

Lecture 6.2

动态内存管理

徐 辉

xuh@fudan.edu.cn



大纲

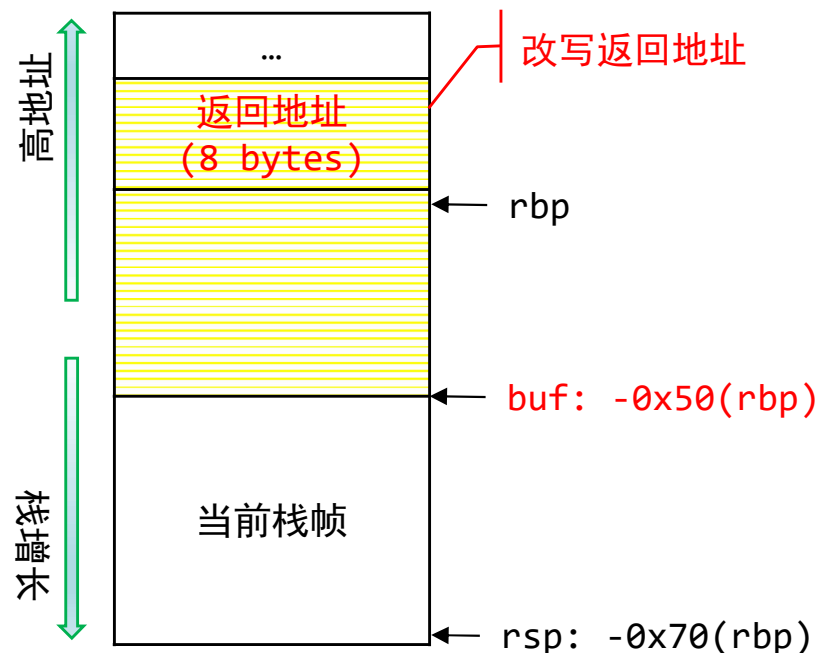
一、内存管理机制

二、智能指针

三、垃圾回收

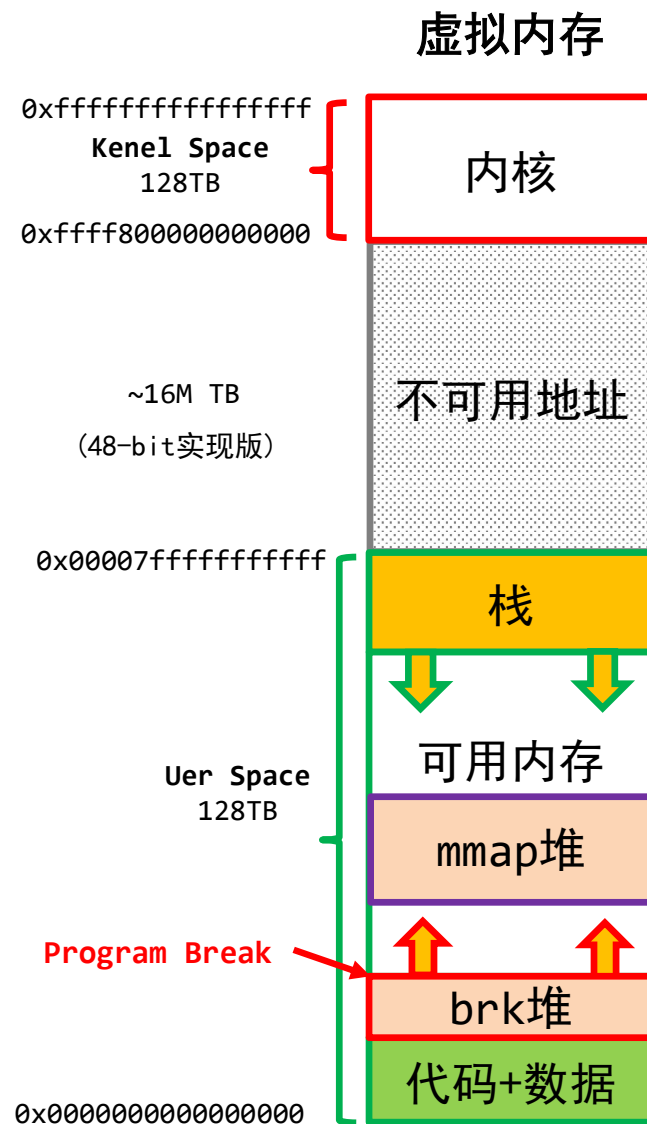
栈的机制相对简单

- 调用规约
 - 新的函数调用会创建栈帧
 - 函数返回自动退栈
- 栈溢出 (stack overflow)
 - 默认8MB，超过则崩溃
 - 可靠性风险
- 缓冲区溢出 (buffer overflow)
 - 写入数据大小超出预留空间
 - 安全问题



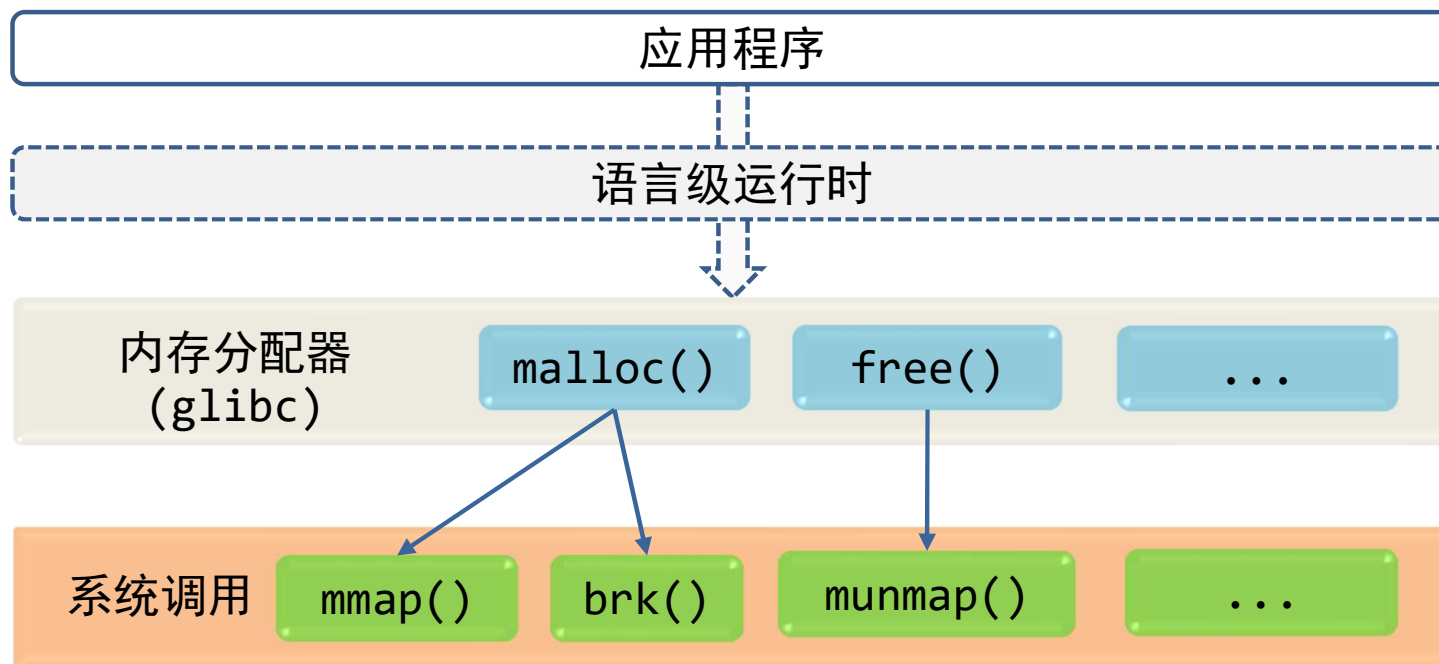
堆的管理比较复杂

- Program break:
 - 如Linux的brk()系统调用
 - 内存小于阈值时使用
 - 分配区间是连续的
 - 一般不主动回收
 - 使用链表管理空闲内存
- 内存映射:
 - 如Linux的mmap()系统调用
 - 早期Unix系统不支持
 - 容易通过munmap()释放
 - 一般内存大于阈值时使用



堆分配API: glibc APIs

- 分配: `malloc(size_t n)`
 - 分配n个字节的内存空间, 并返回指向该内存的指针
 - 内存并未被清理
- 堆释放: `free(void * p)`
 - 释放p指向的内存空间;
 - 不会直接返还操作系统 (brk)
 - 如果`free(p)`之前被调用过, 会导致未定义行为
 - 如果p是空指针, 则不会进行任何操作



堆分配器: Doug Lea's Allocator (dlmalloc)

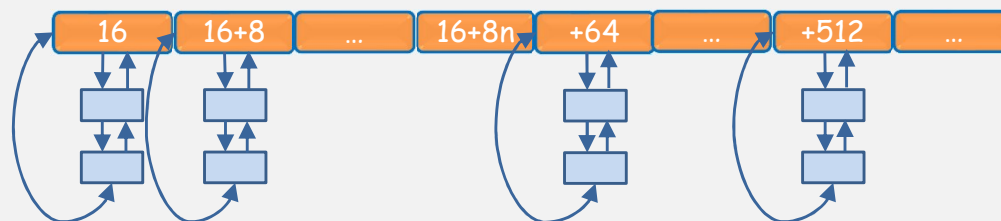
- 通过bins管理空闲内存块 (chunks)
- 每个Regular bin是一个双向链表, 包含大小固定的块
 - Fastbin采用单向链表
- malloc()进行内存分配时需找到合适bin

Bins for sizes < 512 bytes contain chunks of all the same size, spaced 8 bytes apart. Larger bins are approximately logarithmically spaced:

small bins

large bins

4 bins of size	8
32 bins of size	64
16 bins of size	512
8 bins of size	4096
4 bins of size	32768
2 bins of size	262144
1 bin of size	what's left



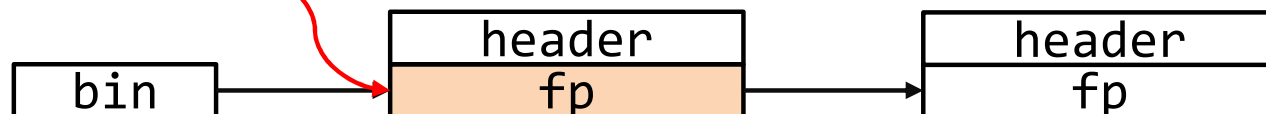
The bins top out around 1MB because we expect to service large requests via mmap.

堆上的问题

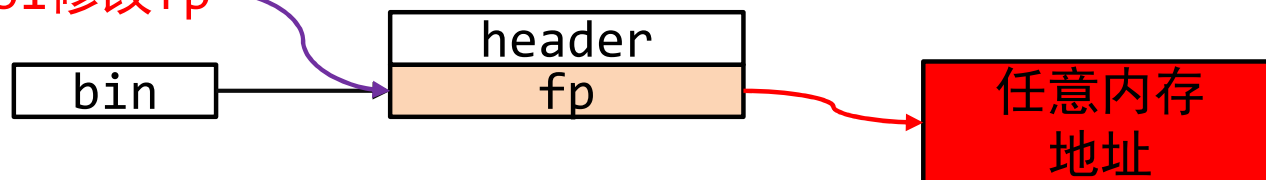
- 安全性缺陷
 - 释放后使用：use-after-free
 - 堆溢出：heap overflow
 - 双重释放：double free
- 可靠性缺陷
 - 内存泄露：memory leakage
 - 堆耗尽：heap exhaustion

Use-After-Free

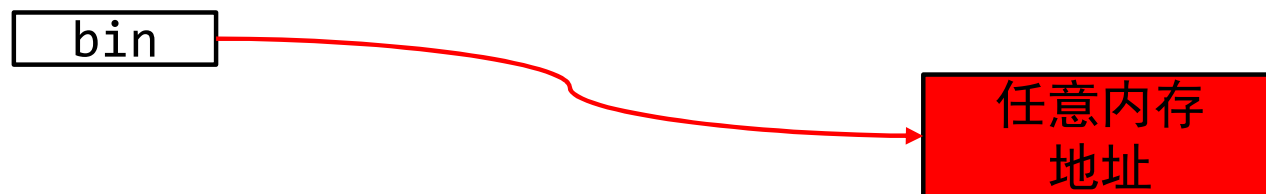
1. 获得悬空指针p1



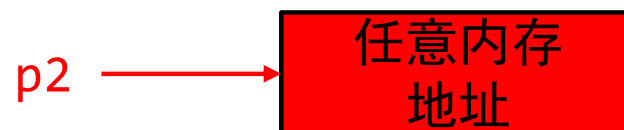
2. 通过p1修改fp



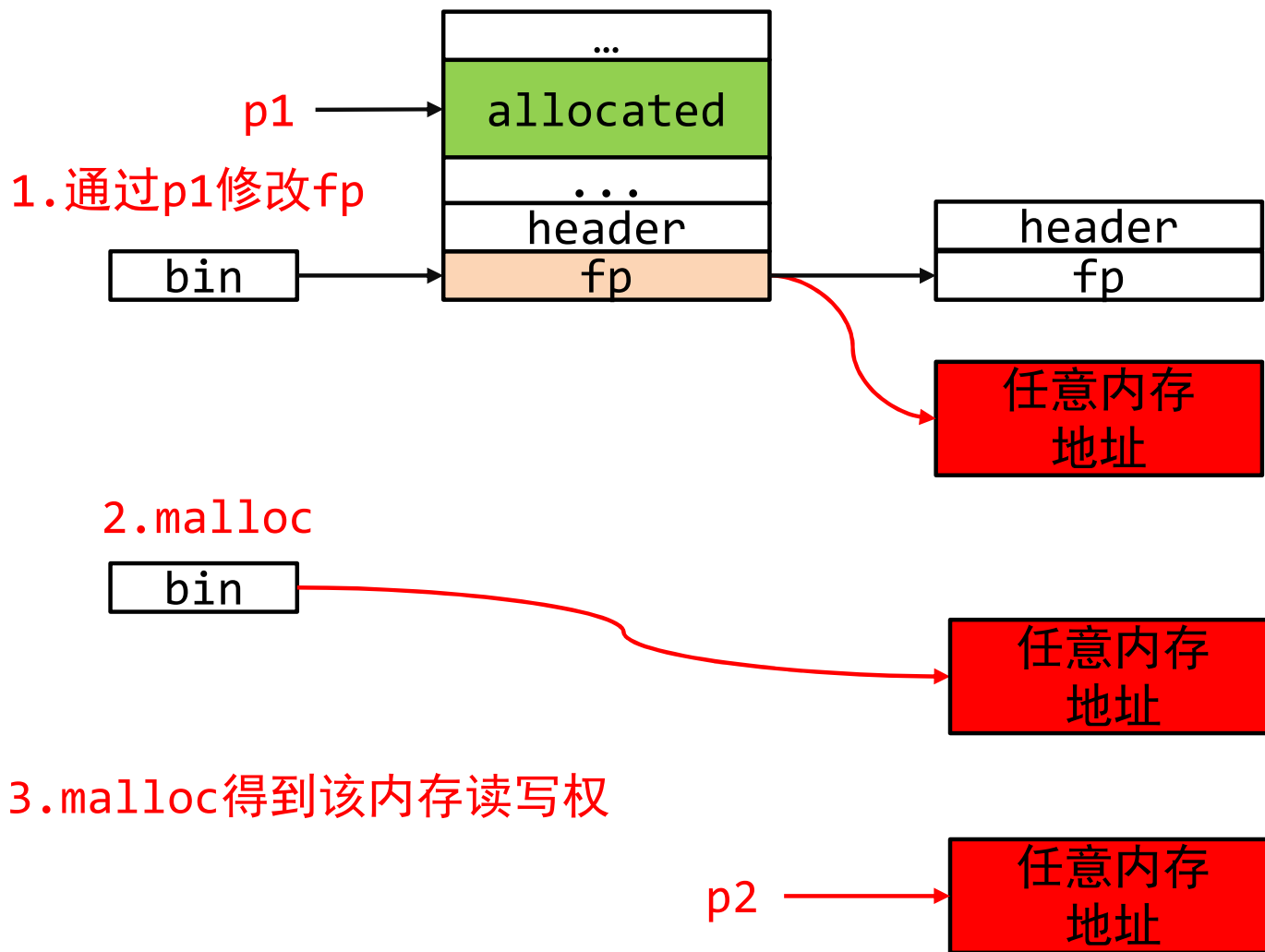
3. malloc



4. malloc得到该内存读写权



堆溢出

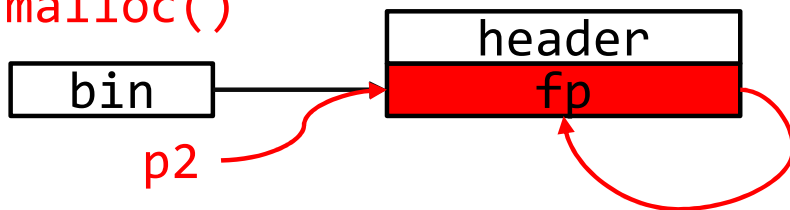


Double Free

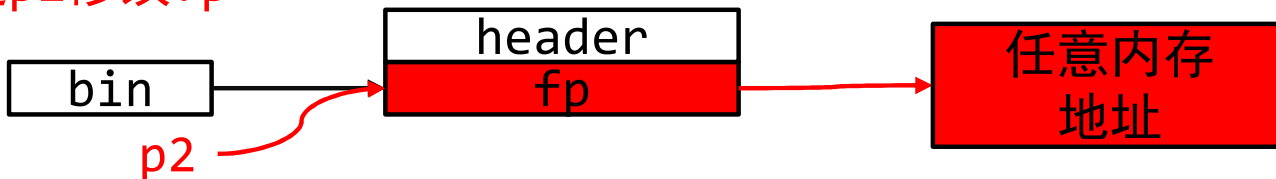
1. free(p1)



2. malloc()



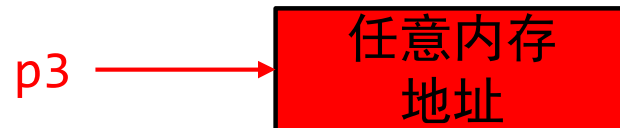
3. 通过p2修改fp



4. malloc()

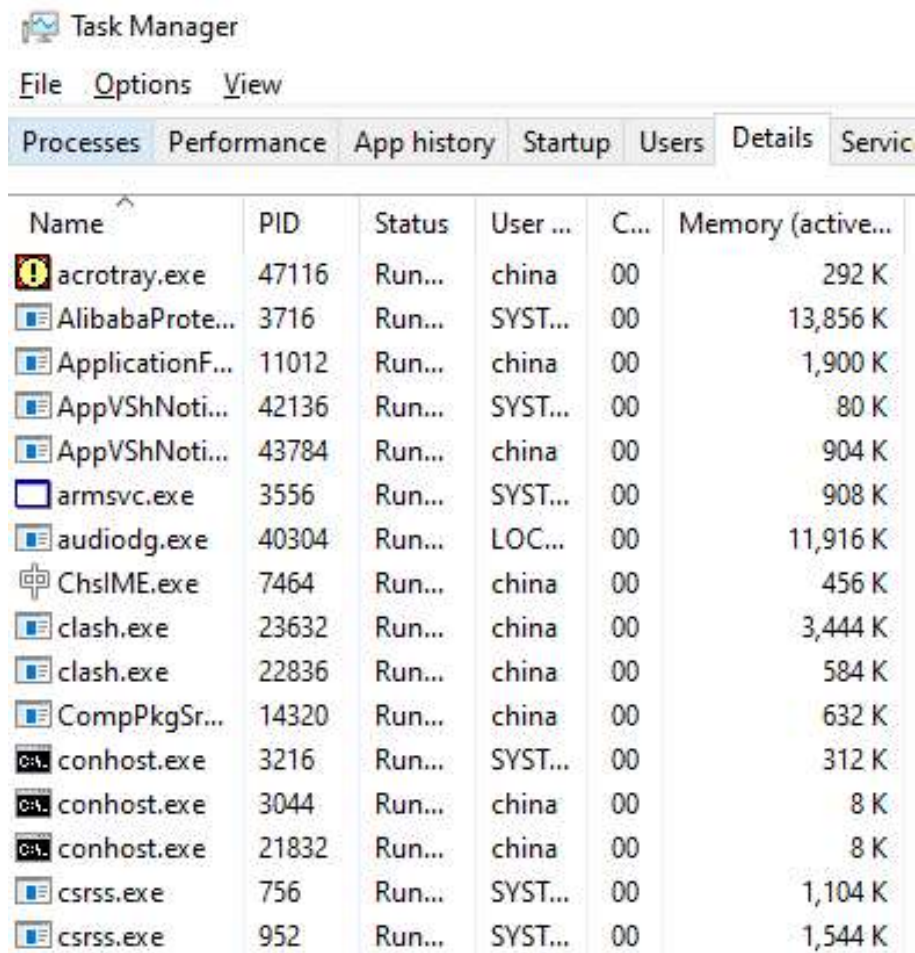


5. malloc()得到该内存读写权



堆空间耗尽：Heap Exhaustion

- 堆空间何时会耗尽？
 - 物理内存用尽？
 - 虚拟内存用尽？
 - 地址空间用尽？
- 不同操作系统存在区别：
 - Windows采用eager的机制；
 - 分配即占用
 - Linux采用lazy的机制；
 - mmap()/brk()只是分配地址
 - 并不分配物理页
 - 访问才占用

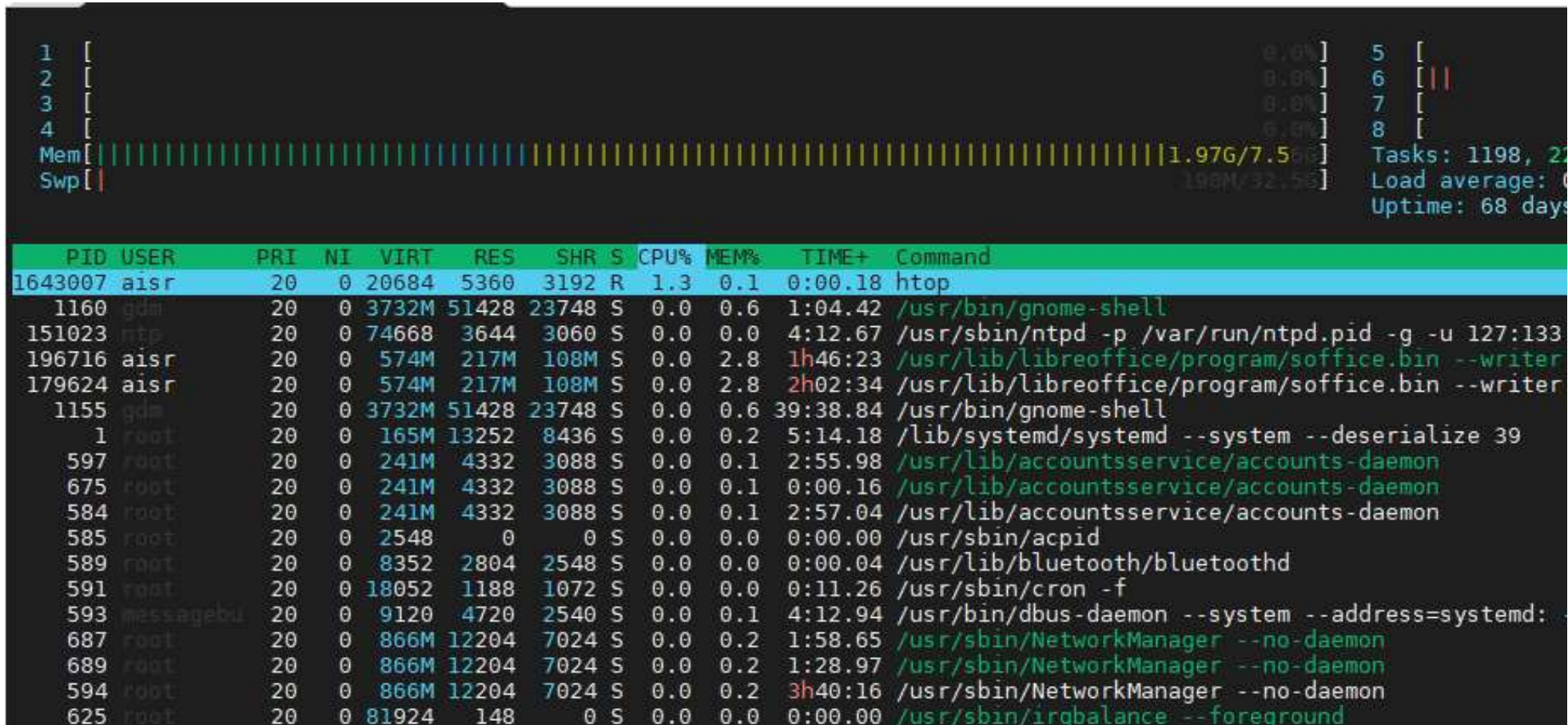


The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It displays a list of running processes, including system processes like csrss.exe and user applications like acrotray.exe and clash.exe. The 'Memory (active...)' column shows the memory usage for each process.

Name	PID	Status	User	C...	Memory (active...)
acrotray.exe	47116	Run...	china	00	292 K
AlibabaProte...	3716	Run...	SYST...	00	13,856 K
ApplicationF...	11012	Run...	china	00	1,900 K
AppVShNoti...	42136	Run...	SYST...	00	80 K
AppVShNoti...	43784	Run...	china	00	904 K
armsvc.exe	3556	Run...	SYST...	00	908 K
audiodg.exe	40304	Run...	LOC...	00	11,916 K
ChslME.exe	7464	Run...	china	00	456 K
clash.exe	23632	Run...	china	00	3,444 K
clash.exe	22836	Run...	china	00	584 K
CompPkgSr...	14320	Run...	china	00	632 K
conhost.exe	3216	Run...	SYST...	00	312 K
conhost.exe	3044	Run...	china	00	8 K
conhost.exe	21832	Run...	china	00	8 K
csrss.exe	756	Run...	SYST...	00	1,104 K
csrss.exe	952	Run...	SYST...	00	1,544 K

Linux内存占用

- VIRT (Virtual Image) : 进程镜像可用的内存地址空间;
- RES (Resident Size) : 物理内存占用, non-swapped;
- SHR (Shared Mem) : 和其它进程共享的内存



内存泄露：Memory Leakage

- 空闲内存不能及时回收造成可用内存越来越少。
 - 忘记free
 - 循环引用

编程语言设计的任务

- 程序员是不可靠的，如何
 - 提高内存使用效率？
 - 预防内存可用性缺陷？
 - 预防内存安全缺陷？
- 实现自动内存管理：
 - 智能指针
 - 垃圾回收

大纲

一、内存管理机制

二、智能指针

三、垃圾回收

如何自动释放内存

- 传统C/C++需要手动释放内存
 - malloc/free
 - constructor/destructor
- 如何自动释放内存？
 - 静态分析目标对象的lifespan
 - 动态分析目标对象的引用数

下面这段C++代码应输出什么？

```
class MyClass{
public:
    int val;
    MyClass(int v) { val = v; }
    int add(MyClass* a) { return val + a->val; }
    int add(MyClass& a) { return val + a.val; }
    int add2(MyClass a) { return val + a.val; }
    ~MyClass() { cout << "delete obj:"<< val << endl; }
};

void foo(MyClass* p){
    MyClass s{3};
    p = &s;
}

int main() {
    MyClass s{1};
    MyClass* p1 = new MyClass(2);
    MyClass* p2;
    foo(p2);
    cout << s.add(p1) << endl;
    cout << p1->add(p2) << endl;
    cout << p1->add2(s) << endl;
}
```

```
delete MyClass obj:3
3
-98693131
3
delete MyClass obj:1
delete MyClass obj:1
```

- s保存在栈上，栈帧销毁时自动析构
- 自动delete new的对象？

编译时分析目标对象的lifespan?

- 基本思路：
 - 第一步：需要确定目标对象的所有别名
 - 指针分析问题，基本不可行
 - 不考虑地址运算
 - 不考虑裸指针
 - 第二步：分析所有别名的def-use
- 如果限制对象只能有一个所有者？
 - Rust所有权机制

Rust所有权模型 => XOR Mutability

- 一个对象有且只有一个所有者
- 所有权可以转移给其它变量（用完不还）
- 所有权可以被其它变量借用（用完归还）
只读（immutable）借用：&
可变（mutable）借用：& mut

```
fn main(){  
    let mut alice = Box::new(1);  
    let bob = alice;  
    println!("bob:{}", bob);  
    println!("alice:{}", alice);  
}
```



alice 拥有Box对象
转移所有权转移给 bob,
alice失去Box对象的所有权

```
fn main(){  
    let mut alice = Box::new(1);  
    let bob = &alice;  
    println!("alice:{}", alice);  
    println!("bob:{}", bob);  
    *alice = 2;  
}
```



bob只读借用Box对象
alice临时失去修改权，保留只读权
alice可读
bob自动归还Box对象，alice恢复修改权

如果必须违背XOR Mutability怎么办？

- 以双向链表为例，中间节点被前后两个节点访问
- 为了提升可用性需要妥协：
 - 智能指针（性能损失）
 - 允许使用裸指针（unsafe）



```
struct List{  
    val: u64,  
    prev: Option<Rc<RefCell<List>>>,  
    next: Option<Rc<RefCell<List>>>,  
}
```

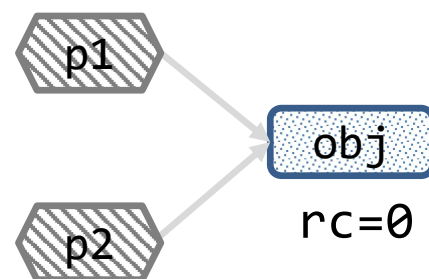
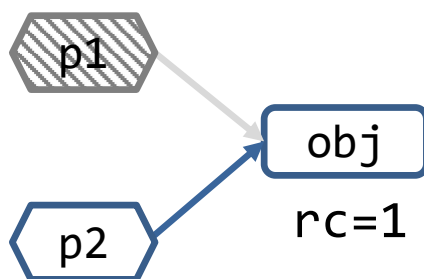
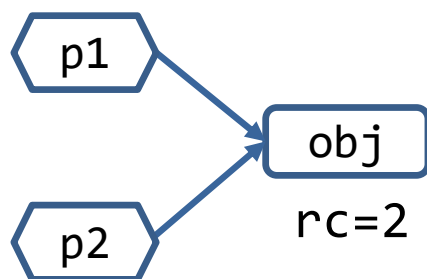
方法一：智能指针

```
struct List{  
    val: u64,  
    next: *mut List,  
    prev: *mut List,  
}
```

方法二：允许使用裸指针

动态分析记录引用数

- 每产生一个新的引用，计数器加1，反之则减1；
- 引用数清零时自动析构
- Rust语言中的Reference Counter
- C++中的智能指针
- 问：如何实现线程安全的引用计数？



C++ (11) 智能指针

- 独占型指针: `unique_ptr`
 - 通过`move`转移所有权
- 共享型指针: `shared_ptr`
 - 可以通过`reset()`主动释放引用数;
 - 引用数为0时自动析构目标对象。

```
int main() {  
    unique_ptr<MyClass> up1(new MyClass(2));  
    //unique_ptr<MyClass> up2 = up1; //编译报错  
    unique_ptr<MyClass> up2 = move(up1);  
    //cout << up1->val << endl; //segmentation fault  
    cout << up2->val << endl;  
  
    shared_ptr<MyClass> sp1(new MyClass(2));  
    shared_ptr<MyClass> sp2 = p1;  
}
```

下面代码会输出什么？

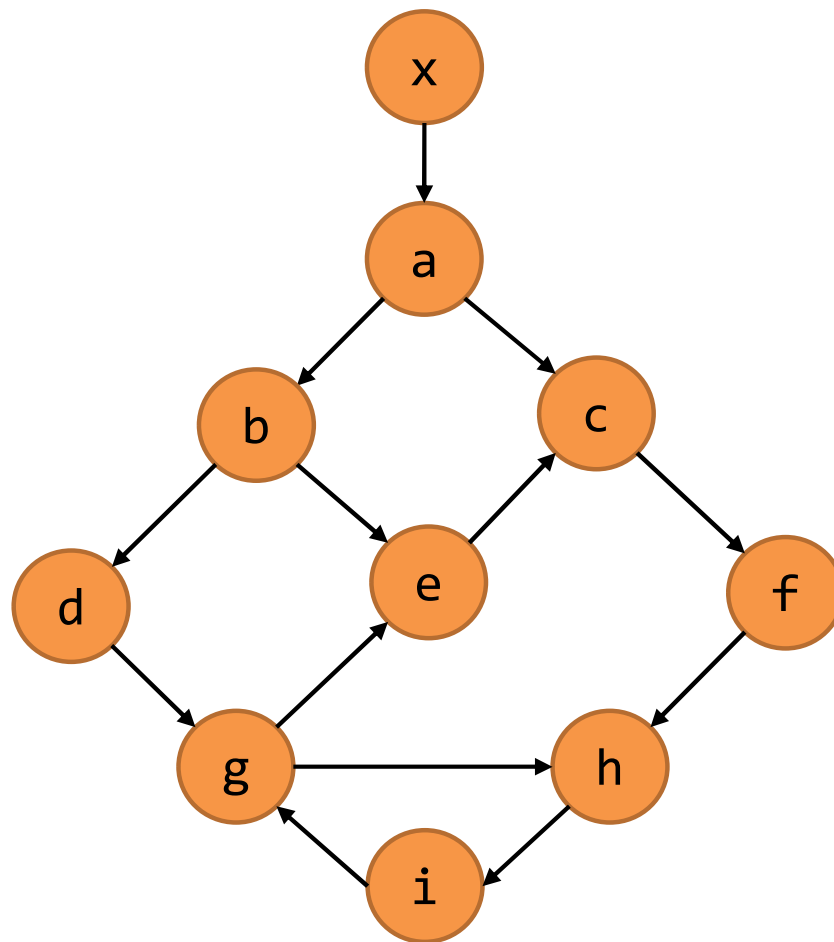
```
class MyClass{
public:
    int val;
    MyClass(int v) { val = v; }
    ~MyClass() { cout << "delete obj:"<< val << endl; }
};

int main() {
    MyClass* p0 = new MyClass(1);
    {
        shared_ptr<MyClass> p1(new MyClass(2));
        shared_ptr<MyClass> p2 = p1;
        shared_ptr<MyClass> p3(p0);
    }
    cout << p0->val << endl;
}
```

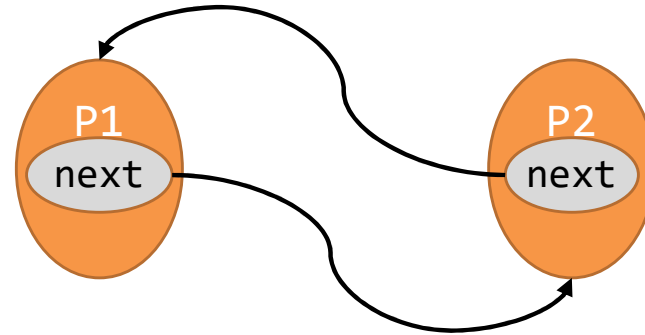
```
./a.out
delete obj:1
delete obj:2
0
```

练习：

- 如果指针指向a->b被修改会发生什么？



智能指针的主要问题：循环引用



```
class MyList{
public:
    int val;
    shared_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< val << endl; }
};

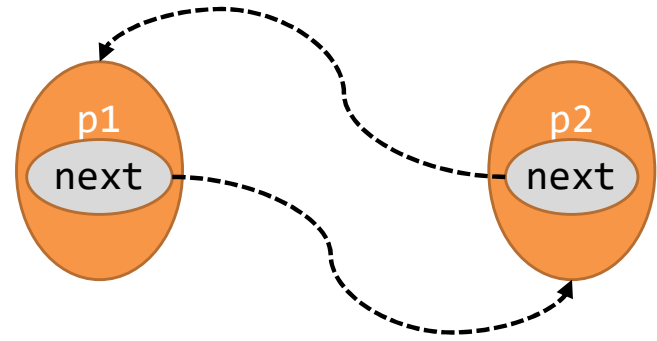
int main() {
    shared_ptr<MyList> p1 = make_shared<MyList>();
    shared_ptr<MyList> p2 = make_shared<MyList>();
    p1->val = 1;
    p2->val = 2;
    p1->next = p2;
    p2->next = p1;
}
```

解决循环引用：weak_ptr

- 不改变引用计数

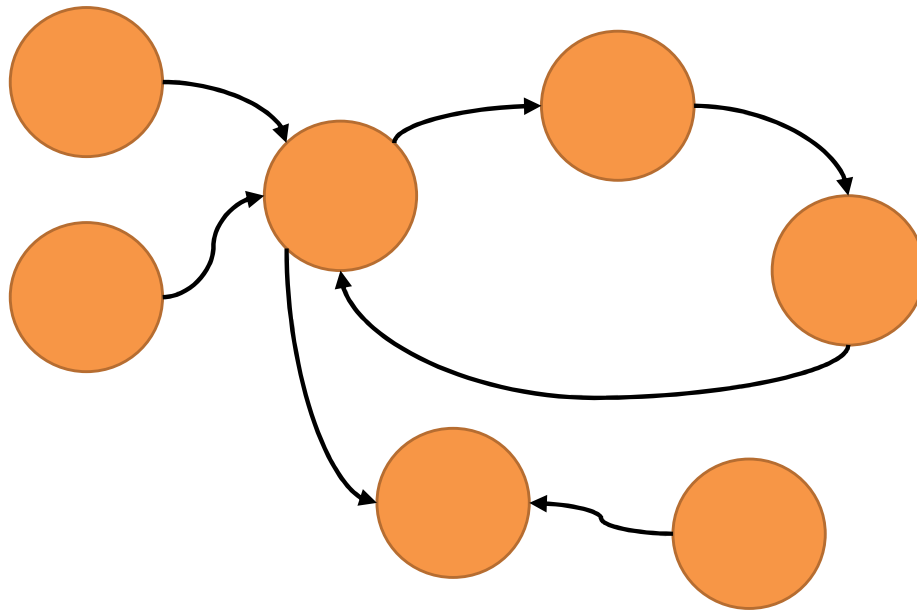
```
class MyList{
public:
    int val;
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< val << endl; }
};

int main() {
    shared_ptr<MyList> p1 = make_shared<MyList>();
    shared_ptr<MyList> p2 = make_shared<MyList>();
    p1->val = 1;
    p2->val = 2;
    p1->next = p2;
    p2->next = p1;
}
```



如何检测循环引用？

- 1) 如何设计检测算法？
- 2) 何时触发算法？
- 3) 如何处理循环引用？



一些易混淆的基本概念

- 胖指针
- deep copy vs shallow copy

大纲

- 一、内存管理机制
- 二、智能指针
- 三、垃圾回收

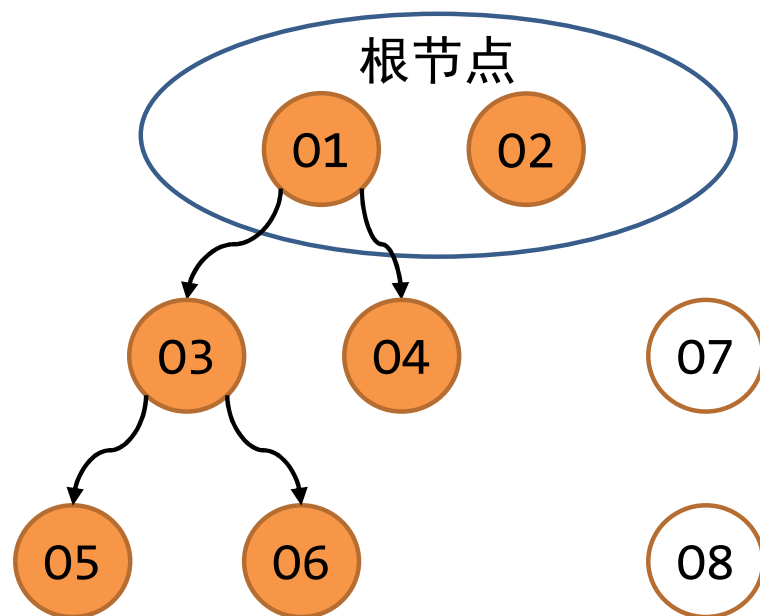


垃圾回收

- 智能指针采用动态计数方法，运行开销平滑。
- 垃圾回收定时清理无效内存，性能代价明显。
- 垃圾回收需要考虑的问题：
 - 何时触发垃圾回收？
 - 哪些内存需要回收？
 - 可达性分析
 - 如何回收性能最优？
 - 卡顿问题
 - 碎片化问题

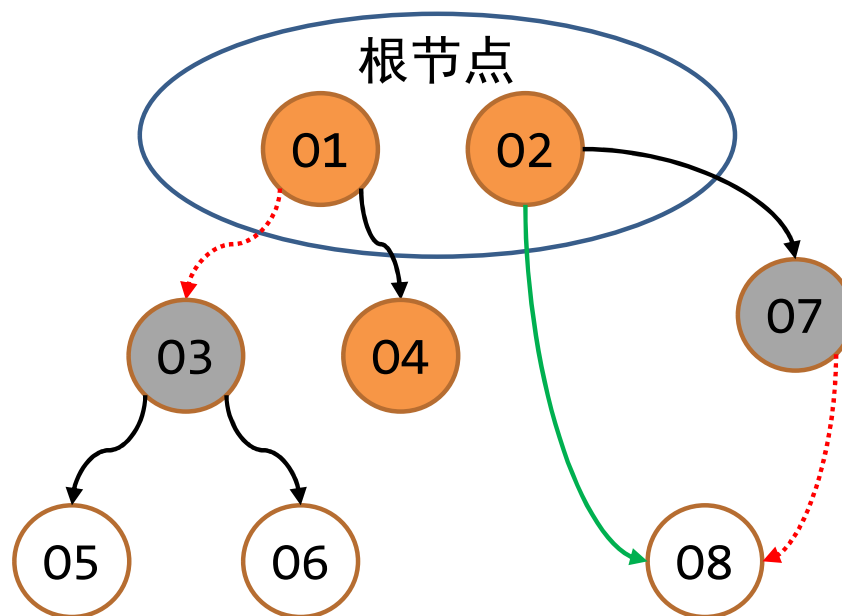
可达性分析

- 一般分析需要暂停程序 (stop the world)
- 从特定的根节点出发
- 不可达的对象即应回收对象 (垃圾)



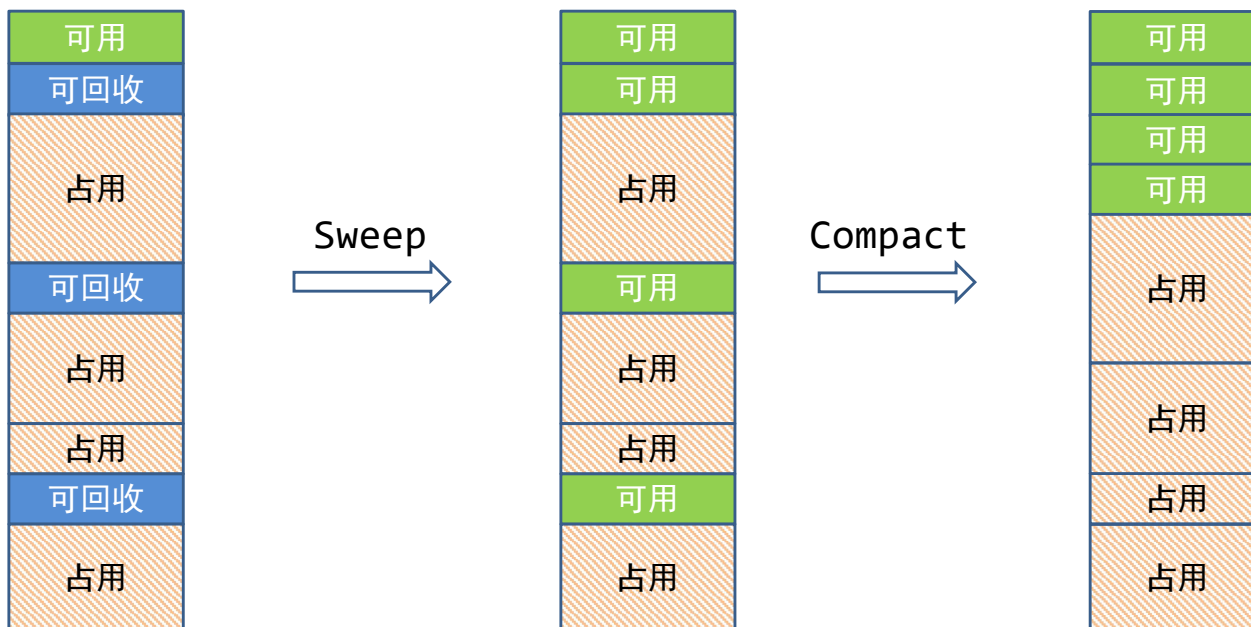
利用空闲时间增量标记？

- 解决stop-the-world问题
- 用第三种颜色（灰色）记录分析过程：
 - 橘色：对象可达，且已分析完毕
 - 灰色：对象可达，还未分析完毕
 - 白色：潜在不可达对象
- 是否会产生误报？
 - false negative
 - false positive
 - 应如何应对？



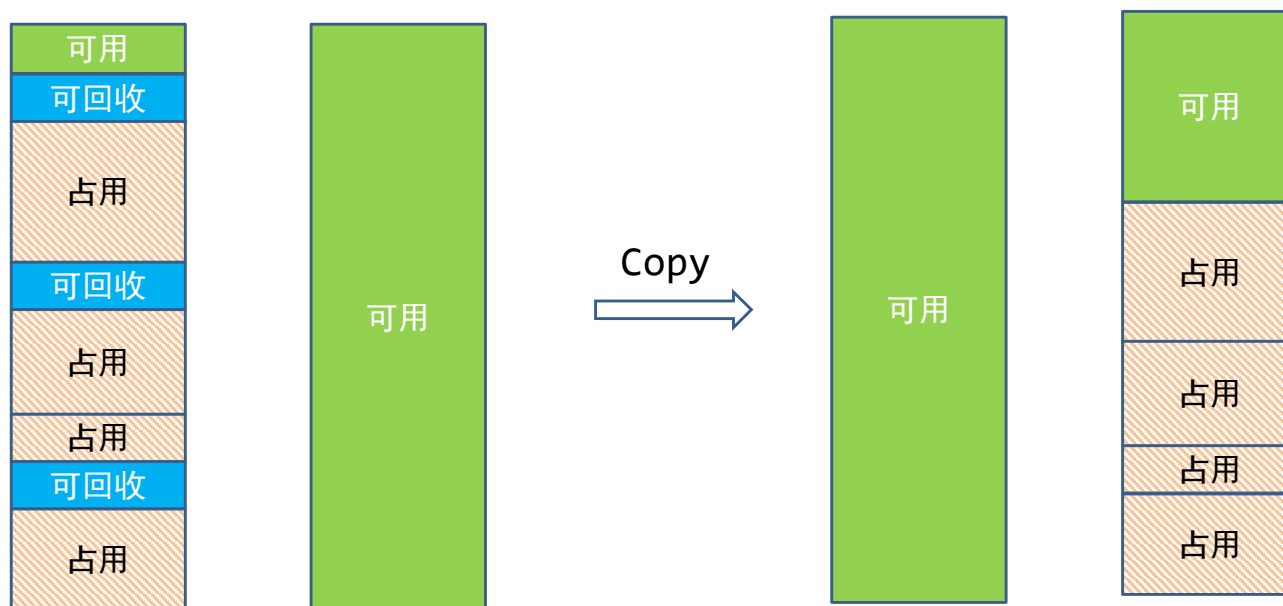
如何回收

- 标记清除方法：mark-sweep
- 解决碎片化问题：mark-compact
 - 碎片化整理开销



如何整理内存时不影响使用？

- 标记复制算法（Mark-Copy）：将内存分为两部分
 - 复制过程中原内存仍可被访问；
 - 空间换时间。



如何进一步优化？

- 观察：
 - 新创建的对象更容易成为垃圾
 - 多次GC存活下来的对象大概率下一轮还能存活
- 利用上述经验降低内存拷贝频率？

分代收集算法：Generational Collection

- 乐园区（Eden）：保存新建对象，空间不足时触发Minor GC
- 幸存区（Survivor）：保存Minor GC后的存活对象
 - 分为from和to两部分，功能完全相同
 - Minor GC($\text{eden} + \text{from}$) \Rightarrow to,
 - Minor GC($\text{eden} + \text{to}$) \Rightarrow from
- 长寿区（Old）：保存多轮Minor GC后存活下来的对象，空间不足时触发Major GC
 - 大对象直接放入长寿区，避免Minor GC时的复制开销



垃圾回收实现参考

- 为C实现垃圾回收
 - 参考教程: <https://maplant.com/gc.html>
 - BoehmGC GC (Malloc)
 - <https://www.hboehm.info/gc/#details>