1. lazy loading延迟加载，就是不是在初始化时loading，而是推迟到必须加载时再loading
2. 数组的查询能力是非常强的，大小为几百万的数组，查询其中的任意一个索引的值所花的时间几乎为0

## 2、单例模式的七种写法

最安全的写法是用单例模式实现的：

先来说下Java枚举的用法：

首先，枚举类似类，一个枚举可以拥有成员变量、成员方法、构造方法。先来看枚举最基本的用法：

public enum EnumEg {  
 *A*,*B*,*C*,*D*;  
}

创建enum时，编译器会自动为我们生成一个继承自java.lang.Enum的类，我们上面的enum可以简单看作：

class EnumEg extends Enum {  
 public static final EnumEg *A*;  
 public static final EnumEg *B*;

…  
}

对于上面的例子，我们可以把EnumEg看作一个类，而把A,B,C,D看作类EnumEg的实例。

当然这个构建的实例的过程不是我们做的，一个enum的构造方法限制是private的，也就是不允许我们在外部调用。

在EnumEg中，我们可以定义变量以及方法，看下面的代码：

public enum EnumEg {  
 *A*,*B*,*C*,*D*;  
 static int *value*;  
 public static int getValue () {  
 return *value*;  
 }  
 String type;  
 public String getType () {  
 return type;  
 }  
}

上面提到A,B,C,D是枚举类EnumEg的实例，那么我们调用方法getType()就可以通过这样调用EnumEg.A.getType()。

另外，每个实例可以实现自己的实例方法来覆盖原本方法，实现属于自己的定制。

public enum EnumEg {  
 *A*{  
 public String getType () {  
 return type;  
 }  
 },*B*,*C*,*D*;  
 static int *value*;  
 public static int getValue () {  
 return *value*;  
 }  
 String type;  
 public String getType () {  
 return type;  
 }  
}

除此之外，我们还可以添加抽象方法来强制各个实例来实现自己的逻辑：

public enum EnumEg {  
 *A*{  
 public String getType () {  
 return type;  
 }  
 },*B*{  
 public String getType () {  
 return type;  
 }  
 },*C*{  
 public String getType () {  
 return type;  
 }  
 },*D*{  
 public String getType () {  
 return type;  
 }  
 };  
 static int *value*;  
 public static int getValue () {  
 return *value*;  
 }  
 String type;  
 public abstract String getType ();  
}

有了上面的基础，我们来看下用枚举来实现单例

public enum SomeThing {  
 *INSTANCE*;  
 private Resource instance;  
 SomeThing () {  
 instance = new Resource();  
 }  
 public Resource getInstance () {  
 return instance;  
 }  
}

SomeThing.INSTANCE.getInstance()来获取实例。

下面来看下，为什么这样可以保证单例：

首先，在枚举中我们明确了构造方法限制为私有，在我们访问枚举实例时会执行构造方法，同时每个枚举实例都是static final类型的，也就表明只能被实例化一次。在调用构造方法时，我们的单例被实例化。也就是说，因为enum中的实例被保证只会被实例化一次，所以我们的INSTANCE也就被保证实例化一次。

3、java类的加载顺序

1静态变量

2静态代码块

3非静态变量

4非静态代码块

5构造方法

4、synchronized的用法

有两种用法，一个是synchronized方法一个是synchronized块。

下面选取其中的一个例子，具体来阐述一下问题吧，并且借此机会也总结一下关于synchronized关键字的问题：

private static AccountSeqHelper instance;

private static Object singleLock = new Object();

public static AccountSeqHelper getInstance()

{

if (instance == null)

{

synchronized (singleLock)

{

if (instance == null)

{

instance = new AccountSeqHelper();//\*\*

}

}

}

return instance;

}

上面的代码就是一个典型的单例模式，且使用了双重检查锁(使用双重检查锁的目的就是减少synchronized所带来的开销，第一次检查instance为null就不再执行后面的synchronized部分)但是此单例模式会存在缺陷。第9行并不是一个原子操作，这里可大体上分为三步，第一步①是先在heap内存上为对象开辟空间，第二步②是为对象的各项参数进行初始化，第三步③是将instance指向分配的内存空间，这里需要考虑指令重排的问题。

由于②要依赖于①，但是③并不依赖于②，所以重排之后会变成①③②，这样当一个线程初始化对象之前就将instance指向了分配的内存空间，恰好另一个线程判断instance不为null，然后将其返回使用，导致出错。

这里的解决方法是使用volatile关键字修饰instance，可以保证内存可见性和防止与instance有关的代码进行指令重排，避免上面的情况出现，但是这里要注意volatile不能保证原子操作。

下面对synchronized关键字进行一下总结：

该关键字可以修饰方法也可以修饰代码块。在修饰非静态方法时，相当于每个类实例对应一把锁。在修饰静态方法时，相当于每个类对应一把锁。

(1)该关键字在修饰方法时的作用范围分两种：某个对象的范围内，某个类的范围内。

①某个对象范围内：作用是可以防止多个线程同时访问这个对象的synchronized方法(这里有一点要注意，如果一个对象有多个synchronized方法，只要一个线程访问了其中一个synchronized方法，其他线程就不能同时访问这个对象中的任何一个synchronized方法，这样做的目的就是为了避免类成员变量的访问冲突--只要所有可能访问类成员变量的方法均被声明为synchronized)

②是某个类的范围，即synchronized修饰静态方法时，可以防止多个线程同时访问这个方法，它可以对类的所有对象起作用。

(2)该关键字不能继承，也就是说，父类的方法synchronized f(){}在子类中并不自动是synchronized f(){}，而是变成了f(){}。

在修饰代码块时：

语法为：synchronized(object){}

当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。其中参数object要看是不是被线程共享的。比如object是一个普通的变量Object object = new Object;这个变量属于这个类实例，所以只能对相同的对象才能实现线程安全；如果static Object object = new Object()，则变量属于这个类范围，所以只要是这个类的实例就能实现线程安全。其中参数就相当于锁，看这个锁所管得范围。相当于对这个参数上锁，其他线程在执行这个方法之前发现这个变量被锁住，就不会向下执行。

synchronized关键字的缺陷：就是如果将一个大的方法声明为synchronized将会大大影响效率，比如，如果将一个线程类的run()方法声明为synchronized，由于在线程的整个声明周期内它一直在运行，所以将会导致它对该线程类的其他synchronized方法的调用永远不能成功。

5、线程的同步和互斥是一个概念，都是指在同一时间内最多只能有一个线程执行某段代码

5、当run()方法执行完毕，线程也就结束并销毁了。

6、哈希表算法

Hash Table的查询速度非常快，几乎是O(1)的时间复杂度

7、归并排序

该方法采用分治法，先将未排序的数据分成若干小组，使每个小组是有序的，然后将各小组合并成整体有序的序列

8、虚拟内存的原理是什么？怎么实现的？

9、从1000万条短信中找出重复次数最多的10条，要求所需内存不超过1G

10、数据结构中的堆

11、最大堆和最小堆

12、trie树

13、eclipse中快速查找类的快捷键ctrl+shift+t，快速查找文件的快捷键是ctrl+shift+r

14、JDBC中的ResultSet与Iterator迭代器稍有不同，对于ResultSet类，迭代器初始化时被设定在第一行之前的位置。必须调用next()方法将它移动到第一行。

15、在SQL中，二进制大对象称作BLOB，字符串大对象称作CLOB。

一个超级简单的服务程序

---------------------------------------------------------------------------------

向浏览器中输入信息，即一个超级简单的服务器程序

package com.ztesoft.lizan;

import java.io.IOException;

import java.io.OutputStream;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;

public class Server {

public static void main(String[] args) {

try {

ServerSocket ss=new ServerSocket(8992);

Socket socket;

OutputStream os;

while(true){

socket=ss.accept();

os=socket.getOutputStream();

PrintWriter out=new PrintWriter(os,true);

out.println("<h1>This is a simple server!</h1>");

Thread.currentThread().sleep(3000);

break;

}

os.close();

socket.close();

ss.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

---------------------------------------------------------------------------------

16、抽象类实现接口，不必实现里面的所有方法

17、new 接口(){}这种用法不是实例化接口，而是内部类的用法

18、解释一下url中有中文时，为什么要在jsp页面中对其进行encodeURI两次编码： request.setCharacterEncoding()只对post有效，对get无效首先要清楚request.getParameter()在取得参数时,如果没有指定编码方式，会自动按照"iso-8859-1"来解码一次，这样如果只对中文进行一次encodeURL编码，那么只是转换成utf-8的编码格式(此时已全部变成英文和数字，成为ascii字符)，这样一来， request.getParameter()按照iso-8859-1解码，已经解码成未知的结果(因为本身这里编码和解码就是不对称的)之后再用URLDecoder.decode()解码，理所当然会解出乱码。

但是如果传递URL之前进行两次编码，相当于对第一次编码后的英文和数字进行utf-8编码(对英文和数字编码，utf-8和iso-8859-1结果相同)这样，当request.getParameter()自动解码时，就会解成第一次经utf-8编码以后的结果，再经过一次URLDecoder.decode()正好将第一次编码的结果解出来

下面可以用流程示意一下：

编码一次：utf-8编码-》ISO-8859-1解码-》utf-8解码 编码解码不对称

编码两次：utf-8编码-》utf-8编码-》ISO-8859-1解码(=utf-8解码)-》utf-8解码 编码解码对称

19、Java中类型后面跟三个点

代表可以接受多个实际参数，这里的多个指的是不限个数。该用法是jdk1.5之后出现的新内容，使用在函数的形参上，相当于一个数组，调用函数时传递多少参数都可传递到这个对象上，需要注意的是，使用这个形参必须放到最后一位形参位置上，否则会报错。

语法结构：String ... name(变量名)

20、\n:newline 表示换行，只是向下移动一行，并不移动左右。

\r:return 表示回车，到当前行的最左边

21、延迟加载就是等到真正使用的时候再去创建实例，不用时不去创建

22、Java当中的对象锁用在对象的实例方法或者对象实例上。类锁用在静态方法或者类实例上。

锁的概念其实就是一个人想占用一个房间(一个线程想执行一个方法)，进入房间把门锁上不让其他人进入(不让其他线程执行该方法)，用完之后把锁释放掉。别人想用的时候，别人再把房间锁上。

23、这里说一下synchronized(param){}中param的类型问题：

如果param是一个普通的变量或者是一个普通的不相关的对象类型，那么就表示是一把锁，当一个线程执行该方法时，表示把这个变量占有，当另一个线程要执行时发现变量被占有，就不会执行该代码，直到变量被释放为止。(这里要注意，param不能是常量或者string name=""或者Long,Integer等

包装类，因为这些变量都是会放在jvm中的常量池中供所有的类共享的，可能在这里锁住param时，会影响很远的八竿子打不着的其他代码的执行)

如果param是this就表示多线程调用同一个对象的此段代码会进行互斥，即只能同时有一个线程执行同一个对象的本段代码。

如果是.class就表示即使是同一个类的不同对象，也只能同时有一个线程进行访问。

24、单例模式中为什么要设定双重检查锁，示例代码如下：

public static AccountSeqHelper getInstance()

{

if (instance == null)

{

synchronized (singleLock)

{

if (instance == null)

{

instance = new AccountSeqHelper();//\*\*

}

}

}

return instance;

}

如果把第一次检查去掉，剩下的就是一个典型的懒加载的单例模式：先进行判断，如果instance为空则创建，如果不为空则直接跳过，这样

保证只有一个变量。

加上第一层判断，是保证其他线程在执行的时候发现instance不为空，下面的同步代码直接不执行，这样就避免了其他线程不必要的等待，从而提高了

性能

25、hashCode()方法的作用一般是用在集合当中。比如，向集合中存入数据，集合要求不能有相同的元素，所以要先判断集合中是否有这个元素，判断的方法

就是先要查找有没有这个元素，一般的查找方法就是一个一个进行比较，但是这样效率会很低，所以就先用hashCode()方法查找hash值，根据hash值

可以很快地查找到元素的存储数组，这样再根据equal()方法就能具体的判断有没有这个值了。

26、为什么哈希表能够加快查找效率？

哈希表的基本原理：原本无序的集合经过哈希算法被重新调整位置，排列成新序列，也就是hash table(与其说是表，不如说是某种数据结构的数组)

以某string集合为例，如图：

原始序列 hash算法 关键字 取模（10） 重排后的数组(somestructure a[])

string1 ------------>> 24 ----->> 4 --------->> a[4]

string2 ------------>> 2940 ---->> 0 --------->> a[0]

string3 ------------>> 598 ---->> 8 --------->> a[8]

string4 ------------>> 97 ----->> 7 --------->> a[7]

此处hash算法其实包括了两部分，（1）把字符串压缩成一个整数关键字（2）对关键字取模，将2^32的整数范围压缩成10。当然由于压缩率太大，

所以发生冲突的概率是很高的，实际问题的解决中不会采用这么大的压缩率。如果有冲突，参见哈希表如何解决冲突。

27、native关键字

为什么要使用native方法？为了让Java与操作系统或者一些底层代码打交道。

native方法就是该方法的实现由非Java代码实现。在定义一个native method时，并不提供实现体，因为实现体是由非Java语言在外面实现的。

注意native不能与abstract连用，因为native暗示这些方法是有实现体的，只不过这些实现体是非Java的，但是abstract却显然的指明这些方法

无实现体。子类也可以用Java重写这个native method。

28、MD5是一种典型的哈希算法。

哈希算法就是将一段任意长度的二进制值映射成较短的固定长度的二进制值，这个小的二进制值称为哈希值。哈希值是一段数据唯一且极其紧凑的数据表示形式。

29、解释型语言和编译型语言

编译型语言就是由编译器全部编译成机器码之后再运行。

而解释性语言是解释器解释一句然后执行一句。

Java既是编译型语言又是解释型语言，它是由虚拟机先将Java代码编译成字节码然后虚拟机再对字节码进行解释。

30、只有内部类可以是私有类，而常规类只能具有包的可见性，或公有的可见性。

31、什么是线程池？

32、类加载器总结：

1、类的加载过程

JVM将类加载过程分为三个步骤：装载(load)、链接(link)和初始化(initialize)，链接又分为三个步骤：验证、准备、解析。

装载---------->>链接

|----验证

|----准备

|----解析---------->>初始化

1)装载：查找并加载类的二进制数据

2)链接：

验证：确保被加载类的正确性；-----------------------------这一步主要就是验证生成的class文件是否符合JVM字节码格式，防止加载其他非法的class文件。

准备：为类的静态变量分配内存，并将其初始化为默认值；-----这里只是先给一个默认值，初始化阶段时再给真正的值。

解析：把类中的符号引用转换为直接引用。

3)初始化：为类的静态变量赋予正确的初始值；

2、类的初始化

类什么时候才被初始化：

1)new一个对象时

2)访问某个类或接口的静态变量，或者对该静态变量赋值

3)调用类的静态方法

4)反射(Class.forName("com.lizan.Load"))

5)初始化一个类的子类(会首先初始化子类的父类)

6)JVM启动时标明的启动类，即文件名和类名相同的那个类

只有以上6步才会导致类的初始化

类的初始化步骤：

1)如果这个类还没有被加载和链接，那先进行加载和链接

2)如果这个类存在父类，并且这个类还没有被初始化(注意：在一个类加载器中类只能初始化一次)，那就初始化直接的父类(不适用于接口)

3)假如类中存在初始化语句(如static变量和static块)，那就依次执行这些初始化语句

3、类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将静态变量和静态块以及方法放在内存空间中的方法区内，将非静态变量

放在堆中，将对象的引用(Student stu=new Student,对象的引用就是stu)放在栈中，

33、SOAP协议 = HTTP协议 + XML数据格式

35、webservice一句话讲就是一种跨语言和操作系统的远程调用技术。也就是说a计算机上的程序可以调用b计算机上的对象的方法(也可以说是服务)，甚至java写的应用程序要和.Net开发的程序进行通信。

webservice平台的三大技术XML+XSD、SOAP和WSDL。

①XML+XSD

Webservice通过http传输数据，通过XML来封装数据。

XML能够封装数据但是它并没有规定采用什么标准来表示数据，例如整数的类型是什么？。所以这里采用XSD来定义一套标准的数据类型，并给出了一种语言

扩展这套数据类型。

②SOAP

Webservice通过http传输数据，发送的请求内容和结果内容通过xml进行封装，并增加了一些特定的http消息头，以说明http消息的内容格式，这些特定的

http消息头和内容格式就是SOAP协议。

③WSDL

要想调用服务首先要知道服务的地址在哪里，服务里有哪些方法，方法有哪些参数，返回值类型是什么，服务通过什么方法来调用。

WSDL就是这样一种基于xml的语言，用于描述web service及其函数参数和返回值。

WSDL保存在web服务器上，通过一个url地址就可以访问它，客户端要调用一个webservice服务之前，要知道该服务的WSDL文件的地址。webservice服务

提供商可以通过两种方式来暴露它的WSDL文件地址，一种是通过是注册到UDDI服务器，以便被人查找；二是直接告诉给调用者。

webservice开发：

服务器开发：把公司内部的方法发布成webservice服务同其他人调用。

客户端开发：调用别人发布的webservice服务，例如调用天气预报的webservice服务。

webservice工作调用原理：

对客户端而言，我们给这各类WebService客户端API传递wsdl文件的url地址，这些API就会创建出底层的代理类，我调用这些代理，

就可以访问到webservice服务。代理类把客户端的方法调用变成soap格式的请求数据再通过HTTP协议发出去，并把接收到的soap数据变成返回值返回。

对服务端而言，各类WebService框架的本质就是一个大大的Servlet，当远程调用客户端给它通过http协议发送过来soap格式的请求数据时，

它分析这个数据，就知道要调用哪个java类的哪个方法，于是去查找或创建这个对象，并调用其方法，再把方法返回的结果包装成soap格式的数据，

通过http响应消息回给客户端。

36、http协议原理

34、数据库中的序列是什么？

为表中的行自动生成序列号，产生一组等间隔的数值(类型为数字)。主要用途是生成表的主键。

35、数组是对象，是一种引用类型。

36、异常机制的一个作用就是当代码出现错误之后，交给catch处理或者直接抛出，之后继续运行下面的代码，保证代码的运行不会中断，并且对出错信息进行打印或者采取一些其他的处理措施，例如：

public static void main(String[] args) {

try {

String name=null;

System.out.println(Long.valueOf(name));

} catch (NumberFormatException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

//=====================================

String[] a={"a","b","c"};

for(String b:a){

System.out.println(b);

}

}

对于注释行上面的部分，如果不加try-catch块，程序只会运行上面的部分输出错误信息，不会输出下面的abc。但是如果加上try-catch块，程序运行到上面部分之后抛出异常信息，接着运行下面的部分

输出abc。

注意：在这里抛出异常(即throws NumberFormatException)程序依然会中断。

当然最常用的作用就是保证程序不会中断并且打印出错信息。

37、异步IO和同步IO

38、事件队列

39、MIME类型：(Multipurpose Internet Mail Extensions)多用途互联网邮件扩展类型，它是一个互联网标准，扩展了电子邮件标准，使其能够支持：非ASCII字符文本；非文本格式附件(二进制、声音、图像等)

；由多部分(multiple parts)组成的消息体；包含非ASCII字符的头信息(Header Information)

40、Base64：要求把每三个8Bit的字节转换为四个6Bit的字节(3\*8=4\*6)，然后把6Bit再添两位高位0，组成四个8Bit的字节，也就是说，转换后的字符串理论上将要比原来的长1/3

41、MD5的功能：

输入任意长度的信息，经过处理，输出为128位的信息(数字指纹)

不同的输入得到不同的结果(唯一性)

根据128的结果不可能反推出输入的信息(不可逆)

MD5的用途：

1、防止被篡改

1)比如发送一个电子文档，发送前先得到MD5的输出结果a，发送给对方之后也得到一个MD5值b，两者进行比对。

2)文件下载

3)SVN checkout判断文件是不是有改动

2、防止直接看到明文：

保存密码时，保存密码的MD5值在数据库中

3、防止抵赖(数字签名)

42、WSDL文件中的标签的说明：

WSDL文档包含7个重要的元素：types import message portType operations binding和service

definition元素：所有WSDL文档的根元素，一般包含若干个XML命名空间，可以带一个name属性。

①types元素：定义了自定义的特殊数据类型(即复杂类型和定制的简单类型)，例如：

<types>

<xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">

<xsd:simpleType name="ISBN">

<xsd:restriction base="xsd:string">

<xsd:pattern value="[0-9]{9}[0-9Xx]" />

<xsd:restriction>

<xsd:simpleType>

<xsd:schema>

<types>

②import元素：可以让当前的文档使用其他WSDL文档中的命名空间。import元素必须声明两个属性，namespace属性和location属性。例如：

<definitions name="AllMhWebServices" xmlns="http://schemas.xmlsoap.org/wsdl/">

<import namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" location="http://www.Monson-Haefel.com/jwsbook/BookPrice.wsdl"/>

<import namespace="http://www.Monson-Haefel.com/jwsbook/po" location="http://www.Monson-Haefel.com/jwsbook/wsdl/PurchaseOrder.wsdl"/>

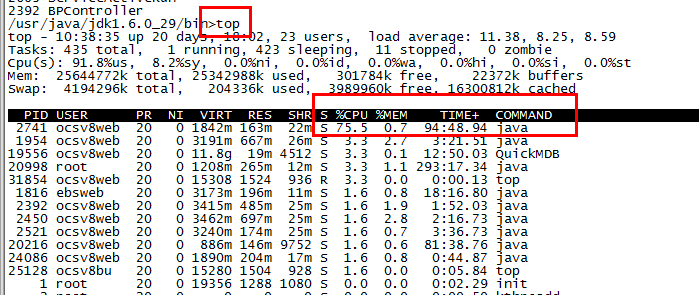
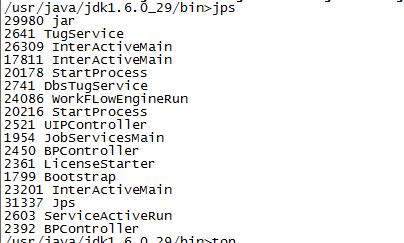
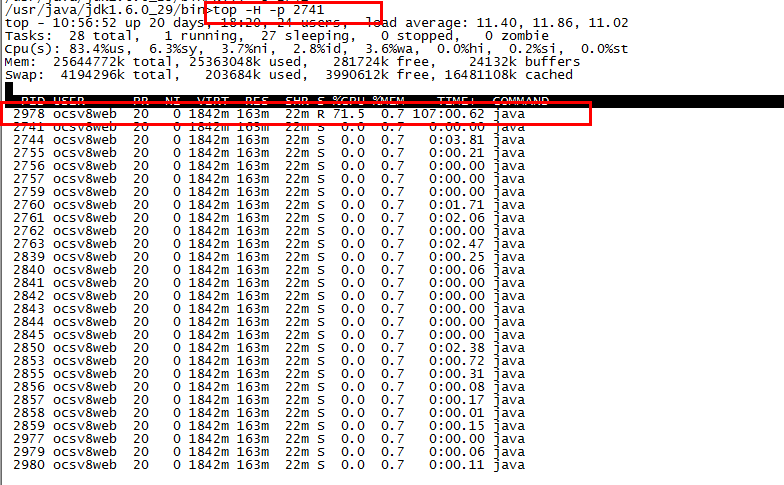
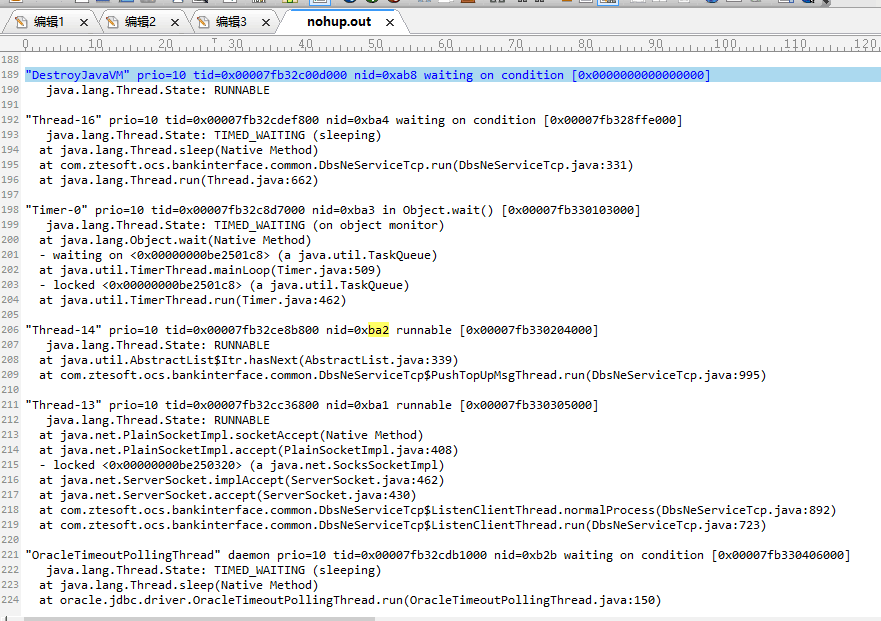
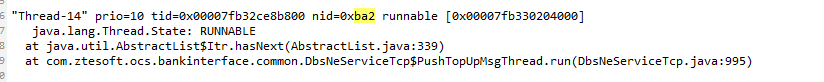
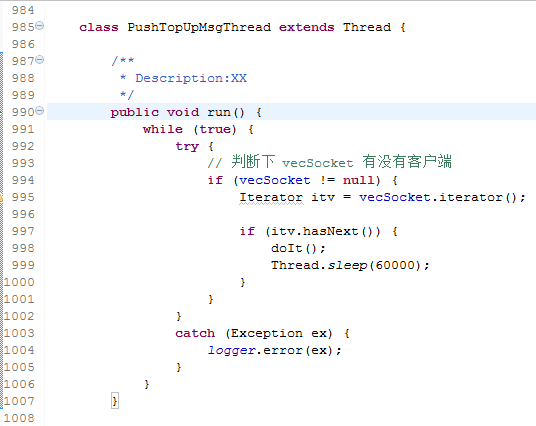
<import namespace="http://www.Monson-Haefel.com/jwsbook/Shipping" location="http://www.Monson-Haefel.com/jwsbook/wsdl/Shipping.wsdl"/>

<definitions />

message portType和operation用于描述web服务的抽象接口，相当于java中的接口。其中portType相当于类接口的名称；operation相当于类接口中包含的函数，message相当于函数的参数和返回值。

binding描述web service的通信协议。<soap:binding/>描述使用soap协议，binding还描述web service的方法和输入输出。

43、Linux下java某个进程CPU占用过高的解决思路：

刚刚测试提了一个cpu过高的问题，在这里分享一下，主要关注解决方案  
郑 2016/12/21 10:53:49  
  
  
郑 2016/12/21 10:54:28  
  
首先可以看到2741的java进程cpu很高 ， 测试反应进程启动以后就一直这样  
  
10:55:34  
郑 2016/12/21 10:55:34  
  
通过jps查看(linux的ps命令也可以)，可以看到该进程是DbsTugService  
郑 2016/12/21 10:55:35  
  
  
郑 2016/12/21 10:56:11  
  
现在可以第一步确认，是银行的接口机耗cpu过高  
郑 2016/12/21 10:56:45  
  
现在第二步，需要找出是这个进程的哪一个线程消耗的cpu资源  
郑 2016/12/21 10:57:26  
  
  
10:57:51  
郑 2016/12/21 10:57:51  
  
命令是top -H -p 2741 ，  2741是java进程号，  这个命令用来查询一个进程下的线程cpu使用情况  
郑 2016/12/21 10:58:17  
  
注意一下这个命令只是给linux用的，unix不支持这个命令，不同的操作系统需要分别处理  
郑 2016/12/21 10:58:37  
  
从上面的图可以看出2978线程耗费了cpu  
郑 2016/12/21 10:59:15  
  
下面抓取java进程的线程堆栈(使用JMAP命令，将堆生成dump文件)，看一下2978对应的是哪个线程  
郑 2016/12/21 10:59:31  
  
  
11:00:13  
郑 2016/12/21 11:00:13  
  
上图里，堆栈里的nid（本地线程ID）都是16进制，所以2978需要做一下进制转换  
郑 2016/12/21 11:00:30  
  
  
郑 2016/12/21 11:00:49  
  
对应的16进制为0xba2  
郑 2016/12/21 11:01:01  
  
线程已经定位  
郑 2016/12/21 11:01:02  
  
  
郑 2016/12/21 11:01:15  
  
下面可以分析代码  
郑 2016/12/21 11:01:47  
  
  
11:02:15  
郑 2016/12/21 11:02:15  
  
结合堆栈（   at com.ztesoft.ocs.bankinterface.common.DbsNeServiceTcp$PushTopUpMsgThread.run(DbsNeServiceTcp.java:995)）  
  
郑 2016/12/21 11:03:28  
  
可以看出在不断的迭代，  当迭代器内有值时，会开始执行任务，然后sleep，释放cpu资源  
郑 2016/12/21 11:04:00  
  
但是当itv.hasNext==false时，将没有任何的sleep动作，导致线程在while里空跑  
11:04:41  
郑 2016/12/21 11:04:41  
  
解决方法：  当itv.hasNext()==false时， 主动sleep一小段时间  (比如50ms)  
郑 2016/12/21 11:05:02  
  
说完了  
郑 2016/12/21 11:05:19  
  
这个场景修改比较简单，主要关注解决的思路  
郑 2016/12/21 11:05:30  
  
cpu过高有各种各样的原因，但是思路基本是这样

1. 流为什么不能重复读取？

这里可以将流看作一个水管，要想喝水就要让水流出，接完了，水管中就没有水了，要想接水，要重新打开。

SVN中的C表示合并并且有冲突，G表示合并并且没有冲突。

PermGen space的全称是Permanent Generation Space，是指内存的永久保存区域，这块内存主要是被JVM存放Class和Meta信息的，Class在被load时就会被放到PermGen space中，它和存放类实例的heap内存不同，GC不会在主程序运行期对PermGen space进行清理，所以如果你的程序中有很多class的话，就很可能出现PermGen space错误，如果你的web APP下用了大量的第三方jar，其大小超过了jvm默认的大小(4M)那么就会产生此错误信息。

Java的方法的变量不能是静态的：因为用static修饰的属性或方法本身就是属于类级别的，也可以说是全局性质的，在方法中用static修饰变量，相当于把局部变量整成全局的，java没有这种用法。

## 平衡二叉搜索树

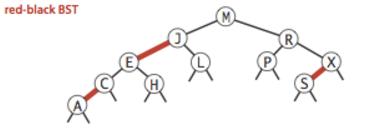
平衡二叉搜索树：它是一颗空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一个平衡二叉树。常用算法有红黑树、AVL、Treap、伸展树等。

## 红黑树

红黑树在实际工程中有着广泛的应用，比如Linux中的线程调度就是使用的红黑树来管理的进程控制块，java中的TreeMap和TreeSet也是基于红黑树来实现的。红黑树相比普通二叉查找树的一个优势就是它的树高为lgN，所以不管是插入/查找/删除操作它均能保证能够在对数时间内完成。

红黑树可以定义成含有红黑链接并且满足下列条件的二叉查找树：

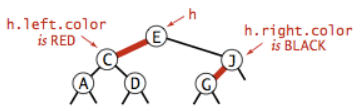
1. 红链接均为左链接(并不保证左链接都是红链接)
2. 没有任何一个节点同时和两条红链接相连
3. 任意空链接到根节点的路径上的黑链接数目相同



先定义一颗红黑树：

public class RedBlackLiteBST<Key extends Comparable<Key>, Value> {  
 private static final boolean *RED* = true;  
 private static final boolean *BLACK* = false;  
 private Node root;  
 private int n;  
 private class Node {  
 private Key key;  
 private Value value;  
 private Node left, right;  
 private boolean color;  
 public Node (Key key, Value value, boolean color) {  
 this.key = key;  
 this.value = value;  
 this.color = color;  
 }  
 }  
}

红黑树的颜色表示：

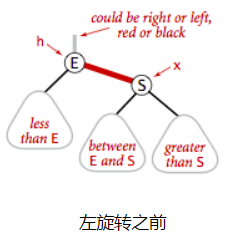


#### 红黑树的几种基本操作

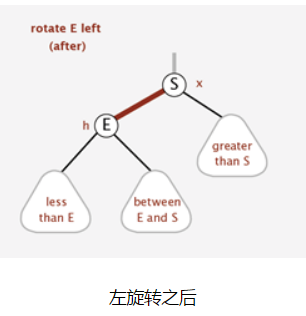
红黑树相比普通二叉查找树的一个重要优势就是插入的高效性，但是正因为如此红黑树的插入操作的算法实现相比普通二叉查找树要复杂一些。在正式实现插入操作之前，先了解下基本操作。

##### 左旋转

如下图所示，我们要将红色右链接转化为左链接：



在保持红黑树平衡的同时，将上图的结构变为如下这样：



## HashMap实现原理

数组：寻址容易，插入和删除困难

链表：寻址困难，插入和删除容易

综合两者的特性----寻址容易，插入和删除容易----哈希表

最常用的实现哈希表的方法----拉链法(链表的数组)

HashMap里面实现一个静态内部类Entry，其重要的属性有key，value，next

(最常用的：数组+链表)，查找数据的时间复杂度

HashSet实现Set接口，HashMap实现Map接口，他们底层的Hash存储机制完全一样。甚至HashSet本身就采用HashMap来实现。

系统会采用hash算法来计算元素的存储位置。

集合当中存储的是对象的引用。

#### HashMap的存储实现：

当程序执行map.put(“语文”,80)时，系统将调用语文的hashCode()方法得到其hashCode值------每个java对象都有hashCode()方法，都可通过该方法获得它的hashCode值。系统会根据该hashCode值来决定该元素的存储位置。

public V put(K key, V value) {  
 if (key == null)  
 return putForNullKey(value);  
 int hash = *hash*(key.hashCode());

//搜索指定hash值在对应table表中的索引  
 int i = *indexFor*(hash, table.length);  
 for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
 Object k;  
 if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  
 V oldValue = e.value;  
 e.value = value;  
 e.recordAccess(this);  
 return oldValue;  
 }  
 }  
  
 modCount++;  
 addEntry(hash, key, value, i);  
 return null;  
}

indexFor(int h, int length)方法的代码如下：

static int indexFor(int h, int length) {  
 return h & (length-1);  
}

这种方法非常巧妙，HashMap底层数组的长度总是2的n次方，这样一来，假设h=5,length=16, 那么 h & length - 1 将得到 5；如果 h=6,length=16, 那么 h & length - 1 将得到 6 ……如果 h=15,length=16, 那么 h & length - 1 将得到 15；但是当 h=16 时 , length=16 时，那么 h & length - 1 将得到 0 了；当 h=17 时 , length=16 时，那么 h & length - 1 将得到 1 了……这样保证计算得到的索引值总是位于 table 数组的索引之内。

所以在调用put方法时，程序首先根据该key的hashCode()返回值决定该Entry的存储位置。当两个Entry对象的key的hashCode()返回值相同时，将由key通过equals()比较值决定是采用覆盖行为还是产生Entry链。

addEntry代码如下：

void addEntry(int hash, K key, V value, int bucketIndex) {

Entry<K,V> e = table[bucketIndex];  
 table[bucketIndex] = new Entry<K,V>(hash, key, value, e);  
 if (size++ >= threshold)  
 resize(2 \* table.length);  
 }

如果索引处有了一个对象，那新添加的Entry对象指向原有Entry对象(产生一个Entry链)，如果索引处没有Entry对象，那么就没有产生Entry链。

Hash算法的性能选项：

上面程序中有两个重要变量：

size：保存了HashMap中键值对的数量

threshold：该变量包含了HashMap能容纳的键值对的极限，它的值等于HashMap的容量乘以负载因子。每扩充一次HashMap的容量扩大一倍。

当创建HashMap时，系统会自动创建一个table数组来保存HashMap中的Entry

HashMap的实际容量总是2的n次方，假如通过构造方法HashMap(int initialCapacity，float loadFactor)输入容量为10，那该方法总是指定实际容量为大于该数的最小的2的倍数，代码如下：

while (capacity < initialCapacity)  
 capacity <<= 1;

系统开始初始化HashMap时，系统会创建一个容量为capacity的Entry数组，这个数组里可以存储元素的位置称为“桶(bucket)”

HashMap存储示意图：



如果HashMap的每个bucket只有一个Entry，那性能是最好的；在发生冲突时，系统必须遍历链中的每个Entry

## HashMap与Hashtable的区别

HashMap是Hashtable的轻量级实现(非线程安全的实现)，它们都完成了Map接口。

主要区别在于：HashMap允许空键(key)值(value)，由于非线程安全，效率上可能高于Hashtable。

HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

HashMap与Hashtable采用的hash/rehash算法都大概一样，所以性能不会有很大差异。

**finally中使用return是一种很不好的编程风格，它会覆盖掉所有的其它返回，并且吃掉catch中抛出的异常。**

## 动态代理

代理的优点：可以隐藏委托类的实现、可以实现客户与委托类间的解耦，在不修改委托类代码的情况下能够做一些额外的处理。

先说静态代理：实现代码如下

先定义一个接口Sell，委托类Vendor和代理类BusinessAgent都实现这个接口

Sell接口：

public interface Sell {  
 void sell();  
 void ad();  
}

委托类Vendor：

public class Vendor implements Sell {  
 @Override  
 public void sell() {  
 System.*out*.println("In sell method");  
 }  
 @Override  
 public void ad() {  
 System.*out*.println("ad method");  
 }  
}

代理类BusinessAgent，内部有一个委托类的引用：

public class BusinessAgent implements Sell {  
 private Vendor mVendor;  
 public BusinessAgent (Vendor vendor) {  
 this.mVendor = vendor;  
 }  
 @Override  
 public void sell() {  
 mVendor.sell();  
 }  
 @Override  
 public void ad() {  
 mVendor.ad();  
 }  
}

上面就是一个简单的静态代理。

下面要增加一个需求：委托类的代码不能动(这里是模拟使用别人的已经编好的代码)，添加一个过滤功能，实现只卖货给特定的人比如大学生。通过静态代理可以在代理类BusinessAgent中添加过滤代码：

public void sell() {  
 if (isColleage == true) {  
 mVendor.sell();  
 }  
}

#### 什么是动态代理

代理类在程序运行时创建的代理方式称为动态代理。也就是说代理类并不是在java代码中定义的，而是在运行时根据我们在java代码中的“提示”动态生成的。**动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类的函数。**

现在有一个新需求：在执行委托类中的方法之前输出“before”，执行之后输出“after”。使用静态代理实现这一需求：

@Override  
public void sell() {  
 System.*out*.println("before");  
 mVendor.sell();  
 System.*out*.println("after");  
}  
@Override  
public void ad() {  
 System.*out*.println("before");  
 mVendor.ad();  
 System.*out*.println("after");  
}

当方法比较少时，这样修改还可以接受，但是当类很多时，这种修改方式就显得很繁琐很不明智了。

比较明智的方法是在委托类和代理类中间添加一个中介类。这个中介类被要求实现一个接口InvocationHandler接口，这个接口的定义如下：

public interface InvocationHandler {  
 Object invoke (Object proxy, Method method, Object[] args);  
}

当我们在调用代理类中的方法时，这个“调用”会转送到这个invoke方法中，proxy是代理类对象实例，method是被调用的代理类的方法，args为方法的参数。这样我们对代理类中所有方法的调用都会转为对invoke方法的调用，这样我们可以在invoke方法中添加统一的处理逻辑

public class DynamicProxy implements InvocationHandler {  
 private Object obj;  
 public DynamicProxy (Object obj) {  
 this.obj = obj;  
 }  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Exception {  
 System.*out*.println("before");  
 Object result = method.invoke(obj, args);  
 System.*out*.println("after");  
 return result;  
 }  
}

#### 动态生成代理类

public class Main {  
 public static void main (String[] args) {  
 //创建中介类实例  
 DynamicProxy inter = new DynamicProxy(new Vendor());  
 //加上这句将会产生一个$Proxy0.class文件，这个文件即为动态生成的代理类文件  
 System.*getProperties*().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");  
 //获取代理类实例sell  
 Sell sell = (Sell) (Proxy.*newProxyInstance*(Sell.class.getClassLoader(), new Class[]{Sell.class},inter));  
 sell.sell();  
 sell.ad();  
 }  
}

## IO中同步异步与阻塞非阻塞的区别

### 概念解释

所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。

所谓异步，当一个异步过程调用发出后，调用者不会立刻得到结果。实际处理这个调用的部件是在调用发出后，通过状态通知来通知调用者，或通过回调函数处理这个调用。以Socket为例，当一个客户端通过Connect函数发出一个请求后，调用者线程不用等待结果，可立刻继续向下运行。当连接真正建立起来以后，socket底层会发出一个消息通知该对象。

## 移位操作

<< 左移运算符，num<<1，相当于num乘以2

>> 右移运算符，num>>1，相当于num除以2

>>> 无符号右移，忽略符号位，空位都以0补齐

## 进程间通信方式

#### 管道

管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子关系。

#### 有名管道

也是半双工通信方式，但是它允许无亲缘关系间通信

#### 信号量

信号量是一个计数器，可以控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

#### 信号

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

#### 消息队列

消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

#### 共享内存

共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但过个进程都可以访问。共享内存是最快的IPC方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间同步和通信。

#### 套接字

套接字也是一种进程间通信方式，与其他方式不同的是，它可用于不同主机间的进程通信。

## 竞态条件

计算的正确性取决于多个线程的交替执行时序时，就会发生竞态条件。

最常见的静态条件为：

1. 先检测后执行。执行依赖于检测的结果，而检测的结果依赖于多个线程的执行时序，而过个线程的执行时序通常情况下是不固定不可判断的，从而导致执行结果出现各种问题。
2. 延迟初始化。

## 判断链表是否交叉并找出交点

1. 第一印象是采用hash，将两个链表的节点进行hash，然后判断出节点
2. 遍历两个链表，在遍历的过程中进行比较
3. 先遍历第一个链表到它尾部，然后将尾部的next指针指向第二个链表，这样两个链表就变成了一个链表，判断原来的两个链表是否相交就变成了判断新的链表是否有环的问题：即判断单链表是否有环。但是这种不太容易找到交点。

## NIO

Java NIO(New IO从Java 1.4开始)是一个可以替代标准Java IO API的IO API，Java NIO提供了与标准IO不同的IO工作方式。

Java NIO 由以下几个核心部分组成：

* Channels
* Buffers
* Selectors

**通道和缓冲区**：标准的IO基于字节流和字符流进行操作，而NIO是基于通道和缓冲区进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

**Non-blocking IO (非阻塞IO)**

Java NIO可以让你非阻塞的使用IO，例如，当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

**选择器**：选择器用于监听多个通道的事件(比如链接打开，数据到达)。因此，单个的线程可以监听多个数据通道。

Channel对应之前的流，Buffer不是新东西，Selector是因为NIO可以使用异步的非阻塞模式才加入的东西。

以前的流总是阻塞的，一个线程只要对它进行操作，其它操作就会被阻塞，也就相当于水管没有阀门，你伸手接水的时候，不管水到了没有，你就都只能耗在接水(流)上。

NIO的Channel的加入相当于接上了水龙头，虽然一个时刻也只能接一个水管的水，但依赖轮换策略，在水量不大的时候，各个水管流出来的水都可以得到妥善接纳，这个关键之处就是增加了一个接水工，也就是Selector，他负责协调，也就是看哪根水管有水了的话，在当前水管的水接到一定程度的时候，就切换一下：临时关上当前水龙头，试着打开另一个水龙头。

当其他人需要用水的时候，不用直接去接水，而是事前提了一个水桶给接水工，这个水桶就是Buffer。也就是其他人虽然也可能要等，但是不会在现场等，而是回家等，可以做其他事去，水接满了，接水工会通知他们。

## 线程池

大体思路就是，假如有15个线程，一个一个添加到线程池(线程池的大小假设为5)中去执行，如果超过5，就将多出的放到缓存队列中等待执行。

MyTask类：

public class MyTask implements Runnable {  
 private int taskNum;  
  
 public MyTask (int num) {  
 taskNum = num;  
 }  
  
 @SuppressWarnings("static-access")  
 public void run() {  
 // *TODO Auto-generated method stub* System.*out*.println("正在执行task" + taskNum);  
 try {  
 Thread.*currentThread*().*sleep*(4000);  
 } catch (InterruptedException e) {  
 // *TODO Auto-generated catch block* e.printStackTrace();  
 }  
 System.*out*.println("task" + taskNum + "执行完毕");  
 }  
}

Test类：

public class Test {  
 public static void main(String[] args) {  
  
 ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 200, TimeUnit.MILLISECONDS, new ArrayBlockingQueue<Runnable>(5));  
 for (int i = 0; i < 15; i++) {  
 MyTask myTask = new MyTask(i+1);  
 executor.execute(myTask);  
 System.*out*.println("线程池中的线程数目：" + executor.getPoolSize() + ", 队列中等待执行的任务数目：" + executor.getQueue().size() + "，已经执行完的任务数目：" + executor.getCompletedTaskCount());  
 executor.shutdown();  
 }  
 }  
}

ThreadPoolExecutor是线程池的最核心的类，它的构造方法中的各个参数的含义如下：

* corePoolSize：核心池的大小。在创建线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从名字可以看出，是预创建线程的意思，即在任务没有到来之前就创建corePoolSize个线程或者一个线程。当线程池中的线程数量达到corePoolSize之后，就会把到达的任务放到缓存队列当中。
* maximumPoolSize：线程池最大线程数，它表示在线程池中，最多能创建多少个线程。
* keepAliveTime：表示线程没有任务执行时最多保持多久会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程数不大于corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，即使线程数不大于corePoolSize，keepAliveTime也会超时。
* unit：参数keepAliveTime的时间单位，有7种取值，天，小时，分，秒，毫秒，微妙，纳秒



* workQueue：阻塞队列，用于存储等待执行的任务，一般使用LinkedBlockingQueue和SynchronousQueue。线程池的排队策略和BlockingQueue有关。
* threadFactory：线程工厂，主要用来创建线程
* handler：表示当拒绝任务时的策略，有四种取值：



executor()是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务

submit()是用来向线程池提交任务的，它实际上还是调用的executor()方法，只不过它通过Future来获取任务执行结果。

shutdown()和shutdownNow()是用来关闭线程池的。

#### 线程池状态



* 当创建线程池后，初始时，线程池处于running状态
* 如果调用了shutdown()方法，则线程池处于shutdown状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕
* 如果调用shutdownNow()方法，则线程池处于stop状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务
* 当线程池处于shutdown或者stop状态，并且所有线程都已销毁，任务缓存队列清空或执行结束后，线程池被设置为terminated状态

#### 任务的执行

举个例子：

假设有一个工厂，工厂里有10个工人，每个工人同时只能做一个任务。

因此当10个工人中有工人是空闲的，来了任务就分配给空闲的人做。

当10个工人都有任务时，如果还来了任务，就把任务进行排队等待；

如果新任务增长的速度远远大于工人处理任务的速度，那么就要采取措施，比如再招4个工人来

然后将任务也分配给这4个人做。

如果说14个工人做任务的速度还是不够，那么就要考虑不再接受新的任务或者抛弃之前的任务。

如果14个人当中有空闲，或者新增任务的速度比较缓慢，那么可能就考虑辞掉4个工人。

在这个例子中corePoolSize就是10，maximumPoolSize就是14

## java中的锁

自旋锁、互斥锁、读写锁

**互斥锁**：就是普通的那种锁，访问资源后加锁，访问完成后解锁，加锁后其他任何试图再次加锁的线程都会被阻塞(进入睡眠状态)，直到锁被释放。

**自旋锁**：自旋锁是比较低级的锁，它与互斥锁的区别是，新的线程发现锁已经被占用时不会阻塞在那里，而是一直会循环在那里看是否锁已经被释放，所以称为“自旋”，这种方式比较低级。

自旋锁存在两个问题：①死锁。试图递归的获得自旋锁必然会引起死锁。死锁阶段发生在第二次调用递归的时候。②过多占用CPU资源。由于申请者一直在循环等待，因此自旋锁在锁定的情况下，如果不成功，不会睡眠，导致CPU占用过高，单CPU时候自旋锁会让其他process动不了。解决办法就是，自旋锁实现会有一个参数限定最多持续尝试次数，超出后，自旋锁放弃当前时间片(time slice)，等下一次机会。

由此可见：自旋锁比较适合保持锁时间比较短的情况。保持锁时间很短，因此选择自旋而不是睡眠是很有必要的，在这一点上自旋锁的效率要远高于互斥锁。

信号量和读写信号量适用于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用。自旋锁适用

于保持时间很短的情况，它可以在任何上下文使用。

**读写锁**：是一种特殊的自旋锁。它把对共享资源的访问者分为读者和写者，读者只对共享资源进行读访问，写者只进行写操作。一个读写锁同时只能有一个写者或多个读者，不能同时既有读者又有写者。

***可重入锁***：举个例子：当一个线程执行到某个synchronized时，比如method1，而在method1中会调用另外一个

synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2.

public class Test {  
 public synchronized void method1 () {  
 method2();  
 }  
 public synchronized void method2 () {  
  
 }  
}

下面解释为什么需要可重入锁：假如某一时刻，线程A执行到了method1，此时线程获取了这个对象的锁，而由于method2也是synchronized方法，假如synchronized不具备可重入性，此时线程A需要重新申请锁，但这时线程A已经获取了这个对象的锁，这样线程就会一直等待永远不会执行method2.

CAS(compareAndSet)原理：CAS原语有3个参数，内存地址、期望值、新值。如果内存地址的值==期望值，表示该值未修改，此时可以修改成新值。否则表示修改失败，返回false，由用户决定后续操作。

#### AtomicReference类的作用是对“对象”进行原子操作。

它是通过volatile和“Unsafe提供的CAS函数”实现原子操作。

1. value是volatile类型。这保证了当某个线程修改了value的值时，其他线程看到的value的值都是最新的，即修改之后的value的值。
2. 通过CAS设置value。这保证了当线程池通过CAS函数(如compareAndSet函数)设置value时，它的操作是原子的，即线程在操作value时不会被中断。
3. public static void main (String[] args) {  
    Test t1 = new Test(101);  
    Test t2 = new Test(102);  
    AtomicReference atom = new AtomicReference(t1);  
    atom.compareAndSet(t1,t2); //如果atom的值为t1的话将其设置为t2  
    Test t3 = (Test) atom.get(); //值为t2  
   }

## TCP的三次握手和四次挥手

#### 三次握手

首先要熟悉TCP的报文格式：



SYN(synchronous)建立连接标志位，1代表请求连接

ACK(acknowledgement确认 )确认标志位，1代表确认号有效

FIN(finish结束)

RST(reset重置)

URG(urgent紧急指针)

PSH(push传送)

Sequence number(seq序列号)

Acknowledge number(ack确认号)

三次握手过程图示：



过程描述：

服务端处于监听状态，客户端发起连接

第一次握手：A向B发送一个数据包，其中的数据为SYN=1表示请求建立连接，初始化序列号isn=200(这个数是随机产生的)

第二次握手：如果B同意建立连接，那B向A发送一个数据包，其中SYN=1，初始化序列号=500(也是随机产生的)，同时确认标志ACK=1，确认号ack=200+1

第三次握手：C收到数据以后检验ACK==1，ack==201，如果正确，发送一个数据包，其中ACK=1，ack=501.B收到以后检验ACK==1，ack==501，如果正确，连接建立成功。

在这里讨论下为什么不能是两次握手？：首先第二次握手表示B告诉A，“你发送的信息我已经收到”。但是B给A的连接B收到没收到并没说，所以这时候还需要A给B发送信息，“你发送的信息我也收到了”。

不采用更多次是因为3次就足够了，避免浪费资源。或者也可以理解成将中间的一次分成2次，三次握手将其进行了合并。

## 四次挥手

任意一方都可以执行close来触发

过程图示：



由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上不再接收数据了，但是仍然可以发送数据，直到另一方也发送FIN。首先进行关闭的一方将进行主动关闭，而另一方将进行被动关闭。流程描述如下：

第一次挥手：客户端发送一个FIN，用来关闭客户端到服务器的数据传送。

第二次挥手：服务器收到一个FIN后，发送一个ACK给客户端，确认序号为收到序号+1

第三次挥手：服务器发送一个FIN，用来关闭与客户端的传送。

第四次挥手：客户端收到FIN，接着发送ACK给服务器，确认序列号为收到序列号+1，服务器进入closed状态，完成四次挥手。

上面是一个主动关闭一个被动关闭。实际上两边可能同时发起主动关闭。

现在来讨论为什么四次挥手中间的两次不能合并：关闭连接时当收到对方的FIN报文时，仅仅表示对方不在发送数据了，但是还能接收数据，乙方也未必全部数据都发送给对方了，所以乙方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，乙方ACK和FIN都会分开发送。三次握手中，ACK和SYN是同时发送的。

## sleep(),wait(),notify()方法

wait()与notify()的使用示例：

public class WaitAndNotify implements Runnable {  
 private byte[] b;  
 private int num;  
 private int count = 5;  
 public WaitAndNotify (int num, byte[] b) {  
 this.num = num;  
 this.b = b;  
 }  
  
 @Override  
 public void run() {  
 while (count -- > 0) {  
 synchronized (b) {  
 b.notify();  
 System.*out*.println("Number:" + num);  
 try {  
 b.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 public static void main (String[] args) {  
 byte[] b = {0};  
 new Thread(new WaitAndNotify(1, b)).start();  
 new Thread(new WaitAndNotify(2, b)).start();  
 }  
}

大体思路就是交替输出1,2,1,2

调用sleep()和wait()方法为什么要处理InterruptException?

举个例子，假如一个线程发出请求，但是该线程正处于阻塞状态，这时线程突然发生中断，如果不处理这个中断的话那么没有人知道有一个线程发出了请求。所以处理这个异常，可以将这个线程发出的请求进行处理。应该是起到一种通知作用。

## 理解HashMap中的加载因子loadfactor

加载因子表示哈希表中元素填满的程度；例如，加载因子越大填满的元素越多，好处是空间利用率高了，但冲突的机会加大了，从而查找的成本变高；反之加载因子越小，好处是冲突的机会减小了，但空间浪费多了。

因此必须在“冲突的机会”和“空间利用率”之间寻找平衡与折中。

## 数据库中delete与truncate的区别

## 在java多线程中对static关键字的理解

虽然static变量在类的各个实例中是共享的，是属于整个类的，但是被static修饰的变量在多线程中并不是线程安全的。也就是说在类的各个实例中共享与线程安全并没有直接的联系。下面举例：

设有一个静态变量为10000，起1000个线程对其进行加1减1操作，但是结果并不总是10000，代码如下：

public class Exercise extends Thread {  
 private static int *sum* = 10000;  
 CountDownLatch latch = null;  
 public Exercise (CountDownLatch latch) {  
 this.latch = latch;  
 }  
  
 public void run () {  
 *sum* = *sum* + 1;  
 System.*out*.println("sum + 1 = " + *sum*);  
 *sum* = *sum* - 1;  
 System.*out*.println("sum - 1 = " + *sum*);  
 latch.countDown();  
 }  
  
 public static void main (String[] arg) throws InterruptedException {  
 CountDownLatch latch = new CountDownLatch(1000);  
 int i = 1000;  
 while (i > 0) {  
 Thread thread = new Thread(new Exercise(latch));  
 thread.start();  
 //System.out.println(thread.getId());  
 //System.out.println(thread.toString());  
 i--;  
 }  
 latch.await();  
 System.*out*.println("sum = " + *sum*);  
 }  
}

得出的结果：







从结果看出用static修饰的变量并不能保证线程安全。

## java当中如何停止一个线程

jdk1.0的时候，java API提供了一些像stop()、suspend()和resume()方法来终止线程，但是由于潜在的死锁威胁，因此在后续的版本中这些方法被废弃了。之后的java API的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。当run()或者call()方法执行完成的时候线程会自动结束，如果要手动结束一个线程，你可以用volatile布尔变量来退出run()的循环或者取消任务来中断线程。

## Java回调机制

先来上一段代码：

先定义一个接口，

public interface MyInterface {  
 void sayYourName();  
}

接着，定义一个类，其中一个方法的参数用接口来声明，

public class MyClass {  
 public void sayYourName (MyInterface myInterface) {  
 System.*out*.println("start");  
 myInterface.sayYourName();  
 System.*out*.println("finished");  
 }  
}

接着，我们在MainActivity中声明这个类，并调用其中的方法，

public class MainActivity {  
 public void onCreate () {  
 MyClass myClass = new MyClass();  
 myClass.sayYourName(new MyInterface() { //这里直接可以new接口，后面直接跟实现方法.其实这是内部类的用法  
 @Override  
 public void sayYourName() {  
 System.*out*.println("CallBack");  
 }  
 });  
 }  
}

这其实就是回调函数的用法。

## 二叉树

满二叉树：二叉树中的每个节点有0或者2个子节点

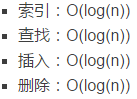
完美二叉树：二叉树中的每个节点有2个子节点，并且所有叶子节点的深度是一样的

完全二叉树：二叉树中除最后一层外其它各层的节点数均达到最大值，最后一层的节点都连续集中在最左边。

## 二叉查找树

二叉查找树(BST)是一种二叉树。其任何节点的值都大于等于左子树中的值，小于等于右子树中的值。

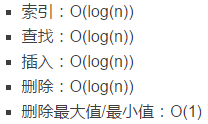
时间复杂度：

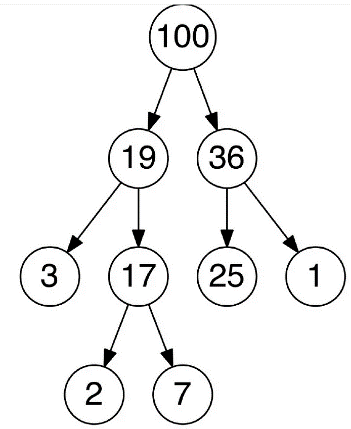


## 堆

堆是一种基于树的满足某些特性的数据结构：整个堆中的所有父子节点的键值都满足相同的排序条件。堆分为最大推和最小堆。在最大堆中，父节点的键值永远大于等于子节点的键值，根节点的键值是最大的。在最小堆中，父节点的键值永远小于等于所有子节点的键值，根节点的键值永远是最小的。

时间复杂度：





## JVM总结

### 11.JVM内存分哪几个区，每个区的作用是什么?

[**Java**](http://lib.csdn.net/base/javase)虚拟机主要分为以下一个区:

**方法区：**  
1. 有时候也成为**永久代**，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载  
2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。  
3. 该区域是被线程共享的。  
4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

**虚拟机栈:**  
1. 虚拟机栈也就是我们平常所称的**栈内存**,它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。  
2. 虚拟机栈是线程私有的，它的生命周期与线程相同。  
3. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定  
4.操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式  
5.每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

**本地方法栈**  
本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

**堆**  
java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

**程序计数器**  
内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

### 12.如和判断一个对象是否存活?(或者GC对象的判定方法)

判断一个对象是否存活有两种方法:  
1. **引用计数法**  
所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.  
引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象A引用对象B，对象B又引用者对象A，那么此时A,B对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种[**算法**](http://lib.csdn.net/base/datastructure)。

2.**可达性算法**(引用链法)  
该算法的思想是：从一个被称为**GC Roots**的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连时，则说明此对象不可用。  
在java中可以作为GC Roots的对象有以下几种:

* 虚拟机栈中引用的对象
* 方法区类静态属性引用的对象
* 方法区常量池引用的对象
* 本地方法栈JNI引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比**不一定会被回收**。当一个对象不可达GC Root时，这个对象并   
**不会立马被回收**，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记  
如果对象在可达性分析中没有与GC Root的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行finalize()方法。当对象没有覆盖finalize()方法或者已被虚拟机调用过，那么就认为是没必要的。  
如果该对象有必要执行finalize()方法，那么这个对象将会放在一个称为F-Queue的对队列中，虚拟机会触发一个Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果finalize()执行缓慢或者发生了死锁，那么就会造成F-Queue队列一直等待，造成了内存回收系统的崩溃。GC对处于F-Queue中的对象进行第二次被标记，这时，该对象将被移除”即将回收”集合，等待回收。

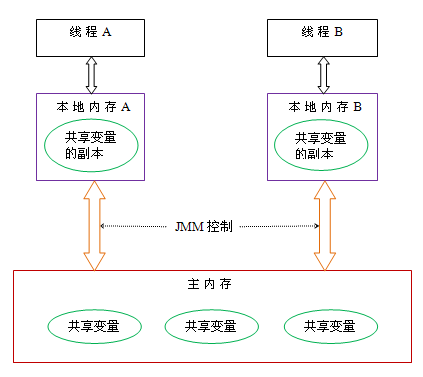
### 13.简述java垃圾回收机制?

在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

### 14.java中垃圾收集的方法有哪些?

1. **标记-清除:**  
   这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作。
2. **复制算法:**  
   为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。  
   于是将该算法进行了改进，内存区域不再是按照1：1去划分，而是将内存划分为8:1:1三部分，较大那份内存交Eden区，其余是两块较小的内存区叫Survior区。每次都会优先使用Eden区，若Eden区满，就将对象复制到第二块内存区上，然后清除Eden区，如果此时存活的对象太多，以至于Survivor不够时，会将这些对象通过分配担保机制复制到老年代中。(java堆又分为新生代和老年代)
3. **标记-整理**  
   该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。
4. **分代收集**   
   现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用**复制**算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用**标记-整理** 或者 **标记-清除**。

### 15.java内存模型

java内存模型(JMM)是线程间通信的控制机制.JMM定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java内存模型的抽象示意图如下：  
  
从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：  
1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。  
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。  
写的很好:<http://www.infoq.com/cn/articles/java-memory-model-1>

### 16.java类加载过程?

java类加载需要经历一下7个过程：  
**加载**  
加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：  
1. 通过一个类的全限定名获取该类的二进制流。  
2. 将该二进制流中的静态存储结构转化为方法去运行时[**数据结构**](http://lib.csdn.net/base/datastructure)。   
3. 在内存中生成该类的Class对象，作为该类的数据访问入口。

**验证**  
验证的目的是为了确保Class文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:  
1. 文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.  
2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。  
3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。  
4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

**准备**  
准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

public static int value=123;//在准备阶段value初始值为0 。在初始化阶段才会变为123 。

**解析**  
该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

**初始化**  
初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

### 17. 简述java类加载机制?

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的java类型。

### 18. 类加载器双亲委派模型机制？

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

### 19.什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。  
主要有一下四种类加载器:  
1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。  
2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。  
3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。  
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

### ****20.简述java内存分配与回收策率以及Minor GC和Major GC****

1. 对象优先在堆的Eden区分配。
2. 大对象直接进入老年代.
3. 长期存活的对象将直接进入老年代.  
   当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor Gc通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生Gc的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

## JIT即时编译器

#### 简介：

Java程序最初是通过解释器解释执行的，当JVM发现某个方法或代码块运行特别频繁的时候，就会认为这是“热点代码”，为了提高热点代码的执行效率，就会将这些“热点代码”编译成与本地机器相关的机器码，进行各个层次的优化。完成这个任务的编译器就是即时编译器。

#### 怎样判断是不是热点代码呢？

热点代码首先被分成两类：1、多次被调用的方法。2、多次被执行的循环体。

热点判定方式，两种：

1. 基于采样的方式探测：周期性检测各个线程的栈顶，发现某个方法经常出现在栈顶，就认为是热点方法。好处就是简单，缺点就是无法精确确认一个方法的热度。容易受线程阻塞或别的原因干扰热点探测。
2. 基于计数器的热点探测：某个方法超过阈值就认为是热点方法，触发JIT编译。(设计计数器的热度半衰减过程)。

两个计数器：方法调用计数器和回边计数器

## 关于final关键字

final修饰的变量是不可变的，**是说变量的引用也就是指向的地址不可变，但是地址上的内容是可以改变的。**

比如用final修饰一个数组 final int[] a = {1,2,3};改变里面的内容a[1] = 1;这样是完全没有问题的。

## 在代码中使用System.exit()的时候，程序不会再走finally块

## 不开辟新的变量空间，原地交换两个变量

int a = 5, b = 10;

a = a+b;

b = a-b;

a = a-b;

## Java中断机制

#### Java中断是一种协作机制

必须记住，中断只是一种协作机制。当一个线程中断另一个线程时，被中断的线程不一定要立即停止正在做的事情。相反，中断是礼貌的请求另一个线程在它愿意并且方便的时候停止它正在做的事情。有些方法，例如Thread.sleep()，很认真的对待这样的请求，但并不是每个方法一定要对中断做出响应。您可以随意忽略中断请求，但是这样做的话会影响响应。

#### 中断状态

每个Java线程都有一个与之相关联的用于表示线程的中断状态的boolean属性。Thread类用于提供三个方法来操作中断状态，这些方法包括：

**public static boolean interrupted()** 检测当前线程是否已经中断。线程的中断状态由该方法清除。

**public boolean isInterrupted()** 测试线程是否已经中断。线程的中断状态不受该方法的影响。

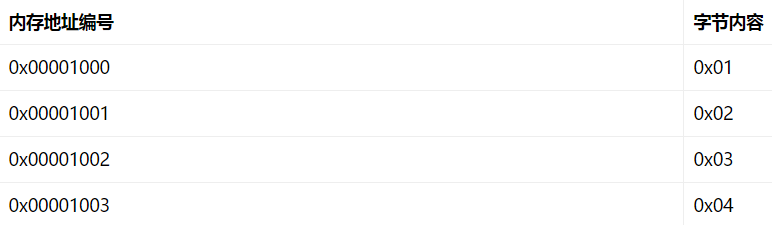
**public void interrupt()**  中断线程

三个方法中，interrupt()方法是唯一一个能将中断状态设置为true的方法，另外两个方法都是用于检测当前中断状态的方法。

## 接口不能定义为protected和private，只能是public和default

## Java中采用的是大端序

高字节数据存放在低地址处，低字节数据存放在高地址处。例如，一个int类型数据占据4个字节，int value = 0x01020304。占据情况如下：



## 多次调用同一个线程的start()方法会抛出IllegalThreadStateException