**线程池的作用：**

线程池作用就是限制系统中执行线程的数量。  
     根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程排队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池中有等待的工作线程，就可以开始运行了；否则进入等待队列。

**为什么要用线程池:**

1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。

**Comparable和Comparator接口是干什么的**？列出它们的区别。

Java提供了只包含一个compareTo()方法的Comparable接口。这个方法可以个给两个对象排序。具体来说，它返回负数，0，正数来表明输入对象小于，等于，大于已经存在的对象。

Java提供了包含compare()和equals()两个方法的Comparator接口。compare()方法用来给两个输入参数排序，返回负数，0，正数表明第一个参数是小于，等于，大于第二个参数。equals()方法需要一个对象作为参数，它用来决定输入参数是否和comparator相等。只有当输入参数也是一个comparator并且输入参数和当前comparator的排序结果是相同的时候，这个方法才返回true。

没有final修饰的变量相加后会被自动提升为int型

**HashSet和TreeSet有什么区别？**

HashSet是由一个hash表来实现的，因此，它的元素是无序的。add()，remove()，contains()方法的时间复杂度是O(1)。

另一方面，TreeSet是由一个树形的结构来实现的，它里面的元素是有序的。因此，add()，remove()，contains()方法的时间复杂度是O(logn)。

**什么是Java虚拟机**？为什么Java被称作是“平台无关的编程语言”？

Java虚拟机是一个可以执行Java字节码的虚拟机进程。Java源文件被编译成能被Java虚拟机执行的字节码文件。

**HashMap和HashTable有何不同？**

（1）HashMap允许key和value为null，而HashTable不允许。

（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。

ArrayList和Vector。

（1）Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。

（2）ArrayList比Vector快，它因为有同步，不会过载、

（3）ArrayList 和vector都是数组实现

LinkedHashMap 继承自 HashMap，具有高效性，同时在 HashMap 的基础上，又在内部增加了一个链表，用以存放元素的顺序。

LinkedHashMap 是基于元素进入集合的顺序或者被访问的先后顺序排序，TreeMap 则是基于元素的固有顺序 (由 Comparator 或者 Comparable 确定)

**遍历hashmap**

通过entry遍历map

for(Entry entry:map.entrySet()){

K key = entry.getKey();

V value = entry.getValue();

}

将map转换为list (ArrayList)map.keySet()(ArrayList)map.valueSet()

(ArrayList)map.entryset()

Hashmap排序

Collections.sort(list,new Comparator(){

Public int compare(Entry e1,Entry e2){

Return e1.getValue().compare(e2.getValue());

}

});

**Collection 和 Collections的区别**   
Collections是个java.util下的类，它包含有各种有关集合操作的静态方法。   
Collection是个java.util下的接口，它是各种集合结构的父接口。

 Collection  
├List  
│├LinkedList对列表中任何位置的成员的增加和删除支持较好，但对基于索引的成

│ 员访问支持性能较差；LinkedList底层采用双向循环列表实现，可以使用Linked

List来实现队列和栈  
│├ArrayList 通过ensureCapacity(int n)方法可提高ArrayList的初始化速度  
│└Vector  
│　└Stack  
└Set

├LinkedHashSet外部按成员的插入顺序遍历成员

├treeset外部有序地遍历成员;附加实现了SortedSet,支持子集等要求顺序的操作

├hashset成员要求实现caparable接口,或者使用Comparator构造TreeSet成员可

为任意Object子类的对象,但如果覆盖了equals方法,同时注意修改hashCode方法

Map  
├Hashtable key和value都不允许出现null值，Hashtable 中的方法是同步的

├TreeMap支持对键有序地遍历，使用时建议先用HashMap增加和删除成员，最后从HashMap生成TreeMap；附加实现了SortedMap接口，支持子Map等要求顺序的操作  
├HashMap中null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值

为Null; 应该用containsKey()方法来判断是否存在某个键  
└LinkedHashMap 保留键的插入顺序，用equals方法检查键和值的相等性

Map对值排序:Collections.sort()

关于set如何保证元素不重复：HashSet的底层采用HashMap来存放数据；当向HashMap中添加元素的时候，首先计算元素的hashcode值，然后根据1处的代码计算出Hashcode的值，再根据2处的代码计算出这个元素的存储位置，如果这个位置为空，就将元素添加进去；如果不为空，遍历索引为i的链上的元素，如果key重复，则替换并返回oldValue值

当从HashSet集合中查找某个对象时，Java系统首先调用对象的hashCode()方法获得该对象的哈希码表，然后根据哈希吗找到相应的存储区域，最后取得该存储区域内的每个元素与该对象进行equals方法比较；

我们重写一个对象的equals方法，就**必须重写他的hashCode**方法，不过不重写他的hashCode方法的话，Object对象中的hashCode方法始终返回的是一个对象的hash地址，而这个地址是永远不相等的。所以这时候即使是重写了equals方法，也不会有特定的效果的，因为hashCode方法如果都不想等的话，就不会调用equals方法进行比较了，所以没有意义了。只要是两个不同的实例对象，即使他们的equals方法比较结果相等，他们默认的hashCode方法(未重写的)的返回值是不同的。

**throw和throws有什么区别**？

throw关键字用来在程序中明确的抛出异常，相反，throws语句用来表明方法不能处理的异常。每一个方法都必须要指定哪些异常不能处理，所以方法的调用者才能够确保处理可能发生的异常，多个异常是用逗号分隔的。

**sendRedirect()和forward()方法有什么区别**？

sendRedirect()方法会创建一个新的请求，而forward()方法只是把请求转发到一个新的目标上。重定向(redirect)以后，之前请求作用域范围request以内的对象就失效了，因为会产生一个新的请求，而转发(forwarding)以后，之前请求作用域范围以内的对象还是能访问的。一般认为sendRedirect()比forward()要慢。

**Java反射机制的作用？**

运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；

1）在运行时判断任意一个对象所属的类。

2）在运行时构造任意一个类的对象。

3）在运行时判断任意一个类所具有的成员变量和方法。

4）在运行时调用任意一个对象的方法

Class 类十分特殊。它和一般类一样继承自Object，其实体用以表达Java程序运行时的classes和interfaces

**类有哪三个基本特性？各特性的优点？**

类具有封装性、继承性和多态性。

封装性：类的封装性为类的成员提供公有、缺省、保护和私有等多级访问权限，目的是隐藏类中的私有变量和类中方法的实现细节。

继承性：类的继承性提供从已存在的类创建新类的机制，继承（inheritance）使一个新类自动拥有被继承类（父类）的全部可继承的成员。

不能降低从基类继承的方法的可见性，即子类方法的权限至少要大于父类方法的权限

多态性：类的多态性提供类中方法执行的多样性，多态性有两种表现形式：重载和覆盖

**怎么理解java的泛型？**

解答： 在Java SE 1.5之前，没有泛型的情况的下，通过对类型Object的引用来实现参数的“任意化”，“任意化”带来的缺点是要做显式的强制类型转换，而这种转换是要求开发者对实际参数类型可以预知的情况下进行的。对于强制类型转换错误的情况，编译器可能不提示错误，在运行的时候才出现异常，这是一个安全隐患。

泛型是Java SE 1.5的新特性，泛型的本质是**参数化类型**，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。

泛型的好处是在**编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，提高代码的重用率**。

**在一个千万级的数据库查寻中，如何提高查询效率？**分别说出在数据库设计、SQL语句、java等层面的解决方案。

1）数据库设计方面：

a. 对查询进行优化，应尽量**避免全表扫描**，首先应考虑在where及order by涉及的列上建立索引。

b. 应尽量**避免在where子句中对字段进行null值判断，否则将导致引擎放弃使用索引而进行全表扫描**，如：select id from t where num is null 可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：select id from t where num=0

c. 并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，**当索引列有大量数据重复时,查询可能不会去利用索引**，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

d. 索引并不是越多越好，索引固然可以提高相应的select的效率，但同时也**降低了insert及update的效率**，因为insert或update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

e. 应尽可能的避免更新索引数据列，因为索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新索引数据列，那么需要考虑是否应将该索引建为索引。

f. **尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销**。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

g. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先**变长字段存储空间小，可以节省存储空间**，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

h. 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

i. 避免频繁创建和删除临时表，以减少系统表资源的消耗。

j. 临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

k. 在新建临时表时，如果一次性插入数据量很大，那么可以使用select into代替create table，避免造成大量log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

l. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先truncate table ，然后drop table ，这样可以避免系统表的较长时间锁定。

2)SQL语句方面：

a. 应尽量**避免在where子句中使用!=或<>操作符**，否则将引擎放弃使用索引而进行全表扫描。

b. 应尽量**避免在where子句中使用or来连接条件**，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num=10 or num=20可以这样查询：select id from t where num=10 union all select id from t where

num=20

c. **in 和 not in 也要慎用，否则会导致全表扫描**，如： select id from t where num in(1,2,3)对于连续的数值，**能用between就不要用in了**：select id from t where num between 1 and 3

d. 下面的查询也将导致全表扫描： select id from t where name like ‘%abc%’

e. 如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：select id from t where num=@num 可以改为强制查询使用索引：select id from t with(index(索引名)) where num=@num

f. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如： select id from t where num/2=100 应改为: select id from t where num=100\*2

g. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如： select id from t where substring(name,1,3)=’abc’–name以abc开头的id select id from t where datediff(day,createdate,’2005-11-30′)=0–‘2005-11-30’生成的id 应改为: select id from t where name like ‘abc%’ select id from t where createdate>=’2005-11-30′ and createdate<’2005-12-1′

h. 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

i. 不要写一些没有意义的查询，如需要生成一个空表结构：select col1,col2 into #t from t where 1=0 这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样： create table #t(„)

j. 很多时候用 exists 代替 in 是一个好的选择：select num from a where num in(select num from b)用下面的语句替换：select num from a where exists(select 1 from b where num=a.num)

k. 任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，**不要返回用不到的任何字段**。

l. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

m. 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

n. 尽量避免大事务操作，提高系统并发能力。

3)java方面:

a.尽可能的少造对象。

b.合理摆正系统设计的位置。大量数据操作，和少量数据操作一定是分开的。大量的数据操作，肯定不是ORM框架搞定的。，

c.使用jDBC链接数据库操作数据

d.控制好内存，让数据流起来，而不是全部读到内存再处理，而是边读取边处理；

e.合理利用内存，有的数据要缓存

**hibernate延迟加载**

Hibernae 的延迟加载是一个非常常用的技术，实体的**集合属性默认会被延迟加载**，实体所**关联的实体默认也会被延迟加载**。Hibernate 通过这种延迟加载来降低系统的内存开销，从而保证 Hibernate 的运

行性能。

属性的延迟加载功能：当Hibernate在查询数据的时候，数据并没有存在与内存中，当程序真正对数据的操作时，对象才存在与内存中，就实现了延迟加载，他节省了服务器的内存开销，从而提高了服务器的性能

关联实体是多个实体时（包括一对多、多对多）：此时关联实体将以集合的形式存在，Hibernate 将使用 PersistentSet、PersistentList、PersistentMap、PersistentSortedMap、PersistentSortedSet 等集合来管理延迟加载的实体

关联实体是单个实体时（包括一对一、多对一）：当 Hibernate 加载某个实体时，延迟的关联实体将是一个动态生成代理对象。

当关联实体是单个实体时，也就是使用 <many-to-one.../> 或 <one-to-one.../> 映射关联实体的情形，这两个元素也可通过 lazy 属性来指定延迟加载。

加载主实体时，并未真正去抓取关联实体对应数据，而**只是动态地生成一个对象作为关联实体的代理。当应用程序真正需要使用关联实体时，代理对象会负责从底层数据库抓取记录，并初始化真正的关联实体**。

在 Hibernate 的延迟加载中，客户端程序开始**获取的只是一个动态生成的代理对象**，而真正的实体则委托给代理对象来管理——这就是典型的代理模式

我们让数据在需要的时候才进行加载，这时我们就用到了延迟加载。

比如部门ENTITY和员工ENTITY,部门与员工1对多，如果lazy设置为 false，那么只要加载了一个部门的po,就会根据一对多配置的关系把所有员工的po也加载出来。但是实际上有时候只是需要用到部门的信息，不需要用到 员工的信息，这时员工po的加载就等于浪费资源。如果lazy设置为true,那么只有当你访问部门po的员工信息时候才回去加载员工的po的信息

假如在student对象中包含一个head对象  
如果你确定在用student对象的时候就要用到head对象里的属性，那你就设置立即加载，因为设置立即加载那么在查询student的同时就会查询 student的head

**Hibernate缓存**

一级缓存就是Session级别的缓存，存在于内存中。一个Session做了一个查询操作，它会把这个操作的结果放在一级缓存中，如果短时间内这个session（一定要同一个session）又做了同一个操作，那么hibernate直接从一级缓存中拿，而不会再去连数据库，取数据。

二级缓存就是SessionFactory级别的缓存，存在于硬盘中。顾名思义，就是查询的时候会把查询结果缓存到二级缓存中，如果同一个sessionFactory创建的某个session执行了相同的操作，hibernate就会从二级缓存中拿结果，而不会再去连接数据库。

**如何优化Hibernate?**

1.使用双向一对多关联，不使用单向一对多

2.灵活使用单向一对多关联

3.不用一对一，用多对一取代

4.配置对象缓存，不使用集合缓存

5.一对多集合使用Bag,多对多集合使用Set

6. 继承类使用显式多态

7. 表字段要少，表关联不要怕多，自己再补充一点，上面说到的都是关于数据库设计 带来的优势，从而起到优化框架的效果，但是真正在实际操作中，我们只有尽量控制不使用懒加载“lazy=true”，如果在涉及的尽量使用Hql语句

**Servlet生命周期**

Servlet被服务器实例化后，容器运行其init方法，请求到达时运行其service方法，service方法自动派遣运行与请求对应的doXXX方法（doGet，doPost）等，当服务器决定将实例销毁的时候调用其destroy方法。

web容器加载servlet，生命周期开始。通过调用servlet的**init()**方法进行servlet的初始化。通过**调用service()**方法实现，根据请求的不同调用不同的do\*\*\*()方法。结束服务，web容器调用servlet的**destroy()**方法

**Java异常处理**

异常的继承结构：基类为Throwable，Error和Exception继承Throwable，RuntimeException和IOException等继承Exception

非RuntimeException一般是外部错误，其必须被 try{}catch语句块所捕获

Error类体系描述了Java运行系统中的内部错误以及资源耗尽的情形，Error不需要捕捉

**java类加载器**

引导类加载器（bootstrap class loader）：它用来加载 Java 的核心库，是用原生代码来实现的

扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。

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

tomcat为每个App创建一个Loader，里面保存着此WebApp的ClassLoader。需要加载WebApp下的类时，就取出ClassLoader来使用

**四种会话跟踪技术**

1）隐藏表单域：<input type="hidden">，非常适合步需要大量数据存储的会话应用。  
2）URL 重写:URL 可以在后面附加参数，和服务器的请求一起发送，这些参数为名字/值对。  
3）Cookie:一个 Cookie是一个小的，已命名数据元素。服务器使用SET-Cookie头标将它作为HTTP响应的一部分传送到客户端，客户端被请求保存Cookie值，在对同一服务器的后续请求使用一个Cookie头标将之返回到服务器。与其它技术比较，Cookie的一个优点是在浏览器会话结束后，甚至在客户端计算机重启后它仍可以保留其值。   
4）Session：使用 setAttribute(String str,Object obj)方法将对象捆绑到一个会话。

隐藏域和URL重写有着共同的优点**:**  
它们在Cookie被禁用或者根本不支持的情况下依旧能够工作。

**关于java初始化的顺序**

不论怎样，必定先执行静态代码；子由父生，所以父类必先执行

执行顺序：1.静态代码块 2.构造代码块3.构造方法

1.静态代码块：是在类的加载过程的第三步初始化的时候进行的，主要目的是给类变量赋予初始值。  
2.构造代码块：构造块：类中直接用{}定义，每一次创建对象时执行。是独立的，必须依附载体才能运行，Java会把构造代码块放到每种构造方法的前面，用于实例化一些共有的实例变量，减少代码量。  
3.构造方法：用于实例化变量。  
1是类级别的，2、3是实例级别的，自然1要优先23.

对子类得主动使用会导致对其父类得主动使用，所以尽管实例化的是子类，但也会导致父类的初始化和实例化，且优于子类执行。

1.首先初始化父类中的静态成员变量和静态代码块,按照在程序中出现的顺序初始化

2.然后初始化子类中的静态成员变量和静态代码块,按照在程序中出现的顺序初始化

3.其次，初始化父类的普通成员变量和代码块，在执行父类的构造方法；

4.最后，初始化子类的普通成员变量和代码块，在执行子类的构造方法；

局部变量没有默认值 必须手动初始化

Win64下：long 8字节、short 2字节、int 4字节、int\* 8字节，C++中内存对齐，按最大长度对齐：8+（2+4+2（补齐2字节））+8 = 24字节

8中基本数据类型是：byte short int long float double boolean char

Double 8字节 float4字节 0.0f char 2字节 int 4字节

**哈希表**

插入删除查找速度O(1);不需要遍历表，关键字和所在位置有函数关系

平方取中法

取关键字平方后的中间几位为哈希地址

使用哈希查找有两个步骤:

1. 使用哈希函数将被查找的**键转换为数组的索引**。在理想的情况下，不同的键会被转换为不同的索引值，但是在有些情况下我们需要处理多个键被哈希到同一个索引值的情况。所以哈希查找的第二个步骤就是处理冲突
2. 处理哈希碰撞冲突。有很多处理哈希碰撞冲突的方法，如拉链法(将大小为M 的数组的每一个元素指向一个条链表，链表中的每一个节点都存储散列值为该索引的键值对)和线性探测法(开放寻址法的一种, 当碰撞发生时即一个键的散列值被另外一个键占用时，直接检查散列表中的下一个位置即将索引值加1)。

**Java线程**

在java中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程。因为每当使用java命令执行一个类的时候，实际上都会启动一个ＪＶＭ，每一个ｊＶＭ实习在就是在操作系统中启动了一个进程

在调用start()方法之前：线程处于新状态中，新状态指有一个Thread对象，但还没有一个真正的线程。

在调用start()方法之后：发生了一系列复杂的事情

启动新的执行线程（具有新的调用栈）；

该线程从新状态转移到可运行状态；

当该线程获得机会执行时，其目标run()方法将运行

1、线程的名字，一个运行中的线程总是有名字的，名字有两个来源，一个是虚拟机自己给的名字，一个是你自己的定的名字。在没有指定线程名字的情况下，虚拟机总会为线程指定名字，并且主线程的名字总是mian，非主线程的名字不确定。

2、线程都可以设置名字，也可以获取线程的名字，连主线程也不例外。

3、获取当前线程的对象的方法是：Thread.currentThread()；

4、在上面的代码中，只能保证：每个线程都将启动，每个线程都将运行直到完成。一系列线程以某种顺序启动并不意味着将按该顺序执行。对于任何一组启动的线程来说，调度程序不能保证其执行次序，持续时间也无法保证。

5、当线程目标run()方法结束时该线程完成。

6、一旦线程启动，它就永远不能再重新启动。只有一个新的线程可以被启动，并且只能一次。一个可运行的线程或死线程可以被重新启动。

7、线程的调度是JVM的一部分，在一个CPU的机器上上，实际上一次只能运行一个线程。一次只有一个线程栈执行。JVM线程调度程序决定实际运行哪个处于可运行状态的线程。

众多可运行线程中的某一个会被选中做为当前线程。可运行线程被选择运行的顺序是没有保障的。

8、尽管通常采用队列形式，但这是没有保障的。队列形式是指当一个线程完成“一轮”时，它移到可运行队列的尾部等待，直到它最终排队到该队列的前端为止，它才能被再次选中。事实上，我们把它称为可运行池而不是一个可运行队列，目的是帮助认识线程并不都是以某种有保障的顺序排列唱呢个一个队列的事实。

9、尽管我们没有无法控制线程调度程序，但可以通过别的方式来影响线程调度的方式。

等待/阻塞/睡眠Thread.sleep(long millis)状态：这是线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的，但是当前没有条件运行。换句话说，**它是可运行的**，但是如果某件事件出现，他可能返回到可运行状态。

线程睡眠到期自动苏醒，并返回到**可运行**状态，不是运行状态。sleep()中指定的时间是线程不会运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就开始执行

死亡态：当线程的run()方法完成时就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。 如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

线程的优先级和线程让步yield()

线程的让步是通过Thread.**yield**()来实现的。**yield()方法的作用是：暂停当前正在执行的线程对象，并执行其他线程**。**线程仍处于可执行状态**。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。Thread.currentThread().yield();

**suspend（）和resume（）方法**：两个方法配套使用，suspend（）使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume（） 被调用，才能使得线程重新进入可执行状态。典型地，suspend（） 和 resume（）被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用resume

线程总是存在优先级，优先级范围在1~10之间。JVM线程调度程序是**基于优先级**的抢先调度机制。在大多数情况下，当前运行的线程优先级将大于或等于线程池中任何线程的优先级

设置线程的优先级：线程默认的优先级是创建它的执行线程的优先级。可以通过setPriority(int newPriority)更改线程的优先级

static int MAX\_PRIORITY 线程可以具有的最高优先级。   
static int MIN\_PRIORITY 线程可以具有的最低优先级。   
static int NORM\_PRIORITY 分配给线程的默认优先级。

Thread的非静态方法join()让一个线程B“加入”到另外一个线程A的尾部。在A执行完毕之前，B不能工作；join()方法还有带超时限制的重载版本。例如t.join(500)则让线程等待500毫秒，如果超过这个时间，则停止等待，变为可运行状态

当 a thread 调用Join方法的时候a.join()，MainThread 就被停止执行，直到 **a thread 线程执行完毕**。

把竞争访问的资源类的C变量x标识为private；同步那些修改变量的代码，使用synchronized关键字同步方法或代码。

**垃圾回收：释放那些不再持有引用的对象的内存**

怎么判断一个对象是否需要收集？

引用计数（最简单古老的方法）：指将资源（可以是对象、内存或磁盘空间等等）的被引用次数保存起来，当被引用次数变为零时就将其释放的过程

对象引用遍历（现在大多数 jvm 使用的方法）：对象引用遍历从一组对象开始，沿着整个对象图上的每条链接，递归确定可到达（reachable）的对象。如果某对象不能从这些根对象的一个（至少一个）到达，则将它作为垃圾收集

**常用垃圾回收算法**

（1）标记－清除收集器

这种收集器首先遍历对象图并标记可到达的对象，然后扫描堆栈以寻找未标记对象并释放它们的内存。这种收集器一般使用单线程工作并停止其他操作。

（2）标记－压缩收集器

有时也叫标记－清除－压缩收集器，与标记－清除收集器有相同的标记阶段。在第二阶段，则把标记对象复制到堆栈的新域中以便压缩堆栈。这种收集器也停止其他操作

（3）复制收集器

这种收集器将堆栈分为两个域，常称为半空间。每次仅使用一半的空间，虚拟机生成的新对象则放在另一半空间中。垃圾回收器运行时，它把可到达对象复制到另一半空间，没有被复制的的对象都是不可达对象，可以被回收。这种方法适用于短生存期的对象，持续复制长生存期的对象由于多次拷贝，导致效率降低。缺点是只有一半的虚拟机空间得到使用。

**锁的原理：**Java中每个对象都有一个内置锁

当程序运行到非静态的synchronized同步方法上时，自动获得与正在执行代码类的当前实例（this实例）有关的锁。获得一个对象的锁也称为获取锁、锁定对象、在对象上锁定或在对象上同步

线程睡眠时，它所持的任何锁都不会释放。

线程可以获得多个锁。比如，在一个对象的同步方法里面调用另外一个对象的同步方法，则获取了两个对象的同步锁。

同步损害并发性，应该尽可能缩小同步范围。同步不但可以同步整个方法，还可以同步方法中一部分代码块。

在使用同步代码块时候，应该指定在哪个对象上同步，也就是说要获取哪个对象的锁

要同步静态方法，需要一个用于整个类对象的锁，这个对象是就是这个类

当在对象上调用wait()方法时，执行该代码的线程立即放弃它在对象上的锁。然而调用notify()时，并不意味着这时线程会放弃其锁

在一个线程中开启另外一个新线程，则新开线程称为该线程的子线程，子线程初始优先级与父线程相同;默认的线程优先级是5.主线程也有可能在子线程结束之前结束。并且子线程不受影响，不会因为主线程的结束而结束

**ThreadLocal和Synchonized**都用于解决多线程并发访问。但是ThreadLocal与synchronized有本质的区别。synchronized是利用锁的机制，使变量或代码块在某一时该只能被一个线程访问。而ThreadLocal为每一个线程都提供了变量的副本，使得每个线程在某一时间访问到的并不是同一个对象，这样就隔离了多个线程对数据的数据共享。而Synchronized却正好相反，它用于在多个线程间通信时能够获得数据共享。

ThreadLocal是解决线程安全问题一个很好的思路，ThreadLocal类中有一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本，由于Key值不可重复，每一个“线程对象”对应线程的“变量副本”，而到达了线程安全

**ThreadLocal使用的一般步骤**

1、在多线程的类（如ThreadDemo类）中，创建一个ThreadLocal对象threadXxx，用来保存线程间需要隔离处理的对象xxx。

2、在ThreadDemo类中，创建一个获取要隔离访问的数据的方法getXxx()，在方法中判断，若ThreadLocal对象为null时候，应该new()一个隔离访问类型的对象，并强制转换为要应用的类型。

3、在ThreadDemo类的run()方法中，通过getXxx()方法获取要操作的数据，这样可以保证每个线程对应一个数据对象，在任何时刻都操作的是这个对象。

**可重入锁ReentrantLock**

ReentrantLock 类实现了Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，**当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上**。）

*reentrant* 锁意味着什么呢？简单来说，它**有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放**。这模仿了 synchronized 的语义；**如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续）synchronized块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个synchronized块时，才释放锁。**

在查看清单 1 中的代码示例时，可以看到 Lock 和 synchronized 有一点明显的区别 —— **lock 必须在 finally 块中释放**。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放！这一点区别看起来可能没什么，但是实际上，它极为重要。忘记在 finally 块中释放锁，可能会在程序中留下一个定时炸弹，当有一天炸弹爆炸时，您要花费很大力气才有找到源头在哪。而使用同步，JVM 将确保锁会获得自动释放

Lock lock = new ReentrantLock();

lock.lock();

try {

// update object state

}

finally {

lock.unlock();

}

当许多线程都在**争用同一个锁时**，使用 ReentrantLock 的总体开支通常要比 synchronized 少得多。可重入锁的劣势：必须在finally中手动释放锁。

**可重入锁的新特性**

1.等待可中断

在持有锁的线程长时间不释放锁的时候,**等待的线程可以选择放弃等待.**  tryLock(long timeout, TimeUnit unit)

2.公平锁

按照申请锁的顺序来一次获得锁称为公平锁.synchronized的是非公平锁,ReentrantLock可以通过构造函数实现公平锁.new RenentrantLock(boolean fair)

3.绑定多个Condition

通过多次newCondition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal();

 ReentrantLock 构造器的一个参数是 boolean 值，它允许您选择想要一个 公平（fair）锁，还是一个 不公平（unfair）锁。公平锁使线程按照请求锁的顺序依次获得锁；而不公平锁则允许讨价还价，在这种情况下，线程有时可以比先请求锁的其他线程先得到锁。

**Wait与notify方法**

wait(),notify(),notifyAll()属于Object基础类,因为都个对象都有锁,锁是每个对象的基础,当然操作锁的方法也是最基础了

如果对象调用了wait方法就会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。  
如果对象调用了notify方法就**会通知某个正在等待这个对象的控制权的线程**可以继续运行。不能确切的唤醒某一个等待状态的线程而是由JVM确定唤醒哪个线程，而且不是按优先级  
如果对象调用了notifyAll方法就会通知**所有等待这个对象控制权的线程继续运行**。

无论是执行对象的wait、notify还是notifyAll方法，必须保证当前运行的线程取得了该对象的控制权（monitor）也就是必须写在synchronized(obj){...} 代码段内. (只能在同步控制方法或者同步控制块里面使用) 如果在没有控制权的线程里执行对象的以上三种方法，就会报java.lang.IllegalMonitorStateException异常。

线程取得控制权的三种方法：  
1.执行对象的某个同步实例方法。  
2.执行对象对应类的同步静态方法。  
3.执行对该对象加同步锁的同步块。

当线程A获得了obj锁后，发现条件condition不满足，无法继续下一处理，于是线程A就wait()。  
在另一线程B中，如果B更改了某些条件，使得线程A的condition条件满足了，就可以唤醒线程A

　synchronized(obj) {  
　　while(!condition) {  
　　obj.wait();  
　　}  
　　obj.doSomething();  
　　}

　synchronized(obj) {  
　　condition = true;  
　　obj.notify();  
　}

**有返回值的线程**

可返回值的任务必须实现Callable接口,无返回值的任务必须Runnable接口。

执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。

 public Object call() throws Exception {   
   return oid+"任务返回的内容";   
 }

**java lock锁**

在执行同步代码之前使用lock.lock()；同步代码之后使用lock.unlock();

替代synchronized关键字

读写锁：

ReadWriteLock内置两个Lock，一个是读的Lock，一个是写的Lock。

ReadWriteLock lock = new ReentrantReadWriteLock();

Lock writeLock = lock.writeLock();

writeLock.lock();

//mutex

writeLock.unlock();

多个线程可同时得到读的Lock，但只有一个线程能得到写的Lock，

而且写的Lock被锁定后，任何线程都不能得到Lock。ReadWriteLock提供的方法有：

readLock(): 返回一个读的lock

writeLock(): 返回一个写的lock, 此lock是排他的。

读的时候可以同时读，但不能写；写的时候既不能同时写也不能读

使用 **volatile** 变量的条件

在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

**对变量的写操作不依赖于当前值**。

该变量没有包含在具有其他变量的不变式中

即只有在状态真正独立于程序内其他内容时才能使用 volatile

Volatile修饰的成员变量在每次被线程访问时，都**强迫从共享内存中重读该成员变量的值**。而且，当成员变量发生变化时，**强迫线程将变化值回写到共享内存**。这样在任何时刻，两个不同的线程**总是看到某个成员变量的同一个值**。**volatile关键字不能保证线程安全**

**作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值**

**优化器在用到被volatile修饰的变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。**  
Java语言规范中指出：为了获得最佳速度，**允许线程保存共享成员变量的私有拷贝，而且只当线程进入或者离开同步代码块时才与共享成员变量的原始值对比**。  
  
这样当多个线程同时与某个对象交互时，就必须要注意到要让线程及时的得到共享成员变量的变化。而volatile关键字就是提示JVM：对于这个成员变量不能保存它的私有拷贝，而应直接与共享成员变量交互

使用建议：在两个或者更多的线程访问的成员变量上使用volatile。当要访问的变量已在synchronized代码块中，或者为常量时，不必使用

**Interface**

接口不能被private,static和protected修饰

抽象类可以有普通成员变量，接口中只能有final类型的成员变量

抽象类和接口中都可以包含静态成员变量。

**Jsp include**

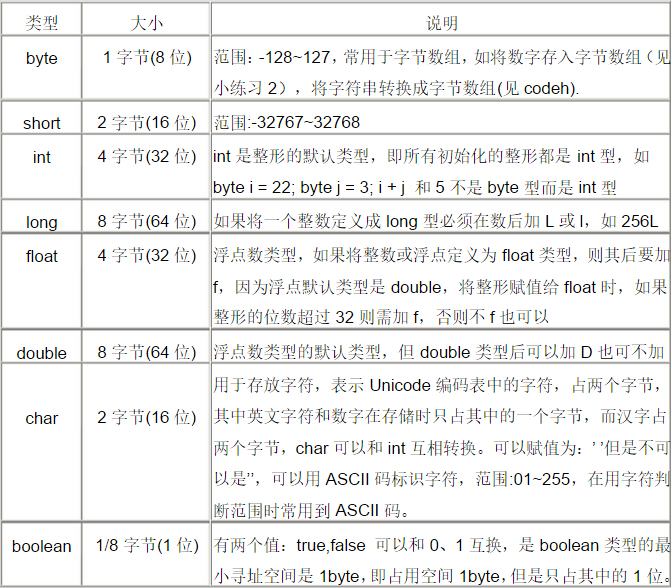
动态INCLUDE用jsp:include 动作实现 <jsp:include page="included.jsp"/>它总是会检查所含文件中的变化,适合用于包含动态页面,并且可以带参数。各个文件分别先编译，然后组合成一个文件.

静态 INCLUDE 用 include 伪码实现 , 定不会检查所含文件的变化 , 适用于包含静态页面 <%@ include file="included.htm" %> 。先将文件的代码被原封不动地加入到了主页面从而合成一个文件，然后再进行翻译，此时不允许有相同的变量。

<%@ include file="relativeURI"%>引入静态文本 (html,jsp), 在 JSP 页面被转化成servlet之前和它融和到一起 .

<jsp:include page="relativeURI" flush="true" />引入执行页面或servlet所生成的应答文本

**Java变量长度**



**抽象类与接口**

接口是公开的，里面不能有私有的方法或变量，是用于让别人使用的，方法全部为public；而抽象类是可以有私有变量的，方法修饰符可以为public protected 和defeult

接口只能包含方法声明，而抽象类还可以包含方法的定义  
  
另外，实现接口的一定要实现接口里定义的所有方法，而实现抽象类可以有选择地重写需要用到的方法，一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现

Java局部变量只能用final修饰，局部变量前不能放置任何访问修饰符

**final关键字**

被final修饰的关键字一旦被初始化便不可改变，这里不可改变的意思对基本类型来说是其值不可变，而对于对象变量来说其引用不可再变。但是对象的成员变量的值是可以改变的

其初始化可以在两个地方，一是其定义处，也就是说在final变量定义时直接给其赋值，二是在构造函数中。这两个地方只能选其一，要么在定义时给值，要么在构造函数中给值，不能同时既在定义时给了值，又在构造函数中给另外的值。

final类型的变量必须显示初始化，且初始化的方法必须是在申明时或者在构造方法中直接赋值，而不能通过调用函数赋值

**Java创建对象的几种方式：**

(1) 用new语句创建对象，这是最常见的创建对象的方法。  
(2) 运用反射手段,调用java.lang.Class或者java.lang.reflect.Constructor类的newInstance()实例方法。  
(3) 调用对象的clone()方法。  
(4) 运用反序列化手段，调用java.io.ObjectInputStream对象的 readObject()方法

**常见设计模式**

装饰者模式：

装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。Source类是被装饰类，Decorator类是一个装饰类，decorator可以为Source类动态的添加一些功能。

public interface Sourceable {//接口

public void method();

}

public class Source implements Sourceable {//装饰者

public void method() {

System.out.println("the original method!");

}

}

public class Decorator implements Sourceable {//被装饰者

private Sourceable source;

public Decorator(Sourceable source){

super();

this.source = source;

}

public void method() {

System.out.println("before decorator!");

source.method();

System.out.println("after decorator!");

}

}

测试类：

public class DecoratorTest {

public static void main(String[] args) {

Sourceable source = new Source();

Sourceable obj = new Decorator(source);

obj.method();

}

}

观察者模式

观察者模式很好理解，类似于邮件订阅和RSS订阅，当我们浏览一些博客或wiki时，经常会看到RSS图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化；对象之间是一种一对多的关系。

适配器模式

public class Person {

private String name;

private String sex;

private int age;

public void speakJapanese(){

System.out.println("I can speak Japanese!");

}

public void speakEnglish(){

System.out.println("I can speak English!");

}

//省略成员变量的get和set方法

}

public interface Job {

public abstract void speakJapanese();

public abstract void speakEnglish();

public abstract void speakFrench();

}

public class Adapter implements Job {

Person person;

public Adapter(Person person) {

this.person = person;

}

public void speakEnglish() {

person.speakEnglish();

}

public void speakJapanese() {

person.speakJapanese();

}

//new add

public void speakFrench() {

}

}

**内部类**

静态内部类，定义在类中，任何方法外，用static定义；静态内部类只能访问外部类的静态成员。生成（new）一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成：Outer.Inner in=new Outer.Inner()；而不需要通过生成外部类对象来生成。这样实际上使静态内部类成为了一个顶级类。可以定义私有静态内部类。

嵌套内部类，就是静态内部类。不需要内部类对象和外部类对象之间的联系，就是说我们可以直接引用outer.inner，即不需要创建外部类，也不需要创建内部类。

嵌套类和普通的内部类还有一个区别：普通内部类不能有static数据和static属性，也不能包含嵌套类，但嵌套类可以。而嵌套类不能声明为private，一般声明为public，方便调用。

非静态外部类可以随意访问其外部类的成员变量以及方法（包括声明为private的方法），但是如果一个内部类被声明为static，则其在访问包括自身的外部类会有诸多的限制。静态内部类不能访问其外部类的非静态成员变量和方法

生成非静态内部类：通用方式Out out = new Out(); out.In in = out.**new** In();在外部类中可以使用的方式：out.In in = new Out.In();

或者在成员内部类要引用外部类对象时，使用outer.this来表示外部类对象.

**Java内存泄露**

静态集合类引起内存泄露：

像HashMap、Vector等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，他们所引用的所有的对象Object也不能被释放，因为他们也将一直被Vector等引用着。

Static Vector v = new Vector(10);   
for (int i = 1; i<100; i++) {   
 Object o = new Object();   
 v.add(o);   
 o = null;   
}  
在这个例子中，循环申请Object 对象，并将所申请的对象放入一个Vector 中，如果仅仅释放引用本身（o=null），那么Vector 仍然引用该对象，所以这个对象对GC 来说是不可回收的。因此，如果对象加入到Vector 后，还必须从Vector 中删除，最简单的方法就是将Vector对象设置为null。

各种连接   
比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。

单例模式

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

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

**equals与hashcode**

equals()相等的两个对象，hashcode()一定相等，equals()不相等的两个对象，却并不能证明他们的hashcode()不相等。换句话说，equals()方法不相等的两个对象，hashCode()有可能相等

不重写equals，那么比较的将是对象的引用是否指向同一块内存地址，重写之后目的是为了比较两个对象的value值是否相等

**java深clone和浅clone**

浅clone：对象类型的成员变量仅仅保存了对象的引用；深clone则将对象类型的成员变量也clone。实现方法：是让UnCloneA类（对象类型的成员变量的类）也实现和CloneB类一样的clone功能（实现Cloneable接口，重载clone()方法）。二是在CloneB的clone()方法中加入一句**o.unCA = (UnCloneA)unCA.clone();**

Cloneable接口是不包含任何方法的，其实这个接口仅仅是一个标志，而且这个标志也仅仅是针对Object类中clone()方法的，如果clone类没有实现Cloneable接口，并调用了Object的clone()方法（也就是调用了super.Clone()方法），那么Object的clone()方法就会抛出CloneNotSupportedException异常

**HTTP响应**由三个部分组成：

状态码(Status Code)：描述了响应的状态。可以用来检查是否成功的完成了请求。请求失败的情况下，状态码可用来找出失败的原因。如果Servlet没有返回状态码，默认会返回成功的状态码HttpServletResponse.SC\_OK。

HTTP头部(HTTP Header)：它们包含了更多关于响应的信息。比如：头部可以指定认为响应过期的过期日期，或者是指定用来给用户安全的传输实体内容的编码格式。

主体(Body)：它包含了响应的内容。它可以包含HTML代码，图片，等等。主体是由传输在HTTP消息中紧跟在头部后面的数据字节组成的。

**Switch的参数类型**

Java 7之前，switch 只能支持 byte、short、char、int或者其对应的封装类以及 Enum 类型。在 Java 7中，switch开始支持String

**java多态**

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的实现方式：接口实现，继承父类进行方法重写，同一个类中进行方法重载。

**当在浏览器地址栏输入一个URL后回车，将会发生的事情**

浏览器查找域名的IP地址

根据ip地址和url中的端口号建立一个socket连接，如果没有端口号则用默认端口号

建立连接之后向web服务器发送请求

Web服务器处理请求

返回http响应 浏览器显示页面

关闭连接

**Java堆内存与栈内存**

函数中定义的一些**基本类型的变量和对象的引用变量都是在函数的栈内存中分配**。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用

**堆内存用于存放由new创建的对象和数组**。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理

**获得一个类的类对象有哪些方式？**

方法1：类型.class，例如：String.class

方法2：对象.getClass()，例如："hello".getClass()

方法3：Class.forName()，例如：Class.forName("java.lang.String")

**Servlet线程安全问题**

Servlet默认是多线程的：编写线程安全的类，避免使用可以修改的类变量和实例变量

实例变量是在堆中分配的，不是线程安全的；

request，response是线程安全的

局部变量是在栈中分配的，是线程安全的

局部变量可作为参数传递

两个客户端访问同一个servlet的时候就会出现问题。

为了解决这个问题，可以使用如下的办法。

实现SingleThreadModel接口，这样就能保证只能有一个线程访问servlet。但是就不能够并发了，影响效率。

**尽量使用临时变量，因为临时变量是存储在栈中的，而每一个线程都有一个独立的栈，所以不会有线程安全的问题。**

request的属性也是线程安全的。

使用synchronized (this) {}代码块包住就可以了。

**Context，Session的线程安全**   
context，session类似，也会存在不安全的问题，需要加入synchronized(session)就可以了。不重写equals，那么比较的将是对象的引用是否指向同一块内存地址，重写之后目的是为了比较两个对象的value值是否相等

**Java内存模型**

Java线程之间的通信由Java内存模型（JMM）控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：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。

线程间通信的步骤：

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。

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。

JMM的编译器重排序规则会禁止特定类型的编译器重排序

**Jdbc java数据库连接**

两种建立数据库的方式：datasource(JNDI)机制与drivermanager机制

dataSource数据库连接池

Application Server启动时，创建通向数据库资源的连接池。连接池包含多个JDBC连接

当应用程序需要访问数据库时，从连接池中取得一个连接，使用该连接与数据库通信；一旦工作完成，关闭数据库连接，释放回连接池中

DataSource是JDBC Connection对象的一个工厂；允许使用已经在JNDI命名服务中注册的DataSource对象建立连接，由驱动程序供应商实现

**计算机网络之滑动窗口协议**

滑动窗口协议是传输层进行流控的一种措施，接收方通过通告发送方自己的窗口大小，从而控制发送方的发送速度，从而达到防止发送方发送速度过快而导致自己被淹没的目的。

**后退n协议**中，发送方在发完一个数据帧后，不停下来等待应答帧，而是连续发送若干个数据帧，即使在连续发送过程中收到了接收方发来的应答帧，也可以继续发送。且发送方在每发送完一个数据帧时都要设置**超时定时器**。只要在所设置的超时时间内仍未收到确认帧，就要重发相应的数据帧。如：当发送方发送了N个帧后，若发现该N帧的前一个帧在计时器超时后仍未返回其确认信息，则该帧被判为出错或丢失，此时发送方就不得不重新**发送出错帧及其后的N帧**。

**选择重传协议**

在后退n协议中，接收方若**发现错误帧就不再接收后续的帧**，即使是正确到达的帧，这显然是一种浪费。另一种效率更高的策略是当接收方发现某帧出错后，其后继续送来的正确的帧虽然**不能立即递交给接收方的高层，但接收方仍可收下来，存放在一个缓冲区中**，同时要求发送方**重新传送出错的那一帧。一旦收到重新传来的帧后，就可以原已存于缓冲区中的其余帧一并按正确的顺序递交高层**。这种方法称为选择重发(SELECTICE REPEAT)。显然，选择重发减少了浪费，但要求接收方有足够大的缓冲区空间。

TCP的特点之一是**提供体积可变的滑动窗口机制**，支持端到端的流量控制。TCP的窗口以字节为单位进行调整，以适应接收方的处理能力。处理过程如下：

TCP连接阶段，双方**协商窗口尺寸**，同时接收方预留数据缓存区；

发送方根据协商的结果，发送符合窗口尺寸的数据字节流，并等待对方的确认；

发送方根据确认信息，改变窗口的尺寸，增加或者减少发送未得到确认的字节流中的字节数。调整过程包括：**如果出现发送拥塞，发送窗口缩小为原来的一半，同时将超时重传的时间间隔扩大一倍**。

滑动窗口机制为端到端设备间的数据传输提供了可靠的流量控制机制。然而，它只能在源端设备和目的端设备起作用，当网络中间设备（例如路由器等）发生拥塞时，滑动窗口机制将不起作用。

**拥塞控制**

**慢启动**：当连接建立时，拥塞窗口被初始化为一个报文段大小，每收到一个ACK，拥塞窗口就会增加一个报文段，发送方取拥塞窗口与通过窗口的最小值作为发送的上限

**Spring事务传播属性**

REQUIRED ，这个是默认的属性

Support a current transaction, create a new one if none exists.

如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

被设置成这个级别时，会为**每一个被调用的方法创建一个逻辑事务域**。如果前面的方法已经创建了事务，那么后面的方法支持当前的事务，如果**当前没有事务会重新建立事务**。

**Hibernate配置**

配置文件默认为hibernate.cfg.xml；如果需要改变，则应在web.xml中配置context

<context-param>

<param-name>**contextConfigLocation**</param-name>

<param-value>/WEB-INF/classes/applicationContext.xml</param-value>

</context-param>

其中配置了数据源datasource，sessionFactory，

通过配置文件建立sessionFactory

Configuration config = new Configuration().configure();

//建立SessionFactory

SessionFactory sessionFactory = config.buildSessionFactory();

session=sessionFactory.openSession();

实体与数据库表的映射关系可以由映射文件也可以由注解方式来实现

简单查询Criteria query

查询所有

Criteria criteria = session.createCriteria(User.class);

List users=criteria.list();

查询符合条件的user

criteria.add(Expression.eq("userName","测试用户2"));

users=criteria.list();

**推荐的查询模式**

**Hibernate Query Language 面向对象的查询语言，查询以对象形式存在的数据**

String hql = "from edu.nju.hbn.User as us where us.userName like '测试用户2%'";

Query query=session.createQuery(hql);

//hql处理分页查询

query.setFirstResult(start);

query.setMaxResults(max);

List users=query.list();

Mysql分页查询：SELECT ... FROM ... WHERE ... ORDER BY ... LIMIT start,pagesize

Hql:1、属性查询2、参数查询、命名参数查询3、关联查询4、分页查询5、统计函数

**使用HibernateTemplete增删改查**

Dao实现类继承HibernateDaoSupport类,该类是抽象类,类中有一个HibernateTemplate的属性，通过this.getHibernateTemplate()可获得一个HibernateTemplate类的对象,

而且在hibernate.cfg.xml中需要配置hibernateTemplete的bean

find(hql语句) save(实体) update(实体) delete(实体)

**Hibernate中Session的get和load**

hibernate中Session接口提供的get()和load()方法都是用来获取一个实体对象，在使用方式和查询性能上有一些区别。

Session接口提供了4个重载的get方法，分别通过“持久类+主键”和“全类名+主键”以及“锁选项”来获取实体对象。

public Object get(Class clazz, Serializable id);

public Object get(Class clazz, Serializable id, LockOptions lockOptions);

public Object get(String entityName, Serializable id);

public Object get(String entityName, Serializable id, LockOptions lockOptions);

①get方法会在调用之后立即向数据库发出sql语句（不考虑缓存的情况下），返回持久化对象；而load方法会在调用后返回一个代理对象，该代理对象只保存了实体对象的id，直到使用对象的非主键属性时才会发出sql语句。

②查询数据库中不存在的数据时，get方法返回null，load方法抛出异常：org.hibernate.ObjectNotFoundException

get方法不使用延迟加载机制，首先查找Session缓存，然后查找二级缓存，然后查询数据库。load方法首先查找Session缓存，然后查找二级缓存，若找不到则返回代理对象，延迟到真正使用对象非主键属性时才发出sql语句加载对象。

若设置lazy="false"，load返回代理对象之后会发出sql语句，若找不到符合条件的记录，依然会抛出ObjectNotFoundException异常。

**Hibernate属性**

inverse属性是用来指定关联的控制方的。inverse属性默认是false，若为false，则关联由自己控制，若为true，则关联由对方控制。

在一对多关联中，在1的一方设置inverse="true",有助于性能的改善。(会减少update语句的数量)

将"inverse"属性设置为"false"，双方都维护关系(因为没有一方交出权力，"inverse"的默认值为"false"，而且"inverse"属性只能在set、list、map等几个标签中设置，像many-to-one这一类的标签都不能设置"inverse"这个属性值，它们只能取值"false"

Cascade属性：当关联双方存在父子关系，就可以在 set 处设定 cascade 为 all-delete-orphan所谓父子关系，即指由父方控制子方的持久化生命周期，子方对象必须和一个父方对象关联。如果删除父方对象，应该级联删除所有关联的子方对象；如果一个子方对象不再和一个父方对象关联，应该把这个子方对象删除。

cascade几种取值：

save-update:级联保存(load以后如果子对象发生了更新,也会级联更新)但它不会级联删除

delete:级联删除, 但不具备级联保存和更新

all-delete-orphan: 在解除父子关系时,自动删除不属于父对象的子对象, 也支持级联删除和级联保存更新.

all:级联删除, 级联更新,但解除父子关系时不会自动删除子对象.

delete-orphan:删除所有和当前对象解除关联关系的对象

none: 在保存，更新，删除当前对象时，忽略其关联的对象

级联保存和级联更新总是集合在一起的, 所以没单独的save 或 updata

**Java调用存储过程**

首先在数据库中创建一个存储过程create procedure prname

CallableStatement c=conn.prepareCall("{call prname(?)}");

//给存储过程的参数设置值

c.setInt(1,100);//将第一个参数的值设置成100

c.execute();//执行存储过程

c.getInt(1)//得到返回值

hibernate调用存储过程

Query query = session.createQuery("{call prname()}").addEntity

(entity.class);

或者Connection connection=session.getConnection(),然后使用与jdbc相同的方法

List list =query.list();

**Spring几种bean注入的方式**

构造器注入、设值注入(setter()方法)、注解方式注入、beanfactory注入

Resource rs = new FileSystemResource("beans-config.xml");

BeanFactory factory = new XmlBeanFactory(rs);

HelloBean helloBean = (HelloBean)factory.getBean("helloBean");

appliationContext=new ClassPathXmlApplicationContext("applicationContext.xml");

userService=(UserManageService)appliationContext.getBean("UserManageService");