1. 自我介绍
2. 讲一件你印象最深刻的事没事怎么解决的
3. JAVA垃圾回收机制怎么工作的
4. AOP是怎么实现的
5. TCP/UDP的区别是什么
6. MAP讲一下你对MAP的认识，然后谈一下HsahMap的性能，主要谈一下哈希冲撞
7. 创造线程的几种方式，Thread和Runnable接口有什么区别
8. 简单说一下你参加的中兴算法大赛
9. 说一下你做的聊天工具，如果遇到服务突然卡顿，怎么解决
10. 讲一下你对于spring框架的了解
11. Mysql数据库，索引是怎么使用的。
12. 你对分布式是否有所了解
13. 你对数据结构了解么
14. 你为什么想转行做软件呢
15. String 和StringBuffer
16. 浅析HashMap与ConcurrentHashMap的线程安全性
17. 堆排序，快速排序
18. 数据库的四大特征，数据库的隔离级别，
19. 线程同步机制用的什么？当初为什么选择这个机制？
20. 互斥和条件变量区别？互斥是阻塞的吗？当初为什么选择阻塞的机制？

21.消息队列

22.分布式

Java中多态性的实现

**什么是多态**

1. **面向对象的三大特性**：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。
2. **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）
3. **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。
4. **多态的作用**：消除类型之间的耦合关系。
5. **现实中，关于多态的例子不胜枚举**。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

下面是多态存在的三个必要条件，要求大家做梦时都能背出来！

多态存在的三个必要条件  
一、要有继承；  
二、要有重写；  
三、父类引用指向子类对象。

Java中多态的实现方式：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。

String与StringBuffer的区别  
简单地说，就是一个变量和常量的关系。StringBuffer对象的内容可以修改；而String对象一旦产生后就不可以被修改，重新赋值其实是两个对象。  
StringBuffer的内部实现方式和String不同，StringBuffer在进行字符串处理时，不生成新的对象，在内存使用上要优于String类。所以在实际使用时，如果经常需要对一个字符串进行修改，例如插入、删除等操作，使用StringBuffer要更加适合一些。  
String:在String类中没有用来改变已有字符串中的某个字符的方法，由于不能改变一个[**Java**](http://lib.csdn.net/base/java)字符串中的某个单独字符，所以在JDK文档中称String类的对象是不可改变的。然而，不可改变的字符串具有一个很大的优点:编译器可以把字符串设为共享的。   
StringBuffer:StringBuffer类属于一种辅助类，可预先分配指定长度的内存块建立一个字符串缓冲区。这样使用StringBuffer类的append方法追加字符 比 String使用 + 操作符添加字符 到 一个已经存在的字符串后面有效率得多。因为使用 + 操作符每一次将字符添加到一个字符串中去时，字符串对象都需要寻找一个新的内存空间来容纳更大的字符串，这无凝是一个非常消耗时间的操作。添加多个字符也就意味着要一次又一次的对字符串重新分配内存。使用StringBuffer类就避免了这个问题。  
StringBuffer是线程安全的，在多线程程序中也可以很方便的进行使用，但是程序的执行效率相对来说就要稍微慢一些。  
StringBuffer的常用方法  
StringBuffer类中的方法要偏重于对字符串的变化例如追加、插入和删除等，这个也是StringBuffer和String类的主要区别。

**String 字符串常量  
StringBuffer 字符串变量（线程安全）  
StringBuilder 字符串变量（非线程安全）**  
 简要的说， String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象, 因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后， JVM 的 GC 就会开始工作，那速度是一定会相当慢的。  
 而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。而在某些特别情况下， String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，而特别是以下的字符串对象生成中， String 效率是远要比 StringBuffer 快的：  
 String S1 = “This is only a” + “ simple” + “ test”;  
 StringBuffer Sb = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);  
 你会很惊讶的发现，生成 String S1 对象的速度简直太快了，而这个时候 StringBuffer 居然速度上根本一点都不占优势。其实这是 JVM 的一个把戏，在 JVM 眼里，这个  
 String S1 = “This is only a” + “ simple” + “test”; 其实就是：  
 String S1 = “This is only a simple test”; 所以当然不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的 String 对象的话，速度就没那么快了，譬如：  
String S2 = “This is only a”;  
String S3 = “ simple”;  
String S4 = “ test”;  
String S1 = S2 +S3 + S4;  
这时候 JVM 会规规矩矩的按照原来的方式去做

在大部分情况下 StringBuffer > String  
**StringBuffer**  
Java.lang.StringBuffer线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。  
可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。  
StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。  
例如，如果 z 引用一个当前内容是“start”的字符串缓冲区对象，则此方法调用 z.append("le") 会使字符串缓冲区包含“startle”，而 z.insert(4, "le") 将更改字符串缓冲区，使之包含“starlet”。  
在大部分情况下 StringBuilder > StringBuffer

**java.lang.StringBuilde**  
java.lang.StringBuilder一个可变的字符序列是5.0新增的。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。两者的方法基本相同。

在数据结构中有一种称为哈希表的数据结构，它实际上是数组的推广。如果有一个数组，要最有效的查找某个元素的位置，如果存储空间足够大，那么可以对每个元素和内存中的某个地址对应起来，然后把每个元素的地址用一个数组（这个数组也称为哈希表）存储起来，然后通过数组下标就可以直接找到某个元素了。这种方法术语叫做直接寻址法。这种方法的关键是要把每个元素和某个地址对应起来，所以如果当一组数据的取值范围很大的时候，而地址的空间又有限，那么必然会有多个映射到同一个地址，术语上称为哈希冲突，这时映射到同一个地址的元素称为同义词。毕竟，存储空间有限，所以冲突是不可避免的，但是可以尽量做到减少冲突。目前有两种比较有效的方法来解决哈希冲突：

链地址法

开放地址法

这里简要说明一下开放地址法，顾名思义，就是哈希表中的每个位置要么存储了一个元素要么为NULL。当数据比较多的时候，查找一个元素挺费事的，但是可以使用探测的方法进行查找。这个话题与本主题关系不大，感兴趣的小伙伴可以自行研究。

从这段注释中可以发现每次执行ConcurrentHashMap的put方法都是调用s.put()方法的，而Segments对象是一个继承了ReentrantLock锁对象的子类，那么剩下的就很清晰了，每一个Segments都有一个锁，只有执行完上面try语句块中的代码才会释放锁，从而保证了多线程并发访问的安全性。

**什么是线程同步？**  
当使用多个线程来访问同一个数据时，非常容易出现线程安全问题(比如多个线程都在操作同一数据导致数据不一致),所以我们用同步机制来解决这些问题。

实现同步机制有两个方法：  
**1.同步代码块：**  
**synchronized(同一个数据){} 同一个数据：就是N条线程同时访问一个数据。**

**2.**同步方法：

**public synchronized 数据返回类型 方法名(){}**  
**就是使用 synchronized 来修饰某个方法，则该方法称为同步方法。对于同步方法而言，无需显示指定同步监视器，同步方法的同步监视器是 this 也就是该对象的本身（这里指的对象本身有点含糊，其实就是调用该同步方法的对象）通过使用同步方法，可非常方便的将某类变成线程安全的类**，具有如下特征：  
1，该类的对象可以被多个线程安全的访问。  
2，每个线程调用该对象的任意方法之后，都将得到正确的结果。  
3，每个线程调用该对象的任意方法之后，该对象状态依然保持合理状态。  
注：synchronized关键字可以修饰方法，也可以修饰代码块，但不能修饰构造器，属性等。

实现同步机制注意以下几点： 安全性高，性能低，在多线程用。性能高，安全性低，在单线程用。  
1，不要对线程安全类的所有方法都进行同步，只对那些会改变共享资源方法的进行同步。  
2，如果可变类有两种运行环境，当线程环境和多线程环境则应该为该可变类提供两种版本：线程安全版本和线程不安全版本(没有同步方法和同步块)。在单线程中环境中，使用线程不安全版本以保证性能，在多线程中使用线程安全版本.

一：进程与线程

概述：几乎任何的操作系统都支持运行多个任务，通常一个任务就是一个程序，而一个程序就是一个进程。当一个进程运行时，内部可能包括多个顺序执行流，每个顺序执行流就是一个线程。

进程：进程是指处于运行过程中的程序，并且具有一定的独立功能。进程是系统进行资源分配和调度的一个单位。当程序进入内存运行时，即为进程。

进程的三个特点：

1：独立性：进程是系统中独立存在的实体，它可以独立拥有资源，每一个进程都有自己独立的地址空间，没有进程本身的运行，用户进程不可以直接访问其他进程的地址空间。

2：动态性：进程和程序的区别在于进程是动态的，进程中有时间的概念，进程具有自己的生命周期和各种不同的状态。

3：并发性：多个进程可以在单个处理器上并发执行，互不影响。

并发性和并行性是不同的概念：并行是指同一时刻，多个命令在多个处理器上同时执行；并发是指在同一时刻，只有一条命令是在处理器上执行的，但多个进程命令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果

线程：

线程是进程的组成部分，一个进程可以拥有多个线程，而一个线程必须拥有一个父进程。线程可以拥有自己的堆栈，自己的程序计数器和自己的局部变量，但不能拥有系统资源。它与父进程的其他线程共享该进程的所有资源。

线程的特点：

线程可以完成一定任务，可以和其它线程共享父进程的共享变量和部分环境，相互协作来完成任务。

线程是独立运行的，其不知道进程中是否还有其他线程存在。

线程的执行是抢占式的，也就是说，当前执行的线程随时可能被挂起，以便运行另一个线程。

一个线程可以创建或撤销另一个线程，一个进程中的多个线程可以并发执行。

1.进程的三种基本状态

进程在运行中不断地改变其运行状态。通常，一个运行进程必须具有以下三种基本状态。

（1）就绪(Ready)状态

当进程已分配到除CPU以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态。

（2）执行（Running）状态当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为执行状态。

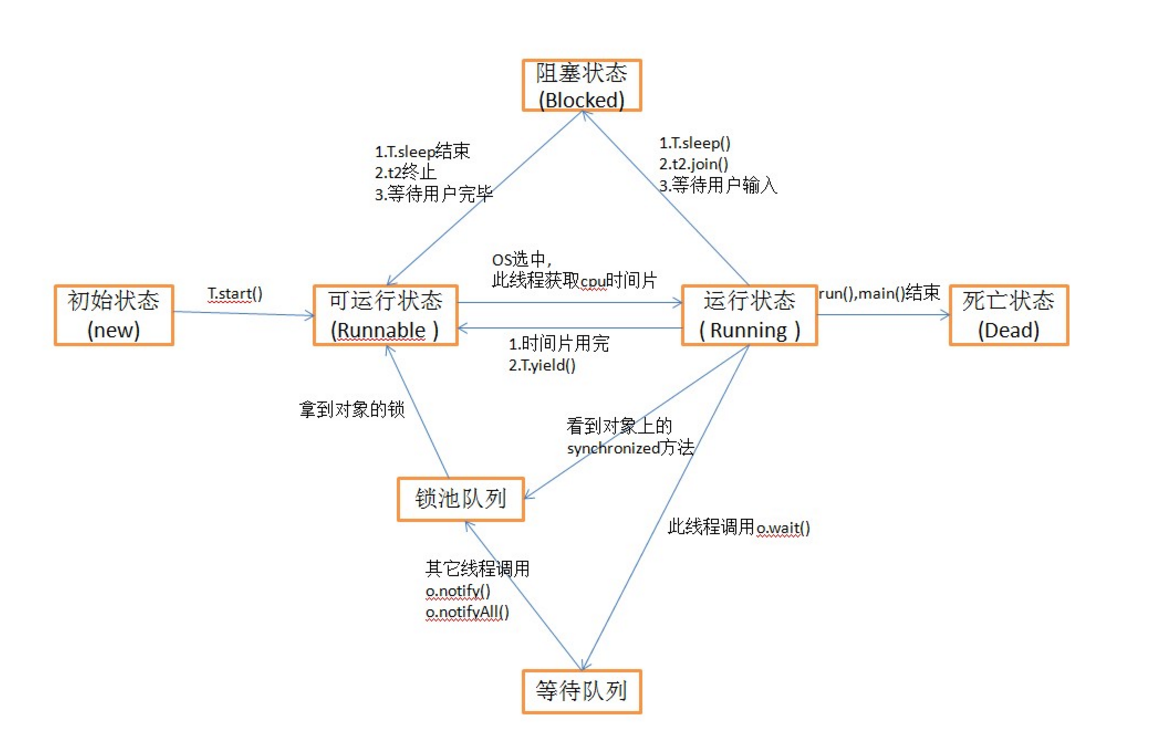
（3）阻塞(Blocked)状态正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态。引起进程阻塞的事件可有多种，例如，等待I/O完成、申请缓冲区不能满足、等待信件(信号)等。

阻塞的情况分三种：

(1)、等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，

(2)、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。

(3)、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。



同步机制：

1、临界区:通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

2、互斥量:为协调共同对一个共享资源的单独访问而设计的。

3、信号量:为控制一个具有有限数量用户资源而设计。

4、事 件:用来通知线程有一些事件已发生，从而启动后继任务的开始。

临界区（Critical Section）

保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

互斥量（Mutex）

互斥量跟临界区很相似，只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下此共享资源都不会同时被多个线程所访问。当前占据资源的线程在任务处理完后应将拥有的互斥对象交出，以便其他线程在获得后得以访问资源。互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。

信号量（Semaphores）

信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。在用CreateSemaphore（）创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1，只要当前可用资源计数是大于0的，就可以发出信号量信号。但是当前可用计数减小到0时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过ReleaseSemaphore（）函数将当前可用资源计数加1。在任何时候当前可用资源计数决不可能大于最大资源计数。

事件（Event）

事件对象也可以通过通知操作的方式来保持线程的同步。并且可以实现不同进程中的线程同步操作。

总结：

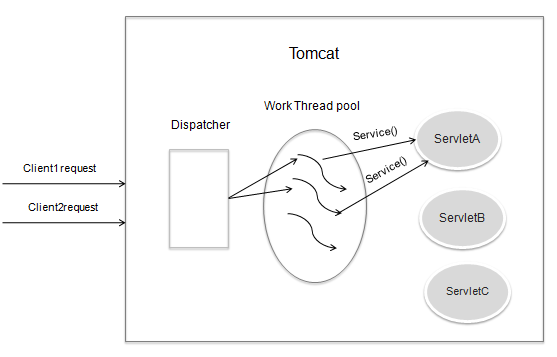
1． 互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。

2． 互斥量（Mutex），信号灯（Semaphore），事件（Event）都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用WaitForSingleObject来等待进程和线程退出。

3． 通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过互斥量就无法处理，比如现在一位用户购买了一份三个并发访问许可的数据库系统，可以根据用户购买的访问许可数量来决定有多少个线程/进程能同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求，信号灯对象可以说是一种资源计数器。

在上一篇关于Serlvet框架和Servlet生命周期的学习中，我们已经知道了在多线程的情况下Servlet是线程不安全的。Servlet体系是建立在[Java](http://lib.csdn.net/base/java)多线程的基础之上的,它的生命周期是由Tomcat来维护的。当客户端第一次请求Servlet的时候,tomcat会根据web.xml配置文件实例化servlet，当又有一个客户端访问该servlet的时候，不会再实例化该servlet，也就是多个线程在使用这个实例。**Servlet线程池**serlvet采用多线程来处理多个请求同时访问，Tomcat容器维护了一个线程池来服务请求。

线程池实际上是等待执行代码的一组线程叫做工作组线程(Worker Thread)，Tomcat容器使用一个调度线程来管理工作组线程(Dispatcher Thead)。



  当容器收到一个Servlet请求，Dispatcher线程从线程池中选出一个工作组线程，将请求传递给该线程，然后由该线程来执行Servlet的service方法。当这个线程正在执行的时候，容器收到另一个请求，调度者线程将从线程池中选出另外一个工作组线程来服务则个新的请求，容器并不关心这个请求是否访问的是同一个Servlet还是另一个Servlet。当容器收到对同一个Servlet的多个请求的时候，那这个servlet的service方法将在多线程中并发的执行。

在浏览器中输入网址访问资源都是通过GET方式;在FORM提交中，可以通过Method指定提交方式为GET或者POST，默认为GET提交。

HTTP 定义了与服务器交互的不同方法，最常用的有4种，Put(增),Delete(删)，Post(改),Get(查)，即增删改查：

1)Get，

它用于获取信息，注意，他只是获取、查询数据，也就是说它不会修改服务器上的数据，从这点来讲，它是数据安全的，而稍后会提到的Post它是可以修改数据的，所以这也是两者差别之一了。

2)

Post，它是可以向服务器发送修改请求，从而修改服务器的，比方说，我们要在论坛上回贴、在博客上评论，这就要用到Post了，当然它也是可以仅仅获取数据的。

3)Delete 删除数据。可以通过Get/Post来实现。

4)Put，增加、放置数据，可以通过Get/Post来实现。

根据HTTP规范，GET用于信息获取，而且应该是安全的和幂等的 。

1.所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET请求一般不应产生副作用。就是说，仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。(注意：这里安全的含义仅仅是指是非修改信息。)

根据HTTP规范，POST表示可能修改变服务器上的资源的请求

。继续引用上面的例子：还是新闻以网站为例，读者对新闻发表自己的评论应该通过POST实现，因为在评论提交后站点的资源已经不同了，或者说资源被修改了。

　 表现形式区别：

HTTP请求：在HTTP请求中，第一行必须是一个请求行(request

line)，用来说明请求类型、要访问的资源以及使用的HTTP版本。紧接着是一个首部(header)小节，用来说明服务器要使用的附加信息。在首部之后是一个空行，再此之后可以添加任意的其他数据[称之为主体(body)]。

两种提交方式的区别：

(1)GET提交，请求的数据会附在URL之后(就是把数据放置在HTTP协议头中)，以?分割URL和传输数据，多个参数用&连接。如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如：

%E4%BD%A0%E5%A5%BD，其中%XX中的XX为该符号以16进制表示的ASCII。

POST提交：把提交的数据放置在是HTTP包的包体中。上文示例中红色字体标明的就是实际的传输数据

因此，GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变

(2)传输数据的大小：首先声明：HTTP协议没有对传输的数据大小进行限制，HTTP协议规范也没有对URL长度进行限制。

而在实际开发中存在的限制主要有：

GET:特定浏览器和服务器对URL长度有限制，例如IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。

因此对于GET提交时，传输数据就会受到URL长度的限制。

POST:由于不是通过URL传值，理论上数据不受限。但实际各个WEB服务器会规定对post提交数据大小进行限制，Apache、IIS6都有各自的配置。

伴随网站业务规模和访问量的逐步发展，原本由单台服务器、单个域名的迷你网站架构已经无法满足发展需要。

此时我们可能会购买更多服务器，并且启用多个二级子域名以频道化的方式，根据业务功能将网站分布部署在独立的服务器上；或通过负载均衡技术（如：DNS轮询、Radware、F5、LVS等）让多个频道共享一组服务器。

OK，头脑中我们已经构思了这样的解决方案，不过进入深入开发后新的技术问题又随之而来：

我们把网站程序分布部署到多台服务器上，而且独立为几个二级域名，由于Session受实现原理的局限（PHP中Session默认以文件的形式保存在本地服务器的硬盘），使得我们的网站用户不得不经常在几个频道间来回输入用户名、密码登入，导致用户体验大打折扣；另外，原本程序可以直接从用户Session变量中读取的资料（如：昵称、积分、登入时间等），因为无法跨服务器同步更新Session 变量，迫使开发人员必须实时读写数据库，从而增加了数据库的负担。

于是，解决网站跨服务器之间的Session共享方案需求变得迫切起来，最终催生了多种解决方案，下面列举4种较为可行的方案进行对比探讨：

1. 基于NFS的Session共享

NFS是Net FileSystem的简称，最早由Sun公司为解决Unix网络主机间的目录共享而研发。

这个方案实现最为简单，无需做过多的二次开发，仅需将共享目录服务器mount到各频道服务器的本地session目录即可，缺点是NFS依托于复杂的安全机制和文件系统，因此并发效率不高，尤其对于session这类高并发读写的小文件，会由于共享目录服务器的io-wait过高，最终拖累前端WEB应用程序的执行效率。

2. 基于数据库的Session共享

首选当然是大名鼎鼎的MySQL数据库，并且建议使用内存表Heap，提高session操作的读写效率。这个方案的实用性比较强，相信大家普遍在使用，它的缺点在于session的并发读写能力取决于mysql数据库的性能，同时需要自己实现session淘汰逻辑，以便定时从数据表中更新、删除 session记录，当并发过高时容易出现表锁，虽然我们可以选择行级锁的表引擎，但不得不否认使用数据库存储Session还是有些杀鸡用牛刀的架势。

3. 基于Cookie的Session共享

这个方案我们可能比较陌生，但它在大型网站中还是比较普遍被使用。原理是将全站用户的Session信息加密、序列化后以Cookie的方式，统一种植在根域名下（如：.host.com），利用浏览器访问该根域名下的所有二级域名站点时，会传递与之域名对应的所有Cookie内容的特性，从而实现用户的Cookie化Session 在多服务间的共享访问。

这个方案的优点无需额外的服务器资源；缺点是由于受http协议头信心长度的限制，仅能够存储小部分的用户信息，同时Cookie化的 Session内容需要进行安全加解密（如：采用DES、RSA等进行明文加解密；再由MD5、SHA-1等算法进行防伪认证），另外它也会占用一定的带宽资源，因为浏览器会在请求当前域名下任何资源时将本地Cookie附加在http头中传递到服务器。

4. 基于Memcache的Session共享

Memcache由于是一款基于Libevent多路异步I/O技术的内存共享系统，简单的Key + Value数据存储模式使得代码逻辑小巧高效，因此在并发处理能力上占据了绝对优势，目前本人所经历的项目达到2000/秒 平均查询，并且服务器CPU消耗依然不到10%。

另外值得一提的是Memcache的内存hash表所特有的Expires数据过期淘汰机制，正好和Session的过期机制不谋而合，降低了过期Session数据删除的代码复杂度，对比“基于数据库的存储方案”，仅这块逻辑就给数据表产生巨大的查询压力。

内存溢出

http://blog.csdn.net/ylyg050518/article/details/52354879

  内存溢出：OOM（OutOfMemoryError）异常，即程序需要内存超出了虚拟机可以分配内存的最大范围。在Java 虚拟机规范的描述中，除了程序计数器外，虚拟机内存的其他区域都可能发生异常。

内存溢出区域

常见的内存溢出分为以下几种：

1. Java 堆溢出

Java 堆用于存储对象实例，只要不断地创建对象，并且保证垃圾回收机制清除这些对象，那么在对象数量达到最大堆限制就会产生内存溢出异常。

测试方案：无限循环new对象实例出来，在List中保存引用，防止GC回收，最终会产生OOM ,异常堆栈信息并提示Java heap space。

2. 虚拟机栈和本地方法栈溢出

关于虚拟机栈和本地方法栈，Java虚拟机规范中定义了两种异常：

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError 异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。

测试方案：

单线程条件下，通过不断递归调用方法，如不断累加的方法，如下所示7

最终会产生StackOverflowError栈溢出异常；

多线程条件下，无限循环地创建线程，并为每个线程无限循环的增加内存，最终会导致OutOfMemoryError异常。

这里有一点要重点说明，在多线程情况下，给每个线程的栈分配的内存越大，反而越容易产生内存产生内存溢出一场。操作系统为每个进程分配的内存是有限制的，虚拟机提供了参数来控制Java堆和方法区这两部分内存的最大值，忽略掉程序计数器消耗的内存（很小），以及进程本身消耗的内存，剩下的内存便给了虚拟机栈和本地方法栈，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少。因此，如果是建立过多的线程导致的内存溢出，在不能减少线程数的情况下，就只能通过减少最大堆和每个线程的栈容量来换取更多的线程。

3. 方法区和运行时常量池溢出

运行时常量池是方法区的一部分。方法区用于存放Class的相关信息，如类名，访问修饰符，常量池，字段描述，方法描述等。

测试方法：

1.对于非常量池部分，运行时生成大量的动态类填满方法区;

2.对于常量池部分，无限循环调用String的intern()方法产生不同的String对象实例，并在List中保存其引用，以防止被GC回收，最终会产生溢出。

4. 本机直接内存溢出

  此类内存溢出一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果发现OOM之后Dump文件很小，而程序中又直接或间接使用了NIO,可以考虑一下是不是这方面原因。

内存泄露

  内存泄漏是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成内存空间的浪费称为内存泄漏。内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你OOM。

Java内存泄漏的根本原因是长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收。

内存泄露场景

具体主要有如下几大类：

1. 静态集合类引起内存泄漏

  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。

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。

2.集合里面的对象属性被修改，再调用remove()方法不生效

3. 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。

4. 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。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。

5. 内部类和外部模块的引用

  内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如： public void registerMsg(Object b); 这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B 是否提供相应的操作去除引用。

6. 单例模式

  不正确使用单例模式是引起内存泄漏的一个常见问题，单例对象在初始化后将在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的引用，那么这个对象将不能被JVM正常回收，导致内存泄漏，考虑下面的例子：

显然B采用singleton模式，它持有一个A对象的引用，而这个A类的对象将不能被回收。想象下如果A是个比较复杂的对象或者集合类型会发生什么情况。

XML文件加载机制：<https://my.oschina.net/zmf/blog/397468>

Session 共享的方式：<http://blog.csdn.net/jimmy1980/article/details/4973021>

Session缓存原理:   
Session缓存：在[hibernate](http://lib.csdn.net/base/javaee)中被称为一级缓存。   
原理：

1. 当应用程序调用Session的CRUD方法、以及调用查询接口的list()、iterate()或filter()方法时，如果在Session缓存中还不存在相应的对象，Hibernate就会把该对象加入到第一级缓存中
2. 当清理缓存时，Hibernate会根据缓存中对象的状态变化来同步更新数据库

Session缓存的作用。   
1.减少访问[数据库](http://lib.csdn.net/base/mysql)的频率   
2.保证缓存中的对象与数据库中的数据同步   
3.当缓存中的持久化对象之间存在循环关联关系时，Session会保证不4.出现访问对象图的死循环，以及由死循环引起的JVM堆栈溢出异常

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

**lock与Syntronized的区别**

转自自：

java并发之Lock与synchronized的区别

1）Lock是一个接口，而synchronized是[Java](http://lib.csdn.net/base/java)中的关键字，synchronized是内置的语言实现；

　　2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

　　3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

　　4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

　　5）Lock可以提高多个线程进行读操作的效率。

　　在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

JVN内存区域以及GC机制 ：<http://www.cnblogs.com/zhguang/p/3257367.html>

类的加载：http://blog.csdn.net/ns\_code/article/details/17881581

**线程间通信：**

**window平台：**

（1）全局变量volatile

（2）事件(Event);

（3）窗口消息实现通信

（4）信号量 二进制信号量 互斥信号量 整数型信号量 记录型信号量

三个区别：

1、接收的参数不一样

2、submit有返回值，而execute没有

Method submit extends base method Executor.execute by creating and returning a Future that can be used to cancel execution and/or wait for completion.

用到返回值的例子，比如说我有很多个做validation的task，我希望所有的task执行完，然后每个task告诉我它的执行结果，是成功还是失败，如果是失败，原因是什么。然后我就可以把所有失败的原因综合起来发给调用者。

个人觉得cancel execution这个用处不大，很少有需要去取消执行的。

而最大的用处应该是第二点。

3、submit方便Exception处理

There is a difference when looking at exception handling. If your tasks throws an exception and if it was submitted with execute this exception will [Go](http://lib.csdn.net/base/go) to the uncaught exception handler (when you don't have provided one explicitly, the default one will just print the stack trace to System.err). If you submitted the task with submit **any** thrown exception, checked or not, is then part of the task's return status. For a task that was submitted with submit and that terminates with an exception, the Future.get will rethrow this exception, wrapped in an ExecutionException.

意思就是如果你在你的task里会抛出checked或者unchecked exception，而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

[Synchronized和Static Synchronized区别](http://blog.csdn.net/liovey/article/details/7456096)

通过分析这两个用法的分析，我们可以理解[Java](http://lib.csdn.net/base/java)中锁的概念。一个是实例锁（锁在某一个实例对象上，如果该类是单例，那么该锁也具有全局锁的概念），一个是全局锁（该锁针对的是类，无论实例多少个对象，那么线程都共享该锁）。实例锁对应的就是synchronized关键字，而类锁（全局锁）对应的就是static synchronized（或者是锁在该类的class或者classloader对象上）。下面的文章做了很好的总结：

1.synchronized与static synchronized 的区别  
       synchronized是对类的**当前实例（当前对象）**进行加锁，防止其他线程同时访问该类的该实例的所有synchronized块，注意这里是“类的当前实例”， 类的两个不同实例就没有这种约束了。

那么static synchronized恰好就是要控制类的所有实例的并发访问，static synchronized是限制多线程中该类的所有实例同时访问jvm中该类所对应的代码块。实际上，在类中如果某方法或某代码块中有 synchronized，那么在生成一个该类实例后，该实例也就有一个监视块，防止线程并发访问该实例的synchronized保护块，而static synchronized则是所有该类的所有实例公用得一个监视块，这就是他们两个的区别。也就是说synchronized相当于 this.synchronized，而static synchronized相当于Something.synchronized.（后面又讲解）