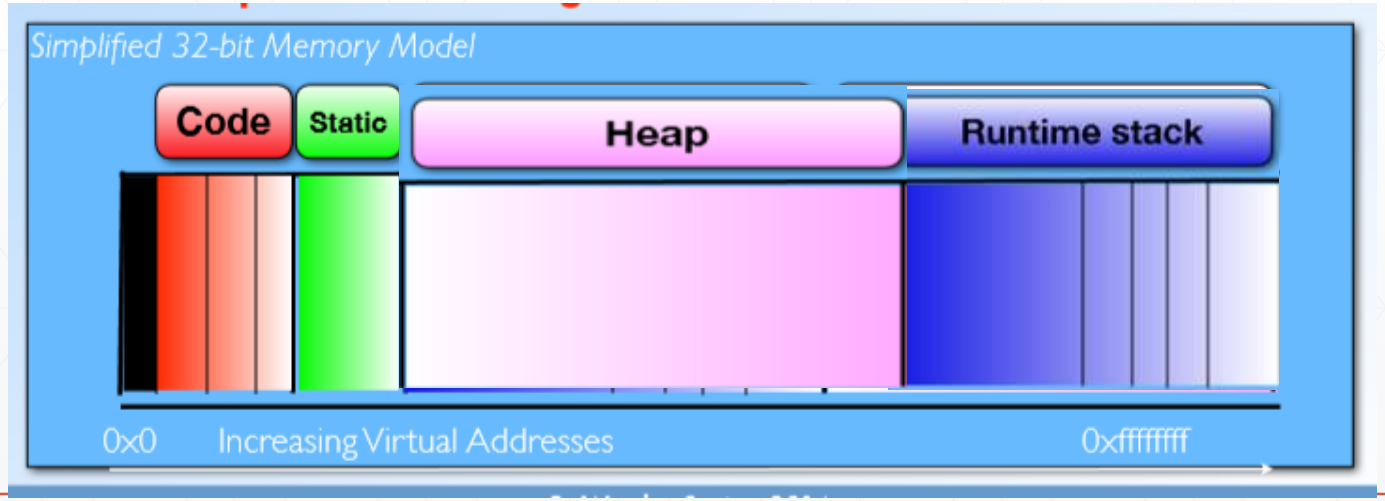
垃圾收集算法

北京理工大学 计算机学院

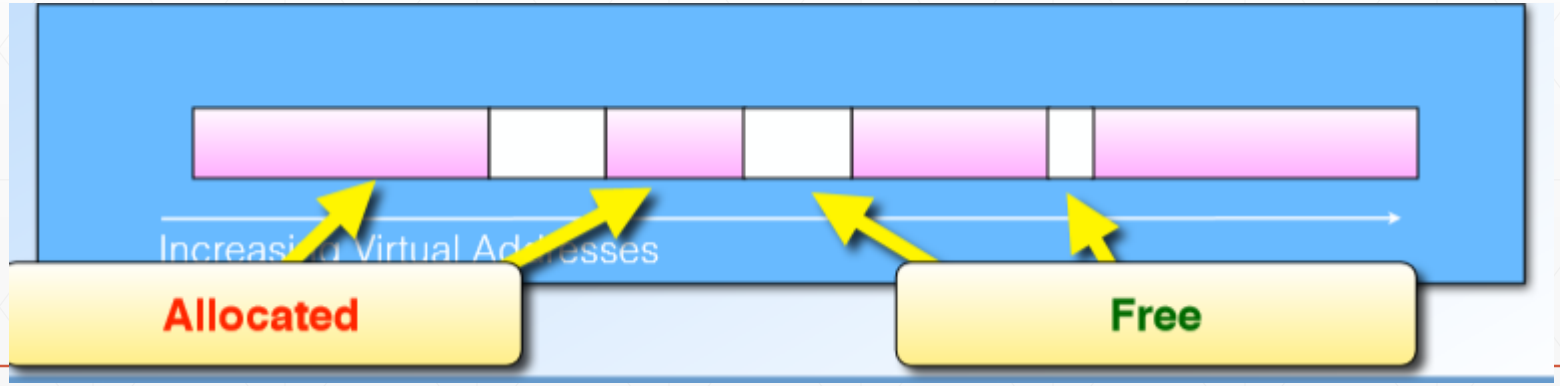
堆/Heap

- 数据对象存活期比较自由
  - 可以在任一点分配和释放
  - 子过程返回之后仍然存活
  - 按需灵活分配，但是管理起来非常复杂



堆/Heap

- 内存分配
  - 多数显式分配
    - C: malloc/alloc
    - C++/Java: new
  - 编译器隐式调用分配器
    - Prolog
- 内存回收
  - 部分显式
    - C: free
    - C++: delete
  - 部分自动回收
    - Java、Python



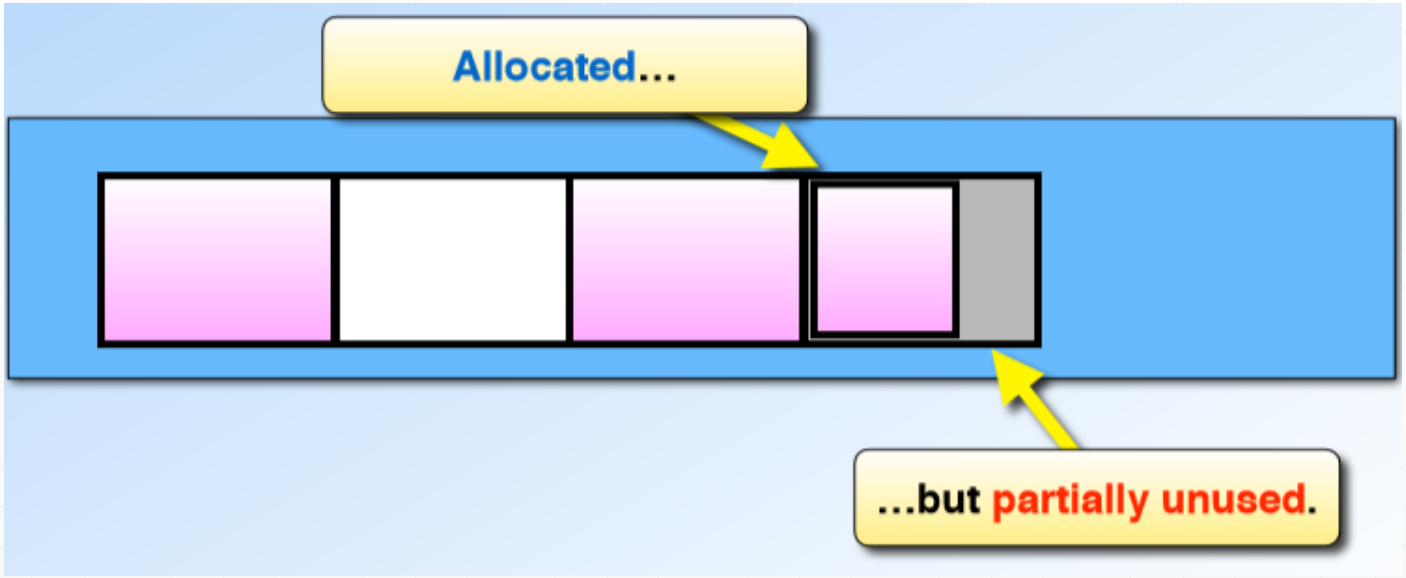
内存分配

- 问题描述：对于给定的大小S，找到连续的内存区域大小至少为S
- 常见技术

可变大小	固定大小	实际OS中
• first-fit，best-fit • 分配和释放均比较耗时	• 分配和释放比较高效 • 2^n或者斐波那契数列 • Buddy allocator、slab allocator	• 分配器的性能对许多负载比较重要 • 采用固定大小 • 高效并发分配器仍然是研究问题

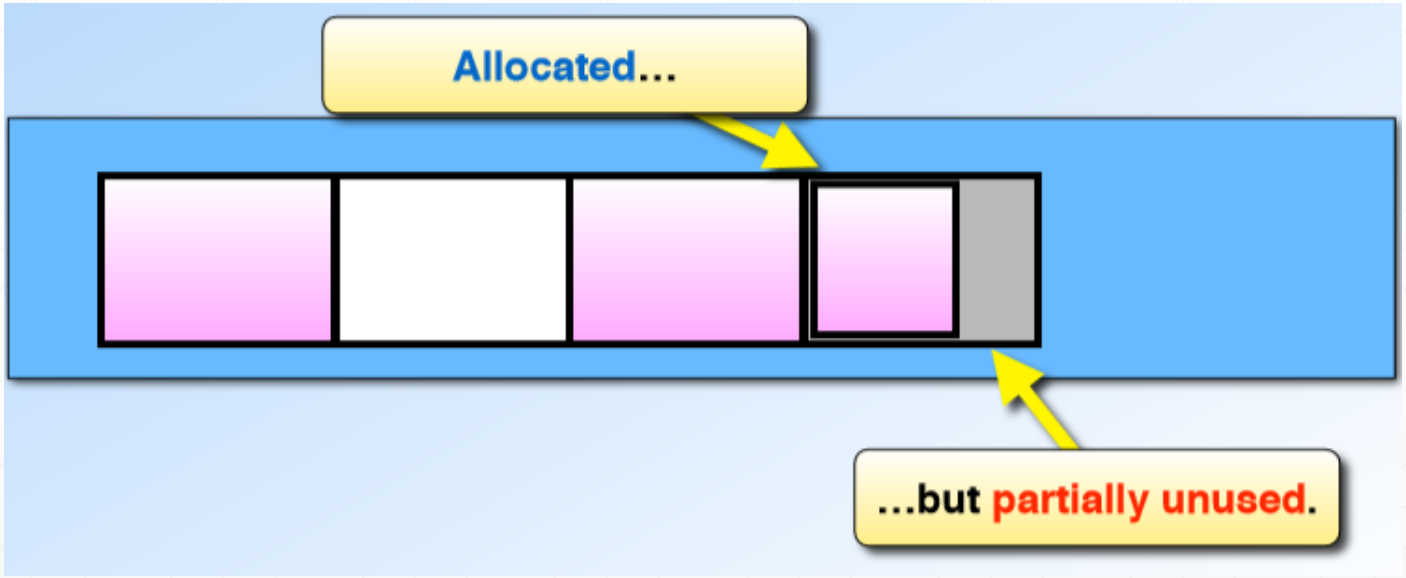
内存分配

- 固定大小的问题
  - 造成一定空间的浪费→内存碎片



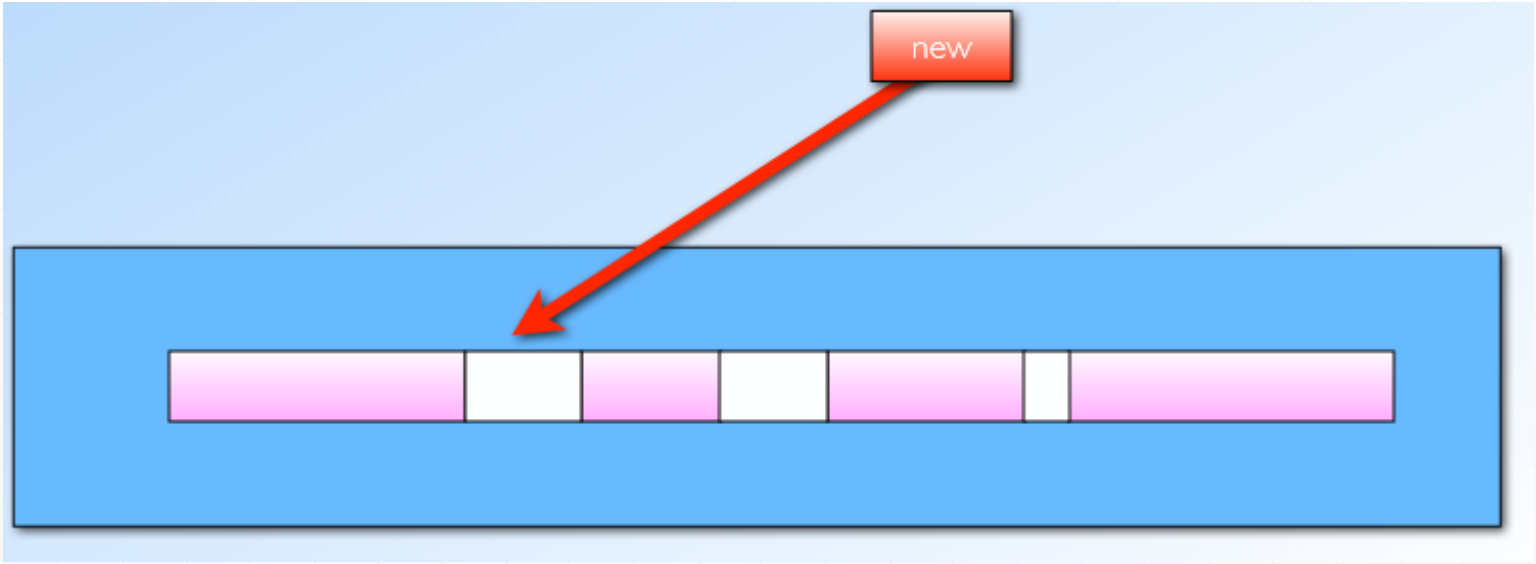
内存分配

- 固定大小的问题
  - 造成一定空间的浪费→内存碎片



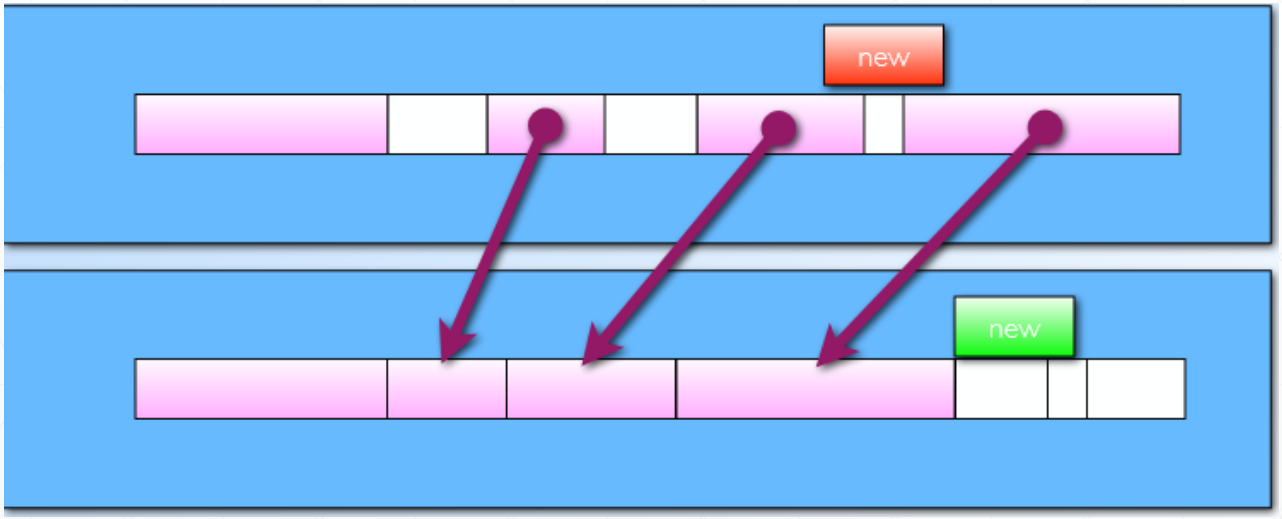
内存分配

- 固定大小的问题
  - 造成一定空间的浪费→内存碎片
  - 空闲的总空间够，但是找不到合适的空间



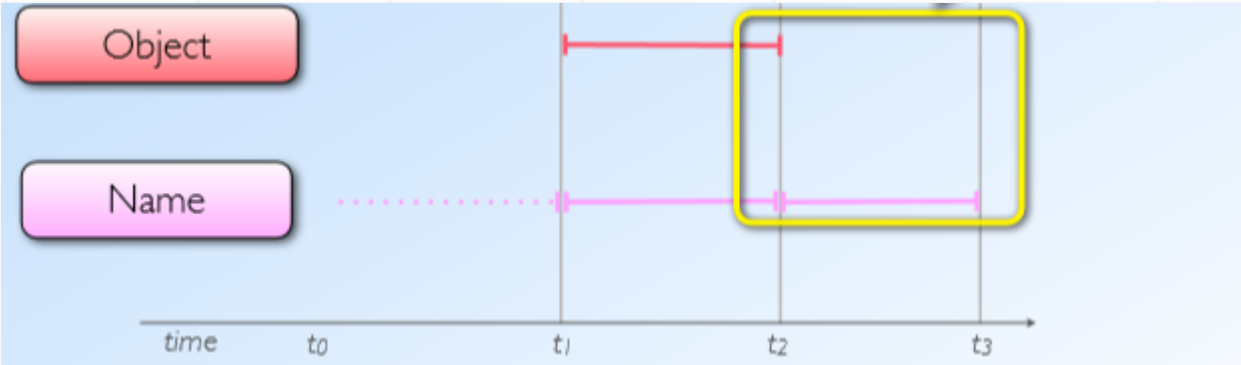
内存分配

- 固定大小的问题
  - 造成一定空间的浪费→内存碎片
  - 内存空间整理：进行合并



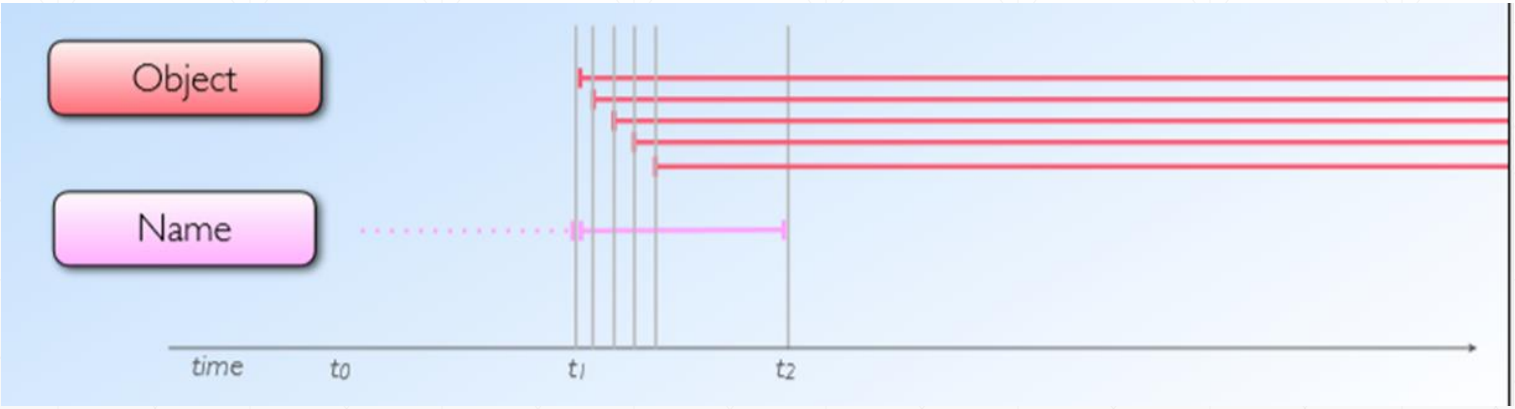
手动内存管理的常见问题

- 问题1：悬空指针（Dangling pointer）
  - 绑定的声明周期长于对象的声明周期
  - 对象释放过早，访问变为非法
  - 导致“use-after-free”的问题



手动内存管理的常见问题

- 问题2：内存泄露（Memory leaks）
  - 忘记释放，对象永远存在
  - 长时间运行导致内存耗尽
  - 进程结束时才会释放



垃圾收集

- 运行时系统安全时自动回收对象
- 自动内存管理技术，用户无需关心内存释放
- 使用垃圾回收的语言
  - 函数式编程语言：Haskell、ML
  - 命令式语言：Python、Ruby、Java、C#

垃圾收集

- 运行时系统安全时自动回收对象
- 自动内存管理技术，用户无需关心内存释放
- 使用垃圾回收的语言
  - 函数式编程语言：Haskell、ML
  - 命令式语言：Python、Ruby、C/C++、Java、C#

垃圾收集

- 问题：如何找出可以回收的对象？
  - 第0步：停止程序运行（Stop the world）
  - 第1步：找出寄存器中和运行时栈里面的对象引用集合（Root set）
  - 第2步：根据Root set找到它们引用其他的对象
  - 第3步：不断迭代，直到找不到新对象
  - 第4步：从Root Set出发不可达的对象就是可以回收的对象

垃圾收集

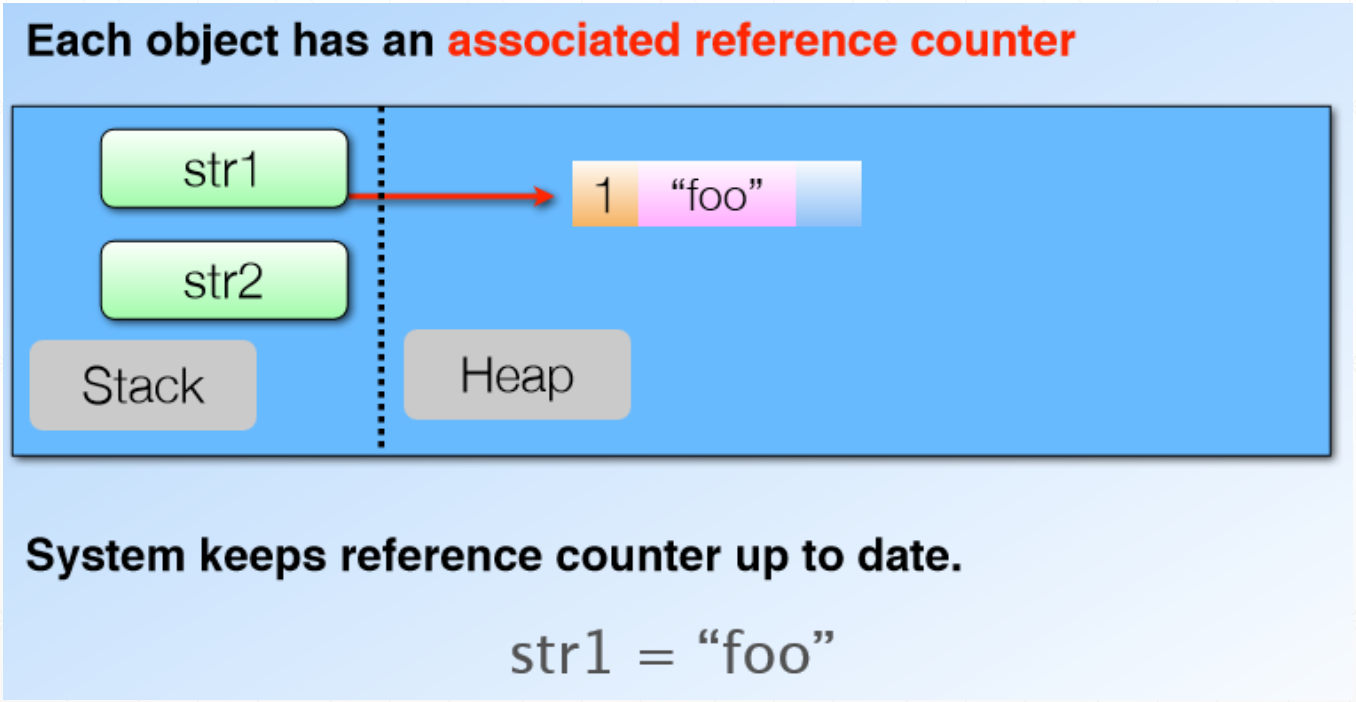
- 主要算法
  - 引用计数算法
  - 拷贝算法
  - 标记清除算法
  - 按代垃圾收集算法

垃圾收集：引用计数算法

- 基本思想
  - 每个对象设置一个引用计数器
  - 每增加一次引用时计数器加1
  - 每减少一次引用时计数器减1
  - 从Root set可达的对象引用计数>0
  - 引用计数=0时可以回收
- 特点
  - 实现简单，在很多项目中得到应用

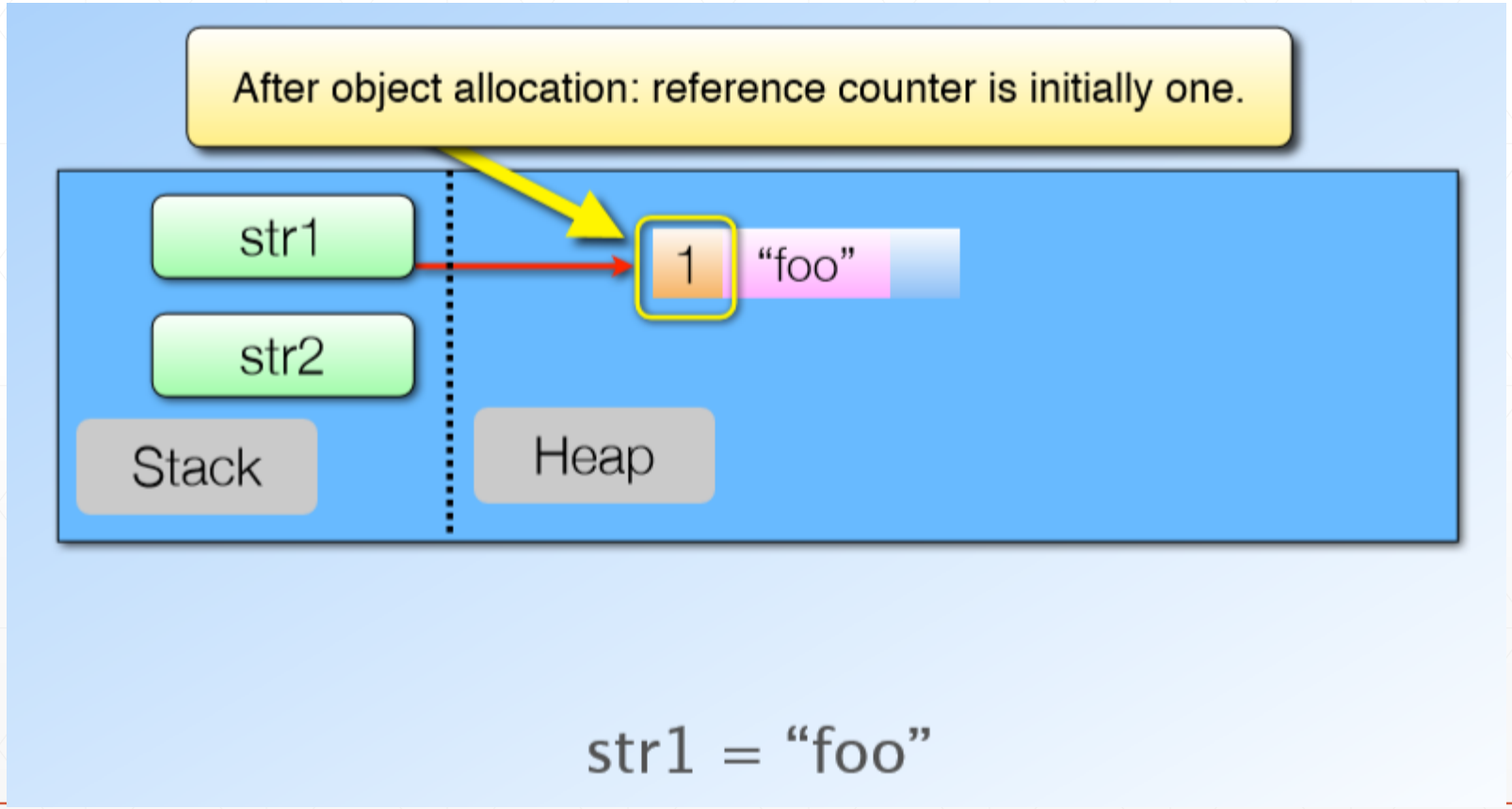
垃圾收集：引用计数算法

- 具体例子



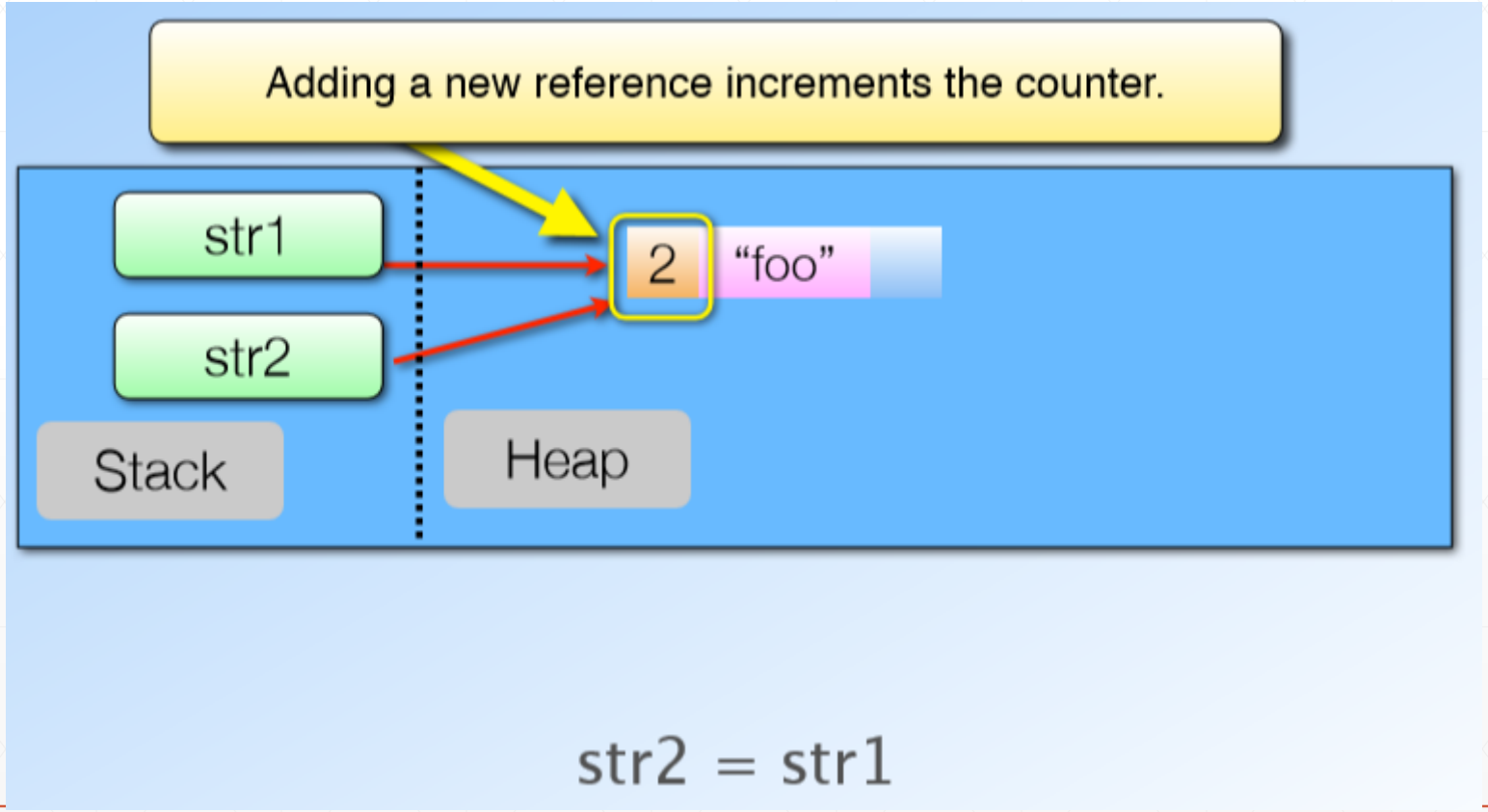
垃圾收集：引用计数算法

- 具体例子



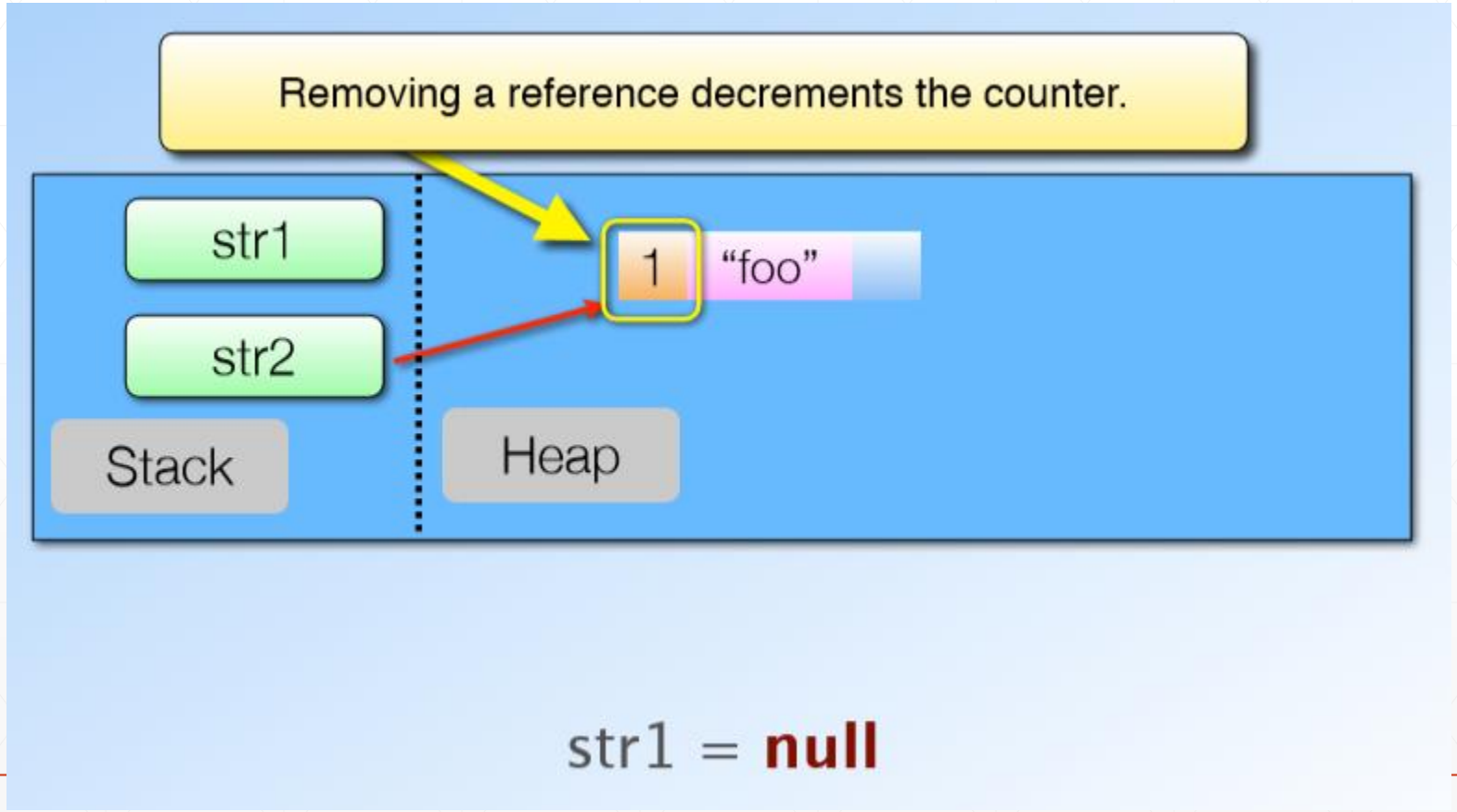
垃圾收集：引用计数算法

- 具体例子



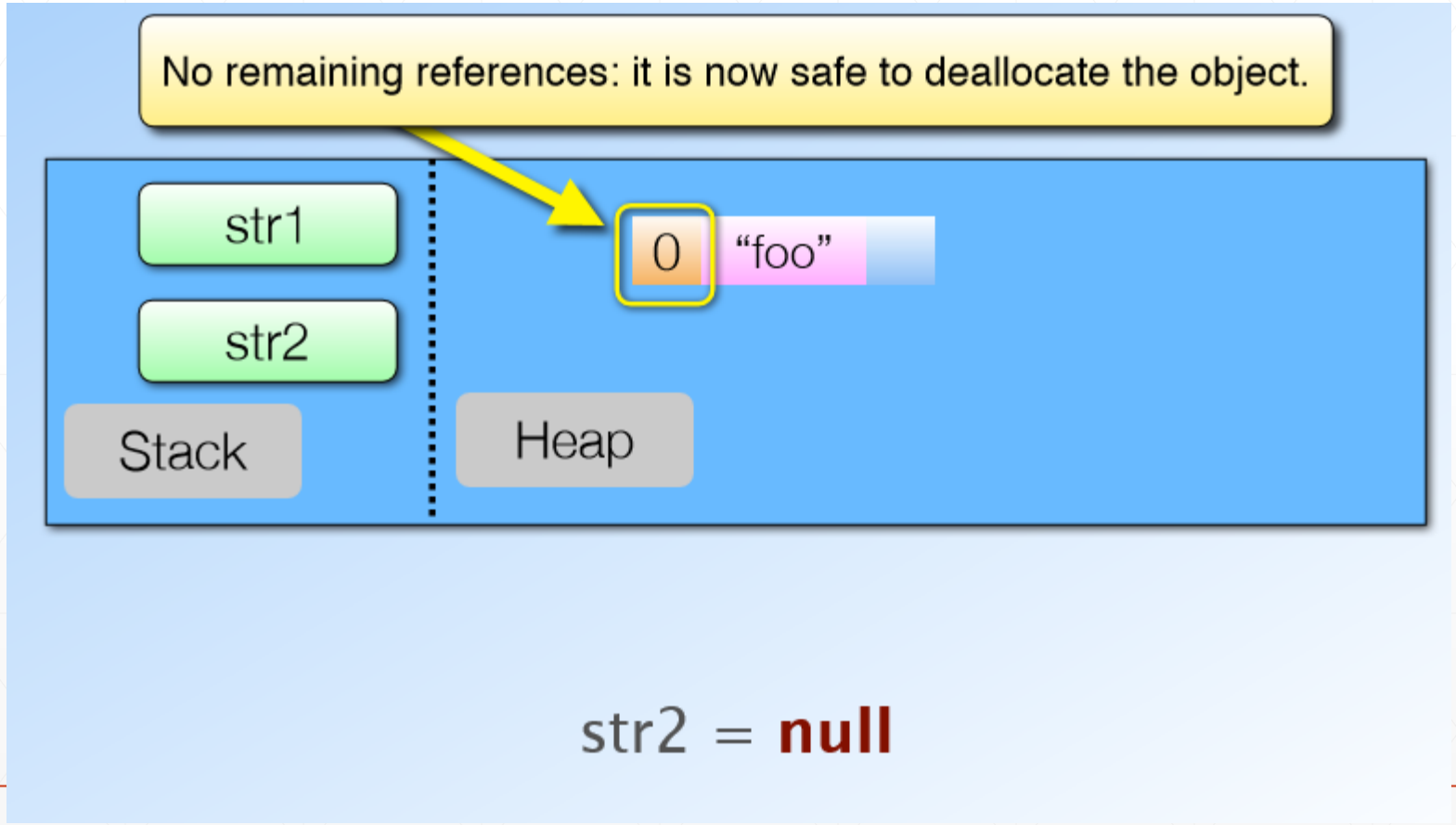
垃圾收集：引用计数算法

- 具体例子



垃圾收集：引用计数算法

- 具体例子

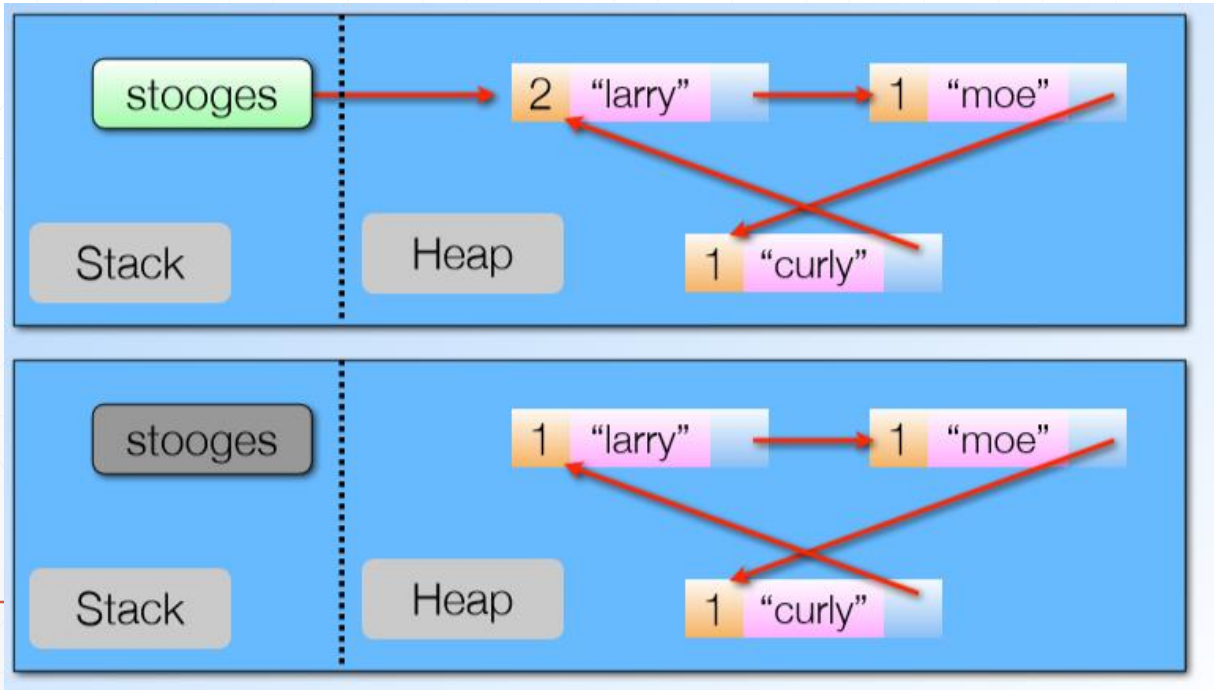


垃圾收集：引用计数算法

- 存在的主要问题
  - 每次引用操作都会需要更新计数器，程序性能受影响
  - 多线程环境下，引用计数器更新需要原子性操作保护
  - Union类型，怎么区分整型和指针？
  - 循环引用问题

垃圾收集：引用计数算法

- 循环引用问题
  - 不可达，但是引用计数不是0
  - 造成内存泄露，需要额外算法检测



垃圾收集：标记清除/Mark & Sweep算法

- 基本思想
  - 为每个对象设置一个标记flag
  - 清除所有对象的标记flag
  - 从Root set出发遍历所有可达对象
  - 对可达对象进行标记（Mark）
  - 没有被标记的对象就是垃圾
  - 回收所有没有被标记的对象（Sweep）
  - 内存空间耗尽时触发

垃圾收集：标记清除/Mark & Sweep算法

- 主要挑战
  - 停止程序运行：Stop the world
  - 防止遍历对象的过程中对象引用关系被修改
  - 内存空间较大时，用户界面无法响应
  - 算法运行时本身需要空间
- 解决办法
  - 并发垃圾收集，垃圾收集与程序交替运行，期间监控引用操作
  - 增量垃圾收集，更加复杂

垃圾收集：标记清除和引用计数比较

标记清除	引用计数
• Stop the world • 其余时间运行比较高效 • 精确算法，不会导致内存泄露 • 实现复杂	• 应用无停顿 • 引用写操作比较慢 • 存在环路引用的问题 • 实现简单

垃圾收集：拷贝算法

- 找到无用对象并释放掉并不能解决内存碎片问题

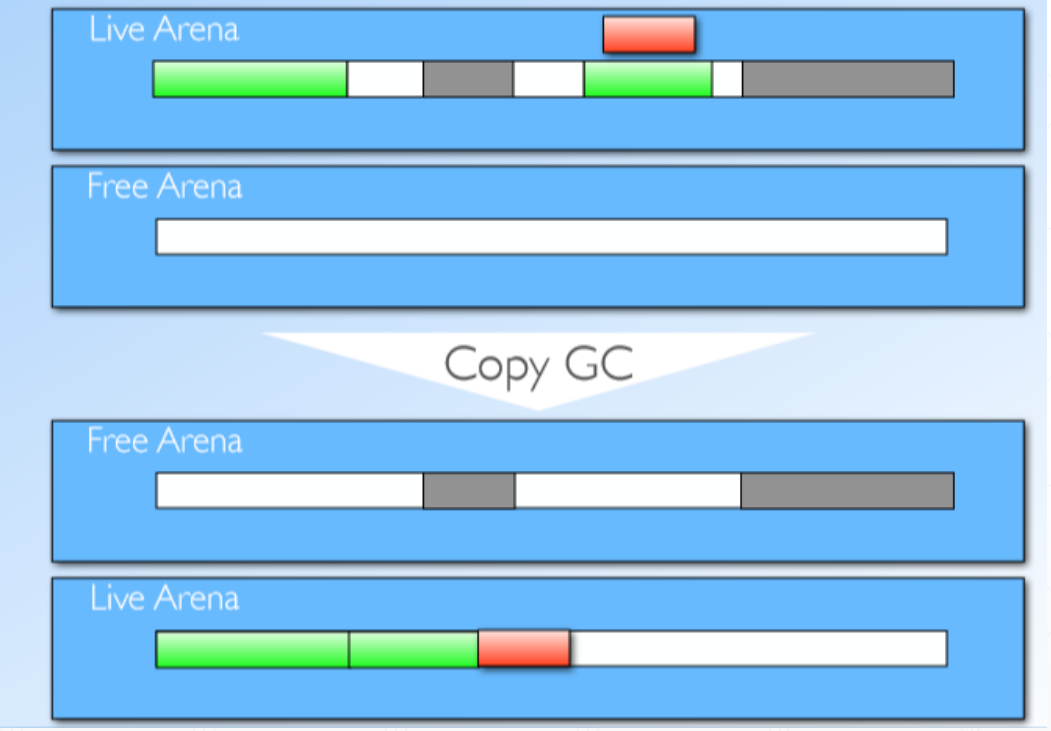
- 基本想法
  - 将内存空间分为两个半区A和B
  - 首先在A半区内紧挨着创建对象
  - A半区满时将存活对象拷贝到B半区
  - 从B半区空闲位置开始创建对象
  - B半区满时重复以上步骤

垃圾收集：拷贝算法

- 某时刻的内存占用情况
  - 绿色表示存活对象，灰色表示无用对象

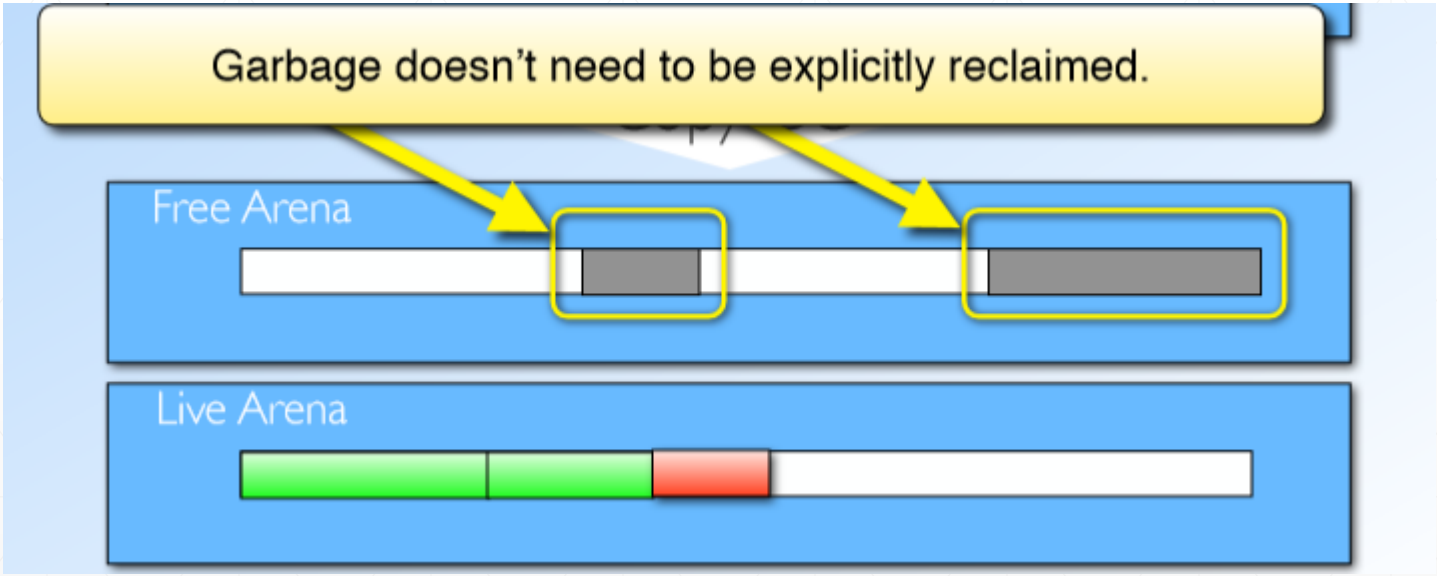
垃圾收集：拷贝算法

- 具体过程
  - 从Live区拷贝存活对象到Free区
  - 交换Live和Free
  - 在新的Live区创建新对象



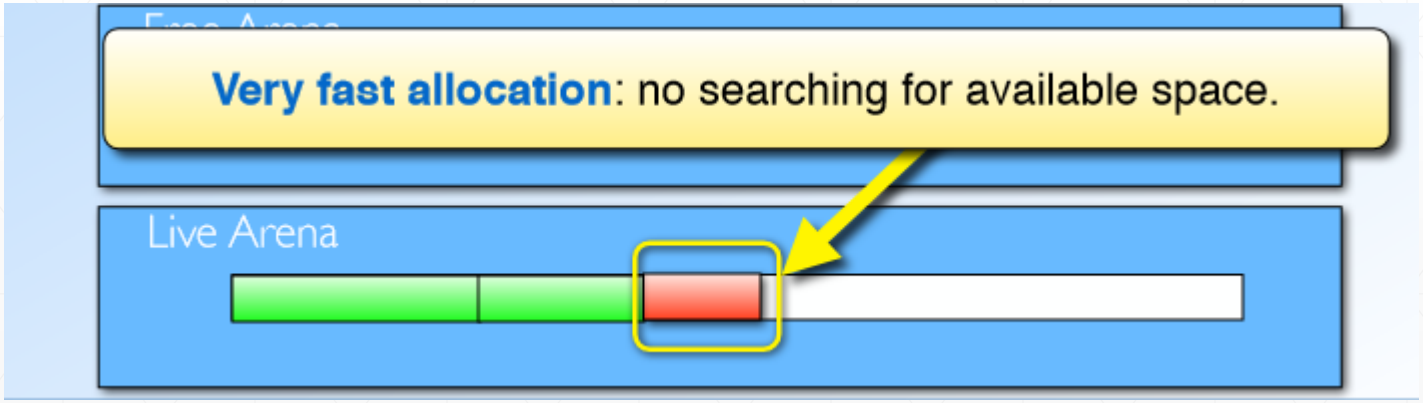
垃圾收集：拷贝算法

- 优点
  - 无需显式回收无用对象，自然被抛弃



垃圾收集：拷贝算法

- 优点
  - 分配空间非常快，地址做加法
- 缺点
  - 任何时刻只有一半空间可以用

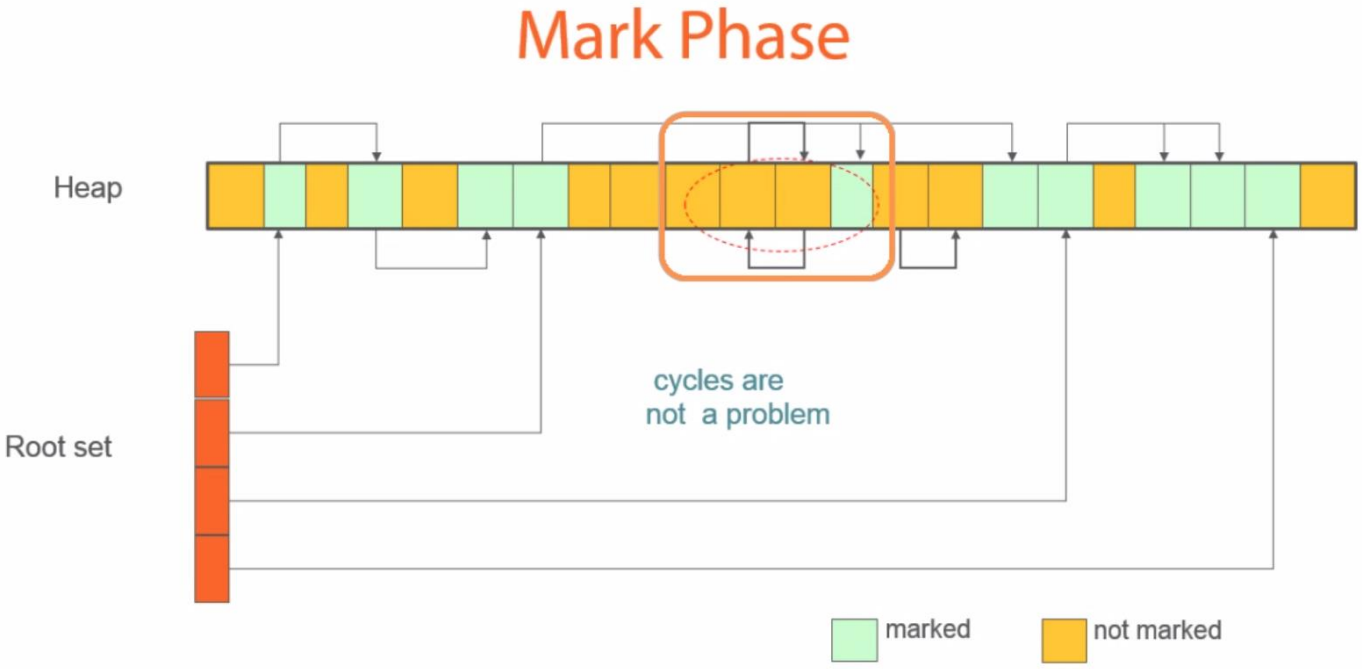


垃圾收集：标记压缩

- 主要思想
  - 从Root set开始标记
  - 存活对象拷贝集中移动到一端
  - 无用对象自然抛弃
  - 从新的空闲空间开始位置创建对象
- 优点：整个存储空间可用
- 缺点：每次垃圾收集耗时较长

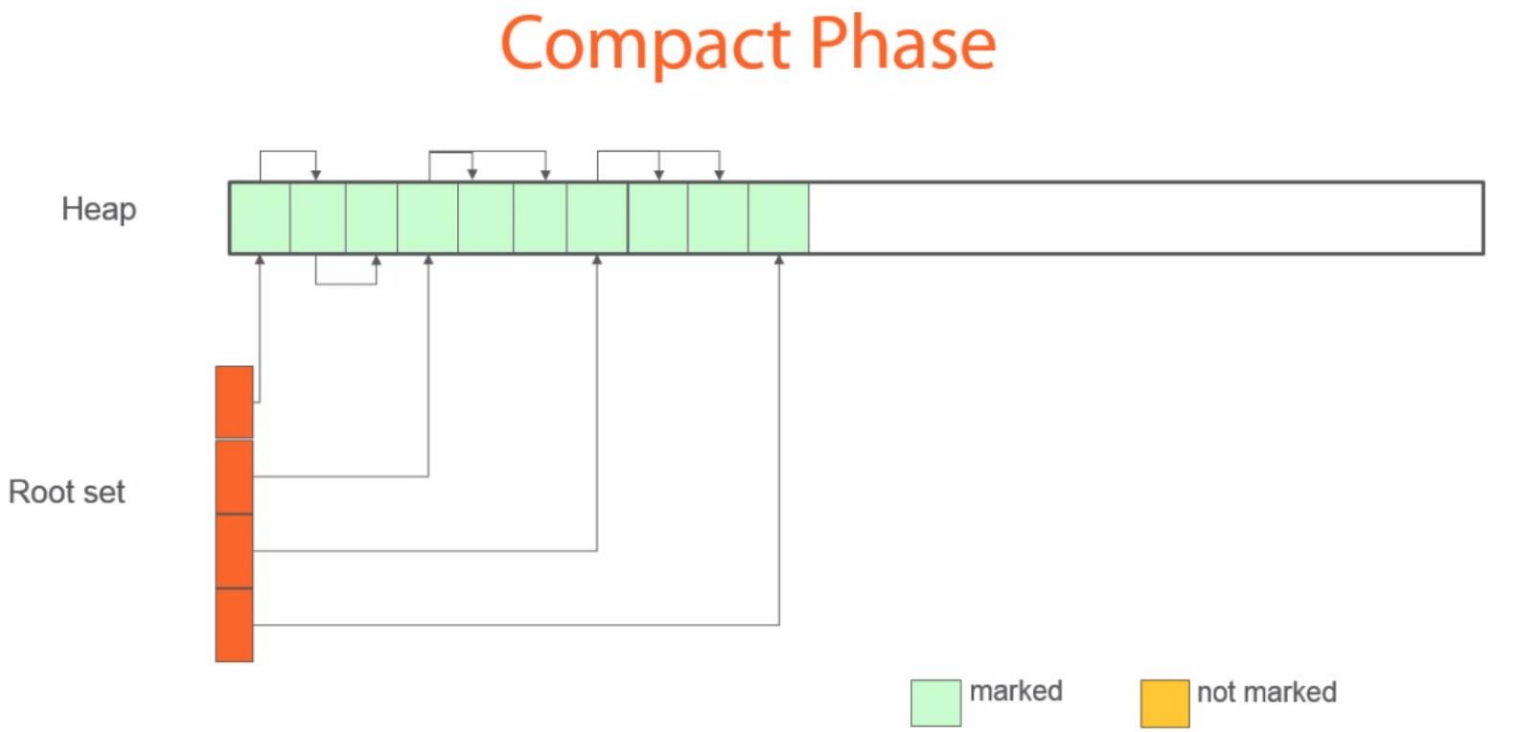
垃圾收集：标记压缩

- 标记阶段



垃圾收集：标记压缩

- 标记阶段



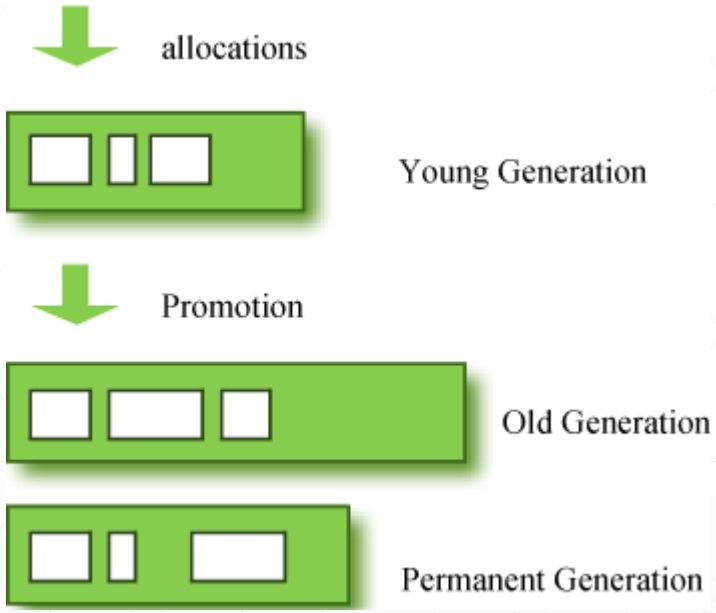
垃圾收集：标记清除与标记压缩

垃圾收集：按代垃圾收集

- 基于如下的观察结果
  - 大部分对象的生命周期比较短
  - 大部分对象在创建不久就变成无用的了
  - 存活时间越久的对象越不可能是垃圾
- 基本思路
  - 将内存划分为不同区域：新生代和旧生代
  - 新生代存活对象拷贝到旧生代
  - 新生代垃圾收集频率较高，旧生代较低
  - 大部分虚拟机采用该算法（JVM）

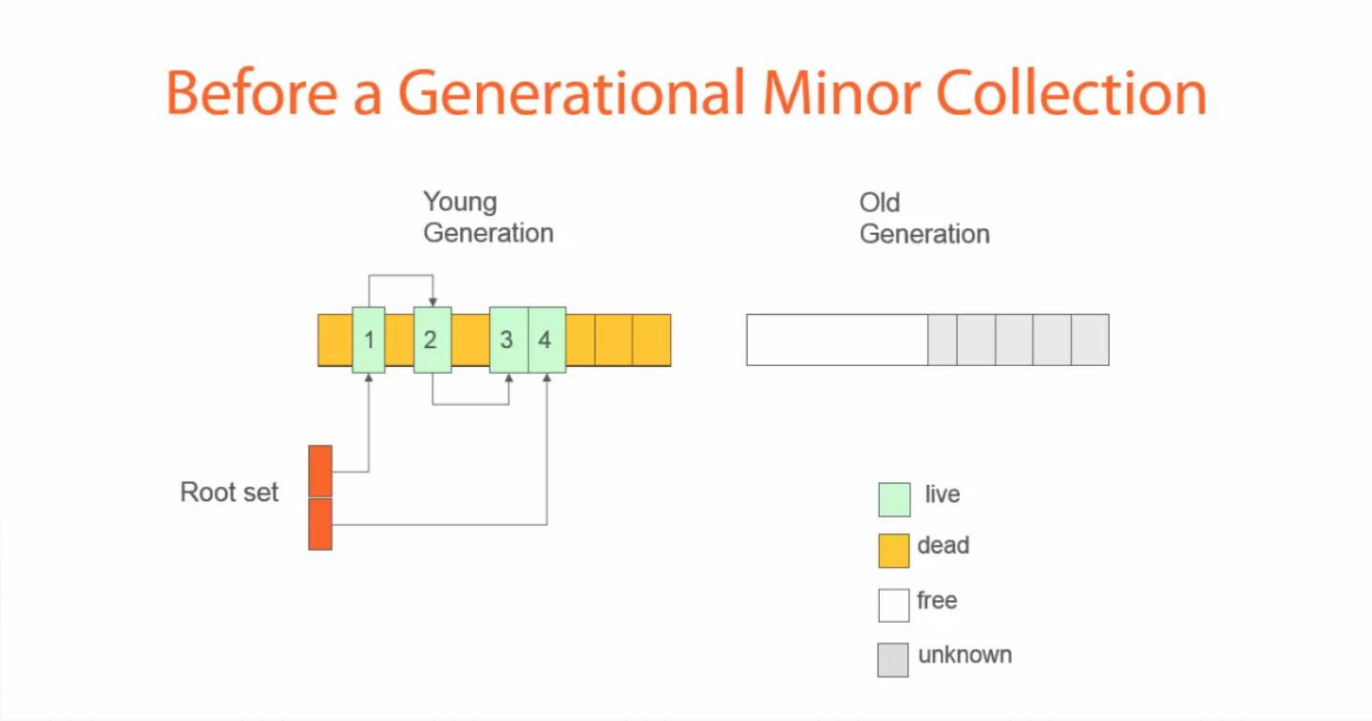
垃圾收集：按代垃圾收集

- 对象首先分配在新生代
- 经过新生代区域垃圾收集后存活下来的进入救生代
- 永久存储区存放代码（类）



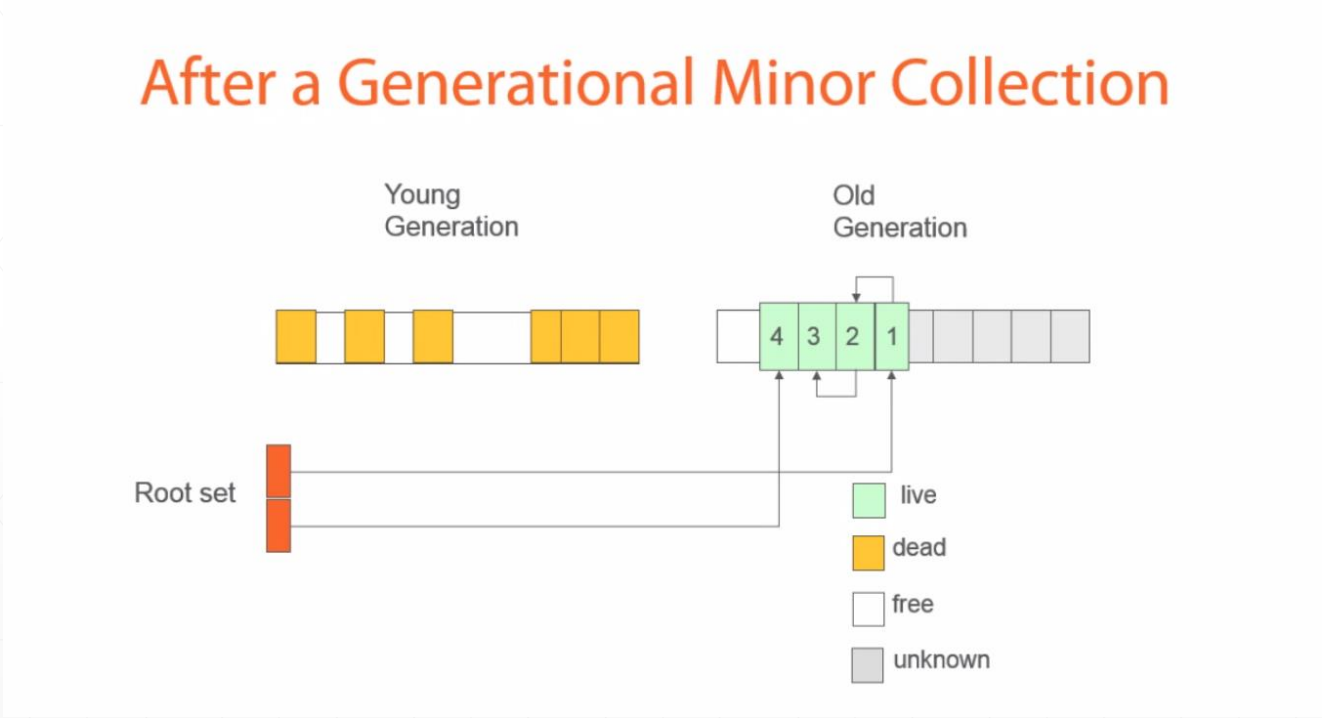
垃圾收集：按代垃圾收集

- 例子



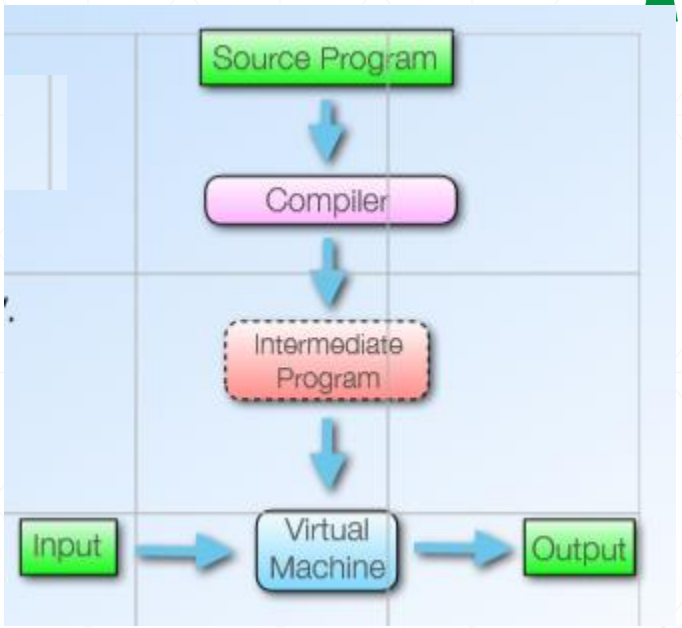
垃圾收集：按代垃圾收集

- 新生代收集后



JIT

- 编译执行与解释执行
- Java执行过程
  - 解释执行效率不高
  - 编译+解释+JIT
- JIT: Just-in-time compilation
- AOT: Ahead-of-time compilation



JIT

- 为什么不在执行前全部编译为机器代码？
  - 显著的启动延迟
  - 解释与JIT并发进行
  - JIT按需进行，每次一小块
  - 平衡：编译时间与运行时间
  - 根据运行时profiling信息决定