**Day02**

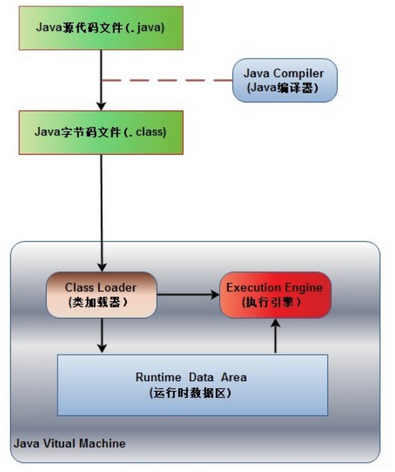
# 课程目标

* Java内存模型
* 线程特性
* 多线程控制
* 容器
* 线程池

# Java内存模型

Java内存模型是每个java程序员必须掌握理解的，这是Java的核心基础，对我们编写代码特别是并发编程时有很大帮助。由于Java程序是交由JVM执行的，所以我们在谈Java内存区域划分的时候事实上是指JVM内存区域划分。

## Java程序执行流程回顾



如图所示

首先Java源代码文件(.java后缀)会被Java编译器编译为字节码文件(.class后缀)，

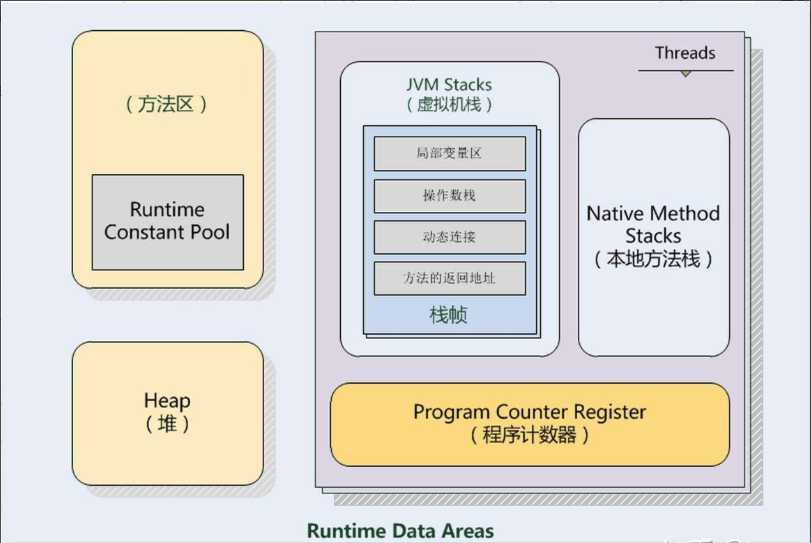
然后由JVM中的类加载器加载各个类的字节码文件，

加载完毕之后，交由JVM执行引擎执行。

Java内存模型指的就是Runtime Data Area（运行时数据区），即程序执行期间用到的数据和相关信息保存区。

## Java内存模型

根据 JVM 规范，JVM 内存共分为虚拟机栈、堆、方法区、程序计数器、本地方法栈五个部分。结构如下图：

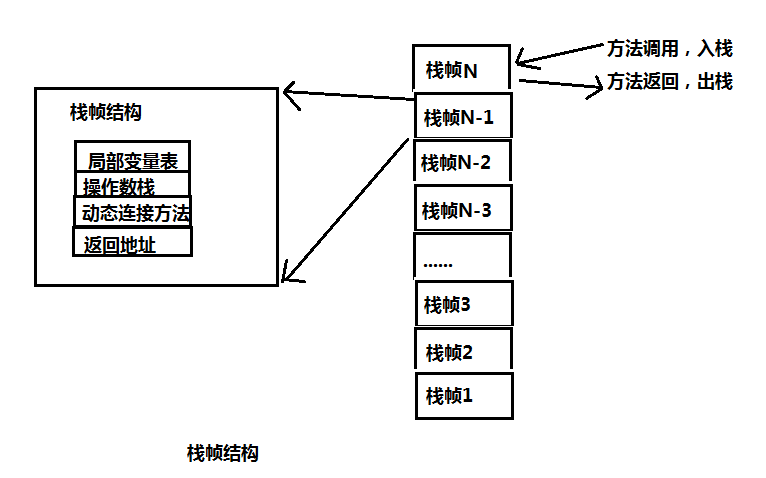


### ****PC程序计数器：****

* 每个线程对应有一个程序计数器。
* 各线程的程序计数器是线程私有的，互不影响，是线程安全的。
* 程序计数器记录线程正在执行的内存地址，以便被中断线程恢复执行时再次按照中断时的指令地址继续执行

### Java栈JavaStack（虚拟机栈JVM Stack）：

* 每个线程会对应一个Java栈；
* 每个Java栈由若干栈帧组成；
* 每个方法对应一个栈帧；
* 栈帧在方法运行时，创建并入栈；方法执行完，该栈帧弹出栈帧中的元素作为该方法返回值，该栈帧被清除；
* 栈顶的栈帧叫活动栈，表示当前执行的方法，才可以被CPU执行；
* 线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；
* 栈扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常；



### 方法区MethodArea

* 方法区是Java堆的永久区（PermanetGeneration）
* 方法区存放了要加载的类的信息（名称、修饰符等）、类中的静态常量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，
* 方法区是被Java线程共享的
* 方法区要使用的内存超过其允许的大小时，会抛出OutOfMemoryError: PremGen space的错误信息。

### 常量池ConstantPool:

* 常量池是方法区的一部分。
* 常量池中存储两类数据：字面量和引用量。

字面量：字符串、final变量等。

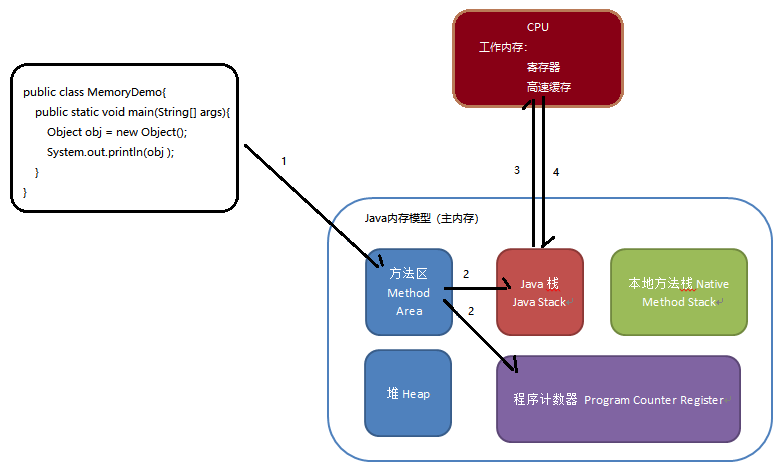
引用量：类/接口、方法和字段的名称和描述符，

* 常量池在编译期间就被确定，并保存在已编译的.class文件中

### 本地方法栈Native Method Stack:

* 本地方法栈和Java栈所发挥的作用非常相似，区别不过是Java栈为JVM执行Java方法服务，而本地方法栈为JVM执行Native方法服务。
* 本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。

## Java内存模型工作示意图



1. 首先类加载器将Java代码加载到方法区
2. 然后执行引擎从方法区找到main方法
3. 为方法创建栈帧放入方法栈，同时创建该栈帧的程序计数器
4. 执行引擎请求CPU执行该方法
5. CPU将方法栈数据加载到工作内存（寄存器和高速缓存），执行该方法
6. CPU执行完之后将执行结果从工作内存同步到主内存

线程计算的时候，原始的数据来自内存，在计算过程中，有些数据可能被频繁读取，这些数据被存储在寄存器和高速缓存中，当线程计算完后，这些缓存的数据在适当的时候应该写回内存。

当个多个线程同时读写某个内存数据时，就会产生多线程并发问题，要解决这些问题就涉及到多线程编程三个特性：原子性，有序性，可见性。

# 多线程特性

多线程编程要保证满足三个特性：原子性、可见性、有序性。

## 原子性

原子性，即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

## 可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。显然，对于单线程来说，可见性问题是不存在的。

## 有序性

有序性即程序执行的顺序按照代码的先后顺序执行。

# 多线程控制类

为了保证多线程的三个特性，Java引入了很多线程控制机制，下面介绍其中常用的几种：

* ThreadLocal
* 原子类
* Lock类
* Volatile关键字

## ThreadLocal

### 作用

ThreadLocal提供线程局部变量，即为使用相同变量的每一个线程维护一个该变量的副本。

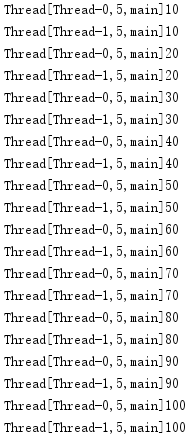
当某些数据是以线程为作用域并且不同线程具有不同的数据副本的时候，就可以考虑采用ThreadLocal，比如数据库连接Connection，每个请求处理线程都需要，但又不相互影响，就是用ThreadLocal实现。

### 示例

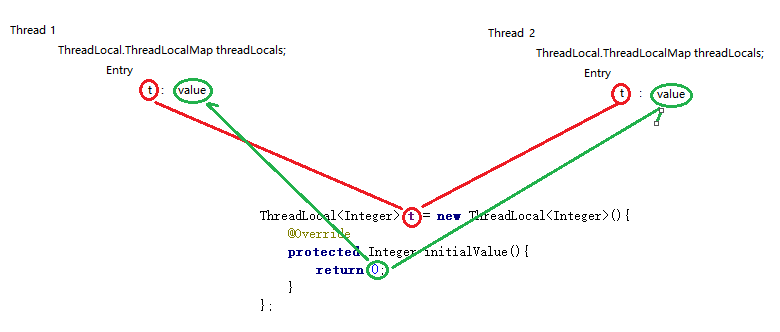
两个线程分别转账

**package** com.multithread.thread;  
  
**public class** Bank {  
 ThreadLocal<Integer> **t** = **new** ThreadLocal<Integer>(){  
 @Override  
 **protected** Integer initialValue(){  
 **return** 0;  
 }  
 };  
 **public** Integer get(){  
 **return t**.get();  
 }  
 **public void** set(){  
 **t**.set(**t**.get()+10);  
 }  
   
 **public static void** main(String[] args){  
 Bank bank = **new** Bank();  
 Transfer transfer = **new** Transfer(bank);  
 Thread t1 = **new** Thread(transfer);  
 Thread t2 = **new** Thread(transfer);  
 t1.start();  
 t2.start();  
 }  
}  
  
**class** Transfer **implements** Runnable{  
 Bank **bank**;  
 **public** Transfer(Bank bank){  
 **this**.**bank** = bank;  
 }  
 **public void** run() {  
 **for** (**int** i=0;i<10;i++){  
 **bank**.set();  
 System.***out***.println(Thread.*currentThread*()+**""**+**bank**.get());  
 }  
 }  
}

打印结果：



### 分析



* 在ThreadLocal类中定义了一个ThreadLocalMap，
* 每一个Thread都有一个ThreadLocalMap类型的变量threadLocals
* threadLocals内部有一个Entry，Entry的key是ThreadLocal对象实例，value就是共享变量副本
* ThreadLocal的get方法就是根据ThreadLocal对象实例获取共享变量副本
* ThreadLocal的set方法就是根据ThreadLocal对象实例保存共享变量副本

## 原子类

Java的java.util.concurrent.atomic包里面提供了很多可以进行原子操作的类，分为以下四类：

* 原子更新基本类型：AtomicInteger、AtomicBoolean、AtomicLong
* 原子更新数组：AtomicIntegerArray、AtomicLongArray
* 原子更新引用：AtomicReference、AtomicStampedReference等
* 原子更新属性：AtomicIntegerFieldUpdater、AtomicLongFieldUpdater

提供这些原子类的目的就是为了解决基本类型操作的非原子性导致在多线程并发情况下引发的问题。

### 非原子性操作问题演0示

非原子性的操作会引发什么问题呢？下面以i++为例演示非原子性操作问题。

i++并不是原子操作，而是由三个操作构成：

tp1 = i;

tp2 = tp1+1;

i = tp2;

所以单线程i的值不是有问题，但多线程下就会出错，多线程示例代码如下：

**package** com.multithread.thread;  
  
**import** java.util.concurrent.atomic.AtomicInteger;  
  
**public class** AtomicClass {  
 **static int** *n* = 0;  
 **public static void** main(String[] args) **throws** InterruptedException {  
 **int** j = 0;  
 **while**(j<100){  
 *n* = 0;  
 Thread t1 = **new** Thread(){  
 **public void** run(){  
 **for**(**int** i=0; i<1000; i++){  
 *n*++;  
 }  
 }  
 };  
 Thread t2 = **new** Thread(){  
 **public void** run(){  
 **for**(**int** i=0; i<1000; i++){  
 *n*++;  
 }  
 }  
 };  
 t1.start();  
 t2.start();  
 t1.join();  
 t2.join();  
 System.***out***.println(**"n的最终值是："**+*n*);  
 j++;  
 }  
  
 }  
}

执行结果如下：发现n的最终值可能不是2000



### 原子类解决非原子性操作问题

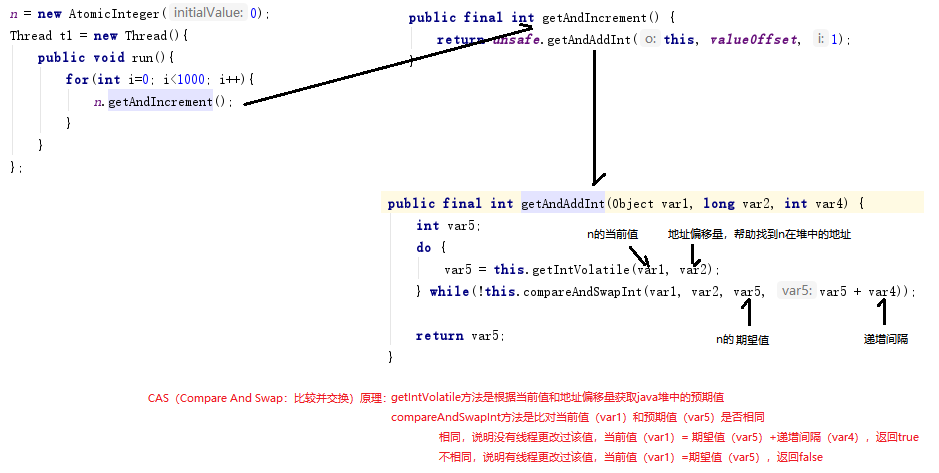
以上代码修改如下：

**package** com.multithread.thread;  
  
**import** java.util.concurrent.atomic.AtomicInteger;  
  
**public class** AtomicClass {  
 **static** AtomicInteger *n*;  
 **public static void** main(String[] args) **throws** InterruptedException {  
 **int** j = 0;  
 **while**(j<100){  
 *n* = **new** AtomicInteger(0);  
 Thread t1 = **new** Thread(){  
 **public void** run(){  
 **for**(**int** i=0; i<1000; i++){  
 *n*.getAndIncrement();  
 }  
 }  
 };  
 Thread t2 = **new** Thread(){  
 **public void** run(){  
 **for**(**int** i=0; i<1000; i++){  
 *n*.getAndIncrement();  
 }  
 }  
 };  
 t1.start();  
 t2.start();  
 t1.join();  
 t2.join();  
 System.***out***.println(**"n的最终值是："**+*n*);  
 j++;  
 }  
  
 }  
}

执行结果如下：n的值永远是2000



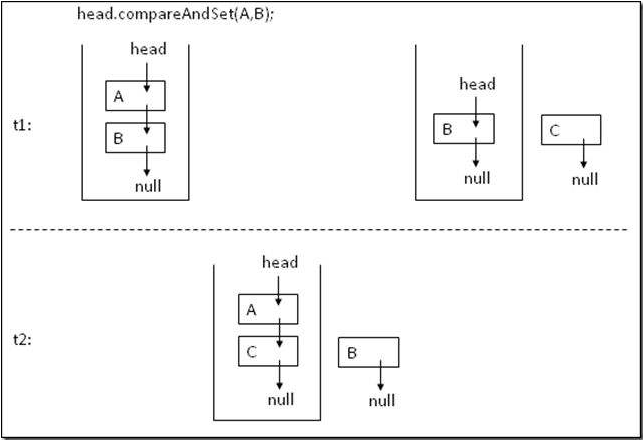
### 原子类CAS原理分析



### CAS的ABA问题及解决

#### ABA问题分析

当前内存的值一开始是A，被另外一个线程先改为B然后再改为A，那么当前线程访问的时候发现是A，则认为它没有被其他线程访问过。在某些场景下这样是存在错误风险的。如下图：



#### ABA问题解决

**package** com.multithread.thread;  
  
**import** java.util.concurrent.atomic.AtomicInteger;  
**import** java.util.concurrent.atomic.AtomicStampedReference;  
  
**public class** AtomicClass {  
 **static** AtomicStampedReference<Integer> *n*;  
 **public static void** main(String[] args) **throws** InterruptedException {  
 **int** j = 0;  
 **while**(j<100){  
 *n* = **new** AtomicStampedReference<Integer>(0,0);  
 Thread t1 = **new** Thread(){  
 **public void** run(){  
 **for**(**int** i=0; i<1000; i++){  
 **int** stamp;  
 Integer reference;  
 **do**{  
 stamp = *n*.getStamp();  
 reference = *n*.getReference();  
 } **while**(!*n*.compareAndSet(reference, reference+1, stamp, stamp+1));  
 }  
 }  
 };  
 Thread t2 = **new** Thread(){  
 **public void** run(){  
 **for**(**int** i=0; i<1000; i++){  
 **int** stamp;  
 Integer reference;  
 **do**{  
 stamp = *n*.getStamp();  
 reference = *n*.getReference();  
  
 } **while**(!*n*.compareAndSet(reference, reference+1, stamp, stamp+1));  
 }  
 }  
 };  
 t1.start();  
 t2.start();  
 t1.join();  
 t2.join();  
 System.***out***.println(**"n的最终值是："**+*n*.getReference());  
 j++;  
 }  
  
 }  
}

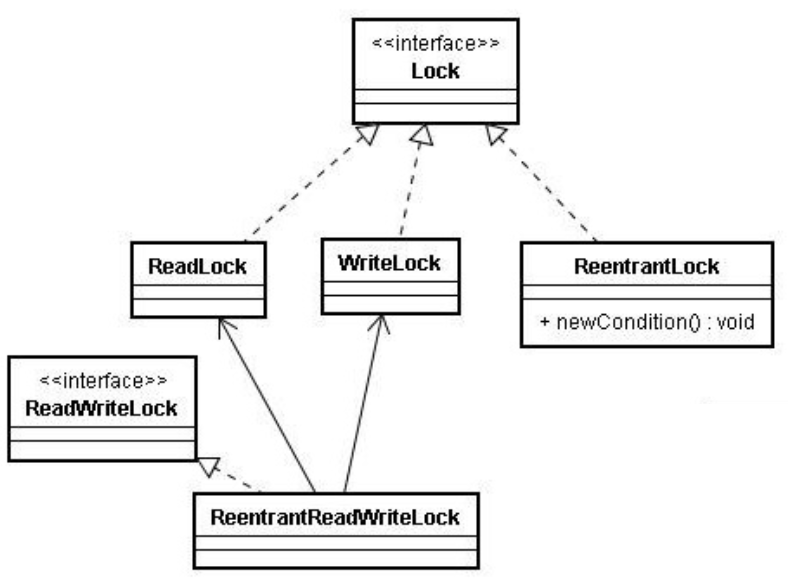
执行效果如下：执行结果也是2000



注意：采用AtomicStampedReference会降低性能，慎用。

## Lock类

### Lock接口关系图



Lock和ReadWriteLock是两大锁的根接口

Lock 接口支持重入、公平等的锁规则：实现类 ReentrantLock、ReadLock和WriteLock。  
ReadWriteLock 接口定义读取者共享而写入者独占的锁，实现类：ReentrantReadWriteLock。

### 可重入锁

不可重入锁，即线程请求它已经拥有的锁时会阻塞。

可重入锁，即线程可以进入它已经拥有的锁的同步代码块。

public class ReentrantLockTest {

public static void main(String[] args) throws InterruptedException {

ReentrantLock lock = new ReentrantLock();

for (int i = 1; i <= 3; i++) {

lock.lock();

}

for(int i=1;i<=3;i++){

try {

} finally {

lock.unlock();

}

}

}

}

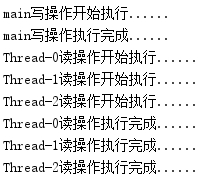
### 读写锁

读写锁，即可以同时读，读的时候不能写；不能同时写，写的时候不能读。

示例代码：

**package** com.multithread.thread;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReadWriteLock;  
**import** java.util.concurrent.locks.ReentrantReadWriteLock;  
  
*/\*\*  
 \* 读写操作类  
 \*/***public class** ReadWriteLockDemo {  
 **private** Map<String, Object> **map** = **new** HashMap<String, Object>();  
 *//创建一个读写锁实例* **private** ReadWriteLock **rw** = **new** ReentrantReadWriteLock();  
 *//创建一个读锁* **private** Lock **r** = **rw**.readLock();  
 *//创建一个写锁* **private** Lock **w** = **rw**.writeLock();  
  
 */\*\*  
 \* 读操作  
 \*  
 \** ***@param key*** *\** ***@return*** *\*/* **public** Object get(String key) {  
 **r**.lock();  
 System.***out***.println(Thread.*currentThread*().getName() + **"读操作开始执行......"**);  
 **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **try** {  
 **return map**.get(key);  
 } **finally** {  
 **r**.unlock();  
 System.***out***.println(Thread.*currentThread*().getName() + **"读操作执行完成......"**);  
 }  
 }  
  
 */\*\*  
 \* 写操作  
 \*  
 \** ***@param key*** *\** ***@param value*** *\*/* **public void** put(String key, Object value) {  
 **try** {  
 **w**.lock();  
 System.***out***.println(Thread.*currentThread*().getName() + **"写操作开始执行......"**);  
 **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **map**.put(key, value);  
 } **finally** {  
 **w**.unlock();  
 System.***out***.println(Thread.*currentThread*().getName() + **"写操作执行完成......"**);  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **final** ReadWriteLockDemo d = **new** ReadWriteLockDemo();  
 d.put(**"key1"**, **"value1"**);  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 d.get(**"key1"**);  
 }  
 }).start();  
  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 d.get(**"key1"**);  
 }  
 }).start();  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 d.get(**"key1"**);  
 }  
 }).start();  
 }  
  
}

执行效果如下：写操作为独占锁，执行期间不能读；读操作可



## Volatile关键字

### 作用

一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

* 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。（注意：不保证原子性）
* 禁止进行指令重排序。（保证变量所在行的有序性）

当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

### 应用场景

基于volatile的作用，使用volatile必须满足以下两个条件：

* 对变量的写操作不依赖于当前值
* 该变量没有包含在具有其他变量的不变式中

常见应用场景如下：

状态量标记：

volatile boolean flag = false;

while(!flag){

    doSomething();

}

public void setFlag() {

    flag = true;

}

volatile boolean inited = false;

//线程1:

context = loadContext();

inited = true;

//线程2:

while(!inited ){

sleep()

}

doSomethingwithconfig(context);

双重校验：

class Singleton{

    private volatile static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {

        if(instance==null) {

            synchronized (Singleton.class) {

                if(instance==null)

                    instance = new Singleton();

            }

        }

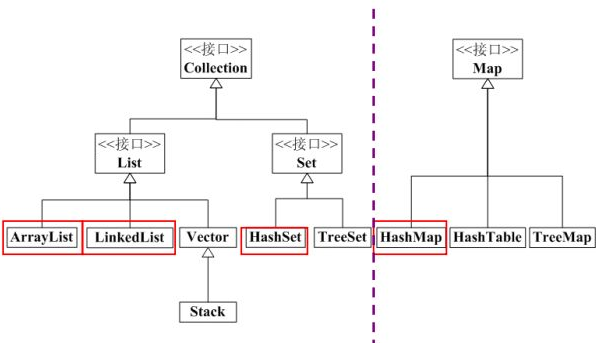
        return instance;

    }

}

# 容器

## 容器类关系图



Collection          接口的接口   对象的集合   
├ List                   子接口      按进入先后有序保存   可重复   
│├ LinkedList                接口实现类   链表   插入删除   没有同步   线程不安全   
│├ ArrayList                  接口实现类   数组   随机访问   没有同步   线程不安全   
│└ Vector                      接口实现类   数组                  同步        线程安全   
│ 　 └ Stack  
└ Set                   子接口 不可重复

├ HashSet

│ 　 └ LinkedHashSet  
└ TreeSet

Map                接口      键值对的集合   
├ Hashtable                  接口实现类         同步           线程安全   
├ HashMap                   接口实现类         没有同步    线程不安全

│├ LinkedHashMap

│└ WeakHashMap

└ TreeMap

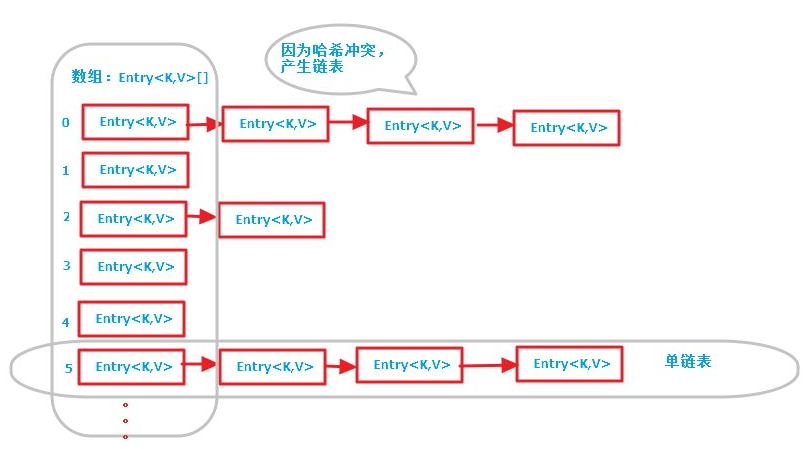
## HashMap实现分析

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。

数组：存储区间连续，占用内存严重，寻址容易，插入删除困难；

链表：存储区间离散，占用内存比较宽松，寻址困难，插入删除容易；  
HashMap综合应用了这两种数据结构，实现了寻址容易，插入删除也容易。

HashMap结构示意图如下：



1.8的HashMap做了很大的变化，故分开分析

### JDK1.8之前并发问题

1. 在hashmap做put操作的时候会调用下面方法：

// 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。

void addEntry(int hash, K key, V value, int bucketIndex) {

// 保存“bucketIndex”位置的值到“e”中

Entry<K,V> e = table[bucketIndex];

// 设置“bucketIndex”位置的元素为“新Entry”，

// 设置“e”为“新Entry的下一个节点”

table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

// 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小

if (size++ >= threshold)

resize(2 \* table.length);

}

 在hashmap做put操作的时候会调用到以上的方法。现在假如A线程和B线程同时对同一个数组位置调用addEntry，两个线程会同时得到现在的头结点，然后A写入新的头结点之后，B也写入新的头结点，那B的写入操作就会覆盖A的写入操作造成A的写入操作丢失

1. 删除键值对会调用以下代码

final Entry<K,V> removeEntryForKey(Object key) {

// 获取哈希值。若key为null，则哈希值为0；否则调用hash()进行计算

int hash = (key == null) ? 0 : hash(key.hashCode());

int i = indexFor(hash, table.length);

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

// 删除链表中“键为key”的元素

// 本质是“删除单向链表中的节点”

while (e != null) {

Entry<K,V> next = e.next;

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next;

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

当多个线程同时操作同一个数组位置的时候，也都会先取得现在状态下该位置存储的头结点，然后各自去进行计算操作，之后再把结果写会到该数组位置去，其实写回的时候可能其他的线程已经就把这个位置给修改过了，就会覆盖其他线程的修改。

1. addEntry中当加入新的键值对后键值对总数量超过门限值的时候会调用一个resize操作，代码如下：

// 重新调整HashMap的大小，newCapacity是调整后的容量

void resize(int newCapacity) {

Entry[] oldTable = table;

int oldCapacity = oldTable.length;

//如果就容量已经达到了最大值，则不能再扩容，直接返回

if (oldCapacity == MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return;

}

// 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，

// 然后，将“新HashMap”赋值给“旧HashMap”。

Entry[] newTable = new Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (int)(newCapacity \* loadFactor);

}

 这个操作会新生成一个新的容量的数组，然后对原数组的所有键值对重新进行计算和写入新的数组，之后指向新生成的数组。当多个线程同时检测到总数量超过门限值的时候就会同时调用resize操作，各自生成新的数组并rehash后赋给该map底层的数组table，结果最终只有最后一个线程生成的新数组被赋给table变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的table作为原始数组，这样也会有问题。

### JDK1.8并发问题

HashMap中的迭代器源码：

**abstract class** HashIterator {  
 Node<K,V> **next**; *// next entry to return* Node<K,V> **current**; *// current entry* **int expectedModCount**; *// for fast-fail* **int index**; *// current slot* HashIterator() {  
 **expectedModCount** = **modCount**;  
 Node<K,V>[] t = **table**;  
 **current** = **next** = **null**;  
 **index** = 0;  
 **if** (t != **null** && **size** > 0) { *// advance to first entry* **do** {} **while** (**index** < t.**length** && (**next** = t[**index**++]) == **null**);  
 }  
 }  
  
 **public final boolean** hasNext() {  
 **return next** != **null**;  
 }  
  
 **final** Node<K,V> nextNode() {  
 Node<K,V>[] t;  
 Node<K,V> e = **next**;  
 **if** (**modCount** != **expectedModCount**)  
 **throw new** ConcurrentModificationException();  
 **if** (e == **null**)  
 **throw new** NoSuchElementException();  
 **if** ((**next** = (**current** = e).**next**) == **null** && (t = **table**) != **null**) {  
 **do** {} **while** (**index** < t.**length** && (**next** = t[**index**++]) == **null**);  
 }  
 **return** e;  
 }  
  
 **public final void** remove() {  
 Node<K,V> p = **current**;  
 **if** (p == **null**)  
 **throw new** IllegalStateException();  
 **if** (**modCount** != **expectedModCount**)  
 **throw new** ConcurrentModificationException();  
 **current** = **null**;  
 K key = p.**key**;  
 removeNode(*hash*(key), key, **null**, **false**, **false**);  
 **expectedModCount** = **modCount**;  
 }  
}

* modCount是hashmap中的成员变量。
* 在调用put(),remove(),clear(),ensureCapacity()这些会修改数据结构的方法中都会使modCount++。
* 在获取迭代器的时候会把modCount赋值给迭代器的expectedModCount变量。此时modCount与expectedModCount肯定相等。
* 在迭代元素的过程中如果hashmap调用自身方法使集合发生变化，那么modCount肯定会变，此时modCount与expectedModCount肯定会不相等。
* 在迭代过程中，只要发现modCount!=expectedModCount，则说明结构发生了变化也就没有必要继续迭代元素了。此时会抛出ConcurrentModificationException，终止迭代操作。

## HashMap并发问题解决方案

HashMap并发问题解决方案有如下几种：

* Synchronized关键字
* Lock锁
* 同步类容器
* 并发类容器

# 同步容器

## 同步容器介绍

在Java中，同步容器主要包括2类：

1）Vector、Stack、HashTable（可以独立创建）

2）Collections类中提供的静态工厂方法创建的类（借助工具类创建）

Vector

实现了List接口，Vector实际上就是一个数组，和ArrayList类似，但是Vector中的方法都是synchronized方法，即进行了同步措施。

Stack

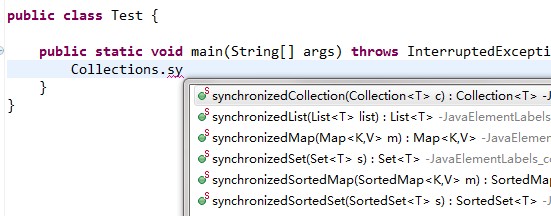
也是一个同步容器，它的方法也用synchronized进行了同步，它实际上是继承于Vector类。

HashTable

实现了Map接口，它和HashMap很相似，但是HashTable进行了同步处理，而HashMap没有。

Collections

Collections类是一个工具提供类，注意，它和Collection不同，Collection是一个顶层的接口。在Collections类中提供了大量的方法，比如对集合或者容器进行排序、查找等操作。最重要的是，在它里面提供了几个静态工厂方法来创建同步容器类，如下图所示：



下面以HashTable为例分析同步容器实现原理及特点

## HashTable

Hashtable在jdk1.1就有了，那么它是怎样实现线程安全的呢？主要看put、remove、get方法猜它肯定进行的同步控制的。于是看源码：

//get它搞成了同步方法，保证了get的安全性

**public synchronized** V get(Object key) {  
 ...

}

//put方法同样

**public synchronized** V put(K key, V value) {  
 ...  
}

//也是搞成了同步方法

**public synchronized** V remove(Object key) {  
 ...

}

所以为什么Hashtable是线程安全的，因为它的remove,put，get等public方法做成了同步方法，保证了HashTable的线程安全性。

# 并发容器

## 并发容器简介

因为同步容器将几乎所有方法添加的synchronized进行同步，这样保证了线程的安全性，但代价就是严重降低了并发性能，当多个线程竞争容器时，吞吐量严重降低。

Java5.0开始针对多线程并发访问重新设计，提供了并发性能较好的并发容器，引入了java.util.concurrent包。

并发容器如下：

ConcurrentHashMap

* 对应的非并发容器：HashMap
* 目标：代替Hashtable、synchronizedMap，支持复合操作
* 原理：JDK6中采用一种更加细粒度的加锁机制Segment“分段锁”，JDK8中采用CAS无锁算法。

CopyOnWriteArrayList

* 对应的非并发容器：ArrayList
* 目标：代替Vector、synchronizedList
* 原理：利用高并发往往是读多写少的特性，对读操作不加锁，对写操作，先复制一份新的集合，在新的集合上面修改，然后将新集合赋值给旧的引用，并通过volatile 保证其可见性，当然写操作的锁是必不可少的了。

CopyOnWriteArraySet

* 对应的费并发容器：HashSet
* 目标：代替synchronizedSet
* 原理：基于CopyOnWriteArrayList实现，其唯一的不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法，其遍历当前Object数组，如Object数组中已有了当前元素，则直接返回，如果没有则放入Object数组的尾部，并返回。

ConcurrentSkipListMap

* 对应的非并发容器：TreeMap
* 目标：代替synchronizedSortedMap(TreeMap)
* 原理：Skip list（跳表）是一种可以代替平衡树的数据结构，默认是按照Key值升序的。Skip list让已排序的数据分布在多层链表中，以0-1随机数决定一个数据的向上攀升与否，通过”空间来换取时间”的一个算法。ConcurrentSkipListMap提供了一种线程安全的并发访问的排序映射表。内部是SkipList（跳表）结构实现，在理论上能够在O（log（n））时间内完成查找、插入、删除操作。

ConcurrentSkipListSet

* 对应的非并发容器：TreeSet
* 目标：代替synchronizedSortedSet
* 原理：内部基于ConcurrentSkipListMap实现

ConcurrentLinkedQueue

* 不会阻塞的队列
* 对应的非并发容器：Queue
* 原理：基于链表实现的FIFO队列（LinkedList的并发版本）

LinkedBlockingQueue、ArrayBlockingQueue、PriorityBlockingQueue

* 对应的非并发容器：BlockingQueue
* 特点：拓展了Queue，增加了可阻塞的插入和获取等操作
* 原理：通过ReentrantLock实现线程安全，通过Condition实现阻塞和唤醒
* 实现类：

LinkedBlockingQueue：基于链表实现的可阻塞的FIFO队列

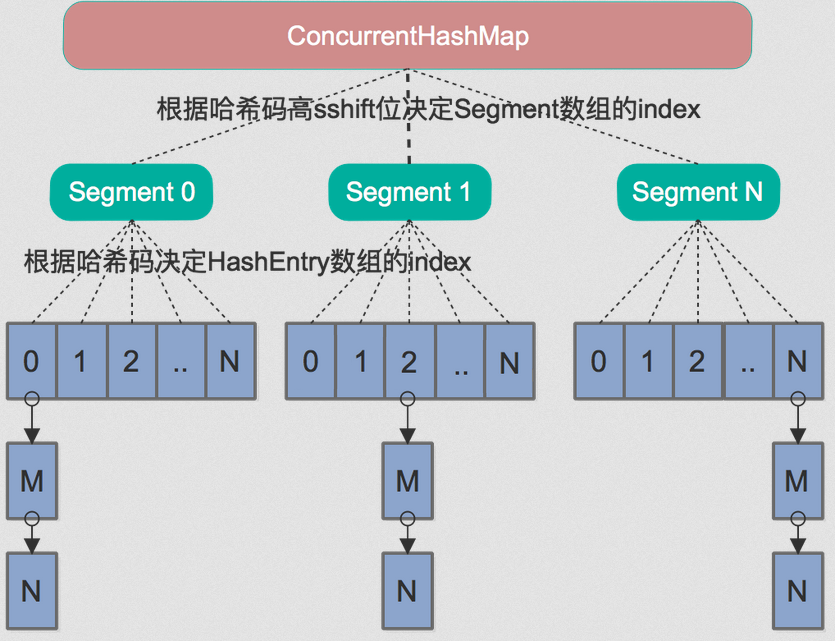
ArrayBlockingQueue：基于数组实现的可阻塞的FIFO队列

PriorityBlockingQueue：按优先级排序的队列

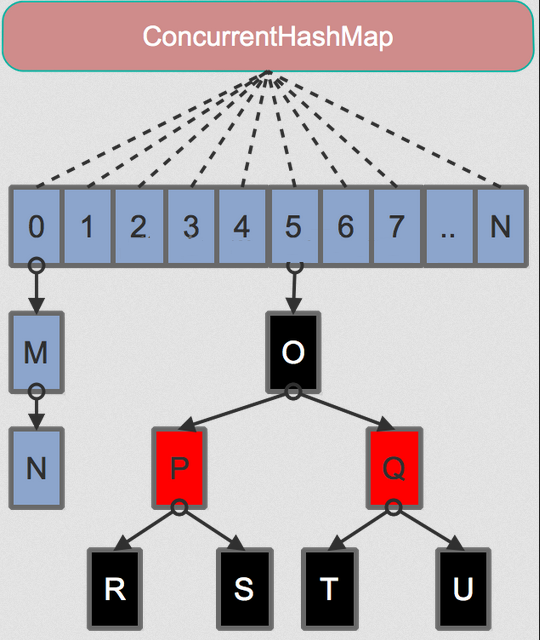
下面以ConcurrentHashMap为例讲解并发包的数据结构和保证安全的方法

## ConcurrentHashMap数据结构

### Java7基于分段的数据结构



### Java8基于CAS的数据结构



## ConcurrentHashMap同步原理

### Java7同步实现分析

put 的主流程：

public V put(K key, V value) {

Segment<K,V> s;

if (value == null)

throw new NullPointerException();

// 1. 计算 key 的 hash 值

int hash = hash(key);

// 2. 根据 hash 值找到 Segment 数组中的位置 j

// hash 是 32 位，无符号右移 segmentShift(28) 位，剩下低 4 位，

// 然后和 segmentMask(15) 做一次与操作，也就是说 j 是 hash 值的最后 4 位，也就是槽的数组下标

int j = (hash >>> segmentShift) & segmentMask;

// 刚刚说了，初始化的时候初始化了 segment[0]，但是其他位置还是 null，

// ensureSegment(j) 对 segment[j] 进行初始化

if ((s = (Segment<K,V>)UNSAFE.getObject // nonvolatile; recheck

(segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment

s = ensureSegment(j);

// 3. 插入新值到 槽 s 中

return s.put(key, hash, value, false);

}

Segment内部的put方法（对应上方标红的s.put(key, hash, value, false)）：

final V put(K key, int hash, V value, boolean onlyIfAbsent) {

// 在往该 segment 写入前，需要先获取该 segment 的独占锁

// 先看主流程，后面还会具体介绍这部分内容

HashEntry<K,V> node = tryLock() ? null :

scanAndLockForPut(key, hash, value);

V oldValue;

try {

// 这个是 segment 内部的数组

HashEntry<K,V>[] tab = table;

// 再利用 hash 值，求应该放置的数组下标

int index = (tab.length - 1) & hash;

// first 是数组该位置处的链表的表头

HashEntry<K,V> first = entryAt(tab, index);

// 下面这串 for 循环虽然很长，不过也很好理解，想想该位置没有任何元素和已经存在一个链表这两种情况

for (HashEntry<K,V> e = first;;) {

if (e != null) {

K k;

if ((k = e.key) == key ||

(e.hash == hash && key.equals(k))) {

oldValue = e.value;

if (!onlyIfAbsent) {

// 覆盖旧值

e.value = value;

++modCount;

}

break;

}

// 继续顺着链表走

e = e.next;

}

else {

// node 到底是不是 null，这个要看获取锁的过程，不过和这里都没有关系。

// 如果不为 null，那就直接将它设置为链表表头；如果是null，初始化并设置为链表表头。

if (node != null)

node.setNext(first);

else

node = new HashEntry<K,V>(hash, key, value, first);

int c = count + 1;

// 如果超过了该 segment 的阈值，这个 segment 需要扩容

if (c > threshold && tab.length < MAXIMUM\_CAPACITY)

rehash(node); // 扩容后面也会具体分析

else

// 没有达到阈值，将 node 放到数组 tab 的 index 位置，

// 其实就是将新的节点设置成原链表的表头

setEntryAt(tab, index, node);

++modCount;

count = c;

oldValue = null;

break;

}

}

} finally {

// 解锁

unlock();

}

return oldValue;

}

scanAndLockForPut方法获取锁：对应上边scanAndLockForPut(key, hash, value);

tryLock() 成功了，循环终止；

重试次数超过了 MAX\_SCAN\_RETRIES，进到 lock() 方法，此方法会阻塞等待，直到成功拿到独占锁。

private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {

    HashEntry<K,V> first = entryForHash(this, hash);

    HashEntry<K,V> e = first;

    HashEntry<K,V> node = null;

    int retries = -1; // negative while locating node

    // 循环获取锁

    while (!tryLock()) {

        HashEntry<K,V> f; // to recheck first below

        if (retries < 0) {

            if (e == null) {

                if (node == null) // speculatively create node

                    // 进到这里说明数组该位置的链表是空的，没有任何元素

                    // 当然，进到这里的另一个原因是 tryLock() 失败，所以该槽存在并发，不一定是该位置

                    node = new HashEntry<K,V>(hash, key, value, null);

                retries = 0;

            }

            else if (key.equals(e.key))

                retries = 0;

            else

                // 顺着链表往下走

                e = e.next;

        }

        // 重试次数如果超过 MAX\_SCAN\_RETRIES（单核1多核64），那么不抢了，进入到阻塞队列等待锁

        //    lock() 是阻塞方法，直到获取锁后返回

        else if (++retries > MAX\_SCAN\_RETRIES) {

            lock();

            break;

        }

        else if ((retries & 1) == 0 &&

                 // 这个时候是有大问题了，那就是有新的元素进到了链表，成为了新的表头

                 //     所以这边的策略是，相当于重新走一遍这个 scanAndLockForPut 方法

                 (f = entryForHash(this, hash)) != first) {

            e = first = f; // re-traverse if entry changed

            retries = -1;

        }

    }

    return node;

}

ensureSegment方法初始化分片中指定位置的元素（槽）：使用CAS保证线程安全

private Segment<K,V> ensureSegment(int k) {

final Segment<K,V>[] ss = this.segments;

long u = (k << SSHIFT) + SBASE; // raw offset

Segment<K,V> seg;

if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {

// 这里看到为什么之前要初始化 segment[0] 了，

// 使用当前 segment[0] 处的数组长度和负载因子来初始化 segment[k]

// 为什么要用“当前”，因为 segment[0] 可能早就扩容过了

Segment<K,V> proto = ss[0];

int cap = proto.table.length;

float lf = proto.loadFactor;

int threshold = (int)(cap \* lf);

// 初始化 segment[k] 内部的数组

HashEntry<K,V>[] tab = (HashEntry<K,V>[])new HashEntry[cap];

if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))

== null) { // 再次检查一遍该槽是否被其他线程初始化了。

Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);

// 使用 while 循环，内部用 CAS，当前线程成功设值或其他线程成功设值后，退出

while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))

== null) {

if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))

break;

}

}

}

return seg;

}

### Java8同步实现分析

put方法主流程

public V put(K key, V value) {

return putVal(key, value, false);

}

final V putVal(K key, V value, boolean onlyIfAbsent) {

if (key == null || value == null) throw new NullPointerException();

// 得到 hash 值

int hash = spread(key.hashCode());

// 用于记录相应链表的长度

int binCount = 0;

for (Node<K,V>[] tab = table;;) {

Node<K,V> f; int n, i, fh;

// 如果数组"空"，进行数组初始化

if (tab == null || (n = tab.length) == 0)

// 初始化数组，后面会详细介绍

tab = initTable();

// 找该 hash 值对应的数组下标，得到第一个节点 f

else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {

// 如果数组该位置为空，

// 用一次 CAS 操作将这个新值放入其中即可，这个 put 操作差不多就结束了，可以拉到最后面了

// 如果 CAS 失败，那就是有并发操作，进到下一个循环就好了

if (casTabAt(tab, i, null,

new Node<K,V>(hash, key, value, null)))

break; // no lock when adding to empty bin

}

// hash 居然可以等于 MOVED，这个需要到后面才能看明白，不过从名字上也能猜到，肯定是因为在扩容

else if ((fh = f.hash) == MOVED)

// 帮助数据迁移，这个等到看完数据迁移部分的介绍后，再理解这个就很简单了

tab = helpTransfer(tab, f);

else { // 到这里就是说，f 是该位置的头结点，而且不为空

V oldVal = null;

// 获取数组该位置的头结点的监视器锁

synchronized (f) {

if (tabAt(tab, i) == f) {

if (fh >= 0) { // 头结点的 hash 值大于 0，说明是链表

// 用于累加，记录链表的长度

binCount = 1;

// 遍历链表

for (Node<K,V> e = f;; ++binCount) {

K ek;

// 如果发现了"相等"的 key，判断是否要进行值覆盖，然后也就可以 break 了

if (e.hash == hash &&

((ek = e.key) == key ||

(ek != null && key.equals(ek)))) {

oldVal = e.val;

if (!onlyIfAbsent)

e.val = value;

break;

}

// 到了链表的最末端，将这个新值放到链表的最后面

Node<K,V> pred = e;

if ((e = e.next) == null) {

pred.next = new Node<K,V>(hash, key,

value, null);

break;

}

}

}

else if (f instanceof TreeBin) { // 红黑树

Node<K,V> p;

binCount = 2;

// 调用红黑树的插值方法插入新节点

if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,

value)) != null) {

oldVal = p.val;

if (!onlyIfAbsent)

p.val = value;

}

}

}

}

// binCount != 0 说明上面在做链表操作

if (binCount != 0) {

// 判断是否要将链表转换为红黑树，临界值和 HashMap 一样，也是 8

if (binCount >= TREEIFY\_THRESHOLD)

// 这个方法和 HashMap 中稍微有一点点不同，那就是它不是一定会进行红黑树转换，

// 如果当前数组的长度小于 64，那么会选择进行数组扩容，而不是转换为红黑树

// 具体源码我们就不看了，扩容部分后面说

treeifyBin(tab, i);

if (oldVal != null)

return oldVal;

break;

}

}

}

//

addCount(1L, binCount);

return null;

}

put 的主流程看完了，下面我们再深入研究三个问题：

1. 如何初始化？
2. 如何扩容？
3. 如何帮助数据迁移？

初始化方法是 initTable()：该方法通过sizeCtl实现CAS初始化

private final Node<K,V>[] initTable() {

Node<K,V>[] tab; int sc;

while ((tab = table) == null || tab.length == 0) {

// 初始化的"功劳"被其他线程"抢去"了

if ((sc = sizeCtl) < 0)

Thread.yield(); //放弃执行权

// CAS 一下，将 sizeCtl 设置为 -1，代表抢到了锁

else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {

try {

if ((tab = table) == null || tab.length == 0) {

// DEFAULT\_CAPACITY 默认初始容量是 16

int n = (sc > 0) ? sc : DEFAULT\_CAPACITY;

// 初始化数组，长度为 16 或初始化时提供的长度

Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];

// 将这个数组赋值给 table，table 是 volatile 的

table = tab = nt;

// 如果 n 为 16 的话，那么这里 sc = 12

// 其实就是 0.75 \* n

sc = n - (n >>> 2);

}

} finally {

// 设置 sizeCtl 为 sc，我们就当是 12 吧

sizeCtl = sc;

}

break;

}

}

return tab;

}

扩容方法是tryPresize：该方法通过sizeCtl实现CAS初始化

// 首先要说明的是，方法参数 size 传进来的时候就已经翻了倍了

private final void tryPresize(int size) {

// c：size 的 1.5 倍，再加 1，再往上取最近的 2 的 n 次方。

int c = (size >= (MAXIMUM\_CAPACITY >>> 1)) ? MAXIMUM\_CAPACITY :

tableSizeFor(size + (size >>> 1) + 1);

int sc;

while ((sc = sizeCtl) >= 0) {

Node<K,V>[] tab = table; int n;

// 这个 if 分支和之前说的初始化数组的代码基本上是一样的，在这里，我们可以不用管这块代码

if (tab == null || (n = tab.length) == 0) {

n = (sc > c) ? sc : c;

if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {

try {

if (table == tab) {

@SuppressWarnings("unchecked")

Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];

table = nt;

sc = n - (n >>> 2); // 0.75 \* n

}

} finally {

sizeCtl = sc;

}

}

}

else if (c <= sc || n >= MAXIMUM\_CAPACITY)

break;

else if (tab == table) {

int rs = resizeStamp(n);

if (sc < 0) {

Node<K,V>[] nt;

if ((sc >>> RESIZE\_STAMP\_SHIFT) != rs || sc == rs + 1 ||

sc == rs + MAX\_RESIZERS || (nt = nextTable) == null ||

transferIndex <= 0)

break;

// 2. 用 CAS 将 sizeCtl 加 1，然后执行 transfer 方法

// 此时 nextTab 不为 null

if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))

transfer(tab, nt);

}

// 1. 将 sizeCtl 设置为 (rs << RESIZE\_STAMP\_SHIFT) + 2)

// 调用 transfer 方法，此时 nextTab 参数为 null

else if (U.compareAndSwapInt(this, SIZECTL, sc,

(rs << RESIZE\_STAMP\_SHIFT) + 2))

transfer(tab, null);

}

}

}

数据迁移方法是transfer：该方法通过CAS和synchronized关键字实现同步。

private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {

int n = tab.length, stride;

// stride 在单核下直接等于 n，多核模式下为 (n>>>3)/NCPU，最小值是 16

// stride 可以理解为”步长“，有 n 个位置是需要进行迁移的，

// 将这 n 个任务分为多个任务包，每个任务包有 stride 个任务

if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN\_TRANSFER\_STRIDE)

stride = MIN\_TRANSFER\_STRIDE; // subdivide range

// 如果 nextTab 为 null，先进行一次初始化

// 前面我们说了，外围会保证第一个发起迁移的线程调用此方法时，参数 nextTab 为 null

// 之后参与迁移的线程调用此方法时，nextTab 不会为 null

if (nextTab == null) {

try {

// 容量翻倍

Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];

nextTab = nt;

} catch (Throwable ex) { // try to cope with OOME

sizeCtl = Integer.MAX\_VALUE;

return;

}

// nextTable 是 ConcurrentHashMap 中的属性

nextTable = nextTab;

// transferIndex 也是 ConcurrentHashMap 的属性，用于控制迁移的位置

transferIndex = n;

}

int nextn = nextTab.length;

// ForwardingNode 翻译过来就是正在被迁移的 Node

// 这个构造方法会生成一个Node，key、value 和 next 都为 null，关键是 hash 为 MOVED

// 后面我们会看到，原数组中位置 i 处的节点完成迁移工作后，

// 就会将位置 i 处设置为这个 ForwardingNode，用来告诉其他线程该位置已经处理过了

// 所以它其实相当于是一个标志。

ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);

// advance 指的是做完了一个位置的迁移工作，可以准备做下一个位置的了

boolean advance = true;

boolean finishing = false; // to ensure sweep before committing nextTab

/\*

\* 下面这个 for 循环，最难理解的在前面，而要看懂它们，应该先看懂后面的，然后再倒回来看

\*

\*/

// i 是位置索引，bound 是边界，注意是从后往前

for (int i = 0, bound = 0;;) {

Node<K,V> f; int fh;

// 下面这个 while 真的是不好理解

// advance 为 true 表示可以进行下一个位置的迁移了

// 简单理解结局：i 指向了 transferIndex，bound 指向了 transferIndex-stride

while (advance) {

int nextIndex, nextBound;

if (--i >= bound || finishing)

advance = false;

// 将 transferIndex 值赋给 nextIndex

// 这里 transferIndex 一旦小于等于 0，说明原数组的所有位置都有相应的线程去处理了

else if ((nextIndex = transferIndex) <= 0) {

i = -1;

advance = false;

}

else if (U.compareAndSwapInt

(this, TRANSFERINDEX, nextIndex,

nextBound = (nextIndex > stride ?

nextIndex - stride : 0))) {

// 看括号中的代码，nextBound 是这次迁移任务的边界，注意，是从后往前

bound = nextBound;

i = nextIndex - 1;

advance = false;

}

}

if (i < 0 || i >= n || i + n >= nextn) {

int sc;

if (finishing) {

// 所有的迁移操作已经完成

nextTable = null;

// 将新的 nextTab 赋值给 table 属性，完成迁移

table = nextTab;

// 重新计算 sizeCtl：n 是原数组长度，所以 sizeCtl 得出的值将是新数组长度的 0.75 倍

sizeCtl = (n << 1) - (n >>> 1);

return;

}

// 之前我们说过，sizeCtl 在迁移前会设置为 (rs << RESIZE\_STAMP\_SHIFT) + 2

// 然后，每有一个线程参与迁移就会将 sizeCtl 加 1，

// 这里使用 CAS 操作对 sizeCtl 进行减 1，代表做完了属于自己的任务

if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {

// 任务结束，方法退出

if ((sc - 2) != resizeStamp(n) << RESIZE\_STAMP\_SHIFT)

return;

// 到这里，说明 (sc - 2) == resizeStamp(n) << RESIZE\_STAMP\_SHIFT，

// 也就是说，所有的迁移任务都做完了，也就会进入到上面的 if(finishing){} 分支了

finishing = advance = true;

i = n; // recheck before commit

}

}

// 如果位置 i 处是空的，没有任何节点，那么放入刚刚初始化的 ForwardingNode ”空节点“

else if ((f = tabAt(tab, i)) == null)

advance = casTabAt(tab, i, null, fwd);

// 该位置处是一个 ForwardingNode，代表该位置已经迁移过了

else if ((fh = f.hash) == MOVED)

advance = true; // already processed

else {

// 对数组该位置处的结点加锁，开始处理数组该位置处的迁移工作

synchronized (f) {

if (tabAt(tab, i) == f) {

Node<K,V> ln, hn;

// 头结点的 hash 大于 0，说明是链表的 Node 节点

if (fh >= 0) {

// 下面这一块和 Java7 中的 ConcurrentHashMap 迁移是差不多的，

// 需要将链表一分为二，

// 找到原链表中的 lastRun，然后 lastRun 及其之后的节点是一起进行迁移的

// lastRun 之前的节点需要进行克隆，然后分到两个链表中

int runBit = fh & n;

Node<K,V> lastRun = f;

for (Node<K,V> p = f.next; p != null; p = p.next) {

int b = p.hash & n;

if (b != runBit) {

runBit = b;

lastRun = p;

}

}

if (runBit == 0) {

ln = lastRun;

hn = null;

}

else {

hn = lastRun;

ln = null;

}

for (Node<K,V> p = f; p != lastRun; p = p.next) {

int ph = p.hash; K pk = p.key; V pv = p.val;

if ((ph & n) == 0)

ln = new Node<K,V>(ph, pk, pv, ln);

else

hn = new Node<K,V>(ph, pk, pv, hn);

}

// 其中的一个链表放在新数组的位置 i

setTabAt(nextTab, i, ln);

// 另一个链表放在新数组的位置 i+n

setTabAt(nextTab, i + n, hn);

// 将原数组该位置处设置为 fwd，代表该位置已经处理完毕，

// 其他线程一旦看到该位置的 hash 值为 MOVED，就不会进行迁移了

setTabAt(tab, i, fwd);

// advance 设置为 true，代表该位置已经迁移完毕

advance = true;

}

else if (f instanceof TreeBin) {

// 红黑树的迁移

TreeBin<K,V> t = (TreeBin<K,V>)f;

TreeNode<K,V> lo = null, loTail = null;

TreeNode<K,V> hi = null, hiTail = null;

int lc = 0, hc = 0;

for (Node<K,V> e = t.first; e != null; e = e.next) {

int h = e.hash;

TreeNode<K,V> p = new TreeNode<K,V>

(h, e.key, e.val, null, null);

if ((h & n) == 0) {

if ((p.prev = loTail) == null)

lo = p;

else

loTail.next = p;

loTail = p;

++lc;

}

else {

if ((p.prev = hiTail) == null)

hi = p;

else

hiTail.next = p;

hiTail = p;

++hc;

}

}

// 如果一分为二后，节点数少于 8，那么将红黑树转换回链表

ln = (lc <= UNTREEIFY\_THRESHOLD) ? untreeify(lo) :

(hc != 0) ? new TreeBin<K,V>(lo) : t;

hn = (hc <= UNTREEIFY\_THRESHOLD) ? untreeify(hi) :

(lc != 0) ? new TreeBin<K,V>(hi) : t;

// 将 ln 放置在新数组的位置 i

setTabAt(nextTab, i, ln);

// 将 hn 放置在新数组的位置 i+n

setTabAt(nextTab, i + n, hn);

// 将原数组该位置处设置为 fwd，代表该位置已经处理完毕，

// 其他线程一旦看到该位置的 hash 值为 MOVED，就不会进行迁移了

setTabAt(tab, i, fwd);

// advance 设置为 true，代表该位置已经迁移完毕

advance = true;

}

}

}

}

}

}

# 线程池

## 为什么使用线程池

多线程的缺点：

处理任务的线程创建和销毁都非常耗时并消耗资源。

多线程之间的切换也会非常耗时并消耗资源。

解决方法：采用线程池

使用时线程已存在，消除了线程创建的时耗

通过设置线程数目，防止资源不足

## ThreadPoolExecutor的全参构造函数参数介绍

在Java中创建线程池常用的类是ThreadPoolExecutor，该类的全参构造函数如下：

**public** ThreadPoolExecutor(**int** corePoolSize,  
 **int** maximumPoolSize,  
 **long** keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {

参数介绍：

* corePoolSize：线程池中核心线程数的最大值
* maximumPoolSize：线程池中能拥有最多线程数
* workQueue：用于缓存任务的阻塞队列，对于不同的应用场景我们可能会采取不同的排队策略，这就需要不同类型的阻塞队列，在线程池中常用的阻塞队列有以下2种：
* SynchronousQueue<Runnable>：此队列中不缓存任何一个任务。向线程池提交任务时，如果没有空闲线程来运行任务，则入列操作会阻塞。当有线程来获取任务时，出列操作会唤醒执行入列操作的线程。从这个特性来看，SynchronousQueue是一个无界队列，因此当使用SynchronousQueue作为线程池的阻塞队列时，参数maximumPoolSizes没有任何作用。
* LinkedBlockingQueue<Runnable>：顾名思义是用链表实现的队列，可以是有界的，也可以是无界的，但在Executors中默认使用无界的。

以上三个参数之间的关系如下：

1. 如果没有空闲的线程执行该任务且当前运行的线程数少于corePoolSize，则添加新的线程执行该任务。
2. 如果没有空闲的线程执行该任务且当前的线程数等于corePoolSize同时阻塞队列未满，则将任务入队列，而不添加新的线程。
3. 如果没有空闲的线程执行该任务且阻塞队列已满同时池中的线程数小于maximumPoolSize，则创建新的线程执行任务。
4. 如果没有空闲的线程执行该任务且阻塞队列已满同时池中的线程数等于maximumPoolSize，则根据构造函数中的handler指定的策略来拒绝新的任务。

* keepAliveTime：表示空闲线程的存活时间。
* unit：表示keepAliveTime的单位。
* handler：表示当workQueue已满，且池中的线程数达到maximumPoolSize时，线程池拒绝添加新任务时采取的策略。一般可以采取以下四种取值。

|  |  |
| --- | --- |
| ThreadPoolExecutor.AbortPolicy() | 抛出RejectedExecutionException异常 |
| ThreadPoolExecutor.CallerRunsPolicy() | 由向线程池提交任务的线程来执行该任务 |
| ThreadPoolExecutor.DiscardOldestPolicy() | 抛弃最旧的任务（最先提交而没有得到执行的任务） |
| ThreadPoolExecutor.DiscardPolicy() | 抛弃当前的任务 |

* threadFactory：指定创建线程的工厂

## 四种常用线程池

ThreadPoolExecutor构造函数的参数很多，使用起来很麻烦，为了方便的创建线程池，JavaSE中又定义了Executors类，Eexcutors类提供了四个创建线程池的方法，分别如下

* newCachedThreadPool
* newFixedThreadPool
* newSingleThreadExecutor
* newScheduleThreadPool

### newCachedThreadPool

该方法创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

此类型线程池特点是：

1. 工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE)
2. 空闲的工作线程会自动销毁，有新任务会重新创建
3. 在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

示例：

public static void main(String[] args) {

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

for (int i = 0; i < 10; i++) {

final int index = i;

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

cachedThreadPool.execute(new Runnable() {

public void run() {

System.out.println(index);

}

});

}

}

### newFixedThreadPool

该方法创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

优点：具有线程池提高程序效率和节省创建线程时所耗的开销。

缺点：在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

示例：

public static void main(String[] args) {

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);

for (int i = 0; i < 10; i++) {

final int index = i;

fixedThreadPool.execute(new Runnable() {

public void run() {

try {

System.out.println(index);

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

}

}

### newSingleThreadExecutor

该方法创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO,优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。

单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

示例：

public static void main(String[] args) {

ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

for (int i = 0; i < 10; i++) {

final int index = i;

singleThreadExecutor.execute(new Runnable() {

public void run() {

try {

System.out.println(index);

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

}

}

### newScheduleThreadPool

该方法创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。

示例：

public static void main(String[] args) {

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

for (int i = 0; i < 10; i++) {

scheduledThreadPool.schedule(new Runnable() {

public void run() {

System.out.println("delay 3 seconds");

}

}, 3, TimeUnit.SECONDS);

}

}