# 1、静态代理，JDK代理和CGLIB代理

[httSs://blog.csdn.net/sinat\_38393872/article/details/93781299](https://blog.csdn.net/sinat_38393872/article/details/93781299)

静态代理的出现是因为为了保证开闭原则，不修改A类中A.f函数源码，而新创建一个静态代理类S，在S中传入一个A的对象，再定义一个S.f函数，在S.f函数中调用A类的A.f函数，在S.f函数中进行修改操作（一般是在A.f前和后）。

JDK动态代理的出现是因为当A类中有多个方法都要进行相同的修改时，那么S当中就要定义多个S.f(n)函数，那么就会有大量重复代码。而使用动态代理的方式，只需要代理这个类，就能给这个类中的所有方法进行相同的修改。

还有就是当有多个类中的方法需要进行相同的修改时，使用动态代理的方法，毫无疑问要用到多个静态代理类，而且其中还包含大量重复代码，而使用动态代理的方式，只需要定义一个动态代理类，只需传入不同的A类就可。

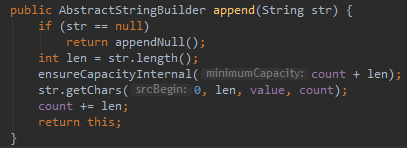
JDK和CGLIB一样，只是实现方式不同。

# 2、StringBuilder为什么线程不安全？StringBuffer为什么线程安全

<https://blog.csdn.net/weixin_38405253/article/details/100151578>

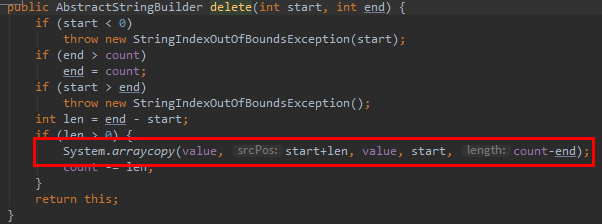
<https://blog.csdn.net/qq_37840807/article/details/97167566>

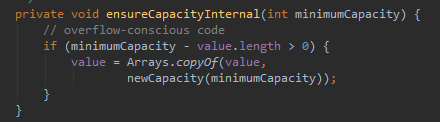
简单的来说StringBuffer类中的每个函数都加了synchronized关键字进行修饰，具体来说的话，以二者的append方法为例，当新建十个进程，每个进程里append 1000个元素，最终计数，按理来说应该最后输出10000个元素，但是以StringBuilder存储的输出往往不足10000个，原因是：其中存在两个线程同时进行了append，那么只会append一个元素，而且因为二者都是继承于AbstractStringBuilder类，调用的都是其的append方法，其中append的方法如下：

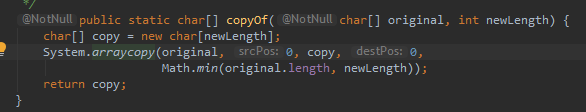


可以看到其中value[]和count是全局变量，前者存值，后者记录最后一个位置。当最后一个位置超过容量大小时就会发生扩容，扩容的操作就是新建一个数组，将旧数组的值都赋值给新数组。其中getChars方法是String中的方法，就是把新加的元素添加到value数组中，这里值得一提的是String、StringBuffer、StringBuilder底层都是利用数组进行实现的，前者的

数组用final修饰不可变，后两者可变，，其中后两者有一个指针count记录最后一个元素的位置，用来增加和删除元素。







线程不安全的StringBuffer会报数组越界的错误，原因是因为当进行到末尾时，假设两个线程对其进行了添加操作，第一个线程执行完毕，count大于了最大数组容量，而且还未对其扩容，那么第二个线程拿到的count就会超过数组容量，这个是够执行String函数的getchars函数进行添加元素时就会出现越界的情况，因为添加的元素是从count位置开始添加的。

# JVM栈

<https://blog.csdn.net/qq_28089993/article/details/90901511>

<https://blog.csdn.net/wb_snail/article/details/80610676>

**代码中的方法的调用在jvm中转换成字节码执行，而栈就是字节码指令执行的数据结构。**

**JVM中的栈包含四个部分：**

**局部变量表**：包含着方法参数和局部变量，但不是方法参数个数+局部变量个数 >= 局部变量表slot数量的大小,因为当pc计数器的值已经超出了某个变量的作用域时，下一个变量不必使用新的Slot空间，可以去覆盖前面那个空间。

**操作数栈**：将操作数入栈，当遇到运算符是，出栈进行计算，并将结果入栈。

**动态链接**：例如String a = “123”，a就是一个动态链接，而字符串常量却放在方法区中的运行时常量池中。其中编译期就刻意确定改方法有没有可能通过继承或者其他方式被重写，在java中静态方法(与类型直接关联)，私有方法(外部不可访问)，构造方法，父类方法，final方法，这五种方法的符号引用可以被静态解析都不可能被重写，可以在运行前确定唯一的调用版本，满足被静态解析的条件，称为非虚方法。其余的都是动态解析。

**返回地址**：正常退出时：方法退出的过程实际上就等同于把当前栈帧出栈， 因此退出时可能执行的操作有： 恢复上层方法的局部变量表和操作数栈， 把返回值（ 如果有的话） 压人调用者栈帧的操作数栈中， 调整PC 计数器的值以指向方法调用指令后面的一条指令。

一般来说， 方法正常退出时， 调用者的PC 计数器的值可以作为返回地址， 栈帧中很可能会保存这个计数器值。而方法异常退出时， 返回地址是要通过异常处理器表来确定的，栈帧中一般不会保存这部分信息。

前两者的大小在编译期间就已经确定，因此，栈帧数据结构的大小仅仅取决于Java虚拟机的实现。实现者可以在调用方法的时候给它们分配内存。

# JVM方法区

<https://blog.csdn.net/PORSCHE_GT3RS/article/details/82635053>

<https://blog.csdn.net/weixin_43320847/article/details/82950756>

JDK8之后方法区被分离出来到堆外内存，不会使用jvm内存，称为MetaStore元空间，但是还是和堆一样属于线程共享区，同理，栈、程序计数器组成线程独占区。元空间里存的是类信息，字节码文件（存放在常量池），常量（**分为字面量（字符串**

**Final常量）和符号引用**）、静态变量等共享数据。

移除的原因是：

1. 字符串存放在永久代中，容易发生性能问题和内存溢出，所以将字符串、 Interned Strings、静态变量存放于堆中的字符串常量池。但类信息(classes metadata)还是放到了 Metaspace 中。
2. 永久代垃圾回收效率偏低
3. 将HotPot和JRockit虚拟机合二为一（后者没有永久代概念。）

**常量池、运行时常量池、字符串常量池：**

**Jdk8后，常量池和运行时常量池存放在堆外内存（物理内存）元空间MetaStore中，字符串常量池放在堆中。**

**其中常量池存放文件常量，也就是字节码文件；**

**运行时常量池存放final常量，符号引用和静态变量，还有运行时常量，如String.intern()；**

**字符串常量存放字符串常量，主要用来存放新建的字符串常量，运行时常量字符串放在上面个。**

# 5、java多线程创建方法

<https://blog.csdn.net/sinat_33921105/article/details/105203972>

一般来说是有五种方法创建线程：

1. 直接实现Runnable接口，重写run方法，称为MyThread类

new Thread（new MyThread）.start()

1. 继承Thread类，直接重写run方法

new Thread（），其中还是有一个target的Runnable对象

1. 匿名内部类

new Thread( new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("匿名内部类的方式创建线程");  
 }  
 }).start();

1. 实现Callable接口，重写其中的run方法，但是却要这样：

//FutureTask继承自RunnableFuture，而后者继承自Runnable

FutureTask<Integer> futureTask = new FutureTask<>(new C());  
 // 传递的仍是一个Runnable对象  
 Thread thread2 = new Thread(futureTask);  
 thread2.start();

1. 采用线程池，就是存储上述方式创建的线程。

**综上：其实可以总结为一个方法，就是最终都要经过Thread这个类，传递给Thread类一个Runnable对象，然后调用Thread的start方法，有一个除外，无需传递Runnable对象，因为继承Thread类，自身携带target Runnable对象，其余的在构造函数时会调用init函数，该函数会将传入的Runnabe对象覆盖target对象。**

**关于你明明重写的是run方法，为什么调用的确实start方法。**

（1）start()方法是来自线程的，真正的实现了多线程的运行，而不用等到run()方法执行完毕之后再执行下面的代码段：

通过调用Thread类的**start()方法来启动一个线程，这时此线程是处于就绪状态**，并没有运行，**其实就是将该线程对象放入了线程组中group中。**  
 然后通过此Thread类调用方法run()来完成其运行操作的，这里方法run()称为线程体，它包含了要执行的这个线程的内容，**Run方法运行结束，此线程终止**，而CPU再运行其它线程。

（2）而run()方法只是类中我们自己定义的方法，是一种普通的方法，顺序执行，只有在执行完这一段代码之后才会继续执行下去，而**如果直接用Run方法，这只是调用一个方法而已，程序中依然只有主线程--这一个线程**，其程序执行路径还是只有一条，这样就没有达到写线程的目的。

# 用户态和内核态的理解和区别

<https://blog.csdn.net/qq_39823627/article/details/78736650>

<https://www.cnblogs.com/gizing/p/10925286.html>

首先明白的是这两种状态都是针对进程来说的，而进程是操作系统创建的，而每个进程所能做的工作都不一样，为了将这些工作分轻重缓急、重要程度，给进程建立了特权级的概念，最关键的工作交给特权级最高的线程去做，这样可以做到集中管理，减少有限资源的访问和冲突。而inter x86的cpu一共有4个级别0-3，其中0的特权级别最高。而用户态的进程就是特权级别为3，内核态的进程就是特权级别为0.

一般而言，自己刚开始创建的进程都会处于用户态，但如果要执行文件操作，网络数据发送等操作必须调用write\send等系统调用，那么这个时候需要将进程从用户态切换到内核态，因为用户态的进程的特权不允许访问本该内核态才能访问的代码和数据。当完成系统调用后，又会切换为用户态。

**总结一下：**

内核态：运行操作系统程序，操作硬件

**用户态：运行用户程序**

**差别在于：**

处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处于占有的处理器是可被抢占的

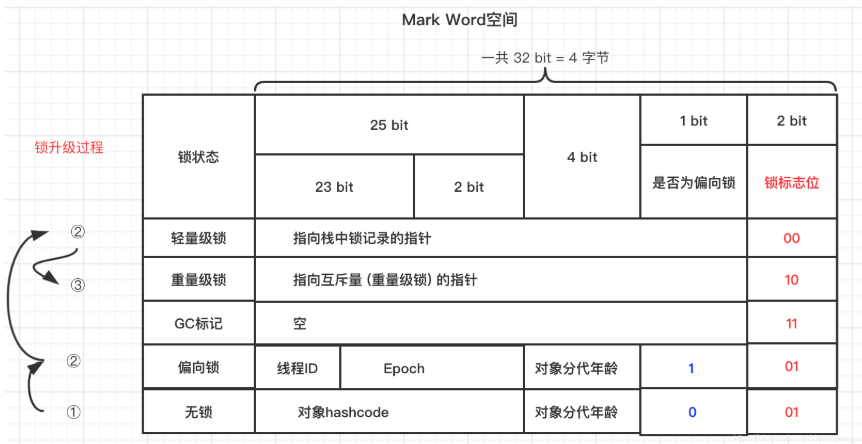
处于内核态执行时，则能访问所有的内存空间和对象，且所占有的处理器是不允许被抢占的。

# java对象头

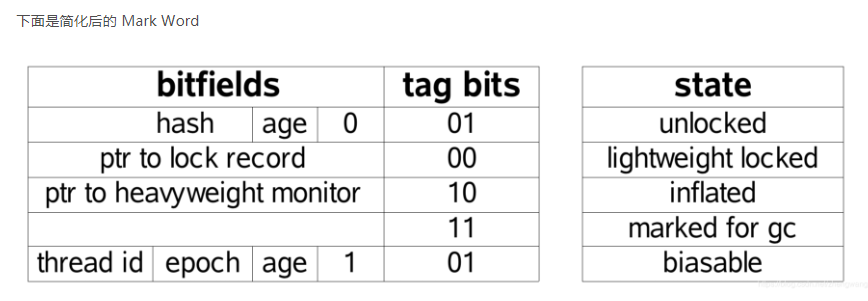
首先，java对象包含三个部分：

**java对象头：**





以上是32位JVM背景下的对象头中的mark word，锁标志位代表该对象是否获得该资源，获的该资源加的锁是什么，01是初始化时的标志位，表示无锁，年龄为1时，表示获得该资源，加了偏向锁，00是轻量级锁，10是重量级锁，11表示该对象要被回收。



**对象实例数据**：

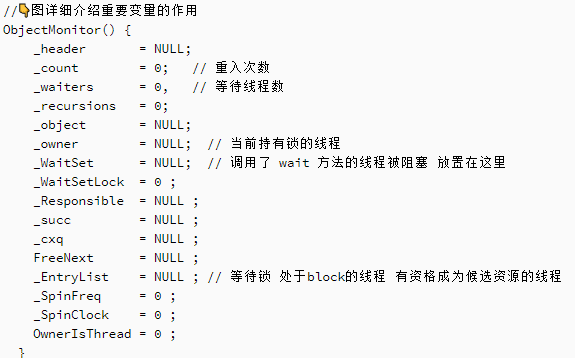
成员变量等

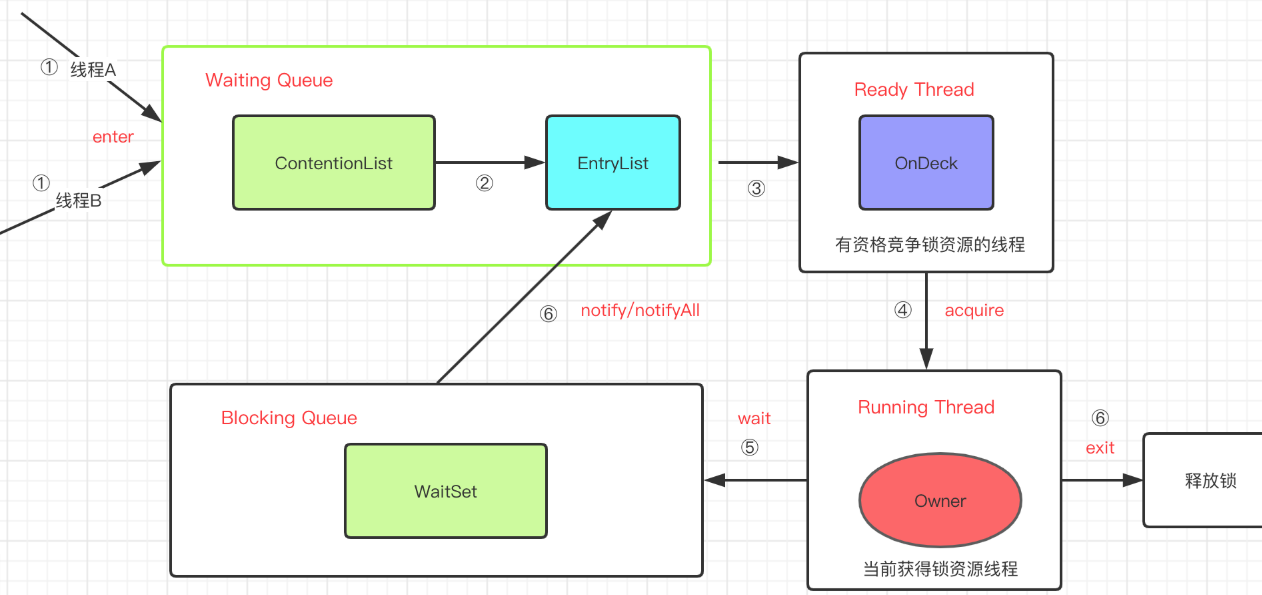
**对齐填充：**

JVM要求对象占用的空间必须是8 的倍数，方便内存分配（以字节为最小单位分配），因此这部分就是用于填满不够的空间凑数用的。

# Monitor

每个对象都有一个与之关联的Monitor 对象





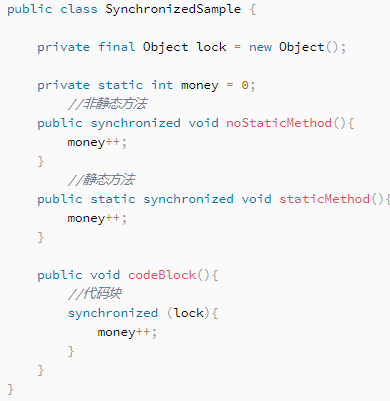
# 9、synchronized详解

<https://blog.csdn.net/zhengwangzw/article/details/105141484>

**Synchronized有三种作用范围：**

1. 在静态方法上加锁
2. 在非静态方法上加锁
3. 在代码块上加锁

注意：锁只能加在对象上。



synchronized 在代码块上加锁是通过 monitorenter 和 monitorexit指令实现，在静态方法和非静态方法上加锁是在方法的flags 中加入 ACC\_SYNCHRONIZED 。JVM 运行方法时检查方法的flags，遇到同步标识开始启动前面的加锁流程，在方法内部遇到monitorenter指令开始加锁。

**JVM怎么通过sychronized在对象上加锁，保证多线程访问竟态资源安全。**

总的来说：

1. jdk6之前，都是重量级锁
2. Jdk6之后，引入了偏向锁和轻量级锁，
   1. 原因1、因为Sun 程序员发现大部分程序大多数时间都不会发生多个线程同时访问竞态资源的情况，每次线程都加锁解锁，每次这么搞都要操作系统在用户态和内核态之间来回切（加锁后，线程处于内核态，不允许抢占该线程占有的资源），太耗性能了。线程获取资源的时候，会发生系统调用，从用户态转化为内核态，假如加了重量锁，线程会重返用户态，来回切换会消耗性能，而假设加了轻量锁，线程会先保持内核态，当自旋操作达到一定数量后，将轻量锁标志位置为重量锁标志位，然后再重返用户态。
   2. 原因2、同一个线程加锁解锁的重复率很高，这也是偏向锁的由来。

偏向锁适合低并发时，同一个线程总是获得该资源，为避免重复的加锁解锁操作，而保持的一直占用，当高并发时，用的都是轻量级锁，因为线程执行的时间可能很短亦或是很少发生线程争抢，所以加轻量级锁就好，使进程保持短暂的内核态比用户态和内核态切换消耗的性能要少，当线程执行时间比较长或者并发度很高时，那么使用重量级锁较好。

**总结：**

并发度极低：使用偏向锁，避免频繁加锁解锁

并发度中等，线程执行时间较短：使用轻量级锁，使进程保持短暂的内核态，避免频繁的用户态和内核态切换。

并发度高或线程执行时间较长：使用重量级锁，防止大量线程执行较长时间的轻量级锁的自旋操作。

加锁具象化理解：

在一个类中，全局变量可以看做是竟态资源，如StringBuffer当中的value数组， 而对其中的append方法加sychronized关键字，实例化一个StringBuffer对象，在多个线程中调用StringBuffer对象的append方法，锁住的就是当前对象，不允许多个线程同时调用append方法，那么也就不会同时操作value数组。也就是说，**sychronized锁住的其实是操作资源的代码块，而不是真的资源。**

