目录

[一、JVM 3](#_Toc118282164)

[1.1 什么是JVM 3](#_Toc118282165)

[二、面向对象 3](#_Toc118282166)

[2.1 3](#_Toc118282167)

[三、流程控制 3](#_Toc118282168)

[3.1 3](#_Toc118282169)

[四、容器 3](#_Toc118282170)

[4.1 List 3](#_Toc118282171)

[4.1.1 ArrayList 3](#_Toc118282172)

[4.1.2 LinkedList 3](#_Toc118282173)

[4.1.3 Vector 3](#_Toc118282174)

[4.1.4 Queue 3](#_Toc118282175)

[4.1.5 Deque 3](#_Toc118282176)

[4.2 Set 3](#_Toc118282177)

[4.2.1 HashSet 3](#_Toc118282178)

[4.2.2 TreeSet 3](#_Toc118282179)

[4.3 Map 3](#_Toc118282180)

[4.3.1 HashMap 3](#_Toc118282181)

[4.3.1 TreeMap 6](#_Toc118282182)

[4.3.1 HashTable 6](#_Toc118282183)

[五、线程 6](#_Toc118282184)

[5.1 多线程用法 6](#_Toc118282185)

[5.1.1 继承Thread类 6](#_Toc118282186)

[5.2.2 实现Runnable接口 6](#_Toc118282187)

[5.2.3 线程的生命周期 6](#_Toc118282188)

[5.2 线程池 6](#_Toc118282189)

[5.2.1 概述 6](#_Toc118282190)

[5.2.2 工作原理 6](#_Toc118282191)

[5.2.4 常见线程池 7](#_Toc118282192)

[5.3 ThreadLocal类 8](#_Toc118282193)

[5.3.1 定义 8](#_Toc118282194)

[5.3.2 实现原理 8](#_Toc118282195)

[5.3.3 使用场景 8](#_Toc118282196)

[5.4 Synchronized 8](#_Toc118282197)

[5.4.1 实现原理 8](#_Toc118282198)

[5.4.2 与volatile的区别 9](#_Toc118282199)

[5.4.3 与Lock的区别 9](#_Toc118282200)

[5.4.4 与ReentrantLock的区别 9](#_Toc118282201)

[5.4.5 CAS 9](#_Toc118282202)

[六、IO 10](#_Toc118282203)

[6.1 10](#_Toc118282204)

[6.2 NIO 10](#_Toc118282205)

[6.2.1 新特性 10](#_Toc118282206)

[6.2.2 核心组件 10](#_Toc118282207)

[七、异常 10](#_Toc118282208)

[7.1 11](#_Toc118282209)

[八、泛型 11](#_Toc118282210)

[8.1 11](#_Toc118282211)

[九、反射 11](#_Toc118282212)

[9.1 11](#_Toc118282213)

[十、注解 11](#_Toc118282214)

[10.1 11](#_Toc118282215)

# 一、JVM

## 1.1 什么是JVM

# 二、面向对象

## 2.1

# 三、流程控制

## 3.1

# 四、容器

## 4.1 List

### 4.1.1 ArrayList

### 4.1.2 LinkedList

### 4.1.3 Vector

### 4.1.4 Queue

### 4.1.5 Deque

## 4.2 Set

### 4.2.1 HashSet

### 4.2.2 TreeSet

## 4.3 Map

### 4.3.1 HashMap

1. 重要参数
2. 容量

必须是2的幂 & <最大容量（2的30次方）。

1. 加载因子

HashMap在其容量自动增加前可达到多满的一种尺度。

1. 扩容阈值

当哈希表的大小 ≥ 扩容阈值时，就会扩容哈希表（即扩充HashMap的容量）。

1. 桶的树化阈值

链表转成红黑树的阈值，在存储数据时，当链表长度 > 该值（8）时，则将链表转换成红黑树。

1. 桶的链表还原阈值

红黑树转为链表的阈值，当在扩容时（此时HashMap的数据存储位置会重新计算），在重新计算存储位置后，当原有的红黑树内数量<6时，则将红黑树转换成链表。

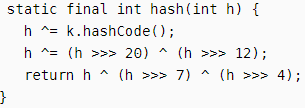
1. 最小树化容量阈值

当哈希表中的容量 > 该值时，才允许树形化链表（即将链表转换成红黑树）。

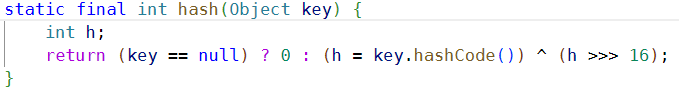
1. put()源码分析
   1. hash（key）扰动处理：

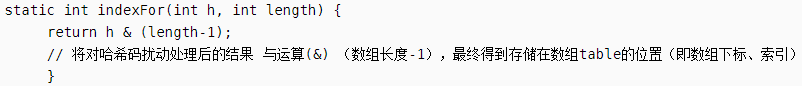
哈希码与数组大小不匹配，提高数据分布的随机性&均匀性。

* + 1. JDK 1.7­：

使用hashCode() + 4次位运算 + 5次异或运算（9次扰动）

* + 1. JDK 1.8：

使用hashCode() + 1次位运算 + 1次异或运算（2次扰动）

* 1. 计算下标

1. putVal(hash(key), key, value, false, true)源码分析
   1. 若哈希表的数组tab为空，则通过resize()创建。所以初始化哈希表的时机为第1次调用put函数时，即调用resize()初始化创建；
   2. 计算插入存储的数组索引；
   3. 判断是否为空节点：
      1. 若为空（即当前table[i] == null），则直接在该数组位置新建节点，插入完毕；
      2. 若不为空，判断是否存在Hash冲突：
         1. 若不存在（key相等），替换value；
         2. 若存在，判断需要插入的数据结构是红黑树or链表：
            1. 若为红黑树，putTreeVal()；
            2. 若为链表，尾部插入数据（JDK1.7是头插入），根据链表大小决定是否树化。
2. resize()源码分析
3. 是否需要初始化？当前容量>最大值？；
4. 新建2倍容量数组；
5. 保存旧数组做转移；
6. JDK1.7：扩容后存储位置和put()时的计算一致；

JDK1.8：hash & oldCap == 0则扩容后存储位置 = 原位置；

hash & oldCap == 1则扩容后存储位置 = 原位置+旧容量。

1. 重设扩容阈值。

### 4.3.1 TreeMap

### 4.3.1 HashTable

# 五、线程

## 5.1 多线程用法

### 5.1.1 继承Thread类

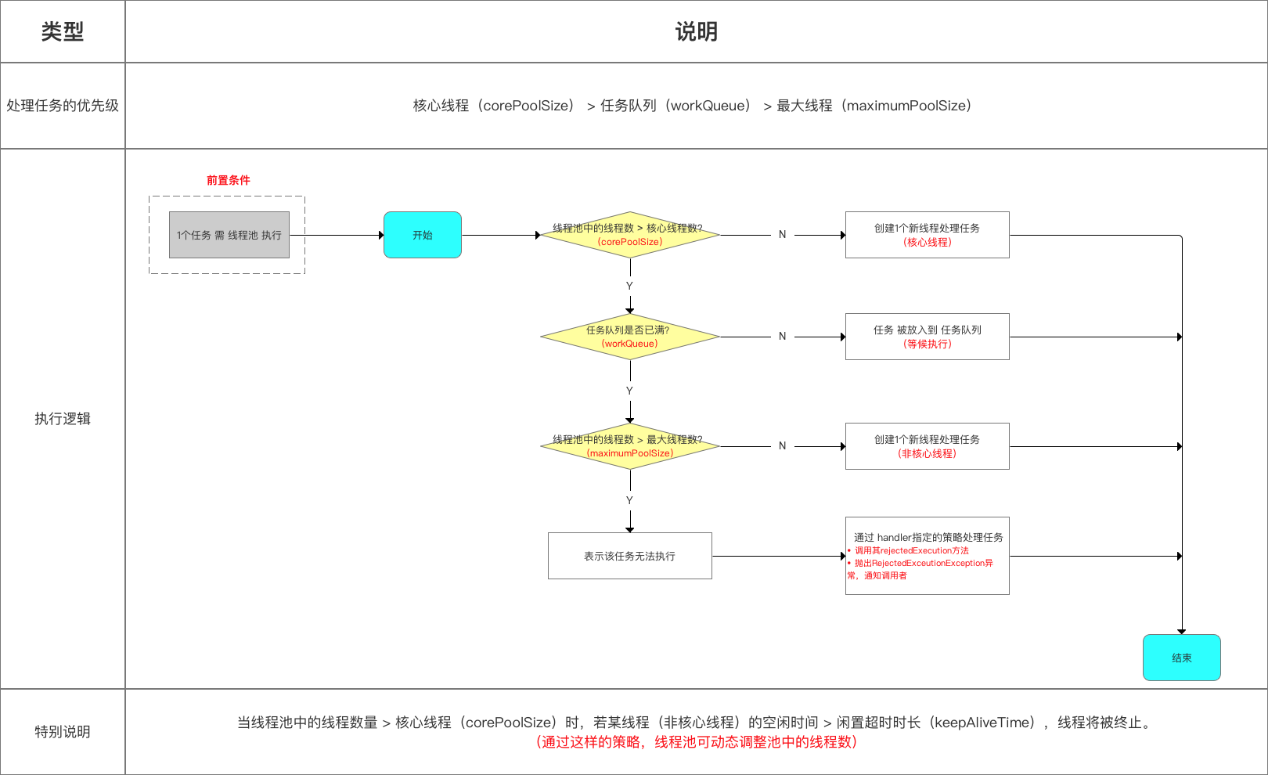
### 5.2.2 实现Runnable接口

### 5.2.3 线程的生命周期

## 5.2 线程池

### 5.2.1 概述

### 5.2.2 工作原理

1. 核心参数
   1. corePoolSize：核心线程数。默认情况下，核心线程会一直存活。
   2. maximumPoolSize：线程池所能容纳的最大线程数。当活动线程数到达该值后，后续的新任务将会阻塞。
   3. keepAliveTime：非核心线程闲置超时时长。超过该时长，非核心线程会被回收。
   4. unit：指定keepAliveTime参数的时间单位。
   5. workQueue：任务队列。通过线程池的execute()方法提交的Runnable对象将存储在该参数中。
   6. threadFactory：线程工厂。为线程池创建新的线程。
2. 内部逻辑

### 5.2.4 常见线程池

1. 定长线程池FixedThreadPool
   1. 特点：只有核心线程 & 不会被回收、线程数量固定、任务队列无大小限制（超出的线程任务会在队列中等待）。
   2. 应用场景：控制线程最大并发数。
2. 定时线程池ScheduledThreadPool
   1. 特点：核心线程数量固定、非核心线程数量无限制（闲置时马上回收）。
   2. 应用场景：执行定时/周期性任务。
3. 可缓存线程池CachedThreadPool
   1. 特点：只有非核心线程、线程数量不固定（可无限大）、灵活回收空闲线程（具备超时机制，全部回收时几乎不占系统资源）、新建线程（无线程可用时）。
   2. 应用场景：执行大量、耗时少的线程任务。
4. 单线程化线程池SingleThreadExecutor
   1. 特点：只有一个核心线程（保证所有任务按照指定顺序在一个线程中执行，不需要处理线程同步的问题）。
   2. 应用场景：不适合并发但可能引起IO阻塞性及影响UI线程响应的操作，如数据库操作，文件操作等。
5. 工作窃取线程池WorkStealingPool
   1. 特点：如果不主动设置它的并发数，那么这个方法就会以当前机器的CPU处理器个数为线程个数，这个线程池会并行处理任务，不能够保证任务执行的顺序。
   2. 应用场景：

## 5.3 ThreadLocal类

### 5.3.1 定义

ThreadLocal为每个使用该变量的线程提供独立的变量副本，使得每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

### 5.3.2 实现原理

ThreadLocalMap

### 5.3.3 使用场景

数据库连接和 session 管理等。

## 5.4 Synchronized

### 5.4.1 实现原理

synchronized 是由一对 monitorenter/monitorexit 指令实现的，monitor 对象是同步的基本实现单元。在 Java 6 之前，monitor 的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作，性能也很低。但在 Java 6 的时候，Java 虚拟机 对此进行了大刀阔斧地改进，提供了三种不同的 monitor 实现，也就是常说的三种不同的锁：偏向锁（Biased Locking）、轻量级锁和重量级锁，大大改进了其性能。

### 5.4.2 与volatile的区别

1. volatile是变量修饰符；synchronized 是修饰类、方法、代码段。
2. volatile仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
3. volatile不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

### 5.4.3 与Lock的区别

1. synchronized可以给类、方法、代码块加锁；而lock只能给代码块加锁。
2. synchronized不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而lock需要自己加锁和释放锁，如果使用不当没有unLock()去释放锁就会造成死锁。
3. 通过Lock可以知道有没有成功获取锁，而synchronized 却无法办到。

### 5.4.4 与ReentrantLock的区别

1. 公平锁。
2. 等待可中断。
3. 绑定多个Condition对象，使用起来更灵活。

### 5.4.5 CAS

1. 定义

Compare And Swap，即比较并交换，是一种解决并发操作的乐观锁。synchronized锁住的代码块：同一时刻只能由一个线程访问，属于悲观锁。

1. 典型应用：AtomicInteger

# 六、IO

## 6.1

## 6.2 NIO

### 6.2.1 新特性

### 6.2.2 核心组件

# 七、异常

## 7.1

# 八、泛型

## 8.1

# 九、反射

## 9.1

# 十、注解

## 10.1