# String 详解

## String类

String字符串是常量 是final类型类，父类继承Object。，在他们创建后他们的值不能被改变。不能被继承。

public final class **String**

extends [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

implements [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html), [Comparable](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html)<[String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html)>, [CharSequence](https://docs.oracle.com/javase/7/docs/api/java/lang/CharSequence.html)

## String 初始化

String 有两种复制方式 第一种str1=”likunran”,第二种是String str2=new String(“likunran”);这两种方式初始化有什么不同呢。我们运行下面代码。看到运行结果不一样。

== java是比较引用。及在内存中地址.程序可以运行看到str1和str2的地址相同。是因为String 针对常量池优化。执行str1 时首先会从常量池检查是否有“likunran”这个常量对象，有就直接赋值给str1没有就创建一个，放到常量池复制给str1。str2 过程一样 。所以 str1和str2有了相同的地址。

str2和str3地址不相同。String str3 = new String("likunran") 会开辟一个新空间然后复制给 str3所以两个地址不相同。str2==str3.intern() 相同。

## Intern 方法

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

All literal strings and string-valued constant expressions are interned. String literals are defined in section 3.10.5 of the The Java™ Language Specification.

上面是官方解释，string执行intern方法时，会类似st1的检查和返回。

public static void main(String[] args){

String str1 = "likunran";

String str2 = "likunran";

String str3 = new String("likunran");

String str4=str3.intern();

System.out.println( str1==str2);

System.out.println(str2==str3);

System.out.println(str2==str3.intern());

System.out.println(str2==str3);

System.out.println(str2==str4);

}

true

false

true

false

true

# Java 中的同步关键字

线程同步，在多线程中，可能有多个线程试图访问一个有限资源，必须限制同一时间最多只有一个线程访问这个资源。当两个并发线程同时访问一个带锁的资源。一段时间内有一个线程在访问一个在等待。ynchronized方法被某个线程执行时，其他线程无法访问该对象的任何synchronized方法（但是可以调用其他非synchronized的方法）。直至该synchronized方法执行完。

变量的作用域对线程操作有一定影响，如果变量是成员变量那么多个线程对同一个对象成员变量进行操作，是同一个共享变量操作。如果是局部变量，那么多个线程对同一个对象进行操作，每个线程操作都会有一个该局部变量的拷贝，他们之间的局部变量互不影响。

Java 中用Synchronized 关键字代表这个方法加锁。

一个简单的锁

public class Counter{

private int count = 0;

public int inc(){

synchronized(this){

return ++count;

}

}

}

如果线程t1.中有语句obj.inc();由于synchornized 锁定，所以先判断是否能锁定，如果可以就锁定并执行++操作，否则就是其他线程在锁定该对象，需要等待线程释放然后执行锁定和执行。

# JVM 调优

什么时候后full gc

1、应用本身在GC堆内的对象行为良好，正常情况下很久都不发生full GC；   
2、应用大量使用了NIO的direct memory，经常、反复的申请DirectByteBuffer   
3、使用了-XX:+DisableExplicitGC

## 1、-XX:+DisableExplicitGC 与 NIO的direct memory

很多人都见过JVM调优建议里使用这个参数，对吧？但是为什么要用它，什么时候应该用而什么时候用了会掉坑里呢？   
  
首先要了解的是这个参数的作用。在Oracle/Sun JDK这个具体实现上，System.gc()的默认效果是引发一次stop-the-world的full GC，对整个GC堆做收集。有几个参数可以改变默认行为，之前[发过一帖简单描述](http://rednaxelafx.iteye.com/blog/1042471" \t "_blank)过，这里就不重复了。关键点是，用了-XX:+DisableExplicitGC参数后，System.gc()的调用就会变成一个空调用，完全不会触发任何GC（但是“函数调用”本身的开销还是存在的哦～）。   
  
为啥要用这个参数呢？最主要的原因是为了防止某些手贱的同学在代码里到处写System.gc()的调用而干扰了程序的正常运行吧。有些应用程序本来可能正常跑一天也不会出一次full GC，但就是因为有人在代码里调用了System.gc()而不得不间歇性被暂停。也有些时候这些调用是在某些库或框架里写的，改不了它们的代码但又不想被这些调用干扰也会用这参数。

在这个例子里，main()里的循环不断申请DirectByteBuffer但并没有引用、使用它们，所以这些DirectByteBuffer应该刚创建出来就已经满足被GC的条件，等下次GC运行的时候就应该可以被回收。   
  
实际上却没这么简单。DirectByteBuffer是种典型的“冰山”对象，也就是说它的Java对象虽然很小很无辜，但它背后却会关联着一定量的native memory资源，而这些资源并不在GC的控制之下，需要自己注意控制好。对JVM如何使用native memory不熟悉的同学可以参考去年JavaOne上IBM的一个演讲，“Where Does All the Native Memory Go”。   
  
Oracle/Sun JDK的实现里，DirectByteBuffer有几处值得注意的地方。   
1、DirectByteBuffer没有finalizer，它的native memory的清理工作是通过sun.misc.Cleaner自动完成的。

## 2、-XX:+DisableExplicitGC 与 Remote Method Invocation (RMI) 与 -Dsun.rmi.dgc.{server|client}.gcInterval=

<http://hllvm.group.iteye.com/group/topic/27945>

3、-XX:+ExplicitGCInvokesConcurrent 或 -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

CMS GC周期中也会做reference processing。所以如果用这两个参数的其中一个，而不是用-XX:+DisableExplicitGC的话，就避开了由full GC带来的长GC pause，同时NIO direct memory的OOM也不会那么容易发生。

4、-XX:+GCLockerInvokesConcurrent

6、-verbose:gc 与 -XX:+PrintGCDetails

经常能看到在推荐的标准参数里这两个参数一起出现。实际上它们有啥关系？   
  
在Oracle/Sun JDK 6里，"java"这个启动程序遇到"-verbosegc"会将其转换为"-verbose:gc"，将启动参数传给HotSpot VM后，HotSpot VM遇到"-verbose:gc"则会当作"-XX:+PrintGC"来处理。   
也就是说 -verbosegc、-verbose:gc、-XX:+PrintGC 三者的作用是完全一样的。   
  
而当HotSpot VM遇到 -XX:+PrintGCDetails 参数时，会顺带把 -XX:+PrintGC 给设置上。   
也就是说 -XX:+PrintGCDetails 包含 -XX:+PrintGC，进而也就包含 -verbose:gc。   
  
既然 -verbose:gc 都被包含了，何必在命令行参数里显式设置它呢？

7、-XX:+UseFastEmptyMethods 与 -XX:+UseFastAccessorMethods

JVM 参数可以设置

Xms  JVM初始分配的堆内存

Xmx JVM最大允许分配的堆内存，按需分配

Xx:MaxPermSize JVM最大允许分配的非堆内存，按需分配

Tomcat 参数可以设置项

maxSpareThreads最大空闲线程如果空闲状态的线程数多于设置的数目，则将这些线程中止，减少这个池中的线程总数。

maxThreads maxThreads:Tomcat可创建的最大的线程数，每一个线程处理一个请求；

minSpareTHreads最小备用线程数，tomcat启动时的初始化的线程数；

acceptCount **acceptCount**：当tomcat起动的线程数达到最大时，接受排队的请求个数，默认值为100

情况1：接受一个请求，此时tomcat起动的线程数没有到达maxThreads，tomcat会起动一个线程来处理此请求。

情况2：接受一个请求，此时tomcat起动的线程数已经到达maxThreads，tomcat会把此请求放入等待队列，等待空闲线程。

情况3：接受一个请求，此时tomcat起动的线程数已经到达maxThreads，等待队列中的请求个数也达到了acceptCount，此时tomcat会直接拒绝此次请求，返回connection refused

**maxThreads如何配置**

一般的服务器操作都包括量方面：1计算（主要消耗cpu），2等待（io、数据库等）

第一种极端情况，如果我们的操作是纯粹的计算，**那么系统响应时间的主要限制就是cpu的运算能力，此时maxThreads应该尽量设的小，降低同一时间内争抢cpu的线程个数，**可以提高计算效率，提高系统的整体处理能力。

第二种极端情况，**如果我们的操作纯粹是IO或者数据库，那么响应时间的主要限制就变为等待外部资源，此时maxThreads应该尽量设的大，这样才能提高同时处理请求的个数，从而提高系统整体的处理能力。**此情况下因为tomcat同时处理的请求量会比较大，所以需要关注一下tomcat的虚拟机内存设置和linux的open file限制。

我在测试时遇到一个问题，**maxThreads**我设置的比较大比如3000，当服务的线程数大到一定程度时，一般是2000出头，单次请求的响应时间就会急剧的增加，

百思不得其解这是为什么，四处寻求答案无果，最后我总结的原因可能是**cpu在线程切换时消耗的时间随着线程数量的增加越来越大，**

cpu把大多数时间都用来在这2000多个线程直接切换上了，当然cpu就没有时间来处理我们的程序了。

以前一直简单的认为多线程=高效率。。其实多线程本身并不能提高cpu效率，线程过多反而会降低cpu效率。

当cpu核心数<线程数时，cpu就需要在多个线程直接来回切换，以保证每个线程都会获得cpu时间，即通常我们说的并发执行。

所以**maxThreads**的配置绝对不是越大越好。

现实应用中，我们的操作都会包含以上两种类型（计算、等待），所以maxThreads的配置并没有一个最优值，一定要根据具体情况来配置。

最好的做法是：在不断测试的基础上，不断调整、优化，才能得到最合理的配置。

**acceptCount的配置**，我一般是设置的跟maxThreads一样大，这个值应该是主要根据应用的访问峰值与平均值来权衡配置的。

**如果设的较小，可以保证接受的请求较快相应，但是超出的请求可能就直接被拒绝**

**如果设的较大，可能就会出现大量的请求超时的情况，因为我们系统的处理能力是一定的。**

## JVM 启动参数

-Dlog.path.prefix=/export/home/tomcat/logs/bs-list/server1 -Ddeploy.project.id=2975 -Ddeploy.app.id=10505 -Ddeploy.app.name=bs-list -Ddeploy.instance.id=7835107 -Ddeploy.data.path=/export/home/tomcat/bs.list.jd.local -Ddeploy.logs.path=/export/home/tomcat/logs/bs.list.jd.local -Ddeploy.instance.logs.path=/export/home/tomcat/logs/bs.list.jd.local/server1 -Dins\_id=7835107 -Ddeploy.dynamic.config.dir=/export/App/bs.list.jd.local/ -Djava.util.logging.config.file=/export/Domains/bs.list.jd.local/server1/conf/logging.properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djava.library.path=/usr/local/lib -Xms1024m -Xmx1024m -XX:MaxPermSize=1024m -Djava.awt.headless=true -Dsun.net.client.defaultConnectTimeout=60000 -Dsun.net.client.defaultReadTimeout=60000 -Djmagick.systemclassloader=no -Dnetworkaddress.cache.ttl=300 -Dsun.net.inetaddr.ttl=300 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/export/Domains/bs.list.jd.local/server1/logs -XX:ErrorFile=/export/Domains/bs.list.jd.local/server1/logs/java\_error\_%p.log -Djava.endorsed.dirs=/export/servers/tomcat7.0.61/endorsed -Dcatalina.base=/export/Domains/bs.list.jd.local/server1 -Dcatalina.home=/export/servers/tomcat7.0.61 -Djava.io.tmpdir=/export/Domains/bs.list.jd.local/server1/temp

# 计算机容量计算

1比特(bit)=0.125字节(b)  
  
1B就是1个字节。  
  
Byte、KB、B、MB、GB之间的关系是：   
Bit——比特  
B ——字节   
KB——千字节   
MB——兆字节   
GB——吉字节   
TB——太字节   
  
1B=8 Bit  
1KB＝1024B   
1MB＝1024KB   
1GB＝1024MB   
1TB=1024GB

Int 4个字节

Char 取值范围 -128-127

当 Runnable run 方法中抛出异常的话，线程就会终止，这种情况称为线程泄漏。JDK 5 新加入了 ThreadPoolExecutor 的线程池执行器，在产生线程泄漏时，会补充一个新的线程到池中去。

Runnable

在java中可有两种方式实现多线程，一种是继承Thread类，一种是实现Runnable接口；Thread类是在java.lang包中定义的。一个类只要继承了Thread类同时覆写了本类中的run()方法就可以实现多线程操作了，但是一个类只能继承一个父类，这是此方法的局限。

在实际开发中一个多线程的操作很少使用Thread类，而是通过Runnable接口完成。

# JVM 常用命令

Java程序运行过程中会遇到各种jVM问题,比如JVM崩溃，挂起，死锁，Full Gc，突然或持续CPU使用率过高,OOM 等这些问题。JAVA那些工具可以查询呢。一般有JVM自带工具或者调用JDK API 来获取JVM 信息。

在本地和可以连接的服务器如果有jvm问题我们一般用JVM自带工具进行问题排查，如果在生产环境有的公司没有服务器权限，就可以利用JDK API 进行问题排除。

[jstatd](http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstatd.html)