



# 数据结构与算法 (Python版)

## 什么是算法分析

陈斌 北京大学 [gischen@pku.edu.cn](mailto:gischen@pku.edu.cn)

# 对比程序，还是算法？

## ❖ 如何对比两个程序？

看起来不同，但解决同一个问题的程序，哪个“更好”？

## ❖ 程序和算法的区别

算法是对问题解决的分步描述

程序则是采用某种编程语言实现的算法，同一个算法通过不同的程序员采用不同的编程语言，能产生很多程序

# 累计求和问题

❖ 我们来看一段程序，完成从1到n的累加，  
输出总和

设置累计变量=0

从1到n循环，逐次累加到累计变量

返回累计变量

```
def sumOfN(n):  
    theSum = 0  
    for i in range(1, n + 1):  
        theSum = theSum + i  
    return theSum  
  
print(sumOfN(10))
```

# 累计求和问题

## ❖ 再看第二段程序，是否感觉怪怪的？

但实际上本程序功能与前面那段相同

这段程序失败之处在于：变量命名词不达意，以及包含了无用的垃圾代码

```
def foo(tom):  
    fred = 0  
    for bill in range(1, tom + 1):  
        barney = bill  
        fred = fred + barney  
    return fred  
  
print(foo(10))
```

# 算法分析的概念

- ❖ 比较程序的“好坏”，有更多因素  
代码风格、可读性等等
  - ❖ 我们主要感兴趣的是算法本身特性
  - ❖ 算法分析主要就是从计算资源消耗的角度来评判和比较算法  
更高效利用计算资源，或者更少占用计算资源的算法，就是好算法
- 从这个角度，前述两段程序实际上和基本相同的，它们都采用了一样的算法来解决累计求和问题

# 说到代码风格和可读性

## ❖ 为什么Python的强制缩进是好的？

语句块功能和视觉效果统一

## ❖ 苹果公司一个低级Bug

由于C语言源代码书写缩进对齐的疏忽

造成SSL连接验证被跳过

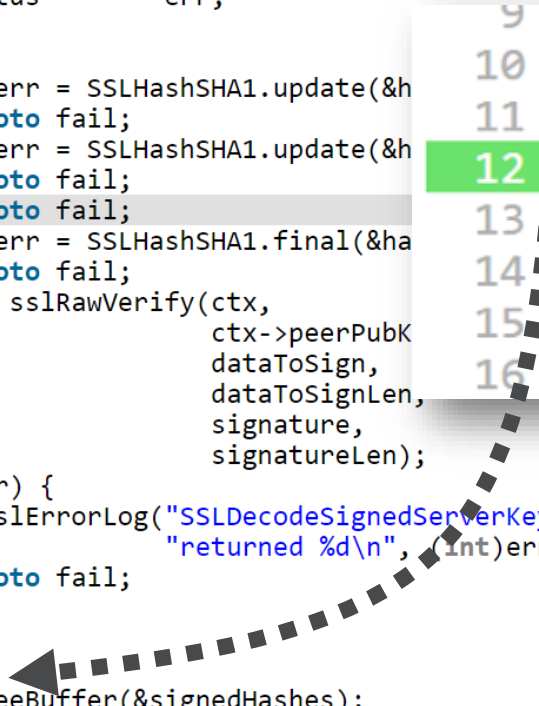
2014.2.22修正iOS7.0.6



# 说到代码风格和可读性

## ❖ 代码不像看起来那样运行

```
1 static OSStatus
2 SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedPa
3     uint8_t *signature, UInt16 signatureLen)
4 {
5     OSStatus      err;
6     ...
7
8     if ((err = SSLHashSHA1.update(&h
9         goto fail;
10    if ((err = SSLHashSHA1.update(&h
11        goto fail;
12    goto fail;
13    if ((err = SSLHashSHA1.final(&ha
14        goto fail;
15    err = sslRawVerify(ctx,
16        ctx->peerPubK
17        dataToSign,
18        dataToSignLen,
19        signature,
20        signatureLen);
21    if(err) {
22        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
23            "returned %d\n", (int)err);
24        goto fail;
25    }
26
27 fail:
28    SSLFreeBuffer(&signedHashes);
29    SSLFreeBuffer(&hashCtx);
30    return err;
31 }
```



The diagram illustrates a jump in code execution. A dashed arrow originates from line 12, which contains the statement `goto fail;`, and points to the `fail:` label at the bottom of the function. This demonstrates how the `goto` statement can lead to a different part of the code, potentially bypassing other logic.



# 计算资源指标

❖ 那么何为计算资源?

❖ 一种是算法解决问题过程中需要的**存储空间或内存**

但存储空间受到问题自身数据规模的变化影响

要区分哪些存储空间是问题本身描述所需，哪些是算法占用，不容易

❖ 另一种是算法的**执行时间**

我们可以对程序进行实际运行测试，获得真实的运行时间



# 运行时间检测

## ❖ Python中有一个time模块，可以获取计算机系统当前时间

在算法开始前和结束后分别记录系统时间，即可得到运行时间

```
>>> help(time.time)
Help on built-in function time in module time:

time(...)
    time() -> floating point number

    Return the current time in seconds since the Epoch.
    Fractions of a second may be present if the system clock provides them.
```

```
>>> import time
>>> time.time()
1565878560.88039
>>>
```

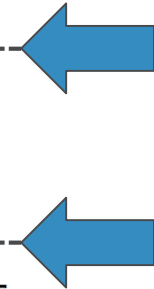
# 运行时间检测

## ❖ 累计求和程序的运行时间检测

用time检测总运行时间

返回累计和，以及运行时间（秒）

```
1  import time
2
3
4  def sumOfN2(n):
5      start = time.time()
6      theSum = 0
7      for i in range(1, n + 1):
8          theSum = theSum + i
9      end = time.time()
10     return theSum, end - start
11
12
13 for i in range(5):
14     print("Sum is %d required %10.7f seconds"
15           % sumOfN2(10000))
```



# 运行时间检测

## ❖ 在交互窗口连续运行5次看看

1到10,000累加

每次运行约需0.0007秒

```
Sum is 50005000 required 0.0007980 seconds
Sum is 50005000 required 0.0007021 seconds
Sum is 50005000 required 0.0007031 seconds
Sum is 50005000 required 0.0007219 seconds
Sum is 50005000 required 0.0007060 seconds
```

# 运行时间检测

## ❖ 如果累加到100,000?

看起来运行时间增加到10,000的10倍

```
Sum is 5000050000 required 0.0078530 seconds
Sum is 5000050000 required 0.0078511 seconds
Sum is 5000050000 required 0.0087960 seconds
Sum is 5000050000 required 0.0082700 seconds
Sum is 5000050000 required 0.0077040 seconds
```

## ❖ 进一步累加到1,000,000?

运行时间又是100,000的10倍了

```
Sum is 500000500000 required 0.0817859 seconds
Sum is 500000500000 required 0.0781529 seconds
Sum is 500000500000 required 0.0803380 seconds
Sum is 500000500000 required 0.0783160 seconds
Sum is 500000500000 required 0.0776238 seconds
```

## 第二种无迭代的累计算法

### ❖ 利用求和公式的无迭代算法

```
17 def sumOfN3(n):  
18     start = time.time()  
19     theSum = (n * (n + 1)) / 2  
20     end = time.time()  
21     return theSum, end - start
```

### ❖ 采用同样的方法检测运行时间

10,000; 100,000; 1,000,000

10,000,000; 100,000,000

```
Sum is 50005000 required 0.0000010 seconds  
Sum is 5000050000 required 0.0000000 seconds  
Sum is 500000500000 required 0.0000010 seconds  
Sum is 50000005000000 required 0.0000000 seconds  
Sum is 5000000050000000 required 0.0000169 seconds
```

## 第二种无迭代的累计算法

### ❖ 需要关注的两点

这种算法的运行时间比前种都短很多

运行时间与累计对象 $n$ 的大小没有关系（前种算法是倍数增长关系）

### ❖ 新算法运行时间几乎与需要累计的数目无关

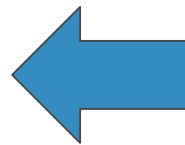
# 运行时间检测的分析

## ❖ 观察一下第一种迭代算法

包含了一个循环，可能会执行更多语句

这个循环运行次数跟累加值 $n$ 有关系， $n$ 增加，循环次数也增加

```
def sumOfN(n):  
    theSum = 0  
    for i in range(1, n + 1):  
        theSum = theSum + i  
    return theSum  
  
print(sumOfN(10))
```





# 运行时间检测的分析

- ❖ 但关于运行时间的实际检测，有点问题  
关于编程语言和运行环境
- ❖ 同一个算法，采用不同的编程语言编写，放在不同的机器上运行，得到的运行时间会不一样，**有时候会大不一样**：  
比如把非迭代算法放在老旧机器上跑，甚至可能慢过新机器上的迭代算法
- ❖ 我们需要更好的方法来衡量算法运行时间  
这个指标与**具体**的机器、程序、运行时段都**无关**