第三章 栈与队列

黄群 主讲

采用教材：《数据结构与算法》，张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6（“十二五”国家级规划教材）

http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

第三章 栈与队列

- 栈的应用：递归函数、递归到非递归的转换
  - 递归的基本概念
  - 递归函数调用原理
  - 机械的递归转换

递归的基本概念

- 递归：计算机科学中最重要的概念之一
  - 在解决一个问题时，问题的解依赖于更小规模的相同问题的解

更小规模

“想要理解递归，你先要理解递归”

相同问题

- 递归与迭代
  - 相同点：都是更小规模的相同问题
  - 不同点：
    - 迭代：小问题 -> 大问题（自底向上）
    - 递归：大问题 -> 小问题（自顶向下）

递归的基本概念

- 更接近人类思维方式
  - 设计递归算法比设计非递归算法往往更容易
  - 大多数编程语言支持递归
  - 许多函数式编程语言，更是直接以递归为基础

- 递归在完成问题定义时，也同时完成了问题求解

递归到非递归的转换

递归函数

- 递归函数：解决递归问题的函数
  - 直接或间接的调用函数自身

阶乘 $f(n)=\begin{cases}1, & n\le 1 \\ n\times f(n-1), & n>0\end{cases}$

边界情形

递归规则

```
long fact(long n) {
  if (n <= 1)
    return 1; // 边界情形
  else
    return n * fact(n-1); // 递归规则
}
```

- 递归函数的2个要素
  - 基本的边界情形：可以直接求解
  - 递归规则：将问题转化为子问题

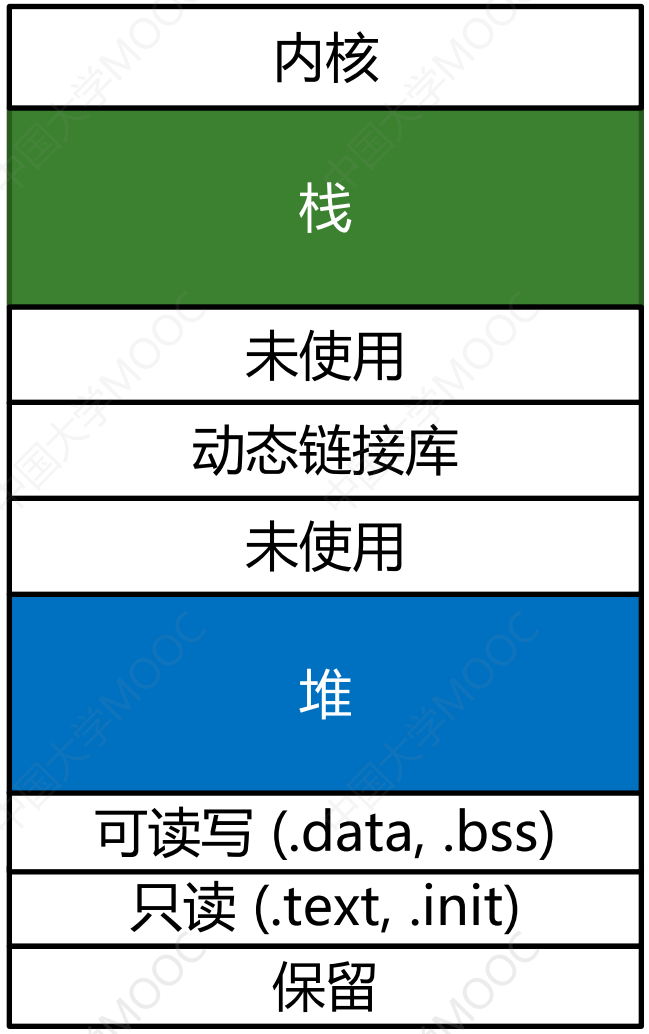
递归函数的执行

- 递归函数：调用自身
- 但是，计算机程序是一组顺序执行的指令序列
- 问题：怎样用指令序列运行一个递归函数？

递归到非递归的转换

程序内存布局

- 栈
  - 用于函数调用，包括递归函数
  - 又被称作 “调用栈”



- 堆
  - 用于动态分配内存空间

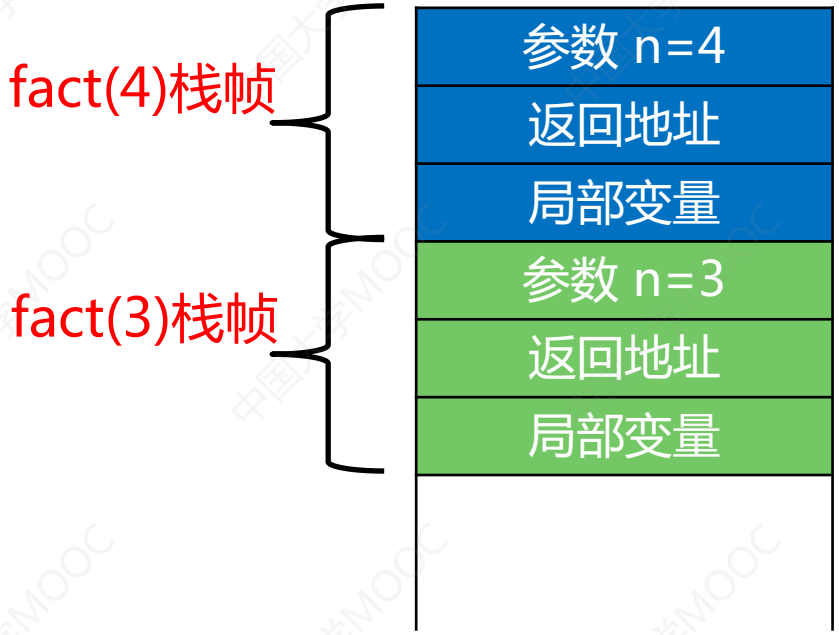
递归到非递归的转换

调用栈

↓栈增长方向

- 调用栈的元素：栈帧
  - 该次调用时，传入的参数
  - 函数返回地址
  - 局部变量

```
long fact(long n) {
  if (n <= 1)
    return 1; // 边界情形
  else
    return n * fact(n-1); // 递归规则
}
```

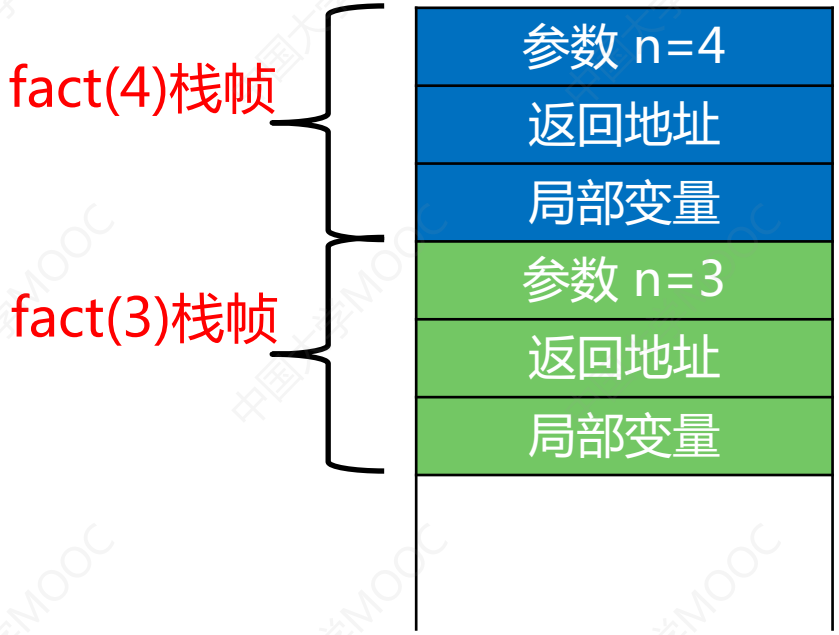


递归到非递归的转换

函数调用与返回：对调用栈的操作

- 函数调用
  - 压入栈帧
    - 压入调用参数
    - 压入返回地址
    - 为局部变量分配空间
    - 其它信息（寄存器信息）
  - 跳转到被调用函数，开始执行

- 函数退出
  - 记录返回值
  - 释放栈帧（局部变量，返回地址，调用函数，其它）
  - 根据返回地址，跳回调用前位置继续执行



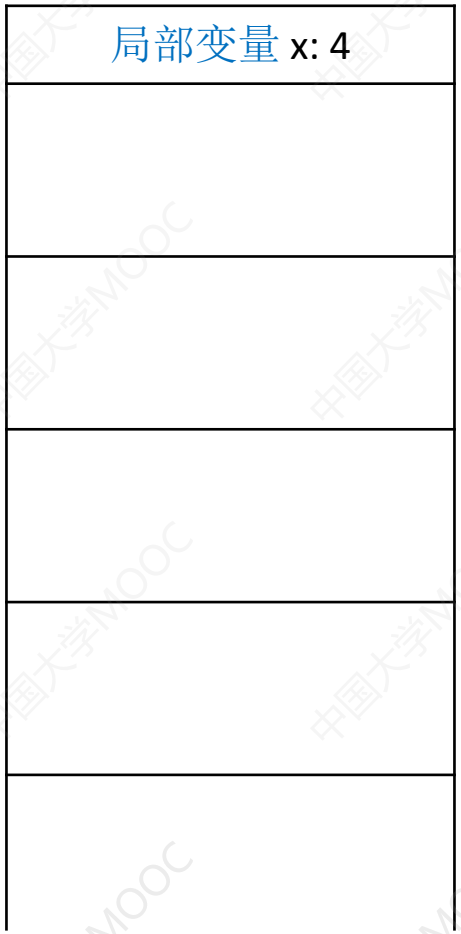
递归到非递归的转换

例：阶乘函数的调用栈

```
long fact(long n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1); // 递归规则
}

int main() {
  int x = 4;
  printf("%d\n", fact(x));
  return 0;
}
```

递归到非递归的转换

例：阶乘函数的调用栈

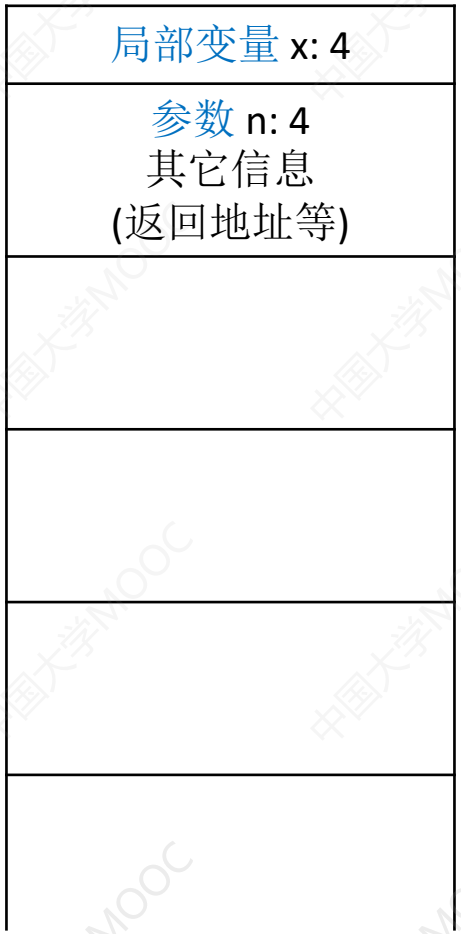


主函数栈帧

↓ 栈增长方向

递归到非递归的转换

例：阶乘函数的调用栈



主函数栈帧

fact(4)栈帧

↓ 栈增长方向

递归到非递归的转换

例：阶乘函数的调用栈

主函数栈帧

fact(4)栈帧

fact(3)栈帧

↓ 栈增长方向

递归到非递归的转换

例：阶乘函数的调用栈

主函数栈帧

fact(4)栈帧

fact(3)栈帧

fact(2)栈帧

↓ 栈增长方向

递归到非递归的转换

例：阶乘函数的调用栈

主函数栈帧

fact(4)栈帧

fact(3)栈帧

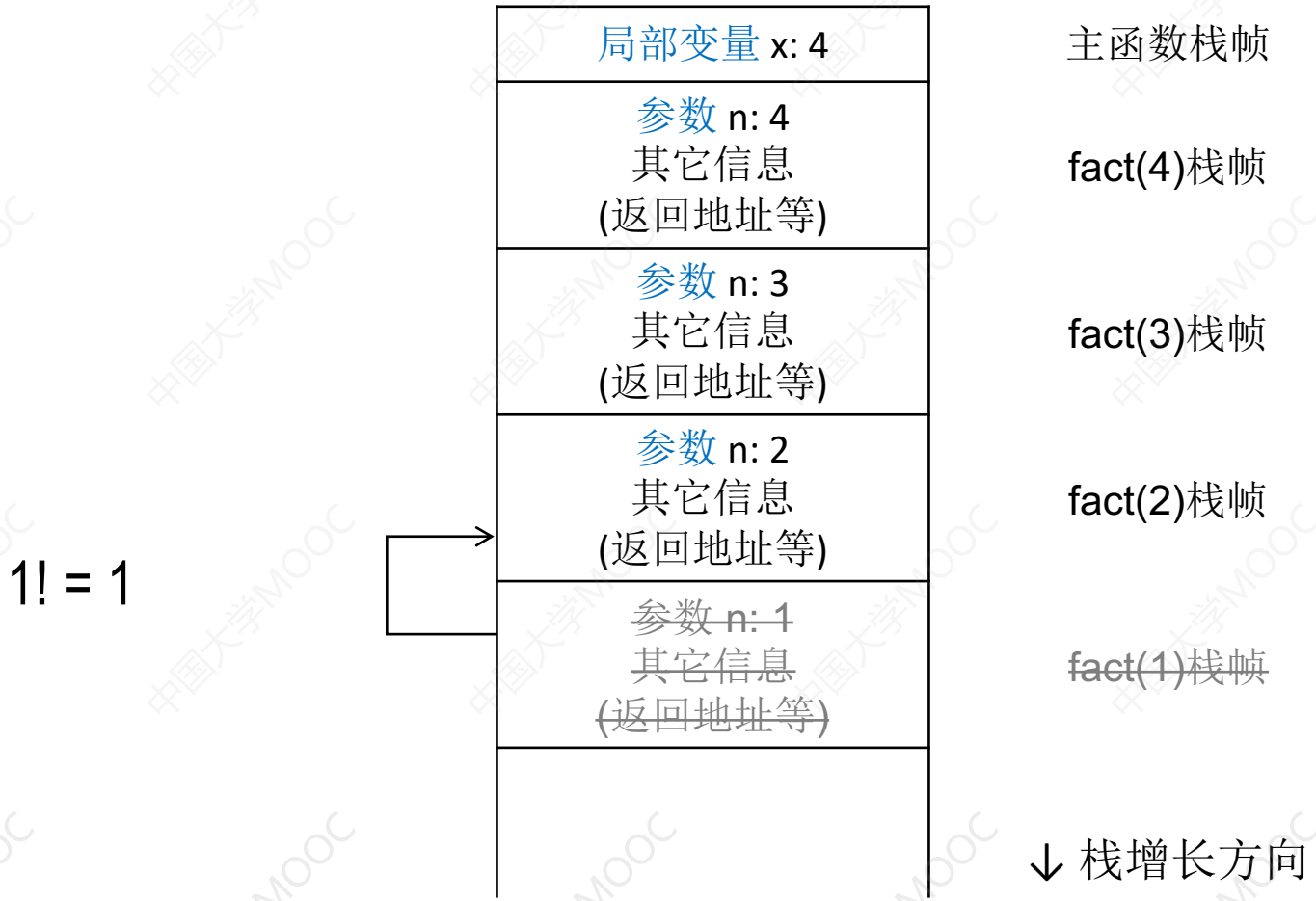
fact(2)栈帧

fact(1)栈帧

↓ 栈增长方向

递归到非递归的转换

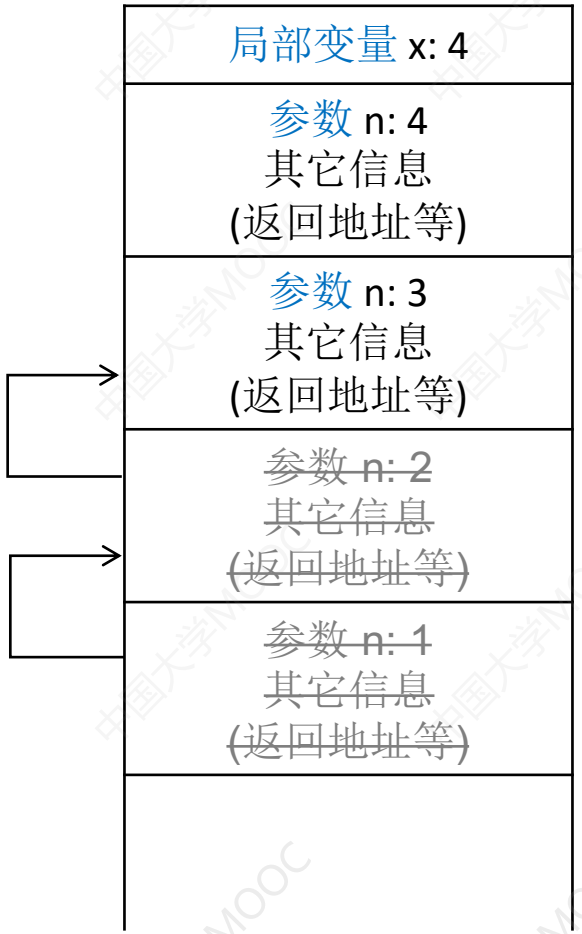
例：阶乘函数的调用栈



例：阶乘函数的调用栈

2! = 2*1! = 2

1! = 1



主函数栈帧

fact(4)栈帧

fact(3)栈帧

~~fact(2)栈帧~~

~~fact(1)栈帧~~

↓ 栈增长方向

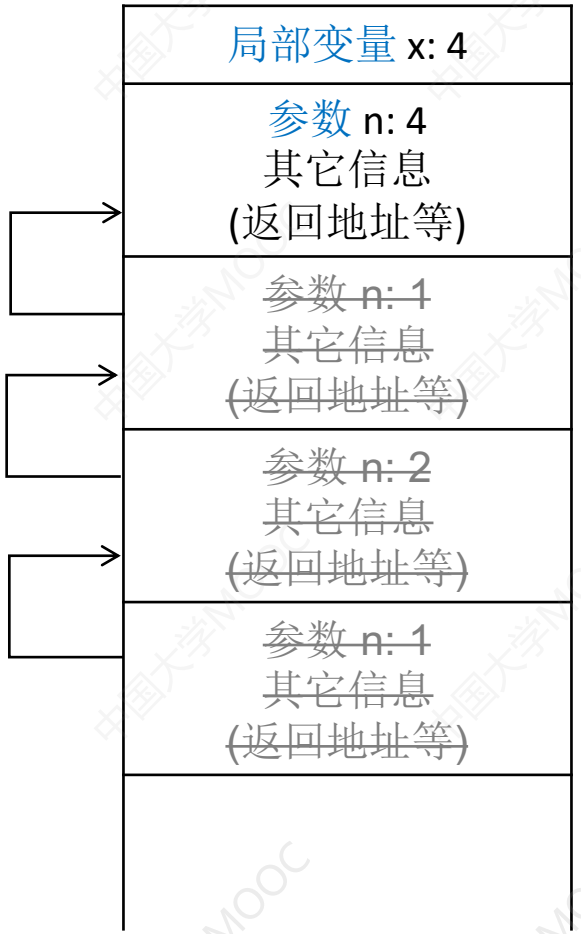
递归到非递归的转换

例：阶乘函数的调用栈

3! = 3*2! = 6

2! = 2*1! = 2

1! = 1



主函数栈帧

fact(4)栈帧

fact(3)栈帧

fact(2)栈帧

fact(1)栈帧

↓ 栈增长方向

递归到非递归的转换

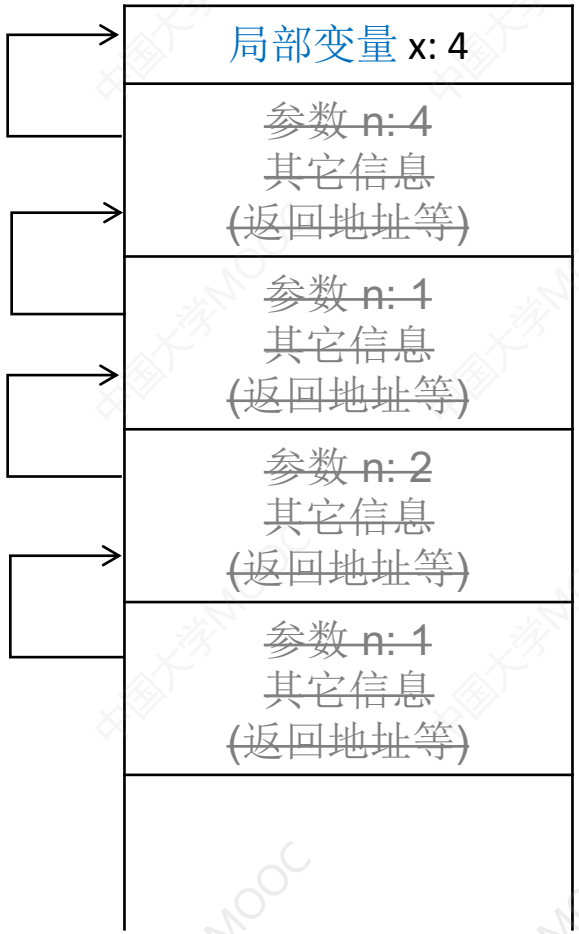
例：阶乘函数的调用栈

4! = 4*3! = 24

3! = 3*2! = 6

2! = 2*1! = 2

1! = 1



主函数栈帧

~~fact(4)栈帧~~

~~fact(3)栈帧~~

~~fact(2)栈帧~~

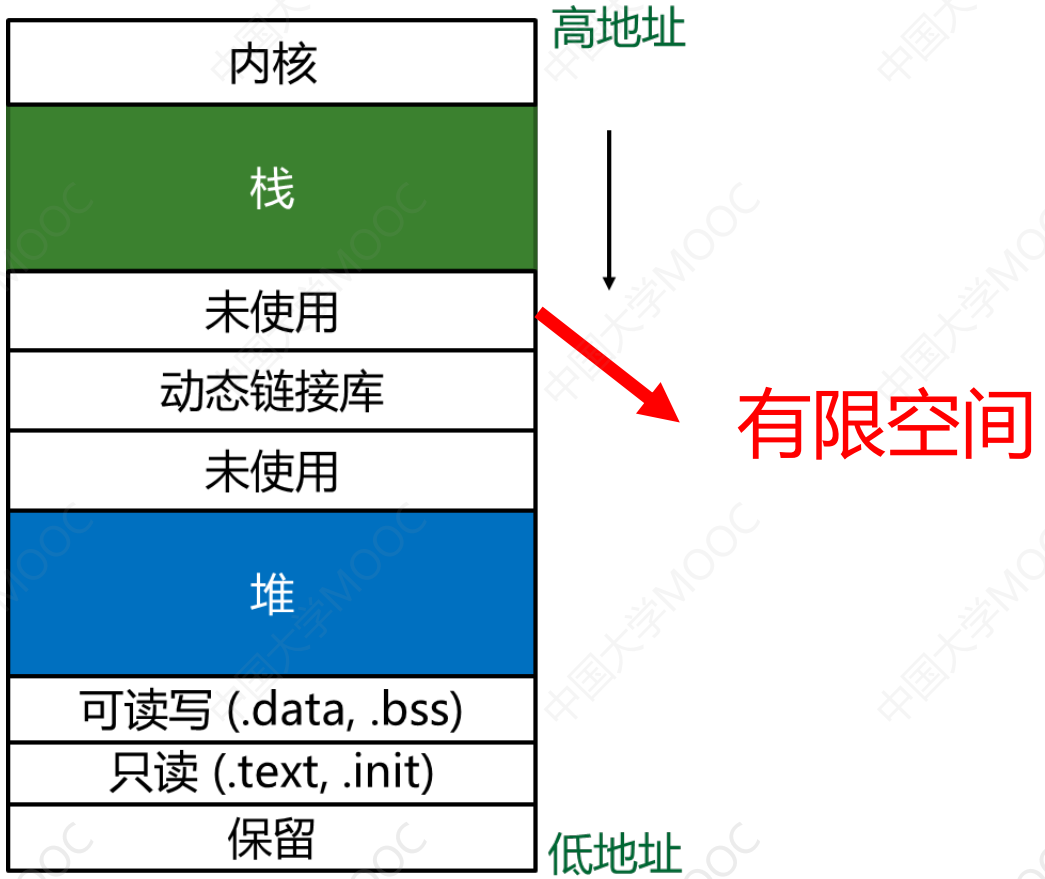
~~fact(1)栈帧~~

↓ 栈增长方向

递归到非递归的转换

递归函数的开销

- 每次函数调用需要创建栈帧
- 空间开销
  - 实际中的栈帧可能很大（包含其它信息）
  - 可能导致栈溢出，造成安全漏洞
- 时间开销
  - 栈操作
  - 函数调用指令



- **递归函数转非递归：减少开销**

递归转非递归：尾递归

- 一类特殊的递归函数：尾递归

- 尾递归：函数仅有一次自身调用，且该调用是函数退出前的最后一个操作

递归转非递归：尾递归

```
long fact(long n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```

不是尾递归

```
long fact_tail_rec(long n, long product) {
  if (n <= 1)
    return product;
  else
    return fact_tail_rec(n-1, product * n);
}
```

改写成尾递归形式

递归转非递归：尾递归

- 尾递归可以很容易转化成非递归形式
  - 尾递归的本质：将单次计算的结果缓存起来，传递给下次调用，相当于自动累积
  - 转化非递归：通过循环迭代，每次保存累积结果
- 转化之后：没有栈开销、没有函数调用开销
  - 递归调用：线性空间
  - 非递归循环：常数空间
- 许多现代编程语言支持对尾递归的优化
  - 编译器/解释器：GCC, LLVM/Clang, Intel 编译器, Java 虚拟机
  - 函数式编程语言：LISP, Scheme, Scala, Haskell, Erlang

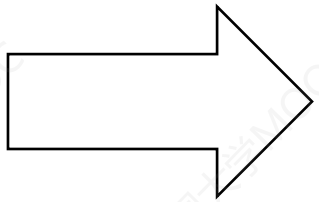
递归到非递归的转换

通用的递归到非递归转化

- 思想：通过显式模拟函数调用过程
  - 特别是模拟调用栈的操作
- 好处：使用于所有递归函数（机械转换）
  - 空间：减少冗余存储
  - 时间：减少冗余操作（主要是函数跳转）

递归版本

```
long fact(long n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```



非递归版本

```
long fact_nr (long n) {
  Stack s;
  int  ret = 1;

  while (n>0)
        s.push(n--);

  while (!isEmpty(s)) {
        ret *= s.pop(s);
  return ret;
}
```

示例问题

- [简化的0-1背包问题]

我们有n件物品，物品i的重量为w[i] 。如果限定每种物品：要么完全放进背包，要么不放进背包；即物品是不可分割的。

问：能否从这n件物品中选择若干件放入背包，使其重量之和恰好为s。

递归到非递归的转换

示例问题

$$knap(S,n)=\begin{cases} true & \text{if } S=0 \\ false & \text{if } S<0 \text{ or } (S>0 \text{ and } n<1) \\ knap(S-w_{n-1},n-1) \text{ or } knap(S,n-1) & \text{if } S>0 \text{ and } n\geq 1 \end{cases}$$

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
    }
```

递归到非递归的转换

步骤1：定义调用栈帧

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```

```
struct Elem {
    int s; int n;
    int rd;

    Elem(int s, int n, int rd):
            s(s), n(n), rd(rd) {}
};
```

调用参数

返回地址(从何处调用了该函数)

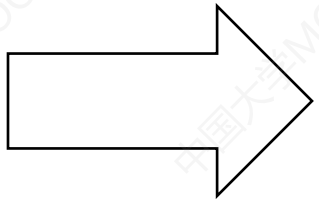
栈帧构造函数

栈帧数据结构

递归到非递归的转换

步骤2：将原问题入栈

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```



rd=0表示这里是原问题调用

```
bool knap(int s, int n) {
    stack<Elem> st; // 创建栈
    Elem x(s,n,0); // 原问题栈帧
    st.push(x); // 入栈
    bool ret; // 最近一次调用结果

    // more
}
```

递归到非递归的转换

步骤3：程序分析

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
    }
}
```

出口1：执行完原问题knap(s, n)后，返回函数调用者

3种递归出口

2处递归调用

出口2：执行完knap(s-w[n-1], n-1)后，继续cout语句

出口3：执行完knap(s, n-1)后，继续return tmp语句

递归到非递归的转换

步骤3：程序分析

```
bool knap(int s, int n) {
    if (s == 0) return true;          // 区域 0: 第一次递归调用之前
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << "";
            return true;
    }                                 // 区域1: 第二次递归调用之前
    else {
            bool tmp = knap(s, n-1);
            return tmp;
    }                                 // 区域 2: 第二次递归调用之后
}
```

小结：t次递归调用，就有t+1个递归出口，将原函数划分为t+1个区域

递归到非递归的转换

步骤4：划分标签、翻译

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```

```
bool knap(int s, int n) {
    stack<Elem> st;
    Elem x(s,n,0);
    st.push(x);
    bool ret;

L0:
    s = st.top().s;  n = st.top().n;
    if (s == 0) {
        ret = true; TODO return; }
    if ((s < 0) || (s > 0 && n < 1)) {
        ret = false; TODO return; }

    // TODO: 第一次递归调用
}
```

递归到非递归的转换

步骤4：划分标签、翻译

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```

```
bool knap(int s, int n) {
    stack<Elem> st;
    Elem x(s,n,0);
    st.push(x);
    bool ret;

L0:
    s = st.top().s;  n = st.top().n;
    if (s == 0) {
        ret = true; TODO return; }
    if ((s < 0) || (s > 0 && n < 1)) {
        ret = false; TODO return; }

    // TODO: 第一次递归调用

L1:
    s = st.top().s;  n = st.top().n;
    if (ret) {
        printf("%d ", w[n-1]);
        ret = true; TODO return;
    } else {
        // TODO: 第二次递归调用

}
```

递归到非递归的转换

步骤4：划分标签、翻译

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```

```
bool knap(int s, int n) {
    stack<Elem> st;
    Elem x(s,n,0);
    st.push(x);
    bool ret;

L0:
    s = st.top().s;  n = st.top().n;
    if (s == 0) {
        ret = true; TODO return; }
    if ((s < 0) || (s > 0 && n < 1)) {
        ret = false; TODO return; }

    // TODO: 第一次递归调用

L1:
    s = st.top().s;  n = st.top().n;
    if (ret) {
        printf("%d ", w[n-1]);
        ret = true; TODO return;
    } else {
        // TODO: 第二次递归调用

L2:
    TODO return;
    }

}
```

递归到非递归的转换

步骤4：划分标签、翻译

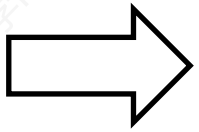
```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```

```
bool knap(int s, int n) {
    stack<Elem> st;
    Elem x(s,n,0);
    st.push(x);
    bool ret;

L0:
    s = st.top().s;  n = st.top().n;
    if (s == 0) {
        ret = true; TODO return; }
    if ((s < 0) || (s > 0 && n < 1)) {
        ret = false; TODO return; }

    // TODO: 第一次递归调用

L1:
    s = st.top().s;  n = st.top().n;
    if (ret) {
        printf("%d ", w[n-1]);
        ret = true; TODO return;
    } else {
        // TODO: 第二次递归调用

L2:
    TODO return;
    }

L3: // TODO: 递归退出，决定返回地址
}
```

递归到非递归的转换

步骤4：划分标签、翻译

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```



```
bool knap(int s, int n) {
   stack<Elem> st;
   Elem x(s,n,0);
   st.push(x);
   bool ret;

L0:
   s = st.top().s;  n = st.top().n;
   if (s == 0) {
      ret = true; TODO return; }
   if ((s < 0) || (s > 0 && n < 1)) {
      ret = false; TODO return; }

   // TODO: 第一次递归调用

L1:
   s = st.top().s;  n = st.top().n;
   if (ret) {
      printf("%d ", w[n-1]);
      ret = true; TODO return;
   } else {
      // TODO: 第二次递归调用

L2:
   TODO return;
   }

L3: // TODO: 递归退出，决定返回地址
}
```

小结：t次递归调用，添加t+2个标签

递归到非递归的转换

步骤5：用goto实现递归调用

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```

```
bool knap(int s, int n) {
   stack<Elem> st;
   Elem x(s,n,0);
   st.push(x);
   bool ret;

L0:
   s = st.top().s;  n = st.top().n;
   if (s == 0) {
      ret = true; TODO return; }
   if ((s < 0) || (s > 0 && n < 1)) {
      ret = false; TODO return; }

   // 第一次递归调用
   st.push(Elem(s-w[n-1],n-1,1)); goto L0;

L1:
   s = st.top().s;  n = st.top().n;
   if (ret) {
      printf("%d ", w[n-1]);
      ret = true; TODO return;
   } else {
      // 第二次递归调用
      st.push(Elem(s,n-1,2)); goto L0;
L2:
   TODO return;
   }

L3: // TODO: 递归退出，决定返回地址
   }
```

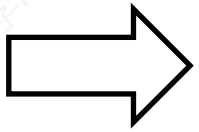
小结：第t次递归调用，rd=t

rd = 1 or rd = 2:
不同的函数调用（递归出口）

递归到非递归的转换

步骤6：处理return语句

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```



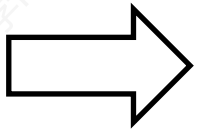
```
bool knap(int s, int n) {
    stack<Elem> st;
    Elem x(s,n,0);
    st.push(x);
    bool ret;

L0:
    s = st.top().s;  n = st.top().n;
    if (s == 0) {
        ret = true; goto L3; }
    if ((s < 0) || (s > 0 && n < 1)) {
        ret = false; goto L3; }

    // 第一次递归调用
    st.push(Elem(s-w[n-1],n-1,1)); goto L0;

L1:
    s = st.top().s;  n = st.top().n;
    if (ret) {
        printf("%d ", w[n-1]);
        ret = true; goto L3;
    } else {
        // 第二次递归调用
        st.push(Elem(s,n-1,2)); goto L0;
L2:
        goto L3;
    }

L3: // TODO: 递归退出，决定返回地址
}
```

小结：所有return都goto到最后一个标签

递归到非递归的转换

步骤7：实现递归退出

```
bool knap(int s, int n) {
    if (s == 0) return true;
    else if ((s < 0) || (s>0 && n <1))
            return false;
    else if (knap (s-w[n-1], n-1)) {
            cout << w[n-1] << " ";
            return true;
    }
    else {
            bool tmp = knap(s, n-1);
            return tmp;
}
```



```
bool knap(int s, int n) {
   stack<Elem> st;
   Elem x(s,n,0);
   st.push(x);
   bool ret;

L0:
   s = st.top().s;  n = st.top().n;
   if (s == 0) {
      ret = true; goto L3; }
   if ((s < 0) || (s > 0 && n < 1)) {
      ret = false; goto L3; }

   // 第一次递归调用
   st.push(Elem(s-w[n-1],n-1,1)); goto L0;

L1:
   s = st.top().s;  n = st.top().n;
   if (ret) {
      printf("%d ", w[n-1]);
      ret = true; goto L3;
   } else {
      // 第二次递归调用
      st.push(Elem(s,n-1,2)); goto L0;
L2:
      goto L3;
   }

L3:
    switch ((x = st.top()).rd) {
      case 0: st.pop(); return ret;
      case 1: st.pop(); goto L1;
      case 2: st.pop(); goto L2;
      default: assert(false);
   }
}
```

小结：t个递归调用，switch语句有t+1个分支
根据rd值，goto到相应位置继续执行

递归到非递归的转换

总结：通用的机械转换步骤

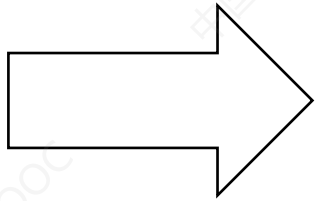
- 步骤 1: 定义栈帧，建立调用栈
- 步骤 2: 在栈中压入原始问题的帧（rd=0）
- 步骤 3: 根据递归调用数t，将程序划分为 t+1 个区域
- 步骤 4: 创建(t+2)个标签，逐区域翻译（除return语句、递归调用）
  - t+2 个标签为 t+1 个区域的边界
- 步骤 5: 用 goto 实现递归调用
  - 形式“push stack; goto label 0”，第i个调用的rd=i
- 步骤 6: 用 goto 实现return语句
  - 将所有 “return” 替换为 “goto label (t+1)”
- 步骤 7: 在标签 t+1 后添加递归出口
  - 使用 “switch” 语句，根据栈顶的 rd 值判断继续执行的标签
- 可选: 代码优化

递归到非递归的转换

思考题

- 问题：将前面提到的阶乘函数使用机械步骤转化为非递归形式

```
long fact(long n) {
  if (n <= 1)
    return 1;

  int tmp1 = fact(n-1);

  int tmp2 = tmp1 * n;
  return tmp2;
}
```



```
int fact_nr(int n) {
  stack<Item> st;
  st.push(Item(n, 0));
  int ret;
L0:
  if (st.top().n <= 1) ret = 1; goto L2;
  st.push(Item(st.top().n-1,1)); goto L0;
L1:
  ret = st.top().n * ret; goto L2;

L2:
  switch (st.top().rd) {
    case 0: st.pop(); return ret;
    case 1: st.pop(); goto L1;
  }
}
```

数据结构与算法

谢谢聆听

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6。“十一五”国家级规划教材