### Java8

1. 新特性：速度更快、代码更少、强大的StreamAPI、便于并行、最大化减少空指针异常 Optional。

* 对底层数据结构进行了更改：

hashMap：采用hash算法，加速了重复key的比较速度。

链表添加时，jdk1.7采用的是头插法，jdk1.8使用的是尾插法。

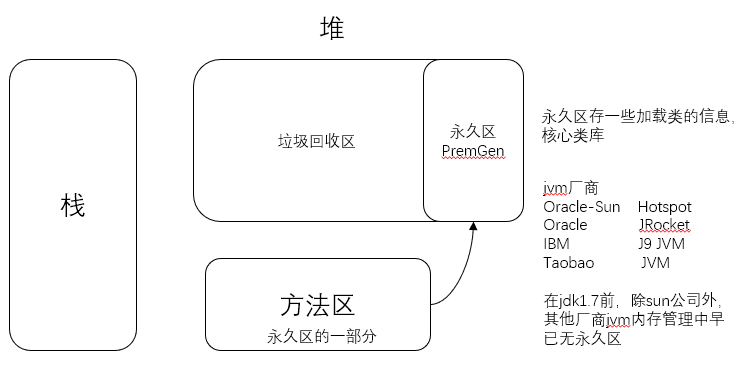
碰撞次数大于8，且总容量大于64时，该链表就转换为红黑树。

jdk1.8扩容时，等于原来哈希表的总长度 + 原来的位置。

ConcurrentHashMap：jdk1.7采用锁分段机制，并发级别concurretLevel默认为16；

jdk1.8采用CAS无锁算法

* 对底层内存分配的变化：



jdk1.7 栈、堆、方法区（堆中永久区PremGen的一部分）

永久区的内容几乎不会被gc回收，注意不是不会，只是回收条件比较苛刻，只有当方法区快满时，才去回收。

jdk1.8中，Hotspot取消了永久区，原方法区变为了MetaSpace元空间（与之前不同的是，该区使用了物理内存[相比变大了很多]，从而提升了垃圾回收机制的效率，降低了OOM的概率）。

1. Lambda表达式
2. Lambda是一个匿名函数。
3. 箭头操作符“->”，将Lambda表达式拆分为两部分，左侧为Lambda表达式的参数列表，右侧为Lambda表达式中需要执行的功能，即Lambda体。
4. 语法格式：

【jdk1.7及以前局部内部类应用同级别的局部变量时，局部变量必须显示加上final】

a. 无参数无返回值 Runnable r1 = () -> System.***out***.println(**"Hello world"**);

b. 单参无返回值 Consumer<String> con = (x) -> System.***out***.println(x);

若左侧只有一个参数，左侧小括号可以不写。

c. 多参多语句有返回值

|  |
| --- |
| Comparator<Integer> com = (x, y) -> {  System.***out***.println(**"x"**);  **return** Integer.*compare*(x, y); }; |

若Lambda体中只有一条语句，则return和大括号都可以省略。

d. Lamdba表达式的参数列表的数据类型可以省略不写，因为JVM编译器可以通过上下文推断出数据类型，即“类型推断”。【若要写，则所有类型都必须写】

1. Lambda表达式需要一个函数式接口的支持。函数式接口是指接口中只有一个抽象方法的接口，可以使用注解@FunctionalInterface修饰，可以检查是否是函数式接口。
2. 内置的四大核心函数式接口

a. Consumer<T>：消费型接口 void accept(T t);

b. Supplier<T>：供给型接口 T get();

c. Function<T, R>：函数型接口 R apply(T t);

d. Predicate<T>：断言型接口 boolean test(T t);

1. 方法引用：若Lamdba体中的内容有方法已经实现了，我们可以使用“方法引用”。(可以理解为方法引用是Lamdba表达式的另外一种表现形式)

语法格式：

对象::实例方法名 Consumer<String> con1 = System.***out***::println

类::静态方法名 Comparator<Integer> com = Integer::*compare*

类::实例方法名 BiPredicate<String, String> bp = String::equals

【注意：

① **Lambda体中调用方法的参数列表和返回值类型必须与函数式接口中抽象方法的参数列表和返回值类型保持一致**。

② Lambda参数列表中的第一个参数是实例方法的调用者，而第二个参数是实例方法的参数时，可以使用 类::实例方法名】

1. 构造器引用：类名::new

Supplier<Employee> sup = Employee::**new**;

注意：需要调用的构造器的参数列表要与函数式接口中抽象方法的参数列表保持一致。

1. 数组引用：Type[]::new

Function<Integer, String[]> fun = String[]::**new**

### java5

1. volatile关键字：当多个线程进行操作共享数据时，可以保证内存中的数据可见。（底层通过内存栅栏解决实时刷新缓存）

内存可见性问题：当多个线程操作共享数据时，彼此不可见。

同步锁能保证每次都刷新缓存。

* 效率：正常 > volatile（重排序导致）> 同步锁
* volatile与synchronized区别

a. volatile不具备“互斥性”。

b. volatile不能保证变量的“原子性”。

1. 原子性

* i++的原子性问题：

i++的操作实际上分为三步：

|  |
| --- |
| int temp = i;  i = i + 1;  i = temp; |

* 原子变量（java.util.concurrent.atomic）

AtomicBoolean、AtomicInteger、AtomicLong、AtomicIntegerArray、AtomicLongArray

特点：a. 该对象内的变量都用volatile修饰，保证了内存可见性

b. 使用CAS（Compare-And-Swap）算法保证了数据原子性

CAS算法是硬件对于并发操作共享数据的支持。

CAS包含了三个操作数：内存值V、预估值A、更新值B。当且仅当V==A时，V = B。否则将不做任何操作。CAS比同步锁效率高，因为其不会导致阻塞挂起。

1. ConcurrentHashMap：线程安全的HashMap

Hashtable通过一个大锁锁整个表，即将并行转化为串行，导致效率低下。有时候还存在复合操作（若不存在则添加，若存在则删除）上的一些安全问题。

ConcurrentHashMap采用“锁分段”机制，concurrentLevel默认为16，即将Map分为16个段（每个段中都有独立的锁），而每个段中也有16个长度的Hash数组。在jdk1.8中取消了分段锁，全部采用CAS算法。

concurrent包中，当期望许多线程访问一个给定collection时，ConcurrentHashMap通常优于同步的HashMap，ConcurrentSkipListMap通常优于同步的TreeMap。当期望的读数和遍历远远大于列表的更新数据时，CopyOnWriteArrayList优于同步的ArrayList。

1. CountDownLatch闭锁

CountDownLatch是一个同步辅助类，在完成一组正在其他线程中执行的操作之前。他允许一个或多个线程一直等待。