

# 编程基础 I

刘 钦

# Outline

- 代码是用来读的（实践经验）
- 降低复杂度 — 分解与抽象（方法学）
- 编程=数据结构+算法（理论逻辑）
- 算法建模（理论逻辑）
- 数据建模（理论逻辑）
- 编程典型场景 — 数据处理（理论逻辑）
- 用日志记录数据

1. 代码是用来读的！

Write once,  
Read many times

- “When you program, you have to think about how someone will read your code, not just how a computer will interpret it.”
- — kent beck

# 代码可读

- 团队的需要
- 维护的需要

# 糟糕的变量名字

- `x = x - xx;`
- `xxx = aretha + SalesTax( aretha );`
- `x = x + LateFee( x1, x ) + xxx;`
- `x = x + Interest( x1, x );`

# 良好的变量名字

- `balance = balance - lastPayment;`
- `monthlyTotal = NewPurchases + SalesTax( newPurchases );`
- `balance = balance + LateFee( customerId, balance ) + monthlyTotal;`
- `balance = balance + Interest( customerId, balance );`

代码是用来读的！

## 2 降低复杂度 — 分解与抽象

为什么要降低复杂度？

世界是复杂的？

程序员是干什么的？

将复杂的问题转化为代码

有什么降低复杂度的方法？

降低复杂度的方法 —  
分解（组合）

\*

//打印1颗\*

```
print("*")
```

#

//打印1颗 #

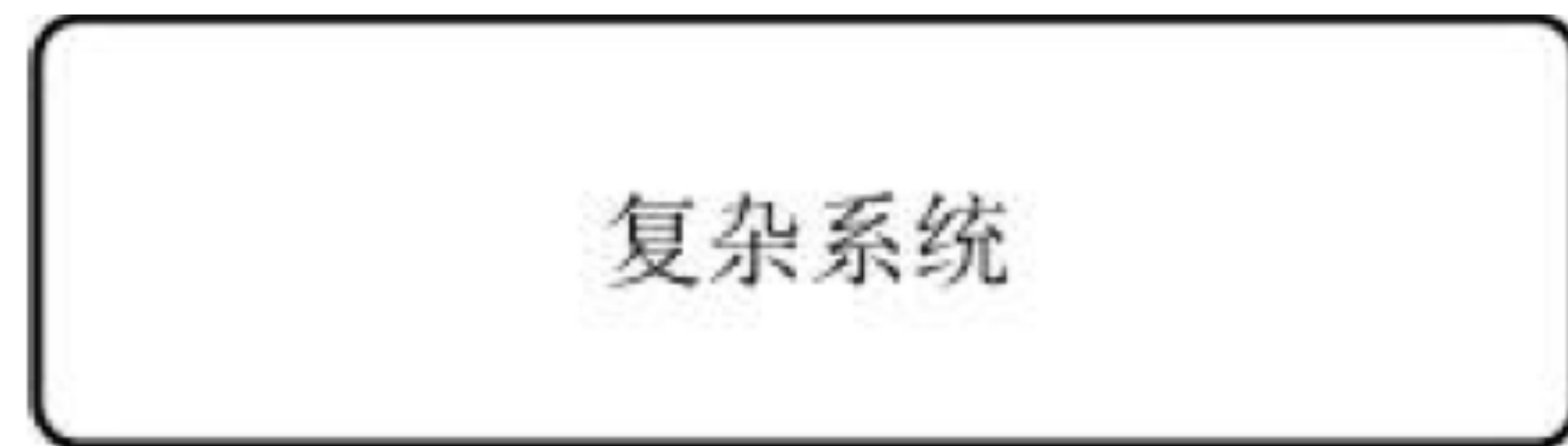
```
print("#")
```

\* #

// 打印 \* #

// = 打印一颗\* + 打印一颗#

```
print("*")  
print("#")
```



分解

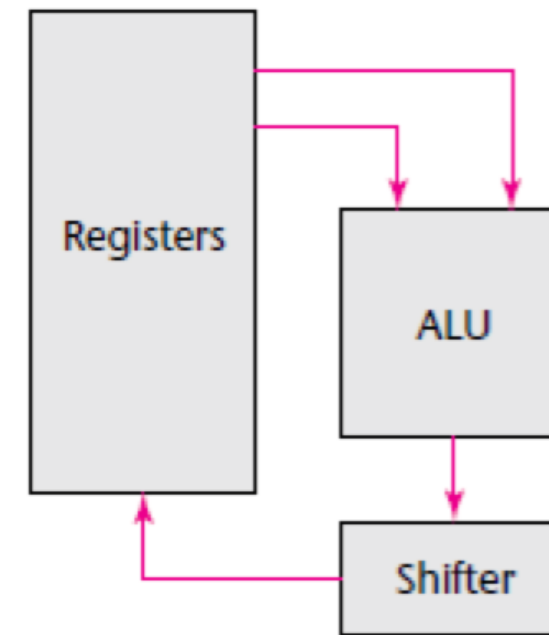


分解 (组合)

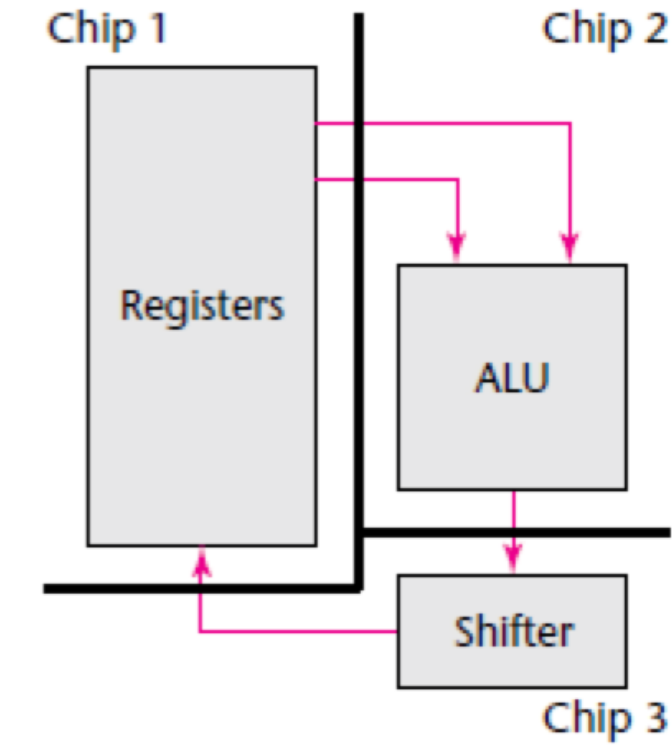
# 分解（组合）的关键点

- 分解之后，每一部分复杂度要变小，
- 相互之间关联要小，相对独立。

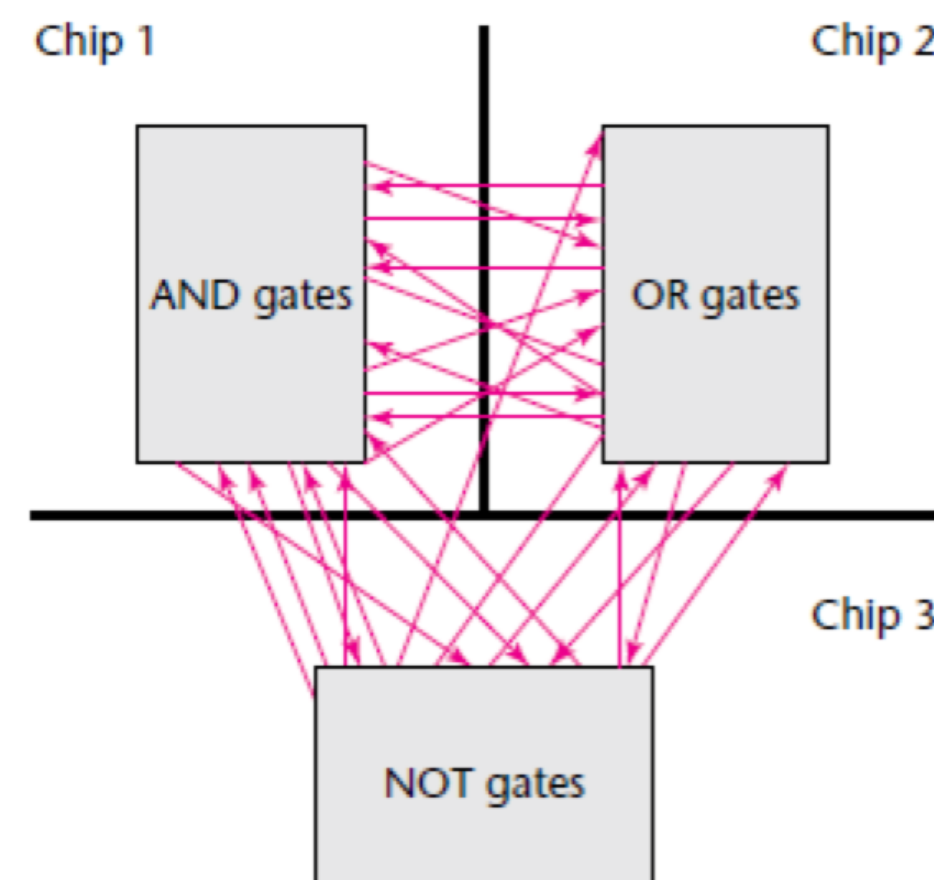
**FIGURE 7.1** The design of a computer.



**FIGURE 7.2** The computer of Figure 7.1 fabricated on three chips.



**FIGURE 7.3** The computer of Figure 7.1 fabricated on three other chips.



# 好的分解和坏的分解

\*#\*#

//打印2对\* #

```
print("*")  
print("#")  
print("*")  
print("#")
```

\*#\*#\*#...

//打印100对\* #

我们愿意输入200遍么？

```
print("*")  
print("#")  
  
...  
print("*")  
print("#")
```

重复100次以下操作

{

print("\*")

print("#")

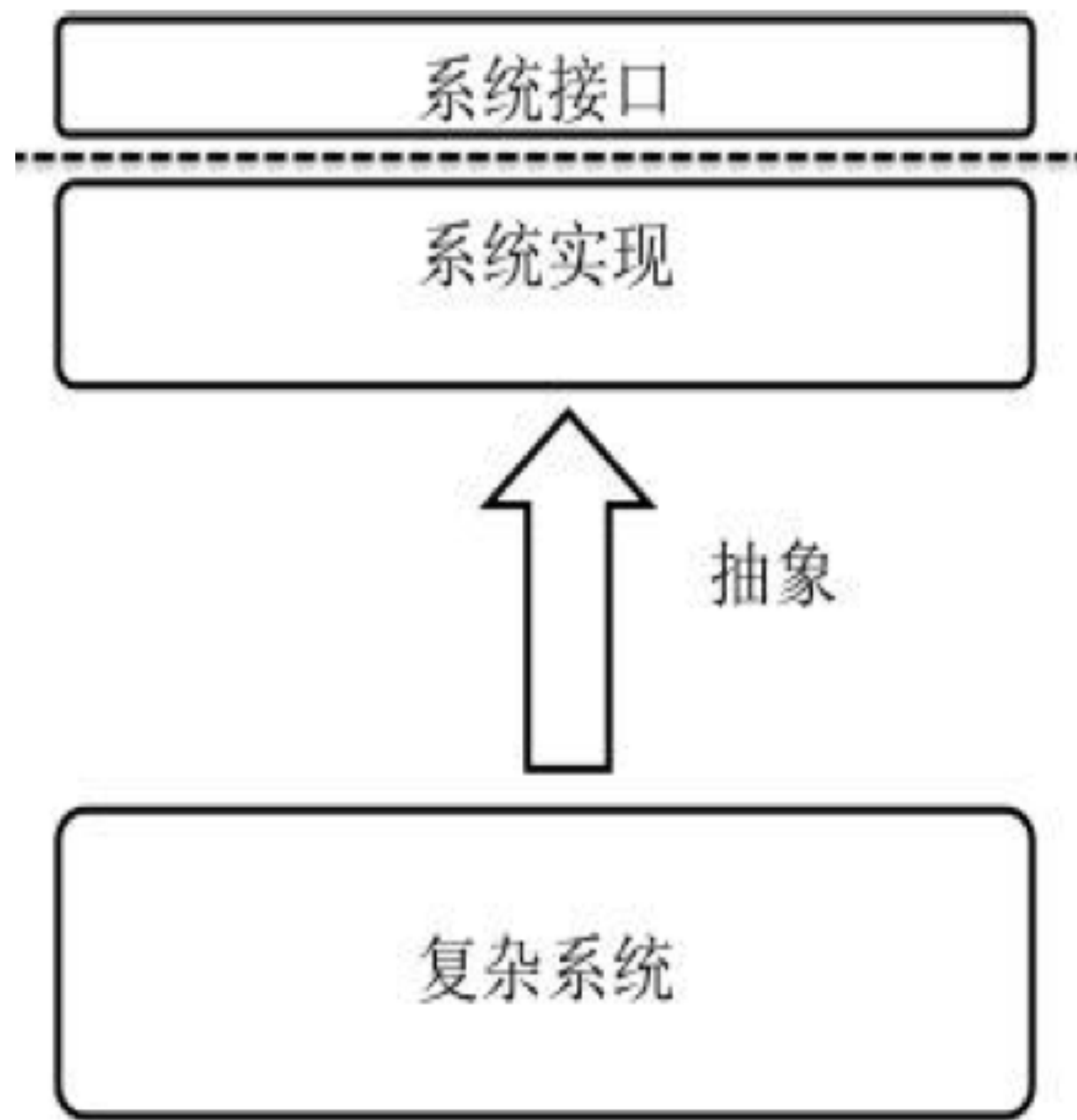
}

如过以下操作特别多

```
print("*")  
print("@")  
print("$")  
print("%")  
...  
print("#")
```

# 降低复杂度的方法 二

## 抽象

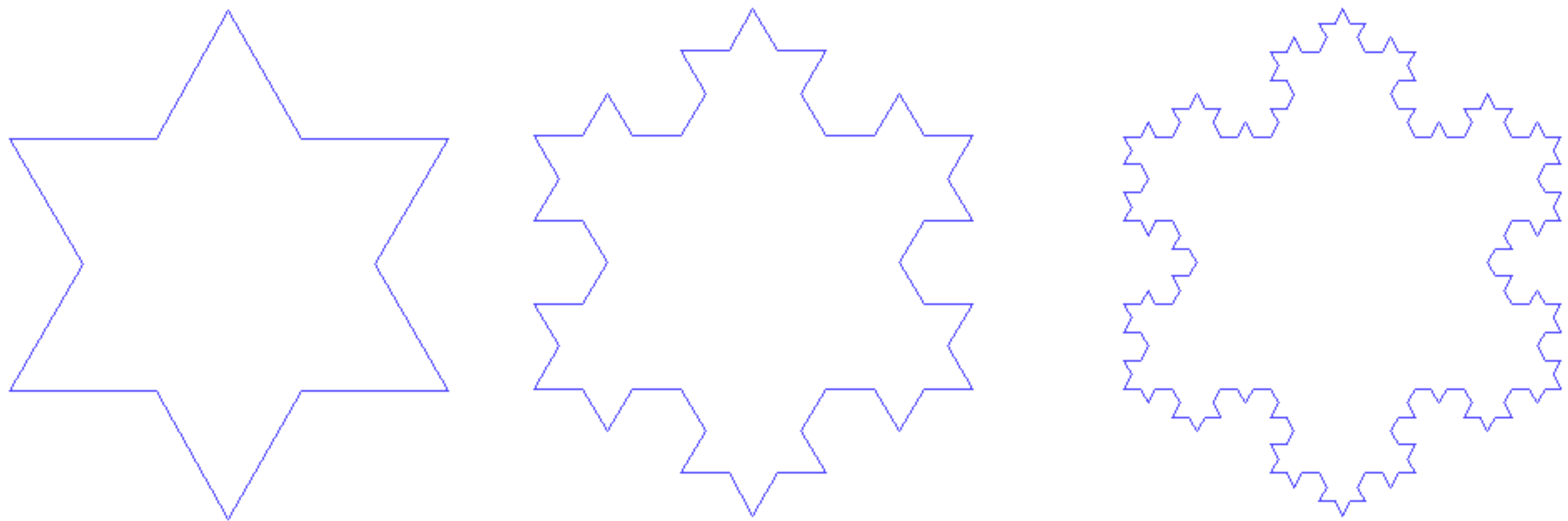


抽象

```
def printSingleStarSharp():  
    print("*")  
    # 特别多种类  
    print("#")  
  
def printStars():  
    printSingleStarSharp()  
    #重复很多遍  
    printSingleStarSharp()  
  
if __name__ == "__main__": #main方法  
    print("main")  
    printStars()
```

# 抽象的关键点

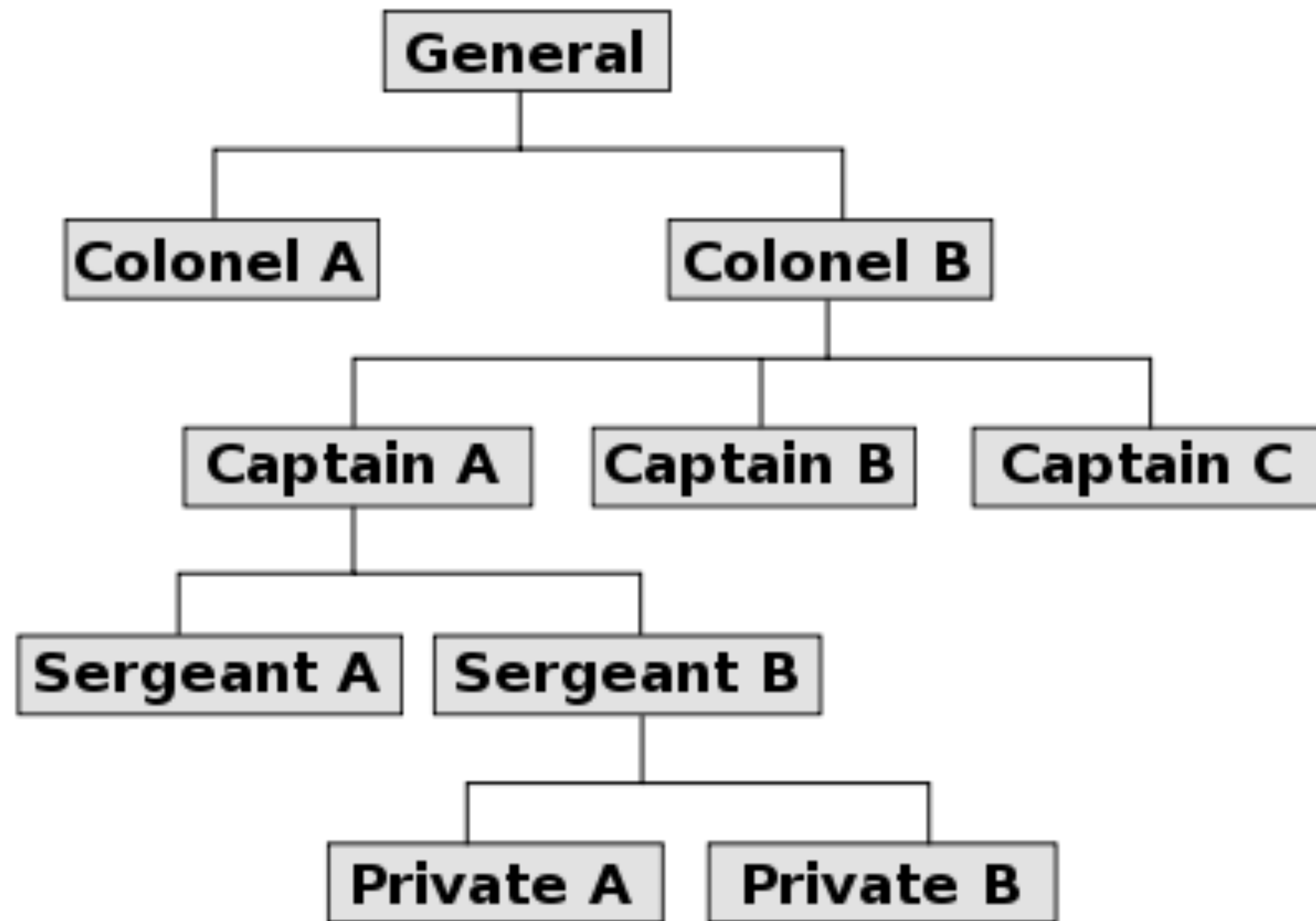
- 抽象之后，接口的复杂度变小，
- 接口和实现之间达成一种契约。



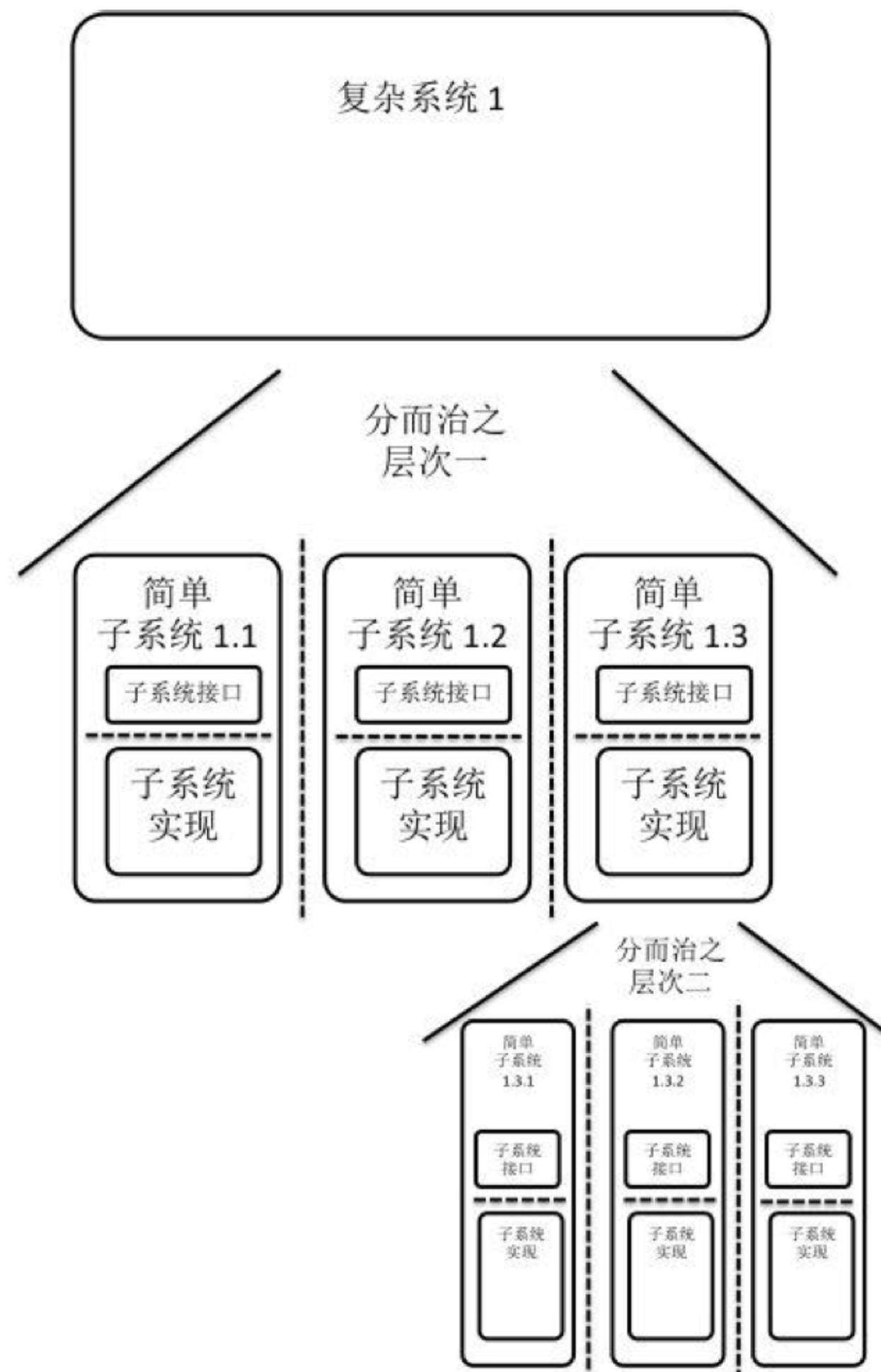
分形 — 科赫雪花



套娃



组织结构



# 分解与抽象的并用和层次性

3. Program = Algorithm + Data  
Structure

# Program

- Algorithms + Data Structures = Programs is a 1976 book written by [Niklaus Wirth](#) covering some of the fundamental topics of computer programming, particularly that algorithms and data structures are inherently related.

算法是计划、过程、步骤。

数据结构是操作的目标、对象。

算法和数据结构是配合的！

## 4. 算法建模 — 三种机制（基本 表达式、分解、抽象）

# 每种语言都会提供的三种机制

- 基本表达式
- 分解（组合）的方法
- 抽象的方法（后续章节再展开）

# 基本表达式

# 基本表达式

- 数字
  - 数学运算
  - $3 * 5$
- 逻辑
  - 逻辑运算
  - True & False

分解（组合）

# 树形表示法

- $( * ( + 2 ( * 4 6 ) ) ( + 3 5 7 ) )$

抽象

# 复合

- 定义平方
  - ( define ( square x) (\* x x) )
- 定义平方和
  - ( define ( sum-of-squares x y)
  - (+ (square x) (square y) ) )

\*...\* //打印n颗星

# 参数与变量

- `def printStars(n):` # n 是形参
- `count = 0` # count 是局部变量
- `while (count < n):`
- `print("*")`
- `count = count + 1`
- `if __name__ == "__main__":`
- `printStars(9)` # 9 是实参

# 5. 算法建模 — 两种思路（迭代与递归）

# 线性的递归和迭代

- 计算 $n$ 的阶乘
  - 递归
  - 迭代

# 递归

- (define (factorial n)
  - (if (= n 1)
    - 1
    - (\* n (factorial (- n 1))))))

# 计算6! 的线性递归过程

- (factorial 6)
- (\* 6 (factorial 5))
- (\* 6 (\* 5 (factorial 4)))
- (\* 6 (\* 5 (\* 4 (factorial 3))))
- (\* 6 (\* 5 (\* 4 (\* 3 (factorial 2)))))
- (\* 6 (\* 5 (\* 4 (\* 3 (\* 2 (factorial 1)))))
- (\* 6 (\* 5 (\* 4 (\* 3 (\* 2 1)))))
- (\* 6 (\* 5 (\* 4 (\* 3 2))))
- (\* 6 (\* 5 (\* 4 6)))
- (\* 6 (\* 5 24))
- (\* 6 120)
- 720

# 迭代

- (define ( factorial n)
  - (fact-iter 1 1 n)
- (define (fact-iter product counter max-count)
  - (if (> counter max-count)
    - product
    - (fact-iter ( \* counter product)
      - (+ counter 1)
      - max-count))))

# 计算6! 的线性迭代过程

- (factorial 6)
- (fact-iter 1 1 6)
- (fact-iter 1 2 6)
- (fact-iter 2 3 6)
- (fact-iter 6 4 6)
- (fact-iter 24 5 6)
- (fact-iter 120 6 6)
- (fact-iter 720 7 6)
- 720

# 案例 - 利用牛顿法求平方根

- 做什么
  - $y = \sqrt{x}$
  - $x$  的平方 =  $y$
- 怎么做
  - 牛顿的逐步逼近方法

# 手动计算2的平方根

猜测	商	平均值
1	$2/1=2$	$(2+1)/2=1.5$
1.5	$2/1.5=1.3333$	$(1.3333+1.5)/2=1.4167$
1.4167	$2/1.4167=1.4118$	$(1.4167+1.4118)/2=1.4142$
1.4142	...	...

# 步骤

- sqrt
  - sqrt-iter
    - good-enough
      - square
      - abs
    - improve
      - average

# Scheme语言版

- (define (sqrt-iter guess x)
  - (if (good-enough? guess x)
    - guess
    - (sqrt-iter (improve guess x)
  - x)))
- (define (improve guess x)
  - (average guess (/ x guess)))
- (define (average x y)
  - (/ (+ x y) 2))
- (define (good-enough? guess x)
  - (< (abs (- (square guess) x)) 0.001))
- (define (sqrt x)
  - (sqrt-iter 1.0 x))

# C语言版

- 1 #define ABS(VAL) (((VAL)>0)?(VAL):(-(VAL)))
- 2 //用牛顿迭代法求浮点数的平方根
- 3 double mysqrt(float x) {
- 4     double g0,g1;
- 5     if(x==0)
- 6         return 0;
- 7     g0=x/2;
- 8     g1=(g0+x/g0)/2;
- 9     while(ABS(g1-g0)>0.01)
- 10     {
- 11         g0=g1;
- 12         g1=(g0+(x/g0))/2;
- 13     }
- 14     return g1;
- 15 }

# Python语言版

- `c = input()`
- `err = 1e-15`
- `t = c`
- `while abs(t - c/t)>err:`
- `t = (c/t+t)/2.0`
- `print(t)`
-

# 对比

- (factorial 6)
- (\* 6 (factorial 5))
- (\* 6 (\* 5 (factorial 4)))
- (\* 6 (\* 5 (\* 4 (factorial 3))))
- (\* 6 (\* 5 (\* 4 (\* 3 (factorial 2)))))
- (\* 6 (\* 5 (\* 4 (\* 3 (\* 2 (factorial 1))))))
- (\* 6 (\* 5 (\* 4 (\* 3 (\* 2 1)))))
- (\* 6 (\* 5 (\* 4 (\* 3 2))))
- (\* 6 (\* 5 (\* 4 6)))
- (\* 6 (\* 5 24))
- (\* 6 120)
- 720

- (factorial 6)
- (fact-iter 1 1 6)
- (fact-iter 1 2 6)
- (fact-iter 2 3 6)
- (fact-iter 6 4 6)
- (fact-iter 24 5 6)
- (fact-iter 120 6 6)
- (fact-iter 720 7 6)
- 720

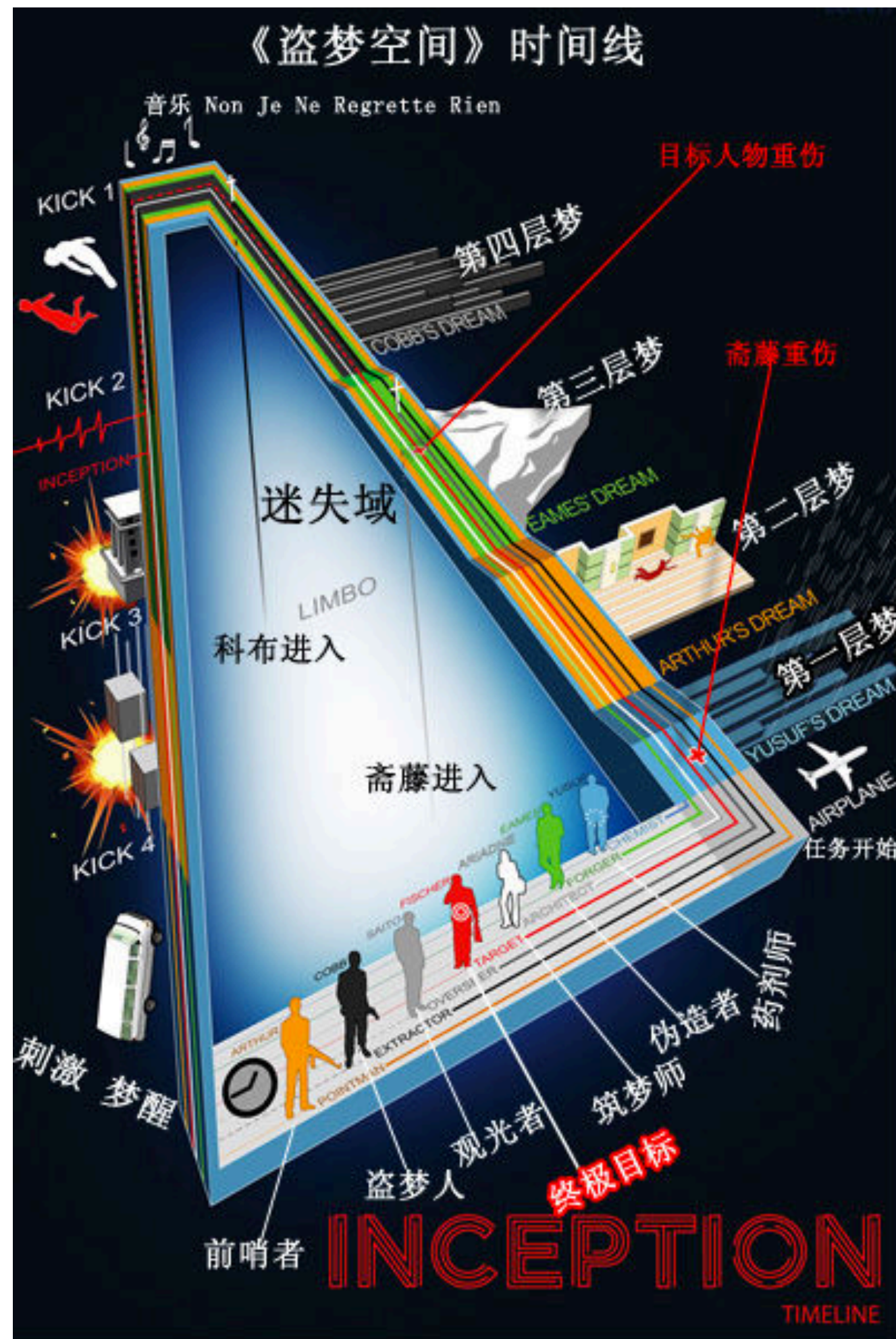
# 斐波拉契数列 — 迭代

- `#include <iostream>`
- `using namespace std;`
- `//迭代实现斐波那契数列`
- `long fab_iteration(int index)`
- `{`
- `if(index == 1 || index == 2)`
- `{`
- `return 1;`
- `}`
- `else`
- `{`
- `long f1 = 1L;`
- `long f2 = 1L;`
- `long f3 = 0;`
- `for(int i = 0; i < index-2; i++)`
- `{`
- `f3 = f1 + f2; //利用变量的原值推算出变量的一个新值`
- `f1 = f2;`
- `f2 = f3;`
- `}`
- `return f3;`
- `}`
- `}`

# 斐波拉契数列 — 递归

- //递归实现斐波那契数列
- long fab\_recursion(int index)
- {
- if(index == 1 || index == 2)
- {
- return 1;
- }
- else
- {
- return fab\_recursion(index-1)+fab\_recursion(index-2);   //递归求值
- }
- }

# 对比



思考题：

加法可不可以用递归和迭代表示？

# 6. 数据建模

\* #

// 打印 \* #

// = 打印一颗\* + 打印一颗#

基于算法建模的方案：

```
print("*")  
print("#")
```

基于数据建模的方案：

```
print("*#")
```

# 基础数据

- 整数
- 浮点数
- 布尔值

# 数据的组合

- 数组
  - 同一类数据的组合
- 结构体 struct
  - 不同数据的组合
- 对象
  - 数据和行为的组合

# 数据的抽象

- 有序对(Ordered Pair)

# 有序对的定义

- Wiener's definition

$$(a, b) := \{\{\{a\}, \emptyset\}, \{\{b\}\}\}.$$

- Haudorff's definition

$$(a, b) := \{\{a, 1\}, \{b, 2\}\}$$

- Kuratowski definition

$$(a, b)_K := \{\{a\}, \{a, b\}\}.$$

# 有序对性质

Let  $(a_1, b_1)$  and  $(a_2, b_2)$  be ordered pairs. Then the characteristic (or *defining*) property of the ordered pair is:

$(a_1, b_1) = (a_2, b_2)$  if and only if  $a_1 = a_2$  and  $b_1 = b_2$ .

# 证明

- If  $a = b$ :
  - $(a, b)K = \{\{a\}, \{a, b\}\} = \{\{a\}, \{a, a\}\} = \{\{a\}\}.$
  - $(c, d)K = \{\{c\}, \{c, d\}\} = \{\{a\}\}.$
  - Thus  $\{c\} = \{c, d\} = \{a\}$ , which implies  $a = c$  and  $a = d$ . By hypothesis,  $a = b$ . Hence  $b = d$ .
- If  $a \neq b$ , then  $(a, b)K = (c, d)K$  implies  $\{\{a\}, \{a, b\}\} = \{\{c\}, \{c, d\}\}.$ 
  - Suppose  $\{c, d\} = \{a\}$ . Then  $c = d = a$ , and so  $\{\{c\}, \{c, d\}\} = \{\{a\}, \{a, a\}\} = \{\{a\}, \{a\}\} = \{\{a\}\}.$  But then  $\{\{a\}, \{a, b\}\}$  would also equal  $\{\{a\}\}$ , so that  $b = a$  which contradicts  $a \neq b$ .
  - Suppose  $\{c\} = \{a, b\}$ . Then  $a = b = c$ , which also contradicts  $a \neq b$ .
  - Therefore  $\{c\} = \{a\}$ , so that  $c = a$  and  $\{c, d\} = \{a, b\}.$
  - If  $d = a$  were true, then  $\{c, d\} = \{a, a\} = \{a\} \neq \{a, b\}$ , a contradiction. Thus  $d = b$  is the case, so that  $a = c$  and  $b = d$ .

思考题：  
如何表达有理数？

思考题：

如何表达树形结构数据？

# 7 编程的典型场景 — 数据处理

# 编程的典型场景

- 数据处理
  - 输入
  - 处理
  - 输出

# 求两个数和

- 输入
  - 控制台输入2个数
- 处理
  - 将输入String转出int
  - 求和
- 输出
  - 输出和

用日志记录数据

你想要执行的任务	此任务最好的工具
对于命令行或程序的应用，结果显示在控制台。	<code>print()</code>
在对程序的普通操作发生时提交事件报告(比如：状态监控和错误调查)	<code>logging.info()</code> 函数(当有诊断目的需要详细输出信息时使用 <code>logging.debug()</code> 函数)
提出一个警告信息基于一个特殊的运行时事件	<code>warnings.warn()</code> 位于代码库中，该事件是可以避免的，需要修改客户端应用以消除告警  <code>logging.warning()</code> 不需要修改客户端应用，但是该事件还是需要引起关注
对一个特殊的运行时事件报告错误	引发异常
报告错误而不引发异常(如在长时间运行中的服务端进程的错误处理)	<code>logging.error()</code> ， <code>logging.exception()</code> 或 <code>logging.critical()</code> 分别适用于特定的错误及应用领域

# 日志使用场景

级别	何时使用
DEBUG	细节信息，仅当诊断问题时适用。
INFO	确认程序按预期运行
WARNING	表明有已经或即将发生的意外（例如：磁盘空间不足）。程序仍按预期进行
ERROR	由于严重的问题，程序的某些功能已经不能正常执行
CRITICAL	严重的错误，表明程序已不能继续执行

# 日志级别

# 日志

```
import logging
logging.warning('Watch out!') # will print a message to the console
logging.info('I told you so') # will not print anything
```

如果你在命令行中输入这些代码并运行，你将会看到：

```
WARNING:root:Watch out!
```

# 日志

如果你的程序包含多个模块，这里有一个如何组织日志记录的示例：

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()

# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

如果你运行 myapp.py，你应该在 myapp.log 中看到：

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```