北京大学计算机学院

2026 年春季学期　　《编译原理(实验班)》

第十四讲　拾遗

Miscellaneous Topics

主要内容

- 运行时环境：非局部数据访问
- 运行时环境：垃圾回收
- 目标代码生成：针对 SSA 的寄存器分配
- 代码优化：基于路径的数据流分析

主要内容

- **运行时环境：非局部数据访问**
- 运行时环境：垃圾回收
- 目标代码生成：针对 SSA 的寄存器分配
- 代码优化：基于路径的数据流分析

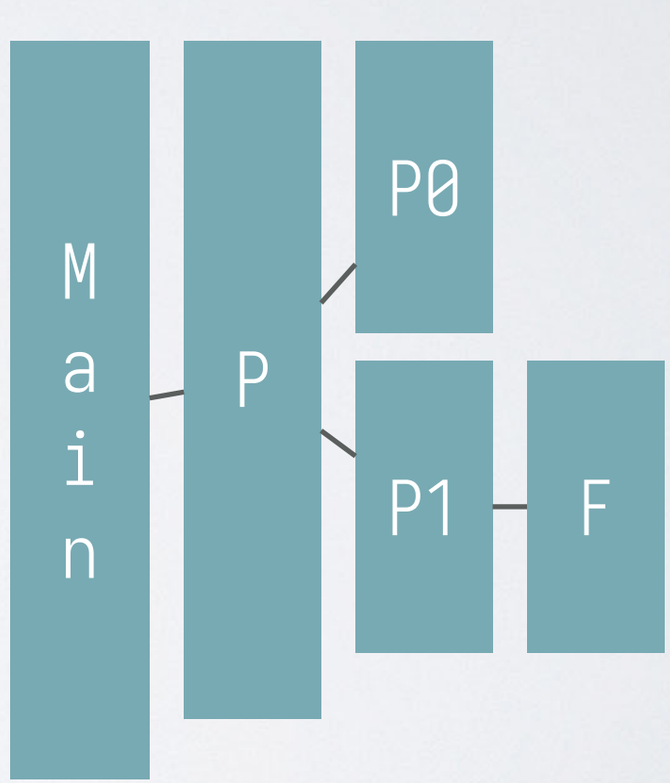
非局部数据的访问

- **过程嵌套**：在过程/函数内部定义过程/函数
  - C、C++、Java 都不允许过程嵌套
  - 新版本的 C++ 和 Java 支持将函数视为「头等(first-class)对象」
- **非局部数据的访问**：一个过程访问在另一个过程中声明的局部变量
- **静态作用域**，也称词法(lexical)作用域
  - 非局部名字的绑定在过程**被定义时**决定
- **动态作用域**
  - 非局部名字的绑定在过程**被调用时**决定

静态作用域

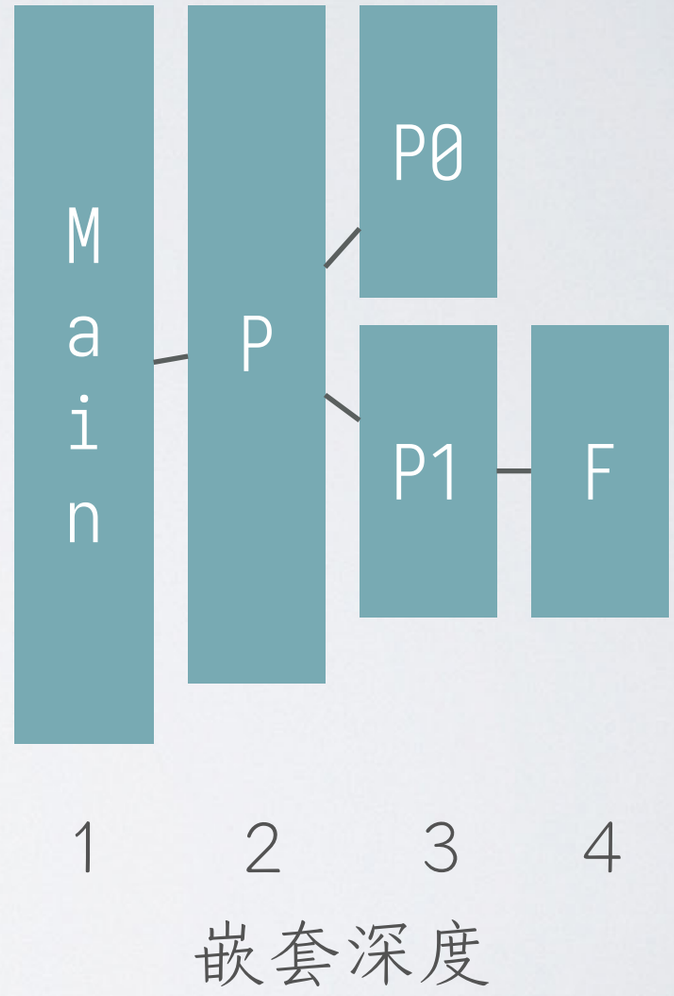
- 过程/函数可以访问包含该过程/函数的所有**外层过程/函数**声明的局部变量和形式参数
  - 允许过程/函数嵌套的语言：Pascal、Lisp、OCaml等

```
procedure P(px: integer, py: integer);
  var i: integer; r: integer;
  procedure P0(p0x: integer);
    begin … end;
  procedure P1(p1x: integer, p1y: integer);
    var j: integer;
    function F(fx: integer): integer;
      var x: integer;
      begin … P0(x) … p1x … px + j … end;
    begin … F(p1y) … F(r) … end;
  begin … P1(…, …) … end;
begin
  …
end.
```

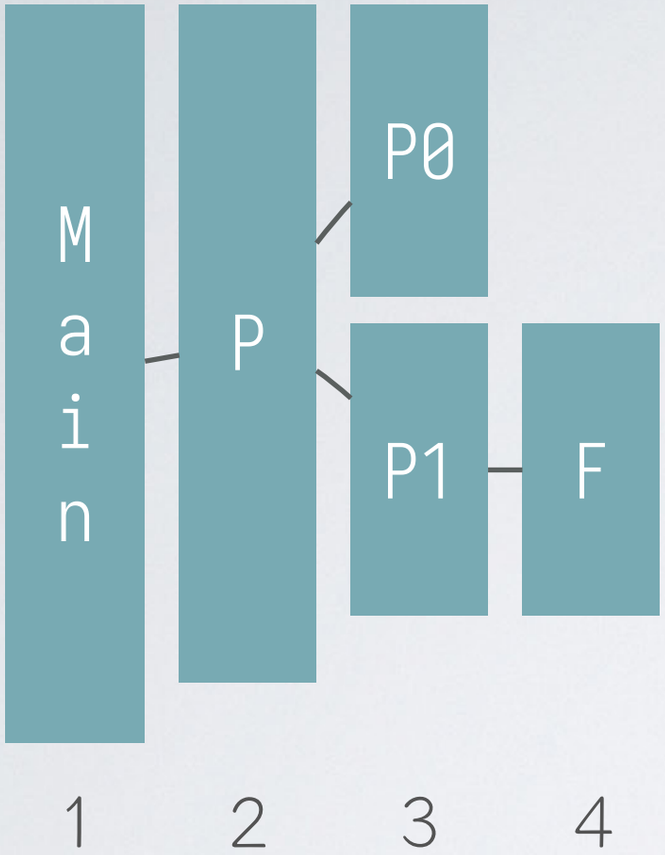


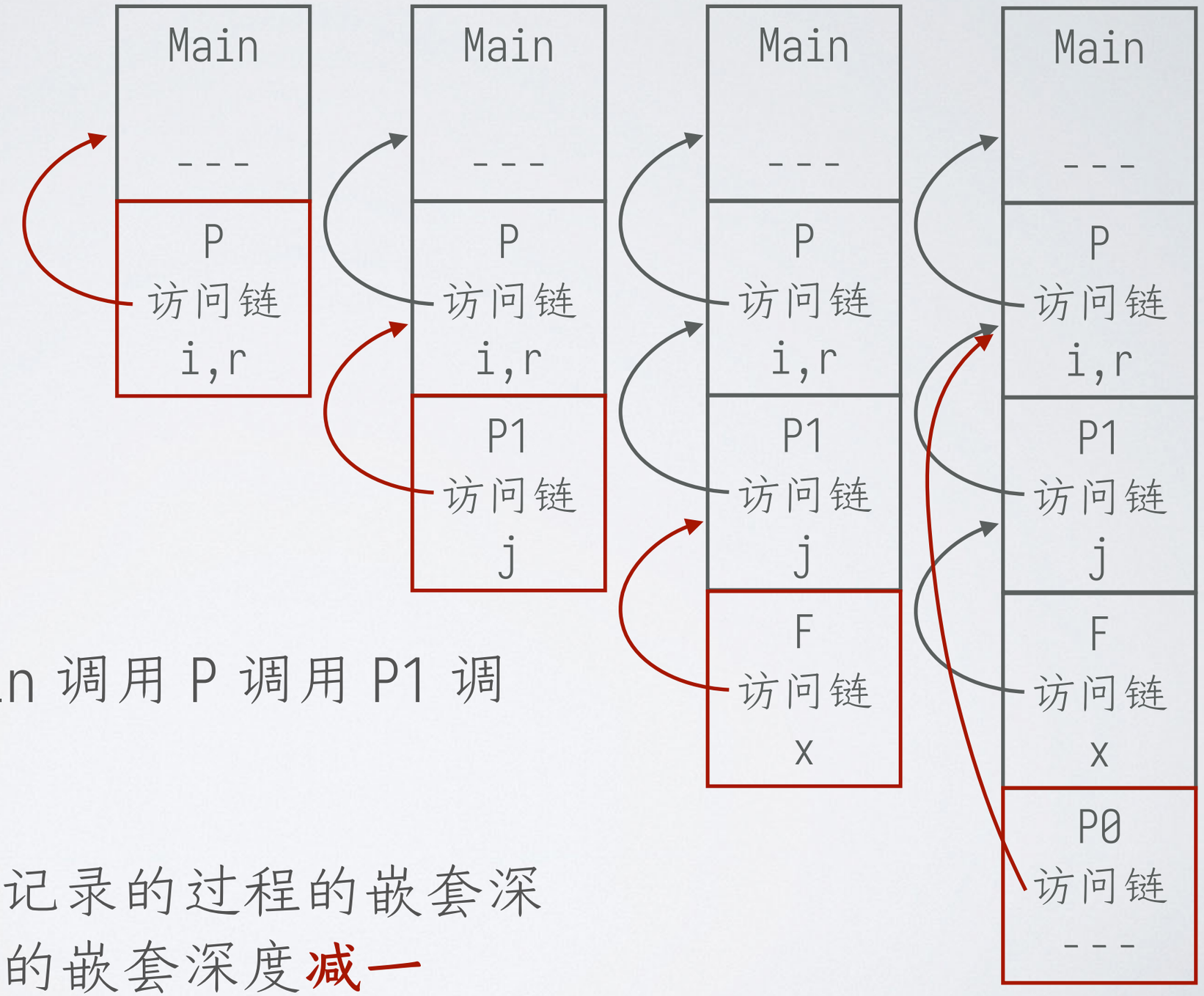
如何支持静态作用域？

- **问题：无法静态确定两个嵌套过程活动记录的相对位置**
- 方法一：**访问链法**
- **嵌套深度：**如果过程 p 在一个嵌套深度为 i 的过程中定义，那么 p 的嵌套深度为 $i+1$
- 如果过程 p 直接嵌套定义在 q 中，那么 p 的一个活动的访问链指向**最近**的 q 的活动
- 沿着访问链可以找到所有可以被当前过程访问的活动
  - 沿访问链走一步可以使嵌套深度恰好少一



访问链示例





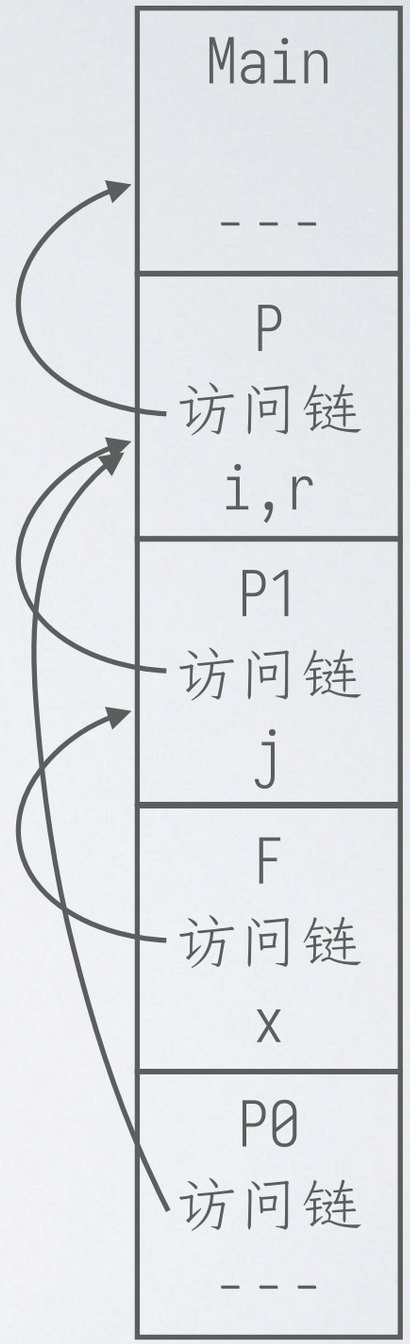
- 考虑过程活动：Main 调用 P 调用 P1 调用 F 调用 P0
- 访问链指向的活动记录的过程的嵌套深度一定是当前过程的嵌套深度**减一**

访问链的建立

- 如果嵌套深度为 m 的过程 q 调用嵌套深度为 n 的过程 p:
  - $m < n$:
    - p 直接声明在 q 中，也就是说 $m + 1 = n$
    - 将 p 的访问链指向 q 的活动记录
  - $m \geq n$:
    - q 和 p 的嵌套深度从 1 到 $n - 1$ 的外围过程是相同的
    - 追踪 q 的访问链 $m - n + 1$ 步，到达直接包含 p 的过程 r 的最近的活动记录
    - 将 p 的访问链指向这个 r 的活动记录

通过访问链访问非局部数据

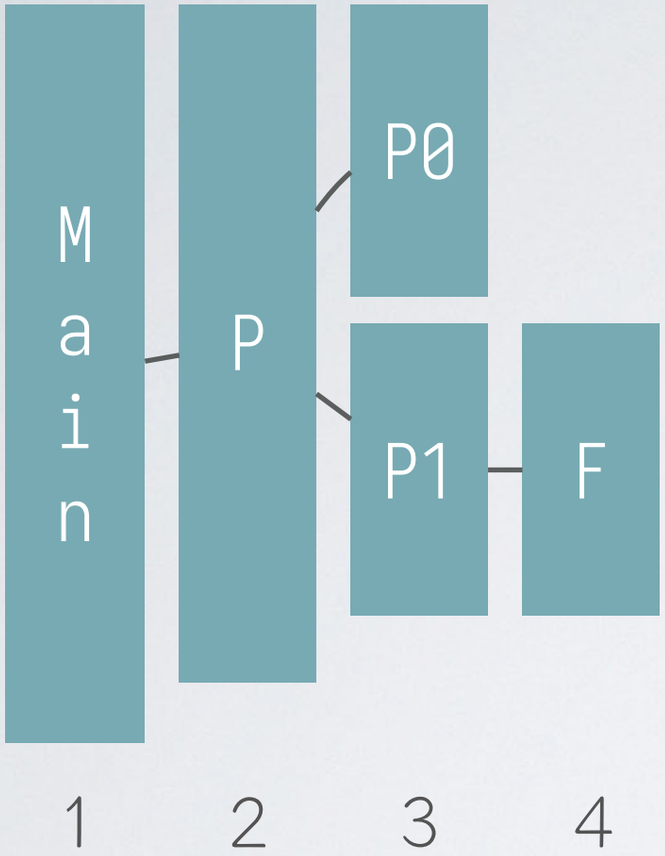
- 从嵌套深度 m 的过程访问嵌套深度 n 的过程中的数据($n \leq m$):
  - 追踪访问链 $m-n$ 次，到达目标活动记录
  - $m-n$ 的值可以在编译时确定
- **以右图为例:**
  - 对于过程 F 而言:
    - 访问 x: 追踪 0 次
    - 访问 j: 追踪 1 次
    - 访问 i 和 r: 追踪 2 次
  - 对于过程 P0 而言:
    - 访问 i 和 r: 追踪 1 次
    - **不能**访问 x 或 j

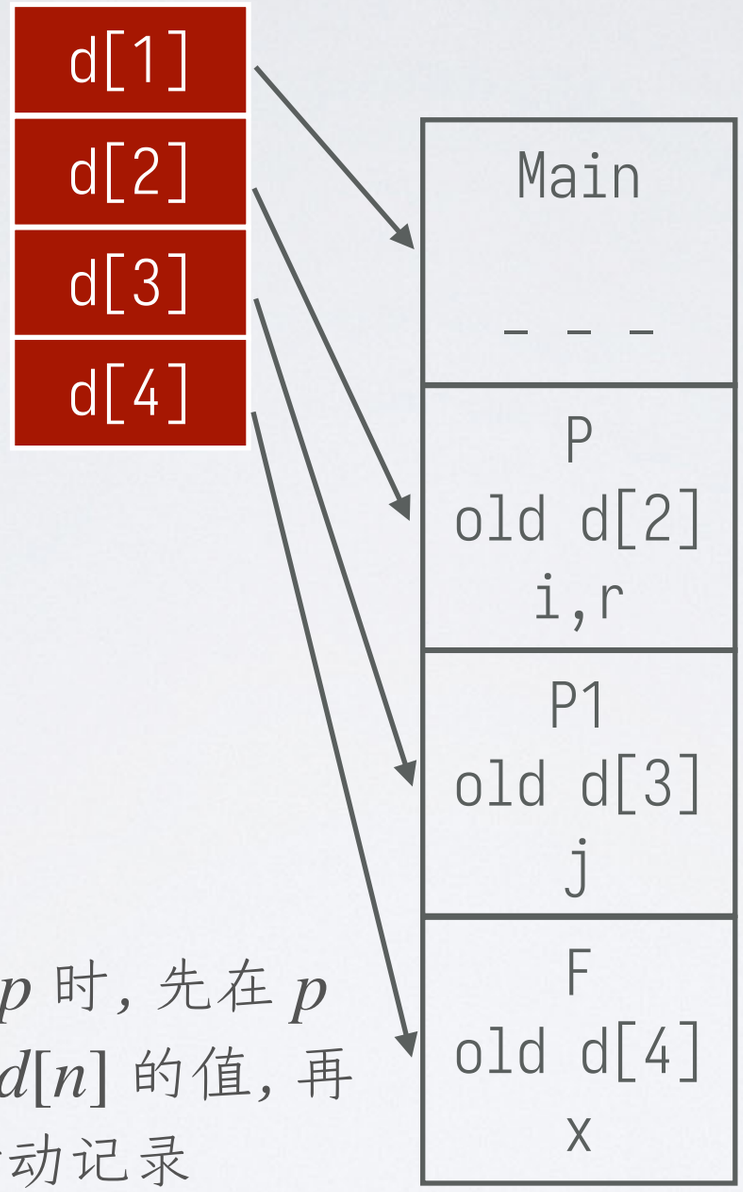


如何支持静态作用域？

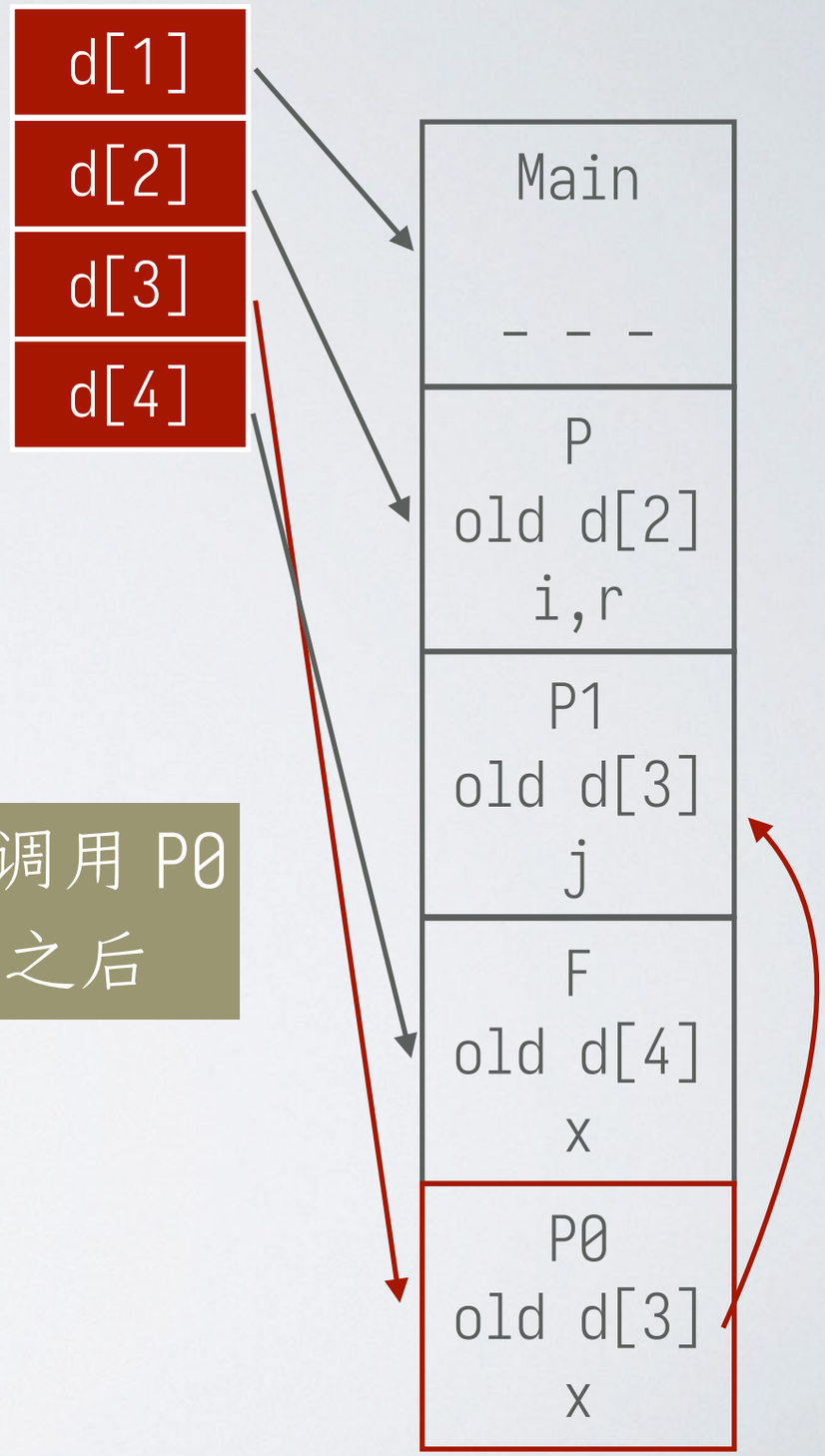
- **问题：用访问链时，访问数据的开销和嵌套深度差有关**
- 方法二：**显示表法**
- **显示表(display)**：运行时环境维护一个数组 d，为每个嵌套深度记录一个指针
  - 指针 $d[i]$ 指向**最近**的嵌套深度为 i 的活动记录
  - 如果过程 p 在运行中访问嵌套深度为 i **(静态可确定)**的过程 q 的数据，则可以通过 $d[i]$ 找到 q 的活动记录
  - 使用显示表可以提高效率，访问开销是常数

显示表示例





F 调用 P0 之后



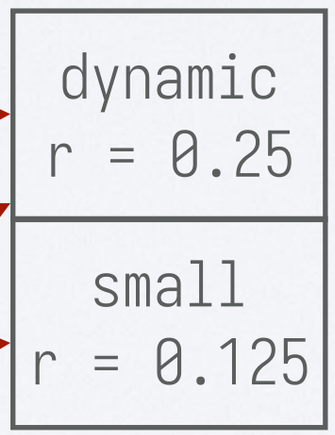
- 调用嵌套深度为 n 的过程 p 时，先在 p 的活动记录中保存原有的 $d[n]$ 的值，再将 $d[n]$ 指向调用的 p 的活动记录
- 从过程 p 返回时，通过活动记录中保存的值恢复之前的 $d[n]$

动态作用域

- 被调用者的非局部名字 a 和其**调用者**中使用相同的存储单元
  - 静态无法确定，只能在运行时确定
- 目前只有少数语言采用，比如 Emacs Lisp
- 运行时环境为每个名字维护一个**全局**的作用域栈

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r:5:3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```

r 的作用域栈



- 动态作用域下输出：

```
0.250   0.125
0.250   0.125
```

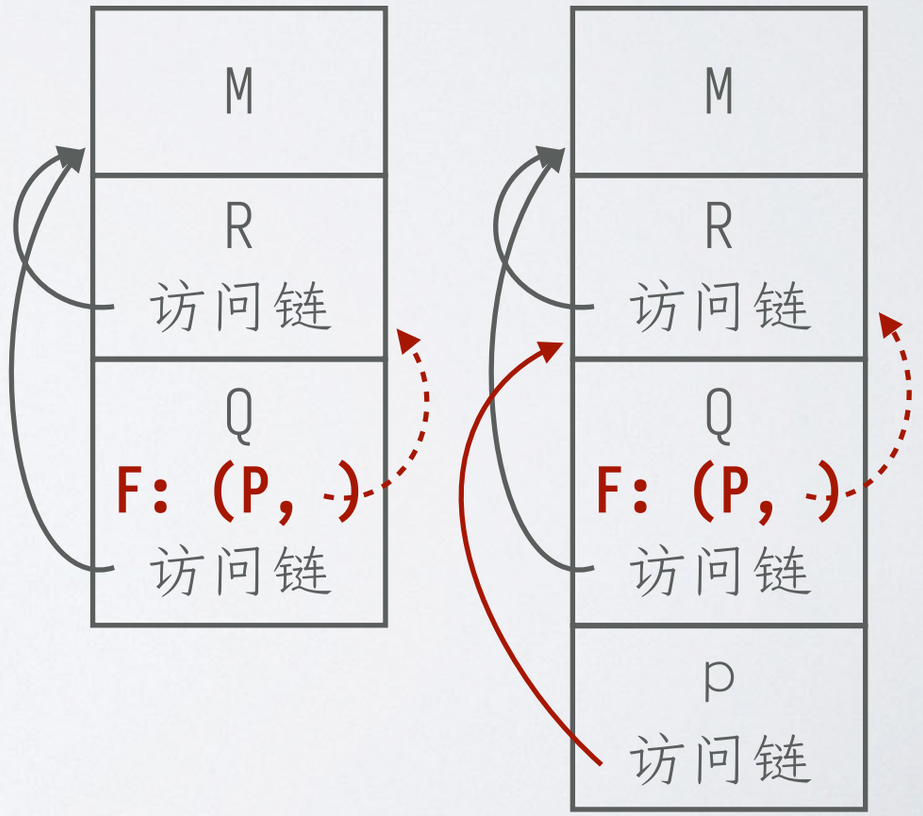
- 静态作用域下输出：

```
0.250   0.250
0.250   0.250
```

过程作为参数传递

- 让我们回到静态作用域
- 当一个过程 p 作为参数传递给另一个过程 q，并且 q 随后调用了这个参数时，有可能 q 并不知道 p 的上下文
- **方法：调用者把 p 作为参数传递时，同时传递其访问链**

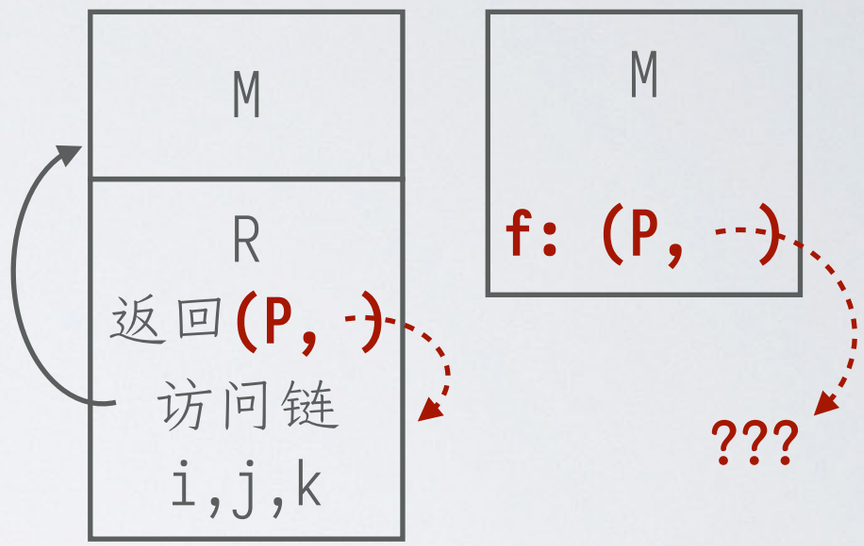
```
procedure M(x: integer);
  procedure Q(F: integer -> integer);
    begin … F(…) … end;
  procedure R(y: integer);
    function P(z: integer): integer;
      begin … end;
    begin … Q(P) … end;
  begin
    … R(…) …
  end;
```



如何支持将过程作为结果返回？

- 问题：栈式管理下，访问链指向的活动记录有可能不在栈中

```
procedure M(x: integer);
  function R(y: integer): integer -> integer;
    var i, j, k: integer;
    function P(z: integer): integer;
      begin … i … j … k … end;
    begin … R := P … end;
begin
  … f := R(…) … f(…) …
end;
```

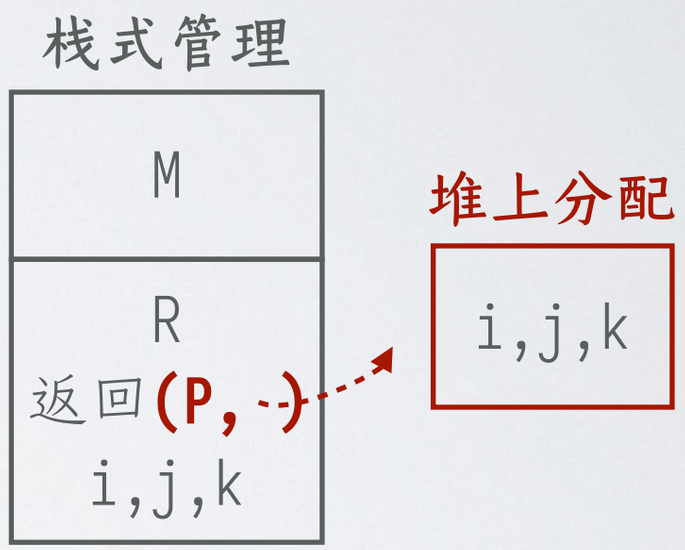


- 栈式活动记录管理不再适合！
- 方法一：完全在堆中分配和管理活动记录
- 方法二：闭包

闭包

- **闭包(closure)**是支持「头等(first-class)函数」的一种技术
- 运行时在堆上分配空间，存储需要的**外层函数**的局部数据

```
procedure M(x: integer);
  function R(y: integer): integer -> integer;
    var i, j, k: integer;
    function P(z: integer): integer;
      begin … i … j … k … end;
    begin … R := P … end;
begin
  … f := R(…) … f(…) …
end;
```



- Lua 的 **upvalue** 机制
  - 闭包中的每个外层变量指向一个 upvalue
  - upvalue 初始时指向栈中数据，若**逃逸(escape)**则转移到堆上

主要内容

- 运行时环境：非局部数据访问
- **运行时环境：垃圾回收**
- 目标代码生成：针对 SSA 的寄存器分配
- 代码优化：基于路径的数据流分析

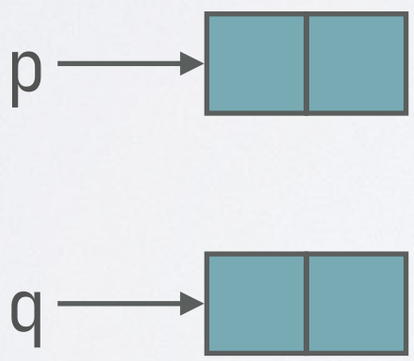
垃圾回收

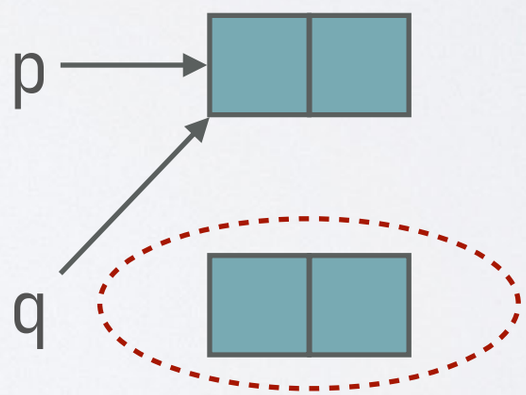
- 垃圾(garbage)
  - 狭义：不能被访问(不可达)的数据
  - 广义：不需要再被使用的数据
- 垃圾回收(garbage collection)
  - 自动回收不可达数据的机制，降低程序员的负担
- 使用垃圾回收的语言：
  - 最早出现在 1958 年 Lisp 语言的实现中
  - OCaml、Haskell、Java、Python、Go、JavaScript、Swift 等

垃圾示例

```
class node {
  int value;
  node next;
}
```

```
node p, q;
p = new node();
q = new node();
q = p;
```





垃圾回收器的设计目标

- 基本要求：**类型安全**
  - 保证回收器能够知道数据元素**是否为**一个指向某内存块的**指针**
  - 类型不安全的语言(比如 C 和 C++)不适合使用垃圾回收
    - 但要用也不是不行

- **性能度量：**
  - **总体运行时间：**不显著增加应用程序的总运行时间
  - **空间使用：**最大限度地利用可用内存
  - **停顿时间：**当垃圾回收器启动时，可能引起应用程序的停顿，应当使得这个停顿尽量短
  - **程序局部性：**改善应用程序的时间局部性和空间局部性

核心概念：可达性

- **可达性**指的是一个对象可以被应用程序访问到
- **根集(root set)**：不需要指针解引用就可以直接访问的数据
  - Java：静态字段成员、栈中变量
- **可达性(reachability)**的定义：
  - 根集中的成员指向的对象都是可达的
  - 对于任意一个对象，如果指向它的一个指针被保存在可达对象的某字段中，那么这个对象也是可达的
- **性质：一个对象一旦变得不可达，它就不会再变成可达的**

改变可达对象集合的操作

- **对象分配:** 返回一个指向新存储块的指针
- **参数传递/返回值:** 对象指针从实在参数传递到形式参数/从返回值传递给调用者
- **引用赋值:** 对于指针 u 和 v, 赋值 $u = v$ 将 u 指向 v 指向的对象, 可能使得 u 原来指向的对象变得不可达, 并递归得使得更多对象不可达
- **过程返回:** 活动记录出栈, 局部变量从根集中移除, 可能使得一些对象变得不可达
- **问: 哪些操作可能增加/减小可达对象集合的大小?**

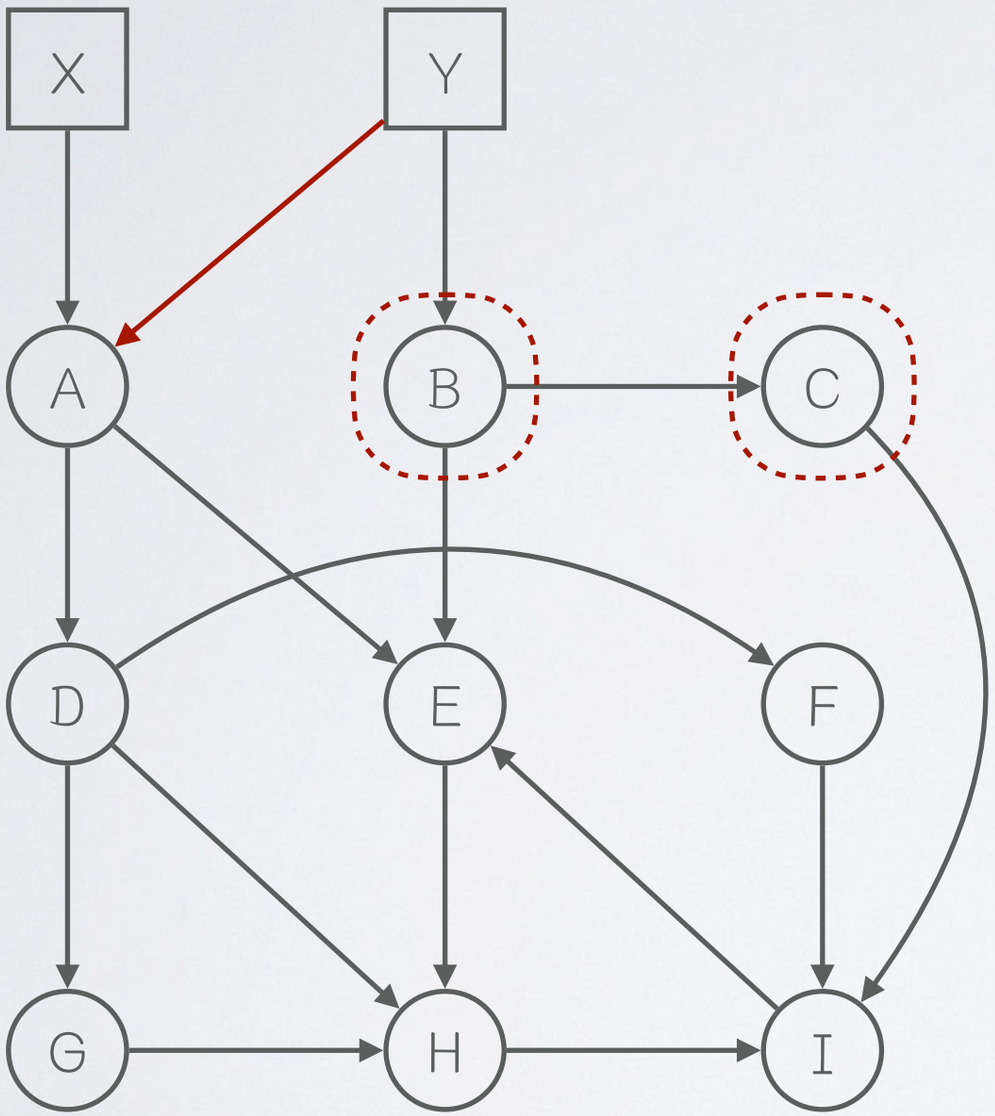
垃圾回收算法

- 基本思想：**寻找不可达的对象**
- 两种基本方法：
  - 跟踪相关操作，捕获对象变得不可达的时刻，回收对象占用的空间
    - 典型例子：**基于引用计数的垃圾回收**
  - 在需要时，标记出所有可达对象，然后回收其它对象
    - 典型例子：**基于跟踪的垃圾回收**

基于引用计数的垃圾回收器

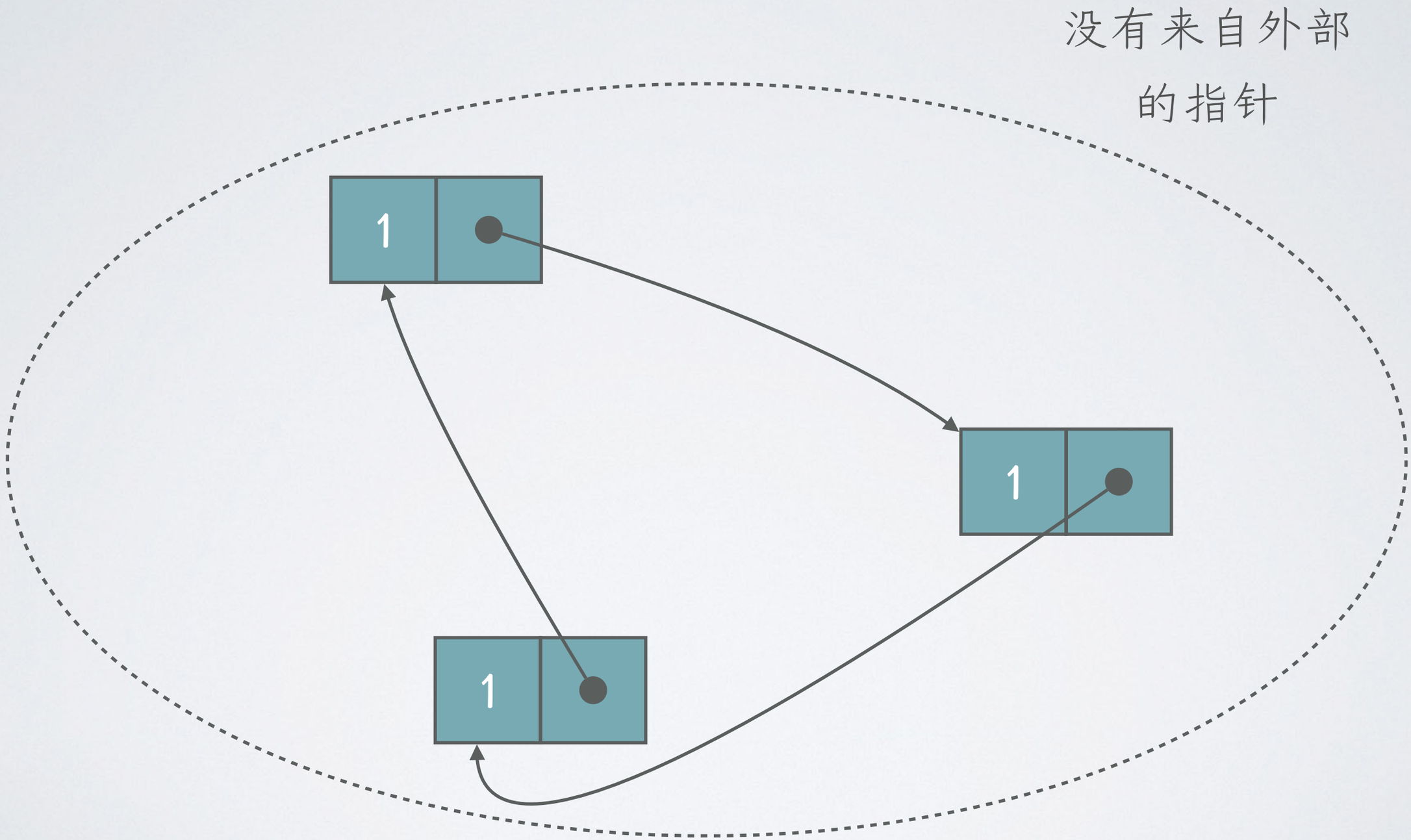
- 每个对象有一个用于存放**引用计数**(reference counting)的字段，并按照如下方式维护：
  - **对象分配**：引用计数设为 1
  - **参数传递**：引用计数加 1
  - **引用赋值**：对于 $u = v$，u 指向的对象引用计数减 1，v 指向的对象引用计数加 1
  - **过程返回**：每个局部变量指向对象的引用计数减 1
  - **问：返回值如何处理？**
- 如果一个对象的引用计数为 0，在删除该对象前，此对象中各个指针所指对象的引用计数减 1
- 使用引用计数的语言：Objective C、Swift 等

引用计数示例



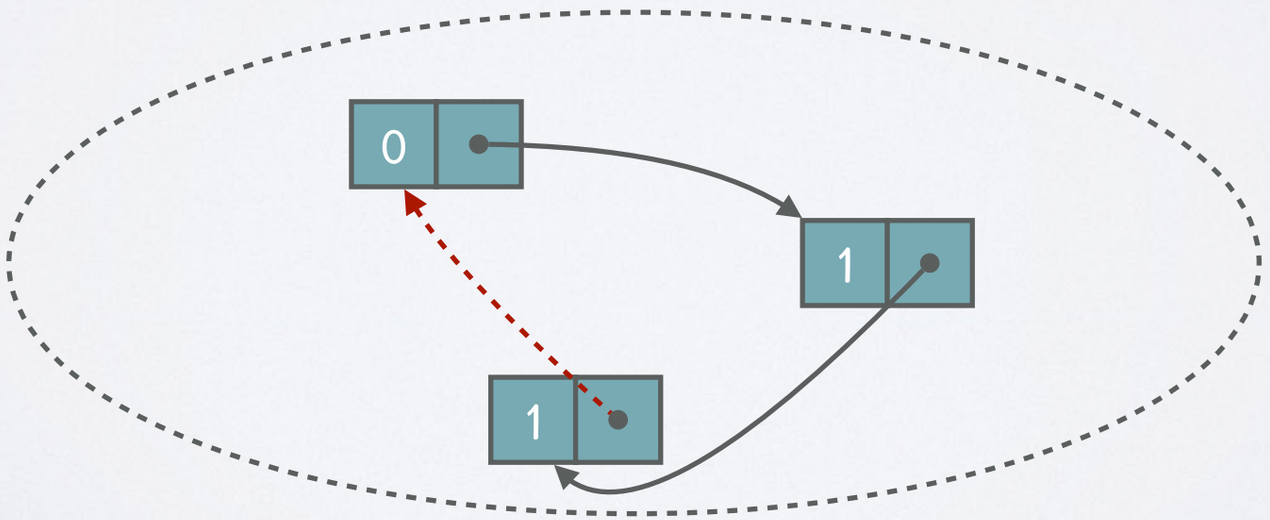
- 考虑赋值 $Y = X$
- 修改计数后总是先考虑是否释放该对象
- 释放一个对象前总是先处理该对象内部的指针

循环垃圾示例



弱引用

- weak reference
- 程序员手动声明一些指针**不影响**引用计数
  - 例如二叉树结点中指向其 parent 的指针
- 可以解决一些循环数据结构的垃圾回收问题
- 使用表达弱引用的指针时须判断其是否有效



小结：引用计数

- **优点：**
  - 以增量方式完成，可以避免长时间停顿
  - 垃圾可以被及时回收
  - 易于实现
  - 可以与其它存储管理机制结合
    - 例如 C++ 中的 shared_ptr<T> 和 Rust 中的 Rc<T>
- **缺点：**
  - 空间代价：每个对象都要保存引用计数
  - 时间代价：每次指针更新都要做多次检查和修改
  - 循环数据结构会造成内存泄漏

基于跟踪的垃圾回收器

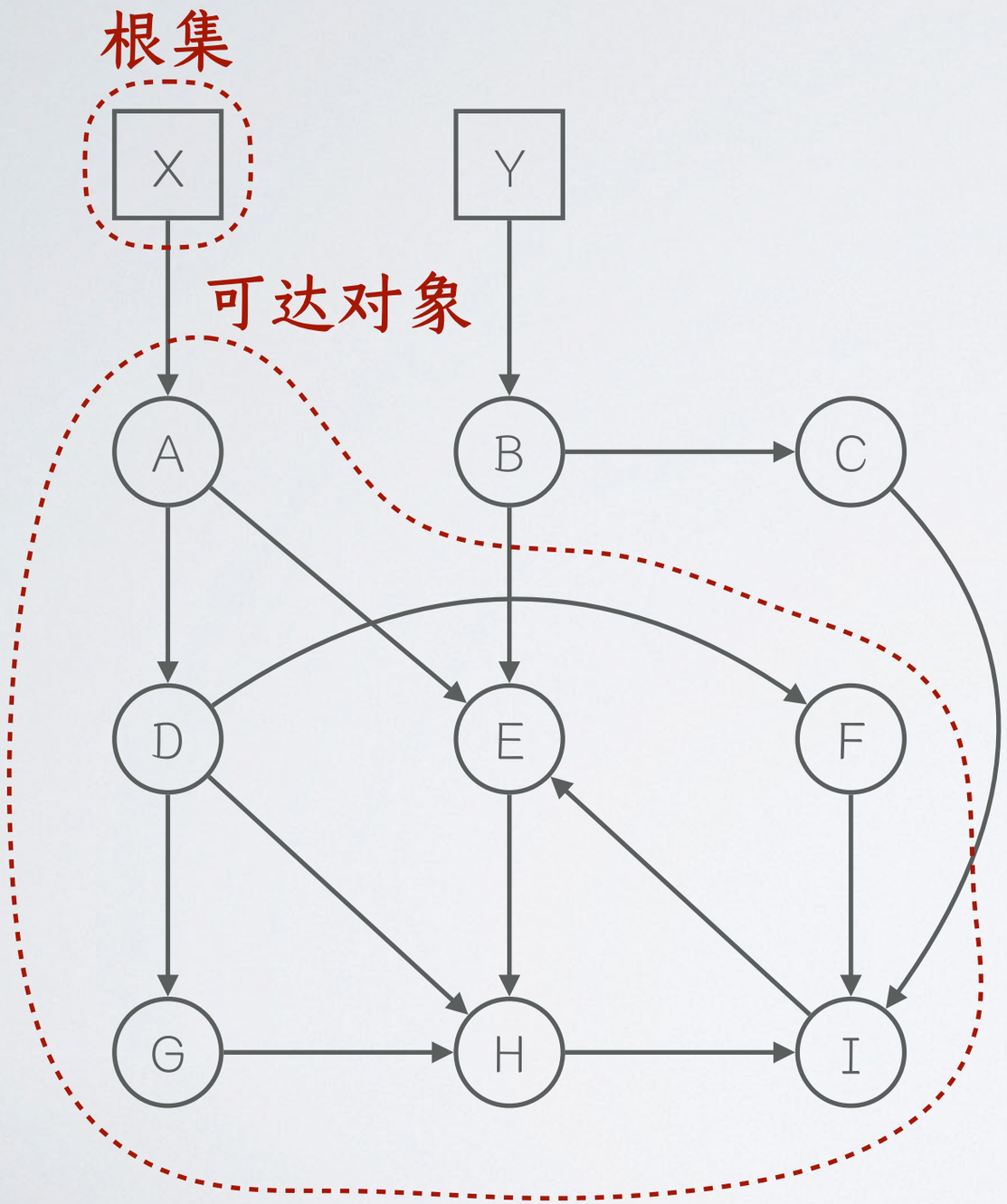
- 以**周期性**的方式运行，在空闲空间耗尽或者低于某个阈值时启动，**寻找不可达对象**并回收其空间

- 标记-清扫式垃圾回收及其优化
- 标记并压缩的垃圾回收
- 拷贝回收
- 世代垃圾回收
- ……

标记-清扫式垃圾回收器

- **mark-and-sweep**
- 一种直接的全面停顿的算法
- 分成两个阶段：
  - **标记**：从根集开始，跟踪并标记出所有可达对象
  - **清扫**：遍历整个堆区，释放不可达对象
- 如果我们把数据对象看作顶点，指向关系看作有向边，那么标记的过程实际上是**从根集开始的图遍历**的过程

标记-清扫示例



- 假设 X 是全局变量，Y 是当前过程活动的局部变量

- 当前过程返回后，进行标记-清扫式垃圾回收
  - A、D、E、F、G、H、I 可达
  - B、C 不可达，从而被释放

标记-清扫垃圾回收算法的优化

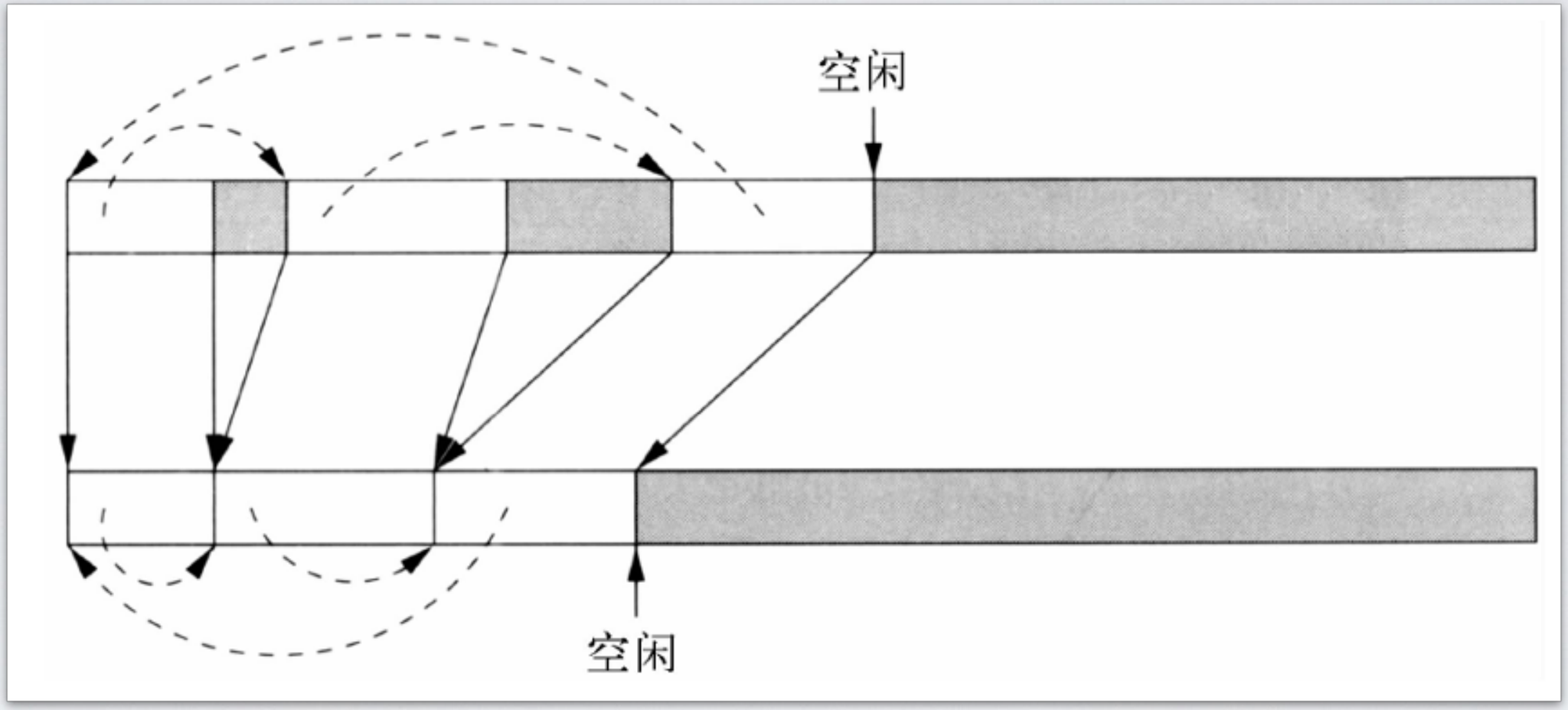
- 基本算法需要扫描整个堆区
- **优化：**
  - 用一个列表记录所有已经分配的对象
  - 不可达对象等于已分配对象去掉可达对象
- **优点：**只需要扫描这个列表就可以完成清扫
- **缺点：**需要维护这个额外的列表

标记并压缩的垃圾回收器

- mark-and-compact
- 对可达对象进行**重定位(relocating)**可以消除存储碎片
  - 把可达对象移动到堆区的一端，另一端则是空闲空间
  - 空闲空间接合成单一块，更容易存储较大的对象
  - 提高应用程序的时间局部性和空间局部性
- 整个过程分成三个步骤：
  - 标记
  - 计算新地址
  - 移动对象并更新其中的指针

标记并压缩示例

- **注意:** 对象的位置发生改变，所有的指针都可能需要更新



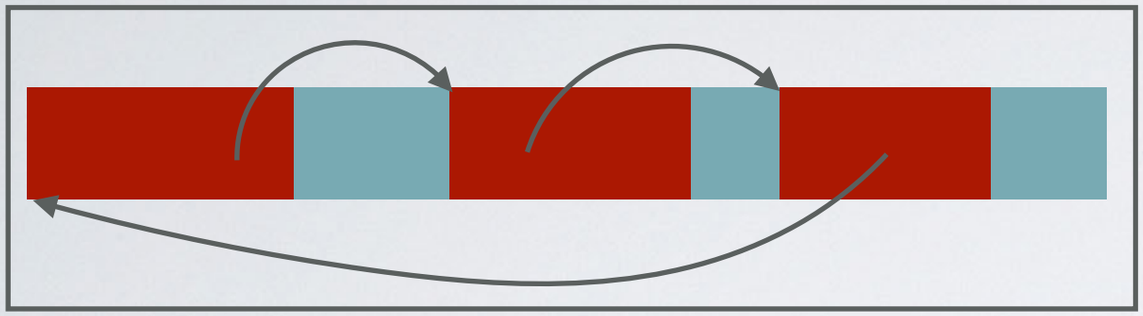
小结：标记-清扫

- **优点：**
  - 基本没有空间代价(一个内存块只需要若干个二进制位)
  - 可以正确处理循环数据结构
- **缺点：**
  - 应用程序必须全面停顿，不适用于实时系统
    - 可以采用**增量式回收**或**部分回收**来改善(参见第 7.7 节)
  - 可能会造成堆区的碎片化
    - 可以用「标记并压缩」来解决
- 实际中可以同时使用「引用计数」和「标记-清扫」
  - 比如 Python：https://docs.python.org/3/library/gc.html

拷贝回收器

- 「标记并压缩」的问题：压缩时需要扫描整个堆区
- copying collector
- 堆区空间被分为两个**半空间(semispace)**：
  - *From* 半空间：在这里分配内存
  - *To* 半空间：拷贝可达对象到这里
- 策略：
  - 在 *From* 半空间里分配内存，当其填满后，开始垃圾回收
  - 回收时，把可达对象拷贝到 *To* 半空间
  - 回收完成后，把两个半空间的角色对换，应用程序继续

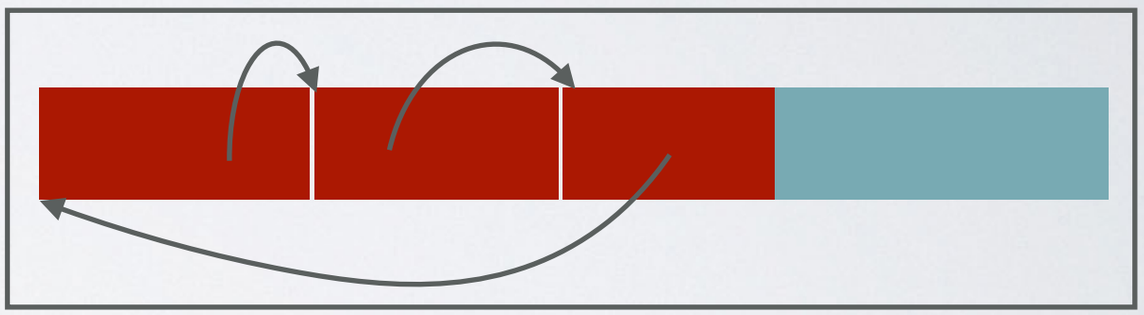
拷贝回收示例



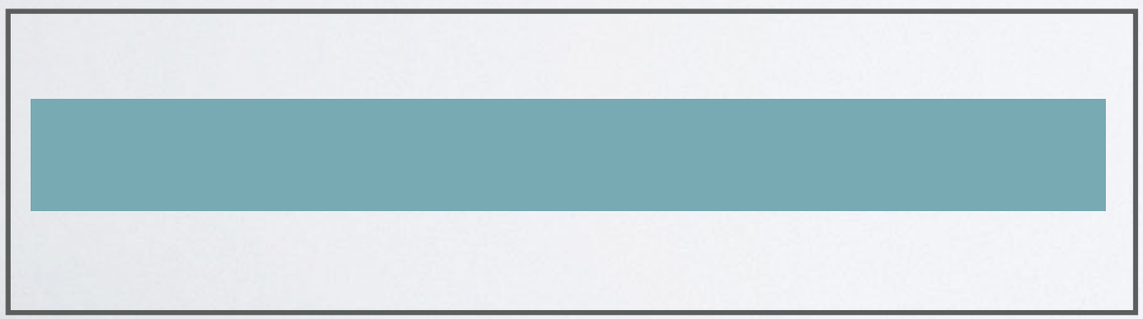
From 半空间

To 半空间

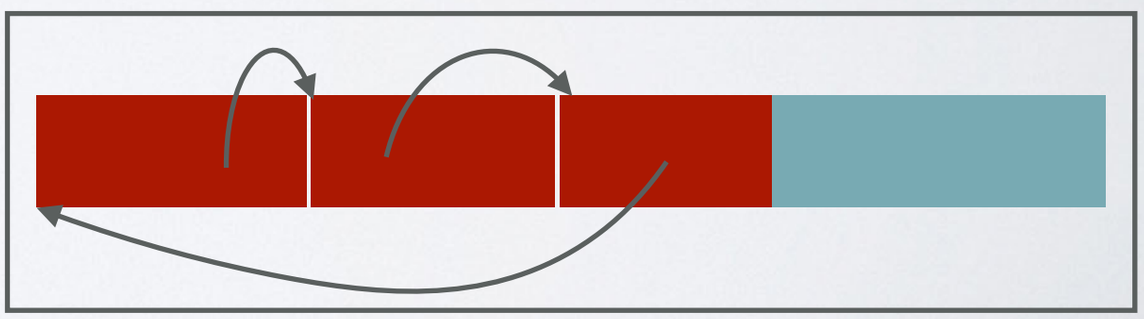
From 半空间



To 半空间



To 半空间

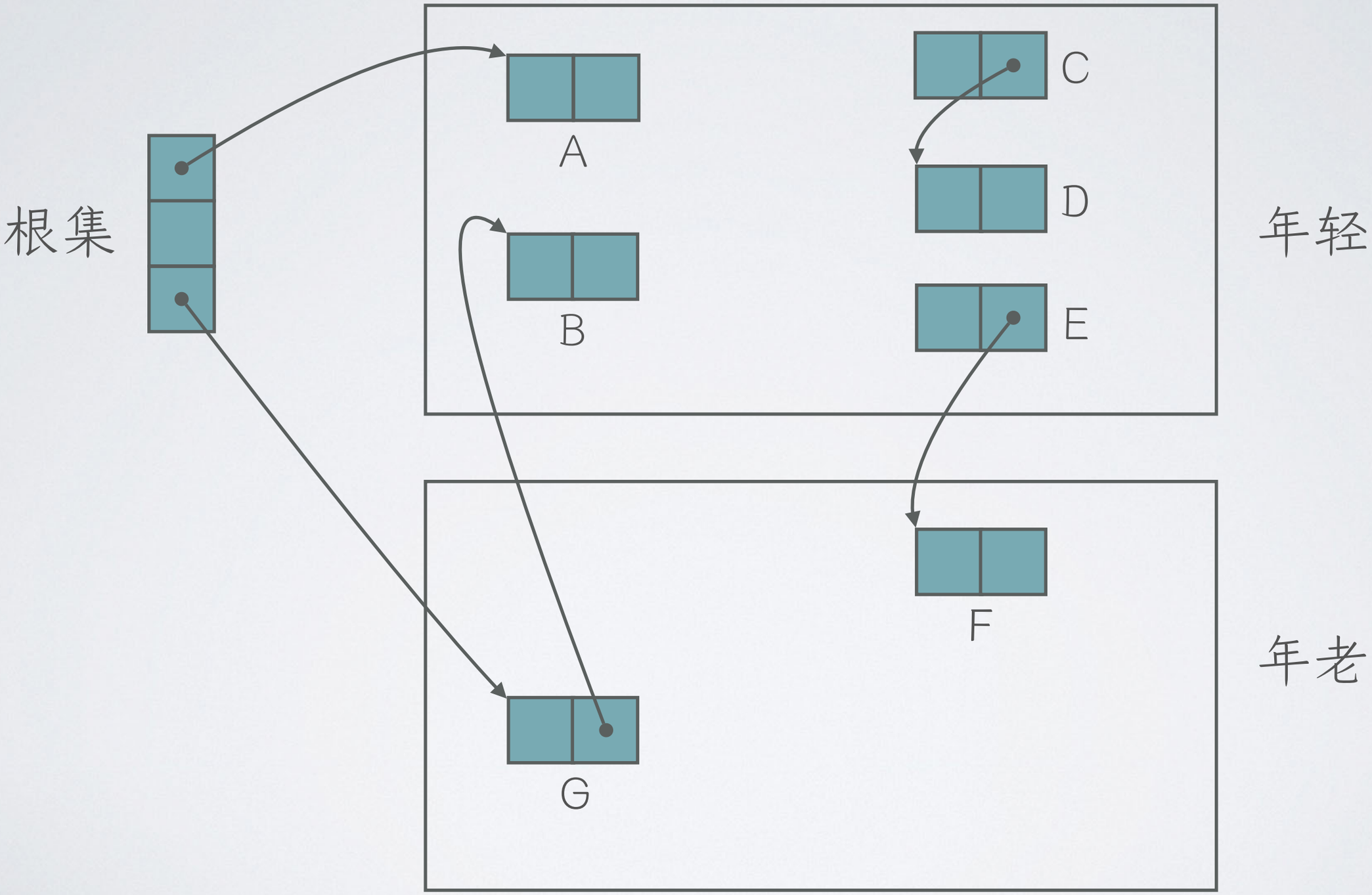


From 半空间

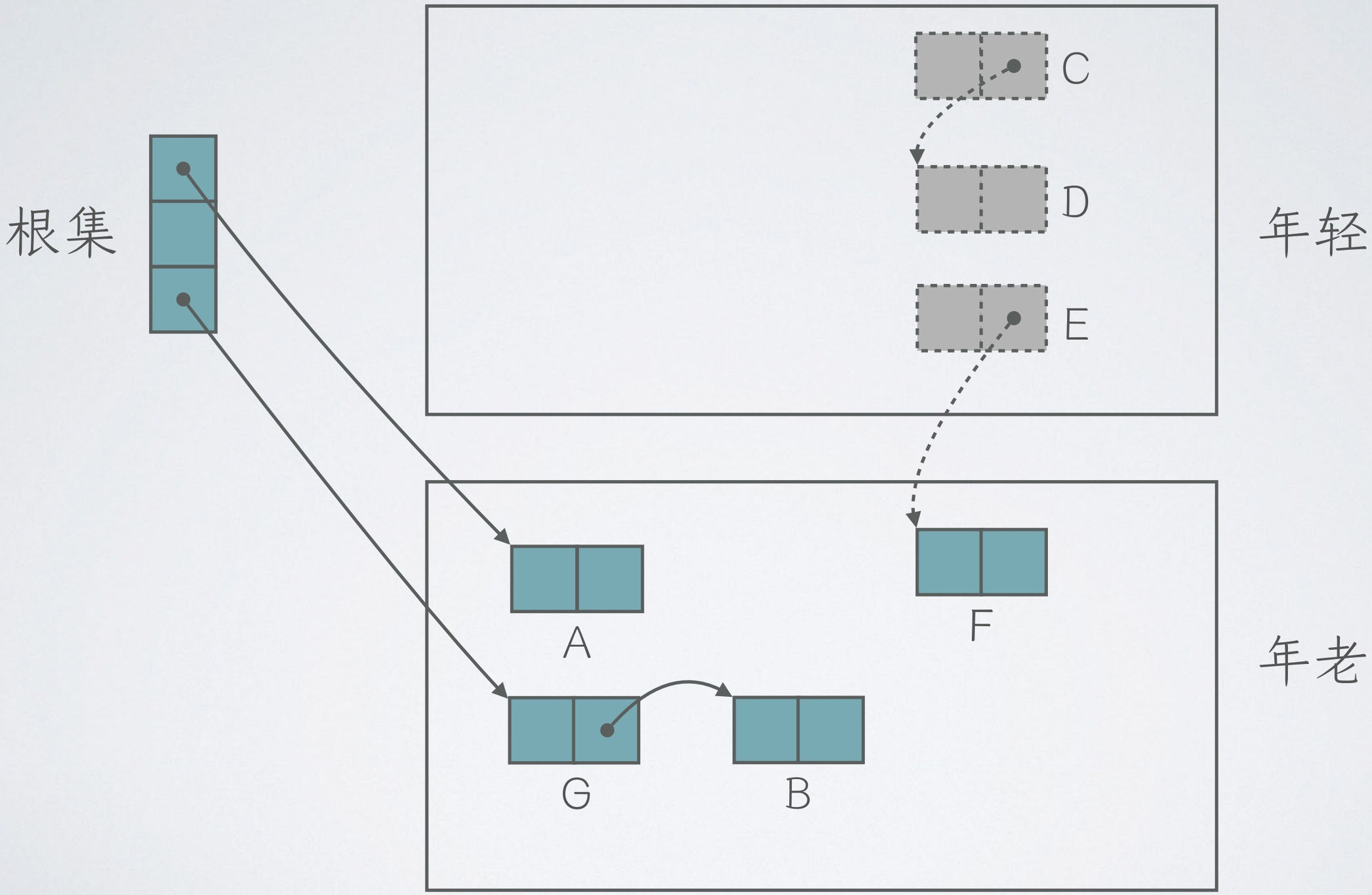
世代垃圾回收器

- generational garbage collector
- **出发点：大多数对象生命周期都很短**
  - 据估计 80-95% 的对象「活」不过几个 MB
- 基本思想：把堆区分成**不同的年龄区域**(代表不同的世代)，对比较年轻的区域进行更加频繁的垃圾回收
  - 在一个回收周期不用跟踪所有的内存单元
  - 周期性地对「较老」的区域进行回收

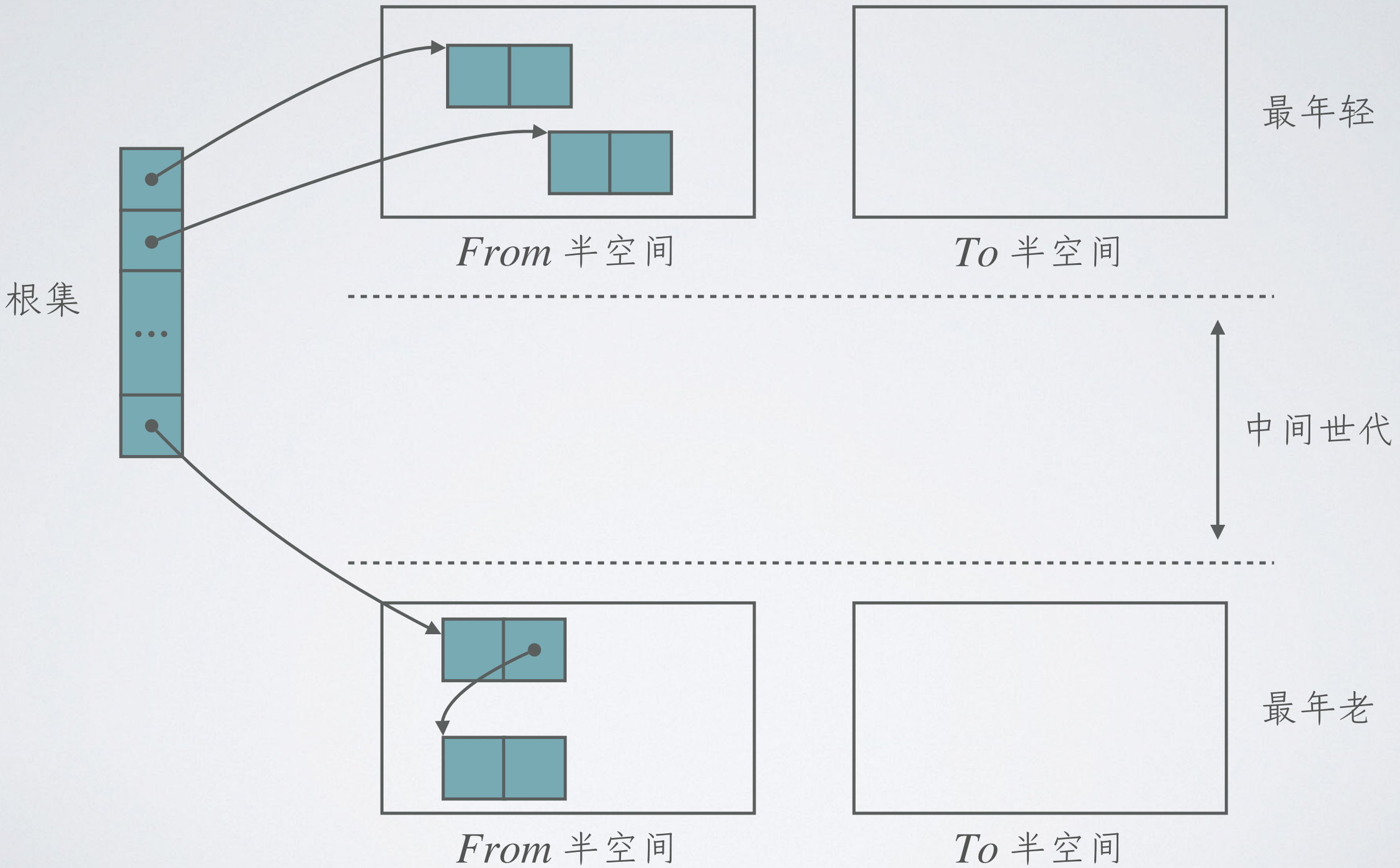
世代垃圾回收示例



世代垃圾回收示例



拷贝世代垃圾回收器



主要内容

- 运行时环境：非局部数据访问
- 运行时环境：垃圾回收
- **目标代码生成：针对 SSA 的寄存器分配**
- 代码优化：基于路径的数据流分析

SSA IR 的目标代码生成：指令选择

- 为**块参数**或者 φ **函数**生成相应的寄存器间的 LD 指令

```
pow(b0,e0):
  r0 = 1
  goto loop(e0,r0)
loop(e1,r1):
  if e1 > 0
    then goto body()
    else goto done()
body():
  r3 = r1 * b0
  e3 = e1 - 1
  goto loop(e3,r3)
done():
  return r1
```

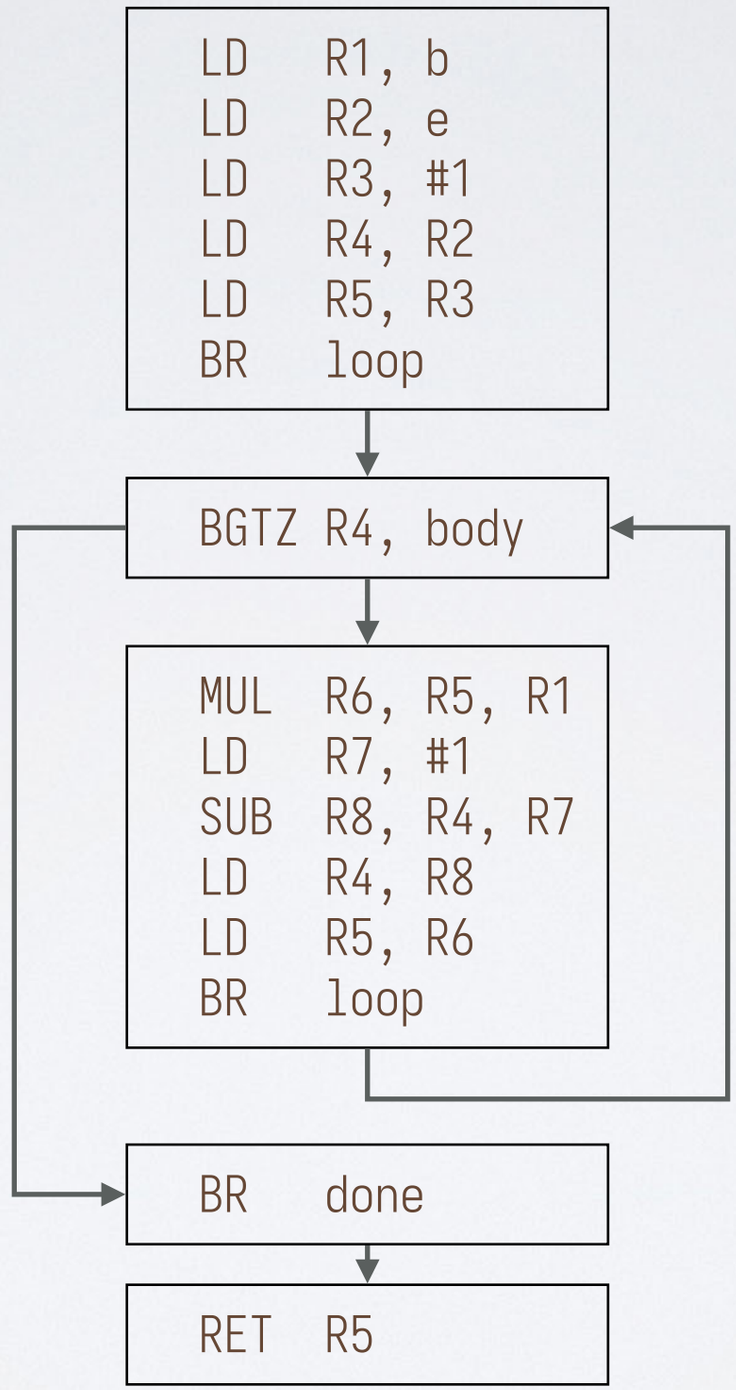
```
pow:
  LD   R1, b          ; R1:b0
  LD   R2, e          ; R2:e0
  LD   R3, #1         ; R3:r0
  LD   R4, R2         ; R4:e1
  LD   R5, R3         ; R5:r1
  BR   loop
loop:
  BGTZ R4, body
  BR   done
body:
  MUL  R6, R5, R1     ; R6:r3
  LD   R7, #1
  SUB  R8, R4, R7     ; R8:e3
  LD   R4, R8
  LD   R5, R6
  BR   loop
done:
  RET  R5
```

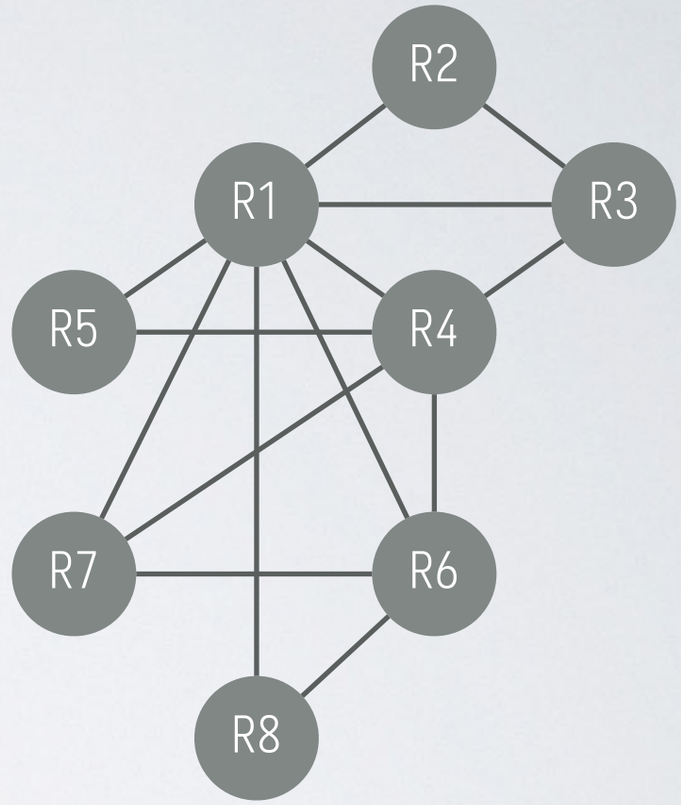
```
pow(b0,e0):
  r0 = 1
  goto loop
loop:
  e1 = φ(e0,e3)
  r1 = φ(r0,r3)
  if e1 > 0
    then goto body
    else goto done
body:
  r3 = r1 * b0
  e3 = e1 - 1
  goto loop
done:
  return r1
```

SSA IR 的目标代码生成：寄存器分配

```
pow:
  LD   R1, b
  LD   R2, e
  LD   R3, #1
  LD   R4, R2
  LD   R5, R3
  BR   loop
loop:
  BGTZ R4, body
  BR   done
body:
  MUL  R6, R5, R1
  LD   R7, #1
  SUB  R8, R4, R7
  LD   R4, R8
  LD   R5, R6
  BR   loop
done:
  RET  R5
```

```
{}
{R1}
{R1,R2}
{R1,R2,R3}
{R1,R3,R4}
{R1,R4,R5}
{R1,R4,R5}
{R1,R4,R5}
{R5}
{R1,R4,R5}
{R1,R4,R5}
{R1,R4,R6}
{R1,R4,R6,R7}
{R1,R6,R8}
{R1,R4,R6}
{R1,R4,R5}
{R5}
{R5}
```

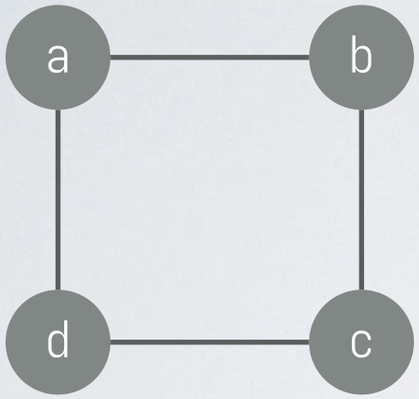




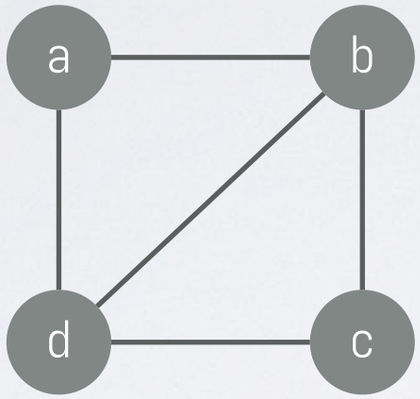
这个冲突图有什么特点？

弦图（Chordal Graph）

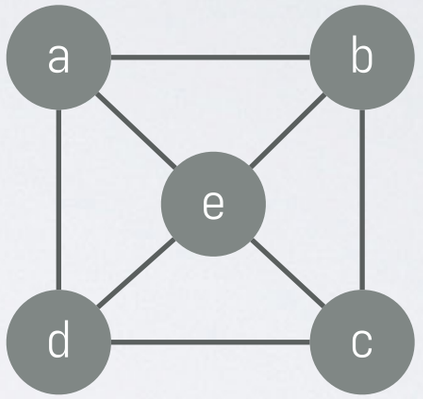
- 无向图满足图中每个长度不小于 4 的环中都有**弦**
  - **弦(chord)**：连接环中两个不相邻点的边



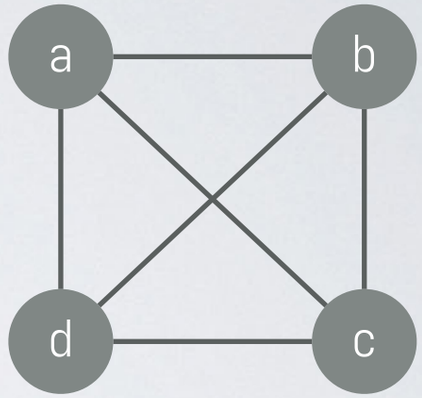
非弦图



弦图



非弦图



弦图

- 前面课件中见过的冲突图都是弦图
  - 研究表明[PP05]，编译实践中产生的冲突图约 95% 都是弦图

- **你能构造一段代码，使得产生的冲突图不是弦图吗？**

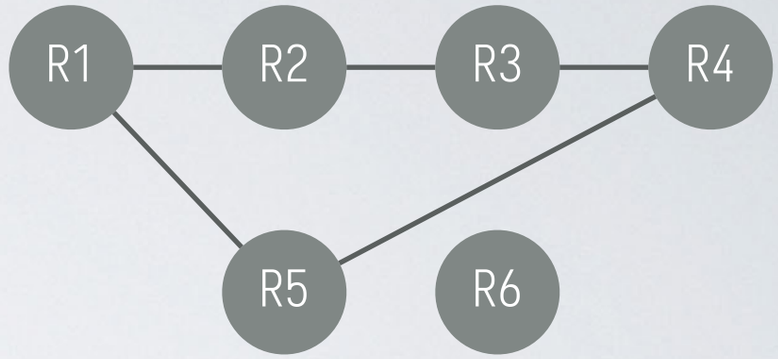
[PP05] Fernando Magno Quintao Pereira and Jens Palsberg. Register Allocation via Coloring of Chordal Graphs. In APLAS'05.

冲突图不是弦图的例子

```
a = 0
b = 1
c = a + b
d = b + c
a = c + d
t = 7
d = a + t
e = t + d
return e
```

```
  LD  R1, #0
  LD  R2, #1
  ADD R3, R1, R2
  ADD R4, R2, R3
  ADD R1, R3, R4
  LD  R5, #7
  ADD R4, R1, R5
  ADD R6, R5, R4
  RET R6
```

```
{}
{R1}
{R1,R2}
{R2,R3}
{R3,R4}
{R1}
{R1,R5}
{R4,R5}
{R6}
```



3-着色

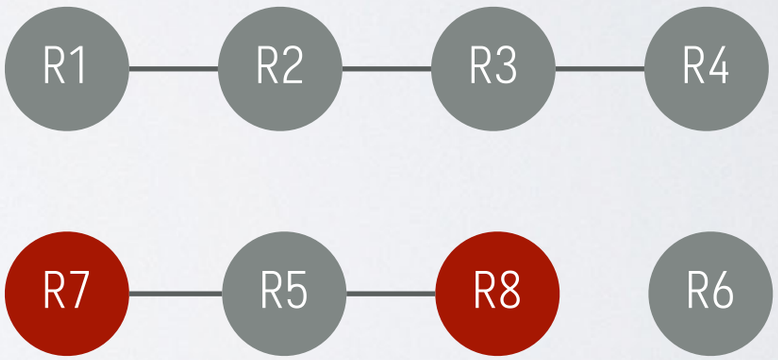
转换为 SSA 形式

研究表明[Hac07]，SSA IR 产生的冲突图都是弦图

```
a = 0
b = 1
c = a + b
d = b + c
a1 = c + d
t = 7
d1 = a1 + t
e = t + d1
return e
```

```
  LD  R1, #0
  LD  R2, #1
  ADD R3, R1, R2
  ADD R4, R2, R3
  ADD R7, R3, R4
  LD  R5, #7
  ADD R8, R7, R5
  ADD R6, R5, R8
  RET R6
```

```
{}
{R1}
{R1,R2}
{R2,R3}
{R3,R4}
{R7}
{R5,R7}
{R5,R8}
{R6}
```

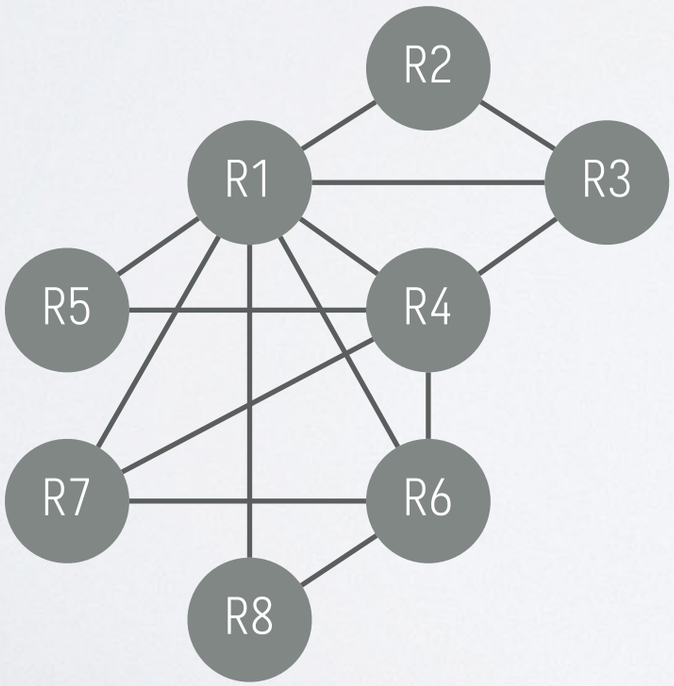


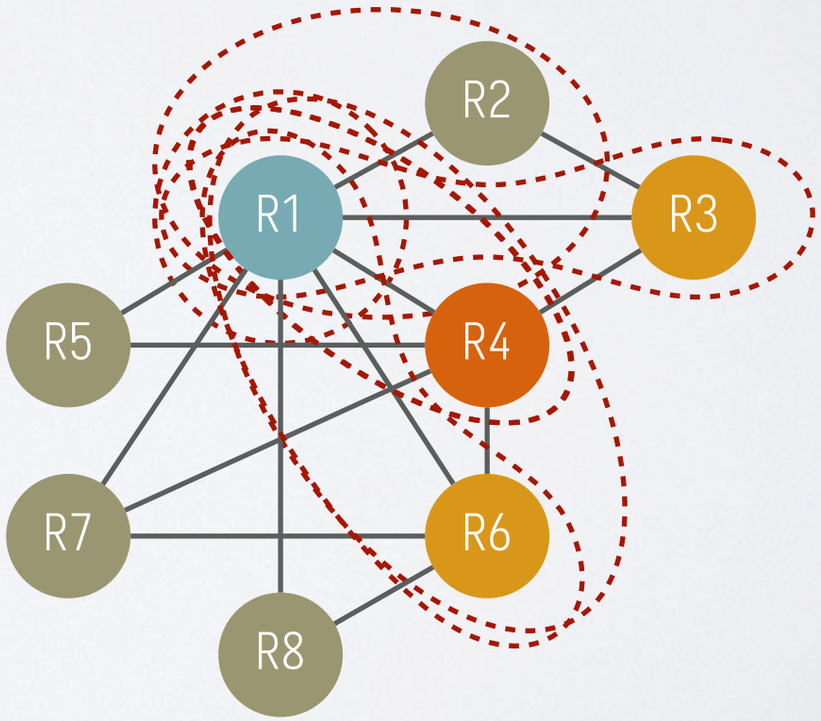
2-着色

[Hac07] Sebastian Hack. Register Allocation for Programs in SSA Form. PhD thesis, Universitat Karlsruhe, October 2007.

弦图的着色

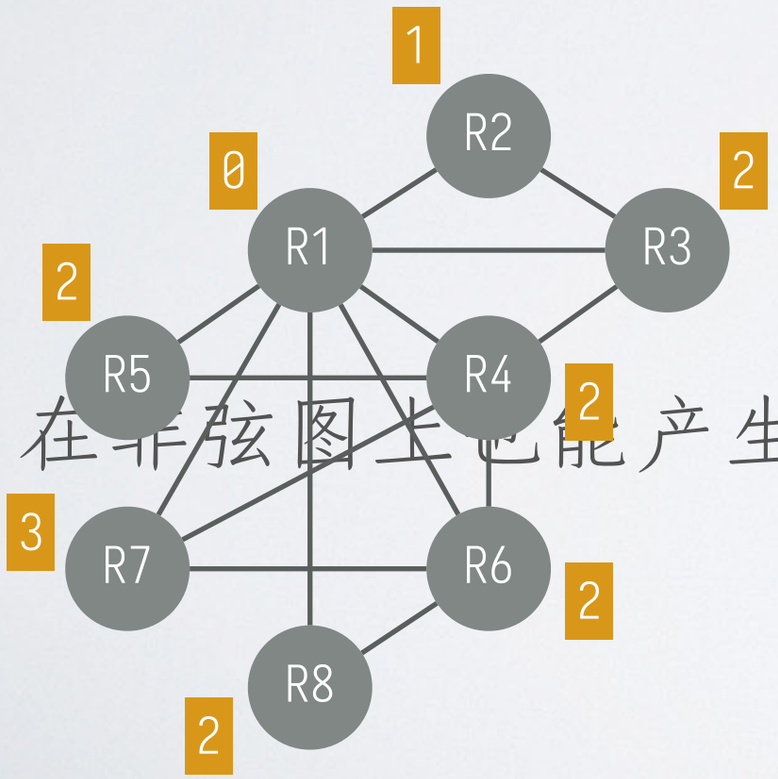
- **单纯点(simplical vertex)**：邻居构成一个**团**的点
- **单纯消除序列**：$v_1, \ldots, v_n$，其中每个点 v_i 都是子图 $v_1, \ldots, v_i$ 中的单纯点
  - 按照单纯消除序列的顺序来着色，可以证明能得到一个**最优方案**！
- **定理**：一个图有单纯消除序列**当且仅当**它是一个弦图

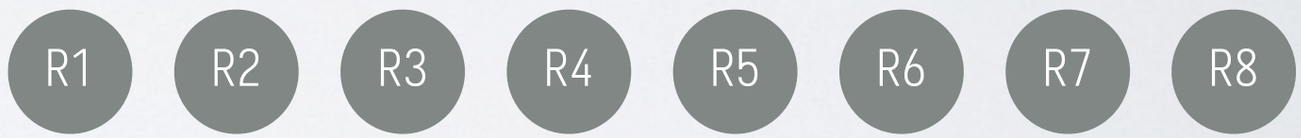




计算弦图的单纯消除序列

- **多项式时间算法**：$O(|V|+|E|)$，其中 V、E 为点集、边集
- **迭代算法**，为每个点 v 维护一个权重 $wt(v)$
  - $wt(v)$ 表示在算法进行中有多少 v 的邻居已经被添加进序列中了
  - 每一次迭代选择一个具有**最大权重**的点，并更新它邻居的权重

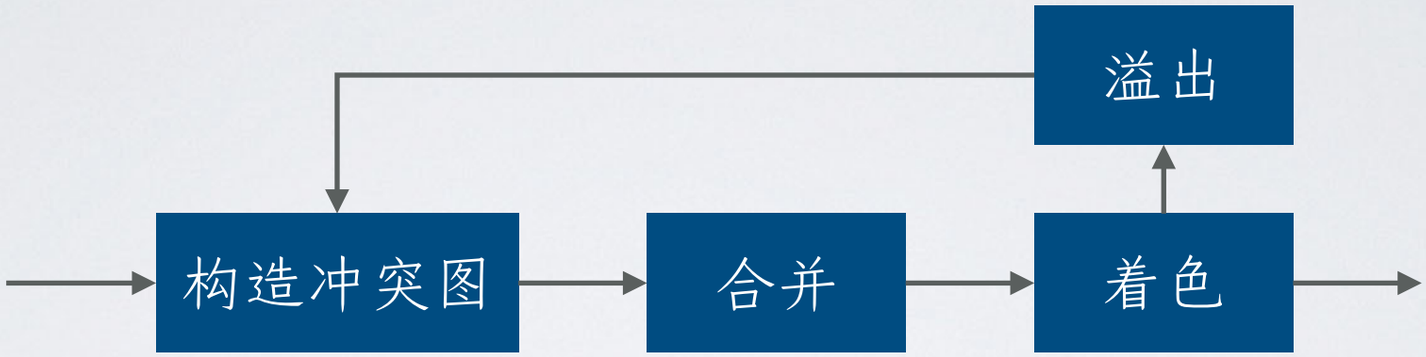




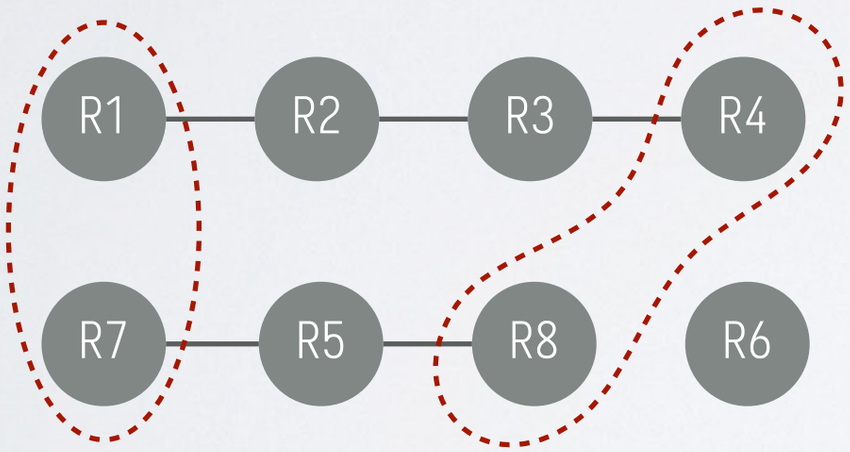
- 在非弦图上也能产生一个序列，只是不保证产生最优着色

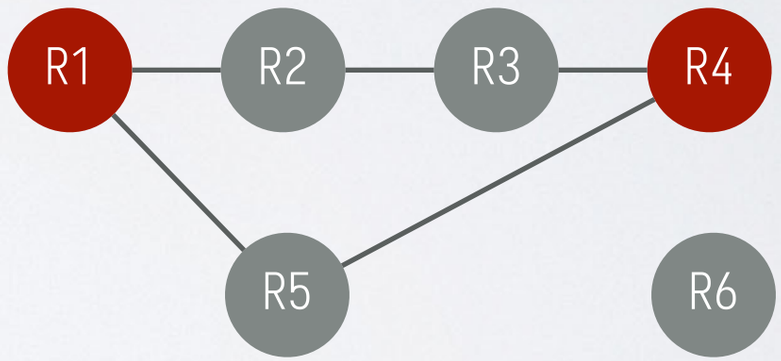
弦图上的寄存器合并

- 回顾：**合并(coalescing)**指的是在冲突图中合并不相邻结点



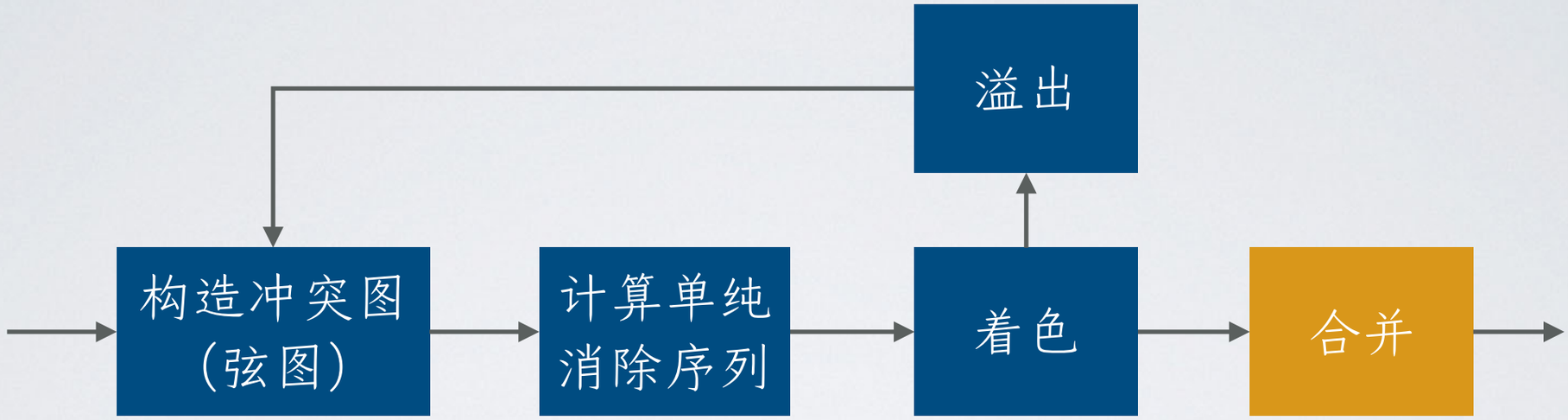
- **问题：**在弦图上合并结点可能导致冲突图不再是弦图





- **一种解决方案：**在完成着色之后再进行合并

弦图上的寄存器合并

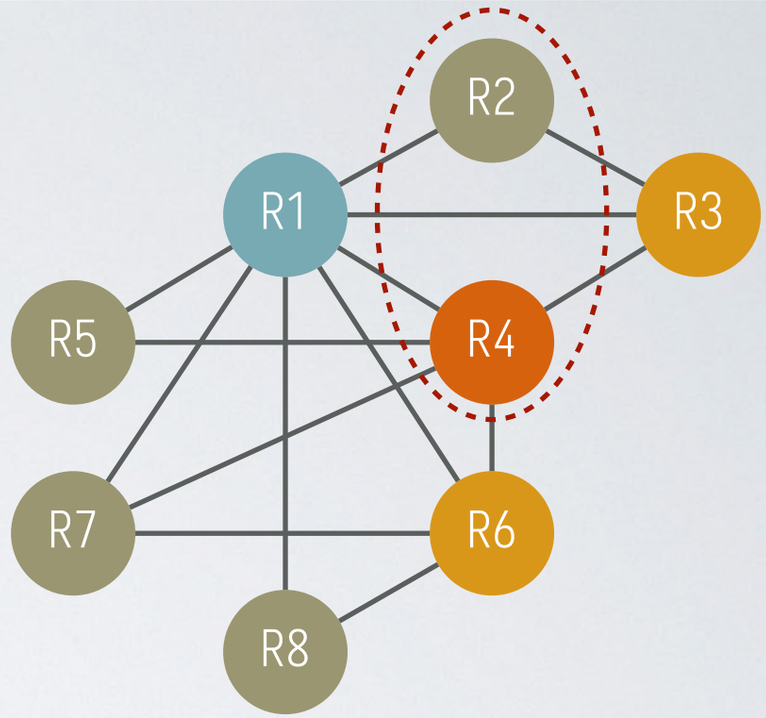


- 在指派给物理寄存器之前可以进行寄存器合并
  - 考虑代码中的每个寄存器间的移动：LD Rs, Rt
  - 如果 Rs 和 Rt 之间有冲突(即在冲突图中相邻)，则不能合并
  - 如果 Rs 和 Rt 的邻居中有一个没有出现过的颜色 c，则可以合并 Rs 和 Rt 为一个具有颜色 c 的结点
    - 操作时可以把 Rs 和 Rt 替换为一个新的符号寄存器 Ru

弦图上的寄存器合并的例子（1）

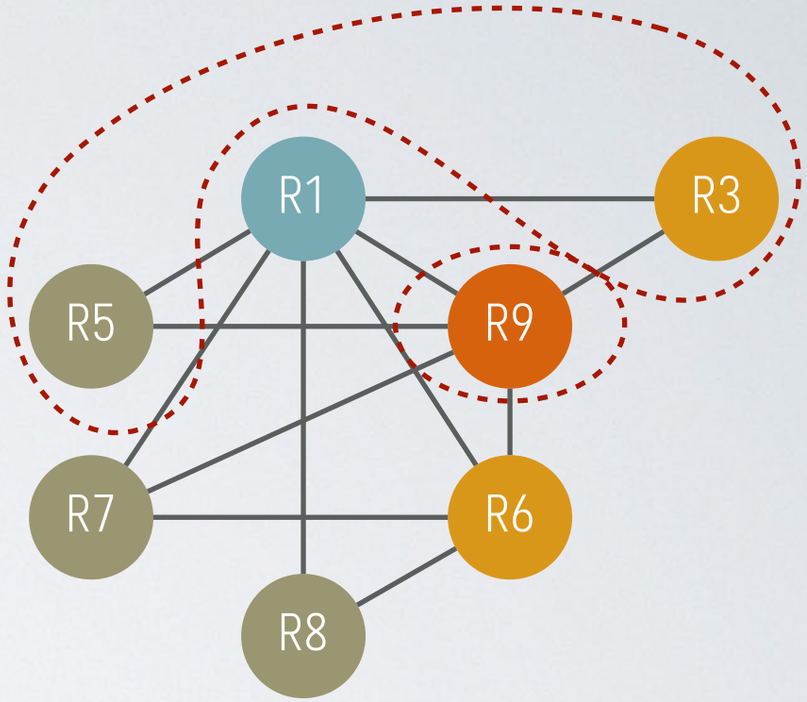
```
pow:
  LD   R1, b
  LD   R2, e
  LD   R3, #1
  LD   R4, R2
  LD   R5, R3
  BR   loop
loop:
  BGTZ R4, body
  BR   done
body:
  MUL  R6, R5, R1
  LD   R7, #1
  SUB  R8, R4, R7
  LD   R4, R8
  LD   R5, R6
  BR   loop
done:
  RET  R5
```

```
pow:
  LD   R1, b
  LD   R2, e
  LD   R3, #1
  LD   R4, R2
  LD   R2, R3
  BR   loop
loop:
  BGTZ R4, body
  BR   done
body:
  MUL  R3, R2, R1
  LD   R2, #1
  SUB  R2, R4, R2
  LD   R4, R2
  LD   R2, R3
  BR   loop
done:
  RET  R2
```



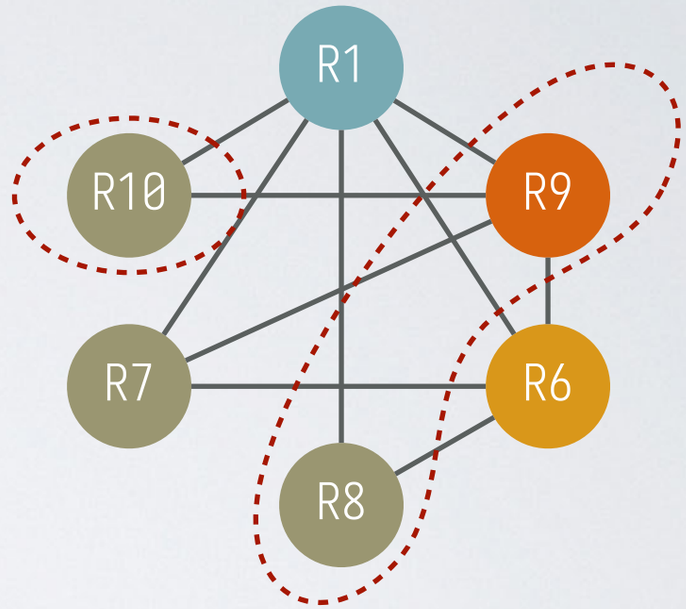
弦图上的寄存器合并的例子（2）

```
pow:
  LD   R1, b
  LD   R9, e
  LD   R3, #1
  LD   R9, R9
  LD   R5, R3
  BR   loop
loop:
  BGTZ R9, body
  BR   done
body:
  MUL  R6, R5, R1
  LD   R7, #1
  SUB  R8, R9, R7
  LD   R9, R8
  LD   R5, R6
  BR   loop
done:
  RET  R5
```



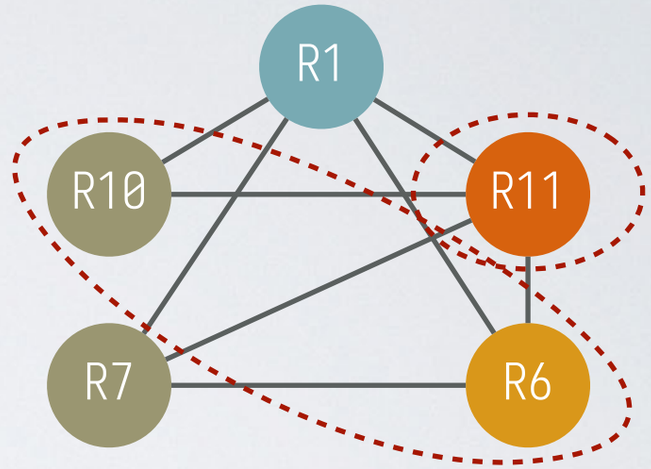
弦图上的寄存器合并的例子（3）

```
pow:
  LD   R1, b
  LD   R9, e
  LD   R10, #1
  LD   R9, R9
  LD   R10, R10
  BR   loop
loop:
  BGTZ R9, body
  BR   done
body:
  MUL  R6, R10, R1
  LD   R7, #1
  SUB  R8, R9, R7
  LD   R9, R8
  LD   R10, R6
  BR   loop
done:
  RET  R10
```



弦图上的寄存器合并的例子（4）

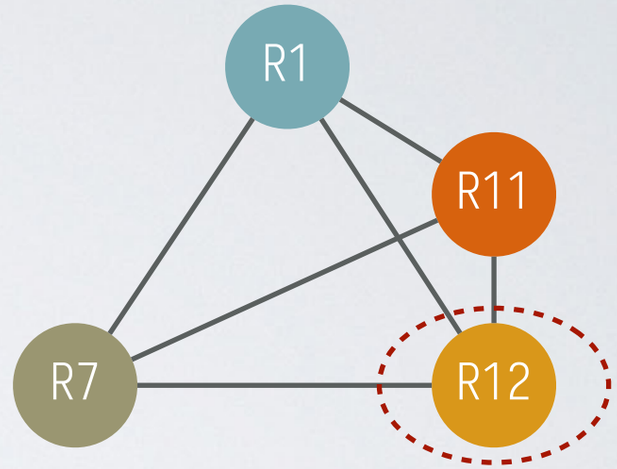
```
pow:
  LD   R1, b
  LD   R11, e
  LD   R10, #1
  LD   R9, R9
  LD   R10, R10
  BR   loop
loop:
  BGTZ R11, body
  BR   done
body:
  MUL  R6, R10, R1
  LD   R7, #1
  SUB  R11, R11, R7
  LD   R11, R11
  LD   R10, R6
  BR   loop
done:
  RET  R10
```



弦图上的寄存器合并的例子（5）

```
pow:
  LD   R1, b
  LD   R11, e
  LD   R12, #1
  LD   R9, R9
  LD   R10, R10
  BR   loop
loop:
  BGTZ R11, body
  BR   done
body:
  MUL  R12, R12, R1
  LD   R7, #1
  SUB  R11, R11, R7
  LD   R11, R11
  LD   R12, R12
  BR   loop
done:
  RET  R12
```

```
pow:
  LD   R1, b
  LD   R2, e
  LD   R3, #1
  BR   loop
loop:
  BGTZ R2, body
  BR   done
body:
  MUL  R3, R3, R1
  LD   R4, #1
  SUB  R2, R2, R4
  BR   loop
done:
  RET  R3
```



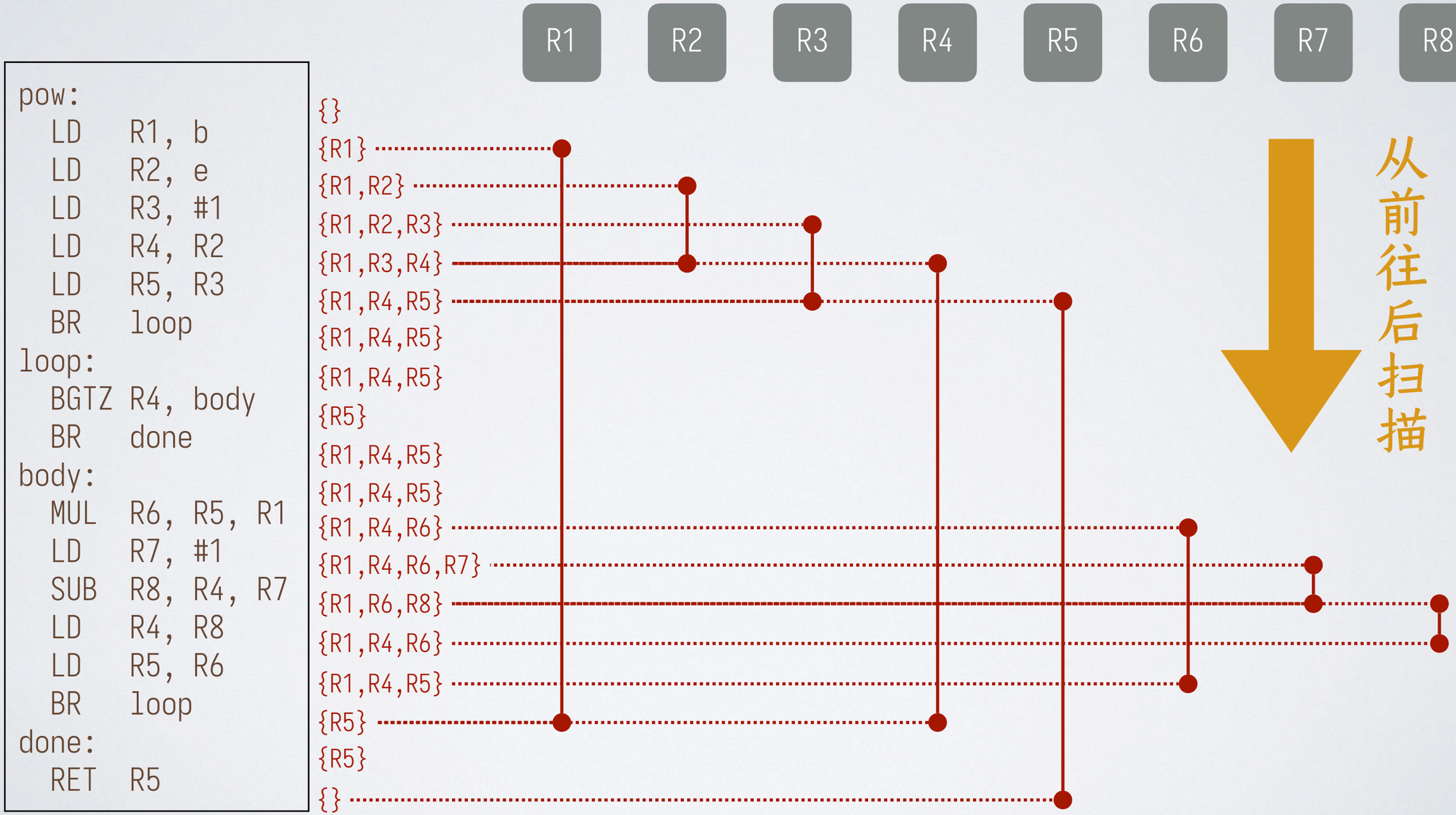
不产生多余的
LD Rs, Rt 指令

线性扫描寄存器分配

- **基于图着色的方法的问题在于时间复杂度比较高**
- Linear Scan Register Allocation (LSRA)
  - 时间复杂度关于代码大小是线性的
  - 能够达到与图着色算法接近的效果
  - 相对于x86 的 8 个寄存器，当代体系结构都有充分多的寄存器
    - x86-64: 16 个
    - RISC-V: 32 个
- LLVM、Java 的 JIT 编译器均默认采用线性扫描寄存器分配
  - JIT 编译器看重编译效率

线性扫描寄存器分配

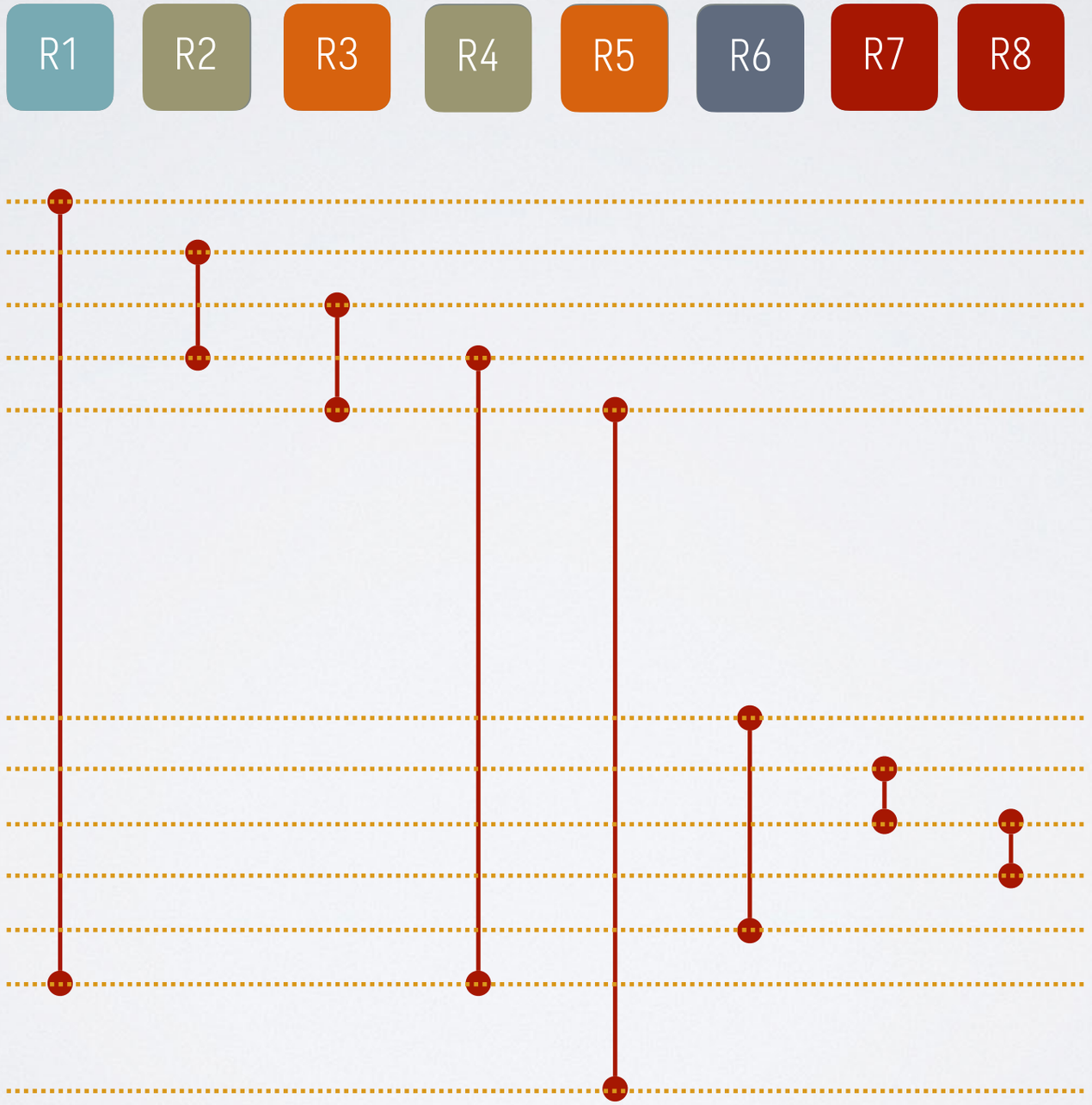
- 核心：为每个符号寄存器计算一个**活性区间(live interval)**



线性扫描寄存器分配

- 核心：**活性区间有重叠**的符号寄存器**不能有相同的指派**

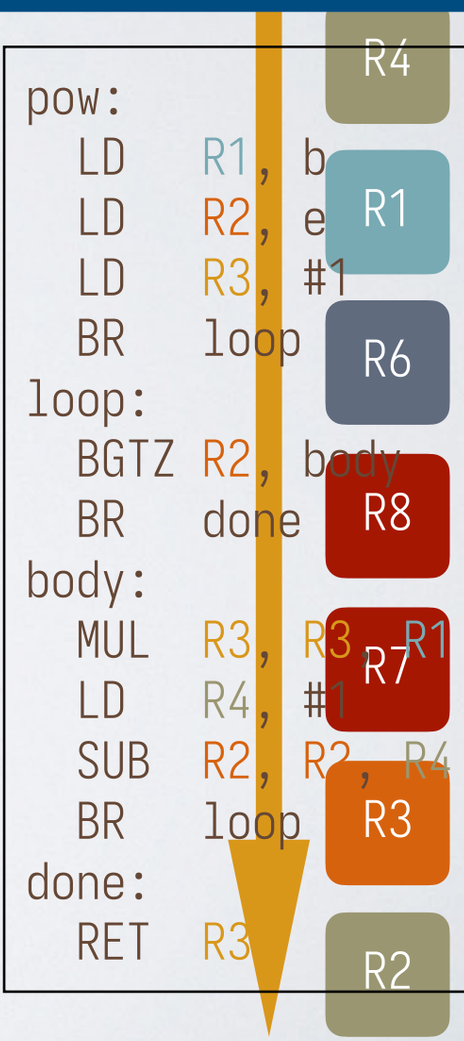
```
pow:
  LD   R1, b
  LD   R2, e
  LD   R3, #1
  LD   R4, R2
  LD   R5, R3
  BR   loop
loop:
  BGTZ R4, body
  BR   done
body:
  MUL  R6, R5, R1
  LD   R7, #1
  SUB  R8, R4, R7
  LD   R4, R8
  LD   R5, R6
  BR   loop
done:
  RET  R5
```



优先队列

前面图着色只需要 4 个寄存器

```
pow:
  LD   R1, b
  LD   R2, e
  LD   R3, #1
  BR   loop
loop:
  BGTZ R2, body
  BR   done
body:
  MUL  R3, R3, R1
  LD   R4, #1
  SUB  R2, R2, R4
  BR   loop
done:
  RET  R3
```



线性扫描寄存器分配

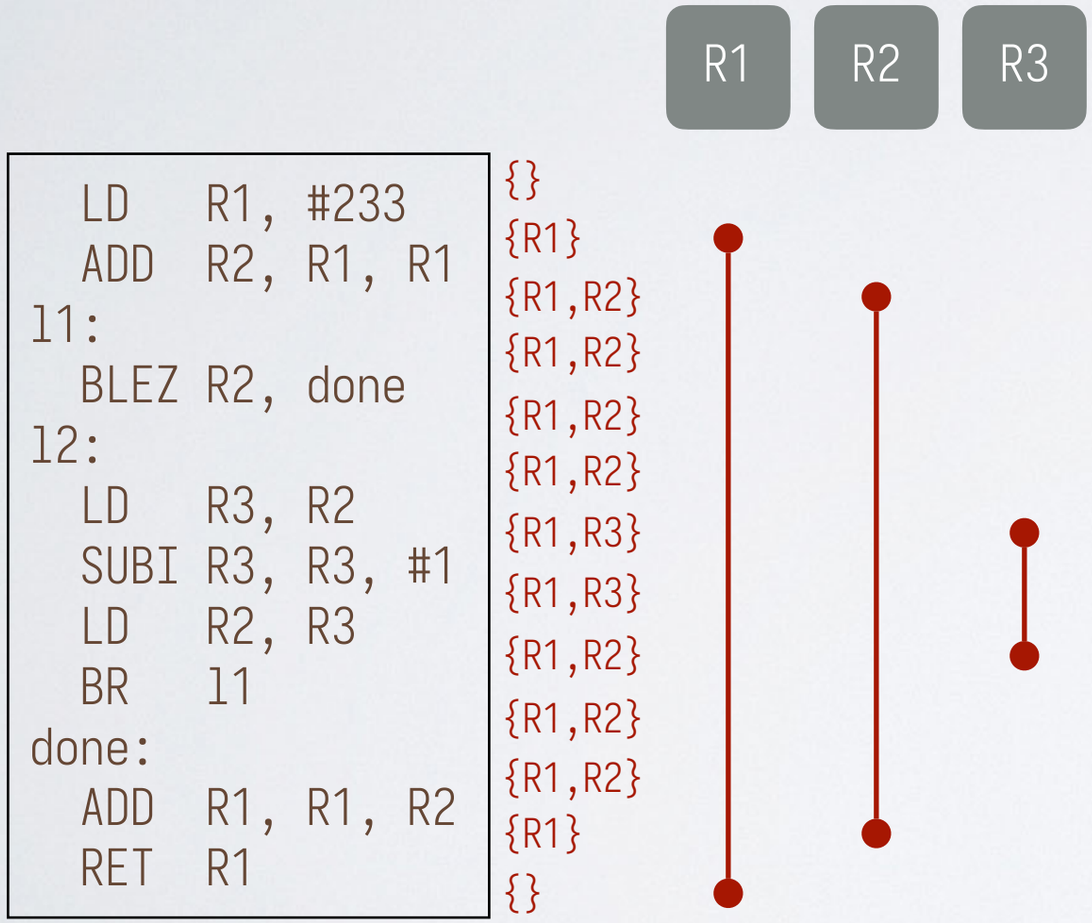
- 算法小结：维护一个**活动(active)优先队列**
  - 优先队列中的符号寄存器以活性区间终点为序
  - 当扫描到一个新的活性区间起点时：
    - 移除队列中的「过期」符号寄存器，即活性区间结束的符号寄存器
    - 将新的活性区间加入队列，并选在与队列中都不同的指派

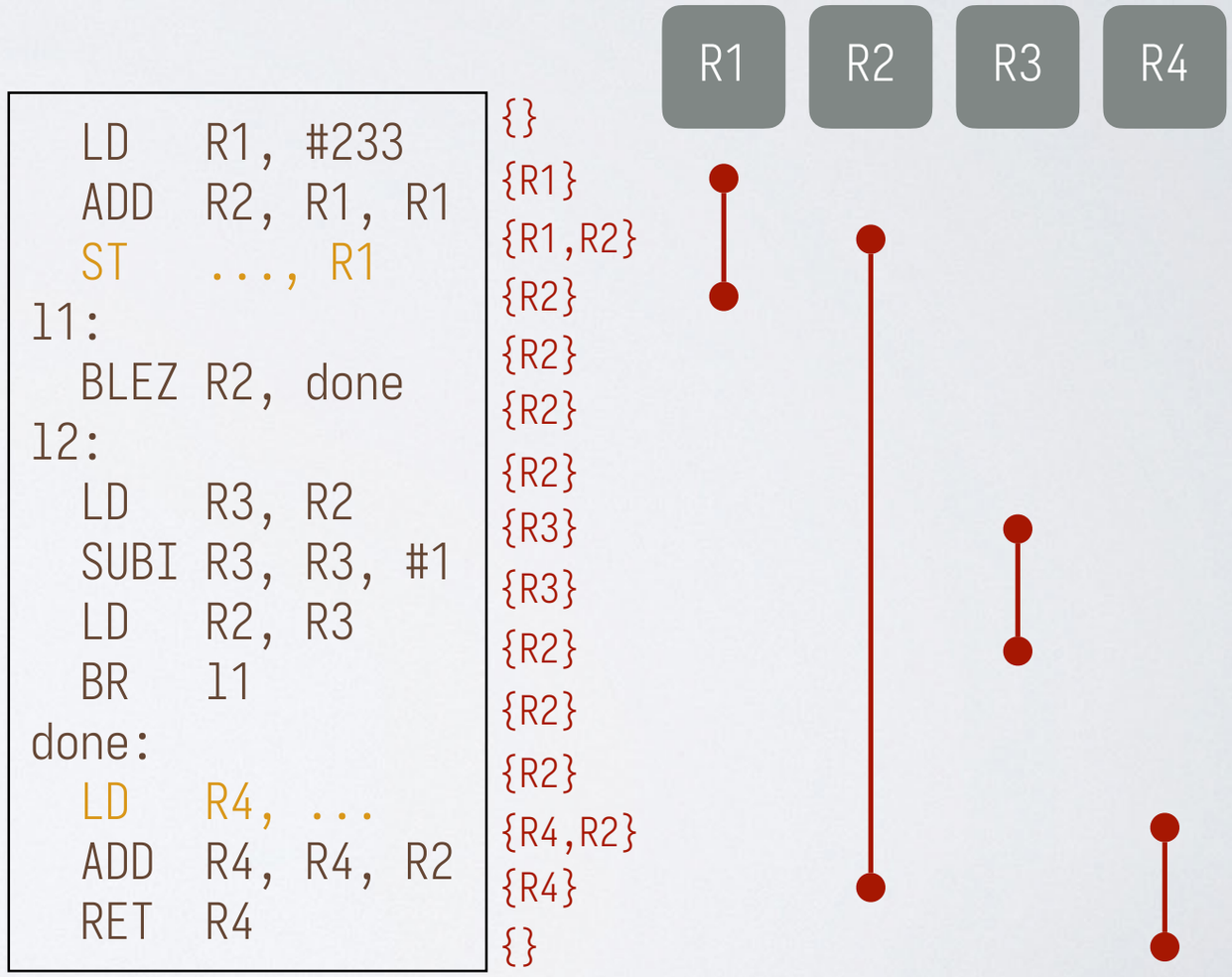
- 控制队列总大小为可用寄存器的数目
  - 无法加入队列时，需要**溢出**队列中的符号寄存器
  - 一种方案：溢出**活性区间结束最晚**的符号寄存器

- 时间复杂度：$O(|V|\log|R|)$
  - 其中 V、R 为符号寄存器和物理寄存器集合

活性区间拆分

- 前面介绍的**拆分(splitting)**技术本质上是在分解活性区间
- 线性扫描算法可以结合活性区间拆分来提升效果

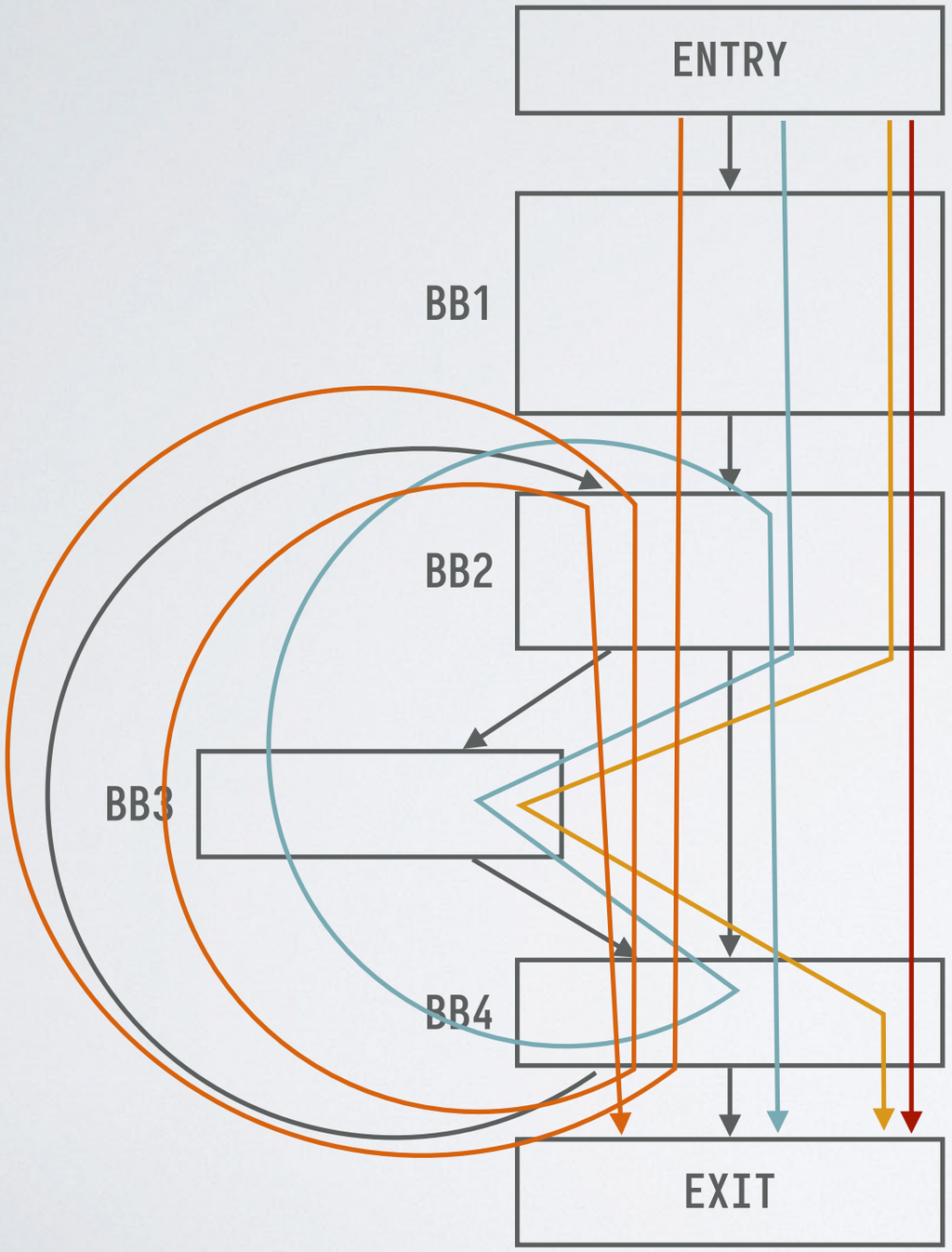




主要内容

- 运行时环境：非局部数据访问
- 运行时环境：垃圾回收
- 目标代码生成：针对 SSA 的寄存器分配
- **代码优化：基于路径的数据流分析**

数据流分析是什么？



- 分析流图上的**路径集合**的性质

  ENTRY->BB1->BB2->BB4->EXIT

  ENTRY->BB1->BB2->BB3->BB4->EXIT

  ENTRY->BB1->BB2->BB3->BB4->BB2->BB4->EXIT

  ENTRY->BB1->BB2->BB4->BB2->BB4->BB2->BB4->EXIT

  ……

- 流图上可以有**无限多条**路径

- **是否可以用一个有限的形式刻画一个无限的集合？**

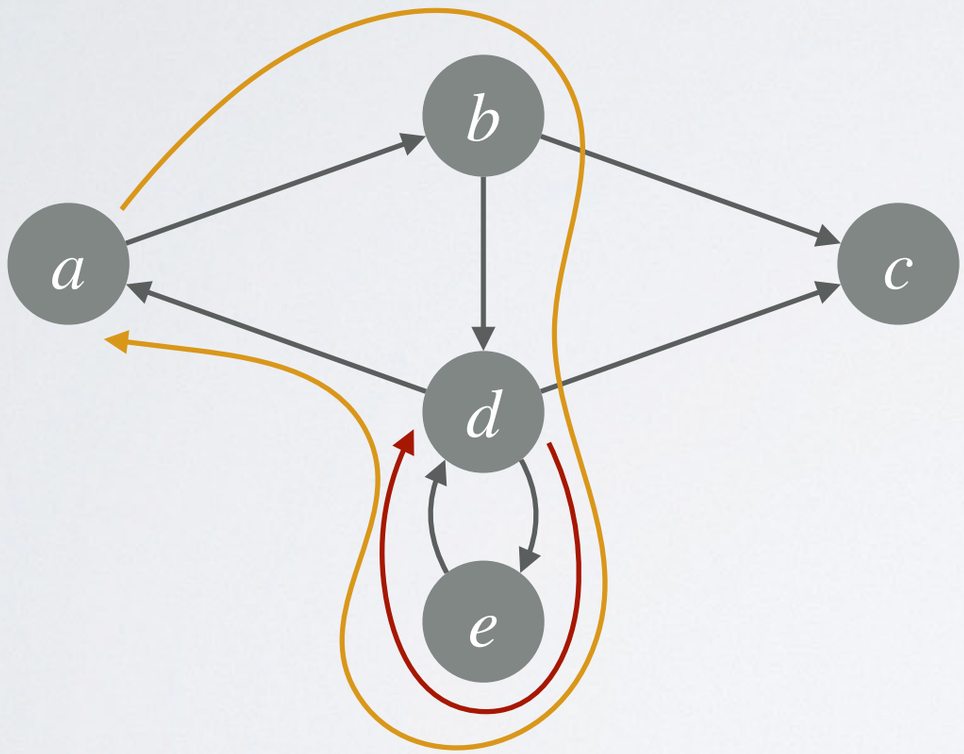
回顾：正则表达式

- 用**有限**形式(正则表达式)来刻画可能**无限**的集合(正则语言)
  - ϵ 表示语言 $L(\epsilon) = \{\epsilon\}$
  - 如果 a 是字母表中的符号，则 a 表示语言 $L(a) = \{a\}$
  - 设 r_1 和 r_2 是正则表达式
    - $r_1 | r_2$ 表示语言 $L(r_1 | r_2) = L(r_1) \cup L(r_2)$
    - $r_1 r_2$ 表示语言 $L(r_1 r_2) = \{xy \mid x \in L(r_1), y \in L(r_2)\}$
    - r_1^* 表示语言 $L(r_1^*) = \{x_1 x_2 \cdots x_n \mid n \geq 0, x_1, x_2, \cdots, x_n \in L(r_1)\}$

- **例子：** `0(0 | 1)*` 刻画了所有以 `0` 开头的 `0/1` 符号串

有向图上的路径表达式

- 考虑有向图 $G=(V,E)$, 一个**路径表达式(path expression)**是一个以 E 为字母表的**正则表达式** r, 且 r 识别的每个符号串都是图 G 中的一条**路径**



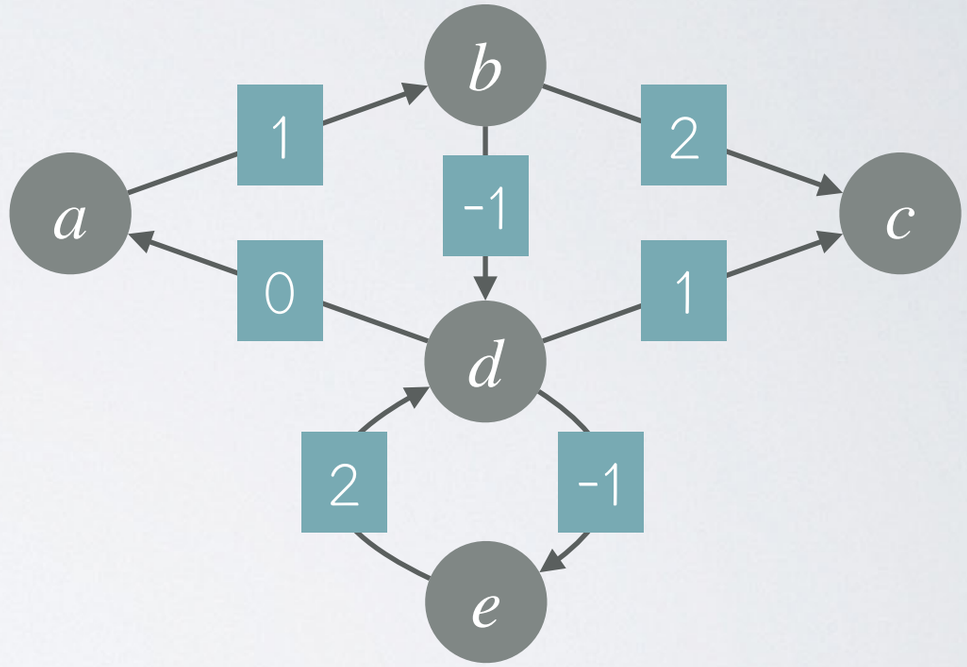
字母表 $\Sigma=\{\langle a,b\rangle,$
$\langle b,d\rangle,\langle b,c\rangle,$
$\langle d,a\rangle,\langle d,c\rangle,\langle d,e\rangle,$
$\langle e,d\rangle\}$

识别所有 a 到 c 的路径的路径表达式:

$$(\langle a,b\rangle\langle b,d\rangle(\langle d,e\rangle\langle e,d\rangle)^*\langle d,a\rangle)^*\langle a,b\rangle(\langle b,c\rangle \mid \langle b,d\rangle(\langle d,e\rangle\langle e,d\rangle)^*\langle d,c\rangle)$$

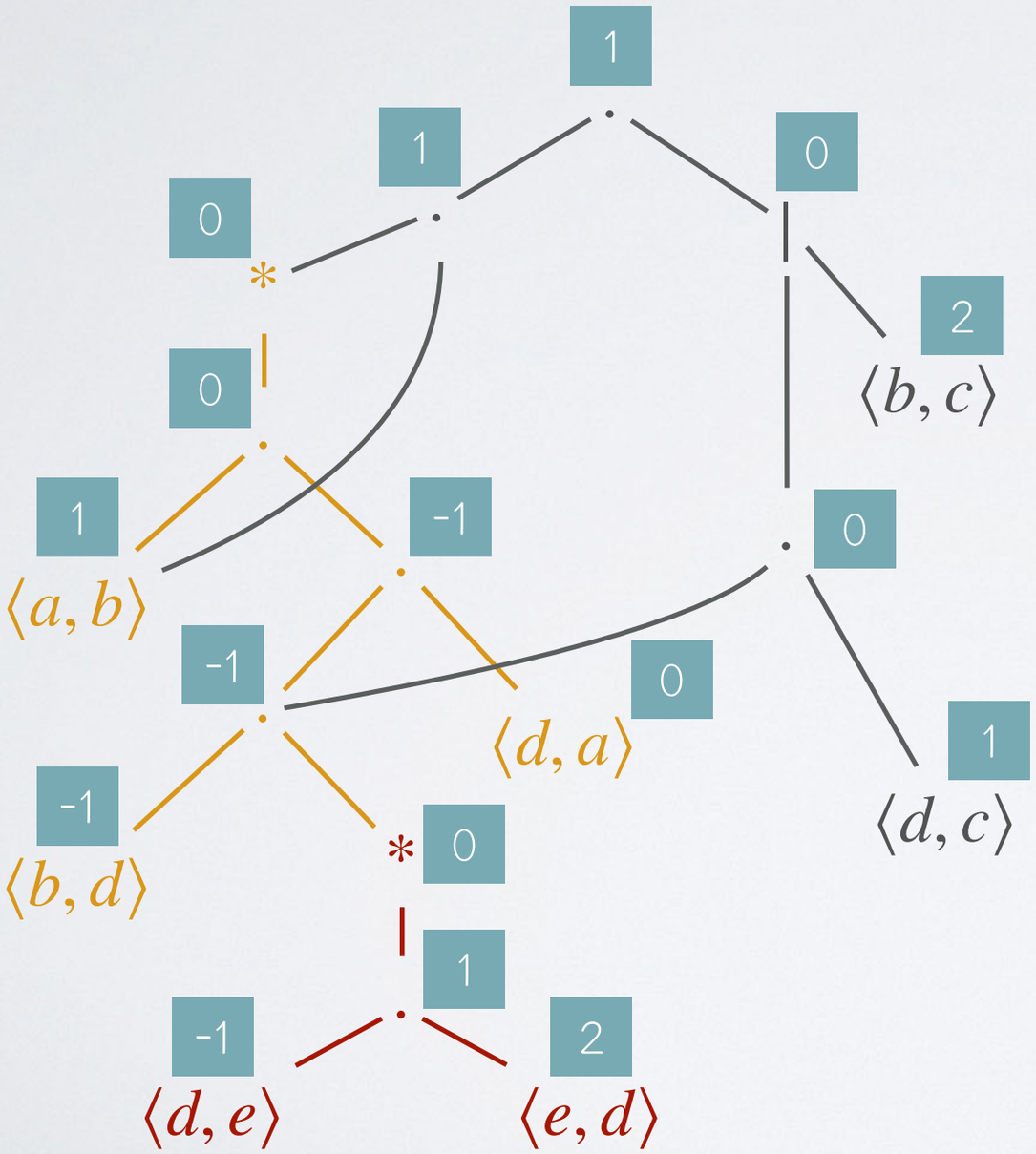
通过路径表达式求最短路（1）

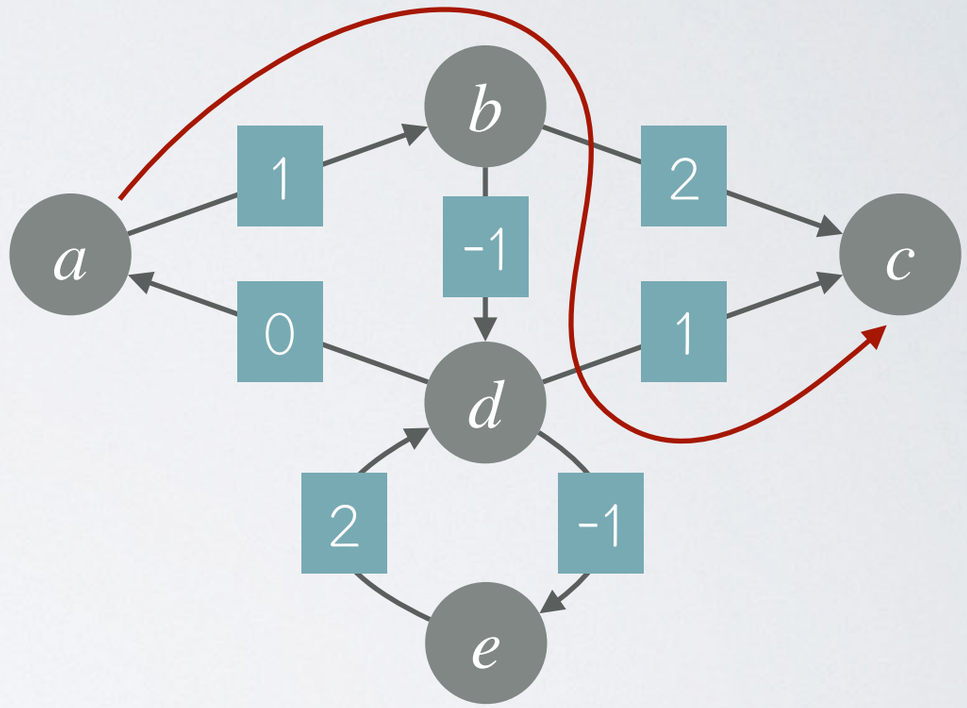
- 用 $F(r)$ 表示 r 能识别的路径的长度的最小值
  - $F(\epsilon) = 0, F(\langle u_1, u_2 \rangle) =$ 边 $\langle u_1, u_2 \rangle$ 的长度
  - $F(r_1 \mid r_2) = \min(F(r_1), F(r_2))$
  - $F(r_1 r_2) = F(r_1) + F(r_2)$
  - $F(r_1^*) = ?$
    - 若 $F(r_1) < 0$，则 $F(r_1^*) = -\infty$
    - 否则 $F(r_1^*) = 0$
- 例子：
  - $F(\langle d, e \rangle) = -1, F(\langle e, d \rangle) = 2$
  - $F(\langle d, e \rangle \langle e, d \rangle) = (-1) + 2 = 1$
  - $F((\langle d, e \rangle \langle e, d \rangle)^*) = 0$



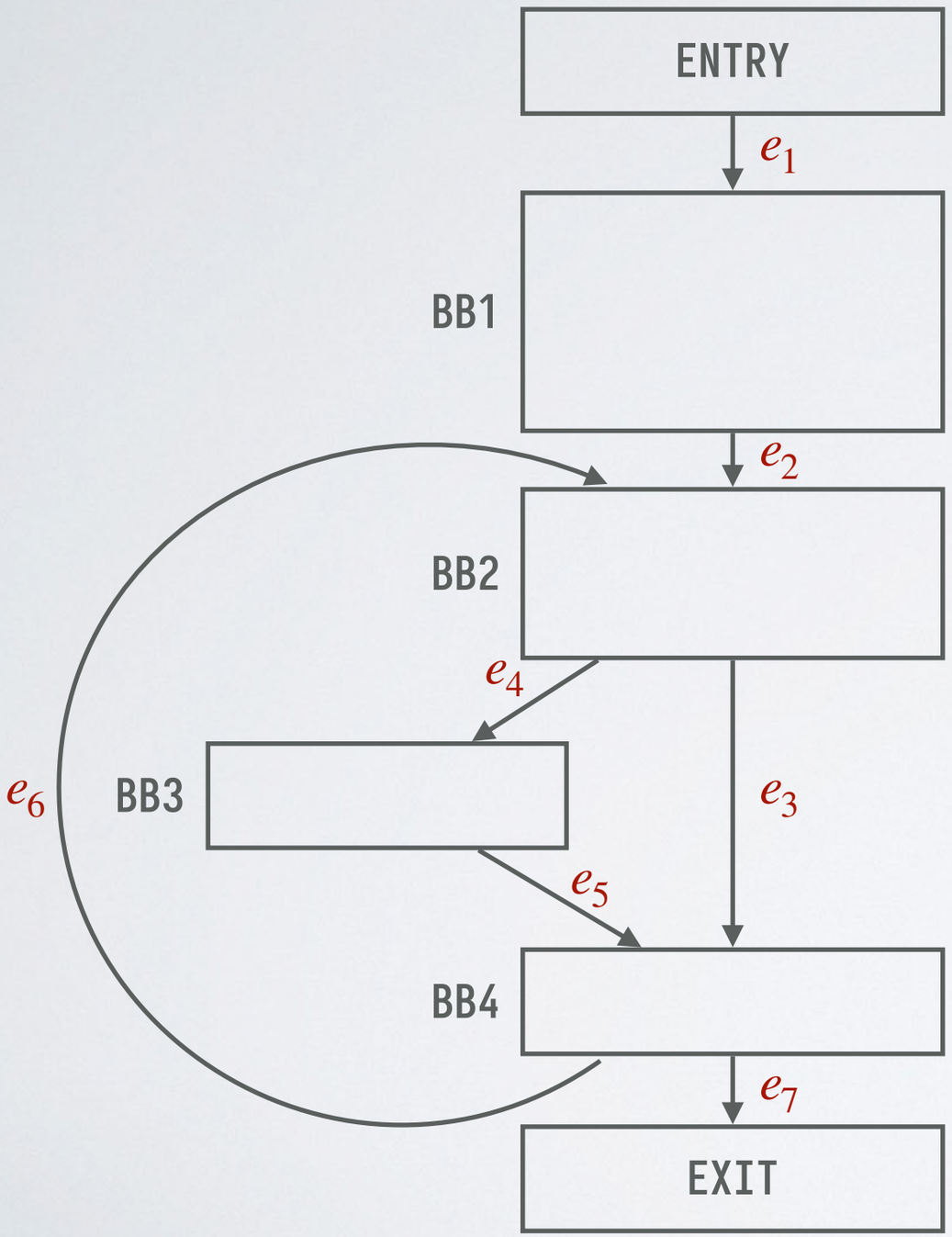
通过路径表达式求最短路（2）

$$F(\ (\langle a,b\rangle\langle b,d\rangle(\langle d,e\rangle\langle e,d\rangle)^*\langle d,a\rangle)^*\langle a,b\rangle(\langle b,c\rangle \mid \langle b,d\rangle(\langle d,e\rangle\langle e,d\rangle)^*\langle d,c\rangle)\ ) = ?$$





流图上的路径表达式



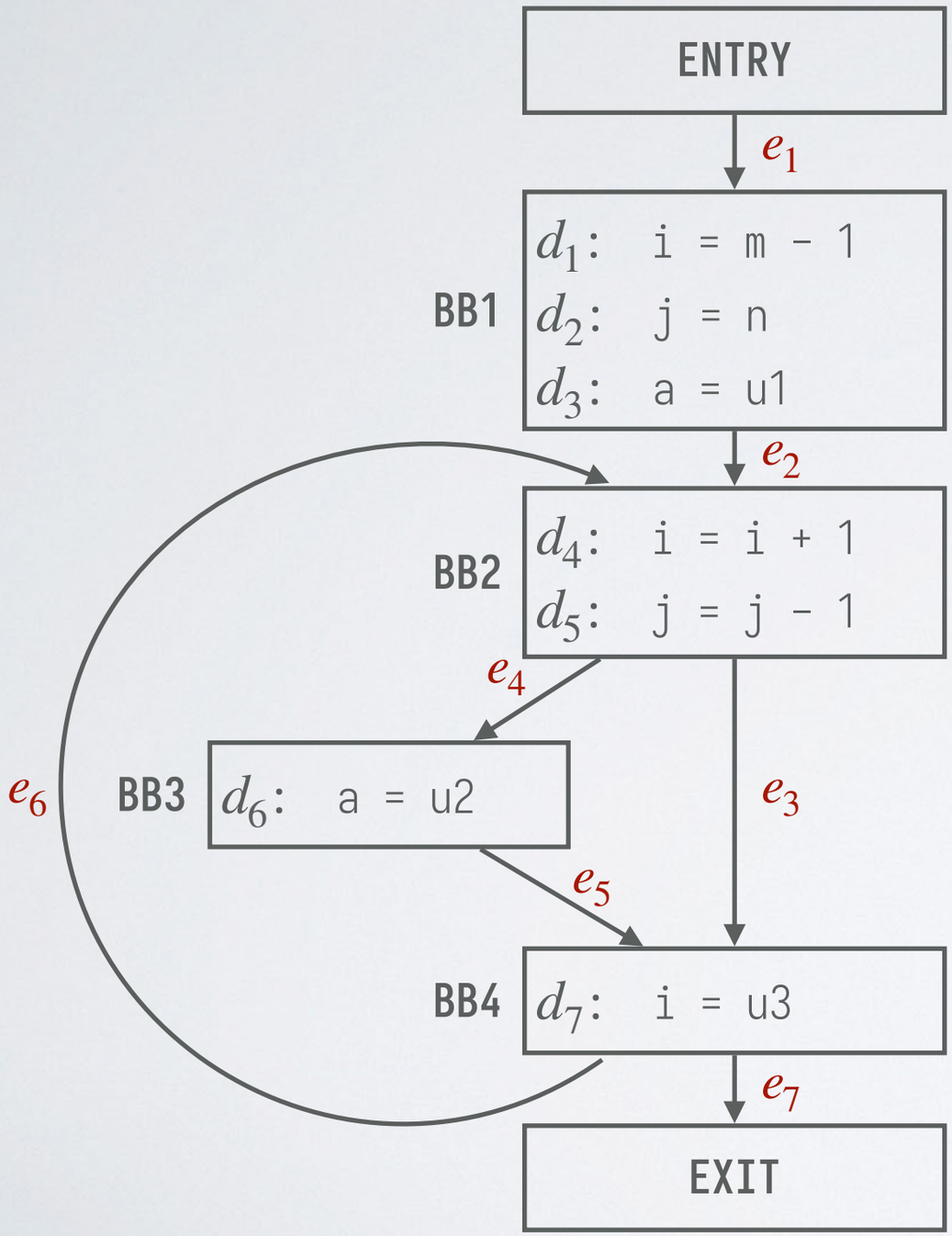
- 给每条边一个**标号**
- **前向分析**：从 **ENTRY** 出发的路径
  - **BB1**: e_1
  - **BB2**: $e_1e_2((e_3 \mid e_4e_5)e_6)^*$
  - **BB3**: $e_1e_2((e_3 \mid e_4e_5)e_6)^*e_4$
  - **BB4**: $e_1e_2((e_3 \mid e_4e_5)e_6)^*(e_3 \mid e_4e_5)$
  - **EXIT**: $e_1e_2((e_3 \mid e_4e_5)e_6)^*(e_3 \mid e_4e_5)e_7$
- **后向分析**：到达 **EXIT** 的路径
  - **BB4**: $(e_6(e_3 \mid e_4e_5))^*e_7$
  - **BB3**: $e_5(e_6(e_3 \mid e_4e_5))^*e_7$
  - **BB2**: $(e_3 \mid e_4e_5)(e_6(e_3 \mid e_4e_5))^*e_7$
  - **BB1**: $e_2(e_3 \mid e_4e_5)(e_6(e_3 \mid e_4e_5))^*e_7$
  - **ENTRY**: $e_1e_2(e_3 \mid e_4e_5)(e_6(e_3 \mid e_4e_5))^*e_7$

基于路径表达式的数据流分析

- 数据流分析的域 V，交汇运算 $\wedge : V \times V \to V$
- 每个基本块 B 的传递函数 $f_B : V \to V$
- **用 $F(r) : V \to V$ 表示 r 能识别的路径的数据流抽象**

- 以前向分析为例
  - $F(\epsilon) =$ 恒等函数，$F(e) = f_{h(e)}$ 其中 $h(e)$ 是边 e 的**起点基本块**
  - $F(r_1 | r_2) = F(r_1) \wedge F(r_2)$
  - $F(r_1 r_2) = F(r_2) \circ F(r_1)$
  - **$F(r_1^*) = \bigwedge_{i \geq 0} F(r_1)^i$，不过有时能找到更高效的算法**

例子：可达定值分析



数据流抽象：$F(r)(I) = (I - kill_r) \cup gen_r$

$gen_{e_1} = \{\}$ $gen_{e_2} = \{d_1, d_2, d_3\}$

$kill_{e_1} = \{\}$ $kill_{e_2} = \{d_4, d_5, d_6, d_7\}$

$gen_{e_3} = gen_{e_4} = \{d_4, d_5\}$ $gen_{e_5} = \{d_6\}$

$kill_{e_3} = kill_{e_4} = \{d_1, d_2, d_7\}$ $kill_{e_5} = \{d_3\}$

$gen_{e_6} = gen_{e_7} = \{d_7\}$

$kill_{e_6} = kill_{e_7} = \{d_1, d_4\}$

$gen_{e_4e_5} = \{d_4, d_5, d_6\}$

$kill_{e_4e_5} = \{d_1, d_2, d_3, d_7\}$

$gen_{e_3|e_4e_5} = \{d_4, d_5, d_6\}$

$kill_{e_3|e_4e_5} = \{d_1, d_2, d_7\}$

$gen_{(e_3|e_4e_5)^*} = \{d_4, d_5, d_6\}$

$kill_{(e_3|e_4e_5)^*} = \{\}$

不需要迭代