1. Java集合框架
   1. HashMap
      1. 简介

HashMap是基于哈希表实现的，每一个元素是一个key-value对，其内部通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

HashMap是非线程安全的，只是用于单线程环境下，多线程环境下可以采用concurrent并发包下的concurrentHashMap。

HashMap 实现了Serializable接口，因此它支持序列化，实现了Cloneable接口，能被克隆。

HashMap存数据的过程是：

HashMap内部维护了一个存储数据的Entry数组，HashMap采用链表解决冲突，每一个Entry本质上是一个单向链表。当准备添加一个key-value对时，首先通过hash(key)方法计算hash值，然后通过indexFor(hash,length)求该key-value对的存储位置，计算方法是先用hash&0x7FFFFFFF后，再对length取模，这就保证每一个key-value对都能存入HashMap中，当计算出的位置相同时，由于存入位置是一个链表，则把这个key-value对插入链表头。

HashMap中key和value都允许为null。key为null的键值对永远都放在以table[0]为头结点的链表中。

了解了数据的存储，那么数据的读取也就很容易就明白了。

图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

HashMap内存储数据的Entry数组默认是16，如果没有对Entry扩容机制的话，当存储的数据一多，Entry内部的链表会很长，这就失去了HashMap的存储意义了。所以HasnMap内部有自己的扩容机制。HashMap内部有：

变量size，它记录HashMap的底层数组中已用槽的数量；

变量threshold，它是HashMap的阈值，用于判断是否需要调整HashMap的容量（threshold = 容量\*加载因子）

变量DEFAULT\_LOAD\_FACTOR = 0.75f，默认加载因子为0.75

HashMap扩容的条件是：当size大于threshold时，对HashMap进行扩容 ，扩容是是新建了一个HashMap的底层数组，而后调用transfer方法，将就HashMap的全部元素添加到新的HashMap中（要重新计算元素在新的数组中的索引位置）。 很明显，扩容是一个相当耗时的操作，因为它需要重新计算这些元素在新的数组中的位置并进行复制处理。因此，我们在用HashMap的时，最好能提前预估下HashMap中元素的个数，这样有助于提高HashMap的性能。

HashMap共有四个构造方法。构造方法中提到了两个很重要的参数：初始容量和加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中槽的数量（即哈希数组的长度），初始容量是创建哈希表时的容量（从构造函数中可以看出，如果不指明，则默认为16），加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 resize 操作（即扩容）。

下面说下加载因子，如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），对空间造成严重浪费。如果我们在构造方法中不指定，则系统默认加载因子为0.75，这是一个比较理想的值，一般情况下我们是无需修改的。

另外，无论我们指定的容量为多少，构造方法都会将实际容量设为不小于指定容量的2的次方的一个数，且最大值不能超过2的30次方

* + 1. java8的性能改善

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。

当插入新元素时，对于红黑树的判断如下：

判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向下面；

遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

* 1. HashTable
     1. 简介

Hashtable同样是基于哈希表实现的，同样每个元素是一个key-value对，其内部也是通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

Hashtable也是JDK1.0引入的类，是线程安全的，能用于多线程环境中。

Hashtable同样实现了Serializable接口，它支持序列化，实现了Cloneable接口，能被克隆。

* + 1. HashMap与HashTable的区别

1、继承的父类不同

Hashtable继承自Dictionary类，而HashMap继承自AbstractMap类。但二者都实现了Map接口。

2、线程安全性不同

javadoc中关于hashmap的一段描述如下：此实现不是同步的。如果多个线程同时访问一个哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

3、是否提供contains方法

HashMap把Hashtable的contains方法去掉了，改成containsValue和containsKey，因为contains方法容易让人引起误解。

Hashtable则保留了contains，containsValue和containsKey三个方法，其中contains和containsValue功能相同。

4、key和value是否允许null值

其中key和value都是对象，并且不能包含重复key，但可以包含重复的value。

通过上面的ContainsKey方法和ContainsValue的源码我们可以很明显的看出：

Hashtable中，key和value都不允许出现null值。但是如果在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。

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。当get()方法返回null值时，可能是 HashMap中没有该键，也可能使该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键， 而应该用containsKey()方法来判断。

5、两个遍历方式的内部实现上不同

Hashtable、HashMap都使用了 Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式 。

6、hash值不同

哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

hashCode是jdk根据对象的地址或者字符串或者数字算出来的int类型的数值。

Hashtable计算hash值，直接用key的hashCode()，而HashMap重新计算了key的hash值，Hashtable在求hash值对应的位置索引时，用取模运算，而HashMap在求位置索引时，则用与运算，且这里一般先用hash&0x7FFFFFFF后，再对length取模，&0x7FFFFFFF的目的是为了将负的hash值转化为正值，因为hash值有可能为负数，而&0x7FFFFFFF后，只有符号外改变，而后面的位都不变。

7、内部实现使用的数组初始化和扩容方式不同

HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

Hashtable和HashMap它们两个内部实现方式的数组的初始大小和扩容的方式。HashTable中hash数组默认大小是11，增加的方式是 old\*2+1。