1、redis的特点：

基于内存、单线程、IO多路复用

2、IO多路复用：

经典的Reactor设计模式，也称为异步阻塞IO，



用户首先将需要进行IO操作的socket添加到select函数中，然后阻塞等待select系统调用返回。当数据到达时，socket被激活，select函数返回。用户线程正式发起read请求，读取数据并继续执行。用户可以注册多个socket，然后不断地调用select读取被激活的socket，即可达到在同一个线程内同时处理多个IO请求的目的。

通过Reactor的方式，用户线程注册事件处理器之后可以继续执行做其他的工作（异步），而Reactor线程负责调用内核的select函数检查socket状态。当有socket被激活时，则通知相应的用户线程，执行handle\_event进行数据读取、处理的工作。由于select函数是阻塞的，因此多路IO复用模型也被称为异步阻塞IO模型。注意，这里的所说的阻塞是指select函数执行时线程被阻塞，而不是指socket。

**3 、redis单线程问题**

    单线程指的是网络请求模块使用了一个线程（所以不需考虑并发安全性），即一个线程处理所有网络请求，其他模块仍用了多个线程。采用单线程,避免了不必要的[上下文切换](https://www.baidu.com/s?wd=%E4%B8%8A%E4%B8%8B%E6%96%87%E5%88%87%E6%8D%A2&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)

4、redis数据结构

String、List、Set、hash、zSet

5、spring的注入方式

常用的注入方式主要有：构造方法注入，setter注入，基于注解的注入。

<https://blog.csdn.net/yinni11/article/details/80212614>

https://blog.csdn.net/qq594913801/article/details/80512451

6、java虚拟机参数指令

-Xms20m 堆最小值20MB

-Xmx20m堆最大值20MB

-Xss128k 栈容量128KB

-XX:MaxPermSize=10M 永久代(方法区)最大值为10MB

7、类加载过程（类的生命周期）

类加载过程包括 加载、验证、准备、解析、初始化五个阶段。

加载：（三步）

a、通过类全限定名获取类的二进制字节流。

b、将该字节流所代表的静态存储结构转化为方法区的运行时数据结构。

c、在内存中生成一个代表该类的java.lang.Class对象，作为方法区这个类的访问入口。

验证：连接阶段的第一步，为了确保Class文件的字节流符合虚拟机的安全。

准备：为类变量（static修饰）分配内存并设置初值（0值）。

解析：将常量池内的符号引用替换为直接引用的过程。

符号引用: 用一组符号来描述所引用的目标，符号引用与虚拟机实现的内存布局无关，引用的目标不一定已经加载到内存中。

直接引用：可以是直接指向目标的指针、相对偏移量或句柄。直接引用和虚拟机实现的内存布局相关，如果有直接引用，那应用的目标已存在在内存中。

初始化：类加载过程的最后一步，执行类构造器<clinit>()方法的过程。

8、类加载器

启动类加载器（Bootstrap ClassLoader）:

加载<JAVA\_HOME>\lib目录中的

扩展类加载器（ExtensionClassLoader）：

加载<JAVA\_HOME>\lib\ext目录中的

应用程序类加载器（ApplicationClassLoader）

加载ClassPath所指定的类库，也称系统类加载器

双亲委派模型：

除启动类加载器外，其余类加载器都有父加载器，关系以“复用”实现而不是继承。

当一个类加载器收到类加载的请求时，它会先把这个请求委派给父类加载器来完成，所有的加载请求都会送到顶层的启动类加载器，只有当父类加载器反馈无法加载时，子加载器才会尝试加载。双亲委派模型使得java类随着其类加载器具有了优先级的层次关系。例如，Object类被任何一个类加载器加载时都是同一个类。

9、垃圾回收判定

可达性分析算法：当一个到GC Roots没有任何应用链相连时，对象会被判定为可回收的对象。

引用类型：

强引用（引用还在就不会被回收）

软引用（有用但并非必需的对象，系统内存溢出前才会回收这类对象）

弱引用（非必需对象，只能生存到下一次垃圾收集发生之前）

虚引用（虚引用不影响对象生存时间，也无法取得对象的实例，唯一作用是在对象被回收时收到系统通知）

垃圾回收时要进行两次标记，可达性标记第一次，第二次标记finalize()方法，若对象覆写该方法并在方法中重建引用链，则对象不会被回收。

10、垃圾收集算法

虚拟机都采用“分代收集”算法，将内存分为新生代与老年代。

(1)标记-清除算法(mark-sweep)

首先标记出所有需要回收的对象，标记完成后统一回收，缺点：会产生不连续的内存碎片。

(2)复制算法(copying)

hotspot将新生代内存分为Eden空间和两块Survivor空间，大小比为8：1，每次使用Eden和其中一块Survivor，当回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor上，最后清理掉Eden与刚才的Survivor。当Survivor空间不够时，将使用老年代进行分配担保。

(3)标记-整理算法(mark-compact)

对老年代使用，标记过程与标记-清除算法一样，但后续不是直接对对象回收而是让存活的对象向一端移动，清理掉边界以外的内存。

11、垃圾收集器



（1）Serial收集器

Serial收集器是一个单线程的垃圾收集器，特点如下：

·仅仅使用一个线程完成垃圾收集工作；

·在垃圾收集时必须暂停其他所有的工作线程，直到垃圾收集结束；

·Stop the World是在用户不可见的情况下执行的，会造成某些应用响应变慢；

·使用复制算法；

Serial收集器的工作流程如下图：



虽然如此，Serial收集器依然是虚拟机运行在Client模式下的默认新生代收集器。它的优点同样明显：简单而高效（单个线程相比），并且由于没有线程交互的开销，专心做垃圾收集自然课获得最高的单线程效率。

（2）ParNew收集器

ParNew收集器其实是Serial收集器的多线程版本，与Serial不同的地方就是在垃圾收集过程中使用多个线程，剩下的所有行为都一样。ParNew收集器也使用复制算法。ParNew收集器的工作流程如下图：



ParNew收集器看似没有多大的创新之处，但却是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为，除了Serial收集器外，目前只有ParNew收集器能够与CMS收集器配合工作，而CMS收集器是HotSpot在JDK 1.5时期推出的具有划时代意义的垃圾收集器。可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

（3）Parallel Scavenge收集器

Parallel Scavenge收集器和ParNew类似，是一个使用复制算法的并行多线程收集器。不过和ParNew不同的是，Parallel Scavenge收集器的关注点不同。

CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目的则是达到一个可控制的吞吐量。吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+运行垃圾收集时间）。如果虚拟机一共运行100分钟，垃圾收集运行了1分钟，那么吞吐量就是99%。

停顿时间越短就越适合与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效的利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge收集器提供了两个参数来精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数。

Parallel Scavenge收集器也叫吞吐量优先收集器，它还有一个参数-XX:UseAdaptiveSizePolicy，这是一个开关参数，当这个参数打开后，就不需要手工指定新生代的大小（-Xmn）、Eden和Survivor的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最适合的停顿时间或最大的吞吐量，这叫GC自适应的调节策略。这也是Parallel Scavenge收集器和ParNew收集器的一个重要区别。

（4）Serial Old收集器

Serial Old是Serial的老年版本，在Serial的工作流程图中可以看到，Serial Old收集器也是一个单线程收集器，使用“标记-整理”算法。这个收集器主要给Client模式下的虚拟机使用。如果在Serve模式下，它有两个用途：一个是在JDK 1.5之前的版本中与Parallel Scavenge收集器搭配使用；另一个就是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。这个收集器的工作流程在Serial的后半部分有所体现。

（5）Parallel Old收集器

Parallel Old收集器是Parallel Scavenge收集器的老年版本，它也使用多线程和“标记-整理”算法。这个收集器是在JDK 1.6开始提供。

在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器的组合。

（6）CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。在重视响应速度和用户体验的应用中，CMS应用很多。

CMS收集器使用“标记-清除”算法，运作过程比较复杂，分为4个步骤：

初始标记（CMS initial mark）

并发标记（CMS Concurrent mark）

重新标记（CMS remark）

并发清除（CMS Concurrent Sweep）

其中，初始标记和并发标记仍然需要Stop the World、初始标记仅仅标记一下GC Roots能直接关联到的对象，速度很快，并发标记就是进行GC RootsTracing的过程，而重新标记阶段则是为了修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段长，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以整体上说，CMS收集器的内存回收过程是与用户线程一共并发执行的。下图是流程图：



CMS的优点就是并发收集、低停顿，是一款优秀的收集器。不过，CMS也有缺点，如下：

CMS收集器对CPU资源非常敏感。CMS默认启动的回收线程数是（CPU数量+3）/4，当CPU个数大于4时，垃圾收集线程使用不少于25%的CPU资源，当CPU个数不足时，CMS对用户程序的影响很大；

CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC；

CMS使用标记-清除算法，会产生内存碎片；

（7）G1收集器

G1（Garbage first）收集器是最先进的收集器之一，是面向服务端的垃圾收集器。与其他收集器相比，G1收集器有如下优点：

并行与并发：有些收集器需要停顿的过程G1仍然可以通过并发的方式让用户程序继续执行；

分代收集：可以不使用其他收集器配合管理整个Java堆；

空间整合：使用标记-整理算法，不产生内存碎片；

可预测的停顿：G1除了降低停顿外，还能建立可预测的停顿时间模型；

G1中也有分代的概念，不过使用G1收集器时，Java堆的内存布局与其他收集器有很大的差别，它将整个Java堆划分为多个大小相等的独立区域（Region），G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划的避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里垃圾堆积的价值大小（回收所获得的空间大小以及回收所需要的时间的经验值），在后台维护一个优先列表，每次优先收集价值最大的那个Region。这样就保证了在有限的时间内尽可能提高效率。

12、java8持久代移除变为元空间

JDK 8.HotSpot JVM开始使用本地化的内存存放类的元数据，这个空间叫做元空间（Metaspace）。新的参数 (MaxMetaspaceSize)可以使用，参数PermSize 和 MaxPermSize会被忽略

元空间的特点：   
- 充分利用了Java语言规范中的好处：类及相关的元数据的生命周期与类加载器的一致。   
- 每个加载器有专门的存储空间，给每个类加载器分配一个内存块的列表。块的大小取决于类加载器的类型。  
- 只进行线性分配   
- 不会单独回收某个类   
- 省掉了GC扫描及压缩的时间   
- 元空间里的对象的位置是固定的   
- 如果GC发现某个类加载器不再存活了，会把相关的空间整个回收掉

对于Java8， HotSpots取消了永久代，那么是不是也就没有方法区了呢？当然不是，方法区是一个规范，规范没变，它就一直在。那么取代永久代的就是元空间。元空间存储类的元信息，静态变量和常量池等并入堆中。相当于永久代的数据被分到了堆和元空间中。

13、常量池

a.Class文件常量池  
       Class 文件常量池指的是编译生成的 class 字节码文件，其结构中有一项是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

这里的字面量是指字符串字面量和声明为 final 的（基本数据类型）常量值，这些字符串字面量除了类中所有双引号括起来的字符串(包括方法体内的)，还包括所有用到的类名、方法的名字和这些类与方法的字符串描述、字段(成员变量)的名称和描述符；声明为final的常量值指的是成员变量，不包含本地变量，本地变量是属于方法的。这些都在常量池的 UTF-8 表中(逻辑上的划分)；  
符号引用，就是指指向 UTF-8 表中向这些字面量的引用，包括类和接口的全限定名(包括包路径的完整名)、字段的名称和描述符、方法的名称和描述符。只不过是以一组符号来描述所引用的目标，和内存并无关，所以称为符号引用，直接指向内存中某一地址的引用称为直接引用；  
b.运行时常量池  
       运行时常量池是方法区的一部分，是一块内存区域。Class 文件常量池将在类加载后进入方法区的运行时常量池中存放。一个类加载到 JVM 中后对应一个运行时常量池，运行时常量池相对于 Class 文件常量池来说具备动态性，Class 文件常量只是一个静态存储结构，里面的引用都是符号引用。而运行时常量池可以在运行期间将符号引用解析为直接引用。可以说运行时常量池就是用来索引和查找字段和方法名称和描述符的。给定任意一个方法或字段的索引，通过这个索引最终可得到该方法或字段所属的类型信息和名称及描述符信息，这涉及到方法的调用和字段获取。

c.字符串常量池  
       字符串常量池是全局的，JVM 中独此一份，因此也称为全局字符串常量池。运行时常量池中的字符串字面量若是成员的，则在类的加载初始化阶段就使用到了字符串常量池；若是本地的，则在使用到的时候（执行此代码时）才会使用到字符串常量池。其实，“使用常量池”对应的字节码是一个 ldc 指令，在给 String 类型的引用赋值的时候会先执行这个指令，看常量池中是否存在这个字符串对象的引用，若有就直接返回这个引用，若没有，就在堆里创建这个字符串对象并在字符串常量池中记录下这个引用（jdk1.7)。String 类的 intern() 方法还可在运行期间把字符串放到字符串常量池中。JVM 中除了字符串常量池，8种基本数据类型中除了两种浮点类型剩余的6种基本数据类型的包装类，都使用了缓冲池技术，但是 Byte、Short、Integer、Long、Character 这5种整型的包装类也只是在对应值在 [-128,127] 时才会使用缓冲池，超出此范围仍然会去创建新的对象。其中：

在 jdk1.6（含）之前也是方法区的一部分，并且其中存放的是字符串的实例；  
在 jdk1.7（含）之后是在堆内存之中，方法区存储的是字符串对象的引用，字符串实例是在堆中；  
jdk1.8 已移除永久代，字符串常量池是在本地内存当中，存储的也只是引用。

14、事务ACID

原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

一致性（Consistency）

事务前后数据的完整性必须保持一致。

隔离性（Isolation）

事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。一般通过悲观或者乐观锁实现。

持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

15、数据库中的事务和锁（乐观、悲观锁，共享、排他锁，死锁）

<https://blog.csdn.net/woshiluoye9/article/details/68954515>

16、mysql中innodb和myisam对比及索引原理区别

<https://blog.csdn.net/qq_27607965/article/details/79925288>

17、redis持久化机制

a、RDB快照

　RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，是默认的持久化方式，将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。**RDB有它的不足，就是一旦数据库出现问题，那么我们的RDB文件中保存的数据并不是全新的**，从上次RDB文件生成到 Redis停机这段时间的数据全部丢掉了。

b、AOF日志

AOF日志的全称是Append Only File，从名字上我们就能看出来，它是一个追加写入的日志文件。与一般数据库不同的是，**AOF文件是可识别的纯文本，它的内容就是一个个的Redis标准命令**。如果需要使用AOF，则需要修改配置文件中的以下条目：  
将appendonly no改为appendonly yes。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据。

18、数据库优化

查找并定位慢查询，在启动数据库时开启慢查询，将执行慢的语句写入日志中，找出慢查询的语句后explain慢查询语句。

a、创建索引

b、分表，当表中数据较多时水平分表、表中某些字段值较长且很少使用时垂直分表

c、读写分离，读写分离前要设置主从同步，然后用负载均衡将写的操作放到主数据库，读操作放到从服务器。

d、缓存，在持久层前做一个缓存层，使用redis作中央缓存。

19、批量插入百万条数据

用JDBC关闭自动提交，将多次提交变为一次提交，prepareStatement.addBatch()添加操作。

20、原子类的实现

乐观锁实现，volatile+CAS实现，更新不成功继续循环更新，效率比每次都加锁(悲观锁)高很多。

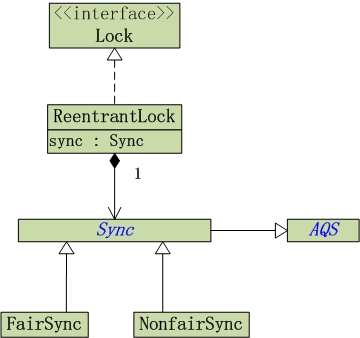
incrementAndGet的实现通过一个无限循环直到increment成功为止.    
循环的内容是  
1.取得当前值  
2.计算+1后的值  
3.如果当前值还有效(没有被修改)的话设置为那个+1后的值  
4.如果设置没成功(当前值已经无效了即被别的线程改过了), 再从1开始.

调用顺序：incrementAndGet();->compareAndSet(current, next);->

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

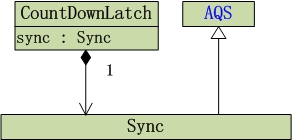
 compareAndSwapInt不是用Java实现的, 而是通过JNI调用操作系统的原生程序,最后会指向一条汇编CPU指令。

21、ReentrantLock实现

[](https://images0.cnblogs.com/blog/497634/201401/271152070311302.jpg)

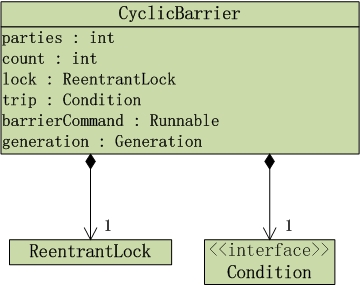
22、CountDownLatch实现

CountDownLatch是通过“共享锁”实现的。在创建CountDownLatch中时，会传递一个int类型参数count，该参数是“锁计数器”的初始状态，表示该“共享锁”最多能被count个线程同时获取。当某线程调用该CountDownLatch对象的await()方法时，该线程会等待“共享锁”可用时，才能获取“共享锁”进而继续运行。而“共享锁”可用的条件，就是“锁计数器”的值为0！而“锁计数器”的初始值为count，每当一个线程调用该CountDownLatch对象的countDown()方法时，才将“锁计数器”-1；通过这种方式，必须有count个线程调用countDown()之后，“锁计数器”才为0，使前面的等待线程运行！

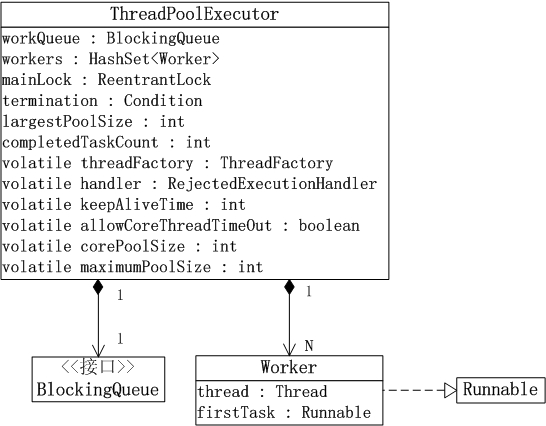
[](https://images0.cnblogs.com/blog/497634/201401/271448592345680.jpg)

23、CyclicBarrier实现

CyclicBarrier是通过[ReentrantLock](http://www.cnblogs.com/skywang12345/p/3496147.html)(独占锁)和[Condition](http://www.cnblogs.com/skywang12345/p/3496716.html)来实现的，await()让当前线程阻塞，直到“有parties个线程到达barrier” 或 “当前线程被中断” 或 “超时”这3者之一发生，breakBarrier()会设置当前中断标记broken为true，意味着“将该Generation中断”；同时重新初始化count；最后，通过signalAll()唤醒CyclicBarrier上所有的等待线程。。CyclicBarrier中通过generation对象，记录属于哪一代。当有parties个线程到达barrier，generation就会被更新换代。

[](https://images0.cnblogs.com/blog/497634/201401/271455098594619.jpg)

24、ThreadPoolExecutor实现



一个Worker对应一个线程，当Worker对应的线程池启动时，它会执行线程池中的任务；当执行完一个任务后，它会从线程池的阻塞队列中取出一个阻塞的任务来继续运行。workQueue是一个阻塞队列。当线程池中的线程数超过它的容量的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能。

25、JIT编译器

<https://www.cnblogs.com/insistence/p/5901457.html>

# 26、ThreadLocal

# <http://www.cnblogs.com/dolphin0520/p/3920407.html>

# 27、页面淘汰算法

# <https://blog.csdn.net/youanyyou/article/details/78989956>

# 28、Mysql undo日志与redo日志

# 事物中，每执行一条SQL语句都会记录与之相反的undo log（撤销日志）undo log是事物原子性的实现。每执行一条sql会记录到redo log,事物提交时，持久化到磁盘，redo log使持久化批量执行，提高了磁盘IO.

# 日志的持久化早于db buffer的写入操作，系统故障时由log进行恢复，先持久化日志的策略叫write ahead log.

# 29、TCP关闭时的Time\_Wait

# <https://blog.csdn.net/u013616945/article/details/77510925>

# Time\_Wait的解决[SO\_REUSEADDR选项](https://www.cnblogs.com/qiaoconglovelife/p/5416715.html)

# <https://www.cnblogs.com/qiaoconglovelife/p/5416715.html>

# 30、java8 streamAPI

# <https://www.ibm.com/developerworks/cn/java/j-lo-java8streamapi/index.html?utm_source=wechat_session&utm_medium=social&utm_oi=955571522857914368>

# 31、LRU

# <https://zhuanlan.zhihu.com/p/34133067>

# 32、单例模式

# 双重检验版

# public class Singleton {     private static volatile Singleton s;     private Singleton(){};     public static Singleton getInstance() {          if(s == null) {             synchronized (Singleton.class) {                 if(s == null) {                     s = new Singleton();                 }             }         }         return s;     } }

# 如果没有volatile关键字,在并发情况下，在第5行会出现问题 对于第5行 s = new Singleton(); //5 可以分解为3个步骤： 1 memory=allocate();// 分配内存 相当于c的malloc 2 ctorInstanc(memory) //初始化对象 3 s=memory //设置s指向刚分配的地址 上面的代码在编译器运行时，可能会出现重排序 从1-2-3 排序为1-3-2 如此在多线程下就会出现问题 例如现在有2个线程A,B 线程A在执行第5行代码时，B线程进来，而此时A执行了 1和3，没有执行2，此时B线程判断s不为null 直接返回一个未初始化的对象，就会出现问题 而用了volatile，上面的重排序就会在多线程环境中禁止，不会出现上述问题。

# 枚举版

# class Resource{

# }

# public enum SingletonEnum {

# INSTANCE;

# private Resource instance;

# SingletonEnum () {

# instance = new Resource();

# }

# public Resource getInstance() {

# return instance;

# }

# }

# 33、内存泄露

# 内存泄露是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，memory leak会最终会导致out of memory。

# 1、静态集合类引起内存泄露：

# 2、当集合里面的对象属性被修改后，再调用remove（）方法时不起作用。

# 3、监听器  在java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

# 4、各种连接  比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的。Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。

6、单例模式

如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露。

不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露，考虑下面的例子：

# 34、一致性哈希算法

# <https://blog.csdn.net/bntX2jSQfEHy7/article/details/79549368>