Servlet

1，生命周期：

1. init：初始化
2. service： 处理客户端请求
3. destroy： 释放资源，自身标记可回收
4. 被垃圾回收器回收

Servlet容器会启动多个线程访问一个servlet方法，也就是多个客户端访问同一个service方法，会涉及线程安全问题。

2，servlet如何同时处理多个请求访问？

默认：单实例多线程

1. 来请求时，会实例化一个servlet，只有一个实例。
2. 容器初始化servlet主要是读取配置文件，如servlet.xml等
3. 请求到达时，servlet容器通过调度线程（Dispatcher Thread）调度他管理下的线程池中的等待执行的线程（Worker Thread）给请求者
4. 线程执行service方法
5. 请求结束，放回线程池中，等待被调用

3，上述功能操作可以看出：

1. servlet单实例，减少创建实例的开销
2. 通过线程池相应多个请求，提升效率
3. Servlet容器不关心到达的servlet请求访问是否是用一个servlet还是别的servlet，直接分配给他一个线程；如果是同一个servlet的多个请求，那么servlet中的service方法将做并发执行。
4. 每个请求由servletRequest对象接受，由servletResponse对象响应

4，过滤器Filter

Servlet范畴。实现filter接口

生命周期：

init：容器启动初始化过滤器时被调用，在整个生命周期只会被调用一次。如果此方法不成功，则过滤器没有作用

doFilter：容器的每一次请求都会调用

destroy：容器销毁

5，拦截器Interceptor

链式调用，一个应用中可以有多个拦截器，一个请求也可以出发多个拦截器，通过HandlerInterceptor实现

PreHandle：此方法在请求处理前调用。如果此方法返回false，将视为当前请求结束

postHandle：preHandle请求为true的时候才会执行。会在controller调用之后，dispatcherServlet返回视图渲染前被调用

AfterCompletion：只有在preHandle返回true才执行。在postHandle之后执行

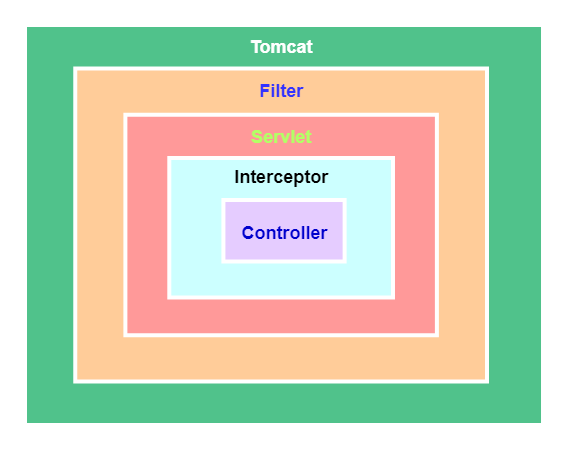
6，过滤器和拦截器

1. 实现原理不同

过滤器：基于函数回调；拦截器：基于java反射机制（动态代理）

1. 触发点不同

执行顺序：Filer处理 -> Interceptor前置 -> controller -> Interceptor 处理中-> Interceptor 处理后



1. 注入Bean情况不同
   1. 过滤器中可以自动注入
   2. 拦截器不能自动注入，可以手动处理

反射

定义：在运行状态中，对于任意一个类，都能知道这个类的属性和方法；对于任何一个对象，都能任意调用它的方法和属性；并且能够改变它的属性。

使用场景：编写灵活代码，代码可以在运行时装配，降低代码耦合度；动态代理实现，AOP等

得到Class的三种方式：

1. 通过调用对象 的 getClass()接口：person.getClass()
2. 直接通过类名.class方式：Person.class
3. 通过Class对象的静态方法forName()获取：Class.foeName(“com.math.BigDecimal”)

JAVA虚拟机

垃圾回收算法：

1. 标记清除：首先标记将要清除的对象，标记后统一回收被标记的对象

不足：效率问题，标记和清除两个阶段效率不高；空间问题，容易产生大量不连续的内存碎片；空间碎片太多会导致以后想要分配较大的内存对象时，不得不再次出发垃圾回收机制。

1. 复制算法：实现简单，性能高效；浪费一半的内存空间。PS：在对象存活率较高的情况下，效率会有所下降。商业虚拟机用此方式，回收新生代
2. 标记整理：在标记清除方法的基础上，将内存放在一起
3. 分代收集算法：将内存分为几块，不同周期的内存采用不同的方法，也就是上述算法的组合

HotSpot算法实现，可作为GCRoots（本身为引用）的对象包括“

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中（Native方法）引用的对象

垃圾回收器

1. Serial收集器

标记-复制；单线程；

1. ParNew收集器

Serial的多线程版本，能与CMS合作

1. Parallel

指多条垃圾数据线程并行工作，但此时用户线程仍然处于等待状态

1. Parallel Scavenge收集器

新生代收集器，复制算法收集器。

目标：达到可控制的吞吐量，吞吐量为CPU用于运行代码的时间与CPU总消耗的时间的比值。即 吞吐量= 运行代码时间 /（运行代码时间+垃圾回收时间）

高吞吐量可以高效的利用CPU时间，主要是和在后台运算而不需要太多交互的任务

精确控制吞吐量参数：

1. 控制最大垃圾回收时间 –XX:MaxGCPauseMillis
2. 直接设置 吞吐量大小 -XX:GCTimeRatio
3. Parallel Old 收集器

本身有PS MarkSweep 收集器来收集老年代。Java8默认垃圾回收器：Parallel Old 收集器和Parallel Scavenge 收集器；注重吞吐量

1. CMS收集器

目标：获取最短回收停顿时间的收集器，现在很大一部分额java应用集中在互联网网站上或者B/S系统的服务端上，希望系统停顿时间最短。CMS适用于这样的应用的需求

算法：标记-清除

步骤：

1. 初始标记-stop the world：标记GC Roots能够直接引用的对象，速度很快
2. 并发标记：GC Roots Tracing的过程（基于JVM中活跃的引用进行搜索，看是否有可达对象，那些不可达的对象，便认为没有用了）
3. 重新标记-stop the world：修正并发标记期间由于用户继续运行而导致标记产生变动的那一部分的标记记录
4. 并发清除

时间最长的是并发标记和并发清除，收集器可以与用户线程一起工作。整体而言，CMS垃圾回收期的内存回收过程可以与用户线程一起并发执行

缺点：

1. 对CPU资源敏感。默认启动回收线程数为（CPU+3）/4
2. 无法清理浮动垃圾（并发清理时产生出来的垃圾）
3. 由于采用的是并发-清除算法，会产生大量内存碎片。默认开：-XX: UseCMSCompactAtFullCollection 会在fullgc时开启内存碎片整理过程；无法并发，碎片没了，但停顿时间长
4. G1

Jdk9中，默认垃圾回收器、

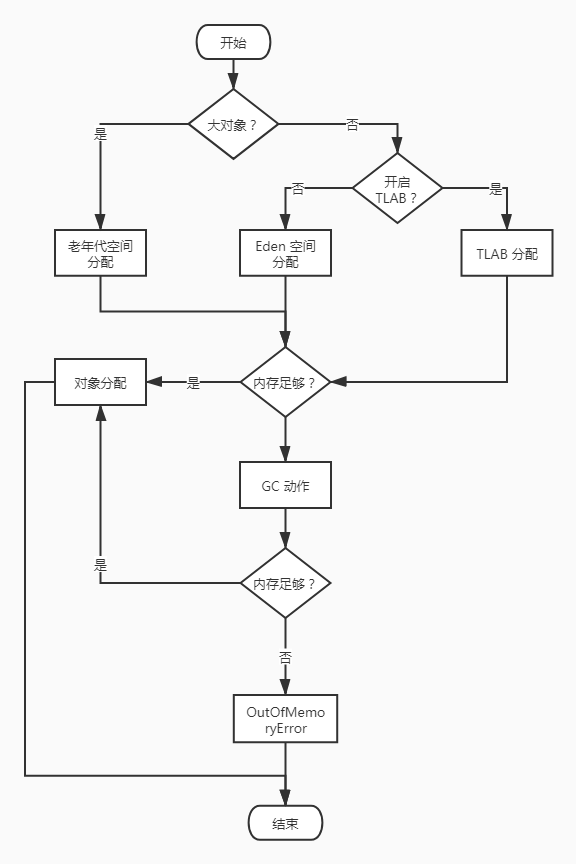
G1将新生代、老年代的物理空间划分取消，不用担心哪一个代空间不够用

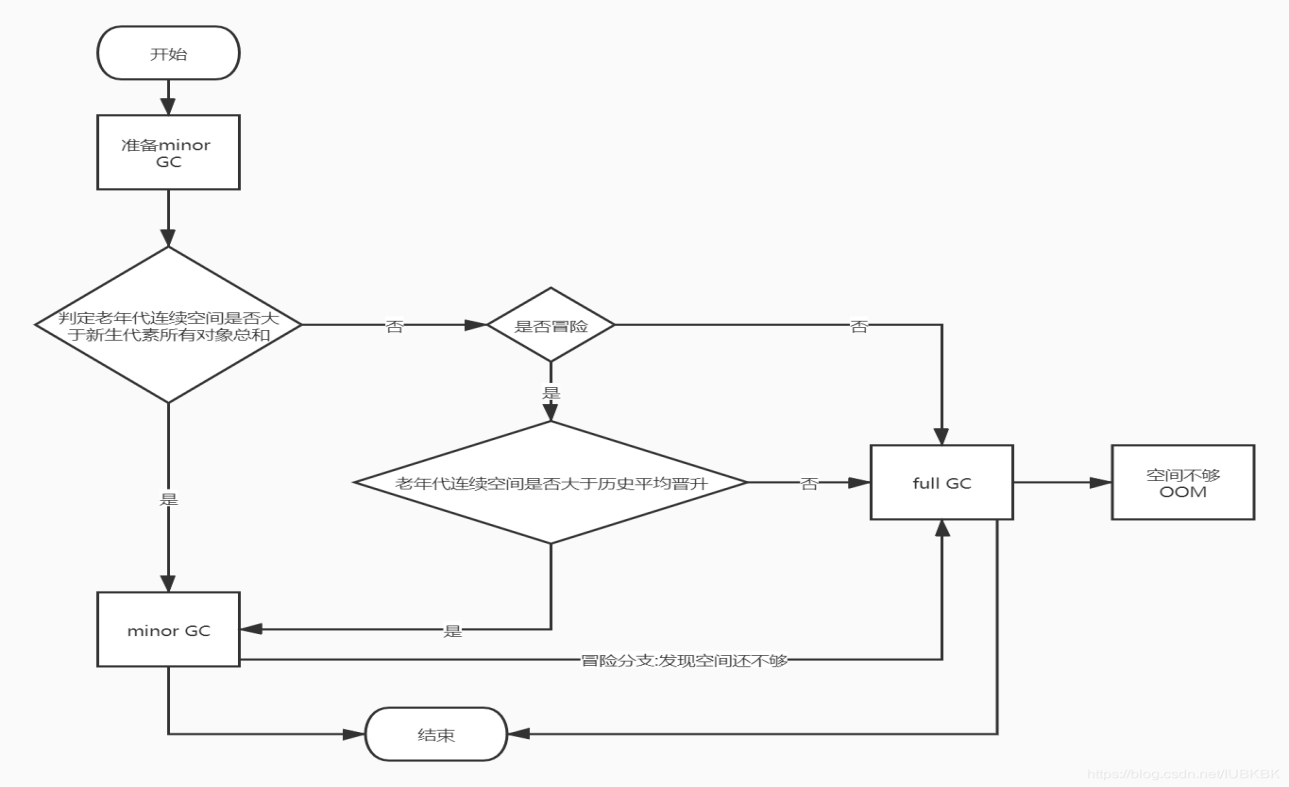
G1算法将堆分成若干个区域（Region）

GC模式：Young GC，Mixed GC，两种都需要STW

对象分配策略

TLAB：本地线程分配缓冲，每个线程在Java堆上预先分配的一小块内存。线程私有，没有锁开销，性能好，jdk7之后默认开启





系统运行越来越快

Java文件便已成为.class文件，再通过解释器将字节码转换成本地机器码才能运行，为了节约内存和执行效率，代码最初被执行时，解释器会先解释执行这段代码。

当虚拟器发现某个方法或代码块运行特别频繁时，就会把这些代码认定为热点代码（HOT SPOT CODE），在运行时，虚拟机会通过即时编译器JIT对这些代码优化，存储在内存中，之后每次运行代码时，直接从内存获取

OOM内存溢出：

OutOfMemoryError异常

堆溢出 JavaHeap spaces

Java堆用于储存对象，不断创建对象，并保持GCRoot到对象之间有可达路径

虚拟机栈和本地方法栈溢出

线程请求栈的深度大于虚拟机允许的最大深度（StackOverFlowError ）无限递归

如果虚拟机再扩展时无法申请足够的空间，则会抛出OutOfMemoryError异常

运行时常量池溢出

比如String对象包含的字符串添加到常量池

方法区溢出

方法区用于存放Class的相关信息，例如类名、访问修饰符、常量池、字段描述、方法描述等，在经常需要动态生成大量class应用中，要特别注意

引用：

强引用：使用最普遍的引用，一个对象如果具有强引用，则不会被回收。当内存不足时，Java虚拟机会抛出OOM，也不会回收这些对象

软引用：用于描述一些有用但并不是必须的对象，只有在内存不足的时候才会回收这些内存。比较适合实现网页缓存，图片缓存等。

弱引用：jvm进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。在java中，用java.lang.ref.WeakReference类表示，可以在缓存中使用弱引用。

虚引用：主要用来追踪对象被垃圾回收的活动。

Java 基础

1，基本类型：

自动类型转换（jvm自动完成）：从范围小到范围大的自动转换

byte -> short(char) -> int -> long -> flout -> double

所以，short k = 1; k = k+ 1 是错的。因为k+1的时候会向上转换成int，但int转为short需要强制转换。

2，重写和重载

重写：对父类允许访问的方法进行重新编写，返回值和形参都不能改变

子类与父类多态的表现

好处：子类可以根据需要，定义自己的行为

方法重写规则：

1. 参数列表必须相同
2. 返回类型可以不同，但必须是父类返回值的派生类（子类）
3. 访问权限不能比父类中被重写的方法的访问权限更低
4. 父类的成员方法只能被子类重写
5. Final 的方法不能被重写
6. Static方法不能被重写，但能够被再次声明
7. 子类和父类在同一包中，子类可重写父类除private，final的所有方法
8. 不在同一包中，只能重写父类public和protected 的非final方法
9. 构造方法不能被重写
10. 不继承，则不能重写此类的方法
11. 重写方法可以抛出非强制异常。强制异常只能抛出范围更窄的

重载：在一个类里，名字相同，但参数不同。返回类型可以相同也可以不同

一个类的多态表现

规则：

1. 被重载的方法必须修改参数列表（参数名或参数类型）
2. 被重载的方法可以改变返回类型
3. 被重载的方法可以改变访问权限
4. 被重载的方法可以声明新的或更广的异常

3，数组实例化：

Int[] I = {1,2,3}

Int[] I = new int[3]

Int[] I = new int[3]{1,2,3}

4，

1. Object的方法

Native方法：

1. getClass() 返回此时运行Object的类
2. hashCode() 返回散列值
3. clone()，Object的projected方法，用于克隆
4. notify() 唤醒此对象监视器上等待的单个线程
5. notifyAll() 唤醒此对象监视器上等待的所有线程
6. wait() 当前线程等待

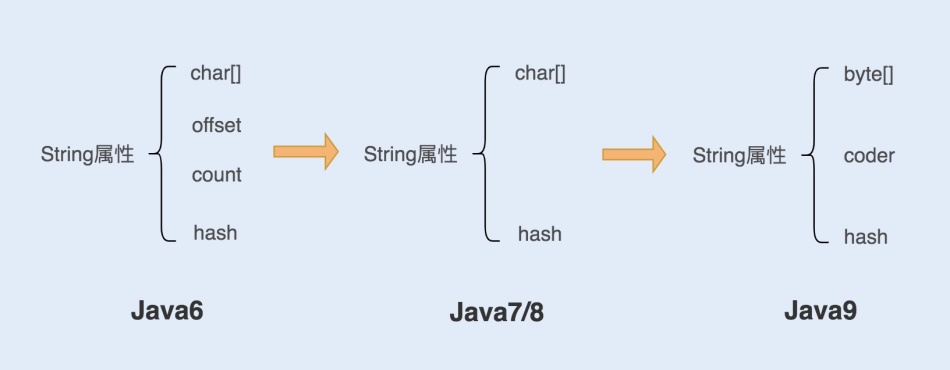
JAVA方法：

1. equals() 判断对象是否相等
2. toString() 默认返回ToStringExample@1312AA2 ；@后面是散列码的无符号十六进制
3. finalize() 垃圾回收时会调用
4. 值传递与引用传递

|  |  |  |
| --- | --- | --- |
|  | 值传递 | 引用传递 |
| 根本区别 | 创建副本（copy） | 不会创建副本 |
| 结果 | 方法中不会改变原始对象 | 方法中会改变原始对象 |

1. 高效使用String

String对象优化过程：



A，1.6版本：String通过offset和count两个属性来定位char[]数组，获取字符串，这么做可以高效快速共享数组对象，同时节省内存空间，但容易造成内存泄露

B，1.7、1.8版本：减少offset和count变量，好处是String对象占用内存稍微减少了，String.subString()方法也不在共享char[]，从而解决了使用该方法导致的内存泄漏的问题。

Substring()好处:

调用substring()时无需复制数组，可重用value[]数组；且substring()的运行是常数时间，非线性，性能得到提高

坏处：

内存泄漏：由substring(start,end)调用构造函数String(int,in,char[])时，实际上是改变offset和count的位置达到取得子字符串的目的，而子字符串里的value[]数组，仍然指向原字符串。假设原字符串s有1GB，且我们需要的是s.substring(1,10)这样一段小的字符串，但由于substring()里的value[]数组仍然指向1GB的原字符串，导致原字符串无法在GC中释放，从而产生了内存泄露

C，1.9开始，将char[]改成byte[]，又维护另一个属性coder，它是一个编码格式的标识

coder作用：在计算字符串长度或使用indexOf方法时，根据此字段判断如何计算长度；coder=0代表Latin-1（单字节编码），1代码utf-16；如果String字符串只包含Latin-1，则coder=0，反之为1

char变byte因为char字符占16位，2个字节，存储单字节的字符串就浪费内存

String不可变性

String类被final修饰；变量char[]也被final修饰。表示不可更改。

好处：

1. 保证String对象安全性。防止被恶意修改
2. 保证hash属性值不会频繁变更，确保了唯一性
3. 可以实现字符串常量池。

使用String s = “abc” 这种方式创建字符串的时候，jvm首先会检查该对象是否在字符串常量池中，如果存在直接返回对象引用，否则新的字符串将在常量池中被创建。这种方式可以减少同一个值的字符串对象重复创建，节约内存。

使用String s = new String(“abc”)这种方式，首先在编译类文件时，”abc”常量字符串将会放到常量结构中，在类加载时，字符串会在常量池中创建；其次，在调用new时，jvm命令先会调用String的构造函数，同时引用常量池中的字符串，在堆内存中创建一个String对象；最后，将str引用String对象

String对象优化：

1，构建大字符串

字符串变量累计，使用+号作为字符串的拼接，也一样被编译器优化成StringBuilder的方式。所以写代码的时候最好显示写StringBuilder。

2，使用String.intern节省内存

String a =new String("abc").intern();String b = new String("abc").intern(); if(a==b) { System.out.print("a==b");}

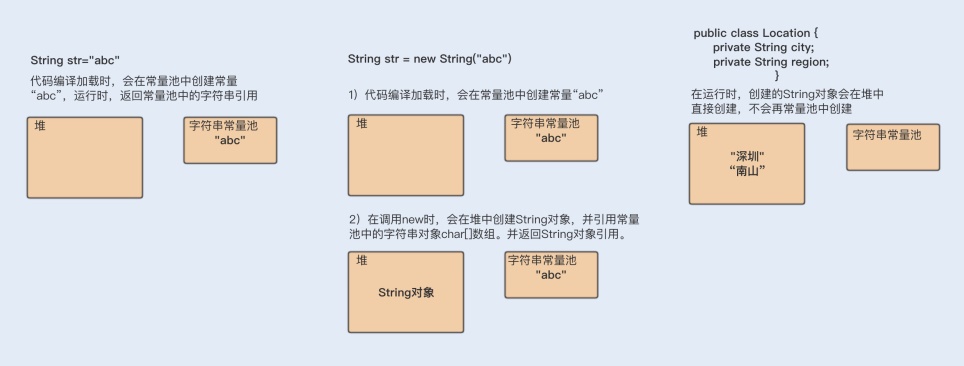
输出结果为 a==b

在字符串常量中，默认将对象放入常量池；在字符串变量中，对象会创建在堆内存中，同时也会在常量池中创建一个字符串对象，String对象中的char数组将会引用常量池中的char数组，并返回堆内存对象引用

如果调用intern方法，会去查看字符串常量池中是否有等于该对象的字符串的引用，如果没有，1.7版本以后（常量池合并到堆中），将首次遇到的字符串的引用添加到常量池中；如果有，就返回常量池中字符串的引用

使用intern要注意，因为常量池的实现类似于一种HashTable的实现方式，Hash Table数据越大，遍历时间越长，数据过大会给字符串常量池造成负担

String创建对象分配内存地址过程：

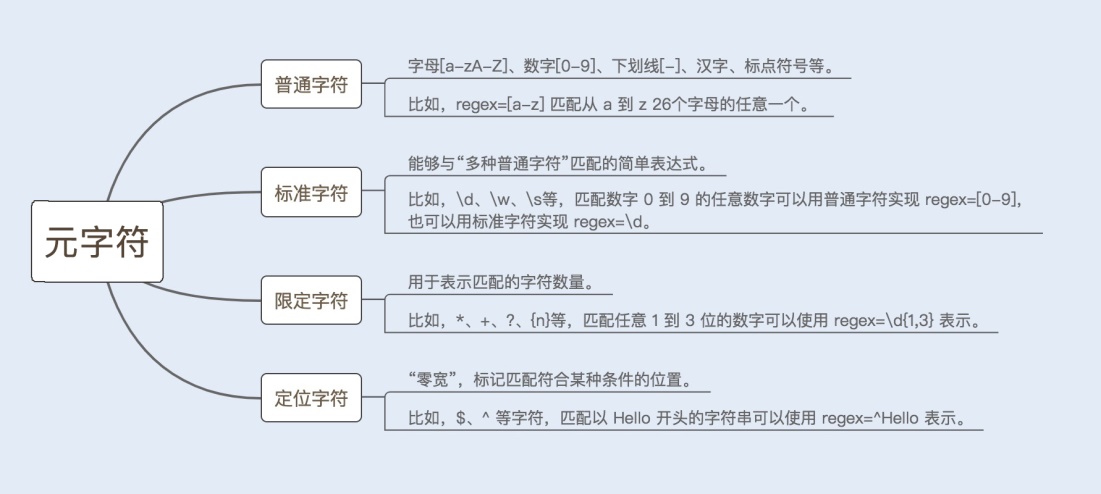


字符串分割：

尽量避免使用split（使用正则），容易发生回溯问题，String.indexOf方法

1. 正则表达式

组成：普通字符，标准字符，限定字符（量词），定位字符（边界字符）



引擎：DFA自动机（确定有限状态自动机）和NFA（非确定有限状态自动机）

减少回溯：NFA的贪婪特性是导火索

贪婪模式：正则表达式会匹配尽可能多的内容

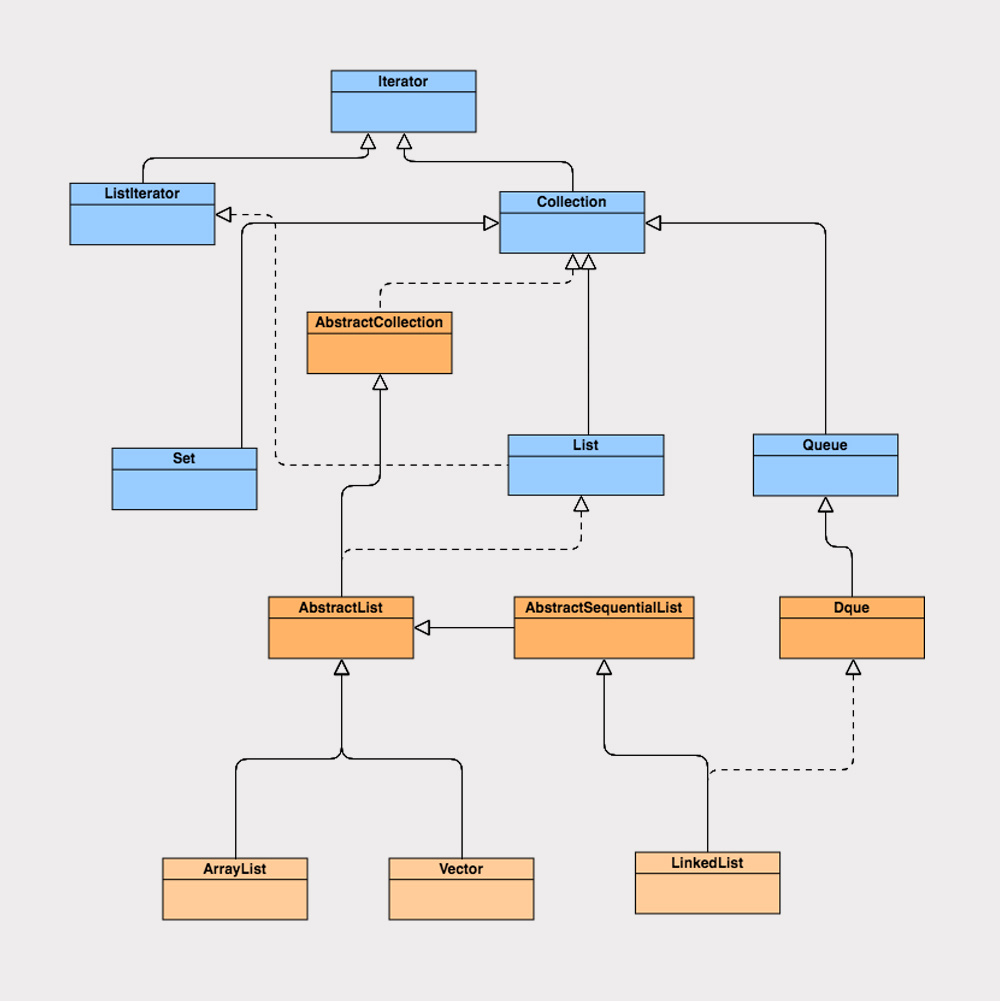
懒惰模式：正则表达式会尽可能少的重复匹配字符（不能完全避免回溯）

独占模式：同贪婪模式一样，独占模式会最大限度的匹配多的内容；不同的是，独占模式下，匹配失败就会结束匹配，不会发生回溯问题

优化：

1. 少用贪婪
2. 减少分支选择（X|Y|Z）
3. 减少捕获嵌套
4. 集合

List集合



**ArrayList** 继承 **AbstractList，**底层数组并且实现了自动扩容；实现了Cloneable接口和 Serializable接口，可以实现克隆和序列化；实现了RandomAccess接口（标志接口，实现此接口的List可以实现快速随机访问）

属性

//默认初始化容量

private static final int DEFAULT\_CAPACITY = 10;

//对象数组

transient Object[] elementData;

//数组长度

private int size;

扩容：1.5

问题1：数组elementData用transient修饰，表示该属性不会被序列化。但文档却没有说，为什么？

因为ArrayList基于数组的方式实现，数组动态扩充，所以不是所有分配空间的内存都有数据。如果采用外部序列化法实现数组的序列化，那么会序列化整个数组。ArrayList 为了避免这些没有存储数据的内存空间被序列化，内部提供了两个私有方法 writeObject 以及 readObject 来自我完成序列化与反序列化，从而在序列化与反序列化数组时节省了空间和时间。

因此使用 transient 修饰数组，是防止对象数组被其他外部方法序列化。

问题2：ArrayList在哪种情况下增加元素不会影响性能?

知道数组大小并指定容量，在不再发生扩容时，不影响性能，反而更好

问题3：如何remove？

使用Iterator it = list.iterator();而不是for中的remove

在foreach循环中调用list中的remove()方法，会走到fastRemove()方法，该方法不是iterator中的方法，而是ArrayList中的方法，在该方法只做了modCount++，而没有同步到expectedModCount。 当再次遍历时，会先调用内部类iteator中的hasNext(),再调用next(),在调用next()方法时，会对modCount和expectedModCount进行比较，此时两者不一致，就抛出了ConcurrentModificationException异常

For中在remove操作时使用的是原始数组list的remove，而不是迭代器的remove。这样就会造成modCound != exceptedModeCount，进而抛出异常。

**LinkedList** 继承 **AbstractSequentialList**实现了List, Deque ,Cloneable接口和 Serializable接口

定义了一个Node结构，元素内容item，前指针和后指针（pre/first，next/last）

private static class Node<E> {

E item;

Node<E> next;

Node<E> prev;

Node(Node<E> prev, E element, Node<E> next) {

this.item = element;

this.next = next;

this.prev = prev;

}

}

好处：

1. first/last属性能更好的表达链表表头和链尾的概念
2. first/last LinkedList初始化节省new一个新的Entry（1.7之前会创建空的Entry来做header，前后指针指向自己做循环链表）
3. first/last 最重要的性能优化是头和尾的插入删除操作更快捷

并发容器：



1. Comparable和Comparator

两者都是接口，两者同时存在则Comparator优先级高

Comparable表示可被排序，实现此接口的类的对象自动拥有排序功能；这个排序称为类的自然顺序；这个类对象的列表可以通过Collections.sort和Arrays.sort来执行排序

Comparator则表示一个比较器，可作为一个参数传递到Collectoins.sort和Arrays.sort中

Spring、Spring boot 和 Spring cloud

1. 重要注解
   1. @SpringBootApplication：@Configuration 和 @EnableAutoConfiguration 和 @ComponentScan 组合
   2. @Configuration：允许在上下文中注册额外的bean或导入其他配置类
   3. @EnableAutoConfiguration：启用SpringBoot的自动配置机制；自动装配核心注解
   4. @ComponentScan：扫描被@Component(Service、Controller)注解的bean，默认会扫描启动类所在的包下所有的类，可以自定义不被扫描的bean。
2. 自动装配

AutoConfigurationImportSelector：加载自动装配类

方法：

selectImports() 获取所有符合条件的类的全限定类名，这些类需要被加载到ioc容器中

getAutoConfigurationEntry() 加载自动配置类，并返回给上一方法

1. 判断自动装配开关是否打开

默认：spring.boot.enableautoconfiguration = true

1. 获取EnableAutoConfiguration注解中的exclude和excludeName
2. 获取需要自动装配的所有配置类，读取 META-INF/spring.factories
3. 筛选，@ConditionOnXXX中的所有条件都满足，该类才会生效

并发编程

1. 线程：操作系统能够进行运算调度的最小单位。
2. 实现Java线程：
   1. 继承Thread类
   2. 实现Runnable接口，重写run方法
3. Start方法和run方法的区别：

Start方法被用来启动新创建的线程，而且start内部调用了run方法，这和直接调用run方法不一样。当只调用run方法的时候，只会在原来的线程中调用，没有启动新的线程，start方法会启动新的线程

1. Runnable和callable区别：

Callable的call方法有返回值，并且可以抛异常，而Runnable的run方法没有。Callable可以返回装载有计算结果的Future对象

要使用ExecutorService.submit方法调用。此方法会产生Future对象。

1. Volatile关键字

特殊修饰符，只有成员变量才可以使用它。

用于确保将变量的更细操作通知到其他线程。当把变量声明为volatile后，编译器和运行时都会注意这个变量时共享的，因此不会将此变量的操作与其他内存操作一起重排序。Volatile变量不会被缓存在寄存器或对其他处理器不可见的地方。因此在读取volatile类型的数据时候总会返回最新写入的值。

1. ThreadLocal

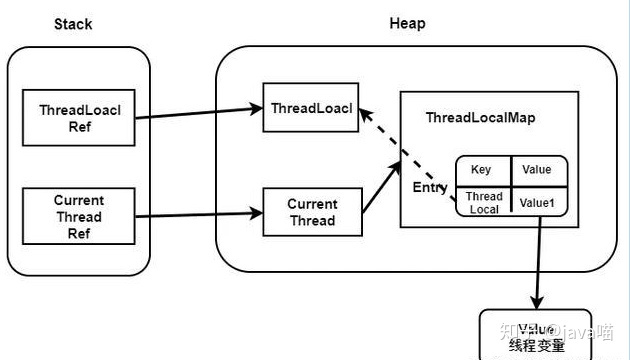
内含有：ThreadLocalMap，是一个内部类。

功能：为每一个线程创建单独的副本

ThreadLocal导致内存泄漏。

内存泄漏：不再会被使用的对象或者变量的内存无法被回收。

ThreadLocalMap中的key为弱引用的ThreadLocal实例，value为线程变量的副本



因为key为弱引用，，如果ThreadLocal不存在外部的强引用，那么key会被GC，到时此时ThreadLocalMap中的key为null。而value存在这强引用，只有在thread线程退出后，value的强引用链条才会断。如果迟迟不结束，则这些value无法回收。

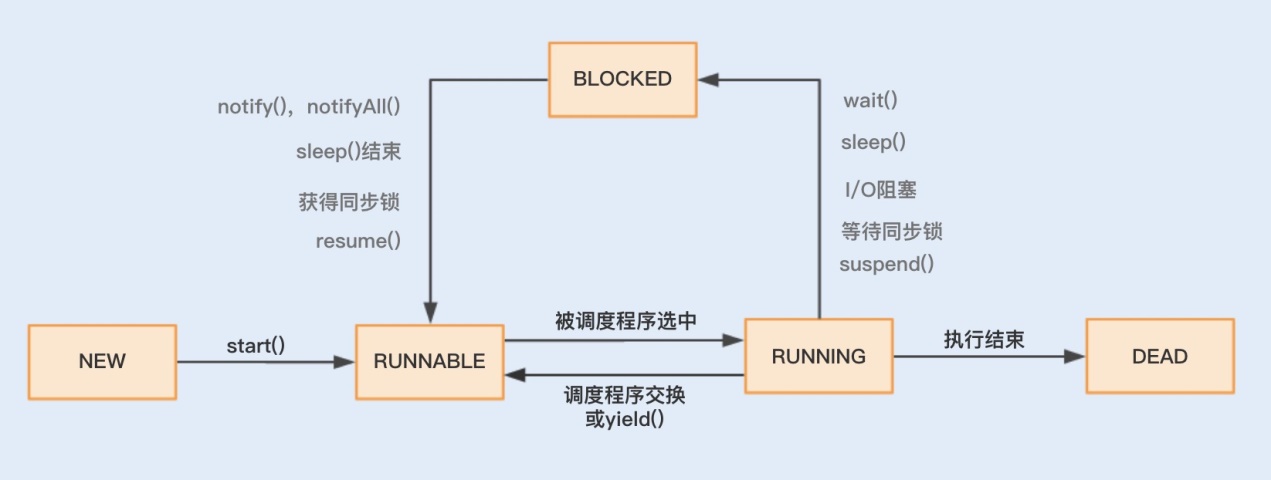
内存泄漏根源：ThreadLocalMap的生命周期和Thread一样长，若没有手动删除对应key，value就会导致内存泄漏

使用注意：

每次使用完Thread Local都要调用他的remove方法

将ThreadLocal变量定义成private static ，这样就一直存在这强引用，保证任何时候都能通过Thread Local的弱引用都能访问到Map中的Entry的value值，进而清除掉。

1. 线程



状态：NEW，RUNNABLE，RUNNING，BLOCKED，DEAD

由RUNNABLE转化为非RUNNABLE的过程就是线程上下文切换

由RUNNING转化为BLOCKED再转化为RUNNABLE，被调度器执行，这就是上下文切换的过程

诱发机制：

自发性上下文切换（程序本身触发）

非自发性上下文切换（由系统或者虚拟机诱发）

自发性上下文切换关键字或方法：

Sleep():

Wait():

Yield();

Join():

Park():

Synchronized:

Lock:

非自发性上下文切换：常见的有 线程被分配的时间片用完、虚拟机垃圾回收导致、执行优先级问题导致

Synchronized

基于底层操作系统的Mutex Lock 实现存在用户态和内核态之间的切换，所以增加了性能的开销。

用户态：线程执行用户自己写的代码时处于用户态；权限有限

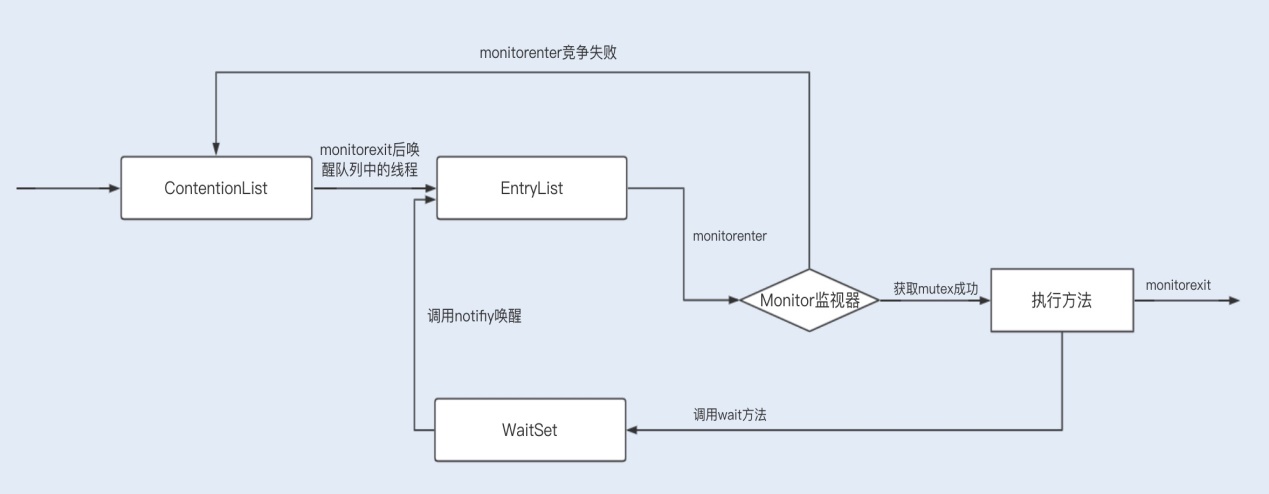
内核态：线程到了系统调用；cpu可以访问内存的所有数据包括外部设备（硬盘等）

用法：1，修饰方法；2，修饰方法块

修饰代码块，是由monitorenter和monitorexit指令实现同步。Enter后线程持有monitor对象，退出则释放此对象

修饰方法，则会出现ACC\_SYNCHRONIZED标志，jvm通过此标志一方法是否为同步方法

每一个对象实例都会有Monitor，monitor可以和对象一起创建、销毁。由C++实现。



多线程访问统一代码块时，多个线程会被存放在contentionList和entryList集合中，处于block状态的线程都会在此集合中。调用wait方法后，线程会释放mutex，进入waitSet中。

1. 线程池

线程池状态：



Running：一旦被创建（初始化）就是此状态，可以接受新任务

Shutdown：调用线程池shutdown接口。不接受新任务，但可以处理已经添加的任务

Stop：调用shutdownNow接口。不接受新任务，不处理已添加的任务。中断正在处理的任务

Tidying：所有任务已终止，线程池会变成tidying状态，此时会调用terminated方法。

Terminated：线程池彻底终止

线程池重要参数：

Corepoolsize：核心线程数

MaxPoolSize：最大线程数

keepAliveTime：除核心线程外的线程最大保留时间

util：上述时间的单位

workQueue：等待队列

threadFactory：创建线程的工厂

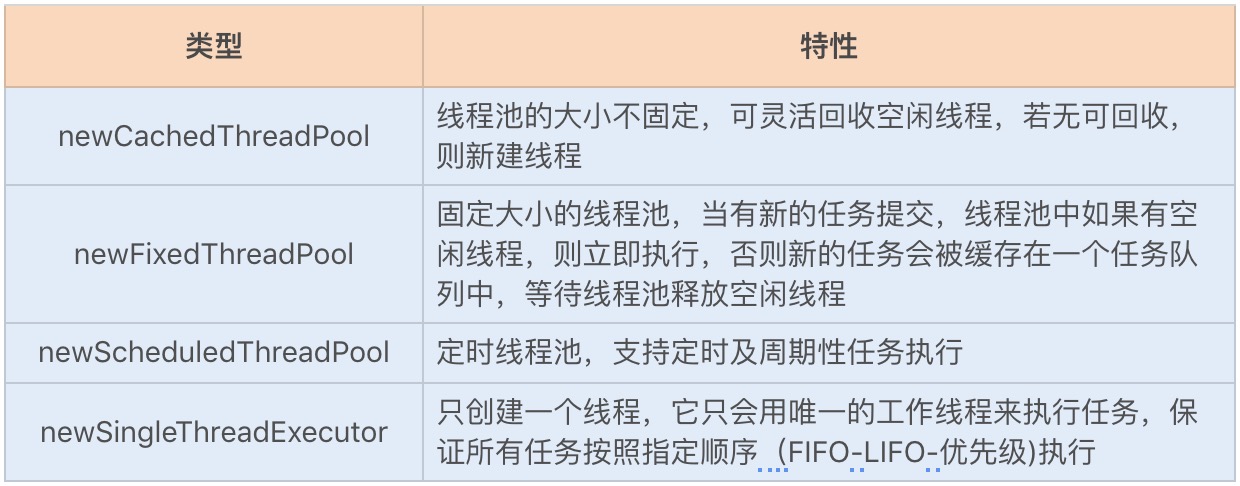
rejecthandler：拒绝策略；实现RejectedExecutionHandler接口可以自定义拒绝策略

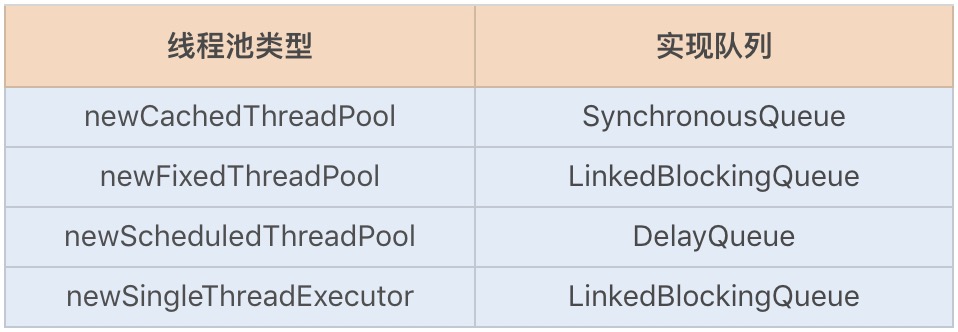
线程池中的submit和execute区别：

Submit：适用于需要关注返回值的场景

Execute：不要关注返回值，将线程直接放入线程池中执行即可

Executors实现了一下四种ThreadPoolExecutor：





阻塞队列：

ArrayBlockingQueue：基于数组结构实现的有界阻塞队列，按照FIFO排序，使用ReenTrantLock、Condition来实现线程安全；

LinkedBlockingQueue：基于链表结构的阻塞队列，吞吐量比ArrayBlockingQueue高

PriorityBlockingQueue：具有优先级的无线阻塞队列，基于二叉堆结构实现

DelayQueue：支持延时获取元素的无界阻塞队列，基于无线阻塞队列的扩展，并且实现了Delay延时接口

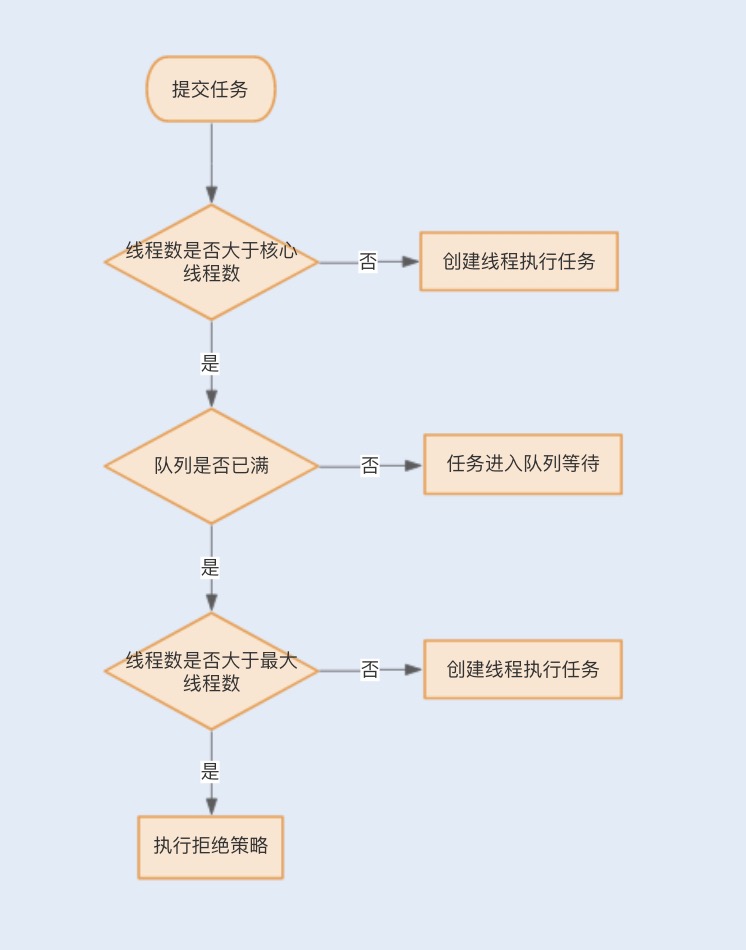
SynchronousQueue：不存储多个元素的阻塞队列，每次放入数据时，必须等待相应的消费者取走数据才可以放入

非阻塞队列：

ConcurrentLinkedQueue：无界线程安全FIFO队列，基于链表实现，利用CAS实现线程安全

线程池预热：调用prestartAllCoreThread()或prestartCoreThread()方法，提前创建等于核心线程数的线程数量，在抢购场景中会用到

线程池线程分配流程：



计算线程数量：

CPU密集型：这种任务主要消耗CPU资源，可以将线程数设置为N（CPU核心数）+1，加一是防止线程偶发中断，充分利用CPU资源

IO密集型：线程数设置为2N，因为IO是不消耗CPU资源的

Java对象头

Jdk1.6中，对象实例在堆内存中分为三部分：对象头、实例数据和对齐填充；对象头由Mark Word、指向类的指针和数组长度 三部分组成

Mark Word记录了对象和锁有关的信息



1. 偏向锁

优化同一线程多次申请同一个锁的竞争。例如：创建一个线程并在线程中执行循环监听的场景，同一线程每次都需要获取释放锁，每次操作都会发生用户态和内核态的切换

作用：当一个线程再次访问此代码块，可先从mark word中判断线程Id，无需再进行monitor竞争

缺点：当偏向锁升级时会发生stop the world，所以在高并发场景下这样会带来大量性能开销。

如何改进：

-XX:-UseBiasedLocking //关闭偏向锁（默认打开）

-XX:+UseHeavyMonitors //设置重量级锁

1. 轻量级锁

当其他线程竞争获取此锁时，通过CAS将对象头中的线程Id改完自己的线程Id，如果成功则保持偏向锁，如果失败，则升级为轻量级锁

1. 自旋锁和重量级锁

轻量级锁CAS失败，线程会处于阻塞状态。如果正在持有锁的线程在很短时间内释放资源，那么进入阻塞状态的线程又要开始竞争锁资源。

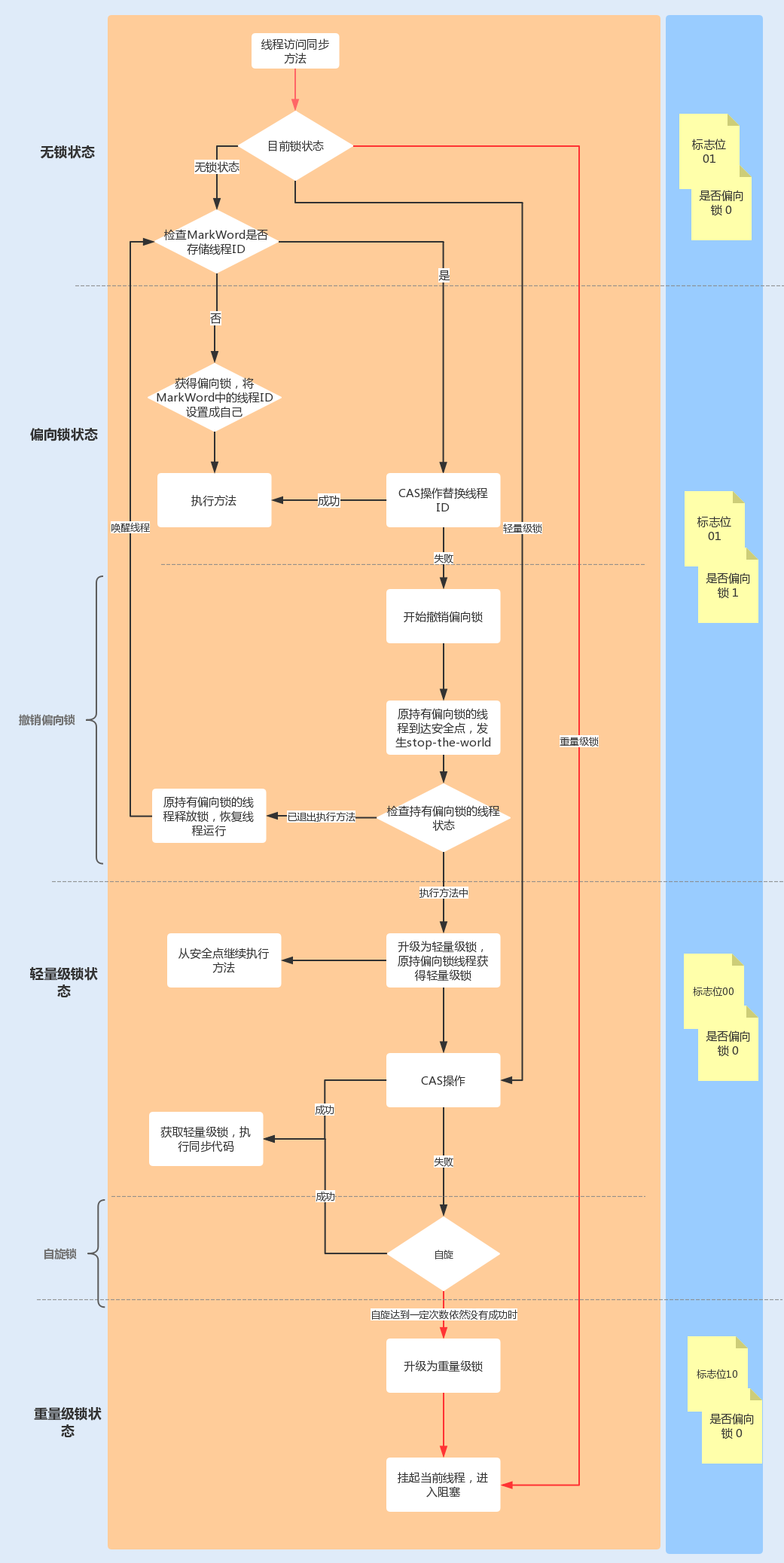
Jvm提供自旋锁，可以通过自旋不断获取资源，避免线程挂起阻塞。自选次数由JVM参数决定。注：在锁竞争不激烈且占用锁时间短的条件下，自旋锁可以提高性能；但设置过多则会导致长时间占用CPU

高并发高负载场景下可以关闭优化系统性能

-XX:-UseSpinning //参数关闭自旋锁优化(默认打开)

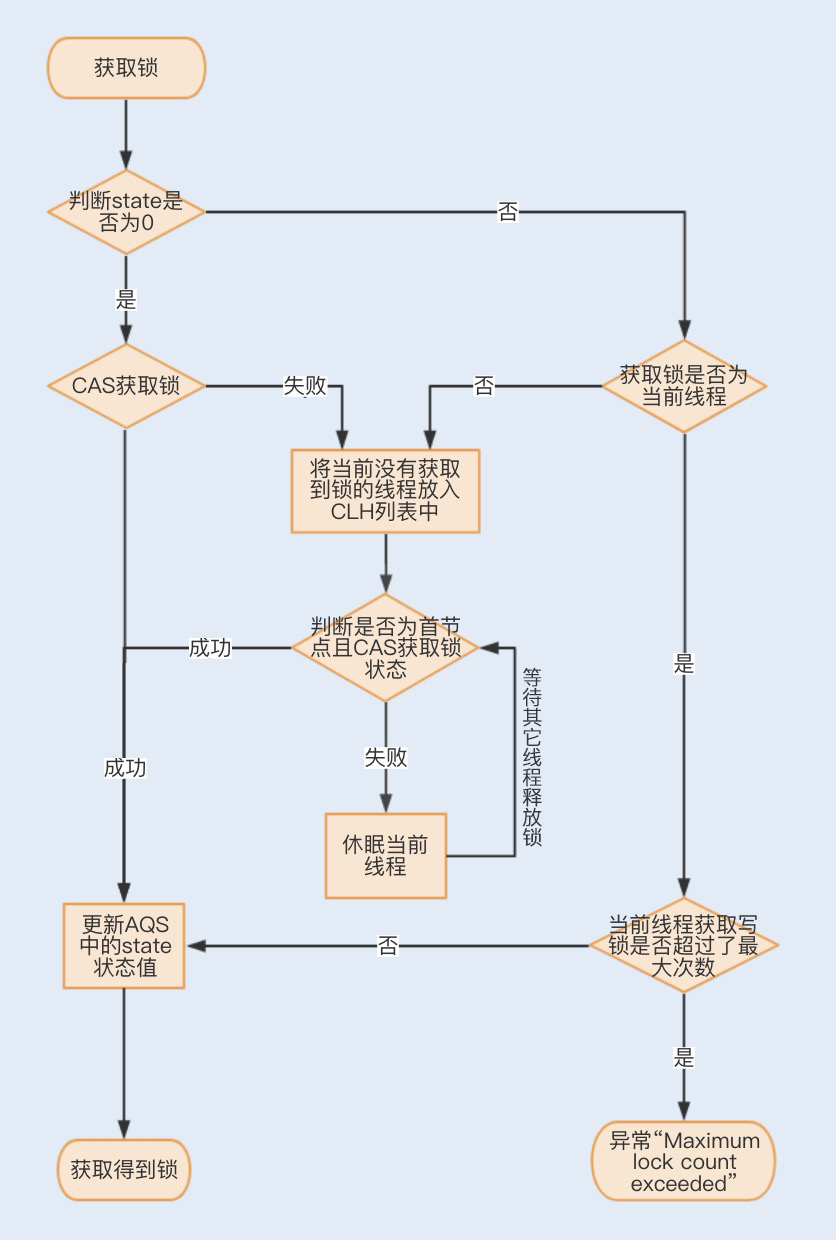
-XX:PreBlockSpin //参数修改默认的自旋次数。JDK1.7后，去掉此参数，由jvm控制

自旋抢锁失败后，同步锁升级为重量级锁，所标志位为10。在这个状态下，为抢到所的线程都会进入monitor，之后被阻塞到



Lock锁：

基于Java实现的锁；Lock是一个接口，常用的实现类有ReenTrantLock、ReenTrantWriteLock（RRW），他们都是依赖于AQS实现。AQS类结构包含一个基于链表实现的等待队列（CLH队列，队列操作皆为CAS操作），用于存储所有阻塞的线程；state变量，表示加锁状态。

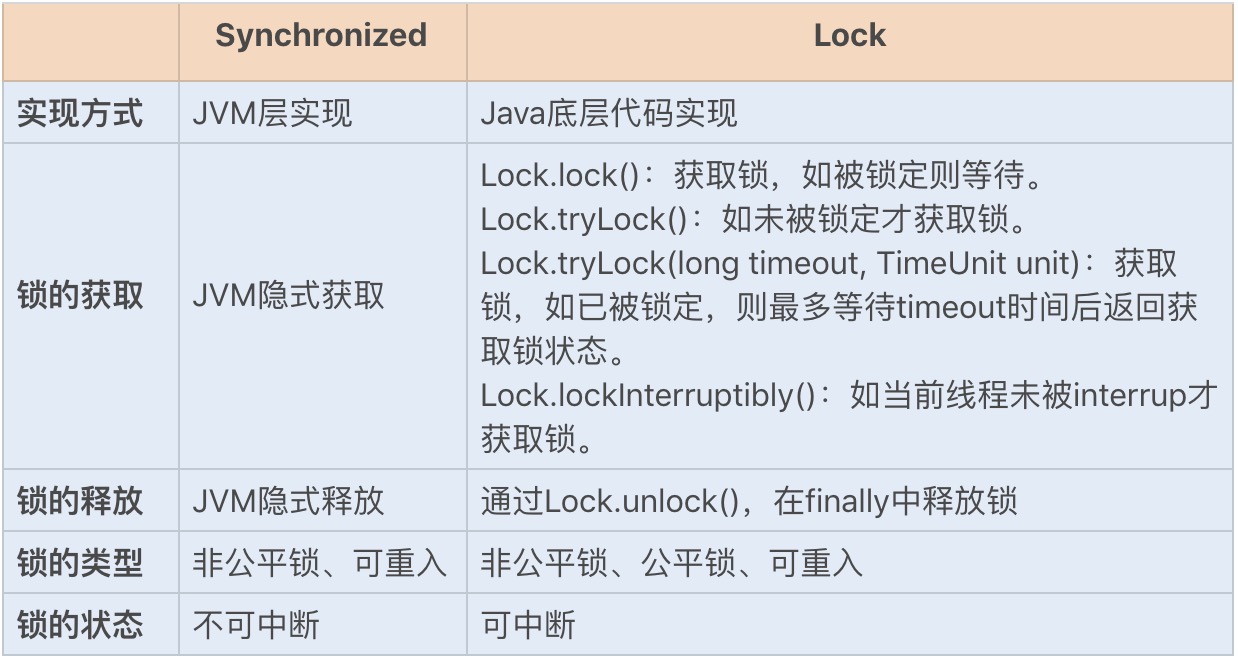


ReenTranLock是独占锁；RRW允许多个读线程同时访问，通过state变量高低位控制，高位表示读，低位表示写。

锁饥饿：写线程会因为迟迟无法获取到锁而一直处于等待状态

StampedLock解决了饥饿问题。

劣势：不支持重入，不支持条件变量，线程被中断可能导致CPU暴涨



当一个线程获得锁，并进入方法A，在方法A没有释放锁的情况下，该线程能否进入需要此锁的方法B

可重入锁：可以进入B方法

不可重入锁：不可以进入B方法

乐观锁

JDK中concurrent包中，atomic下的类都是基于CAS实现的。会依赖本地方法Unsafe类，类中的操作方法会调用CPU底层指令实现原子操作。处理器提供**总线锁定**和**缓存锁定**两个机制来保证复杂内存原子性的操作

总线锁定：操作共享资源时，总线会发出一个Lock信号，这时其他处理器就不能操作这个共享资源了。但是会造成性能开销

缓存锁定：当某个处理器对缓存中的共享变量进行操作，就会通知其他处理器放弃存储该共享资源或者重新读取该共享资源

ABA问题，使用AtomicStampedReference或StampdLock等，通过版本号或者时间戳解决此问题

**性能：**

影响性能的因素：

CPU：大量计算的应用会长时间占用CPU资源，导致其他资源无法争夺CPU而相应缓慢。

例如，代码递归导致无限循环；正则表达式引起的回溯；jvm频繁的fullgc；多线程编程造成的大量上下文切换；

内存：Java程序一般在jvm中对内存进行管理，主要是用堆来存储Java创建的对象。当内存空间被占满，对象无法回收，会导致内存溢出，内存泄露等问题

磁盘IO：磁盘的读写比内存慢很多

网络：网络对于性能有着至关重要的作用。带宽过低，对于传输数据比较大、或者并发量比较大的情况，网络很容易变成瓶颈

异常：Java应用中，抛出异常需要构建异常栈，对异常捕获或处理，很消耗性能。尤其是在高并发情况持续的异常处理。

数据库：数据库IO等等，影响性能

锁竞争：并发编程中，多个线程共享读写同一个资源，为了保证其原子性，我们会用到锁。锁的使用可能会造成上下文切换。为了提高性能，对jvm内部锁已经做了多次优化（jdk1.6以后）。例如增加了偏向锁、自旋锁、轻量级锁、锁粗化、锁消除等