第六讲　运行时环境

Runtime Environments

主要内容

- 运行时环境的作用
- 运行时环境的设计
- 运行时环境的实现

- 对应章节：第 7 章

主要内容

- 运行时环境的作用
- 运行时环境的设计
- 运行时环境的实现

下面的三地址代码如何运行？

```
int fib(int n) {
  int a; a = 1;
  int b; b = 1;
  while (!(n == 0)) {
    int t;
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}

int main() {
  return fib(10);
}
```

```
  a  = 1
  b  = 1
L1:
  if n == 0 goto L0
  goto L2
L2:
  t0 = a + b
  t  = t0
  a  = b
  b  = t
  t1 = n - 1
  n  = t1
  goto L1
L0:
  return a
```

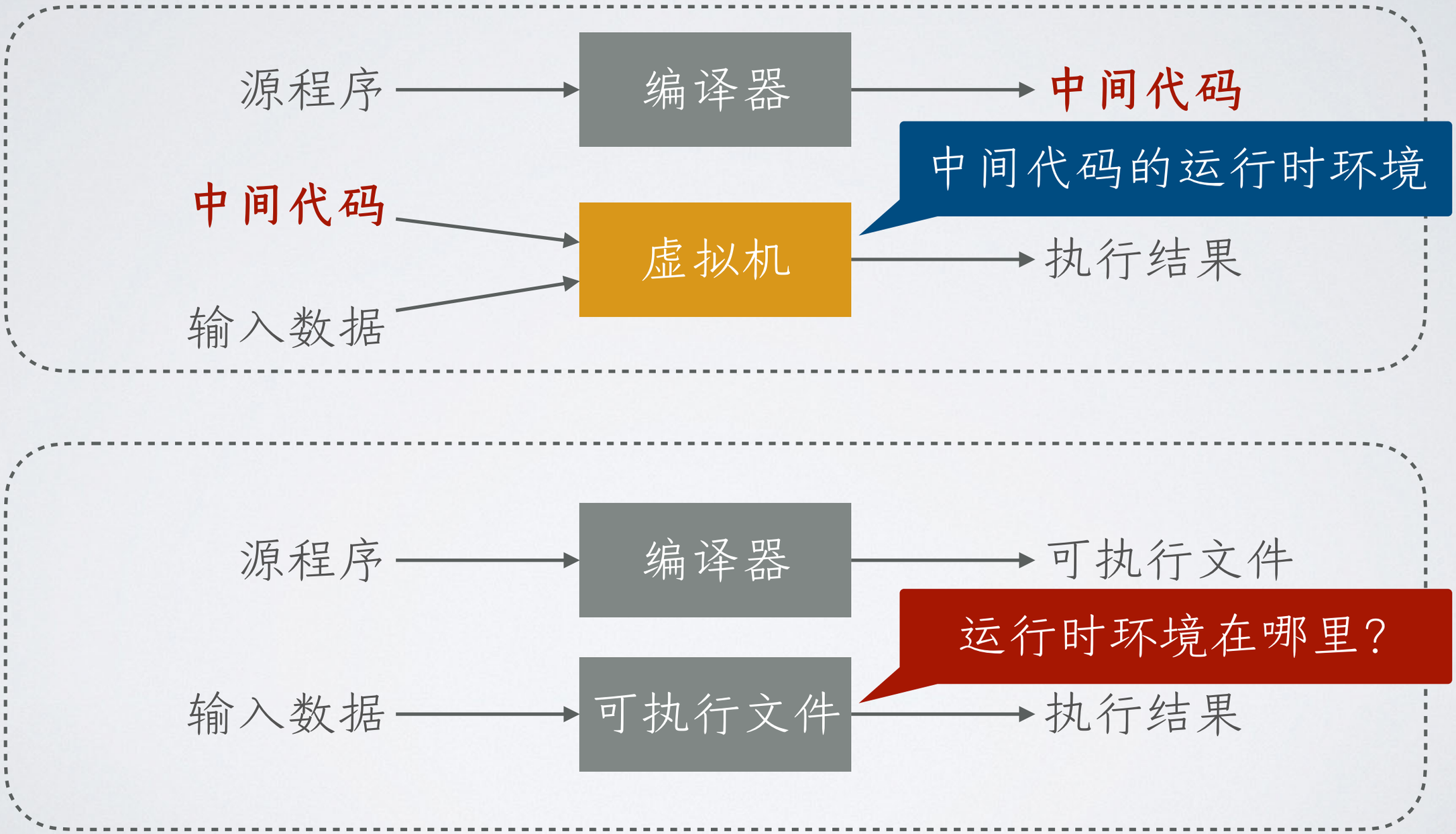
数据如何进行存储和访问？

过程/函数如何进行调用和返回？

```
param 10
t0 = call fib,1
return t0
```

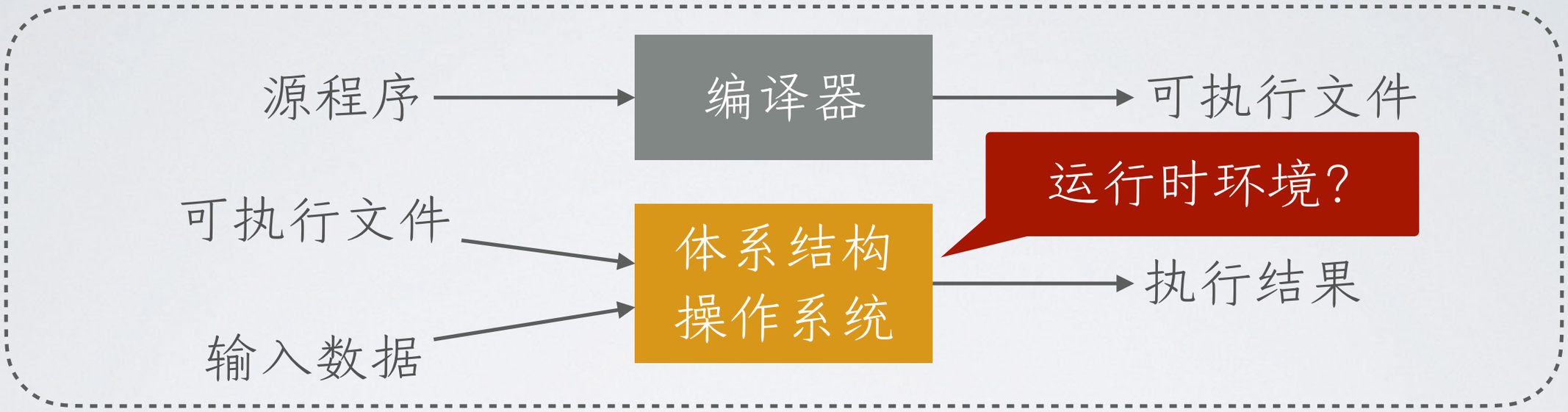
运行时环境

- 运行时环境的主要作用是实现**存储组织**和**过程抽象**



运行时环境

- 运行时环境的主要作用是实现**存储组织**和**过程抽象**



- **问题：运行时刻环境需要考虑源语言本身的特性**
  - 比如：面向对象语言需要处理类之间的继承关系、虚函数表等

- **可执行文件 =**
  源程序代表的计算 + 通过体系结构/操作系统接口实现的运行时环境

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

虚拟机实现的接口：

- vm_get(name)
- vm_set(name, value)
- vm_param(value)
- vm_call(name, nargs)
- vm_ret(value)

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a")  // 89
23. vm_ret(89)
24. vm_set("t0",89)
25. vm_get("t0") // 89
26. vm_ret(89)
```

一个运行三地址代码的虚拟机

代码

```
(1) ...
(2) ...
(3) ...
(4) ...
```

状态

pc	(1)
ra	[]
a0	0
st	[]

当前指令下标

返回地址栈

函数返回值

函数参数栈

数据

变量	值
a	0
b	0
n	0
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

虚拟机实现的接口：

- vm_get(name)
- vm_set(name, value)
- vm_param(value)
- vm_call(name, nargs)
- vm_ret(value)

如何实现接口？

一个运行三地址代码的虚拟机

代码

```
(1) ...
(2) ...
(3) ...
(4) ...
```

状态

pc	(1)
ra	[]
a0	0
st	[]

数据

变量	值
a	0
b	0
n	0
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

vm_get(name):

- 查变量表读取变量值

vm_set(name,value):

- 更改变量表中的值

vm_param(value):

- 把参数值压入参数栈

vm_call(name,nargs):

- 把函数返回后的下一条指令的下标压入返回地址栈
- 查函数表找到函数入口的第一条指令
- 根据函数表中的形参信息从参数栈弹出相应的参数值并设置变量表

vm_ret(value):

- 设置 a0 为返回值
- 从 ra 弹出返回后的指令位置

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	0
b	0
n	0
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(1)
ra	[]
a0	0
st	[]

```
1.  vm_call("main",0)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	0
b	0
n	0
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(3)
ra	[(2)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	0
b	0
n	0
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(4)
ra	[(2)]
a0	0
st	[10]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	0
b	0
n	10
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(7)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	0
n	10
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(8)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	1
n	10
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(9)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	1
n	10
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(10)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	1
n	10
t	0
t_0	0
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(11)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	1
n	10
t	0
t_0	2
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(12)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	1
n	10
t	2
t_0	2
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(13)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	1
n	10
t	2
t_0	2
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(14)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	2
n	10
t	2
t_0	2
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(15)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	2
n	10
t	2
t_0	2
t_1	9

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(16)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	1
b	2
n	9
t	2
t_0	2
t_1	9

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(17)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
```

一个运行三地址代码的虚拟机

若干轮循环后

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	89
b	144
n	0
t	144
t_0	144
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(18)
ra	[(2),(5)]
a0	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a")  // 89
23. vm_ret(89)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	89
b	144
n	0
t	144
t_0	144
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(5)
ra	[(2)]
a0	89
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a")  // 89
23. vm_ret(89)
24. vm_set("t0",89)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	89
b	144
n	0
t	144
t_0	89
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(6)
ra	[(2)]
a0	89
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a")  // 89
23. vm_ret(89)
24. vm_set("t0",89)
25. vm_get("t0") // 89
26. vm_ret(89)
```

一个运行三地址代码的虚拟机

代码

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

数据

变量	值
a	89
b	144
n	0
t	144
t_0	89
t_1	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

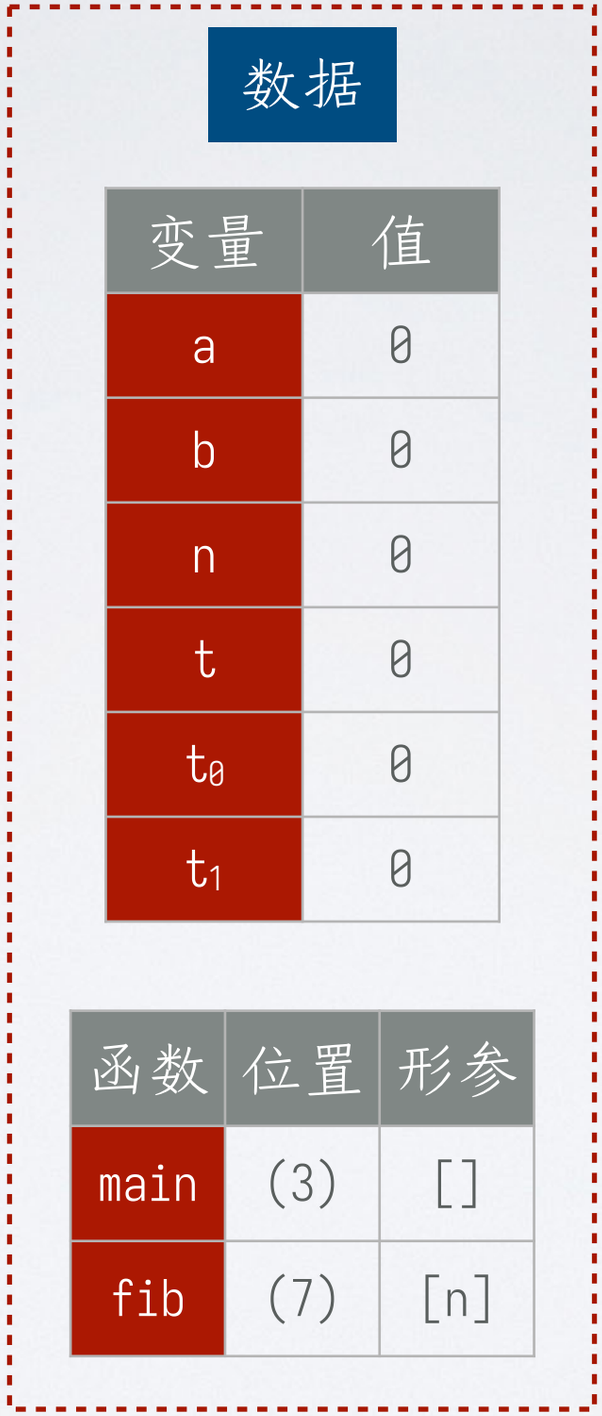
状态

pc	(2)
ra	[]
a0	89
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")  // 10
8.  vm_get("a")  // 1
9.  vm_get("b")  // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")  // 1
```

```
14. vm_set("a",1)
15. vm_get("t")  // 2
16. vm_set("b",2)
17. vm_get("n")  // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a")  // 89
23. vm_ret(89)
24. vm_set("t0",89)
25. vm_get("t0") // 89
26. vm_ret(89)
```

一个运行三地址代码的虚拟机



这个设计有什么问题？

所有变量都处于同一个全局作用域中！

- 可能与源语言的作用域规则不一致
- 也不支持递归过程/函数的实现

为支持存储组织和过程抽象，需要对运行时环境进行细致的设计

从三地址代码到目标代码

- **可执行文件 =**

  源程序代表的计算 + 通过体系结构/操作系统接口实现的运行时环境

```
(1)   call main,0
(2)   halt
(3)   param 10
(4)   call fib,1
(5)   retval t0
(6)   return t0
(7)   a  = 1
(8)   b  = 1
(9)   if n == 0 goto (18)
(10)  goto (11)
(11)  t0 = a + b
(12)  t  = t0
(13)  a  = b
(14)  b  = t
(15)  t1 = n - 1
(16)  n  = t1
(17)  goto (9)
(18)  return a
```

RISC-V 汇编

```
  .data
a:
  .zero 4
b:
  .zero 4
n:
  .zero 4
...
```

数据段：存储全局变量

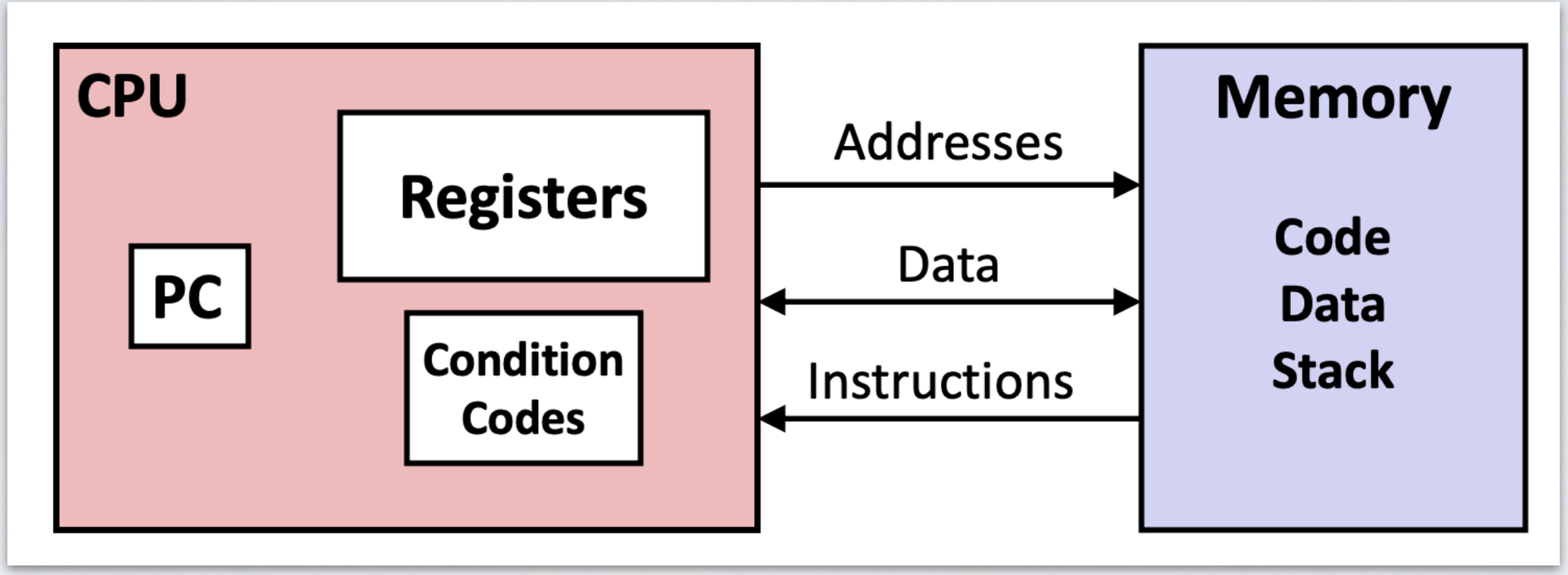
```
  .text
main:
  li a0, 10
  call fib
  ret
fib:
  la t6, n
  sw a0, 0(t6)
...
```

代码段：存储程序代码

RISC-V 调用约定：寄存器 a0 被用来传递第一个参数以及返回值

RISC-V 体系结构：寄存器 ra 被用来记录函数的返回地址

从三地址代码到目标代码



- 体系结构和操作系统提供了**非常底层**的操作
- 运行时环境用这些操作来实现数据存储和过程调用
- **需要在生成的目标代码中插入这些底层操作**
  - 本地(native)运行 ≈ 用机器指令「模拟」虚拟机

「机器无关」的运行时环境

- 本讲内容**不限定**本地运行或虚拟机运行，也**不限定**具体的体系结构或操作系统
- 本讲内容着重介绍设计和实现运行时环境的**通用技术**
- **重点：存储组织和过程抽象**

主要内容

- 运行时环境的作用
- **运行时环境的设计**
- 运行时环境的实现

存储组织

- 在代码生成前，编译器需要进行**目标运行环境的设计**和**数据空间的分配**

- 编译器在操作系统/虚拟机规定的区域中存储生成的**目标代码**与代码运行时的**数据空间**
  - 比如 RISC-V 中，.text 段存储代码，.data 段存储全局变量等

- 需要存储空间的对象：
  - 源代码声明的各种类型的数据对象
  - 用来保留中间结果和传递参数的临时数据对象
  - 过程调用时需要记录的上下文信息
  - ……

静态和动态存储分配

- **难点：**区分程序的**编译**时刻和**运行**时刻
- 编译时刻对应**静态分配**
  - 编译器通过程序文本即可做出分配决定
  - 例如：常量、全局变量、静态变量(C 中的 static 变量)
- 运行时刻对应**动态分配**
  - 程序在运行过程中才能做出分配决定
  - 例如：局部变量、动态变量(C 中的 malloc 函数分配的数据)
- **注意：**静态确定的存储空间大小**并不意味**静态分配
  - 很多时候空间大小可以由类型信息得出

纯静态存储分配

- 所有分配决定都在编译时得到
- **优点：不需要运行时的支持**
- **缺点：不支持递归调用过程，不能动态建立数据结构**

- 案例：标准 Fortran 语言使用纯静态存储分配

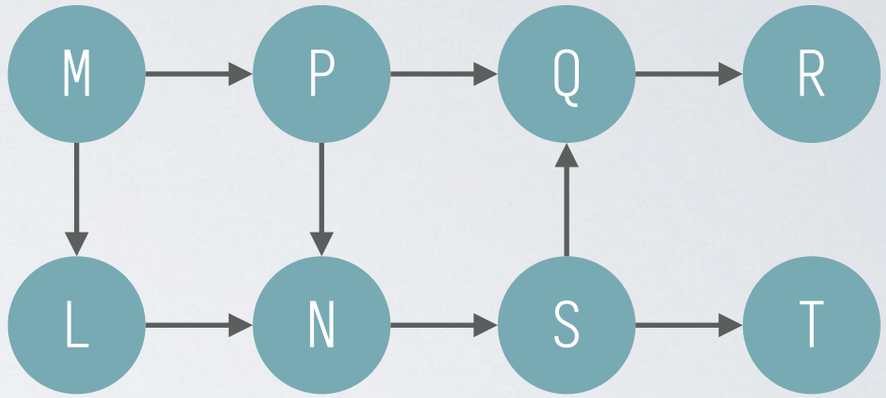
Fortran 语言的存储组织

过程列表

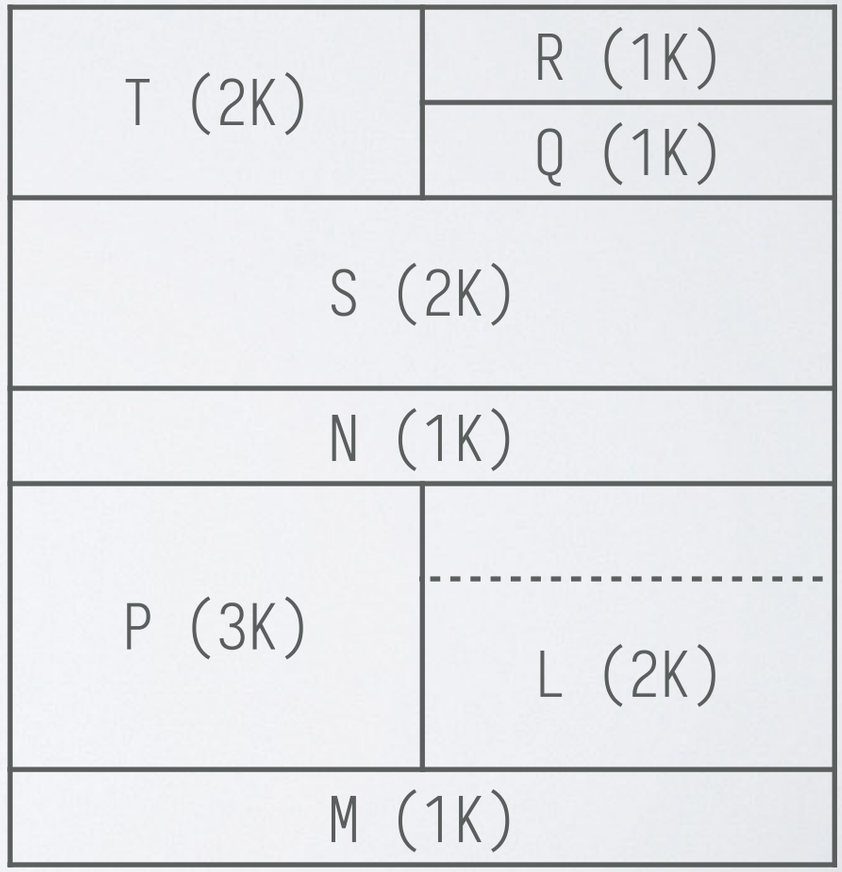
过程名	空间大小
M	1K
P	3K
Q	1K
R	1K
L	2K
N	1K
S	2K
T	2K

M 为主过程

调用关系



存储分配



分时复用！

如何支持递归调用过程？

- 过程调用的次数一般不能静态确定，所以需要**动态分配**
- 过程调用在时间上总是嵌套的
  - 后调用的先返回(last-in-first-out, LIFO)
  - 适合使用**栈(stack)**来管理过程活动的存储空间
  - 例如：过程的局部变量、形式参数等，这些值**与过程的生命周期相同**
- **栈式存储管理**

如何支持动态建立数据结构？

- 动态数据结构的大小一般不能静态确定，所以需要**动态分配**
- 其生命周期可能并不和某个过程的调用/返回相匹配
  - 数据对象可能比创建它的过程调用更长寿
  - **非词法作用域(non-lexical scope)**

- **堆(heap)式存储管理**
  - 不是数据结构中的堆
  - 可以理解为**「可用内存池」**

过程抽象

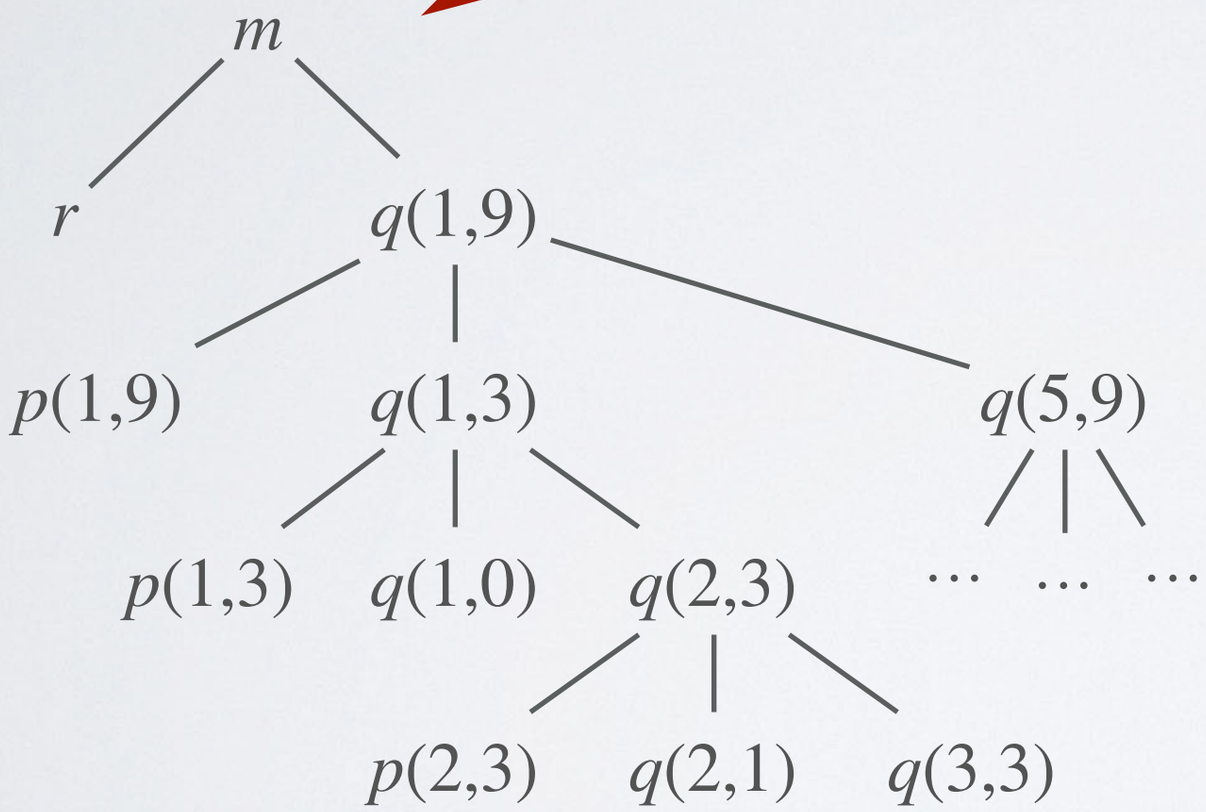
- **所谓「抽象」，不过是隐藏一定的「细节」**

- 多数现代编程语言都提供了过程抽象
  - 通过定义过程/函数的形式隐藏其代表的计算的**细节**
  - 每个过程/函数对外部提供的**抽象**则是其名字、参数、类型声明等

- **活动树**(activation tree)：程序运行的过程活动可以用树表示
  - 每个结点对应于一个过程活动
  - 根结点对应于主过程(或入口过程)的活动
  - 过程 p 的某次活动对应的结点的所有子结点：此次活动所调用的各个过程活动(从左到右，表示调用的先后顺序)

活动树示例

问：前序遍历和后序遍历分别对应什么？

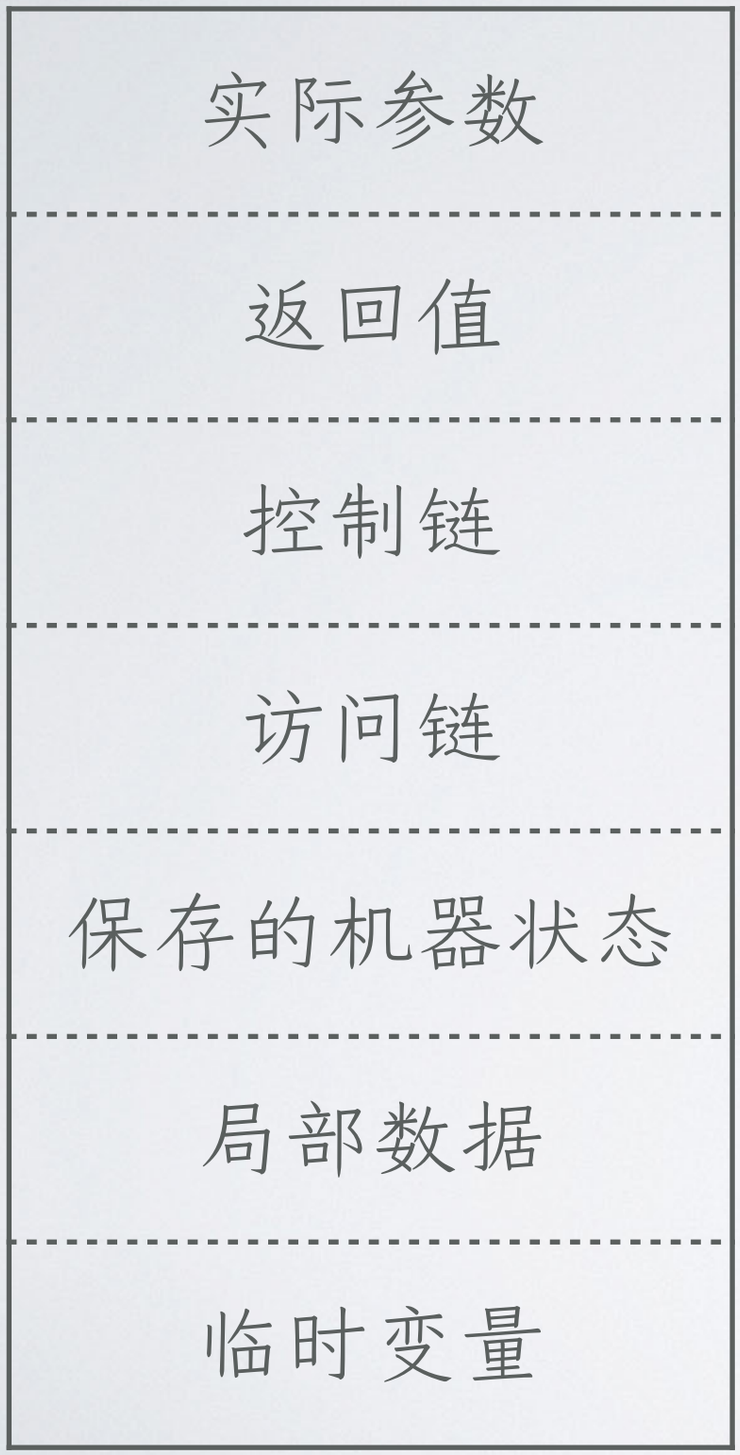



```
enter main()
  enter readArray()
  leave readArray()
  enter quickSort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quickSort(1,3)
      …
    leave quickSort(1,3)
    enter quickSort(5,9)
      …
    leave quickSort(5,9)
  leave quickSort(1,9)
leave main()
```

活动记录

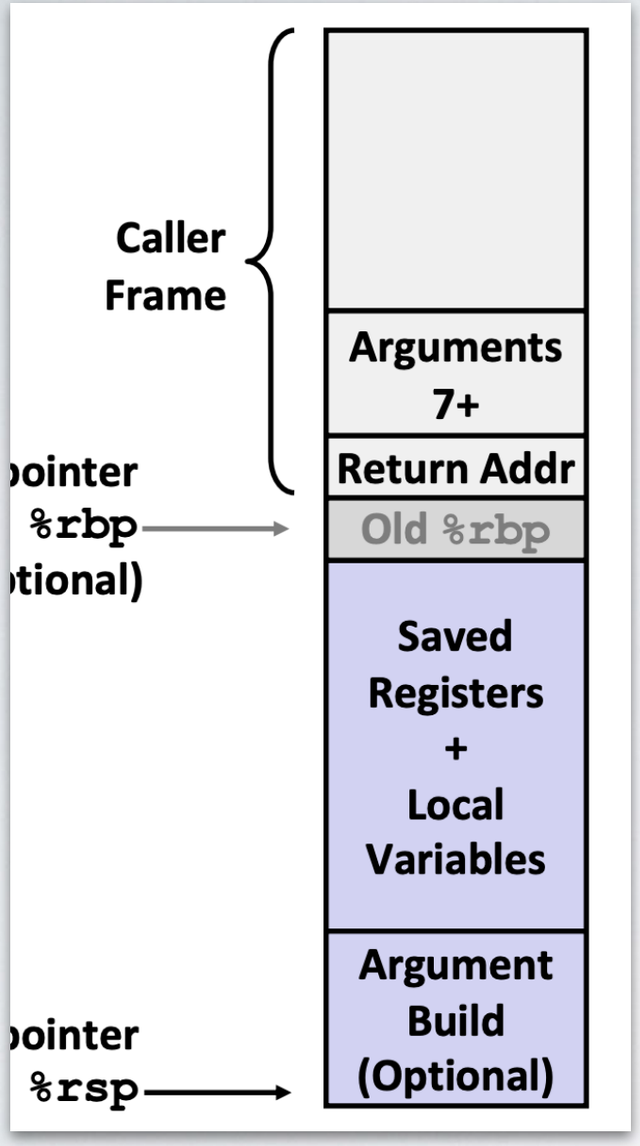
- 子程序(过程或函数)的一次运行(由于被调用而引发),称为子程序的一次**活动(activity)**
- 子程序运行时所用到的局部信息(比如局部变量)存储在一个地址连续的存储块内,这个存储块称为**活动记录(activation record)**,又称**帧(frame)**
- 本地运行:
  - 现代体系结构通常提供**栈式**活动记录管理,比如 x86、RISC-V 等
    - **栈帧(stack frame)**
- 虚拟机运行:
  - 虚拟机自身通过数据结构进行**栈式或堆式**管理,比如 Java、Lua 等

活动记录的结构



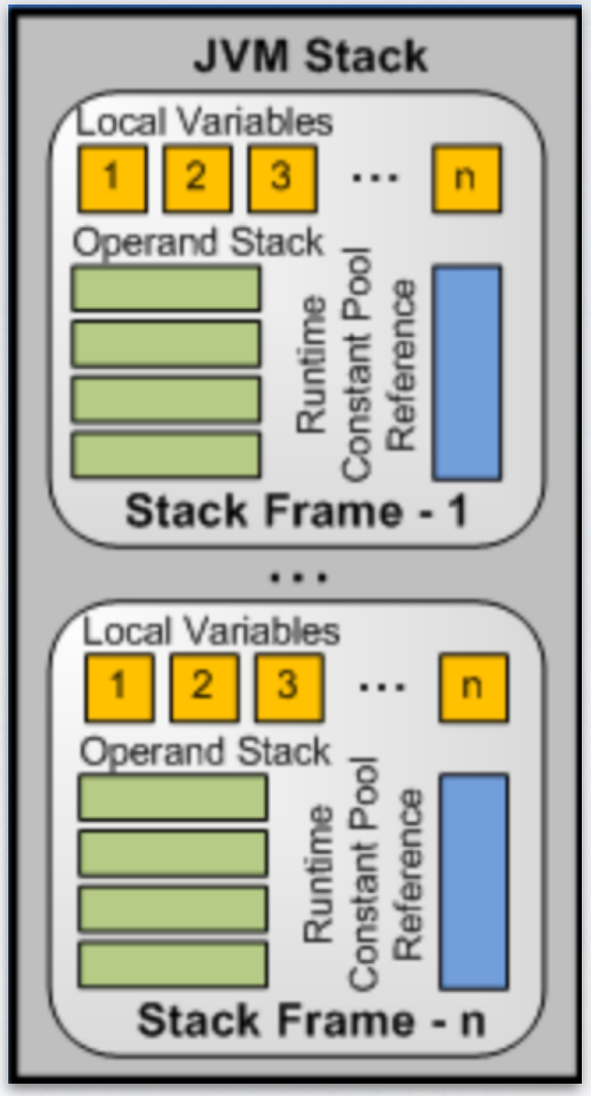
- **保存的机器状态:** 此次调用之前的机器状态信息，包括**返回地址**
- **局部数据:** 过程中声明和使用的局部变量
- **临时变量:** 中间代码或目标代码生成时产生的临时值
- **控制链:** 指向调用者的活动记录
- **访问链:** 指向过程中要访问的**非局部数据**所在的活动记录

案例：本地运行和虚拟机运行的帧



- 前 6 个参数通过寄存器传递，更多的则放在调用者的栈帧里
- 返回地址也放在调用者的栈帧里
- 返回值通过寄存器传递
- 当前栈帧可能存放的 Old %rbp 可认为是实现了控制链

x86-64/Linux

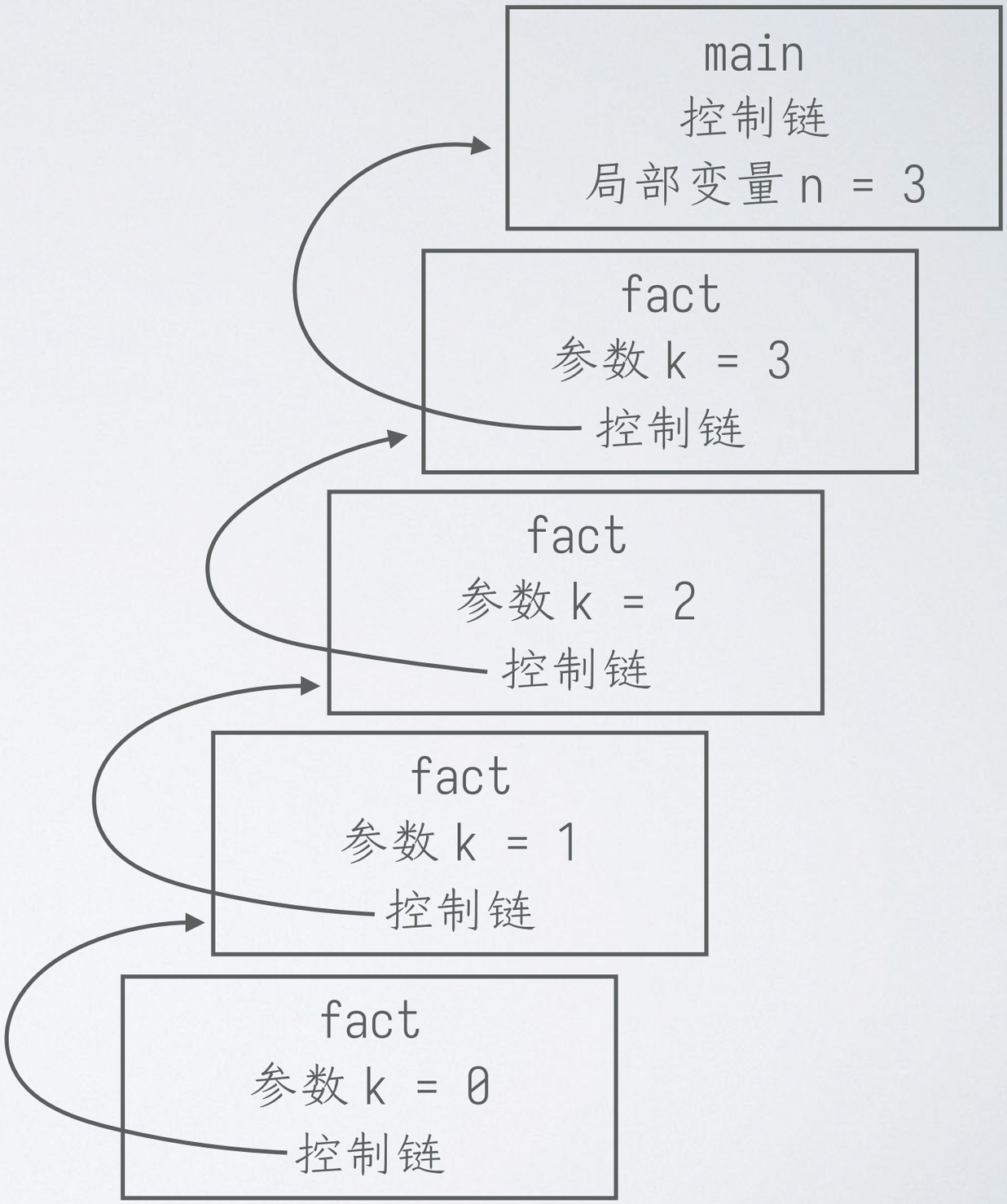


- JVM 没有寄存器，运算通过栈帧中的操作数栈(operand stack)进行
- 返回值会在返回后压入调用者的操作数栈
- 栈帧中还存储了一些跟动态链接相关的数据结构

Java VM

活动记录示例

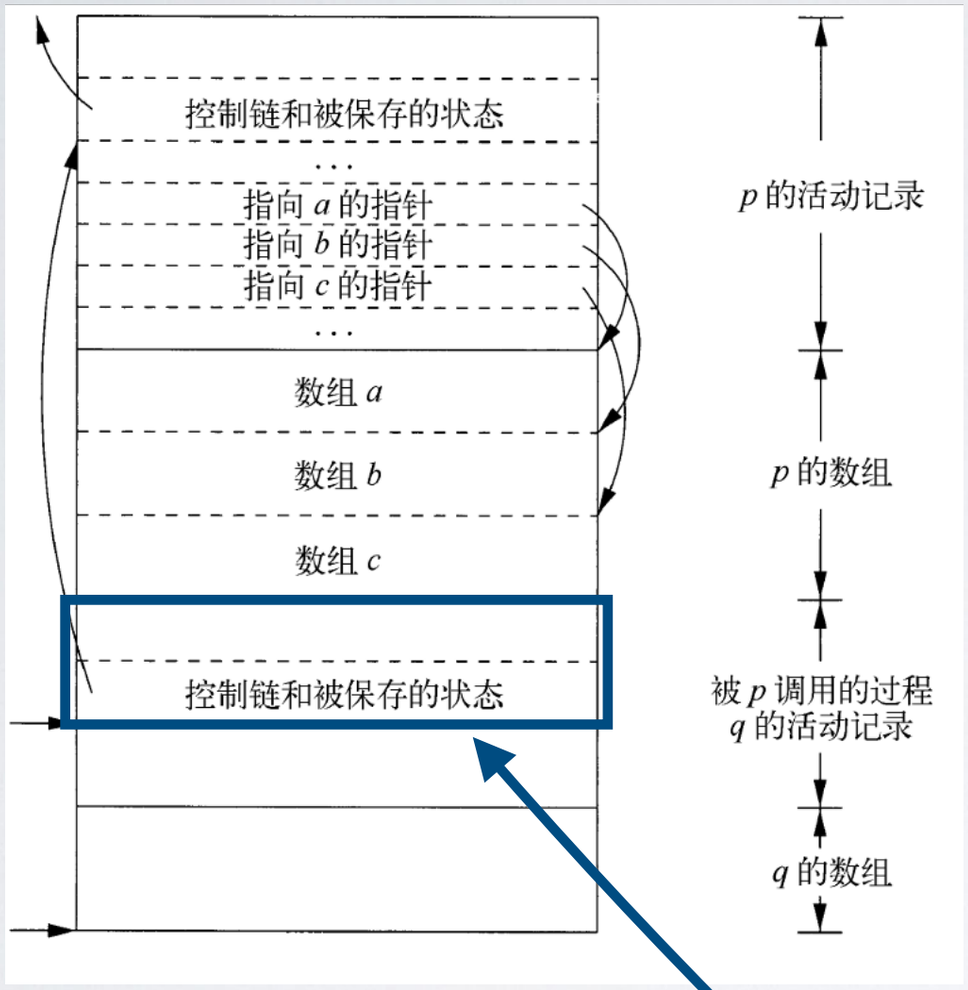
```
int fact(int k) {
  if (k == 0)
    return 1;
  else
    return k * fact(k - 1);
}

void main() {
  int n;
  printf("Enter a number:");
  scanf("%d", &n);
  printf("Results = %d\n",
         fact(n));
}
```



活动记录指针

- 运行时环境中通过**活动记录指针**(activation-record pointer)指向正在运行的当前过程的活动记录
- 可以设计 ARP 为指向活动记录中**固定长度**数据的末端
  - 编译器可静态计算得出长度
  - 生成的代码可以根据相对 ARP 的偏移来获取这些数据(比如参数、控制链等)
- 这样的设计也支持运行时**动态确定局部数据的长度**

栈帧中的变长局部数据



栈式活动记录管理：运行时环境通常会维护一个栈顶指针 **TOP**

- 如果数据对象的生命周期局限于过程活动的生命周期，就可以分配在运行时刻栈中
- **变长数组**也可以放在栈中
- **问：**被调用过程 q 返回时，如何恢复调用过程 p 对应的 **ARP** 和 **TOP** 指针？
  - **ARP**：读取控制链存放的指针
  - **TOP**：把（未恢复的）**ARP** 与被调用过程 q 的**固定长度字段**的长度相加

数据的存储与访问

常量、全局变量的存储与访问

- 运行时环境中开辟一块存储空间来存取这些数据
- 虚拟机可以维护常量表，并设计读取常量表的指令
- RISC-V 支持 .data 段放全局变量，并配合 la/lw/sw 等指令进行存取

过程活动中局部变量的存储与访问

- 活动记录中的局部数据块可用于存储局部变量的数据
- 虚拟机可以在活动记录中维护变量名到值的映射表，并设计读取局部变量表的指令
- RISC-V、x86 等采取栈式活动记录管理，在栈上存储局部变量，访问可以通过相对 **ARP** 或 **TOP** 的偏移来进行

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

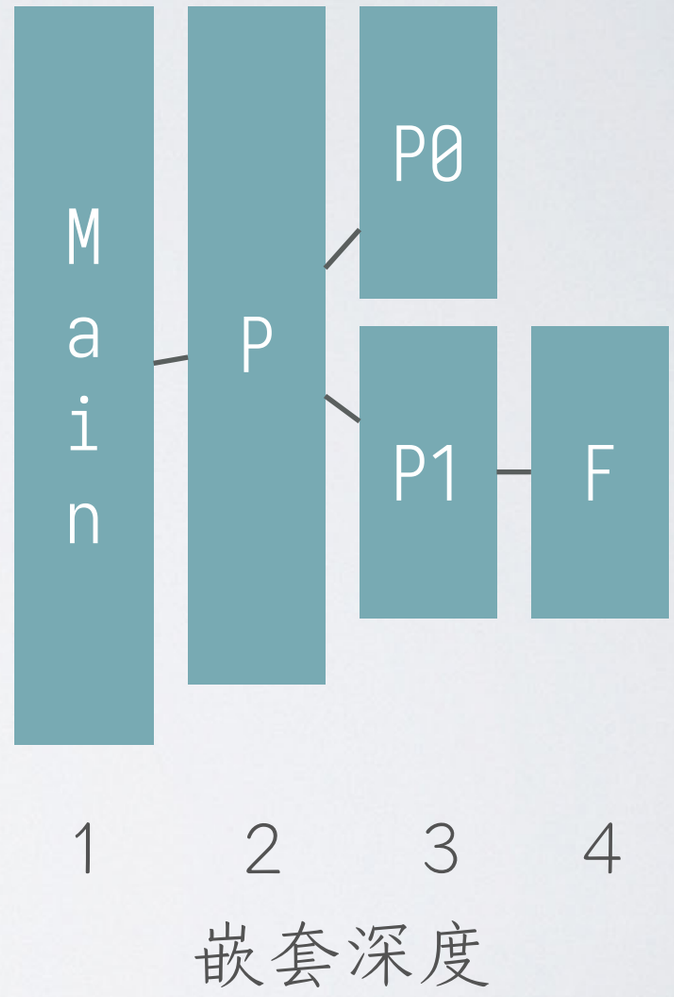
静态作用域

- 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

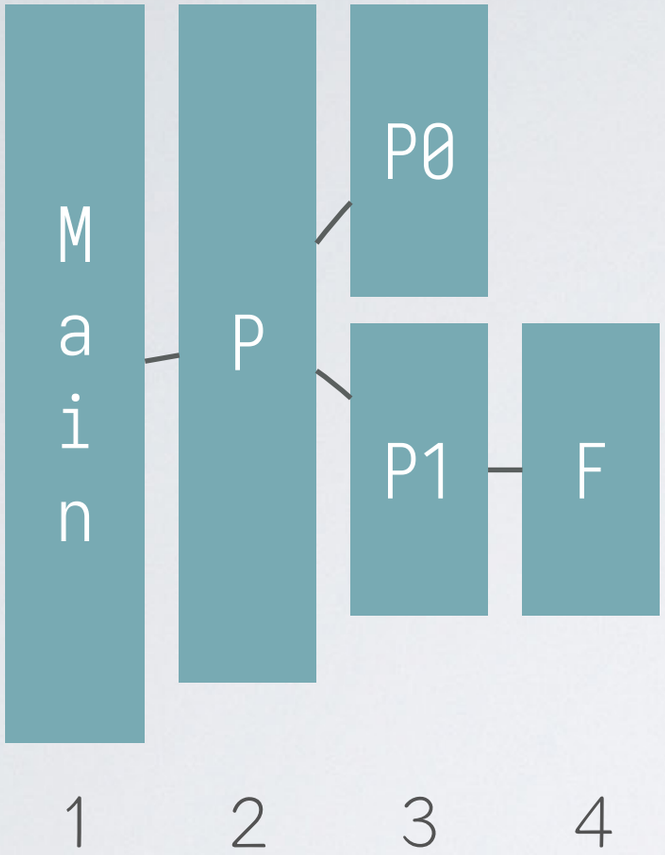
```
procedure P(px: integer, py: integer);
  var i: integer; r: integer;
  procedure P0(p0x: integer);
    begin … end;
  procedure P1(p1x: integer, p1y: integer);
    var j: integer;
    function F(fx: integer): integer;
      var x: integer;
      begin … P0(x) … p1x … px + j … end;
    begin … F(p1y)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

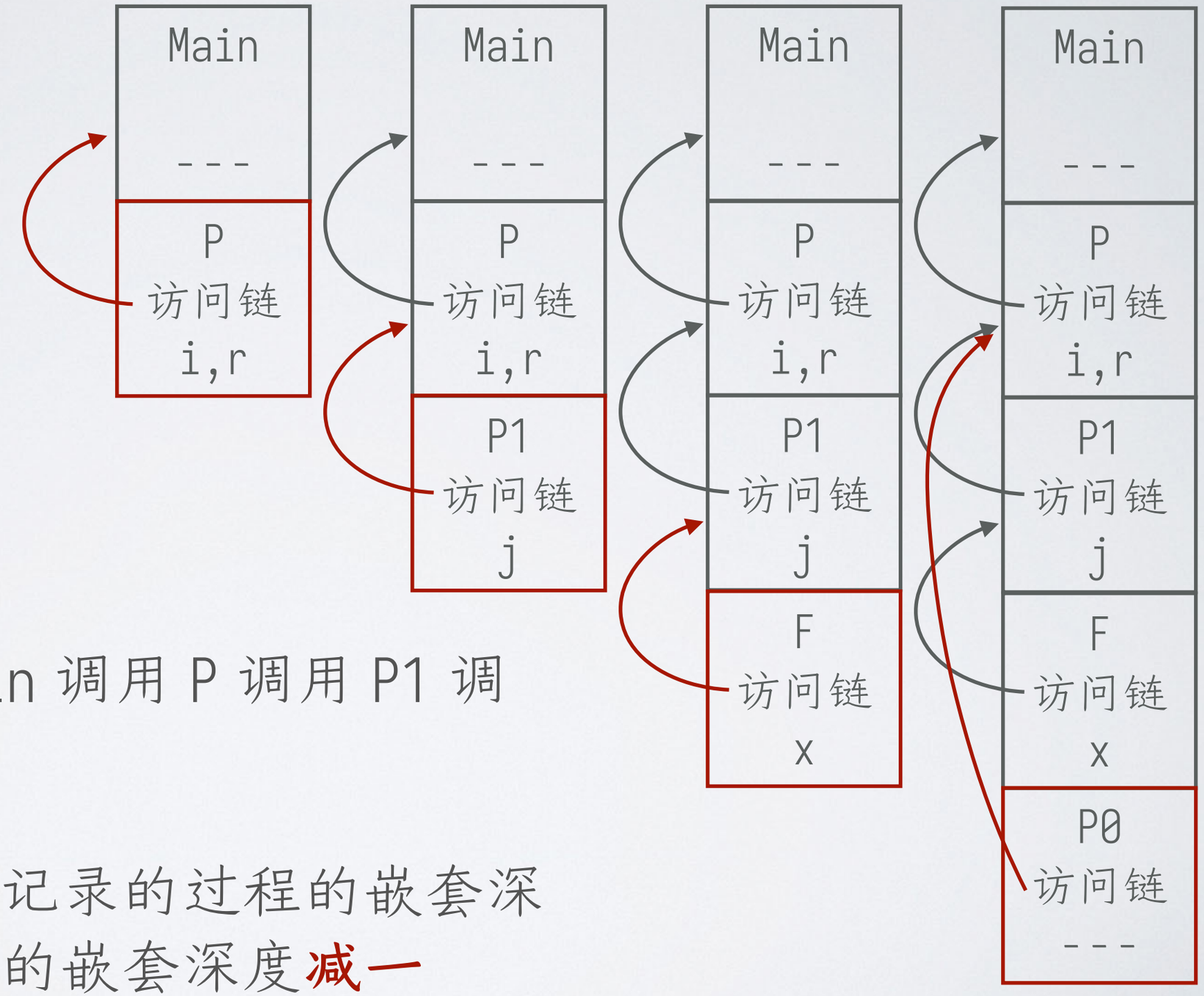
如何支持静态作用域？

- **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





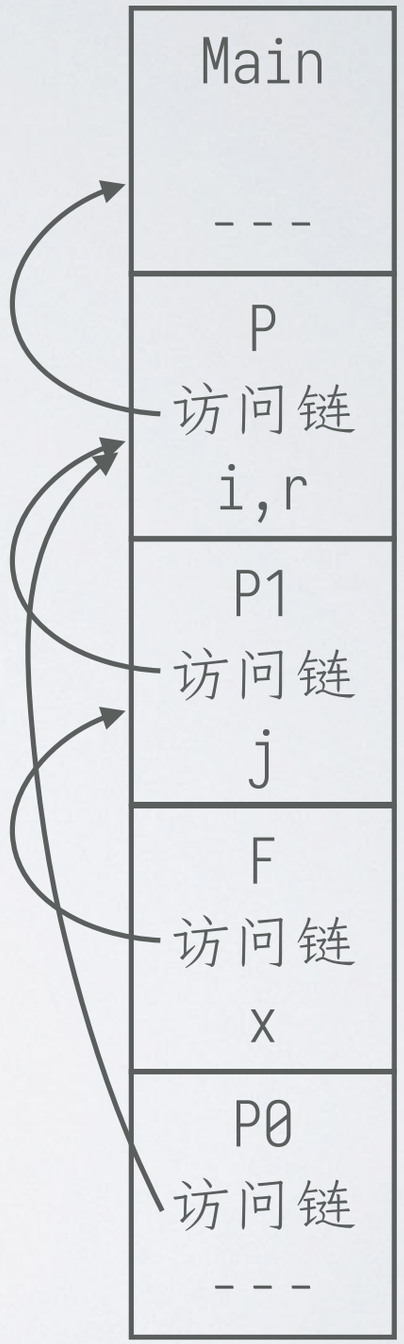
- 考虑过程活动：Main 调用 P 调用 P1 调用 F 调用 P0
- 访问链指向的活动记录的过程的嵌套深度一定是当前过程的嵌套深度**减一**

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

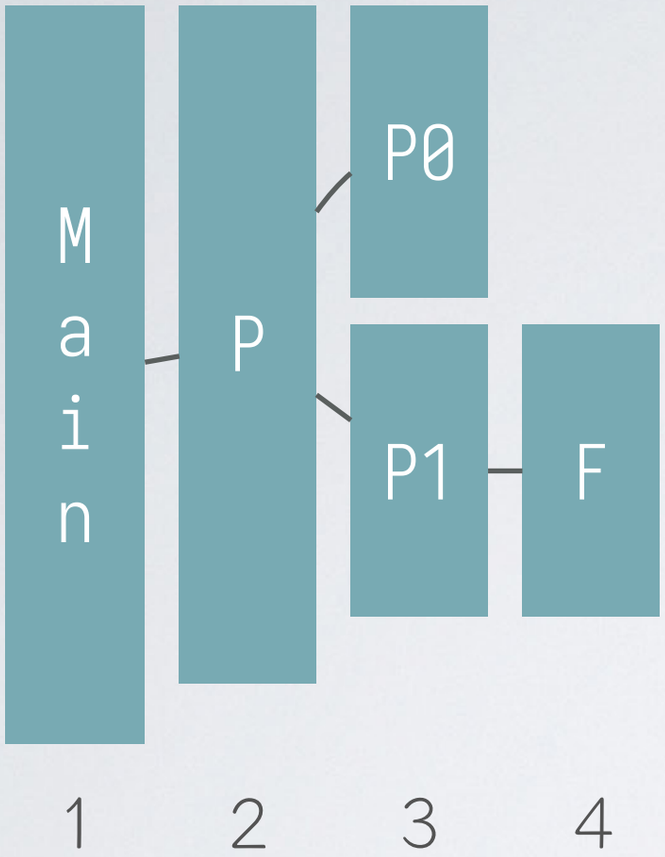
- 从嵌套深度 m 的过程访问嵌套深度 n 的过程中的数据($n \leq m$):
  - 追踪访问链 $m-n$ 次，到达目标活动记录
  - $m-n$ 的值可以在编译时确定
- **以右图为例:**
  - 对于过程 F 而言:
    - 访问 x: 追踪 0 次
    - 访问 j: 追踪 1 次
    - 访问 i 和 r: 追踪 2 次
  - 对于过程 P0 而言:
    - 访问 i 和 r: 追踪 1 次
    - **不能**访问 x 或 j

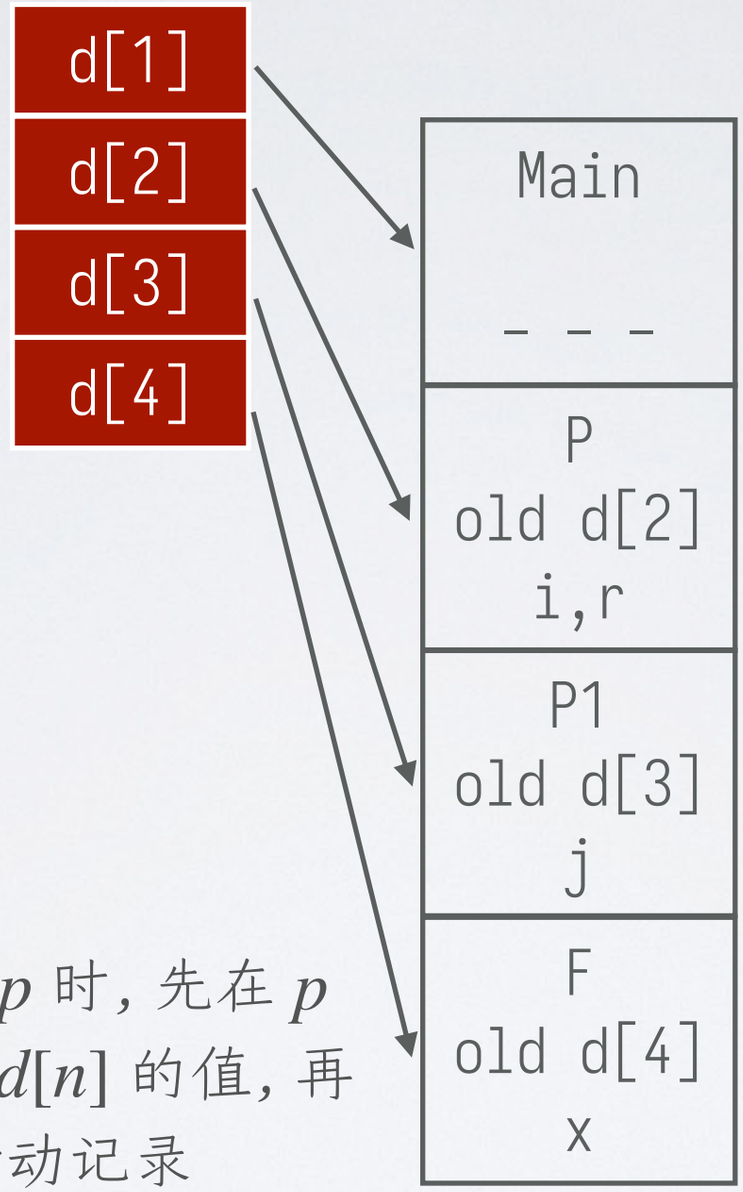


如何支持静态作用域？

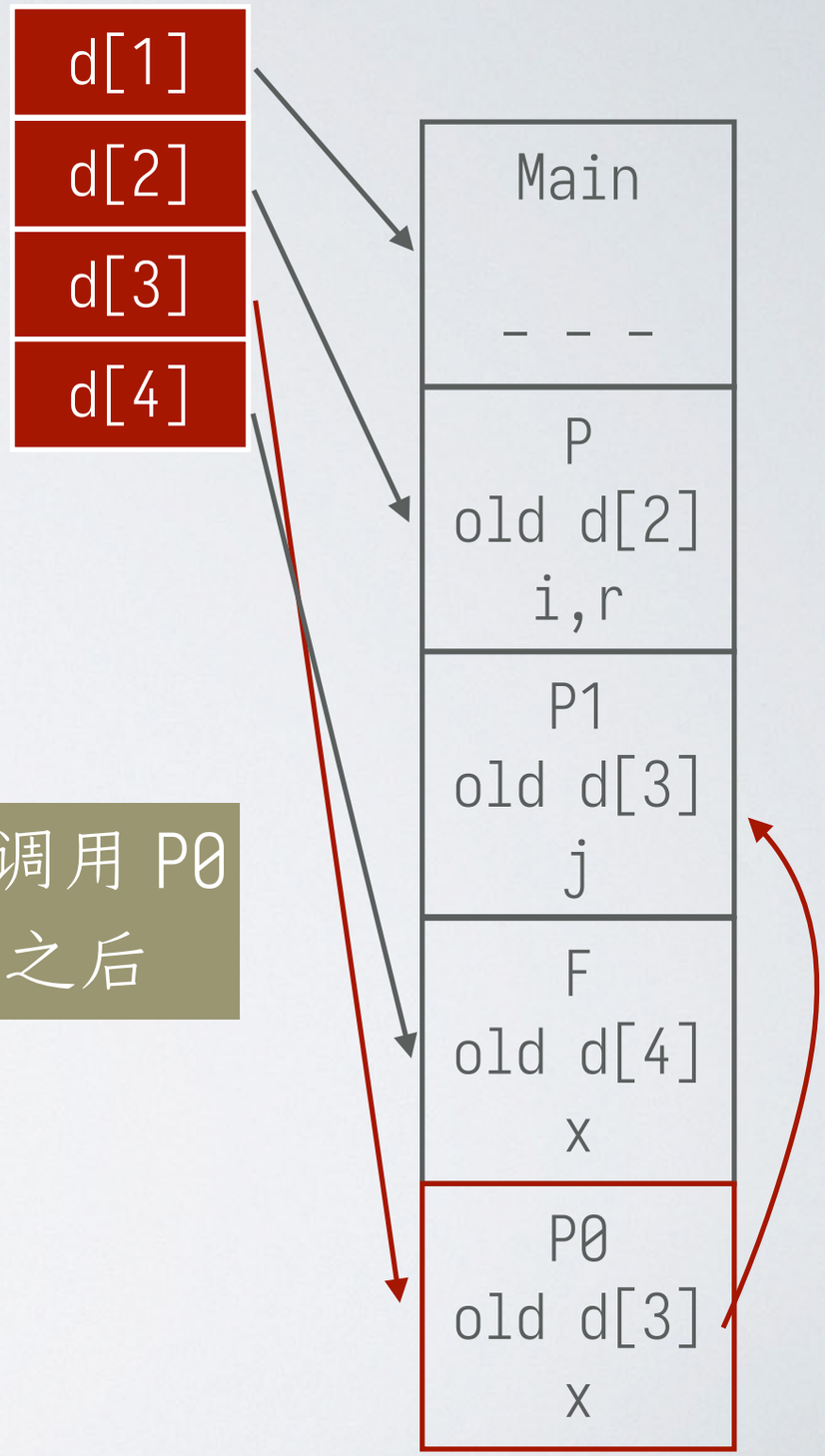
- **问题：用访问链时，访问数据的开销和嵌套深度差有关**
- 方法二：**显示表法**
- **显示表(display)**：运行时环境维护一个数组 d，为每个嵌套深度记录一个指针
  - 指针 $d[i]$ 指向**最近**的嵌套深度为 i 的活动记录
  - 如果过程 p 在运行中访问嵌套深度为 i **(静态可确定)**的过程 q 的数据，则可以通过 $d[i]$ 找到 q 的活动记录
  - 使用显示表可以提高效率，访问开销是常数

显示表示例





F 调用 P0 之后



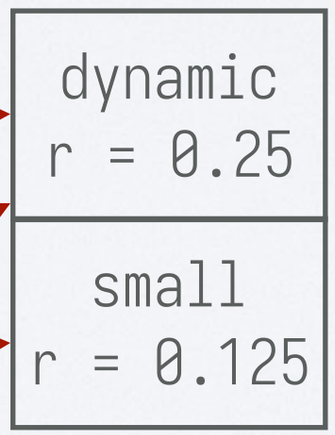
- 调用嵌套深度为 n 的过程 p 时，先在 p 的活动记录中保存原有的 $d[n]$ 的值，再将 $d[n]$ 指向调用的 p 的活动记录
- 从过程 p 返回时，通过活动记录中保存的值恢复之前的 $d[n]$

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
```

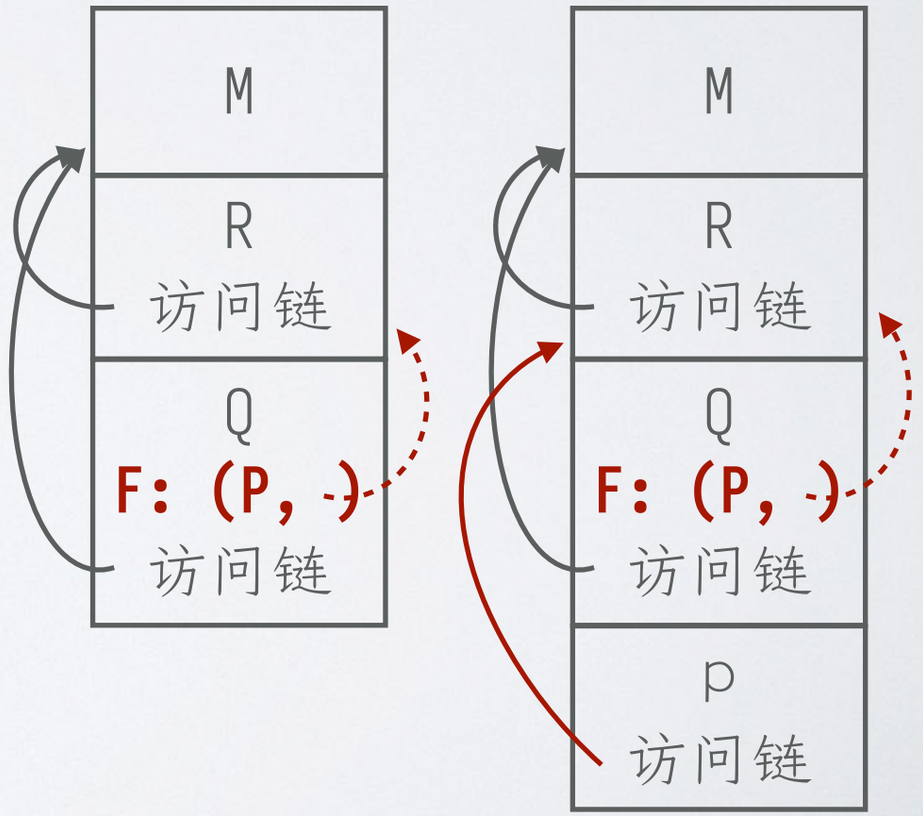
- 静态作用域下输出：

```
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**

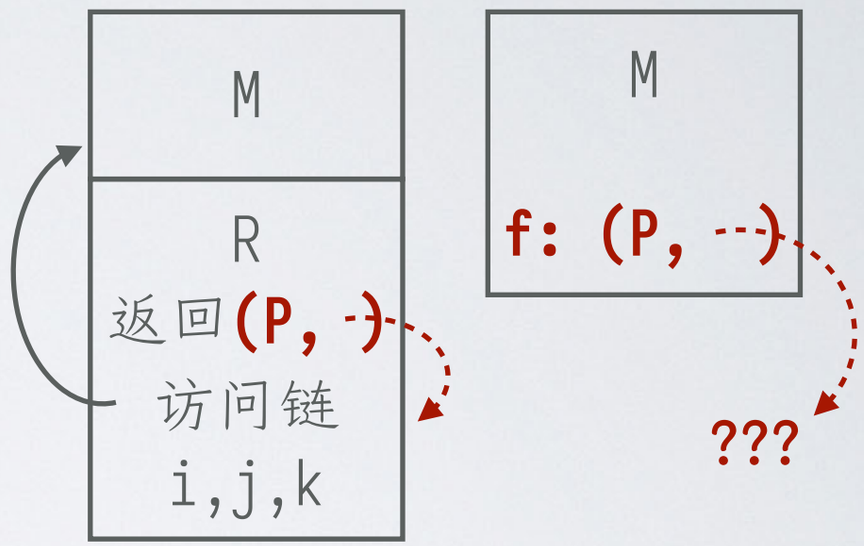
```
procedure M(x: integer);
  procedure Q(F: integer -> integer);
    begin … F(…) … end;
  procedure R(y: integer);
    function P(z: integer): integer;
      begin … end;
    begin … Q(P) … end;
  begin
    … R(…) …
  end;
```



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;
```

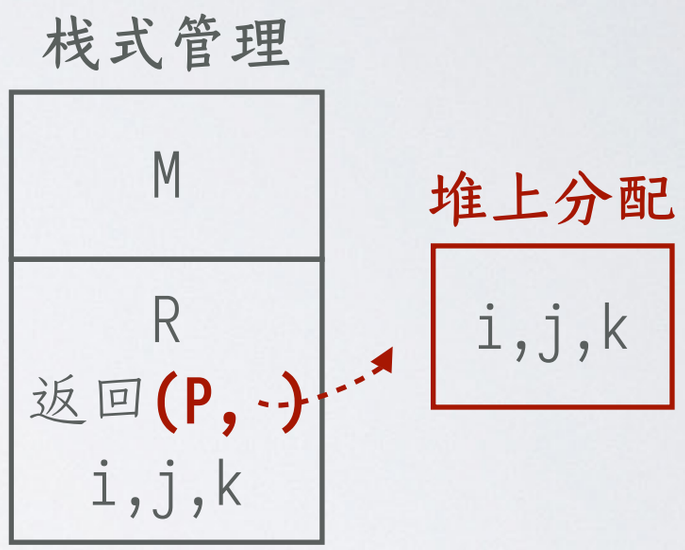


- 栈式活动记录管理不再适合！
- 方法一：完全在堆中分配和管理活动记录
- 方法二：闭包

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

- 运行时环境的作用
- 运行时环境的设计
- **运行时环境的实现**

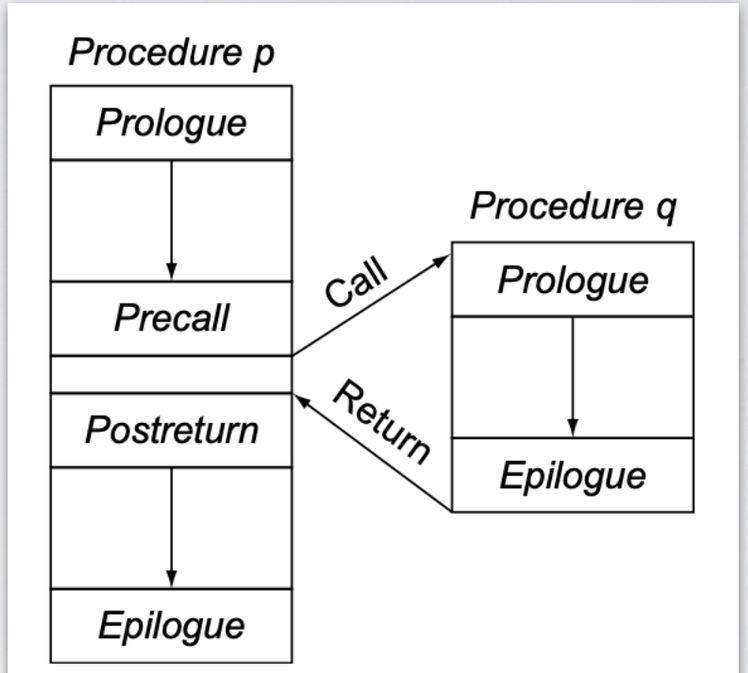
过程抽象的实现

- **问题：如何创建和维护活动记录？**
- 如果是实现虚拟机……
  - 你想怎么实现就怎么实现！
- 如果是需要生成目标代码？
  - **需要和操作系统和体系结构达成一致**
  - 在目标代码中插入指令实现运行时环境

- **总体策略：**
  - **调用代码序列：**调用者中的 precall 和被调用者的 prologue
  - **返回代码序列：**被调用者的 epilogue 和调用者中的 postreturn

过程链接

- **调用代码序列**：给被调用过程的活动记录分配空间，填写记录中的信息
  - 分割为 precall 和 prologue
- **返回代码序列**：释放被调用过程的活动记录，恢复机器状态，返回到调用过程中继续执行
  - 分割为 epilogue 和 postreturn
- 根据源语言、体系结构、操作系统的规定，可以有不同的分割方案
  - **如果调用者做得多**：生成的代码会比较长
    - 每处调用都需要重复生成
  - **如果被调用者做得多**：可能会有冗余的存储操作
    - 考虑被调用者保存(callee-saved)寄存器



一种调用代码序列的设计

- 调用者中的 precall：
  - 计算实际参数的值，并存入被调用者的活动记录
  - 保存机器状态信息(比如 caller-saved 寄存器)
  - 将返回地址和原来的 ARP 指针存入被调用者的活动记录，并更新 ARP 指向被调用者的活动记录
- 被调用者的 prologue：
  - 保存机器状态信息(比如 callee-saved 寄存器)
  - 初始化其局部数据并开始执行
- **问：上述过程与 x86-64/Linux 的规定有什么异同？**

一种返回代码序列的设计

- 被调用者的 epilogue：
  - 在自己的活动记录的返回值字段中放一个返回值
  - 利用保存的控制链和机器状态的信息，恢复 **ARP** 和其它相关状态(比如 callee-saved 寄存器)
  - 按照返回地址转移到调用者的代码之中
- 调用者中的 postreturn：
  - 从被调用者的活动记录中获取返回值
  - 利用保存的机器状态的信息恢复相关状态(比如 caller-saved 寄存器)
- **问：上述过程与 x86-64/Linux 的规定有什么异同？**

栈式存储管理的实现

- 如果是实现虚拟机……
  - 你想怎么实现就怎么实现！
- 如果是需要生成目标代码？
  - **现有的体系结构和操作系统基本都提供了方便的栈式存储管理**
  - 比如：RISC-V 的栈指针 sp、帧指针 fp，过程调用 call、返回 ret

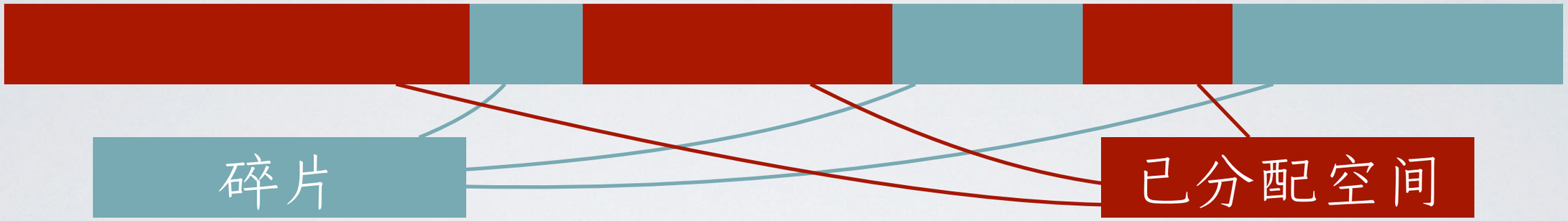
堆式存储管理的实现

- 需要实现的基本功能：分配/回收堆区空间
  - **分配：**为每个内存请求分配一块连续的、适当大小的堆空间
    - 必要时请求操作系统增加堆空间
  - **回收：**把被回收的空间返回空闲空间缓冲池，以满足后续的内存请求

- 评价存储管理器的特性：
  - **空间效率：**使程序需要的堆空间最小，即减少**碎片**
  - **程序效率：**充分利用存储层次结构，提高效率
  - **额外开销：**使分配/回收内存的操作尽可能高效

- **问：程序效率和额外开销的区别是什么？**

堆空间的碎片问题



- 随着运行时分配/回收内存，堆区被逐渐割裂成为若干空闲存储块(「窗口」，hole)和已用存储块的交错
- 对于一个分配请求，通常是使用一个足够大的窗口的一部分，其余部分成为更小的窗口
- 对于一个回收请求，被释放的存储块被放回缓冲池，通常需要把连续的窗口**接合(coalesce)**成更大的窗口

堆存储分配策略

- 假设分配请求的大小为 N 字节
- **最佳适配(best-fit)**：把长度大于或等于 N 的空闲存储块中的长度最小者作为待分配块
- **首先适配(first-fit)**：把构成堆中的空闲存储块中第一个长度大于或等于 N 的作为待分配块
  - **下次适配(next-fit)**：从上次分配的位置开始寻找待分配块
- **最大适配(largest-fit)**：把长度大于或等于 N 的空闲存储块中的长度最大者作为待分配块
- **问：什么场景适合最大适配？**

人工存储管理的问题

- 内存泄漏(memory leak)
  - 一直未能删除不能被访问的数据

```
void f() {
  int *a = malloc(sizeof(int));
  return;
}
```

- 悬空指针解引用(dangling-pointer dereference)
  - 解引用已经被删除的数据

```
int *a = malloc(sizeof(int));
free(a);
…
int b = *a;
```

现代存储管理机制

- **静态机制**
  - 静态推导对象的生命周期(lifetime)、对象的所有者(ownership)等
  - RUST: Regions, Uniqueness, Ownership & Types (链接)
- **动态机制**
  - 智能指针(smart pointers)
  - **垃圾回收**

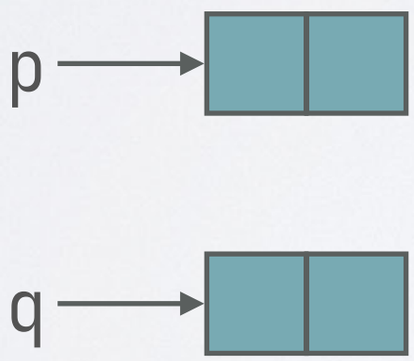
垃圾回收

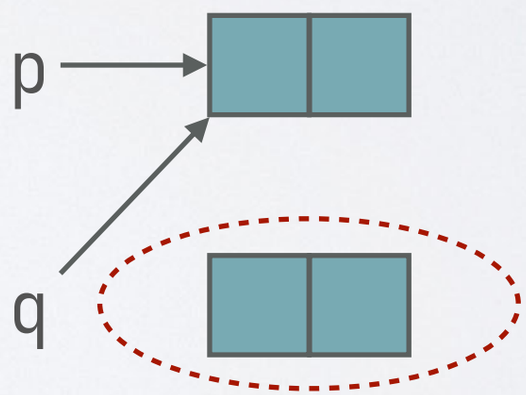
- **垃圾(garbage)**
  - 狭义：不能被访问(不可达)的数据
  - 广义：不需要再被使用的数据
- **垃圾回收(garbage collection)**
  - 自动回收不可达数据的机制，降低程序员的负担
- 使用垃圾回收的语言：
  - 最早出现在 1958 年 Lisp 语言的实现中
  - OCaml、Haskell、Java、Python、Go、JavaScript、Swift 等

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?**

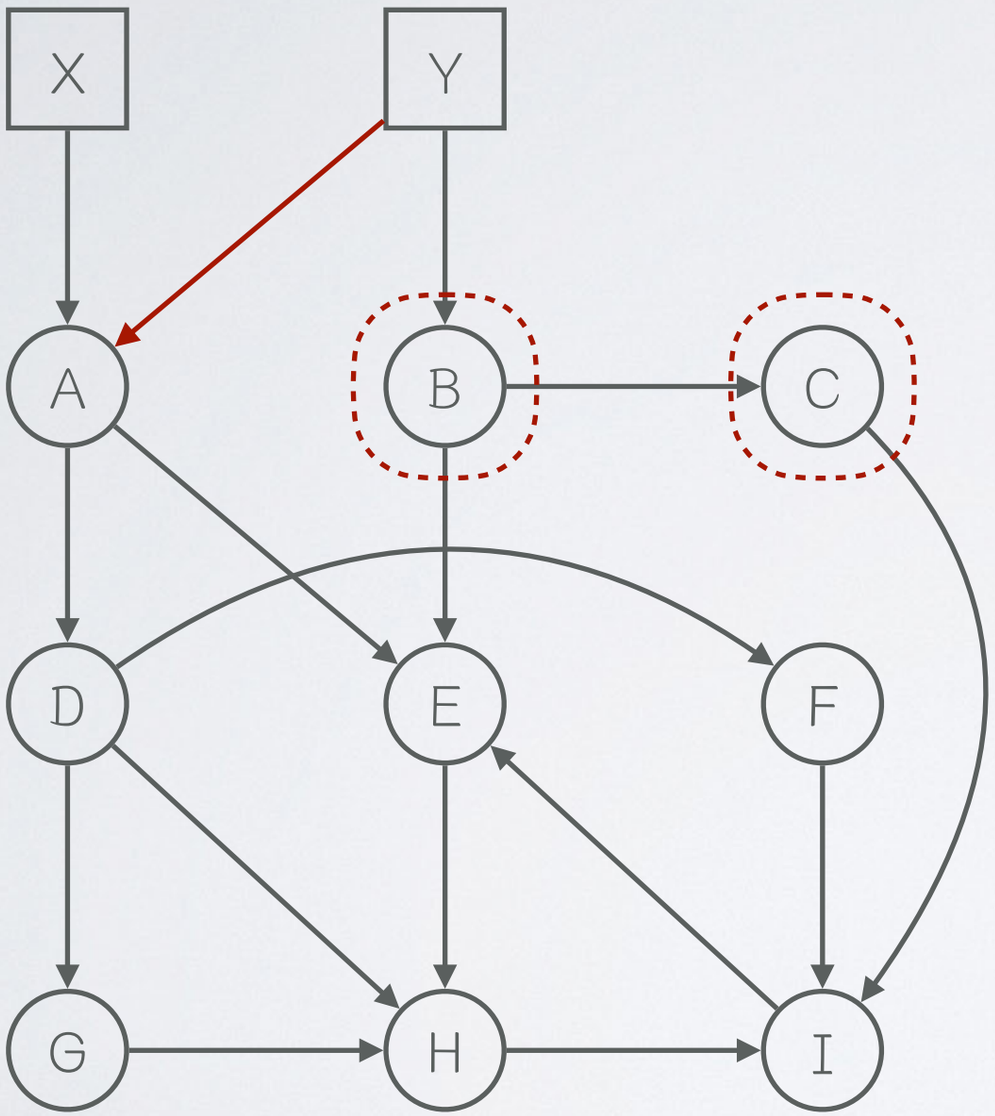
垃圾回收算法

- 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

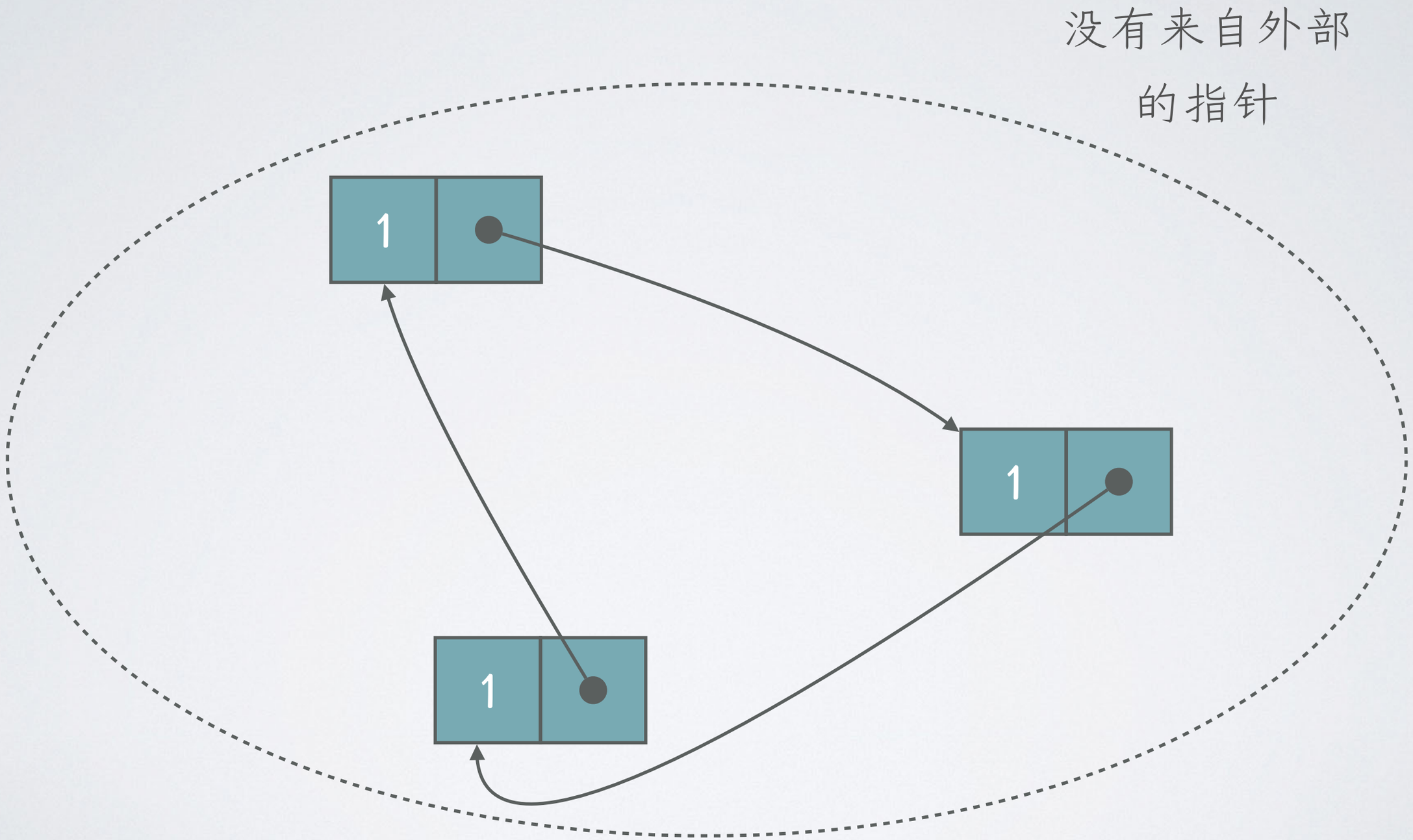
- 每个对象有一个用于存放**引用计数**(reference counting)的字段，并按照如下方式维护：
  - **对象分配**：引用计数设为 1
  - **参数传递**：引用计数加 1
  - **引用赋值**：对于 $u = v$，u 指向的对象引用计数减 1，v 指向的对象引用计数加 1
  - **过程返回**：每个局部变量指向对象的引用计数减 1
  - **问：返回值如何处理？**
- 如果一个对象的引用计数为 0，在删除该对象前，此对象中各个指针所指对象的引用计数减 1
- 使用引用计数的语言：Objective C、Swift 等

引用计数示例



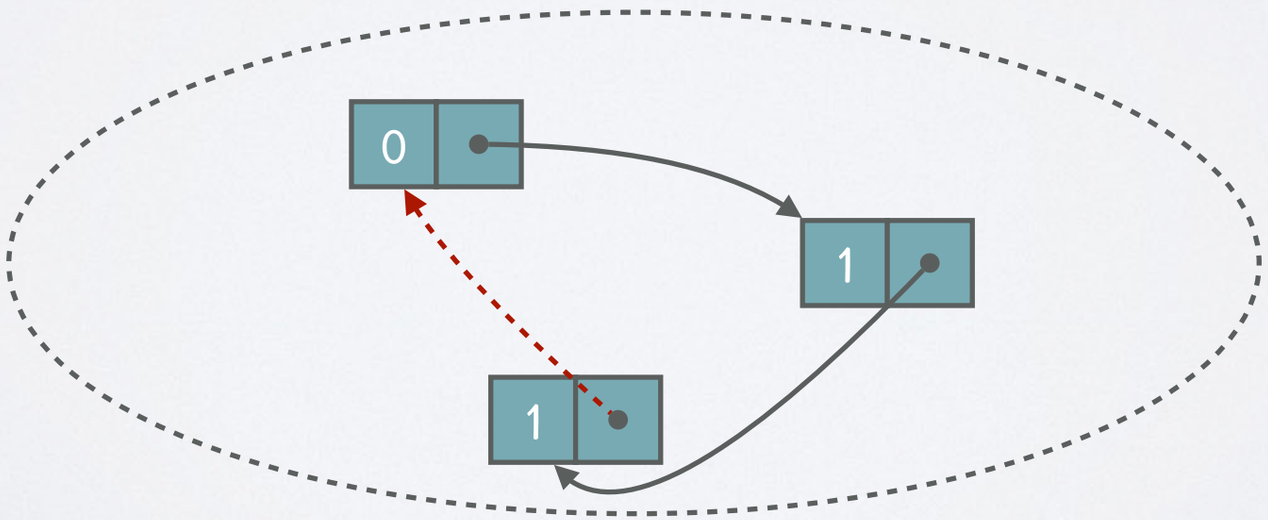
- 考虑赋值 $Y = X$
- 修改计数后总是先考虑是否释放该对象
- 释放一个对象前总是先处理该对象内部的指针

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

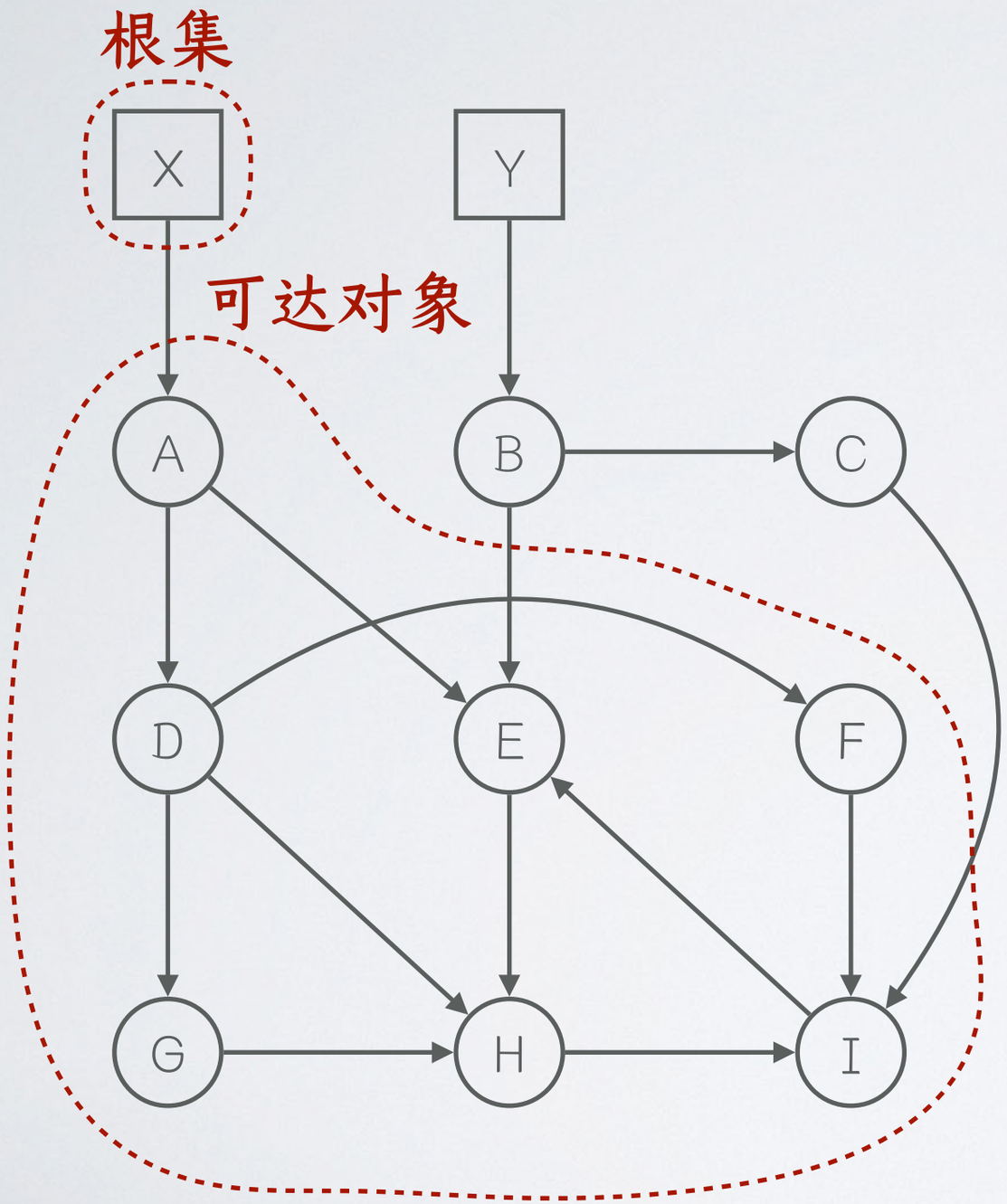
- 以**周期性**的方式运行，在空闲空间耗尽或者低于某个阈值时启动，**寻找不可达对象**并回收其空间

- 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

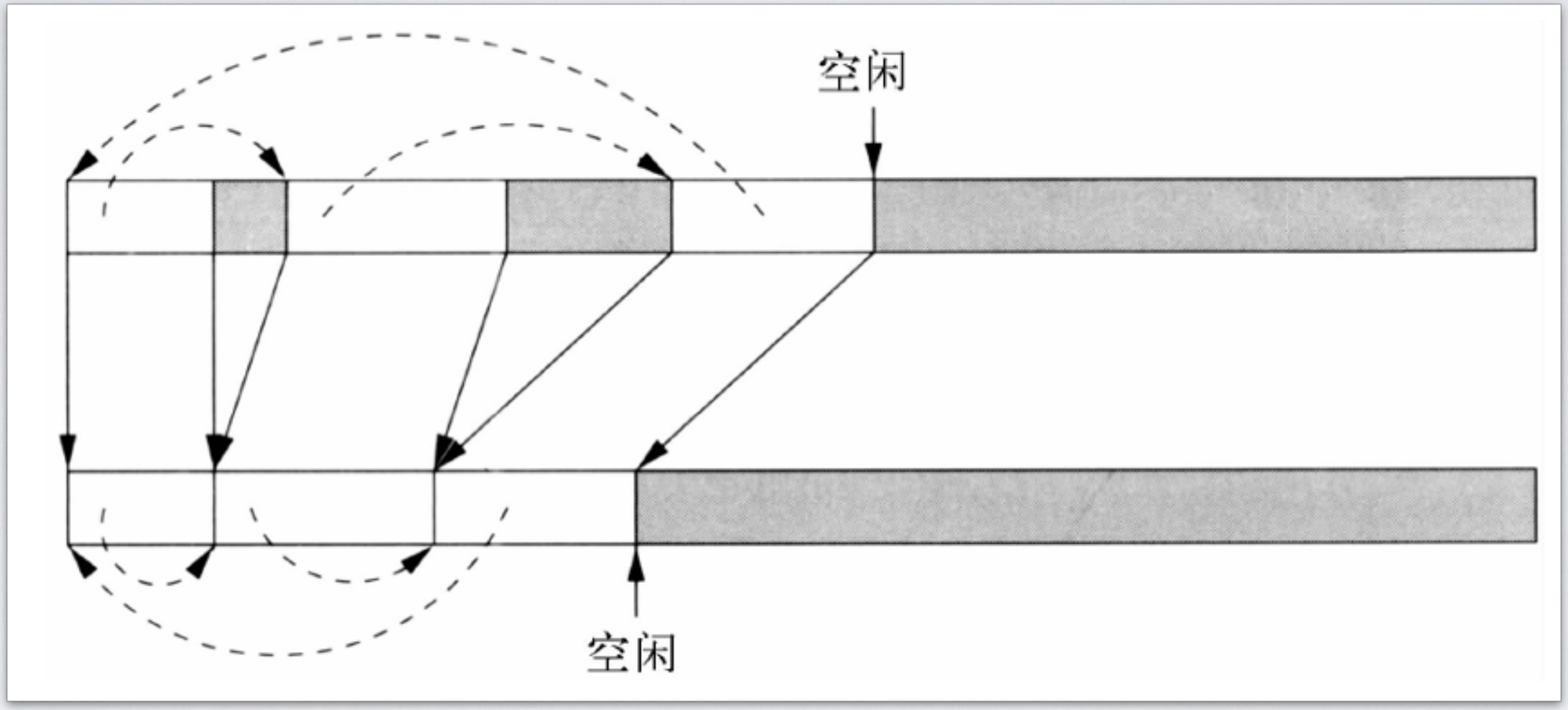
- 基本算法需要扫描整个堆区
- **优化：**
  - 用一个列表记录所有已经分配的对象
  - 不可达对象等于已分配对象去掉可达对象
- **优点：**只需要扫描这个列表就可以完成清扫
- **缺点：**需要维护这个额外的列表

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



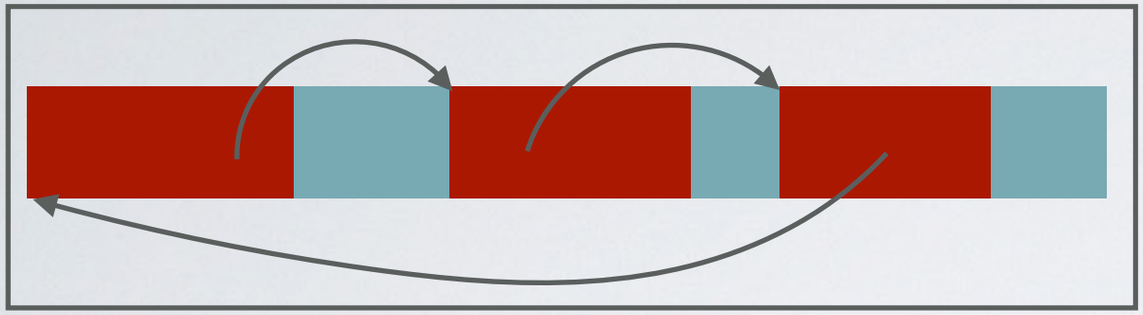
小结：标记-清扫

- **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

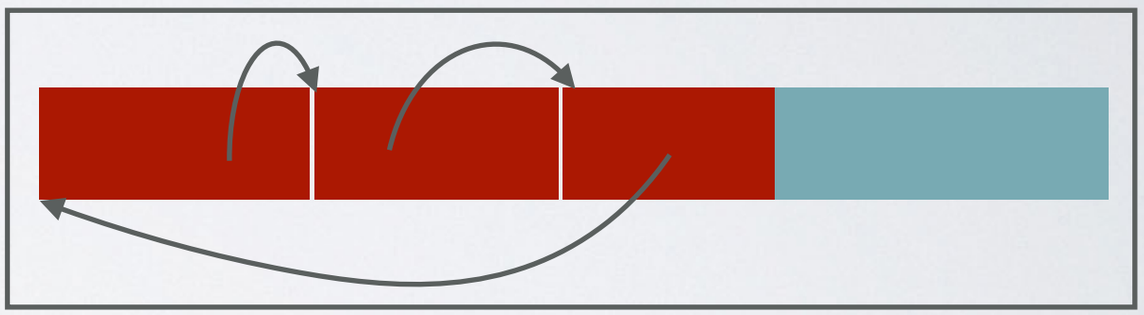
拷贝回收示例



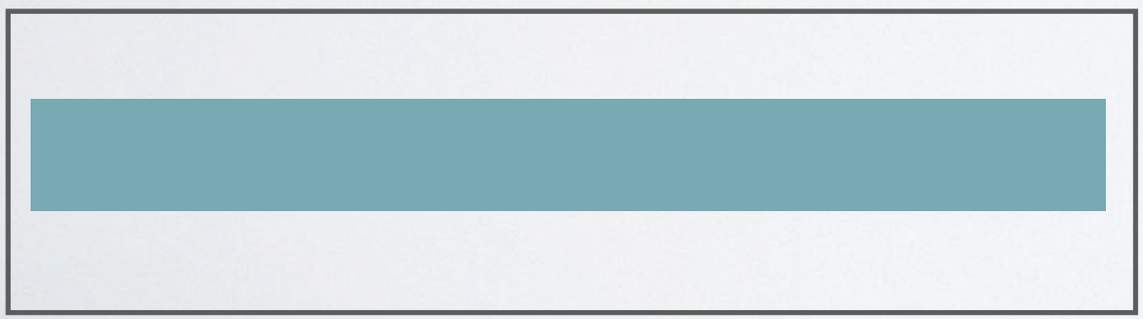
From 半空间

To 半空间

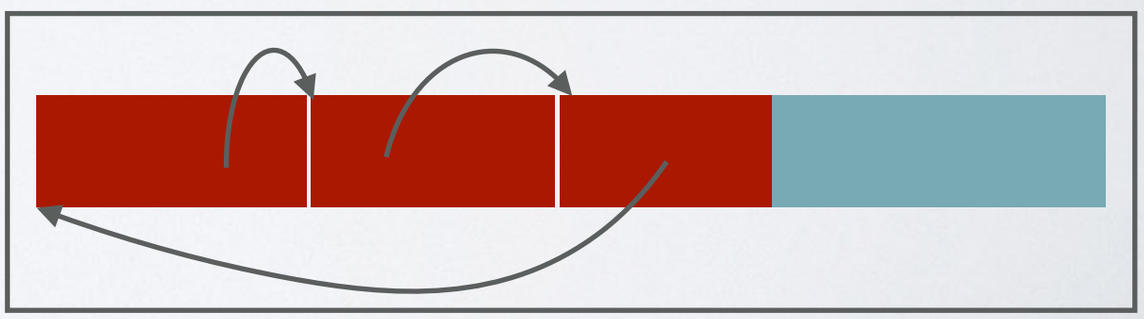
From 半空间



To 半空间



_To_ 半空间

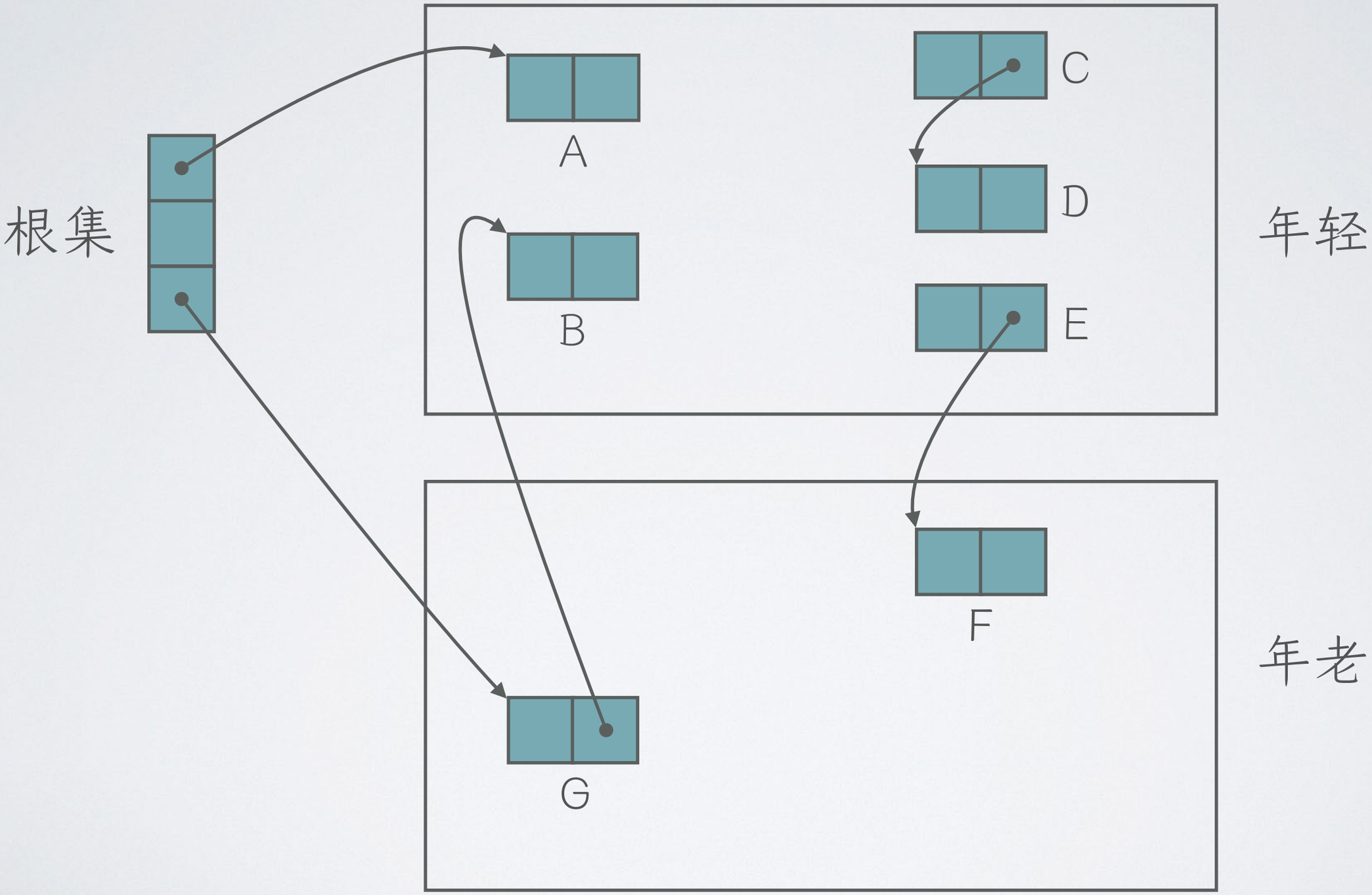


From 半空间

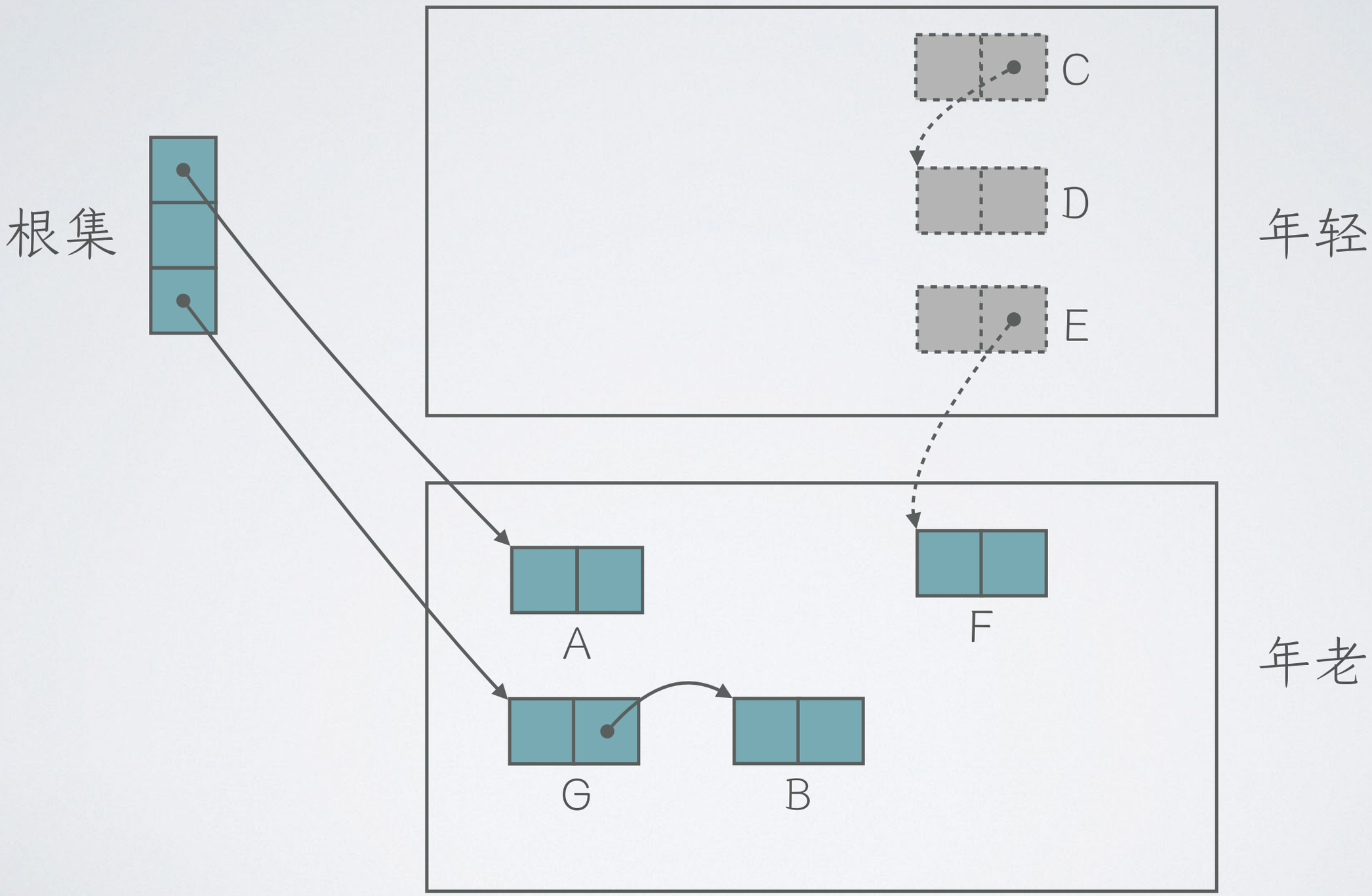
世代垃圾回收器

- 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

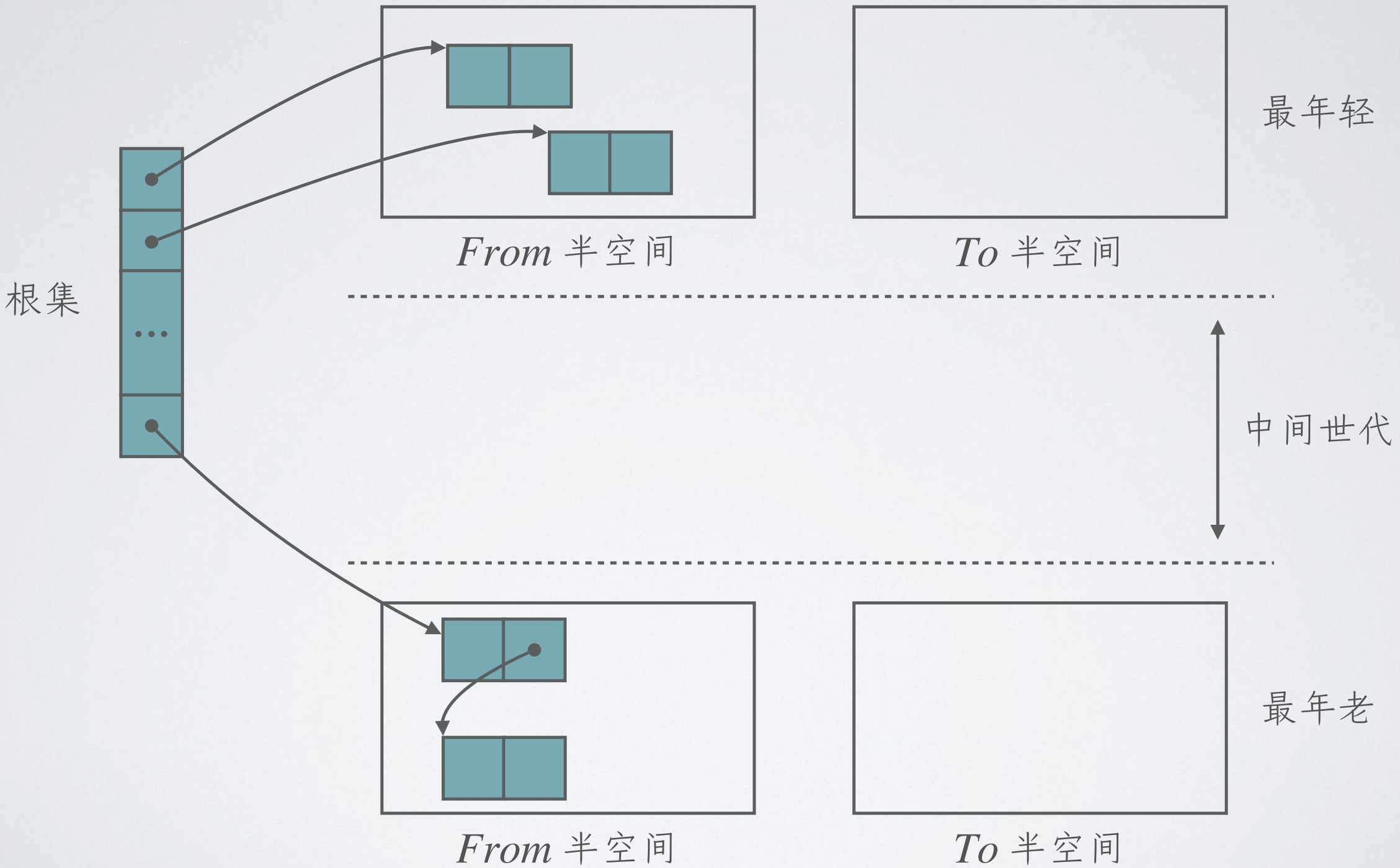
世代垃圾回收示例



世代垃圾回收示例



拷贝世代垃圾回收器



主要内容

- 运行时环境的作用
- 运行时环境的设计
- 运行时环境的实现

- One More Thing

案例：各种虚拟机

- Lox：栈式字节码，支持闭包、面向对象、垃圾回收，纯 C 实现
  - 玩具项目，适合学习
- Lua：寄存器式字节码，支持闭包、垃圾回收，可嵌入 C 中
  - 工业级，但实现好，有很多源码解读的资料
- Wasm：栈式字节码，面向浏览器设计，安全、跨平台、高性能
  - 当红炸子鸡
- EVM：栈式字节码，面向区块链设计，是智能合约的编译目标
  - 有 Rust、Python、C++、JavaScript 等语言做的独立实现

本讲小结

- 运行时环境的作用
  - 实现代码运行所需的存储组织和过程抽象
- 运行时环境的设计
  - 存储组织：纯静态存储管理、栈式存储管理、堆式存储管理
  - 过程抽象：活动记录、控制链、访问链
- 运行时环境的实现
  - 过程抽象：调用代码序列、返回代码序列、寄存器的保存和恢复
  - 栈式存储管理：多数体系结构/操作系统都支持得很好
  - 堆式存储管理：分配和回收策略、垃圾回收(引用计数、标记清扫)

思考问题

- 为什么编译过程需要设计运行时刻环境?
- 运行时刻环境中需要保留和维护编程语言的哪些信息?
- 虚拟机考虑的代码形式通常都是线性的(比如各种字节码),这是为什么呢?可以设计成图状的代码形式吗?
- 虚拟机实现难度相对低,但是直接生成的目标机器代码的效率相对高,有没有什么方法可以结合两者的优点?
- 面向对象语言、函数式语言的运行时刻环境需要考虑哪些设计问题?你能想象出什么实现难点?
- 人工智能时代,是否能够面向机器学习代码设计虚拟机?