# 520 第二周学习总结

# 学习内容总结

第二周学习的内容有：哈希表、映射集合、图、二叉树、递归、分治回溯等知识。

内容量挺大的，因为之前刚好第二周的时候面临学校的硕士毕设论文，实在没法挤出来时间，趁期中，补上。

实际上，在工作中使用得更多的当属set和map,前者可以起到过滤重复元素的作用，比如在爬虫的时候，可以放入到set里面来防止出现重复的元素。而map可以用key-value的方式让查询和使用更快速，LeetCode著名的第一题便可以用它很快和轻松解决，时间复杂度也没有直接二重遍历那么高。

哈希在后面还有LRU内容，期待。

哈希表本质是定义了一个哈希函数然后在计算之后把元素放入定义好的“桶”中。如果哈希值有冲突了，也就是出现了哈希碰撞了，可以提升维度，使用拉链式的方法解决。但是如果某一条链太长了，查找的时候时间复杂度会很悲催的退化成O(n)，这简直和链表没啥区别了。所以下面也有分析，在JDK8之后的HashMap中，在长度大于8的链表内部会转换成红黑树实现。

超哥分享的时间复杂度的这张图很赞：

计算机生成了可选文字:
复杂度分析
DataStructure
A又
Stac&
TimeComplexity
Average
Access
Search
Insertion
土山上
Qouhly-LinkedList习
(n))
旦山旦〔Y一Sh．卫000g（n））
O(log(n))O(log(n))
O(log(n))0（10g（n））
Deletion
为0
O(log(n)
0（10g（n））
Worst
Access
O(n)
O(n)
Search
O(n)
O(n)
Insertion
O(n)
〔00
O(n)
CartesianTree
a-Tree
N/A]
Clog(n))
粼@〈诩粼@甸习
000g（n））000g（n））O(log(n))oaog(n))0（10g@））0（10g（n））0（10g（n））
Red-BlackTreeO(log(n))为0>习0（10g（n））0（10g（n））〔00〈习0《》]0（10g（n））
Deletion
OCn)
O(n)
0（10g（n））
O(log(n))
O(log(n)
SpaceComplexity
Worst
O(n10(n)
O(n)
[00
O(n)
O(n)
Splay-Lee
AVLTree
KDTree
N/A'
O(log(n))O(log(n))O(log(n))
O(log(n))O(log(n))
@《囟在》]@《囟〈0习@0囟〈0习@0囟〈0习0（10g（n））0囟在>习000g（n））
诩习囗訌到

赞叹于前人的探索的成果，将数据结构整个大厦搭建得如此让人叹为观止，不同的结构又拥有自己的长处和短处，仿佛置身于一个不同凡响的数字世界。每次刷题的时候，都有这样的感慨。

第五课中介绍了字母异位词和字母异位词组的问题，这个题确实反复出现，因为是这周写的第二周内容，而在期中考试中却也碰上了。超哥说，这种问题写熟练了即可，不需要怕题目那看起来吓人的意思，再次强调题目刷一次是不够的，最大的误区就是题目只刷一次，而且可以养成一个好习惯，把算法题目分门别类地总结出来。

树的相关内容再怎么强调都不为过，因为在企业中使用范围和场景非常之多，比如数据库使用B+树进行存储，我曾经看到一篇经常面试别人的面试官写的文章，大致意思是，“很多工作了很多年的工程师，基本功不过关，比如现场写二叉树的层次遍历都写不出来”，所以，如果没准备好树相关的题目，我觉得别去面试了，不然面试官可能都把你列入黑名单。

树和图的差别主要是树没有环，而且经常的说法是“Linked List是特殊化的Tree，Tree是特殊化的Graph”。树的出现本身也体现了人们升维度，加速的这种思想。

树的前中后序的递归和非递归遍历是很重要和经典的，我花了两天，差不多熟练掌握。但是实际上，超哥在课上也有讲说，**有关树的面试题一般都是递归解决**的，因为如果现场去想BFS，往往写循环比较麻烦，而递归调用相较来说比较简单。

来到递归，其实递归和循环本质没有区别，递归利用了计算机本身的栈去简化计算的代码，比如斐波那契数列的计算。递归的代码比较容易，一般单独写一个函数，然后里面分为四步。但是我也在LeetCode上面看到高票的解法，比如求树的深度，直接利用题目的函数进行递归，这个多做，记住，还是大有裨益的。

最后，尽量不要做人肉递归，找到可以拆分成可重复解决的最近子问题，也就是拆分成可以用if-else, for-loop实现的问题。学习数学归纳法思维，也就是**观察、归纳、发现规律的能力**。

面试的题目一般五行，十行，最多20行解决，一般都有重复性。

希望能够养成一个习惯：在LeetCode国际站上面看到工整、简单、重剑无锋，大巧不工的代码，会有惊叹，会有抄下来并且练会并且写出来的冲动，不断提高自己写的代码，就算是真正“上道了”。

别说，我还真有这样的感触，例如在104树的深度那道题，LeetCode上面有一行代码可以解决的，我当时真的惊叹了，抄下来并且裱起来了。当然后面还有，就不多写了，因为有很多……

应当记住，二叉树的中序遍历输出的结果是有序的，可以用来验证是否为二叉搜索树。

来到分治和回溯，分治和回溯的本质，其实就是递归，可以理解是稍微复杂的递归。对于递归或者分治，遇到问题正确的解决思路，是去找它的重复性。遇到问题，就思考怎么去把它分成子问题。不能分成子问题的其实很少，非要说有，那就只能是很简单的问题……

总的来说，分治是一种思想，具体怎么做，差不多还是当做递归处理。

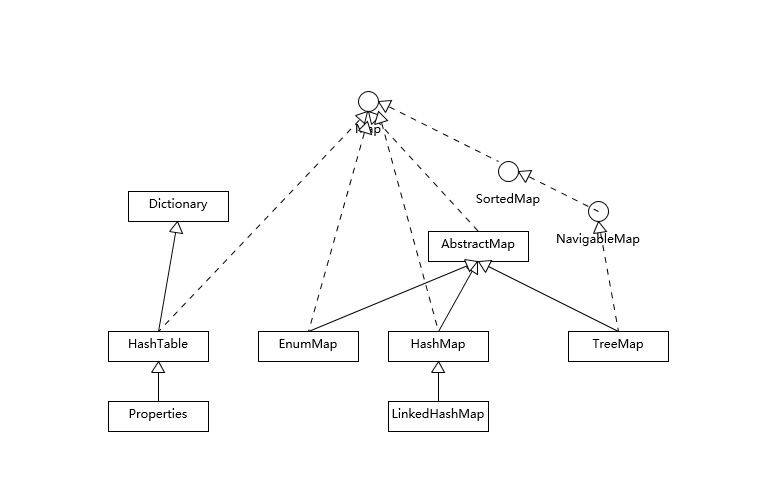
而回溯则一般使用最简单的递归来实现，而回溯和递归最主要的区别在于，回溯可以在一定程度上进行剪枝，以便加快递归的结束时间。

# HashMap相关内容总结

Java的早期的类库中，实际上提供的是Hashtable来实现哈希表的。Hashtable不支持null键和值，由于同步导致的性能开销，现在已经很少被使用了，而替代它的，正是HashMap。HashMap大致行为和Hashtable一致，只是它不是同步的，支持null键和值等。通常情况下HashMap执行get和put都是常数时间的性能，所以**它是绝大部分利用键值对存取的场景的首选**。比如，实现一个用户ID和用户信息对应的运行时存储结构。

有关HashMap的并发的问题是很有意思的而且经常被问道的内容，因为HashMap在并发环境中可能出现无限循环占用CPU、size不准确等诡异的问题所以经常被讨论。但是实际上因为HashMap明确声明了不是线程安全的数据结构，所以它不能被简单使用在多线程场景中的。

Map的整体结构（内容来源于）



可以看出除了HashTable继承的是Dictionary类，其他的Map都是继承AbstractMap的。大部分Map的使用场景，都是放入、访问或者删除，对顺序没有特别要求，那么可以说HashMap是最好的选择。HashMap的性能表现非常依赖于哈希码的有效性，所以要强调hashCode和equals的一些基本约定，比如：

* equals相等，hashCode也一定要相等
* 重写了hashCode，也一定要重写equals
* hashCode需要保持一致性，状态改变返回的哈希值仍然要一致
* equals的对称、反射、传递等特性

虽然LinkedHashMap和TreeMap都可以保证某种顺序，但是二者其实非常不同。

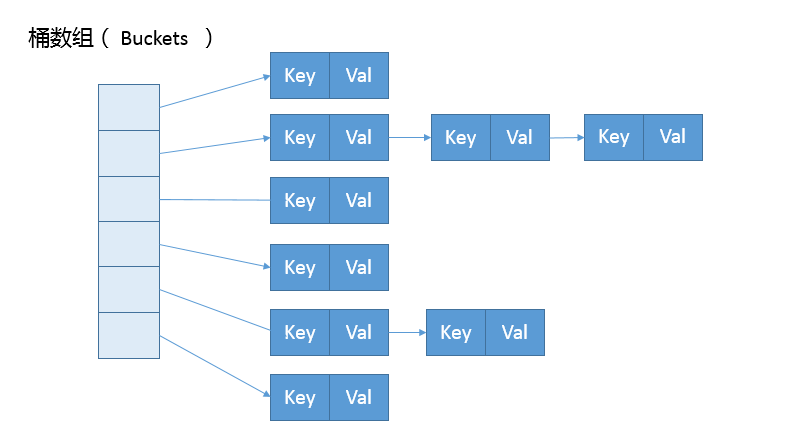
LinkedHashMap通常提供的是遍历顺序符合插入顺序，通过键值对（条目）维护一个双向链表。需要注意通过特定构造函数，我们可以创建反应访问顺序的实例，所谓的put、get、compute等，都算“访问”

HashMap的源码分析，更是**非常高频的面试题**。这里主要围绕三个点介绍讨论：

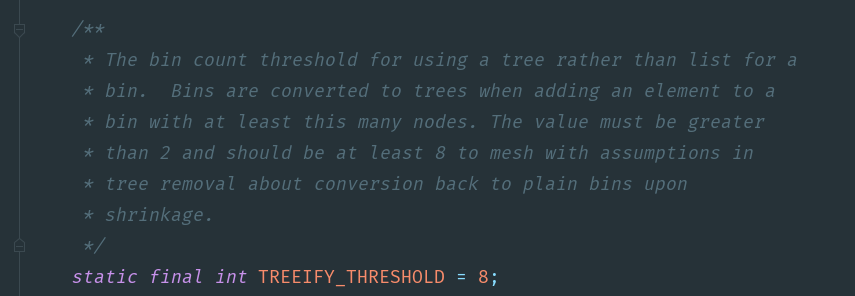
1. HashMap内部实现基本点分析
2. 容量（capacity）和负载系数（load factor）
3. 树化

首先，JDK8对HashMap的底层进行了优化，比如引入了红黑树的数据结构和扩容的优化等。

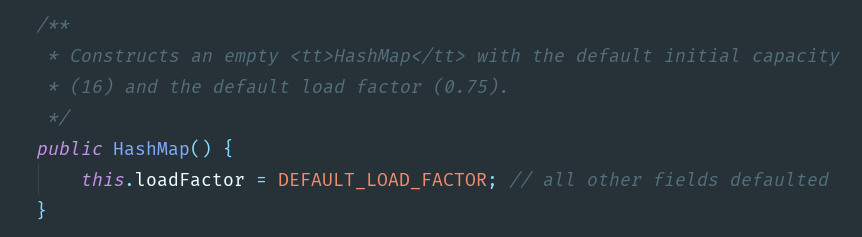
**HashMap的结构是数组和链表组合而成的复合结构**，数组会被分成一个一个的“桶”（bucket），通过哈希值决定键值对在这个数组中的寻址。哈希值相同的键值对，则以链表形式存储。如下图：



值得注意的是，如果某条链表大小超过阈值（从源码中可以看到TREEIFY\_THRESHOLD,8），那么图中的链表会被改造为树形结构。

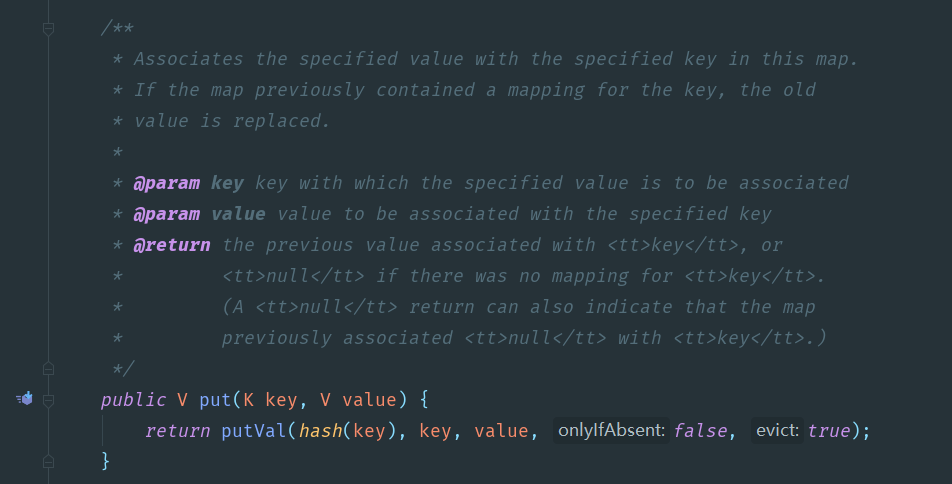


而HashMap在构造函数中可以看到，它似乎不是在最初就初始化好的，仅仅设置了一些初始值。源码中有四种构造方法，下面这个是最常用的，因为大多采用无参构造方法。它仅初始化了一些值



所以我们可以说HashMap也许是按照懒加载原则，在首次使用的时候被初始化。

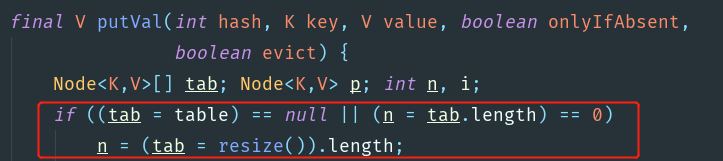
对于比较重要的put方法：



可以看到put关键的是调用了putVal这个函数，那么毫无疑问它是值得被好好研究一下的。



上图是putVal的所有部分，单独看几个部分的作用：

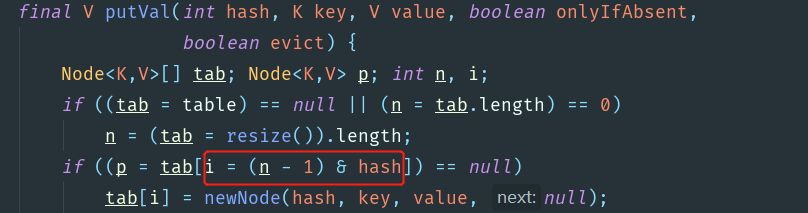


* 如果表格是空的，那么会调用resize()这个函数对tab进行初始化。
* resize方法主要有两个职责，创建初始存储表格，或者在容量不够用的时候进行扩容。
* 比如，在存放新的键值对的过程中，进行扩容的条件：

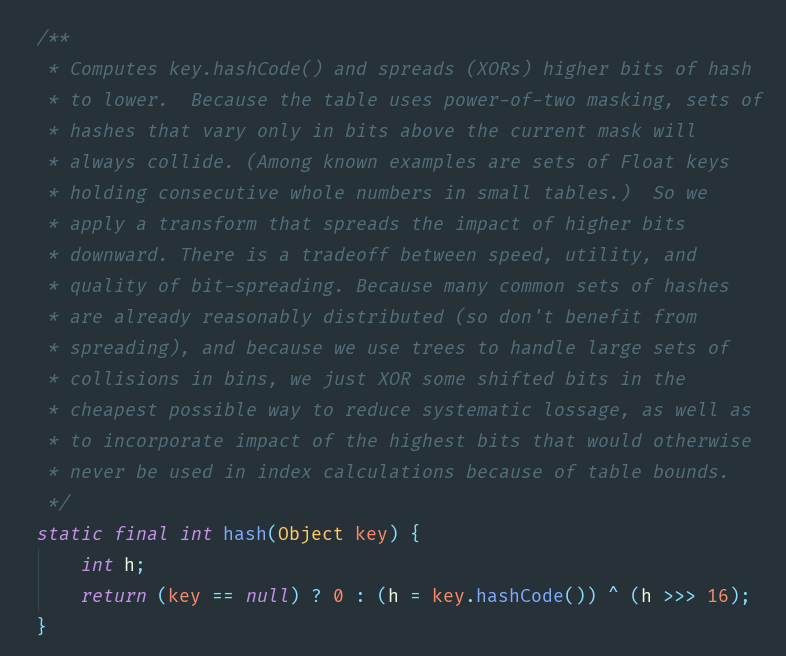


即，存放的size大于阈值了，会发生扩容。

而具体键值对在哈希表中所处的位置（数组的index）取决于这个位运算操作：



也就是：i = (n - 1) & hash

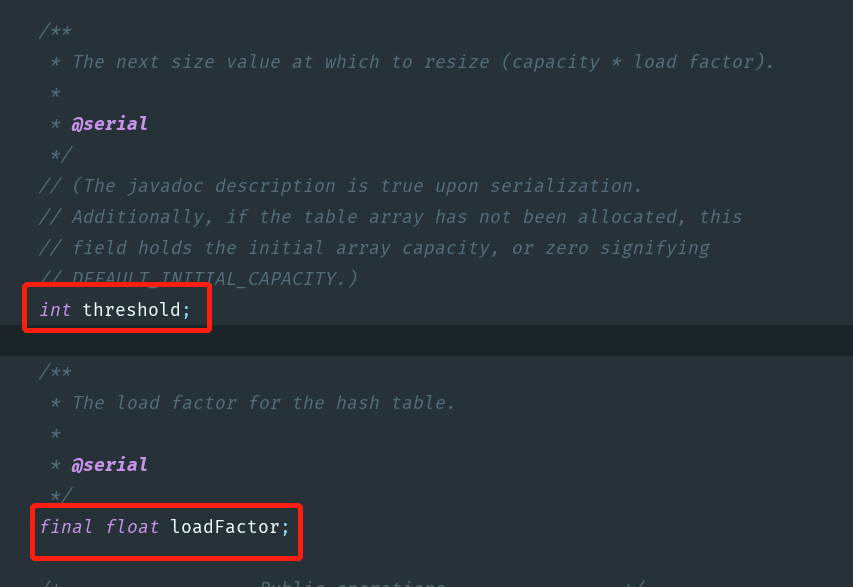


这段hash方法上方的注释解释道，这个方法目的是计算hashCode，也就是说得到的并不是这个key本身的hashCode，而是通过hash计算之后的hashCode。为什么这里需要将高位数据移动到低位进行异或（XOR）运算呢？这是因为**有些数据计算出的哈希值差异主要在高位，而HashMap里的哈希寻址是忽略大于容量以上的高位的，所以这种处理可以有效避免类似情况下计算的哈希碰撞**。

哈希冲突，可以这么理解，就是不同的“key”经过哈希计算之后得到的哈希值之间相等了，那么想把它们存入到HashMap的时候，两者的地址就冲突了。这种现象被称为**哈希冲突**。哈希表解决哈希冲突的方法：很多，比如开放地址法、再哈希函数法和链地址法。而经过考虑，HashMap使用链地址法解决哈希冲突。

这种方法才用了数组（哈希表） + 链表的数据结构，当发生哈希冲突时，就用一个链表结构存储相同Hash值的数据。

HashMap中另外两个非常重要的属性：加载因子(loadFactor)和边界值（threshold）。在初始化的时候，就会涉及这两个关键初始化参数。



LoadFactor这个属性默认用来设置Entry数组的内存空间大小，默认情况下它为0.75，这个值结合了很多实践，一般来说是最好效果，不要轻易去修改它。

putVal方法本身的逻辑非常集中，从初始化，扩容到树化，全都和它有关。

而resize方法更是身兼多职，研究一下它的源码设计：



根据源码，不考虑极端情况（容量理论最大极限由MAXIMUM\_CAPACITY指定，数值为1<<30，也就是2的30次方）。可以归纳为：

* 门限值等于（负载因子）\*（容量），如果构建HashMap的时候没有指定它们，那么就是根据相应的默认常量值。
* 门限通常是以倍数进行调整（newThr = oldThr << 1），前面提到，根据putVal中的逻辑，当元素个数超过门限大小的时候，调整Map的大小。
* 扩容后，需要将老的数组中的元素重新放置到新的数组，这是扩容的一个主要开销来源。

下面介绍**容量、负载因子和树化**。

前面我们考虑了HashMap从创建到放入键值对的相关逻辑。但是现在看来，我们为什么需要容量和负载因子呢？

答：因为容量和负载因子系数决定了可用的桶的数量，空桶太多会浪费空间，但如果使用得太满又会严重影响操作的性能。极端情况下，假设只有一个桶，那么他就退化成了链表，完全不能提供所谓常数时间存储的性能。

既然容量和负载因子很重要，那么使用过程中它们的值应当被如何选择？

如果可以预先知道HashMap要存储的键值对的数量，可以考虑预先设置合适容量的大小。具体数值可以根据扩容发生的条件进行简单预估，根据前面的分析，可以大致知道它符合的计算条件为：

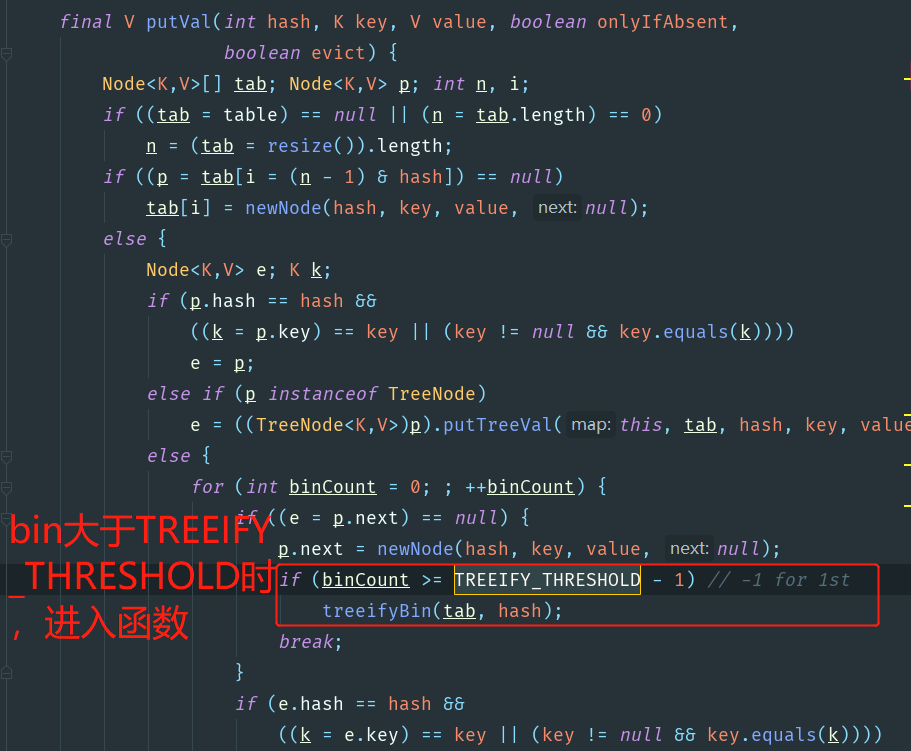
负载因子 \* 容量 > 元素数量

因此，预先设置的容量需要满足：大于“预估元素数量/负载因子”，同时它是2的幂数。

对于负载因子的赋值建议：

* 如果没有特别需求，不要轻易更改，因为JDK自身的默认负载因子已经很符合通用场景的需求了。
* 如果确实需要调整，也建议不要超过0.75这个值，因为会显著增加冲突，降低HashMap的性能。
* 如果负载因子太小了，那么按照上面的公式，元素容量也应该要调整，否则可能导致更加频繁地扩容，增加无谓的开销，本身访问的性能也会受影响。

改造树的逻辑主要在putVal和treeifyBin中：





可以从上图中看出，树化逻辑改造的逻辑很清晰，可以理解为，当bin的数量大于TREEIFY\_CAPACITY的时候：

* 如果容量小于MIN\_TREEIFY\_CAPACITY，只会进行简单的扩容。
* 如果容量大于MIN\_TREEIFY\_CAPACITY，则会进行树化改造。

那么，HashMap为什么要进行树化？（具体说是红黑树化）

可以说这本质是**安全问题**。因为在元素放置的过程中，如果一个对象哈希冲突，都被放置到同一个桶里，如果不做树化，会形成链表。而链表的查询性能很低，是线性的，会严重影响存取的性能。

而恶意代码可以利用这些数据大量与服务器端交互，导致服务器端CPU大量占用，这就构成了哈希碰撞拒绝服务攻击，国内一线互联网公司就曾经发生过这样的攻击事件。