# 1.maven：项目管理工具，统一开发规范与工具，统一管理jar包

# 2.序列化反序列化：序列化（将数据结构或对象转换成二进制串的过程），反序列化（将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程），java实现Serializable接口。

# 3.servlet：servlet是一种运行服务器端的java应用程序，就是继承HttpServlet的类，实例化创建Servlet，初始化init()，调用service方法，销毁。每个 Servlet 类在容器中只存在一个实例，每一个用户请求都会产生一个新的线程，所以是非线程安全的。

filter：filter是一个可以复用的代码片段，可以用来转换HTTP请求、响应和头信息。Filter不像Servlet，它不能产生一个请求或者响应，它只是修改对某一资源的请求，或者修改从某一的响应。

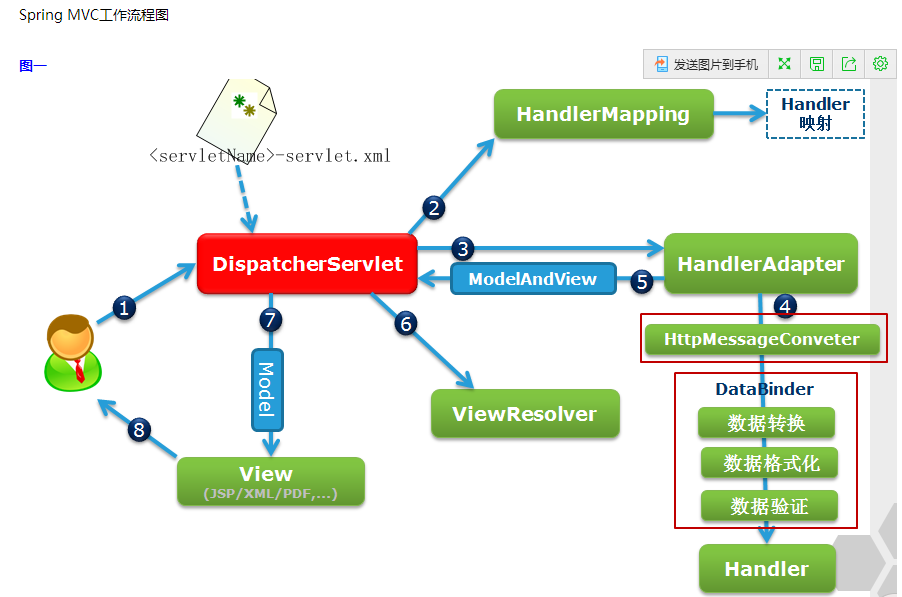
# 4.sprirngMVC与struts2区别：

Spring MVC主要由DispatcherServlet、处理器映射、处理器(控制器)、视图解析器、视图组成。两个核心

**处理器映射**：选择使用哪个控制器来处理请求

**视图解析器**：选择结果应该如何渲染

Struts2是类级别的拦截， 一个类对应一个request上下文，SpringMVC是方法级别的拦截，一个方法对应一个request上下文；SpringMVC的入口是servlet，而Struts2是filter；SpringMVC集成了Ajax，使用非常方便，只需一个注解@ResponseBody就可以实现；SpringMVC开发效率和性能高于Struts2。SpringMVC的入口是servlet，而Struts2是filter。



# 6.http

HTTPS 是由 HTTP 协议+SSL 协议构成。SSL（数字证书是支持这种安全通信的文件） 协议通过对信息进行加密，为网络通信提供安全保障。它运用了非对称密钥机制，这种机制是将公钥自由对外分发，而私钥只有信息接收者才有。SSL介于应用层和TCP层之间。应用层数据不再直接传递给传输层，而是传递给SSL层，SSL层对从应用层收到的数据进行加密，并增加自己的SSL头。

握手协议：客户机和服务器用SSL连接通信时使用的第一个子协议，允许服务器和客户机相互验证，握手协议是在应用程序的数据传输之前使用的。

记录协议：记录协议在客户机和服务器握手成功后使用，记录协议向SSL连接提供两个服务：保密性，完整性。

警报协议：客户机和服务器发现错误时，向对方发送一个警报消息。如果是致命错误，则算法立即关闭SSL连接，双方还会先删除相关的会话号，秘密和密钥。每个警报消息共2个字节，第1个字节表示错误类型，如果是警报，则值为1，如果是致命错误，则值为2；第2个字节制定实际错误类型。

对称密码算法：是指加密和解密使用相同的密钥，典型的有DES、RC5、IDEA（分组加密），RC4（序列加密）；

非对称密码算法：又称为公钥加密算法，是指加密和解密使用不同的密钥（公开的公钥用于加密，私有的私钥用于解密）

散列算法：散列变换是指把文件内容通过某种公开的算法，变成固定长度的值（散列值），这个过程可以使用密钥也可以不使用。这种散列变换是不可逆的，也就是说不能从散列值变成原文。因此，散列变换通常用于验证原文是否被篡改。典型的算法有：MD5，SHA，Base64，CRC等。

1.建立TCP连接；2.Web浏览器向Web服务器发送请求命令；3.Web浏览器发送请求头信息；4.Web服务器应答；5.Web服务器发送应答头信息；6.Web服务器向浏览器发送数据；7.Web服务器关闭TCP连接

HTTP请求信息由3部分组成：协议/版本；请求头（浏览器语言，主机，编码，链接时间）；请求正文。

# 7.RestFul :URL定位资源，用HTTP动词（GET（select）,POST（create）,DELETE（deletet）,PUT(update)）描述操作，json数据操作。

# 8.equals 与 ==

# ==比较的是两个对象的地址，equals比较的是两个对象的内容。

# 9.Object的共有方法：clone，finalize,tostring，hashcode，equals,wait,notify

# 10.抽象类，接口

1.抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract 方法；

2.抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的；

3.接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法

4.一个类只能继承一个抽象类，而一个类却可以实现多个接口

# 11java类加载过程：

# 加载：指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象

# 连接：

# 验证：这是类装载中最复杂的过程，并且花费的时间也是最长的。任务是确保导入类型的准确性，验证阶段做的检查，运行时不需要再做，虽然减慢加了载速度，但是避免了多次检查。

# 准备：分配一个结构用来存储类信息，这个结构中包含了类中定义的成员变量，方法和接口的信息。

# 解析：可选阶段，把这个类的常量池中的所有的符号引用改变成直接引用。如果不执行，符号解析要等到字节码指令使用这个引用时才会进行。

# 初始化：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。

# 12.hashmap原理区别：是数组+链表的数据结构。 hash值+数组长度取模得到index，存在数组中。HashMap里面实现一个静态内部类Entry，其重要的属性有 key , value, next

# 13.静态代理也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了，动态代理是在程序运行期间由JVM根据反射等机制动态的生成，所以不存在代理类的字节码文件。代理类和委托类的关系是在程序运行时确定。

补充：

当一个对象（客户端）不能或者不想直接引用另一个对象（目标对象），这时可以应用代理模式在这两者之间构建一个桥梁--代理对象。按照代理对象的创建时期不同，可以分为两种：

静态代理：程序员事先写好代理对象类，在程序发布前就已经存在了；

动态代理：应用程序发布后，通过动态创建代理对象。

其中动态代理又可分为：

1.JDK动态代理

此时代理对象和目标对象实现了相同的接口，目标对象作为代理对象的一个属性，具体接口实现中，可以在调用目标对象相应方法前后加上其他业务处理逻辑。

代理模式在实际使用时需要指定具体的目标对象，如果为每个类都添加一个代理类的话，会导致类很多，同时如果不知道具体类的话，怎样实现代理模式呢？这就引出动态代理。

JDK动态代理只能针对**实现了接口的类**生成代理。

# 2.CGLIB代理

CGLIB（CODE GENERLIZE LIBRARY）代理是针对类实现代理，主要是对指定的类生成一个**子类**，覆盖其中的所有方法，所以该类或方法不能声明称final的。

如果目标对象没有实现接口，则默认会采用CGLIB代理；

如果目标对象实现了接口，可以强制使用CGLIB实现代理（添加CGLIB库，并在**[spring](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "_blank)**配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）。

# 14.SpringIOC，AOP原理

IOC ：Spring来控制对象的生命周期和对象之间的关系，（1）接口注入（2）Construct注入（3）Setter注入。

AOP：一是采用动态代理技术（InvocationHandler接口时使用jdk动态代理，否则使用cglib代理），利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

# 17.JVM结构：

1.类加载器（ClassLoader）

2.执行引擎 ：通过类装载器装载的，被分配到JVM的运行时数据区的字节码会被执行引擎执行。执行引擎以指令为单位读取 Java 字节码。它就像一个 CPU 一样，一条一条地执行机器指令。每个字节码指令都由一个1字节的操作码和附加的操作数组成。执行引擎取得一个操作码，然后根据操作数来执行任务，完成后就继续执行下一条操作码。    不过 Java 字节码是用一种人类可以读懂的语言编写的，而不是用机器可以直接执行的语言。因此，执行引擎必须把字节码转换成可以直接被 JVM 执行的语言

3.内存区：私有的（程序计数器（正在执行的虚拟机字节码文件地址），虚拟机栈（方法执行内存区，每个方法执行会创建栈帧），本地方法栈（执行native方法））；共有的（java堆（对象分配内存区域），方法区（存放类信息，常量，静态变量，编译后的代码。其中还有个常量池存各种字面量和符号引用））

# 18.异常怎么处理

# 19.mysql索引（联合索引什么时候生效：使用联合索引的全部索引键，可触发索引的使用）

# 21.IO 及 NIO（有缓冲）

字节流（InputStream,OutputStream）,字符流（Reader,Writer）常见方法：InputStream.read/close,OutputStream.write/close;

FileInputStream：字节文件输入流，从文件系统中的某个文件中获得输入字节，用于读取诸如图像数据之类的原始字节流,InputStream inputStream = new FileInputStream(new File("f://hello//test.txt"));

FileOutputStream：字节文件输出流是用于将数据写入到File，从程序中写入到其他位置。

OutputStream outputStream = new FileOutputStream(new File("test.txt"));

outputStream.write("ABCD".getBytes());

BufferedInputStream：字节缓冲输入流，提高了读取效率。

InputStream in = new FileInputStream("test.txt");

BufferedInputStream bis = new BufferedInputStream(in);

BufferedOutputStream：字节缓冲输出流，提高了写出效率。

InputStreamReader：字节流转字符流，它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

BufferedReader：字符缓冲流，从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。

file.createNewFile() , file.mkdir()：创建文件夹,只能创建一级

# 22 StringBuffer 线程安全怎么实现（synchronized）

# 23.事务应该具有4个属性：原子性（事务是一个不可再分割的工作单位，事务中的操作要么都发生，要么都不发生）、一致性（事务开始之前和事务结束以后，数据库的完整性约束没有被破坏）、隔离性（多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果）、持久性。这四个属性通常称为ACID特性。

读取未提交数据(会出现脏读, 不可重复读) 基本不使用

读取已提交数据(会出现不可重复读和幻读)

可重复读(会出现幻读)

串行化

**脏读** : 一个事务读取到另一事务未提交的更新数据

**不可重复读** : 在同一事务中, 多次读取同一数据返回的结果有所不同

**幻读** : 一个事务读到另一个事务已提交的insert数据

# 24.Concurrent阻塞队列，添加方法：offer，put；获取方法：poll，take，drainTo；ArrayBlockingQueue

# 25.ClassLoader使用的是双亲委托模型来搜索类的，每个ClassLoader实例都有一个父类加载器的引用（不是继承的关系，是一个包含的关系），虚拟机内置的类加载器（Bootstrap ClassLoader）本身没有父类加载器，但可以用作其它ClassLoader实例的的父类加载器，当一个ClassLoader实例需要加载某个类时，先把这个任务委托给它的父类加载器，这个过程是由上至下依次检查的，首先由最顶层的类加载器Bootstrap ClassLoader试图加载，如果没加载到，则把任务转交给Extension ClassLoader试图加载，如果也没加载到，则转交给App ClassLoader 进行加载，如果它也没有加载得到的话，则返回给委托的发起者，由它到指定的文件系统或网络等URL中加载该类。如果它们都没有加载到这个类时，则抛出ClassNotFoundException异常。否则将这个找到的类生成一个类的定义，并将它加载到内存当中，最后返回这个类在内存中的Class实例对象。

# 26. 异常

1.java.lang.nullpointerexception 解释是"程序遇上了空指针",就是调用了未经初始化的对象或者是不存在。

1. java.lang.classnotfoundexception 解释是"指定的类不存在"，这里主要考虑一下类的名称和路径是否正确即可。
2. java.lang.arrayindexoutofboundsexception 解释是"数组下标越界"，现在程序中大多都有对数组的操作，因此在调用数组的时候一定要认真检查，看自己调用的下标是不是超出了数组的范围。
3. FileNotFoundException 解释是“文件未找到异常”。
4. IOException 解释是”输入输出流异常“。
5. NoSuchMethodException 解释是"方法未找到异常"。

# 27.finally什么时候执行：return语句已经执行了再去执行finally语句，不过并没有直接返回，而是等finally语句执行完了再返回结果（若不调用system.exit(0)）

# 28:网络安全协议SSL,介于应用层和TCP层之间,提供两个基本的安全服务：鉴别与保密。

# 29.cookie和session

cookie是存在于客户端（浏览器）,不安全，大小限制，单个4k，一个站点最多20个

session是存在于服务器端，多了影响服务器性能，session是通过cookie来工作的（sessionid）

# 30. GC是后台的守护进程。它的特别之处是它是一个低优先级进程

方法一：引用计数法。简单但速度很慢。缺陷是：不能处理循环引用的情况。

方法二：停止-复制(stop and copy)。效率低，需要的空间大，优点，不会产生碎片。

方法三：标记 - 清除算法 (mark and sweep)。速度较快，占用空间少，标记清除后会产生大量的碎片。

JVM 分别对新生代（停止-复制）和旧生代（标记 - 清除）采用不同的垃圾回收机制.

# 31.SQL中的数据锁

排它锁(其他的事务不能对它读取和修改),这种锁在Insert, Update, Delete命令执行的时候放置的，每一条记录同一时间只能有一个排他锁。

共享锁(可以被其他事务读取，但不能修改),这个锁是数据在被select的时候放置的。

Oracle数据库锁可以分为以下几大类：

DML锁（data locks，数据锁），用于保护数据的完整性；包含：TM锁称为表级锁，TX锁称为事务锁或行级锁

DDL锁（dictionary locks，字典锁），用于保护数据库对象的结构，如表、索引等的结构定义；

当Oracle执行DML语句时，系统自动在所要操作的表上申请TM类型的锁。当TM锁获得后，系统再自动申请TX类型的锁

select \* from account where name=”Erica” for update

lock table tbl\_t1 in row share mode nowait;

Hibernate 通过使用数据库的 for update 子句实现了悲观锁机制，LockMode.NONE/Write/Read

应用层实现：Criteria.setLockMode， Query.setLockMode ，Session.lock

悲观锁在整个数据处理过程中，将数据处于锁定状态。

乐观锁，大多是基于数据版本   Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来 实现。 读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提 交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据 版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据. hibernate的配置xml中在id后加上<version column="version" name="version" />

# 32.线程池是如何实现的

ThreadPool，ThreadPoolExecutor（ExecutorService的默认实现）。方法newFixedThreadPool等

ExecutorService（线程池实现类） cachedThreadPool = Executors.newCachedThreadPool();

一个线程池包括以下四个基本组成部分：

1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；

2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；

4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

线程锁：在java.util.concurrent.locks包下，主要有三个接口Condition、Lock、ReadEWriteLock。

1、**同步锁**通过synchronized关键字来进行同步

2、**独占锁**（可重入的互斥锁）可以被单个线程多次获取，它分为**“公平锁”和“非公平锁”**。Java中通过ReentrantLock实现独占锁，默认为非公平锁。

3、**公平锁** 通过CLH等待线程按照先来先得的规则，线程依次排队，公平的获取锁　Lock lock = new ReetrantLock(true)。

4、**非公平锁**　ReetrantLock默认为非公平锁，或——Lock lock = new ReetrantLock(false)。

**5、共享锁**能被多个线程同时获取、共享的锁，比如读锁 ReentrantReadWriteLock，ReadLock

//先得到lock

lock.lock();//然后获取锁

try {

//各种控制操作

}catch(Exception e){

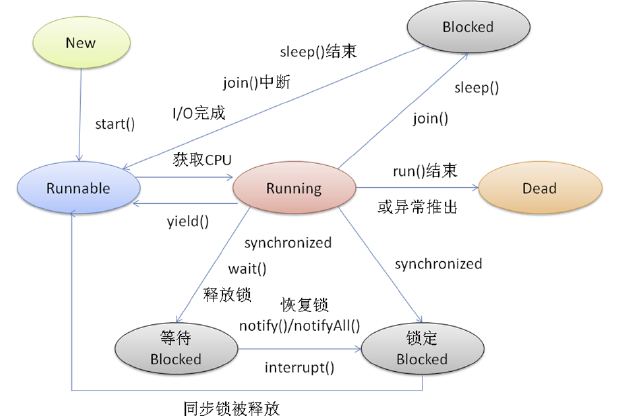
}finally {

lock.unlock();//解锁

}

LockSupport中的静态方法park() 和 unpark() 的作用分别是阻塞线程和解除阻塞线程，而不会导致死锁

Condition condition = lock.newCondition();可以通过await(),signal()来休眠、唤醒线程



基本线程类指的是Thread类，Runnable接口，Callable接口（future模式：并发模式的一种，可以有两种形式，即无阻塞和阻塞，分别是isDone和get。其中Future对象用来存放该线程的返回值以及状态）。

Thread.yield()：当前线程可转让cpu控制权，让别的就绪状态线程运行（切换）

Thread.sleep()：暂停一段时间

join()：在一个线程中调用other.join(),将等待other执行完后才继续本线程。

interrupte()

1.ThreadLocal类

2.原子类（AtomicInteger、AtomicBoolean……）

3.Lock类

lock更灵活，性能更高

可以自由定义多把锁的枷锁解锁顺序（synchronized要按照先加的后解顺序）

提供多种加锁方案，lock 阻塞式, trylock 无阻塞式, lockInterruptily 可打断式， 还有trylock的带超时时间版本。

本质上和监视器锁（即synchronized是一样的），和Condition类的结合。

**ReentrantLock**：可重入的意义在于持有锁的线程可以继续持有，并且要释放对等的次数后才真正释放该锁

4.容器类（BlockingQueue，ConcurrentHashMap）

BlockingQueue（线程安全的）：

①出现阻塞的场景：

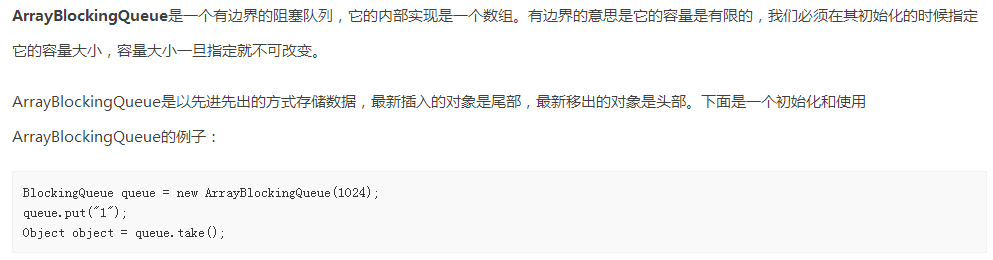
1. 当队列满了的时候进行入队列操作

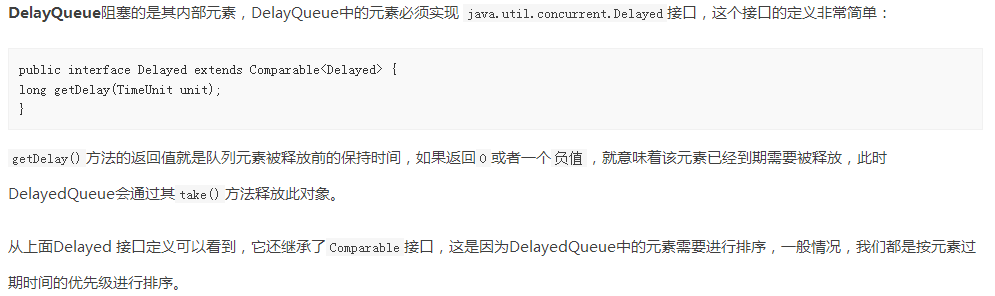
2. 当队列空了的时候进行出队列操作

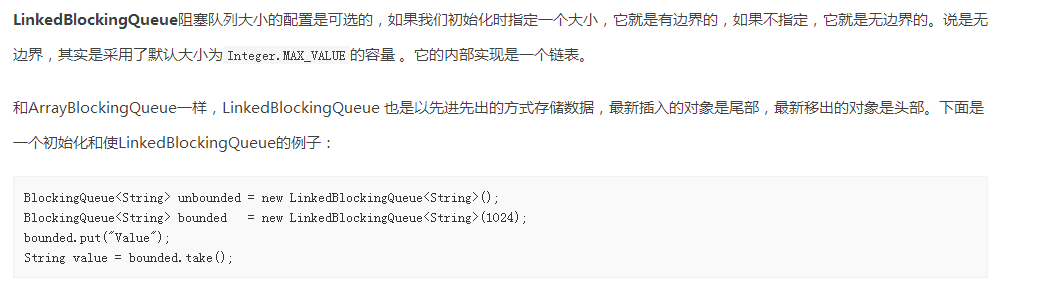
②接口中的方法：

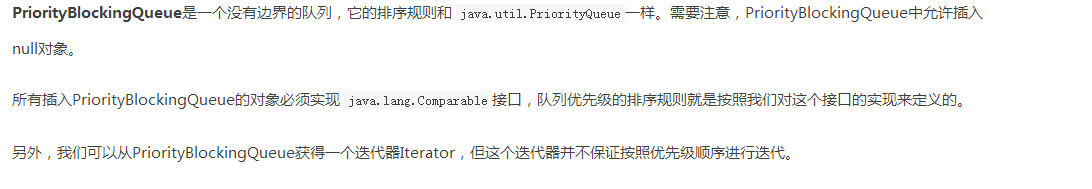














ConcurrentHashMap和hashTable区别：

HashTable性能差主要是由于所有操作需要竞争同一把锁，而如果容器中有多把锁，每一把锁锁一段数据，这样在多线程访问时不同段的数据时，就不会存在锁竞争了，这样便可以有效地提高并发效率。这就是ConcurrentHashMap所采用的"**分段锁**"思想。主干是个Segment数组，Segment继承了ReentrantLock，一个Segment就是一个子哈希表，Segment里维护了一个HashEntry数组。

# 33.常见的设计模式

单例模式：

class Singleton {

 private static Singleton instance = new Singleton();// 在内部产生本类的实例化对象

  public static Singleton getInstance() { // 通过静态方法返回instance对象

  return instance;

}

   private Singleton() { // 将构造方法封装为私有化

}

}

# 34.web优化：

1、压缩源码和图片

2、选择合适的图片格式

3、合并静态资源

4、开启服务器端的Gzip压缩

5、使用CDN

6、延长静态资源缓存时间

7、把CSS放在页面头部，把JavaScript放在页面底部

# 35.高并发网站解决：

1、HTML静态化

2、图片服务器分离

3、数据库集群、库表散列

4、缓存

5、镜像

6、负载均衡

7、最新：CDN加速技术

# 36.索引：平衡树（非二叉）也就是b tree或者 b+ tree，一个没加主键的表，它的数据无序的放置在磁盘存储器上，一行一行的排列的很整齐， 跟我认知中的「表」很接近。如果给表上了主键，那么表在磁盘上的存储结构就由整齐排列的结构转变成了树状结构，也就是上面说的「平衡树」结构，换句话说，就是整个表就变成了一个索引。没错， 再说一遍， 整个表变成了一个索引，也就是所谓的「聚集索引」；因为主键的作用就是把「表」的数据格式转换成「索引（平衡树）」的格式放置。索引能让数据库查询数据的速度上升， 而使写入数据的速度下降，原因很简单的， 因为平衡树这个结构必须一直维持在一个正确的状态， 增删改数据都会改变平衡树各节点中的索引数据内容，破坏树结构， 因此，在每次数据改变时， DBMS必须去重新梳理树（索引）的结构以确保它的正确，这会带来不小的性能开销，也就是为什么索引会给查询以外的操作带来副作用的原因。每次给字段建一个新索引， 字段中的数据就会被复制一份出来， 用于生成索引。 因此， 给表添加索引，会增加表的体积， 占用磁盘存储空间。非聚集索引和聚集索引的区别在于， 通过聚集索引可以查到需要查找的数据， 而通过非聚集索引可以查到记录对应的主键值 ， 再使用主键的值通过聚集索引查找到需要的数据。

# 37.SQL优化

1.选择最有效率的表名顺序，按照从右到左的顺序处理FROM子句中的表名，选择记录条数最少的表作为基础表。

2.WHERE子句中的连接顺序，采用自下而上的顺序解析WHERE子句，过滤掉最大数量记录的条件必须写在WHERE子句的末尾。

3.SELECT子句中避免使用\*，会将'\*' 依次转换成所有的列名, 这个工作是通过查询数据字典完成的, 这意味着将耗费更多的时间.

4.减少访问数据库的次数，可使用使用存储过程。

5.使用DECODE函数来减少处理时间，可以避免重复扫描相同记录或重复连接相同的表。

6. 删除重复记录（使用了ROWID高效）

DELETE FROM EMP E WHERE E.ROWID > (SELECT MIN(X.ROWID)

FROM EMP X WHERE X.EMP\_NO = E.EMP\_NO)。

7. 用TRUNCATE替代DELETE，TRUNCATE只在删除全表适用,TRUNCATE是DDL不是DML，删除无法恢复。

8.尽量多使用COMMIT。

9.用Where子句替换HAVING子句，HAVING 只会在检索出所有记录之后才对结果集进行过滤。

10.通过内部函数提高SQL效率。

11. 使用表的别名(Alias)，连接多个表时, 请使用表的别名并把别名前缀于每个Column上。

12.用EXISTS替代IN、用NOT EXISTS替代NOT IN。

13.使用索引。

14. 用EXISTS替换DISTINCT。

38.MyBatis缓存分为一级缓存和二级缓存，MyBatis的一级缓存指的是在一个Session域内,session为关闭的时候执行的查询会根据SQL为key被缓存(跟mysql缓存一样,修改任何参数的值都会导致缓存失效)。开启二级缓存，在配置文件中添加如下配置<cache/>。

一级缓存是sqlSession级别的缓存,在操作数据库时需要构造sqlSession对象，在对象中有一个数据结构(HashMap),用于存储缓存数据。

二级缓存是mapper级别的缓存，多个sqlSession去操作同一个Mapper的sql语句，多个SqlSession可以公用二级缓存，二级缓存是跨sqlSession的。

# Maven 命令

Mvn –v

编译项目  
mvn compile

打包发布  
mvn package

多线程，线程安全怎么实现的

**值传递！Java中只有按值传递，没有按引用传递！**Java 编程语言只有值传递参数。当一个对象实例作为一个参数被传递到方法中时，参数的值就是该对象的引用一个副本。指向同一个对象,对象的内容可以在被调用的方法中改变（基本数据类型不会在调用方法中改变），但对象的引用(不是引用的副本)是永远不会改变的。

39.HashMap

默认容量为16的Entry数组，默认加载因子为0.75

扩容：使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里。put过程是先计算hash然后通过hash与table.length取摸计算index值，然后将key放到table[index]位置，当table[index]已存在其它元素时，会在table[index]位置形成一个链表，将新添加的元素放在table[index]，原来的元素通过Entry的next进行链接，这样以链表形式解决hash冲突问题，当元素数量达到临界值(capactiy\*factor)时，则进行扩容，是table数组长度变为table.length\*2

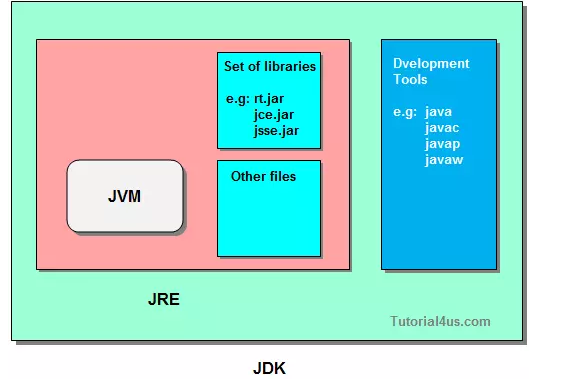
Hashtable：初始容量为11，扩容时是容量翻倍+1即:capacity\*2+1。

HashSet：底层实现和hashmap一样，不是key value结构，仅仅是存储不重复的元素，HashSet内部就是使用HashMap实现，只不过HashSet里面的HashMap所有的value都是同一个Object而已，因此HashSet也是非线程安全的

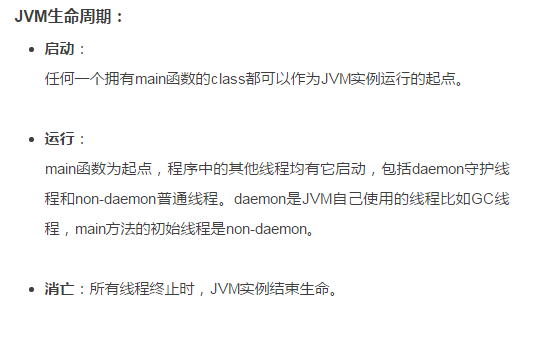
 ArrayList是实现List接口的动态数组，初始容量10，扩容(原始容量x3)/2 + 1

# 额外：jvm：

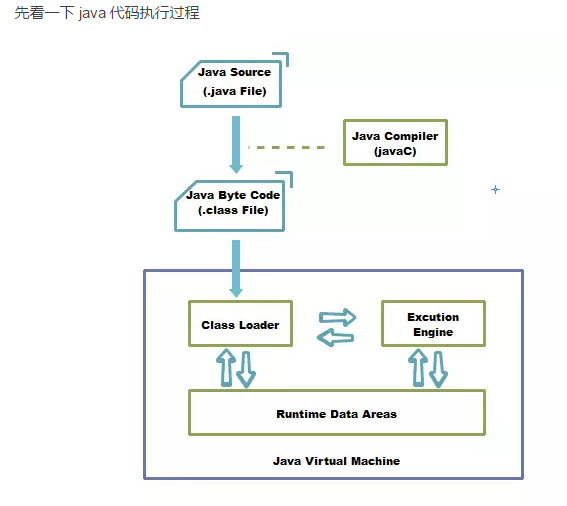
1. jdk，jre，jvm（一个虚拟的概念，不是物理存在）直接的关系和组织结构



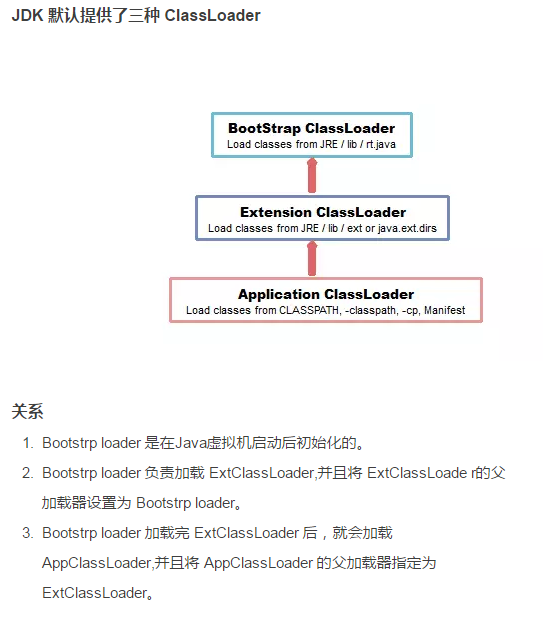
# jvm的生命周期



1. jvm的整体架构



1）.classs loader



**双亲委托模型**

Java中ClassLoader的加载采用了双亲委托机制，采用双亲委托机制加载类的时候采用如下的几个步骤：

1. 当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

2. 当前classLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrp ClassLoader。

3. 当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

**为什么使用双亲委托模型——ClassLoader 隔离问题**

每个类装载器都有一个自己的命名空间用来保存已装载的类。当一个类装载器装载一个类时，它会通过保存在命名空间里的类全局限定名(Fully Qualified Class Name)进行搜索来检测这个类是否已经被加载了。

大家觉得一个运行程序中有没有可能同时存在两个包名和类名完全一致的类？

JVM 及 Dalvik 对类唯一的识别是 ClassLoader id + PackageName + ClassName，所以一个运行程序中是有可能存在两个包名和类名完全一致的类的。并且如果这两个”类”不是由一个 ClassLoader 加载，是无法将一个类的示例强转为另外一个类的，这就是 ClassLoader 隔离。

双亲委托是 ClassLoader 问题的一种解决方案，也是 Android 差价化开发和热修复的基础。

Android 插件化 动态升级  
Android 热补丁动态修复框架小结

++++++++++++++++++++++++++++++++特别章节++++++++++++++++++++++++++++++

1. 高并发及其解决办法

尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与数据库的交互，提高性能。

用jprofiler等工具找出性能瓶颈，减少额外的开销。

优化数据库查询语句，减少直接使用hibernate等工具的直接生成语句（仅耗时较长的查询做优化）。

优化数据库结构，多做索引，提高查询效率。

统计的功能尽量做缓存，或按每天一统计或定时统计相关报表，避免需要时进行统计的功能。

能使用静态页面的地方尽量使用，减少容器的解析（尽量将动态内容生成静态html来显示）。

解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

基本上以上述问题解决后，达到系统最优。

1. redis缓存数据类型和过期策略

**1、设置过期时间**

* expire key time(以秒为单位)--这是最常用的方式
* setex(String key, int seconds, String value)--字符串独有的方式
* **注意**：
* 除了字符串自己独有设置过期时间的方法外，其他方法都需要依靠expire方法来设置时间
* 如果没有设置时间，那缓存就是永不过期
* 如果设置了过期时间，之后又想让缓存永不过期，使用persist key

**2、三种过期策略**

* 定时删除
  + 含义：在设置key的过期时间的同时，为该key创建一个定时器，让定时器在key的过期时间来临时，对key进行删除
  + 优点：保证内存被尽快释放
  + 缺点：
    - 若过期key很多，删除这些key会占用很多的CPU时间，在CPU时间紧张的情况下，CPU不能把所有的时间用来做要紧的事儿，还需要去花时间删除这些key
    - 定时器的创建耗时，若为每一个设置过期时间的key创建一个定时器（将会有大量的定时器产生），性能影响严重
    - 没人用
* 惰性删除
  + 含义：key过期的时候不删除，每次从数据库获取key的时候去检查是否过期，若过期，则删除，返回null。
  + 优点：删除操作只发生在从数据库取出key的时候发生，而且只删除当前key，所以对CPU时间的占用是比较少的，而且此时的删除是已经到了非做不可的地步（如果此时还不删除的话，我们就会获取到了已经过期的key了）
  + 缺点：若大量的key在超出超时时间后，很久一段时间内，都没有被获取过，那么可能发生内存泄露（无用的垃圾占用了大量的内存）
* 定期删除
  + 含义：每隔一段时间执行一次删除过期key操作
  + 优点：
    - 通过限制删除操作的时长和频率，来减少删除操作对CPU时间的占用--处理"定时删除"的缺点
    - 定期删除过期key--处理"惰性删除"的缺点
  + 缺点
    - 在内存友好方面，不如"定时删除"
    - 在CPU时间友好方面，不如"惰性删除"
  + 难点
    - 合理设置删除操作的执行时长（每次删除执行多长时间）和执行频率（每隔多长时间做一次删除）（这个要根据服务器运行情况来定了）

注意：

* memcached只是用了惰性删除，而redis同时使用了惰性删除与定期删除，这也是二者的一个不同点（可以看做是redis优于memcached的一点）

1. 分布式系统，及其分布式系统的一致性问题

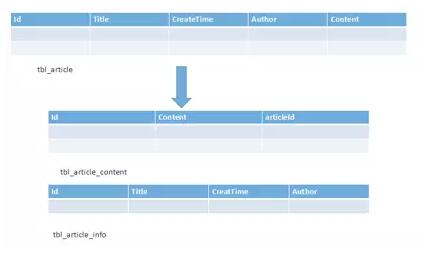
解决事一致性问题可以使用Paxos算法实现

1. 分库分表

详情：http://www.111cn.net/database/mysql/103264.htm

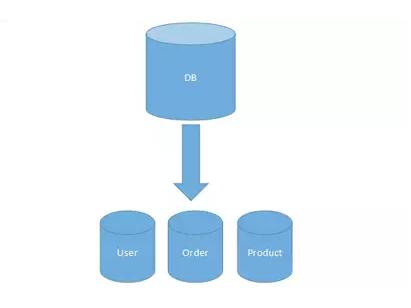
## 垂直分表

垂直分表在日常开发和设计中比较常见，通俗的说法叫做“大表拆小表”，拆分是基于关系型数据库中的“列”（字段）进行的。通常情况，某个表中的字段比较多，可以新建立一张“扩展表”，将不经常使用或者长度较大的字段拆分出去放到“扩展表”中，如下图所示：



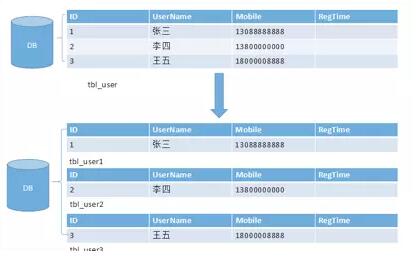
## 垂直分库

垂直分库在“微服务”盛行的今天已经非常普及了。基本的思路就是按照业务模块来划分出不同的数据库，而不是像早期一样将所有的数据表都放到同一个数据库中。如下图： 



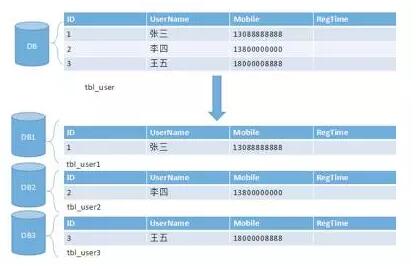
## 水平分表 （不建议用）

水平分表也称为横向分表，比较容易理解，就是将表中不同的数据行按照一定规律分布到不同的数据库表中（这些表保存在同一个数据库中），这样来降低单表数据量，优化查询性能。最常见的方式就是通过主键或者时间等字段进行Hash和取模后拆分。如下图所示：



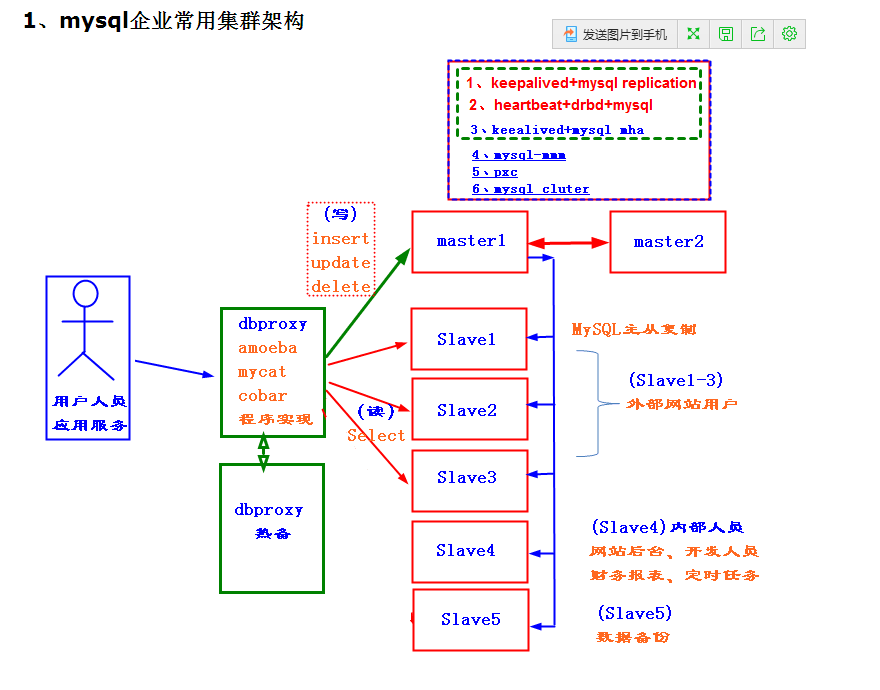
**水平分库**

水平分库分表与上面讲到的水平分表的思想相同，唯一不同的就是将这些拆分出来的表保存在不同的数据中。这也是很多大型互联网公司所选择的做法。如下图：



某种意义上来讲，有些系统中使用的“冷热数据分离”（将一些使用较少的历史数据迁移到其他的数据库中。而在业务功能上，通常默认只提供

1. 集群



在中小型互联网的企业中。mysql的集群一般就是上图的架构。WEB节点读取数据库的时候读取dbproxy服务器。dbproxy服务器通过对SQL语句的判断来进行数据库的读写分离。读请求负载到从库（也可以把主库加上），写请求写主库。

这里的dbproxy是数据库集群的唯一出口所以也需要做高可用。

dbproxy是数据库读写分离的常用软件，amoeba、mycat、cobar也很常用。这类软件不仅带有读写分离功能，还可以实现负载均衡以及后端节点的健康检查。

数据库的读写分离除了通过这类数据库中间件软件实现，还可以写在程序中。

通常我们的主库要做双主高可用，实现主库挂掉另一个主库立刻接管。如果不做双主，从库接管主库的时候需要做状态迁移，会有延迟。

数据库主库的高可用重点需要考虑的是数据同步。比较常用的高可用方案有：

1、keepalived+mysql replication。通过keepalived实现VIP飘逸，通过mysql自带的同步方案replication实现数据同步。

2、hearbeat+drbd。通过drbd实现双主数据的同步，这个数据同步是基于块设备的。比一般的同步方案要快很多。通过heartbeat实现VIP漂移以及drbd资源的切换管理。

3、keepalived+mha。

对于从库，最好不要超过5个。我们可以把其中的三个作为用户访问的节点，把另外一个作为内部人员的查询节点。因为内部人员查询节点的时候一般是按照时间段查询，不经过索引，占用的资源比较多，所以要把这个节点单独专用，以免影响客户访问。最后我们应该留一个从库进行数据库的数据备份。

从库的数据一致性保持可以通过直接于主库进行主从辅助，也可以从其他从库那进行主从复制（优点是减少主库压力，缺点是延迟稍大）。

1. myabtis中${}和#{}的区别
   1. ${ } 变量的替换阶段是在动态 SQL 解析阶段，而 #{ }变量的替换是在 DBMS 中。
   2. 能使用 #{ } 的地方就用 #{ }首先这是为了性能考虑的，相同的预编译 sql 可以重复利用。其次，${ } 在预编译之前已经被变量替换了，这会存在 sql 注入问题
   3. 表名作为变量时，必须使用 ${ }这是因为，表名是字符串，使用 sql 占位符替换字符串时会带上单引号 ''，这会导致 sql 语法错误
2. 线程池大小的设定

cpu数量x(线程等待时间/线程cup执行时间+1)

1. jvm优化策略

参考<http://pengjiaheng.iteye.com/blog/552456>

<http://blog.csdn.net/madun/article/details/7913043>

1. **有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：**

  --年老代年轻代大小划分是否合理

  --内存泄漏

  --垃圾回收算法设置是否合理

2. **热点分析**

**CPU热点**：检查系统哪些方法占用的大量CPU时间

**内存热点**：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

3.**快照**

    快照是系统运行到某一时刻的一个定格。在我们进行调优的时候，不可能用眼睛去跟踪所有系统变化，依赖快照功能，我们就可以进行系统两个不同运行时刻，对象（或类、线程等）的不同，以便快速找到问题

举例说，我要检查系统进行垃圾回收以后，是否还有该收回的对象被遗漏下来的了。那么，我可以在进行垃圾回收前后，分别进行一次堆情况的快照，然后对比两次快照的对象情况。

## 4. ****内存泄漏检查****

    内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

    内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。

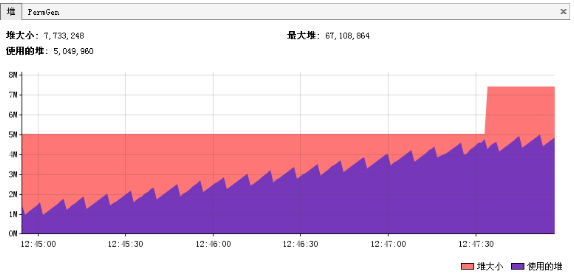
    内存泄漏对系统危害比较大，因为他可以直接导致系统的崩溃。

需要区别一下，内存泄漏和系统超负荷两者是有区别的，虽然可能导致的最终结果是一样的。内存泄漏是用完的资源没有回收引起错误，而系统超负荷则是系统确实没有那么多资源可以分配了（其他的资源都在使用）。

**5.年老代堆空间被占满**

**异常：** java.lang.OutOfMemoryError: Java heap space

**说明：**



    这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。

如上图所示，这是非常典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有谷底部分表示是一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随时间的推移，系统的堆空间被不断占满，最终会占满整个堆空间。因此可以初步认为系统内部可能有内存泄漏。（上面的图仅供示例，在实际情况下收集数据的时间需要更长，比如几个小时或者几天）

**解决：**

    这种方式解决起来也比较容易，一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

6. **持久代被占满**

**异常：**java.lang.OutOfMemoryError: PermGen space

**说明：**

    Perm空间被占满。无法为新的class分配存储空间而引发的异常。这个异常以前是没有的，但是在Java反射大量使用的今天这个异常比较常见了。主要原因就是大量动态反射生成的类不断被加载，最终导致Perm区被占满。

    更可怕的是，不同的classLoader即便使用了相同的类，但是都会对其进行加载，相当于同一个东西，如果有N个classLoader那么他将会被加载N次。因此，某些情况下，这个问题基本视为无解。当然，存在大量classLoader和大量反射类的情况其实也不多。

**解决：**

    1. -XX:MaxPermSize=16m

2. 换用JDK。比如JRocket(大量的行业基准测试显示，基本JRockit JVM是世界上最快的JVM; JRockit面向延迟敏感型应用的解决方案JRockit Real Time提供以毫秒或微秒级的JVM响应时间)。

7. **堆栈溢出**

**异常：**java.lang.StackOverflowError

**说明：**这个就不多说了，一般就是递归没返回，或者循环调用造成

8. **线程堆栈满**

**异常**：Fatal: Stack size too small

**说明**：java中一个线程的空间大小是有限制的。JDK5.0以后这个值是1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。

**解决**：增加线程栈大小。-Xss2m。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

9. **系统内存被占满**

**异常**：java.lang.OutOfMemoryError: unable to create new native thread

**说明**：

    这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。

分配给Java虚拟机的内存愈多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。同时，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总共内生产的线程数。

**解决：**

    1. 重新设计系统减少线程数量。

    2. 线程数量不能减少的情况下，通过-Xss减小单个线程大小。以便能生产更多的线程。

# ++++++++++++++++[Xms Xmx PermSize MaxPermSize 区别](http://www.cnblogs.com/mingforyou/archive/2012/03/03/2378143.html)++++++++++++

Eclipse崩溃，错误提示：  
MyEclipse has detected that less than 5% of the 64MB of Perm   
Gen (Non-heap memory) space remains. It is strongly recommended  
that you exit and restart MyEclipse with new virtual machine memory  
paramters to increase this memory.   Failure to do so can result in  
data loss. The recommended Eclipse memory parameters are:   
eclipse.exe -vmargs -Xms128M -Xmx512M -XX:PermSize=64M -XX:MaxPermSize=128M  
   
1.参数的含义  
**-vmargs -Xms128M -Xmx512M -XX:PermSize=64M -XX:MaxPermSize=128M  
-vmargs 说明后面是VM的参数，所以后面的其实都是JVM的参数了  
-Xms128m JVM初始分配的堆内存  
-Xmx512m JVM最大允许分配的堆内存，按需分配  
-XX:PermSize=64M JVM初始分配的非堆内存  
-XX:MaxPermSize=128M JVM最大允许分配的非堆内存，按需分配**

我们首先了解一下JVM内存管理的机制，然后再解释每个参数代表的含义。

**1)堆(Heap)和非堆(Non-heap)内存**

 按照官方的说法：“Java 虚拟机具有一个堆，堆是运行时数据区域，所有类实例和数组的内存均从此处分配。堆是在 Java 虚拟机启动时创建的。”“在JVM中堆之外的内存称为非堆内存(Non-heap memory)”。  
 可以看出JVM主要管理两种类型的内存：堆和非堆。简单来说堆就是Java代码可及的内存，是留给开发人员使用的；非堆就是JVM留给自己用的，  
 所以方法区、JVM内部处理或优化所需的内存(如JIT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法的代码都在非堆内存中。

**堆内存分配**

 JVM初始分配的堆内存由-Xms指定，默认是物理内存的1/64；JVM最大分配的堆内存由-Xmx指定，默认是物理内存的1/4。默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；  
 空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制。因此服务器一般设置-Xms、-Xmx 相等以避免在每次GC 后调整堆的大小。  
 说明：如果-Xmx 不指定或者指定偏小，应用可能会导致java.lang.OutOfMemory错误，此错误来自JVM，不是Throwable的，无法用try...catch捕捉。

**非堆内存分配**

 JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4。（还有一说：MaxPermSize缺省值和-server -client选项相关，  
 -server选项下默认MaxPermSize为64m，-client选项下默认MaxPermSize为32m。这个我没有实验。）  
 上面错误信息中的PermGen space的全称是Permanent Generation space，是指内存的永久保存区域。还没有弄明白PermGen space是属于非堆内存，还是就是非堆内存，但至少是属于了。  
XX:MaxPermSize设置过小会导致java.lang.OutOfMemoryError: PermGen space 就是内存益出。   
说说为什么会内存益出：   
（1）这一部分内存用于存放Class和Meta的信息，Class在被 Load的时候被放入PermGen space区域，它和存放Instance的Heap区域不同。   
（2）GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的APP会LOAD很多CLASS 的话,就很可能出现PermGen space错误。  
  这种错误常见在web服务器对JSP进行pre compile的时候。

**2)JVM内存限制(最大值)**

 首先JVM内存限制于实际的最大物理内存，假设物理内存无限大的话，JVM内存的最大值跟操作系统有很大的关系。简单的说就32位处理器虽然可控内存空间有4GB,但是具体的操作系统会给一个限制，  
 这个限制一般是2GB-3GB（一般来说Windows系统下为1.5G-2G，Linux系统下为2G-3G），而64bit以上的处理器就不会有限制了。

2. 为什么有的机器我将-Xmx和-XX:MaxPermSize都设置为512M之后Eclipse可以启动，而有些机器无法启动？  
 通过上面对JVM内存管理的介绍我们已经了解到JVM内存包含两种：堆内存和非堆内存，另外JVM最大内存首先取决于实际的物理内存和操作系统。所以说设置VM参数导致程序无法启动主要有以下几种原因：  
1) 参数中-Xms的值大于-Xmx，或者-XX:PermSize的值大于-XX:MaxPermSize；  
2) -Xmx的值和-XX:MaxPermSize的总和超过了JVM内存的最大限制，比如当前操作系统最大内存限制，或者实际的物理内存等等。说到实际物理内存这里需要说明一点的是，  
 如果你的内存是1024MB，但实际系统中用到的并不可能是1024MB，因为有一部分被硬件占用了。

3. 为何将上面的参数写入到eclipse.ini文件Eclipse没有执行对应的设置？  
 那为什么同样的参数在快捷方式或者命令行中有效而在eclipse.ini文件中是无效的呢？这是因为我们没有遵守eclipse.ini文件的设置规则：  
参数形如“项 值”这种形式，中间有空格的需要换行书写，如果值中有空格的需要用双引号包括起来。比如我们使用-vm C:/Java/jre1.6.0/bin/javaw.exe参数设置虚拟机，  
在eclipse.ini文件中要写成这样：  
-vm   
C:/Java/jre1.6.0/bin/javaw.exe   
-vmargs   
-Xms128M   
-Xmx512M   
-XX:PermSize=64M   
-XX:MaxPermSize=128M   
实际运行的结果可以通过Eclipse中“Help”-“About Eclipse SDK”窗口里面的“Configuration Details”按钮进行查看。  
另外需要说明的是，Eclipse压缩包中自带的eclipse.ini文件内容是这样的：  
-showsplash   
org.eclipse.platform   
--launcher.XXMaxPermSize   
256m   
-vmargs   
-Xms40m   
-Xmx256m   
其中–launcher.XXMaxPermSize（注意最前面是两个连接线）跟-XX:MaxPermSize参数的含义基本是一样的，我觉得唯一的区别就是前者是eclipse.exe启动的时候设置的参数，  
而后者是eclipse所使用的JVM中的参数。其实二者设置一个就可以了，所以这里可以把–launcher.XXMaxPermSize和下一行使用#注释掉。

4. 其他的启动参数。 如果你有一个双核的CPU，也许可以尝试这个参数:  
-XX:+UseParallelGC  
让GC可以更快的执行。（只是JDK 5里对GC新增加的参数）

补充：  
　　如果你的WEB APP下都用了大量的第三方jar，其大小超过了服务器jvm默认的大小，那么就会产生内存益出问题了。  
解决方法： 设置MaxPermSize大小   
可以在myelipse里选中相应的服务器比如tomcat5，展开里面的JDK子项页面，来增加服务器启动的JVM参数设置：  
-Xms128m   
-Xmx256m   
-XX:PermSize=128M   
-XX:MaxNewSize=256m   
-XX:MaxPermSize=256m  
或者手动设置MaxPermSize大小,比如tomcat，  
修改TOMCAT\_HOME/bin/catalina.bat，在echo "Using CATALINA\_BASE: $CATALINA\_BASE"上面加入以下行：   
JAVA\_OPTS="-server -XX:PermSize=64M -XX:MaxPermSize=128m

建议：将相同的第三方jar文件移置到tomcat/shared/lib目录下，这样可以减少jar 文档重复占用内存