**1 线程安全**

定义

某个类的行为与其规范一致。

不管多个线程是怎样的执行顺序和优先级,或是wait,sleep,join等控制方式,，如果一个类在多线程访问下运转一切正常，并且访问类不需要进行额外的同步处理或者协调，那么我们就认为它是线程安全的。

如何保证线程安全？

对变量使用volitate

对程序段进行加锁(synchronized,lock)

**2 Java实现多线程的方式及三种方式的区别**

实现多线程的方式

继承Thread类，重写run函数。

实现Runnable接口，实现该接口的run（）方法

实现Callable接口

三种方式的区别：

实现Runnable接口可以避免Java单继承特性而带来的局限；增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；适合多个相同程序代码的线程区处理同一资源的情况。

继承Thread类和实现Runnable方法启动线程都是使用start方法，然后JVM虚拟机将此线程放到就绪队列中，如果有处理机可用，则执行run方法。

实现Callable接口要实现call方法，并且线程执行完毕后会有返回值。其他的两种都是重写run方法，没有返回值。

3 Java序列化，如何实现序列化和反序列化？

1. Java 序列化定义

将哪些实现了Serializable接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。

1. Java 序列化的作用
2. Java 远程方法调用
3. 对javabeasn进行序列化

3. 如何实现序列化和反序列化？

只有实现了Serializable和Externalizable接口的类的对象才能被序列化。Externalizable接口继承自 Serializable接口，实现Externalizable接口的类完全由自身来控制序列化的行为，而仅实现Serializable接口的类可以 采用默认的序列化方式 。

4 static 和 final的区别和用途？

Static

修饰变量：静态变量随着类加载时被完成初始化，内存中只有一个，且JVM也只会为它分配一次内存，所有类共享静态变量。

修饰方法：在类加载的时候就存在，不依赖任何实例；**static方法必须实现，不能用abstract修饰。**

修饰代码块：在类加载完之后就会执行代码块中的内容。

父类静态代码块->子类静态代码块->父类非静态代码块->父类构造方法->子类非静态代码块->子类构造方法

Final

修饰变量：

编译期常量： 类加载的过程完成初始化，编译后带入到任何计算式中。只能是基本类型。

运行时常量：**基本数据类型或引用数据类型**。引用不可变，但引用的对象内容可变。

修饰方法：不能被继承，不能被子类修改。

修饰类：不能被继承。

修饰形参：final形参不可变

**5. HashMap的实现原理，HashMap是如何解决hash冲突的问题？**

1. hashmap的工作原理

数组：

**数组具有遍历快，增删慢的特点。**数组在堆中是一块连续的存储空间。（数组遍历的时间复杂度为O(1)，增删的时间复杂度为O(n)）。

链表：

**链表增删快，遍历慢的特点**。链表中各元素的内存空间是不连续的。需要从第一个开始，依次拿到后面元素的地址，进行遍历，直到遍历到第n个元素。（数组遍历的时间复杂度为O(n)，增删的时间复杂度为O(1)）。

HashMap：

**底层维护一个Entry<V,V>的数组**，数组元素Node<K,V>实现了Entry接口。首先有一个每个元素都是链表（可能表述不准确）的数组，当添加一个元素（key-value）时，就首先计算元素key的hash值，以此确定插入数组中的位置，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在数组的同一位置，但是形成了链表，同一各链表上的Hash值是相同的，所以说数组存放的是链表。而当链表长度太长时，链表就转换为红黑树，这样大大提高了查找的效率。

加载因子（默认0.75）：为什么需要使用加载因子，为什么需要扩容呢？因为如果填充比很大，说明利用的空间很多，如果一直不进行扩容的话，链表就会越来越长，这样查找的效率很低，因为链表的长度很大（当然最新版本使用了红黑树后会改进很多），扩容之后，将原来链表数组的每一个链表分成奇偶两个子链表分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，增加查找效率。

get(key)方法时获取key的hash值，计算hash&(n-1)得到在链表数组中的位置first=tab[hash&(n-1)],先判断first的key是否与参数key相等，不等就遍历后面的链表找到相同的key值返回对应的Value值即可

下面简单说下添加键值对put(key,value)的过程：

1，判断键值对数组tab[]是否为空或为null，否则以默认大小resize()；

2，根据键值key计算hash值得到插入的数组索引i，如果tab[i]==null，直接新建节点添加，否则转入3

3，判断当前数组中处理hash冲突的方式为链表还是红黑树(check第一个节点类型即可),分别处理

构造hash表时，如果不指明初始大小，默认大小为16（即Node数组大小16），如果Node[]数组中的元素达到（填充比\*Node.length）重新调整HashMap大小 变为原来2倍大小,扩容很耗时。

**在jdk1.7中hashmap扩容：多线程下的死循环和丢失。**

因为多线程会导致HashMap的Entry链表形成环形数据结构。

Resize()方法中有transfer方法（把旧表中的元素添加到新表中），这是引起死循环的根本原因所在。那么在多个线程同时扩容的情况下就可能导致产生一个存在闭环的单链表

**Hash解决冲突的方法？**

1. **开放定址法**

这种方法也称再散列法，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。

1. **再哈希法**

这种方法是同时构造多个不同的哈希函数：

Hi=RH1（key） i=1，2，…，k

当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

1. **链地址法**

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

（4）**建立公共溢出区**

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

6. Hash Map和Hash Table的区别，Hash Map中的key可以是任何对象或数据类型吗？HashTable是线程安全的么？

Hashtable的方法是同步的，HashMap未经同步，所以在多线程场合要手动同步HashMap这个区别就像Vector和ArrayList一样。

Hashtable不允许 null 值(key 和 value 都不可以)，HashMap允许 null 值(key和value都可以)。

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。

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

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

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。

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。

1. **Hash Map中的key可以是任何对象或数据类型吗**

**可以为null，但不能是可变对象，如果是可变对象的话，对象中的属性改变，则对象HashCode也进行相应的改变，导致下次无法查找到已存在Map中的数据。**

**如果可变对象在HashMap中被用作键，那就要小心在改变对象状态的时候，不要改变它的哈希值了。我们只需要保证成员变量的改变能保证该对象的哈希值不变即可。**

**7.Concurrent HashMap 线程安全吗， Concurrent HashMap如何保证 线程安全？**

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是**ConcurrentHashMap所使用的锁分段技术**，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。

**get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空的才会加锁重读**。get方法里将要使用的共享变量都定义成volatile，**如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量**，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），**在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。**

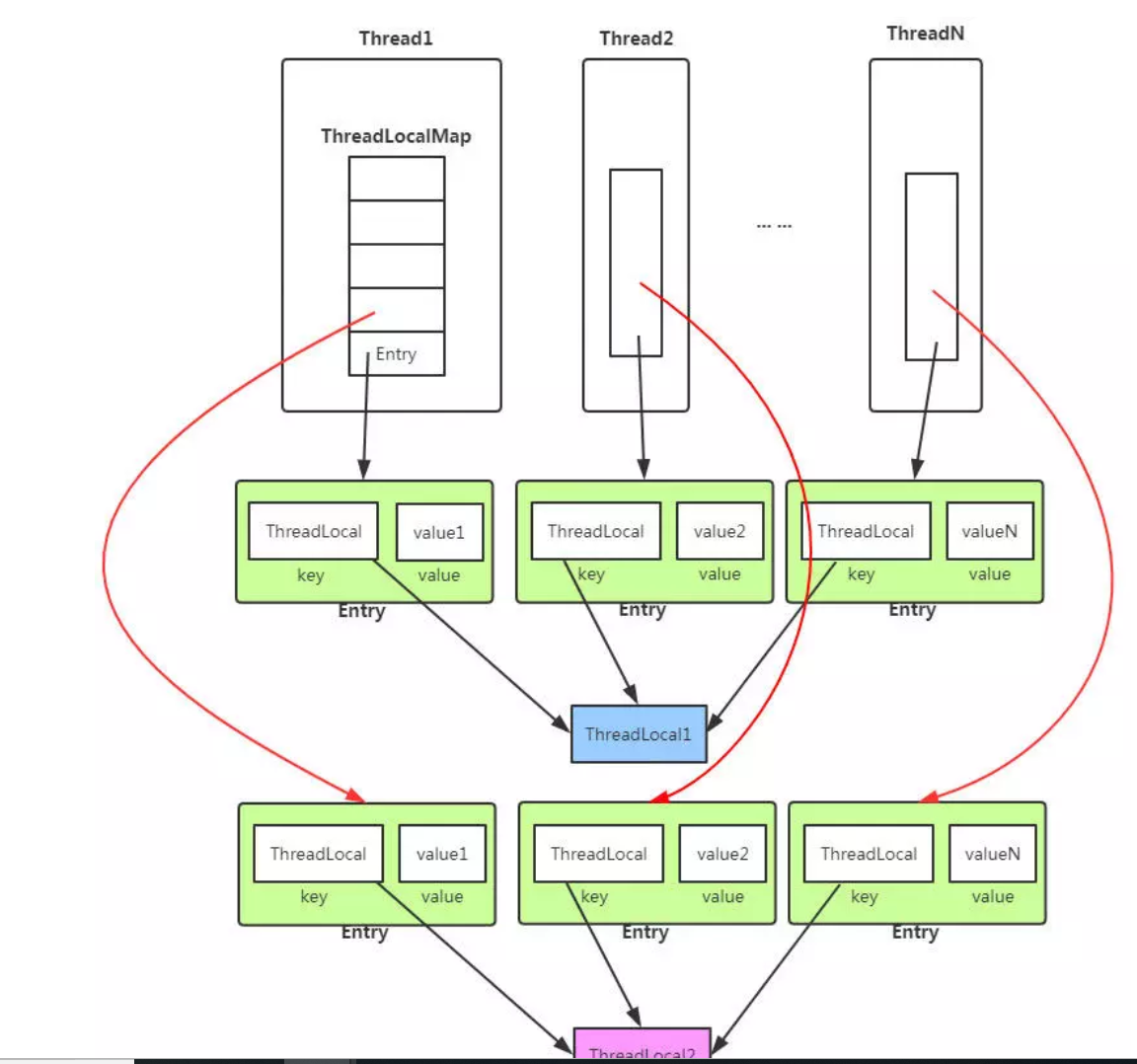
Put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行**扩容**，第二步定位添加元素的位置**然后**放在HashEntry数组里。

然而在jdk1.8中的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。

**改进一：取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。**

**10. ThreadLocal（补充）**

Threadlocal 是本地线程副本工具类，主要用于将私有线程和该线程存访的副本对象做一个映射，各个线程之间的变量互补干扰。在高并发场景下，可以实现无状态的调用，特别适用于各个线程依赖不通的变量值完成操作的场景。



**11. concurrent包下面，都用过什么？**

concurrent下面的包

Executor 用来创建线程池，在实现Callable接口时，添加线程。

FeatureTask 此 FutureTask 的 get 方法所返回的结果类型。

TimeUnit

Semaphore

LinkedBlockingQueue

所用过的类

Executor

**12 volatile和synchronized区别**

volatile不会进行加锁操作：

**volatile变量是一种稍弱的同步机制在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比synchronized关键字更轻量级的同步机制。**

volatile 变量作用类似于同步变量读写操作：

从内存可见性的角度看，写入volatile变量相当于退出同步代码块，而读取volatile变量相当于进入同步代码块。

volatile 不如 synchronized安全：

在代码中如果过度依赖volatile变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当volatile变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。

volatile 无法同时保证内存可见性和原则性：

加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而volatile变量只能确保可见性，原因是声明为volatile的简单变量如果当前值与该变量以前的值相关，那么volatile关键字不起作用，也就是说如下的表达式都不是原子操作：“count++”、“count = count+1”。

**13 sleep和wait分别是那个类的方法，有什么区别**

sleep和wait

sleep是Thread类的方法

wait是Object类的方法

有什么区别

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态）。

wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

**14 常见异常分为那两种(Exception，Error)，常见异常的基类以及常见的异常**

Throwable是java语言中所有错误和异常的超类（万物即可抛）。它有两个子类：Error、Exception。

异常种类

Error：Error为错误，是程序无法处理的，如OutOfMemoryError、ThreadDeath等，出现这种情况你唯一能做的就是听之任之，交由JVM来处理，不过JVM在大多数情况下会选择终止线程。

Exception：Exception是程序可以处理的异常。它又分为两种CheckedException（受捡异常），一种是UncheckedException（不受检异常）。

CheckException发生在**编译阶段**，必须要使用**try…catch**（或者throws）否则编译不通过。

UncheckedException**发生在运行期，具有不确定性，主要是由于程序的逻辑问题所引起的**，难以排查，我们一般都需要纵观全局才能够发现这类的异常错误，所以在程序设计中我们需要认真考虑，好好写代码，尽量处理异常，即使产生了异常，也能尽量保证程序朝着有利方向发展。

常见异常的基类

IOException

RuntimeException