## JavaSE面试常见问题

## 01、TCP、UDP和HTTP的区别

**1、**TCP/IP是个协议组，可分为三个层次：网络层、传输层和应用层。

在网络层有IP协议、ICMP协议、ARP协议、RARP协议和BOOTP协议。

在传输层中有TCP协议与UDP协议。

在应用层有FTP、HTTP、TELNET、SMTP、DNS等协议。

因此，HTTP本身就是一个协议，是从Web服务器传输超文本到本地浏览器的传送协议。

TCP 是基于 TCP 协议实现的网络文本协议,属于传输层。

UDP 是和TCP 对等的，属于传输层，UDP 和 TCP 有重要的区别。

**2**、HTTP协议是建立在请求/响应模型上的。首先由客户建立一条与服务器的TCP链接，并发送一个请求到服务器，请求中包含请求方法、URI、协议版本以及相关的MIME样式的消息。服务器响应一个状态行，包含消息的协议版本、一个成功和失败码以及相关的MIME式样的消息。

HTTP/1.0为每一次HTTP的请求/响应建立一条新的TCP链接，因此一个包含HTML内容和图片的页面将需要建立多次的短期的TCP链接。一次TCP链接的建立将需要3次握手。

另外，为了获得适当的传输速度，则需要TCP花费额外的回路链接时间（RTT）。每一次链接的建立需要这种经常性的开销，而其并不带有实际有用的数据，只是保证链接的可靠性，因此HTTP/1.1提出了可持续链接的实现方法。HTTP/1.1将只建立一次TCP的链接而重复地使用它传输一系列的请求/响应消息，因此减少了链接建立的次数和经常性的链接开销。

这里有必要再讲一下**三次握手**的过程:

1.第一次握手：建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x;然后,客户端进入SYN\_SEND状态，等待服务器的确认; （SYN：同步序列编号）

2.第二次握手：服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1);同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y;服务器端将上述所有信息放到一个报文段(即SYN+ACK报文段)中，一并发送给客户端，此时服务器进入SYN\_RECV状态;（ACK:确认字符）

3.第三次握手：客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态（TCP连接成功），完成TCP三次握手。

当客户端和服务器通过三次握手建立了TCP连接以后，当数据传送完毕，肯定是要断开TCP连接的啊。那对于TCP的断开连接，这里就有了神秘的“**四次挥手**”。

1.第一次挥手：主机1(可以是客户端，也可以是服务器端)，设置Sequence Number和Acknowledgment Number，向主机2发送一个FIN报文段;此时，主机1进入FIN\_WAIT\_1状态;这表示主机1没有数据要发送给主机2了;

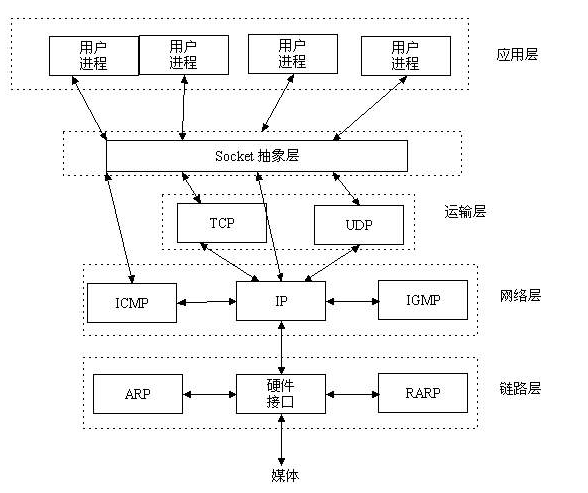
2.第二次挥手：主机2收到了主机1发送的FIN报文段，向主机1回一个ACK报文段，Acknowledgment Number为Sequence Number加1;主机1进入FIN\_WAIT\_2状态;主机2告诉主机1，我也没有数据要发送了，可以进行关闭连接了;

3.第三次挥手：主机2向主机1发送FIN报文段，请求关闭连接，同时主机2进入CLOSE\_WAIT状态;

4.第四次挥手：主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入TIME\_WAIT状态;主机2收到主机1的ACK报文段以后，就关闭连接;此时，主机1等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

**3、**结论：虽然HTTP本身是一个协议，但其最终还是基于TCP的。不过，目前，有人正在研究基于TCP+UDP混合的HTTP协议。

Socket是什么呢？  
       Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。



TCP（Transmission Control Protocol，传输控制协议）是基于连接的协议，也就是说，在正式收发数据前，必须和对方建立可靠的连接。一个TCP连接必须要经过三次“对话”才能建立起来，其中的过程非常复杂，我们这里只做简单、形象的介绍，你只要做到能够理解这个过程即可。我们来看看这三次对话的简单过程：主机A向主机B发出连接请求数据包：“我想给你发数据，可以吗？”，这是第一次对话；主机B向主机A发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包：“可以，你什么时候发？”，这是第二次对话；主机A再发出一个数据包确认主机B的要求同步：“我现在就发，你接着吧！”，这是第三次对话。三次“对话”的目的是使数据包的发送和接收同步，经过三次“对话”之后，主机A才向主机B正式发送数据。

UDP（User Data Protocol，用户数据报协议）是与TCP相对应的协议。它是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去！   
 UDP适用于一次只传送少量数据、对可靠性要求不高的应用环境。比如，我们经常使用“ping”命令来测试两台主机之间TCP/IP通信是否正常，其实“ping”命令的原理就是向对方主机发送UDP数据包，然后对方主机确认收到数据包，如果数据包是否到达的消息及时反馈回来，那么网络就是通的。例如，在默认状态下，一次“ping”操作发送4个数据包。大家可以看到，发送的数据包数量是4包，收到的也是4包（因为对方主机收到后会发回一个确认收到的数据包）。这充分说明了UDP协议是面向非连接的协议，没有建立连接的过程。正因为UDP协议没有连接的过程，所以它的通信效果高；但也正因为如此，它的可靠性不如TCP协议高。QQ就使用UDP发消息，因此有时会出现收不到消息的情况。

|  |  |  |
| --- | --- | --- |
| tcp协议和udp协议的差别 | | |
|  | TCP | UDP |
| 是否连接 | 面向连接 | 面向非连接 |
| 传输可靠性 | 可靠 | 不可靠 |
| 应用场合 | 传输大量数据 | 传输少量数据 |
| 速度 | 慢 | 快 |

## 02、java内存分配

**简单通俗的讲，一个完整的Java程序运行过程会涉及以下内存区域：**

l **寄存器：**JVM内部虚拟寄存器，存取速度非常快，程序不可控制。

l **栈：**保存局部变量的值，包括：

1.用来保存基本数据类型的值；

2.保存类的**实例**，即堆区**对象**的引用(指针)。

3.也可以用来保存加载方法时的帧 (在堆栈中为当前正在运行的函数分配的区域（或空间）)。

l **堆：**用来存放动态产生的数据，比如new出来的**对象**。注意创建出来的对象只包含属于各自的成员变量，并不包括成员方法。因为同一个类的对象拥有各自的成员变量，存储在各自的堆中，但是他们共享该类的方法，并不是每创建一个对象就把成员方法复制一次。

l **常量池：**JVM为每个已加载的类型维护一个常量池，常量池就是这个类型用到的常量的一个有序集合。包括直接常量(基本类型，String)和对其他类型、方法、字段的**符号引用(1)**。池中的数据和数组一样通过索引访问。由于常量池包含了一个类型所有的对其他类型、方法、字段的符号引用，所以常量池在Java的动态链接中起了核心作用。

java中基本类型的包装类的大部分都实现了常量池技术，这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另外Byte,Short,Integer,Long,Character这5种整型的包装类也只是在对应值小于等于127时才可使用对象池，也即对象不负责创建和管理大于127的这些类的对象。

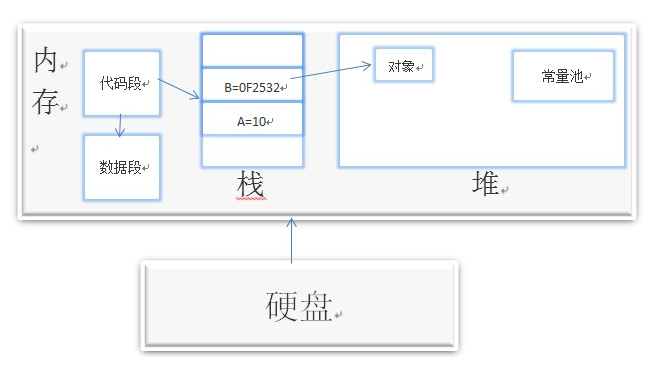
**关于常量池的说明：**

Java8之前，常量池是存放在堆中的，常量池就相当于是在永久代中，所以永久代存放在堆中。

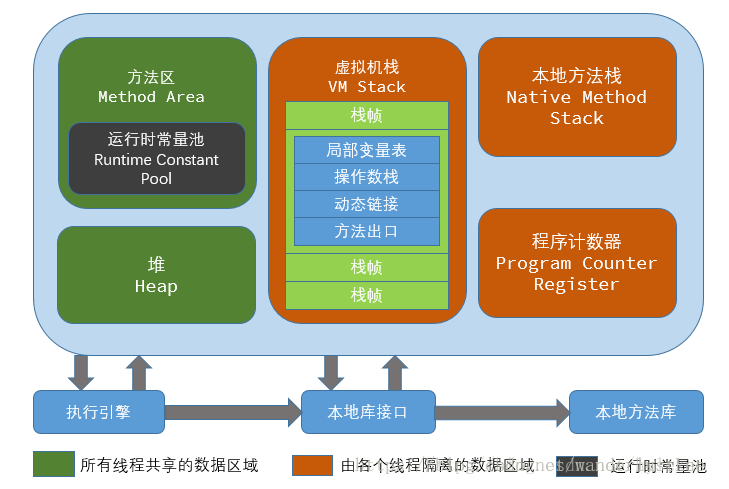
Java8之后，取消了整个永久代区域，取而代之的是元空间。常量池就不存放在堆中了，而是存放在方法区里面，与堆栈是并列关系。永久代也就不存放在堆中了

l **代码段：**用来存放从硬盘上读取的源程序代码。

l **数据段：**用来存放static定义的静态成员。



java 8之前



java8之后

## 03、数据库事务的四大特性

**⑴ 原子性（Atomicity）**

　　原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

**⑵ 一致性（Consistency）**

　　一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

**⑶ 隔离性（Isolation）**

　　隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

　　即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

　　关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

**⑷ 持久性（Durability）**

　　持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

　　例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

以上介绍完事务的四大特性(简称ACID)，现在重点来说明下事务的隔离性，当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题：

**1，脏读**

　　脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

　　当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元，对应SQL命令如下

update account set money=money+**100** where name=’B’; (此时A通知B)

update account set money=money - **100** where name=’A’;

　　当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

**2，不可重复读**

　　不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

　　例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

　　不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

　　在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据A和B依次查询就可能不同，A和B就可能打起来了……

**3，虚读(幻读)**

　　幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

　　幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

## 04、事务的四种隔离级别

现在来看看MySQL数据库为我们提供的四种**隔离级别**：

　　① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

　　② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

　　③ Read committed (读已提交)：可避免脏读的发生。

　　④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

## 05、this 和super的区别

属性的区别：this访问本类中的属性，如果本类没有此属性则从父类中继续查找。super访问父类中的属性。

方法的区别：this访问本类中的方法，如果本类没有此方法则从父类中继续查找。super访问父类中的方法。

构造的区别：this调用本类构造，必须放在构造方法的首行。super调用父类构造，必须放在子类构造方法首行。

其他区别：this表示当前对象。super不能表示当前对象

**1、this.变量和super.变量**

this.变量 调用的当前对象的变量；

super.变量 直接调用的是父类中的变量。

**2、this(参数)和super(参数)方法**

this(参数) 调用（转发）的是当前类中的构造器；

super(参数) 用于确认要使用父类中的哪一个构造器。

**注意点**：

1）在对拥有父类的子类进行初始化时，父类的构造方法也会执行，且优先于子类的构造函数执行；因为每一个子类的构造函数中的第一行都有一条默认的隐式语句super();

2）this() 和super()都只能写在构造函数的第一行；

3）this() 和super() 不能存在于同一个构造函数中。第一，this()和super()都必须写在构造函数的第一行；第二，this()语句调用的是当前类的另一个构造函数而这个另一个构造函数中必然有一个父类的构造器，再使用super()又调用一次父类的构造器，就相当于调用了两次父类的构造器，编译器不会通过；

4）this和super不能用于static修饰的变量，方法，代码块；因为this和super都是指的是对象（实例）。

js中this的指向问题（javascript）、

## 06、抽象类和接口的区别

抽象类里面可以有非抽象的方法，接口里只能有抽象方法。

　　抽象类中的抽象方法声明时不能有大括号，而接口中的所有方法都没有大括号。

抽象类（abstract class）：

　　a.抽象类是仅供派生的类，无法实例化一个抽象类，只能实例化从它派生的类。

　　b.抽象类的主要特征在于它包含抽象成员（abstract member），抽象成员是不具有实现的一个方法或属性，其作用是强制所有派生类提供实现。

　　c.由于抽象成员应当是要被重写（override）的，因此这类成员会自动成为virtual成员，而且不能这样显示地声明。

　　d.抽象成员不能是private的，否则派生类看不见他们。

接口（interface）：

　　a.接口是抽象类的变体。

　　b.接口的一个关键特征是它既不包含实现，也不包含数据。

　　c.字段（也就是数据）不能出现在一个接口中，如果一个接口要求派生类包含特定的数据，那么它会使用属性而不是字段。由于属性不会包含任何实现作为接口声明的一部分，所以他不会引用一个支持字段。

　　d.接口的宗旨是定义由多个类共同遵守的一个契约，所以接口中所有成员都必须为public类型。

接口的特点

（1）接口中的方法是抽象的，不能实例化对象。

（2）implements可以实现接口，可以实现多个接口，被实现的接口之间需要用逗号隔开。

（3）当一个类实现接口时，如果这个类时抽象类，则实现接口中的部分方法即可，否则需要实现全部全部方法

## 07、JavaSE和JavaEE的区别是什么？

J2SE全称是java 2 Standard Edition(标准版)，J2SE 包含那些构成Java语言核心的类。

比如：数据库连接、接口定义、输入/输出、网络编程

J2EE全称是java 2 enterprise edition（企业版），现在最新称谓是java EE 5，两个是同样的东东，Enterprise Edition(企业版) J2EE 包含J2SE 中的类，并且还包含用于开发企业级应用的类。比如：EJB、servlet、JSP、XML、事务控制

J2SE包含于J2EE中，简单点说，J2EE是J2SE在企业应用需求上的扩张，语法是以J2SE为基础，通过添加新的API,框架技术来实现大规模，高复杂度的企业级项目的开发。

笼统的讲，可以这样理解：J2SE是基础；压缩一点，再增加一些CLDC等方面的特性就是J2ME；扩充一点，再增加一些EJB等企业应用方面的特性就是J2EE。

补充一点J2EE更恰当的说，应该是JAVA2企业开发的技术规范，不仅仅是比标准版多了一些类。J2EE又包括许多组件，如Jsp, Servlet, JavaBean, EJB, JDBC, JavaMail等。

J2SE商业版本，标准版本 (Java2 Standard Edition) 定位在客户端，主要用于桌面应用软件的编程。J2SE 包含那些构成Java语言核心的类。

比如：数据库连接、接口定义、输入/输出、网络编程 　　J2SE 是J2EE的基础，他大量的JDK代码库是每个要学习J2EE的编程人员必须掌握的。

## 08、java多线程中run和start的区别是什么？

Java的线程是通过java.lang.Thread类来实现的。VM启动时会有一个由主方法所定义的线程。可以通过创建Thread的实例来创建新的线程。每个线程都是通过某个特定Thread对象所对应的方法run（）来完成其操作的，方法run()称为线程体。通过调用Thread类的start()方法来启动一个线程。

实现并启动线程有两种方法：

1、写一个类继承自Thread类，重写run方法。用start方法启动线程

2、写一个类实现Runnable接口，实现run方法。用new Thread(Runnable target).start()方法来启动

调用start（）后，线程会被放到等待队列，等待CPU调度，并不一定要马上开始执行，只是将这个线程置于可动行状态。然后通过JVM，线程Thread会调用run（）方法，执行本线程的线程体。先调用start后调用run，这么麻烦，为了不直接调用run？就是为了实现多线程的优点，没这个start不行。

1.start（）方法来启动线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码；通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

2.run（）方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到写线程的目的。

记住：多线程就是分时利用CPU，宏观上让所有线程一起执行 ，也叫并发。

## 09、[ArrayList的实现原理](http://zhangshixi.iteye.com/blog/674856)

ArrayList概述：

   ArrayList是List接口的可变数组的实现。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。  
   每个ArrayList实例都有一个容量，该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向ArrayList中不断添加元素，其容量也自动增长。自动增长会带来数据向新数组的重新拷贝，因此，如果可预知数据量的多少，可在构造ArrayList时指定其容量。在添加大量元素前，应用程序也可以使用ensureCapacity操作来增加ArrayList实例的容量，这可以减少递增式再分配的数量。   
   注意，此实现不是同步的。如果多个线程同时访问一个ArrayList实例，而其中至少一个线程从结构上修改了列表，那么它必须保持外部同步。

2. ArrayList的实现：

   对于ArrayList而言，它实现List接口、底层使用数组保存所有元素。其操作基本上是对数组的操作。下面我们来分析ArrayList的源代码：

   1) 底层使用数组实现：

Java代码

1. private transient Object[] elementData;

    2) 构造方法：   
   ArrayList提供了三种方式的构造器，可以构造一个默认初始容量为10的空列表、构造一个指定初始容量的空列表以及构造一个包含指定collection的元素的列表，这些元素按照该collection的迭代器返回它们的顺序排列的。

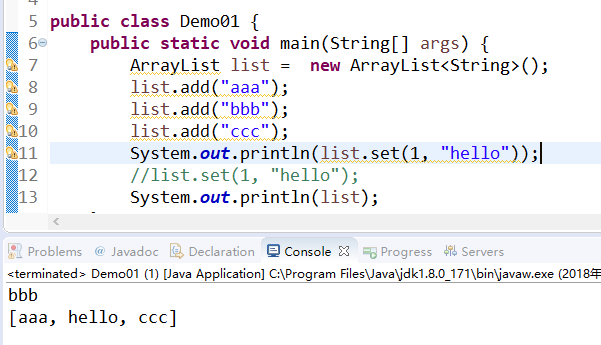
Java代码

1. public ArrayList() {
2. this(10);
3. }
5. public ArrayList(int initialCapacity) {
6. super();
7. if (initialCapacity < 0)
8. throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
9. this.elementData = new Object[initialCapacity];
10. }
12. public ArrayList(Collection<? extends E> c) {
13. elementData = c.toArray();
14. size = elementData.length;
15. // c.toArray might (incorrectly) not return Object[] (see 6260652)
16. if (elementData.getClass() != Object[].class)
17. elementData = Arrays.copyOf(elementData, size, Object[].class);
18. }

    3) 存储：   
   ArrayList提供了set(int index, E element)、add(E e)、add(int index, E element)、addAll(Collection<? extends E> c)、addAll(int index, Collection<? extends E> c)这些添加元素的方法。下面我们一一讲解：

Java代码

1. // 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。
2. public E set(int index, E element) {
3. RangeCheck(index);
5. E oldValue = (E) elementData[index];
6. elementData[index] = element;
7. return oldValue;
8. }



Java代码

1. // 将指定的元素添加到此列表的尾部。
2. public boolean add(E e) {
3. ensureCapacity(size + 1);
4. elementData[size++] = e;
5. return true;
6. }

Java代码

1. // 将指定的元素插入此列表中的指定位置。
2. // 如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（将其索引加1）。
3. public void add(int index, E element) {
4. if (index > size || index < 0)
5. throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);
6. // 如果数组长度不足，将进行扩容。
7. ensureCapacity(size+1);  // Increments modCount!!
8. // 将 elementData中从Index位置开始、长度为size-index的元素，
9. // 拷贝到从下标为index+1位置开始的新的elementData数组中。
10. // 即将当前位于该位置的元素以及所有后续元素右移一个位置。
11. System. (elementData, index, elementData, index + 1, size - index);
12. elementData[index] = element;
13. size++;
14. }

Java代码

1. // 按照指定collection的迭代器所返回的元素顺序，将该collection中的所有元素添加到此列表的尾部。
2. public boolean addAll(Collection<? extends E> c) {
3. Object[] a = c.toArray();
4. int numNew = a.length;
5. ensureCapacity(size + numNew);  // Increments modCount
6. System.arraycopy(a, 0, elementData, size, numNew);
7. size += numNew;
8. return numNew != 0;
9. }

Java代码

1. // 从指定的位置开始，将指定collection中的所有元素插入到此列表中。
2. public boolean addAll(int index, Collection<? extends E> c) {
3. if (index > size || index < 0)
4. throw new IndexOutOfBoundsException(
5. "Index: " + index + ", Size: " + size);
7. Object[] a = c.toArray();
8. int numNew = a.length;
9. ensureCapacity(size + numNew);  // Increments modCount
11. int numMoved = size - index;
12. if (numMoved > 0)
13. System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
15. System.arraycopy(a, 0, elementData, index, numNew);
16. size += numNew;
17. return numNew != 0;
18. }

    4) 读取：

Java代码

1. // 返回此列表中指定位置上的元素。
2. public E get(int index) {
3. RangeCheck(index);
5. return (E) elementData[index];
6. }

    5) 删除：   
   ArrayList提供了根据下标或者指定对象两种方式的删除功能。如下：

Java代码

1. // 移除此列表中指定位置上的元素。
2. public E remove(int index) {
3. RangeCheck(index);
5. modCount++;
6. E oldValue = (E) elementData[index];
8. int numMoved = size - index - 1;
9. if (numMoved > 0)
10. System.arraycopy(elementData, index+1, elementData, index, numMoved);
11. elementData[--size] = null; // Let gc do its work
13. return oldValue;
14. }

Java代码

1. // 移除此列表中首次出现的指定元素（如果存在）。这是应为ArrayList中允许存放重复的元素。
2. public boolean remove(Object o) {
3. // 由于ArrayList中允许存放null，因此下面通过两种情况来分别处理。
4. if (o == null) {
5. for (int index = 0; index < size; index++)
6. if (elementData[index] == null) {
7. // 类似remove(int index)，移除列表中指定位置上的元素。
8. fastRemove(index);
9. return true;
10. }
11. } else {
12. for (int index = 0; index < size; index++)
13. if (o.equals(elementData[index])) {
14. fastRemove(index);
15. return true;
16. }
17. }
18. return false;
19. }

    注意：从数组中移除元素的操作，也会导致被移除的元素以后的所有元素的向左移动一个位置。  
   6) 调整数组容量：   
   从上面介绍的向ArrayList中存储元素的代码中，我们看到，每当向数组中添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容通过一个公开的方法ensureCapacity(int minCapacity)来实现。在实际添加大量元素前，我也可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。

Java代码

1. public void ensureCapacity(int minCapacity) {
2. modCount++;
3. int oldCapacity = elementData.length;
4. if (minCapacity > oldCapacity) {
5. Object oldData[] = elementData;
6. int newCapacity = (oldCapacity \* 3)/2 + 1;
7. if (newCapacity < minCapacity)
8. newCapacity = minCapacity;
9. // minCapacity is usually close to size, so this is a win:
10. elementData = Arrays.copyOf(elementData, newCapacity);
11. }
12. }

   从上述代码中可以看出，数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的1.5倍。这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。  
   ArrayList还给我们提供了将底层数组的容量调整为当前列表保存的实际元素的大小的功能。它可以通过trimToSize方法来实现。代码如下：

Java代码

1. public void trimToSize() {
2. modCount++;
3. int oldCapacity = elementData.length;
4. if (size < oldCapacity) {
5. elementData = Arrays.copyOf(elementData, size);
6. }
7. }

   7) Fail-Fast机制：   
ArrayList也采用了快速失败的机制，通过记录modCount参数来实现。在面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。具体介绍请参考我之前的文章 [HashMap的实现原理](http://zhangshixi.iteye.com/blog/672697" \t "_blank" \o "深入Java集合学习系列：HashMap的实现原理) 中的Fail-Fast机制。  
   8) 关于其他 的一些方法的实现都很简单易懂，读者可参照API文档和源代码，一看便知，这里就不再多说。

## 10、[HashMap的实现原理](http://zhangshixi.iteye.com/blog/672697)

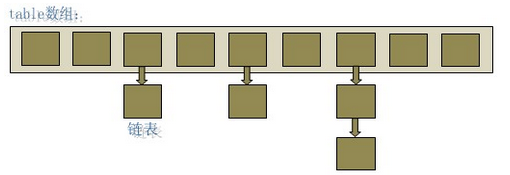
1. HashMap概述：

   HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。（除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）

Hashtable不允许值为null。

2. HashMap的数据结构：

   在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。



   从上图中可以看出，HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个数组。

   源码如下：

Java代码

1. /\*\*
2. \* The table, resized as necessary. Length MUST Always be a power of two.
3. \*/
4. transient Entry[] table;
6. static class Entry<K,V> implements Map.Entry<K,V> {
7. final K key;
8. V value;
9. Entry<K,V> next;
10. final int hash;
11. ……
12. }

   可以看出，Entry就是数组中的元素，每个 Map.Entry 其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。

3.    HashMap的存取实现：

   1) 存储：

Java代码

1. public V put(K key, V value) {
2. // HashMap允许存放null键和null值。
3. // 当key为null时，调用putForNullKey方法，将value放置在数组第一个位置。
4. if (key == null)
5. return putForNullKey(value);
6. // 根据key的keyCode重新计算hash值。
7. int hash = hash(key.hashCode());
8. // 搜索指定hash值在对应table中的索引。
9. int i = indexFor(hash, table.length);
10. // 如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素。
11. for (Entry<K,V> e = table[i]; e != null; e = e.next) {
12. Object k;
13. if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
14. V oldValue = e.value;
15. e.value = value;
16. e.recordAccess(this);
17. return oldValue;
18. }
19. }
20. // 如果i索引处的Entry为null，表明此处还没有Entry。
21. modCount++;
22. // 将key、value添加到i索引处。
23. addEntry(hash, key, value, i);
24. return null;
25. }

   从上面的源代码中可以看出：当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

   addEntry(hash, key, value, i)方法根据计算出的hash值，将key-value对放在数组table的i索引处。addEntry 是 HashMap 提供的一个包访问权限的方法，代码如下：

Java代码

1. void addEntry(int hash, K key, V value, int bucketIndex) {
2. // 获取指定 bucketIndex 索引处的 Entry
3. Entry<K,V> e = table[bucketIndex];
4. // 将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
5. table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
6. // 如果 Map 中的 key-value 对的数量超过了极限
7. if (size++ >= threshold)
8. // 把 table 对象的长度扩充到原来的2倍。
9. resize(2 \* table.length);
10. }

   当系统决定存储HashMap中的key-value对时，完全没有考虑Entry中的value，仅仅只是根据key来计算并决定每个Entry的存储位置。我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

   hash(int h)方法根据key的hashCode重新计算一次散列。此算法加入了高位计算，防止低位不变，高位变化时，造成的hash冲突。

Java代码

1. static int hash(int h) {
2. h ^= (h >>> 20) ^ (h >>> 12);
3. return h ^ (h >>> 7) ^ (h >>> 4);
4. }

   我们可以看到在HashMap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。如何计算这个位置就是hash算法。前面说过HashMap的数据结构是数组和链表的结合，所以我们当然希望这个HashMap里面的 元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表，这样就大大优化了查询的效率。

   对于任意给定的对象，只要它的 hashCode() 返回值相同，那么程序调用 hash(int h) 方法所计算得到的 hash 码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，“模”运算的消耗还是比较大的，在HashMap中是这样做的：调用 indexFor(int h, int length) 方法来计算该对象应该保存在 table 数组的哪个索引处。indexFor(int h, int length) 方法的代码如下：

Java代码

1. static int indexFor(int h, int length) {
2. return h & (length-1);
3. }

   这个方法非常巧妙，它通过 h & (table.length -1) 来得到该对象的保存位，而HashMap底层数组的长度总是 2 的 n 次方，这是HashMap在速度上的优化。在 HashMap 构造器中有如下代码：

Java代码

1. int capacity = 1;
2. while (capacity < initialCapacity)
3. capacity <<= 1;

   这段代码保证初始化时HashMap的容量总是2的n次方，即底层数组的长度总是为2的n次方。

当length总是 2 的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。

   这看上去很简单，其实比较有玄机的，我们举个例子来说明：

   假设数组长度分别为15和16，优化后的hash码分别为8和9，那么&运算后的结果如下：

       h & (table.length-1)                     hash                             table.length-1

       8 & (15-1)：                                 0100                   &              1110                   =                0100

       9 & (15-1)：                                 0101                   &              1110                   =                0100

       -----------------------------------------------------------------------------------------------------------------------

       8 & (16-1)：                                 0100                   &              1111                   =                0100

       9 & (16-1)：                                 0101                   &              1111                   =                0101

   从上面的例子中可以看出：当它们和15-1（1110）“与”的时候，产生了相同的结果，也就是说它们会定位到数组中的同一个位置上去，这就产生了碰撞，8和9会被放到数组中的同一个位置上形成链表，那么查询的时候就需要遍历这个链 表，得到8或者9，这样就降低了查询的效率。同时，我们也可以发现，当数组长度为15的时候，hash值会与15-1（1110）进行“与”，那么 最后一位永远是0，而0001，0011，0101，1001，1011，0111，1101这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！而当数组长度为16时，即为2的n次方时，2n-1得到的二进制数的每个位上的值都为1，这使得在低位上&时，得到的和原hash的低位相同，加之hash(int h)方法对key的hashCode的进一步优化，加入了高位计算，就使得只有相同的hash值的两个值才会被放到数组中的同一个位置上形成链表。

   所以说，当数组长度为2的n次幂的时候，不同的key算得得index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

   根据上面 put 方法的源代码可以看出，当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部——具体说明继续看 addEntry() 方法的说明。

   2) 读取：

Java代码

1. public V get(Object key) {
2. if (key == null)
3. return getForNullKey();
4. int hash = hash(key.hashCode());
5. for (Entry<K,V> e = table[indexFor(hash, table.length)];
6. e != null;
7. e = e.next) {
8. Object k;
9. if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
10. return e.value;
11. }
12. return null;
13. }

   有了上面存储时的hash算法作为基础，理解起来这段代码就很容易了。从上面的源代码中可以看出：从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

   3) 归纳起来简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

4.    HashMap的resize（rehash）：

   当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，这是一个常用的操作，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

   那么HashMap什么时候进行扩容呢？当HashMap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

5.    HashMap的性能参数：

   HashMap 包含如下几个构造器：

   HashMap()：构建一个初始容量为 16，负载因子为 0.75 的 HashMap。

   HashMap(int initialCapacity)：构建一个初始容量为 initialCapacity，负载因子为 0.75 的 HashMap。

   HashMap(int initialCapacity, float loadFactor)：以指定初始容量、指定的负载因子创建一个 HashMap。

   HashMap的基础构造器HashMap(int initialCapacity, float loadFactor)带有两个参数，它们是初始容量initialCapacity和加载因子loadFactor。

   initialCapacity：HashMap的最大容量，即为底层数组的长度。

   loadFactor：负载因子loadFactor定义为：散列表的实际元素数目(n)/ 散列表的容量(m)。

   负载因子衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。

   HashMap的实现中，通过threshold字段来判断HashMap的最大容量：

Java代码

1. threshold = (int)(capacity \* loadFactor);

   结合负载因子的定义公式可知，threshold就是在此loadFactor和capacity对应下允许的最大元素数目，超过这个数目就重新resize，以降低实际的负载因子。默认的的负载因子0.75是对空间和时间效率的一个平衡选择。当容量超出此最大容量时， resize后的HashMap容量是容量的两倍：

Java代码

1. if (size++ >= threshold)
2. resize(2 \* table.length);

6.    Fail-Fast机制：

   我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

   这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

Java代码

1. HashIterator() {
2. expectedModCount = modCount;
3. if (size > 0) { // advance to first entry
4. Entry[] t = table;
5. while (index < t.length && (next = t[index++]) == null)
6. ;
7. }
8. }

   在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map：

   注意到modCount声明为volatile，保证线程之间修改的可见性。

Java代码

1. final Entry<K,V> nextEntry() {
2. if (modCount != expectedModCount)
3. throw new ConcurrentModificationException();

   在HashMap的API中指出：

   由所有HashMap类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

   注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

## 11、[HashSet的实现原理](http://zhangshixi.iteye.com/blog/673143)

1.    HashSet概述：

   HashSet实现Set接口，由哈希表（实际上是一个HashMap实例）支持。它不保证set 的迭代顺序；特别是它不保证该顺序恒久不变。此类允许使用null元素。

2.    HashSet的实现：

   对于HashSet而言，它是基于HashMap实现的，HashSet底层使用HashMap来保存所有元素，因此HashSet 的实现比较简单，相关HashSet的操作，基本上都是直接调用底层HashMap的相关方法来完成， HashSet的源代码如下：

Java代码

1. public class HashSet<E>
2. extends AbstractSet<E>
3. implements Set<E>, Cloneable, java.io.Serializable
4. {
5. static final long serialVersionUID = -5024744406713321676L;
7. // 底层使用HashMap来保存HashSet中所有元素。
8. private transient HashMap<E,Object> map;
10. // 定义一个虚拟的Object对象作为HashMap的value，将此对象定义为static final。
11. private static final Object PRESENT = new Object();
13. /\*\*
14. \* 默认的无参构造器，构造一个空的HashSet。
15. \*
16. \* 实际底层会初始化一个空的HashMap，并使用默认初始容量为16和加载因子0.75。
17. \*/
18. public HashSet() {
19. map = new HashMap<E,Object>();
20. }
22. /\*\*
23. \* 构造一个包含指定collection中的元素的新set。
24. \*
25. \* 实际底层使用默认的加载因子0.75和足以包含指定
26. \* collection中所有元素的初始容量来创建一个HashMap。
27. \* @param c 其中的元素将存放在此set中的collection。
28. \*/
29. public HashSet(Collection<? extends E> c) {
30. map = new HashMap<E,Object>(Math.max((int) (c.size()/.75f) + 1, 16));
31. addAll(c);
32. }
34. /\*\*
35. \* 以指定的initialCapacity和loadFactor构造一个空的HashSet。
36. \*
37. \* 实际底层以相应的参数构造一个空的HashMap。
38. \* @param initialCapacity 初始容量。
39. \* @param loadFactor 加载因子。
40. \*/
41. public HashSet(int initialCapacity, float loadFactor) {
42. map = new HashMap<E,Object>(initialCapacity, loadFactor);
43. }
45. /\*\*
46. \* 以指定的initialCapacity构造一个空的HashSet。
47. \*
48. \* 实际底层以相应的参数及加载因子loadFactor为0.75构造一个空的HashMap。
49. \* @param initialCapacity 初始容量。
50. \*/
51. public HashSet(int initialCapacity) {
52. map = new HashMap<E,Object>(initialCapacity);
53. }
55. /\*\*
56. \* 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。
57. \* 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支持。
58. \*
59. \* 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。
60. \* @param initialCapacity 初始容量。
61. \* @param loadFactor 加载因子。
62. \* @param dummy 标记。
63. \*/
64. HashSet(int initialCapacity, float loadFactor, boolean dummy) {
65. map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);
66. }
68. /\*\*
69. \* 返回对此set中元素进行迭代的迭代器。返回元素的顺序并不是特定的。
70. \*
71. \* 底层实际调用底层HashMap的keySet来返回所有的key。
72. \* 可见HashSet中的元素，只是存放在了底层HashMap的key上，
73. \* value使用一个static final的Object对象标识。
74. \* @return 对此set中元素进行迭代的Iterator。
75. \*/
76. public Iterator<E> iterator() {
77. return map.keySet().iterator();
78. }
80. /\*\*
81. \* 返回此set中的元素的数量（set的容量）。
82. \*
83. \* 底层实际调用HashMap的size()方法返回Entry的数量，就得到该Set中元素的个数。
84. \* @return 此set中的元素的数量（set的容量）。
85. \*/
86. public int size() {
87. return map.size();
88. }
90. /\*\*
91. \* 如果此set不包含任何元素，则返回true。
92. \*
93. \* 底层实际调用HashMap的isEmpty()判断该HashSet是否为空。
94. \* @return 如果此set不包含任何元素，则返回true。
95. \*/
96. public boolean isEmpty() {
97. return map.isEmpty();
98. }
100. /\*\*
101. \* 如果此set包含指定元素，则返回true。
102. \* 更确切地讲，当且仅当此set包含一个满足(o==null ? e==null : o.equals(e))
103. \* 的e元素时，返回true。
104. \*
105. \* 底层实际调用HashMap的containsKey判断是否包含指定key。
106. \* @param o 在此set中的存在已得到测试的元素。
107. \* @return 如果此set包含指定元素，则返回true。
108. \*/
109. public boolean contains(Object o) {
110. return map.containsKey(o);
111. }
113. /\*\*
114. \* 如果此set中尚未包含指定元素，则添加指定元素。
115. \* 更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))
116. \* 的元素e2，则向此set 添加指定的元素e。
117. \* 如果此set已包含该元素，则该调用不更改set并返回false。
118. \*
119. \* 底层实际将将该元素作为key放入HashMap。
120. \* 由于HashMap的put()方法添加key-value对时，当新放入HashMap的Entry中key
121. \* 与集合中原有Entry的key相同（hashCode()返回值相等，通过equals比较也返回true），
122. \* 新添加的Entry的value会将覆盖原来Entry的value，但key不会有任何改变，
123. \* 因此如果向HashSet中添加一个已经存在的元素时，新添加的集合元素将不会被放入HashMap中，
124. \* 原来的元素也不会有任何改变，这也就满足了Set中元素不重复的特性。
125. \* @param e 将添加到此set中的元素。
126. \* @return 如果此set尚未包含指定元素，则返回true。
127. \*/
128. public boolean add(E e) {
129. return map.put(e, PRESENT)==null;
130. }
132. /\*\*
133. \* 如果指定元素存在于此set中，则将其移除。
134. \* 更确切地讲，如果此set包含一个满足(o==null ? e==null : o.equals(e))的元素e，
135. \* 则将其移除。如果此set已包含该元素，则返回true
136. \* （或者：如果此set因调用而发生更改，则返回true）。（一旦调用返回，则此set不再包含该元素）。
137. \*
138. \* 底层实际调用HashMap的remove方法删除指定Entry。
139. \* @param o 如果存在于此set中则需要将其移除的对象。
140. \* @return 如果set包含指定元素，则返回true。
141. \*/
142. public boolean remove(Object o) {
143. return map.remove(o)==PRESENT;
144. }
146. /\*\*
147. \* 从此set中移除所有元素。此调用返回后，该set将为空。
148. \*
149. \* 底层实际调用HashMap的clear方法清空Entry中所有元素。
150. \*/
151. public void clear() {
152. map.clear();
153. }
155. /\*\*
156. \* 返回此HashSet实例的浅表副本：并没有复制这些元素本身。
157. \*
158. \* 底层实际调用HashMap的clone()方法，获取HashMap的浅表副本，并设置到HashSet中。
159. \*/
160. public Object clone() {
161. try {
162. HashSet<E> newSet = (HashSet<E>) super.clone();
163. newSet.map = (HashMap<E, Object>) map.clone();
164. return newSet;
165. } catch (CloneNotSupportedException e) {
166. throw new InternalError();
167. }
168. }
169. }



HashSet：

　　HashSet实现了Set接口，它不允许集合中出现重复元素。当我们提到HashSet时，第一件事就是在将对象存储在

HashSet之前，要确保重写hashCode（）方法和equals（）方法，这样才能比较对象的值是否相等，确保集合中没有

储存相同的对象。如果不重写上述两个方法，那么将使用下面方法默认实现：

　public boolean add(Object obj)方法用在Set添加元素时，如果元素值重复时返回 "false"，如果添加成功则返回"true"

HashMap：

　　HashMap实现了Map接口，Map接口对键值对进行映射。Map中不允许出现重复的键（Key）。Map接口有两个基本的实现

TreeMap和HashMap。TreeMap保存了对象的排列次序，而HashMap不能。HashMap可以有空的键值对（Key（null）-Value（null））

HashMap是非线程安全的（非Synchronize），要想实现线程安全，那么需要调用collections类的静态方法synchronizeMap（）实现。

public Object put(Object Key,Object value)方法用来将元素添加到map中。

## 12、[LinkedHashMap的实现原理](http://zhangshixi.iteye.com/blog/673789)

[Java](http://www.iteye.com/blogs/tag/Java)[算法](http://www.iteye.com/blogs/tag/%E7%AE%97%E6%B3%95)

1. LinkedHashMap概述：

   LinkedHashMap是Map接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。  
   LinkedHashMap实现与HashMap的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。  
   注意，此实现不是同步的。如果多个线程同时访问链接的哈希映射，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。

2. LinkedHashMap的实现：

   对于LinkedHashMap而言，它继承与HashMap、底层使用哈希表与双向链表来保存所有元素。其基本操作与父类HashMap相似，它通过重写父类相关的方法，来实现自己的链接列表特性。下面我们来分析LinkedHashMap的源代码：

   1) Entry元素：

   LinkedHashMap采用的hash算法和HashMap相同，但是它重新定义了数组中保存的元素Entry，该Entry除了保存当前对象的引用外，还保存了其上一个元素before和下一个元素after的引用，从而在哈希表的基础上又构成了双向链接列表。看源代码：

Java代码

1. /\*\*
2. \* 双向链表的表头元素。
3. \*/
4. private transient Entry<K,V> header;
6. /\*\*
7. \* LinkedHashMap的Entry元素。
8. \* 继承HashMap的Entry元素，又保存了其上一个元素before和下一个元素after的引用。
9. \*/
10. private static class Entry<K,V> extends HashMap.Entry<K,V> {
11. Entry<K,V> before, after;
12. ……
13. }

    2) 初始化：

   通过源代码可以看出，在LinkedHashMap的构造方法中，实际调用了父类HashMap的相关构造方法来构造一个底层存放的table数组。如：

Java代码

1. public LinkedHashMap(int initialCapacity, float loadFactor) {
2. super(initialCapacity, loadFactor);
3. accessOrder = false;
4. }

    HashMap中的相关构造方法：

Java代码

1. public HashMap(int initialCapacity, float loadFactor) {
2. if (initialCapacity < 0)
3. throw new IllegalArgumentException("Illegal initial capacity: " +
4. initialCapacity);
5. if (initialCapacity > MAXIMUM\_CAPACITY)
6. initialCapacity = MAXIMUM\_CAPACITY;
7. if (loadFactor <= 0 || Float.isNaN(loadFactor))
8. throw new IllegalArgumentException("Illegal load factor: " +
9. loadFactor);
11. // Find a power of 2 >= initialCapacity
12. int capacity = 1;
13. while (capacity < initialCapacity)
14. capacity <<= 1;
16. this.loadFactor = loadFactor;
17. threshold = (int)(capacity \* loadFactor);
18. table = new Entry[capacity];
19. init();
20. }

    我们已经知道LinkedHashMap的Entry元素继承HashMap的Entry，提供了双向链表的功能。在上述HashMap的构造器  
中，最后会调用init()方法，进行相关的初始化，这个方法在HashMap的实现中并无意义，只是提供给子类实现相关的初始化调用。  
   LinkedHashMap重写了init()方法，在调用父类的构造方法完成构造后，进一步实现了对其元素Entry的初始化操作。

Java代码

1. void init() {
2. header = new Entry<K,V>(-1, null, null, null);
3. header.before = header.after = header;
4. }

    3) 存储：

   LinkedHashMap并未重写父类HashMap的put方法，而是重写了父类HashMap的put方法调用的子方法void addEntry(int hash, K key, V value, int bucketIndex) 和void createEntry(int hash, K key, V value, int bucketIndex)，提供了自己特有的双向链接列表的实现。

Java代码

1. void addEntry(int hash, K key, V value, int bucketIndex) {
2. // 调用create方法，将新元素以双向链表的的形式加入到映射中。
3. createEntry(hash, key, value, bucketIndex);
5. // 删除最近最少使用元素的策略定义
6. Entry<K,V> eldest = header.after;
7. if (removeEldestEntry(eldest)) {
8. removeEntryForKey(eldest.key);
9. } else {
10. if (size >= threshold)
11. resize(2 \* table.length);
12. }
13. }

Java代码

1. void createEntry(int hash, K key, V value, int bucketIndex) {
2. HashMap.Entry<K,V> old = table[bucketIndex];
3. Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
4. table[bucketIndex] = e;
5. // 调用元素的addBrefore方法，将元素加入到哈希、双向链接列表。
6. e.addBefore(header);
7. size++;
8. }

Java代码

1. private void addBefore(Entry<K,V> existingEntry) {
2. after  = existingEntry;
3. before = existingEntry.before;
4. before.after = this;
5. after.before = this;
6. }

    4) 读取：

   LinkedHashMap重写了父类HashMap的get方法，实际在调用父类getEntry()方法取得查找的元素后，再判断当排序模式accessOrder为true时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失。

Java代码

1. public V get(Object key) {
2. // 调用父类HashMap的getEntry()方法，取得要查找的元素。
3. Entry<K,V> e = (Entry<K,V>)getEntry(key);
4. if (e == null)
5. return null;
6. // 记录访问顺序。
7. e.recordAccess(this);
8. return e.value;
9. }

Java代码

1. void recordAccess(HashMap<K,V> m) {
2. LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
3. // 如果定义了LinkedHashMap的迭代顺序为访问顺序，
4. // 则删除以前位置上的元素，并将最新访问的元素添加到链表表头。
5. if (lm.accessOrder) {
6. lm.modCount++;
7. remove();
8. addBefore(lm.header);
9. }
10. }

    5) 排序模式：

   LinkedHashMap定义了排序模式accessOrder，该属性为boolean型变量，对于访问顺序，为true；对于插入顺序，则为false。

Java代码

1. private final boolean accessOrder;

 一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。看LinkedHashMap的构造方法，如：

Java代码

1. public LinkedHashMap(int initialCapacity, float loadFactor) {
2. super(initialCapacity, loadFactor);
3. accessOrder = false;
4. }

    这些构造方法都会默认指定排序模式为插入顺序。如果你想构造一个LinkedHashMap，并打算按从近期访问最少到近期访问最多的顺序（即访问顺序）来保存元素，那么请使用下面的构造方法构造LinkedHashMap：

Java代码

1. public LinkedHashMap(int initialCapacity,
2. float loadFactor,
3. boolean accessOrder) {
4. super(initialCapacity, loadFactor);
5. this.accessOrder = accessOrder;
6. }

    该哈希映射的迭代顺序就是最后访问其条目的顺序，这种映射很适合构建LRU缓存。LinkedHashMap提供了removeEldestEntry(Map.Entry<K,V> eldest)方法，在将新条目插入到映射后，put和 putAll将调用此方法。该方法可以提供在每次添加新条目时移除最旧条目的实现程序，默认返回false，这样，此映射的行为将类似于正常映射，即永远不能移除最旧的元素。

Java代码

1. protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
2. return false;
3. }

    此方法通常不以任何方式修改映射，相反允许映射在其返回值的指引下进行自我修改。如果用此映射构建LRU缓存，则非常方便，它允许映射通过删除旧条目来减少内存损耗。  
   例如：重写此方法，维持此映射只保存100个条目的稳定状态，在每次添加新条目时删除最旧的条目。

Java代码

1. private static final int MAX\_ENTRIES = 100;
2. protected boolean removeEldestEntry(Map.Entry eldest) {
3. return size() > MAX\_ENTRIES;
4. }

## 13、[LinkedHashSet的实现原理](http://zhangshixi.iteye.com/blog/673319)

1.  LinkedHashSet概述：

   LinkedHashSet是具有可预知迭代顺序的Set接口的哈希表和链接列表实现。此实现与HashSet的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可为插入顺序或是访问顺序。

   注意，此实现不是同步的。如果多个线程同时访问链接的哈希Set，而其中至少一个线程修改了该Set，则它必须保持外部同步。

2. LinkedHashSet的实现：

   对于LinkedHashSet而言，它继承与HashSet、又基于LinkedHashMap来实现的。

   LinkedHashSet底层使用LinkedHashMap来保存所有元素，它继承与HashSet，其所有的方法操作上又与HashSet相同，因此LinkedHashSet 的实现上非常简单，只提供了四个构造方法，并通过传递一个标识参数，调用父类的构造器，底层构造一个LinkedHashMap来实现，在相关操作上与父类HashSet的操作相同，直接调用父类HashSet的方法即可。LinkedHashSet的源代码如下：

Java代码

1. public class LinkedHashSet<E>
2. extends HashSet<E>
3. implements Set<E>, Cloneable, java.io.Serializable {
5. private static final long serialVersionUID = -2851667679971038690L;
7. /\*\*
8. \* 构造一个带有指定初始容量和加载因子的新空链接哈希set。
9. \*
10. \* 底层会调用父类的构造方法，构造一个有指定初始容量和加载因子的LinkedHashMap实例。
11. \* @param initialCapacity 初始容量。
12. \* @param loadFactor 加载因子。
13. \*/
14. public LinkedHashSet(int initialCapacity, float loadFactor) {
15. super(initialCapacity, loadFactor, true);
16. }
18. /\*\*
19. \* 构造一个带指定初始容量和默认加载因子0.75的新空链接哈希set。
20. \*
21. \* 底层会调用父类的构造方法，构造一个带指定初始容量和默认加载因子0.75的LinkedHashMap实例。
22. \* @param initialCapacity 初始容量。
23. \*/
24. public LinkedHashSet(int initialCapacity) {
25. super(initialCapacity, .75f, true);
26. }
28. /\*\*
29. \* 构造一个带默认初始容量16和加载因子0.75的新空链接哈希set。
30. \*
31. \* 底层会调用父类的构造方法，构造一个带默认初始容量16和加载因子0.75的LinkedHashMap实例。
32. \*/
33. public LinkedHashSet() {
34. super(16, .75f, true);
35. }
37. /\*\*
38. \* 构造一个与指定collection中的元素相同的新链接哈希set。
39. \*
40. \* 底层会调用父类的构造方法，构造一个足以包含指定collection
41. \* 中所有元素的初始容量和加载因子为0.75的LinkedHashMap实例。
42. \* @param c 其中的元素将存放在此set中的collection。
43. \*/
44. public LinkedHashSet(Collection<? extends E> c) {
45. super(Math.max(2\*c.size(), 11), .75f, true);
46. addAll(c);
47. }
48. }

   在父类HashSet中，专为LinkedHashSet提供的构造方法如下，该方法为包访问权限，并未对外公开。

Java代码

1. /\*\*
2. \* 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。
3. \* 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支持。
4. \*
5. \* 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。
6. \* @param initialCapacity 初始容量。
7. \* @param loadFactor 加载因子。
8. \* @param dummy 标记。
9. \*/
10. HashSet(int initialCapacity, float loadFactor, boolean dummy) {
11. map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);
12. }

    由上述源代码可见，LinkedHashSet通过继承HashSet，底层使用LinkedHashMap，以很简单明了的方式来实现了其自身的所有功能。

## 14、HashMap 与HashTable的区别

HashMap 与HashTable的区别

HashMap与Hashtable的区别是面试中经常遇到的一个问题。这个问题看似简单，但如果深究进去，也能了解到不少知识。本文对两者从来源，特性，算法等多个方面进行对比总结。力争多角度，全方位的展示二者的不同，做到此问题的终结版。

1 作者

Hashtable的作者：

这里写图片描述

HashMap的作者：

这里写图片描述

Hash Map的作者比Hashtable的作者多了著名顶顶的并发大神Doug Lea。他写了util.concurrent包。著有并发编程圣经Concurrent Programming in Java: Design Principles and Patterns 一书。他的个人主页： http://g.oswego.edu/

Josh Bloch 为领导了众多Java平台特性的设计和实现，其中包括Java Collection框架、java.math包以及assert机制。著有 Effective Java 一书。

Arthur van Hoff最早任职于硅谷的Sun Microsystems公司，从事Java程序语言的早期开发工作。设计并实现了JDK 1.0的许多方面，包括Java编译器、Java调试器、许多标准Java类以及HotJava浏览器。随后创立了多家成功的企业，其中包括Marimba（1999年IPO）、Strangeberry（后被TiVo收购）、ZING（后被Dell收购）和Ellerdale（后被Flipboard收购）。Java命名来源有这么一种说法，来源于开发人员名字的组合：James Gosling、Arthur Van Hoff和Andy Bechtolsheim首字母的缩写。

Neal Gafter是Java SE 4和5语言增强的主要设计者和实现者，他的Java闭包实现赢得了OpenJDK创新者挑战赛的大奖。他也在继续参与SE 7和8的语言发展。之前Neal在为Google的在线日历工作，也曾经是C++标准委员会的一员，并曾在Sun微系统公司，MicroTec研究院和德州仪器领导开发C和C++编译器。如今Neal在微软开发.NET平台编程语言。Neal是《Java Puzzlers：Traps, Pitfalls and Corner Cases》（Addison Wesley，2005）一书的合作者。他拥有罗彻斯特大学计算机科学的博士学位。

可见这些作者都是java乃至整个it领域大名鼎鼎的人物。也只有这些大师级人物才能写出HashMap这么大道至简的数据类型了。

2 产生时间

Hashtable是java一开始发布时就提供的键值映射的数据结构，而HashMap产生于JDK1.2。虽然Hashtable比HashMap出现的早一些，但是现在Hashtable基本上已经被弃用了。而HashMap已经成为应用最为广泛的一种数据类型了。造成这样的原因一方面是因为Hashtable是线程安全的，效率比较低。另一方面可能是因为Hashtable没有遵循驼峰命名法吧。。。

3 继承的父类不同

HashMap和Hashtable不仅作者不同，而且连父类也是不一样的。HashMap是继承自AbstractMap类，而HashTable是继承自Dictionary类。不过它们都实现了同时实现了map、Cloneable（可复制）、Serializable（可序列化）这三个接口

这里写图片描述

这里写图片描述

Dictionary类是一个已经被废弃的类（见其源码中的注释）。父类都被废弃，自然而然也没人用它的子类Hashtable了。

\* NOTE: This class is obsolete. New implementations should

\* implement the Map interface, rather than extending this class.

4 对外提供的接口不同

Hashtable比HashMap多提供了elments() 和contains() 两个方法。

elments() 方法继承自Hashtable的父类Dictionnary。elements() 方法用于返回此Hashtable中的value的枚举。

contains()方法判断该Hashtable是否包含传入的value。它的作用与containsValue()一致。事实上，contansValue() 就只是调用了一下contains() 方法。

这里写图片描述

5 对Null key 和Null value的支持不同

Hashtable既不支持Null key也不支持Null value。Hashtable的put()方法的注释中有说明。

这里写图片描述

当key为Null时，调用put() 方法，运行到下面这一步就会抛出空指针异常。因为拿一个Null值去调用方法了。

这里写图片描述

当value为null值时，Hashtable对其做了限制，运行到下面这步也会抛出空指针异常。

这里写图片描述

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。

6 线程安全性不同

Hashtable是线程安全的，它的每个方法中都加入了Synchronize方法。在多线程并发的环境下，可以直接使用Hashtable，不需要自己为它的方法实现同步

HashMap不是线程安全的，在多线程并发的环境下，可能会产生死锁等问题。具体的原因在下一篇文章中会详细进行分析。使用HashMap时就必须要自己增加同步处理，

虽然HashMap不是线程安全的，但是它的效率会比Hashtable要好很多。这样设计是合理的。在我们的日常使用当中，大部分时间是单线程操作的。HashMap把这部分操作解放出来了。当需要多线程操作的时候可以使用线程安全的ConcurrentHashMap。ConcurrentHashMap虽然也是线程安全的，但是它的效率比Hashtable要高好多倍。因为ConcurrentHashMap使用了分段锁，并不对整个数据进行锁定。

7 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 。

HashMap的Iterator是fail-fast迭代器。当有其它线程改变了HashMap的结构（增加，删除，修改元素），将会抛出ConcurrentModificationException。不过，通过Iterator的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。

JDK8之前的版本中，Hashtable是没有fast-fail机制的。在JDK8及以后的版本中 ，HashTable也是使用fast-fail的， 源码如下：

这里写图片描述

modCount的使用类似于并发编程中的CAS（Compare and Swap）技术。我们可以看到这个方法中，每次在发生增删改的时候都会出现modCount++的动作。而modcount可以理解为是当前hashtable的状态。每发生一次操作，状态就向前走一步。设置这个状态，主要是由于hashtable等容器类在迭代时，判断数据是否过时时使用的。尽管hashtable采用了原生的同步锁来保护数据安全。但是在出现迭代数据的时候，则无法保证边迭代，边正确操作。于是使用这个值来标记状态。一旦在迭代的过程中状态发生了改变，则会快速抛出一个异常，终止迭代行为。

8 初始容量大小和每次扩充容量大小的不同

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。

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。也就是说Hashtable会尽量使用素数、奇数。而HashMap则总是使用2的幂作为哈希表的大小。

之所以会有这样的不同，是因为Hashtable和HashMap设计时的侧重点不同。Hashtable的侧重点是哈希的结果更加均匀，使得哈希冲突减少。当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀。而HashMap则更加关注hash的计算效率问题。在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。HashMap为了加快hash的速度，将哈希表的大小固定为了2的幂。当然这引入了哈希分布不均匀的问题，所以HashMap为解决这问题，又对hash算法做了一些改动。这从而导致了Hashtable和HashMap的计算hash值的方法不同

9 计算hash值的方法不同

为了得到元素的位置，首先需要根据元素的 KEY计算出一个hash值，然后再用这个hash值来计算得到最终的位置。

Hashtable直接使用对象的hashCode。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。然后再使用除留余数发来获得最终的位置。

这里写图片描述

Hashtable在计算元素的位置时需要进行一次除法运算，而除法运算是比较耗时的。

HashMap为了提高计算效率，将哈希表的大小固定为了2的幂，这样在取模预算时，不需要做除法，只需要做位运算。位运算比除法的效率要高很多。

HashMap的效率虽然提高了，但是hash冲突却也增加了。因为它得出的hash值的低位相同的概率比较高，而计算位运算

为了解决这个问题，HashMap重新根据hashcode计算hash值后，又对hash值做了一些运算来打散数据。使得取得的位置更加分散，从而减少了hash冲突。当然了，为了高效，HashMap只做了一些简单的位处理。从而不至于把使用2 的幂次方带来的效率提升给抵消掉。

这里写图片描述

附上关于这个问题的说明：

Joshua Bloch: The downside of using a power-of-two is that the resulting hash table is very sensitive to the quality of the hash function (hashCode). It is imperative that any change in the input must affect the low order bits of the hash value. (Ideally, it should affect all bits of the hash value with equal likelihood.) Because we have no assurance that this is true, we put in a secondary (or “defensive”) hash function when we switched to the power-of-two hash table. This hash function is applied to the results of hashCode before masking off the low order bits. Its job is to scatter the information over all the bits, and in particular, into the low order bits. Of course it has to run very fast, or you lose the benefit of switching to the power-of-two-sized table. The original secondary hash function in 1.4 turned out to be insufficient. We knew that this was a theoretical possibility, but we thought that it didn’t affect any practical data sets. We were wrong. The replacement secondary hash function (which I developed with the aid of a computer) has strong statistical properties that pretty much guarantee good bucket distribution.

---------------------

作者：hahahaha233

来源：CSDN

原文：https://blog.csdn.net/wangxing233/article/details/79452946

版权声明：本文为博主原创文章，转载请附上博文链接！

**HashTable**

* 底层数组+链表实现，无论key还是value都**不能为null**，线程**安全**，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化
* 初始size为**11**，扩容：newsize = olesize\*2+1
* 计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

**HashMap**

* 底层数组+链表实现，可**以存储null键和null值**，线程**不安全**
* 初始size为**16**，扩容：newsize = oldsize\*2，size一定为2的n次幂
* 扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入
* 插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）
* 当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀
* 计算index方法：index = hash & (tab.length – 1)

HashMap的初始值还要考虑加载因子:

* **哈希冲突**：若干Key的哈希值按数组大小取模后，如果落在同一个数组下标上，将组成一条Entry链，对Key的查找需要遍历Entry链上的每个元素执行equals()比较。
* **加载因子**：为了降低哈希冲突的概率，默认当HashMap中的键值对达到数组大小的75%时，即会触发扩容。因此，如果预估容量是100，即需要设定100/0.75＝134的数组大小。
* ***空间换时间***：如果希望加快Key查找的时间，还可以进一步降低加载因子，加大初始大小，以降低哈希冲突的概率。

HashMap和Hashtable都是用hash算法来决定其元素的存储，因此HashMap和Hashtable的hash表包含如下属性：

* 容量（capacity）：hash表中桶的数量
* 初始化容量（initial capacity）：创建hash表时桶的数量，HashMap允许在构造器中指定初始化容量
* 尺寸（size）：当前hash表中记录的数量
* 负载因子（load factor）：负载因子等于“size/capacity”。负载因子为0，表示空的hash表，0.5表示半满的散列表，依此类推。轻负载的散列表具有冲突少、适宜插入与查询的特点（但是使用Iterator迭代元素时比较慢）

除此之外，hash表里还有一个“负载极限”，“负载极限”是一个0～1的数值，“负载极限”决定了hash表的最大填满程度。当hash表中的负载因子达到指定的“负载极限”时，hash表会自动成倍地增加容量（桶的数量），并将原有的对象重新分配，放入新的桶内，这称为rehashing。

HashMap和Hashtable的构造器允许指定一个负载极限，HashMap和Hashtable默认的“负载极限”为0.75，这表明当该hash表的3/4已经被填满时，hash表会发生rehashing。

“负载极限”的默认值（0.75）是时间和空间成本上的一种折中：

* 较高的“负载极限”可以降低hash表所占用的内存空间，但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap的get()与put()方法都要用到查询）
* 较低的“负载极限”会提高查询数据的性能，但会增加hash表所占用的内存开销

程序猿可以根据实际情况来调整“负载极限”值。

**ConcurrentHashMap**

* 底层采用分段的数组+链表实现，线程**安全**
* 通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)
* Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术
* 有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁
* 扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

Hashtable和HashMap都实现了Map接口，但是Hashtable的实现是基于Dictionary抽象类的。Java5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。

HashMap基于哈希思想，实现对数据的读写。当我们将键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，然后找到bucket位置来存储值对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞时，对象将会储存在链表的下一个节点中。HashMap在每个链表节点中储存键值对对象。当两个不同的键对象的hashcode相同时，它们会储存在同一个bucket位置的链表中，可通过键对象的equals()方法来找到键值对。如果链表大小超过阈值（TREEIFY\_THRESHOLD,8），链表就会被改造为树形结构。

在HashMap中，null可以作为键，这样的键只有一个，但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，即可以表示HashMap中没有该key，也可以表示该key所对应的value为null**。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个key，应该用**containsKey()**方法来判断。而在Hashtable中，无论是key还是value都不能为null。

Hashtable是线程安全的，它的方法是同步的，可以直接用在多线程环境中。而HashMap则不是线程安全的，在多线程环境中，需要手动实现同步机制。

Hashtable与HashMap另一个区别是HashMap的迭代器（Iterator）是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。

先看一下简单的类图：

从类图中可以看出来在存储结构中ConcurrentHashMap比HashMap多出了一个类Segment，而Segment是一个可重入锁。

ConcurrentHashMap是使用了锁分段技术来保证线程安全的。

**锁分段技术**：首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap提供了与Hashtable和SynchronizedMap不同的锁机制。Hashtable中采用的锁机制是一次锁住整个hash表，从而在同一时刻只能由一个线程对其进行操作；而ConcurrentHashMap中则是一次锁住一个桶。

ConcurrentHashMap默认将hash表分为16个桶，诸如get、put、remove等常用操作只锁住当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。

## 15、不同数据之间的相互转化

### 1、基本数据类型

基本类型有以下四种：  
1）int长度数据类型有：byte(8bits)、short(16bits)、int(32bits)、long(64bits)  
2）float长度数据类型有：单精度（32bits float）、双精度（64bits double），JAVA中将小数默认为double类型，若要定义float需在数据后面用f声明；  
3）boolean类型变量的取值有：ture、false  
4）char数据类型有：unicode字符,16位  
对应的类类型：Integer、Float、Boolean、Character、Double、Short、Byte、Long

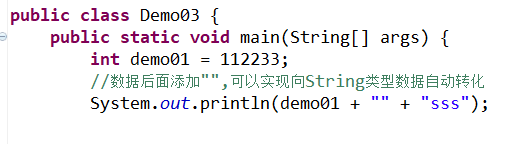
基本的数据类型由低级到高级分别为：（byte、short、char）——int——long——float——double

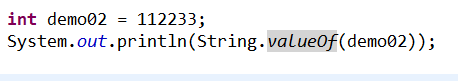
### 2、字符串与其他数据类型转换

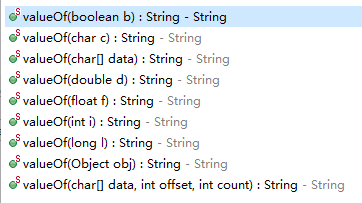
⑴其它类型向字符串的转换

①调用类的串转换方法:X.toString();

  
②自动转换:X + “”;

  
③使用String的方法:String.volueOf(X);





⑵ 字符串作为值,向其它类型的转换

①先转换成相应的封装器实例,再调用对应的方法转换成其它类型  
例如，字符中“32.1”转换double型的值的格式为:new Float(“32.1”).doubleValue()。也可以用:Double.valueOf(“32.1”).doubleValue()  
②静态parseXXX方法  
String s = "1";  
byte b = Byte.parseByte( s );  
short t = Short.parseShort( s );  
int i = Integer.parseInt( s );  
long l = Long.parseLong( s );  
Float f = Float.parseFloat( s );  
Double d = Double.parseDouble( s );  
③ Character的getNumericValue(char ch)方法

转换实例

1）基本类型向类类型转换

正向转换：通过类包装器来new出一个新的类类型的变量  
Integer a= new Integer(2);  
反向转换：通过类包装器来转换  
int b=a.intValue();

通过类包装器——>基本数据类型  
eg1:int i=Integer.parseInt(“123”)  
说明：此方法只能适用于字符串转化成整型变量  
eg2: float f=Float.valueOf(“123”).floatValue()  
说明：上例是将一个字符串转化成一个Float对象，然后再调用这个对象的floatValue()方法返回其对应的float数值。  
eg3: boolean b=Boolean.valueOf(“123”).booleanValue()  
说明：上例是将一个字符串转化成一个Boolean对象，然后再调用这个对象的booleanValue()方法返回其对应的boolean数值。  
eg4:double d=Double.valueOf(“123”).doubleValue()  
说明：上例是将一个字符串转化成一个Double对象，然后再调用这个对象的doubleValue()方法返回其对应的double数值。  
eg5: long l=Long.valueOf(“123”).longValue()  
说明：上例是将一个字符串转化成一个Long对象，然后再调用这个对象的longValue()方法返回其对应的long数值。  
eg6: char=Character.valueOf(“123”).charValue()  
说明：上例是将一个字符串转化成一个Character对象，然后再调用这个对象的charValue()方法返回其对应的char数值。

2）基本类型向字符串的转换  
正向转换：  
如下：  
System.out.println(""+2+3);// “""”把2转成字符串操作；  
System.out.println(2+3);   // 不存在转换。  
System.out.println(2+3+"");// 前两个数值相加后，被“""”转成字符串。  
System.out.println(2+""+3);// 同第一个

输出显示为：23，5，5，23

3）类类型向字符串转换

正向转换：因为每个类都是object类的子类，而所有的object类都有一个toString()函数，所以通过toString()函数来转换即可  
反向转换：通过类包装器new出一个新的类类型的变量  
eg1: int i=Integer.valueOf(“123”).intValue()  
说明：上例是将一个字符串转化成一个Integer对象，然后再调用这个对象的intValue()方法返回其对应的int数值。  
eg2: float f=Float.valueOf(“123”).floatValue()  
说明：上例是将一个字符串转化成一个Float对象，然后再调用这个对象的floatValue()方法返回其对应的float数值。  
eg3: boolean b=Boolean.valueOf(“123”).booleanValue()  
说明：上例是将一个字符串转化成一个Boolean对象，然后再调用这个对象的booleanValue()方法返回其对应的boolean数值。  
eg4:double d=Double.valueOf(“123”).doubleValue()  
说明：上例是将一个字符串转化成一个Double对象，然后再调用这个对象的doubleValue()方法返回其对应的double数值。  
eg5: long l=Long.valueOf(“123”).longValue()  
说明：上例是将一个字符串转化成一个Long对象，然后再调用这个对象的longValue()方法返回其对应的long数值。  
eg6: char=Character.valueOf(“123”).charValue()  
说明：上例是将一个字符串转化成一个Character对象，然后再调用这个对象的charValue()方法返回其对应的char数值。

Char数组转化为字符串：

