链滴

数据结构 - 程序复杂度分析

作者：chestnutLeaves
原文链接：https://ld246.com/article/1657995992565
来源网站：链滴

　　数据结构和算法，是解决 “快” 和 ” 省 “ 的问题，如何衡量算法的执行效率，就用到时间复杂度分析、空间复杂度分析

一、复杂度分析的意义

问题：通过监控和统计，能实际获取算法执行的时间和内存，为何仍需要时间复杂度分析和空间复杂分析?

原因：

1. 通过运行代码来统计复杂度也是有名字的，称为 ” 事后统计法”
2. 事后统计法，看似能获取精确数据，但是受到运行环境影响很大
3. 事后统计法，受测试数据的影响很大，如果数据规模太小体现不出算法差距。

**结论: **我们需要一种，不依赖测试环境和测试数据就能粗略估计算法执行效率的方法。

二、大O复杂度分析法和时间复杂度

```
//示例代码 求累加和
int cal(int n){
    int sum=0;
    int i = 1;
    for(;i<n;++i){
        sum+=i;
    }
    return sum;
}
```

1. **假设每条语句执行时间都是unit_time **

如上代码，将每条语句执行时间粗略估计，当成统一的时间unit_time，那么代码执行的语句数量可拿来评估方法的执行时间；那么如上代码 3、4、7 占用3个unit_time ,5、6 共占用2n个unit_time 那这段代码耗时可写为，（2n+3）*unit_time

1. **规律：代码执行时间 T(n) 与每一条语句执行次数成正比。**
2. **规律公式：T(n)=O(f(n))**

** **其中T(n) 表示代码执行总时间，n表示数据规模；f(n) 表示语句执行次数的累加和，大O O() 表代码执行时间T(n) 与f(n) 成正比

1. **时间复杂度 大O时间复杂度，并不具体表示代码真正的执行时间，只是表示随着数据规模增大时T(n的变化趋势，因此也称为 渐进时间复杂度 （asymptotic time complexity）**
简称**时间复杂度**
2. **忽略常量和系数 **当n足够大时，公式中的，常量、系数和增长的趋势就无关了，因此可以忽略如此一来上面代码复杂度可以记录为 T(n)=O(n)

三、时间复杂度分析法

1 加法法则

加法法则：代码总复杂度，等于量级最大的那段代码的复杂度。

代码举例：

```
int cal(int n){

    //复杂度是常量 执行100次
    int sum_1=0;
    int p=1;
    for(;p<=100;++p){
        sum_1=sum_1+p;
    }
    //复杂度是 O(n) 执行n次
    int sum_2=0;
    int q=1;
    for(;q<=n;++1){
        sum_2=sum_2+q;
    }
    //复杂度是 O(n²)
    int sum_3=0;
    int j=1;
    int i=1;
    for(;i<=n;++i){
        j=1;
        for(j<=n;++j){
            sum_3=sum_3+i*j;
        }
    }

    return sum_1+sum_2+sum_3;
}
```

其中三段代码中，复杂度是 O(n)、O(n²)、常量，**复杂度取其中最大值，所以是O(n²)。**

加法法则用公式表达

T1(n)=O(f(n)); T2(n)=O(g(n));

T(n)=T1(n)+T2(n)=max(O(f(n)),O(g(n)))=O(max(f(n),g(n)))

2 乘法法则

乘法法则：嵌套代码的复杂度，等于嵌套内外代码复杂度的乘积

T1(n)=O(f(n)); T2(n)=O(g(n));

T(n)=T1(n) x T2(n)=O(f(n)) x O(g(n))=O(f(n) x g(n))

代码举例：

```
//f(i) 的复杂度是O(n) 外层循环的复杂度是O(n) 所以 复杂度是O(n²)
int cal(int n){
```

```
    int ret=0;
    int i=1;
    for(;p<=n;++i){
        ret=ret+f(i)
    }
}

//复杂度是 O(n)
int f(int n){
    int sum=0;
    int i=1;
    for(;p<=n;++i){
        sum=sum+i;
    }
}
```

常见的时间复杂度量级

O(1) 常量级时间复杂度

只要代码的执行时间，不随着数据规模n变化而变化，代码就是常量级别时间复杂度，统一计为O(1)。

O(log^n)、O(nlog^n) 对数阶时间复杂度

代码举例

```
i=1;
while(i<=n){
    i=i*2;
}
```

分析：根据加法法则，这段代码的时间复杂度，取决于第三行代码执行的次数而第三行代码执行的条为 i*2^x > n，在上面这段代码中，那么x=log2n（2为底，n的对数），因此这段代码复杂度就是 O(2n)。

忽略底数：无论以2或者3或者10为底数，都可以把所有的对数时间复杂度记为O(log**n**)，因为底数是个系数，不会对趋势造成影响。根据换底公式所有不同底数的对数都可以变换成系数和相同底数的对相乘，而时间复杂度统计时一般忽略常量和系数，所以可以忽略底数。

O(nlogn)：即是根据上面的乘法法则，如果对复杂度为O(logn
的代码，循环执行了N遍，那么复杂度就是**O(nlogn)**了

O(m+n)、O(mn) 时间复杂度

代码举例

```
int cal(int m ,int n){
    int sum_1=0;
    int i=1;
    for(;i<=m;++i){
        sum_1=sum_1+i;
    }
```

```
    int sum_2=0;
    int j=1;
    for(;j<=n;++j){
        sum_2=sum_2+j;
    }

}
```

对于上面的代码，m和n是两个无关的数据规模，无法实现评估哪个更大，所以标识时间复杂度都需保留，所以时间复杂度可以记为O(m+n)

四、空间复杂度分析

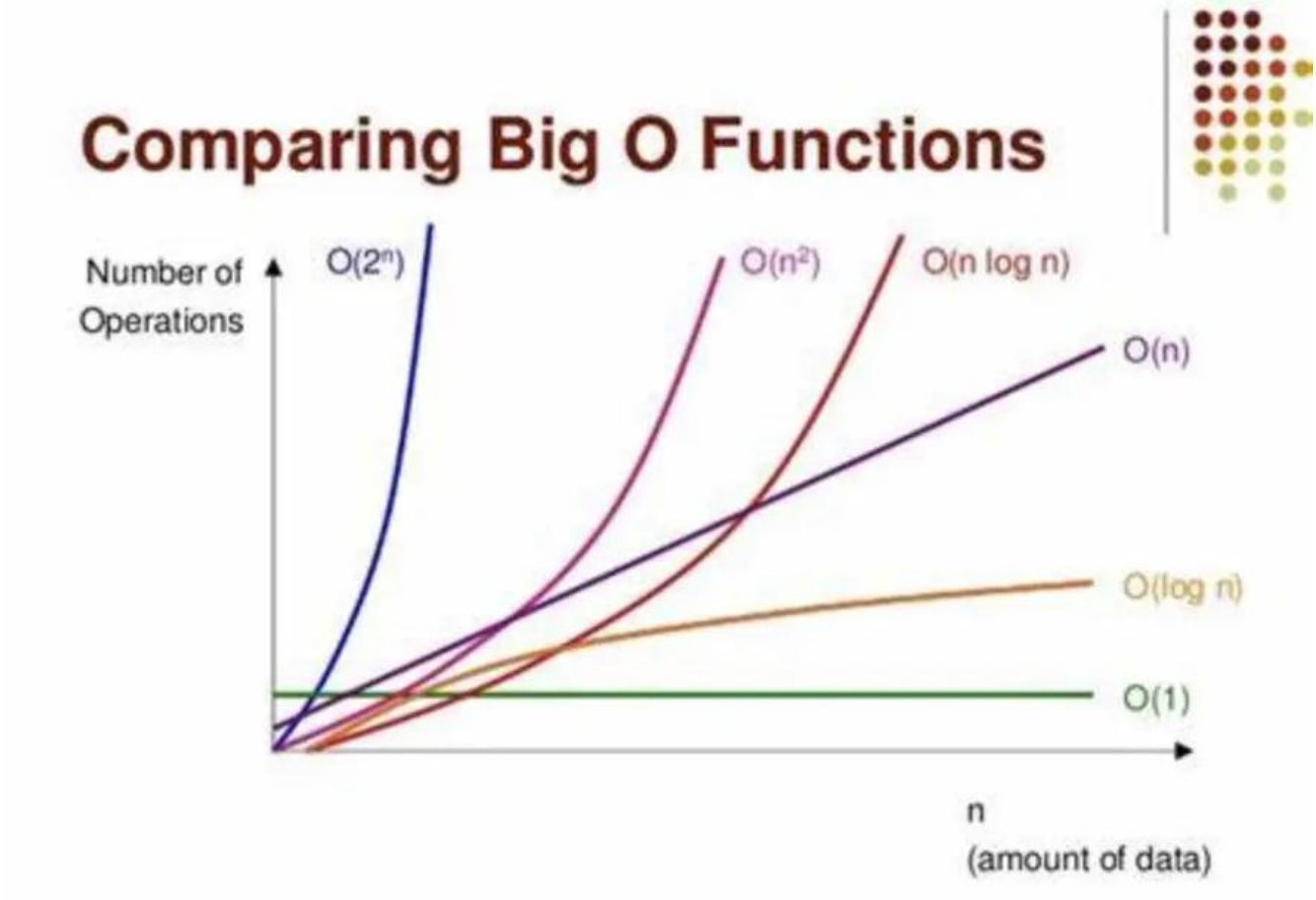
空间复杂度全称是，渐进空间复杂度(asymptoyic space complexity)表示算法的存储空间，与数据模之间的增长关系。

```
public void reverse(int a[],int n){
    int tmp[]=new int[n];
    //将a[] 逆向存入到tem[]
    for(int i=0;i<n;++i){
        temp[i]=a[n-i-1];
    }
    //将tem[] 一对一存回a[]
    for(int i=0;i<n;++i){
        a[i]=tem[i]
    }
}
```

对于上面的代码，申请了变量i作为游标，又申请了大小为n的tem[]数组，其中i的存储空间不会随着n变化而变化，所以空间复杂度就是O(n)

常见的空间复杂度有：O(1)、O(n)、$O(n^2)$、O(logn)、O(nlogn)，其中的对数级复杂度，常见于递代码。

五、常见复杂度的增长趋势对比



六、最好、最坏、平均、均摊时间复杂度

作用：更全面的表示一段代码的执行效率。

```
//复杂度为O(n)的写法
//n表示，数组array的长度
int find(int[] array,int n,int x){
    int i=0;
    int pos=1;
    for(;i<n;++i){
        if(array[i]==x){
            pos=i;
        }
    }
    return pos;
}

//优化后，时间复杂度还为O(n)吗?
//n表示，数组array的长度
int find(int[] array,int n,int x){
    int i=0;
    int pos=1;
    for(;i<n;++i){
        if(array[i]==x){
            pos=i;
            break;
        }
```

原文链接：数据结构 - 程序复杂度分析

```
    }
    return pos;
}
```

如代码示例中，第一种写法的复杂度为O(n) 但是优化后的代码，仍用O(n)表示的话，便显得不是很切了。所以需要引入三个概念

1. 最好情况时间复杂度
2. 最坏情况时间复杂度
3. 平均时间复杂度

最好情况时间复杂度

则是要找的变量x正好是数组第一个元素，那么代码的复杂度为o(1)

最坏情况复杂度

则是类似第一种情况，数组中不存在对应的x那么需要遍历一边数组，时间复杂度为O(n)

平均时间复杂度

则是在计算每种情况时，加入每种情况出现的期望值然后进行计算。

1. 首先可以计算出，查找x在数组中的位置，包括不存在的情况，共有n+1种情况
2. 计算每种情况遍历的个数累加，除以所有情况可以得出需要遍历元素个数的平均值 (1+2+3+…+n+)/(n+1)=n(n+3)/2(n+1) 简化后复杂度为O(n)
3. 按照第二种情况，虽然计算出了平均遍历元素个数，但是并未考虑这n+1种情况的概率是否相同，以加上每种情况出现的期望一起计算才更合理
4. 带入每种情况的期望，首先假设变量x出现在数组里和不出现在数组里的概率各是1/2,那么出现在组里任意一个地方的概率就是1/2n，不出现在数组里的概率就是1/2
5. 带入期望后，第二步的公式变成了 1x1/2n +2x1/2n ……+nx1/2n +n*1/2=(3n+1)/4 去除系数和量后，复杂度仍然是 O(n)

均摊时间复杂度-特殊的平均时间复杂度

代码示例

```
//array和count是类成员变量，或者全局变量
int[] array=new int[n];
int count=0;//标识数组中的元素个数

//尝试插入一个值 val 如果数组已满，那么求和打印后，清空数组从头插入
void insert(int val){
    if(count==array.length){
        int sum=0;
        for(int i=0;i<array.length;++i){
            sum=sum+array[i];
        }
        System.out.println(sum);
```

```
        count=0;
    }
    array[count]=val;
    count++;
}
```

1. 对于上述代码，如果要计算平均时间复杂度，按照之前平均时间复杂度的算法来计算的话计算过程下

    1. 如果数组有剩余空间，位置在0~n之间，复杂度都是O(1)
    2. 如果数组已满，那么对应的复杂度是O(n)
    3. 1和2共n+1种情况，且每种情况出现的概率相等都为 1/(n+1)
    4. 根据加权平均值计算方法，他的平均时间复杂度如下
    5. 1x1/(n+1)+1x1/(n+1)+⋯⋯+1x1/(n+1)+nx1/(n+1)=O(n)

2. 但是对于计算insert() 方法的平均时间复杂度，不需要如此无需引入概率论的知识，因为insert()对find()方法

    1. find()方法极端情况复杂度才为O(1)，而insert方法大部分情况下都是O(1)
    2. 对于insert方法，O(1) 和O(n)出现的情况是有规律的，O(n) 出现后必定伴随n-1个O(1)不断循环

3. 针对这样的情况，可以不用概率论的方法，而是引入一种更简单的分析方法**“摊还分析法” ，**于摊还分析法得到的时间复杂度，称为**”均摊时间复杂度”**

4. 使用摊还分析法，上面的insert()方法，每次O(n)的操作，都会跟着n-1次的O(1)操作，将O(n)的耗，均摊下来，n次执行的耗时为O(1)

5. 使用场景：连续的一组操作下，大部分复杂度很低，个别复杂度很高并且存在前后连贯的时序关系就可以将这一组操作一起分析，看看是否能均摊。一般在能应用均摊时间复杂度的场景中，均摊时间杂度一般等于最好时间复杂度。