# Java基础

## 经典基础问题

### 1、为什么局部/匿名内部类在使用外部局部变量时, 只能使用被final修饰的变量?

首先，内部类会以拷贝的形式把外部局部变量final a复制了一个副本到内部类中, 内部类有一个变量a指向外部变量a所指向的值；现在我们知道内部类的变量a和外部局部变量a是两个完全不同的变量, 那么如果在执行run()方法的过程中, 内部类中修改了a变量所指向的值, 就会产生数据不一致问题。

### 2、final修饰的变量、方法、类有什么特点？

被final修饰的变量是不能够被改变的，final修饰的是一个基本数据类型数据时, 这个数据的值在初始化后将不能被改变; 当final修饰的是一个引用类型数据时, 也就是修饰一个对象时, 引用在初始化后将永远指向一个内存地址, 不可修改. 但是该内存地址中保存的对象信息, 是可以进行修改的. 并且被final修饰的常量在编译阶段会被放入常量池中。

使用final修饰方法有两个作用, 首要作用是锁定方法, 不让任何继承类对其进行修改.

使用final修饰类的目的简单明确: 表明这个类不能被继承. 被final修饰的类所有成员方法都将被隐式修饰为final方法.

### 3、String类型默认为final的好处是？

1、只有当字符串是不可变的，字符串池才有可能实现。

2、如果字符串是可变的，那么会引起很严重的安全问题。

3、因为字符串是不可变的，所以是多线程安全的。

4、类加载器要用到字符串，不可变性提供了安全性,以便正确的类被加载。

5、作为Map的key，提高了访问效率。

### 4、BIO、NIO、AIO 有什么区别？

5种IO模型介绍：<https://mp.weixin.qq.com/s?__biz=Mzg3MjA4MTExMw==&mid=2247484746&idx=1&sn=c0a7f9129d780786cabfcac0a8aa6bb7&source=41#wechat_redirect>

一次IO操作的两个阶段？

（1）请求准备：内核从磁盘中的数据加载至内核缓存区（读操作）

（2）数据拷贝：内核缓存区数据再复制到用户进程内存（这个是真正执行IO的阶段）

同步和异步指的是？

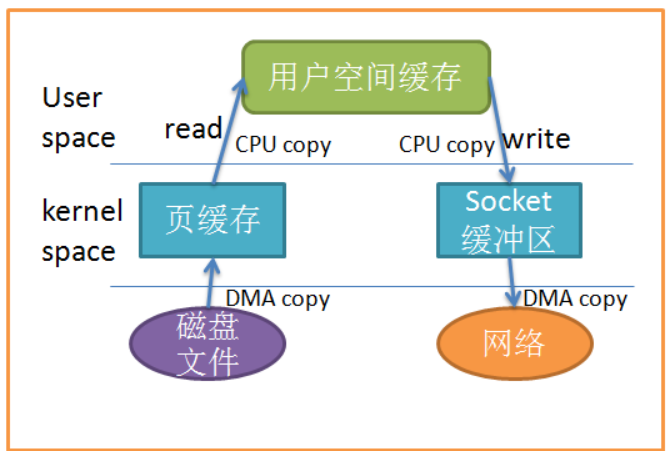
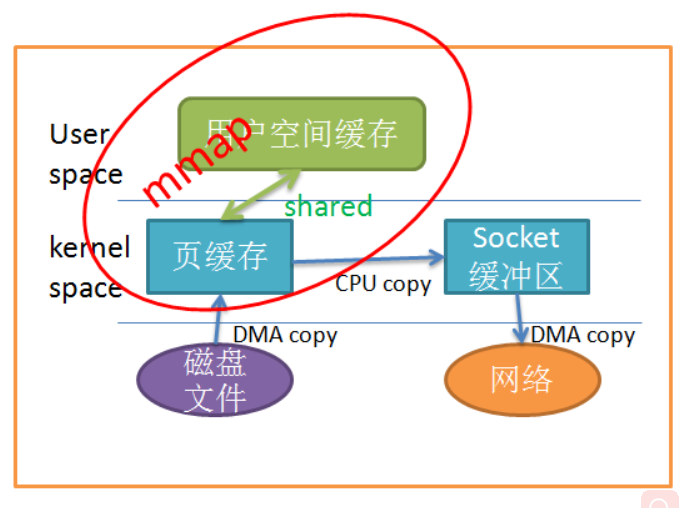
指的是消息的通信机制，由数据拷贝阶段来确定，如果使用用户线程进行拷贝，则为同步模式，但如果是有操作系统内核来完成，则为异步模式。

阻塞和非阻塞指的是？

指的是程序在等待调用结果（消息，返回值）时的状态，由请求准备阶段来确定。

* Java1.5+使用的IO模型为I/O复用模型

### 5、零拷贝技术？

### 6、为什么说Java中只有值传递?

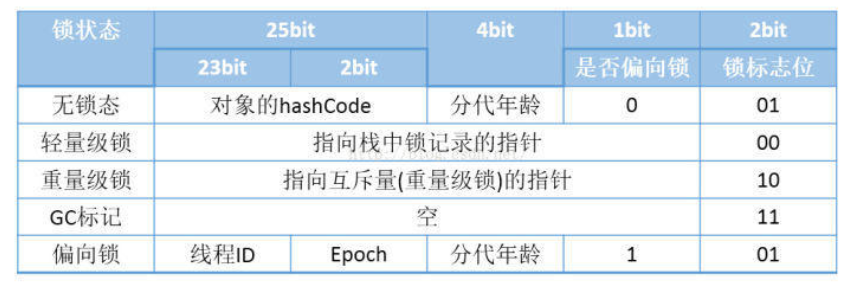
值传递（pass by value）是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

引用传递（pass by reference）是指在调用函数时将实际参数的地址直接传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。



### 7、synchronized锁的底层实现？

对象在内存中的存储布局方式可以分为3块区域：对象头、实例数据、对齐填充。其中对象头，便是我们今天的主角。

锁的种类：

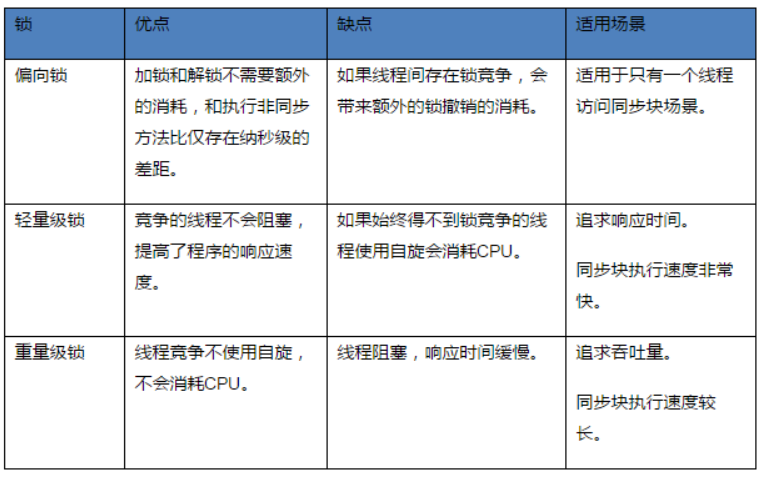
1. 重量级锁：重量级锁是我们常说的传统意义上的锁，其利用操作系统底层的同步机制去实现Java中的线程同步。重量级锁的状态下，对象的mark word为指向一个堆中monitor对象的指针。运行时并没有多线程竞争，或两个线程接近于交替执行的情况，使用传统锁机制无疑效率是会比较低的。
2. 轻量级锁：程序运行时，同步块中的代码都是不存在竞争的，不同的线程交替的执行同步块中的代码。这种情况下，用重量级锁是没必要的。因此JVM引入了轻量级锁的概念。期间利用CAS指令来改变对象头的信息，发生失败后膨胀为重量级锁。
3. 偏向锁：在JDK1.6中为了提高一个对象在一段很长的时间内都只被一个线程用做锁对象场景下的性能，引入了偏向锁，在第一次获得锁时，会有一个CAS操作，之后该线程再获取锁，只会执行几个简单的命令，而不是开销相对较大的CAS命令。当然在条件不满足的情况下会膨胀为轻量级锁。
4. 自旋锁：即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源。
   * 自旋锁优点：线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。同时我们发现在许多应用上面，对象锁的锁状态只会持续很短一段时间，为了这一段很短的时间频繁地阻塞和唤醒线程是非常不值得的，所以引入自旋锁。

锁的获取过程：（也是jvm底层对synchronized的优化策略）

在所有的锁都启用的情况下线程进入临界区时会先去获取偏向锁，如果已经存在偏向锁了，则会尝试获取轻量级锁，如果以上两种都失败，则启用自旋锁，如果自旋也没有获取到锁，则使用重量级锁，没有获取到锁的线程阻塞挂起，直到持有锁的线程执行完同步块唤醒他们；

* + 过程：偏向锁--》轻量级锁--》自旋锁--》重量级锁

锁的优缺点：



### 8、程序中如何进行的锁优化？

1、减少锁的时间

2、减小锁的粒度

3、锁粗化

4、使用读写锁

### 9、synchronized 和 ReentrantLock 区别是什么？

1、synchronized 竞争锁时会一直等待；ReentrantLock 可以尝试获取锁，并得到获取结果

2、synchronized 获取锁无法设置超时；ReentrantLock 可以设置获取锁的超时时间

3、synchronized 无法实现公平锁；ReentrantLock 可以满足公平锁，即先等待先获取到锁

4、synchronized 控制等待和唤醒需要结合加锁对象的 wait() 和 notify()、notifyAll()；ReentrantLock 控制等待和唤醒需要结合 Condition 的 await() 和 signal()、signalAll() 方法

5、synchronized 是 JVM 层面实现的；ReentrantLock 是 JDK 代码层面实现

6、synchronized 在加锁代码块执行完或者出现异常，自动释放锁；ReentrantLock 不会自动释放锁，需要在 finally{} 代码块显示释放

7、都支持锁重入

### 10、HashMap的扩容机制？

1、容器初始化大小选择：当数组长度（也就是初始化容器大小）为2的n次幂的时候，不同的key算得得index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。具体index的取值方式为：首先算得key得hashcode值，然后跟数组的长度-1做一次“与”运算（&），即：hashcode & (length-1)

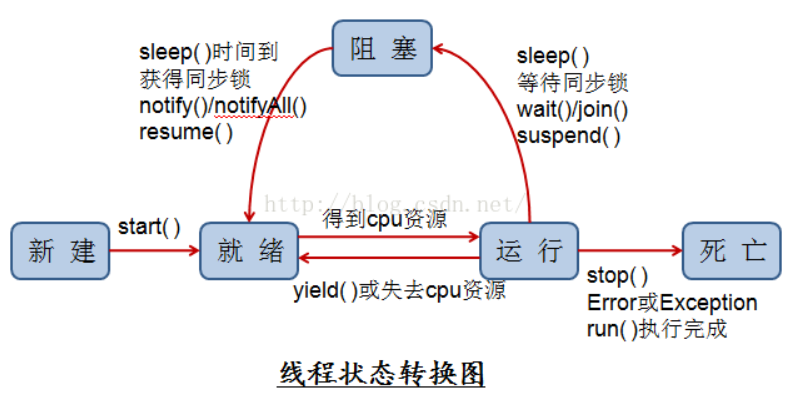
2、resize过程：而在hashmap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。当hashmap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75。所以我们在初始化容器大小的时候需要考虑到：预估值 < 0.75\*初始化值，比如存1000个元素，最好初始化值设为2048。

### 11、HashMap多线程并发的问题？

1、造成数据脏读。

2、会发生死锁，造成CPU、内存飙升100%的问题，原因是因为在put操作过程中发生扩容，形成了循环链表而导致的。

### 12、线程的生命周期



### 13、线程之间通信方式（基于共享内存或消息传递）

1、通过synchronized关键字：线程A和线程B持有同一个MyObject类的对象object,尽管这两个线程需要调用不同的方法,但是它们是同步执行，本质上就是“共享内存”式的通信

2、while轮询：在这种方式下，线程A不断地改变条件，线程ThreadB不停地通过while语句检测某个条件是否成立 ，从而实现了线程间的通信。但是这种方式会浪费CPU资源。

3、wait/notify机制：线程唤醒与等待

4、使用 volatile 关键字：多个线程同时监听一个变量，当这个变量发生变化的时候 ，线程能够感知并执行相应的业务

5、使用JUC工具类 CountDownLatch

6、使用 ReentrantLock 结合 Condition：等同于wait/notify

### 14、ReentrantLock的底层实现原理

**ReetrantLock底层是基于AQS实现的（CAS+CHL），有公平和非公平两种区别。**

ReentrantLock的基本实现可以概括为：先通过CAS尝试获取锁。如果此时已经有线程占据了锁，那就加入AQS队列并且被挂起。当锁被释放之后，排在CLH队列队首的线程会被唤醒，然后CAS再次尝试获取锁。在这个时候，如果：

* + 非公平锁：如果同时还有另一个线程进来尝试获取，那么有可能会让这个线程抢先获取；
  + 公平锁：如果同时还有另一个线程进来尝试获取，当它发现自己不是在队首的话，就会排到队尾，由队首的线程获取到锁。（区别）

CLH队列：带头结点的双向非循环链表。

### 15、AQS底层原理解析

AQS(AbstractQueuedSynchronizer)是一个抽象同步队列，JUC(java.util.concurrent)中很多同步锁都是基于AQS实现的。AQS中有一个被volatile声明表示状态的字段state，例如ReentrantLock用它来表示线程重入锁的次数，Semphore用它表示剩余的许可数量，FutureTask用它表示任务的状态，CountDownLatch用它表示需要被后续调用countDown()的次数。对state变量值的更新都采用CAS操作保证更新操作的原子性。

AQS的基本原理就是当一个线程请求共享资源的时候会判断是否能够成功操作这个共享资源，如果可以就会把这个共享资源设置为锁定状态，如果当前共享资源已经被锁定了，那就把这个请求的线程阻塞住，也就是放到队列中等待。

独占锁、共享锁区别：

1. 要实现一个独占锁（悲观锁），那就去重写或封装tryAcquire，tryRelease方法.
2. 要实现共享锁（乐观锁），就去重写tryAcquireShared，tryReleaseShared.

### 16、CountDownLatch和CyclicBarrier的区别？

CountDownLatch是计数器，只能使用一次，而CyclicBarrier的计数器提供reset功能，可以多次使用。

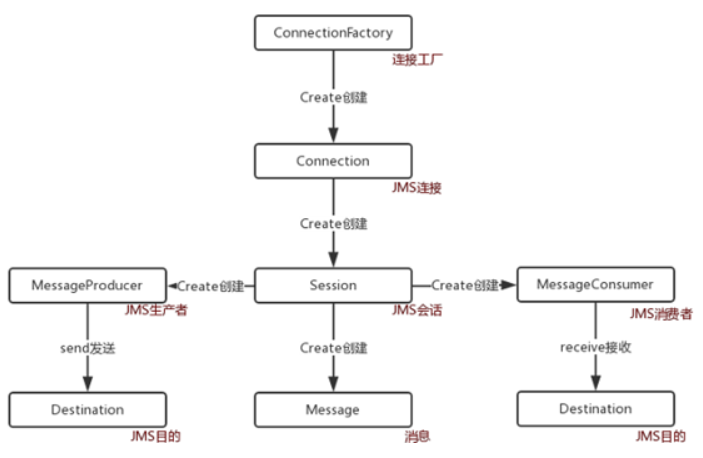
JavaDoc中的定义：

CountDownLatch：一个或者多个线程，等待其他线程完成某件事情之后，等待其他多个线程完成某件事情之后才能执行。

CyclicBarrier：多个线程互相等待，直到到达同一个同步点，再继续一起执行。

### 17、JMS的设计原理（java message service）

JMS对象模型：



在JMS标准中，有两种消息模型PTP（Point to Point）,Publish/Subscribe(Pub/Sub)。

### 18、JMS和AMQP的区别？

## 高级

# Springboot

# Springcloud

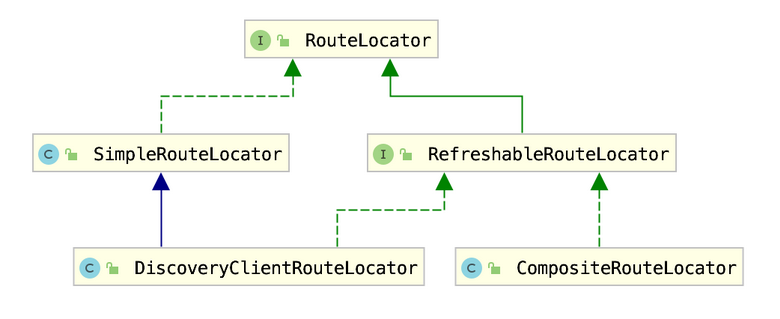
## 基础使用

1. ES+Logstash+Kibana整合spingcloud

2、

## 源码分析

1. 如何使用zuul实现动态路由？（添加、修改、删除）



原理：普通情况下，我们的路由规则是配置在properties文件中，当有请求到达ZuulServlet时会根据过滤规则选取合适的路由过滤器（SimpleHostRoutingFilter、RibbonRoutingFilter）来完成请求转发；路由定位器客户端（DiscoveryClientRouteLocator）根据properties信息动态加载路由数据，刷新路由。 CompositeRouteLocator具备组合多个RouteLocator的能力，用Collection存储多个RouteLocator，调用 getRoutes()、getMatchingRoute()、refresh() 时都会逐一调用每个RouteLocator相应的方法。

方案：

* 通过自定义 RouteLocator （extends SimpleRouteLocator implements RefreshableRouteLocator）来实现动态路由，自定义的RouteLocator最终会添加到CompositeRouteLocator中。

# Mysql

# Redis