**八种基本数据类型的大小，以及他们的封装类。**

八种基本数据类型，int ,double ,long ,float, short,byte,character,boolean

对应的封装类型是：Integer ,Double ,Long ,Float, Short,Byte,Character,Boolean

**Switch能否用string做参数？**

从Java 7开始，expr还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的

**equals与==的区别**

使用==比较有两种情况

1. 比较基础数据类型(Java中基础数据类型包括八中short, int, long, float, double, char, byte, boolen )：这种情况下，==比较的是他们的值是否相等。
2. 引用间的比较：在这种情况下，==比较的是他们在内存中的地址，也就是说，除非引用指向的是同一个new出来的对象，此时他们使用`==`去比较得到true，否则，得到false。

使用equals进行比较：

1. equals追根溯源，是Object类中的一个方法，在该类中，equals的实现也仅仅只是比较两个对象的内存地址是否相等，但在一些子类中，如：String、Integer 等，该方法将被重写。

以String类为例子说明eqauls与==的区别：

在开始这个例子之前，同学们需要知道JVM处理String的一些特性。Java的虚拟机在内存中开辟出一块单独的区域，用来存储字符串对象，这块内存区域被称为字符串缓冲池。当使用 *String a = "abc"*这样的语句进行定义一个引用的时候，首先会在字符串缓冲池中查找是否已经相同的对象，如果存在，那么就直接将这个对象的引用返回给a，如果不存在，则需要新建一个值为"abc"的对象，再将新的引用返回a。String a = new String("abc");这样的语句明确告诉JVM想要产生一个新的String对象，并且值为"abc"，于是就在堆内存中的某一个小角落开辟了一个新的String对象。

String str1 = "abc";

String str2 = "abc";

System.out.println(str1 == str2); //true

System.out.println(str1.equals(str2)); //true

String str2 = new String("abc");

System.out.println(str1 == str2); //false(两个对象的引用不一样)

System.out.println(str1.equals(str2)); //true（String重写了equals方法）

以上代码将会输出

true

true

false

true

\*\*第一个true：\*\*因为在str2赋值之前，str1的赋值操作就已经在内存中创建了一个值为"abc"的对象了，然后str2将会与str1指向相同的地址。

\*\*第二个true：\*\*因为`String`已经重写了`equals`方法：为了方便大家阅读我贴出来，并且在注释用进行分析：

```

public boolean equals(Object anObject) {

//如果比较的对象与自身内存地址相等的话

//就说明他两指向的是同一个对象

//所以此时equals的返回值跟==的结果是一样的。

if (this == anObject) {

return true;

}

//当比较的对象与自身的内存地址不相等，并且

//比较的对象是String类型的时候

//将会执行这个分支

if (anObject instanceof String) {

String anotherString = (String)anObject;

int n = value.length;

if (n == anotherString.value.length) {

char v1[] = value;

char v2[] = anotherString.value;

int i = 0;

//在这里循环遍历两个String中的char

while (n-- != 0) {

//只要有一个不相等，那么就会返回false

if (v1[i] != v2[i])

return false;

i++;

}

return true;

}

}

return false;

}

```

进行以上分析之后，就不难理解第一段代码中的实例程序输出了。

**总结：**

1. 使用==比较原生类型如：boolean、int、char等等，使用equals()比较对象。
2. ==返回true如果两个引用指向相同的对象，equals()返回结果依赖于具体业务实现
3. 字符串的对比使用equals()代替==操作符

以上就是关于equals方法和==操作符的区别，其主要的不同是一个是操作符一个是方法，==用于对比原生类型而equals()方法比较对象的相等性。

**Object有哪些公用方法？**

1．clone方法

保护方法，实现对象的浅复制，只有实现了Cloneable接口才可以调用该方法，否则抛出CloneNotSupportedException异常。主要是JAVA里除了8种基本类型传参数是值传递，其他的类对象传参数都是引用传递，我们有时候不希望在方法里讲参数改变，这是就需要在类中复写clone方法。

2．getClass方法

final方法，获得运行时类型。

3．toString方法

该方法用得比较多，一般子类都有覆盖。

4．finalize方法

该方法用于释放资源。因为无法确定该方法什么时候被调用，很少使用。

5．equals方法

该方法是非常重要的一个方法。一般equals和==是不一样的，但是在Object中两者是一样的。子类一般都要重写这个方法。

6．hashCode方法

该方法用于哈希查找，可以减少在查找中使用equals的次数，重写了equals方法一般都要重写hashCode方法。这个方法在一些具有哈希功能的Collection中用到。

一般必须满足obj1.equals(obj2)==true。可以推出obj1.hashCode()==obj2.hashCode()，但是hashCode相等不一定就满足equals。不过为了提高效率，应该尽量使上面两个条件接近等价。

如果不重写hashCode(),在HashSet中添加两个equals的对象，会将两个对象都加入进去。

7．wait方法

wait方法就是使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。wait()方法一直等待，直到获得锁或者被中断。wait(long timeout)设定一个超时间隔，如果在规定时间内没有获得锁就返回。

调用该方法后当前线程进入睡眠状态，直到以下事件发生。

（1）其他线程调用了该对象的notify方法。

（2）其他线程调用了该对象的notifyAll方法。

（3）其他线程调用了interrupt中断该线程。

（4）时间间隔到了。

此时该线程就可以被调度了，如果是被中断的话就抛出一个InterruptedException异常。

8．notify方法

该方法唤醒在该对象上等待的某个线程。

9．notifyAll方法

该方法唤醒在该对象上等待的所有线程

**String、StringBuffer与StringBuilder的区别。**

Java 平台提供了两种类型的字符串：String和StringBuffer / StringBuilder，它们可以储存和操作字符串。其中String是只读字符串，也就意味着String引用的字符串内容是不能被改变的。而StringBuffer和StringBulder类表示的字符串对象可以直接进行修改。StringBuilder是JDK1.5引入的，它和StringBuffer的方法完全相同，区别在于它是单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer略高。

**try catch finally，try里有return，finally还执行么？**

会执行，在方法 返回调用者前执行。Java允许在finally中改变返回值的做法是不好的，因为如果存在finally代码块，try中的return语句不会立马返回调用者，而是纪录下返回值待finally代码块执行完毕之后再向调用者返回其值，然后如果在finally中修改了返回值，这会对程序造成很大的困扰，C#中就从语法规定不能做这样的事。

**Excption与Error区别**

Error表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的状况；Exception表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

**Excption与Error包结构。OOM你遇到过哪些情况，SOF你遇到过哪些情况。**

Java异常架构图

[](https://camo.githubusercontent.com/2248cdf95b8f2569ee0e58aac3d8ed179c89f190/687474703a2f2f696d61676573323031352e636e626c6f67732e636f6d2f626c6f672f3637393930342f3230313531302f3637393930342d32303135313032353231303831333938392d3932313932373931362e6a7067)

1. Throwable：Throwable是 Java 语言中所有错误或异常的超类。 Throwable包含两个子类: Error 和 Exception 。它们通常用于指示发生了异常情况。 Throwable包含了其线程创建时线程执行堆栈的快照，它提供了printStackTrace()等接口用于获取堆栈跟踪数据等信息。
2. Exception：Exception及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。
3. RuntimeException： RuntimeException是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。 编译器不会检查RuntimeException异常。 例如，除数为零时，抛出ArithmeticException异常。RuntimeException是ArithmeticException的超类。当代码发生除数为零的情况时，倘若既"没有通过throws声明抛出ArithmeticException异常"，也"没有通过try...catch...处理该异常"，也能通过编译。这就是我们所说的"**编译器不会检查RuntimeException异常**"！ 如果代码会产生RuntimeException异常，则需要通过修改代码进行避免。 例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！
4. Error 和Exception一样， Error也是Throwable的子类。 它用于指示合理的应用程序不应该试图捕获的严重问题，大多数这样的错误都是异常条件。 和RuntimeException一样， 编译器也不会检查Error。

J**ava将可抛出(Throwable)的结构分为三种类型**： 被检查的异常(Checked Exception)，运行时异常(RuntimeException)和错误(Error)。

(01) **运行时异常**

**定义** : RuntimeException及其子类都被称为运行时异常。

**特点** : Java编译器不会检查它。 也就是说，当程序中可能出现这类异常时，倘若既"没有通过throws声明抛出它"，也"没有用try-catch语句捕获它"，还是会编译通过。例如，除数为零时产生的ArithmeticException异常，数组越界时产生的IndexOutOfBoundsException异常，fail-fail机制产生的ConcurrentModificationException异常等，都属于运行时异常。 虽然Java编译器不会检查运行时异常，但是我们也可以通过throws进行声明抛出，也可以通过try-catch对它进行捕获处理。 如果产生运行时异常，则需要通过修改代码来进行避免。 例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

(02) **被检查的异常**

**定义** : Exception类本身，以及Exception的子类中除了"运行时异常"之外的其它子类都属于被检查异常。

**特点** : Java编译器会检查它。 此类异常，要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。例如，CloneNotSupportedException就属于被检查异常。当通过clone()接口去克隆一个对象，而该对象对应的类没有实现Cloneable接口，就会抛出CloneNotSupportedException异常。 被检查异常通常都是可以恢复的。

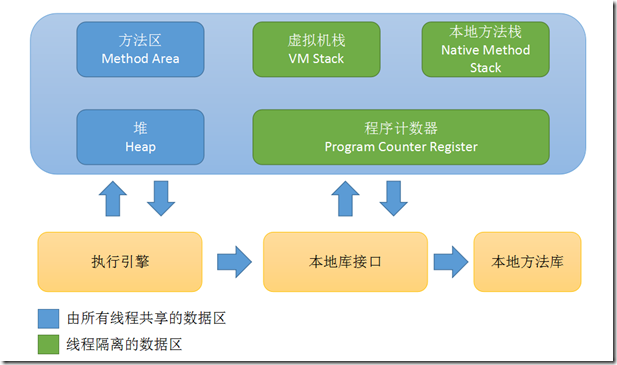
(03) **错误**

**定义** : Error类及其子类。

**特点** : 和运行时异常一样，编译器也不会对错误进行检查。 当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。例如，VirtualMachineError就属于错误。 按照Java惯例，我们是不应该是实现任何新的Error子类的！

对于上面的3种结构，我们在抛出异常或错误时，到底该哪一种？《Effective Java》中给出的建议是： 对于可以恢复的条件使用被检查异常，对于程序错误使用运行时异常。

**Java虚拟机所管理的内存将会包括一下几个运行时数据区域**：

[](http://images2015.cnblogs.com/blog/1182497/201706/1182497-20170616192739978-1176032049.png)

**1.程序计数器**

程序计数器（Program Counter Register） 是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条执行字节码指令。

每条线程都有一个独立的程序计数器。

如果执行的是java方法，这个计数器记录的是正在执行的虚拟机字节码指令地址。如果是native方法，计数器为空。此内存区域是唯一一个在java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

**2.Java虚拟机栈**

同样是线程私有，描述Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。一个方法对应一个栈帧。

局部变量表存放了各种基本类型、对象引用和returnAddress类型（指向了一条字节码指令地址）。其中64位长度long 和 double占两个局部变量空间，其他只占一个。

规定的异常情况有两种：1.线程请求的栈的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；2.如果虚拟机可以动态扩展，如果扩展时无法申请到足够的内存，就抛出OutOfMemoryError异常。

**3.本地方法栈**

和Java虚拟机栈很类似，不同的是本地方法栈为Native方法服务。

**4.Java堆**

是Java虚拟机所管理的内存中最大的一块。由所有线程共享，在虚拟机启动时创建。堆区唯一目的就是存放对象实例。

堆中可细分为新生代和老年代，再细分可分为Eden空间、From Survivor空间、To Survivor空间。

堆无法扩展时，抛出OutOfMemoryError异常

**5.方法区**

所有线程共享，存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

当方法区无法满足内存分配需求时，抛出OutOfMemoryError

**6.运行时常量池**

它是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项是常量池（Const Pool Table），用于存放编译期生成的各种字面量和符号引用。并非预置入Class文件中常量池的内容才进入方法运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。

当方法区无法满足内存分配需求时，抛出OutOfMemoryError

**7.直接内存**

并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域。

JDK1.4加入了NIO，引入一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。因为避免了在Java堆和Native堆中来回复制数据，提高了性能。

当各个内存区域总和大于物理内存限制，抛出OutOfMemoryError异常。

**OOM：**

1. **OutOfMemoryError异常**

除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生OutOfMemoryError(OOM)异常的可能，

Java Heap 溢出

一般的异常信息：java.lang.OutOfMemoryError:Java heap spacess

java堆用于存储对象实例，我们只要不断的创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量达到最大堆容量限制后产生内存溢出异常。

出现这种异常，一般手段是先通过内存映像分析工具(如Eclipse Memory Analyzer)对dump出来的堆转存快照进行分析，重点是确认内存中的对象是否是必要的，先分清是因为内存泄漏(Memory Leak)还是内存溢出(Memory Overflow)。

如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链。于是就能找到泄漏对象时通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收。

如果不存在泄漏，那就应该检查虚拟机的参数(-Xmx与-Xms)的设置是否适当。

1. **虚拟机栈和本地方法栈溢出**

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常

这里需要注意当栈的大小越大可分配的线程数就越少。

1. **运行时常量池溢出**

异常信息：java.lang.OutOfMemoryError:PermGen space

如果要向运行时常量池中添加内容，最简单的做法就是使用String.intern()这个Native方法。该方法的作用是：如果池中已经包含一个等于此String的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize和-XX:MaxPermSize限制方法区的大小，从而间接限制其中常量池的容量。

1. **方法区溢出**

方法区用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。

异常信息：java.lang.OutOfMemoryError:PermGen space

方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收，判定条件是很苛刻的。在经常动态生成大量Class的应用中，要特别注意这点。

**Java面向对象的三个特征与含义。**

**继承**：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。

**封装**：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口（可以想想普通洗衣机和全自动洗衣机的差别，明显全自动洗衣机封装更好因此操作起来更简单；我们现在使用的智能手机也是封装得足够好的，因为几个按键就搞定了所有的事情）。

**多态**：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当A系统访问B系统提供的服务时，B系统有多种提供服务的方式，但一切对A系统来说都是透明的（就像电动剃须刀是A系统，它的供电系统是B系统，B系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A系统只会通过B类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1. 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2. 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

**foreach与正常for循环效率对比。**

直接for循环效率最高，其次是迭代器和 ForEach操作。 作为语法糖，其实 ForEach 编译成 字节码之后，使用的是迭代器实现的，反编译后，testForEach方法如下：

public static void testForEach(List list) {

for (Iterator iterator = list.iterator(); iterator.hasNext();) {

Object t = iterator.next();

Object obj = t;

}

}

可以看到，只比迭代器遍历多了生成中间变量这一步，因为性能也略微下降了一些。

Try-catch-finally的用法

结论一：

   return语句并不是函数的最终出口，如果有finally语句，这在return之后还会执行finally（return的值会暂存在栈里面，等待finally执行后再返回）  
结论二：

   finally里面不建议放return语句，根据需要，return语句可以放在try和catch里面和函数的最后。可行的做法有四：  
   （1）return语句只在函数最后出现一次。  
   （2）return语句仅在try和catch里面都出现。  
   （3）return语句仅在try和函数的最后都出现。  
   （4）return语句仅在catch和函数的最后都出现。  
   注意，除此之外的其他做法都是不可行的，编译器会报错。

**反射机制**

JAVA反射机制是在运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法; 对于任意一个对象, 都能够调用它的任意一个方法和属性; 这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制.

主要作用有三：

运行时取得类的方法和字段的相关信息。

创建某个类的新实例(.newInstance())

取得字段引用直接获取和设置对象字段，无论访问修饰符是什么。

用处如下：

观察或操作应用程序的运行时行为。

调试或测试程序，因为可以直接访问方法、构造函数和成员字段。

通过名字调用不知道的方法并使用该信息来创建对象和调用方法。

**泛型的优缺点**

优点：

使用泛型类型可以最大限度地重用代码、保护类型的安全以及提高性能。

泛型最常见的用途是创建集合类。

缺点：

在性能上不如数组快。

**泛型常用特点，List<String>能否转为List<Object>**

能，但是利用类都继承自Object，所以使用是每次调用里面的函数都要通过强制转换还原回原来的类，这样既不安全，运行速度也慢。

**ArrayList、LinkedList、Vector的底层实现和区别**

* 从同步性来看，ArrayList和LinkedList是不同步的，而Vector是的。所以线程安全的话，可以使用ArrayList或LinkedList，可以节省为同步而耗费的开销。但在多线程下，有时候就不得不使用Vector了。当然，也可以通过一些办法包装ArrayList、LinkedList，使我们也达到同步，但效率可能会有所降低。
* 从内部实现机制来讲ArrayList和Vector都是使用Object的数组形式来存储的。当你向这两种类型中增加元素的时候，如果元素的数目超出了内部数组目前的长度它们都需要扩展内部数组的长度，Vector缺省情况下自动增长原来一倍的数组长度，ArrayList是原来的50%，所以最后你获得的这个集合所占的空间总是比你实际需要的要大。如果你要在集合中保存大量的数据，那么使用Vector有一些优势，因为你可以通过设置集合的初始化大小来避免不必要的资源开销。
* ArrayList和Vector中，从指定的位置（用index）检索一个对象，或在集合的末尾插入、删除一个对象的时间是一样的，可表示为O(1)。但是，如果在集合的其他位置增加或者删除元素那么花费的时间会呈线性增长O(n-i)，其中n代表集合中元素的个数，i代表元素增加或移除元素的索引位置，因为在进行上述操作的时候集合中第i和第i个元素之后的所有元素都要执行(n-i)个对象的位移操作。LinkedList底层是由双向循环链表实现的，LinkedList在插入、删除集合中任何位置的元素所花费的时间都是一样的O(1)，但它在索引一个元素的时候比较慢，为O(i)，其中i是索引的位置，如果只是查找特定位置的元素或只在集合的末端增加、移除元素，那么使用Vector或ArrayList都可以。如果是对其它指定位置的插入、删除操作，最好选择LinkedList。

**HashMap和HashTable的底层实现和区别，两者和ConcurrentHashMap的区别。**

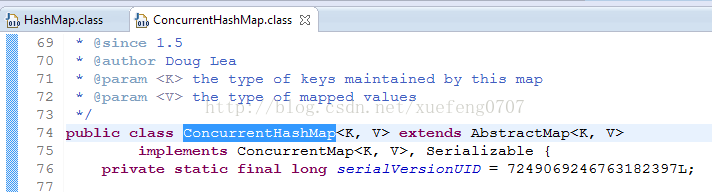
**总结：**HashTable线程安全则是依靠方法简单粗暴的sychronized修饰，HashMap则没有相关的线程安全问题考虑。。

在以前的版本貌似ConcurrentHashMap引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的HashTable，根据key.hashCode()来决定把key放到哪个HashTable中。在ConcurrentHashMap中，就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中。

通过把整个Map分为N个Segment（类似HashTable），可以提供相同的线程安全，但是效率提升N倍。

从JDK1.2起，就有了HashMap，正如前一篇文章所说，HashMap不是线程安全的，因此多线程操作时需要格外小心。

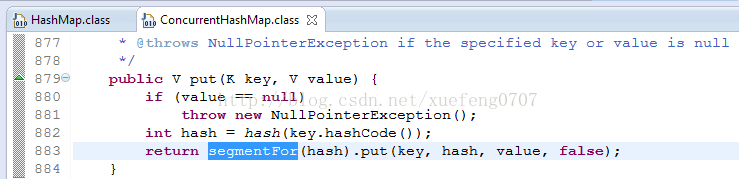
在JDK1.5中，伟大的Doug Lea给我们带来了concurrent包，从此Map也有安全的了。

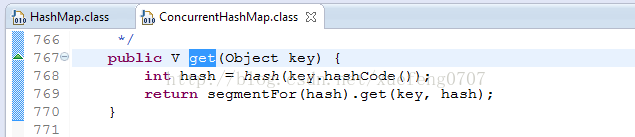


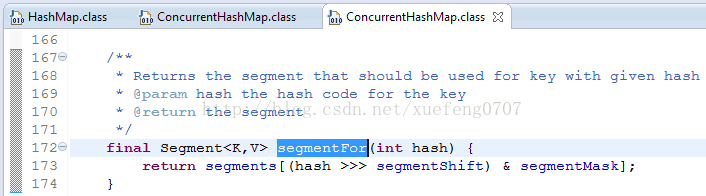
ConcurrentHashMap具体是怎么实现线程安全的呢，肯定不可能是每个方法加synchronized，那样就变成了HashTable。

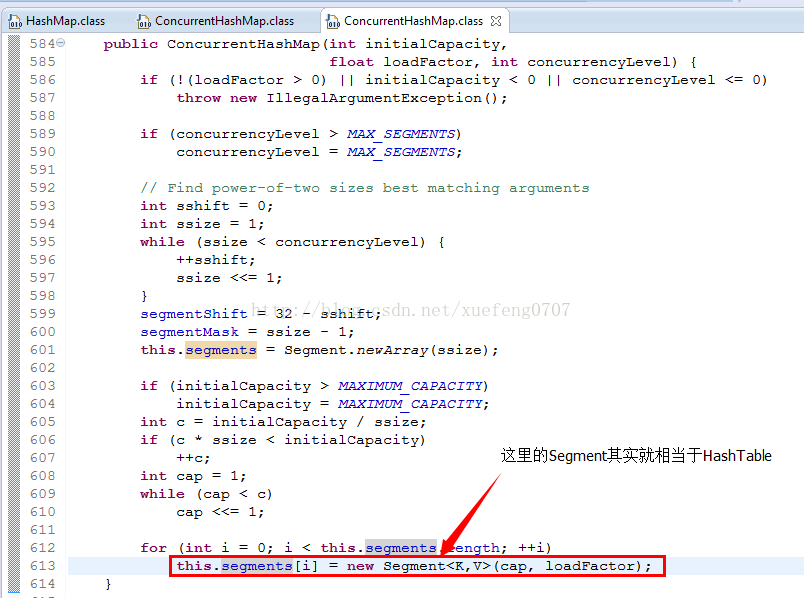
从ConcurrentHashMap代码中可以看出，它引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的HashTable，根据key.hashCode()来决定把key放到哪个HashTable中。

在ConcurrentHashMap中，就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中：









测试程序：

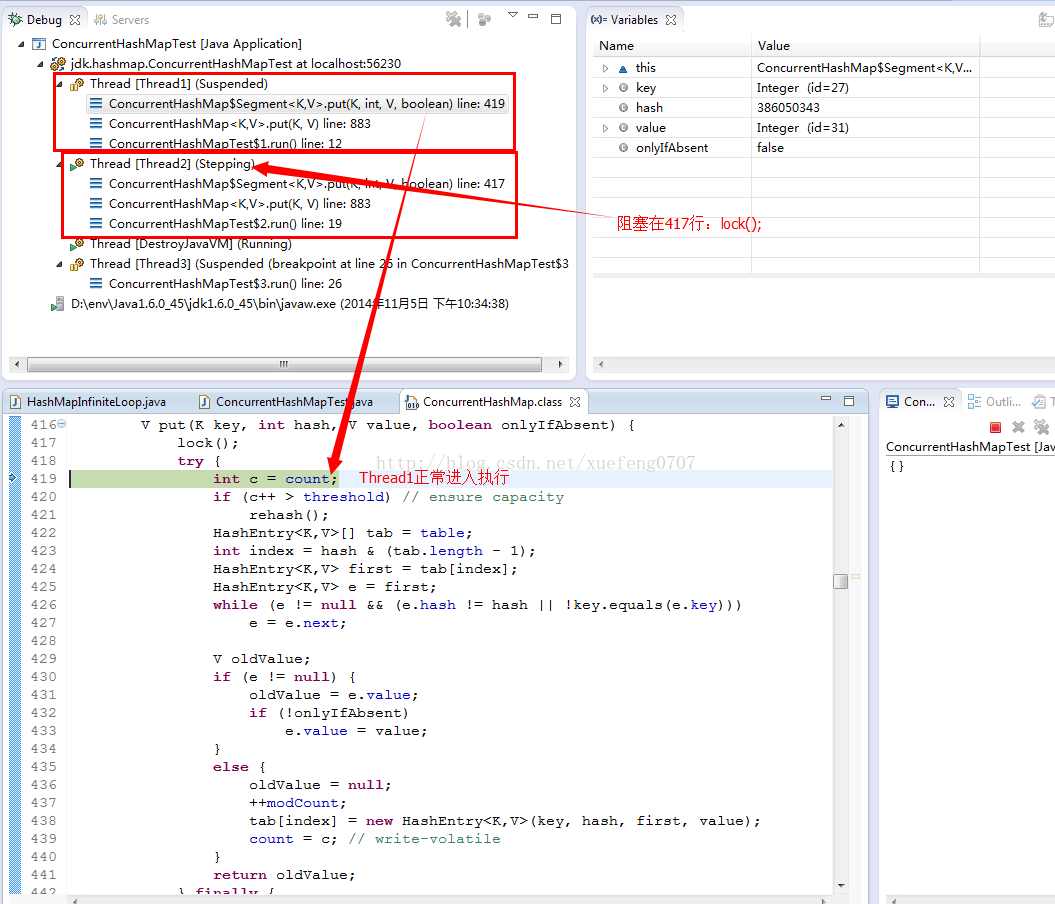
**[java]** [view plain](http://blog.csdn.net/xuefeng0707/article/details/40834595) [copy](http://blog.csdn.net/xuefeng0707/article/details/40834595)

1. **import** java.util.concurrent.ConcurrentHashMap;
3. **public** **class** ConcurrentHashMapTest {
5. **private** **static** ConcurrentHashMap<Integer, Integer> map = **new** ConcurrentHashMap<Integer, Integer>();
6. **public** **static** **void** main(String[] args) {
7. **new** Thread("Thread1"){
8. @Override
9. **public** **void** run() {
10. map.put(3, 33);
11. }
12. };
14. **new** Thread("Thread2"){
15. @Override
16. **public** **void** run() {
17. map.put(4, 44);
18. }
19. };
21. **new** Thread("Thread3"){
22. @Override
23. **public** **void** run() {
24. map.put(7, 77);
25. }
26. };
27. System.out.println(map);
28. }
29. }

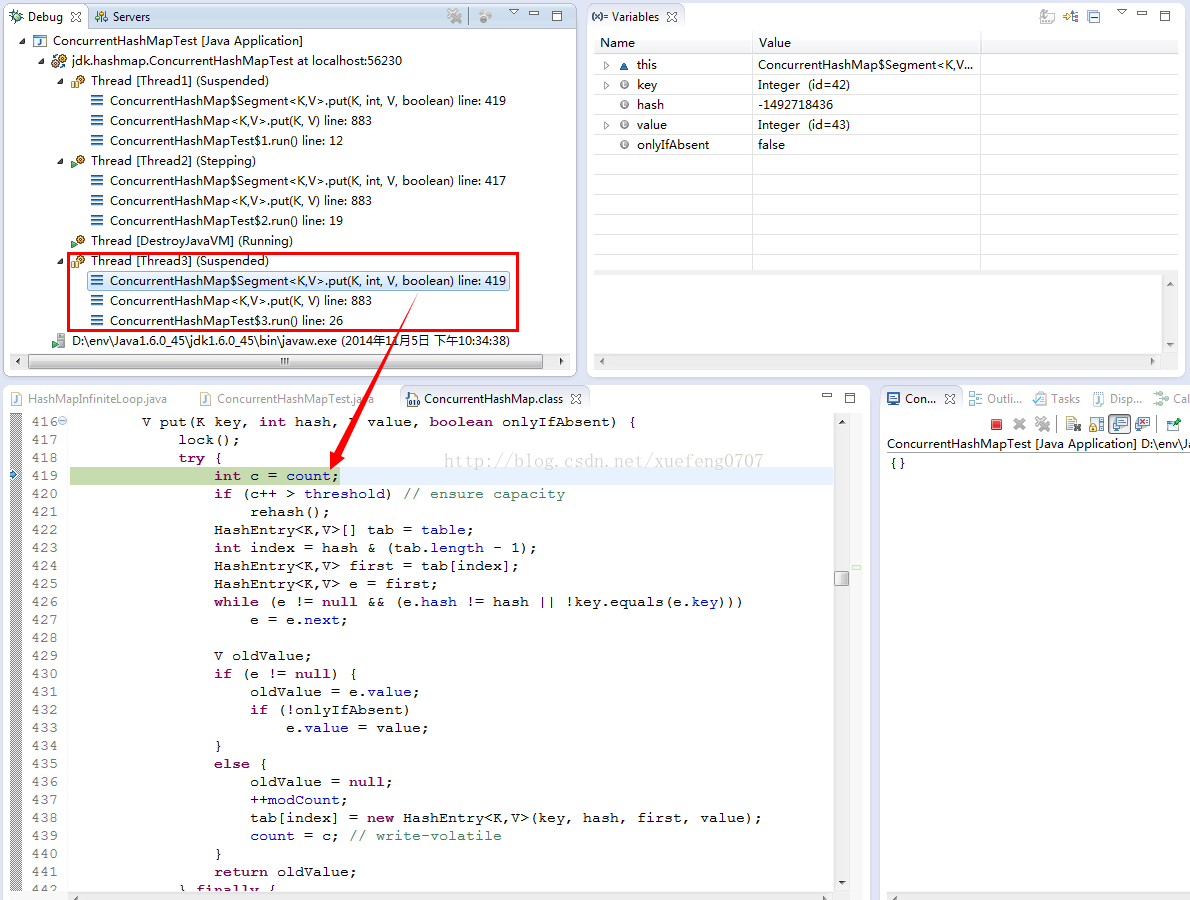
ConcurrentHashMap中默认是把segments初始化为长度为16的数组。

根据ConcurrentHashMap.segmentFor的算法，3、4对应的Segment都是segments[1]，7对应的Segment是segments[12]。

（1）Thread1和Thread2先后进入Segment.put方法时，Thread1会首先获取到锁，可以进入，而Thread2则会阻塞在锁上：



（2）切换到Thread3，也走到Segment.put方法，因为7所存储的Segment和3、4不同，因此，不会阻塞在lock()：



以上就是ConcurrentHashMap的工作机制，通过把整个Map分为N个Segment（类似HashTable），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。