类的加载顺序:  
 1.静态代码块

-->父类静态初始化块,静态属性,但不包括静态方法

-->子类静态初始化块,静态属性,但不包括静态方法

2.非静态代码块

构造函数

final的作用:

1.类不可以被继承

2.方法不能被重写

3.变量不可变,修饰引用,表示引用不可变,指向引用的内容可变

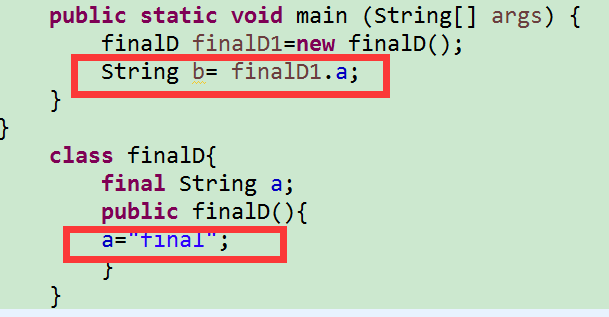
4.被final修饰的方法,jvm会尝试与其内联,以提高运行效率

5.被final修饰的常量,在编译阶段会存到常量池.

编译器对final域要遵循的两个重排序规则

1.在构造函数内对一个final域的写入,随后把这个被构造对象的引用赋值给一个引用变量,者两个操作不能重排序

暂时理解:



2.初次读一个包含final域的对象的引用,和随后初次读这个final域,这两个操作之前不能重排序.

线程阻塞的方法:

1.sleep

2.suspend和resume

3.yield 放弃当前已经分的cpu时间片,但仍处于可执行状态,随时可以获得时间片,

4.wait和notify wait毫秒,不需要唤醒,没有参数,必须使用notify

wait,notify和suspend,resume

初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的notify()方法则导致从调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和synchronized关键字一起使用，将它们和操作系统进程间通信机制作一个比较就会发现它们的相似性：synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点：

第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

synchronized与reentrantlock的区别

1.synchronized是关键字,reentrantlock是类

2.类的话,更灵活,可以获取锁的等待时间进行设置,避免死锁,可以获取锁的各种信息,可以灵活的实现多路通知.

3.reentrantlock底层使用的unsafe的park方法加锁,synchronized使用的对象头中的Markword.

concurrenthashmap的并发度是16

hashtable的并发度是2

concurrenthashmap1.6的实现原理:

hashtable是通过对hash表结构进行锁定,是阻塞式的,当一个线程占有这个锁时,其他线程必须等待其释放锁.concurrenthashmap是采用分离锁的方式,并没有对整个hash表进行锁定,而是局部锁定,也就是说当一个线程占有这个局部锁时,不影响其他线程对hash其他局部的使用,具体是concurrenthashmap内部有一个segment分离锁.

1.8中不在使用

|  |  |  |
| --- | --- | --- |
|  | CyclicBarrier(栅栏回环) | Countdownlatch |
|  | 某个线程运行到某个点停止,所有到达,才继续运行,(打篮球,5个人都到了,才能打) | 有一个条件 条件变为0后不能运行(大概是公交车,坐麻了就不能做了) |
|  | 只能唤起一个任务 | 唤起多个任务 |
|  | 可重用 | 不可重用,值为0后不可用, |
|  |  |  |

用segment分离锁,而是采用乐观锁cas算法来实现同步问题,底层仍然是数组+链表->红黑树的实现.

多线程开发的良好习惯

给线程命名,最小化同步范围,优先使用volatile,尽量使用阻塞队列,信号量,而非wait(),notify(),优先使用并发容器而非同步容器.使用线程池.

priorityqueue是一个优先级队列,保证优先级最高或者最低的元素总在队列头部,

linkedhashmap维护的顺序是插入的顺序,

当遍历一个priorityqueue,没有任何顺序保证,但是linkedhashmap保证遍历使用的时插入的顺序.

weakhashmap使用弱引用作为key,意思是当可以没有任何引用时,key/value将被回收.

arraylist-数组,随机访问,时间复杂度O(1)

linkedlist-双向循环链表,O(n)

java的7大设计原则:

1.开闭原则:一个软件实体,类,模块函数,对扩展开发,对修改关闭

2.里氏替换原则:如果对于每个类型 T1的对象o1,都有类型为T2的对象o2,使得T1定义的所有程序P在所有对象o1替换为o2时,程序P的行为没有发生变化,那么T2是T1的子类型. 子类型必须能够替换掉他们的父类型.

3.依赖倒转原则:高层模块不应该依赖于底层模块,二者都应该依赖于其抽象,抽象不应该依赖于抽象,细节应该依赖于抽象.即针对接口编程,不针对实现编程.

4.接口隔离原则:建立单一接口,尽量细化接口,接口中的方法尽量少,为各个类创建专用的接口.

5.组合/聚合复用原则:尽量使用合成和聚合,不使用继承.

6.迪米特法则:耦合最低原则

7.单一职责原则:一个类只负责一项职责.

设计模式:

创建型模式：

1.简单工厂模式（不包含在gof23中）

2.工厂模式

3.抽象工厂模式

4.单例模式

5.原型模式

创建者模式

6.结构型模式：

7.组合模式

8.装饰者模式

9.外观模式

10.适配器模式

11.代理模式

12.享元模式

13.桥接模式

行为型模式：

14.观察者模式

15.策略模式

16.状态模式

17.中介模式

18.模板方法

19.命令模式

20.备忘录模式

21.访问者模式

22.解释器模式

23.迭代器模式

24.职责链模式

一些集合的接口,set list map queue dequeue sortedset sortedmap listIterator

fail-fast快速失败:当在迭代一个集合时,另一个线程正在修改你正在访问的那个线程,就会能抛出concurrentmodificationException 在java.util下的都是快速失败

fail-safe:安全失败: 在迭代的时候回去底层做一个拷贝,在修改该上层集合时不受影响,不会抛出异常. 在java.util.concurrent包下的都是安全失败

bucket存储空间.

hashmap是一个链表散列,最左侧是一个数组.数组中的每一个元素都是链表,链表的每一个元素都是entry,调用put方法时,首先对键调用hashcode方法,返回的hashcode,用于找到bucket位置来存储Entry对象.

对map的增删改的操作使用hashmap,对有序key集合的遍历使用treemap.

arraylist和vector :两者都是基于索引的,内部有一个数组支持,两者维护插入的顺序,我们可以根据插入的顺序获取元素,迭代器都是fail-fast.两者都允许空值,也可以使用索引值对元素进行随机访问

不同vector是同步的,迭代的时候对列表进行改变使用copyonwritearraylist

arraylist快.

装饰者模式和代理模式.

装饰者模式:以对客户端透明度的方式扩展对象的功能,是继承关系的扩展方案,

代理模式:给对象提供一个新的代理对象,并由对象来控制对原有对象的引用.

springmvc 核心流程,控制反转

用户发送请求到前端控制器,也是springmvc的核心控制器,前端控制器收到请求,自己不进行处理,而是委托给其他的解析器进行处理,作为统一访问点,进行全局的流程控制.

前端控制器调用处理器映射器,处理器映射器会把请求映射为处理器解析链对象(安博涵处理器对象,也就是controller层,多个处理器拦截器)返回给前端控制器;

String a="abc";

调用的是String.valueOf来返回String实例对象,至于调用那个取决于你的赋值,比如 String a=1;调用的是

public static String valueOf(int i){

return Integer.toString(i);

}

String a=new String("abc"); 调用的是

public String(String original){

this.value=original.value;

this.hash=original.hash;

} 最后将变量存储在一个char数组中.

Stringbuffer的toString的方法会对对象进行缓存,减少元素赋值开销

Stringbuilder的 toString的时候会直接返回一个新对象

类的加载顺序

加载连接初始化

=====

1静态变量

1静态代码块

1.1

2非静态变量(成员变量)

2构造函数

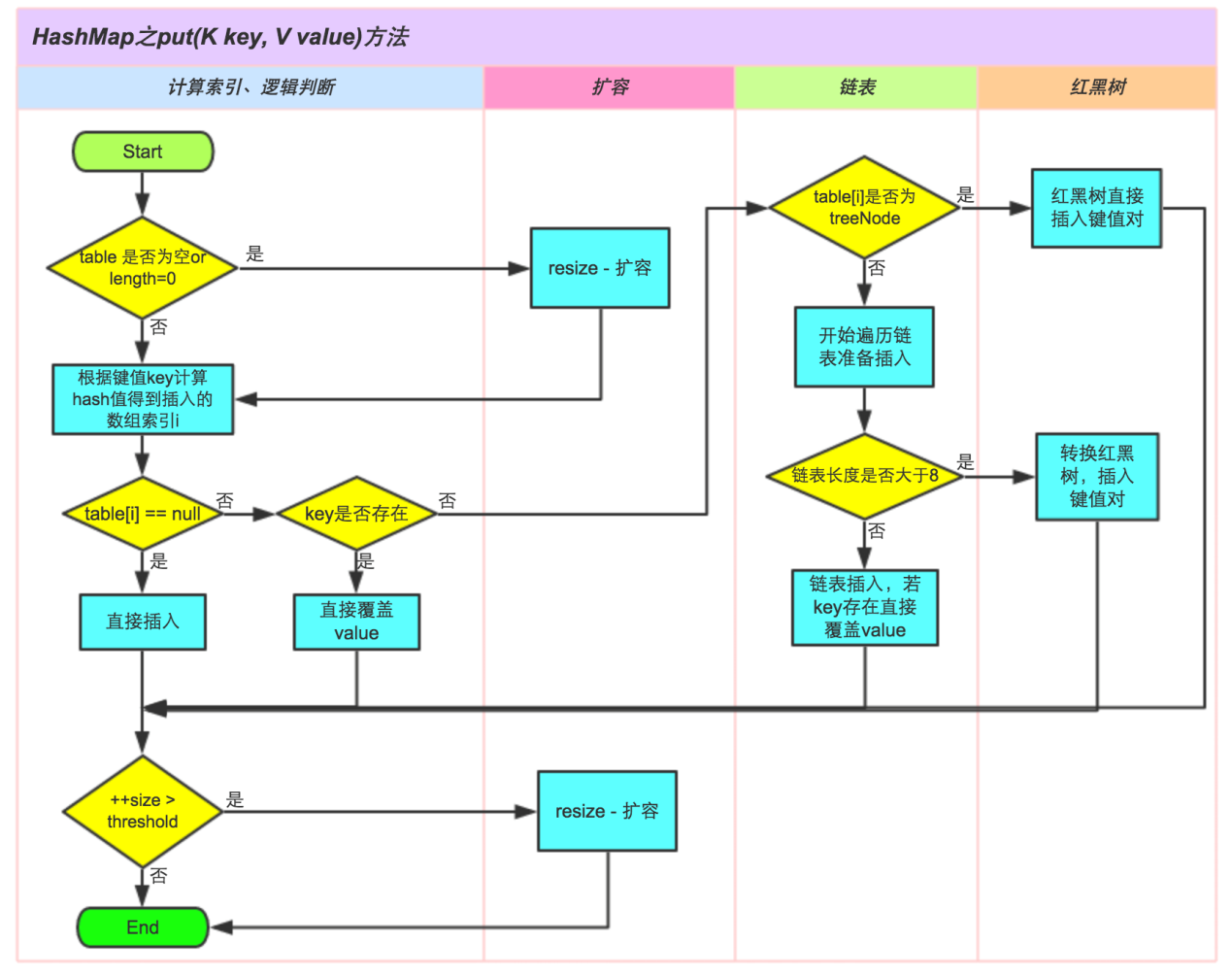
2.1

父类静态变量-->父类静态代码块-->子类静态变量-->子类静态代码块

-->父类成员变量-->父类构造函数--子类成员变量-->子类构造函数

有序的map:treeamap(默认升序),LinkedHashMap(则是记录了插入顺序)

hashmap的插入方法:



hashset是基于hashmap实现的,容量默认是16 .加载因子0.75

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Hashmap | HashSet | Hashtable | Linkedhashmap | Linkedhashset | Arraylist | Linkedlist | Concurrenthashmap |
| 初始容量 | 16 | 16 | 11 | 16 | 16 | 10 1.8(0) | 没有 | 16 |
| 加载因子  (警戒线) | 0.75(12) | 0.75 | 0.75 | 0.75  直接使用hashmap | 0.75 | 1 | 没有 | 0.75 |
| 底层 | 数组和链表的结合体 | Hashmap(数组和链表) | 拉链式的哈希表 | 双重连接链表  维持迭代顺序 | 双重连接链表  维持迭代顺序 | 数组 | 双向链表 | 分情况, |
| 安全同步 | 不同步 | 不同步 | 同步并发数2 | 不同步 | 不同步 | 不同步 | 不同步 | 同步并发数16 |
| Null | 允许,方法  putForNullKey | 不允许 | 不允许 | 允许 | 不允许 | 不允许 | 不允许 | 允许 |
| 元素重复 | No | No | No | Yes | No | Yes |  | No |
| 扩容数 | 翻倍 | 翻倍 | 2\*11+1 | 翻倍 | 翻倍 | 1.5+1/  1.5 | 直接新增 | 翻倍 |
| 顺序 | No | No | No | Yes | Yes | Yes | Yes | No |

时间复杂度以后在研究

entryset效率高于keyset

util中的都是fail-fast,

entry是一个单向链表

threshold阈值,理解上就是警戒线

hashmap

|  |  |  |
| --- | --- | --- |
|  | Hashmap | Hashset |
|  | Map | Set |
|  | Put | Add |
|  | 键值对 |  |
|  | 快 |  |
|  |  |  |

内存屏障:为了保存执行顺序和可见性的一条cpu指令

重排序:为了提供性能,编译器和处理器会对执行命令进行重排序

happen-before:操作间执行的顺序,有些操作先发生

主内存:共享变量存储的区域就是主内存

工作内存:每个线程copy的本地内存,存储了该线程以读/写共享变量的副本