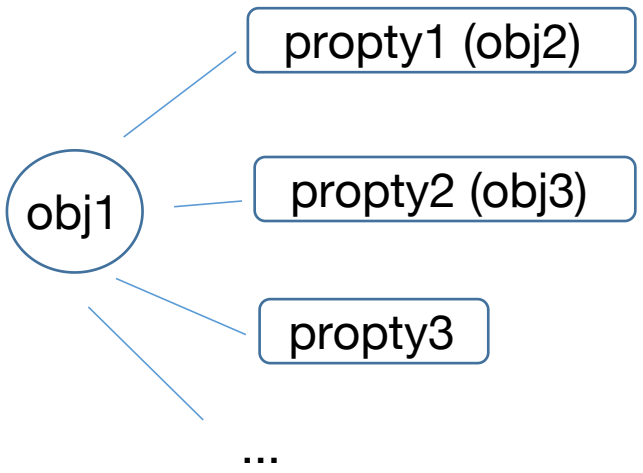
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

ref_count = 1
mark = 0

gc_decref :



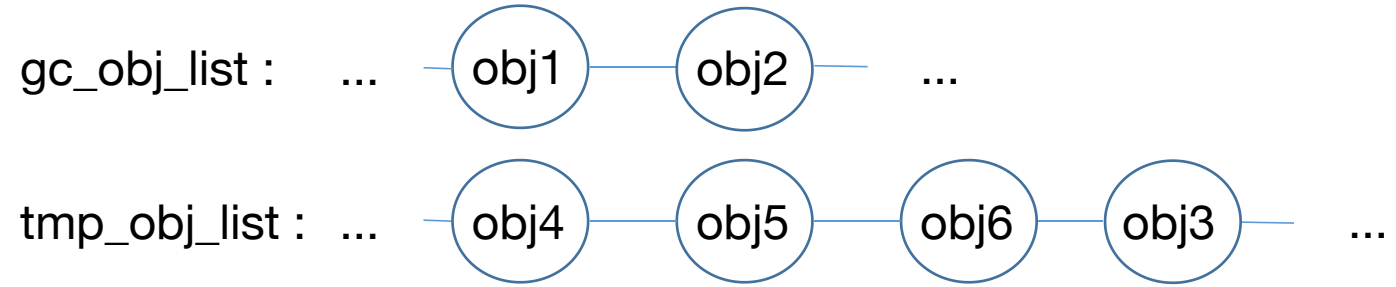
ref_count = 0
mark = 0

ref_count = 0
mark = 1

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

```
ref_count --;
if (ref_count == 1 && mark == 1)
    put propty into tmp_obj_list
```

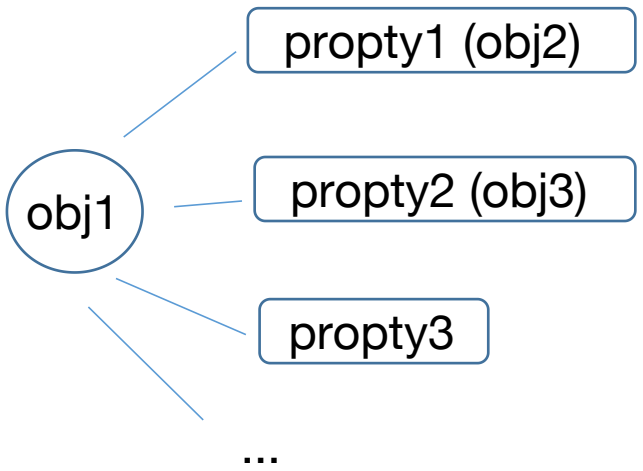
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

gc_decref :

ref_count = 1
mark = 1



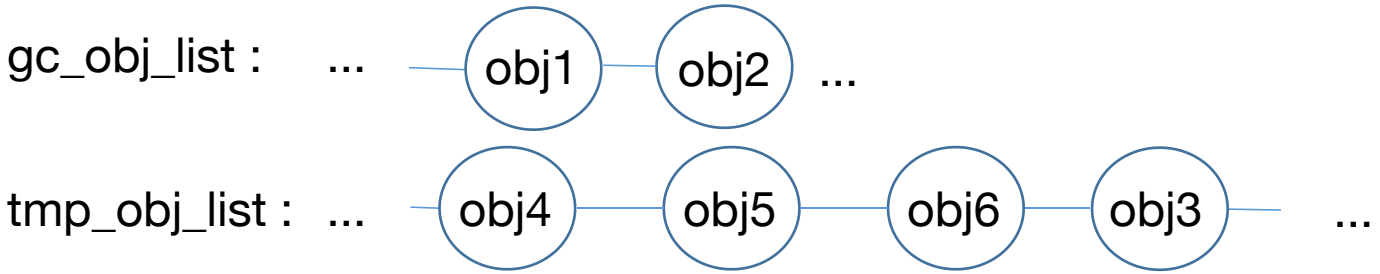
ref_count = 0
mark = 0

ref_count = 0
mark = 1

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

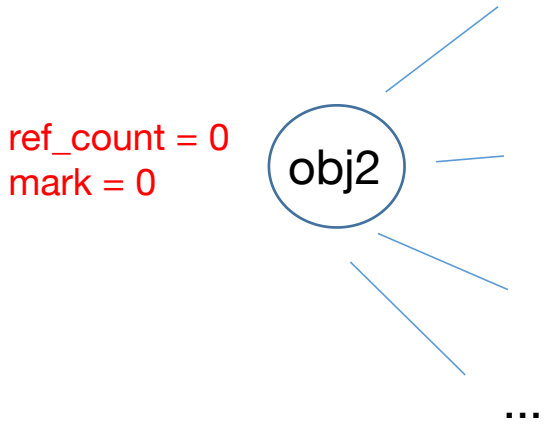
```
mark = 1;
if (ref_count == 0)
    put obj1 into tmp_obj_list
```

```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

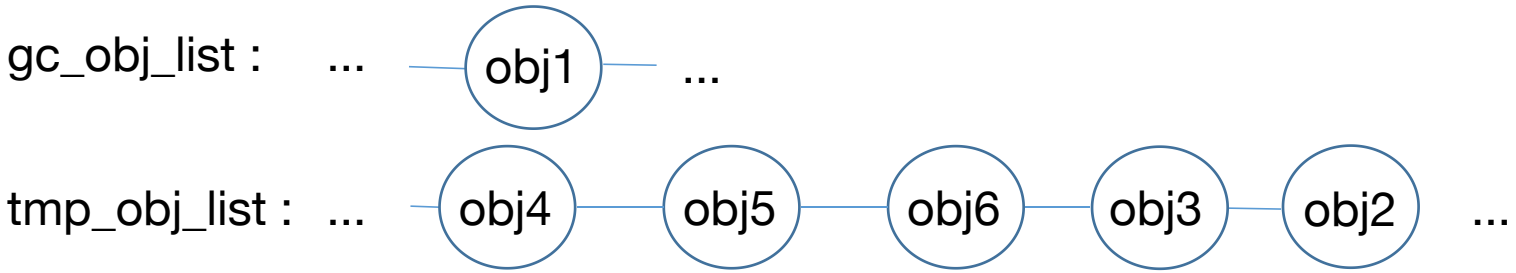
gc_decref :




```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

```
mark = 1;
if (ref_count == 0)
    put obj2 into tmp_obj_list
```

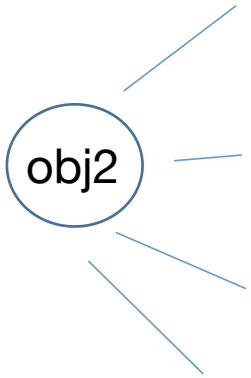
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

gc_decref :

ref_count = 0
mark = 1

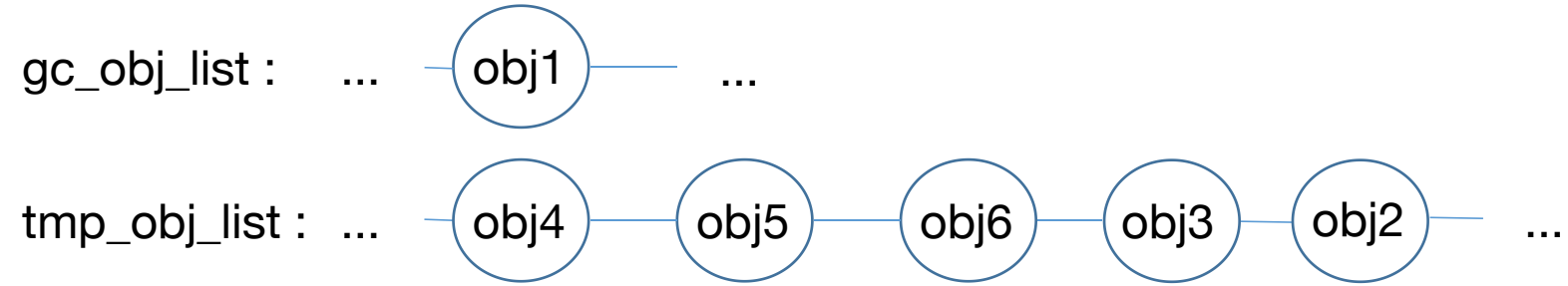


...

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

```
mark = 1;
if (ref_count == 0)
    put obj2 into tmp_obj_list
```

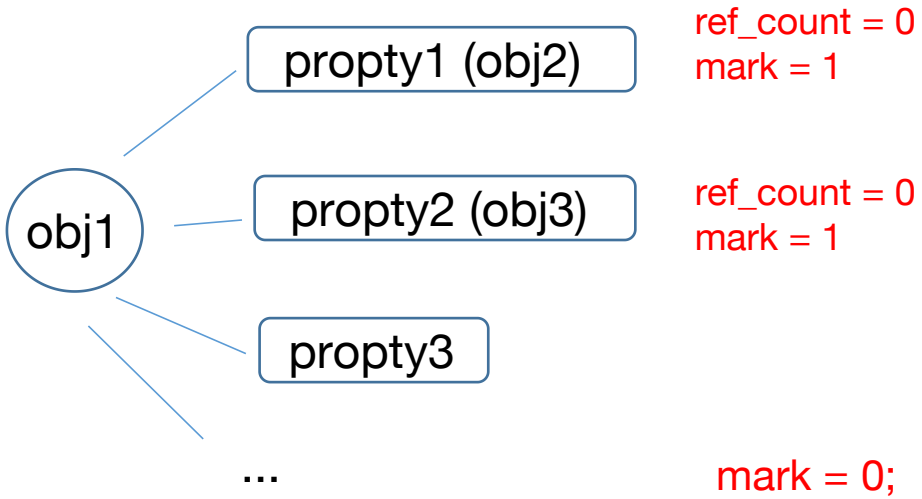
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

ref_count = 1
mark = 1

gc_scan :



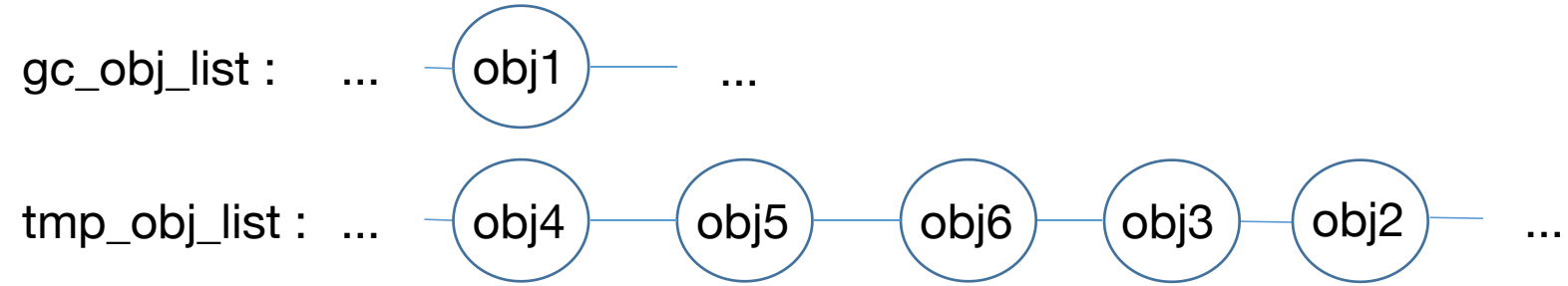
ref_count = 0
mark = 1

ref_count = 0
mark = 1

mark = 0;

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

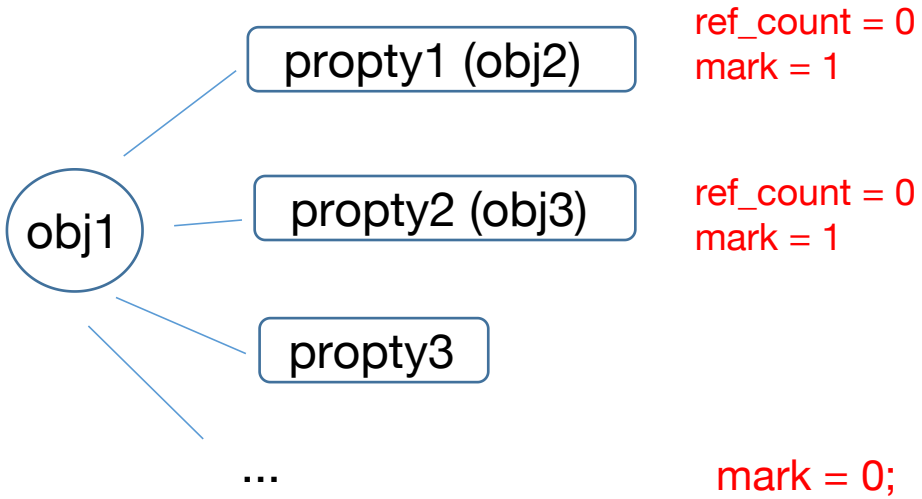
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

ref_count = 1
mark = 0

gc_scan :



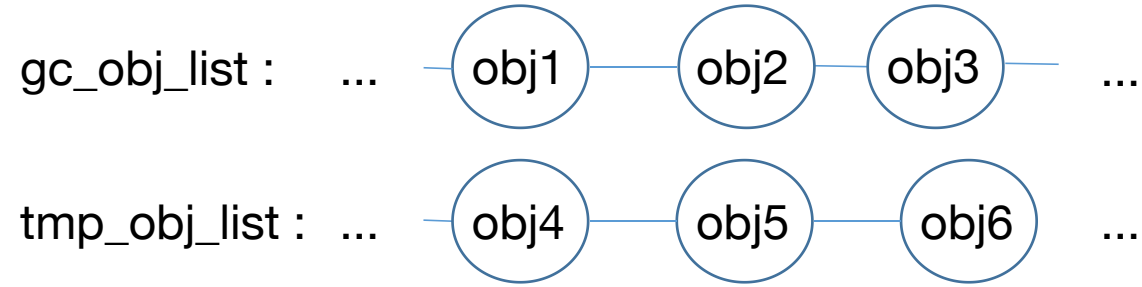
ref_count = 0
mark = 1

ref_count = 0
mark = 1

mark = 0;

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

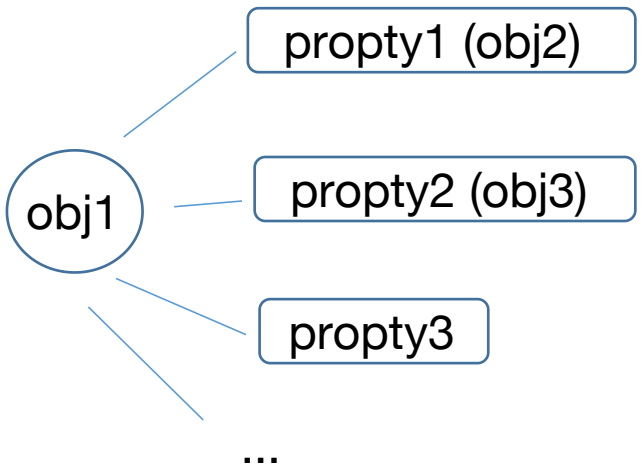
```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}
```




```
JSGCObjectHeader {
    ...
    mark;
    ref_count;
}
```

ref_count = 1
mark = 0

gc_scan :



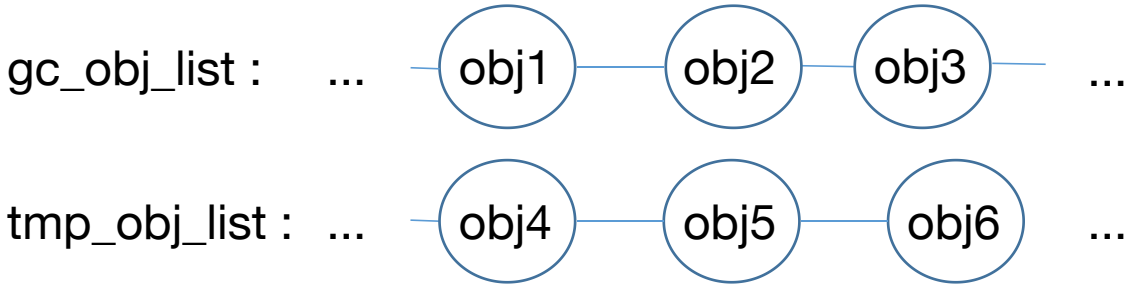
ref_count = 1
mark = 0

ref_count = 1
mark = 0

```
JSObject {
    ...
    JSGCObjectHeader *p;
}
```

```
ref_count = 1;
if (ref_count == 1)
    put propty into gc_obj_list;
    mark = 0;
```

```
JSRuntime {
    ...
    malloc_gc_threshold;
    malloc_size;
    gc_obj_list;
    tmp_obj_list;
}

JSGCObjectHeader {
    ...
    mark;
    ref_count;
}

JSObject {
    ...
    JSGCObjectHeader *p;
}
```



gc_free_cycles :

free all the obj in the tmp_obj_list

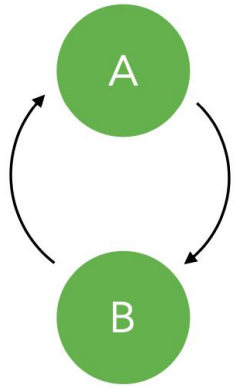
去环成功的例子

```
let a = {b: undefined};
let b = {a: a};
a.b = b;
```

存放在gc_obj_list

存放在temp_obj_list

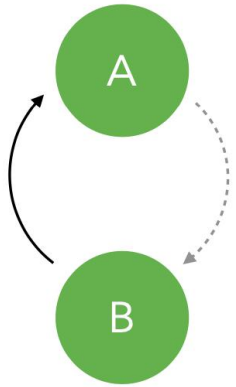
gc_decref



引用计数

A: 1

B: 1

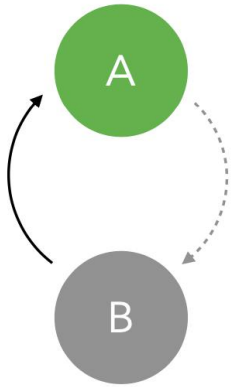


引用计数

A: 1

B: 0

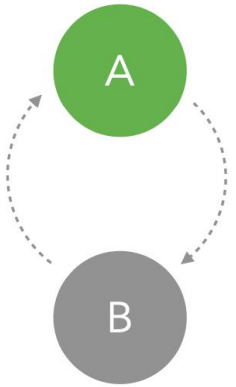
B将移动到temp_obj_list



引用计数

A: 1

B: 0

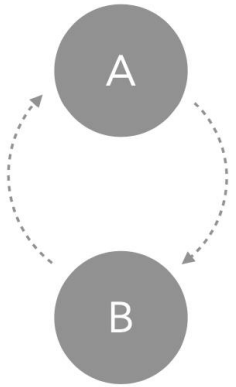


引用计数

A: 0

B: 0

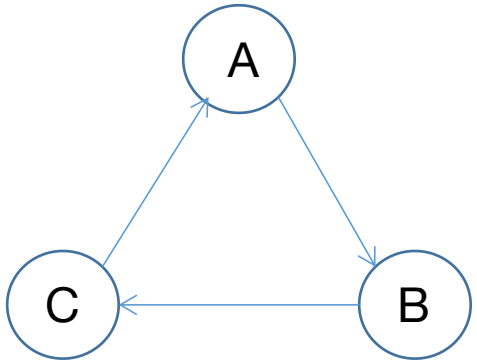
A将移动到temp_obj_list



引用计数

A: 0

B: 0



放入 tmp_obj_list 的顺序为：B->A->C

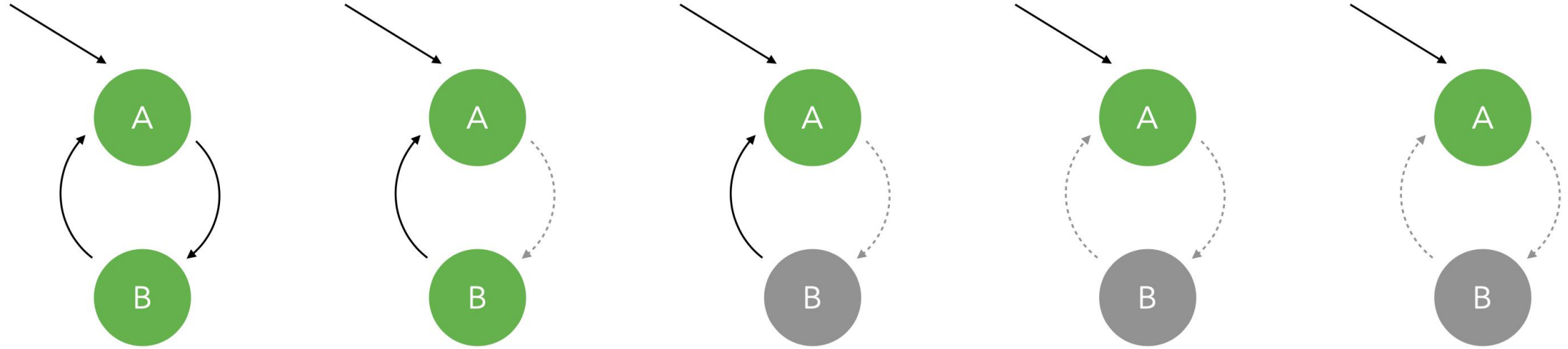
去环失败的例子

```
let a = {b: undefined};
let b = {a: a};
a.b = b;
global.a = a;
```

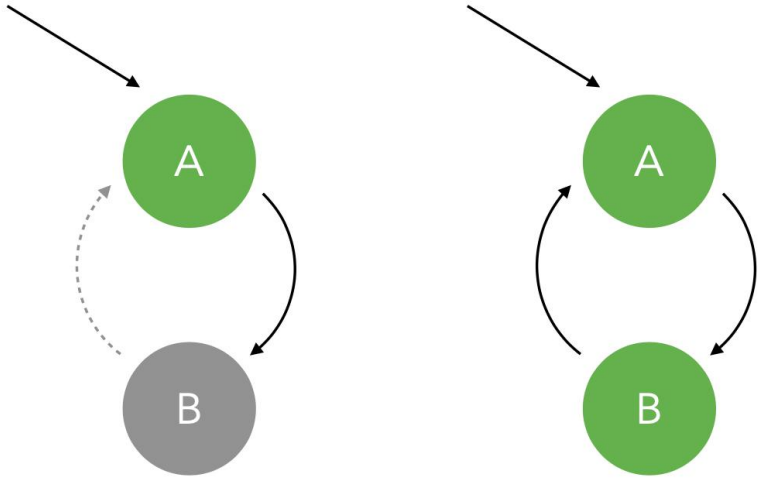
存放在gc_obj_list

存放在temp_obj_list

gc_decref



gc_scan



因为A还被其他地方引用，不能被移动到temp_obj_list
而移过去的B因为A的引用又给添加回来了

触发 js_trigger_gc ( ) 的时机：

1. JS_NewObjectFromShape ( ) 创建新对象。

2. JS_CallInternal ( ) 指令形式的函数执行完毕返回时，先利用 JS_FreeValue ( ) 回收栈上参数和变量 (JSValue)，再调用 js_trigger_gc ( )。

3. 重新赋值情况时没有额外调用 js_trigger_gc ( )。

4. 在整个程序结束的时候一定会执行一次 gc。

QuickJS vs. JerryScript

QuickJS :

```
Sullivans-MacBook-Pro:quickjs源码分析 kxw$ qjs examples/gc.js
GC: size=77840
runGC
GC: time=0.000105
GC: size=19904
```

```
for (var i = 0; i < 10; i++) {
  var obj = {}, obj_l;
  obj_l = obj;

  for (var k = 0; k < 150; k++) {
    obj_l.prop = {};
    obj_l = obj_l.prop;
  }
}
```

JerryScript :

```
Sullivans-MacBook-Pro:bin kxw$ ./jerry gc.js
GC: size=8184
RunGC
GC: time=0.000020
GC: size=3384

GC: size=8176
RunGC
GC: time=0.000014
GC: size=3376

GC: size=8176
RunGC
GC: time=0.000018
GC: size=3376

GC: size=8176
RunGC
GC: time=0.000024
GC: size=3376

GC: size=8128
RunGC
GC: time=0.000023
GC: size=2584
```

QuickJS GC 特性：

1. 整个程序结束的时候，会调用一次 JS_RunGC
2. rt->malloc_gc_threshold = 256*1024 = 2^18 (初始化)
3. rt->malloc_gc_threshold = rt->malloc_state.malloc_size
   + (rt->malloc_state.malloc_size >> 1) (更新)

JerryScript GC 特性：

1. 阈值更新规则：保持大于实际使用内存不超过一个 limit 的大小
2. CONFIG_GC_LIMIT = 2^13 (默认)

对比可以发现，JerryScript GC 堆大小小于 QuickJS GC 堆；

因此，针对同一段程序，Jerry 会执行更多次的 GC;

但总体而言，Jerry GC 的耗时小于 QuickJS