HTTP 状态码

200 ok 成功返回状态，对应，GET,PUT,PATCH,DELETE.

201 created 成功创建。

304 not modified HTTP缓存有效。

400 bad request 请求格式错误。

401 unauthorized 未授权。

403 forbidden 鉴权成功，但是该用户没有权限。

404 not found 请求的资源不存在

405 method not allowed 该http方法不被允许。

410 gone 这个url对应的资源现在不可用。

415 unsupported media type 请求类型错误。

422 unprocessable entity 校验错误时用。

429 too many request 请求过多。

GET操作是安全的。所谓安全是指不管进行多少次操作，资源的状态都不会改变。比如我用GET浏览文章，不管浏览多少次，那篇文章还在那，没有变化。当然，你可能说每浏览一次文章，文章的浏览数就加一，这不也改变了资源的状态么？这并不矛盾，因为这个改变不是GET操作引起的，而是用户自己设定的服务端逻辑造成的。

PUT，DELETE操作是幂等的。所谓幂等是指不管进行多少次操作，结果都一样。比如我用PUT修改一篇文章，然后在做同样的操作，每次操作后的结果并没有不同，DELETE也是一样。顺便说一句，因为GET操作是安全的，所以它自然也是幂等的。

POST操作既不是安全的，也不是幂等的，比如常见的POST重复加载问题：当我们多次发出同样的POST请求后，其结果是创建出了若干的资源。

集合

**ArrayList**:线程不同步。数组形式，默认初始容量为10，扩容是当前 50%，它继承于 **AbstractList**，实现了 **List**, **RandomAccess**, **Cloneable**, **java.io.Serializable** 这些接口。插入删除元素的时间复杂度为**O（n），**求表长以及增加元素，取第 i 元素的时间复杂度为**O（1）**

**遍历方式：**  
  

**Vector**:线程同步，默认容量10，扩容是当前 100%，它的同步是通过iterator方法加synchronized实现的

**LinkedList**:一个实现了List接口和Deque接口的双端链表, 支持高效插入和删除操作，实现了Deque接口，具有队列的特性; LinkedList不是线程安全的，调用静态类Collections类中的synchronizedList方法可以保证线程安全

**Arraylist 与 LinkedList 异同**

1. 是否保证线程安全： ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

2. 底层数据结构： Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向循环链表数据结构；

3. 插入和删除是否受元素位置的影响： ① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行add(E e) 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element) ）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。

4. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index) 方法)。

5. 内存空间占用： ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

**HashMap**:主要用来存放键值对，它基于哈希表的Map接口实现，是常用的Java集合之一。

JDK1.8 之前 HashMap 由 数组+链表 组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）.JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间

遍历方式：



**HashMap的底层实现**

JDK1.8 之前 HashMap 底层是 数组和链表 结合在一起使用也就是 链表散列。HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 (n - 1) & hash 判断当前元素存放的位置（这里的 n 指的时数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

**HashMap 和 Hashtable 的区别**

线程是否安全： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；

对Null key 和Null value的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。

初始容量大小和每次扩充容量大小的不同 ： ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的tableSizeFor()方法保证，下面给出了源代码）。也就是说 HashMap 总是使用2的幂作为哈希表的大小,后面会介绍到为什么是2的幂次方。

底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制

**HashMap 的长度为什么是2的幂次方**

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648到2147483648，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ (n - 1) & hash ”。（n代表数组长度）。这也就解释了 HashMap 的长度为什么是2的幂次方。

这个算法应该如何设计呢？

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是2的 n 次方；）。” 并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

**HashMap 多线程操作导致死循环问题**

在多线程下，进行 put 操作会导致 HashMap 死循环，原因在于 HashMap 的扩容 resize()方法。由于扩容是新建一个数组，复制原数据到数组。由于数组下标挂有链表，所以需要复制链表，但是多线程操作有可能导致环形链表。复制链表过程如下:

以下模拟2个线程同时扩容。假设，当前 HashMap 的空间为2（临界值为1），hashcode 分别为 0 和 1，在散列地址 0 处有元素 A 和 B，这时候要添加元素 C，C 经过 hash 运算，得到散列地址为 1，这时候由于超过了临界值，空间不够，需要调用 resize 方法进行扩容，那么在多线程条件下，会出现条件竞争，模拟过程如下：

线程一：读取到当前的 HashMap 情况，在准备扩容时，线程二介入

线程二：读取 HashMap，进行扩容

线程一：继续执行

这个过程为，先将 A 复制到新的 hash 表中，然后接着复制 B 到链头（A 的前边：B.next=A），本来 B.next=null，到此也就结束了（跟线程二一样的过程），但是，由于线程二扩容的原因，将 B.next=A，所以，这里继续复制A，让 A.next=B，由此，环形链表出现：B.next=A; A.next=B

**ConcurrentHashMap 和 Hashtable 的区别**

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

底层数据结构： JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

实现线程安全的方式（重要）： ① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配16个Segment，比Hashtable效率提高16倍。） 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

Public > 类 、包、子类、其他包

Protected >类、包、子类

Default >类、包

Private >类