java总结

# 系统需求

# 项目负责

# 项目前思考

# 项目前攻破

# 项目中使用的技术总结

## webapp项目中的filter与Servlet关系

### 例子1:



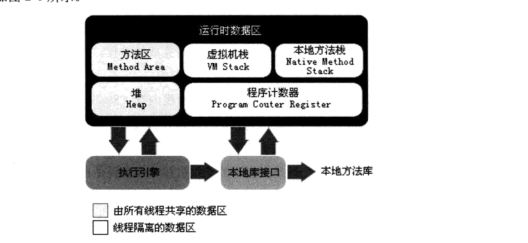
### 例子2：



### 总结：



## java内存



### 程序计数器(Program Counter register)

相当于运行的指令，每条线程间的指令是要独立的，因此每一个这个区域的线程都是隔离的

### 虚拟机栈 (Vm Stack)

对于线程来说，也是单独隔离的。周期与线程一致。虚拟机栈相当于方式执行的内存模型，每一个方法执行时，都会创建一个栈帧(Stack Frame)，用于存储局部变量表、操作栈、动态链接、出口等。每个方法的调用过程，其实就是入栈出栈的过程。其实也就是一些局部变量表。

对于栈溢出的异常，我目前只想到了：无限递归和重复调用(比如A调用B，而在B中又调用A，导致了栈无法停止)，这样就有可能出现StackOverFlowError

### 堆(Heap)

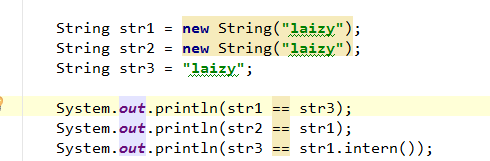
线程共享，此空间用于存放对象实例空间。java的内存管理都是针对此区域的回收，将其按照内存回收的角度，可分为新生代和年老代，具体可分为Edge、From、To和Old

### 方法区(Method Area、MaxPermSize)

线程共享，存储被虚拟机装载了的类信息、常量、静态变量、即时编译器后的代码等数据，该区域还有一个别名叫Non-heap(非堆)，用于跟heap堆分开

#### 运行时常量池(Runtime Constant Pool)

属于方法区的一部分，类中包含的字面量和符号引用，都存放于运行时常量池。这一块区域并不一定是在编译时，就放入常量池，运行区间也有可能将常量放入常量池，常见的有String类的intern()方法。再补充介绍一点：存在于.class文件中的常量池，在运行期被JVM装载，并且可以扩充。String的intern()方法就是扩充常量池的一个 方法；当一个String实例str调用intern()方法时，Java查找常量池中是否有相同Unicode的字符串常量，如果有，则返回其的引用， 如果没有，则在常量池中增加一个Unicode等于str的字符串并返回它的引用；

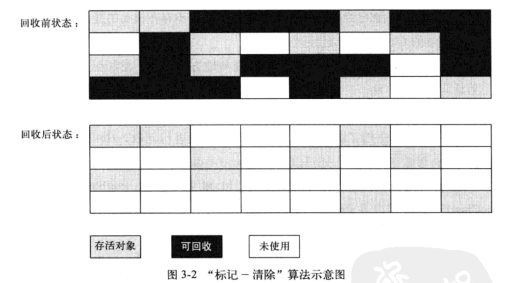


### 本地方法栈 (Native Method Stack)

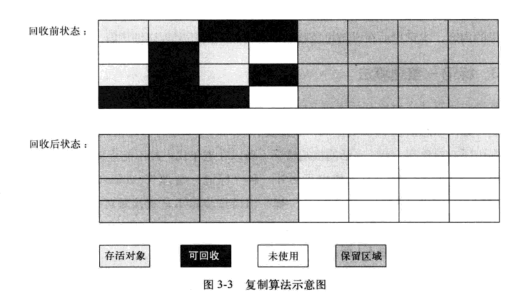
也就是Native的方法服务区

## java内存的回收算法

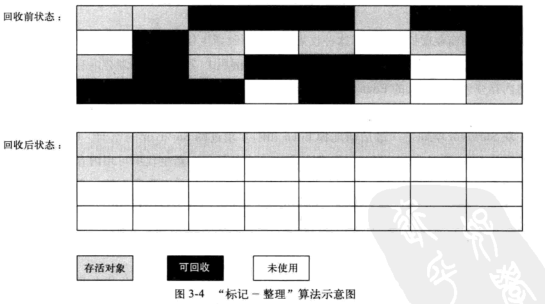
### 标记-清除



### 复制



### 标记-整理



### 分代收集

java堆分为新生代和老生代，新生代由于只有少量存活，因此采用了复制算法，而老生代中对象存活率高，没有额外空间做担保，因此采用了标记清除、标记整理

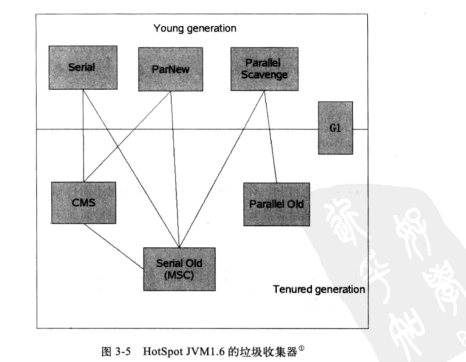
|  |  |  |
| --- | --- | --- |
| 算法 | 优点 | 缺点 |
| 标记-整理 | 效率高，回收高 | 存在内存碎片，使得空间无法再次利用 |
| 复制 | 不会存在内存碎片，空间充分利用 | 内存只能使用一半 |
| 标记-整理 | 集合了前两种算法的优点 |  |

## 垃圾收集器

回收算法的具体实现，这里解释一下，并行与并发

并行：多线程收集器同时工作，但是用户线程还是等待

并发：收集线程与用户线程并发操作



### Serial收集器

在jdk1.3以前，是新生代的唯一收集器，其属于单线程收集，当它开始运行的时候，必须停止所有的工作线程(Stop the world)，适用于-Client环境下



### ParNew收集器

ParNew是Serial的多线程版本，还适用于控制参数-XX:SurviviorRatio、-XX:PretenureSizeThresHold、-XX:HandlePromotionFailure



许多模式下-Server的首先垃圾收集器，因为也只有它能够与CMS(老生代回收器)结合使用，可以使用参数-XX:UseConcMarkSweepGC来默认指定ParNew收集器，或者-XX:UseNewParNew来指定

### Paralle Scavenge

并行的收集器，复制算法，该收集器的目的在于，可控制程序的吞吐量。

吞吐量=运行时间/(运行时间+垃圾收集时间)

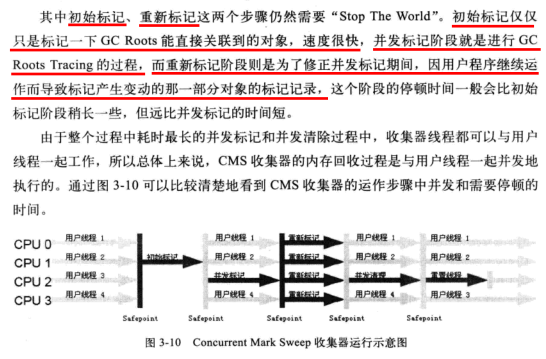
-XX:MaxGCPauseMills(gc收集停顿时间)

-XX:GCTimeRatio(吞吐量)，那么垃圾回收时间就占了(1/(1+n))，默认值是99

### CMS

以获取最短收集暂停时间的回收器，目前大部分运用于服务器B/s架构上等注重服务器响应速度上面。采用了标志-清除的算法。其算法也是比较复杂的：

1. 初始标记
2. 并发标记
3. 重新标记
4. 并发清除



缺点：

1.CPU敏感

2.无法收集浮动垃圾，-XX:CMSInitialtingOccupancyFraction来控制老生代使用的空间所占比开始收集

3.存在空间碎片，可使用-XX:UseCMSCompactAtFullCollection来执行碎片处理；-XX:CMSFullGCsBeforeCompaction，经过多少次回收后进行碎片处理

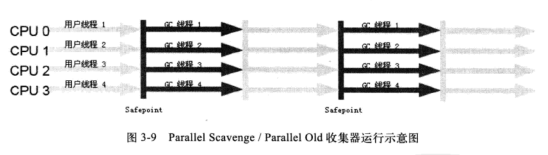
### Serial Old

老生代的收集器，采用了单线程的标记-整理算法，其主要也是在client的情况下运行，在server环境下，则是配合Paralle进行收集，或者作为CMS Failure的时候备选者

=

### Paralle Old

Paralle老生代的垃圾收集器，采用了标记-整理算法，jdk1.6开始



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 垃圾回收器 | 适用命令 | 内存分配规则 | 回收算法 | 适用区域 |
| Serial | -XX:+UseSerialGC | 先新生代、后老生代，如果新生代不足空间，发生gc，会将新生代空间拷贝到老年代，然后再继续从新生代开辟  gc输出信息  def  new generation  tenured generation  compacting perm gen | 复制算法，单线程执行，简单高效 | 适用于-client、新生代 |
| Serial Old | 同上 | 同上 | 同上 | 适用于老生代 |
| ParNew | -XX:+UseParNewGC | 输出  par  new generation  老生代使用了Serial | serial的多线程版 | 新生代 |
| Parallel Scavenge | -XX:+UseParallelGC | 先新生代、再老生代，如果新生代空间不足，老年代空间足够，则直接在老生代开辟空间；如果此时老年代无法开辟足够空间，则发生一次gc，然后将新生代的空间拷贝复制到老生代，接着在新生代开辟空间  输出  PSYoungGen  ParOldGen(老生代使用了ParOld)  PSPermGen | 并行收集器，复制算法 | 新生代 |
| CMS | -XX:+UseConcMarkSweepGC | 新生代使用了ParNew的垃圾回收器  新分配新生代再分配老生代，如果新生代空间不足，则移入老生代。  par new  generation新生代  concurrent mark-sweep generation cms老生代 | 标记整理 | 老生代，只能搭配ParNew新生代 |
|  |  |  |  |  |
|  |  |  |  |  |

## jvm优化

### -server:

tomcat默认是以一种叫java –client的模式来运行的，-server意味着更大、更高并发处理能力

### -Xms

JVM初始堆内存

即JVM内存设置了，把Xms与Xmx两个值设成一样是最优的做法

一个系统随着并发数越来越高，它的内存使用情况逐步上升，上升到最高点不能上升了，开始回落，你们不要认为这个回落就是好事情，由其是大起大落，在内存回落时它付出的代价是CPU高速开始运转进行垃圾回收，此时严重的甚至会造成你的系统出现“卡壳”就是你在好好的操作，突然网页像死在那边一样几秒甚至十几秒时间，因为JVM正在进行垃圾回收。

### -Xmx

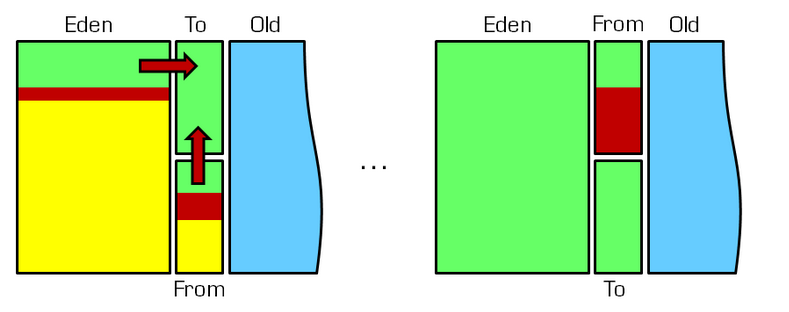
JVM最大堆内存

### -Xmn

年轻代堆栈, the size of the heap for the young generation，可分为一个Eden、两个Survior区(又可分为FROM与To)

虚拟机默认的Eden与Survior的比例为8:1，(Survivor的大小是512等的倍数，如果超过，则向下取整。比如如今分配mn12m，那么照计算survior的大小为1.2M，但是实际只有1024k)

年老代就是Old，就是mx-mn的值就是剩下年老代的大小



### -Xss

java线程的栈大小，在java中，线程的开辟，都会独立去申请栈空间，但是会与主线程共用堆空间、常量空间、方法空间等

### -verbose:gc

输出gc的过程

### -XX:+PrintGCDetails

输出内存具体分配，新生代、老生代、内存等分配情况

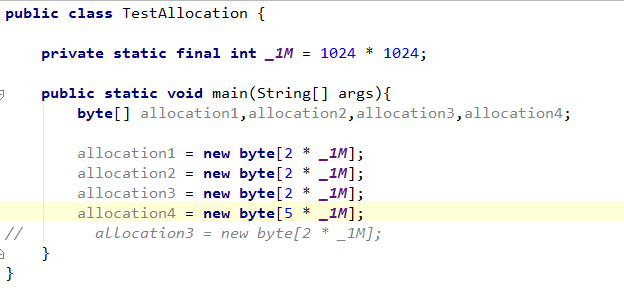
### 举例1

#### jvm参数

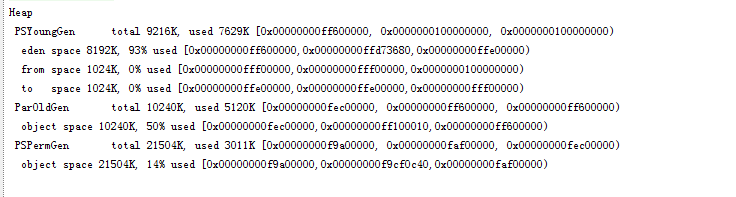
-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8 -XX:+PrintGCDetails

默认采用了UseParallelGC的算法

#### 代码



#### 运行结果



新生代：

mn为10M，而SurvivorRatio=8，那么eden:from:to=8:1:1，所以eden为8M，from和to都是1m

老生代

总对大小mx为20M，所以老生代为20-10=10M

#### PrintGC描述

[GC [PSYoungGen(年轻代回收器): 1096K(回收前大小)->504K(回收后大小)(4608K(年轻代总代下))] 6216K(回收前堆大小)->5712K(回收后堆大小)(11776K(总大小)), 0.0033323 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[GC [PSYoungGen: 504K->488K(4608K)] 5712K->5704K(11776K), 0.0023845 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC [PSYoungGen: 488K->0K(4608K)] [ParOldGen: 5216K->555K(6144K)] 5704K->555K(10752K) [PSPermGen: 3073K->3072K(16384K)], 0.0276401 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]

### final占用的空间是在堆还是非堆？



之前错误的以为，以上会占用非堆内存6M；真实情况是，非堆内存只占用了6个字节，引用地址的空间。具体开辟的空间6m是在堆内存中

## Transient

表示该属性不持久化，即不被序列化

# java多线程

## 扩展集合类，实现没有就添加

### 拓展原来的集合类



优点：所有的api都能够以vector的方式来使用

缺点：如果后续vector的同步变量有变化，则会导致该同步机制失败

### 新增辅助类(客户端加锁)



缺点：无法真正实现同步，比如线程1，调用了putIfAbsend，线程2调用了list.add的方式，并无法保证。而且引入了新类，每一次操作都得新增辅助类，使用复杂。



虽然实现了同步的机制，可查看Collections.synchron的方法，发现其是新增一个扩展类SynchronList的，并以list作为同步信号量。

缺点：无法正常使用List的api方式，必须每次都以辅助类来操作。是扩展了功能，并不是扩展了类

### 扩展类，并使用组合方式



优点：既不受底层List的同步信号量影响，又可以以最简单的方式来调用api

## SyncheronzedQueue

同步队列，只有先take的任务要求，put才能正常插入，而且数量只能保持1

## 闭锁(CountDownlaunch)

countDownload计数减1

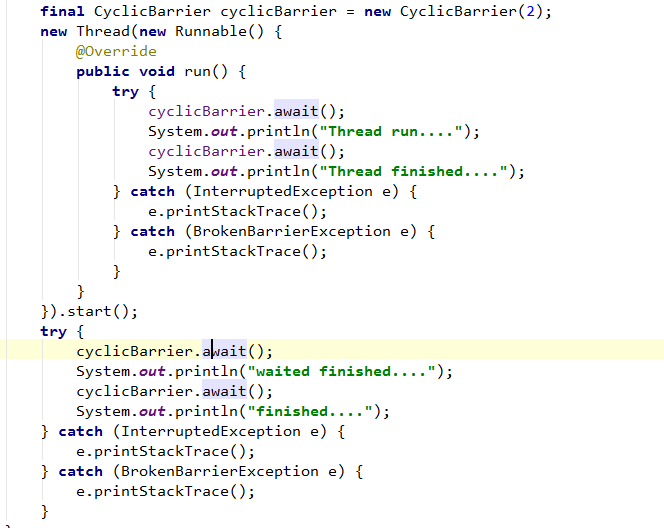
await等待，是否次数已经减至0

相当于一个总任务，在等待其它分支任务，是否都处理了。如果分支任务都处理了，总任务在继续执行

## 栅栏(CyclicBarrier)

所有任务都在一起执行，但是都在等待互相的任务完成，才能进行下一步。

比如三个人在打桩，只有等三个人都打桩完成，才能继续建桥



如上图，当执行完成一次之后，可进入下一轮循环等待

## 多线程同步

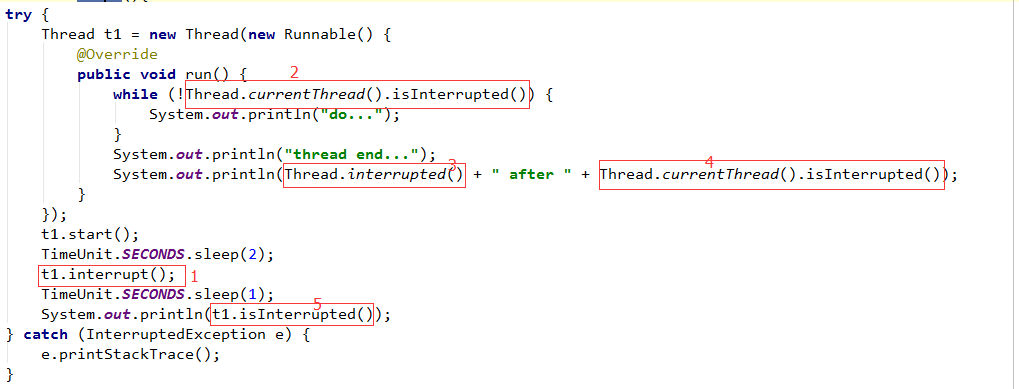
|  |  |  |
| --- | --- | --- |
| 同步方式 | 优点 | 缺点 |
| 内置锁(Synchronize) | 简单方便 | 无法响应线程的interrupt |
| Lock | 1. 可以通过tryLock来轮询获取锁，从而避免死锁 2. 通过lockInterruptibly，是会马上响应对应的interrupt信号，从而中断 | 控制不好，可能产生死锁。比如当lock之后，没有在finally中unlock的话 |

|  |  |
| --- | --- |
|  | 含义 |
| CountDownLatch | 主任务，等待其它子任务完成 |
| CyclicBarrier | 子任务都在等在对方完成，然后进入下一轮等待 |
| Semaphore | 资源的控制，比如当前只有5个厕所，那么在acquire之后，如果已经达到5个，就必须等待。当用完之后可用release归还 |

## 线程

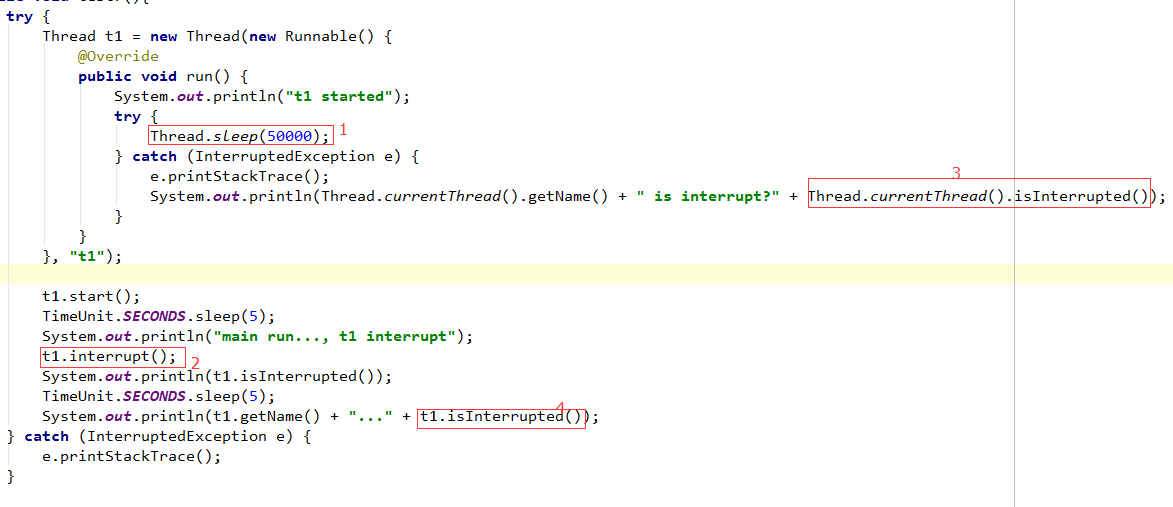
### 中断策略

#### 即时响应



1. 在1中使用了interrupt的方式来中断线程，但是这只是设置终端状态为true，线程并没有响应。
2. 所以在2中获取的状态为true，结束循环
3. 在调用了interrupted之后，返回了上次的中断状态，并清空
4. 由于3中清除了中断状态，所以为false
5. 同4

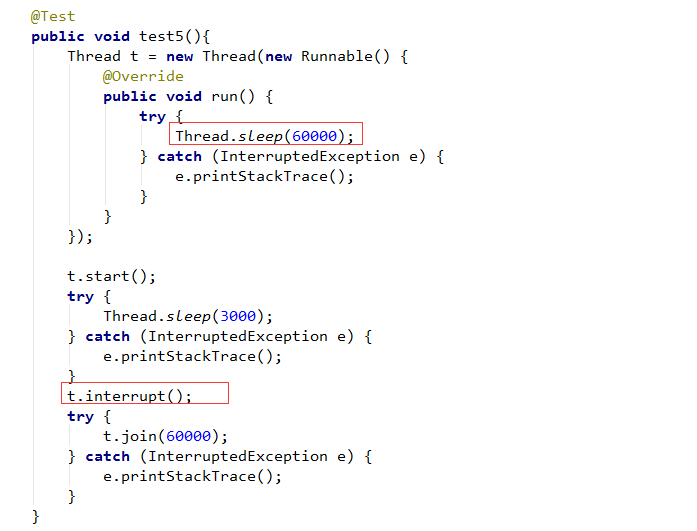
#### sleep、await、join



1. 采用了sleep方式，将线程挂取，这里可以响应中断信号
2. 采用interrupt来设置中断状态
3. 1中响应了中断信号，抛出InterruptedException，但是同时jvm将中断状态清除，所以输出false
4. 同3

### interrupt

中断线程，并设置中断状态，需要处理线程对终端状态进行处理



如上图，t会被主线程的interrupt，从而抛出InterruptedException

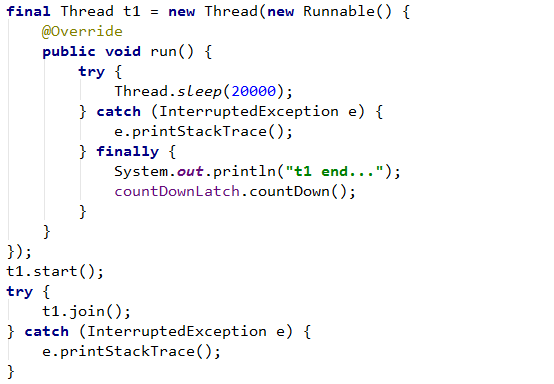
### interrupted

清除终端的状态，并返回上一次的中断状态。比如当处理的时候，catch InterruptException，表示线程已经中断了，这时候需要将异常抛出，并调用该interrupted来清除状态。因为有异常，表示线程已经中断

### isInterrupted

当前线程的中断状态

### join



表示在调用join的线程中，会等待join的对象的完成

### 线程池

awaitTermination等待线程池关闭，每x的unit的时间检查一下

isTerminated线程池是否关闭状态

# Java编程

## 泛化

也叫类型擦除，在编译时检测，但是运行时就会无法获取该类型

### 不要使用原生类型

比如在编码中使用List等原生类型，虽然编译后，也会被类型擦除。

### 不能创建泛型数组

因为数组指定的类型必须是具体化，如果使用List<String>[]，由于在运行时无法确定List对应的类型，所以不是一个固定类型

所以不能通过new T[]这种泛型来创建数组，但是可使用(T[])(new Object[x])这样子来转换

### 数组是协变的

即Sub[] 是 Parent[]的类型，而List<Sub>则不是List<Parent>的类型

### List<?>

通用匹配，匹配任何一种的List，但是无法添加任何元素(除了null)

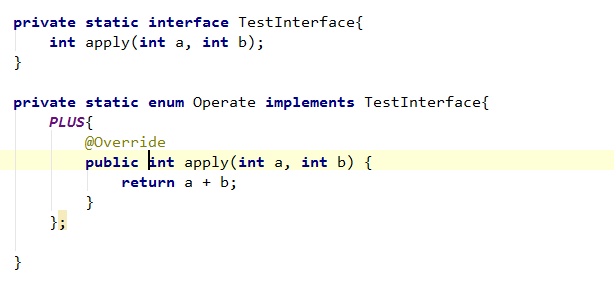
### PECS

Product- Extends(为列表新增元素)

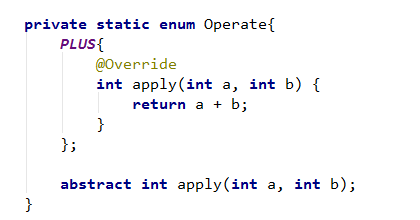
Customer-Super(从列表获取元素)

## 枚举

### 实现接口



### 抽象方法



## 重载与重写

重载的工作是在编译时就指定了，而重写的话，是根据不同子类在运行时，不同多态。

## 异常

如果在方法可用Exception作为程序的分支，但是如果只是抛出Exception，会使得这种异常没有具体的意义。所以findbug需要用户自定义异常，然后再抛出。

而对于有时候底层有抛出IOException和其他类型的异常，用户在service可以针对这些异常进行转化为自己的异常，但是要保持统一做法。这样才能够让异常统一，不要在一个service尝试抛出不同的异常

## HashCode与equals

hashCode类似于对象的开辟地址，当hashCode一样的时候，会将该hashCode一致的放在一个list中；获取也一样；比如当map.get(null)和map.get(0)，虽然hashCode一样，但是equals不一样，所以可以取到不同对象

equals:两个对象是否等价，不一定是地址一致。理论上来说，当对象equals，那么对应的hashCode也必须一致。

## Java –cp

Java –cp aa.jar;as.jar XXX

类似于aa.jar、as.jar添加到classpath中

## TreeMap妙用



为不同的提供商插入视频，

普通一点的json串：{1:[{“index”:1,”videoId”:20}]}

高级一点的json串：{1:{2:3,4:5}}，采用了treeMap来使用index:videoId的对应关系

## JMI



RmiServer通过url注册到RMIRegistry；提供外部访问接口1099，或者其它。并且本身也提供了一个远程访问的UnicastRemoteObject，所以这里会再提供另外一个端口

RmiClient通过RMIRegistry.find来获取对应的Server实例，如Stud。通过远程调用，与服务端开启UnicastRemoteObject的socket进行通讯，

这样处理的目的是：具体远程Server对外是隐秘的，只有RMIRegistry的端口对外开放，并能提供多个实例注册

## Permgen溢出跟踪



由于图中，将loader加载在内存中cache住，导致permgen oom时，无法进行回收，然后就抛出异常了。

## tomcat假死状态跟踪

### Jstack

jstack –F pid 可以查看线程状态

### jstat



S0: survivor space第0块区域的使用大小，也就是from

S1: survivor space第1块区域的使用大小，也就是to

E：Eden新生代的占用大小

O：Old老生代的使用大小

P: Permspace的使用大小

YGC: 新生代的回收次数

YGCT: 新生代回收使用时长

FGC: Full GC的次数

FGCT: full GC的时间

GCT: 垃圾回收总时间

### jmap

jmap –heap pid

后来发现是在hibernate中使用了List的去查询数据，会在session中缓存数据，导致内存越来越大。下图是hibernate加载list的一个大概流程图。读list会加载到sesseion中。



## java子类覆盖父类，Exception不应该超出父类的

这是因为如果：

try{ parent.do()} catch(ParentException e) {}

如果children的do方法，抛出了比ParentException范围更广的话，会导致代码出错

## KeyTool生成的证书及https加密

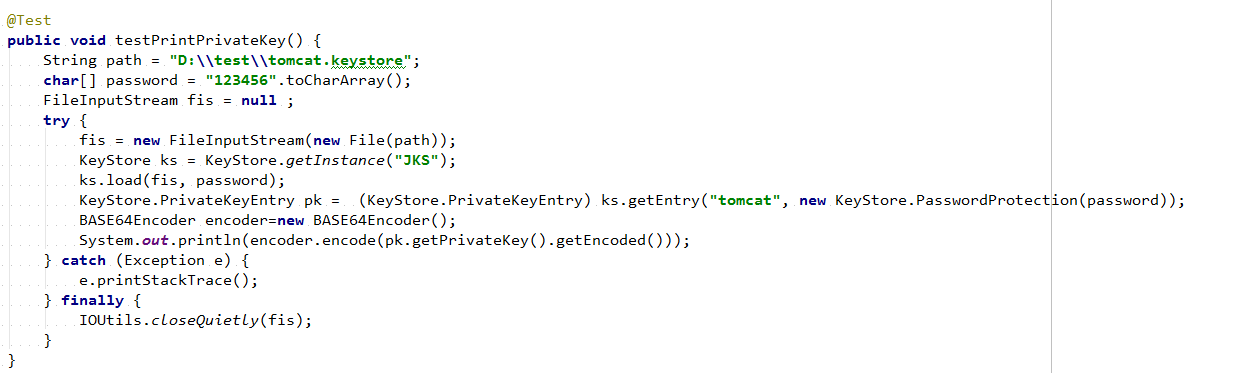
以前不明白https的加密规则；现在看了http协议的那本书，算是有点理解。

客户端-》使用证书（https，通过CA认证的话就是绿色，否则就是红色）中的公钥-》服务端-》使用私钥(保存在服务端)进行解密，不会被外界给破解。

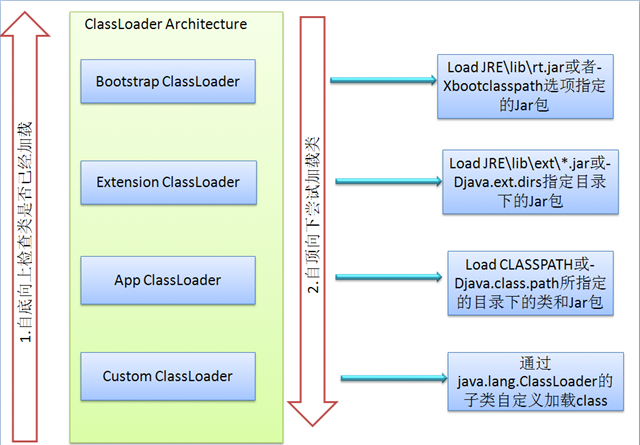
keytool生成了对应的keystore之后，这个里面是含有私钥和公钥。可使用

keytool –export来导出公钥(证书)，这个证书就可以丢给CA机构去认证

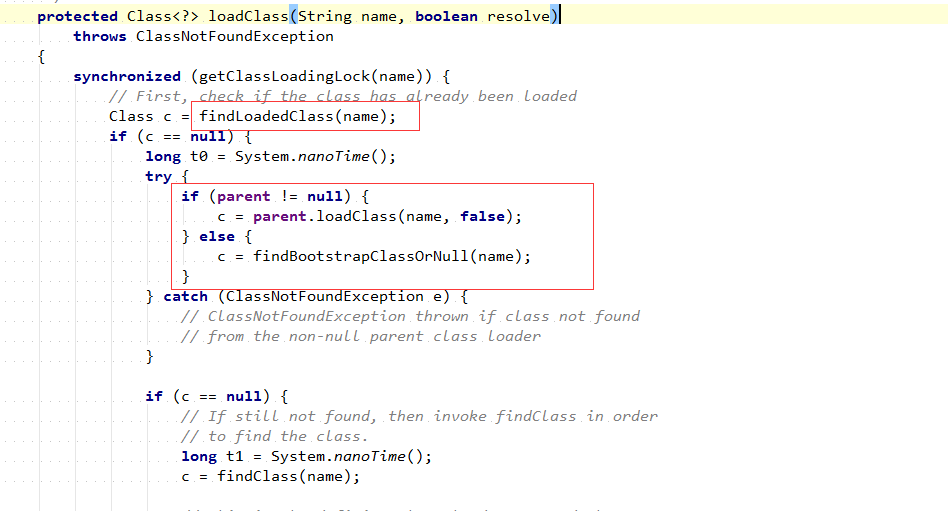
但是无法导出私钥。需要使用java代码协助来输出



## ClassLoader的加载顺序



可根据代码查看，如下图(ClassLoader代码片段)。首先第一个矩形先找出当前是否已经加载过了，如果没有，先判断CLassLoad是否有parent，如果有，先用parent去load，除非parent为空，才使用root去加载。因为可以看出：是由上图的自底而上判断类是否加载，自顶而上加载类



ClassLoader代码片段

# 项目后期思考

# 项目中觉得亮点

## jdk1.7.0.55中HttpClient出现java.util.NoSuchElementException: null

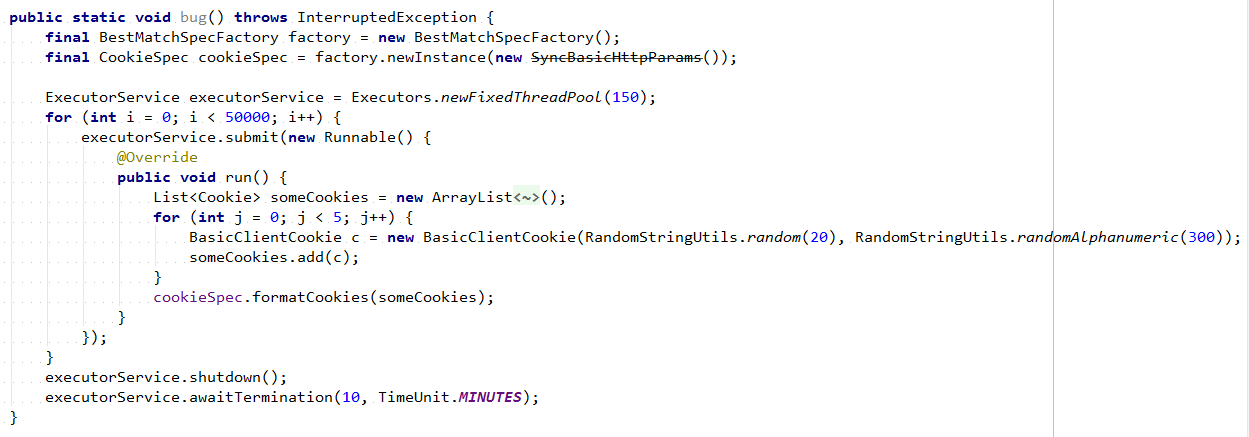


问题原因：此异常是由于jdk  JIT compiler optimization 导致，bug id 8021898；jdk1.7.0\_25到1.7.0\_55这几个版本都存在此bug。但是目前还没定位出来，具体触发需要达到极限的条件。

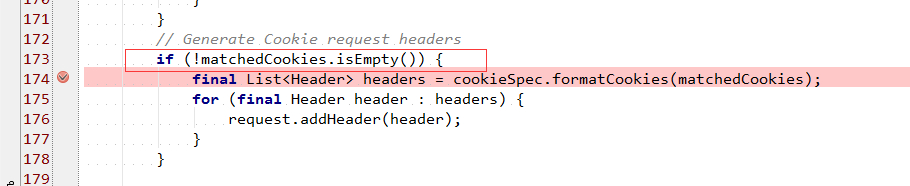
<http://bugs.java.com/view_bug.do?bug_id=8021898>。<https://issues.apache.org/jira/browse/HTTPCLIENT-1173>

解决方法：升级jdk版本至1.7.0.60以上既可。

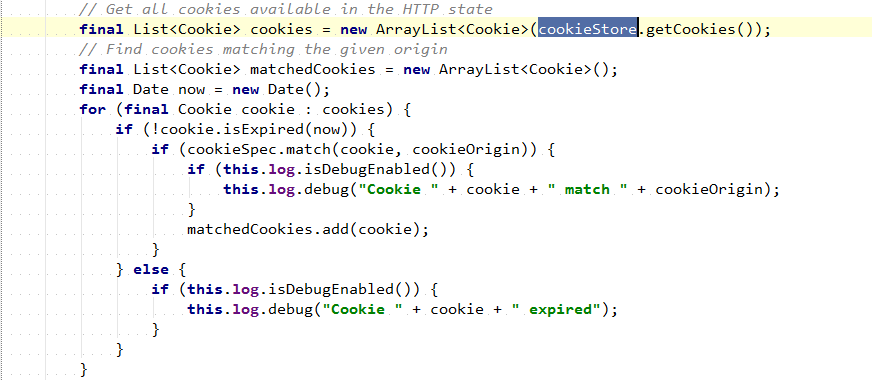
* 分析：可是为啥重启完tomcat之后，又恢复正常？是不是要达到某种场景极限呢？对于httpClient的使用达到某种要求？
* 后来网上提供了一个demo:可以快速复现出bug，但是也没有指出具体场景。也算概率性，只不过出现的机会更简单及更大。代码内容大概如下：就是模拟那个出错代码，给它新增不一样的cookies



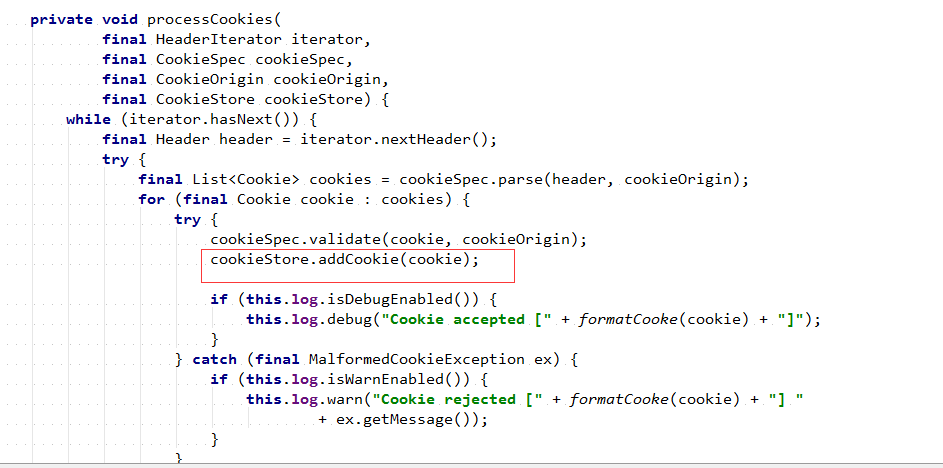
* 想去尝试利用HttpClient发送http请求，是否也会起到该错误。首先将maxTotal和defaultMaxPerRoutCount都设置为较大。但是却没有引起错误。跟踪代码发现。在RequestAddCookies中，必须满足已有cookies不能为空，才会触发BestCookieSpec中的formatCookies



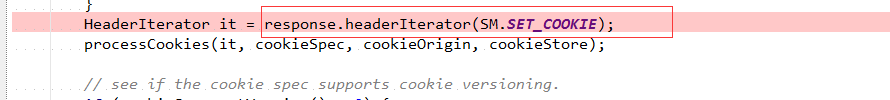
* 而上面的cookies来源，是由cookieStore获取得来。那么cookieStore又是什么东西呢？



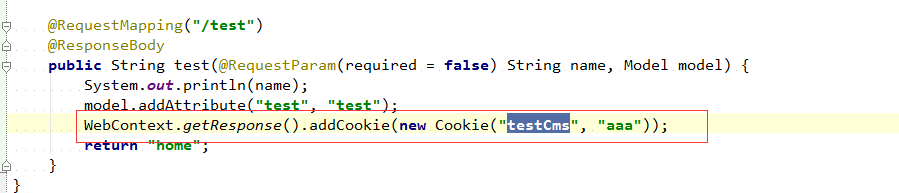
* ResponseProcessCookies中process，如果发现response中有cookie，才会往本地的cookieStore加上cookie.



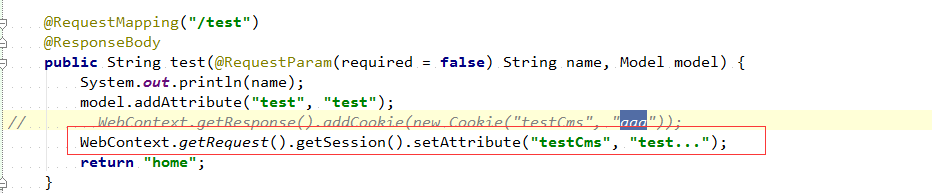
* 上面所说的cookies是由response里面而来。



也就是请求处理的源端的response.addCookie而来

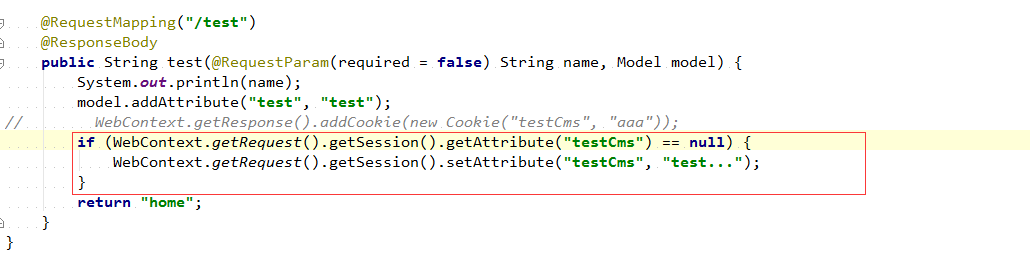


或者往session里面塞数据也是可以的

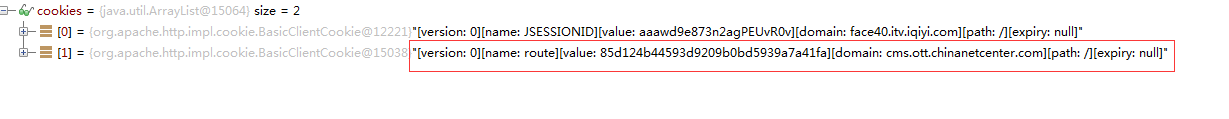


* 可是代码中并没有给response.addCookie的方法。那么到底是什么时候addCookie的呢？是tomcat可以给response addCookie还是nginx可以给response addCookie。?又或者是客户端上传cookie，然后服务端会自动response?

1. 从以下代码中，如果客户端已经有session的值，就不会再回塞。但是这样客户端就不会收到cookie的信息，说明客户端上传cookie，服务端不会再回塞。



1. 后面发现是由于在nginx中配置了SessionStricky，那么nginx会往response加塞route的cookie信息。这样就会在httpClient中触发



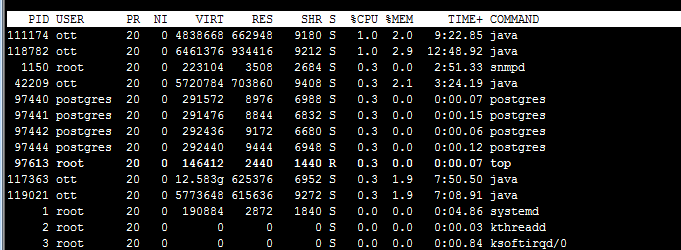
## 在windows配置了tomcat的acceptCount=2,maxConnections=1，然后使用httpClient分别请求3次一个接口，正常；而在Linux下却会出现一个connectionTime out。

1. 首先tomcat中对于maxConnections的解析是：在XXEndPoint（比如NioEndPoint）中有个countUpOrAwaitConnection，这里就表示如果达到maxConnections之后，必须等待
2. 而对于acceptCount，则是使用了serverSocket.bind(xxx,acceptCount)这里表示socket连接后等待处理的队列数。如果满了，之后就是直接refuse connection
3. 使用了httpClient的包，在linux发送了3次请求，结果会有一个refuse，那么跟配置解释不通。于是怀疑是不是socket的配置问题？然后直接使用了socket进行操作之后，也是出现了同样的结果？于是使用了wireshark的抓包。结果发现3个同时发包的时候，会出现一个rest。那么是不是因为有其它形式的在这个端口发包，导致失败呢。(tcp.port=9012)来过滤发包。结果并不是。

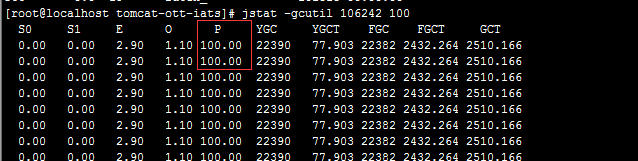
**总结：**如果延迟1s之后，就能够正常100%的加入队列，不会缺少一个。不晓得是不是因为backlog会加1的问题

## tomcat假死原因追查

1. 查看tomcat进程的占用cpu和使用内存情况。ps –ef，因为这次假死的pid为106242。没有发现该线程有较大的异常



1. 使用jstack pid > iats.log。可以打出线程的堆栈信息。但是打了几次，发现都是WAITING，也算比较正常
2. jstat –gcutil pid 1000.每1s打印gc的信息



可以看出perfermemory占了100%，所以常量区内存爆出。然后去查看运行情况，发现没有配置performemory的值，所以后来增加了JAVA\_OPTS中加了-XPermSize=128M。