# **海涯面试大全**

## 一、通用问题

### 1自我介绍？

你好，名字，户籍，学校，工作经验和经历，职业规划，收尾

### 2什么时候可以到岗？住在哪里？上班是否方便？

随时，龙华，方便

### 3怎么看加班？

项目进度紧急，上线，有紧急线上问题的时候可以接受加班

### 4平时工作中遇到什么困难，怎么解决的？

@Activiti异步task，去除异步选项

@报表缓慢，查询中间表并且同步

@同步axis接口超时连接中断，更换方案

### 5为什么离职？

个人原因

### 6做过什么自认为有挑战性的工作？

@切换工作流到activiti, 整个电商系统完全重构，所有场景需要回归测试

@三条业务线并行研发，基本能够准时交付

@elk收视分析，客服困难

### 7说下自己的工作中的优势和缺点？

@优势，积极主动，工作效率高，态度好，有责任心

@缺点，事情比较集中有点急躁

### 8 经常溜达的IT学习网站是？

CSDN; 博客园; Github; 码云 ；微信公众号

### 9最近一个项目的前后端技术栈是什么？技术框架的版本号？

Spring4.1,mybatis3.2,activiti5.21

### 10你在最近一个项目中担任什么角色，具体的工作是什么？

部门经理，三个项目的整体进度把控，跟进转测问题，解决生产问题，需求分解

### 11最近一个项目构建和部署方式是什么？

Jenkins+maven+git

### 12开发模式和开发流程？有没有设计阶段？用什么工具设计？画过什么图？

敏捷开发，需求分析--概要设计--编码自测试---测试---预发布---生产环境

有，visio和powerdesigner; 类图，状态图等

### 13有没有采用分布式架构？为什么？

采用了，业务量比较大，按照订单，框架，流程，组件，交付 进行垂直分割，分割后每个模块有独立的集群环部署环境，独立的数据库。

### 14项目开发中哪些场景采用设计模式，用了哪些？

适配器模式：activiti监听器

单例模式：activiti事件逻辑

代理模式：aop记录日志

命令模式：流程监控功能

### 15项目的单点登录怎么实现的？

CAS服务器

### 16项目开发中有没有性能优化的例子？采用什么方法解决什么问题？

有，我的任务性能优化，用sql语句取代后台逻辑，简化设计，去除繁冗的逻辑

### 17项目怎么保证接口安全问题？

Ip白名单，签名机制和数字证书

### 18有集群部署吗？怎么解决多台服务器共享session问题？

有，CAS或者Redis

### 19 介绍下最有代表性的项目的整体情况？

Ideal电商切换底层流程引擎

### 20 介绍下最近一个参与项目的整体情况？

聚引客

## 二、初中级开发

### 1 说下java中内存泄漏的场景 ？

Java的一个重要特性就是通过垃圾收集器(GC)自动管理内存的回收，而不需要程序员自己来释放内存。理论上Java中所有不会再被利用的对象所占用的内存，都可以被GC回收，但是Java也存在内存泄露，但它的表现与C++不同。

释放对象的根本原则就是对象不会再被使用：

给对象赋予了空值null，之后再没有调用过。

另一个是给对象赋予了新值，这样重新分配了内存空间。

不再会被使用的对象的内存不能被回收，就是内存泄露。

对象都是有生命周期的，有的长，有的短，如果长生命周期的对象持有短生命周期的引用，就很可能会出现内存泄露

在堆中的分配的内存，在没有将其释放掉的时候，就将所有能访问这块内存的方式都删掉（如指针重新赋值），这是针对c++等语言的，Java中的GC会帮我们处理这种情况，所以我们无需关心。

在内存对象明明已经不需要的时候，还仍然保留着这块内存和它的访问方式（引用），这是所有语言都有可能会出现的内存泄漏方式。编程时如果不小心，我们很容易发生这种情况，如果不太严重，可能就只是短暂的内存泄露。

解决的原则就是尽量减小对象的作用域（将类的成员变量改写为方法内的局部变量）以及手动设置null值。

例子1：

Java容器ArrayList是数组实现的（可参考：Java之ArrayList源码解读（JDK 1.8）），如果我们要为其写一个pop()（弹出）方法，可能会是这样：:

public E pop(){

if(size == 0)

return null;

else

return (E) elementData[--size];

}

写法很简洁，但这里却会造成内存溢出：elementData[size-1]依然持有E类型对象的引用，并且暂时不能被GC回收。我们可以如下修改：

public E pop(){

if(size == 0)

return null;

else{

E e = (E) elementData[--size];

elementData[size] = null;

return e;

}

}

http://blog.csdn.net/anxpp/article/details/51325838

例子2：

void method(){

Vector vector = new Vector();

for (int i = 1; i<100; i++)

{

Object object = new Object();

vector.add(object);

object = null;

}

//...对v的操作

vector = null;

//...与v无关的其他操作

}

例子3：

比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，以及使用其他框架的时候，除非其显式的调用了其close()方法（或类似方法）将其连接关闭，否则是不会自动被GC回收的。

其实原因依然是长生命周期对象持有短生命周期对象的引用。

可能很多人使用过Hibernate，我们操作数据库时，通过SessionFactory获取一个session：

Session session=sessionFactory.openSession();

完成后我们必须调用close()方法关闭：

session.close();

SessionFactory就是一个长生命周期的对象，而session相对是个短生命周期的对象，但是框架这么设计是合理的：它并不清楚我们要使用session到多久，于是只能提供一个方法让我们自己决定何时不再使用。

因为在close()方法调用之前，可能会抛出异常而导致方法不能被调用，我们通常使用try语言，然后再finally语句中执行close()等清理工作：

try{

session=sessionFactory.openSession();

//...其他操作

}finally{

session.close();

}

例子4：单例模式，很多时候我们可以把它的生命周期与整个程序的生命周期看做差不多的，所以是一个长生命周期的对象。如果这个对象持有其他对象的引用，也很容易发生内存泄露。

gc()对于程序员来说，GC基本是透明的，不可见的。运行GC的函数是System.gc()，调用后启动垃圾回收器开始清理。但是根据Java语言规范定义， 该函数不保证JVM的垃圾收集器一定会执行。因为，不同的JVM实现者可能使用不同的算法管理GC。通常，GC的线程的优先级别较低。

finalize()，finalize()是Object类中的方法。当在java中调用非java代码（如c和c++）时，在这些非java代码中可能会用到相应的申请内存的操作（如c的malloc()函数），而在这些非java代码中并没有有效的释放这些内存，就可以使用finalize()方法，并在里面调用本地方法的free()等函数。

所以finalize()并不适合用作普通的清理工作。

内存泄露问题，还是编码不认真导致的，我们并不能责怪JVM没有更合理的清理。

### 2 hashmap和hashtable什么区别？

Hashmap：线程不安全,允许有null的键和值,效率高,方法不是Synchronize的要提供外同步

Hashtable：线程安全,不允许有null的键和值,效率稍低,方法是是Synchronize的

### 3单例模式有什么优缺点？枚举实现单例好处？

【采用原因】

对于系统中的某些类来说，只有一个实例很重要，

例如，一个系统中可以存在多个打印任务，但是只能有一个正在工作的任务；一个系统只能有一个窗口管理器或文件系统；一个系统只能有一个计时工具或ID(序号)生成器。

如在Windows中就只能打开一个任务管理器。如果不使用机制对窗口对象进行唯一化，将弹出多个窗口，如果这些窗口显示的内容完全一致，则是重复对象，浪费内存资源；

比如说一些大对象，如果能够不断初始化实例，那么内存消耗会不断变大。

【单例优点】

1、提供了对唯一实例的受控访问，保证了重要交易的准确性。

2、由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象单例模式无疑可以提高系统的性能。

【单例缺点】

1、在多线程情况下为了线程安全 同步可能会影响性能

2、单例类的职责过重，在一定程度上违背了单一职责原则。

3、滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

【 枚举单例】

枚举单例是java中使用枚举提供一个实例对象来实现单例模式的一种新方法，虽然单例模式在java中早已存在，但枚举单例实际上从java5引入枚举作为它的关键特性之后相对来说还是一个新的概念

1）枚举单例模式代码简洁

public class Singleton{

private static final Singleton INSTANCE = new Singleton();

private Singleton(){}

public static Singleton getSingleton(){

return INSTANCE;

}

}

2）枚举单例可以自己处理序列化，传统的单例模式的另外一个问题是一旦你实现了serializable接口，他们就不再是单例的了，因为readObject()方法总是返回一个 新的实例对象，就像java中的构造器一样。你可以使用readResolve()方法来避免这种情况

3）枚举单例是线程安全的，由于枚举实例的创建默认就是线程安全的，你不需要担心双检锁问题。

### 4静态代理和动态代理区别？

动态代理做到了开闭原则，静态代理没有做到对修改开放

### 5策略模式和模板方法模式解决什么问题？

模板方法模式的主要思想：定义一个算法流程，将一些特定步骤的具体实现、延迟到子类。使得可以在不改变算法流程的情况下，通过不同的子类、来实现“定制”流程中的特定的步骤。

策略模式的主要思想：使不同的算法可以被相互替换，而不影响客户端的使用。

在思想和意图上看，模板方法更加强调：

1）定义一条线（算法流程），线上的多个点是可以变化的（具体实现在子类中完成），线上的多个点一定是会被执行的，并且一定是按照特定流程被执行的。

2）算法流程只有唯一的入口，对于点的访问是受限的【通常用受保护的虚函数来定义可变点】。

策略模式更注重于： 一个“策略”是一个 整体的(完整的) 算法，算法是可以被整体替换的。而模板方法只能被替换其中的特定点，算法流程是固定不可变的。

在这样的细节上看来，模板方法 和 一组策略模式 是不可以划等号的。

比如说定义一个业务抽象类，有一个execute方法，包括 initData,doValidate,doBusiness,doAfterBusiness等方法，那么 execute就是一个模板方法，由不同的子类继承，可以走不同的路线，可以去除doValidate但是整体顺序是不变的

子类也可以完全重写doAfterBusiness是策略模式。

### 6 linux命令

包含XXX名字的进程；ps -ef|grep nginx

包含XXX名字的文件；find /home -name \*.log

包含XXX名字的日志；find /home -type f -name \*erp\*.log

监控日志最后20行；tail -f -n 1000 ops-web-erp.log | perl -pe 's/(22)/\e[1;31m$1\e[0m/g'

切换用户；su -ddd

退出编辑文本；:q!

授权777；chmod dd 777

CPU 内存监控情况 ；top free

### 7类的加载器有哪几种？分别加载什么类？

（1） 根类加载器(Bootstrap) 它负责加载虚拟机的核心类库，如java.lang.\*等

使用C++编写，它的实现依赖于底层操作系统，它并没有继承java.lang.ClassLoader类

（2） 扩展类加载器(Extension) 它的父加载器为根类加载器，它从jre\lib\ext子目录下加载类库，它使用Java代码实现，是java.lang.ClassLoader类的子类。

（3） 系统(应用)类加载器(System\App) 它的父加载器为扩展类加载器，它从环境变量classpath中加载类，它是用户自定义的类加载器的默认父加载器。它使用java实现，是java.lang.ClassLoader类的子类。

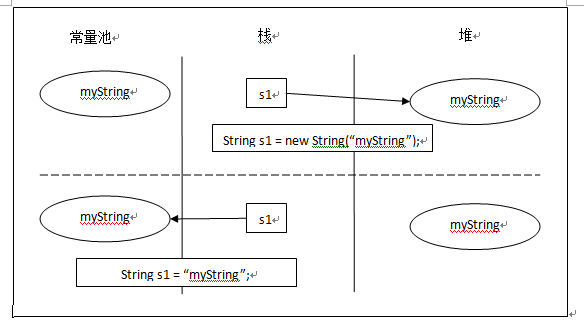
（4）自定义加载器(必须继承ClassLoader)

java的类加载器 ，采用父委托机制,类加载机制是为了保障jvm的安全，先去父类找，父类没有才去子类找，这样就避免了系统自身的方法不被覆盖。如果你试图写一个lang的类去覆盖java.lang，因为类的加载机制，所以不可能成功。

父亲委托机制的优点是能够提高软件系统的安全性。因为在此机制下，用户自定义的类加载器不可能加载应该由父加载器加载的可靠类，从而防止不可靠甚至恶意的代码代替由父加载器加载的可靠代码。

例如，java.lang.Object类总是由根类加载器加载，其他任何用户自定义的类加载器都不可能加载含有恶意代码的java.lang.Object类。

### 8讲下字符串的jvm内存分配,堆和栈是什么？



（1）使用关键字new，如：String s1 = new String(“myString”);

第一种方式通过关键字new定义过程：在程序编译期，编译程序先去字符串常量池检查，是否存在“myString”,如果不存在，

则在常量池中开辟一个内存空间存放“myString”；如果存在的话，则不用重新开辟空间，保证常量池中只有一个“myString”常量，节省内存空间。

然后在内存堆中开辟一块空间存放new出来的String实例，在栈中开辟一块空间，命名为“s1”，存放的值为堆中String实例的内存地址，这个过程就是将引用s1指向new出来的String实例。

各位，最模糊的地方到了！堆中new出来的实例和常量池中的“myString”是什么关系呢？等我们分析完了第二种定义方式之后再回头分析这个问题。

（2）直接定义，如：String s1 = “myString”;

第二种方式直接定义过程：在程序编译期，编译程序先去字符串常量池检查，是否存在“myString”，如果不存在，则在常量池中开辟一个内存空间存放“myString”；如果存在的话，则不用重新开辟空间。

然后在栈中开辟一块空间，命名为“s1”，存放的值为常量池中“myString”的内存地址。常量池中的字符串常量与堆中的String对象有什么区别呢？为什么直接定义的字符串同样可以调用String对象的各种方法呢？

常量池中的字符串常量实质上是一个String实例，与堆中的String实例是克隆关系

（3）String str=”kv”+”ill”+” “+”ans”;

由于String类的immutable性质,这一说又要说很多，大家只 要知道String的实例一旦生成就不会再改变了，比如说：String str=”kv”+”ill”+” “+”ans”; 就是有4个字符串常量，

首先”kv”和”ill”生成了”kvill”存在内存中，然后”kvill”又和” ” 生成 “kvill “存在内存中，最后又和生成了”kvill ans”;并把这个字符串的地址赋给了str,就是因为String的

”不可变”产生了很多临时变量，这也就是为什么建议用StringBuffer的原 因了，因为StringBuffer是可改变的。

### 9设计工具用过哪些？继承和接口实现用什么箭头表示？

Powerdesigner和visio

实线三角形是继承，虚线三角形是实线接口

### 10 hashmap内部实现原理？

一、HashMap概述

HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

二、HashMap的数据结构

在Java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

从上图中可以看出，HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个数组。

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry[] table;

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

final int hash;

……

}

可以看出，Entry就是数组中的元素，每个 Map.Entry 其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。

三、HashMap的存取实现

1 存储

public V put(K key, V value) {

// HashMap允许存放null键和null值。

// 当key为null时，调用putForNullKey方法，将value放置在数组第一个位置。

if (key == null)

return putForNullKey(value);

// 根据key的keyCode重新计算hash值。

int hash = hash(key.hashCode());

// 搜索指定hash值在对应table中的索引。

int i = indexFor(hash, table.length);

// 如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素。

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;

}

}

// 如果i索引处的Entry为null，表明此处还没有Entry。

modCount++;

// 将key、value添加到i索引处。

addEntry(hash, key, value, i);

return null;

}

从上面的源代码中可以看出：当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标）， 如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

addEntry(hash, key, value, i)方法根据计算出的hash值，将key-value对放在数组table的i索引处。addEntry 是 HashMap 提供的一个包访问权限的方法，代码如下：

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

// 获取指定 bucketIndex 索引处的 Entry

Entry<K,V> e = table[bucketIndex];

// 将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry

table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

// 如果 Map 中的 key-value 对的数量超过了极限

if (size++ >= threshold)

// 把 table 对象的长度扩充到原来的2倍。

resize(2 \* table.length);

}

当系统决定存储HashMap中的key-value对时，完全没有考虑Entry中的value，仅仅只是根据key来计算并决定每个Entry的存储位置。我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

hash(int h)方法根据key的hashCode重新计算一次散列。此算法加入了高位计算，防止低位不变，高位变化时，造成的hash冲突。

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

我们可以看到在HashMap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。如何计算这个位置就是hash算法。前面说过HashMap的数据结构是数组和链表的结合，所以我们当然希望这个HashMap里面的 元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表，这样就大大优化了查询的效率。

对于任意给定的对象，只要它的 hashCode() 返回值相同，那么程序调用 hash(int h) 方法所计算得到的 hash 码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，“模”运算的消耗还是比较大的，在HashMap中是这样做的：调用 indexFor(int h, int length) 方法来计算该对象应该保存在 table 数组的哪个索引处。indexFor(int h, int length) 方法的代码如下：

static int indexFor(int h, int length) {

return h & (length-1);

}

这个方法非常巧妙，它通过 h & (table.length -1) 来得到该对象的保存位，而HashMap底层数组的长度总是 2 的 n 次方，这是HashMap在速度上的优化。在 HashMap 构造器中有如下代码：

int capacity = 1;

while (capacity < initialCapacity)

capacity <<= 1;

这段代码保证初始化时HashMap的容量总是2的n次方，即底层数组的长度总是为2的n次方。

当length总是 2 的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。

这看上去很简单，其实比较有玄机的，我们举个例子来说明：

假设数组长度分别为15和16，优化后的hash码分别为8和9，那么&运算后的结果如下：

h & (table.length-1) hash table.length-1

8 & (15-1)： 0100 & 1110 = 0100

9 & (15-1)： 0101 & 1110 = 0100

8 & (16-1)： 0100 & 1111 = 0100

9 & (16-1)： 0101 & 1111 = 0101

从上面的例子中可以看出：当它们和15-1（1110）“与”的时候，产生了相同的结果，也就是说它们会定位到数组中的同一个位置上去，这就产生了碰撞，8和9会被放到数组中的同一个位置上形成链表，那么查询的时候就需要遍历这个链 表，得到8或者9，这样就降低了查询的效率。同时，我们也可以发现，当数组长度为15的时候，hash值会与15-1（1110）进行“与”，那么最后一位永远是0，而0001，0011，0101，1001，1011，0111，1101这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！而当数组长度为16时，即为2的n次方时，2n-1得到的二进制数的每个位上的值都为1，这使得在低位上&时，得到的和原hash的低位相同，加之hash(int h)方法对key的hashCode的进一步优化，加入了高位计算，就使得只有相同的hash值的两个值才会被放到数组中的同一个位置上形成链表。

所以说，当数组长度为2的n次幂的时候，不同的key算得得index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

根据上面 put 方法的源代码可以看出，当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部——具体说明继续看 addEntry() 方法的说明。

2 读取

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

有了上面存储时的hash算法作为基础，理解起来这段代码就很容易了。从上面的源代码中可以看出：从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

3 归纳

简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，

也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

四、HashMap的resize（rehash）

当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，这是一个常用的操作，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

那么HashMap什么时候进行扩容呢？当HashMap中的元素个数超过数组大小loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过160.75=12的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

五、HashMap的性能参数

HashMap 包含如下几个构造器：

HashMap()：构建一个初始容量为 16，负载因子为 0.75 的 HashMap。

ashMap(int initialCapacity)：构建一个初始容量为 initialCapacity，负载因子为 0.75 的 HashMap。

HashMap(int initialCapacity, float loadFactor)：以指定初始容量、指定的负载因子创建一个 HashMap。

HashMap的基础构造器HashMap(int initialCapacity, float loadFactor)带有两个参数，它们是初始容量initialCapacity和负载因子loadFactor。

负载因子loadFactor衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。

HashMap的实现中，通过threshold字段来判断HashMap的最大容量：

1 threshold = (int)(capacity \* loadFactor);

结合负载因子的定义公式可知，threshold就是在此loadFactor和capacity对应下允许的最大元素数目，超过这个数目就重新resize，以降低实际的负载因子。默认的的负载因子0.75是对空间和时间效率的一个平衡选择。当容量超出此最大容量时， resize后的HashMap容量是容量的两倍：

六、Fail-Fast机制

我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

HashIterator() {

expectedModCount = modCount;

if (size > 0) { // advance to first entry

Entry[] t = table;

while (index < t.length && (next = t[index++]) == null)

;

}

}

在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map：

注意到modCount声明为volatile，保证线程之间修改的可见性。

final Entry<K,V> nextEntry() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

在HashMap的API中指出：

由所有HashMap类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

七、HashMap的两种遍历方式

第一种

Map map = new HashMap();

Iterator iter = map.entrySet().iterator();

while (iter.hasNext()) {

Map.Entry entry = (Map.Entry) iter.next();

Object key = entry.getKey();

Object val = entry.getValue();

}

效率高,以后一定要使用此种方式！

第二种

Map map = new HashMap();

Iterator iter = map.keySet().iterator();

while (iter.hasNext()) {

Object key = iter.next();

Object val = map.get(key);

}

效率低,以后尽量少使用！

### 11 ArrayList和LinkedList从数据结构有何区别，用于什么场景？

LinkedList和ArrayList的差别主要来自于Array和LinkedList数据结构的不同。如果你很熟悉Array和LinkedList，你很容易得出下面的结论：

1) 因为Array是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。Array获取数据的时间复杂度是O(1),但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

2) 相对于ArrayList，LinkedList插入是更快的。因为LinkedList不像ArrayList一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是ArrayList最坏的一种情况，时间复杂度是O(n)，

而LinkedList中插入或删除的时间复杂度仅为O(1)。ArrayList在插入数据时还需要更新索引（除了插入数组的尾部）。

3) 类似于插入数据，删除数据时，LinkedList也优于ArrayList。

4) LinkedList需要更多的内存，因为ArrayList的每个索引的位置是实际的数据，而LinkedList中的每个节点中存储的是实际的数据和前后节点的位置。

5) 不会随机访问数据。因为如果你需要LinkedList中的第n个元素的时候，你需要从第一个元素顺序数到第n个数据，然后读取数据。

6) 更多的插入和删除元素，更少的读取数据。因为插入和删除元素不涉及重排数据，所以它要比ArrayList要快。

7) 以上就是关于ArrayList和LinkedList的差别。你需要一个不同步的基于索引的数据访问时，请尽量使用ArrayList。ArrayList很快，也很容易使用。但是要记得要给定一个合适的初始大小，尽可能的减少更改数组的大小。

### 12怎么预防SQL注入？

1.PreparedStatement

采预编译语句集，它内置了处理SQL注入的能力，只要使用它的setXXX方法传值即可。

使用好处：

(1).代码的可读性和可维护性.

(2).PreparedStatement尽最大可能提高性能.

(3).最重要的一点是极大地提高了安全性.

原理：

sql注入只对sql语句的准备(编译)过程有破坏作用

而PreparedStatement已经准备好了,执行阶段只是把输入串作为数据处理,

而不再对sql语句进行解析,准备,因此也就避免了sql注入问题.

2.使用正则表达式过滤传入的参数

3.字符串过滤

4.jsp中调用该函数检查是否包函非法字符

5.JSP页面判断代码：

### 13 unicode和utf-8有什么关系？

1. ASCII码

我们知道，在计算机内部，所有的信息最终都表示为一个二进制的字符串。每一个二进制位（bit）有0和1两种状态，因此八个二进制位就可以组合出256种状态，这被称为一个字节（byte）。也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从0000000到11111111。

上个世纪60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为ASCII码，一直沿用至今。

ASCII码一共规定了128个字符的编码，比如空格“SPACE”是32（二进制00100000），大写的字母A是65（二进制01000001）。这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的1位统一规定为0。

2、非ASCII编码

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。比如，在法语中，字母上方有注音符号，它就无法用ASCII码表示。于是，一些欧洲国家就决定，利用字节中闲置的最高位编入新的符号。比如，法语中的é的编码为130（二进制10000010）。这样一来，这些欧洲国家使用的编码体系，可以表示最多256个符号。

但是，这里又出现了新的问题。不同的国家有不同的字母，因此，哪怕它们都使用256个符号的编码方式，代表的字母却不一样。比如，130在法语编码中代表了é，在希伯来语编码中却代表了字母Gimel (?)，在俄语编码中又会代表另一个符号。但是不管怎样，所有这些编码方式中，0—127表示的符号是一样的，不一样的只是128—255的这一段。

至于亚洲国家的文字，使用的符号就更多了，汉字就多达10万左右。一个字节只能表示256种符号，肯定是不够的，就必须使用多个字节表达一个符号。比如，简体中文常见的编码方式是GB2312，使用两个字节表示一个汉字，所以理论上最多可以表示256x256=65536个符号。

中文编码的问题需要专文讨论，这篇笔记不涉及。这里只指出，虽然都是用多个字节表示一个符号，但是GB类的汉字编码与后文的Unicode和UTF-8是毫无关系的。

3.Unicode

正如上一节所说，世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。为什么电子邮件常常出现乱码？就是因为发信人和收信人使用的编码方式不一样。

可以想象，如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是Unicode，就像它的名字都表示的，这是一种所有符号的编码。

Unicode当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，U+4E25表示汉字“严”。具体的符号对应表，可以查询unicode.org，或者专门的汉字对应表。

4. Unicode的问题

需要注意的是，Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

比如，汉字“严”的unicode是十六进制数4E25，转换成二进制数足足有15位（100111000100101），也就是说这个符号的表示至少需要2个字节。表示其他更大的符号，可能需要3个字节或者4个字节，甚至更多。

这里就有两个严重的问题，第一个问题是，如何才能区别unicode和ascii？计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号呢？第二个问题是，我们已经知道，英文字母只用一个字节表示就够了，如果unicode统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必然有二到三个字节是0，这对于存储来说是极大的浪费，文本文件的大小会因此大出二三倍，这是无法接受的。

它们造成的结果是：1）出现了unicode的多种存储方式，也就是说有许多种不同的二进制格式，可以用来表示unicode。2）unicode在很长一段时间内无法推广，直到互联网的出现。

5.UTF-8

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8就是在互联网上使用最广的一种unicode的实现方式。其他实现方式还包括UTF-16和UTF-32，不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8是Unicode的实现方式之一。

UTF-8最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

UTF-8的编码规则很简单，只有二条：

1）对于单字节的符号，字节的第一位设为0，后面7位为这个符号的unicode码。因此对于英语字母，UTF-8编码和ASCII码是相同的。

2）对于n字节的符号（n>1），第一个字节的前n位都设为1，第n+1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的unicode码。

下表总结了编码规则，字母x表示可用编码的位。

Unicode符号范围 | UTF-8编码方式

(十六进制) | （二进制）

--------------------+---------------------------------------------

0000 0000-0000 007F | 0xxxxxxx

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

下面，还是以汉字“严”为例，演示如何实现UTF-8编码。

已知“严”的unicode是4E25（100111000100101），根据上表，可以发现4E25处在第三行的范围内（0000 0800-0000 FFFF），因此“严”的UTF-8编码需要三个字节，即格式是“1110xxxx 10xxxxxx 10xxxxxx”。然后，从“严”的最后一个二进制位开始，依次从后向前填入格式中的x，多出的位补0。这样就得到了，“严”的UTF-8编码是“11100100 10111000 10100101”，转换成十六进制就是E4B8A5。

6. Unicode与UTF-8之间的转换

通过上一节的例子，可以看到“严”的Unicode码是4E25，UTF-8编码是E4B8A5，两者是不一样的。它们之间的转换可以通过程序实现。

在Windows平台下，有一个最简单的转化方法，就是使用内置的记事本小程序Notepad.exe。打开文件后，点击“文件”菜单中的“另存为”命令，会跳出一个对话框，在最底部有一个“编码”的下拉条。

里面有四个选项：ANSI，Unicode，Unicode big endian 和 UTF-8。

1）ANSI是默认的编码方式。对于英文文件是ASCII编码，对于简体中文文件是GB2312编码（只针对Windows简体中文版，如果是繁体中文版会采用Big5码）。

2）Unicode编码指的是UCS-2编码方式，即直接用两个字节存入字符的Unicode码。这个选项用的little endian格式。

3）Unicode big endian编码与上一个选项相对应。我在下一节会解释little endian和big endian的涵义。

4）UTF-8编码，也就是上一节谈到的编码方法。

选择完”编码方式“后，点击”保存“按钮，文件的编码方式就立刻转换好了。

### 14 String,StringBuffer,StringBuilder使用场景？有何不同？

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

### 15 wait和sleep不同点？怎么唤醒？有没释放锁？

1、这两个方法来自不同的类分别是，sleep来自Thread类，和wait来自Object类。

sleep是Thread的静态类方法，谁调用的谁去睡觉，即使在a线程里调用了b的sleep方法，实际上还是a去睡觉，要让b线程睡觉要在b的代码中调用sleep。

2、最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法

sleep不出让系统资源；wait是进入线程等待池等待，出让系统资源，其他线程可以占用CPU。一般wait不会加时间限制，因为如果wait线程的运行资源不够，再出来也没用，要等待其他线程调用notify/notifyAll唤醒等待池中的所有线程，才会进入就绪队列等待OS分配系统资源。sleep(milliseconds)可以用时间指定使它自动唤醒过来，如果时间不到只能调用interrupt()强行打断。

Thread.Sleep(0)的作用是“触发操作系统立刻重新进行一次CPU竞争”。

3、使用范围：wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

synchronized(x){

x.notify()

//或者wait()

}

1. sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

### 16 concurrentHashMap和普通的map有何区别？采用什么同步技术？

并发编程实践中，ConcurrentHashMap是一个经常被使用的数据结构，相比于Hashtable以及Collections.synchronizedMap()，ConcurrentHashMap在线程安全的基础上提供了更好的写并发能力，但同时降低了对读一致性的要求（这点好像CAP理论啊 O(∩\_∩)O）。ConcurrentHashMap的设计与实现非常精巧，大量的利用了volatile，final，CAS等lock-free技术来减少锁竞争对于性能的影响，无论对于Java并发编程的学习还是Java内存模型的理解，ConcurrentHashMap的设计以及源码都值得非常仔细的阅读与揣摩。

这篇日志记录了自己对ConcurrentHashMap的一些总结，由于JDK6，7，8中实现都不同，需要分开阐述在不同版本中的ConcurrentHashMap。

之前已经在ConcurrentHashMap原理分析中解释了ConcurrentHashMap的原理，主要是从代码的角度来阐述是源码是如何写的，本文仍然从源码出发，挑选个人觉得重要的点（会用红色标注）再次进行回顾，以及阐述ConcurrentHashMap的一些注意点。

一、 设计思路

ConcurrentHashMap采用了分段锁的设计，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。相比于对整个Map加锁的设计，分段锁大大的提高了高并发环境下的处理能力。但同时，由于不是对整个Map加锁，导致一些需要扫描整个Map的方法（如size(), containsValue()）需要使用特殊的实现，另外一些方法（如clear()）甚至放弃了对一致性的要求（ConcurrentHashMap是弱一致性的，具体请查看ConcurrentHashMap能完全替代HashTable吗？）。

ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock）。ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性，代码如下：

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

二、 并发度（Concurrency Level）

并发度可以理解为程序运行时能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。ConcurrentHashMap默认的并发度为16，但用户也可以在构造函数中设置并发度。当用户设置并发度时，ConcurrentHashMap会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）。运行时通过将key的高n位（n = 32 – segmentShift）和并发度减1（segmentMask）做位与运算定位到所在的Segment。segmentShift与segmentMask都是在构造过程中根据concurrency level被相应的计算出来。

如果并发度设置的过小，会带来严重的锁竞争问题；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中，CPU cache命中率会下降，从而引起程序性能下降。（文档的说法是根据你并发的线程数量决定，太多会导性能降低）

三、创建分段锁

和JDK6不同，JDK7中除了第一个Segment之外，剩余的Segments采用的是延迟初始化的机制：每次put之前都需要检查key对应的Segment是否为null，如果是则调用ensureSegment()以确保对应的Segment被创建。

ensureSegment可能在并发环境下被调用，但与想象中不同，ensureSegment并未使用锁来控制竞争，而是使用了Unsafe对象的getObjectVolatile()提供的原子读语义结合CAS来确保Segment创建的原子性。代码段如下：

if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))

== null) { // recheck

Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);

while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))

== null) {

if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))

break;

}

}

四、 put/putIfAbsent/putAll

和JDK6一样，ConcurrentHashMap的put方法被代理到了对应的Segment（定位Segment的原理之前已经描述过）中。与JDK6不同的是，JDK7版本的ConcurrentHashMap在获得Segment锁的过程中，做了一定的优化 - 在真正申请锁之前，put方法会通过tryLock()方法尝试获得锁，在尝试获得锁的过程中会对对应hashcode的链表进行遍历，如果遍历完毕仍然找不到与key相同的HashEntry节点，则为后续的put操作提前创建一个HashEntry。当tryLock一定次数后仍无法获得锁，则通过lock申请锁。

需要注意的是，由于在并发环境下，其他线程的put，rehash或者remove操作可能会导致链表头结点的变化，因此在过程中需要进行检查，如果头结点发生变化则重新对表进行遍历。而如果其他线程引起了链表中的某个节点被删除，即使该变化因为是非原子写操作（删除节点后链接后续节点调用的是Unsafe.putOrderedObject()，该方法不提供原子写语义）可能导致当前线程无法观察到，但因为不影响遍历的正确性所以忽略不计。

之所以在获取锁的过程中对整个链表进行遍历，主要目的是希望遍历的链表被CPU cache所缓存，为后续实际put过程中的链表遍历操作提升性能。

在获得锁之后，Segment对链表进行遍历，如果某个HashEntry节点具有相同的key，则更新该HashEntry的value值，否则新建一个HashEntry节点，将它设置为链表的新head节点并将原头节点设为新head的下一个节点。新建过程中如果节点总数（含新建的HashEntry）超过threshold，则调用rehash()方法对Segment进行扩容，最后将新建HashEntry写入到数组中。

put方法中，链接新节点的下一个节点（HashEntry.setNext()）以及将链表写入到数组中（setEntryAt()）都是通过Unsafe的putOrderedObject()方法来实现，这里并未使用具有原子写语义的putObjectVolatile()的原因是：JMM会保证获得锁到释放锁之间所有对象的状态更新都会在锁被释放之后更新到主存，从而保证这些变更对其他线程是可见的。

五、 rehash

相对于HashMap的resize，ConcurrentHashMap的rehash原理类似，但是Doug Lea为rehash做了一定的优化，避免让所有的节点都进行复制操作：由于扩容是基于2的幂指来操作，假设扩容前某HashEntry对应到Segment中数组的index为i，数组的容量为capacity，那么扩容后该HashEntry对应到新数组中的index只可能为i或者i+capacity，因此大多数HashEntry节点在扩容前后index可以保持不变。基于此，rehash方法中会定位第一个后续所有节点在扩容后index都保持不变的节点，然后将这个节点之前的所有节点重排即可。这部分代码如下：

private void rehash(HashEntry<K,V> node) {

HashEntry<K,V>[] oldTable = table;

int oldCapacity = oldTable.length;

int newCapacity = oldCapacity << 1;

threshold = (int)(newCapacity \* loadFactor);

HashEntry<K,V>[] newTable =

(HashEntry<K,V>[]) new HashEntry[newCapacity];

int sizeMask = newCapacity - 1;

for (int i = 0; i < oldCapacity ; i++) {

HashEntry<K,V> e = oldTable[i];

if (e != null) {

HashEntry<K,V> next = e.next;

int idx = e.hash & sizeMask;

if (next == null) // Single node on list

newTable[idx] = e;

else { // Reuse consecutive sequence at same slot

HashEntry<K,V> lastRun = e;

int lastIdx = idx;

for (HashEntry<K,V> last = next;

last != null;

last = last.next) {

int k = last.hash & sizeMask;

if (k != lastIdx) {

lastIdx = k;

lastRun = last;

}

}

newTable[lastIdx] = lastRun;

// Clone remaining nodes

for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {

V v = p.value;

int h = p.hash;

int k = h & sizeMask;

HashEntry<K,V> n = newTable[k];

newTable[k] = new HashEntry<K,V>(h, p.key, v, n);

}

}

}

}

int nodeIndex = node.hash & sizeMask; // add the new node

node.setNext(newTable[nodeIndex]);

newTable[nodeIndex] = node;

table = newTable;

}

六、 remove

和put类似，remove在真正获得锁之前，也会对链表进行遍历以提高缓存命中率。

七、 get与containsKey

get与containsKey两个方法几乎完全一致：他们都没有使用锁，而是通过Unsafe对象的getObjectVolatile()方法提供的原子读语义，来获得Segment以及对应的链表，然后对链表遍历判断是否存在key相同的节点以及获得该节点的value。但由于遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据，这一点是ConcurrentHashMap在弱一致性上的体现。如果要求强一致性，那么必须使用Collections.synchronizedMap()方法。

八、 size、containsValue

这些方法都是基于整个ConcurrentHashMap来进行操作的，他们的原理也基本类似：首先不加锁循环执行以下操作：循环所有的Segment（通过Unsafe的getObjectVolatile()以保证原子读语义），获得对应的值以及所有Segment的modcount之和。如果连续两次所有Segment的modcount和相等，则过程中没有发生其他线程修改ConcurrentHashMap的情况，返回获得的值。

当循环次数超过预定义的值时，这时需要对所有的Segment依次进行加锁，获取返回值后再依次解锁。值得注意的是，加锁过程中要强制创建所有的Segment，否则容易出现其他线程创建Segment并进行put，remove等操作。代码如下：

for(int j =0; j < segments.length; ++j)

ensureSegment(j).lock();// force creation

一般来说，应该避免在多线程环境下使用size和containsValue方法。

注1：modcount在put, replace, remove以及clear等方法中都会被修改。

注2：对于containsValue方法来说，如果在循环过程中发现匹配value的HashEntry，则直接返回true。

最后，与HashMap不同的是，ConcurrentHashMap并不允许key或者value为null，按照Doug Lea的说法，这么设计的原因是在ConcurrentHashMap中，一旦value出现null，则代表HashEntry的key/value没有映射完成就被其他线程所见，需要特殊处理。在JDK6中，get方法的实现中就有一段对HashEntry.value == null的防御性判断。但Doug Lea也承认实际运行过程中，这种情况似乎不可能发生

### 17 注解有什么好处？

配置文件

1，遵循OCP开发原则，修改配置文件即可进行功能扩展（OCP 开闭原则 Open Closed Principle）

2，集中管理对象和对象之间的组合关系，易于阅读

3，开发速度相对较慢；

4，编译时很难检查出错误，运行中的错误很难定位，调试难度较大。

注解

1，开发速度快

2，编译期间容易发现错误的出处

3，管理分散，基本每个类上都有

4，扩展功能时，没有遵循OCP开发原则

### 18 数组与ArrayList的关系与区别？

存储结构一样 。

数组（[]）：最高效；但是其容量固定且无法动态改变；

ArrayList：容量可动态增长；但牺牲效率；

建议首先使用数组，无法确定数组大小时才使用ArrayList！

数组扩容是对ArrayList效率影响比较大的一个因素。

每当执行Add、AddRange、Insert、InsertRange等添加元素的方法，都会检查内部数组的容量是否不够了，如果是，它就会以当前容量的1.5倍来重新构建一个数组，将旧元素Copy到新数组中，然后丢弃旧数组，在这个临界点的扩容操作，应该来说是比较影响效率的。

### 19 序列化应用什么场景？

1 简述

在分布式环境下，当进行远程通信时，无论是何种类型的数据，都会以二进制序列的形式在网络上传输。序列化是一种将对象以一连串的字节描述的过程，

用于解决在对对象流进行读写操作时所引发的问题。序列化可以将对象的状态写在流里进行网络传输，或者保存到文件、数据库等系统中，并在需要时把

该流读取出来重新构造一个相同的对象。

所有实现序列化的类都必须实现Serializable接口，Serializable接口位于java.lang包中，没有任何实现方法，使用一个输出流（例如FileOutputStream）

来构造一个ObjectOutputStream对象，紧接着使用该对象的writeObject(Object obj)方法就可以将obj对象写出，要恢复时可以使用其对应的输入流ObjectInputStream.

Java平台允许我们在内存中创建可复用的Java对象，但一般情况下，只有当JVM处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比JVM的生命周期更长。

但在现实应用中，就可能要求在JVM停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java对象序列化就能够帮助我们实现该功能。使用Java对

象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的"状态"，即它的成员变量。由此可

知，对象序列化不会关注类中的静态变量。除了在持久化对象时会用到对象序列化之外，当使用RMI(远程方法调用)，或在网络中传递对象时，都会用到对象序列化。Java序

列化API为处理对象序列化提供了一个标准机制，该API简单易用，在本文的后续章节中将会陆续讲到。

2 序列化特点

（1）如果一个类能被序列化，那么它的子类也能够被序列化。

（2）由于static(静态)代表类的成员，transient（Java关键字，如果用transient声明一个实例变量，当对象存储时，它的值不需要维持）代表对象的临时数据，因此被声明为这两种类型的数据成员是不能够被序列化的。

（3）每个枚举类型都会默认继承类java.lang.Enum，而该类实现了Serializable接口，所以枚举类型对象都是默认可以被序列化的。

3 序列化场景

1）当你想把的内存中的对象状态保存到一个文件中时候；

2）当你想把的内存中的对象状态保存到数据库时候；

3）当你想用套接字在网络上传送对象的时候；

4）当你想通过RMI传输对象的时候；

4 序列化

如下是序列化的代码，首先声明一个Student类继承Serializable接口，由代码中的输出（注释即为输出内容）可得静态变量SCHOOLNAME和SCHOOLID的值发生了变化，难道是静态变量也被序列化

了吗？其实不是的，因为在当前的运行程序中，Student类的静态变量的值已经发生了变化，如果真的已经序列化了，那么我们将序列化的那个函数去掉，让程序从SourceFile/Student文件中反

序列化，那么得到的SCHOOLNAME应该为HeBei University，SCHOOLID的值为2，然而当我们去掉序列化代码，直接从文件反序列化，输出SCHOOLNAME是Beijing University of Post and Telecommunications，SCHOOLID为1，说明static变量并没有被序列化。

5 反序列化

与序列化相对的是反序列化，它将流转换为对象，在序列化与反序列化的过程中，serialVersionUID起着重要的作用，每一个类都有一个特定的serialVersionUID，在反序列化的过程

自定义serialVersionUID主要由如下3个优点：

（1）提高程序运行效率。如果在类中未显示声明serialVersionUID，那么在序列化时会通过计算得到一个serialVersionUID。通过显示声明serialVersionUID的方式省去了计算的过程，提高了程序效率。

（2）提高程序不同平台上的兼容性。由于各个平台计算serialVersionUID的方式可能不同，通过显示的方式可以完全避免该问题。

（3）增强程序各个版本的可兼容性。在默认情况下每个类都有唯一的一个serialVersionUID，因此当后期对类进行修改时，类的serialVersionUID值将会发生变化，这将会导致类在修改前对象的文件在修

改后无法进行反序列化操作。同样通过显示声明serialVersionUID也会解决该问题。

6 Serializable接口和serialVersionUID

序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。

如果接收者加载的该对象的类的 serialVersionUID 与对应的发送者的类的版本号不同，则反序列化将会导致 InvalidClassException。可序列化类可以通过声明名为 "serialVersionUID"

的字段（该字段必须是静态 (static)、最终 (final) 的 long 型字段）显式声明其自己的 serialVersionUID：

如果可序列化类未显式声明 serialVersionUID，则序列化运行时将基于该类的各个方面计算该类的默认 serialVersionUID 值，如“Java(TM) 对象序列化规范”中所述。不过，强烈建议所有

可序列化类都显式声明 serialVersionUID 值，原因计算默认的 serialVersionUID 对类的详细信息具有较高的敏感性，根据编译器实现的不同可能千差万别，这样在反序列化过程中可能会

导致意外的 InvalidClassException。因此，为保证 serialVersionUID 值跨不同 java 编译器实现的一致性，序列化类必须声明一个明确的 serialVersionUID 值。还强烈建议使用 private 修改

器显示声明 serialVersionUID，原因是这种声明仅应用于立即声明类 -- serialVersionUID 字段作为继承成员没有用处。

开始指定了一个serialVersionUID，序列化了一个对象，后来这个类有修改，但是要固定serialVersionUID否则反序列化失败。

　 实现serializable接口的作用是就是可以把对象存到字节流，然后可以恢复。所以你想如果你的对象没实现序列化怎么才能进行网络传输呢，要网络传输就得转为字节流，所以在分布式应用

中，你就得实现序列化，如果你不需要分布式应用，那就没那个必要实现序列化。实际上serializable就是一个空接口，一个标识，标识一个类可以序列化。

### 20 JDK 6/7/8有什么新特性?

java5 泛型,枚举,拆箱与装箱,静态导入

java6 支持JDBC4.0规范,轻量级HttpServer,JSR223脚本引擎

java7 捕获多个异常,try-with-resources，支持JDBC4.1规范

java8 Stream函数式操作流元素集合，类型注解，新增base64加解密API

### 21数据库连接池解决什么问题？

一般来说java应用程序访问数据库的过程是：

　①装载数据库驱动程序；

　②通过jdbc建立数据库连接；

　③访问数据库，执行sql语句；

　④断开数据库连接。

程序开发过程中，存在很多问题：首先，每一次web请求都要建立一次数据库连接。建立连接是一个费时的活动，每次都得花费0.05s～1s的时间，而且系统还要分配内存资源。

这个时间对于一次或几次数据库操作，或许感觉不出系统有多大的开销。可是对于现在的web应用，尤其是大型电子商务网站，同时有几百人甚至几千人在线是很正常的事。

在这种情况下，频繁的进行数据库连接操作势必占用很多的系统资源，网站的响应速度必定下降，严重的甚至会造成服务器的崩溃。不是危言耸听，这就是制约某些电子商务网站

发展的技术瓶颈问题。其次，对于每一次数据库连接，使用完后都得断开。否则，如果程序出现异常而未能关闭，将会导致数据库系统中的内存泄漏，最终将不得不重启数据库。

还有，这种开发不能控制被创建的连接对象数，系统资源会被毫无顾及的分配出去，如连接过多，也可能导致内存泄漏，服务器崩溃。

连接池还要考虑更多的问题

1、并发问题

为了使连接管理服务具有最大的通用性，必须考虑多线程环境，即并发问题。这个问题相对比较好解决，因为java语言自身提供了对并发管理的支持，使用synchronized关键字即可确保线程是同步的。使用方法为直接在类方法前面加上synchronized关键

publicsynchronized connection getconnection（）

2、多数据库服务器和多用户

对于大型的企业级应用，常常需要同时连接不同的数据库（如连接oracle和sybase）。如何连接不同的数据库呢？我们采用的策略是：设计一个符合单例模式的连接池管理类，在连接池管理类的唯一实例被创建时读取一个资源文件，

其中资源文件中存放着多个数据库的url地址等信息。根据资源文件提供的信息，创建多个连接池类的实例，每一个实例都是一个特定数据库的连接池。连接池管理类实例为每个连接池实例取一个名字，通过不同的名字来管理不同的连接池。

对于同一个数据库有多个用户使用不同的名称和密码访问的情况，也可以通过资源文件处理，即在资源文件中设置多个具有相同url地址，但具有不同用户名和密码的数据库连接信息。

3、事务处理

我们知道，事务具有原子性，此时要求对数据库的操作符合“all-all-nothing”原则即对于一组sql语句要么全做，要么全不做。

在java语言中，connection类本身提供了对事务的支持，可以通过设置connection的autocommit属性为false 然后显式的调用commit或rollback方法来实现。但要高效的进行connection复用，就必须提供相应的事务支持机制。可采用每一个事务独占一个连

接来实现，这种方法可以大大降低事务管理的复杂性。

4、连接池的分配与释放

连接池的分配与释放，对系统的性能有很大的影响。合理的分配与释放，可以提高连接的复用度，从而降低建立新连接的开销，同时还可以加快用户的访问速度。

对于连接的管理可使用空闲池。即把已经创建但尚未分配出去的连接按创建时间存放到一个空闲池中。每当用户请求一个连接时，系统首先检查空闲池内有没有空闲连接。如果有就把建立时间最长（通过容器的顺序存放实现）的那个连接分配给他（实际是

先做连接是否有效的判断，如果可用就分配给用户，如不可用就把这个连接从空闲池删掉，重新检测空闲池是否还有连接）；如果没有则检查当前所开连接池是否达到连接池所允许的最大连接数（maxconn）如果没有达到，就新建一个连接，如果已经达到，

就等待一定的时间（timeout）。如果在等待的时间内有连接被释放出来就可以把这个连接分配给等待的用户，如果等待时间超过预定时间timeout 则返回空值（null）。系统对已经分配出去正在使用的连接只做计数，当使用完后再返还给空闲池。对于空

闲连接的状态，可开辟专门的线程定时检测，这样会花费一定的系统开销，但可以保证较快的响应速度。也可采取不开辟专门线程，只是在分配前检测的方法。

5、连接池的配置与维护

连接池中到底应该放置多少连接，才能使系统的性能最佳？系统可采取设置最小连接数（minconn）和最大连接数（maxconn）来控制连接池中的连接。最小连接数是系统启动时连接池所创建的连接数。如果创建过多，则系统启动就慢，但创建后系统的响应

速度会很快；如果创建过少，则系统启动的很快，响应起来却慢。这样，可以在开发时，设置较小的最小连接数，开发起来会快，而在系统实际使用时设置较大的，因为这样对访问客户来说速度会快些。最大连接数是连接池中允许连接的最大数目，具体设

置多少，要看系统的访问量，可通过反复测试，找到最佳点。如何确保连接池中的最小连接数呢？有动态和静态两种策略。动态即每隔一定时间就对连接池进行检测，如果发现连接数量小于最小连接数，则补充相应数量的新连接以保证连接池的正常运转。静

态是发现空闲连接不够时再去检查。

### 22运行时异常是什么？和检查异常有什么区别？

1.可查异常和不可查异常

通常，Java的异常(包括Exception和Error)分为可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）。

可查异常（编译器要求必须处置的异常）：正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。

除了RuntimeException及其子类以外，其他的Exception类及其子类都属于可查异常。这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它，否则编译不会通过。

不可查异常(编译器不要求强制处置的异常):包括运行时异常（RuntimeException与其子类）和错误（Error）。

如果使用throw在方法体中抛出可查异常，则需要在方法头部声明方法可能抛出的异常类型。程序会在throw语句后立即终止，它后面的语句执行不到，然后在包含它的所有try块中（可能在上层调用函数中）从里向外寻找含有与其匹配的catch子句的try块。

FileNotFoundException,IOException,SQLException.

2.运行时异常和非运行时异常

(1)运行时异常都是RuntimeException类及其子类异常，如NullPointerException、IndexOutOfBoundsException等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

当出现RuntimeException的时候，我们可以不处理。当出现这样的异常时，总是由虚拟机接管。比如：我们从来没有人去处理过NullPointerException异常，它就是运行时异常，并且这种异常还是最常见的异常之一。

出现运行时异常后，如果没有捕获处理这个异常（即没有catch），系统会把异常一直往上层抛，一直到最上层，如果是多线程就由Thread.run()抛出，如果是单线程就被main()抛出。抛出之后，如果是线程，这个线程也就退出了。如果是主程序抛出的异常，那么这整个程序也就退出了。运行时异常是Exception的子类，也有一般异常的特点，是可以被catch块处理的。只不过往往我们不对他处理罢了。也就是说，你如果不对运行时异常进行处理，那么出现运行时异常之后，要么是线程中止，要么是主程序终止。

如果不想终止，则必须捕获所有的运行时异常，决不让这个处理线程退出。队列里面出现异常数据了，正常的处理应该是把异常数据舍弃，然后记录日志。不应该由于异常数据而影响下面对正常数据的处理。

(2)非运行时异常是RuntimeException以外的异常，类型上都属于Exception类及其子类。如IOException、SQLException等以及用户自定义的Exception异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch并处理，否则程序就不能编译通过。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆catch块去处理可能的异常。

### 23 得到线程运行结果有几种方式实现？

（1）最官方办法 Future接口

public class ThreadReturnValue implements Callable {

@Override

public Object call() throws Exception {

int temp = 0;

StringBuffer sb = new StringBuffer("");

synchronized (App.GLOBALARRAY) {

for (int i = 0; i < App.GLOBALARRAY.length; i++) {

System.out.println(Thread.currentThread().getName() + "\n" + App.GLOBALARRAY[i]);

App.GLOBALARRAY[i] += 1;

temp += App.GLOBALARRAY[i];

sb.append("+").append(App.GLOBALARRAY[i]);

}

}

return sb.toString() + "=" + temp;

}

}

ExecutorService pool = Executors.newFixedThreadPool(3);

Future<String> future = pool.submit(new ThreadReturnValue());

Future<String> future2 = pool.submit(new ThreadReturnValue());

Future<String> future3 = pool.submit(new ThreadReturnValue());

pool.shutdown(); // 不允许再想线程池中增加线程

String r = future.get();

String r2 = future2.get();

String r3 = future3.get();

（2）join方法

（3）循环判断线程的状态如果已结束在获取结果

### 24 如何避免超卖和超发的情况发生？

@悲观锁思路，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待

@FIFO内存队列思路，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。

@乐观锁思路，乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，

其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。

@ConrenthashMap思路，支持同步和并发一起。

### 25 如何实现list中根据对象的某个属性排序？

Bean实现Comparator接口，覆写compare方法

public class ComparatorUser implements Comparator{

public int compare(Object obj0, Object obj1) {

User user0=(User)obj0;

User user1=(User)obj1;

int flag=user0.getAge().compareTo(user1.getAge());

if(flag==0){

return user0.getName().compareTo(user1.getName());

}else{

return flag;

}

}

}

### 26 Memcache和redis存储什么区别？

1)、Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等。

2)、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储。

3)、虚拟内存--Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘

4)、过期策略--memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10

5)、分布式--设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从

6)、存储数据安全--memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）

7)、灾难恢复--memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复

8)、Redis支持数据的备份，即master-slave模式的数据备份。

### 27 Executors有几种模式，分别是什么？

Java通过Executors提供四种线程池，分别为：

newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

### 28 线程中Join方法用法是什么？停止线程执行最官方的方法是？

join方法用得不多,当A线程执行到了B线程的join()方法时,A就会等待,等B线程都运行完,A线程才会执行。

interrupt使线程停止，stop不建议使用

### 29 Lock对比synchronized有什么优势，Condition接口中await和signal方法对比object的wait、nofity有何不同？

synchronized是基于jvm底层实现的数据同步，lock是基于Java编写，主要通过硬件依赖CPU指令实现数据同步。下面一一介绍

一、synchronized的实现方案

1.synchronized能够把任何一个非null对象当成锁，实现由两种方式：

a.当synchronized作用于非静态方法时，锁住的是当前对象的事例，当synchronized作用于静态方法时，锁住的是class实例，又因为Class的相关数据存储在永久带，因此静态方法锁相当于类的一个全局锁。

b.当synchronized作用于一个对象实例时，锁住的是对应的代码块。

2.synchronized锁又称为对象监视器（object）。

3.当多个线程一起访问某个对象监视器的时候，对象监视器会将这些请求存储在不同的容器中。

>Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中

>Entry List：Contention List中那些有资格成为候选资源的线程被移动到Entry List中

>Wait Set：哪些调用wait方法被阻塞的线程被放置在这里

>OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为OnDeck

>Owner：当前已经获取到所资源的线程被称为Owner

> !Owner：当前释放锁的线程

下图展示了他们之前的关系

4.synchronized在jdk1.6之后提供了多种优化方案：

>自旋锁

jdk1.6之后默认开启，可以使用参数-XX:+UseSpinning控制，自旋等待不能代替阻塞，且先不说对处理器数量的要求，自旋等待本身虽然避免了线程切换的开销，但它是要占用处理器时间的，因此，如果锁被占用的时间很短，自旋等待的效果就会非常好，反之，如果锁被占用的时候很长，那么自旋的线程只会白白消耗处理器资源，而不会做任何有用的工作，反而会带来性能上的浪费。自旋次数的默认值是 10 次，用户可以使用参数 -XX:PreBlockSpin 来更改。

自旋锁的本质：执行几个空方法，稍微等一等，也许是一段时间的循环，也许是几行空的汇编指令。

>锁消除

即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除，依据来源于逃逸分析的数据支持，那么是什么是逃逸分析？对于虚拟机来说需要使用数据流分析来确定是否消除变量底层框架的同步代码，因为有许多同步的代码不是自己写的。

例1.1

public static String concatString(String s1, String s2, String s3) {

return s1 + s2 + s3;

}

由于 String 是一个不可变的类，对字符串的连接操作总是通过生成新的 String 对象来进行的，因此 Javac 编译器会对 String 连接做自动优化。在 JDK 1.5 之前，会转化为 StringBuffer 对象的连续 append() 操作，在 JDK 1.5 及以后的版本中，会转化为 StringBuilder 对象的连续 append() 操作，这里的stringBuilder.append是线程不同步的（假设是同步）。

Javac 转化后的字符串连接代码为：

public static String concatString(String s1, String s2, String s3) {

StringBuffer sb = new StringBuffer();

sb.append(s1);

sb.append(s2);

sb.append(s3);

return sb.toString();

}

此时的锁对象就是sb,虚拟机观察变量 sb，很快就会发现它的动态作用域被限制在 concatString() 方法内部。也就是说，sb 的所有引用永远不会 “逃逸” 到concatString() 方法之外，其他线程无法访问到它，虽然这里有锁，但是可以被安全地消除掉，在即时编译之后，这段代码就会忽略掉所有的同步而直接执行了。

>锁粗化

将同步块的作用范围限制得尽量小——只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待锁的线程也能尽快拿到锁。

>轻量级锁

加锁过程：在代码进入同步块的时候，如果此同步对象没有被锁定（锁标志位为 “01” 状态）虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的 Mark Word 的拷贝，这时候线程堆栈与对象头的状态如图 13-3 所示

然后，虚拟机将使用 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Record 的指针。如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象 Mark Word 的锁标志位 （Mark Word 的最后 2bit）将转变为 “00”，即表示此对象处于轻量级锁定状态，这时线程堆栈与对象头的状态如图13-4

如果上述更新操作失败，则说明这个锁对象被其他锁占用，此时轻量级变为重量级锁，标志位为“10”，后面等待的线程进入阻塞状态。

解锁过程：也是由CAS进行操作的，如果对象的 Mark Word 仍然指向着线程的锁记录，那就用 CAS 操作把对象当前的 Mark Word 和线程中复制的 Displaced Mark Word 替换回来，如果替换成功，整个同步过程就完成了。如果替换失败，说明有其他线程尝试过获取该锁，那就要释放锁的同时，唤醒被挂起的线程。

轻量级锁能提升程序同步性能的依据是 “对于绝大部分的锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥量的开销，但如果存在锁竞争，除了互斥量的开销外，还额外发生了 CAS 操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢。

>偏向锁

偏向锁也是 JDK 1.6 中引入的一项锁优化，它的目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能。如果说轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把整个同步都消除掉，连 CAS 操作都不做了。

实质就是设置一个变量，判断这个变量是否是当前线程，是就避免再次加锁解锁操作，从而避免了多次的CAS操作。坏处是如果一个线程持有偏向锁，另外一个线程想争用偏向对象，拥有者想释放这个偏向锁，释放会带来额外的性能开销，但是总体来说偏向锁带来的好处还是大于CAS的代价的。在具体问题具体分析的前提下，有时候使用参数 -XX:-UseBiasedLocking 来禁止偏向锁优化反而可以提升性能。

二、lock的实现方案

与synchronized不同的是lock是纯java手写的，与底层的JVM无关。在java.util.concurrent.locks包中有很多Lock的实现类，常用的有ReenTrantLock、ReadWriteLock(实现类有ReenTrantReadWriteLock)

，其实现都依赖java.util.concurrent.AbstractQueuedSynchronizer类（简称AQS），实现思路都大同小异，因此我们以ReentrantLock作为讲解切入点。

分析之前我们先来花点时间看下AQS。AQS是我们后面将要提到的CountDownLatch/FutureTask/ReentrantLock/RenntrantReadWriteLock/Semaphore的基础，因此AQS也是Lock和Excutor实现的基础。它的基本思想就是一个同步器，支持获取锁和释放锁两个操作。

要支持上面锁获取、释放锁就必须满足下面的条件：

1、 状态位必须是原子操作的

2、 阻塞和唤醒线程

3、 一个有序的队列，用于支持锁的公平性

场景：可定时的、可轮询的与可中断的锁获取操作，公平队列，或者非块结构的锁。

主要从以下几个特点介绍：

1.可重入锁

如果锁具备可重入性，则称作为可重入锁。像synchronized和ReentrantLock都是可重入锁，可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。

2.可中断锁

可中断锁：顾名思义，就是可以相应中断的锁。

在Java中，synchronized就不是可中断锁，而Lock是可中断锁。

如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

3.公平锁和非公平锁

公平锁以请求锁的顺序来获取锁，非公平锁则是无法保证按照请求的顺序执行。synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

参数为true时表示公平锁，不传或者false都是为非公平锁。

ReentrantLock lock = new ReentrantLock(true);

4.读写锁

读写锁将对一个资源（比如文件）的访问分成了2个锁，一个读锁和一个写锁。

正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。

ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。

可以通过readLock()获取读锁，通过writeLock()获取写锁。

三、总结

1.synchronized

优点：实现简单，语义清晰，便于JVM堆栈跟踪，加锁解锁过程由JVM自动控制，提供了多种优化方案，使用更广泛

缺点：悲观的排他锁，不能进行高级功能

2.lock

优点：可定时的、可轮询的与可中断的锁获取操作，提供了读写锁、公平锁和非公平锁

缺点：需手动释放锁unlock，不适合JVM进行堆栈跟踪

3.相同点

都是可重入锁

**Condition**，Condition 将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set （wait-set）。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。下面将之前写过的一个线程通信的例子替换成用Condition实现(Java线程(三))，代码如下：

public class ThreadTest2 {

public static void main(String[] args) {

final Business business = new Business();

new Thread(new Runnable() {

@Override

public void run() {

threadExecute(business, "sub");

}

}).start();

threadExecute(business, "main");

}

public static void threadExecute(Business business, String threadType) {

for(int i = 0; i < 100; i++) {

try {

if("main".equals(threadType)) {

business.main(i);

} else {

business.sub(i);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Business {

private boolean bool = true;

private Lock lock = new ReentrantLock();

private Condition condition = lock.newCondition();

public /\*synchronized\*/ void main(int loop) throws InterruptedException {

lock.lock();

try {

while(bool) {

condition.await();//this.wait();

}

for(int i = 0; i < 100; i++) {

System.out.println("main thread seq of " + i + ", loop of " + loop);

}

bool = true;

condition.signal();//this.notify();

} finally {

lock.unlock();

}

}

public /\*synchronized\*/ void sub(int loop) throws InterruptedException {

lock.lock();

try {

while(!bool) {

condition.await();//this.wait();

}

for(int i = 0; i < 10; i++) {

System.out.println("sub thread seq of " + i + ", loop of " + loop);

}

bool = false;

condition.signal();//this.notify();

} finally {

lock.unlock();

}

}

}

在Condition中，用await()替换wait()，用signal()替换notify()，用signalAll()替换notifyAll()，传统线程的通信方式，Condition都可以实现，这里注意，Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法。

这样看来，Condition和传统的线程通信没什么区别，Condition的强大之处在于它可以为多个线程间建立不同的Condition，下面引入API中的一段代码，加以说明。

class BoundedBuffer {

final Lock lock = new ReentrantLock();//锁对象

final Condition notFull = lock.newCondition();//写线程条件

final Condition notEmpty = lock.newCondition();//读线程条件

final Object[] items = new Object[100];//缓存队列

int putptr/\*写索引\*/, takeptr/\*读索引\*/, count/\*队列中存在的数据个数\*/;

public void put(Object x) throws InterruptedException {

lock.lock();

try {

while (count == items.length)//如果队列满了

notFull.await();//阻塞写线程

items[putptr] = x;//赋值

if (++putptr == items.length) putptr = 0;//如果写索引写到队列的最后一个位置了，那么置为0

++count;//个数++

notEmpty.signal();//唤醒读线程

} finally {

lock.unlock();

}

}

public Object take() throws InterruptedException {

lock.lock();

try {

while (count == 0)//如果队列为空

notEmpty.await();//阻塞读线程

Object x = items[takeptr];//取值

if (++takeptr == items.length) takeptr = 0;//如果读索引读到队列的最后一个位置了，那么置为0

--count;//个数--

notFull.signal();//唤醒写线程

return x;

} finally {

lock.unlock();

}

}

}

这是一个处于多线程工作环境下的缓存区，缓存区提供了两个方法，put和take，put是存数据，take是取数据，内部有个缓存队列，具体变量和方法说明见代码，这个缓存区类实现的功能：有多个线程往里面存数据和从里面取数据，其缓存队列(先进先出后进后出)能缓存的最大数值是100，多个线程间是互斥的，当缓存队列中存储的值达到100时，将写线程阻塞，并唤醒读线程，当缓存队列中存储的值为0时，将读线程阻塞，并唤醒写线程，这也是ArrayBlockingQueue的内部实现。下面分析一下代码的执行过程：

1. 一个写线程执行，调用put方法；

2. 判断count是否为100，显然没有100；

3. 继续执行，存入值；

4. 判断当前写入的索引位置++后，是否和100相等，相等将写入索引值变为0，并将count+1；

5. 仅唤醒读线程阻塞队列中的一个；

6. 一个读线程执行，调用take方法；

7. ……

8. 仅唤醒写线程阻塞队列中的一个。

这就是多个Condition的强大之处，假设缓存队列中已经存满，那么阻塞的肯定是写线程，唤醒的肯定是读线程，相反，阻塞的肯定是读线程，唤醒的肯定是写线程，那么假设只有一个Condition会有什么效果呢，缓存队列中已经存满，这个Lock不知道唤醒的是读线程还是写线程了，如果唤醒的是读线程，皆大欢喜，如果唤醒的是写线程，那么线程刚被唤醒，又被阻塞了，这时又去唤醒，这样就浪费了很多时间

### 30 线程同步方法有哪些？

（1）同步方法

（2）同步代码块

（3）wait与notify

wait():使一个线程处于等待状态，并且释放所持有的对象的lock。

sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉InterruptedException异常。

notify():唤醒一个处于等待状态的线程，注意的是在调用此方法的时候，并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。

Allnotity():唤醒所有处入等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争。

（4）使用特殊域变量(volatile)实现线程同步

//需要同步的变量加上volatile

private volatile int account = 100;

a.volatile关键字为域变量的访问提供了一种免锁机制

b.使用volatile修饰域相当于告诉虚拟机该域可能会被其他线程更新

c.因此每次使用该域就要重新计算，而不是使用寄存器中的值

d.volatile不会提供任何原子操作，它也不能用来修饰final类型的变量

（5）ReentrantLock() : 创建一个ReentrantLock实例

lock() : 获得锁

unlock() : 释放锁

（6）ThreadLocal方法

a.ThreadLocal与同步机制都是为了解决多线程中相同变量的访问冲突问题。

b.前者采用以"空间换时间"的方法，后者采用以"时间换空间"的方式

（7）LinkedBlockingQueue阻塞队列实现线程同步

（8）原子操作就是指将读取变量值、修改变量值、保存变量值看成一个整体来操作即-这几种行为要么同时完成，要么都不完成。在java的util.concurrent.atomic包中提供了创建了原子类

型变量的工具类，使用该类可以简化线程同步。其中AtomicInteger 表可以用原子方式更新int的值，可用在应用程序中(如以原子方式增加的计数器)，但不能用于替换Integer；

可扩展Number，允许那些处理机遇数字类的工具和实用工具进行统一访问。

1. ConcurrentHashMap并发16个，桶的方式同步

### 31 Callable和Runnable不同点？

(1)Callable规定的方法是call()，Runnable规定的方法是run()。其中Runnable可以提交给Thread来包装下，直接启动一个线程来执行，而Callable则一般都是提交给ExecuteService来执行。

(2)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得

(3)call方法可以抛出异常，run方法不可以

(4)运行Callable任务可以拿到一个Future对象，c表示异步计算的结果。

### 32 Jdk中BlockingQueue有几个实现类？各自有什么特征？

@ArrayBlockingQueue：一个由数组支持的有界阻塞队列，规定大小的BlockingQueue,其构造函数必须带一个int参数来指明其大小.其所含的对象是以FIFO(先入先出)顺序排序的。

@LinkedBlockingQueue：大小不定的BlockingQueue,若其构造函数带一个规定大小的参数,生成的BlockingQueue有大小限制,若不带大小参数,所生成的BlockingQueue的大小由Integer.MAX\_VALUE来决定.其所含的对象是以FIFO(先入先出)顺序排序的。

@PriorityBlockingQueue:类似于LinkedBlockQueue,但其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序.

@SynchronousQueue:特殊的BlockingQueue,对其的操作必须是放和取交替完成的.

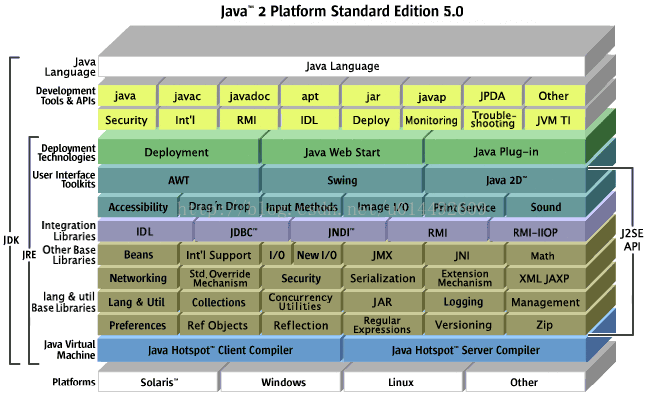
其中LinkedBlockingQueue和ArrayBlockingQueue比较起来,它们背后所用的数据结构不一样,导致LinkedBlockingQueue的数据吞吐量要大于ArrayBlockingQueue,但在线程数量很大时其性能的可预见性低于ArrayBlockingQueue.

### 33现在有1、2、3三个线程，你怎样保证2在1执行完后执行，3在2执行完后执行？

Java多线程中的join方法

### 34 JDK，JRE，jvm的区别和联系是什么？

我们利用JDK（调用JAVA API）开发了属于我们自己的JAVA程序后，通过JDK中的编译程序（javac）将我们的文本java文件编译成JAVA字节码，在JRE上运行这些JAVA字节码，JVM解析这些字节码，映射到CPU指令集或OS的系统调用。



### 35 int 和Integer 有什么区别?

int 是基本类型，直接存数值，进行初始化时int类的变量初始为0。

integer是对象，用一个引用指向这个对象，Integer的变量则初始化为null。

原始数据类型，分为boolean,byte,in,char,long,short,double,float

为了能够将这些基本数据类型当成对象操作，Java为每 一个基本数据类型都引入了对应的包装类型，int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Integer num = 9;

解析原因：归结于java对于Integer与int的自动装箱与拆箱的设计，是一种模式：叫享元模式（flyweight）。加大对简单数字的重利用，Java定义在自动装箱时对于值从–128到127之间的值，它们被装箱为Integer对象后，会存在内存中被重用，始终只存在一个对象。

### 36 解释内存中的栈（stack）、堆(heap)和静态存储区的用法？

Runtime Data Areas运行时数据区可以划分为6个区域：

PC 寄存器(Program Counter Register)，根据不同线程启动而被创建

JVM 栈(JVM Stacks)，根据不同线程启动而被创建，只保存基础数据类型的对象和自定义对象的引用，每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。

本地方法栈(Native Method Stacks)，根据不同线程启动而被创建

堆(Heap)，随JVM 启动而被创建，被所有线程公用，存储的是new出来的对象和数组，每个对象都包含一个与之对应的class的信息。(class的目的是得到操作指令)；

jvm只有一个堆区(heap)被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身，可能抛出OutOfMemoryError.堆是GC管理的主要区域。

一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。

分为新生代、老年代、永久代

在方法中去new一个对象，那这方法调用完毕后，对象就会被回收，这就是一个典型的新生代对象

在新生代中经历了N次垃圾回收后仍然存活的对象就会被放到老年代中。而且大对象直接进入老年代

永久代即方法区

方法区(Method Area)，随JVM 启动而被创建，被所有线程公用，又叫静态区，方法区包含所有的class和static变量。都是在整个程序中永远唯一的元素，如class，static变量。

字符串常量池放在方法区。将会抛出OutOfMemoryError

运行时常量池(Runtime Constant Pool)，随JVM 启动而被创建，被所有线程公用，每个类或接口的常量池在类和接口被JVM 创建时建立

### 37描述一下JVM 加载class文件的原理机制?

（1）JVM组成：

JVM 全称是Java Virtual Machine ，Java 虚拟机，也就是在计算机上再虚拟一个计算机，这和我们使用 VMWare不一样，那个虚拟的东西你是可以看到的，这个JVM 你是看不到的，它存在内存中。

JVM 存储就是内存了，我们写的所有类、常量、变量、方法都在内存中

整个JVM 分为四部分：

Class Loader 类加载器 ：Class Loader 只管加载，只要符合文件结构就加载，至于说能不能运行，则不是它负责的，那是由Execution Engine 负责的，

Execution Engine 执行引擎 ：负责解释命令，提交操作系统执行

Native Interface 本地接口：融合不同的编程语言为Java 所用

Runtime data area 运行数据区 ：

整个JVM 框架由加载器加载文件，然后执行器在内存中处理数据，需要与异构系统交互是可以通过本地接口进行。

（2）加载原理：

都需要由类加载器装载到JVM中才能运行。类加载器本身也是一个类，而它的工作就是把class文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，

像是反射，就需要显式的加载所需要的类。类装载方式，有两种

1.隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用类装载器加载对应的类到jvm中，

2.显式装载， 通过class.forname()等方法，显式加载需要的类

Java的类加载器有三个，对应Java的三种类:（java中的类大致分为三种： 1.系统类 2.扩展类 3.由程序员自定义的类 ）

委托模型机制的工作原理很简单：当类加载器需要加载类的时候，先请示其Parent(即上一层加载器)在其搜索路径载入，如果找不到，才在自己的搜索路径搜索该类。这样的顺序其实就是加载器层次上自顶而下的搜索，因为加

载器必须保证基础类的加载。之所以是这种机制，还有一个安全上的考虑：如果某人将一个恶意的基础类加载到jvm，委托模型机制会搜索其父类加载器，显然是不可能找到的，自然就不会将该类加载进来。

（3）加载步骤：

@装载:查找和导入class文件;

@连接:检查:检查载入的class文件数据的正确性;

准备:为类的静态变量分配存储空间;

解析:将符号引用转换成直接引用(这一步是可选的)

@初始化:初始化静态变量，静态代码块。

### 38 try{}里有一个return语句，那么紧跟在这个try后的finally{}里的code会不会被执行，什么时候被执行，在return前还是后?

在return前执行finally里面的语句。当运行到try或者catch里面的return语句时，[JVM](https://www.baidu.com/s?wd=JVM&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPvn3rjF9uWIBnAnLujbz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1fdPWD1nW0Y" \t "https://zhidao.baidu.com/question/_blank)会先去执行finally里面的语句，然后再返回执行return语句，当然，如果finally里面有return语句就从这里返回了，而不会去执行之前的return语句。

### 39 Iterator符合哪个设计模式？

迭代器模式，提供一个方法按顺序遍历一个集合内的元素，而又不需要暴露该对象的内部表示。访问一个聚合的对象，而不需要暴露对象的内部表示；支持对聚合对象的多种遍历；对遍历不同的对象，提供统一的接口。

## 三、前端开发

### 1 用过哪些前端框架？

Jquery, bootstrap,angularjs,extjs,easy ui

### 2 Ext和jquery有何特点和不同？如何解决ext的性能问题？

ExtJS是个html的UI库,以其漂亮的UI和强大的交互功能著称,为html提供了大量的控件支持,而且很优美

缺点:操作DOM对象的能力和灵活度不够

jQuery从名字可知,就是简化对DOM操作,以其精简灵活的语法著称,比如,要获取页面所有div对象,只要一行代码就可以了即$("div"),

缺点:UI设计不够强大，但是支持第三方的UI插件

1 减少要加载的东西不要动不动就ext-all.js，extjs是可以定制的，如果你用不到tree，就不要包含这个组件了。ExtJs的官网上有详细的定制方法和工具。

2 动态加载，参考使用ExtJs开发MIS系统（2）：Js的动态加载

3 内存泄露 Ext性能调优方案

4 将JS进行合并压缩。使用yahoo的yui-compress.jar进行压缩JS，去掉过多的空格和注释，并合并，减少IO的支出。

5 压缩ext

### 3 闭包用在那些场景，有什么特点？

闭包就是由函数创造的一个词法作用域，里面创建的变量被引用后，可以在这个词法环境之外自由使用。闭包通常用来创建内部变量，使得这些变量不能被外部随意修改，同时又可以通过指定的函数接口来操作。

保护函数内的变量安全：如迭代器、生成器。在内存中维持变量：如果缓存数据、柯里化。

### 4 js实现继承有几种方式？原型链prototype有何特点？

第一种，prototype的方式

第二种，apply的方式：

第三种，call+prototype的方式

### 5 js深度复制和浅度复制有什么区别？

1 在谈javascript的浅复制和深复制之前，我们有必要在来讨论下js的数据类型。我们都知道有Number，Boolean，String，Null，Undefined，Object五种类型。

而Object又包含Function，Array和Object自身。前面的五种类型叫做基本类型，而Object是引用类型。

2 js中基本类型的赋值为深复制，而引用类型的赋值为浅复制。

浅复制：就是把数据的地址赋值给对应变量，而没有把具体的数据复制给变量，变量会随数据值的变化而变化。

深复制：就是把数据赋值给对应的变量，从而产生一个与源数据不相干的新数据(数据地址已变化)

var a = "dengkunming";

var a1 = a;

alert(a1);//dengkunming

a="abc";

alert(a1);//dengkunming;

var a = [0,1,2,3];

var a1= a;

alert(a1);//[0,1,2,3]

a[1]="变";

alert(a1);//[0,"变",2,3]

3 浅复制不会随着存储数据地址的变化而变化，只会随着数据值的变化而变化。

var a = {w1:2,w2:3}

var a1= a;

alert(a1);//{w1:2,w2:3}

var a={x1:7,x2:8}

alert(a1);//{w1:2,w2:3}

按照我们上面的理论来讲，这里是浅复制。a1应该随着a的变化而变化呀，可在这里为什么会事与愿违了？这就是引用类型惹的祸了。对象赋值其实都是引用传值，传递的是一个地址。

那么实验五中的第四行其实就是把变量a指向了一个新的地址。而a1还是指向的原来那个地址，原来地址中的值没变，所以a1就不会变。所以请记住：浅复制不会随着存储数据地址的

变化而变化，只会随着数据值的变化而变化。

### 6 JSON和xml比较？

（1）可读性方面：基本相同，XML的可读性比较好；

　　（2）可扩展性方面：都具有良好的扩展性；

　　（3）编码难度方面：相对而言，JSON的编码比较容易；

　　（4）解码难度：JSON的解码难度基本为零，XML需要考虑子节点和父节点；

　　（5）数据体积方面：JSON相对于XML来讲，数据体积小，传递的速度比较快；

　　（6）数据交互方面：JSON与javascript的交互更加方便，更容易解析处理，更好的数据交互；

　　（7）数据描述方面：XML对数据描述性比较好；

（8）传输速度方面：JSON的速度远远快于XML。

### 7 this的用法有哪些？

1. 指向window全局变量

alert(this) //返回 [object Window]

function sayHello(){

alert(this);

}

sayHello();

2. 指向该对象（在全局里面this指向window,在某个对象里面this指向该对象，在闭包里面this指向window）

var user="the Window";

var box={

user:'the box',

getThis:function(){

return this.user;

},

getThis2:function(){

return function (){

return this.user;

}

}

};

alert(this.user);//the Window

alert(box.getThis());//the box

alert(box.getThis2()());

//the Window （由于使用了闭包，这里的this指向window）

alert(box.getThis2().call(box));

//the box 对象冒充（这里的this指向box对象）

3. 用apply,call改变函数的this指向

function sum(num1, num2){

return num1+num2;

}

function box(num1, num2){

return sum.apply(this, [num1, num2]);

//this 表示window的作用域 box冒充sum来执行

}

console.log(box(10,10)); //20

4. new 对象

function Person(){

console.log(this) //将 this 指向一个新建的空对象

}

var p = new Person();

### 8外边距内边距用CSS什么属性 ？

margin是指从自身边框到另一个容器边框之间的距离，就是容器外距离。

padding是指自身边框到自身内部另一个容器边框之间的距离，就是容器内距离。

### 9 jquery循环数组什么function?

$(array).each(function (idx)

{

var obj = this;

if(obj.name == 'eee')

{

return false;

}

return true;

});

### 10 怎么理解ext的mvc模式？

1，每个应用都有一个实体，就是Application对象，而每个应用同样采用单一入口结构，有个快捷函数就是Ext.application({config})，创建一个Application对象实例，并且运行它。Application在创建之初，会去加载Controller类

2，Application在lunch的时候，会创建一个Viewport实例，这个东西就像一个骨架一样，上面可以拼装各种View，具体说，就是各种布局形式和窗体控件，可以说是view界面的载体，一个页面只能有一个Viewport实例。

3，View纯粹是一个界面组件，或者说窗体控件的集合（比如form,grid和window）。它其实就是利用窗体控件panel,grid或form等进行用户界面展示,表格可以通过Ext.grid.GridPanel的getView()函数获取当前表格使用的视图实例,当我们希望在创建Gridview初始化一些参数可以使用Ext.grid.GridPanel的viewConfig参数,具体属性可以查看api.

通过Store来加载数据并且展现到界面上，界面控件的响应都写在Controller里面，View对Controller的存在全无所知，也没有代码上的依赖。

4，Controller的角色完全是个粘合剂，它在加载之初，会帮忙加载跟其有关的Model，Store，View类，而其真正的作用，是通过一系列的事件处理函数(比如点击保存按钮)，确定了每个View上面界面组件对用户交互行为的响应方法，可以说是一堆事件处理器函数的集合；这里面主要通过一个control成员函数来进行事件绑定，通过另一个叫ComponentQuery的组件，使用类似css selector的语法来定位界面上的组件，并为其绑定事件处理器。

5，Model是对抽象数据的具体化，简单理解就是数据库里面的一行记录。

6，Store是对通过网络加载数据的过程的一个抽象，Store通过data发送请求(一般为ajax请求)到后台获取数据(一般返回json格式)，Store依赖于Model，通过关联的Model对象才知道如何将取回的数据对象化。

不管是grid(表格),tree(树),form(表单)都可以通过model格式化字段,这样就可以把后台字段转化为Ext想要的字段,有时需要用到mapping属性

a.Model模型：模型是字段和它们的数据的集合，例如User模型带有username和password字段，模型知道如何持久化自己的数据，并且可以和其他模型关联，模型跟Ext JS 3 中的Record类有点像（区别是，Record只是单纯的扁平结构，而Model可以nest），通常都用在Store中去展示grid和其他组件的数据。

Store就是多个model实例的集合。这里的model实例相当于 Java中的对象，Model就相当于Class

Store如果看成是 一张数据库表，

record(s)就是 表中的记录 （行），

Model 代表（封装）了表的字段信息（属性）。

Ext.define('FWY.store.Students', {

extend: 'Ext.data.Store',

fields: ['id','name', 'age','sex'],

data: [

{id:1,name: '张三', age: 30,sex:'男'},

{id:2,name: '李四', age: 20,sex:'女'}

]

});

Ext.define('FWY.model.Student', {

extend: 'Ext.data.Model',

fields: ['id','name','age','sex']

});

b.View视图：视图是组件的一种，专注于界面展示 – grid, tree, panel 都是view。

Ext.define('FWY.view.student.List' ,{

extend: 'Ext.grid.Panel',

alias : 'widget.studentlist',

title : '学生信息列表',

initComponent: function() {

this.store = {

fields: ['id','name', 'age','sex'],

data : [

{id:1,name: 'zhangsan', age: 18,sex:'boy'},

{id:2,name: 'lishi', age: 20,sex:'girl'}

]};

this.columns = [

{header: '编号', dataIndex: 'id', flex: 1},

{header: '姓名', dataIndex: 'name', flex: 1},

{header: '年龄', dataIndex: 'age', flex: 1},

{header: '性别', dataIndex: 'sex', flex: 1}

];

this.callParent(arguments);

}

});

c.Controllers控制器：一个安放所有使你的app正确工作的代码的位置，具体一点应该是所有动作，例如如何渲染view，如何初始化model，和app的其他逻辑。

ExtJS 4 应用都遵循一个统一的目录结构，每个应有都相同 控制器是应用的粘合剂，它们所作的事情就是监听事件并执行动作，继续我们的应用，创建一个控制器。创建app/controller/Students.js这个文件，并添加如下代码Ext.define('FWY.controller.Students', {

extend: 'Ext.app.Controller',

views:

[

'student.List',

'student.Edit'

],

stores: ['Students'],//加载store

model: 'FWY.model.Student',

init: function() {

this.control({

'viewport > panel': {

render: this.onPanelRendered

}

});

},

editStudent: function(grid, record) {

console.log('Double clicked on ' + record.get('name'));

}

});

MVC中，所有类都放在app目录里面，这个目录可以有子目录，代表的是命名空间（一个子目录对应一个命名空间），使用不同的目录存放views,models,controllers,stores。当我们完成例子的时候，目录结构应该和下图一样

### 11 AngularJS有什么特性？解决什么问题？

A:四大特点：

　（1）MVC模式：

　　　　Model:数据,其实就是angular变量($scope.XX,$rootScope.XX);

　　　　View:数据的呈现,Html+Directive(指令);

　　　　Controller:操作数据,就是function,数据的增删改查;

　（2）双向绑定:

　　　　方向一:Model--->View

　　　　　　　　{{Model数据}} 或<XXX ng-xxx="Model数据"> Model变View跟着变;

　　　　方向二:View--->Model

　　　　　　　　<表单控件 ng-model="Model数据名"> View变Model跟着变;

　（3）依赖注入:

依赖注入(Dependency Injection,简称DI)是一种设计模式, 指某个对象依赖的其他对象无需手工创建，只需要“吼一嗓子”，则此对象在创建时，其依赖的对象由框架来自动创建并注入进来,其实就是最少知识法则;模块中所有的service和provider两类对象，都可以根据形参名称实现DI.

myModule.factory('$alert', function($window) {

return {

alert: function(text) {

$window.alert(text);

}

};

});

var myController = function($scope, $alert) {

$scope.message = function(msg) {

console.log(msg);

$alert.alert(msg);

};

};

myController.$inject = ['$scope', '$alert'];

　（4）模块化设计:高内聚低耦合法则

　　1)官方提供的模块 ng、ngRoute、ngAnimate、ngTouch

2)用户自定义的模块 angular.module('模块名',[ ])

B:优点集合

1 AngularJS是一套完整的框架，angular有自带的数据绑定、render渲染、angularUI库,过滤器,directive(模板),服务q(defer),http，inject(依赖注入),

factory,provider……，等等一系列工具，基本上只要你在做web开发用过的东西，它都有一个。但是这些东西React自身都没有。

2 angularjs的架构清晰，分工明确，扩展性良好，model，view，controller谁在什么时候做什么事情说的很清楚，angular能够让程序员真正专注于业务逻辑，

而且因为对html侵入不大，非常易于和designer协作。整个框架充满了DI的思路，耦合性非常低，对象都是被inject的，也就是说每个对象都可以轻易被替换而

不影响其他对象。

3 Angular生产效率高，单向数据流什么的想法非常好，但是写起来太麻烦！我只想变更个很简单的数据还要经过action、dispatcher、reduce、view四步

，angular里一行代码就搞定的事情在react里却如此麻烦

C：缺点集合

1 性能 ，双向数据绑定是一把双刃剑。

随着组件增加，项目越来越复杂，双向数据绑定带来性能问题。双向数据绑定是如何影响性能的？在JavaScript（ES5）中，并没有实现当变量或对象改变时发出通知的功能，Angular的实现方法被叫做“Dirty-checking（脏检查机制），通过跟踪数据的改变再动态更新用户界面（UI）。在Angular的作用域中任何操作的执行都会引发Dirty-checking，随着绑定数量的增加性能就会越低。

2 Angular 2.0推翻重做使得目前不宜采用此框架

Angular 1.x版本其实是个比较旧的东西了，现在看来有些理念过时了，比如依赖注入、自己独特的模块化，这些东西其实在ES6下已经很好地被解决了。

Angular的2.0几乎是一个推翻重做的框架，估计不会有1.X的upgrade方案。所以如果现在新开始的项目采用Angular的话，会是一个很尴尬的时机。同样，如此大的改动似乎也反面印证了1.X并不是那么好。

### 12 什么是模块化？AMD和CMD的区别？

1 模块化

随着浏览器功能越来越完善，前端已经不仅仅是切图做网站，前端在某些方面已经媲美桌面应用。越来越庞大的前端项目，越来越复杂的代码，前端开发者们对于模块化的需求空前强烈。随着浏览器功能越来越完善，前端已经不仅仅是切图做网站，前端在某些方面已经媲美桌面应用。越来越庞大的前端项目，越来越复杂的代码，前端开发者们对于模块化的需求空前强烈。

后来node出现了，跟随node出现的还有commonjs，这是一种js模块化解决方案，像Node.js主要用于服务器的编程，加载的模块文件一般都已经存在本地硬盘，所以加载起来比较快，不用考虑

异步加载的方式，CommonJS 加载模块是同步的，所以只有加载完成才能执行后面的操作。但是浏览器环境不同于Node，浏览器中获取一个资源必须要发送http请求，从服务器端获取，采用同

步模式必然会阻塞浏览器进程出现假死现象。在这方面dojo曾经做了伟大尝试，早期dojo便是采用xhr+eval的方式，结果可想而知，阻塞现象是必然的。后来出现无阻塞加载脚本方式在开发

中广泛应用，在此基础结合commonjs规范，前端模块化迎来了两种方案：AMD、CMD. 异步和管理依赖。

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出，CMD是SeaJS 在推广过程中被广泛认知。

2 AMD

Asynchronous Module Definition，用白话文讲就是 异步模块定义，对于 JSer 来说，异步是再也熟悉不过的词了，所有的模块将被异步加载，模块加载不影响后面语句运行

。所有依赖某些模块的语句均放置在回调函数中。AMD规范定义了一个自由变量或者说是全局变量 define 的函数。

第一个参数 id 为字符串类型，表示了模块标识，为可选参数。若不存在则模块标识应该默认定义为在加载器中被请求脚本的标识。如果存在，那么模块标识必须为顶层的或者一个绝对的标识。

第二个参数，dependencies ，是一个当前模块依赖的，已被模块定义的模块标识的数组字面量。

第三个参数，factory，是一个需要进行实例化的函数或者一个对象。

创建模块标识为 alpha 的模块，依赖于 require， export，和标识为 beta 的模块

3 CMD

通用模块定义（CMD）是Common Module Definition的缩写，是SeaJS 在推广过程中对模块定义的规范化产出。

RequireJS 和 SeaJS 都是模块化框架的代表，AMD和CMD，是他们各自定义模块化的方式，大同小异，主要是代码风格和API不同。

### 13 seajs和requirejs 区别？

（1）两者定位有差异。RequireJS 想成为浏览器端的模块加载器，同时也想成为 Rhino / Node 等环境的模块加载器。SeaJS 则专注于 Web 浏览器端，同时通过 Node 扩展的方式可以很方便跑在 Node 服务器端

（2）两者遵循的标准有差异。RequireJS 遵循的是 AMD（异步模块定义）规范，SeaJS 遵循的是 CMD （通用模块定义）规范。规范的不同，导致了两者 API 的不同。SeaJS 更简洁优雅。

（3）两者社区理念有差异。RequireJS 在尝试让第三方类库修改自身来支持 RequireJS，目前只有少数社区采纳。SeaJS 不强推，而采用自主封装的方式来“海纳百川”，目前已有较成熟的封装策略。

（4）两者代码质量有差异。RequireJS 是没有明显的 bug，SeaJS 是明显没有 bug。

（5）两者对调试等的支持有差异。SeaJS 通过插件，可以实现 Fiddler 中自动映射的功能，还可以实现自动 combo 等功能，非常方便便捷。RequireJS 无这方面的支持。

（6）两者的插件机制有差异。RequireJS 采取的是在源码中预留接口的形式，源码中留有为插件而写的代码。SeaJS 采取的插件机制则与 Node 的方式一致：开放自身.

### 14 对nodejs的理解？

nodejs是一个基于Chrome V8 引擎的JS运行环境，也就是让javascript运行在服务器（server）端，

NodeJS使用了一个事件驱动，非阻塞式的I/O模型，使得其轻量又高效。

Nodejs包管理器npm是全球最大的开源生态系统。

nodejs是服务端的js平台。

npm grunt express 等强大的代码与项目管理应用在nodeJS上。

关于NodeJS与传统服务器处理平台（Apache）的区别

Apache的多线程高并发模式

Apache是一种多线程处理并发，但是在一些大型的web应用上也会发生阻塞。

线程和进程

线程是可以独立运行的最小的CPU单位。

线程可以在同一个进程中并发运行，并共享该进程下的内存地址空间。

进程可以支持多个线程，它们看似同时执行，但是相互之间并不同步。

一个进程中的多个线程共享相同的内存地址空间，意味着可以访问相同的变量和对象，并且从同一堆中分配对象。

这样让线程之间共享信息变得容易，但是也要确保他们不会妨碍同一进程中想的其他线程。

NodeJS的异步I/O原理

用数据库的调用举例：

Apache，执行到第一个线程的时候会等待query返回结果，一方面会导致线程长期阻塞等待，另一方面会为了新请求不断增加线程，会浪费大量的资源，同时线程增加会占用大量的CPU时间来处理内存上下文切换。

NodeJS是异步单线程的，应用的是异步回调的方法，也就是异步的I/O。

解释：当进程执行的时候，不会等待结果的返回，而是直接执行下面的语句，直到进入事件循环，当数据库执行返回结果的时候会将事件发送到事件队列，等线程进入事件循环之后才会调用之前的回调函数。

也就是nodejs的工作原理其实就是事件循环。每一条nodejs的逻辑都是写在回调函数里面的，而回调函数都是返回之后才异步执行的。

NodeJS也会发生阻塞，但是阻塞发生在自己的单个线程当中，不是发生在后续回调的流程当中。

与php区别

用node来做网站开应用的是分块加载的模式，不用像php那样把所有的数据一次性加载到客户端。

NodeJS相比于php和Apache开的新线程来讲，节省了CPU内存和上下文切换的时间。

NodeJS的应用场景

NodeJs适合应用在具有大量的细小的http请求环境下，例如web的即时聊天程序，或者上万人同时在线的游戏服务器。不用考虑http请求次数过多的问题。

### 15 请你谈谈Cookie的弊端?

优点：

1.通过良好的编程，控制保存在cookie中的session对象的大小。

2.通过加密和安全传输技术（SSL），减少cookie被破解的可能性。

3.只在cookie中存放不敏感数据，即使被盗也不会有重大损失。

4.控制cookie的生命期，使之不会永远有效。偷盗者很可能拿到一个过期的cookie。

缺点：

1.`Cookie`数量和长度的限制。每个domain最多只能有20条cookie，每个cookie长度不能超过4KB，否则会被截掉。

2.安全性问题。如果cookie被人拦截了，那人就可以取得所有的session信息。即使加密也与事无补，因为拦截者并不需要知道cookie的意义，他只要原样转发cookie就可以达到目的了。

3.有些状态不可能保存在客户端。例如，为了防止重复提交表单，我们需要在服务器端保存一个计数器。如果我们把这个计数器保存在客户端，那么它起不到任何作用。

### 16 谈谈浏览器本地存储？

1) Cookie ： 广泛应用，局限明显。支持数据存储量相对较少，每个 domain 最多只能有 20 条 cookie ，每个 cookie 长度不能超过 4KB ，否则会被截掉；同时，存在安全性问题，如果被拦截，就可以取得所有的 session 信息。

2) Flash SharedObject ：使用的是 kissy 的 store 模块来调用 Flash SharedObject 。

优点：容量适中，基本上不存在兼容性问题

缺点：要在页面中引入特定的 Flash 和 JS ，增加额外负担，处理繁琐；还是有部分机子没有 flash 运行环境。

3) Google Gears ： Google 的离线方案，已经停止更新，官方推荐使用 HTML5 的 localStorage 方案。

4) User Data ： 是微软为 IE 专门在系统中开辟的一块存储空间，只支持 Windows+IE 的组合。单个文件的大小限制是 128KB ，一个域名下总共可以保存 1024KB 的文件，文件个数应该没有限制。在受限站点里这两个值分别是 64KB 和 640KB 。

（所以如果考虑到各种情况的话，单个文件最好能控制 64KB 以下。）

（实际测试 2000 （ IE5.5 ）、 XP （ IE6 、 IE7 ）， Vista （ IE7 ）下都是可正常使用。）

5) Web Storage

在较高版本的浏览器中， JS 提供了 sessionStorage 和 globalStorage 。

在 HTML5 中提供了 sessionStorage 和 localStorage 。

sessionStorage 用于本地存储一个会话（ session ）中的数据，这些数据只有在同一个会话中的页面才能访问，会话结束后数据随之销毁。因此 sessionStorage 不是一种持久化的本地存储，仅仅是会话级别的存储。

globalStorage 跨越会话存储数据。有特定访问限制，要指定哪些域可访问该数据。

localStorage 用于持久化的本地存储，除非主动删除数据，否则数据是永远不会过期的。不能给 localStorage 指定任何规则，要访问同一个 localStorage ，页面必须使用同一个域名，使用同一种协议，在同一个端口上。

优点：容量大、易用、强大、原生支持

缺点： a) 兼容性差（ Chrome, Safari, Firefox,Opera,IE8+ 支持 ， IE8 以下版本不支持）

b) 安全性差（所以请勿使用 localStorage 保存敏感信息）

### 17 web storage和cookie的区别？

1、cookie数据始终在同源的http请求中携带(即使不需要)，即cookie在浏览器和服务器间来回传递

2、cookie数据还有路径（path）的概念，可以限制。cookie只属于某个路径下、

3、存储大小限制也不同，cookie数据不能超过4K，同时因为每次http请求都会携带cookie，所以cookie只适合保存很小的数据，如回话标识

4、webStorage虽然也有存储大小的限制，但是比cookie大得多，可以达到5M或更大

5、数据的有效期不同

sessionStorage：仅在当前的浏览器窗口关闭有效；

localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；

cookie：只在设置的cookie过期时间之前一直有效，即使窗口和浏览器关闭

6、作用域不同

sessionStorage：不在不同的浏览器窗口中共享，即使是同一个页面；

localStorage：在所有同源窗口都是共享的；

cookie：也是在所有同源窗口中共享的

7、webStorage支持事件通知机制，可以将数据更新的通知发生给监听者

8、webStorage的API借口使用更方便 。setItem getItem clearItem

window。sessionStorage/window。 localStorage

setItem（key,val）设置

getItem（key）获取

webStorage。removeItem（key）删除单个

webStorage.clear（）清除所有

webStorage只能操作字符串对象，所有的存储值都会为字符串数据

### 18 display:none和visibility:hidden的区别？

display:none;会让元素完全从渲染树中消失，渲染的时候不占据任何空间；

visibility: hidden;不会让元素从渲染树消失，渲染师元素继续占据空间，只是内容不可见

display: none;是非继承属性，子孙节点消失由于元素从渲染树消失造成，通过修改子孙节点属性无法显示；

visibility: hidden;是继承属性，子孙节点消失由于继承了hidden，通过设置visibility: visible;可以让子孙节点显示；

修改常规流中元素的display通常会造成文档重排。修改visibility属性只会造成本元素的重绘。

读屏器不会读取display: none;元素内容；会读取visibility: hidden;元素内容

### 19 position的值， relative和absolute分别是相对于谁进行定位的？

1、static（静态定位）：默认值。没有定位，元素出现在正常的流中（忽略 top, bottom, left, right 或者 z-index 声明）。

2、relative（相对定位）：生成相对定位的元素，通过top,bottom,left,right的设置相对于其正常（原先本身）位置进行定位。可通过z-index进行层次分级。

3、absolute（绝对定位）：生成绝对定位的元素，相对于 static 定位以外的第一个父元素进行定位。元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。可通过z-index进行层次分级。

4、fixed（固定定位）：生成绝对定位的元素，相对于浏览器窗口进行定位。元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。可通过z-index进行层次分级

### 20 如何实现浏览器内多个标签页之间的通信?

本题主要考察数据存储的知识，数据存储有本地和服务器存储两种方式，对于前端开发来讲，只需要讲解用本地存储的方式来解决就好。当然也能知道服务器端的方式更好。本题的难易程度一般，只要能够说出思路就可以，至少说两种解决方法。

方法一：使用localStorage

使用localStorage.setItem(key,value);添加内容

使用storage事件监听添加、修改、删除的动作

window.addEventListener("storage",function(event){

$("#name").val(event.key+”=”+event.newValue);

});

方法二、使用cookie+setInterval

HTML代码

<inputidinputid="name"><input type="button" id="btnOK"value="发送">

JS代码-页面1

$(function(){

$("#btnOK").click(function(){

varname=$("#name").val();

document.cookie="name="+name;

});

});

JS代码-页面2

//获取Cookie天的内容

function getKey(key) {

return JSON.parse("{\""+ document.cookie.replace(/;\s+/gim,"\",\"").replace(/=/gim, "\":\"") +"\"}")[key];

}

//每隔1秒获取Cookie的内容

setInterval(function(){

console.log(getKey("name"));

},1000);

### 21 请说出三种减少页面加载时间的方法？

1、减少http请求（合并文件、合并图片）

2、优化图片文件，减小其尺寸，特别是缩略图，一定要按尺寸生成缩略图然后调用，不要在网页中用resize方法实现，虽然这样看到的图片外形小了，但是其加载的数据量一点也没减少。曾经见过有人在网页中加载的缩略图，其真实尺寸有10M之巨…普通图像、icon也要尽可能压缩后，可以采用web图像保存、减少颜色数等等方法实现。

3、图像格式的选择（GIF：提供的颜色较少，可用在一些对颜色要求不高的地方）

4、 压缩Javascript、CSS代码：一般js、css文件中存在大量的空格、换行、注释，这些利于阅读，如果能够压缩掉，将会很有利于网络传输。这方面的工具也有很多，可以在百度里搜索一下关键字“css代码压缩”，或者“js代码压缩”将会发现有很多网站都提供这样的功能，当然了你也可以自己写程序来做这个工作，如果你会的话。就拿我们这个网站来说吧。刚开始上传这个网站的时候，我的很多Css代码都没有压缩，后面发现了这个问题，我就上网找了相关的网站的压缩代码的功能，最后就把很多CSS文件都压缩了。这个压缩比率还是比较高的，一般都有百分五十左右。这个代码压缩对于网页的加载还是很有用的。

5、 服务器启用gzip压缩功能：将要传输的文件压缩后传输到客户端再解压，在网络传输 数据量会大幅减小。在服务器上的Apache、Nginx可直接启用，也可用代码直接设置传输文件头，增加gzip的设置，也可从 负载均衡设备直接设置。不过需要留意的是，这个设置会略微增加服务器的负担。服务器性能不是很好的网站，要慎重考虑。

6.标明高度和宽度（如果浏览器没有找到这两个参数，它需要一边下载图片一边计算大小，如果图片很多，浏览器需要不断地调整页面。这不但影响速度，也影响浏览体验。 当浏览器知道了高度和宽度参数后，即使图片暂时无法显示，页面上也会腾出图片的空位，然后继续加载后面的内容。从而加载时间快了，浏览体验也更好了。）

7、网址后面加上“/”:对服务器而言，不加斜杠服务器会多一次判断的过程，加斜杠就会直接返回网站设置的存放在网站根目录下的默认页面。

### 22 null和undefined的区别？

undefined表示变量声明但未初始化时的值，

null表示准备用来保存对象，还没有真正保存对象的值。从逻辑角度看，null值表示一个空对象指针。

JavaScript（ECMAScript标准）里共有5种基本类型：Undefined, Null, Boolean, Number, String，和一种复杂类型Object。可以看到null和undefined分属不同的类型，未初始化定义的值用typeof检测出来是"undefined"(字符串)，而null值用typeof检测出来是"object"（字符串）。

任何时候都不建议显式的设置一个变量为undefined，但是如果保存对象的变量还没有真正保存对象，应该设置成null。

实际上，undefined值是派生自null值的，ECMAScript标准规定对二者进行相等性测试要返回true，即

alert(null==undefined); // true

### 23 如何解决跨域问题?

1、JSONP：

使用方式就不赘述了，但是要注意JSONP只支持GET请求，不支持POST请求。

利用<script>标签没有跨域限制的“漏洞”（历史遗迹啊）来达到与第三方通讯的目的。当需要通讯时，本站脚本创建一个<script>元素，地址指向第三方的API网址，形如：

<script src="http://www.example.net/api?param1=1&param2=2"></script>

并提供一个回调函数来接收数据（函数名可约定，或通过地址参数传递）。

第三方产生的响应为json数据的包装（故称之为jsonp，即json padding），形如：

callback({"name":"hax","gender":"Male"})

这样浏览器会调用callback函数，

并传递解析后json对象作为参数。本站脚本可在callback函数里处理所传入的数据。

2、代理：

例如www.123.com/index.html需要调用www.456.com/server.php，可以写一个接口www.123.com/server.php，由这个接口在后端去调用www.456.com/server.php并拿到返回值，然后再返回给index.html，这就是一个代理的模式。相当于绕过了浏览器端，自然就不存在跨域问题。

3、PHP端修改header（XHR2方式）

在php接口脚本中加入以下两句即可：

header('Access-Control-Allow-Origin:\*');//允许所有来源访问

header('Access-Control-Allow-Method:POST,GET');//允许访问的方式

### 24 call() 和 apply() 的区别和作用？

JavaScript中的每一个Function对象都有一个apply()方法和一个call()方法，它们的语法分别为：

function.apply(thisObj[, argArray]);

function.call(thisObj[, arg1[, arg2[, [,...argN]]]]);

它们各自的定义：

apply：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.apply(A, arguments);即A对象应用B对象的方法。

call：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.call(A, args1,args2);即A对象调用B对象的方法。

它们的共同之处：

都“可以用来代替另一个对象调用一个方法，将一个函数的对象上下文从初始的上下文改变为由thisObj指定的新对象”。

它们的不同之处：

apply：最多只能有两个参数——新this对象和一个数组argArray。如果给该方法传递多个参数，则把参数都写进这个数组里面，当然，即使只有一个参数，也要写进数组里。如果argArray不是一个有效的数组或arguments对象，那么将导致一个TypeError。如果没有提供argArray和thisObj任何一个参数，那么Global对象将被用作thisObj，并且无法被传递任何参数。

call：它可以接受多个参数，第一个参数与apply一样，后面则是一串参数列表。这个方法主要用在js对象各方法相互调用的时候，使当前this实例指针保持一致，或者在特殊情况下需要改变this指针。如果没有提供thisObj参数，那么 Global 对象被用作thisObj。

### 25 你遇到过比较难的技术问题是？你是如何解决的？

略过

### 26 WEB应用从服务器主动推送Data到客户端有那些方式？

通常情况下,打开网页或app去查询或者刷新时，客户端向服务器发出请求然后返回数据,客户端与服务端对应的模式是: 客户端请求--服务端响应, 而在有些情况下,服务端会主动推送一些信息到客户端,例如:新闻的订阅,

天气的提醒等等,那么在这样的模式下,会有些问题值得思考:

1.应用服务器如何确定每一个应用所在的设备

2.服务端把消息推到哪，客户端又不像服务器有一个固定的地址

服务端主动推送到客户端是怎么一个过程?

结合一个实际问题分析下:

问题提出: 外卖app, 商家在商家后台需要实时的获取到有没有新订单，有的话是几个；这个需求类似与日常中使用QQ或者微信时的新信息提醒一样，只要有新信息就需要提醒

最近工作中遇到一个场景，商家在商家后台需要实时的获取到有没有新订单，有的话是几个；这个需求类似与日常中使用QQ或者微信时的新信息提醒一样，只要有新信息就需要提醒；商家基本在PC上使用，各式浏览器都有:如 IE系列（7.0，8.0，9.0及以上），chrome内核，firefox等；功能所属的部署在Tomcat 6.0上，如果技术需要可以部署到 Tomcat 7.0上;

我们先做做技术调研，这种浏览器与服务器实时通信的方式有哪些方式。

（1）AJAX轮询

这是我们最自然想到的。 采用常规AJAX轮询的方式，每10s或者30s轮询一次，既可以判断出有有多少个新订单进入，且这种时间间隔对于消息提醒也是可以接受的。这种技术方式实现起来非常简单，目前的机器都是可以机器的，前端浏览器也都支持。

但是这种方式会有非常严重的问题，就是需要不断的向服务器发送消息询问，如果有1w个商家打开了浏览器，采用10s轮询的方式，则服务器则会承担1000 的QPS，这1w个商家可能只有10个有订单通知；这种方式会对服务器造成极大的性能浪费。

还有一个类似的轮询是使用JSONP跨域请求的方式轮询，在实现起来有差别，但基本原理都是相同的，都是客户端不断的向服务器发起请求。

优点实现简单。

缺点这是通过模拟服务器发起的通信，不是实时通信，不顾及应用的状态改变而盲目检查更新，导致服务器资源的浪费，且会加重网络负载，拖累服务器。

（2）comet

Comet是一种用于Web的推送技术，能使服务器实时地将更新的信息传送到客户端，而无须客户端发出请求，目前有两种实现方式：

长轮询（long polling)

长轮询 (long polling) 是在打开一条连接以后保持，等待服务器推送来数据再关闭，可以采用HTTP长轮询和XHR长轮询两种方式。

HTTP 和JSONP方式的长轮询

把 script 标签附加到页面上以让脚本执行。服务器会挂起连接直到有事件发生，接着把脚本内容发送回浏览器，然后重新打开另一个 script 标签来获取下一个事件，从而实现长轮询的模型。

XHR长轮询

这种方式是使用比较多的长轮询模式。

客户端打开一个到服务器端的 AJAX 请求然后等待响应；服务器端需要一些特定的功能来允许请求被挂起，只要一有事件发生，服务器端就会在挂起的请求中送回响应并关闭该请求。客户端 JavaScript 响应处理函数会在处理完服务器返回的信息后，再次发出请求，重新建立连接；如此循环。

现在浏览器已经支持CROS的跨域方式请求，因此HTTP和JSONP的长轮询方式是慢慢被淘汰的一种技术，建议采用XHR长轮询。

长轮询优缺点

优点客户端很容易实现良好的错误处理系统和超时管理，实现成本与Ajax轮询的方式类似。

缺点需要服务器端有特殊的功能来临时挂起连接。当客户端发起的连接较多时，服务器端会长期保持多个连接，具有一定的风险。

（3）iframe

iframe 是很早就存在的一种 HTML 标记， 通过在 HTML 页面里嵌入一个隐蔵帧，然后将这个隐蔵帧的 SRC 属性设为对一个长连接的请求，服务器端就能源源不断地往客户端输入数据。

优点：这种方式每次数据传送不会关闭连接，连接只会在通信出现错误时，或是连接重建时关闭（一些防火墙常被设置为丢弃过长的连接， 服务器端可以设置一个超时时间， 超时后通知客户端重新建立连接，并关闭原来的连接）。

缺点IE、Morzilla Firefox 下端的进度栏都会显示加载没有完成，而且 IE 上方的图标会不停的转动，表示加载正在进行。

Google 的天才们使用一个称为“htmlfile”的 ActiveX 解决了在 IE 中的加载显示问题，并将这种方法用到了 gmail+gtalk 产品中。Alex Russell 在 “What else is burried down in the depth’s of Google’s amazing JavaScript?”文章中介绍了这种方法。Zeitoun 网站提供的 comet-iframe.tar.gz，封装了一个基于 iframe 和 htmlfile 的 JavaScript comet 对象，支持 IE、Mozilla Firefox 浏览器，可以作为参考。

我们常用的网页版的gtalk就是这种实现方式,Google的开发人员使使用一个称为“htmlfile”的 ActiveX 解决了在 IE 中的加载显示问题。

（4）Comet实现框架

CometD

CometD 框架是基于 HTTP 的事件驱动通信解决方案，使用了Bayeux通信协议，提供了一个 Java 服务器部件和一个 Java 客户端部件，还有一个基于 jQuery 和 Dojo 的 JavaScript 客户端库。

Bayeux 通信协议主要是基于 HTTP，提供了客户端与服务器之间的响应性双向异步通信。Bayeux 协议基于通道进行通信，通过该通道从客户端到服务器、从服务器到客户端或从客户端到客户端（但是是通过服务器）路由和发送消息。Bayeux 是一种 “发布- 订阅” 协议。

CometD 与三个传输协议绑定在一起：JSON、JSONP 和 WebSocket。他们都依赖于 Jetty Continuations 和 Jetty WebSocket API。在默认情况下，可以在 Jetty 6、Jetty 7、和 Jetty 8 中以及其他所有支持 Servlet 3.0 Specification 的服务中使用 CometD。

（5）服务器和内部构件

Atmosphere框架

Atmosphere提供了一个通用 API，以便使用许多 Web 服务器（包括 Tomcat、Jetty、GlassFish、Weblogic、Grizzly、JBossWeb、JBoss 和 Resin）的 Comet 和 WebSocket 特性。它支持任何支持 Servlet 3.0 Specification 的 Web 服务器。

Atmosphere 提供了一个 jQuery 客户端库，该库可以使连接设置变得更容易，它能够自动检测可以使用的最佳传输协议（WebSockets 或 CometD）。Atmosphere 的 jQuery 插件的用法与 HTML5 WebSockets API 相似。

（6）Pushlet

Pushlet 使用了观察者模型：客户端发送请求，订阅感兴趣的事件；服务器端为每个客户端分配一个会话 ID 作为标记，事件源会把新产生的事件以多播的方式发送到订阅者的事件队列里。

Pushlet 最后更新于2010年2月5号，之后至今没有再更新。

（7）websocket

WebSocket是HTML5开始提供的一种在单个 TCP 连接上进行全双工通讯的协议。WebSocket通讯协议于2011年被IETF定为标准RFC 6455，WebSocketAPI被W3C定为标准。在WebSocket API中，浏览器和服务器只需要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送。

浏览器支持

浏览器 版本支持

Chrome 4+

Firefox 4+

IE 10+

Opera 10+

Safari 5+

总结下来长轮询不是一个很好的方案，而且对于服务器而言是有风险的；另外支持WebSocket协议的浏览器都比较新，特比是IE需要10以上的版本；而我们的业务是面向于商家端，商家的浏览器版本相对较低，很多对WebSocket都不支持；相对而言Comet的方式比较适合，也有相应的实现框架，实现成本最低；因此最后我们还是决定使用Comet的方式来实现，后面上线运行一段时间之后再来给大家介绍。

### 27 你有哪些性能优化的方法？

（1） 减少http请求次数：CSS Sprites, JS、CSS源码压缩、图片大小控制合适；网页Gzip，CDN托管，data缓存，图片服务器。

（2） 前端模板 JS+数据，减少由于HTML标签导致的带宽浪费，前端用变量保存AJAX请求结果，每次操作本地变量，不用请求，减少请求次数

（3） 用innerHTML代替DOM操作，减少DOM操作次数，优化javascript性能。

（4） 当需要设置的样式很多时设置className而不是直接操作style。

（5） 少用全局变量、缓存DOM节点查找的结果。减少IO读取操作。

（6） 避免使用CSS Expression（css表达式)又称Dynamicproperties(动态属性)。

（7） 图片预加载，将样式表放在顶部，将脚本放在底部 加上时间戳。

### 28 一个页面从输入 URL 到页面加载显示完，此过程中都发生了什么？

分为4个步骤：

（1），当发送一个URL请求时，不管这个URL是Web页面的URL还是Web页面上每个资源的URL，浏览器都会开启一个线程来处理这个请求，同时在远程DNS服务器上启动一个DNS查询。这能使浏览器获得请求对应的IP地址。

（2）， 浏览器与远程Web服务器通过TCP三次握手协商来建立一个TCP/IP连接。该握手包括一个同步报文，一个同步-应答报文和一个应答报文，这三个报文在 浏览器和服务器之间传递。该握手首先由客户端尝试建立起通信，而后服务器应答并接受客户端的请求，最后由客户端发出该请求已经被接受的报文。

（3），一旦TCP/IP连接建立，浏览器会通过该连接向远程服务器发送HTTP的GET请求。远程服务器找到资源并使用HTTP响应返回该资源，值为200的HTTP响应状态表示一个正确的响应。

（4），此时，Web服务器提供资源服务，客户端开始下载资源。

请求返回后，便进入了我们关注的前端模块

简单来说，浏览器会解析HTML生成DOM Tree，其次会根据CSS生成CSS Rule Tree，而javascript又可以根据DOM API操作DOM

### 29 怎么理解sql注入原理和预防？

1.永远不要信任用户的输入，要对用户的输入进行校验，可以通过正则表达式，或限制长度，对单引号和双"-"进行转换等。

2.永远不要使用动态拼装SQL，可以使用参数化的SQL或者直接使用存储过程进行数据查询存取。

3.永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。

4.不要把机密信息明文存放，请加密或者hash掉密码和敏感的信息。

5.应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装，把异常信息存放在独立的表中。

### 30 请解释一下 JavaScript 的同源策略？

一、同源策略的产生

JS可以读取/修改网页的值。

一个浏览器中，打开一个银行网站和一个恶意网站，如果恶意网站能够对银行网站进行修改，那么就会很危险。

你打开了恶意网站和另一个网站，如果没有同源限制，该恶意网站就可以构造AJAX请求频繁在另一个网站发广告帖。

同源策略就是为了解决这类问题而出现的。

二、什么是同源策略（举例）

同源策略，即拥有相同的协议（protocol），端口（如果指定），主机（域名）的两个页面是属于同一个源。

然而在IE中比较特殊，IE中没有将端口号加入同源的条件中，因此上图中端口不同那一项，在IE中是算同源的

三、不遵循同源策略的标签

<script> <img> <iframe>中的src，href都可以任意链接网络资源，相当于对所要求的源进行了一次请求。

四、源继承

来自about:blank，javascript:和data:URLs中的内容，继承了将其载入的文档所指定的源，因为它们的URL本身未指定任何关于自身源的信息。

五、变更源

假设在 http://store.company.com/dir/other.html 中的一个脚本执行了下列语句：

document.domain = "company.com";

这条语句执行之后，页面将会成功地通过对 http://company.com/dir/page.html 的同源检测。而同理，company.com不能设置 document.domain 为 othercompany.com.

### 31 什么是 “use strict”? 使用它的好处和坏处分别是什么？

ECMAscript 5添加了第二种运行模式："严格模式"（strict mode）。顾名思义，这种模式使得Javascript在更严格的条件下运行。

设立"严格模式"的目的，主要有以下几个：

1. 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;

2. 消除代码运行的一些不安全之处，保证代码运行的安全；

3. 提高编译器效率，增加运行速度；

4. 为未来新版本的Javascript做好铺垫。

注：经过测试 IE6,7,8,9 均不支持严格模式。

缺点：现在网站的 JS 都会进行压缩，一些文件用了严格模式，而另一些没有。这时这些本来是严格模式的文件，被 merge 后，这个串就到了文件的中间，不仅没有指示严格模式，反而在压缩后浪费了字节。

### 32 哪些地方会出现css阻塞，哪些地方会出现js阻塞？

js 的阻塞特性：所有浏览器在下载 JS 的时候，会阻止一切其他活动，比如其他资源的下载，内容的呈现等等。直到 JS 下载、解析、执行完毕后才开始继续并行下载其他资源并呈现内容。为了提高用户体验，新一代浏览器都支持并行下载 JS，但是 JS 下载仍然会阻塞其它资源的下载(例如.图片，css文件等)。

　　由于浏览器为了防止出现 JS 修改 DOM 树，需要重新构建 DOM 树的情况，所以就会阻塞其他的下载和呈现。

　　嵌入 JS 会阻塞所有内容的呈现，而外部 JS 只会阻塞其后内容的显示，2 种方式都会阻塞其后资源的下载。也就是说外部样式不会阻塞外部脚本的加载，但会阻塞外部脚本的执行。

　　CSS 怎么会阻塞加载了?CSS 本来是可以并行下载的，在什么情况下会出现阻塞加载了(在测试观察中，IE6 下 CSS 都是阻塞加载)

　　当 CSS 后面跟着嵌入的 JS 的时候，该 CSS 就会出现阻塞后面资源下载的情况。而当把嵌入 JS 放到 CSS 前面，就不会出现阻塞的情况了。

　　根本原因：因为浏览器会维持 html 中 css 和 js 的顺序，样式表必须在嵌入的 JS 执行前先加载、解析完。而嵌入的 JS 会阻塞后面的资源加载，所以就会出现上面 CSS 阻塞下载的情况。

　　嵌入JS应该放在什么位置?

　　1. 放在底部，虽然放在底部照样会阻塞所有呈现，但不会阻塞资源下载。

　　2. 如果嵌入JS放在head中，请把嵌入JS放在CSS头部。

　　3. 使用 defer(只支持IE)

　　4. 不要在嵌入的JS中调用运行时间较长的函数，如果一定要用，可以用 setTimeout 来调用

　　Javascript无阻塞加载具体方式：

　　1. 将脚本放在底部。还是放在head中，用以保证在js加载前，能加载出正常显示的页面。

<script>

var script=document.createElement("script");

script.type="text/javascript";

script.src="file.js";

document.getElementsByTagName("head")[0].appendChild(script);

</script>

　　此技术的重点在于：无论在何处启动下载，文件额下载和运行都不会阻塞其他页面处理过程，即使在head里（除了用于下载文件的 http 链接）。

### 33 Javascript无阻塞加载具体方式有哪些？

1、将脚本放在底部

<link>还是在head中，用以保证在js加载前，能加载出正常显示的页面。

<script>放在</body>前。

2、成组脚本

由于每个<script>标签下载时阻塞页面解析过程，所以限制页面的<script>总数也可以改善性能。适用于内联脚本和外部脚本。

3、非阻塞脚本

等页面完成加载后，再加载js代码。也就是，在window.load事件发出后开始下载代码。

（1）defer属性：支持IE4和fierfox3.5更高版本浏览器

<script defer>...</script>

内联和外部文件

带defer属性的<script>可出现在文档的任何位置，对应的js文件将在<script>被解析时启动下载，但代码不会执行，直到DOM加载完毕（在onload事件句柄被调用之前）。所以实现了和也卖弄其他资源一起并行下载。

（2）动态脚本元素

文档对象模型（DOM）允许你使用js动态创建HTML的几乎全部文档内容。

复制代码 代码如下:

var script=document.createElement("script");

script.type="text/javascript";

script.src="file.js";

document.getElementByTagName\_r("head")[0].appendChild(script);

此技术的重点在于：无论在何处启动下载，文件额下载和运行都不会阻塞其他页面处理过程。即使在head里（除了用于下载文件的http链接）。

（3）The YUI3 approach

理念：用一个很小的初始代码，下载其余的功能代码，先引入文件：

复制代码 代码如下:

<script type="text/javascript src=http://files.jb51.net/file\_images/article/201306/yuanma/combo.js></script>

此种子文件大约10KB，

使用：

复制代码 代码如下:

YUI().use("dom",function(Y){

　　Y.Dom.addclass(...)

})

当所有代码可用时，回调函数被调用，YUI实例作为参数传入，就可以立即使用新下载的功能。

The LazyLoad library

使用：先引入：lazyload-min.js

(4)

复制代码 代码如下:

LazyLoad.js("a.js",function(){

Appliction.init();

})

多个文件：

复制代码 代码如下:

LazyLoad.js(["a.js","b.js"],function(){

Application.init();

})

(5)The LABjs library

先引入：lab.js

复制代码 代码如下:

$LAB.script("a.js").wait(function(){

Application.init();

})

多个文件，就链式写法

他的独特之处在于能够管理依赖关系。

可以通过wait()函数指定哪些文件应该等待其他文件。

例如：b.js的代码保不在a.js之前运行

复制代码 代码如下:

$LAB.script("a.js").wait().script("b.js").wait(function(){

Application.init();

})

这样，虽然两个文件是并行下载的，却能保证a.js能在b.js之前执行

### 34 事件、IE与火狐的事件机制有什么区别？ 如何阻止冒泡？

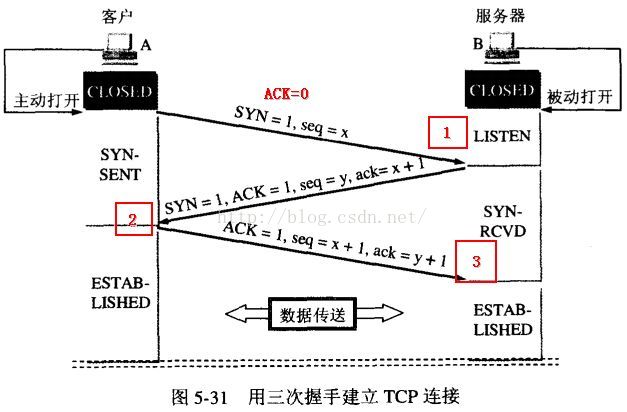
1.事件流描述的是从页面中接受事件的顺序，分为冒泡流和捕获流；

2.事件冒泡是指事件从最具体的元素接收，然后逐级向上传播，直到不具体的节点（通常指文档节点）；而事件捕获相反，它是从不具体的节点开始，逐步到最具体的节点；

3.IE的事件流是冒泡流，而火狐同时支持冒泡流和捕获流；

4.阻止事件冒泡：e.stopPropagation()，IE则是使用e.cancelBubble = true；

### 35 说说TCP传输的三次握手策略？



首先由Client发出请求连接即 SYN=1 ACK=0 (请看头字段的介绍), TCP规定SYN=1时不能携带数据，但要消耗一个序号,因此声明自己的序号是 seq=x

然后 Server 进行回复确认，即 SYN=1 ACK=1 seq=y, ack=x+1, 再然后 Client 再进行一次确认，但不用SYN 了，这时即为 ACK=1, seq=x+1, ack=y+1.

然后连接建立，为什么要进行三次握手呢（两次确认）。

建立三次握手主要是因为A发送了再一次的确认，那么A为什么会再确认一次呢，主要是为了防止已失效的连接请求报文段又突然传送给B，从而产生了错误。所谓“已失效的连接请求报文”是这样产生的，正常情况下，A发出连接请求，但是因为连接报文请求丢失而未收到确认，于是A再重传一次连接请求，后来收到了请求，并收到了确认，建立了连接，数据传输完毕后，就释放链接，A共发送了两次连接请求报文段，其中第一个丢失，第二个到达了B，没有“已失效的连接请求报文段”，但是还有异常情况下，A发送的请求报文连接段并没有丢失，而是在某个网络节点滞留较长时间，以致延误到请求释放后的某个时间到达B，本来是一个早已失效的报文段，但是B收到了此失效连接请求报文段后，就误以

为A又重新发送的连接请求报文段，并发送确认报文段给A，同意建立连接，如果没有三次握手，那么B发送确认后，连接就建立了，而此时A没有发送建立.

连接的请求报文段，于是不理会B的确认，也不会给B发送数据，而B却一直等待A发送数据，因此B的许多资源就浪费了，采用三次握手的方式就可以防止这.

种事情发生，例如刚刚，A不理会B，就不会给B发送确认，B收不到A的确认，就知道A不要求建立连接，就不会白白浪费资源.

### 36 Javascript垃圾回收方法？

一、垃圾回收的必要性

　　由于字符串、对象和数组没有固定大小，所有当他们的大小已知时，才能对他们进行动态的存储分配。JavaScript程序每次创建字符串、数组或对象时，解释器都必须分配内存来存储那个实体。只要像这样动态地分配了内存，最终都要释放这些内存以便他们能够被再用，否则，JavaScript的解释器将会消耗完系统中所有可用的内存，造成系统崩溃。

　　这段话解释了为什么需要系统需要垃圾回收，JS不像C/C++，他有自己的一套垃圾回收机制（Garbage Collection）。JavaScript的解释器可以检测到何时程序不再使用一个对象了，当他确定了一个对象是无用的时候，他就知道不再需要这个对象，可以把它所占用的内存释放掉了。例如：

var a = "before";

var b = "override a";

var a = b; //重写a

　　这段代码运行之后，“before”这个字符串失去了引用（之前是被a引用），系统检测到这个事实之后，就会释放该字符串的存储空间以便这些空间可以被再利用。

二、垃圾回收原理浅析

　　现在各大浏览器通常用采用的垃圾回收有两种方法：标记清除、引用计数。

1、标记清除

　　这是javascript中最常用的垃圾回收方式。当变量进入执行环境是，就标记这个变量为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到他们。当变量离开环境时，则将其标记为“离开环境”。

　　垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

关于这一块，建议读读Tom大叔的几篇文章，关于作用域链的一些知识详解，读完差不多就知道了，哪些变量会被做标记。

2、引用计数

　　另一种不太常见的垃圾回收策略是引用计数。引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变成0时，则说明没有办法再访问这个值了，因而就可以将其所占的内存空间给收回来。这样，垃圾收集器下次再运行时，它就会释放那些引用次数为0的值所占的内存。

但是用这种方法存在着一个问题，下面来看看代码：

复制代码

function problem() {

var objA = new Object();

var objB = new Object();

objA.someOtherObject = objB;

objB.anotherObject = objA;

}

　　在这个例子中，objA和objB通过各自的属性相互引用；也就是说这两个对象的引用次数都是2。在采用引用计数的策略中，由于函数执行之后，这两个对象都离开了作用域，函数执行完成之后，objA和objB还将会继续存在，因为他们的引用次数永远不会是0。这样的相互引用如果说很大量的存在就会导致大量的内存泄露。

　　我们知道，IE中有一部分对象并不是原生JavaScript对象。例如，其BOM和DOM中的对象就是使用C++以COM（Component Object

Model，组件对象）对象的形式实现的，而COM对象的垃圾回收器就是采用的引用计数的策略。因此，即使IE的Javascript引擎使用标记清除的策略来实现的，但JavaScript访问的COM对象依然是基于引用计数的策略的。说白了，只要IE中涉及COM对象，就会存在循环引用的问题。看看下面的这个简单的例子：

var element = document.getElementById("some\_element");

var myObj =new Object();

myObj.element = element;

element.someObject = myObj;

　　上面这个例子中，在一个DOM元素(element)与一个原生JavaScript对象(myObj)之间建立了循环引用。其中，变量myObj有一个名为element的属性指向element；而变量element有一个名为someObject的属性回指到myObj。由于循环引用，即使将例子中的DOM从页面中移除，内存也永远不会回收。

　　不过上面的问题也不是不能解决，我们可以手动切断他们的循环引用。

myObj.element = null;

element.someObject =null;

这样写代码的话就可以解决循环引用的问题了，也就防止了内存泄露的问题。

### 37 什么是Etag？

在HTTP1.1规范中，新增了一个HTTP头信息：ETag。对Web开发者来说，它是一个非常重要的信息。它是用作缓存使

用的两个主要的头信息之一 (另一个是Expires)。除此之外，在REST架构中，它还可以用于控制并发操作(上节中已经大

致介绍AtomPub中控制并发的流程)。那么ETag是什么？它又几种类型？强ETag与弱ETag之间有什么区别。？如何计算

ETag值?它与Last-Modified头信息在使用上有什么区别？本节主要围绕这几个方面叙述一下自己的理解。

### 38 关于Http 2.0 你怎么理解？

1、什么是HTTP 2.0

HTTP/2（超文本传输协议第2版，最初命名为HTTP 2.0），是HTTP协议的的第二个主要版本，使用于万维网。HTTP/2是HTTP协议自1999年HTTP 1.1发布后的首个更新，主要基于SPDY协议（是Google开发的基于TCP的应用层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验）。

2、与HTTP 1.1相比，主要区别包括

HTTP/2采用二进制格式而非文本格式

HTTP/2是完全多路复用的，而非有序并阻塞的——只需一个连接即可实现并行

使用报头压缩，HTTP/2降低了开销

HTTP/2让服务器可以将响应主动“推送”到客户端缓存中

## 四、数据库

### 1口述oracle和mysql分页的写法？

SELECT \* FROM

(

SELECT A.\*, ROWNUM RN

FROM (SELECT \* FROM TABLE\_NAME) A

WHERE ROWNUM <= 40

)

WHERE RN >= 21

Limit后的两个参数中，参数m是起始下标，它从0开始；参数n是返回的记录数。我们需要分页的话指定这两个值即可

### 2 SQL优化的思路？

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

2.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

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

3.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。

4.应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，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 Name = 'admin'

可以这样查询： select id from t where num = 10 union allselect id from t where Name = 'admin'

5.in和 not in 也要慎用，否则会导致全表扫描，如：

6.下面的查询也将导致全表扫描：

select id from t where name like ‘%abc%’

若要提高效率，可以考虑全文检索。

7.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

8.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 wherenum = 100\*2

9.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'

10.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。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12.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

13.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。

14.对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

15.select count(\*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。

16.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。

17.应尽可能的避免更新 clustered 索引数据列，因为 clustered 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 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

18.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。

19.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。

20.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。

21.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）。

22. 避免频繁创建和删除临时表，以减少系统表资源的消耗。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。

23.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。

24.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。

25.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。

26.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

29.尽量避免大事务操作，提高系统并发能力。

30.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。

### 3 sql函数decode和case when用法什么区别？

　１，DECODE Oracle 特有;

　２，CASE WHEN Oracle , SQL Server, MySQL 都可用;

　３，DECODE 只能用做相等判断,但是可以配合sign函数进行大于，小于，等于的判断,CASE 　可用于=,>=,<,<=,<>,is null,is not null 等的判断;

４，DECODE 使用其来比较简洁，CASE 虽然复杂但更为灵活;

### 4索引什么优缺点？

（1）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。

（2）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。

### 5建立索引规则，什么情况应该建立，什么情况不应该？

数据库建立索引常用的规则如下：

1、表的主键、外键必须有索引；

2、数据量超过300的表应该有索引；

3、经常与其他表进行连接的表，在连接字段上应该建立索引；

4、经常出现在Where子句中的字段，特别是大表的字段，应该建立索引；

5、索引应该建在选择性高的字段上；

6、索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引；

7、复合索引的建立需要进行仔细分析；尽量考虑用单字段索引代替：

A、正确选择复合索引中的主列字段，一般是选择性较好的字段；

B、复合索引的几个字段是否经常同时以AND方式出现在Where子句中？单字段查询是否 极少甚至没有？如果是，则可以建立复合索引；否则考虑单字段索引；

C、如果复合索引中包含的字段经常单独出现在Where子句中，则分解为多个单字段索引；

D、如果复合索引所包含的字段超过3个，那么仔细考虑其必要性，考虑减少复合的字段；

E、如果既有单字段索引，又有这几个字段上的复合索引，一般可以删除复合索引；

8、频繁进行数据操作的表，不要建立太多的索引；

9、删除无用的索引，避免对执行计划造成负面影响；

10、BLOB，CLOB字段肯定也不适合建索引。

11、索引不会包含有NULL值的列

12、般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “a%” 不会使用索引而like “aaa%”可以使用索引。

### 6 redis备份机制有哪2种？有什么区别？

RDB是在某个时间点将数据写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复。

优点：使用单独子进程来进行持久化，主进程不会进行任何IO操作，保证了redis的高性能

缺点：RDB是间隔一段时间进行持久化，如果持久化间redis发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候

这里说的这个执行数据写入到临时文件的时间点是可以通过配置来自己确定的，通过配置redis在n秒内如果超过m个key被修改这执行一次RDB操作。

这个操作就类似于在这个时间点来保存一次Redis的所有数据，一次快照数据。所有这个持久化方法也通常叫做snapshots。

Append-only file，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在append操作返回后(已经写入到文件或者即将写入)，才进行实际的数据变更，

“日志文件”保存了历史所有的操作过程；当server需要数据恢复时，可以直接replay此日志文件，即可还原所有的操作过程。AOF相对可靠，它和mysql中bin.log、

apache.log、zookeeper中txn-log简直异曲同工。AOF文件内容是字符串，非常容易阅读和解析。

优点：可以保持更高的数据完整性，如果设置追加file的时间是1s，如果redis发生故障，最多会丢失1s的数据；且如果日志写入不完整支持redis-check-aof来进行日志

修复；AOF文件没被rewrite之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的flushall）。

缺点：AOF文件比RDB文件大，且恢复速度慢。

我们可以简单的认为AOF就是日志文件，此文件只会记录“变更操作”(例如：set/del等)，如果server中持续的大量变更操作，将会导致AOF文件非常的庞大，意味着server失效后，

数据恢复的过程将会很长；事实上，一条数据经过多次变更，将会产生多条AOF记录，其实只要保存当前的状态，历史的操作记录是可以抛弃的；因为AOF持久化模式还伴生了“AOF rewrite”。

AOF的特性决定了它相对比较安全，如果你期望数据更少的丢失，那么可以采用AOF模式。如果AOF文件正在被写入时突然server失效，有可能导致文件的最后一次记录是不完整，你

可以通过手工或者程序的方式去检测并修正不完整的记录，以便通过aof文件恢复能够正常；同时需要提醒，如果你的redis持久化手段中有aof，那么在server故障失效后再次启动

前，需要检测aof文件的完整性。

### 7事物的传播特性有哪些，隔离级别有哪些？

事务的第一个方面是传播行为。传播行为定义关于客户端和被调用方法的事务边界。Spring定义了7中传播行为。

| **传播行为** | **意义** |
| --- | --- |
| PROPAGATION\_MANDATORY | 表示该方法必须运行在一个事务中。如果当前没有事务正在发生，将抛出一个异常 |
| PROPAGATION\_NESTED | 表示如果当前正有一个事务在进行中，则该方法应当运行在一个嵌套式事务中。被嵌套的事务可以独立于封装事务进行提交或回滚。如果封装事务不存在，行为就像PROPAGATION\_REQUIRES一样。 |
| PROPAGATION\_NEVER | 表示当前的方法不应该在一个事务中运行。如果一个事务正在进行，则会抛出一个异常。 |
| PROPAGATION\_NOT\_SUPPORTED | 表示该方法不应该在一个事务中运行。如果一个现有事务正在进行中，它将在该方法的运行期间被挂起。 |
| PROPAGATION\_SUPPORTS | 表示当前方法不需要事务性上下文，但是如果有一个事务已经在运行的话，它也可以在这个事务里运行。 |
| PROPAGATION\_REQUIRES\_NEW | 表示当前方法必须在它自己的事务里运行。一个新的事务将被启动，而且如果有一个现有事务在运行的话，则将在这个方法运行期间被挂起。 |
| PROPAGATION\_REQUIRES | 表示当前方法必须在一个事务中运行。如果一个现有事务正在进行中，该方法将在那个事务中运行，否则就要开始一个新事务。 |

**隔离级别**

声明式事务的第二个方面是隔离级别。隔离级别定义一个事务可能受其他并发事务活动活动影响的程度。另一种考虑一个事务的隔离级别的方式，是把它想象为那个事务对于事物处理数据的自私程度。

在一个典型的应用程序中，多个事务同时运行，经常会为了完成他们的工作而操作同一个数据。并发虽然是必需的，但是会导致一下问题：

脏读（Dirty read）-- 脏读发生在一个事务读取了被另一个事务改写但尚未提交的数据时。如果这些改变在稍后被回滚了，那么第一个事务读取的数据就会是无效的。

不可重复读（Nonrepeatable read）-- 不可重复读发生在一个事务执行相同的查询两次或两次以上，但每次查询结果都不相同时。这通常是由于另一个并发事务在两次查询之间更新了数据。

幻影读（Phantom reads）-- 幻影读和不可重复读相似。当一个事务（T1）读取几行记录后，另一个并发事务（T2）插入了一些记录时，幻影读就发生了。在后来的查询中，第一个事务（T1）就会发现一些原来没有的额外记录。

在理想状态下，事务之间将完全隔离，从而可以防止这些问题发生。然而，完全隔离会影响性能，因为隔离经常牵扯到锁定在数据库中的记录（而且有时是锁定完整的数据表）。侵占性的锁定会阻碍并发，要求事务相互等待来完成工作。

考虑到完全隔离会影响性能，而且并不是所有应用程序都要求完全隔离，所以有时可以在事务隔离方面灵活处理。因此，就会有好几个隔离级别

隔离级别 含义

ISOLATION\_DEFAULT 使用后端数据库默认的隔离级别。

ISOLATION\_READ\_UNCOMMITTED 允许读取尚未提交的更改。可能导致脏读、幻影读或不可重复读。

ISOLATION\_READ\_COMMITTED 允许从已经提交的并发事务读取。可防止脏读，但幻影读和不可重复读仍可能会发生。

ISOLATION\_REPEATABLE\_READ 对相同字段的多次读取的结果是一致的，除非数据被当前事务本身改变。可防止脏读和不可重复读，但幻影读仍可能发生。

ISOLATION\_SERIALIZABLE 完全服从ACID的隔离级别，确保不发生脏读、不可重复读和幻影读。这在所有隔离级别中也是最慢的，因为它通常是通过完全锁定当前事务所涉及的数据表来完成的。

### 8列举几种表连接方式,有什么区别？

（1）SQL几种常用连接方式（旧式写法）

一、NATURAL JOIN（自然连接）

两张表通过NATURAL JOIN连接的时候，相当于有个隐含的WHERE子句，对两张表中同名的对应列相比较看是否相等。

SQL> select \* from emp natural join dept;

返回14行数据 相当于select \* from emp , dept where emp.depno = dept.depno;

二、CROSS JOIN（创建笛卡尔积）

对两张表通过交叉联合产生第三张返回结果集的表。相当于普通的连接。如下返回56行=14\*4

select \* from emp cross join dept;

三、INNER JOIN（内连接）

内连接就相当于普通的CROSS JOIN，只是格式不一样，INNER JOIN在后面有一个ON子句（相当于WHERE）的搜索条件，用于过滤返回的行。

四、OUTER JOIN （外连接）

select \* from ta outer join tb on (ta.c1=tb.c1)

outer join告诉DBMS生成结果表，在此表中不仅带有相关(ta.c1=tb.c1)行对，而且还有来自两个源表中任一表的不匹配的行。

五、LEFT OUTER JOIN（左连接） RIGHT OUTER JOIN（右连接）

select \* from ta left outer join tb on (ta.c1=tb.c1)

select \* from ta right outer join tb on (ta.c1=tb.c1)

left outer join(left join) 告诉DBMS生成包括联合行和任何不匹配的行的结果表，但是不匹配的行系来自查询的FROM子句中LEFT OUTER JOIN关键词左边的表中。

right outer join(right join)与left outer join(left join)刚好相反。

（2）Oracle中常用新式写法

1. 相等连接

SELECT \* FROM EMP,DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO;

2. 外连接

◆左条件(+) = 右条件;

代表除了显示匹配相等连接条件的信息之外,还显示右条件所在的表中无法匹配相等连接条件的信息。

此时也称为"右外连接".另一种表示方法是:

SELECT ... FROM 表1 RIGHT OUTER JOIN 表2 ON 连接条件

◆左条件 = 右条件(+);

代表除了显示匹配相等连接条件的信息之外,还显示左条件所在的表中无法匹配相等连接条件的信息。

此时也称为"左外连接".

SELECT ... FROM 表1 LEFT OUTER JOIN 表2 ON 连接条件

3. 不等连接

两个表中的相关的两列进行不等连接,比较符号一般为>,<,...,BETWEEN.. AND..

SELECT EMPNO,ENAME,SAL,GRADE FROM SALGRADE,EMP

WHERE EMP.SAL BETWEEN LOSAL AND HISAL;

4. 自连接

自连接是数据库中经常要用的连接方式，使用自连接可以将自身表的一个镜像当作另一个表来对待，从而能够得到一些特殊的数据。

### 9 union和union all有什么不同?

UNION和UNION ALL关键字都是将两个结果集合并为一个，但这两者从使用和效率上来说都有所不同。

1、对重复结果的处理：UNION在进行表链接后会筛选掉重复的记录，Union All不会去除重复记录。

2、对排序的处理：Union将会按照字段的顺序进行排序；UNION ALL只是简单的将两个结果合并后就返回。

从效率上说，UNION ALL 要比UNION快很多，所以，如果可以确认合并的两个结果集中不包含重复数据且不需要排序时的话，那么就使用UNION ALL。

### 10 truncate与 delete区别？

TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同：

二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。

DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。

TRUNCATE,DELETE,DROP放在一起比较：

TRUNCATE TABLE：删除内容、释放空间但不删除定义。

DELETE TABLE:删除内容不删除定义，不释放空间。

DROP TABLE：删除内容和定义，释放空间。

### 11在一个字符串中搜索指定的字符,返回发现指定的字符的位置用什么函数？

INSTR(string,set[,start [,occurrence ] ] ) 如果指定start，oracle则跳过前面所有字符串到该位置开始搜索，occurence，

是强迫instr跳过前几次与字符串匹配，给出下一次匹配的位置，如果occurence指定3,那就是匹配第三次的位置了。

例 instr('ABACAAA','A',2,2) 从ABACAAA中匹配A这个字符串，从2个位置开始匹配，匹配第2次A所在的位置。PS：

如果set中不止有一个字符而是有几个字符组成的，则INSTR给出该字符集中的第一个字符的位置。

### 12 exists和in用法？

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

如果查询的两个表大小相当，那么用in和exists差别不大。

如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in：

例如：表A（小表），表B（大表）

1：

select \* from A where cc in (select cc from B) 效率低，用到了A表上cc列的索引；

select \* from A where exists(select cc from B where cc=A.cc) 效率高，用到了B表上cc列的索引。

相反的

2：

select \* from B where cc in (select cc from A) 效率高，用到了B表上cc列的索引；

select \* from B where exists(select cc from A where cc=B.cc) 效率低，用到了A表上cc列的索引。

not in 和not exists如果查询语句使用了not in 那么内外表都进行全表扫描，没有用到索引；而not extsts 的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

in 与 =的区别

select name from student where name in ('zhang','wang','li','zhao');

与

select name from student where name='zhang' or name='li' or name='wang' or name='zhao'

的结果是相同的。

### 13 [MYSQL复制原理及其流程](http://www.cnblogs.com/huixuexidezhu/p/7059202.html)？

1）在Slave 服务器上执行sart slave命令开启主从复制开关，开始进行主从复制。

2）此时，Slave服务器的IO线程会通过在master上已经授权的复制用户权限请求连接master服务器，并请求从执行binlog日志文件的指定位置（日志文件名和位置就是在配置主从复制服务时执行change master命令指定的）之后开始发送binlog日志内容

3）Master服务器接收到来自Slave服务器的IO线程的请求后，二进制转储IO线程会根据Slave服务器的IO线程请求的信息分批读取指定binlog日志文件指定位置之后的binlog日志信息，然后返回给Slave端的IO线程。返回的信息中除了binlog日志内容外，还有在master服务器端记录的新的binlog文件名称，以及在新的binlog中的下一个指定更新位置。

4）当Slave服务器的IO线程获取到Master服务器上IO线程发送的日志内容、日志文件及位置点后，会将binlog日志内容依次写到Slave端自身的Relay Log（即中继日志）文件（MySQL-relay-bin.xxx）的最末端，并将新的binlog文件名和位置记录到master-info文件中，以便下一次读取master端新binlog日志时能告诉Master服务器从新binlog日志的指定文件及位置开始读取新的binlog日志内容

5）Slave服务器端的SQL线程会实时检测本地Relay Log 中IO线程新增的日志内容，然后及时把Relay LOG 文件中的内容解析成sql语句，并在自身Slave服务器上按解析SQL语句的位置顺序执行应用这样sql语句，并在relay-log.info中记录当前应用中继日志的文件名和位置点

### 14 MyISAM和innodb 的区别有哪些？

1. InnoDB支持事务，MyISAM不支持，对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；

2. InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；

3. InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而MyISAM是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。

4. InnoDB不保存表的具体行数，执行select count(\*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；

5. Innodb不支持全文索引，而MyISAM支持全文索引，查询效率上MyISAM要高；

### 15 单个索引、联合索引、主键索引有何不同？

@ 索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。

普通索引(由关键字KEY或INDEX定义的索引)的唯一任务是加快对数据的访问速度。

@ 普通索引允许被索引的数据列包含重复的值。如果能确定某个数据列将只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字UNIQUE把它定义为一个唯一索引。也就是说，唯一索引可以保证数据记录的唯一性。

@ 主键，是一种特殊的唯一索引，在一张表中只能定义一个主键索引，主键用于唯一标识一条记录，使用关键字 PRIMARY KEY 来创建。

索引可以覆盖多个数据列，如像INDEX(columnA, columnB)索引，这就是联合索引。

@ 主键分为复合主键和联合主键

复合主键就是指你表的主键含有一个以上的字段组成 。

例如；

create table test

(

name varchar(19),

id number,

value varchar(10),

primary key (id,name)

)

上面的id和name字段组合起来就是你test表的复合主键 （若其一为单索引字段时，左边的id才会有索引）

它的出现是因为你的name字段可能会出现重名，所以要加上ID字段这样就可以保证你记录的唯一性

一般情况下，主键的字段长度和字段数目要越少越好

@ 联合主键，顾名思义就是多个主键联合形成一个主键组合，体现在联合。

(主键原则上是唯一的，别被唯一值所困扰。)

索引可以极大的提高数据的查询速度，但是会降低插入、删除、更新表的速度，因为在执行这些写操作时，还要操作索引文件。

### 16 MySql的存储引擎的不同？

1. MyISAM : 是旧版本mysql的默认引擎，现在默认引擎是InnoDB。MyISAM引擎的主要特点就是快，没有事务处理操作，也不支持外键操作。适合于多读取插入，少更新删除的操作表。存储数据分成三个文件：.frm(存储表定义) .MYD(存储数据) .MYI(存储索引)

用法： engine=myisam default charset=utf-8 ;

2.InnoDB ：是新版本mysql的默认引擎，支持事务处理和外键，但是其缺点几就是慢了些。存储方式分为两种：1.共享表空间存储。[.frm(表结构) 和 innodb\_data\_home(数据)和innodb\_data\_file\_path(索引)] 2.多表空间存储。 [.frm(表结构) 和 .idb（数据） ]。

适用于对于事务由较高要求的表的创建。

用法：engine=innodb default charset=utf-8 ;

3.MEMORY： 数据访问非常快的引擎，存储格式同一放在内存中的一个磁盘文件中格式是.frm 。默认使用hash索引。一旦服务器关闭表中的数据就会丢失。数据大小有限制。

用法：engine=memory ;

4.MERGE：本身是融合的意思，实质是MyISUM表的融合，这些融合的表必须结构完全相同。MERGE本身是没有数据的。插入操作可以是first或者是last。删除只是删除MERGE表定义，并不删除真正表的数据。存储方式：.frm(文件存储表定义信息) .MRG(描述组合表的信息，比如由哪些表组成，插入时的依据)。

适用于：将一系列等同的MyISAM表逻辑方式组合在一起，作为一个对象引用它们。

用法：engine=merge union=(\_\_,\_\_) insert\_method=last/first ;

### 17 Mysql怎么分表，以及分表后如果想按条件分页查询怎么办？

一、分表概述

1.如果只是为了分页，可以考虑这种分表，就是表的id是范围性的，且id是连续的，比如第一张表id是1到10万，第二张是10万到20万，这样分页应该没什么问题。

2.如果是其他的分表方式，建议用sphinx先建索引，然后查询分页，我们公司现在就是这样干的

Mysql分库分表方案

1.为什么要分表：

当一张表的数据达到几千万时，你查询一次所花的时间会变多，如果有联合查询的话，我想有可能会死在那儿了。分表的目的就在于此，减小数据库的负担，缩短查询时间。

mysql中有一种机制是表锁定和行锁定，是为了保证数据的完整性。表锁定表示你们都不能对这张表进行操作，必须等我对表操作完才行。行锁定也一样，别的sql必须等我对这条数据操作完了，才能对这条数据进行操作。

2. mysql proxy：amoeba

做mysql集群,利用amoeba。

从上层的java程序来讲，不需要知道主服务器和从服务器的来源，即主从数据库服务器对于上层来讲是透明的。可以通过amoeba来配置。

3.大数据量并且访问频繁的表，将其分为若干个表

比如对于某网站平台的数据库表-公司表，数据量很大，这种能预估出来的大数据量表，我们就事先分出个N个表，这个N是多少，根据实际情况而定。

某网站现在的数据量至多是5000万条，可以设计每张表容纳的数据量是500万条，也就是拆分成10张表，

那么如何判断某张表的数据是否容量已满呢？可以在程序段对于要新增数据的表，在插入前先做统计表记录数量的操作，当<500万条数据，就直接插入，当已经到达阀值，可以在程序段新创建数据库表（或者已经事先创建好），再执行插入操作。

4. 利用merge存储引擎来实现分表

如果要把已有的大数据量表分开比较痛苦，最痛苦的事就是改代码，因为程序里面的sql语句已经写好了。用merge存储引擎来实现分表, 这种方法比较适合.

举例子：

二、数据库架构

1、简单的MySQL主从复制:

MySQL的主从复制解决了数据库的读写分离，并很好的提升了读的性能，其图如下：

其主从复制的过程如下图所示：

但是，主从复制也带来其他一系列性能瓶颈问题：

1. 写入无法扩展

2. 写入无法缓存

3. 复制延时

4. 锁表率上升

5. 表变大，缓存率下降

那问题产生总得解决的，这就产生下面的优化方案，一起来看看。

2、MySQL垂直分区

如果把业务切割得足够独立，那把不同业务的数据放到不同的数据库服务器将是一个不错的方案，而且万一其中一个业务崩溃了也不会影响其他业务的正常进行，并且也起到了负载分流的作用，大大提升了数据库的吞吐能力。经过垂直分区后的数据库架构图如下：

然而，尽管业务之间已经足够独立了，但是有些业务之间或多或少总会有点联系，如用户，基本上都会和每个业务相关联，况且这种分区方式，也不能解决单张表数据量暴涨的问题，因此为何不试试水平分割呢？

3、MySQL水平分片（Sharding）

这是一个非常好的思路，将用户按一定规则（按id哈希）分组，并把该组用户的数据存储到一个数据库分片中，即一个sharding，这样随着用户数量的增加，只要简单地配置一台服务器即可，原理图如下：

三、分页方案

要是整体的分页显示那就更简单了

1、每个表的记录数是已知的，应在每次发生变化时记录到目录表中

2、无论是否排序（如果排序只是表的次序不同）至多会 union 两个分表

如假定共3个分表，记录数分别为 90，120，80 总记录数为 290

设分页是每页显示40条，则

第1页 表一的 1 到 40

第2页 表一的 41 到 80

第3页 表一的 81 到 90 + 表二的 1 到 30

第4页 表二的 31 到 70

第5页 表二的 71 到 110

第6页 表二的 111 到 120 + 表三的 1 到 30

### 18 MySql的主从实时备份同步的配置，以及原理(从库读主库的binlog)，读写分离？

一、主从同步概述

Mysql的Replication(复制)是一个异步的复制过程，从一个 Mysql instance(我们称之为 Master)复制到另一个Mysql instance(我们称之 Slave)。在 Master 与 Slave之间的实现整个复制过程主要由三个线程来完成，其中两个线程(Sql线程和IO线程)在 Slave 端，另外一个线程(IO线程)在Master端。

要实现 MySQL 的 Replication ，首先必须打开 Master 端的BinaryLog(mysql-bin.xxxxxx)功能，否则无法实现。因为整个复制过程实际上就是Slave从Master端获取该日志然后再在自己身上完全顺序的执行日志中所记录的各种操作。打开 MySQL 的 Binary Log 可以通过在启动 MySQL Server 的过程中使用“—log-bin” 参数选项，或者在 my.cnf 配置文件中的 mysqld 参数组([mysqld]标识后的参数部分)增加“log-bin” 参数项。

二、主从同步过程

MySQL 复制的基本过程如下：

1.Slave上面的IO线程连接上Master，并请求从指定日志文件的指定位置(或者从最开始的日志)之后的日志内容;

2.Master接收到来自Slave的IO线程的请求后，通过负责复制的IO线程根据请求信息读取指定日志指定位置之后的日志信息，返回给Slave端的 IO线程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息在Master端的Binary Log文件的名称以及在Binary Log中的位置;

3.Slave的IO线程接收到信息后，将接收到的日志内容依次写入到 Slave 端的RelayLog文件(mysql-relay-bin.xxxxxx)的最末端，并将读取到的Master端的bin-log的文件名和位置记录到master-info文件中，以便在下一次读取的时候能够清楚的告诉Master“我需要从某个bin-log的哪个位置开始往后的日志内容，请发给我”。

4.Slave的SQL线程检测到Relay Log中新增加了内容后，会马上解析该Log文件中的内容成为在Master 端真实执行时候的那些可执行的Query语句，并在自身执行这些Query。这样，实际上就是在Master端和Slave端执行了同样的Query，所以两端的数据是完全一样的。

三、 风险和性能

实际上，在老版本中，MySQL 的复制实现在 Slave 端并不是由 SQL 线程和 IO线程这两个线程共同协作而完成的，而是由单独的一个线程来完成所有的工作。但是 MySQL的工程师们很快发现，这样做存在很大的风险和性能问题，主要如下：

1.首先，如果通过一个单一的线程来独立实现这个工作的话，就使复制 Master 端的，BinaryLog日志，以及解析这些日志，然后再在自身执行的这个过程成为一个串行的过程，性能自然会受到较大的限制，这种架构下的Replication 的延迟自然就比较长了。

3.其次，Slave 端的这个复制线程从 Master 端获取 Binary Log 过来之后，需要接着解析这些内容，还原成Master 端所执行的原始 Query，然后在自身执行。在这个过程中，Master端很可能又已经产生了大量的变化并生成了大量的Binary Log 信息。如果在这个阶段 Master端的存储系统出现了无法修复的故障，那么在这个阶段所产生的所有变更都将永远的丢失，无法再找回来。这种潜在风险在Slave端压力比较大的时候尤其突出，因为如果 Slave压力比较大，解析日志以及应用这些日志所花费的时间自然就会更长一些，可能丢失的数据也就会更多。

所以，在后期的改造中，新版本的 MySQL 为了尽量减小这个风险，并提高复制的性能，将 Slave端的复制改为两个线程来完成，也就是前面所提到的 SQL 线程和 IO线程。最早提出这个改进方案的是Yahoo!的一位工程师“JeremyZawodny”。通过这样的改造，这样既在很大程度上解决了性能问题，缩短了异步的延时时间，同时也减少了潜在的数据丢失量。

当然，即使是换成了现在这样两个线程来协作处理之后，同样也还是存在 Slave数据延时以及数据丢失的可能性的，毕竟这个复制是异步的。只要数据的更改不是在一个事务中，这些问题都是存在的。

四、读写分离简介

对于很多大型网站（pv值百万、千万）来说，在所处理的业务中，其中有70%的业务是查询（select）相关的业务操作（新闻网站，插入一条新闻。查询操作），剩下的则是写（insert、update、delete，只要能对MySQL的数据造成更改的操作都叫写操作）操作。在使用负载均衡集群之后，可以很大程度的提升网站的整体性能，但是最终的数据处理的压力还是会落到MySQL数据库上，所有很有必要使用一些技术来提升MySQL的负载能力。（读写分离）

写专门交给写服务器处理（一般网站来说写是比较少的 读写比 4:1） 那么需要把读的任务分配多台服务器来完成的架构，就叫做读写分离。

五、读写分离实现方式

第一种：php程序上自己做逻辑判断，写php代码的时候，自己在程序上做逻辑判读写匹配。select，insert、update、delete做正则匹配，根据结果选择写服务器（主服务器）。如果是select操作则选择读服务器（从服务器器） mysql\_connect('读写的区分')

第二种：MySQL- Proxy是实现"读写分离(Read/Write Splitting)"的一个软件（MySQL官方提供 ，也叫中间件），基本的原理是让主数据库处理写操作（insert、update、delete），而从数据库处理查询操作（select）。而数据库的一致性则通过主从复制来实现。所以说主从复制是读写分离的基础。

注意：MySQL-proxy它能实现读写语句的区分主要依靠的是内部一个lua脚本（能实现读写语句的判断）。

注意：如果只在主服务器（写服务器）上完成数据的写操作话；这个时候从服务器上没有执行写操作，是没有数据的。这个时候需要使用另外一个技术来实现主从服务器的数据一致性，这个技术叫做 主从复制技术。所以说主从复制是读写分离的基础。

### 19 索引的数据结构，B+树？

一、数据库索引

是数据库管理系统中一个排序的数据结构以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

二、索引的优点

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。

三、索引的缺点

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。

四、应该在这些列上创建索引

1、在经常需要搜索的列上，可以加快搜索的速度；

2、在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；

3、在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；

4、在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

5、在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

6、在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

五、不应该创建索引的的这些列具有下列特点

1、对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

2、对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

3、对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

4、当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

六、B-树的特性

1.关键字集合分布在整颗树中；

2.任何一个关键字出现且只出现在一个结点中；

3.搜索有可能在非叶子结点结束；

4.其搜索性能等价于在关键字全集内做一次二分查找；

5.自动层次控制；

B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点。

七、B+树的特性：

1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

2.不可能在非叶子结点命中；

3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

4.更适合文件索引系统。

在B+Tree的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的B+Tree。做这个优化的目的是为了提高区间访问的性能。

B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

所以 B+树有两种搜索方法：

一种是按叶节点自己拉起的链表顺序搜索。

一种是从根节点开始搜索，和B树类似，不过如果非叶节点的关键码等于给定值，搜索并不停止，而是继续沿右指针，一直查到叶节点上的关键码。所以无论搜索是否成功，都将走完树的所有层。

B+ 树中，数据对象的插入和删除仅在叶节点上进行。

八、这两种处理索引的数据结构的不同之处：

1、B-树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而B+树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持B+树的平衡。

2、因为B-树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B+树相比来说是一种较好的折中。

3、B-树的查询效率与键在树中的位置有关，最大时间复杂度与B+树相同(在叶结点的时候)，最小时间复杂度为1(在根结点的时候)。而B+树的时候复杂度对某建成的树是固定的。

九、为什么选用B+、B-树

索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。

内存读取，内存是由一系列的存储单元组成的，每个存储单元存储固定大小的数据，且有一个唯一地址。当需要读内存时，将地址信号放到地址总线上传给内存，内存解析信号并定位到存储单元，然后把该存储单元上的数据放到数据总线上，回传。

写内存时，系统将要写入的数据和单元地址分别放到数据总线和地址总线上，内存读取两个总线的内容，做相应的写操作。

内存存取效率，跟次数有关，先读取A数据还是后读取A数据不会影响存取效率。而磁盘存取就不一样了，磁盘I/O涉及机械操作。磁盘是由大小相同且同轴的圆形盘片组成，磁盘可以转动(各个磁盘须同时转动)。磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不动，磁盘转动，但磁臂可以前后动，用于读取不同磁道上的数据。磁道就是以盘片为中心划分出来的一系列同心环(如图标红那圈)。磁道又划分为一个个小段，叫扇区，是磁盘的最小存储

磁盘读取时，系统将数据逻辑地址传给磁盘，磁盘的控制电路会解析出物理地址，即哪个磁道哪个扇区。于是磁头需要前后移动到对应的磁道，消耗的时间叫寻道时间，然后磁盘旋转将对应的扇区转到磁头下，消耗的时间叫旋转时间。所以，适当的操作顺序和数据存放可以减少寻道时间和旋转时间。

为了尽量减少I/O操作，磁盘读取每次都会预读，大小通常为页的整数倍。即使只需要读取一个字节，磁盘也会读取一页的数据(通常为4K)放入内存，内存与磁盘以页为单位交换数据。因为局部性原理认为，通常一个数据被用到，其附近的数据也会立马被用到。

### 20 事务的四个特性，以及各自的特点（原子、隔离）等等，项目怎么解决这些问题？

一．什么是事务

事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤消。也就是事务具有原子性，一个事务中的一系列的操作要么全部成功，要么一个都不做。

事务的结束有两种，当事务中的所以步骤全部成功执行时，事务提交。如果其中一个步骤失败，将发生回滚操作，撤消撤消之前到事务开始时的所以操作。

二．事务的 ACID

事务具有四个特征：原子性（ Atomicity ）、一致性（ Consistency ）、隔离性（ Isolation ）和持续性（ Durability ）。这四个特性简称为 ACID 特性。

1 、原子性

事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做

2 、一致性

事 务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统 运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是 不一致的状态。

3 、隔离性

一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

4 、持续性

也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

三、解决高并发锁的问题

1. 版本检查

在数据库中保留“版本”字段，跟随数据同时读写，以此判断数据版本。版本可能是时间戳或状态字段。

下例中的 WHERE 子句就实现了简单的版本检查：

UPDATE table SET status = 1 WHERE id=1 AND status = 0;

版本检查能够作为“乐观锁”，解决更新丢失的问题。

2. 锁

2.1 共享锁与排它锁

共享锁（Shared locks, S-locks）

共享锁又称读锁，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。

能给未加锁和添加了S锁的对象添加S锁。对象可以接受添加多把S锁。

如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据。

用法：

SELECT ... LOCK IN SHARE MODE;

在查询语句后面增加LOCK IN SHARE MODE，Mysql会对查询结果中的每行都加共享锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请共享锁，否则会被阻塞。其他线程也可以读取使用了共享锁的表，而且这些线程读取的是同一个版本的数据。

排它锁（Exclusive locks, X-locks）

排他锁又称写锁，如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的封锁。获得排他锁的事务既能读数据，又能修改数据。

只能给未加锁的对象添加X锁。对象只能接受一把X锁。加X锁的对象不能再加任何锁。

用法：

SELECT ... FOR UPDATE;

在查询语句后面增加FOR UPDATE，Mysql会对查询结果中的每行都加排他锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请排他锁，否则会被阻塞。

对于insert、update、delete，InnoDB会自动给涉及的数据加排他锁（X）；对于一般的Select语句，InnoDB不会加任何锁，事务可以通过以下语句给显示加共享锁或排他锁。

共享锁：SELECT ... LOCK IN SHARE MODE;

排他锁：SELECT ... FOR UPDATE;

2.2 意向锁

InnoDB还有两个表锁：

意向共享锁（IS）：表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁。

意向排他锁（IX）：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。

意向锁是InnoDB自动加的，不需要用户干预。

2.3 临时锁与持续锁

锁的时效性，指明了加锁生效期是到当前语句结束还是当前事务结束。

2.4 表级锁与行级锁

锁的粒度，指明了加锁的对象是当前表还是当前行。

2.5 悲观锁与乐观锁

悲观锁（Pessimistic Locking）

悲观锁假定当前事务操纵数据资源时，肯定还会有其他事务同时访问该数据资源，为了避免当前事务的操作受到干扰，先锁定资源。悲观锁需使用数据库的锁机制实现，如使用行级排他锁或表级排它锁。

尽管悲观锁能够防止丢失更新和不可重复读这类问题，但是它非常影响并发性能，因此应该谨慎使用。

乐观锁（Optimistic Locking）

乐观锁假定当前事务操纵数据资源时，不会有其他事务同时访问该数据资源，因此不在数据库层次上的锁定。乐观锁使用由程序逻辑控制的技术来避免可能出现的并发问题。

唯一能够同时保持高并发和高可伸缩性的方法就是使用带版本检查的乐观锁。

乐观锁不能解决脏读的问题，因此仍需要数据库至少启用“读已提交”的事务隔离级别。

3. 三级加锁协议

三级加锁协议也称为三级封锁协议,是为了保证正确的调度事务的并发操作,事务在对数据库对象加锁,解锁是必须遵守的一种规则。

3.1 一级加锁协议

事务在修改数据前必须加X锁，直到事务结束（事务结束包括正常结束(COMMIT)和非正常结束(ROLLBACK)）才可释放；如果仅仅是读数据，不需要加锁。

如下例：

SELECT xxx FOR UPDATE;

UPDATE xxx;

一级封锁协议可以防止丢失修改，并保证事务T是可恢复的。使用一级封锁协议可以解决丢失修改问题。

在一级封锁协议中，如果仅仅是读数据不对其进行修改，是不需要加锁的，它不能保证可重复读和不读“脏”数据。

3.2 二级加锁协议

满足一级加锁协议，且事务在读取数据之前必须先加S锁，读完后即可释放S锁。

二级封锁协议除防止了丢失修改，还可以进一步防止读“脏”数据。但在二级封锁协议中，由于读完数据后即可释放S锁，所以它不能保证可重复读。

3.3 三级加锁协议

满足一级加锁协议，且事务在读取数据之前必须先加S锁，直到事务结束才释放。

三级封锁协议除防止了丢失修改和不读“脏”数据外，还进一步防止了不可重复读。

上述三级协议的主要区别在于什么操作需要申请封锁，以及何时释放。

4. 两段锁协议（2-phase locking）

两段锁协议是指每个事务的执行可以分为两个阶段：生长阶段（加锁阶段）和衰退阶段（解锁阶段）。

加锁阶段：在该阶段可以进行加锁操作。在对任何数据进行读操作之前要申请并获得S锁，在进行写操作之前要申请并获得X锁。加锁不成功，则事务进入等待状态，直到加锁成功才继续执行。

解锁阶段：当事务释放了一个封锁以后，事务进入解锁阶段，在该阶段只能进行解锁操作不能再进行加锁操作。

若并发执行的所有事务均遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的。

遵循两段锁协议的事务调度处理的结果是可串行化的充分条件，但是可串行化并不一定遵循两段锁协议。

两段锁协议和防止死锁的一次封锁法的异同之处，一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行，因此一次封锁法遵守两段锁协议；但是两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议的事务可能发生死锁。

### 21 数据库的锁：行锁，表锁；乐观锁，悲观锁？

一、乐观锁

乐观锁不是数据库自带的，需要我们自己去实现。乐观锁是指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在操作数据时，并不进行任何其他的特殊处理（也就是不加锁），而在进行更新后，再去判断是否有冲突了。

通常实现是这样的：在表中的数据进行操作时(更新)，先给数据表加一个版本(version)字段，每操作一次，将那条记录的版本号加1。也就是先查询出那条记录，获取出version字段,如果要对那条记录进行操作(更新),则先判断此刻version的值是否与刚刚查询出来时的version的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，将version字段的值加1；如果更新时发现此刻的version值与刚刚获取出来的version的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。

举例：

下单操作包括3步骤：

1.查询出商品信息

select (status,status,version) from t\_goods where id=#{id}

2.根据商品信息生成订单

3.修改商品status为2

update t\_goods

set status=2,version=version+1

where id=#{id} and version=#{version};

除了自己手动实现乐观锁之外，现在网上许多框架已经封装好了乐观锁的实现，如hibernate，需要时，可能自行搜索"hiberate 乐观锁"试试看。

二、悲观锁

与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。

说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。

共享锁

共享锁指的就是对于多个不同的事务，对同一个资源共享同一个锁。相当于对于同一把门，它拥有多个钥匙一样。就像这样，你家有一个大门，大门的钥匙有好几把，你有一把，你女朋友有一把，你们都可能通过这把钥匙进入你们家，进去啪啪啪啥的，一下理解了哈，没错，这个就是所谓的共享锁。

刚刚说了，对于悲观锁，一般数据库已经实现了，共享锁也属于悲观锁的一种，那么共享锁在mysql中是通过什么命令来调用呢。通过查询资料，了解到通过在执行语句后面加上lock in share mode就代表对某些资源加上共享锁了。

比如，我这里通过mysql打开两个查询编辑器，在其中开启一个事务，并不执行commit语句

city表DDL如下：

[plain] view plain copy

CREATE TABLE `city` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) DEFAULT NULL,

`state` varchar(255) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=18 DEFAULT CHARSET=utf8;

begin;

SELECT \* from city where id = "1" lock in share mode;

然后在另一个查询窗口中，对id为1的数据进行更新

update city set name="666" where id ="1";

此时，操作界面进入了卡顿状态，过几秒后，也提示错误信息

[SQL]update city set name="666" where id ="1";

[Err] 1205 - Lock wait timeout exceeded; try restarting transaction

那么证明，对于id=1的记录加锁成功了，在上一条记录还没有commit之前，这条id=1的记录被锁住了，只有在上一个事务释放掉锁后才能进行操作，或用共享锁才能对此数据进行操作。

再实验一下：

update city set name="666" where id ="1" lock in share mode;

[Err] 1064 - You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'lock in share mode' at line 1

加上共享锁后，也提示错误信息了，通过查询资料才知道，对于update,insert,delete语句会自动加排它锁的原因

于是，我又试了试SELECT \* from city where id = "1" lock in share mode;

这下成功了。

三、排它锁

排它锁与共享锁相对应，就是指对于多个不同的事务，对同一个资源只能有一把锁。

与共享锁类型，在需要执行的语句后面加上for update就可以了

四、行锁

行锁，由字面意思理解，就是给某一行加上锁，也就是一条记录加上锁。

比如之前演示的共享锁语句

SELECT \* from city where id = "1" lock in share mode;

由于对于city表中,id字段为主键，就也相当于索引。执行加锁时，会将id这个索引为1的记录加上锁，那么这个锁就是行锁。

五、表锁

表锁，和行锁相对应，给这个表加上锁。

MyISAM引擎里有的，暂时研究了

六、共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

七、排他锁（X)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。

八、意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。

九、意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。

### 22 存储过程与函数的区别？

本质上没区别，执行的本质都一样。

只是函数有如：只能返回一个变量的限制。而存储过程可以返回多个。

函数是可以嵌入在sql中使用的,可以在select中调用，而存储过程要让sql的query 可以执行， 需要把 mysql\_real\_connect 的最后一个参数设置为CLIENT\_MULTI\_STATEMENTS。

函数限制比较多，比如不能用临时表，只能用表变量．还有一些函数都不可用等等．而存储过程的限制相对就比较少。

### 23 索引类型有哪些？

@ 从数据结构角度

1、B+树索引(O(log(n)))：关于B+树索引，可以参考 MySQL索引背后的数据结构及算法原理

2、hash索引：

a 仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询

b 其检索效率非常高，索引的检索可以一次定位，不像B-Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问，所以 Hash 索引的查询效率要远高于 B-Tree 索引

c 只有Memory存储引擎显示支持hash索引

3、FULLTEXT索引（现在MyISAM和InnoDB引擎都支持了）

4、R-Tree索引（用于对GIS数据类型创建SPATIAL索引）

@从物理存储角度

1、聚集索引（clustered index）

2、非聚集索引（non-clustered index）

@从逻辑角度

1、主键索引：主键索引是一种特殊的唯一索引，不允许有空值

2、普通索引或者单列索引

3、多列索引（复合索引）：复合索引指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用复合索引时遵循最左前缀集合

4、唯一索引或者非唯一索引

5、空间索引：空间索引是对空间数据类型的字段建立的索引，MYSQL中的空间数据类型有4种，分别是GEOMETRY、POINT、LINESTRING、POLYGON。

MYSQL使用SPATIAL关键字进行扩展，使得能够用于创建正规索引类型的语法创建空间索引。创建空间索引的列，必须将其声明为NOT NULL，空间索引只能在存储引擎为MYISAM的表中创建

CREATE TABLE table\_name[col\_name data type]

[unique|fulltext|spatial][index|key][index\_name](col\_name[length])[asc|desc]

### 24 数据库三范式是什么?

◆ 第一范式（1NF）：强调的是列的原子性，即列不能够再分成其他几列。

◆ 第二范式（2NF）：首先是 1NF，另外包含两部分内容，一是表必须有一个主键；二是没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。

◆ 第三范式（3NF）：首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

### 25 行转列、列换行怎么转？

一、场景

在生成报表的时候，很多时候对某列取值固定或者有限的几个值时，进行转列分析。如下：

facility列取值只有四种，每种一列，遇到相同facility求和。如下图：

二、实现方式

1 DECODE

利用DECODE函数、聚合函数SUM、GROUP BY分组实现。

WITH CO\_ORDER AS(

SELECT 'DOM1' Customer, 'ZHA01' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA01' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA02' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA03' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA04' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA01' facility , 6500 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA02' facility , 6000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA03' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA04' facility , 3000 TEU FROM dual

)

SELECT t.Customer, SUM(decode(t.facility, 'ZHA01', t.teu)) AS ZHA01,

SUM(decode(t.facility, 'ZHA02', t.teu)) AS ZHA02,

SUM(decode(t.facility, 'ZHA03', t.teu)) AS ZHA03,

SUM(decode(t.facility, 'ZHA04', t.teu)) AS ZHA04

FROM CO\_ORDER t

GROUP BY t.Customer;

该方式简单直观，当然每列具体要标示出来，所以只能处理固定种类的，不易程序后期扩展。

2 case

case函数的写法相对比较简洁，如下

WITH CO\_ORDER AS(

SELECT 'DOM1' Customer, 'ZHA01' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA01' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA02' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA03' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA04' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA01' facility , 6500 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA02' facility , 6000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA03' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA04' facility , 3000 TEU FROM dual

)

SELECT Customer,

sum (CASE WHEN facility = 'ZHA01' THEN TEU ELSE 0 END) AS ZHA01,

sum (CASE WHEN facility = 'ZHA02' THEN TEU ELSE 0 END) AS ZHA02,

sum (CASE WHEN facility = 'ZHA03' THEN TEU ELSE 0 END) AS ZHA03,

sum (CASE WHEN facility = 'ZHA04' THEN TEU ELSE 0 END) AS ZHA04

FROM CO\_ORDER GROUP BY Customer ORDER BY Customer ASC;

值得注意的是 case函数只返回第一个符合条件的值，剩下的Case部分将会被自动忽略

3 PIVOT

直接使用Oracle 关键字PIVOT。

WITH CO\_ORDER AS(

SELECT 'DOM1' Customer, 'ZHA01' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA01' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA02' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA03' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA04' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA01' facility , 6500 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA02' facility , 6000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA03' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA04' facility , 3000 TEU FROM dual

)

SELECT \* FROM (

SELECT t.Customer, t.facility, SUM(TEU) EU FROM CO\_ORDER t

GROUP BY t.Customer, t.facility

) t pivot(SUM(t.EU) FOR facility IN ('ZHA01', 'ZHA02', 'ZHA03', 'ZHA04'));

改方式更加的简便，最重要的是，分列条件是动态的，不需要考虑取值的可能性，利于扩展。不过，PIVOT是Oracle 11g后才出现的，所以使用时要注意环境。

另外，列名是动态的，所以在select中写列名报错，这点不知道怎么解决，有高手的话帮忙看下。

4 多列求和

转列之后，很多时候需要对同一行中某些列求和。

WITH CO\_ORDER AS(

SELECT 'DOM1' Customer, 'ZHA01' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA01' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA02' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA03' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA04' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA01' facility , 6500 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA02' facility , 6000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA03' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA04' facility , 3000 TEU FROM dual

)

select Customer,ZHA01,ZHA02,ZHA03,ZHA04,ZHA01+ZHA02+ZHA03 +ZHA04 as TOTAL from (

SELECT t.Customer, sum(decode(t.facility, 'ZHA01', t.teu)) AS ZHA01,

sum(decode(t.facility, 'ZHA02', t.teu)) AS ZHA02,

sum(decode(t.facility, 'ZHA03', t.teu)) AS ZHA03,

sum(decode(t.facility, 'ZHA04', t.teu)) AS ZHA04

FROM CO\_ORDER t

group by t.Customer);

5 WM\_CONCAT

该函数用于把列值以逗号分隔显示成一行。如下：

WITH CO\_ORDER AS(

SELECT 'DOM1' Customer, 'ZHA01' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA01' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA02' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA03' facility , 9000 TEU FROM dual UNION ALL

SELECT 'DOM1' Customer, 'ZHA04' facility , 4000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA01' facility , 6500 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA02' facility , 6000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA03' facility , 5000 TEU FROM dual UNION ALL

SELECT 'DOM2' Customer, 'ZHA04' facility , 3000 TEU FROM dual

)

SELECT Customer,WM\_CONCAT(facility) ALL\_FACILITY FROM CO\_ORDER GROUP BY Customer;

6 去重

例子中，(wb\_id, reply, addre) 作为表的unique key， 如果(wb\_id, reply, addre) 相同，保留字段TEU 最大值的记录。删除其他记录。

WITH tt AS (

SELECT 'DOM1' wb\_id, 'ZHA01' reply , 'ZHA01' addre ,4000 TEU FROM dual UNION ALL

SELECT 'DOM1' wb\_id, 'ZHA01' reply , 'ZHA01' addre ,5000 TEU FROM dual UNION ALL --- (wb\_id, reply, addre)与上条记录相同，不过 TEU 更大，保留。

SELECT 'DOM1' wb\_id, 'ZHA02' reply , 'ZHA01' addre ,9000 TEU FROM dual UNION ALL

SELECT 'DOM2' wb\_id, 'ZHA01' reply , 'ZHA01' addre ,9000 TEU FROM dual UNION ALL

SELECT 'DOM2' wb\_id, 'ZHA04' reply , 'ZHA01' addre ,4000 TEU FROM dual UNION ALL

SELECT 'DOM2' wb\_id, 'ZHA01' reply , 'ZHA01' addre ,6500 TEU FROM dual --- (wb\_id, reply, addre)与第四条记录相同，不过 TEU 小，删除。

)

-- 选出待删除的数据集合

SELECT \* FROM tt

WHERE (wb\_id, reply,addre) IN (SELECT wb\_id, reply, addre FROM tt GROUP BY wb\_id, reply, addre HAVING count(1) > 1)

AND (wb\_id, reply,reply, TEU) NOT IN (SELECT wb\_id, reply, addre, max(TEU) FROM tt group by wb\_id, reply, addre)

;

7 列转行

上面说到用 PRIVOT 进行行转列，现在看下 UNPRIVOT 操作。从名字上看，就是 PRIVOT 加前缀 UN，意义不言而喻，就是进行列转行。先看例子。

WITH CO\_ORDER2 AS(

SELECT 'DOM1' Customer, 'office1' ZHA02, 'office1' ZHA03, 'office2' ZHA04 FROM dual UNION ALL

SELECT 'DOM1' Customer, 'office2' ZHA02, 'office3' ZHA03, 'office2' ZHA04 FROM dual UNION ALL

SELECT 'DOM2' Customer, 'office4' ZHA02, 'office1' ZHA03, 'office2' ZHA04 FROM dual UNION ALL

SELECT 'DOM2' Customer, 'office3' ZHA02, 'office1' ZHA03, 'office1' ZHA04 FROM dual

)

--SELECT \* FROM CO\_ORDER2

SELECT Customer, offcs, facility FROM CO\_ORDER2 UNPIVOT (offcs FOR facility IN ( ZHA02, ZHA03, ZHA04) );

首先数据原始结构如下：

而利用 UNPRIVOT 处理后，结果如下：

UNPRIVOT 帮我们生成了两个新列，一列用于存放 office1 之类的值，另一个新列存的是 这个数据来源于之前那个原始列名。

其实它做的很简单，就是对每行原始数据的指定列的每个格子进行遍历，生成一条条数据。将原有的结构进行打平。因此，

处理后数据的行数 = 原始数据行数 \* 指定列数

因此上面 产生 了 12 条数据。

不过这也不是绝对的，当某个数据为空是，最终生成的数据会被 UNPRIVOT 给删除。看下面：

WITH CO\_ORDER2 AS(

SELECT 'DOM1' Customer, 'office1' ZHA02, 'office1' ZHA03, 'office2' ZHA04 FROM dual UNION ALL

SELECT 'DOM1' Customer, 'office2' ZHA02, '' ZHA03, 'office2' ZHA04 FROM dual UNION ALL

SELECT 'DOM2' Customer, 'office4' ZHA02, 'office1' ZHA03, 'office2' ZHA04 FROM dual UNION ALL

SELECT 'DOM2' Customer, null ZHA02, 'office1' ZHA03, 'office1' ZHA04 FROM dual

)

--SELECT \* FROM CO\_ORDER2

SELECT Customer, offcs, facility FROM CO\_ORDER2 UNPIVOT (offcs FOR facility IN ( ZHA02, ZHA03, ZHA04) );

此时下面只产生了 10 行数据。

在上面的 数据中，我们 给其中两个格子里面，一个为 空字符串，一个是 null。最终这两条数据没出现在结果集中。

### 26 mysql支持事务吗？

在缺省模式下，MYSQL是autocommit模式的，所有的数据库更新操作都会即时提交，所以在缺省情况下，mysql是不支持事务的。

但是如果你的MYSQL表类型是使用InnoDB Tables 或 BDB tables的话，你的MYSQL就可以使用事务处理,使用SET AUTOCOMMIT=0就可以使MYSQL允许在非autocommit模式，

在非autocommit模式下，你必须使用COMMIT来提交你的更改，或者用ROLLBACK来回滚你的更改。示例如下：

START TRANSACTION;

### 27 mysql 与其他数据库比较的特点？

MySQL是一个小型关系型数据库管理系统，开发者为瑞典MySQL AB公司，现在已经被Sun公司收购，支持FreeBSD、Linux、MAC、Windows等多种操作系统与其他的大型数据库例如Oracle、DB2、SQL Server等相比功能稍弱一些。其特点有：

1、可以处理拥有上千万条记录的大型数据；

2、支持常见的SQL语句规范；

3、可移植行高，安装简单小巧；

4、良好的运行效率，有丰富信息的网络支持；

5、调试、管理，优化简单（相对其他大型数据库）。

### 28 innodb中四种事务的隔离级别？

| **隔离级别** | **脏读（Dirty Read）** | **不可重复读（NonRepeatable Read）** | **幻读（Phantom Read）** |
| --- | --- | --- | --- |
| 未提交读（Read uncommitted） | 可能 | 可能 | 可能 |
| 已提交读（Read committed） | 不可能 | 可能 | 可能 |
| 可重复读（Repeatable read） | 不可能 | 不可能 | 可能 |
| 可串行化（SERIALIZABLE） | 不可能 | 不可能 | 不可能 |

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, 换句话说, 后续读取可以读到另一事务已提交的更新数据. 相反, “可重复读”在同一事务中多次读取数据时, 能够保证所读数据一样, 也就是后续读取不能读到另一事务已提交的更新数据。

幻读 :一个事务读到另一个事务已提交的insert数据

### 29 NVL与NVL2两个函数的使用方法和差别？

NVL2(expr1,expr2,expr3)

功能：如果参数表达式expr1值为NULL，则NVL2()函数返回参数表达式expr3的值；如果参数表达式expr1值不为NULL，则NVL2()函数返回参数表达式expr2的值。

需要注意的是value1和value2要保持字段类型相同。

NVL( string1, replace\_with)

功能：如果string1为NULL，则NVL函数返回replace\_with的值，否则返回string1的值，如果两个参数都为NULL ，则返回NULL。

需要注意的是参数value2 value3可以是除了LONG类型之外的任意数据类型

### 30 对字符串操作的函数?

一、字符转换函数

1、ASCII()

返回字符表达式最左端字符的ASCII 码值。在ASCII（）函数中，纯数字的字符串可不用‘’括起来，但含其它字符的字符串必须用‘’括起来使用，否则会出错。

2、CHAR()

将ASCII 码转换为字符。如果没有输入0 ~ 255 之间的ASCII 码值，CHAR（） 返回NULL 。

3、LOWER()和UPPER()

LOWER()将字符串全部转为小写；UPPER()将字符串全部转为大写。

4、STR()

把数值型数据转换为字符型数据。

STR (<float\_expression>[，length[， <decimal>]])

length 指定返回的字符串的长度，decimal 指定返回的小数位数。如果没有指定长度，缺省的length 值为10， decimal 缺省值为0。

当length 或者decimal 为负值时，返回NULL；

当length 小于小数点左边（包括符号位）的位数时，返回length 个\*；

先服从length ，再取decimal ；

当返回的字符串位数小于length ，左边补足空格。

二、去空格函数

1、LTRIM() 把字符串头部的空格去掉。

2、RTRIM() 把字符串尾部的空格去掉。

三、取子串函数

1、left()

LEFT (<character\_expression>， <integer\_expression>)

返回character\_expression 左起 integer\_expression 个字符。

2、RIGHT()

RIGHT (<character\_expression>， <integer\_expression>)

返回character\_expression 右起 integer\_expression 个字符。

3、SUBSTRING()

SUBSTRING (<expression>， <starting\_ position>， length)

返回从字符串左边第starting\_ position 个字符起length个字符的部分。

四、字符串比较函数

1、CHARINDEX()

返回字符串中某个指定的子串出现的开始位置。

CHARINDEX (<’substring\_expression’>， <expression>)

其中substring \_expression 是所要查找的字符表达式，expression 可为字符串也可为列名表达式。如果没有发现子串，则返回0 值。

此函数不能用于TEXT 和IMAGE 数据类型。

2、PATINDEX()

返回字符串中某个指定的子串出现的开始位置。

PATINDEX (<’%substring \_expression%’>， <column\_ name>)其中子串表达式前后必须有百分号“%”否则返回值为0。

与CHARINDEX 函数不同的是，PATINDEX函数的子串中可以使用通配符，且此函数可用于CHAR、 VARCHAR 和TEXT 数据类型。

五、字符串操作函数

1、QUOTENAME()

返回被特定字符括起来的字符串。

QUOTENAME (<’character\_expression’>[， quote\_ character]) 其中quote\_ character 标明括字符串所用的字符，缺省值为“[]”。

2、REPLICATE()

返回一个重复character\_expression 指定次数的字符串。

REPLICATE (character\_expression integer\_expression) 如果integer\_expression 值为负值，则返回NULL 。

3、REVERSE()

将指定的字符串的字符排列顺序颠倒。

REVERSE (<character\_expression>) 其中character\_expression 可以是字符串、常数或一个列的值。

4、REPLACE()

返回被替换了指定子串的字符串。

REPLACE (<string\_expression1>， <string\_expression2>， <string\_expression3>) 用string\_expression3 替换在string\_expression1 中的子串string\_expression2。

4、SPACE()

返回一个有指定长度的空白字符串。

SPACE (<integer\_expression>) 如果integer\_expression 值为负值，则返回NULL 。

5、STUFF()

用另一子串替换字符串指定位置、长度的子串。

STUFF (<character\_expression1>， <start\_ position>， <length>，<character\_expression2>)

如果起始位置为负或长度值为负，或者起始位置大于character\_expression1 的长度，则返回NULL 值。

如果length 长度大于character\_expression1 中 start\_ position 以右的长度，则character\_expression1 只保留首字符。

六、数据类型转换函数

1、CAST()

CAST (<expression> AS <data\_ type>[ length ])

2、CONVERT()

CONVERT (<data\_ type>[ length ]， <expression> [， style])

1）data\_type为SQL Server系统定义的数据类型，用户自定义的数据类型不能在此使用。

2）length用于指定数据的长度，缺省值为30。

3）把CHAR或VARCHAR类型转换为诸如INT或SAMLLINT这样的INTEGER类型、结果必须是带正号或负号的数值。

4）TEXT类型到CHAR或VARCHAR类型转换最多为8000个字符，即CHAR或VARCHAR数据类型是最大长度。

5）IMAGE类型存储的数据转换到BINARY或VARBINARY类型，最多为8000个字符。

6）把整数值转换为MONEY或SMALLMONEY类型，按定义的国家的货币单位来处理，如人民币、美元、英镑等。

7）BIT类型的转换把非零值转换为1，并仍以BIT类型存储。

8）试图转换到不同长度的数据类型，会截短转换值并在转换值后显示“+”，以标识发生了这种截断。

9）用CONVERT（） 函数的style 选项能以不同的格式显示日期和时间。style 是将DATATIME 和SMALLDATETIME 数据转换为字符串时所选用的由SQL Server 系统提供的转换样式编号，不同的样式编号有不同的输出格式。

七、日期函数

1、day(date\_expression)

返回date\_expression中的日期值

2、month(date\_expression)

返回date\_expression中的月份值

3、year(date\_expression)

返回date\_expression中的年份值

4、DATEADD()

DATEADD (<datepart>， <number>， <date>)

返回指定日期date 加上指定的额外日期间隔number 产生的新日期。参数“datepart” 取值如下：

5、DATEDIFF()

DATEDIFF (<datepart>， <date1>， <date2>)

返回两个指定日期在datepart 方面的不同之处，即date2 超过date1的差距值，其结果值是一个带有正负号的整数值。

6、DATENAME()

DATENAME (<datepart>， <date>)

以字符串的形式返回日期的指定部分此部分。由datepart 来指定。

7、DATEPART()

DATEPART (<datepart>， <date>)

以整数值的形式返回日期的指定部分。此部分由datepart 来指定。

DATEPART (dd， date) 等同于DAY (date)

DATEPART (mm， date) 等同于MONTH (date)

DATEPART (yy， date) 等同于YEAR (date)

8、GETDATE()

以DATETIME 的缺省格式返回系统当前的日期和时间

### 31 对于精通的数据库系统描述其数据一致性的保证机制,包括lock,事务一致性等？

对oracle 系统而言,描述sga的结构; 后台pmon,ckpt、lgwr,smon等进程的功能;表空间的分配策略; 回滚段的结构

oracle的sga（系统全局区）包括的主要区有：数据库缓存区，重做日志缓存区，共享池（数据字典缓存和库缓存），大池等。数据库缓存区用来存放最近使用过的数据块主要和后台进程中的数据库写进程(DBWR)以及数据文件发生关系；重做日志缓存区用于存放操作数据库数据所产生的重做日志信息，与之合作的有重做日志写进程(LGWR)和重做日志文件；共享池主要缓存SQL/PLSQL，资源锁，控制信息等，其中的库缓存主要缓存被解析执行过的SQL/PLSQL库缓存可分为共享SQL和私有SQL两个区，共享SQL用于存放SQL语句的语法分析结果和执行计划，私有SQL则用来存放与具体SQL语句执行有关的绑定变量，会话参数等。

ORACLE实例的另外一个重要部分就是其后台进程，主要的后台进程有：数据库写进程（DBWR），重做日志写进程（LGWR），系统监视器（SMON），进程监视器（PMON），检查点进程（CKPT）。DBWR主要是对数据库缓存区中的脏冷数据进行写入数据文件操作；LGWR主要是将对数据库数据操作所产生的重做日志信息写入到重做日志文件中；SMON完成由于非正常关闭数据库的情况下重起数据库时对数据库的恢复；PMON用来恢复失败的用户进程和服务进程，并释放其所占的系统资源；CKPT可以表示数据库在此出处于完整状态。

逻辑存储结构：数据块BLOCK，区EXTENT，段SEGMENT，表空间TABLESPACE

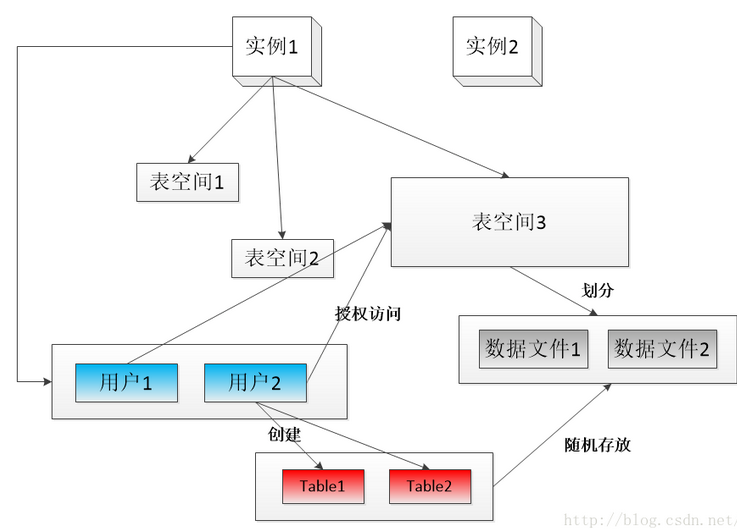
物理存储空间：表空间，数据文件，控制文件，日志文件，数据字典

软件体系结构就是上边对SGA和后台进程的描述。

### 32 一张表有10万条记录，如何删除其中的任意20条记录?请用SQL语句进行操作？

delete from ACT\_EVT\_LOG order by rand() limit 2

### 33 介绍一下oracle的体系结构?



Oracle的体系结构中有这么几个概念：实例、表空间、数据文件、用户、表。

Oracle数据库不同其他数据库，Oracle可以理解为一个大的数据库，就只有这一个。你所操作的“多个数据库”都是Oracle数据库下的不同的实例。安装Oracle的时候会默认给你创建一个实例orcl。

表空间是一个逻辑概念，一个数据库实例可以有多个表空间。可以说表空间是对这个数据库实例物理磁盘内存上的划分，也许你平时使用的时候并没有创建表空间，因为这个时候使用的是系统默认的表空间，后面详说。一个表空间会创建多个数据文件，这是个一对多的关系。

数据文件是用来存储数据的，这个顾名思义。是什么数据？平时接触最多的是我们平时创建的表，create table...但是表的概念还不能从这里引出。先说用户。

我看有的理解是一个表空间下有很多个用户，这个理解可以说是勉强正确。因为我觉得用户和表空间并不是上司与下属的关系，算是同级吧，都是隶属于数据库实例的。只不过在创建用户的时候都要给用户指定一个表空间，授权该用户可以访问的表空间，他们的关系是多对一，多个用户可以访问一个表空间，但一个用户只能访问一个表空间。（总感觉这句话逻辑不顺）如果没有指明表空间，则是默认表空间users表空间。而表只能由用户来创建，用户将这个表放入与用户绑定的表空间，但最后管理表的存放的确是这个表空间，表空间随机的将表存放到它所在的一个或多个数据文件中。

需要特别指出的是：

在创建用户时是只能指明一个默认表空间和一个默认临时表空间，你后续create table都是在这个默认表空间中，但是可以更改。

alter user xx default tablespace xxxx

说到这差不多结构已经出来了，不知道你有没有晕，画个图梳理一下吧。

### 34 谈谈对oracle的row\_ id是否理解?请简述?

ORACLE的row\_id是一个伪列，其个是为18个字节可将这18个字节用6363来划分，分别表示段编号，数据文件编号，数据块编号和记录编号。

Row\_id表示的是一个记录的物理存储地址。

### 35 如何判断游标已经到最后一行?

open my\_cursor;

fetch my\_cursor into last\_row;

while my\_cursor%found loop

fetch my\_cursor into current\_row;

if my\_cursor%notfound then

无条件插入last\_row;

else

编辑last\_row,插入;

last\_row := current\_row;

end if;

end loop;

close my\_cursor;

### 36 简述Oracle的归档与不归档工作模式，分别说明。

1 归档模式和非归档模式

在DBA部署数据库之初，必须要做出的最重要决定之一就是选择归档模式（ARCHIVELOG）或者非 归档模式（NOARCHIVELOG ）下运行数据库。我们知道，Oracle 数据库需要至少两组联机日志，每当一组 联机日志写满后会发生日志切换，继续向下一组联机日志写入。如果是归档模式，日志切换会触发归档进程 （ARCn）进行归档，生成归档日志。Oracle 保证归档完成前，联机日志不会被覆盖，如果是非归档模式， 则不会触发归档动作。

原文地址：http://blog.sunansheng.com/2016/04/29/archivelog-noarchivelog/

2 归档模式的优缺点

归档日志文件中保留了数据库的改动信息。

在这种模式下可以获得如下好处:

可以进行完全、不完全恢复：由于对数据库所做的全部改动都记录在日志文件中，如果发生硬盘故 障等导致数据文件丢失的话，则可以利用物理备份和归档日志完全恢复数据库，不会丢失任何数据。

可以进行联机热备：所谓联机热备，就是在数据库运行状态下，对数据库进行备份。备份时用户对 数据库的使用不受任何影响。

可以实施 Data Guard：可以部署 1 个或多个备用数据库，从而最大限度地提供灾难保护手段。

可以实施 Stream：利用 Stream 技术，可以实现最简单的单向复制到复杂的双向复制、多向复制， 提供更加灵活的数据冗余方案。

表空间可以脱机：可以备份部分数据库，比如重要的表空间。

能够增量备份：只需做一次完全备份，以后只备份发生改变的数据，可以提高备份速度。

更多的优化选项：随着 Oracle 版本升级，在联机热备方面不断有新的优化策略出现。

使用归档模式的缺点在于：

需要更多的磁盘空间保存归档日志；

DBA 会有更多的管理工作，包括维护归档空间、备份归档日志。

3 非归档模式的优缺点

非归档模式不生成归档日志，从数据安全角度来说，这种模式缺点是主要的，而优点可以忽略不计。

非归档模式的缺点包括：

只能进行脱机备份，也就是所谓的“ 冷备份”，和联机备份的“ 热备份” 相对应，数据库必须完全 关闭后备份，在备份过程中数据库不可用；

必须备份整个数据库，不能只备份部分数据库；

不能增量备份，对于 TB 级数据库（VLDB） ，这是一个非常大的缺点；

只能部分恢复，如果数据文件丢失需要恢复，DBA 只能恢复最后一次的完全备份，而之后的所有 数据库改变全部丢失。

非归档模式的优点包括：

DBA 的管理工作减少，因为非归档模式不产生归档日志，因此 DBA 不用考虑对归档的管理；

4 性能会有提升。

非归档模式转换成归档模式

数据库创建过程中需要指定归档和非归档模式，如果选择的是非归档模式，可以在数据库创建完成后 手工改变成归档模式，具体操作步骤如下。

（1 ）关闭数据库：

shutdown immediate;

（2 ）启动数据库到 mount 状态：

startup mount;

（3 ）修改数据库归档模式：

alter database archivelog;

（4 ）启动数据库：

alter database open;

（5 ）定义归档位置，也就是归档日志保存路径：

alter syste set log\_archive\_dest\_1="location=d:\oradata\example\archive" scope=both;

（6 ）确认配置生效：

archive log list;

## 五、高级开发

### 1分布式事物一致性如何保证？

在OLTP系统领域，我们在很多业务场景下都会面临事务一致性方面的需求，例如最经典的Bob给Smith转账的案例。传统的企业开发，系统往往是以单体应用形式存在的，也没有横跨多个数据库。我们通常只需借助开发平台中特有数据访问技术和框架（例如Spring、JDBC、ADO.NET），结合关系型数据库自带的事务管理机制来实现事务性的需求。关系型数据库通常具有ACID特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

而大型互联网平台往往是由一系列分布式系统构成的，开发语言平台和技术栈也相对比较杂，尤其是在SOA和微服务架构盛行的今天，一个看起来简单的功能，内部可能需要调用多个“服务”并操作多个数据库或分片来实现，情况往往会复杂很多。单一的技术手段和解决方案，已经无法应对和满足这些复杂的场景了。

分布式系统的特性

对分布式系统有过研究的读者，可能听说过“CAP定律”、“Base理论”等，非常巧的是，化学理论中ACID是酸、Base恰好是碱。这里笔者不对这些概念做过多的解释，有兴趣的读者可以查看相关参考资料。CAP定律如下图：

在分布式系统中，同时满足“CAP定律”中的“一致性”、“可用性”和“分区容错性”三者是不可能的，这比现实中找对象需同时满足“高、富、帅”或“白、富、美”更加困难。在互联网领域的绝大多数的场景，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

分布式事务

提到分布式系统，必然要提到分布式事务。要想理解分布式事务，不得不先介绍一下两阶段提交协议。先举个简单但不精准的例子来说明：

第一阶段，张老师作为“协调者”，给小强和小明（参与者、节点）发微信，组织他们俩明天8点在学校门口集合，一起去爬山，然后开始等待小强和小明答复。

第二阶段，如果小强和小明都回答没问题，那么大家如约而至。如果小强或者小明其中一人回答说“明天没空，不行”，那么张老师会立即通知小强和小明“爬山活动取消”。

细心的读者会发现，这个过程中可能有很多问题的。如果小强没看手机，那么张老师会一直等着答复，小明可能在家里把爬山装备都准备好了却一直等着张老师确认信息。更严重的是，如果到明天8点小强还没有答复，那么就算“超时”了，那小明到底去还是不去集合爬山呢？

这就是两阶段提交协议的弊病，所以后来业界又引入了三阶段提交协议来解决该类问题。

两阶段提交协议在主流开发语言平台，数据库产品中都有广泛应用和实现的，下面来介绍一下XOpen组织提供的DTP模型图：

XA协议指的是TM（事务管理器）和RM（资源管理器）之间的接口。目前主流的关系型数据库产品都是实现了XA接口的。JTA(Java Transaction API)是符合X/Open DTP模型的，事务管理器和资源管理器之间也使用了XA协议。 本质上也是借助两阶段提交协议来实现分布式事务的，下面分别来看看XA事务成功和失败的模型图：

在JavaEE平台下，WebLogic、Webshare等主流商用的应用服务器提供了JTA的实现和支持。而在Tomcat下是没有实现的（其实笔者并不认为Tomcat能算是JavaEE应用服务器），这就需要借助第三方的框架Jotm、Automikos等来实现，两者均支持spring事务整合。

而在Windows .NET平台中，则可以借助ado.net中的TransactionScop API来编程实现，还必须配置和借助Windows操作系统中的MSDTC服务。如果你的数据库使用的mysql，并且mysql是部署在Linux平台上的，那么是无法支持分布式事务的。 由于篇幅关系，这里不展开，感兴趣的读者可以自行查阅相关资料并实践。

总结：这种方式实现难度不算太高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况。但分布式事务对性能的影响会比较大，不适合高并发和高性能要求的场景。

提供回滚接口

在服务化架构中，功能X，需要去协调后端的A、B甚至更多的原子服务。那么问题来了，假如A和B其中一个调用失败了，那可怎么办呢？

在笔者的工作中经常遇到这类问题，往往提供了一个BFF层来协调调用A、B服务。如果有些是需要同步返回结果的，我会尽量按照“串行”的方式去调用。如果调用A失败，则不会盲目去调用B。如果调用A成功，而调用B失败，会尝试去回滚刚刚对A的调用操作。

当然，有些时候我们不必严格提供单独对应的回滚接口，可以通过传递参数巧妙的实现。

这样的情况，我们会尽量把可提供回滚接口的服务放在前面。举个例子说明：

我们的某个论坛网站，每天登录成功后会奖励用户5个积分，但是积分和用户又是两套独立的子系统服务，对应不同的DB，这控制起来就比较麻烦了。解决思路：

把登录和加积分的服务调用放在BFF层一个本地方法中。

当用户请求登录接口时，先执行加积分操作，加分成功后再执行登录操作

如果登录成功，那当然最好了，积分也加成功了。如果登录失败，则调用加积分对应的回滚接口（执行减积分的操作）。

总结：这种方式缺点比较多，通常在复杂场景下是不推荐使用的，除非是非常简单的场景，非常容易提供回滚，而且依赖的服务也非常少的情况。

这种实现方式会造成代码量庞大，耦合性高。而且非常有局限性，因为有很多的业务是无法很简单的实现回滚的，如果串行的服务很多，回滚的成本实在太高。

本地消息表

这种实现方式的思路，其实是源于ebay，后来通过支付宝等公司的布道，在业内广泛使用。其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。如果不考虑性能及设计优雅，借助关系型数据库中的表即可实现。

举个经典的跨行转账的例子来描述。

第一步伪代码如下，扣款1W，通过本地事务保证了凭证消息插入到消息表中。

第二步，通知对方银行账户上加1W了。那问题来了，如何通知到对方呢？

通常采用两种方式：

采用时效性高的MQ，由对方订阅消息并监听，有消息时自动触发事件

采用定时轮询扫描的方式，去检查消息表的数据。

两种方式其实各有利弊，仅仅依靠MQ，可能会出现通知失败的问题。而过于频繁的定时轮询，效率也不是最佳的（90%是无用功）。所以，我们一般会把两种方式结合起来使用。

解决了通知的问题，又有新的问题了。万一这消息有重复被消费，往用户帐号上多加了钱，那岂不是后果很严重？

仔细思考，其实我们可以消息消费方，也通过一个“消费状态表”来记录消费状态。在执行“加款”操作之前，检测下该消息（提供标识）是否已经消费过，消费完成后，通过本地事务控制来更新这个“消费状态表”。这样子就避免重复消费的问题。

总结：上诉的方式是一种非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。但是，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。所以，在真正的高并发场景下，该方案也会有瓶颈和限制的。

MQ（非事务消息）

通常情况下，在使用非事务消息支持的MQ产品时，我们很难将业务操作与对MQ的操作放在一个本地事务域中管理。通俗点描述，还是以上述提到的“跨行转账”为例，我们很难保证在扣款完成之后对MQ投递消息的操作就一定能成功。这样一致性似乎很难保证。

先从消息生产者这端来分析，请看伪代码：

根据上述代码及注释，我们来分析下可能的情况：

操作数据库成功，向MQ中投递消息也成功，皆大欢喜

操作数据库失败，不会向MQ中投递消息了

操作数据库成功，但是向MQ中投递消息时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚

从上面分析的几种情况来看，貌似问题都不大的。那么我们来分析下消费者端面临的问题：

消息出列后，消费者对应的业务操作要执行成功。如果业务执行失败，消息不能失效或者丢失。需要保证消息与业务操作一致

尽量避免消息重复消费。如果重复消费，也不能因此影响业务结果

如何保证消息与业务操作一致，不丢失？

主流的MQ产品都具有持久化消息的功能。如果消费者宕机或者消费失败，都可以执行重试机制的（有些MQ可以自定义重试次数）。

如何避免消息被重复消费造成的问题？

保证消费者调用业务的服务接口的幂等性

通过消费日志或者类似状态表来记录消费状态，便于判断（建议在业务上自行实现，而不依赖MQ产品提供该特性）

总结：这种方式比较常见，性能和吞吐量是优于使用关系型数据库消息表的方案。如果MQ自身和业务都具有高可用性，理论上是可以满足大部分的业务场景的。不过在没有充分测试的情况下，不建议在交易业务中直接使用。

MQ（事务消息）

举个例子，Bob向Smith转账，那我们到底是先发送消息，还是先执行扣款操作？

好像都可能会出问题。如果先发消息，扣款操作失败，那么Smith的账户里面会多出一笔钱。反过来，如果先执行扣款操作，后发送消息，那有可能扣款成功了但是消息没发出去，Smith收不到钱。除了上面介绍的通过异常捕获和回滚的方式外，还有没有其他的思路呢？

下面以阿里巴巴的RocketMQ中间件为例，分析下其设计和实现思路。

RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事物，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。细心的读者可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，这时候发现了Prepared消息，它会向消息发送者确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。如下图：

总结：据笔者的了解，各大知名的电商平台和互联网公司，几乎都是采用类似的设计思路来实现“最终一致性”的。这种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大。目前主流的开源MQ（ActiveMQ、RabbitMQ、Kafka）均未实现对事务消息的支持，所以需二次开发或者新造轮子。比较遗憾的是，RocketMQ事务消息部分的代码也并未开源，需要自己去实现。

其他补偿方式

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了success字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。

其实这就是一个很典型的补偿例子，跟一些MQ重试补偿机制很类似。

一般成熟的系统中，对于级别较高的服务和接口，整体的可用性通常都会很高。如果有些业务由于瞬时的网络故障或调用超时等问题，那么这种重试机制其实是非常有效的。

当然，考虑个比较极端的场景，假如系统自身有bug或者程序逻辑有问题，那么重试1W次那也是无济于事的。那岂不是就发生了“明明已经付款，却显示未付款不发货”类似的悲剧？

其实为了交易系统更可靠，我们一般会在类似交易这种高级别的服务代码中，加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。

在某些特殊的情况下，还会有“人工补偿”的，这也是最后一道屏障。

小结

上诉的几种方案中，笔者也大致总结了其设计思路，优势，劣势等，相信读者已经有了一定的理解。其实分布式系统的事务一致性本身是一个技术难题，目前没有一种很简单很完美的方案能够应对所有场景。具体还是要使用者根据不同的业务场景去抉择。

### 2 docker比linux虚拟机有什么优势？

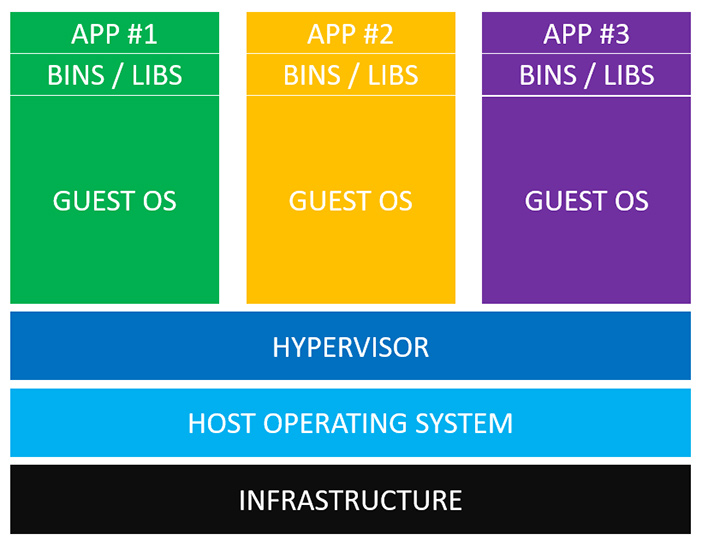
首先，大家需要明确一点，Docker容器不是虚拟机。

2014年，当我第一次接触Docker的时候，我把它比做一种轻量级的虚拟机。这样做无可厚非，因为Docker最初的成功秘诀，正是它比虚拟机更节省内存，启动更快。Docker不停地给大家宣传，”虚拟机需要数分钟启动，而Docker容器只需要50毫秒”。

然而，Docker容器并非虚拟机，我们不妨来比较一下它们。

理解虚拟机

使用虚拟机运行多个相互隔离的应用时，如下图:



从下到上理解上图:

基础设施(Infrastructure)。它可以是你的个人电脑，数据中心的服务器，或者是云主机。

主操作系统(Host Operating System)。你的个人电脑之上，运行的可能是MacOS，Windows或者某个Linux发行版。

虚拟机管理系统(Hypervisor)。利用Hypervisor，可以在主操作系统之上运行多个不同的从操作系统。类型1的Hypervisor有支持MacOS的HyperKit，支持Windows的Hyper-V以及支持Linux的KVM。类型2的Hypervisor有VirtualBox和VMWare。

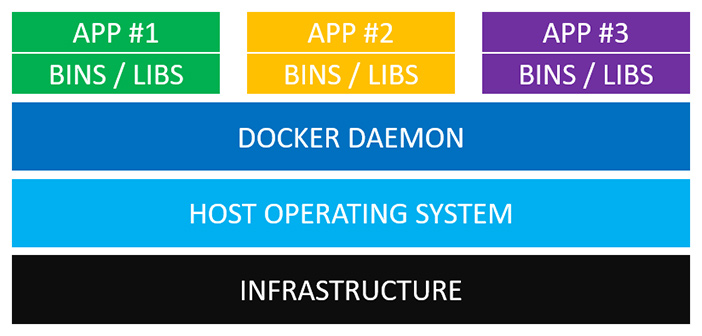
从操作系统(Guest Operating System)。假设你需要运行3个相互隔离的应用，则需要使用Hypervisor启动3个从操作系统，也就是3个虚拟机。这些虚拟机都非常大，也许有700MB，这就意味着它们将占用2.1GB的磁盘空间。更糟糕的是，它们还会消耗很多CPU和内存。

各种依赖。每一个从操作系统都需要安装许多依赖。如果你的的应用需要连接PostgreSQL的话，则需要安装libpq-dev；如果你使用Ruby的话，应该需要安装gems；如果使用其他编程语言，比如Python或者Node.js，都会需要安装对应的依赖库。

应用。安装依赖之后，就可以在各个从操作系统分别运行应用了，这样各个应用就是相互隔离的。

理解Docker容器

使用Docker容器运行多个相互隔离的应用时，如下图:



不难发现，相比于虚拟机，Docker要简洁很多。因为我们不需要运行一个臃肿的从操作系统了。

从下到上理解上图:

基础设施(Infrastructure)。

主操作系统(Host Operating System)。所有主流的Linux发行版都可以运行Docker。对于MacOS和Windows，也有一些办法”运行”Docker。

Docker守护进程(Docker Daemon)。Docker守护进程取代了Hypervisor，它是运行在操作系统之上的后台进程，负责管理Docker容器。

各种依赖。对于Docker，应用的所有依赖都打包在Docker镜像中，Docker容器是基于Docker镜像创建的。

应用。应用的源代码与它的依赖都打包在Docker镜像中，不同的应用需要不同的Docker镜像。不同的应用运行在不同的Docker容器中，它们是相互隔离的。

对比虚拟机与Docker

Docker守护进程可以直接与主操作系统进行通信，为各个Docker容器分配资源；它还可以将容器与主操作系统隔离，并将各个容器互相隔离。虚拟机启动需要数分钟，而Docker容器可以在数毫秒内启动。由于没有臃肿的从操作系统，Docker可以节省大量的磁盘空间以及其他系统资源。

说了这么多Docker的优势，大家也没有必要完全否定虚拟机技术，因为两者有不同的使用场景。虚拟机更擅长于彻底隔离整个运行环境。例如，云服务提供商通常采用虚拟机技术隔离不同的用户。而Docker通常用于隔离不同的应用，例如前端，后端以及数据库。

### 3 MyBatis的#和$有什么区别？

1.#将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user\_id#，如果传入的值是111,那么解析成sql时的值为order by "111", 如果传入的值是id，则解析成的sql为order by "id".

2.$将传入的数据直接显示生成在sql中。如：order by $user\_id$，如果传入的值是111,那么解析成sql时的值为order by user\_id, 如果传入的值是id，则解析成的sql为order by id.

3.#方式能够很大程度防止sql注入。

4.$方式无法防止Sql注入。

5.$方式一般用于传入数据库对象，例如传入表名.

6.一般能用#的就别用$.

### 4 Spring bean作用域有哪些？默认是哪个？

1. singleton作用域 scope="singleton" 默认值

bean设置为该作用域时，Spring IOC容器中只会存在一个共享的bean实例，也就意味着Spring IOC容器只会创建该bean定义的唯一实例。

这个单一实例会被存储到单例缓存（singleton cache）中。

2. prototype作用域 scope="prototype"

prototype作用域部署的bean，每一次请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean()）都会产生一个新的bean

实例，相当于一个new操作。

注意：Spring不能对一个prototype bean的整个生命周期负责，容器在初始化、配置、装饰或者是装配完一个prototype实例后，将它交给客户端，

随后就对 prototype实例不闻不问了。此时清楚任何prototype作用域的对象并释放其所持有的昂贵资源，都是客户端代码的职责。

注意：request作用域、session作用域、global-session作用域使用的时候首先要在初始化web的web.xml中做以下配置

<listener>

<listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>

</listener>

3. request作用域

表示该作用域下，针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP-request内有效

4. session作用域

session作用域表示针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP-session内有效

5. global-session作用域

全局会话内有效

### 5 spring mvc的control是单例吗？怎么保证线程安全？Struts2是单例还是多例？

Spring MVC单例，struts1单例，struts2 多例

由于Spring MVC默认是Singleton的，所以会产生一个潜在的安全隐患。根本核心是instance变量保持状态的问题。这意味着每个request过来，系统都会用原有的instance去处理，这样导致了两个结果：

一是我们不用每次创建Controller，

二是减少了对象创建和垃圾收集的时间；

由于只有一个Controller的instance，当多个线程同时调用它的时候，它里面的instance变量就不是线程安全的了，会发生窜数据的问题。

当然大多数情况下，我们根本不需要考虑线程安全的问题，比如dao,service等，除非在bean中声明了实例变量。因此，我们在使用spring mvc 的contrller时，应避免在controller中定义实例变量。

如：

publicclassControllerextendsAbstractCommandController{

......

protectedModelAndView handle(HttpServletRequest request,HttpServletResponse response,

Object command,BindException errors)throwsException{

company =................;

}

protectedCompany company;

}

在这里有声明一个变量company，这里就存在并发线程安全的问题。

如果控制器是使用单例形式，且controller中有一个私有的变量a,所有请求到同一个controller时，使用的a变量是共用的，即若是某个请求中修改了这个变量a，则，在别的请求中能够读到这个修改的内容。。

有几种解决方法：

1、在控制器中不使用实例变量

2、将控制器的作用域从单例改为原型，即在spring配置文件Controller中声明 scope="prototype"，每次都创建新的controller

3、在Controller中使用ThreadLocal变量

这几种做法有好有坏，第一种，需要开发人员拥有较高的编程水平与思想意识，在编码过程中力求避免出现这种BUG，而第二种则是容器自动的对每个请求产生一个实例，由JVM进行垃圾回收，因此做到了线程安全。

使用第一种方式的好处是实例对象只有一个，所有的请求都调用该实例对象，速度和性能上要优于第二种，不好的地方，就是需要程序员自己去控制实例变量的状态保持问题。第二种由于每次请求都创建一个实例，所以会消耗较多的内存空间。

所以在使用spring开发web 时要注意，默认Controller、Dao、Service都是单例的

### 6 Dubbo支持哪些协议,有哪几种容错机制？

Dubbo支持dubbo、rmi、hessian、http、webservice、thrift、redis等多种协议。

容错机制有：

Failover Cluster

失败自动切换，当出现失败，重试其它服务器。(缺省)

通常用于读操作，但重试会带来更长延迟。

可通过retries="2"来设置重试次数(不含第一次)。正是文章刚开始说的那种情况.

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。

通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。

通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。

通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。

通常用于实时性要求较高的读操作，但需要浪费更多服务资源。

可通过forks="2"来设置最大并行数。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0开始支持)

通常用于通知所有提供者更新缓存或日志等本地资源信息。

重试次数配置如：(failover集群模式生效)

<dubbo:serviceretries="2"/>

或：<dubbo:referenceretries="2"/>

或：<dubbo:reference>

<dubbo:methodname="findFoo"retries="2"/>

</dubbo:reference>

### 7 NIO和IO方面有何不同？有没使用过Netty？

下表总结了Java IO和NIO之间的主要区别：

|  |  |
| --- | --- |
| IO | NIO |
| 面向流 | 面向缓冲 |
| 阻塞IO | 非阻塞IO |
| 无 | 选择器 |

1、面向流与面向缓冲

     Java IO和NIO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

2、阻塞与非阻塞IO

     Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

3、选择器（Selectors）

     Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

### 8 怎么理解CAS原语无锁？

**无锁编程 / lock-free / 非阻塞同步**

无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。

实现非阻塞同步的方案称为“无锁编程算法”（ Non-blocking algorithm）。

lock-free是目前最常见的无锁编程的实现级别（一共三种级别）。

**为什么要 Non-blocking sync ？**

使用lock实现线程同步有很多缺点：

\* 产生竞争时，线程被阻塞等待，无法做到线程实时响应。

\* dead lock。

\* live lock。

\* 优先级翻转。

\* 使用不当，造成性能下降。

如果在不使用 lock 的情况下，实现变量同步，那就会避免很多问题。虽然目前来看，无锁编程并不能替代 lock。

**实现级别**

非同步阻塞的实现可以分成三个级别：wait-free/lock-free/obstruction-free。

wait-free

是最理想的模式，整个操作保证每个线程在有限步骤下完成。

保证系统级吞吐（system-wide throughput）以及无线程饥饿。

截止2011年，没有多少具体的实现。即使实现了，也需要依赖于具体CPU。

lock-free

允许个别线程饥饿，但保证系统级吞吐。

确保至少有一个线程能够继续执行。

wait-free的算法必定也是lock-free的。

obstruction-free

在任何时间点，一个线程被隔离为一个事务进行执行（其他线程suspended），并且在有限步骤内完成。在执行过程中，一旦发现数据被修改（采用时间戳、版本号），则回滚。

也叫做乐观锁，即乐观并发控制(OOC)。事务的过程是：1读取，并写时间戳；2准备写入，版本校验；3校验通过则写入，校验不通过，则回滚。

lock-free必定是obstruction-free的。

**CAS原语**

LL/SC, atom read-modify-write

如果CPU提供了Load-Link/Store-Conditional（LL/SC）这对指令，则就可以轻松实现变量的CPU级别无锁同步。

LL [addr],dst：从内存[addr]处读取值到dst。

SC value,[addr]：对于当前线程，自从上次的LL动作后内存值没有改变，就更新成新值。

上述过程就是实现lock-free的 read-modify-write 的原子操作。

CAS （Compare-And-Swap）

LL/SC这对CPU指令没有实现，那么就需要寻找其他算法，比如CAS。

CAS是一组原语指令，用来实现多线程下的变量同步。

在 x86 下的指令CMPXCHG实现了CAS，前置LOCK既可以达到原子性操作。截止2013，大部分多核处理器均支持CAS。

CAS原语有三个参数，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。

Bool CAS(T\* addr, T expected, T newValue)

{

if( \*addr == expected )

{

\*addr = newValue;

return true;

}

else

return false;

}

ABA 问题

thread1意图对val=1进行操作变成2，cas(\*val,1,2)。

thread1先读取val=1；thread1被抢占（preempted），让thread2运行。

thread2 修改val=3，又修改回1。

thread1继续执行，发现期望值与“原值”（其实被修改过了）相同，完成CAS操作。

使用CAS会造成ABA问题，特别是在使用指针操作一些并发数据结构时。

解决方案

ABAʹ：添加额外的标记用来指示是否被修改。

**语言实现**

Java demo

AtomicInteger atom = new AtomicInteger(1);

boolean r = atom.compareAndSet(1, 2);

C# demo

int i=1;

Interlocked.Increment(ref i);

### 9讲下微服务？

一、单块架构的优缺点：

优点：

易于开发： 开发方式简单，IDE 支持好，方便运行和调试。

易于测试： 所有功能运行在一个进程中，一旦进程启动，便可以进行系统测试。

易于部署： 只需要将打好的一个软件包发布到服务器即可。

易于水平伸缩： 只需要创建一个服务器节点，配置好运行时环境，再将软件包发布到新服务器节点即可运行程序（当然也需要采取分发策略保证请求能有效地分发到新节点）。

缺点：

维护成本大： 当应用程序的功能越来越多、团队越来越大时，沟通成本、管理成本显著增加。当出现 bug 时，可能引起 bug 的原因组合越来越多，导致分析、定位和修复的成本增加；并且在对全局功能缺乏深度理解的情况下，容易在修复 bug 时引入新的 bug。

持续交付周期长： 构建和部署时间会随着功能的增多而增加，任何细微的修改都会触发部署流水线。

新人培养周期长： 新成员了解背景、熟悉业务和配置环境的时间越来越长。

技术选型成本高： 单块架构倾向于采用统一的技术平台或方案来解决所有问题，如果后续想引入新的技术或框架，成本和风险都很大。

可扩展性差： 随着功能的增加，垂直扩展的成本将会越来越大；而对于水平扩展而言，因为所有代码都运行在同一个进程，没办法做到针对应用程序的部分功能做独立的扩展。

二、微服务本质：

1. 服务作为组件

微服务也可以被认为是一种组件，但是跟传统组件的区别在于它可以独立部署，因此它的一个显著的优势。另外一个优点是，它在组件与组件之间定义了清晰的、语言无关、平台无关的规范接口，耦合度低，灵活性非常高。但它的不足之处是，分布式调用严重依赖于网络的可靠性和稳定性。

2. 围绕业务组织团队

在单块架构中，企业一般会根据技能划分团队，在这种组织架构下，即便是简单的需求变更都有可能需要跨团队协作，沟通成本很高。而在微服务架构中，它提倡以业务为核心，按照业务能力来组织团队，团队中的成员具有多样性的技能。

3. 关注产品而非项目

在单块架构中，应用基本上是基于“项目模式”构建的，即项目启动时从不同技能资源池中抽取相关资源组成团队，项目结束后释放所有资源。这种情况下团队成员缺乏主人翁意识和产品成就感。

在微服务架构中，提倡采用“产品模式”构建，即更倾向于让团队负责整个服务的生命周期，以便提供更优质的服务。

4. 技术多样性

微服务架构中，提倡针对不同的业务特征选择合适的技术方案，有针对性的解决具体业务问题，而不是像单块架构中采用统一的平台或技术来解决所有问题。

5. 业务数据独立

微服务架构提供自主管理其相关的业务数据，这样可以随着业务的发展提供数据接口集成，而不是以数据库的方式同其他服务集成。另外，随着业务的发展，可以方便地选择更合的工具管理或者迁移业务数据。

6. 基础设施自动化

在微服务架构的实践过程中，对持续交付和部署流水线的要求很高，将促进企业不断寻找更高效的方式完成基础设施的自动化及 DevOps 运维能力的提升。

三、微服务的挑战：

1. 分布式系统的复杂性

微服务架构是基于分布式的系统，而构建分布式系统必然会带来额外的开销。

性能： 分布式系统是跨进程、跨网络的调用，受网络延迟和带宽的影响。

可靠性： 由于高度依赖于网络状况，任何一次的远程调用都有可能失败，随着服务的增多还会出现更多的潜在故障点。因此，如何提高系统的可靠性、降低因网络引起的故障率，是系统构建的一大挑战。

异步： 异步通信大大增加了功能实现的复杂度，并且伴随着定位难、调试难等问题。

数据一致性： 要保证分布式系统的数据强一致性，成本是非常高的，需要在 C（一致性）A（可用性）P（分区容错性） 三者之间做出权衡。

2. 运维成本

运维主要包括配置、部署、监控与告警和日志收集四大方面。微服务架构中，每个服务都需要独立地配置、部署、监控和收集日志，成本呈指数级增长。

3. 自动化部署

在微服务架构中，每个服务都独立部署，交付周期短且频率高，人工部署已经无法适应业务的快速变化。因此如何有效地构建自动化部署体系，是微服务面临的另一个挑战。

4. DevOps 与组织架构

在微服务架构的实施过程中，开发人员和运维人员的角色发生了变化，开发者将承担起整个服务的生命周期的责任，包括部署和监控；而运维则更倾向于顾问式的角色，尽早考虑服务如何部署。因此，按需调整组织架构、构建全功能的团队，也是一个不小的挑战。

5. 服务间的依赖测试

单块架构中，通常使用集成测试来验证依赖是否正常。而在微服务架构中，服务数量众多，每个服务都是独立的业务单元，服务主要通过接口进行交互，如何保证依赖的正常，是测试面临的主要挑战。

6. 服务间的依赖管理

微服务架构中，服务数量众多，如何清晰有效地展示服务间的依赖关系也是个不小的挑战。

### 10怎么解决跨域问题？

1 JSONP

　　JSONP是JSON with Padding的略称。它是一个非官方的协议，它允许在服务器端集成Script tags返回至客户端，通过javascript callback的形式实现跨域访问（这仅仅是JSONP简单的实现形式）。关于jsonp的使用方式，可以参考http://blog.csdn.net/alen1985/article/details/6365394，优缺点可以参考http://blog.csdn.net/z69183787/article/details/19191385

2 添加响应头，允许跨域

　　addHeader(‘Access-Control-Allow-Origin:\*’);//允许所有来源访问

　　addHeader(‘Access-Control-Allow-Method:POST,GET’);//允许访问的方式

3 代理的方式

服务器A的test01.html页面想访问服务器B的后台action，返回“test”字符串，此时就出现跨域请求，浏览器控制台会出现报错提示，由于跨域是浏览器的同源策略造成的，对于服务器后台不存在该问题，可以在服务器A中添加一个代理action，在该action中完成对服务器B中action数据的请求，然后在返回到test01.html页面。

### 11负载均衡一般几种算法？

负载均衡，英文 名称为Load Balance，指由多台服务器以对称的方式组成一个服务器集合，每台服务器都具有等价的地位，都可以单独对外提供服务而无须其他服务器的辅助。通过某种 负载分担技术，将外部发送来的请求均匀分配到对称结构中的某一台服务器上，而接收到请求的服务器独立地回应客户的请求。负载均衡能够平均分配客户请求到服 务器阵列，借此提供快速获取重要数据，解决大量并发访问服务问题，这种集群技术可以用最少的投资获得接近于大型主机的性能。

负载均衡分为软件负载均衡和硬件负载均衡，前者的代表是阿里章文嵩博士研发的LVS，后者则是均衡服务器比如F5，当然这只是提一下，不是重点。

本文讲述的是"将外部发送来的请求均匀分配到对称结构中的某一台服务器上"的各种算法，并以Java代码演示每种算法的具体实现，OK，下面进入正题，在进入正题前，先写一个类来模拟Ip列表：

1 public class IpMap

2 {

3 // 待路由的Ip列表，Key代表Ip，Value代表该Ip的权重

4 public static HashMap<String, Integer> serverWeightMap =

5 new HashMap<String, Integer>();

6

7 static

8 {

9 serverWeightMap.put("192.168.1.100", 1);

10 serverWeightMap.put("192.168.1.101", 1);

11 // 权重为4

12 serverWeightMap.put("192.168.1.102", 4);

13 serverWeightMap.put("192.168.1.103", 1);

14 serverWeightMap.put("192.168.1.104", 1);

15 // 权重为3

16 serverWeightMap.put("192.168.1.105", 3);

17 serverWeightMap.put("192.168.1.106", 1);

18 // 权重为2

19 serverWeightMap.put("192.168.1.107", 2);

20 serverWeightMap.put("192.168.1.108", 1);

21 serverWeightMap.put("192.168.1.109", 1);

22 serverWeightMap.put("192.168.1.110", 1);

23 }

24 }

**轮询（Round Robin）法**

轮询法即Round Robin法，其代码实现大致如下：

1 public class RoundRobin

2 {

3 private static Integer pos = 0;

4

5 public static String getServer()

6 {

7 // 重建一个Map，避免服务器的上下线导致的并发问题

8 Map<String, Integer> serverMap =

9 new HashMap<String, Integer>();

10 serverMap.putAll(IpMap.serverWeightMap);

11

12 // 取得Ip地址List

13 Set<String> keySet = serverMap.keySet();

14 ArrayList<String> keyList = new ArrayList<String>();

15 keyList.addAll(keySet);

16

17 String server = null;

18 synchronized (pos)

19 {

20 if (pos > keySet.size())

21 pos = 0;

22 server = keyList.get(pos);

23 pos ++;

24 }

25

26 return server;

27 }

28 }

由于serverWeightMap中的地址列表是动态的，随时可能有机器上线、 下线或者宕机，因此为了避免可能出现的并发问题，方法内部要新建局部变量serverMap，现将serverMap中的内容复制到线程本地，以避免被多 个线程修改。这样可能会引入新的问题，复制以后serverWeightMap的修改无法反映给serverMap，也就是说这一轮选择服务器的过程中， 新增服务器或者下线服务器，负载均衡算法将无法获知。新增无所谓，如果有服务器下线或者宕机，那么可能会访问到不存在的地址。因此，服务调用端需要有相应的容错处理，比如重新发起一次server选择并调用。

对于当前轮询的位置变量pos，为了保证服务器选择的顺序性，需要在操作时对其加锁，使得同一时刻只能有一个线程可以修改pos的值，否则当pos变量被并发修改，则无法保证服务器选择的顺序性，甚至有可能导致keyList数组越界。

轮询法的优点在于：试图做到请求转移的绝对均衡。

轮询法的缺点在于：为了做到请求转移的绝对均衡，必须付出相当大的代价，因为为了保证pos变量修改的互斥性，需要引入重量级的悲观锁synchronized，这将会导致该段轮询代码的并发吞吐量发生明显的下降。

**随机（Random）法**

通过系统随机函数，根据后端服务器列表的大小值来随机选择其中一台进行访问。由概率统计理论可以得知，随着调用量的增大，其实际效果越来越接近于平均分配流量到每一台后端服务器，也就是轮询的效果。

随机法的代码实现大致如下：

1 public class Random

2 {

3 public static String getServer()

4 {

5 // 重建一个Map，避免服务器的上下线导致的并发问题

6 Map<String, Integer> serverMap =

7 new HashMap<String, Integer>();

8 serverMap.putAll(IpMap.serverWeightMap);

9

10 // 取得Ip地址List

11 Set<String> keySet = serverMap.keySet();

12 ArrayList<String> keyList = new ArrayList<String>();

13 keyList.addAll(keySet);

14

15 java.util.Random random = new java.util.Random();

16 int randomPos = random.nextInt(keyList.size());

17

18 return keyList.get(randomPos);

19 }

20 }

整体代码思路和轮询法一致，先重建serverMap，再获取到server列表。在选取server的时候，通过Random的nextInt方法取0~keyList.size()区间的一个随机值，从而从服务器列表中随机获取到一台服务器地址进行返回。基于概率统计的理论，吞吐量越大，随机算法的效果越接近于轮询算法的效果。

**源地址哈希（Hash）法**

源地址哈希的思想是获取客户端访问的IP地址值，通过哈希函数计算得到一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是要访问的服务器的序号。源地址哈希算法的代码实现大致如下：

1 public class Hash

2 {

3 public static String getServer()

4 {

5 // 重建一个Map，避免服务器的上下线导致的并发问题

6 Map<String, Integer> serverMap =

7 new HashMap<String, Integer>();

8 serverMap.putAll(IpMap.serverWeightMap);

9

10 // 取得Ip地址List

11 Set<String> keySet = serverMap.keySet();

12 ArrayList<String> keyList = new ArrayList<String>();

13 keyList.addAll(keySet);

14

15 // 在Web应用中可通过HttpServlet的getRemoteIp方法获取

16 String remoteIp = "127.0.0.1";

17 int hashCode = remoteIp.hashCode();

18 int serverListSize = keyList.size();

19 int serverPos = hashCode % serverListSize;

20

21 return keyList.get(serverPos);

22 }

23 }

前两部分和轮询法、随机法一样就不说了，差别在于路由选择部分。通过客户端的ip也就是remoteIp，取得它的Hash值，对服务器列表的大小取模，结果便是选用的服务器在服务器列表中的索引值。

源地址哈希法的优点在于：保证了相同客户端IP地址将会被哈希到同一台后端服务器，直到后端服务器列表变更。根据此特性可以在服务消费者与服务提供者之间建立有状态的session会话。

源地址哈希算法的缺点在于：除非集群中服务器的非常稳定，基本不会上下线，否则一旦有服务器上线、下线，那么通过源地址哈希算法路由到的服务器是服务器上线、下线前路由到的服务器的概率非常低，如果是session则取不到session，如果是缓存则可能引发"雪崩"。如果这么解释不适合明白，可以看我之前的一篇文章MemCache超详细解读，一致性Hash算法部分。

**加权轮询（Weight Round Robin）法**

不同的服务器可能机器配置和当前系统的负载并不相同，因此它们的抗压能力也不尽相 同，给配置高、负载低的机器配置更高的权重，让其处理更多的请求，而低配置、高负载的机器，则给其分配较低的权重，降低其系统负载。加权轮询法可以很好地 处理这一问题，并将请求顺序按照权重分配到后端。加权轮询法的代码实现大致如下：

1 public class WeightRoundRobin

2 {

3 private static Integer pos;

4

5 public static String getServer()

6 {

7 // 重建一个Map，避免服务器的上下线导致的并发问题

8 Map<String, Integer> serverMap =

9 new HashMap<String, Integer>();

10 serverMap.putAll(IpMap.serverWeightMap);

11

12 // 取得Ip地址List

13 Set<String> keySet = serverMap.keySet();

14 Iterator<String> iterator = keySet.iterator();

15

16 List<String> serverList = new ArrayList<String>();

17 while (iterator.hasNext())

18 {

19 String server = iterator.next();

20 int weight = serverMap.get(server);

21 for (int i = 0; i < weight; i++)

22 serverList.add(server);

23 }

24

25 String server = null;

26 synchronized (pos)

27 {

28 if (pos > keySet.size())

29 pos = 0;

30 server = serverList.get(pos);

31 pos ++;

32 }

33

34 return server;

35 }

36 }

与轮询法类似，只是在获取服务器地址之前增加了一段权重计算的代码，根据权重的大小，将地址重复地增加到服务器地址列表中，权重越大，该服务器每轮所获得的请求数量越多。

**加权随机（Weight Random）法**

与加权轮询法类似，加权随机法也是根据后端服务器不同的配置和负载情况来配置不同的权重。不同的是，它是按照权重来随机选择服务器的，而不是顺序。加权随机法的代码实现如下：

1 public class WeightRandom

2 {

3 public static String getServer()

4 {

5 // 重建一个Map，避免服务器的上下线导致的并发问题

6 Map<String, Integer> serverMap =

7 new HashMap<String, Integer>();

8 serverMap.putAll(IpMap.serverWeightMap);

9

10 // 取得Ip地址List

11 Set<String> keySet = serverMap.keySet();

12 Iterator<String> iterator = keySet.iterator();

13

14 List<String> serverList = new ArrayList<String>();

15 while (iterator.hasNext())

16 {

17 String server = iterator.next();

18 int weight = serverMap.get(server);

19 for (int i = 0; i < weight; i++)

20 serverList.add(server);

21 }

22

23 java.util.Random random = new java.util.Random();

24 int randomPos = random.nextInt(serverList.size());

25

26 return serverList.get(randomPos);

27 }

28 }

这段代码相当于是随机法和加权轮询法的结合，比较好理解，就不解释了。

**最小连接数（Least Connections）法**

前面几种方法费尽心思来实现服务消费者请求次数分配的均衡，当然这么做是没错的，可以为后端的多台服务器平均分配工作量，最大程度地提高服务器的利用率，但是实际情况是否真的如此？实际情况中，请求次数的均衡真的能代表负载的均衡吗？这是一个值得思考的问题。

上面的问题，再换一个角度来说就是：以后端服务器的视角来观察系统的负载，而非请求发起方来观察。最小连接数法便属于此类。

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有 快有慢，它正是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前请求，尽可能地提高后端服务器的利用效率，将负载 合理地分流到每一台机器。由于最小连接数设计服务器连接数的汇总和感知，设计与实现较为繁琐，此处就不说它的实现了。

### 12 Zookeeper最少几个节点分别是什么？怎么实现选举？如何做到多个系统访问同一个资源的强一致性？如何保证挂机数据的备份一致性？

一、在zookeeper的选举过程中，为了保证选举过程最后能选出leader，就一定不能出现两台机器得票相同的僵局，所以一般的，要求zk集群的server数量一定要是奇数，也就是2n+1台，

并且，如果集群出现问题，其中存活的机器必须大于n+1台，否则leader无法获得多数server的支持，系统就自动挂掉。所以一般是3个或者3个以上节点。

群首（leader），追随者（follower），观察者（observer）。

Leader，作为整个ZooKeeper集群的主节点，负责响应所有对ZooKeeper状态变更的请求。它会将每个状态更新请求进行排序和编号，以便保证整个集群内部消息处理的FIFO。

这里补充一下ZooKeeper的请求类型。对于exists，getData，getChildren等只读请求，收到该请求的zk服务器将会在本地处理，因为由第一讲的ZAB理论可知，每个服务器看到的名字空间内容都是一致的，无所谓在哪台机器上读取数据，因此如果ZooKeeper集群的负载是读多写少，并且读请求分布得均衡的话，效率是很高的。对于create，setData，delete等有写操作的请求，则需要统一转发给leader处理，leader需要决定编号、执行操作，这个过程称为一个事务（transaction）。

事务的编号就不说了，ZAB一章中已经把zxid的格式说得很清楚，已经忘了的可以回头查阅，重点来说说事务的执行。ZooKeeper事务和关系型数据库事务相似之处是都具备原子性，即整个事务（编号+执行）要么一起成功要么一起失败。另外事务还具备幂等性，即对一个事务执行多次，结果永远都是一致的。但ZooKeeper事务不具备关系型数据库事务的回滚机制，原因是不需要，因为ZAB协议已经保证消息是严格FIFO的，并且只有一个leader实际处理事务。（回忆两阶段提交2PC，之所以需要2PC的原因，归根结底是有不止一个“主”，必须保证这么多“主”看到的结果都是一致的）

Follower，Follower的逻辑就比较简单了。除了响应本服务器上的读请求外，follower还要处理leader的提议，并在leader提交该提议时在本地也进行提交。Follower处理提议的过程已经在ZAB一章中描述过了。

另外需要注意的是，leader和follower构成ZooKeeper集群的法定人数，也就是说，只有他们才参与新leader的选举、响应leader的提议。

Observer，如果ZooKeeper集群的读取负载很高，或者客户端多到跨机房，可以设置一些observer服务器，以提高读取的吞吐量。Observer和Follower比较相似，只有一些小区别：首先observer不属于法定人数，即不参加选举也不响应提议；其次是observer不需要将事务持久化到磁盘，一旦observer被重启，需要从leader重新同步整个名字空间。

二、ZooKeeper的工作原理

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

每个Server在工作过程中有三种状态：

LOOKING：当前Server不知道leader是谁，正在搜寻

LEADING：当前Server即为选举出来的leader

FOLLOWING：leader已经选举出来，当前Server与之同步

三、 选主流程

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。先介绍basic paxos流程：

1 .选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

2 .选举线程首先向所有Server发起一次询问(包括自己)；

3 .选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

4. 收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

5. 线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数， 设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。

通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1.

每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。选主的具体流程图如下所示：

fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。其流程图如下所示：

四、强一致性如何保证

ZooKeeper是个集群，内部有多个server，每个server都可以连接多个client，每个client都可以修改server中的数据

ZooKeeper可以保证每个server内的数据完全一致，是如何实现的呢？

答：数据一致性是靠Paxos算法保证的，Paxos可以说是分布式一致性算法的鼻祖，是ZooKeeper的基础

Paxos的基本思路：(深入解读zookeeper一致性原理)

假设有一个社团，其中有团员、议员（决议小组成员）两个角色

团员可以向议员申请提案来修改社团制度

议员坐在一起，拿出自己收到的提案，对每个提案进行投票表决，超过半数通过即可生效

为了秩序，规定每个提案都有编号ID，按顺序自增

每个议员都有一个社团制度笔记本，上面记着所有社团制度，和最近处理的提案编号，初始为0

投票通过的规则：

新提案ID

是否大于 议员本中的ID，是议员举手赞同

如果举手人数大于议员人数的半数，即让新提案生效

例如：

刚开始，每个议员本子上的ID都为0，现在有一个议员拿出一个提案：团费降为100元，这个提案的ID自增为1

每个议员都和自己ID对比，一看

1>0，举手赞同，同时修改自己本中的ID为1

发出提案的议员一看超过半数同意，就宣布：1号提案生效

然后所有议员都修改自己笔记本中的团费为100元

以后任何一个团员咨询任何一个议员："团费是多少？"，议员可以直接打开笔记本查看，并回答：团费为100元

可能会有极端的情况，就是多个议员一起发出了提案，就是并发的情况

例如

刚开始，每个议员本子上的编号都为0，现在有两个议员（A和B）同时发出了提案，那么根据自增规则，这两个提案的编号都为1，但只会有一个被先处理

假设A的提案在B的上面，议员们先处理A提案并通过了，这时，议员们的本子上的ID已经变为了1，接下来处理B的提案，由于它的ID是1，不大于议员本子上的ID，B提案就被拒绝了，B议员需要重新发起提案

上面就是Paxos的基本思路，对照ZooKeeper，对应关系就是：

团员 -client

议员 -server

议员的笔记本 -server中的数据

提案 -变更数据的请求

提案编号 -zxid（ZooKeeper Transaction Id）

提案生效 -执行变更数据的操作

ZooKeeper中还有一个leader的概念，就是把发起提案的权利收紧了，以前是每个议员都可以发起提案，现在有了leader，大家就不要七嘴八舌了，先把提案都交给leader，由leader一个个发起提案

Paxos算法就是通过投票、全局编号机制，使同一时刻只有一个写操作被批准，同时并发的写操作要去争取选票，只有获得过半数选票的写操作才会被批准，所以永远只会有一个写操作得到批准，其他的写操作竞争失败只好再发起一轮投票

1）一致性保证

Zookeeper 是一种高性能、可扩展的服务。 Zookeeper 的读写速度非常快，并且读的速度要比写的速度更快。另外，在进行读操作的时候， ZooKeeper 依然能够为旧的数据提供服务。这些都是由于 ZooKeepe 所提供的一致性保证，它具有如下特点：

顺序一致性

客户端的更新顺序与它们被发送的顺序相一致。

原子性

更新操作要么成功要么失败，没有第三种结果。

单系统镜像

无论客户端连接到哪一个服务器，客户端将看到相同的 ZooKeeper 视图。

可靠性

一旦一个更新操作被应用，那么在客户端再次更新它之前，它的值将不会改变。。这个保证将会产生下面两种结果：

1 ．如果客户端成功地获得了正确的返回代码，那么说明更新已经成果。如果不能够获得返回代码（由于通信错误、超时等等），那么客户端将不知道更新操作是否生效。

2 ．当从故障恢复的时候，任何客户端能够看到的执行成功的更新操作将不会被回滚。

实时性

在特定的一段时间内，客户端看到的系统需要被保证是实时的（在十几秒的时间里）。在此时间段内，任何系统的改变将被客户端看到，或者被客户端侦测到。

给予这些一致性保证， ZooKeeper 更高级功能的设计与实现将会变得非常容易，例如： leader 选举、队列以及可撤销锁等机制的实现。

2）Leader选举

ZooKeeper 需要在所有的服务（可以理解为服务器）中选举出一个 Leader ，然后让这个 Leader 来负责管理集群。此时，集群中的其它服务器则成为此 Leader 的 Follower 。并且，当 Leader 故障的时候，需要 ZooKeeper能够快速地在 Follower 中选举出下一个 Leader 。这就是 ZooKeeper 的 Leader 机制，下面我们将简单介绍在ZooKeeper 中， Leader 选举（ Leader Election ）是如何实现的。

此操作实现的核心思想是：首先创建一个 EPHEMERAL 目录节点，例如“ /election ”。然后。每一个ZooKeeper 服务器在此目录下创建一个 SEQUENCE| EPHEMERAL 类型的节点，例如“ /election/n\_ ”。在SEQUENCE 标志下， ZooKeeper 将自动地为每一个 ZooKeeper 服务器分配一个比前一个分配的序号要大的序号。此时创建节点的 ZooKeeper 服务器中拥有最小序号编号的服务器将成为 Leader 。

在实际的操作中，还需要保障：当 Leader 服务器发生故障的时候，系统能够快速地选出下一个 ZooKeeper 服务器作为 Leader 。一个简单的解决方案是，让所有的 follower 监视 leader 所对应的节点。当 Leader 发生故障时， Leader 所对应的临时节点将会自动地被删除，此操作将会触发所有监视 Leader 的服务器的 watch 。这样这些服务器将会收到 Leader 故障的消息，并进而进行下一次的 Leader 选举操作。但是，这种操作将会导致“从众效应”的发生，尤其当集群中服务器众多并且带宽延迟比较大的时候，此种情况更为明显。

在 Zookeeper 中，为了避免从众效应的发生，它是这样来实现的：每一个 follower 对 follower 集群中对应的比自己节点序号小一号的节点（也就是所有序号比自己小的节点中的序号最大的节点）设置一个 watch 。只有当follower 所设置的 watch 被触发的时候，它才进行 Leader 选举操作，一般情况下它将成为集群中的下一个 Leader。很明显，此 Leader 选举操作的速度是很快的。因为，每一次 Leader 选举几乎只涉及单个 follower 的操作。

### 13 依赖注入有什么优势？

优点

传统的代码，每个对象负责管理与自己需要依赖的对象，导致如果需要切换依赖对象的实现类时，需要修改多处地方。同时，过度耦合也使得对象难以进行单元测试。

依赖注入把对象的创造交给外部去管理,很好的解决了代码紧耦合（tight couple）的问题，是一种让代码实现松耦合（loose couple）的机制。

松耦合让代码更具灵活性，能更好地应对需求变动，以及方便单元测试。

缺点

在java语言中基本采用反射的方式来实现依赖注入，在一定程度会影响性能

### 14 为什么cglib可以代理class，jdk只能代理接口?

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配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）。

AOP包括切面（aspect）、通知（advice）、连接点（joinpoint），实现方式就是通过对目标对象的代理在连接点前后加入通知，完成统一的切面操作。

### 15 Mybatis底层执行SQL语句的过程是什么？接口实现类怎么生成的？

mybatis在java开发中已经成为主流，它有很多优点，例如

1. 易于上手和掌握。

2. sql写在xml里，便于统一管理和优化。

3. 解除sql与程序代码的耦合。

4. 提供映射标签，支持对象与数据库的orm字段关系映射

5. 提供对象关系映射标签，支持对象关系组建维护

6. 提供xml标签，支持编写动态sql。

Controller调用service

service有实现类serviceImpl

serviceImpl调用DAO

但是，不用写DAO的实现，你知道这是为什么吗？

一：配置spring集成mybatis

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="configLocation" value="classpath:conf/mybatis/config.xml"/>

</bean>

要实现对数据库的操作就要有sqlSession,而sqlSession就是有sqlSessionFactory创建的。

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="com.lzz.aspp.\*\*.dao,com.lzz.lsp.\*\*.dao" />

</bean>

这段配置主要是配置映射文件的路径，这样做的好处就是不用写Dao的实现了，简单的说就是接好接口，写好映射文件 会自动映射到方法和sql语句。

二底层原理：

mybatis通过JDK的动态代理方式，在启动加载配置文件时，根据配置mapper的xml去生成Dao的实现。

session.getMapper()使用了代理，当调用一次此方法，都会产生一个代理class的instance,看看这个代理class的实现.

public class MapperProxy implements InvocationHandler {

...

public static <T> T newMapperProxy(Class<T> mapperInterface, SqlSession sqlSession) {

ClassLoader classLoader = mapperInterface.getClassLoader();

Class<?>[] interfaces = new Class[]{mapperInterface};

MapperProxy proxy = new MapperProxy(sqlSession);

return (T) Proxy.newProxyInstance(classLoader, interfaces, proxy);

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if (!OBJECT\_METHODS.contains(method.getName())) {

final Class<?> declaringInterface = findDeclaringInterface(proxy, method);

final MapperMethod mapperMethod = new MapperMethod(declaringInterface, method, sqlSession);

final Object result = mapperMethod.execute(args);

if (result == null && method.getReturnType().isPrimitive()) {

throw new BindingException("Mapper method '" + method.getName() + "' (" + method.getDeclaringClass() + ") attempted to return null from a method with a primitive return type (" + method.getReturnType() + ").");

}

return result;

}

return null;

}

这里是用到了JDK的代理Proxy。 newMapperProxy()可以取得实现interfaces 的class的代理类的实例。

当执行interfaces中的方法的时候，会自动执行invoke()方法，其中public Object invoke(Object proxy, Method method, Object[] args)中 method参数就代表你要执行的方法.

MapperMethod类会使用method方法的methodName 和declaringInterface去取 sqlMapxml 取得对应的sql，也就是拿declaringInterface的类全名加上 sqlid..

### 16 struts底层实现机制是什么？

一、strut2的处理流程

1、浏览器发出请求，即\*\*\*.action

2、请求被核心控制器的StrutsPrepareAndExecuteFilter拦截

3、搜索struts.xml配置文件，寻找Action的属性name为\*\*\*的配置信息，而后根据class属性，通过反射原理创建Action实例

actionInst

4、调用request.getParameterNames返回所有的请求参数名存入paramMap

使用循环（String paramName : paramMap.keySet()）//传入所有参数

{

//1、得到请求参数名对应的set方法

Method setter = Action类.getMethod("set"+paramName的首字母大写)

//2、获得参数名对应的参数值

String paramValue = paramMap.get(paramName)

//3、以Action实例为调用者，调用setter方法，把参数值作为参数传入

setter.invoke(actionInst , paramValue)

}

5、再次通过反射来调用method属性所指定的方法，返回一个字符串

6、根据字符串匹配struts.xml中<result>元素的属性name值，进行相对应物理视图的跳转

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*附模拟核心控制器代码\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

二、底层机制

//自定义的Action类==========

1 class MyAction{

2 　　//待传值的参数

3 private String name ;

4 private String pass;

5

6 　　//空构造函数

7 public MyAction() {

8 }

9

10 public MyAction(String name, String pass) {

11 this.name = name;

12 this.pass = pass;

13 }

14

15 public String getName() {

16 return name;

17 }

18

19 public void setName(String name) {

20 this.name = name;

21 }

22

23 public String getPass() {

24 return pass;

25 }

26

27 public void setPass(String pass) {

28 this.pass = pass;

29 }

30

31 public String regist(){

32 System.out.println("name----->"+name);

33 System.out.println("pass----->"+pass);

34 return "sucess";

35 }

36 }

//核心控制器==============

1 public class CoreFilter {

2

3

4 //模拟配置文件struts.xml

5 /\*\*

6 <action name = "abc" class = "MyAcion" method = "regist">

7 　　<result name = "success">\*\*\*\*\*\*\*\*.jsp</result>

8 　　　　　......

9 </action>

10 \*\*/

11 public static void main(String[] args) throws Exception {

12 //假如控制器收到abc.action请求,通过解析structs.xml

13 String classProp = "MyAction";

14

15 //反射原理：获取action类所对应的Class对象

16 Class actionClazz = Class.forName("test.MyAction");

17

18 //利用Class对象得到类实例

19 Object actionInst = actionClazz.newInstance(); //此处要求action含有无参构造函数的原因

20

21 //利用Map来模拟请求参数

22 // struct2的核心Filter用下面的代码。得到所有的请求<参数名--参数值>,组成Map

23 //Map<String , String > paramMap = request.getParamMap();

24 //相当于如下模拟过程：

25 Map <String,String> paramMap = new HashMap<String, String>();

26 paramMap.put("name","张三");

27 paramMap.put("pass","123");

28 for(String paramName : paramMap.keySet()){

29 //得到请求参数名对应的setter方法

30 //setName() and setPass()

31 Method setter = actionClazz.getMethod("set"

32 +paramName.substring(0,1).toUpperCase()

33 +paramName.substring(1),String.class);

34

35 //得到请求参数对应的值

36 String paramValue = paramMap.get(paramName);

37

38 //以Action实例为调用者，调用setter方法,吧参数值作为参数传入

39 setter.invoke(actionInst,paramValue);//主调者actionInst，参数值为paramValue

40 }

41 Method targetMethod = actionClazz.getMethod("regist");

42 String result = (String)targetMethod.invoke(actionInst);

43

44 if(result.equals("某个result的name属性值")){

45 //request.getRequestDispatcher().forward()

46 }

47 }

### 17怎么保证集群中只有一个应用执行定时任务，该应用宕机了如何处理？

最近项目中，需要实现分布式的定时任务，结合项目现有的中间件和技术，设计出了2种简单分布式定时任务的实现（这里的前提是定时任务是可拆分的）

需求背景：由于DB需要通过定时任务执行数据清理，每天需要清理N个分库M张分表的线下渠道的数据。（N>20,M>=10）

所以初步设计的定时任务是拆分成了N\*M个子任务，假设服务器的数量大于N\*M。关键点就是如何实现分布式，让每个服务器都利用起来执行任务，以下是设计出的2种实现方案

一、基于zookeeper和redis实现的一种分布式定时任务

由于目前存在基于zookeeper开发出的支持分布式服务器的统一配置管理中间件，还有分布式redis缓存的中间件，所以设计出了基于zookeeper和redis的一种分布式定时任务。步骤如下：

1、修改任务配置

配置一个简单的定时任务，假设每天0点触发，仅由一台服务器执行任务，具体逻辑是：

修改统一配置中的归档任务文件中针对于该任务的节点内容：

[html] view plain copy

#数据清理任务

clearArchive=【当前服务器唯一标识的信息即可】 （可以是IP+时间戳，需要保证每次任务触发的值都不一样）

2、获取子任务

由于zookeeper的功能，修改内容会被所有的服务器监听到，此时可以进行数据清理任务的执行了。但如何保证所有服务器在N\*M个子任务分配中，一台服务器仅分配一个子任务，且不会与其他服务器领取到的子任务重复呢（注意，此时属于高并发场景，所有服务器都会同时监听到）？这时就要用到分布式redis了，步骤如下：

a) 获取子任务号

遍历N\*M次子任务的子任务号（由分库号+"-"+分表号+第一步获取的clearArchive对应的值构成），把该子任务号作为key,服务器ip等信息作为value进行setnx操作，若结果为0，表示已经被其他服务器获取到了，遍历下一个。若结果为1，表示获取成功，则返回此子任务号。

b) 若子任务号不为空，则表示该服务器成功领取了一个子任务了，根据该任务号，执行对应分库分表的数据清理任务。

分析：此设计方案可以保证每一台服务器仅执行一个子任务，可以达到分布式定时任务的处理效果，但缺陷是扩展性和适用性不够，若服务器的数量过小，若子任务过大，还是会存在服务器领取的子任务不均匀的现象。但在此种场景下，应该是比较适合的，目前的技术实现就是采用这种的。

二、内联MQ

1、技术背景

由于项目中存在中间件ActiveMQ消息队列和分布式缓存锁jedis，所以使用了ActiveMQ消息队列的本地通讯的实现。

2、实现步骤如下：

a) 配置一个简单的定时任务，假设每天0点触发，仅由一台服务器执行任务，发送N\*M个对应不同分库号和分表号的MQ消息。

b) 集群的所有服务器进行消息的处理，根据消息里的分库号和分表号，进行对应子任务的处理，从而实现了分布式的定时任务处理。

综合来比较：内联MQ的方式相对实现简单，且仅需要配置对应的JMS和Listen即可，但缺点是服务器分布式处理的可能不均匀，有可能某台服务器处理了多条子任务，但往往能流传下来的都是简单的方式。

### 18 有没有开发过spring aop?什么场景？

AOP是Aspect Oriented Programing的简称，面向切面编程。AOP适合于那些具有横切逻辑的应用：如性能监测，访问控制，事务管理、缓存、对象池管理以及日志记录。AOP将这些分散在各个业务逻辑中的代码通过横向切割的方式抽取到一个独立的模块中。AOP 实现的关键就在于 AOP 框架自动创建的 AOP 代理，AOP 代理则可分为静态代理和动态代理两大类，其中静态代理是指使用 AOP 框架提供的命令进行编译，从而在编译阶段就可生成 AOP 代理类，因此也称为编译时增强；而动态代理则在运行时借助于 JDK 动态代理、CGLIB 等在内存中“临时”生成 AOP 动态代理类，因此也被称为运行时增强。

代理对象的方法 = 增强处理 + 被代理对象的方法

Spring AOP 则采用运行时生成 AOP 代理类，因此无需使用特定编译器进行处理。由于 Spring AOP 需要在每次运行时生成 AOP 代理，因此性能略差一些。

一、AOP使用场景

AOP用来封装横切关注点，具体可以在下面的场景中使用

Authentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading 懒加载

Debugging 调试

logging, tracing, profiling and monitoring 记录跟踪 优化 校准

Performance optimization 性能优化

Persistence 持久化

Resource pooling 资源池

Synchronization 同步

Transactions 事务

二、AOP相关概念

方面（Aspect）：一个关注点的模块化，这个关注点实现可能另外横切多个对象。事务管理是J2EE应用中一个很好的横切关注点例子。方面用Spring的 Advisor或拦截器实现。

连接点（Joinpoint）: 程序执行过程中明确的点，如方法的调用或特定的异常被抛出

通知（Advice）: 在特定的连接点，AOP框架执行的动作。各种类型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多AOP框架包括Spring都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。Spring中定义了四个advice: BeforeAdvice, AfterAdvice, ThrowAdvice和DynamicIntroductionAdvice

切入点（Pointcut）: 指定一个通知将被引发的一系列连接点的集合。AOP框架必须允许开发者指定切入点：例如，使用正则表达式。 Spring定义了Pointcut接口，用来组合MethodMatcher和ClassFilter，可以通过名字很清楚的理解， MethodMatcher是用来检查目标类的方法是否可以被应用此通知，而ClassFilter是用来检查Pointcut是否应该应用到目标类上

引入（Introduction）: 添加方法或字段到被通知的类。 Spring允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified接口，来简化缓存。Spring中要使用Introduction, 可有通过DelegatingIntroductionInterceptor来实现通知，通过DefaultIntroductionAdvisor来配置Advice和代理类要实现的接口

目标对象（Target Object）: 包含连接点的对象。也被称作被通知或被代理对象。POJO

AOP代理（AOP Proxy）: AOP框架创建的对象，包含通知。 在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。

织入（Weaving）: 组装方面来创建一个被通知对象。这可以在编译时完成（例如使用AspectJ编译器），也可以在运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。

三、日志应用：

实现登陆和日志管理（使用Spring AOP

1）LoginService LogService TestMain

2）用Spring 管理 LoginService 和 LogService 的对象

3）确定哪些连接点是切入点，在配置文件中

4）将LogService封装为通知

5）将通知植入到切入点

6）客户端调用目标

<aop:config>

<aop:pointcut expression="execution(\* cn.com.spring.service.impl.\*.\*(..))" id="myPointcut"/>

<!--将哪个-->

<aop:aspect id="dd" ref="logService">

<aop:before method="log" pointcut-ref="myPointcut"/>

</aop:aspect>

</aop:config>

execution(\* \* cn.com.spring.service.impl.\*.\*(..))

1)\* 所有的修饰符

2)\* 所有的返回类型

3)\* 所有的类名

4)\* 所有的方法名

5)\* ..所有的参数名

1.ILoginService.java

package cn.com.spring.service;

public interface ILoginService {

public boolean login(String userName, String password);

}

2.LoginServiceImpl.java

package cn.com.spring.service.impl;

import cn.com.spring.service.ILoginService;

public class LoginServiceImpl implements ILoginService {

public boolean login(String userName, String password) {

System.out.println("login:" + userName + "," + password);

return true;

}

}

3.ILogService.java

package cn.com.spring.service;

import org.aspectj.lang.JoinPoint;

public interface ILogService {

//无参的日志方法

public void log();

//有参的日志方法

public void logArg(JoinPoint point);

//有参有返回值的方法

public void logArgAndReturn(JoinPoint point,Object returnObj);

}

4.LogServiceImpl.java

package cn.com.spring.service.impl;

import org.aspectj.lang.JoinPoint;

import cn.com.spring.service.ILogService;

public class LogServiceImpl implements ILogService {

@Override

public void log() {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*Log\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

//有参无返回值的方法

public void logArg(JoinPoint point) {

//此方法返回的是一个数组，数组中包括request以及ActionCofig等类对象

Object[] args = point.getArgs();

System.out.println("目标参数列表：");

if (args != null) {

for (Object obj : args) {

System.out.println(obj + ",");

}

System.out.println();

}

}

//有参并有返回值的方法

public void logArgAndReturn(JoinPoint point, Object returnObj) {

//此方法返回的是一个数组，数组中包括request以及ActionCofig等类对象

Object[] args = point.getArgs();

System.out.println("目标参数列表：");

if (args != null) {

for (Object obj : args) {

System.out.println(obj + ",");

}

System.out.println();

System.out.println("执行结果是：" + returnObj);

}

}

}

5.applicationContext.java

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:aop="http://www.springframework.org/schema/aop" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<bean id="logService" class="cn.com.spring.service.impl.LogServiceImpl"></bean>

<bean id="loginService" class="cn.com.spring.service.impl.LoginServiceImpl"></bean>

<aop:config>

<!-- 切入点 -->

<aop:pointcut

expression="execution(\* cn.com.spring.service.impl.LoginServiceImpl.\*(..))"

id="myPointcut" />

<!-- 切面： 将哪个对象中的哪个方法，织入到哪个切入点 -->

<aop:aspect id="dd" ref="logService">

<!-- 前置通知

<aop:before method="log" pointcut-ref="myPointcut" />

<aop:after method="logArg" pointcut-ref="myPointcut">

-->

<aop:after-returning method="logArgAndReturn" returning="returnObj" pointcut-ref="myPointcut"/>

</aop:aspect>

</aop:config>

</beans>

6.TestMain.java

public class TestMain {

public static void testSpringAOP(){

ApplicationContext ctx = new ClassPathXmlApplicationContext("app\*.xml");

ILoginService loginService = (ILoginService)ctx.getBean("loginService");

loginService.login("zhangsan", "12344");

}

public static void main(String[] args) {

testSpringAOP();

}

}

7.输出结果：

login:zhangsan,12344

目标参数列表：

zhangsan,

12344,

执行结果是：true

解析:1.先调用了login()方法System.out.println("login:" + userName + "," + password);

### 19 说说Spring boot ，有哪些好处？

优点：

1.去除了大量的xml配置文件

2.简化复杂的依赖管理

3.配合各种starter使用，基本上可以做到自动化配置

4.快速启动容器

5. 配合Maven或Gradle等构件工具打成Jar包后，Java -jar 进行部署运行还是蛮简单的，创建独立Spring应用程序，嵌入式Tomcat，Jetty容器，无需部署WAR包，简化Maven及Gradle配置，尽可能的自动化配置Spring，直接植入产品环境下的实用功能，比如度量指标、健康检查及扩展配置等，无需代码生成及XML配置。

缺点：

1.从原来的xml配置方式转换到JAVA配置方式变化有点大，不太适应

2.感觉Spring Boot 比较适合做微服务，不适合做比较大型的项目。

### 20 Spring transcation控制底层用什么技术实现的？

我们先来分析一下Spring事务管理机制的实现原理。由于Spring内置AOP默认使用动态代理模式实现，我们就先来分析一下动态代理模式的实现方 法。动态代理模式的核心就在于代码中不出现与具体应用层相关联的接口或者类引用，如上所说，这个代理类适用于任何接口的实现。下面我们来看一个例子。

public class TxHandler implements InvocationHandler {

private Object originalObject;

public Object bind(Object obj) {

　this.originalObject = obj;

　return Proxy.newProxyInstance(obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),this);

}

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

　Object result = null;

　if (!method.getName().startsWith("save")) {

　　UserTransaction tx = null;

　　try {

　　　tx = (UserTransaction) (new InitialContext().lookup("java/tx"));

　　　result = method.invoke(originalObject, args);

　　　tx.commit();

　　} catch (Exception ex) {

　　　if (null != tx) {

　　　　try {

　　　　　tx.rollback();

　　　　} catch (Exception e) {

　　　}

　　}

　}

} else {

　result = method.invoke(originalObject, args);

}

return result;

}

}

下面我们来分析一下上述代码的关键所在。

首先来看一下这段代码：

return Proxy.newProxyInstance(

　obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),this);

　 　java.lang.reflect.Proxy.newProxyInstance方法根据传入的接口类型 （obj.getClass.getInterfaces()）动态构造一个代理类实例返回，这也说明了为什么动态代理实现要求其所代理的对象一定要实现 一个接口。这个代理类实例在内存中是动态构造的，它实现了传入的接口列表中所包含的所有接口。

　　再来分析以下代码：

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

　……

　result = method.invoke(originalObject, args);

　……

　return result;

}

　 　InvocationHandler.invoke方法将在被代理类的方法被调用之前触发。通过这个方法，我们可以在被代理类方法调用的前后进行一些处 理，如代码中所示，InvocationHandler.invoke方法的参数中传递了当前被调用的方法（Method）,以及被调用方法的参数。同 时，可以通过method.invoke方法调用被代理类的原始方法实现。这样就可以在被代理类的方法调用前后写入任何想要进行的操作。

　 　Spring的事务管理机制实现的原理，就是通过这样一个动态代理对所有需要事务管理的Bean进行加载，并根据配置在invoke方法中对当前调用的 方法名进行判定，并在method.invoke方法前后为其加上合适的事务管理代码，这样就实现了Spring式的事务管理。Spring中的AOP实 现更为复杂和灵活，不过基本原理是一致的。

### 21对activiti流程引擎的理解，里面运用了什么设计模式？怎么部署流程？多少个表？哪个版本？里面的任务监听器？Execution和流程实例区别？怎么开发一个简单的命令模式？

一、Activiti表结构

5.18版本有23张表，支持多种关系数据库。

所有的表都以ACT\_开头。 第二部分是表示表的用途的两个字母标识。

ACT\_RE\_\*: 'RE'表示repository。 这个前缀的表包含了流程定义和流程静态资源 （图片，规则，等等）。

ACT\_RU\_\*: 'RU'表示runtime。 这些运行时的表，包含流程实例，任务，变量，异步任务，等运行中的数据。 Activiti只在流程实例执行过程中保存这些数据， 在流程结束时就会删除这些记录。 流程结束时会把runtime中的数据移入到history中，包括流程实例，任务，变量等。这样运行时的表可以一直很小保持在一个范围内，保证速度很快。

ACT\_ID\_\*: 'ID'表示identity。 这些表包含身份信息，比如用户，组等等。

ACT\_HI\_\*: 'HI'表示history。 这些表包含历史数据，比如历史流程实例， 变量，任务等等。

ACT\_GE\_\*: 通用数据， 如存放资源文件

二、核心API

ProcessEngine 是Activiti中最核心的类，其他的类都是由他而来。

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

RepositoryService repositoryService =processEngine.getRepositoryService();

RuntimeService runtimeService = processEngine.getRuntimeService();

TaskService taskService =processEngine.getTaskService();

各个Service的作用：

RepositoryService Activiti的仓库服务类。所谓的仓库指流程定义文档的两个文件：bpmn文件和流程图片

该service可以用来删除部署的流程定义。

RuntimeService 执行管理，包括启动，推进，删除流程实例等操作

TaskService 是activiti的任务服务类。可以从这个类中获取任务的相关信息，如当前正在执行的个人待办和用户组待办任务。

HistoryService 是activiti的查询历史信息的类，在一个流程执行完成后，这个对象为我们提供查询历史信息，可以跟踪流程实例对应所有待办节点的运行情况。

IdentityService 认证服务，在工作流执行过程中进行用户查询、认证等操作

FormService Activiti表单引擎产生的用户任务表单服务

三、网关分类

1 互斥网关

只会返回一条结果。当流程执行到排他网关时，流程引擎会自动检索网关出口，从上到下检索如果发现第一条决策结果为true或者没有设置条件的(默认为成立)，则流出。如果没有任何一个出口符合条件，则抛出异常。

${stuType==1}

${stuType==2}

2 并行网关

无条件触发，不会解析条件，写了条件也会被忽略。进入和外出的数目不一定相等。分支(fork)： 并行后的所有外出顺序流，为每个顺序流都创建一个并发分支。汇聚(join)： 所有到达并行网关，在此等待的进入分支， 直到所有进入顺序流的分支都到达以后， 流程就会通过汇聚网关。

流入下一个节点可以设置条件，可以是全部子任务都完成还是50%完成。

3 包容性网关

集中了前两个网关的特点，可以定义条件，多条件执行，只要条件返回true就会执行，包含网关只会等待被选中执行了的进入顺序流。

4 事件网关

基于事件网关允许根据事件判断流向。网关的每个外出顺序流都要连接到一个中间捕获事件。 当流程到达一个基于事件网关，网关会进入等待状态：会暂停执行。 与此同时，会为每个外出顺序流创建相对的事件订阅。

四、任务分类

用户任务：指定一个人或者一个变量，设置的人

直接指定办理人：${assignee}

任务监听器设置处理人：

TaskListenerImpl implements TaskListener

delegateTask.setAssignee("XXX");

脚本任务：

脚本任务是一个自动化活动。当一个流程执行到达脚本任务时，执行相应的脚本。

https://www.cnblogs.com/dengjiahai/p/6942433.html

服务任务：

public class ServiceTask implements JavaDelegate{

@Override

public void execute(DelegateExecution execution) throws Exception

{ 业务逻辑... } }

消息任务：当流程到达这个节点处于停滞状态，程序收到一个消息之后signal 流程实例。

runtimeService.signal(execution.getId());

五、监听器

1 全局监听器

主要使用的场景就是监控这个流程的启动和结束。流程开始的时候可以监控，流程结束的时候可以监控，这里说的是流程实例启动结束的监控

包括流程的启动和停止，线条的take监听（）

ExecutionListener定义如下：流程实例start、end、take的时候调用。take是监控连线的时候使用的。

public interface ExecutionListener extends Serializable {

  String EVENTNAME\_START = "start";

  String EVENTNAME\_END = "end";

  String EVENTNAME\_TAKE = "take";

 void notify(DelegateExecution execution) throws Exception;

}

package cm.daling.ch1.listener;

import org.activiti.engine.delegate.DelegateExecution;

import org.activiti.engine.delegate.ExecutionListener;

public class MyExecutionListener implements ExecutionListener {

public void notify(DelegateExecution execution) throws Exception {

String eventName = execution.getEventName();

//start

if ("start".equals(eventName)) {

System.out.println("start=========");

}else if ("end".equals(eventName)) {

System.out.println("end=========");

}

}

}

2 任务节点监听器

节点监听器的定义接口org.activiti.engine.delegate.TaskListener

public class MyExecutionListener implements TaskListener {

public void notify(DelegateTask delegateTask) {

String eventName = delegateTask.getEventName();

}

}

3 AOP式的监听器

（1） 一般思路ACT\_RE\_actdef\_ext表配置每个节点的处理人（变量，角色，用户组，团队），邮件通知人，回调服务等信息。

在配置文件中配置ParseHandler，启动服务器会自动为所有流程的 process(start,end); usertask(start,complete,end); sequence(take) 添加监听器,根据 配置的节点对应的信息做相应的业务逻辑处理。

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">

<property name="dataSource" ref="dataSource" />

<property name="transactionManager" ref="transactionManager" />

<property name="databaseSchemaUpdate" value="true" />

<property name="jpaHandleTransaction" value="true" />

<property name="jpaCloseEntityManager" value="true" />

<property name="jobExecutorActivate" value="false" />

<property name="idGenerator" ref="uuidGenerator"/>

<!-- <property name="deploymentResources" value="classpath\*:diagrams/\*.\*" /> -->

<property name="customDefaultBpmnParseHandlers">

<list>

<bean class="com.parse.ActivitiProcessExtParseHandler" />

<bean class="com.parse.ActivitiReceiveTaskExtParseHandler" />

<bean class="com.parse.ActivitiSequenceFlowExtParseHandler" />

<bean class="com.parse.ActivitiServiceTaskExtParseHandler" />

<bean class="com.parse.ActivitiUserTaskExtParseHandler" />

</list>

</property>

</bean>

（2）扩展Activiti流程定义文件,实现自定义节点属性

customDefaultBpmnParseHandlers 注册各种监听器

监听器包括启动流程日志，任务完成日志，流程开始，流程结束，消息任务开始，消息任务结束，服务任务开始，服务任务结束，用户任务开始，用户任务完成等监听器。

六、命令模式和责任链模式

Activiti任务的执行 用到了设计模式中的命令模式和责任链模式。

命令模式是将行为请求者和行为实现者解耦合的方式。对命令进行封装，将命令和执行命令分隔开。请求的一方发出命令，要求执行某些操作，接受一方收到命令，执行这些操作的真正实现。请求的一方不必知道接受方的接口，以及如何被操作。

以启动流程为例：

ProcessEngineConfigurationImpl 会初始化一系列配置和注入信息，包括 initCommandExecutors();

RuntimeServiceImpl 继承ServiceImpl ，

在初始化的时候已经注入commandExecutor命令执行器。

public class RuntimeServiceImpl extends ServiceImpl implements RuntimeService

public ProcessInstance startProcessInstanceByKey(String processDefinitionKey, String businessKey) {

return commandExecutor.execute(new StartProcessInstanceCmd<ProcessInstance>(processDefinitionKey, null, businessKey, null));

}

protected void initService(Object service)

{

if (service instanceof ServiceImpl) ((ServiceImpl) service).setCommandExecutor(this.commandExecutor);

}

启动流程类StartProcessInstanceCmd实现Command接口，重写execute方法。注入CommandContext对象，commandContext可以得到流程实例，流程定义，历史数据，任务数据的API，做相应的流程逻辑处理。

以启动流程拦截器链为，一般是默认的拦截器，可以自定义before/after拦截器。

logger拦截器-->spring事务拦截器-->CommandContext拦截器-->CommandInvoker拦截器

其中CommandContext拦截器的工作主要是设置Context：

[java] view plain copy

Context.setCommandContext(context);

Context.setProcessEngineConfiguration(processEngineConfiguration);

return next.execute(config, command);

Activiti默认的拦截器：

    1. LogInterceptor日志拦截器，拦截器打印执行的日志。

    2.事务拦截器。

    3.CommandContextInterceptor 命令上下文拦截器设置Context.

     4 CommandInvoker拦截器,执行命令

### 22 消息队列用过什么？吞吐量多少？什么场景？

a.异步处理，提高吞吐量，减少开销

b.应用解耦，防止接口端应用崩溃，数据阻塞丢失

c.流量销锋,如秒杀业务中将所有请求放在消息队列中，并设置队列的长度超过则转到错误页面

d.日记处理

c.消息通讯，通过消息订阅，时间通讯收发功能

RabbitMQ

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了Broker构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

Redis

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

ZeroMQ

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果down机，数据将会丢失。其中，Twitter的Storm中默认使用ZeroMQ作为数据流的传输。

ActiveMQ

ActiveMQ是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

Kafka/Jafka

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式Publish/Subscribe消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现复杂均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制来统一了在线和离线的消息处理，这一点也是本课题所研究系统所看重的。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

### 23 对Lucene，Solr和ELK有了解吗？分片是什么技术？Es中的索引/类型/文档什么关系？

Cluster包含多个node，Indices不应该理解成动词索引，Indices可理解成关系数据库中的databases，Indices可包含多个Index，Index对应关系数据库中的database，它是用来存储相关文档的。

Elasticsearch与关系数据的类比对应关系如下：

Relational DB ⇒ Databases ⇒ Tables ⇒ Rows

ColumnsElasticsearch ⇒ Indices ⇒ Types ⇒ Documents ⇒ Fields

这里的document的可以理解为一个JSON序列对象。

每个document可包含多个field。

再来说说Shard，每个Index（对应Database）包含多个Shard，默认是5个，分散在不同的Node上，但不会存在两个相同的Shard存在一个Node上，这样就没有备份的意义了。Shard是一个最小的Lucene索引单元。当来一个document的时候，Elasticsearch通过对docid进行hash来确定其放在哪个shard上面，然后在shard上面进行索引存储。replicas就是备份，Elasticsearch采用的是Push Replication模式，当你往 master主分片上面索引一个文档，该分片会复制该文档(document)到剩下的所有 replica副本分片中，这些分片也会索引这个文档。我个人觉得这种模式很nice，有些时候对于一个document的进行索引可能生成很大的索引文件，会很占带宽，而只传输原始文件会好很多。当进行查询是，如果提供了查询的DocID，Elasticsearch通过hash就知道Doc存在哪个shard上面，再通过routing table查询就知道再哪个node上面，让后去node上面去取就好了。如果不提供DocID,那么Elasticsearch会在该Index（indics）shards所在的所有node上执行搜索预警，然后返回搜索结果，由coordinating node gather之后返回给用户。

配置上需要把握什么样的原则，应该去看一个ES优化方面的资料，推荐两本书ElasticSearch Server (豆瓣)，Mastering ElasticSearch (豆瓣)一个机器不应定只部署一个node，一台机器可以虚拟化出多个虚拟机(至于虚拟化出多少看性能)，每个虚拟机都可以部署一个node.尤其在云计算环境下，如果租用云虚拟主机，谁知道它们是不是在同一台机器上。。。

### 24 用过哪些nosql数据库？

虽然SQL数据库是非常有用的工具，但经历了15年的一支独秀之后垄断即将被打破。这只是时间问题：被迫使用关系数据库，但最终发现不能适应需求的情况不胜枚举。

但是NoSQL数据库之间的不同，远超过两 SQL数据库之间的差别。这意味着软件架构师更应该在项目开始时就选择好一个适合的 NoSQL数据库。针对这种情况，这里对 Cassandra、Mongodb、CouchDB、Redis、 Riak、Membase、Neo4j 和 HBase 进行了比较：

编注1：NoSQL：是一项全新的数据库革命性运动，NoSQL的拥护者们提倡运用非关系型的数据存储。现今的计算机体系结构在数据存储方面要求具 备庞大的水平扩 展性，而NoSQL致力于改变这一现状。目前Google的 BigTable 和Amazon 的Dynamo使用的就是NoSQL型数据库。 参见NoSQL词条。）

1. CouchDB

所用语言： Erlang

特点：DB一致性，易于使用

使用许可： Apache

协议： HTTP/REST

双向数据复制，

持续进行或临时处理，

处理时带冲突检查，

因此，采用的是master-master复制（见编注2）

MVCC – 写操作不阻塞读操作

可保存文件之前的版本

Crash-only（可靠的）设计

需要不时地进行数据压缩

视图：嵌入式 映射/减少

格式化视图：列表显示

支持进行服务器端文档验证

支持认证

根据变化实时更新

支持附件处理

因此， CouchApps（独立的 js应用程序）

需要 jQuery程序库

最佳应用场景：适用于数据变化较少，执行预定义查询，进行数据统计的应用程序。适用于需要提供数据版本支持的应用程序。

例如： CRM、CMS系统。 master-master复制对于多站点部署是非常有用的。

（编注2：master-master复制：是一种数据库同步方法，允许数据在一组计算机之间共享数据，并且可以通过小组中任意成员在组内进行数据更新。）

2. Redis

所用语言：C/C++

特点：运行异常快

使用许可： BSD

协议：类 Telnet

有硬盘存储支持的内存数据库，

但自2.0版本以后可以将数据交换到硬盘（注意， 2.4以后版本不支持该特性！）

Master-slave复制（见编注3）

虽然采用简单数据或以键值索引的哈希表，但也支持复杂操作，例如 ZREVRANGEBYSCORE。

INCR & co （适合计算极限值或统计数据）

支持 sets（同时也支持 union/diff/inter）

支持列表（同时也支持队列；阻塞式 pop操作）

支持哈希表（带有多个域的对象）

支持排序 sets（高得分表，适用于范围查询）

Redis支持事务

支持将数据设置成过期数据（类似快速缓冲区设计）

Pub/Sub允许用户实现消息机制

最佳应用场景：适用于数据变化快且数据库大小可遇见（适合内存容量）的应用程序。

例如：股票价格、数据分析、实时数据搜集、实时通讯。

（编注3：Master-slave复制：如果同一时刻只有一台服务器处理所有的复制请求，这被称为 Master-slave复制，通常应用在需要提供高可用性的服务器集群。）

3. MongoDB

所用语言：C++

特点：保留了SQL一些友好的特性（查询，索引）。

使用许可： AGPL（发起者： Apache）

协议： Custom, binary（ BSON）

Master/slave复制（支持自动错误恢复，使用 sets 复制）

内建分片机制

支持 javascript表达式查询

可在服务器端执行任意的 javascript函数

update-in-place支持比CouchDB更好

在数据存储时采用内存到文件映射

对性能的关注超过对功能的要求

建议最好打开日志功能（参数 –journal）

在32位操作系统上，数据库大小限制在约2.5Gb

空数据库大约占 192Mb

采用 GridFS存储大数据或元数据（不是真正的文件系统）

最佳应用场景：适用于需要动态查询支持；需要使用索引而不是 map/reduce功能；需要对大数据库有性能要求；需要使用 CouchDB但因为数据改变太频繁而占满内存的应用程序。

例如：你本打算采用 MySQL或 PostgreSQL，但因为它们本身自带的预定义栏让你望而却步。

4. Riak

所用语言：Erlang和C，以及一些Javascript

特点：具备容错能力

使用许可： Apache

协议： HTTP/REST或者 custom binary

可调节的分发及复制(N, R, W)

用 JavaScript or Erlang在操作前或操作后进行验证和安全支持。

使用JavaScript或Erlang进行 Map/reduce

连接及连接遍历：可作为图形数据库使用

索引：输入元数据进行搜索（1.0版本即将支持）

大数据对象支持（ Luwak）

提供“开源”和“企业”两个版本

全文本搜索，索引，通过 Riak搜索服务器查询（ beta版）

支持Masterless多站点复制及商业许可的 SNMP监控

最佳应用场景：适用于想使用类似 Cassandra（类似Dynamo）数据库但无法处理 bloat及复杂性的情况。适用于你打算做多站点复制，但又需要对单个站点的扩展性，可用性及出错处理有要求的情况。

例如：销售数据搜集，工厂控制系统；对宕机时间有严格要求；可以作为易于更新的 web服务器使用。

5. Membase

所用语言： Erlang和C

特点：兼容 Memcache，但同时兼具持久化和支持集群

使用许可： Apache 2.0

协议：分布式缓存及扩展

非常快速（200k+/秒），通过键值索引数据

可持久化存储到硬盘

所有节点都是唯一的（ master-master复制）

在内存中同样支持类似分布式缓存的缓存单元

写数据时通过去除重复数据来减少 IO

提供非常好的集群管理 web界面

更新软件时软无需停止数据库服务

支持连接池和多路复用的连接代理

最佳应用场景：适用于需要低延迟数据访问，高并发支持以及高可用性的应用程序

例如：低延迟数据访问比如以广告为目标的应用，高并发的 web 应用比如网络游戏（例如 Zynga）

6. Neo4j

所用语言： Java

特点：基于关系的图形数据库

使用许可： GPL，其中一些特性使用 AGPL/商业许可

协议： HTTP/REST（或嵌入在 Java中）

可独立使用或嵌入到 Java应用程序

图形的节点和边都可以带有元数据

很好的自带web管理功能

使用多种算法支持路径搜索

使用键值和关系进行索引

为读操作进行优化

支持事务（用 Java api）

使用 Gremlin图形遍历语言

支持 Groovy脚本

支持在线备份，高级监控及高可靠性支持使用 AGPL/商业许可

最佳应用场景：适用于图形一类数据。这是 Neo4j与其他nosql数据库的最显著区别

例如：社会关系，公共交通网络，地图及网络拓谱

7. Cassandra

所用语言： Java

特点：对大型表格和 Dynamo支持得最好

使用许可： Apache

协议： Custom, binary (节约型)

可调节的分发及复制(N, R, W)

支持以某个范围的键值通过列查询

类似大表格的功能：列，某个特性的列集合

写操作比读操作更快

基于 Apache分布式平台尽可能地 Map/reduce

我承认对 Cassandra有偏见，一部分是因为它本身的臃肿和复杂性，也因为 Java的问题（配置，出现异常，等等）

最佳应用场景：当使用写操作多过读操作（记录日志）如果每个系统组建都必须用 Java编写（没有人因为选用 Apache的软件被解雇）

例如：银行业，金融业（虽然对于金融交易不是必须的，但这些产业对数据库的要求会比它们更大）写比读更快，所以一个自然的特性就是实时数据分析

8. HBase

（配合 ghshephard使用）

所用语言： Java

特点：支持数十亿行X上百万列

使用许可： Apache

协议：HTTP/REST （支持 Thrift，见编注4）

在 BigTable之后建模

采用分布式架构 Map/reduce

对实时查询进行优化

高性能 Thrift网关

通过在server端扫描及过滤实现对查询操作预判

支持 XML, Protobuf, 和binary的HTTP

Cascading, hive, and pig source and sink modules

基于 Jruby（ JIRB）的shell

对配置改变和较小的升级都会重新回滚

不会出现单点故障

堪比MySQL的随机访问性能

### 25 对hadoop,spark,elk,hive,flume,pig了解多少？

一、FLUME 介绍

Apache Flume 用于移动大规模批量流数据到 HDFS 系统。从Web服务器收集当前日志文件数据到HDFS聚集用于分析，一个常见的用例是Flume。

Flume 支持多种来源，如：

“tail”(从本地文件，该文件的管道数据和通过Flume写入 HDFS，类似于Unix命令“tail”)

系统日志

Apache log4j (允许Java应用程序通过Flume事件写入到HDFS文件)。

二、Pig简介

在Map Reduce框架，需要的程序将其转化为一系列 Map 和 Reduce阶段。 但是，这不是一种编程模型，它被数据分析所熟悉。因此，为了弥补这一差距，一个抽象概念叫 Pig 建立在 Hadoop 之上。

Pig是一种高级编程语言，分析大数据集非常有用。 Pig 是雅虎努力开发的结果

Pig 使人们能够更专注于分析大量数据集和花更少的时间来写map-reduce程序。

类似猪吃东西，Pig 编程语言的目的是可以在任何类型的数据工作。

Pig 由两部分组成：

Pig Latin，这是一种语言

运行环境，用于运行PigLatin程序

Pig Latin 程序由一系列操作或变换应用到输入数据，以产生输出。这些操作描述被翻译成可执行到数据流，由 Pig 环境执行。下面，这些转换的结果是一系列的 MapReduce 作业，程序员是不知道的。所以，在某种程度上，Pig 允许程序员关注数据，而不是执行过程。

Pig Latin 是一种相对硬挺的语言，它采用熟悉的关键字来处理数据，例如，Join, Group 和 Filter。

三、HIVE 介绍

在某种程度上数据集收集的大小并在行业用于商业智能分析正在增长，它使传统的数据仓库解决方案更加昂贵。HADOOP与MapReduce框架，被用于大型数据集分析的替代解决方案。虽然，Hadoop 地庞大的数据集上工作证明是非常有用的，MapReduce框架是非常低级别并且它需要程序员编写自定义程序，这导致难以维护和重用。 Hive 就是为程序员设计的。

Hive 演变为基于Hadoop的Map-Reduce 框架之上的数据仓库解决方案。

Hive 提供了类似于SQL的声明性语言，叫作：HiveQL, 用于表达的查询。使用 Hive-SQL，用户能够非常容易地进行数据分析。

Hive 引擎编译这些查询到 map-reduce作业中并在 Hadoop 上执行。此外，自定义 map-reduce 脚本，也可以插入查询。Hive运行存储在表中，它由基本数据类型，如数组和映射集合的数据类型的数据。

四、Hadoop和Spark

Spark是一个计算框架

Hadoop是包含计算框架MapReducehe分布式文件系统HDFS。

Spark是MapReduce的替代方案，而且兼容HDFS、Hive等分布式存储系统，可融入Hadoop生态。

Spark与Hadoop MapReduce优势如下

@中间结果输出

MapReduce的计算引擎将中间结果存储在磁盘上，进行存储和容错。

Spark将执行模型抽象为有向无环图执行计划（DAG），这可以将多个Stage的任务串联或者并行执行，而无须将Stage中间结果输出到HDFS中。

@数据格式和内存布局

MapReduce Schema on Read处理方式会引起较大的处理开销。Spark抽象出分布式内存存储结构弹性分布式数据集RDD，进行数据的存储。RDD能支持粗粒度写操作，但是对于读取操作，RDD可以精确到每条记录，这使得RDD可以用来作分布式索引。Spark的特性是能控制数据节点上的分区，用户可以自定义分区策略，如Hash分区等。Shark 和Spark SQL在Spark的基础之上实现了列存储和列存储压缩。

@执行策略

MapReduce在数据Shuffle之前花费了大量的时间来排序，Spark则可以减轻上述问题带来的开销。因为Spark任务在Shuffle中不是所有情景都需要排序，所以支持基于Hash的分布式聚合，调度中采用更为通用的任务执行计划图（DAG），每一轮次的输出结果在内存缓存。

@任务调度开销

传统的MapReduce系统，如Hadoop，是为了运行长达数小时的批量作业而设计的，在某些极端情况下，提交一个任务的延迟非常高。

Spark采用了事件驱动的类库AKKA来启动任务，通过线程池复用线程来避免进程或线程启动和切换开销。

1、 Spark VSHadoop有哪些异同点？

Hadoop:分布式批处理计算，强调批处理，常用于数据挖掘和数据分析。

Spark:是一个基于内存计算的开源的集群计算系统，目的是让数据分析更加快速, Spark 是一种与 Hadoop 相似的开源集群计算环境，但是两者之间还存在一些不同之处，这些有用的不同之处使 Spark 在某些工作负载方面表现得更加优越，换句话说，Spark 启用了内存分布数据集，除了能够提供交互式查询外，它还可以优化迭代工作负载。

Spark 是在 Scala 语言中实现的，它将 Scala 用作其应用程序框架。与 Hadoop 不同，Spark 和 Scala 能够紧密集成，其中的 Scala 可以像操作本地集合对象一样轻松地操作分布式数据集。

尽管创建 Spark 是为了支持分布式数据集上的迭代作业，但是实际上它是对 Hadoop 的补充，可以在 Hadoop 文件系统中并行运行。通过名为Mesos的第三方集群框架可以支持此行为。

虽然 Spark 与 Hadoop 有相似之处，但它提供了具有有用差异的一个新的集群计算框架。首先，Spark 是为集群计算中的特定类型的工作负载而设计，即那些在并行操作之间重用工作数据集（比如机器学习算法）的工作负载。为了优化这些类型的工作负载，Spark 引进了内存集群计算的概念，可在内存集群计算中将数据集缓存在内存中，以缩短访问延迟。

在大数据处理方面相信大家对hadoop已经耳熟能详，基于GoogleMap/Reduce来实现的Hadoop为开发者提供了map、reduce原语，使并行批处理程序变得非常地简单和优美。Spark提供的数据集操作类型有很多种，不像Hadoop只提供了Map和Reduce两种操作。比如map,filter, flatMap,sample, groupByKey, reduceByKey, union,join, cogroup,mapValues, sort,partionBy等多种操作类型，他们把这些操作称为Transformations。同时还提供Count,collect, reduce, lookup, save等多种actions。这些多种多样的数据集操作类型，给上层应用者提供了方便。各个处理节点之间的通信模型不再像Hadoop那样就是唯一的Data Shuffle一种模式。用户可以命名，物化，控制中间结果的分区等。可以说编程模型比Hadoop更灵活。

2、Spark对于数据处理能力和效率有哪些特色？

Spark提供了高的性能和大数据处理能力，使得用户可以快速得到反馈体验更好。另一类应用是做数据挖掘，因为Spark充分利用内存进行缓存，利用DAG消除不必要的步骤，所以比较合适做迭代式的运算。而有相当一部分机器学习算法是通过多次迭代收敛的算法，所以适合用Spark来实现。

Spark配有一个流数据处理模型，与Twitter的 Storm框架相比，Spark采用了一种有趣而且独特的办法。Storm基本上是像是放入独立事务的管道，在其中事务会得到分布式的处理。相反，Spark采用一个模型收集事务，然后在短时间内（我们假设是5秒）以批处理的方式处理事件。所收集的数据成为他们自己的RDD，然后使用Spark应用程序中常用的一组进行处理。作者声称这种模式是在缓慢节点和故障情况下会更加稳健，而且5秒的时间间隔通常对于大多数应用已经足够快了。这种方法也很好地统一了流式处理与非流式处理部分

五、ELK

一般我们需要进行日志分析场景：直接在日志文件中 grep、awk 就可以获得自己想要的信息。但在规模较大的场景中，此方法效率低下，面临问题包括日志量太大如何归档、文本搜索太慢怎么办、如何多维度查询。需要集中化的日志管理，所有服务器上的日志收集汇总。常见解决思路是建立集中式日志收集系统，将所有节点上的日志统一收集，管理，访问。

一般大型系统是一个分布式部署的架构，不同的服务模块部署在不同的服务器上，问题出现时，大部分情况需要根据问题暴露的关键信息，定位到具体的服务器和服务模块，构建一套集中式日志系统，可以提高定位问题的效率。

一个完整的集中式日志系统，需要包含以下几个主要特点：

收集－能够采集多种来源的日志数据

传输－能够稳定的把日志数据传输到中央系统

存储－如何存储日志数据

分析－可以支持 UI 分析

警告－能够提供错误报告，监控机制

ELK提供了一整套解决方案，并且都是开源软件，之间互相配合使用，完美衔接，高效的满足了很多场合的应用。目前主流的一种日志系统。

ELK简介：

ELK是三个开源软件的缩写，分别表示：Elasticsearch , Logstash, Kibana , 它们都是开源软件。新增了一个FileBeat，它是一个轻量级的日志收集处理工具(Agent)，Filebeat占用资源少，适合于在各个服务器上搜集日志后传输给Logstash，官方也推荐此工具。

Elasticsearch是个开源分布式搜索引擎，提供搜集、分析、存储数据三大功能。它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，restful风格接口，多数据源，自动搜索负载等。

Logstash 主要是用来日志的搜集、分析、过滤日志的工具，支持大量的数据获取方式。一般工作方式为c/s架构，client端安装在需要收集日志的主机上，server端负责将收到的各节点日志进行过滤、修改等操作在一并发往elasticsearch上去。

Kibana 也是一个开源和免费的工具，Kibana可以为 Logstash 和 ElasticSearch 提供的日志分析友好的 Web 界面，可以帮助汇总、分析和搜索重要数据日志。

Filebeat隶属于Beats。目前Beats包含四种工具：

Packetbeat（搜集网络流量数据）

Topbeat（搜集系统、进程和文件系统级别的 CPU 和内存使用情况等数据）

Filebeat（搜集文件数据）

Winlogbeat（搜集 Windows 事件日志数据）

### 26 云计算包括哪三种架构？



1、软件即服务SaaS

软件即服务（SaaS）为商用软件提供基于网络的访问。您有可能已经使用过SaaS，即使您当时并不知道。SaaS的示例太多了，例如Netflix、Photoshop.com、Acrobat.com、Intuit QuickBooks Online、Gmail、Google Docs、Office Web Apps、Zoho、WebQQ、新浪微盘等等。可能不太明显的SaaS实现包括移动应用程序市场中的相当一部分。

SaaS为企业提供一种降低软件使用成本的方法 — 按需使用软件而不是为每台计算机购买许可证。尤其是考虑到大多数计算机在差不多70%的时间是空闲的，SaaS可能非常有效。企业不必为单一用户购买多个许可证，而是让许可证的使用时间尽可能接近100%，从而尽可能节省成本。

1.1 SaaS的优势

SaaS 给软件厂商提供了新的机会。尤其是，SaaS软件厂商可以通过四个因素提高 ROI（投资回报）：提高部署的速度、增加用户接受率、减少支持的需要、降低实现和升级的成本。

（1）提高部署的速度

在过去，部署传统的桌面应用程序需要很大的工作量。实际上，我曾经多次听到桌面应用程序开发人员把更新他们的应用程序称为 “部署噩梦”。正如Tariq Ahmed在Flex 4 in Action (Manning Press) 的第1章中指出的，“要想让数千甚至数万客户机同时运行软件的某一版本，后勤方面的复杂性是非常高的。”

Ahmed说，复杂性这么高，以致于大多数桌面软件开发公司甚至认为这根本不合理或不可行。过去受到这个问题困扰的开发商应该考虑部署软件的SaaS版本。但是，妨碍传统软件开发公司进入SaaS市场的最大障碍是让桌面应用程序能够作为SaaS应用程序运行。在许多情况下，这需要在某种程度上重新编写软件，一些公司觉得这么做成本太高。

这正是向云计算转移的过程比较缓慢且平缓的主要原因之一。在大多数情况下，符合逻辑的解决方案是分阶段地把软件转移到云中，首先以SaaS的形式提供原应用程序的高度简化的版本。考虑到开发商对版本控制的控制水平，这么做是很合理的。在这里，分析一下SaaS的特点会很有帮助。

您可以看出在云计算与过去的 “LAN 计算” 之间有许多相似之处。典型的LAN架构由站内的许多工作站组成，它们常常被称为哑终端，它们通过连接强大的大型机（常常由IBM 提供）运行应用程序。这种计算类型过去非常适合企业，因 IT部门能够完全控制版本，可以非常方便地多次部署更新。同样，过去妨碍桌面软件应用程序开发商进行版本控制的后勤障碍在云中也不存在，因为软件在开发公司能够直接访问的基础设施上运行。

考虑到SaaS必须能够服务的客户机数量，SaaS基础设施的规模要比LAN大得多。但是底层的概念是相同的。大型机能够驻留足够多的软件实例，从而为本地网络中连接它的所有客户机提供服务；而云由许多不同的计算机资源组成，它们共同提供计算能力，从而运行为世界各地的客户机提供服务所需的许多软件实例。

（2）增加接受率

如果您走出企业，看看SaaS对于一般消费者的意义，就会发现以前一些软件的许可证费用太高，而现在SaaS让一般消费者能够以合理的价格使用它们。一个好例子是 Adobe 以SaaS的形式提供Adobe Photoshop。尽管这项工作是Adobe正在做的试验，但是已经取得了一些效果。例如，我注意到在需要执行简单的照片编辑任务时，在我的朋友和家庭成员中越来越多的人开始使用Photoshop.com进行基本的照片编辑，而不是启动全功能的版本。出现这种趋势的原因是，不需要完整版本中的功能的人现在可以省钱。与此同时，过去不使用Photoshop的人也开始使用Photoshop.com了，这给Adobe带来了争取新的长期客户的机会，扩大了潜在客户的范围。

SaaS提供的多种业务模型尤其有吸引力。例如，Intuit 以SaaS的形式提供QuickBooks Online，按月收取服务费。作为经常旅行的企业主，我发现这种服务非常有用，尤其是因为我的业务伙伴住在400英里外的另一个州里。同时，Adobe在Photoshop.com和Acrobat.com中应用了SaaS，以freemium服务的形式提供软件 — freemium服务是指一种基于许可证软件产品的SaaS缩略版的业务模型。

freemium SaaS基于的收入模型是，预计免费用户中的一部分最终会觉得软件很有用，他们会升级到启用了更多特性的SaaS付费版本，或者购买包含所有特性和功能的桌面版本的许可证。这种方法往往比通过 “受限制的演示” 模式试用软件更好，因为演示模式要求用户在桌面计算机上安装他们可能不会购买的应用程序。另外，如果免费用户中升级的比例低于预期，还可以通过广告进一步补充这个模型。随着云计算的发展，传统的桌面软件厂商经常使用这种方法适应市场的变化。

（3）减少支持的需要

大型客户服务中心的成本很高，不得不支持多种平台会导致支持问题增加，而SaaS可以大大缓解这些难题。首先，部署的简便性让开发人员能够在发现bug之后很快进行修复，这意味着大多数bug可以在大量用户遇到它们之前被修复，这会减少客户支持部门接到的电话数量，提高客户满意度，降低客户流失的可能性。

另外，传统桌面软件应用程序的开发商常常必须支持多种平台。例如，开发商可能必须支持Windows 7和Apple Mac OS X 10.6操作系统，添加对第二种操作系统的支持差不多会让开发成本加倍；而且，如果支持这些操作系统的许多不同版本，问题会更多。支持操作系统的多个版本还会产生限制。

例如，如果您要构建一个在Windows 7上运行的程序，但是它必须与Windows XP兼容，就必须非常小心，要确保特性和功能在这两个版本上都能够运行；否则，就必须把项目分为两个分支，为每个版本开发单独的代码，这会不可避免地降低生产力和效率，延长完成项目的预期时间。让业务执行官心跳加速的最快方法之一是，告诉他后两年的预期开发进度要减慢一半儿。另外，支持不同的操作系统和这些操作系统的不同版本会增加预算；这个问题和其他因素导致目前软件开发项目的失败率非常高。

（4）降低实现和升级的成本

SaaS推动ROI的第四个因素与第一个因素有点儿相似。但是，部署的速度是指快速、简便地部署应用程序更新所带来的好处。与之相反，降低实现和升级的成本是指开发公司由于能够控制版本和运行软件的基础设施所获得的经济利益。

因为开发商可以控制运行软件的平台（平台通常对于用户完全透明），所以他们不必负担在多个平台上测试和部署bug补丁和新特性的额外开销，这会节省大量资金。这让SaaS应用程序的升级成本更低。节省的大量时间和资金让开发商有机会更好地响应客户的请求并增强易用性，从而提高客户满意度，降低客户流失的可能性，这会带来间接的经济利益。

1.2 SaaS的用户体验设计

SaaS 应用程序代表着一种新一代应用程序设计方式。尽管在我目前看到的文档中没有明确地指出，但是看起来SaaS程序也带来了一种新的UI设计方式，这种方式与大多数其他行业中的产品设计流程更一致。这种方式包含一个称为用户体验设计 (UXD) 的流程，在这个流程中由产品团队而不是开发团队设计GUI。

UXD的主要目的是，确定哪些特性会让应用程序对于目标客户最有价值，并在设计中融入这些知识。尽管对于是否应该在所有类型的软件的开发中都执行这个流程有争议，但是在SaaS应用程序开发中这种做法非常普遍。出现这种现象的原因可能是，SaaS可以实现的业务模型与传统软件不同，需要执行UXD；而且通过开发SaaS可以节省大量时间和资金，让开发商有能力执行UXD。

1.3 SaaS使用的技术

由于SaaS层离普通用户非常接近，所以在SaaS层所使用到的技术，大多耳熟能详，下面是其中最主要的五种：

（1）HTML ：标准的Web页面技术，现在主要以HTML4为主，但是即将推出的HTML5会在很多方面推动Web页面的发展，比如视频和本地存储等方面。

（2）JavaScript ：一种用于Web页面的动态语言，通过JavaScript ，能够极大地丰富Web页面的功能，最流行的JS框架有jQuery和Prototype 。

（3）CSS ：主要用于控制Web页面的外观，而且能使页面的内容与其表现形式之间进行优雅地分离。

（4）Flash ：业界最常用的RIA（Rich Internet Applications）技术，能够在现阶段提供HTML等技术所无法提供的基于Web的富应用，而且在用户体验方面，非常不错。

（5）Silverlight ：来自业界巨擎微软的RIA技术，虽然其现在市场占有率稍逊于Flash，但由于其可以使用C#来进行编程，所以对开发者非常友好。

在SaaS层的技术选型上，首先，由于通用性和较低的学习成本，大多数云计算产品都会比较倾向HTML 、JavaScript和CSS这对黄金组合，但是在HTML5被大家广泛接受之前，RIA技术在用户体验方面，还是具有一定的优势，所以Flash和Silverlight也将会有一定的用武之地，比如VMware vCloud就采用了基于Flash的Flex技术，而微软的云计算产品肯定会在今后大量使用Silverlight技术。

2、平台即服务PaaS

平台即服务（Platform as a Service，PaaS）提供对操作系统和相关服务的访问。它让用户能够使用提供商支持的编程语言和工具把应用程序部署到云中。用户不必管理或控制底层基础架构，而是控制部署的应用程序并在一定程度上控制应用程序驻留环境的配置。PaaS的提供者包括Google App Engine、Windows Azure、Force.com、Heroku等。小企业软件工作室是非常适合使用PaaS的企业。通过使用云平台，可以创建世界级的产品，而不需要负担内部生产的开销。

通过PaaS这种模式，用户可以在一个提供SDK（Software Development Kit，即软件开发工具包）、文档、测试环境和部署环境等在内的开发平台上非常方便地编写和部署应用，而且不论是在部署，还是在运行的时候，用户都无需为服务器、 操作系统、网络和存储等资源的运维而操心，这些繁琐的工作都由PaaS云供应商负责。而且PaaS在整合率上面非常惊人，比如一台运行Google App Engine的服务器能够支撑成千上万的应用，也就是说， PaaS是非常经济的。 PaaS主要面对的用户是开发人员。

2.1 PaaS的优势

在软件开发过程中，一些东西常常会出问题。以我的经验，设置服务器环境以驻留开发团队要构建的Web应用程序可能会带来许多争吵。即使在最大的企业中，通常一位网络管理员要负责为几个开发团队服务。在不使用PaaS的情况下，设置开发或测试环境通常需要完成以下任务：

\* 获取并部署服务器。

\* 安装操作系统、运行时环境、源代码控制存储库和必需的所有其他中间件。

\* 配置操作系统、运行时环境、存储库和其他中间件。

\* 转移或复制现有的代码。

\* 测试并运行代码以确保一切正常。

在很多情况下，管理员已经非常忙了，所以让他们抽出时间部署新环境会很困难。对于客户机和服务器端的web应用程序开发人员来说，另一个主要问题是在本地复制运行时环境以便执行测试。

现在，想像一下您是使用PaaS的开发团队的成员。在这种情况下，您会有一个虚拟机 (VM)，其中包含完整的服务器环境，可以把它放在USB闪存驱动器中带在身边。这无疑是非常方便的。一般来说，和现有的基于本地的开发和部署环境相比，PaaS平台主要有下面这六方面有非常大的优势：

（1）友好的开发环境：通过提供SDK和IDE（Integrated Development Environment ，集成开发环境）等工具来让用户不仅能在本地方便地进行应用的开发和测试，而且能进行远程部署。

（2）丰富的服务：PaaS平台会以API的形式将各种各样的服务提供给上层的应用。

（3）精细的管理和监控：PaaS能够提供应用层的管理和监控，比如能够观察应用运行的情况和具体数值（比如吞吐量Throughput和响应时间Response Time等）来更好地衡量应用的运行状态，还有能够通过精确计量应用使用所消耗的资源来更好地计费。

（4）伸缩性强： PaaS 平台会自动调整资源来帮助运行于其上的应用更好地应对突发流量。

（5） 多租户（Multi-Tenant）机制：许多PaaS平台都自带多租户机制，不仅能更经济地支撑庞大的用户规模，而且能提供一定的可定制性以满足用户的特殊需求。

（6）整合率和经济性：PaaS平台整合率是非常高，比如PaaS的代表Google App Engine能在一台服务器上承载成千上万的应用。

2.2 PaaS的主要成分

了解 PaaS 的最好方法可能是把它分解为主要组件：平台和服务。现在，考虑提供的服务，这称为解决方案堆。也就是说，PaaS 的两个主要成分是计算平台和解决方案堆。

为了说明这两个 “成分”，我们进一步研究一下它们的定义。按照最简单的形式，计算平台是指一个可以一致地启动软件的地方（只要代码满足平台的标准）。平台的常见示例包括 Windows、Apple Mac OS X和Linux操作系统；用于移动计算的Google Android、Windows Mobile和Apple iOS；以及作为软件框架的Adobe AIR和Microsoft .NET Framework。要记住的重点是，计算平台不是指软件本身，而是指构建并运行软件的平台。图4提供一张示意图以帮助理解这种关系。

图2-2. 云计算分类与PaaS元素之间关系的图形化解释

既然理解了计算平台的概念，现在就来看看什么是解决方案堆。解决方案堆由应用程序组成，这些应用程序有助于开发过程和应用程序部署。这些应用程序是指操作系统、运行时环境、源代码控制存储库和必需的所有其他中间件。

2.3 PaaS供应商的选择

解决方案堆也反映不同PaaS公司的差异，在决定采用PaaS之前，需要深入考察各个提供商提供的解决方案堆。在与某家PaaS提供商签约之前，您应该问几个基本问题：

\* 它支持哪些框架和语言？理想情况下，PaaS应该支持基于此平台选用的语言的任何框架。

\* 可以创建多少个应用程序？大多数PaaS提供商会根据您签订的计划或服务包限制可以构建的应用程序数量。要确保提供商提供的计划或服务包能够满足您的需要。

\* 允许哪些内容类型？支持PaaS的基础设施通常涉及多租用者计算的概念，也就是说许多 “租用者” 分享单一服务器上的 “空间”，这些空间由系统管理程序管理的VM实例分隔。PaaS提供商可能会对要驻留的应用程序和内容的类型加以限制。

\* 支持哪些数据库类型？如果您的数据要随应用程序转移，这个问题就是非常重要的。必须确保提供商提供的数据库与您想要用来导入数据的格式兼容。

\* 它是否支持SSL (HTTPS)？这个问题对于确保安全性非常重要。如果您打算通过应用程序处理事务，但是发现不支持SSL，您就遇到大麻烦了。

在比较PaaS提供商时应该考虑的特性：应用程序开发框架、容易使用、业务流程建模（BPM）工具、可用性、可伸缩性、安全性、包容性、可移植性、移植工具、API等。

2.4 PaaS使用的技术

PaaS 层的技术比较多样性，下面是常见的五种：

（1）REST ：通过 REST（Representational State Transfer，表述性状态转移）技术，能够非常方便和优雅地将中间件层所支撑的部分服务提供给调用者。

（2）多租户：就是能让一个单独的应用实例可以为多个组织服务，而且能保持良好的隔离性和安全性，并且通过这种技术，能有效地降低应用的购置和维护成本。

（3）并行处理：为了处理海量的数据，需要利用庞大的x86集群进行规模巨大的并行处理，Google的MapReduce是这方面的代表之作。

（4）应用服务器：在原有的应用服务器的基础上为云计算做了一定程度的优化，比如用于Google App Engine的Jetty应用服务器。

（5）分布式缓存：通过分布式缓存技术，不仅能有效地降低对后台服务器的压力，而且还能加快相应的反应速度，最著名的分布式缓存例子莫过于Memcached 。

对于很多PaaS平台，比如用于部署Ruby应用的Heroku云平台，应用服务器和分布式缓存都是必备的，同时REST技术也常用于对外的接口，多租户技术则主要用于SaaS应用的后台，比如用于支撑Salesforce 的CRM等应用的Force.com多租户内核，而并行处理技术常被作为单独的服务推出，比如Amazon的Elastic MapReduce 。

2.5 主要的PaaS产品

（1）Force.com：Force.com是业界第一个PaaS平台，其主要通过提供完善的开发环境和强健的基础设施等来帮助企业和第三方供应商交付健壮的、可靠的和可伸缩的在线应用。还有， Force.com本身是基于Salesforce著名的多租户的架构。

（2）Google App Engine：Google App Engine提供Google的基础设施来让大家部署应用，它还提供一整套开发工具和SDK来加速应用的开发，并提供大量的免费额度来节省用户的开支。

（3）Windows Azure Platform ：它是微软推出的PaaS产品，并运行在微软数据中心的服务器和网络基础设施上的，通过公共互联网来对外提供服务，它由具有高扩展性云操作系统、数据存储网络和相关服务组成，而且服务都是通过物理或虚拟的Windows Server 2008实例提供。还有，其附带的Windows Azure SDK（软件开发包）提供了一整套开发、部署和管理Windows Azure云服务所需要的工具和API 。

（4）Heroku：是一个用于部署Ruby On Rails应用的PaaS平台，并且其底层基于Amazon EC2 的IaaS服务，而且在Ruby程序员中有非常好的口碑。

3、基础架构即服务IaaS

基础架构，或称基础设施（Infrastructure）是云的基础。它由服务器、网络设备、存储磁盘等物理资产组成。在使用IaaS时，用户并不实际控制底层基础架构，而是控制操作系统、存储和部署应用程序，还在有限的程度上控制网络组件的选择。

通过IaaS这种模式，用户可以从供应商那里获得他所需要的计算或者存储等资源来装载相关的应用，并只需为其所租用的那部分资源进行付费，而同时这些基础设施繁琐的管理工作则交给IaaS供应商来负责。

IaaS的关键概念：

\* 云爆发（cloud bursting）

\* 多租户计算（multi-tenant computing）

\* 资源共用（resources pooling）

\* 虚拟机监控程序（hypervisor）

IaaS最与众不同的两个方面：可伸缩性和虚拟化（elasticity and virtualization）。

对于企业而言，IaaS的巨大价值通过云爆发（cloudbursting）概念实现。云爆发是指当业务瞬间增长，需要大量的计算资源时，将任务负载扩展到云环境的过程。云爆发促成的资本节约潜力巨大，因为企业无需额外投资利用率很低的服务器，那些服务器一年中只有两三次使用 70% 的容量，其余时间仅有 7-10% 的负荷。

3.1 从”基础架构即资产“到”基础架构即服务”

在过去50年之内，大量成功的公司花费大量宝贵时间和资源来构建基础架构，其目标是通过创建一个更大、更快、更强的网络来获取战胜其竞争对手的竞争优势。IT 行业中的 “基础架构即资产” 范式拥有上世纪六七十年代的 “暴力跑车（muscle cars）” 所拥有的相同或类似的低效率和不利特征。对于企业计算，这些低效率包括：

\* 大量未使用的计算能力和容量，它们耗费的成本与大型、昂贵的数据中心中的硬件消耗的大量空间相关联。比如服务器一年中可能只有两三次使用70%的容量，其余时间仅有7-10%的负荷。

\* 昂贵的人力资源需求，包括要求基础架构资产（服务器、路由器、交换机等）所在的数据中心的网络管理员进行24小时监控。

\* 旨在应对高水平能源浪费的Green Computing计划的一个巨大障碍。

IaaS标志着从 “基础架构即资产” 到 “基础架构即服务” 的转变。云计算的其他两个类别也标志着范式转变。对于Platform as a Service (PaaS)，转变来自 “平台即资产” 范式，该范式的特征是大量采购许可。同样的转变也适用Software as a Service (SaaS)，这种转变是从 “软件以许可形式作为组织资产” 到 “软件以服务形式提供”。

3.2 IaaS的主要特征

（1）可伸缩性

可伸缩性是IaaS的首要关键特征。为了阐述可伸缩的概念，我需要您展开想象。假设云由一些粘在一起的棉花糖簇组成，这样人们就可以坐在它们上面。每个棉花糖都能承载一定数量的人，具体取决于组成云的棉花糖簇的数量和那些簇中包含的棉花糖的数量。随着越来越多的人登上棉花糖云，您可以通过粘贴更多的棉花糖来扩展棉花糖簇，增加表面面积。您可能已经明白，人代表需要计算资源的应用程序，比如承载网站并运行软件的资源。棉花糖簇代表VM集群，每个棉花糖代表一 VM。

尽管这听起来有点像Seuss博士的书中可能出现的内容，但它提供了一种方法来理解许多黑魔法（dark art）考虑的一个概念：可伸缩的集群（elastic clustering）。集群化几个物理服务器来形成一个虚拟云称为云集群（cloud clustering），如果它真是一种黑魔法，则精通程度可以通过一位艺术家的系统设计的可伸缩性来衡量。

我们来看一个例子。假设您是一位为美国政府工作的统计研究员。政府有点人手不足，您刚刚接受一个任务，需要编辑最近的美国人口统计的所有数据。您负责制定必要的统计数据，以便议会能够制定关于经济恢复资金分配和从现在起三天内的税收金额的重要决策。毋庸讳言，这是一项非常重要的工作，您的时间有点紧张。而且，您必须处理的数据量简直是个天文数字，您刚刚发现，编辑那些统计数据需要的计算资源需要IT部门三周时间才能准备好！

这种问题正是您可以使用IaaS轻松缓解的。事实上，使用IaaS，您可以在一小时之内完成全美人口普查数据分析。您首先创建一个服务器的单个实例，这个服务器包含在数据上运行查询需要的数据库软件。这个实例称为一个映像。当您部署映像并将数据导入数据库之后，就可以根据需要复制那个映像任意多次，并开始运行您的数据处理任务。当任务运行时，您可以手动或自动添加和移除资源。例如，如果计算任务的运行速度不够快，只需将更多机器实例副本添加到集群。

（2）虚拟化

IaaS很容易定位，因为它通常是独立于平台的。IaaS有一个硬件和软件资源组合组成。IaaS软件是低级代码，独立于操作系统运行，例如虚拟机监控程序。虚拟机监控程序负责管理硬件资源的库存并根据需要分配上述资源（见图 2-3）。这个过程称为资源共用（resource pooling）。虚拟机监控程序实现的资源共用使得虚拟化成为可能，虚拟化使多租户计算（multi-tenant computing）成为可能。多租户计算概念指由几个组织共享的一个基础架构，这些组织在安全需求和遵从性问题方面有类似的兴趣。

图2-3. VMs、虚拟机监控程序和计算机之间的关系

通过 IaaS，您拥有提供处理、存储、网络和其他计算资源的能力，您可以在那里部署和运行任意软件，比如操作系统和应用程序。大多数云计算用例遵循您已经习惯的基础分层结构：一个软件解决方案堆栈或平台被部署在一个网络基础架构上，一些应用程序在那个平台之上运行。但是，虚拟化使得云范式独一无二。

3.3 IaaS的优势

IaaS服务和传统的企业数据中心相比，在很多方面都存在一定的优势，下面是最明显的五个。

（1）免维护：主要的维护工作都有IaaS云供应商负责，所以不必用户操心。

（2）非常经济：首先免去了用户前期的硬件购置成本，而且由于IaaS云大都采用虚拟化技术，所以在应用和服务器的整合率普遍在10以上，这样能有效降低使用成本。

（3）开放标准：虽然很多IaaS平台都存在一定的私有功能，但是由于OVF等应用发布协议的诞生，使得IaaS在跨平台方面稳步前进，从而使得应用能在多个IaaS云上灵活的迁移，而不会被固定在某个企业数据中心内。

（4）支持的应用：因为IaaS主要是提供虚拟机，而且普通的虚拟机能支持多种操作系统，所以IaaS所支持应用的范围是非常广泛的。

（5）伸缩性强：IaaS云只需几分钟就能提供用户一个新的计算资源，而传统的企业数据中心则往往需要几周时间，并且计算资源可以根据用户需求来调整其资源的大小。

3.4 IaaS使用的技术

在IaaS所采用的技术方面，都是一些比较底层的技术，其中有四种技术是比较常用的：

（1）虚拟化：也可以理解它为基础设施层的“多租户”，因为通过虚拟化技术，能够在一个物理服务器上生成多个虚拟机，并且能在这些虚拟机之间能实现全面的隔离， 这样不仅能减低服务器的购置成本，而且还能同时降低服务器的运维成本，成熟的x86虚拟化技术有VMware的ESX和开源的Xen 。

（2）分布式存储：为了承载海量的数据，同时也要保证这些数据的可管理性，所以需要一整套分布式的存储系统，在这方面， Google 的GFS是典范之作。

（3）关系型数据库：基本是在原有的关系型数据库的基础上做了扩展和管理等方面的优化，使其在云中更适应。

（4）NoSQL：为了满足一些关系数据库所无法满足的目标，比如支撑海量的数据等，一些公司特地设计一批不是基于关系模型的数据库，比如Google的BigTable和Facebook的Cassandra等。

现在大多数的IaaS服务都是基于Xen的，比如Amazon的EC2等，但VMware也推出了基于ESX技术的vCloud ，同时业界也有几个基于关系型数据库的云服务，比如Amazon 的RDS（Relational Database Service，关系型数据库服务）和Windows Azure SDS（SQL Data Services， SQL数据库服务）等。关于分布式存储和NoSQL，它们已经被广泛用于云平台的后端，比如Google App Engine的Datastore就是基于BigTable和GFS这两个技术之上的，而Amazon则推出基于NoSQL技术的Simple DB 。

3.5 主要的IaaS产品

最具代表性的IaaS产品有Amazon EC2、IBM Blue Cloud、Cisco UCS和Joyent。

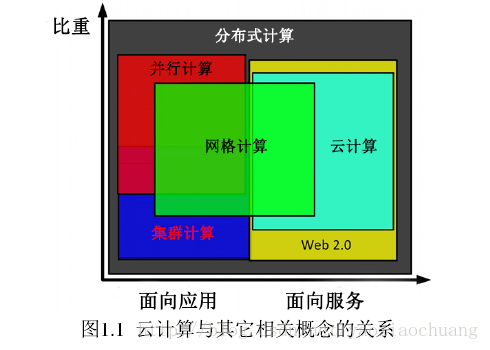
（1）Amazon EC2。EC2主要以提供不同规格的计算资源（也就是虚拟机）为主。它基于著名的开源虚拟化技术Xen。通过Amazon的各种优化和创新， EC2不论在性能上还是在稳定性上都已经满足企业级的需求。而且它还提供完善的API和Web管理界面来方便用户使用。这种IaaS产品得到业界广泛地认可和接受，其中就包括部分大型企业，比如著名的纽约时报。

（2）IBM Blue Cloud。“蓝云”解决方案是由IBM云计算中心开发的业界第一个，同时也是在技术上比较领先的企业级云计算解决方案。该解决方案可以对企业现有的基础架构进行整合，通过虚拟化技术和自动化管理技术来构建企业自己的云计算中心，并实现对企业硬件资源和软件资源的统一管理、统一分配、统一部署、统一监控和统一备份，也打破了应用对资源的独占，从而帮助企业能享受到云计算所带来的诸多优越性。

（3）Cisco UCS。它是下一代数据中心平台，在一个紧密结合的系统中整合了计算、网络、存储与虚拟化功能。该系统包含一个低延时、无丢包和支持万兆以太网的统一网络阵列以及多台企业级x86架构刀片服务器等设备，并在一个统一的管理域中管理所有资源。用户可以通过在UCS上安装VMWare vSphere来支撑多达几千台虚拟机的运行。通过Cisco UCS，能够让企业快速在本地数据中心搭建基于虚拟化技术的云环境。

（4）Joyent。它提供基Open Solaris技术的IaaS服务。其IaaS服务中最核心的是Joyent SmartMachine。与大多数的IaaS服务不同的是，它并不是将底层硬件按照预计的额度直接分配给虚拟机，而是维护了一个大的资源池，让虚拟机上层的应用直接调用资源，并且这个资源池也有公平调度的功能，这样做的好处是优化资源的调配，并且易于应对流量突发情况，同时使用人员也无需过多关注操作系统级管理和运维。

### 27 并行计算和分布式计算什么区别？



1 并行计算、分布式计算、网格计算与云计算都属于高性能计算（High Performance Computing，HPC）的范畴，主要目的在于对大数据的分析与处理，但它们却存在很多差异。电子海图云服务是基于高性能计算的理论技术、通过对传统服务体系结构的改进，以实现海量电子海图数据的快速存取与处理操作，使其更好地为海洋地理信息科学领域中的计算密集型和数据密集型问题提供相应的计算和处理能力。高性能计算体系结构是海量电子海图数据存储与处理的技术基础，也是顺利进行电子海图云服务原型系统开发的保证。以下将对并行计算、分布式计算、网格计算与云计算之间的区别与联系做分析。

并行计算

2 并行计算是相对于串行计算而言的，它是指一种能够让多条指令同时进行的计算模式，可分为时间并行和空间并行。时间并行即利用多条流水线同时作业，空间并行是指使用多个处理器执行并发计算，以降低解决复杂问题所需要的时间。从程序开发人员的角度看，并行计算又可分为数据并行与功能并行，数据并行是通过对数据的分解实现相同子任务的并行作业，功能并行是通过对任务的分解实现相同数据不同任务的并行作业。相比较而言，数据并行较易实现，因此本文在并行算法研究时也将基于数据并行的原则进行设。对于并行计算的研究在上世纪70年代开始，就已有相关理论研究，如单指令多数据流（SIMD）与多指令多数据流（MIMD）的并行机制研究，到了80年代在并行体系结构方面有了很大成果，出现了利用网络组成多台计算机的并行结构与利用共享存储器组成多处理器的并行计算机，科学家利用这种并行计算结构在技术高精尖领域缩减了解决复杂问题的时间。

从以上分析可得出，早期的并行计算主要应用于科学研究领域，具有特定的应用环境，需要利用很高的技术技巧才能完成任务所需要的并行程序设计。虽然当时的并行计算离普

通大众还很遥远，但已经为复杂问题求解（如功能并行、数据并行、通信协调等）奠定了方法论基础。可以说，并行计算是云计算的初始阶段或者说是萌芽期，它为云计算的发展提供了实际而朴素的思想和基本思路。

分布式计算

3 分布式计算是一种把需要进行大量计算的工程数据分区成小块，由多台联网计算机分别处理，在上传处理结果后，将结果统一合并得出数据结论的科学。在90年代，随着TCP/IP协议的最终确定，计算机网络快速发展，Web Service等网络新技术随之而来，为基于广域网的分布式计算做好了硬件与软件基础。首先来比较下分布式计算与并行计算的异同。其相同之处都是将复杂任务化简为多个子任务，然后在多台计算机同时运算。不同之处在于分布式计算是一个比较松散的结构，实时性要求不高，可以跨越局域网在因特网部署运行，大量的公益性项目（如黑洞探索、药物研究、蛋白质结构分析等）大多采用这种方式，而并行计算是需要各节点之间通过高速网络进行较为频繁地通信，节点之间具有较强的关联性，主要部署在局域网内。在分布式计算的算法中，我们更加关注的是计算机间的通信而不是算法的步骤，因为分布式计算的通信代价比起单节点对整体性能的影响权重要大得多。

从以上分析可得出，分布式计算是网络发展的产物，是由并行计算演化出的新模式：网络并行计算。如果说并行计算为云计算奠定了理论基础，那么分布式计算则为云计算的实现打下了坚固的网络技术基石。

4 网格计算网格计算是指通过利用多个独立实体或机构中大量异构的计算机资源（处理器周期和磁盘存储），采用统一开放的标准化访问协议及接口，实现非集中控制式的资源访问与协同式的问题求解，以达到系统服务质量高于其每个网格系统成员服务质量累加的总和。

在90年代中期，分布式计算发展到一定阶段后，网格计算开始出现，其目的在于利用分散的网络资源解决密集型计算问题。当时由于高端的计算机硬件价格不菲，研究人员试图通过定义专门的协议机制以实现对分散异构且动态变化的网络资源管理，以解决高端计算机才能解决的密集型运算问题。网格计算与虚拟组织的概念由此产生，它通过定义一系列的标准协议、中间件以及工具包，以实现对虚拟组织中资源的分配和调度。它的焦点在于支持跨域计算与异构资源整合的能力，这使它与传统计算机集群或简单分布式计算相区别。为使网格计算能够成为类似于水电网的日常公共服务，Ian Foster提出应该定义一个在网络中获取计算或存储资源的标准协议，在这一理论指导下，世界各组织设计了一系列的网格系统，如OSG、ESG、EGEE等，这些网格系统能够按照设计人员的要求提供所需要的计算资存储资源甚至多种数据服务与功能服务。OASIS、OGF等国际标准组织也制定了相关标准，网格计算曾一度被认为是集群计算的市场化。然而迄今为止，商业化的网格系统仍然没有出现。过于庞大的概念、异常复杂的协议标准使得真正实现实用化的网格项目都是由国家行为推动的，如EUGrid、DataGrid、ChinaGrid、EduGrid等。然而网格计算的发展，为云计算的出现提供了基本的网络框架支持。

5 云计算

云计算是一种由大数据存储分析与资源弹性扩缩需求驱动的计算模式，它通过一个虚拟化、动态化、规模化的资源池为用户提供高可用性、高效性、弹性的计算存储资源与

数据功能服务。其具备五个关键特点：①基于分布式并行计算技术；②能够实现规模化、弹性化的计算存储；③用户服务的虚拟化与多级化；④受高性能计算与大数据存储驱动；⑤服务资源的动态化、弹性化。近年来云计算能够获得普遍关注的原因主要有以下三点：①设备存储计算能力的提升与成本的下降，多核、多处理器技术的诞生与普及；②各行业积累了越来越多的专业数据，亟需得到有效利用；③网络服务和Web2.0应用的广泛使用。

从以上分析可知，在概念层次上云计算与并行计算、集群计算、网格计算、分布式计算存在交叉，正如上一节所描述的云计算不仅是从网格计算演化来的，而且网格计算还为

云计算提供了基本的网络框架支持。网格计算的焦点在于计算与存储能力的提供，而云计算更注重于资源与服务能力的抽象，这就是网格计算向云计算的演化。与分布式计算比较，云计算是一种成熟稳定的流式商业资源，它为用户提供可量算的抽象服务就如同水电厂提供可量算的水电资源一样便捷可靠。图1.1显示了云计算与其他相关概念的关系。Web2.0诠释了面向服务的发展方向，云计算成为其中的主力；并行计算和集群计算更注重于传统面向应用的程序设计；网格计算由于其概念的庞大与这四个领域都有交叉，从广义的角度讲，分布式计算包含了整个概念域。

6 根据以上分析，我们可以得出这几个概念的关系。从计算机用户角度来讲，并行计算是由单个用户完成，分布式计算是由多用户合作完成，网格计算是由庞大的异构计算组织完成，云计算是没有用户参与由另一端的弹性服务集群完成。

### 28 maven package/install/deploy/build 命令，是否部署过私服Nexus？

1、maven的用途

　　maven是主要服务基于java平台的项目构建，依赖管理和项目信息管理的一个工具。项目构建包括清理，编译，测试到生成测试报告，再到打包和部署；依赖管理是maven借助于坐标来实现的。maven是跨平台的，可以在windows，linux和mac OS等系统上运行。

2、maven目录结构

（1）bin：maven运行脚本，命令。

（2）boot：包含plexus-classworlds-2.2.3.jar （plexus-classworlds是一个类加载器框架，相对于java 默认类加载器，他提供了更丰富的语法以方便配置，maven使用该框架加载自己的类库）。

（3）conf：配置文件 setting.xml，修改该文件就可以在机器上全局的制定maven的行为。但一般情况下，更倾向于复制该文件到~（用户目录）/.m2/目录下，然后修改该文件，在用户范围内制定maven的行为。

（4）lib：包含maven运行时所需要的java类库，包括第三方的依赖类库。

（5）LICENSE.txt：记录了maven使用该软件的许可证。

（6）NOTICE.txt：记录了maven包含的第三方软件。

（7）README.txt：记录了maven的简要介绍。

3、maven常用几个命令

（1）mvn help:system：该命令打印出所有的java系统属性和环境变量。

（2）mvn clean：清理输出目录默认target/。

（3）mvn clean compline：编译项目主代码，默认编译至target/classes目录下。

（4）mvn clean test：maven测试，但实际执行的命令有：clean:clean，resource:resources，compiler:compile, resources:testResources, compiler:testCompile，maven在执行test之前，会先自动执行项目主资源处理，主代码编译，测试资源处理，测试代码编译等工作，测试代码编译通过之后默认在target/test-calsses目录下生成二进制文件，紧接着surefile:test 任务运行测试，并输出测试报告，显示一共运行了多少次测试，失败成功等等。

（5）mvn celan package：maven打包，maven会在打包之前默认执行编译，测试等操作，打包成功之后默认输出在target/目录中。

（6）mvn clean install：maven安装，让其他的项目直接引用这个项目。

（7）echo %MAVEN\_HOME%：查看maven安装路径。

（8）mvn：检查是否安装了maven。

（9）mvn dependency:list：查看当前项目中的已解析依赖

（10）mvn dependency:tree：查看当前项目的依赖树

（11）mvn dependency:analyse： 查看当前项目中使用未声明的依赖和已声明但未使用的依赖

4、maven的项目结构

（1）项目骨架：项目的根目录下方pom.xml

（2）项目主代码：maven认为项目的主代码是位于src/main/java目录下的，所以通常我们会在此目录下创建文件，比如…/java/com/shuyun/myweb/hello/hello.java，该java类的包名是com.shuyun.myweb.hello

（3）资源文件：maven的资源文件位于src/main/resources目录下

（4）测试文件：maven默认的测试代码目录src/test/java目录

5、Nexus简单说明

用途：指定私服的中央地址、将自己的Maven项目指定到私服地址、从私服下载中央库的项目索引、从私服仓库下载依赖组件、将第三方项目jar上传到私服供其他项目组使用

仓库：

hosted 类型的仓库，内部项目的发布仓库

releases 内部的模块中release模块的发布仓库

snapshots 发布内部的SNAPSHOT模块的仓库

3rd party 第三方依赖的仓库，这个数据通常是由内部人员自行下载之后发布上去

proxy 类型的仓库，从远程中央仓库中寻找数据的仓库

group 类型的仓库，组仓库用来方便我们开发人员进行设置的仓库

### 29 Mybatis的优缺点？

MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以对配置和原生Map使用简单的 XML 或注解，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java对象)映射成数据库中的记录。

一、MyBatis框架的优点：

1. 与JDBC相比，减少了50%以上的代码量。

2. MyBatis是最简单的持久化框架，小巧并且简单易学。

3. MyBatis相当灵活，不会对应用程序或者数据库的现有设计强加任何影响，SQL写在XML里，从程序代码中彻底分离，降低耦合度，便于统一管理和优化，并可重用。

4. 提供XML标签，支持编写动态SQL语句。

5. 提供映射标签，支持对象与数据库的ORM字段关系映射。

二、MyBatis框架的缺点：

1. SQL语句的编写工作量较大，尤其是字段多、关联表多时，更是如此，对开发人员编写SQL语句的功底有一定要求。

2. SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

三、MyBatis框架适用场合：

MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。

对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

### 30 AOP的切面和连接点是什么？通知有哪几类？

一、关于AOP

1. 什么是 AOP?

Aspect Orentied Programming：面向切面编程

Object Orentied Programming：面向对象编程

AOP 编程是以 OOP 为基础，OOP 侧重点是对象抽象和封装，AOP 侧重点时共通处理部分的封装和使用，用于改善共通组件。是对 OOP 的补充和完善

2. AOP 的作用

分离系统中的各种关注点，将核心关注点和横切关注点分离开来。举个例子：开发中为了调试，或在生产环境后为了对系统进行监控，需要为业务需求的实现对象添加日志记录功能，或者，业务方法的执行需要一定的权限限制。如果以面向对象的方式实现，就需要把系统中的每个业务对象都加入日志记录，加入相应的安全检查，那么，这些实现代码就会遍及所有的业务对象中。如果对这些代码进行模块化的组织，简化系统需求与实现之间的对比关系，通过某种方式在适当的时候调用这些代码，那么就不用每个业务方法里都写着相同的代码了

3. 什么时候用AOP

日志功能

安全检查

事务管理

缓存

二、AOP 相关概念

1.Joinpoint

Joinpoint 就是进行织入操作的系统执行点

2.Pointcut

是 Joinpoint的表述方式，指定了系统中符合条件的一组 Joinpoint

3.Advice

Advice 代表的是织入到 Joinpoint 的横切逻辑

分类：

Before Advice：在指定位置之前执行，可以做一些初始化工作，比如设置系统初始值，获取必要系统资源

After Advice

After returning Advice ：当 当前 Joinpoint 正常执行完后，比如方法正常返回而没有抛出异常

After throwing Advice：当 当前 Joinpoint 执行过程中抛出异常的情况下，才会执行

After Advice：不管是否正常返回，都会执行，相当于 finally 块一样

Around Advice：可以在 Joinpoint 之前 或者 之后都能执行相应的逻辑，Filter功能就是其中的一种实现。可以用来系统安全验证及检查，简单的日志记录以及系统附加行为的添加

4.Aspect

是对系统中的横切关注点逻辑进行模块化封装的 AOP 概念实体，其可以包含多个 Pointcut 以及相关 Advice定义。

如果将 Aspect 比作 OOP 中的Class，那么 Advice 就相当于 CLass 中的 Method

5.织入和织入器

AspectJ 有专门的编译器来完成织入操作，ProxyFactory 是 Spring AOP 中最通用的织入器

6.目标对象

符合 Pointcut 所指定的条件，在织入过程中被织入横切逻辑的对象

三、Spring AOP 的实现机制

1. 实现机制

spring AOP 属于第二代AOP（动态AOP），采用动态代理机制和字节码生成技术实现，与最初的 AspectJ采用编译器将横切逻辑织入目标对象不同，动态代理机制和字节码生成都是在运行期间为目标对象生成一个代理对象，而将横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象

2.动态代理

一般的代理模式是有多少需要代理的对象就要写多少代理类，但是jdk 1.3 之后引入了一种称之为动态代理的机制，使用该机制，可以为指定的接口在系统运行期间动态的生成代理对象

实现：

java.lang.reflect.Proxy 类 和 java.lang.reflect.InvocatingHandler 接口。当 Proxy动态生成的代理对象上的相应的接口方法被调用时，对应的InvocationHandler就会拦截相应的方法调用，并进行相应的处理。InvocationHandler 就是实现横切逻辑的地方，是横切逻辑的载体

适用：

只能对实现了相应 Interface 的类使用。如果目标对象没有实现任何 Interface，spring AOP 会尝试使用一个称为 CGLIB 的开源的动态字节码生成类库，为目标对象生成动态的代理对象实例

3.动态字节码生成

原理：对目标对象进行继承扩展，为其生成相应的子类，而子类可以通过重写类扩展父类的行为，只要将横切逻辑的实现放到子类中，然后让系统使用扩展后的目标对象的子类，就OK了。（在系统运行期间动态为目标对象生成相应的扩展子类）

### 31自己开发一个线程池，有什么思路？

**（队列+调度+主任务）**

1 什么是线程池

线程池就是以一个或多个线程[循环执行]多个应用逻辑的线程集合.

一般而言,线程池有以下几个部分:

完成主要任务的一个或多个线程.

用于调度管理的管理线程.

要求执行的任务队列.

2 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。

3 自己实现线程池

根据如上对线程池的理解，我们自己编写一个属于自己的简单线程池:

4 简单的线程池接口:

public interface ThreadPool<Job extends Runnable>{

//执行一个任务(Job),这个Job必须实现Runnable

void execute(Job job);

//关闭线程池

void shutdown();

//增加工作者线程，即用来执行任务的线程

void addWorkers(int num);

//减少工作者线程

void removeWorker(int num);

//获取正在等待执行的任务数量

void getJobSize();

}

客户端可以通过execute(Job)方法将Job提交入线程池来执行，客户端完全不用等待Job的执行完成。除了execute(Job)方法以外，线程池接口提供了增加/减少工作者线程以及关闭线程池的方法。每个客户端提交的Job都会进入到一个工作队列中等待工作者线程的处理。

线程池接口的默认实现

public class DefaultThreadPool<Job extends Runnable> implements ThreadPool<Job>{

// 线程池维护工作者线程的最大数量

private static final int MAX\_WORKER\_NUMBERS = 10;

// 线程池维护工作者线程的默认值

private static final int DEFAULT\_WORKER\_NUMBERS = 5;

// 线程池维护工作者线程的最小数量

private static final int MIN\_WORKER\_NUMBERS = 1;

// 维护一个工作列表,里面加入客户端发起的工作

private final LinkedList<Job> jobs = new LinkedList<Job>();

// 工作者线程的列表

private final List<Worker> workers = Collections.synchronizedList(new ArrayList<Worker>());

// 工作者线程的数量

private int workerNum;

// 每个工作者线程编号生成

private AtomicLong threadNum = new AtomicLong();

//生成线程池

public DefaultThreadPool() {

this.workerNum = DEFAULT\_WORKER\_NUMBERS;

initializeWorkers(this.workerNum);

}

public DefaultThreadPool(int num) {

if (num > MAX\_WORKER\_NUMBERS) {

this.workerNum =DEFAULT\_WORKER\_NUMBERS;

} else {

this.workerNum = num;

}

initializeWorkers(this.workerNum);

}

//初始化每个工作者线程

private void initializeWorkers(int num) {

for (int i = 0; i < num; i++) {

Worker worker = new Worker();

//添加到工作者线程的列表

workers.add(worker);

//启动工作者线程

Thread thread = new Thread(worker);

thread.start();

}

}

public void execute(Job job) {

if (job != null) {

//根据线程的"等待/通知机制"这里必须对jobs加锁

synchronized (jobs) {

jobs.addLast(job);

jobs.notify();

}

}

}

//关闭线程池即关闭每个工作者线程

public void shutdown() {

for (Worker w : workers) {

w.shutdown();

}

}

//增加工作者线程

public void addWorkers(int num) {

//加锁，防止该线程还么增加完成而下个线程继续增加导致工作者线程超过最大值

synchronized (jobs) {

if (num + this.workerNum > MAX\_WORKER\_NUMBERS) {

num = MAX\_WORKER\_NUMBERS - this.workerNum;

}

initializeWorkers(num);

this.workerNum += num;

}

}

//减少工作者线程

public void removeWorker(int num) {

synchronized (jobs) {

if(num>=this.workerNum){

throw new IllegalArgumentException("超过了已有的线程数量");

}

for (int i = 0; i < num; i++) {

Worker worker = workers.get(i);

if (worker != null) {

//关闭该线程并从列表中移除

worker.shutdown();

workers.remove(i);

}

}

this.workerNum -= num;

}

}

public int getJobSize() {

// TODO Auto-generated method stub

return workers.size();

}

//定义工作者线程

class Worker implements Runnable {

// 表示是否运行该worker

private volatile boolean running = true;

public void run() {

while (running) {

Job job = null;

//线程的等待/通知机制

synchronized (jobs) {

if (jobs.isEmpty()) {

try {

jobs.wait();//线程等待唤醒

} catch (InterruptedException e) {

//感知到外部对该线程的中断操作，返回

Thread.currentThread().interrupt();

return;

}

}

// 取出一个job

job = jobs.removeFirst();

}

//执行job

if (job != null) {

job.run();

}

}

}

// 终止该线程

public void shutdown() {

running = false;

}

}

}

从线程池的实现中可以看出，当客户端调用execute(Job)方法时，会不断地向任务列表jobs中添加Job，而每个工作者线程会不读的从jobs上获取Job来执行，当jobs为空时，工作者线程进入WAITING状态。

当添加一个Job后，对工作队列jobs调用其notify()方法来唤醒一个工作者线程。此处我们不调用notifyAll(),避免将等待队列中的线程全部移动到阻塞队列中而造成资源浪费。

线程池的本质就是使用了一个线程安全的工作队列连接工作者线程和客户端线程。客户端线程把任务放入工作队列后便返回，而工作者线程则不端的从工作队列中取出工作并执行。当工作队列为空时，

工作者线程进入WAITING状态，当有客户端发送任务过来后会通过任意一个工作者线程，随着大量任务的提交，更多的工作者线程被唤醒。

### 32什么是开闭原则？

　 软件开发之所以会有这些原则，就是因为复杂多变且不可预料的需求。并不是说在实际项目开发中对这六大原则中的每一条都遵循到极致，而是说在项目开发的过程中，

根据项目的实际需求尽量的去遵守这些原则。当然要做到这些肯定是不容易的，能真正做到并且做好的恐怕也只能是有经验之人。

（1）Single Responsibility Principle　 : 单一职责原则，应该有且只有一个原因引起类的变更。换句话说就是一个接口只做一件事，即一个职责一个接口。但是困难的是划分职责时并没有一个标准，最终都是需要从实际的

项目去考虑。我们在设计的时候，尽量单一，然后对于其实现类就要多方面的考虑。不能死套单一职责原则，否则会增加很多类，给维护带来不便。

　　（2）Liskov Substitution Principle : 里氏替换原则，里氏替换原则简单易懂一点的定义就是只要父类出现的地方子类就可以出现，且替换成子类也不会出现任何错误或者异常。(但是反过来，有子类出现的地方，父类不一定可以适用)。

　　（3）Dependence Inversion Principle ：依赖倒置原则，精简的定义就是面向接口编程。在Java语言中的表现就是为以下的三点

　　　　① 模块间的依赖关系通过接口和抽象类产生，实体类之间不直接发生依赖关系。

　 ②接口和抽象类不依赖于实现类。

　　 ③实现类依赖接口或者抽象类。

　　（4）Interface Segregation Principle : 接口隔离原则， 建立单一接口，不要建立臃肿庞大的接口。即接口尽量细化，同时接口中的方法尽量少。在这里提一下单一职责和接口隔离原则的区别。首先两个侧重点是不一样的，单一职责要求类和接口，或者方法的职责单一，侧重点在职责，这是根据业务逻辑进行划分的。而接口隔离原则要接口中的方法尽量少。比如，一个接口或者一个中有十个方法，不同的方法做不同的事情，但是这个接口总体就是处理一件事情，然后具体细分成了10个方法。不同的模块根据不同的权限进行访问，这是符单一职责原则的。但是按照接口隔离的原则是要求接口接口中的方法尽量少，落实到这个实例就是要求尽量多几个专门的接口供不同的模块使用，而不是只有一个臃肿的接口，依据权限去限制不同模块可以访问的方法。

　　（5）Law of Demeter　　　　 : 迪米特法则，迪米特法则也叫做最少知识原则(Least Knowledge Principle,LKP)，即一个对象应该对其他对象有最少的了解，也就是说一个类要对自己需要耦合或者调用的类知道的最少。我只知道你有多少public方法可以供我调用，而其他的一切都与我无关。

　　（6）Open Closed Principle : 开闭原则，开闭原则是Java里最基础的设计原则。具体的定义是：一个软件实体，比如类，模块，函数应该对扩展开放，对修改关闭。说的通熟易懂一些就是一个软件实体应该通过扩展来实现变化，而不是通过修改已有的代码来实现改变。

### 33工厂方法模式和抽象工厂设计模式有什么区别？

开闭原则：对扩展开放，对修改封闭。静态工厂增加需要是修改源代码，对修改不封闭，不符合开闭原则。

一、Simple Factory 简单工厂模式（静态工厂）

1）Simple Factory模式属于创建型模式

2）简单工厂模式是由一个工厂（注意是一个！）对象决定创建出哪一种产品类的实例（例如你到肯德基说你要鸡腿，要薯条，要饮料还是其他的，这时肯德基是一个工厂，客户端只需要点明自己要什么就行）

3）实现方式的实质：由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（这些产品类继承自一个父类或接口）的实例。

UML类图如下：

这里写图片描述

分析：

工厂角色：被客户端直接调用，根据客户端指定传入的参数，动态创建客户端需要的对象；

抽象产品角色：所有对象的父类（接口）；

具体产品角色：即工厂的创建目标，工厂创建的对象就是这些具体类的对象。

可以看出，客户端只面对工厂，不用管产品的具体细节，客户只需向工厂要求你需要什么，其他的事情都交给工厂了。

优点：

通过使用工厂类,外界可以从直接创建具体产品对象的尴尬局面摆脱出来,仅仅需要负责“消费”对象就可以了。而不必管这些对象究竟如何创建及如何组织的．明确了各自的职责和权利。（简单地说，你到肯德基去只需要说你要鸡腿还是鸡翅就行了，不需要去管鸡腿和鸡翅是怎么做出来的，工厂为你提供了这样一个界面）。

不足：

由于工厂类集中了所有实例的创建逻辑，违反了高内聚责任分配原则，将全部创建逻辑集中到了一个工厂类中；它所能创建的类只能是事先考虑到的，如果需要添加新的类，则就需要改变工厂类了。

当系统中的具体产品类不断增多时候，可能会出现要求工厂类根据不同条件创建不同实例的需求．这种对条件的判断和对具体产品类型的判断交错在一起，很难避免模块功能的蔓延，对系统的维护和扩展非常不利；

使用场景

工厂类负责创建的对象比较少；

客户只知道传入工厂类的参数，对于如何创建对象（逻辑）不关心；

由于简单工厂很容易违反高内聚责任分配原则，因此一般只在很简单的情况下应用。

从上面的例子也可以看出来，工厂类往往是用反射机制来产生具体对象的。（因为不同类都继承自同一接口），故其扩展性很有限，产品种类必须是事先就知道的哪几种，什么时候你想要添加一个不是公共接口下的具体类就不行了。

另外，如果你不用反射机制，也不要公共接口，在工厂中使用其他逻辑（例如判断传入的字符串）来根据用户参数创建对象也行，那样扩展性也是很糟糕的，逻辑和添加只会越来多。

工厂方法模式

这个和简单工厂有区别，简单工厂模式只有一个工厂，工厂方法模式对每一个产品都有相应的工厂。

好处：增加一个运算类（例如N次方类），只需要增加运算类和相对应的工厂，两个类，不需要修改工厂类。

缺点：增加运算类，会修改客户端代码，工厂方法只是把简单工厂的内部逻辑判断移到了客户端进行。

二、Abstract Factory 抽象工厂

UML类图如下：

这里写图片描述

参与者

AbstractFactory——声明一个创建抽象产品对象的操作接口

ConcreteFactory——实现创建具体产品对象的操作

AbstractProduct——为一类产品对象声明一个接口

ConcreteProduct

① 定义一个将被相应的具体工厂创建的产品对象

② 实现AbstractProduct接口

Client

① 仅使用由AbstractFactory和AbstractProduct类声明的接口

协作

① 通常在运行时刻创建一个ConcreteFactory类的实例。这一具体的工厂创建具有特定实现的产品对象。为创建不同的产品对象，客户应适用不同的具体工厂。

②AbstractFactory将产品对象的创建延迟到它的ConcreteFactory子类。

三、工厂方法模式：

① 一个抽象产品类，可以派生出多个具体产品类。

② 一个抽象工厂类，可以派生出多个具体工厂类。

③ 每个具体工厂类只能创建一个具体产品类的实例。

抽象工厂模式：

① 多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。

② 一个抽象工厂类，可以派生出多个具体工厂类。

③ 每个具体工厂类可以创建多个具体产品类的实例。

区别：

① 工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个。

② 工厂方法模式的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个。

### 34 nginx静态代理怎么理解，应用什么场景？

github 中很多项目都有一个 readme 文件，很多人喜欢在文件中添加自己的创作或封面图片，比如 substack 为他的每个项目绘制了一个 logo。这些图片在 github 中能直接在页面中显示出来，不过 url 被替换成了 github 自己的。比如在 browserify 项目中，logo 的链接变成了

https://camo.githubusercontent.com/e19e230a9371a44a2eeb484b83ff4fcf8c824cf7/687474703a2f2f737562737461636b2e6e65742f696d616765732f62726f777365726966795f6c6f676f2e706e67

而我们通过查看 raw 能发现原 url 是

http://substack.net/images/browserify\_logo.png

这样做的一个好处是防止因为在 https 网站中出现 http 链接，否则在客户端会得到一个风险警告。github 在细节上真是考虑的十分周到。

既然有需求，我们就来实现它。通常的做法是写一个应用去抓取远程的静态资源，然后反馈给前端，这就是一个简单地反向代理了。但是这样做比较繁琐，效率也未见得高，其实我们可以直接通过 nginx 来代理这些静态文件。

nginx 的 proxy\_pass 支持填写任意地址，并且支持 dns 解析。所以我的思路是，将原 url 加密转成网站自身的 url。比如上面的

http://substack.net/images/browserify\_logo.png

可以加密成

764feebffb1d3f877e9e0d0fadcf29b85e8fe84ae4ce52f7dc4ca4b3e05bf1718177870a996fe5804a232fcae5b893ea (加密和序列化算法网上有很多，在此就不赘述了)

然后放在我们自己的域名下：

https://ssl.youdomain.com/camo/764feebffb1d3f877e9e0d0fadcf29b85e8fe84ae4ce52f7dc4ca4b3e05bf1718177870a996fe5804a232fcae5b893ea

解密的步骤用 nginx 会比较难实现，所以当用户通过上述链接请求时，先讲请求传递给解密程序，这里有一个 coffeescript 版本的例子：

express = require 'express'

app = express()

app.get '/camo/:eurl', (req, res) ->

{eurl} = req.params

{camoSecret} = config # 这里使用自己的密钥

rawUrl = util.decrypt eurl, camoSecret

return res.status(403).end('INVALID URL') unless rawUrl

res.set 'X-Origin-Url', rawUrl

res.set 'X-Accel-Redirect', '/remote'

res.end()

app.listen 3000

然后写入 X-Accel-Redirect 响应头做内部跳转，下面的步骤就由 nginx 完成了。

下面是一个完整的 nginx 配置文件例子：

server {

listen 80;

server\_name ssl.youdomain.com;

location /camo/ {

proxy\_pass http://localhost:3000;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $http\_host;

proxy\_set\_header X-NginX-Proxy true;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection "Upgrade";

proxy\_redirect off;

break;

}

location /remote {

internal;

resolver 192.168.0.21; # 必须加上 dns 服务器地址，否则 nginx 无法解析域名

set $origin\_url $upstream\_http\_x\_origin\_url;

proxy\_pass $origin\_url;

add\_header Host "file.local.com";

break;

}

}

nginx 的 upstream 模块会把所有的响应头加上 $upstream\_http\_ 前缀当成一个变量保存，所以在上面的例子中我们将原 url 放在 X-Origin-Url 响应头中，在 nginx 就变成了 $upstream\_http\_x\_origin\_url 变量，但是在 proxy\_pass 中不能直接引用，非要通过 set 来设置才能引用，这个我不是很理解，希望有高手能解答。

这样下来，每次当用户请求

https://ssl.youdomain.com/camo/764feebffb1d3f877e9e0d0fadcf29b85e8fe84ae4ce52f7dc4ca4b3e05bf1718177870a996fe5804a232fcae5b893ea

时，nginx 就会去抓取

http://substack.net/images/browserify\_logo.png

的内容返回给用户。我们还可以在 nginx 之前加上 varnish，用以缓存静态文件的内容。这样就跟 githubusercontent 的做法更加一致了。

### 35 kafka怎么防止数据丢失？吞吐量所少？

Kafka到底会不会丢数据(data loss)? 通常不会，但有些情况下的确有可能会发生。下面的参数配置及Best practice列表可以较好地保证数据的持久性(当然是trade-off，牺牲了吞吐量)。笔者会在该列表之后对列表中的每一项进行讨论，有兴趣的同学可以看下后面的分析。

block.on.buffer.full = true

acks = all

retries = MAX\_VALUE

max.in.flight.requests.per.connection = 1

使用KafkaProducer.send(record, callback)

callback逻辑中显式关闭producer：close(0)

unclean.leader.election.enable=false

replication.factor = 3

min.insync.replicas = 2

replication.factor > min.insync.replicas

enable.auto.commit=false

消息处理完成之后再提交位移

给出列表之后，我们从两个方面来探讨一下数据为什么会丢失：

1. Producer端

　　目前比较新版本的Kafka正式替换了Scala版本的old producer，使用了由Java重写的producer。新版本的producer采用异步发送机制。KafkaProducer.send(ProducerRecord)方法仅仅是把这条消息放入一个缓存中(即RecordAccumulator，本质上使用了队列来缓存记录)，同时后台的IO线程会不断扫描该缓存区，将满足条件的消息封装到某个batch中然后发送出去。显然，这个过程中就有一个数据丢失的窗口：若IO线程发送之前client端挂掉了，累积在accumulator中的数据的确有可能会丢失。

　　Producer的另一个问题是消息的乱序问题。假设客户端代码依次执行下面的语句将两条消息发到相同的分区

producer.send(record1);

producer.send(record2);

如果此时由于某些原因(比如瞬时的网络抖动)导致record1没有成功发送，同时Kafka又配置了重试机制和max.in.flight.requests.per.connection大于1(默认值是5，本来就是大于1的)，那么重试record1成功后，record1在分区中就在record2之后，从而造成消息的乱序。很多某些要求强顺序保证的场景是不允许出现这种情况的。

　　鉴于producer的这两个问题，我们应该如何规避呢？？对于消息丢失的问题，很容易想到的一个方案就是：既然异步发送有可能丢失数据， 我改成同步发送总可以吧？比如这样：

producer.send(record).get();

这样当然是可以的，但是性能会很差，不建议这样使用。因此特意总结了一份配置列表。个人认为该配置清单应该能够比较好地规避producer端数据丢失情况的发生：(特此说明一下，软件配置的很多决策都是trade-off，下面的配置也不例外：应用了这些配置，你可能会发现你的producer/consumer 吞吐量会下降，这是正常的，因为你换取了更高的数据安全性)

block.on.buffer.full = true 尽管该参数在0.9.0.0已经被标记为“deprecated”，但鉴于它的含义非常直观，所以这里还是显式设置它为true，使得producer将一直等待缓冲区直至其变为可用。否则如果producer生产速度过快耗尽了缓冲区，producer将抛出异常

acks=all 很好理解，所有follower都响应了才认为消息提交成功，即"committed"

retries = MAX 无限重试，直到你意识到出现了问题:)

max.in.flight.requests.per.connection = 1 限制客户端在单个连接上能够发送的未响应请求的个数。设置此值是1表示kafka broker在响应请求之前client不能再向同一个broker发送请求。注意：设置此参数是为了避免消息乱序

使用KafkaProducer.send(record, callback)而不是send(record)方法 自定义回调逻辑处理消息发送失败

callback逻辑中最好显式关闭producer：close(0) 注意：设置此参数是为了避免消息乱序

unclean.leader.election.enable=false 关闭unclean leader选举，即不允许非ISR中的副本被选举为leader，以避免数据丢失

replication.factor >= 3 这个完全是个人建议了，参考了Hadoop及业界通用的三备份原则

min.insync.replicas > 1 消息至少要被写入到这么多副本才算成功，也是提升数据持久性的一个参数。与acks配合使用

保证replication.factor > min.insync.replicas 如果两者相等，当一个副本挂掉了分区也就没法正常工作了。通常设置replication.factor = min.insync.replicas + 1即可

2. Consumer端

　　consumer端丢失消息的情形比较简单：如果在消息处理完成前就提交了offset，那么就有可能造成数据的丢失。由于Kafka consumer默认是自动提交位移的，所以在后台提交位移前一定要保证消息被正常处理了，因此不建议采用很重的处理逻辑，如果处理耗时很长，则建议把逻辑放到另一个线程中去做。为了避免数据丢失，现给出两点建议：

enable.auto.commit=false 关闭自动提交位移

在消息被完整处理之后再手动提交位移

### 36 Java 内存模型中的可见性、原子性和有序性怎么理解？

Java内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性这3个特征来建立的。

**原子性（Atomicity）**：由Java内存模型来直接保证的原子性变量操作包括read、load、use、assign、store、write，我们大致可以认为基本数据类型的访问读写是具备原子性的。

如果需要一个更大范围的原子性操作，Java内存模型提供了lock和unlock操作来满足这种需求，尽管虚拟机未把lock和unlock操作直接开放给用户使用，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐式地使用这两个操作，这两个字节码指令反应到Java代码中就是同步块---synchronized关键字，所以在synchronized块之间的操作也具备原子性。

**可见性（Visibility）**：可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。volatile、synchronized、final都可以实现可见性，synchronized可见性是由对一个变量执行unlock操作之前，必须先把此变量同步回主内存中（执行store、write操作）这条规则获得的， 而final关键字的可见性是指：被final修饰的字段在构造器中一旦被初始化完成，并且构造器没有把this的引用传递出去，那在其他线程中就能见final字段的值。

**有序性（Ordering）**：Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的。前半句是指线程内表现为串行的语义（within-Thread AS-if-Serial Semantics），后半句是指指令重排序现象和工作内存和猪呢从同步延迟现象。

Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由一个变量在同一时刻只允许一条线程对其进行lock做这条规则获得的，这条规则决定了持有同一个锁的两个同步块只能串行的进入。

先行发生原则（happens-before）是针对于内存模型中的有序性来说的。先行发生是java内存模型中定义的两项操作之间的偏序关系。

以下是java内存模型下一些天然的先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用，如果通过下面关系无法确定顺序性，虚拟机可以对其随意地进行重排序。

程序次序规则（Program Order Rule）：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为考虑分支、循环等结构。

管程锁定规则（Monitor Lock Rule）：一个unlock操作先行发生于后面对同一个锁的lock操作，这里必须强调的是同一个多，而“后面”是指时间上的先后顺序。

volatile变量规则（Volatile Variable Rule）：对一个volatile变量的写操作先行发生于后面对这个变量的读操作，“后面”是指时间上的先后顺序。

线程启动规则（Thread Start Rule）：Thread对象的start（）方法先行发生于此线程的每一个动作。

线程终止规则（Thread Termination Rule）：线程中的所有操作都先行发生于对此线程的终止检测，可以通过Thread.join()方法结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。

线程中断规则（Thread Interruption Rule）：对线程interrupt（）方法的调用先行发生于被中断线程的代码检测到中断时间的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。

对象终结规则（Finalizer Rule）：一个对象的初始化完成（构造函数执行结束）先行发生于它的finalize（）方法的开始。

传递性（Transitivity）：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论。

时间先后顺序与先行发生原则之间基本没有太大的关系，所以衡量并发安全问题的时候不要受到时间顺序的干扰，一切必须以先行发生原则为准。

### 37 有没使用过[thread dump和heap dump](http://www.cnblogs.com/toSeeMyDream/p/7151635.html)？

一.dump基本概念

        在故障定位(尤其是out of memory)和性能分析的时候，经常会用到一些文件来帮助我们排除代码问题。这些文件记录了JVM运行期间的内存占用、线程执行等情况，这就是我们常说的dump文件。常用的有heap dump和thread dump（也叫javacore，或java dump）。我们可以这么理解：heap dump记录内存信息的，thread dump是记录CPU信息的。

        heap dump：

        heap dump文件是一个二进制文件，它保存了某一时刻JVM堆中对象使用情况。HeapDump文件是指定时刻的Java堆栈的快照，是一种镜像文件。Heap Analyzer工具通过分析HeapDump文件，哪些对象占用了太多的堆栈空间，来发现导致内存泄露或者可能引起内存泄露的对象。

        thread dump：

        thread dump文件主要保存的是java应用中各线程在某一时刻的运行的位置，即执行到哪一个类的哪一个方法哪一个行上。thread dump是一个文本文件，打开后可以看到每一个线程的执行栈，以stacktrace的方式显示。通过对thread dump的分析可以得到应用是否“卡”在某一点上，即在某一点运行的时间太长，如数据库查询，长期得不到响应，最终导致系统崩溃。单个的thread dump文件一般来说是没有什么用处的，因为它只是记录了某一个绝对时间点的情况。比较有用的是，线程在一个时间段内的执行情况。

两个thread dump文件在分析时特别有效，困为它可以看出在先后两个时间点上，线程执行的位置，如果发现先后两组数据中同一线程都执行在同一位置，则说明此处可能有问题，因为程序运行是极快的，如果两次均在某一点上，说明这一点的耗时是很大的。通过对这两个文件进行分析，查出原因，进而解决问题。

二.利用JDK自带的工具获取thread dump文件和heap dump文件

        使用的JDK工具在JDK\_HOME/bin/目录下，使用到jmap和jstack这两个命令。

1.获取heap dump文件

        windows下切换到JDK\_HOME/bin/，执行以下命令：jmap -dump:format=b,file=heap.hprof 2576

        linux下切换到JDK\_HOME/bin/，执行以下命令：./jmap -dump:format=b,file=heap.hprof 2576

        这样就会在当前目录下生成heap.hprof文件，这就是heap dump文件。

2.获取thread dump文件

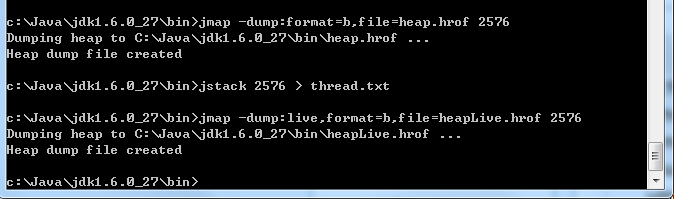
        windows下执行：jstack 2576 > thread.txt

        linux下执行：./jstack 2576 > thread.txt

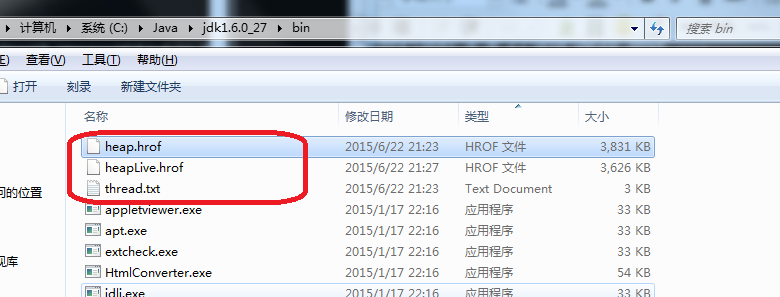
        windows/linux则会将命令执行结果转储到thread.txt，这就是thread dump文件。有了dump文件后，我们就能借助性能分析工具获取dump文件中的信息。

3.如果我们只需要将dump中存活的对象导出，那么可以使用:live参数

jmap -dump:live,format=b,file=heapLive.hprof 2576



        执行完后，我们在当前目录C:\Java\jdk1.6.0\_27\bin下看到刚生成的三个文件，如下所示：

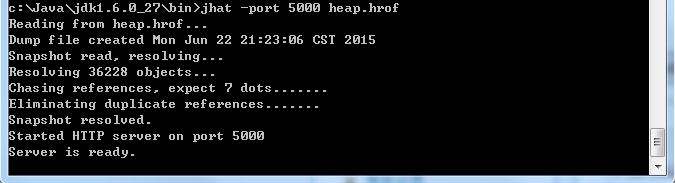
  
        说明：如上实例的2576是我当前需要分析的java进程PID，关于Windows下如何获得指定的JAVA时空程PID可参考：<http://bijian1013.iteye.com/blog/2221238>

三.使用工具分析java heap dump文件

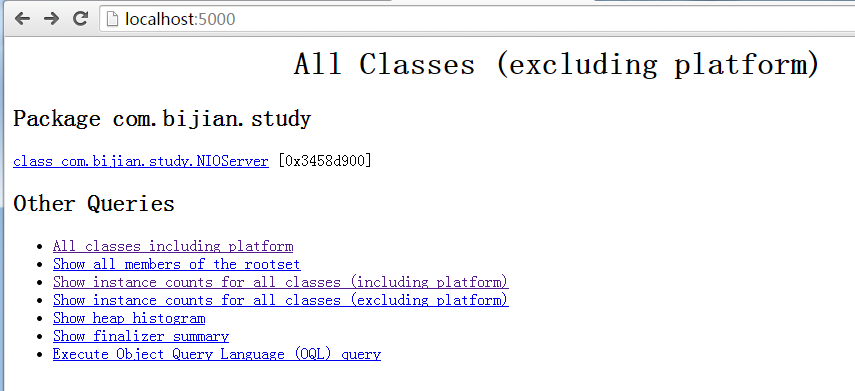
        现在我们使用一些图形化工具，来帮助我们分析文件中的信息，有效地定位问题。

1.使用JDK自带的jhat命令

        jhat是用来分析java堆的命令，可以将堆中的对象以html的形式显示出来，包括对象的数量，大小等等，并支持对象查询语言。

jhat -port 5000 heap.hrof

        当服务启动完成后，我们就可以在浏览器中，通过http://localhost:5000/进行访问，如下所示：

  
2.使用eclipse MAT工具

        一般来说，应用程序的dump文件都是很大的，jdk自带命令难以分析这些大文件。在实际的生产环境下，我们必须要借助第三方工具，才能快速打开这些大文件，进行分析定位。eclipse memory analyzer是一款优秀的heap分析工具，能够帮我们快速定位内存泄露问题。

### 38 tomcat给你你怎样去调优?

Tomcat 的缺省配置是不能稳定长期运行的，也就是不适合生产环境，它会死机，让你不断重新启动，甚至在午夜时分唤醒你。对于操作系统优化来说，是尽可能的增大可使用的内存容量、提高CPU 的频率，保证文件系统的读写速率等。经过压力测试验证，在并发连接很多的情况下，CPU 的处理能力越强，系统运行速度越快。

Tomcat 的优化不像其它软件那样，简简单单的修改几个参数就可以了，它的优化主要有三方面，分为系统优化，Tomcat 本身的优化，Java 虚拟机（JVM）调优。系统优化就不在介绍了，接下来就详细的介绍一下 Tomcat 本身与 JVM 优化，以 Tomcat 7 为例。

一、Tomcat 本身优化

Tomcat 的自身参数的优化，这块很像 ApacheHttp Server。修改一下 xml 配置文件中的参数，调整最大连接数，超时等。此外，我们安装 Tomcat 是，优化就已经开始了。

1、工作方式选择

为了提升性能，首先就要对代码进行动静分离，让 Tomcat 只负责 jsp 文件的解析工作。如采用 Apache 和 Tomcat 的整合方式，他们之间的连接方案有三种选择，JK、http\_proxy 和 ajp\_proxy。相对于 JK 的连接方式，后两种在配置上比较简单的，灵活性方面也一点都不逊色。但就稳定性而言不像JK 这样久经考验，所以建议采用 JK 的连接方式。

2、Connector 连接器的配置

之前文件介绍过的 Tomcat 连接器的三种方式： bio、nio 和 apr，三种方式性能差别很大，apr 的性能最优， bio 的性能最差。而 Tomcat 7 使用的 Connector 默认就启用的 Apr 协议，但需要系统安装 Apr 库，否则就会使用 bio 方式。

3、配置文件优化

配置文件优化其实就是对 server.xml 优化，可以提大大提高 Tomcat 的处理请求的能力，下面我们来看 Tomcat 容器内的优化。

默认配置下，Tomcat 会为每个连接器创建一个绑定的线程池（最大线程数 200），服务启动时，默认创建了 5 个空闲线程随时等待用户请求。

首先，打开 ${TOMCAT\_HOME}/conf/server.xml，搜索【<Executor name="tomcatThreadPool"】，开启并调整为

<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"

maxThreads="500" minSpareThreads="20" maxSpareThreads="50" maxIdleTime="60000"/>

注意， Tomcat 7 在开启线程池前，一定要安装好 Apr 库，并可以启用，否则会有错误报出，shutdown.sh 脚本无法关闭进程。

然后，修改<Connector …>节点，增加 executor 属性，搜索【port="8080"】，调整为

<Connector executor="tomcatThreadPool"

port="8080" protocol="HTTP/1.1"

URIEncoding="UTF-8"

connectionTimeout="30000"

enableLookups="false"

disableUploadTimeout="false"

connectionUploadTimeout="150000"

acceptCount="300"

keepAliveTimeout="120000"

maxKeepAliveRequests="1"

compression="on"

compressionMinSize="2048"

compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain,image/gif,image/jpg,image/png"

redirectPort="8443" />

maxThreads :Tomcat 使用线程来处理接收的每个请求，这个值表示 Tomcat 可创建的最大的线程数，默认值是 200

minSpareThreads：最小空闲线程数，Tomcat 启动时的初始化的线程数，表示即使没有人使用也开这么多空线程等待，默认值是 10。

maxSpareThreads：最大备用线程数，一旦创建的线程超过这个值，Tomcat 就会关闭不再需要的 socket 线程。

上边配置的参数，最大线程 500（一般服务器足以），要根据自己的实际情况合理设置，设置越大会耗费内存和 CPU，因为 CPU 疲于线程上下文切换，没有精力提供请求服务了，最小空闲线程数 20，线程最大空闲时间 60 秒，当然允许的最大线程连接数还受制于操作系统的内核参数设置，设置多大要根据自己的需求与环境。当然线程可以配置在“tomcatThreadPool”中，也可以直接配置在“Connector”中，但不可以重复配置。

URIEncoding：指定 Tomcat 容器的 URL 编码格式，语言编码格式这块倒不如其它 WEB 服务器软件配置方便，需要分别指定。

connnectionTimeout： 网络连接超时，单位：毫秒，设置为 0 表示永不超时，这样设置有隐患的。通常可设置为 30000 毫秒，可根据检测实际情况，适当修改。

enableLookups： 是否反查域名，以返回远程主机的主机名，取值为：true 或 false，如果设置为false，则直接返回IP地址，为了提高处理能力，应设置为 false。

disableUploadTimeout：上传时是否使用超时机制。

connectionUploadTimeout：上传超时时间，毕竟文件上传可能需要消耗更多的时间，这个根据你自己的业务需要自己调，以使Servlet有较长的时间来完成它的执行，需要与上一个参数一起配合使用才会生效。

acceptCount：指定当所有可以使用的处理请求的线程数都被使用时，可传入连接请求的最大队列长度，超过这个数的请求将不予处理，默认为100个。

keepAliveTimeout：长连接最大保持时间（毫秒），表示在下次请求过来之前，Tomcat 保持该连接多久，默认是使用 connectionTimeout 时间，-1 为不限制超时。

maxKeepAliveRequests：表示在服务器关闭之前，该连接最大支持的请求数。超过该请求数的连接也将被关闭，1表示禁用，-1表示不限制个数，默认100个，一般设置在100~200之间。

compression：是否对响应的数据进行 GZIP 压缩，off：表示禁止压缩；on：表示允许压缩（文本将被压缩）、force：表示所有情况下都进行压缩，默认值为off，压缩数据后可以有效的减少页面的大小，一般可以减小1/3左右，节省带宽。

compressionMinSize：表示压缩响应的最小值，只有当响应报文大小大于这个值的时候才会对报文进行压缩，如果开启了压缩功能，默认值就是2048。

compressableMimeType：压缩类型，指定对哪些类型的文件进行数据压缩。

noCompressionUserAgents="gozilla, traviata"： 对于以下的浏览器，不启用压缩。

如果已经对代码进行了动静分离，静态页面和图片等数据就不需要 Tomcat 处理了，那么也就不需要配置在 Tomcat 中配置压缩了。

以上是一些常用的配置参数属性，当然还有好多其它的参数设置，还可以继续深入的优化，HTTP Connector 与 AJP Connector 的参数属性值，可以参考官方文档的详细说明：

https://tomcat.apache.org/tomcat-7.0-doc/config/http.html

https://tomcat.apache.org/tomcat-7.0-doc/config/ajp.html

二、JVM 优化

Tomcat 启动命令行中的优化参数，就是 JVM 的优化 。Tomcat 首先跑在 JVM 之上的，因为它的启动其实也只是一个 java 命令行，首先我们需要对这个 JAVA 的启动命令行进行调优。不管是 YGC 还是 Full GC，GC 过程中都会对导致程序运行中中断，正确的选择不同的 GC 策略，调整 JVM、GC 的参数，可以极大的减少由于 GC 工作，而导致的程序运行中断方面的问题，进而适当的提高 Java 程序的工作效率。但是调整 GC 是以个极为复杂的过程，由于各个程序具备不同的特点，如：web 和 GUI 程序就有很大区别（Web可以适当的停顿，但GUI停顿是客户无法接受的），而且由于跑在各个机器上的配置不同（主要 cup 个数，内存不同），所以使用的 GC 种类也会不同。

1、JVM 参数配置方法

Tomcat 的启动参数位于安装目录 ${JAVA\_HOME}/bin目录下，Linux 操作系统就是 catalina.sh 文件。JAVA\_OPTS，就是用来设置 JVM 相关运行参数的变量，还可以在 CATALINA\_OPTS 变量中设置。关于这 2 个变量，还是多少有些区别的：

JAVA\_OPTS：用于当 Java 运行时选项“start”、“stop”或“run”命令执行。

CATALINA\_OPTS：用于当 Java 运行时选项“start”或“run”命令执行。

为什么有两个不同的变量？它们之间都有什么区别呢？

首先，在启动 Tomcat 时，任何指定变量的传递方式都是相同的，可以传递到执行“start”或“run”命令中，但只有设定在 JAVA\_OPTS 变量里的参数被传递到“stop”命令中。对于 Tomcat 运行过程，可能没什么区别，影响的是结束程序，而不是启动程序。

第二个区别是更微妙，其他应用程序也可以使用 JAVA\_OPTS 变量，但只有在 Tomcat 中使用 CATALINA\_OPTS 变量。如果你设置环境变量为只使用 Tomcat，最好你会建议使用 CATALINA\_OPTS 变量，而如果你设置环境变量使用其它的 Java 应用程序，例如 JBoss，你应该把你的设置放在JAVA\_OPTS 变量中。

2、JVM 参数属性

32 位系统下 JVM 对内存的限制：不能突破 2GB ，那么这时你的 Tomcat 要优化，就要讲究点技巧了，而在 64 位操作系统上无论是系统内存还是 JVM 都没有受到 2GB 这样的限制。

针对于 JMX 远程监控也是在这里设置，以下为 64 位系统环境下的配置，内存加入的参数如下：

CATALINA\_OPTS="

-server

-Xms6000M

-Xmx6000M

-Xss512k

-XX:NewSize=2250M

-XX:MaxNewSize=2250M

-XX:PermSize=128M

-XX:MaxPermSize=256M

-XX:+AggressiveOpts

-XX:+UseBiasedLocking

-XX:+DisableExplicitGC

-XX:+UseParNewGC

-XX:+UseConcMarkSweepGC

-XX:MaxTenuringThreshold=31

-XX:+CMSParallelRemarkEnabled

-XX:+UseCMSCompactAtFullCollection

-XX:LargePageSizeInBytes=128m

-XX:+UseFastAccessorMethods

-XX:+UseCMSInitiatingOccupancyOnly

-Duser.timezone=Asia/Shanghai

-Djava.awt.headless=true"

为了看着方便，将每个参数单独写一行。上面参数好多啊，可能有人写到现在都没见过一个在 Tomcat 的启动命令里加了这么多参数，当然，这些参数只是我机器上的，不一定适合你，尤其是参数后的 value（值）是需要根据你自己的实际情况来设置的。

上述这样的配置，基本上可以达到：

系统响应时间增快；

JVM回收速度增快同时又不影响系统的响应率；

JVM内存最大化利用；

线程阻塞情况最小化。

JVM 常用参数详解：

-server：一定要作为第一个参数，在多个 CPU 时性能佳，还有一种叫 -client 的模式，特点是启动速度比较快，但运行时性能和内存管理效率不高，通常用于客户端应用程序或开发调试，在 32 位环境下直接运行 Java 程序默认启用该模式。Server 模式的特点是启动速度比较慢，但运行时性能和内存管理效率很高，适用于生产环境，在具有 64 位能力的 JDK 环境下默认启用该模式，可以不配置该参数。

-Xms：表示 Java 初始化堆的大小，-Xms 与-Xmx 设成一样的值，避免 JVM 反复重新申请内存，导致性能大起大落，默认值为物理内存的 1/64，默认（MinHeapFreeRatio参数可以调整）空余堆内存小于 40% 时，JVM 就会增大堆直到 -Xmx 的最大限制。

-Xmx：表示最大 Java 堆大小，当应用程序需要的内存超出堆的最大值时虚拟机就会提示内存溢出，并且导致应用服务崩溃，因此一般建议堆的最大值设置为可用内存的最大值的80%。如何知道我的 JVM 能够使用最大值，使用 java -Xmx512M -version 命令来进行测试，然后逐渐的增大 512 的值,如果执行正常就表示指定的内存大小可用，否则会打印错误信息，默认值为物理内存的 1/4，默认（MinHeapFreeRatio参数可以调整）空余堆内存大于 70% 时，JVM 会减少堆直到-Xms 的最小限制。

-Xss：表示每个 Java 线程堆栈大小，JDK 5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。根据应用的线程所需内存大小进行调整，在相同物理内存下，减小这个值能生成更多的线程，但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。一般小的应用， 如果栈不是很深， 应该是128k 够用的，大的应用建议使用 256k 或 512K，一般不易设置超过 1M，要不然容易出现out ofmemory。这个选项对性能影响比较大，需要严格的测试。

-XX:NewSize：设置新生代内存大小。

-XX:MaxNewSize：设置最大新生代新生代内存大小

-XX:PermSize：设置持久代内存大小

-XX:MaxPermSize：设置最大值持久代内存大小，永久代不属于堆内存，堆内存只包含新生代和老年代。

-XX:+AggressiveOpts：作用如其名（aggressive），启用这个参数，则每当 JDK 版本升级时，你的 JVM 都会使用最新加入的优化技术（如果有的话）。

-XX:+UseBiasedLocking：启用一个优化了的线程锁，我们知道在我们的appserver，每个http请求就是一个线程，有的请求短有的请求长，就会有请求排队的现象，甚至还会出现线程阻塞，这个优化了的线程锁使得你的appserver内对线程处理自动进行最优调配。

-XX:+DisableExplicitGC：在 程序代码中不允许有显示的调用“System.gc()”。每次在到操作结束时手动调用 System.gc() 一下，付出的代价就是系统响应时间严重降低，就和关于 Xms，Xmx 里的解释的原理一样，这样去调用 GC 导致系统的 JVM 大起大落。

-XX:+UseConcMarkSweepGC：设置年老代为并发收集，即 CMS gc，这一特性只有 jdk1.5

后续版本才具有的功能，它使用的是 gc 估算触发和 heap 占用触发。我们知道频频繁的 GC 会造面 JVM

的大起大落从而影响到系统的效率，因此使用了 CMS GC 后可以在 GC 次数增多的情况下，每次 GC 的响应时间却很短，比如说使用了 CMS

GC 后经过 jprofiler 的观察，GC 被触发次数非常多，而每次 GC 耗时仅为几毫秒。

-XX:+UseParNewGC：对新生代采用多线程并行回收，这样收得快，注意最新的 JVM 版本，当使用 -XX:+UseConcMarkSweepGC 时，-XX:UseParNewGC 会自动开启。因此，如果年轻代的并行 GC 不想开启，可以通过设置 -XX：-UseParNewGC 来关掉。

-XX:MaxTenuringThreshold：设置垃圾最大年龄。如果设置为0的话，则新生代对象不经过 Survivor 区，直接进入老年代。对于老年代比较多的应用（需要大量常驻内存的应用），可以提高效率。如果将此值设置为一 个较大值，则新生代对象会在 Survivor 区进行多次复制，这样可以增加对象在新生代的存活时间，增加在新生代即被回收的概率，减少Full GC的频率，这样做可以在某种程度上提高服务稳定性。该参数只有在串行 GC 时才有效，这个值的设置是根据本地的 jprofiler 监控后得到的一个理想的值，不能一概而论原搬照抄。

-XX:+CMSParallelRemarkEnabled：在使用 UseParNewGC 的情况下，尽量减少 mark 的时间。

-XX:+UseCMSCompactAtFullCollection：在使用 concurrent gc 的情况下，防止 memoryfragmention，对 live object 进行整理，使 memory 碎片减少。

-XX:LargePageSizeInBytes：指定 Java heap 的分页页面大小，内存页的大小不可设置过大， 会影响 Perm 的大小。

-XX:+UseFastAccessorMethods：使用 get，set 方法转成本地代码，原始类型的快速优化。

-XX:+UseCMSInitiatingOccupancyOnly：只有在 oldgeneration 在使用了初始化的比例后 concurrent collector 启动收集。

-Duser.timezone=Asia/Shanghai：设置用户所在时区。

-Djava.awt.headless=true：这个参数一般我们都是放在最后使用的，这全参数的作用是这样的，有时我们会在我们的 J2EE 工程中使用一些图表工具如：jfreechart，用于在 web 网页输出 GIF/JPG 等流，在 winodws 环境下，一般我们的 app server 在输出图形时不会碰到什么问题，但是在linux/unix 环境下经常会碰到一个 exception 导致你在 winodws 开发环境下图片显示的好好可是在 linux/unix 下却显示不出来，因此加上这个参数以免避这样的情况出现。

-Xmn：新生代的内存空间大小，注意：此处的大小是（eden+ 2 survivor space)。与 jmap -heap 中显示的 New gen 是不同的。整个堆大小 = 新生代大小 + 老生代大小 + 永久代大小。在保证堆大小不变的情况下，增大新生代后，将会减小老生代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的 3/8。

-XX:CMSInitiatingOccupancyFraction：当堆满之后，并行收集器便开始进行垃圾收集，例如，当没有足够的空间来容纳新分配或提升的对象。对于 CMS 收集器，长时间等待是不可取的，因为在并发垃圾收集期间应用持续在运行（并且分配对象）。因此，为了在应用程序使用完内存之前完成垃圾收集周期，CMS 收集器要比并行收集器更先启动。因为不同的应用会有不同对象分配模式，JVM 会收集实际的对象分配（和释放）的运行时数据，并且分析这些数据，来决定什么时候启动一次 CMS 垃圾收集周期。这个参数设置有很大技巧，基本上满足(Xmx-Xmn)\*(100-CMSInitiatingOccupancyFraction)/100 >= Xmn 就不会出现 promotion failed。例如在应用中 Xmx 是6000，Xmn 是 512，那么 Xmx-Xmn 是 5488M，也就是老年代有 5488M，CMSInitiatingOccupancyFraction=90 说明老年代到 90% 满的时候开始执行对老年代的并发垃圾回收（CMS），这时还 剩 10% 的空间是 5488\*10% = 548M，所以即使 Xmn（也就是新生代共512M）里所有对象都搬到老年代里，548M 的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的 promotion failed，因此这个参数的设置必须与 Xmn 关联在一起。

-XX:+CMSIncrementalMode：该标志将开启 CMS 收集器的增量模式。增量模式经常暂停 CMS 过程，以便对应用程序线程作出完全的让步。因此，收集器将花更长的时间完成整个收集周期。因此，只有通过测试后发现正常 CMS 周期对应用程序线程干扰太大时，才应该使用增量模式。由于现代服务器有足够的处理器来适应并发的垃圾收集，所以这种情况发生得很少，用于但 CPU情况。

-XX:NewRatio：年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代），-XX:NewRatio=4 表示年轻代与年老代所占比值为 1:4，年轻代占整个堆栈的 1/5，Xms=Xmx 并且设置了 Xmn 的情况下，该参数不需要进行设置。

-XX:SurvivorRatio：Eden 区与 Survivor 区的大小比值，设置为 8，表示 2 个 Survivor 区（JVM 堆内存年轻代中默认有 2 个大小相等的 Survivor 区）与 1 个 Eden 区的比值为 2:8，即 1 个 Survivor 区占整个年轻代大小的 1/10。

-XX:+UseSerialGC：设置串行收集器。

-XX:+UseParallelGC：设置为并行收集器。此配置仅对年轻代有效。即年轻代使用并行收集，而年老代仍使用串行收集。

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集，JDK6.0 开始支持对年老代并行收集。

-XX:ConcGCThreads：早期 JVM 版本也叫-XX:ParallelCMSThreads，定义并发 CMS 过程运行时的线程数。比如 value=4 意味着 CMS 周期的所有阶段都以 4 个线程来执行。尽管更多的线程会加快并发 CMS 过程，但其也会带来额外的同步开销。因此，对于特定的应用程序，应该通过测试来判断增加 CMS 线程数是否真的能够带来性能的提升。如果还标志未设置，JVM 会根据并行收集器中的 -XX:ParallelGCThreads 参数的值来计算出默认的并行 CMS 线程数。

-XX:ParallelGCThreads：配置并行收集器的线程数，即：同时有多少个线程一起进行垃圾回收，此值建议配置与 CPU 数目相等。

-XX:OldSize：设置 JVM 启动分配的老年代内存大小，类似于新生代内存的初始大小 -XX:NewSize。

以上就是一些常用的配置参数，有些参数是可以被替代的，配置思路需要考虑的是 Java 提供的垃圾回收机制。虚拟机的堆大小决定了虚拟机花费在收集垃圾上的时间和频度。收集垃圾能够接受的速度和应用有关，应该通过分析实际的垃圾收集的时间和频率来调整。假如堆的大小很大，那么完全垃圾收集就会很慢，但是频度会降低。假如您把堆的大小和内存的需要一致，完全收集就很快，但是会更加频繁。调整堆大小的的目的是最小化垃圾收集的时间，以在特定的时间内最大化处理客户的请求。在基准测试的时候，为确保最好的性能，要把堆的大小设大，确保垃圾收集不在整个基准测试的过程中出现。

假如系统花费很多的时间收集垃圾，请减小堆大小。一次完全的垃圾收集应该不超过 3-5 秒。假如垃圾收集成为瓶颈，那么需要指定代的大小，检查垃圾收集的周详输出，研究垃圾收集参数对性能的影响。当增加处理器时，记得增加内存，因为分配能够并行进行，而垃圾收集不是并行的。

3、设置系统属性

之前说过，Tomcat 的语言编码，配置起来很慢，要经过多次设置才可以了，否则中文很有可能出现乱码情况。譬如汉字“中”，以 UTF-8 编码后得到的是 3 字节的值 %E4%B8%AD，然后通过 GET 或者 POST 方式把这 3 个字节提交到 Tomcat 容器，如果你不告诉 Tomcat 我的参数是用 UTF-8编码的，那么 Tomcat 就认为你是用 ISO-8859-1 来编码的，而 ISO8859-1（兼容 URI 中的标准字符集 US-ASCII）是兼容 ASCII 的单字节编码并且使用了单字节内的所有空间，因此 Tomcat 就以为你传递的用 ISO-8859-1 字符集编码过的 3 个字符，然后它就用 ISO-8859-1 来解码。

设置起来不难使用“ -D<名称>=<值> ”来设置系统属性：

-Djavax.servlet.request.encoding=UTF-8

-Djavax.servlet.response.encoding=UTF-8

-Dfile.encoding=UTF-8

-Duser.country=CN

-Duser.language=zh

4、常见的 Java 内存溢出有以下三种

（1） java.lang.OutOfMemoryError: Java heap space —-JVM Heap（堆）溢出

JVM 在启动的时候会自动设置 JVM Heap 的值，其初始空间（即-Xms）是物理内存的1/64，最大空间（-Xmx）不可超过物理内存。可以利用 JVM提供的 -Xmn -Xms -Xmx 等选项可进行设置。Heap 的大小是 Young Generation 和 Tenured Generaion 之和。在 JVM 中如果 98％ 的时间是用于 GC，且可用的 Heap size 不足 2％ 的时候将抛出此异常信息。

解决方法：手动设置 JVM Heap（堆）的大小。

（2） java.lang.OutOfMemoryError: PermGen space —- PermGen space溢出。

PermGen space 的全称是 Permanent Generation space，是指内存的永久保存区域。为什么会内存溢出，这是由于这块内存主要是被 JVM 存放Class 和 Meta 信息的，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，sun 的 GC 不会在主程序运行期对 PermGen space 进行清理，所以如果你的 APP 会载入很多 CLASS 的话，就很可能出现 PermGen space 溢出。

解决方法： 手动设置 MaxPermSize 大小

（3） java.lang.StackOverflowError —- 栈溢出

栈溢出了，JVM 依然是采用栈式的虚拟机，这个和 C 与 Pascal 都是一样的。函数的调用过程都体现在堆栈和退栈上了。调用构造函数的 “层”太多了，以致于把栈区溢出了。通常来讲，一般栈区远远小于堆区的，因为函数调用过程往往不会多于上千层，而即便每个函数调用需要 1K 的空间（这个大约相当于在一个 C 函数内声明了 256 个 int 类型的变量），那么栈区也不过是需要 1MB 的空间。通常栈的大小是 1－2MB 的。

通常递归也不要递归的层次过多，很容易溢出。

### 39 分布式系统含义是什么？要解决什么问题？

一、三个步骤完成华丽转身——任意软件变为“分布式”

分布式——一个高大上的名词，是计算机软件设计中人民群众喜闻乐见的“逼格满满”、“不明觉厉”的几个名词之一。但很可惜，这玩意儿一点也不复杂，甚至有些“简单”。不信？你只要遵循下述步骤即可将任何一个软件拆分为“分布式”的：

将你的整个软件视为一个系统（不管它有多复杂）

将整个系统分割为一系列的 Process（进程）， 每个 Process 完成一定的功能

将这些 Process 分散到不同的机器上。分散后，选择若干种（没错一种可能不够）通信协议把他们连接起来

蹬蹬蹬蹬～大功告成。哈哈别打我，这真的是很严肃的通用型三步骤大杀器，对付任何软件，是任何软件都可以的。接下来我当然要解释清楚为什么。

二、跳出误区——分布式不等于并行计算

人们常常把分布式系统自然而然的和并行计算联系起来。然而这并不正确。实际上，分布式系统并不一定是并行的，举个简单的例子就能理解——

某软件，功能如下：

提示用户输入两个数 A 和 B

在内部，对 A 和 B 执行某数学运算，获得 C

输出 C

很简单吧？这三个步骤是无法并行的。它们需要被依次执行。但是这个软件依然可以被改为分布式的，方法就是用前面提到的方法，把步骤 2 的计算过程独立为一个 Process 移动到另外一台计算机上完成。

如果我们从整个系统流程的观点来看，并没有什么并行。整个过程都是顺序执行的。只不过执行时出现了“跨设备”的现象而已。可见，分布式本身就只如其字面意思所指，指的仅仅是从结构角度的分散而已。

当然啊，现实世界中，我们更多的时候钟情于分布式，还是因为它与并行之间可以相互配合。例如实现既是分布同时也是并行的系统。

好了，理解这一点之后就不难解释为什么我会说前文提到的三步骤是万用大法了。接下来我们继续讨论分布式本身。

三、拆分+连接是分布式系统的本质

所谓分布式，无非就是”将一个系统拆分成多个子系统并散布到不同设备“的过程而已。

本质上而言，实现一个分布式系统，最核心的部分无非有两点：

如何拆分——可以有很多方式，核心依据一是业务需求，二是成本限制。这是实践中构建分布式系统时最主要的设计依据。

如何连接——光把系统拆开成 Process 还不够，关键是拆开后的 Process 之间还要能通信，因此涉及通信协议设计的问题，需要考虑的因素很多，好消息是这部分其实成熟方案很多

四、为什么你要使用分布式？

分布式系统并非灵丹妙药，解决问题的关键还是看你对问题本身的了解。通常我们需要使用分布式的常见理由是：

为了性能扩展——系统负载高，单台机器无法承载，希望通过使用多台机器来提高系统的负载能力

为了增强可靠性——软件不是完美的，网络不是完美的，甚至机器本身也不可能是完美的，随时可能会出错，为了避免故障，需要将业务分散开保留一定的冗余度

在以提供 Service 为主的服务端软件开发过程中常常遇到这些问题。

五、一些分布式方案能解决你的问题，另一些却不能，要学会的其实是选择

笼统的讨论分布式没有太大的意义，就如我刚才所谈的，实际上分布式很容易实现。真正难的地方在于如何选择正确的分布方案。

例如，当你想要建立一个分布式的数据管理系统的时候，你就必须得面对“一致性”问题。如果你对数据一致性要求很高，你就不得不容忍一些缺陷例如规模伸缩困难；而如果你放弃它，你可以轻松伸缩规模，但你必须解决好由此带来的一系列数据不一致导致的问题。（CAP 问题）

于是你会意识到，有许多种分布方案，为了正确解决你的问题，你需要对每一个方案都进行了解，并评估，选择不同的方案有时候区别不大，有时候却会深刻的影响整个系统中其他部分的工作方式，甚至影响用户界面中用户操作时的流程。这是我们学习分布式系统的重点所在。

六、分布式学习入门——基础知识要点

如我前面所讲，分布式入门不难。主要包含如下知识点：

Process（进程）。在分布式系统中，进程是基本单元

通信协议。Process 间需要相互配合才能完成工作，因此通信协议是最基本要解决的问题。这部分其实挺复杂，牵涉面光，不过核心还是抓住两方面，一是存在哪些需求，二是各个协议如何满足这些需求

命名法。两个 Process 要通信，必须相互知道对方的名字，名字可以是数字，也可以是结构化的字符串。例如众所周知域名系统就是一种命名方案，但是方案还有很多，各有特点

协作。上面都在谈 Process 之间的通信，可是为什么要通信？因为要协作。协作是个复杂的主题，其中最基本最基本的一个问题就是同步问题。而聊同步问题必然要聊“锁”……知识点就这么展开了

上面几点是最基础的知识。了解了这些其实就算入门了。可是如何进阶呢？那么必然要开始学习下面的问题：

一致性。数据存储时，最基本的问题。其实也是实际设计系统时常常需要反复考虑的问题

容错。冗余是容错的基础，但并不是全部，分布式本身为实现容错提供了一些便利，这也是实际设计系统时常常需要考虑的问题

好了，如果这些你都学的差不多了，那咱们“纸上谈兵”也就告一段落了。接下来进入实战演练。

七、实战演练？其实你已经开发过分布式系统了

你有没有开发过简单的增删改查软件？这类软件通常都需要搭配一个独立的数据库管理系统共同完成功能。实际上，只要你开发过这么简单的软件，那么你就已经开发过分布式系统了。

“什么，基于数据库管理系统开发出来的软件就可以算分布式呀？我做了很多这类软件，怎么我从来没听过这种说法？

真的，我没开玩笑。还记得我们前面提到的吗，什么是分布式？不就是一个大系统拆分成多个小系统分散到不同的设备上吗。回想一下，当你写一个简单的增删改查软件时，只要用到数据库管理系统，是不是具有如下特点：

整个系统中，你写的代码跑在 A 进程里，而数据库管理系统则跑在另外一个进程 B 里

A 进程与 B 进程通过某种通信协议连接

既可以使 A 进程与 B 进程运行在同一台机器上，也可以将它们分开运行于不同的机器上，并且系统依然可以照常运行

### 40如何理解微服务的熔断和隔离，限流？

一、为什么需要做服务隔离与熔断

微服务是当前业界的一个趋势，其原理是将职责单一的功能独立化成子服务，一个后台服务依赖多个微服务。假设某服务由30个微服务组成，每个微服务的可用性是99.99%，那么99.99%的30次方≈99.7%，也就是说有0.3%的请求会失败，若有一亿次请求则有300000次失败。

微服务间通过RPC来进行数据交换，所以我们可以做一个假设：在IO型服务中，假设服务A依赖服务B和服务C，而B服务和C服务有可能继续依赖其他的服务，继续下去会使得调用链路过长，技术上称1->N扇出。如果在A的链路上某个或几个被调用的子服务不可用或延迟较高，则会导致调用A服务的请求被堵住，堵住的请求会消耗占用掉系统的线程、io等资源，当该类请求越来越多，占用的计算机资源越来越多的时候，会导致系统瓶颈出现，造成其他的请求同样不可用，最终导致业务系统崩溃，又称：雪崩效应。

二、解决或缓解服务雪崩的方案

一般情况对于服务依赖的保护主要有3中解决方案：

（1）熔断模式：这种模式主要是参考电路熔断，如果一条线路电压过高，保险丝会熔断，防止火灾。放到我们的系统中，如果某个目标服务调用慢或者有大量超时，此时，熔断该服务的调用，对于后续调用请求，不在继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。

（2）隔离模式：这种模式就像对系统请求按类型划分成一个个小岛的一样，当某个小岛被火少光了，不会影响到其他的小岛。例如可以对不同类型的请求使用线程池来资源隔离，每种类型的请求互不影响，如果一种类型的请求线程资源耗尽，则对后续的该类型请求直接返回，不再调用后续资源。这种模式使用场景非常多，例如将一个服务拆开，对于重要的服务使用单独服务器来部署，再或者公司最近推广的多中心。

（3）限流模式：上述的熔断模式和隔离模式都属于出错后的容错处理机制，而限流模式则可以称为预防模式。限流模式主要是提前对各个类型的请求设置最高的QPS阈值，若高于设置的阈值则对该请求直接返回，不再调用后续资源。这种模式不能解决服务依赖的问题，只能解决系统整体资源分配问题，因为没有被限流的请求依然有可能造成雪崩效应。

三、熔断设计

熔断的设计主要参考了hystrix的做法。其中最重要的是三个模块：熔断请求判断算法、熔断恢复机制、熔断报警

（1）熔断请求判断机制算法：使用无锁循环队列计数，每个熔断器默认维护10个bucket，每1秒一个bucket，每个blucket记录请求的成功、失败、超时、拒绝的状态，默认错误超过50%且10秒内超过20个请求进行中断拦截。

（2）熔断恢复：对于被熔断的请求，每隔5s允许部分请求通过，若请求都是健康的（RT<250ms）则对请求健康恢复。

（3）熔断报警：对于熔断的请求打日志，异常请求超过某些设定则报警

四、超时机制设计

超时分两种，一种是请求的等待超时，一种是请求运行超时。

等待超时：在任务入队列时设置任务入队列时间，并判断队头的任务入队列时间是否大于超时时间，超过则丢弃任务。

运行超时：直接可使用线程池提供的get方法。

### 41 简单的谈一下SpringMVC的工作流程？

在整个Spring MVC框架中，DispatcherServlet处于核心位置，它负责协调和组织不同组件完成请求处理并返回响应的工作。具体流程为：

1）客户端发送http请求，web应用服务器接收到这个请求，如果匹配DispatcherServlet的映射路径（在web.xml中配置），web容器将请求转交给DispatcherServlet处理；

2）DispatcherServlet根据请求的信息及HandlerMapping的配置找到处理该请求的Controller；

3）Controller完成业务逻辑处理后，返回一个ModelAndView给DispatcherServlet；

4）DispatcherServlet借由ViewResolver完成ModelAndView中逻辑视图名到真实视图对象View的解析工作；

5）DispatcherServlet根据ModelAndView中的数据模型对View对象进行视图渲染，最终客户端得到的响应消息可能是一个普通的html页面，也可能是一个xml或json串，甚至是一张图片或一个PDF文档等不同的媒体形式。

### 42 SpringMVC与Struts2的主要区别？

区别1：

Struts2 的核心是基于一个Filter即StrutsPreparedAndExcuteFilter

SpringMvc的核心是基于一个Servlet即DispatcherServlet(前端控制器)

区别2：

Struts2是基于类开发的，传递的参数是通过类的属性传递(属性驱动和模型驱动),所以只能设计成多例prototype

SpringMvc是基于类中的方法开发的，也就是一个url对应一个方法，传递参数是传到方法的形参上面，所以既可以是单例模式也可以是多例模式singiton

区别3：

Struts2采用的是值栈存储请求以及响应数据，OGNL存取数据

SpringMvc采用request来解析请求内容，然后由其内部的getParameter给方法中形参赋值，再把后台处理过的数据通过ModelAndView对象存储，Model存储数据，View存储返回的页面，再把对象通过request传输到页面去。

### 43 Spring 4.0新特性有哪些？

2004年Spring框架首次发布，然后陆续发布了一些重要的版本：Spring2.0提供XML命名空间和AspectJ的支持；Spring2.5包含了注释驱动配置；Spring3.0在框架基础代码中引入了强大的Java5+，并且提供诸如基于Java的@Configuration模式。

4.0版本是Spring框架最新发布的主版本，并且首次完全支持Java8的功能。你依然可以使用较早的Java版本，但是现在所需的最小的版本已经被提升的Java SE6。同时还利用主版本发布的机会删除了很多废弃的类和方法。

在Spring框架的GitHub Wiki上提供了升级到Spring4.0的指南。

1. 入门体验的改善

新的spring.io网站提供完整的入门指南系列来帮助初学者。在本文档的第一章Spring入门中，你可以读到更多的有关这个指南的信息。这个网站还提供很多在Spring体系下发布的其他一些工程的详细的概要。

如果你是一个Maven用户，那么你可能还要关注与每个Spring框架版本一起发布的资料清单（POM）。

2. 被删除的废弃的包和方法

所有被废弃的包和很多被废弃的类和方法已经从4.0版中删除，如果从Spring之前的发布版本中升级，要确保修正那些对被废弃的内容的调用，以免使用过期的API。

对于全部改变内容，可查阅API差异报告。

注意，那些可选的第三方依赖已经被升级到2010/2011以后的版本（也就是说Spring 4只支持2010年以后释放的版本）：尤其是Hibernate3.6+、EhCache2.1+、Quartz1.8+、Groovy1.8+、和Joda-Time2.0+。一个例外的规则是：目前Spring4要求最近的Hibernate验证器4.3+，并且支持的Jackson已经聚焦到2.0+（在Spring3.2中保留的对Jackson1.8/1.9的支持，现在已经过时了）。

3. Java 8（包括6和7）

Spring框架4.0提供了对几个Java 8功能的支持。你可以通过Spring的回调接口来使用lambda表达式和方法引用。 首先支持的类是java.time(JSR-310)和几个已经被改造成@Repeatable的既存标注。你还可以使用Java8 的参数名称发现机制（-parameters编译标记）来对调试信息的使用进行选择性编译。

Spring保留了跟Java和JDK较早版本的兼容性：具体的是Java SE6（JDK 6最小的版本级别要升级到18，这是2010年2月释放的版本）及以上的版本依然是完全支持的。但是基于Spring4最新开发的项目，我们推荐使用Java7或8。

4. Java EE 6和7

Spring框架4使用Java EE 6或以上的版本来作为基线，同时包含了相关的JPA2.0和Servlet3.0的规范。为了保留跟Google App引擎和旧的应用服务的兼容性，可能要把Spring 4应用程序部署到Servlet 2.5的环境中。但是在安装了Spring的测试和模拟包的开发环境中，强烈推荐使用Servlet3.0+。

提示：如果你是WebSphere 7的用户，要确保安装JPA2.0的补丁包。在WebLogic 10.3.3或更高的版本中带有JAP2.0的补丁包。这样就会把与Spring 4兼容的部署环境带到这两类服务器中。

前面介绍了当前Spring 框架 4.0支持的Java EE 7的应用程序规范级别，包括：JMS 2.0、JTA 1.2、JPA2.1、Bean Validation 1.1和JSR-236 Concurrency Utilities。通常这种支持只是关注这些规范的独立使用，如在Tomcat或独立的环境中。同样当把Spring应用程序部署到Java EE 7服务上时它的工作方式也是一样的。

注意，Hibernate 4.3是JPA2.1的提供器，因此只在Spring 框架4.0中支持。同样Hibernate Validator 5.0作为Bean Validation提供器也适用于这个约束。这两者都没有提供官方的对Spring框架3.2的支持。

5. 使用Groovy的DSL（DomainSpecific Languages）来定义 Bean

从Spring框架4.0开始，可以使用Groovy的 DSL来定义外部的bean配置。这有点类似使用XML Bean定义的概念，但是它允许使用更加简洁的语法。使用Groovy也允许更加容易的把bean定义嵌入到你的应用程序的启动代码中，如：

6. 内核容器方面的改善

以下是对核心容器的几个方面的常规改善：

A. 现在Spring可以在注入Bean的时候处理修饰样式的的泛型。例如，如果要使用Spring的数据资源库（Repository），就可以很容易的注入一个特定的实现：@AutowiredRepository<Customer> customerRepository。

B. 如果使用Spring的元注解支持，那么现在可以开发暴露来自源注解中的特定属性的个性化注解。

C. 现在列表和数组中的bean是可以被排序的，它同时支持@Order注解和ordered接口。

D. 现在在注入点可以使用@Lazy注解，跟@Bean定义一样。

E. 引入了@Description注解，方便开发者使用基于Java的配置。

F. 通过@Conditional注解来作为条件过滤bean的常用模式。这有点类似@Profile支持，但它允许用户给开发的程序定义一些策略。

G. 基于CGLIB的代理类不再需要默认的构造器，它通过被重新包装在内部的objenesis类库来提供支持，这个类库是作为Spring框架的一部分来发布。使用这种策略，不再有用于代理示例调用的构造器了。

H. 通过框架提供管理时区的支持，如LocaleContext。

7. 常用的Web方面的改善

部署到Servlet 2.5依然是一个可选项，但当前的Spring 框架4.0主要关注Servlet3.0+环境。如果你是在使用Spring的MVC测试框架，那么就需要确保在测试的类路径中有与Servlet3.0兼容的JAR包。

另外，稍后会介绍WebSocket，下面是Spring的Web模块常规改善：

A. 你可以在Spring的MVC应用程序中使用新的@RestController注解，不再需要给每个@RequestMapping方法添加@ResponseBody。

B. 添加了AsyncRestTemplate类，它允许在开发REST客户端时支持非阻塞的异步支持。

C. 在开发Spring MVC应用程序时，Spring提供了全面的时区支持。

8. WebSocket、SockJS、和STOMP消息

新的spring-websocket模块提供了全面的基于WebSocket的支持，在Web应用程序的客户端和服务端之间有两种通信方式。它跟JSR-356兼容，用于浏览器的Java的WebSocket API和额外提供的基于SockJS的回退选项（如WebSocket模拟器）依然不支持WebSocket协议（如IE以前的版本）。

新的spring-messaging模块添加了对WebSocket的子协议STOMP的支持，它在应用程序中跟注解编程模式一起用于路由和处理来自WebSocket客户端的STOMP消息。现在一个@Controller就能够包含处理HTTP请求和来自被连接的WebSocket客户端的@RequestMapping和@MessageMapping方法的结果。这个模块还包含了来自Spring集成项目的关键抽象原型，如Message、MessageChannel、MessageHandler以及其他的基于消息的应用的基础服务。

更多的信息，请看第25章---WebSocket支持

9. 测试的改善

Spring框架4.0中删除了spring-test模块中的废弃代码，还引入了几个用于单元和集成测试的新功能：

A. 在spring-test模块中几乎所有的注解（如@ContextConfiguration、@WebAppConfiguration、@ContextHierarchy、@ActiveProfiles等）都可以使用元注解来创建个性化的组合注解并减少跨测试单元的配置成本。

B. 通过简单的编程实现个性化的ActiveProfilesResolver接口，并使用@ActiveProfiles的resolver属性就可以激活bean定义的配置。

C. 在spring-core模块中引入了新的SocketUtils类，它确保你可以扫描到本地主机上的闲置的TCP和UDP服务端口。这个功能不是专门提供给测试的，但是在编写需要使用套接字的集成测试代码是就非常有用，例如，测试内存中启动的SMTP服务、FTP服务、Servlet容器等。

D. 在Spring4.0的org.springframework.mock.web包中，有一组基于Servlet3.0 API的模仿器。此外，还增强了几个Servlet的API模仿器（如MockHttpServletRequest、MockServletContext等）的功能，并改善了可配置性。

### 44 weblogic 负载均衡的原理和集群的配置？

一、 Cluster的概念及优势

Weblogic支持集群技术，即让一组Server指向同一域名一起工作从而提供一个更强大、更可靠的应用平台。对于客户端而言，无论Cluster中有几个Server在工作，看上去都是一个。集群技术有两个最明显的特色：

(1)可伸缩性：Cluster对加入其中的Server在性能上没有限制，为了提高性能，当客户端的请求大幅增加时，可以动态地向Cluster中添加Server。并且，配置Cluster当一台机器的资源没有被完全利用时，可以在同一机器上启动多个Server，但要求每一个Server使用不同的IP，而不能用同一IP的不同端口。

(2)高可用性：由于在Cluster中同一service在多个Server上同时存放或放在一个共享文件系统中，因此相同的请求可以有多个Server提供，并且Server间还可以复制状态信息。这样，当其中某一Server宕机或无法响应请求时，其它的Server会立即接管它的任务，从而把应用和客户端完全隔离开来。

二、Cluster的工作机制

每一个Clustered service,在每一个server上都会有一个instance，即一个replica，这些replicas集合在一起形成一个replica-aware stub。这些stubs负责客户端与相关的服务器段对象的通信，当客户端请求该service时，实际上是向stub发出请求，stub根据不同的算法调用集合中某一replica，如果调用失败，stub会检测到错误并重新调用其它的replica。Cluster支持多种算法：随机、轮循、基于性能的负载均衡的轮循(Weight-based round-robin)、根据参数值调用(Parameter-based routing)。

Weblogic Cluster通过负载均衡和容错最大程度的实现了它的可伸缩性和可用性。

为了提高Cluster的可伸缩性，必须保证充分利用每一个Server。Weblogic可以在不同平台、不同性能的机器上安装Server并进行Cluster,然后采用Weight-based round-robin算法达到负载均衡，从而使每一个Server都得到充分的利用。

为了使Cluster具有高可用性，必须具备故障恢复的能力，这一点可以通过replica-aware stub的容错功能来实现。Stub 主要是通过在检测到错误信息时重新进行调用的方式实现容错。当重新调用不会导致错误的结果时(如stub确认failed server不能接收到请求)，容错功能自动实现。而有些情况下，重新调用可能会导致某一service被请求了多次的错误结果。例如：客户端C请求Clustered购物车服务中的additem()方法，replica-aware stub接收到请求，根据算法调用Server1上的service,Server1响应请求并返回结果，但在结果成功到达客户端之前，Server1出现错误。此时stub接收到错误信息，因此重新调用Server2上的这一方法，但实际上Server1已经将item加入购物车，这样就造成重复。为了解决这种问题，可以为服务添加一个唯一标识，如上述的additem()方法中可添加一个参数——序列号。每一个item有一个唯一的sequence,相同sequence的item不能被重复添加。

三、 Cluster的命名服务

在Weblogic Server中使用命名服务时，客户端通过JNDI存取service，JNDI tree上绑定了Server提供的所有的公共服务。Server提供一个新的service时，是将service以某一名称绑定到JNDI tree上，客户端和Server建立连接并按照名称获取相应的stub。

Custer扩展了Server的这种命名服务机制，它不仅包含了每一个Server上的非Clustered的stub,而且包含了多个Server间的Clustered 的replica-aware stub。

四、 Cluster的服务类型

在Weblogic中，有多种服务可以进行cluster，如：RMI对象、EJB、Servlets、Jsp、Web Application。

（1）RMI和EJB Clustering

RMI和EJB对象在Cluster过程中使用JNDI命名服务机制。RMI和EJB对象发送remote stubs到客户端，客户端获取的这些stubs可以是已经clustered的，也可以是没有clustered的。对于Clustered的服务，Stubs根据负载均衡和容错的不同需求调用Cluster中合适的Server；而对没有Clustered的服务，所有对此stub的调用只能由提供此服务的Server来处理。

有些有状态的RMI和EJB对象是不可以进行clustered的，因为客户端必须总是和同一个Server上的对象实例相联系。所有的EJB都是clusterable，虽然EJB也有有状态的，但是EJB home interface 都是无状态的，可以进行clustered，这样就可以从JNDI tree上获取 Clusterable EJB 的home stub 对象。然后通过home stub的方法创建或检索相应的EJB bean,若为stateful session bean 或entity bean，那么此时得到的stub就是不可clusterable。为了使有状态的对象可以更好的cluster，可以将一些操作作为一个事务来执行，如果工作中的Server出现意外，可以重新获取此对象并进行事务操作。

RMI和EJB不同，RMI没有定义有状态和无状态分类，因此必须特意绑定一个有状态的RMI对象到Server上。可以仿效EJB home interface的方式即客户端从JNDI tree上获取一个clusterable factory method,然后factory method 可以调用集群中的任意一台Server,但是被调用的Server上必须有由此factory调用的对象。

（2）Clustered Servlets

Servlets也是可以进行Cluster的。对于Servlets，它用replica-aware proxy替代了replica-aware。这个proxy接受web server上所有请求，并转给集群中的某一Server。Proxy对cluster的所有请求进行负载均衡，并且当请求失败时会进行恢复处理。Proxy还可以在cluster中特别是Server没有正常完成请求响应时保持session状态。当session初始化时，proxy按照负载均衡算法选择一台Server保存session，此后，所有与此session相关的请求都由这同一台Server处理。为了避免当此Server出错时，无法保存客户端状态信息，所以session会被复制下来，并且session的所有变化都会在备份中进行及时更新，这样，当原有Server在响应请求过程中失败时，proxy会立即获取session的备份，并由此继续响应客户端请求，同时做新的复制。

(3)JDBC clustering

为了利用Weblogic Server cluster的负载均衡和容错的性能，Weblogic JDBC连接池也可以在replicated naming tree上注册。通常情况下，cluster中的每一个Server都进行连接池属性配置来访问同一个后端的DBMS实例，即对相同数据库的访问，每一个Server都有一个连接池。然后通过在配置文件中定义一个DataSource属性来在naming tree 上注册连接池。客户端使用Weblogic JDBC/RMI JDBC 驱动程序从cluster中获取数据库连接，即客户端按照DataSource name获取连接池，然后按照负载均衡的算法选择相应的Weblogic Server来响应请求。

五、集群配置

Weblogic域是一个服务器/集群的管理组，用户可以通过某个中央位置来管理、监控整个服务器域，这个中心就是管理服务器，相对的，受控服务器或者叫被管服务器是指域中除了管理服务器以外的所有的服务器。

Weblogic集群是一个weblogic Server组，在一个集群中的服务器可以获得集群范围的命名服务、负载平衡以及容错等功能。

所以，一个weblogic域可以同时管理多个集群以及多个服务器实例。

另外，还有一个节点管理器的概念，节点管理器是指在物理的机器上运行的一个守护进程，通过它，管理服务器可以获得诸如在控制台中强制重启远程机器上的server等功能（没有节点管理器的情况下也是可以强制关闭远程服务器的，但是不能重启）。

除此之外，据说节点管理器还可以自动检查自己机器上的server实例（健康检查），当发现某个实例出现故障的时候，可以自动进行重启，这个没试过。

注意事项

1. 本文的weblogic均配置在linux环境下

2. 集群中管理服务器和被管服务器之间有用到ip组播UDP，所以这些服务器之间必须是UDP可以到达的。

3. 好像动态ip的机器不能用于集群当中

4. 最佳实践说：请不要把管理服务器设置在集群当中

5. 本文所配置的集群只是实现了web层次上的负载平衡和容错，也就是servlet和jsp，其他的j2ee组件如ejb，jdbc，jms等都可以集群化。

### 45 Nginx+Tomcat+Redis实现负载均衡、资源分离、session共享？

一、准备工作

CentOS安装Nginx

http://centoscn.com/CentosServer/www/2013/0910/1593.html

CentOS安装Tomcat

http://blog.csdn.net/zhuying\_linux/article/details/6583096

CentOS安装Redis

http://www.cnblogs.com/zhuhongbao/archive/2013/06/04/3117997.html

多个Tomcat负载均衡实例：可在服务器上复制出多个Tomcat分别修改Tomcat的

http访问端口（默认为8080端口）

Shutdown端口（默认为8005端口）

JVM启动端口（默认为8009端口）

二、Nginx实现多Tomcat负载均衡

Tomcat服务

192.168.1.177:8001

192.168.1.177:8002

192.168.1.177:8003

Nginx配置

upstream mytomcats {

server 192.168.1.177:8001;

server 192.168.1.177:8002;

server 192.168.1.177:8003;

}

server {

listen 80;

server\_name www.iu14.com;

location ~\* \.(jpg|gif|png|swf|flv|wma|wmv|asf|mp3|mmf|zip|rar)$ {

root /web/www/html/;

}

location / {

proxy\_pass http://mytomcats;

proxy\_redirect off;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

client\_max\_body\_size 10m;

client\_body\_buffer\_size 128k;

proxy\_connect\_timeout 90;

proxy\_send\_timeout 90;

proxy\_read\_timeout 90;

proxy\_buffer\_size 4k;

proxy\_buffers 4 32k;

proxy\_busy\_buffers\_size 64k;

proxy\_temp\_file\_write\_size 64k;

}

}

upstream指定负载均衡组，指定其Tomcat成员

location ~\* \.(jpg|gif|……实现了静态资源分离。ps：在location指令使用正则表达式后再用alias指令，Nginx是不支持的。

三、Nginx实现静态资源分离

Tomcat服务

192.168.1.177:8000

Nginx配置

server {

listen 80;

server\_name www.iu14.com;

root /web/www/html;

location /img/ {

alias /web/www/html/img/;

}

location ~ (\.jsp)|(\.do)$ {

proxy\_pass http://192.168.1.177:8000;

proxy\_redirect off;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

client\_max\_body\_size 10m;

client\_body\_buffer\_size 128k;

proxy\_connect\_timeout 90;

proxy\_send\_timeout 90;

proxy\_read\_timeout 90;

proxy\_buffer\_size 4k;

proxy\_buffers 4 32k;

proxy\_busy\_buffers\_size 64k;

proxy\_temp\_file\_write\_size 64k;

}

}

第一个location指令将/web/www/html/img/目录下的静态文件交给Nginx来完成。最后一个location指令将所有以.jsp、.do结尾的文件都交给Tomcat服务器的8080端口来处理。

四、Nginx+Tomcat+Redis实现session共享

Redis服务

192.168.1.178:6379

Tomcat服务

192.168.1.177:8001

192.168.1.177:8002

192.168.1.177:8003

Nginx服务

192.168.1.179

配置Tomcat让其session保存到redis上，在context.xml配置(Value标签一定要在Manager标签前面)：

配置Nginx

upstream mytomcats {

server 192.168.1.177:8001;

server 192.168.1.177:8002;

server 192.168.1.177:8003;

}

log\_format www\_iu14\_com '$remote\_addr - $remote\_user [$time\_local] $request ' '"$status" $body\_bytes\_sent "$http\_referer"' '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

server {

listen 80;

server\_name www.iu14.com;

location / {

proxy\_pass http:// mytomcats;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

}

access\_log /usr/tmp/logs/redis.iu14.log www\_iu14\_com;

}

依次启动Redis、Tomcat、Nginx，访问Nginx

### 46 nginx配置文件详解——nginx.conf？

######Nginx配置文件nginx.conf中文详解#####

#定义Nginx运行的用户和用户组

user www www;

#nginx进程数，建议设置为等于CPU总核心数。

worker\_processes 8;

#全局错误日志定义类型，[ debug | info | notice | warn | error | crit ]

error\_log /usr/local/nginx/logs/error.log info;

#进程pid文件

pid /usr/local/nginx/logs/nginx.pid;

#指定进程可以打开的最大描述符：数目

#工作模式与连接数上限

#这个指令是指当一个nginx进程打开的最多文件描述符数目，理论值应该是最多打开文件数（ulimit -n）与nginx进程数相除，但是nginx分配请求并不是那么均匀，所以最好与ulimit -n 的值保持一致。

#现在在linux 2.6内核下开启文件打开数为65535，worker\_rlimit\_nofile就相应应该填写65535。

#这是因为nginx调度时分配请求到进程并不是那么的均衡，所以假如填写10240，总并发量达到3-4万时就有进程可能超过10240了，这时会返回502错误。

worker\_rlimit\_nofile 65535;

events

{

#参考事件模型，use [ kqueue | rtsig | epoll | /dev/poll | select | poll ]; epoll模型

#是Linux 2.6以上版本内核中的高性能网络I/O模型，linux建议epoll，如果跑在FreeBSD上面，就用kqueue模型。

#补充说明：

#与apache相类，nginx针对不同的操作系统，有不同的事件模型

#A）标准事件模型

#Select、poll属于标准事件模型，如果当前系统不存在更有效的方法，nginx会选择select或poll

#B）高效事件模型

#Kqueue：使用于FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0 和 MacOS X.使用双处理器的MacOS X系统使用kqueue可能会造成内核崩溃。

#Epoll：使用于Linux内核2.6版本及以后的系统。

#/dev/poll：使用于Solaris 7 11/99+，HP/UX 11.22+ (eventport)，IRIX 6.5.15+ 和 Tru64 UNIX 5.1A+。

#Eventport：使用于Solaris 10。 为了防止出现内核崩溃的问题， 有必要安装安全补丁。

use epoll;

#单个进程最大连接数（最大连接数=连接数\*进程数）

#根据硬件调整，和前面工作进程配合起来用，尽量大，但是别把cpu跑到100%就行。每个进程允许的最多连接数，理论上每台nginx服务器的最大连接数为。

worker\_connections 65535;

#keepalive超时时间。

keepalive\_timeout 60;

#客户端请求头部的缓冲区大小。这个可以根据你的系统分页大小来设置，一般一个请求头的大小不会超过1k，不过由于一般系统分页都要大于1k，所以这里设置为分页大小。

#分页大小可以用命令getconf PAGESIZE 取得。

#[root@web001 ~]# getconf PAGESIZE

#4096

#但也有client\_header\_buffer\_size超过4k的情况，但是client\_header\_buffer\_size该值必须设置为“系统分页大小”的整倍数。

client\_header\_buffer\_size 4k;

#这个将为打开文件指定缓存，默认是没有启用的，max指定缓存数量，建议和打开文件数一致，inactive是指经过多长时间文件没被请求后删除缓存。

open\_file\_cache max=65535 inactive=60s;

#这个是指多长时间检查一次缓存的有效信息。

#语法:open\_file\_cache\_valid time 默认值:open\_file\_cache\_valid 60 使用字段:http, server, location 这个指令指定了何时需要检查open\_file\_cache中缓存项目的有效信息.

open\_file\_cache\_valid 80s;

#open\_file\_cache指令中的inactive参数时间内文件的最少使用次数，如果超过这个数字，文件描述符一直是在缓存中打开的，如上例，如果有一个文件在inactive时间内一次没被使用，它将被移除。

#语法:open\_file\_cache\_min\_uses number 默认值:open\_file\_cache\_min\_uses 1 使用字段:http, server, location 这个指令指定了在open\_file\_cache指令无效的参数中一定的时间范围内可以使用的最小文件数,如果使用更大的值,文件描述符在cache中总是打开状态.

open\_file\_cache\_min\_uses 1;

#语法:open\_file\_cache\_errors on | off 默认值:open\_file\_cache\_errors off 使用字段:http, server, location 这个指令指定是否在搜索一个文件是记录cache错误.

open\_file\_cache\_errors on;

}

#设定http服务器，利用它的反向代理功能提供负载均衡支持

http

{

#文件扩展名与文件类型映射表

include mime.types;

#默认文件类型

default\_type application/octet-stream;

#默认编码

#charset utf-8;

#服务器名字的hash表大小

#保存服务器名字的hash表是由指令server\_names\_hash\_max\_size 和server\_names\_hash\_bucket\_size所控制的。参数hash bucket size总是等于hash表的大小，并且是一路处理器缓存大小的倍数。在减少了在内存中的存取次数后，使在处理器中加速查找hash表键值成为可能。如果hash bucket size等于一路处理器缓存的大小，那么在查找键的时候，最坏的情况下在内存中查找的次数为2。第一次是确定存储单元的地址，第二次是在存储单元中查找键 值。因此，如果Nginx给出需要增大hash max size 或 hash bucket size的提示，那么首要的是增大前一个参数的大小.

server\_names\_hash\_bucket\_size 128;

#客户端请求头部的缓冲区大小。这个可以根据你的系统分页大小来设置，一般一个请求的头部大小不会超过1k，不过由于一般系统分页都要大于1k，所以这里设置为分页大小。分页大小可以用命令getconf PAGESIZE取得。

client\_header\_buffer\_size 32k;

#客户请求头缓冲大小。nginx默认会用client\_header\_buffer\_size这个buffer来读取header值，如果header过大，它会使用large\_client\_header\_buffers来读取。

large\_client\_header\_buffers 4 64k;

#设定通过nginx上传文件的大小

client\_max\_body\_size 8m;

#开启高效文件传输模式，sendfile指令指定nginx是否调用sendfile函数来输出文件，对于普通应用设为 on，如果用来进行下载等应用磁盘IO重负载应用，可设置为off，以平衡磁盘与网络I/O处理速度，降低系统的负载。注意：如果图片显示不正常把这个改成off。

#sendfile指令指定 nginx 是否调用sendfile 函数（zero copy 方式）来输出文件，对于普通应用，必须设为on。如果用来进行下载等应用磁盘IO重负载应用，可设置为off，以平衡磁盘与网络IO处理速度，降低系统uptime。

sendfile on;

#开启目录列表访问，合适下载服务器，默认关闭。

autoindex on;

#此选项允许或禁止使用socke的TCP\_CORK的选项，此选项仅在使用sendfile的时候使用

tcp\_nopush on;

tcp\_nodelay on;

#长连接超时时间，单位是秒

keepalive\_timeout 120;

#FastCGI相关参数是为了改善网站的性能：减少资源占用，提高访问速度。下面参数看字面意思都能理解。

fastcgi\_connect\_timeout 300;

fastcgi\_send\_timeout 300;

fastcgi\_read\_timeout 300;

fastcgi\_buffer\_size 64k;

fastcgi\_buffers 4 64k;

fastcgi\_busy\_buffers\_size 128k;

fastcgi\_temp\_file\_write\_size 128k;

#gzip模块设置

gzip on; #开启gzip压缩输出

gzip\_min\_length 1k; #最小压缩文件大小

gzip\_buffers 4 16k; #压缩缓冲区

gzip\_http\_version 1.0; #压缩版本（默认1.1，前端如果是squid2.5请使用1.0）

gzip\_comp\_level 2; #压缩等级

gzip\_types text/plain application/x-javascript text/css application/xml; #压缩类型，默认就已经包含textml，所以下面就不用再写了，写上去也不会有问题，但是会有一个warn。

gzip\_vary on;

#开启限制IP连接数的时候需要使用

#limit\_zone crawler $binary\_remote\_addr 10m;

#负载均衡配置

upstream piao.jd.com {

#upstream的负载均衡，weight是权重，可以根据机器配置定义权重。weigth参数表示权值，权值越高被分配到的几率越大。

server 192.168.80.121:80 weight=3;

server 192.168.80.122:80 weight=2;

server 192.168.80.123:80 weight=3;

#nginx的upstream目前支持4种方式的分配

#1、轮询（默认）

#每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。

#2、weight

#指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。

#例如：

#upstream bakend {

# server 192.168.0.14 weight=10;

# server 192.168.0.15 weight=10;

#}

#2、ip\_hash

#每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，可以解决session的问题。

#例如：

#upstream bakend {

# ip\_hash;

# server 192.168.0.14:88;

# server 192.168.0.15:80;

#}

#3、fair（第三方）

#按后端服务器的响应时间来分配请求，响应时间短的优先分配。

#upstream backend {

# server server1;

# server server2;

# fair;

#}

#4、url\_hash（第三方）

#按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，后端服务器为缓存时比较有效。

#例：在upstream中加入hash语句，server语句中不能写入weight等其他的参数，hash\_method是使用的hash算法

#upstream backend {

# server squid1:3128;

# server squid2:3128;

# hash $request\_uri;

# hash\_method crc32;

#}

#tips:

#upstream bakend{#定义负载均衡设备的Ip及设备状态}{

# ip\_hash;

# server 127.0.0.1:9090 down;

# server 127.0.0.1:8080 weight=2;

# server 127.0.0.1:6060;

# server 127.0.0.1:7070 backup;

#}

#在需要使用负载均衡的server中增加 proxy\_pass http://bakend/;

#每个设备的状态设置为:

#1.down表示单前的server暂时不参与负载

#2.weight为weight越大，负载的权重就越大。

#3.max\_fails：允许请求失败的次数默认为1.当超过最大次数时，返回proxy\_next\_upstream模块定义的错误

#4.fail\_timeout:max\_fails次失败后，暂停的时间。

#5.backup： 其它所有的非backup机器down或者忙的时候，请求backup机器。所以这台机器压力会最轻。

#nginx支持同时设置多组的负载均衡，用来给不用的server来使用。

#client\_body\_in\_file\_only设置为On 可以讲client post过来的数据记录到文件中用来做debug

#client\_body\_temp\_path设置记录文件的目录 可以设置最多3层目录

#location对URL进行匹配.可以进行重定向或者进行新的代理 负载均衡

}

#虚拟主机的配置

server

{

#监听端口

listen 80;

#域名可以有多个，用空格隔开

server\_name www.jd.com jd.com;

index index.html index.htm index.php;

root /data/www/jd;

#对\*\*\*\*\*\*进行负载均衡

location ~ .\*.(php|php5)?$

{

fastcgi\_pass 127.0.0.1:9000;

fastcgi\_index index.php;

include fastcgi.conf;

}

#图片缓存时间设置

location ~ .\*.(gif|jpg|jpeg|png|bmp|swf)$

{

expires 10d;

}

#JS和CSS缓存时间设置

location ~ .\*.(js|css)?$

{

expires 1h;

}

#日志格式设定

#$remote\_addr与$http\_x\_forwarded\_for用以记录客户端的ip地址；

#$remote\_user：用来记录客户端用户名称；

#$time\_local： 用来记录访问时间与时区；

#$request： 用来记录请求的url与http协议；

#$status： 用来记录请求状态；成功是200，

#$body\_bytes\_sent ：记录发送给客户端文件主体内容大小；

#$http\_referer：用来记录从那个页面链接访问过来的；

#$http\_user\_agent：记录客户浏览器的相关信息；

#通常web服务器放在反向代理的后面，这样就不能获取到客户的IP地址了，通过$remote\_add拿到的IP地址是反向代理服务器的iP地址。反向代理服务器在转发请求的http头信息中，可以增加x\_forwarded\_for信息，用以记录原有客户端的IP地址和原来客户端的请求的服务器地址。

log\_format access '$remote\_addr - $remote\_user [$time\_local] "$request" '

'$status $body\_bytes\_sent "$http\_referer" '

'"$http\_user\_agent" $http\_x\_forwarded\_for';

#定义本虚拟主机的访问日志

access\_log /usr/local/nginx/logs/host.access.log main;

access\_log /usr/local/nginx/logs/host.access.404.log log404;

#对 "/" 启用反向代理

location / {

proxy\_pass http://127.0.0.1:88;

proxy\_redirect off;

proxy\_set\_header X-Real-IP $remote\_addr;

#后端的Web服务器可以通过X-Forwarded-For获取用户真实IP

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

#以下是一些反向代理的配置，可选。

proxy\_set\_header Host $host;

#允许客户端请求的最大单文件字节数

client\_max\_body\_size 10m;

#缓冲区代理缓冲用户端请求的最大字节数，

#如果把它设置为比较大的数值，例如256k，那么，无论使用firefox还是IE浏览器，来提交任意小于256k的图片，都很正常。如果注释该指令，使用默认的client\_body\_buffer\_size设置，也就是操作系统页面大小的两倍，8k或者16k，问题就出现了。

#无论使用firefox4.0还是IE8.0，提交一个比较大，200k左右的图片，都返回500 Internal Server Error错误

client\_body\_buffer\_size 128k;

#表示使nginx阻止HTTP应答代码为400或者更高的应答。

proxy\_intercept\_errors on;

#后端服务器连接的超时时间\_发起握手等候响应超时时间

#nginx跟后端服务器连接超时时间(代理连接超时)

proxy\_connect\_timeout 90;

#后端服务器数据回传时间(代理发送超时)

#后端服务器数据回传时间\_就是在规定时间之内后端服务器必须传完所有的数据

proxy\_send\_timeout 90;

#连接成功后，后端服务器响应时间(代理接收超时)

#连接成功后\_等候后端服务器响应时间\_其实已经进入后端的排队之中等候处理（也可以说是后端服务器处理请求的时间）

proxy\_read\_timeout 90;

#设置代理服务器（nginx）保存用户头信息的缓冲区大小

#设置从被代理服务器读取的第一部分应答的缓冲区大小，通常情况下这部分应答中包含一个小的应答头，默认情况下这个值的大小为指令proxy\_buffers中指定的一个缓冲区的大小，不过可以将其设置为更小

proxy\_buffer\_size 4k;

#proxy\_buffers缓冲区，网页平均在32k以下的设置

#设置用于读取应答（来自被代理服务器）的缓冲区数目和大小，默认情况也为分页大小，根据操作系统的不同可能是4k或者8k

proxy\_buffers 4 32k;

#高负荷下缓冲大小（proxy\_buffers\*2）

proxy\_busy\_buffers\_size 64k;

#设置在写入proxy\_temp\_path时数据的大小，预防一个工作进程在传递文件时阻塞太长

#设定缓存文件夹大小，大于这个值，将从upstream服务器传

proxy\_temp\_file\_write\_size 64k;

}

#设定查看Nginx状态的地址

location /NginxStatus {

stub\_status on;

access\_log on;

auth\_basic "NginxStatus";

auth\_basic\_user\_file confpasswd;

#htpasswd文件的内容可以用apache提供的htpasswd工具来产生。

}

#本地动静分离反向代理配置

#所有jsp的页面均交由tomcat或resin处理

location ~ .(jsp|jspx|do)?$ {

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_pass http://127.0.0.1:8080;

}

#所有静态文件由nginx直接读取不经过tomcat或resin

location ~ .\*.(htm|html|gif|jpg|jpeg|png|bmp|swf|ioc|rar|zip|txt|flv|mid|doc|ppt|

pdf|xls|mp3|wma)$

{

expires 15d;

}

location ~ .\*.(js|css)?$

{

expires 1h;

}

}

}

######Nginx配置文件nginx.conf中文详解#####

### 47 高并发量网站解决方案有哪些？

一个小型的网站，可以使用最简单的html静态页面就实现了，配合一些图片达到美化效果，所有的页面均存放在一个目录下，这样的网站对系统架构、性能的要求都很简单。随着互联网业务的不断丰富，网站相关的技术经过这些年的发展，已经细分到很细的方方面面，尤其对于大型网站来说，所采用的技术更是涉及面非常广，从硬件到软件、编程语言、数据库、WebServer、防火墙等各个领域都有了很高的要求，已经不是原来简单的html静态网站所能比拟的。

大型网站，比如门户网站，在面对大量用户访问、高并发请求方面，基本的解决方案集中在这样几个环节：使用高性能的服务器、高性能的数据库、高效率的编程语言、还有高性能的Web容器。这几个解决思路在一定程度上意味着更大的投入。

1. HTML静态化

其实大家都知道，效率最高、消耗最小的就是纯静态化的html页面，所以我们尽可能使我们的网站上的页面采用静态页面来实现，这个最简单的方法其实也是最有效的方法。但是对于大量内容并且频繁更新的网站，我们无法全部手动去挨个实现，于是出现了我们常见的信息发布系统CMS，像我们常访问的各个门户站点的新闻频道，甚至他们的其他频道，都是通过信息发布系统来管理和实现的，信息发布系统可以实现最简单的信息录入自动生成静态页面，还能具备频道管理、权限管理、自动抓取等功能，对于一个大型网站来说，拥有一套高效、可管理的CMS是必不可少的。

除了门户和信息发布类型的网站，对于交互性要求很高的社区类型网站来说，尽可能的静态化也是提高性能的必要手段，将社区内的帖子、文章进行实时的静态化、有更新的时候再重新静态化也是大量使用的策略，像Mop的大杂烩就是使用了这样的策略，网易社区等也是如此。

同时，html静态化也是某些缓存策略使用的手段，对于系统中频繁使用数据库查询但是内容更新很小的应用，可以考虑使用html静态化来实现。比如论坛中论坛的公用设置信息，这些信息目前的主流论坛都可以进行后台管理并且存储在数据库中，这些信息其实大量被前台程序调用，但是更新频率很小，可以考虑将这部分内容进行后台更新的时候进行静态化，这样避免了大量的数据库访问请求。

2. 图片服务器分离

大家知道，对于Web服务器来说，不管是Apache、IIS还是其他容器，图片是最消耗资源的，于是我们有必要将图片与页面进行分离，这是基本上大型网站都会采用的策略，他们都有独立的、甚至很多台的图片服务器。这样的架构可以降低提供页面访问请求的服务器系统压力，并且可以保证系统不会因为图片问题而崩溃。

在应用服务器和图片服务器上，可以进行不同的配置优化，比如apache在配置ContentType的时候可以尽量少支持、尽可能少的LoadModule，保证更高的系统消耗和执行效率。

3. 数据库集群、库表散列

大型网站都有复杂的应用，这些应用必须使用数据库，那么在面对大量访问的时候，数据库的瓶颈很快就能显现出来，这时一台数据库将很快无法满足应用，于是我们需要使用数据库集群或者库表散列。

在数据库集群方面，很多数据库都有自己的解决方案，Oracle、Sybase等都有很好的方案，常用的MySQL提供的Master/Slave也是类似的方案，您使用了什么样的DB，就参考相应的解决方案来实施即可。

上面提到的数据库集群由于在架构、成本、扩张性方面都会受到所采用DB类型的限制，于是我们需要从应用程序的角度来考虑改善系统架构，库表散列是常用并且最有效的解决方案。

我们在应用程序中安装业务和应用或者功能模块将数据库进行分离，不同的模块对应不同的数据库或者表，再按照一定的策略对某个页面或者功能进行更小的数据库散列，比如用户表，按照用户ID进行表散列，这样就能够低成本的提升系统的性能并且有很好的扩展性。

sohu的论坛就是采用了这样的架构，将论坛的用户、设置、帖子等信息进行数据库分离，然后对帖子、用户按照板块和ID进行散列数据库和表，最终可以在配置文件中进行简单的配置便能让系统随时增加一台低成本的数据库进来补充系统性能。

4. 缓存

缓存一词搞技术的都接触过，很多地方用到缓存。网站架构和网站开发中的缓存也是非常重要。这里先讲述最基本的两种缓存。高级和分布式的缓存在后面讲述。

架构方面的缓存，对Apache比较熟悉的人都能知道Apache提供了自己的缓存模块，也可以使用外加的Squid模块进行缓存，这两种方式均可以有效的提高Apache的访问响应能力。

网站程序开发方面的缓存，Linux上提供的Memory Cache是常用的缓存接口，可以在web开发中使用，比如用Java开发的时候就可以调用MemoryCache对一些数据进行缓存和通讯共享，一些大型社区使用了这样的架构。另外，在使用web语言开发的时候，各种语言基本都有自己的缓存模块和方法，PHP有Pear的Cache模块，Java就更多了，.net不是很熟悉，相信也肯定有。（.net是System.HttpRuntime.Cache，但是前提是你必须引用System.Web.dll。这也就说明了，System.Web.Caching.Cache这个对象完全是可以脱离于System.Web这个名称空间，而作为一个独立的缓存框架而存在。）

5. 镜像

镜像是大型网站常采用的提高性能和数据安全性的方式，镜像的技术可以解决不同网络接入商和地域带来的用户访问速度差异，比如ChinaNet和EduNet之间的差异就促使了很多网站在教育网内搭建镜像站点，数据进行定时更新或者实时更新。在镜像的细节技术方面，这里不阐述太深，有很多专业的现成的解决架构和产品可选。也有廉价的通过软件实现的思路，比如Linux上的rsync等工具。

6. 负载均衡

负载均衡将是大型网站解决高负荷访问和大量并发请求采用的高端解决办法。负载均衡技术发展了多年，有很多专业的服务提供商和产品可以选择，我个人接触过一些解决方法，其中有两个架构可以给大家做参考。

（1）、硬件四层交换

第四层交换使用第三层和第四层信息包的报头信息，根据应用区间识别业务流，将整个区间段的业务流分配到合适的应用服务器进行处理。

第四层交换功能就像是虚IP，指向物理服务器。它传输的业务服从的协议多种多样，有HTTP、FTP、NFS、Telnet或其他协议。这些业务在物理服务器基础上，需要复杂的载量平衡算法。在IP世界，业务类型由终端TCP或UDP端口地址来决定，在第四层交换中的应用区间则由源端和终端IP地址、TCP和UDP端口共同决定。

在硬件四层交换产品领域，有一些知名的产品可以选择，比如Alteon、F5等，这些产品很昂贵，但是物有所值，能够提供非常优秀的性能和很灵活的管理能力。“Yahoo中国”当初接近2000台服务器，只使用了三、四台Alteon就搞定了。

(2)、软件四层交换

大家知道了硬件四层交换机的原理后，基于OSI模型来实现的软件四层交换也就应运而生，这样的解决方案实现的原理一致，不过性能稍差。但是满足一定量的压力还是游刃有余的，有人说软件实现方式其实更灵活，处理能力完全看你配置的熟悉能力。

软件四层交换我们可以使用Linux上常用的LVS来解决，LVS就是Linux Virtual Server，他提供了基于心跳线heartbeat的实时灾难应对解决方案，提高系统的强壮性，同时可供了灵活的虚拟VIP配置和管理功能，可以同时满足多种应用需求，这对于分布式的系统来说必不可少。

一个典型的使用负载均衡的策略就是，在软件或者硬件四层交换的基础上搭建squid集群，这种思路在很多大型网站包括搜索引擎上被采用，这样的架构低成本、高性能还有很强的扩张性，随时往架构里面增减节点都非常容易。

对于大型网站来说，前面提到的每个方法可能都会被同时使用到，这里介绍得比较浅显，具体实现过程中很多细节还需要大家慢慢熟悉和体会。有时一个很小的squid参数或者apache参数设置，对于系统性能的影响就会很大。

7. 最新：CDN加速技术

什么是CDN？

CDN的全称是Content Delivery Network，即内容分发网络。CDN的通俗理解就是网站加速，CPU均衡负载，可以解决跨运营商，跨地区，服务器负载能力过低，带宽过少等带来的网站打开速度慢等问题。

CDN的基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。通过在网络各处放置节点服务器所构成的在现有的互联网基础之上的一层智能虚拟网络，CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。其目的是使用户可就近取得所需内容，解决 Internet网络拥挤的状况，提高用户访问网站的响应速度。

CDN有别于镜像，因为它比镜像更智能，或者可以做这样一个比喻：CDN=更智能的镜像+缓存+流量导流。因而，CDN可以明显提高Internet网络中信息流动的效率。从技术上全面解决由于网络带宽小、用户访问量大、网点分布不均等问题，提高用户访问网站的响应速度。

CDN的类型特点

CDN的实现分为三类：镜像、高速缓存、专线。

镜像站点（Mirror Site），是最常见的，它让内容直接发布，适用于静态和准动态的数据同步。但是购买和维护新服务器的费用较高，还必须在各个地区设置镜像服务器，配备专业技术人员进行管理与维护。对于大型网站来说，更新所用的带宽成本也大大提高了。

高速缓存，成本较低，适用于静态内容。Internet的统计表明，超过80%的用户经常访问的是20%的网站的内容，在这个规律下，缓存服务器可以处理大部分客户的静态请求，而原始的服务器只需处理约20%左右的非缓存请求和动态请求，于是大大加快了客户请求的响应时间，并降低了原始服务器的负载。

CDN服务一般会在全国范围内的关键节点上放置缓存服务器。

专线，让用户直接访问数据源，可以实现数据的动态同步。

CDN的实例

举个例子来说，当某用户访问网站时，网站会利用全球负载均衡技术，将用户的访问指向到距离用户最近的正常工作的缓存服务器上，直接响应用户的请求。

当用户访问已经使用了CDN服务的网站时，其解析过程与传统解析方式的最大区别就在于网站的授权域名服务器不是以传统的轮询方式来响应本地DNS的解析请求，而是充分考虑用户发起请求的地点和当时网络的情况，来决定把用户的请求定向到离用户最近同时负载相对较轻的节点缓存服务器上。

通过用户定位算法和服务器健康检测算法综合后的数据，可以将用户的请求就近定向到分布在网络“边缘”的缓存服务器上，保证用户的访问能得到更及时可靠的响应。

由于大量的用户访问都由分布在网络边缘的CDN节点缓存服务器直接响应了，这就不仅提高了用户的访问质量，同时有效地降低了源服务器的负载压力。

### 48 平台上的图片如何防盗链？

http标准协议中有专门的字段记录referer

一来可以追溯上一个入站地址是什么

二来对于资源文件，可以跟踪到包含显示他的网页地址是什么。

因此所有防盗链方法都是基于这个Referer字段

网上比较多的2种

一种是使用apache文件FileMatch限制，在httpd.conf中增加 ( 其实也可以将把下面的语句存成一个.htaccess文件)，并放到你的网站的根目录（就是www/html目录），这样子别人就没有办法盗连你的东东了~~

SetEnvIfNoCase Referer"^http://yahoo.com/" local\_ref=1

Order Allow,Deny

Allow from env=local\_ref

Allow from 127.0.0.1

这种很方便禁止非允许访问URL引用各种资源文件

请大家注意，把第一句"^http://yahoo.com/"改为你的网站，比如我的网站是： http://www.linji.cn

我应该这么写的

"^http://www.linji.cn/"

第二种是使用rewrite,需要增加apache的mode\_rewrite，支持.htaccess文件目录权限限制

在虚拟主机根目录增加.htaccess文件，描述从定向，把非本地地址refer的图片文件都从定向到警告图片或者警告网页上。

首先要确认你的服务器或空间的服务器解译引擎为Apache2，还有支持.htaccess客户设置文件，

如果你有自己的服务器就请先对./conf/httpd.conf 文件做以下修改

找到：#LoadModule rewrite\_module modules/mod\_rewrite.so

把前面的 # 给去丢

找到等一个AllowOverride None 改为 AllowOverride All

重启Apache2服务器

接下就是做一个.htaccess 文件了，其.htaccess 文件内容为

RewriteEngine on

RewriteCond %{HTTP\_REFERER} !^http://aaoo.net/.\*$ [NC]

RewriteCond %{HTTP\_REFERER} !^http://aaoo.net$ [NC]

RewriteCond %{HTTP\_REFERER} !^http://www.aaoo.net/.\*$ [NC]

RewriteCond %{HTTP\_REFERER} !^http://www.aaoo.net$ [NC]

RewriteRule .\*.(jpg|jpeg|gif|png|bmp|rar|zip|exe)$ http://down.yoyo.com.ru/err.html [R,NC]

其中有色的地方都是要改为你的：

红色：就是改为你提供下载页面的地址，也就是只有通过这个地址才可以下载你所提供的东东。

蓝色：就是要保护文件的扩展名(以|分开)，也就是说以这些为扩展名的文件只有通过红色的地址才可以访问。

绿色：如果不是通过红色的地址访问蓝色这些为扩展名的文件时就回重定向到绿色地址上。

这个方法有个好处是，不同的虚拟主机用不同的描述定义。

接下就是怎么用.htaccess 文件来实现防盗链了。

首先要在空间上建两个目录(当然目录名随你)，一个为 web 另一个为 down ，

web 是用来放下载页面的(或下载程序)，down 当然就是放你提供的东东的啦，

把.htaccess 文件的红色部分改一下，改为http://你的域名/web。蓝色部分

改为你要保护文件的扩展名。绿色部分改为http://你的域名/web。改后保存

.htaccess 文件把它上传到 down 目录。

还有第三种：

我在解决plog禁止盗链的时候，发现个问题，也算个好方法。

plog把所有资源都自己管理起来，用resserver.php来动态显示，这样统一的入口方便添加权限操作。

同时造成上面2种方法无法使用，因为不再是apache直接访问资源文件，而是php通过文件读取。

因此只能在代码中做手脚：在读取资源文件输出之前，加如下判断代码

引用

$referer = $\_SERVER['HTTP\_REFERER'];

$selfurl = $\_SERVER['HTTP\_HOST'];

if(false == strpos($referer,$selfurl))

{

echo '非法盗链!';

exit(1);

}

这里有些偷懒，直接看引用地址中是否包含host地址，不过原理就是这样，判断referer是否是本站地址。

我们常常在下载的时候，也碰到盗链网站无法下载，报盗链的问题。要下载这类文件最简单的方法就是改referer

比方flashget中，网址下面的"引用"一栏中，直接填写下载地址就可以了

### 49 消息队列的原理和实现？

消息队列技术是分布式应用间交换信息的一种技术。消息队列可驻留在内存或磁盘上,队列存储消息直到它们被应用程序读走。通过消息队列，应用程序可独立地执行--它们不需要知道彼此的位置、或在继续执行前不需要等待接收程序接收此消息。

在分布式计算环境中，为了集成分布式应用，开发者需要对异构网络环境下的分布式应用提供有效的通信手段。为了管理需要共享的信息，对应用提供公共的信息交换机制是重要的。

消息队列为构造以同步或异步方式实现的分布式应用提供了松耦合方法。消息队列的API调用被嵌入到新的或现存的应用中，通过消息发送到内存或基于磁盘的队列或从它读出而提供信息交换。消息队列可用在应用中以执行多种功能，比如要求服务、交换信息或异步处理等。

中间件是一种独立的系统软件或服务程序，分布式应用系统借助这种软件在不同的技术之间共享资源，管理计算资源和网络通讯。它在计算机系统中是一个关键软件，它能实现应用的互连和互操作性，能保证系统的安全、可靠、高效的运行。中间件位于用户应用和操作系统及网络软件之间，它为应用提供了公用的通信手段，并且独立于网络和操作系统。中间件为开发者提供了公用于所有环境的应用程序接口，当应用程序中嵌入其函数调用，它便可利用其运行的特定操作系统和网络环境的功能，为应用执行通信功能。

如果没有消息中间件完成信息交换，应用开发者为了传输数据，必须要学会如何用网络和操作系统软件的功能，编写相应的应用程序来发送和接收信息，且交换信息没有标准方法，每个应用必须进行特定的编程从而和多平台、不同环境下的一个或多个应用通信。例如，为了实现网络上不同主机系统间的通信，将要求具备在网络上如何交换信息的知识（比如用TCP/IP的socket程序设计）；为了实现同一主机内不同进程之间的通讯，将要求具备操作系统的消息队列或命名管道(Pipes)等知识。

MQ的通讯模式

1) 点对点通讯：点对点方式是最为传统和常见的通讯方式，它支持一对一、一对多、多对多、多对一等多种配置方式，支持树状、网状等多种拓扑结构。

2) 多点广播：MQ适用于不同类型的应用。其中重要的，也是正在发展中的是"多点广播"应用，即能够将消息发送到多个目标站点(Destination List)。可以使用一条MQ指令将单一消息发送到多个目标站点，并确保为每一站点可靠地提供信息。MQ不仅提供了多点广播的功能，而且还拥有智能消息分发功能，在将一条消息发送到同一系统上的多个用户时，MQ将消息的一个复制版本和该系统上接收者的名单发送到目标MQ系统。目标MQ系统在本地复制这些消息，并将它们发送到名单上的队列，从而尽可能减少网络的传输量。

3) 发布/订阅(Publish/Subscribe)模式：发布/订阅功能使消息的分发可以突破目的队列地理指向的限制，使消息按照特定的主题甚至内容进行分发，用户或应用程序可以根据主题或内容接收到所需要的消息。发布/订阅功能使得发送者和接收者之间的耦合关系变得更为松散，发送者不必关心接收者的目的地址，而接收者也不必关心消息的发送地址，而只是根据消息的主题进行消息的收发。在MQ家族产品中，MQ Event Broker是专门用于使用发布/订阅技术进行数据通讯的产品，它支持基于队列和直接基于TCP/IP两种方式的发布和订阅。

4) 群集(Cluster)：为了简化点对点通讯模式中的系统配置，MQ提供Cluster(群集)的解决方案。群集类似于一个域(Domain)，群集内部的队列管理器之间通讯时，不需要两两之间建立消息通道，而是采用群集(Cluster)通道与其它成员通讯，从而大大简化了系统配置。此外，群集中的队列管理器之间能够自动进行负载均衡，当某一队列管理器出现故障时，其它队列管理器可以接管它的工作，从而大大提高系统的高可靠性。

### 50 java堆，简述新生代老年代？

java堆，分新生代老年代，新生代有Eden，from surviver，to surviver三个空间，堆被所有线程共。eden内存不足时，发生一次minor GC，会把from survivor和eden的对象复制到to survivor，这次的to survivor就变成了下次的from survivor，经过多次minor GC，默认15次，达到次数的对象会从survivor进行老年代。1次new如果新生代装不下，则直接进入老年代。

堆的年轻代大则老年代小，GC少，但是每次时间会比较长。年轻代小则老年代大，会缩短每次GC的时间，但是次数频繁。可以让老年代尽量缓存常用对象，JVM默认年轻代和老年代的大小比例为1:2,。观察峰值老年代内存，不影响full GC，加大老年代可调1:1，但是要给老年代预留三分之一的空间。减少使用全局变量和大对象 ，调整新生代，老年代到最合适。

### 51 内存溢出可能原因和解决？

原因可能是

A，数据加载过多，如1次从数据库中取出过多数据

B，集合类中有对对象的引用，用完后没有清空或者集合对象未置空导致引用存在等，是的JVM无法回收

C，死循环，过多重复对象

D，第三方软件的bug

E，启动参数内存值设定的过小。

例如方法：修改JVM启动参数，加内存(-Xms，-Xmx)；错误日志，是否还有其他错误；代码走查

### 52 http是无状态通信，http的请求方式有哪些，可以自己定义新的请求方式么？

一、无状态描述

无状态是指，当浏览器发送请求给服务器的时候，服务器响应，但是同一个浏览器再发送请求给服务器的时候，他会响应，但是他不知道你就是刚才那个浏览器，简单地说，就是服务器不会去记得你，所以是无状态协议。

而DNS是有状态协议 。

HTTP是一个属于应用层的面向对象的协议，HTTP协议一共有五大特点，

1、支持客户/服务器模式；

2、简单快速；

3、灵活；

4、无连接；

5、无状态；

“无状态”是HTTP协议的主要特点之一，以下为“无状态”的解释。

无状态：是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。 www.2cto.com

HTTP是Hyper Text Transfer Protocol的缩写，顾名思义，这个协议支持着超文本的传输。那么什么是超文本呢？说白了就是使用HTML编写的页面。通常，我们使用客户端浏览器访问服务器的资源，最常见的URL也是以html为后缀的文件。因此，我们可以说超文本是网络上最主要的资源。

二、八种方法

HTTP/1.1协议中共定义了八种方法（有时也叫“动作”）来表明Request-URI指定的资源的不同操作方式。

OPTIONS 返回服务器针对特定资源所支持的HTTP请求方法。也可以利用向Web服务器发送'\*'的请求来测试服务器的功能性。

　HEAD 向服务器索要与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。

　GET 向特定的资源发出请求。注意：GET方法不应当被用于产生“副作用”的操作中，例如在web app.中。其中一个原因是GET可能会被网络蜘蛛等随意访问。

　POST 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。

　PUT 向指定资源位置上传其最新内容。

　DELETE 请求服务器删除Request-URI所标识的资源。

　TRACE 返回显服务器收到的请求，主要用于测试或诊断。

　CONNECT HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

详述：

0，GET：GET可以说是最常见的了，它本质就是发送一个请求来取得服务器上的某一资源。资源通过一组HTTP头和呈现数据（如HTML文本，或者图片或者视频等）返回给客户端。GET请求中，永远不会包含呈现数据。

　　1，HEAD：HEAD和GET本质是一样的，区别在于HEAD不含有呈现数据，而仅仅是HTTP头信息。想象一个业务情景：欲判断某个资源是否存在，我们通常使用GET，但这里用HEAD则意义更加明确。

　　2，PUT：这个方法比较少见。HTML表单也不支持这个。本质上来讲， PUT和POST极为相似，都是向服务器发送数据，但它们之间有一个重要区别，PUT通常指定了资源的存放位置，而POST则没有，POST的数据存放位置由服务器自己决定。举个例子：如一个用于提交博文的URL，/addBlog。如果用PUT，则提交的URL会是像这样的”/addBlog/abc123”，其中abc123就是这个博文的地址。而如果用POST，则这个地址会在提交后由服务器告知客户端。目前大部分博客都是这样的。显然，PUT和POST用途是不一样的。具体用哪个还取决于当前的业务场景。

　　3，DELETE：删除某一个资源。基本上这个也很少见，不过还是有一些地方比如amazon的S3云服务里面就用的这个方法来删除资源。

　　4，POST：向服务器提交数据。这个方法用途广泛，几乎目前所有的提交操作都是靠这个完成。

　　5，OPTIONS：这个方法很有趣，但极少使用。它用于获取当前URL所支持的方法。若请求成功，则它会在HTTP头中包含一个名为“Allow”的头，值是所支持的方法，如“GET, POST”。

　　其实还有一个TRACE方法，不过这个基本上不会用到，这里就不介绍了。以上的六种方法，我们可以跟数据库的CRUD增删改查操作对应起来：

　　CREATE ：PUT

　　READ：GET

　　UPDATE：POST

　　DELETE：DELETE

### 53 socket通信，以及长连接，分包，连接异常断开的处理？

一、一些概念

1 短连接：

连接->传输数据->关闭连接

HTTP是无状态的，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。

也可以这样说：短连接是指SOCKET连接后发送后接收完数据后马上断开连接。

2 长连接：

连接->传输数据->保持连接 -> 传输数据-> 。。。 ->关闭连接。

长连接指建立SOCKET连接后不管是否使用都保持连接，但安全性较差。

之所以出现粘包和半包现象,是因为TCP当中,只有流的概念,没有包的概念.

3 半包

指接受方没有接受到一个完整的包，只接受了部分，这种情况主要是由于TCP为提高传输效率，将一个包分配的足够大，导致接受方并不能一次接受完。（在长连接和短连接中都会出现）。

4 粘包与分包

指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。出现粘包现象的原因是多方面的，它既可能由发送方造成，也可能由接收方造成。发送方引起的粘包是由TCP协议本身造成的，TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据。若连续几次发送的数据都很少，通常TCP会根据优化算法把这些数据合成一包后一次发送出去，这样接收方就收到了粘包数据。接收方引起的粘包是由于接收方用户进程不及时接收数据，从而导致粘包现象。这是因为接收方先把收到的数据放在系统接收缓冲区，用户进程从该缓冲区取数据，若下一包数据到达时前一包数据尚未被用户进程取走，则下一包数据放到系统接收缓冲区时就接到前一包数据之后，而用户进程根据预先设定的缓冲区大小从系统接收缓冲区取数据，这样就一次取到了多包数据。分包是指在出现粘包的时候我们的接收方要进行分包处理。（在长连接中都会出现）

5 什么时候需要考虑半包的情况?

从备注中我们了解到Socket内部默认的收发缓冲区大小大概是8K，但是我们在实际中往往需要考虑效率问题，重新配置了这个值，来达到系统的最佳状态。

一个实际中的例子：用mina作为服务器端，使用的缓存大小为10k，这里使用的是短连接，所有不用考虑粘包的问题。

6 问题描述：在并发量比较大的情况下，就会出现一次接受并不能完整的获取所有的数据。

处理方式：

1.通过包头+包长+包体的协议形式，当服务器端获取到指定的包长时才说明获取完整。

2.指定包的结束标识，这样当我们获取到指定的标识时，说明包获取完整。

7 什么时候需要考虑粘包的情况?

@当时短连接的情况下，不用考虑粘包的情况

@如果发送数据无结构，如文件传输，这样发送方只管发送，接收方只管接收存储就ok，也不用考虑粘包

@如果双方建立连接，需要在连接后一段时间内发送不同结构数据

处理方式：

接收方创建一预处理线程，对接收到的数据包进行预处理，将粘连的包分开

注：粘包情况有两种，一种是粘在一起的包都是完整的数据包，另一种情况是粘在一起的包有不完整的包

备注:

一个包没有固定长度，以太网限制在46－1500字节，1500就是以太网的MTU，超过这个量，TCP会为IP数据报设置偏移量进行分片传输，现在一般可允许应用层设置8k（NTFS系）的缓冲区，8k的数据由底层分片，而应用看来只是一次发送。windows的缓冲区经验值是4k,Socket本身分为两种，流(TCP)和数据报(UDP)，你的问题针对这两种不同使用而结论不一样。甚至还和你是用阻塞、还是非阻塞Socket来编程有关。

1、通信长度，这个是你自己决定的，没有系统强迫你要发多大的包，实际应该根据需求和网络状况来决定。对于TCP，这个长度可以大点，但要知道，Socket内部默认的收发缓冲区大小大概是8K，你可以用SetSockOpt来改变。但对于UDP，就不要太大，一般在1024至10K。注意一点，你无论发多大的包，IP层和链路层都会把你的包进行分片发送，一般局域网就是1500左右，广域网就只有几十字节。分片后的包将经过不同的路由到达接收方，对于UDP而言，要是其中一个分片丢失，那么接收方的IP层将把整个发送包丢弃，这就形成丢包。显然，要是一个UDP发包佷大，它被分片后，链路层丢失分片的几率就佷大，你这个UDP包，就佷容易丢失，但是太小又影响效率。最好可以配置这个值，以根据不同的环境来调整到最佳状态。

send()函数返回了实际发送的长度，在网络不断的情况下，它绝不会返回(发送失败的)错误，最多就是返回0。对于TCP你可以字节写一个循环发送。当send函数返回SOCKET\_ERROR时，才标志着有错误。但对于UDP，你不要写循环发送，否则将给你的接收带来极大的麻烦。所以UDP需要用SetSockOpt来改变Socket内部Buffer的大小，以能容纳你的发包。明确一点，TCP作为流，发包是不会整包到达的，而是源源不断的到，那接收方就必须组包。而UDP作为消息或数据报，它一定是整包到达接收方。

2、关于接收，一般的发包都有包边界，首要的就是你这个包的长度要让接收方知道，于是就有个包头信息，对于TCP，接收方先收这个包头信息，然后再收包数据。一次收齐整个包也可以，可要对结果是否收齐进行验证。这也就完成了组包过程。UDP，那你只能整包接收了。要是你提供的接收Buffer过小，TCP将返回实际接收的长度，余下的还可以收，而UDP不同的是，余下的数据被丢弃并返回WSAEMSGSIZE错误。注意TCP，要是你提供的Buffer佷大，那么可能收到的就是多个发包，你必须分离它们，还有就是当Buffer太小，而一次收不完Socket内部的数据，那么Socket接收事件(OnReceive)，可能不会再触发，使用事件方式进行接收时，密切注意这点。这些特性就是体现了流和数据包的区别。

二、现象：服务器端等待客户断连接，当socket连接建立后，如果客户端异常断开，服务器会抛出异常，从而导致程序运行中断

目标：希望服务器一直等待连接，客户端中断后程序不退出，而客户端重新恢复后可以继续保持连接

代码：

public class Receive

{

public static byte[] buffer= new byte[1024];

public static ManualResetEvent socketEvent = new ManualResetEvent(false);

public static Socket sListener = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

public static Socket handler = null;

public static string ClientBroken = "An connection was forcibly closed by the remote host";

public static void receive()

{

try

{

Console.WriteLine("Main ThreadID:" + AppDomain.GetCurrentThreadId());

byte[] bytes = new byte[1024];

IPAddress ipAddr = IPAddress.Parse("127.0.0.1");

int Port = 10001;

IPEndPoint EPServer = new IPEndPoint(ipAddr, Port);

//Binding a socket

sListener.Bind(EPServer);

//Start listening

sListener.Listen(10);

while(true)

{

if (handler==null)

{

//first must make a connect

Console.WriteLine("waiting for a connection...");

//asychronous function for accepting connections

sListener.BeginAccept(new AsyncCallback(AcceptCallback), sListener);

socketEvent.WaitOne();

handler.BeginReceive(buffer,0,buffer.Length,0,new AsyncCallback(ReceiveCallback),handler);

socketEvent.WaitOne();

}

else

{

Console.WriteLine("waiting next message...");

socketEvent.Reset();

handler.BeginReceive(buffer,0,buffer.Length,0,new AsyncCallback(ReceiveCallback),handler);

socketEvent.WaitOne();

}

}

Console.ReadLine();

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

Console.ReadLine();

}

public static void AcceptCallback(IAsyncResult ar)

{

try

{

Console.WriteLine("AcceptCallback Thread ID:" + AppDomain.GetCurrentThreadId());

Socket listener = (Socket)ar.AsyncState;

//new socket

handler = listener.EndAccept(ar);

handler.BeginReceive(buffer,0,buffer.Length,0,new AsyncCallback(ReceiveCallback),handler);

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

}

public static void ReceiveCallback(IAsyncResult ar)

{

string err\_message=null;

try

{

Console.WriteLine("ReceiveCallback Thread ID:" + AppDomain.GetCurrentThreadId());

string content = String.Empty;

handler = (Socket)ar.AsyncState;

int bytesRead = handler.EndReceive(ar);

//if there is some data...

if (bytesRead>0)

{

//append it to the main string

content += Encoding.ASCII.GetString(buffer,0,bytesRead);

//if we encounter the end of message character

if (content.IndexOf((char)3)> -1 || content.IndexOf((char)16)>-1)

{

Console.WriteLine("Read "+content.Length+" bytes from socket. /n Data:"+content);

socketEvent.Set();

}

else

{

//otherwise receive the remaining data

handler.BeginReceive(buffer,0,buffer.Length,0,new AsyncCallback(ReceiveCallback),handler);

}

}

}

catch(Exception e)

{

err\_message = e.Message;

if (err\_message.IndexOf("An existing connection was forcibly closed by the remote host")> -1)

{

Console.WriteLine("An existing connection was forcibly closed by the remote host");

//handler.Shutdown(SocketShutdown.Both);

//handler.Close();

Console.WriteLine("waiting for a connection...");

//asychronous function for accepting connections

sListener.BeginAccept( new AsyncCallback(AcceptCallback), sListener);

}

else

{

Console.WriteLine(e.ToString());

}

}

}

}

说明：关键在于最后这段的异常处理，接收中断后，服务器端重新等待接收。

现象：客户端与服务器连接，当socket连接建立后，如果服务器端异常断开，客户端会抛出异常，从而导致程序运行中断

目标：希望客户端出现提示，服务器端中断后程序不退出，而服务器端重新恢复后可以继续保持连接

代码：

public class AsyncComm

{

public static string theResponse = "";

public static byte[] buffer = new byte[1024];

public static ManualResetEvent socketEvent = new ManualResetEvent(false);

public static Socket sClient= new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

public static IPEndPoint EPServer = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 10001);

public static void send(string data)

{

byte[] byteData=null;

byteData = Encoding.ASCII.GetBytes(data);

try

{

if (!sClient.Connected)

{

Console.WriteLine(System.DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:ffff")+" "+"Connect begining......");

sClient.BeginConnect(EPServer, new AsyncCallback(ConnectCallback),sClient);

socketEvent.WaitOne();

}

sClient.BeginSend(byteData,0,byteData.Length,0,new AsyncCallback(SendCallback),sClient);

socketEvent.WaitOne();

}

catch (Exception e)

{

Console.WriteLine("Server side is broken...");

socketEvent.Reset();

return;

}

}

public static void ConnectCallback(IAsyncResult ar)

{

try

{

Thread thr = Thread.CurrentThread;

Console.WriteLine("ConnectCallback Thread State:" + AppDomain.GetCurrentThreadId());

Socket sClient = (Socket)ar.AsyncState;

sClient.EndConnect(ar);

Console.WriteLine("Socket connected to " + sClient.RemoteEndPoint.ToString());

socketEvent.Set();

}

catch (Exception ex)

{

Console.WriteLine(System.DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss:ffff")+"||"+AppDomain.GetCurrentThreadId()+"||3--Level 3 Server connection is broken, waiting for Level 3 Server connection......");

sClient= new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

socketEvent.Set();

}

}

receive函数相同，可以参照写出

说明：

在每次发送或接收时检测当前socket是否连接，如果没有连接，就启动连接，并阻塞线程等待ConnectCallback的返回

### 54 socket通信模型的使用，AIO和NIO？

客户端:创建套接字，连接服务器，然后不停的发送和接收数据。

比较容易想到的一种服务器模型就是采用一个主线程，负责监听客户端的连接请求，当接收到某个客户端的连接请求后，创建一个专门用于和该客户端通信的套接字和一个辅助线程。以后该客户端和服务器的交互都在这个辅助线程内完成。这种方法比较直观，程序非常简单而且可移植性好，但是不能利用平台相关的特性。例如，如果连接数增多的时候（成千上万的连接），那么线程数成倍增长，操作系统忙于频繁的线程间切换，而且大部分线程在其生命周期内都是处于非活动状态的，这大大浪费了系统的资源。所以，如果你已经知道你的代码只会运行在Windows平台上，建议采用Winsock I/O模型。

一.Select模型: 轮询fd\_set集合

利用select函数，实现对I/O 的管理。最初设计该模型时，主要面向的是某些使用UNIX操作系统的计算机，它们采用的是Berkeley套接字方案。Select模型已集成到 Winsock 1.1中，它使那些想避免在套接字调用过程中被无辜“锁定”的应用程序，采取一种有序的方式，同时进行对多个套接字的管理。

int select(

int nfds,

fd\_set\* readfds,

fd\_set\* writefds,

fd\_set\* exceptfds,

const struct timeval\* timeout

);

nfds：本参数忽略，仅起到兼容作用。

readfds：（可选）指针，指向一组等待可读性检查的套接口。

writefds：（可选）指针，指向一组等待可写性检查的套接口。

exceptfds：（可选）指针，指向一组等待错误检查的套接口。

timeout：select()最多等待时间，对阻塞操作则为NULL。

FD\_CLR(s,\*set)：从集合set中删除描述字s。

FD\_ISSET(s,\*set)：若s为集合中一员，非零；否则为零。

FD\_SET(s,\*set)：向集合添加描述字s。

FD\_ZERO(\*set)：将set初始化为空集NULL。

timeout参数控制select()完成的时间。若timeout参数为空指针，则select()将一直阻塞到有一个描述字满足条件。否则的话，timeout指向一个timeval结构，其中指定了select()调用在返回前等待多长时间。如果timeval为{0,0}，则 select()立即返回，这可用于探询所选套接口的状态。

服务器来轮询查看某个套接字是否仍然处于读集中，如果是，则接收数据。如果接收的数据长度为0，或者发生WSAECONNRESET错误，则表示客户端套接字主动关闭，这时需要将服务器中对应的套接字所绑定的资源释放掉，然后调整我们的套接字数组（将数组中最后一个套接字挪到当前的位置上）

除了需要有条件接受客户端的连接外，还需要在连接数为0的情形下做特殊处理，因为如果读集中没有任何套接字，select函数会立刻返回。

当调用非阻塞模式时，可以说socket在select上设置超时时间阻塞应用。

select 会在超时时间测试fd\_set集合是否可用，如果超时/没有数据可读了会清除当前集合成员。

二.异步选择

应用程序可以在一个套接字上接收以WINDOWS消息为基础的网络事件通知。该模型的实现方法是通过调用WSAAsynSelect函数 自动将套接字设置为非阻塞模式，并向WINDOWS注册一个或多个网络时间，并提供一个通知时使用的窗口句柄。当注册的事件发生时，对应的窗口将收到一个基于消息的通知。

三.事件选择

Winsock 提供了另一个有用的异步I/O模型。和WSAAsyncSelect模型类似的是，它也允许应用程序在一个或多个套接字上，接收以事件为基础的网络事件通知。

基本思想是将每个套接字都和一个WSAEVENT对象对应起来，并且在关联的时候指定需要关注的哪些网络事件。一旦在某个套接字上发生了我们关注的事件（FD\_READ和FD\_CLOSE），与之相关联的WSAEVENT对象被Signaled。

四.重叠I/O模型

readfile或者writefile的调用马上就会返回，这时候你可以去做你要做的事，系统会自动替你完成readfile或者writefile,在你调用了readfile或者writefile后，你继续做你的事，系统同时也帮你完成readfile或writefile的操作，这就是所谓的重叠。

1.用事件通知方式实现的重叠I/O模型

异步I/O函数WSARecv。在调用WSARecv时，指定一个 WSAOVERLAPPED结构，这个调用不是阻塞的，也就是说，它会立刻返回。一旦有数据到达的时候，被指定的WSAOVERLAPPED结构中的 hEvent被Signaled。使得与该套接字相关联的WSAEVENT对象也被Signaled，所以WSAWaitForMultipleEvents的调用操作成功返回。

2.用完成例程方式实现的重叠I/O模型

WSARecv时传递CompletionROUTINE指针，回调函数，当IO请求完成时调用该回调函数完成我们需要处理的工作，在这个模型中，主线程只用不停的接受连接即可；辅助线程判断有没有新的客户端连接被建立，如果有，就为那个客户端套接字激活一个异步的WSARecv操作，然后调用SleepEx使线程处于一种可警告的等待状态，以使得I/O完成后 CompletionROUTINE可以被内核调用。如果辅助线程不调用SleepEx，则内核在完成一次I/O操作后，无法调用完成例程（因为完成例程的运行应该和当初激活WSARecv异步操作的代码在同一个线程之内）。

Windows提供了四种异步IO技术，机制几乎是相同的，区别在于通知结果的方式不同：

1、使一个设备内核对象变为有信号

Windows将设备句柄看作可同步的对象，即它可以处于有信号或处于无信号状态，当创建设备句柄、以异步的方式发送IO请求时，该句柄处于无信号状态，当异步IO完成之后，该句柄受信，通过WaitForSingleobject或WatiForMultipleObjects函数可以判断设备操作合适完成。该技术只能用于一个设备只发送一个IO请求，否则，若一个设备对应多个操作，当句柄受信时无法判断是该设备的那个操作完成。

2、使一个事件内核对象变为有信号

针对每个I/O操作绑定一个内核事件对象，并将等待事件等待函数等待该事件的受信，当I/O操作完成后系统使得与该操作绑定的事件受信，从而判断那个操作完成。该技术解决了使一个设备内核对象变为有信号技术中一个设备只能对应一个操作的不足。

3、警告I/O

在该技术中，当发出设备IO请求时，同时要求我们传递一个被称为完成例程的回调函数，当IO请求完成时调用该回调函数完成我们需要处理的工作。该技术允许单个设备同时进行多个I/O请求。

4、完成端口

完成端口技术多用于处理大规模的请求，通过内在的进程池技术可以达到很高的性能。

-

五.完成端口模型

只有在你的应用程序需要同时管理数百乃至上千个套接字的时候，而且希望随着系统内安装的CPU数量的增多，应用程序的性能也可以线性提升，才应考虑采用“完成端口”模型。

完成端口内部提供了线程池的管理，可以避免反复创建线程的开销，同时可以根据CPU的个数灵活的决定线程个数，而且可以让减少线程调度的次数从而提高性能。

首先要创建一个 I / O完成端口对象

HANDLE CreateIoCompletionPort (

HANDLE FileHandle, // handle to file

HANDLE ExistingCompletionPort, // handle to I/O completion port

ULONG\_PTR CompletionKey, // completion key

DWORD NumberOfConcurrentThreads // number of threads to execute concurrently

);

我们深入探讨其中的各个参数之前，首先要注意该函数实际用于两个明显有别的目的：

■ 用于创建一个完成端口对象。

■ 将一个句柄同完成端口关联到一起。

最开始创建一个完成端口时，唯一感兴趣的参数便是 NumberOfConcurrentThreads（并发

线程的数量）；前面三个参数都会被忽略。NumberOfConcurrentThreads参数的特殊之处在于，它定义了在一个完成端口上，同时允许执行的线程数量。理想情况下，我们希望每个处理器各自负责一个线程的运行，为完成端口提供服务，避免过于频繁的线程“场景”切换。若将该参数设为0，表明系统内安装了多少个处理器，便允许同时运行多少个线程！可用下述代码创建一个I / O完成端口：

CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, 0, 0)

该语句的作用是返回一个句柄，在为完成端口分配了一个套接字句柄后，用来对那个端

口进行标定（引用）。

工作者线程调用 GetQueuedCompletionStatus 来轮询完成端口队列

如果你想在Windows平台上构建服务器应用，那么I/O模型是你必须考虑的。Windows操作系统提供了

选择（Select）、异步选择（WSAAsyncSelect）、事件选择（WSAEventSelect）、重叠I/O（Overlapped I/O）和完成端口（Completion Port)

共五种I/O模型。每一种模型均适用于一种特定的应用场景。程序员应该对自己的应用需求非常明确，而且综合考虑到程序的扩展性和可移植性

等因素，作出自己的选择。

第一点，NIO少了1次从内核空间到用户空间的拷贝。

ByteBuffer.allocateDirect()分配的内存使用的是本机内存而不是Java堆上的内存，和网络或者磁盘交互都在操作系统的内核空间中发生。allocateDirect()的区别在于这块内存不由java堆管理, 但仍然在同一用户进程内。

第二点，NIO以块处理数据，IO以流处理数据

第三点，非阻塞，NIO1个线程可以管理多个输入输出通道

### 55 HTTP协议、  HTTPS协议，SSL协议及完整交互过程；

1.        安全套接字（Secure Socket Layer，SSL）协议是Web浏览器与Web服务器之间安全交换信息的协议。

2.    SSL协议的三个特性

Ø  保密：在握手协议中定义了会话密钥后，所有的消息都被加密。

Ø  鉴别：可选的客户端认证，和强制的服务器端认证。

Ø  完整性：传送的消息包括消息完整性检查（使用MAC）。

3.    SSL的位置



HTTPS

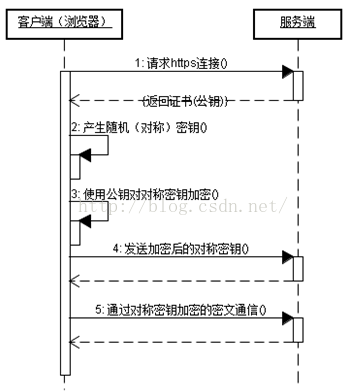
1.     HTTPS基于SSL的HTTP协议。

2.     HTTPS使用与HTTP不同的端口(，一个加密、身份验证层（HTTP与TCP之间）)。

3.     提供了身份验证与加密通信方法，被广泛用于互联网上安全敏感的通信。

交互过程

客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤，如图所示。



1)    客户端请求建立SSL连接，并将自己支持的一套加密规则发送给网站。

2)    网站从中选出一组加密算法与HASH算法，并将自己的身份信息以证书的形式发回给浏览器。证书里面包含了网站地址，加密公钥，以及证书的颁发机构等信息

3)    获得网站证书之后浏览器要做以下工作：

Ø  验证证书的合法性

Ø  如果证书受信任，浏览器会生成一串随机数的密码，并用证书中提供的公钥加密。

Ø  使用约定好的HASH计算握手消息，

Ø  使用生成的随机数对消息进行加密，最后将之前生成的所有信息发送给网站。

4)    网站接收浏览器发来的数据之后要做以下的操作：

Ø  使用自己的私钥将信息解密取出密码

Ø  使用密码解密浏览器发来的握手消息，并验证HASH是否与浏览器发来的一致。

Ø  使用密码加密一段握手消息，发送给浏览器

5)    浏览器解密并计算握手消息的HASH，如果与服务端发来的HASH一致，此时握手结束。

6)    使用随机密码和对称加密算法对传输的数据加密，传输。

4.     密与HASH算法如下：

1)     非对称加密算法：RSA，DSA/DSS，用于在握手过程中加密生成的密码。

2)     对称加密算法：AES，RC4，3DES，用于对真正传输的数据进行加密。

3)     HASH算法：MD5，SHA1，SHA256，验证数据的完整性。

5.     HTTP与HTTPS的区别：

1)     https协议需要申请证书。

2)     http是超文本传输协议，明文传输；https使用的是具有安全性的SSL加密传输协议。

3)     http端口80,；https端口443。

4)     http连接简单无状态；https由SSL+HTTP协议构件的可进行加密传输、身份验证的网络协议。

### 56 假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题？

1、 引言

QoS(Quality of Service，服务质量)控制技术作为下一代网络的核心技术之一，越来越成为计算机网络中研究与开发的热点问题。QoS控制的基本目标是为Internet应用提供性能保证和区分服务。随着Internet上Web应用的爆炸性增长和电子商务的飞速发展，如何为用户提供满意的服务性能保证成了一个新的研究课题，由于传统的Web服务器无法为Web应用提供服务区分和性能保证，因此，随着QoS技术研究和应用的深入，Web QoS作为QoS技术的一个新的重要研究领域应运而生。这种面向Web客户和Http请求的技术，是属于应用层的QoS，是企业对企业交易的一个重要条件，也是Web服务器中的一个必要元素，它度量的是用户在与Web站点进行交互时所感受到的服务性能，如：交易时间，交易的可靠性等等。在Web服务应用程序的实现中需要满足各种QoS属性，比如：可用性、可访问性、完整性、性能、可靠性、常规性和安全性等[1]

。

目前，Web QoS控制研究已经越来越受到国内外学界和业界的重视并取得了一定的成果。概括起来，实现Web QoS控制技术可以分为这样几大类：Web请求的分类机制、Web服务器软件的QoS控制机制、操作系统的Web QoS控制机制以及Web服务器集群的QoS控制机制。本文主要研究一个Web服务的性能瓶颈，并对其进行分析，提出相应的策略。

2、 Http对Web QoS的影响及对策

由于底层消息传递和传输协议的局限性，Web服务会遇到性能瓶颈。然而，对公众普遍接受的协议（例如 HTTP 和SOAP）的依赖却使它们成了必须承担的永久的负担。因此，本文对其加以分析并得出相应的解决措施。

2.1、 Http成为制约Web服务性能的一个瓶颈

Http是一种尽力而为的传输服务，是一个无状态的数据转发机制，它不保证数据包会被传输到目的地，而且不保证数据包到达的顺序。因而产生一个可怕的问题：在没有可用带宽的情况下，数据包就会被简单地丢弃。这样，许多服务级别高的付费用户就无得到服务级别的保证。比如：企业和企业之间的交易就比一般的浏览需要更可靠的服务保证，一个股票在线交易就比普通的下载更加需要实时保障。所以，随着运行在网络上的用户和数据量的增加和电子商务的飞速发展，在有限带宽和网络资源的条件下，Http显然是一个制约Web服务性能的一个瓶颈，Http协议无法为Web服务器提供区分服务和性能保证。

虽然可以使用新设计的协议如“可靠 HTTP”（Reliable HTTP，HTTPR）、“块可扩展交换协议”（Blocks Extensible Exchange Protocol，BEEP）和“直接因特网消息封装”（Direct Internet Message Encapsulation，DIME） [2]

,但这些用于 Web 服务传输的新协议（如 HTTPR 和 BEEP）的广泛采用还需要一些时间。因此，使用 Web 服务的应用程序设计人员在设计系统时应该理解 Web 服务的性能问题，比如延迟和可用性。下面给出了一些改善 Web 服务性能的策略来解决这个问题。

2.2四解决策略

2 .1.1使用异步消息队列

习惯上，许多应用程序使用同步消息传递，当在自己的计算机上运行应用程序时，同步的消息传递是没什么问题的；组件通信的延迟以几毫秒计。但是，对于 Web 服务来说，它们是通过因特网进行通信，这意味着延迟要以几十、几百甚至几千微秒计。

依赖远程 Web 服务的应用程序可以使用消息排队来改善可靠性，但要以响应时间为代价。 一个企业内的应用程序和Web 服务可以使用消息排队如“Java 消息传递服务”（Java Messaging Service，JMS）或 IBM MQSeries 进行 Web服务调用[2]

。企业消息传递为整个企业内的关键数据异步交换提供可靠、灵活的服务。 消息队列有两个主要优势：

(1)它是异步的：一个消息传递服务提供者可以在消息到达时向请求者传递消息，请求者不必为接收消息而请求消息。

(2)它是可靠的：消息传递服务可以确保一条消息被传递一次，且仅传递一次。

将来，因特网上的发布和订阅消息传递系统如 alphaWorks 上的 Utility Services 包可以用于 Web 服务调用[3]

。

2.1.2对到来的Http请求进行分类

实现Web QoS的一个重要环节是对到来的Http请求进行分类，在传统的Web服务中 HTTP的请求是直接由工作进程侦听的,它对于所有的请求均采用先到先服务的处理方式，显然这种方法忽略了客户的优先级别。现在通常使用一种用连接管理模块，可以对不同的请求进行分类，并设定其优先级，这样就可以实现对不同用户的差别服务。请求分类是实现Web差别服务的核心模块，它对不同的请求设定不同的优先级并将其放入相应的队列。分类的方法有很多，可以根据实际的需要进行选择，目前常用的方法可以分为以下几类：

（1） 根据不同的用户分类

要对客户进行分类，客户可以按服务器的要求输入一定的信息，服务器以此来判断客户的身份。一种方法是用客户的IP地址来区分客户，这种方法是在QoS Web服务器模型中，服务器可以对客户的服务请求设定不同的服务等级，按照预先定义的资源分配策略对客户的服务请求作出响应[4]

。这种方法具有占用带宽小，容易实现，客户等等时延小的优点。但缺点是客户端的IP地址经常会被代理服务器或者防火墙所屏蔽，因此它的应用也受到限制。

另外一种方法是基于Http Cookie的分类，它是将Web Cookie嵌入Http请求内，以表明客户所属的类别。HTTP请求中的Cookie是可以由服务器发送给浏览器的唯一标识符，它可以内嵌在HTTP请求中，用来表示不同的服务级别。服务商可以给某个特定的服务提供一个永久的Cookie以供给用户使用。这样，就可以为付费用户和免费用户设置不同的优先级，利用cookie 来识别用户信息,记录下用户在一段时间内的访问倾向,例如经常浏览哪一类的网页,或者常常购买哪一类的商品;并将有相同兴趣的用户归类分组。当用户访问网站时,服务器可以根据他们的兴趣倾向推荐他们可能接受的网页,而且可以预测用户将来可能的行为,以此来提高服务质量。

和基于Http Cookie的分类相似，基于浏览器plug-in的分类是将特定的标识符嵌入Http请求内以表明客户所属的类别。浏览器中的plug-in插件是内嵌在客户端的又一种标识方法，购买了某种优先级服务的用户可以从服务器端下载特定的插件，把它放入HTTP的请求中。这些方法可以对客户进行分组，从而对高级别的客户提供更好的Web QoS保证。

以上这些方法虽然很准确，但是比较繁琐，增加了客户的等待时延，同时也为判断客户的身份占用了额外的带宽。

（2） 根据请求的目标分类

根据请求的目标所特有的一些属性和特点，我们可以对客户进行分类。我们都知道，由于URL请求类型或文件名路径可以区分不同请求，以及若多个站点访问同一Web服务器节点的时候，服务器可以识别其IP。所以可以用基于URL请求类型或请求的文件路径的分类和基于目标IP或端口的分类两种方法来实现Web QoS控制。这种分类方法同样也可以为高级别用户提供优先服务。较好地消除由于在网络少量较大时，由Http协议而产生的瓶颈[5]

。

不同的URL请求类型或者不同的请求文件路径表明了请求的不同的重要程度，在这种情况下请求的重要性与发送者是无关的。它侧重于对于不同的请求动作和请求目的进行分类。按照其重要程度，一般可以将请求分为紧急的 (Mission-critical) 、对时延敏感(delay-sensitive)的、和尽力而为(best-effort)传送3种。 例如,在电子商务应用中，购买商品的用户显然应当比仅仅浏览的用户获取更高的优先级目的地的IP地址，如果在同一个网络节点中架设多个Web站点，那么就要用目的地址来区分请求的重要性。

（3） 利用其他网络参数

Web服务器也可以将传输和路由中对数据包分级的参数集成到自己的连接管理模块中。例如,在因特网的差分服务体系中,IP数据报头的TOS域常被用作包的优先级标识,Web服务器可直接从IP头取出TOS域中的数据作为请求的优先级[5]

。这样，也可以达到高级别用户服务的保证。

2.1.3、通过备份使服务平滑降级

在每个服务器上存储多份不同质量的Web 内容。当服务器超载时,可以使服务器有选择地为客户提供适宜质量的Web 内容,即以体面的方式为低优先级客户提供平滑的服务降级[6]

,而保证高优先级的客户不会受到降级服务。这样就可以在服务器过载的情况下自适应地提供连续的内容降级服务而不是简单地拒绝请求,从而能够更好地为用户提供Web QoS。

2.1.4、其它的保证Web Qos的方法

除了上文中所述的方法之外，我们还可以提供主动Web服务 QoS 的方法，如服务请求的高速缓存和负载平衡，服务提供者可以主动向服务请求者提供很高的 QoS。在Web服务器级别上和 Web 应用程序服务器级别上都可以完成高速缓存和负载平衡。负载平衡区分各种类型通信的优先次序，并确保适当地按照每个请求所表现出的价值对待它。

3、 结论和展望

随着Web 服务的广泛扩大,服务质量QoS 将变成一个判定服务提供者是否成功的重要因素。QoS 决定服务的可用性和实用性, 本文所列出的消除Http对于Web服务性能瓶颈的方法针对性强，易于实现。随着Web QoS研究技术的发展，基于中间件技术和Web服务器集群的QoS控制机制必将给Web提供更可靠的服务保障，有待我们进行更深入的研究。

### 57 如果出现大面积并发，在不增加服务器的基础上，如何解决服务器响应不及时问?

### 58 如果有一个特别大的访问量，到数据库上，怎么做优化？

（DB设计，DBIO，SQL优化，Java优化）

一、数据库结构的设计

如果不能设计一个合理的数据库模型，不仅会增加客户端和服务器段程序的编程和维护的难度，而且将会影响系统实际运行的性能。所以，在一个系统开始实施之前，完备的数据库模型的设计是必须的。

在一个系统分析、设计阶段，因为数据量较小，负荷较低。我们往往只注意到功能的实现，而很难注意到性能的薄弱之处，等到系统投入实际运行一段时间后，才发现系统的性能在降低，这时再来考虑提高系统性能则要花费更多的人力物力，而整个系统也不可避免的形成了一个打补丁工程。

所以在考虑整个系统的流程的时候，我们必须要考虑，在高并发大数据量的访问情况下，我们的系统会不会出现极端的情况。（例如：对外统计系统在7月16日出现的数据异常的情况，并发大数据量的的访问造成，数据库的响应时间不能跟上数据刷新的速度造成。具体情况是：在日期临界时（00：00：00），判断数据库中是否有当前日期的记录，没有则插入一条当前日期的记录。在低并发访问的情况下，不会发生问题，但是当日期临界时的访问量相当大的时候，在做这一判断的时候，会出现多次条件成立，则数据库里会被插入多条当前日期的记录，从而造成数据错误。），数据库的模型确定下来之后，我们有必要做一个系统内数据流向图，分析可能出现的瓶颈。

为了保证数据库的一致性和完整性，在逻辑设计的时候往往会设计过多的表间关联，尽可能的降低数据的冗余。（例如用户表的地区，我们可以把地区另外存放到一个地区表中）如果数据冗余低，数据的完整性容易得到保证，提高了数据吞吐速度，保证了数据的完整性，清楚地表达数据元素之间的关系。而对于多表之间的关联查询（尤其是大数据表）时，其性能将会降低，同时也提高了客户端程序的编程难度，因此，物理设计需折衷考虑，根据业务规则，确定对关联表的数据量大小、数据项的访问频度，对此类数据表频繁的关联查询应适当提高数据冗余设计但增加了表间连接查询的操作，也使得程序的变得复杂，为了提高系统的响应时间，合理的数据冗余也是必要的。设计人员在设计阶段应根据系统操作的类型、频度加以均衡考虑。

另外，最好不要用自增属性字段作为主键与子表关联。不便于系统的迁移和数据恢复。对外统计系统映射关系丢失（\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*）。

原来的表格必须可以通过由它分离出去的表格重新构建。使用这个规定的好处是，你可以确保不会在分离的表格中引入多余的列，所有你创建的表格结构都与它们的实际需要一样大。应用这条规定是一个好习惯，不过除非你要处理一个非常大型的数据，否则你将不需要用到它。（例如一个通行证系统，我可以将USERID，USERNAME，USERPASSWORD，单独出来作个表，再把USERID作为其他表的外键）

表的设计具体注意的问题：

1、数据行的长度不要超过8020字节，如果超过这个长度的话在物理页中这条数据会占用两行从而造成存储碎片，降低查询效率。

2、能够用数字类型的字段尽量选择数字类型而不用字符串类型的（电话号码），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回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。

3、对于不可变字符类型char和可变字符类型varchar 都是8000字节,char查询快，但是耗存储空间，varchar查询相对慢一些但是节省存储空间。在设计字段的时候可以灵活选择，例如用户名、密码等长度变化不大的字段可以选择CHAR，对于评论等长度变化大的字段可以选择VARCHAR。

4、字段的长度在最大限度的满足可能的需要的前提下，应该尽可能的设得短一些，这样可以提高查询的效率，而且在建立索引的时候也可以减少资源的消耗。

二、查询的优化

保证在实现功能的基础上，尽量减少对数据库的访问次数；通过搜索参数，尽量减少对表的访问行数,最小化结果集，从而减轻网络负担；能够分开的操作尽量分开处理，提高每次的响应速度；在数据窗口使用SQL时，尽量把使用的索引放在选择的首列；算法的结构尽量简单；在查询时，不要过多地使用通配符如SELECT \* FROM T1语句，要用到几列就选择几列如：SELECT COL1,COL2 FROM T1；在可能的情况下尽量限制尽量结果集行数如：SELECT TOP 300 COL1,COL2,COL3 FROM T1,因为某些情况下用户是不需要那么多的数据的。

在没有建索引的情况下，数据库查找某一条数据，就必须进行全表扫描了，对所有数据进行一次遍历，查找出符合条件的记录。在数据量比较小的情况下，也许看不出明显的差别，但是当数据量大的情况下，这种情况就是极为糟糕的了。

SQL语句在SQL SERVER中是如何执行的，他们担心自己所写的SQL语句会被SQL SERVER误解。比如：

select \* from table1 where name='zhangsan' and tID > 10000

和执行:

select \* from table1 where tID > 10000 and name='zhangsan'

一些人不知道以上两条语句的执行效率是否一样，因为如果简单的从语句先后上看，这两个语句的确是不一样，如果tID是一个聚合索引，那么后一句仅仅从表的10000条以后的记录中查找就行了；而前一句则要先从全表中查找看有几个name='zhangsan'的，而后再根据限制条件条件tID>10000来提出查询结果。

事实上，这样的担心是不必要的。SQL SERVER中有一个“查询分析优化器”，它可以计算出where子句中的搜索条件并确定哪个索引能缩小表扫描的搜索空间，也就是说，它能实现自动优化。虽然查询优化器可以根据where子句自动的进行查询优化，但有时查询优化器就会不按照您的本意进行快速查询。

在查询分析阶段，查询优化器查看查询的每个阶段并决定限制需要扫描的数据量是否有用。如果一个阶段可以被用作一个扫描参数（SARG），那么就称之为可优化的，并且可以利用索引快速获得所需数据。

SARG的定义：用于限制搜索的一个操作，因为它通常是指一个特定的匹配，一个值的范围内的匹配或者两个以上条件的AND连接。形式如下：

列名 操作符 <常数 或 变量> 或 <常数 或 变量> 操作符 列名

列名可以出现在操作符的一边，而常数或变量出现在操作符的另一边。如：

Name=’张三’

价格>5000

5000<价格

Name=’张三’ and 价格>5000

如果一个表达式不能满足SARG的形式，那它就无法限制搜索的范围了，也就是SQL SERVER必须对每一行都判断它是否满足WHERE子句中的所有条件。所以一个索引对于不满足SARG形式的表达式来说是无用的。

所以，优化查询最重要的就是，尽量使语句符合查询优化器的规则避免全表扫描而使用索引查询。

具体要注意的：

1.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

2.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。优化器将无法通过索引来确定将要命中的行数,因此需要搜索该表的所有行。

3.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

4.in 和 not in 也要慎用，因为IN会使系统无法使用索引,而只能直接搜索表中的数据。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

5.尽量避免在索引过的字符数据中，使用非打头字母搜索。这也使得引擎无法利用索引。

见如下例子：

SELECT \* FROM T1 WHERE NAME LIKE ‘%L%’

SELECT \* FROM T1 WHERE SUBSTING(NAME,2,1)=’L’

SELECT \* FROM T1 WHERE NAME LIKE ‘L%’

即使NAME字段建有索引，前两个查询依然无法利用索引完成加快操作，引擎不得不对全表所有数据逐条操作来完成任务。而第三个查询能够使用索引来加快操作。

6.必要时强制查询优化器使用某个索引，如在 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

7.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 \* FROM T1 WHERE F1/2=100

应改为:

SELECT \* FROM T1 WHERE F1=100\*2

SELECT \* FROM RECORD WHERE SUBSTRING(CARD\_NO,1,4)=’5378’

应改为:

SELECT \* FROM RECORD WHERE CARD\_NO LIKE ‘5378%’

SELECT member\_number, first\_name, last\_name FROM members

WHERE DATEDIFF(yy,datofbirth,GETDATE()) > 21

应改为:

SELECT member\_number, first\_name, last\_name FROM members

WHERE dateofbirth < DATEADD(yy,-21,GETDATE())

即：任何对列的操作都将导致表扫描，它包括数据库函数、计算表达式等等，查询时要尽可能将操作移至等号右边。

8.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'

9.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。

10.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

11.很多时候用 exists是一个好的选择：

elect 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)

SELECT SUM(T1.C1)FROM T1 WHERE(

(SELECT COUNT(\*)FROM T2 WHERE T2.C2=T1.C2>0)

SELECT SUM(T1.C1) FROM T1WHERE EXISTS(

SELECT \* FROM T2 WHERE T2.C2=T1.C2)

两者产生相同的结果，但是后者的效率显然要高于前者。因为后者不会产生大量锁定的表扫描或是索引扫描。

如果你想校验表里是否存在某条纪录，不要用count(\*)那样效率很低，而且浪费服务器资源。可以用EXISTS代替。如：

IF (SELECT COUNT(\*) FROM table\_name WHERE column\_name = 'xxx')

可以写成：

IF EXISTS (SELECT \* FROM table\_name WHERE column\_name = 'xxx')

经常需要写一个T\_SQL语句比较一个父结果集和子结果集，从而找到是否存在在父结果集中有而在子结果集中没有的记录，如：

SELECT a.hdr\_key FROM hdr\_tbl a---- tbl a 表示tbl用别名a代替

WHERE NOT EXISTS (SELECT \* FROM dtl\_tbl b WHERE a.hdr\_key = b.hdr\_key)

SELECT a.hdr\_key FROM hdr\_tbl a

LEFT JOIN dtl\_tbl b ON a.hdr\_key = b.hdr\_key WHERE b.hdr\_key IS NULL

SELECT hdr\_key FROM hdr\_tbl

WHERE hdr\_key NOT IN (SELECT hdr\_key FROM dtl\_tbl)

三种写法都可以得到同样正确的结果，但是效率依次降低。

12.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）。

13.避免频繁创建和删除临时表，以减少系统表资源的消耗。

14.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。

15.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。

16.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。

17.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

18.尽量避免大事务操作，提高系统并发能力。

19.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。

20. 避免使用不兼容的数据类型。例如float和int、char和varchar、binary和varbinary是不兼容的。数据类型的不兼容可能使优化器无法执行一些本来可以进行的优化操作。例如:

SELECT name FROM employee WHERE salary > 60000

在这条语句中,如salary字段是money型的,则优化器很难对其进行优化,因为60000是个整型数。我们应当在编程时将整型转化成为钱币型,而不要等到运行时转化。

21.充分利用连接条件，在某种情况下，两个表之间可能不只一个的连接条件，这时在 WHERE 子句中将连接条件完整的写上，有可能大大提高查询速度。

例：

SELECT SUM(A.AMOUNT) FROM ACCOUNT A,CARD B WHERE A.CARD\_NO = B.CARD\_NO

SELECT SUM(A.AMOUNT) FROM ACCOUNT A,CARD B WHERE A.CARD\_NO = B.CARD\_NO AND A.ACCOUNT\_NO=B.ACCOUNT\_NO

第二句将比第一句执行快得多。

22、使用视图加速查询

把表的一个子集进行排序并创建视图，有时能加速查询。它有助于避免多重排序 操作，而且在其他方面还能简化优化器的工作。例如：

SELECT cust.name，rcvbles.balance，……other columns

FROM cust，rcvbles

WHERE cust.customer\_id = rcvlbes.customer\_id

AND rcvblls.balance>0

AND cust.postcode>“98000”

ORDER BY cust.name

如果这个查询要被执行多次而不止一次，可以把所有未付款的客户找出来放在一个视图中，并按客户的名字进行排序：

CREATE VIEW DBO.V\_CUST\_RCVLBES

AS

SELECT cust.name，rcvbles.balance，……other columns

FROM cust，rcvbles

WHERE cust.customer\_id = rcvlbes.customer\_id

AND rcvblls.balance>0

ORDER BY cust.name

然后以下面的方式在视图中查询：

SELECT ＊ FROM V\_CUST\_RCVLBES

WHERE postcode>“98000”

视图中的行要比主表中的行少，而且物理顺序就是所要求的顺序，减少了磁盘I/O，所以查询工作量可以得到大幅减少。

23、能用DISTINCT的就不用GROUP BY

SELECT OrderID FROM Details WHERE UnitPrice > 10 GROUP BY OrderID

可改为：

SELECT DISTINCT OrderID FROM Details WHERE UnitPrice > 10

24.能用UNION ALL就不要用UNION

UNION ALL不执行SELECT DISTINCT函数，这样就会减少很多不必要的资源

35.尽量不要用SELECT INTO语句。

SELECT INOT 语句会导致表锁定，阻止其他用户访问该表。

上面我们提到的是一些基本的提高查询速度的注意事项,但是在更多的情况下,往往需要反复试验比较不同的语句以得到最佳方案。最好的方法当然是测试，看实现相同功能的SQL语句哪个执行时间最少，但是数据库中如果数据量很少，是比较不出来的，这时可以用查看执行计划，即：把实现相同功能的多条SQL语句考到查询分析器，按CTRL+L看查所利用的索引，表扫描次数（这两个对性能影响最大），总体上看询成本百分比即可。

三、算法的优化

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。.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

游标提供了对特定集合中逐行扫描的手段，一般使用游标逐行遍历数据，根据取出的数据不同条件进行不同的操作。尤其对多表和大表定义的游标（大的数据集合）循环很容易使程序进入一个漫长的等特甚至死机。

在有些场合，有时也非得使用游标，此时也可考虑将符合条件的数据行转入临时表中，再对临时表定义游标进行操作，可时性能得到明显提高。

（例如：对内统计第一版）

封装存储过程

四、建立高效的索引

创建索引一般有以下两个目的：维护被索引列的唯一性和提供快速访问表中数据的策略。大型数据库有两种索引即簇索引和非簇索引，一个没有簇索引的表是按堆结构存储数据，所有的数据均添加在表的尾部，而建立了簇索引的表，其数据在物理上会按照簇索引键的顺序存储，一个表只允许有一个簇索引，因此，根据B树结构，可以理解添加任何一种索引均能提高按索引列查询的速度，但会降低插入、更新、删除操作的性能，尤其是当填充因子（Fill Factor）较大时。所以对索引较多的表进行频繁的插入、更新、删除操作，建表和索引时因设置较小的填充因子，以便在各数据页中留下较多的自由空间，减少页分割及重新组织的工作。

索引是从数据库中获取数据的最高效方式之一。95% 的数据库性能问题都可以采用索引技术得到解决。作为一条规则，我通常对逻辑主键使用唯一的成组索引，对系统键（作为存储过程）采用唯一的非成组索引，对任何外键列[字段]采用非成组索引。不过，索引就象是盐，太多了菜就咸了。你得考虑数据库的空间有多大，表如何进行访问，还有这些访问是否主要用作读写。

实际上，您可以把索引理解为一种特殊的目录。微软的SQL SERVER提供了两种索引：聚集索引（clustered index，也称聚类索引、簇集索引）和非聚集索引（nonclustered index，也称非聚类索引、非簇集索引）。下面，我们举例来说明一下聚集索引和非聚集索引的区别：

其实，我们的汉语字典的正文本身就是一个聚集索引。比如，我们要查“安”字，就会很自然地翻开字典的前几页，因为“安”的拼音是“an”，而按照拼音排序汉字的字典是以英文字母“a”开头并以“z”结尾的，那么“安”字就自然地排在字典的前部。如果您翻完了所有以“a”开头的部分仍然找不到这个字，那么就说明您的字典中没有这个字；同样的，如果查“张”字，那您也会将您的字典翻到最后部分，因为“张”的拼音是“zhang”。也就是说，字典的正文部分本身就是一个目录，您不需要再去查其他目录来找到您需要找的内容。

我们把这种正文内容本身就是一种按照一定规则排列的目录称为“聚集索引”。

如果您认识某个字，您可以快速地从自动中查到这个字。但您也可能会遇到您不认识的字，不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而需要去根据“偏旁部首”查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。但您结合“部首目录”和“检字表”而查到的字的排序并不是真正的正文的排序方法，比如您查“张”字，我们可以看到在查部首之后的检字表中“张”的页码是672页，检字表中“张”的上面是“驰”字，但页码却是63页，“张”的下面是“弩”字，页面是390页。很显然，这些字并不是真正的分别位于“张”字的上下方，现在您看到的连续的“驰、张、弩”三字实际上就是他们在非聚集索引中的排序，是字典正文中的字在非聚集索引中的映射。我们可以通过这种方式来找到您所需要的字，但它需要两个过程，先找到目录中的结果，然后再翻到您所需要的页码。

我们把这种目录纯粹是目录，正文纯粹是正文的排序方式称为“非聚集索引”。

进一步引申一下，我们可以很容易的理解：每个表只能有一个聚集索引，因为目录只能按照一种方法进行排序。

（一）何时使用聚集索引或非聚集索引

下面的表总结了何时使用聚集索引或非聚集索引（很重要）。

动作描述 使用聚集索引 使用非聚集索引

列经常被分组排序 应 应

返回某范围内的数据 应 不应

一个或极少不同值 不应 不应

小数目的不同值 应 不应

大数目的不同值 不应 应

频繁更新的列 不应 应

外键列 应 应

主键列 应 应

频繁修改索引列 不应 应

事实上，我们可以通过前面聚集索引和非聚集索引的定义的例子来理解上表。如：返回某范围内的数据一项。比如您的某个表有一个时间列，恰好您把聚合索引建立在了该列，这时您查询2004年1月1日至2004年10月1日之间的全部数据时，这个速度就将是很快的，因为您的这本字典正文是按日期进行排序的，聚类索引只需要找到要检索的所有数据中的开头和结尾数据即可；而不像非聚集索引，必须先查到目录中查到每一项数据对应的页码，然后再根据页码查到具体内容。

（二）结合实际，谈索引使用的误区

理论的目的是应用。虽然我们刚才列出了何时应使用聚集索引或非聚集索引，但在实践中以上规则却很容易被忽视或不能根据实际情况进行综合分析。下面我们将根据在实践中遇到的实际问题来谈一下索引使用的误区，以便于大家掌握索引建立的方法。

1、主键就是聚集索引

这种想法笔者认为是极端错误的，是对聚集索引的一种浪费。虽然SQL SERVER默认是在主键上建立聚集索引的。

通常，我们会在每个表中都建立一个ID列，以区分每条数据，并且这个ID列是自动增大的，步长一般为1。我们的这个办公自动化的实例中的列Gid就是如此。此时，如果我们将这个列设为主键，SQL SERVER会将此列默认为聚集索引。这样做有好处，就是可以让您的数据在数据库中按照ID进行物理排序，但笔者认为这样做意义不大。

显而易见，聚集索引的优势是很明显的，而每个表中只能有一个聚集索引的规则，这使得聚集索引变得更加珍贵。

从我们前面谈到的聚集索引的定义我们可以看出，使用聚集索引的最大好处就是能够根据查询要求，迅速缩小查询范围，避免全表扫描。在实际应用中，因为ID号是自动生成的，我们并不知道每条记录的ID号，所以我们很难在实践中用ID号来进行查询。这就使让ID号这个主键作为聚集索引成为一种资源浪费。其次，让每个ID号都不同的字段作为聚集索引也不符合“大数目的不同值情况下不应建立聚合索引”规则；当然，这种情况只是针对用户经常修改记录内容，特别是索引项的时候会负作用，但对于查询速度并没有影响。

在办公自动化系统中，无论是系统首页显示的需要用户签收的文件、会议还是用户进行文件查询等任何情况下进行数据查询都离不开字段的是“日期”还有用户本身的“用户名”。

通常，办公自动化的首页会显示每个用户尚未签收的文件或会议。虽然我们的where语句可以仅仅限制当前用户尚未签收的情况，但如果您的系统已建立了很长时间，并且数据量很大，那么，每次每个用户打开首页的时候都进行一次全表扫描，这样做意义是不大的，绝大多数的用户1个月前的文件都已经浏览过了，这样做只能徒增数据库的开销而已。事实上，我们完全可以让用户打开系统首页时，数据库仅仅查询这个用户近3个月来未阅览的文件，通过“日期”这个字段来限制表扫描，提高查询速度。如果您的办公自动化系统已经建立的2年，那么您的首页显示速度理论上将是原来速度8倍，甚至更快。

2、只要建立索引就能显著提高查询速度

事实上，我们可以发现上面的例子中，第2、3条语句完全相同，且建立索引的字段也相同；不同的仅是前者在fariqi字段上建立的是非聚合索引，后者在此字段上建立的是聚合索引，但查询速度却有着天壤之别。所以，并非是在任何字段上简单地建立索引就能提高查询速度。

从建表的语句中，我们可以看到这个有着1000万数据的表中fariqi字段有5003个不同记录。在此字段上建立聚合索引是再合适不过了。在现实中，我们每天都会发几个文件，这几个文件的发文日期就相同，这完全符合建立聚集索引要求的：“既不能绝大多数都相同，又不能只有极少数相同”的规则。由此看来，我们建立“适当”的聚合索引对于我们提高查询速度是非常重要的。

3、把所有需要提高查询速度的字段都加进聚集索引，以提高查询速度

上面已经谈到：在进行数据查询时都离不开字段的是“日期”还有用户本身的“用户名”。既然这两个字段都是如此的重要，我们可以把他们合并起来，建立一个复合索引（compound index）。

很多人认为只要把任何字段加进聚集索引，就能提高查询速度，也有人感到迷惑：如果把复合的聚集索引字段分开查询，那么查询速度会减慢吗？带着这个问题，我们来看一下以下的查询速度（结果集都是25万条数据）：（日期列fariqi首先排在复合聚集索引的起始列，用户名neibuyonghu排在后列）

我们可以看到如果仅用聚集索引的起始列作为查询条件和同时用到复合聚集索引的全部列的查询速度是几乎一样的，甚至比用上全部的复合索引列还要略快（在查询结果集数目一样的情况下）；而如果仅用复合聚集索引的非起始列作为查询条件的话，这个索引是不起任何作用的。当然，语句1、2的查询速度一样是因为查询的条目数一样，如果复合索引的所有列都用上，而且查询结果少的话，这样就会形成“索引覆盖”，因而性能可以达到最优。同时，请记住：无论您是否经常使用聚合索引的其他列，但其前导列一定要是使用最频繁的列。

（三）其他注意事项

“水可载舟，亦可覆舟”，索引也一样。索引有助于提高检索性能，但过多或不当的索引也会导致系统低效。因为用户在表中每加进一个索引，数据库就要做更多的工作。过多的索引甚至会导致索引碎片。

所以说，我们要建立一个“适当”的索引体系，特别是对聚合索引的创建，更应精益求精，以使您的数据库能得到高性能的发挥

### 59 集群如何同步会话状态?

在做了web集群后，你肯定会首先考虑session同步问题，因为通过负载均衡后，同一个IP访问同一个页面会被分配到不同的服务器上，如果session不同步的话，一个登录用户，一会是登录状态，一会又不是登录状态。所以本文就根据这种情况给出三种不同的方法来解决这个问题：

一，利用数据库同步session

在做多服务器session同步时我没有用这种方法，如果非要用这种方法的话，我想过二种方法：

1，用一个低端电脑建个数据库专门存放web服务器的session，或者，把这个专门的数据库建在文件服务器上，用户访问web服务器时，会去这个专门的数据库check一下session的情况，以达到session同步的目的。

2，这种方法是把存放session的表和其他数据库表放在一起，如果mysql也做了集群了话，每个mysql节点都要有这张表，并且这张session表的数据表要实时同步。

说明：用数据库来同步session，会加大数据库的负担，数据库本来就是容易产生瓶颈的地方，如果把session还放到数据库里面，无疑是雪上加霜。上面的二种方法，第一点方法较好，把放session的表独立开来，减轻了真正数据库的负担

二，利用cookie同步session

session是文件的形势存放在服务器端的，cookie是文件的形势存在客户端的，怎么实现同步呢？方法很简单，就是把用户访问页面产生的session放到cookie里面，就是以cookie为中转站。你访问web服务器A，产生了session把它放到cookie里面了，你访问被分配到web服务器B，这个时候，web服务器B先判断服务器有没有这个session，如果没有，在去看看客户端的cookie里面有没有这个session，如果也没有，说明session真的不存，如果cookie里面有，就把cookie里面的sessoin同步到web服务器B，这样就可以实现session的同步了。

说明：这种方法实现起来简单，方便，也不会加大数据库的负担，但是如果客户端把cookie禁掉了的话，那么session就无从同步了，这样会给网站带来损失；cookie的安全性不高，虽然它已经加了密，但是还是可以伪造的。

三，利用memcache同步session

memcache可以做分布式，如果没有这功能，他也不能用来做session同步。他可以把web服务器中的内存组合起来，成为一个"内存池"，不管是哪个服务器产生的sessoin都可以放到这个"内存池"中，其他的都可以使用。

优点：以这种方式来同步session，不会加大数据库的负担，并且安全性比用cookie大大的提高，把session放到内存里面，比从文件中读取要快很多。

缺点：memcache把内存分成很多种规格的存储块，有块就有大小，这种方式也就决定了，memcache不能完全利用内存，会产生内存碎片，如果存储块不足，还会产生内存溢出。

四，总结

上面三种方法都是可行的

第一种方法，最影响系统速度的那种，不推荐使用；

第二种方法，效果不错，不过安全隐患一样的存在；

第三种方法，个人觉得第三种方法是最好的，推荐大家使用

### 60 [负载均衡](http://www.codeceo.com/article/balanced-algorithm.html" \t "http://blog.csdn.net/u014352080/article/details/_blank)的原理?

不能狭义地理解为分配给所有实际服务器一样多的工作量，因为多台服务器的承载能力各不相同，这可能体现在硬件配置、网络带宽的差异，也可能因为某台服务器身兼多职，我们所说的“均衡”，也就是希望所有服务器都不要过载，

并且能够最大程序地发挥作用。

一、http重定向

当http代理（比如浏览器）向web服务器请求某个URL后，web服务器可以通过http响应头信息中的Location标记来返回一个新的URL。这意味着HTTP代理需要继续请求这个新的URL，完成自动跳转。

性能缺陷：

1、吞吐率限制

主站点服务器的吞吐率平均分配到了被转移的服务器。现假设使用RR（Round Robin）调度策略，子服务器的最大吞吐率为1000reqs/s，那么主服务器的吞吐率要达到3000reqs/s才能完全发挥三台子服务器的作用，那么如果有100台子服务器，那么主服务器的吞吐率可想而知得有大？相反，如果主服务的最大吞吐率为6000reqs/s，那么平均分配到子服务器的吞吐率为2000reqs/s，而现子服务器的最大吞吐率为1000reqs/s，因此就得增加子服务器的数量，增加到6个才能满足。

2、重定向访问深度不同

有的重定向一个静态页面，有的重定向相比复杂的动态页面，那么实际服务器的负载差异是不可预料的，而主站服务器却一无所知。因此整站使用重定向方法做负载均衡不太好。

我们需要权衡转移请求的开销和处理实际请求的开销，前者相对于后者越小，那么重定向的意义就越大，例如下载。你可以去很多镜像下载网站试下，会发现基本下载都使用了Location做了重定向。

二、DNS负载均衡

DNS负责提供域名解析服务，当访问某个站点时，实际上首先需要通过该站点域名的DNS服务器来获取域名指向的IP地址，在这一过程中，DNS服务器完成了域名到IP地址的映射，同样，这样映射也可以是一对多的，这时候，DNS服务器便充当了负载均衡调度器，它就像http重定向转换策略一样，将用户的请求分散到多台服务器上，但是它的实现机制完全不同。

使用dig命令来看下"baidu"的DNS设置

可见baidu拥有三个A记录

相比http重定向，基于DNS的负载均衡完全节省了所谓的主站点，或者说DNS服务器已经充当了主站点的职能。但不同的是，作为调度器，DNS服务器本身的性能几乎不用担心。因为DNS记录可以被用户浏览器或者互联网接入服务商的各级DNS服务器缓存，只有当缓存过期后才会重新向域名的DNS服务器请求解析。也说是DNS不存在http的吞吐率限制，理论上可以无限增加实际服务器的数量。

特性:

1、可以根据用户IP来进行智能解析。DNS服务器可以在所有可用的A记录中寻找离用记最近的一台服务器。

2、动态DNS：在每次IP地址变更时，及时更新DNS服务器。当然，因为缓存，一定的延迟不可避免。

不足：

1、没有用户能直接看到DNS解析到了哪一台实际服务器，加服务器运维人员的调试带来了不便。

2、策略的局限性。例如你无法将HTTP请求的上下文引入到调度策略中，而在前面介绍的基于HTTP重定向的负载均衡系统中，调度器工作在HTTP层面，它可以充分理解HTTP请求后根据站点的应用逻辑来设计调度策略，比如根据请求不同的URL来进行合理的过滤和转移。

3、如果要根据实际服务器的实时负载差异来调整调度策略，这需要DNS服务器在每次解析操作时分析各服务器的健康状态，对于DNS服务器来说，这种自定义开发存在较高的门槛，更何况大多数站点只是使用第三方DNS服务。

4、DNS记录缓存，各级节点的DNS服务器不同程序的缓存会让你晕头转向。

5、基于以上几点，DNS服务器并不能很好地完成工作量均衡分配，最后，是否选择基于DNS的负载均衡方式完全取决于你的需要。

三、反向代理负载均衡

这个肯定大家都有所接触，因为几乎所有主流的Web服务器都热衷于支持基于反向代理的负载均衡。它的核心工作就是转发HTTP请求。

相比前面的HTTP重定向和DNS解析，反向代理的调度器扮演的是用户和实际服务器中间人的角色：

1、任何对于实际服务器的HTTP请求都必须经过调度器

2、调度器必须等待实际服务器的HTTP响应，并将它反馈给用户（前两种方式不需要经过调度反馈，是实际服务器直接发送给用户）

特性：

1、调度策略丰富。例如可以为不同的实际服务器设置不同的权重，以达到能者多劳的效果。

2、对反向代理服务器的并发处理能力要求高，因为它工作在HTTP层面。

3、反向代理服务器进行转发操作本身是需要一定开销的，比如创建线程、与后端服务器建立TCP连接、接收后端服务器返回的处理结果、分析HTTP头部信息、用户空间和内核空间的频繁切换等，虽然这部分时间并不长，但是当后端服务器处理请求的时间非常短时，转发的开销就显得尤为突出。例如请求静态文件，更适合使用前面介绍的基于DNS的负载均衡方式。

4、反向代理服务器可以监控后端服务器，比如系统负载、响应时间、是否可用、TCP连接数、流量等，从而根据这些数据调整负载均衡的策略。

5、反射代理服务器可以让用户在一次会话周期内的所有请求始终转发到一台特定的后端服务器上（粘滞会话），这样的好处一是保持session的本地访问，二是防止后端服务器的动态内存缓存的资源浪费。

四、IP负载均衡(LVS-NAT)

因为反向代理服务器工作在HTTP层，其本身的开销就已经严重制约了可扩展性，从而也限制了它的性能极限。那能否在HTTP层面以下实现负载均衡呢？

NAT服务器:它工作在传输层，它可以修改发送来的IP数据包，将数据包的目标地址修改为实际服务器地址。

从Linux2.4内核开始，其内置的Neftilter模块在内核中维护着一些数据包过滤表，这些表包含了用于控制数据包过滤的规则。可喜的是，Linux提供了iptables来对过滤表进行插入、修改和删除等操作。更加令人振奋的是，Linux2.6.x内核中内置了IPVS模块，它的工作性质类型于Netfilter模块，不过它更专注于实现IP负载均衡。

想知道你的服务器内核是否已经安装了IPVS模块，可以

有输出意味着IPVS已经安装了。IPVS的管理工具是ipvsadm，它为提供了基于命令行的配置界面，可以通过它快速实现负载均衡系统。这就是大名鼎鼎的LVS(Linux Virtual Server，Linux虚拟服务器)。

1、打开调度器的数据包转发选项

echo 1 > /proc/sys/net/ipv4/ip\_forward

2、检查实际服务器是否已经将NAT服务器作为自己的默认网关，如果不是，如添加

route add default gw xx.xx.xx.xx

3、使用ipvsadm配置

ipvsadm -A -t 111.11.11.11:80 -s rr

添加一台虚拟服务器，-t 后面是服务器的外网ip和端口，-s rr是指采用简单轮询的RR调度策略（这属于静态调度策略，除此之外，LVS还提供了系列的动态调度策略，比如最小连接（LC）、带权重的最小连接（WLC），最短期望时间延迟（SED）等）

ipvsadm -a -t 111.11.11.11:80 -r 10.10.120.210:8000 -m

ipvsadm -a -t 111.11.11.11:80 -r 10.10.120.211:8000 -m

添加两台实际服务器（不需要有外网ip），-r后面是实际服务器的内网ip和端口，-m表示采用NAT方式来转发数据包

运行ipvsadm -L -n可以查看实际服务器的状态。这样就大功告成了。

实验证明使用基于NAT的负载均衡系统。作为调度器的NAT服务器可以将吞吐率提升到一个新的高度，几乎是反向代理服务器的两倍以上，这大多归功于在内核中进行请求转发的较低开销。但是一旦请求的内容过大时，不论是基于反向代理还是NAT，负载均衡的整体吞吐量都差距不大，这说明对于一睦开销较大的内容，使用简单的反向代理来搭建负载均衡系统是值考虑的。

这么强大的系统还是有它的瓶颈，那就是NAT服务器的网络带宽，包括内部网络和外部网络。当然如果你不差钱，可以去花钱去购买千兆交换机或万兆交换机，甚至负载均衡硬件设备，但如果你是个屌丝，咋办？

一个简单有效的办法就是将基于NAT的集群和前面的DNS混合使用，比如５个100Mbps出口宽带的集群，然后通过DNS来将用户请求均衡地指向这些集群，同时，你还可以利用DNS智能解析实现地域就近访问。这样的配置对于大多数业务是足够了，但是对于提供下载或视频等服务的大规模站点，NAT服务器还是不够出色。

五、直接路由(LVS-DR)

NAT是工作在网络分层模型的传输层（第四层），而直接路由是工作在数据链路层（第二层），貌似更屌些。它通过修改数据包的目标MAC地址（没有修改目标IP），将数据包转发到实际服务器上，不同的是，实际服务器的响应数据包将直接发送给客户羰，而不经过调度器。

1、网络设置

这里假设一台负载均衡调度器，两台实际服务器，购买三个外网ip，一台机一个，三台机的默认网关需要相同，最后再设置同样的ip别名，这里假设别名为10.10.120.193。这样一来，将通过10.10.120.193这个IP别名来访问调度器，你可以将站点的域名指向这个IP别名。

2、将ip别名添加到回环接口lo上

这是为了让实际服务器不要去寻找其他拥有这个IP别名的服务器，在实际服务器中运行：

另外还要防止实际服务器响应来自网络中针对IP别名的ARP广播，为此还要执行：

echo "1" > /proc/sys/net/ipv4/conf/lo/arp\_ignore

echo "2" > /proc/sys/net/ipv4/conf/lo/arp\_announce

echo "1" > /proc/sys/net/ipv4/conf/all/arp\_ignore

echo "1" > /proc/sys/net/ipv4/conf/all/arp\_announce

配置完了就可以使用ipvsadm配置LVS-DR集群了

ipvsadm -A -t 10.10.120.193:80 -s rr

ipvsadm -a -t 10.10.120.193:80 -r 10.10.120.210:8000 -g

ipvsadm -a -t 10.10.120.193:80 -r 10.10.120.211:8000 -g

-g 就意味着使用直接路由的方式转发数据包

LVS-DR 相较于LVS-NAT的最大优势在于LVS-DR不受调度器宽带的限制，例如假设三台服务器在WAN交换机出口宽带都限制为10Mbps，只要对于连接调度器和两台实际服务器的LAN交换机没有限速，那么，使用LVS-DR理论上可以达到20Mbps的最大出口宽带，因为它的实际服务器的响应数据包可以不经过调度器而直接发往用户端啊，所以它与调度器的出口宽带没有关系，只能自身的有关系。而如果使用LVS-NAT，集群只能最大使用10Mbps的宽带。所以，越是响应数据包远远超过请求数据包的服务，就越应该降低调度器转移请求的开销，也就越能提高整体的扩展能力，最终也就越依赖于WAN出口宽带。

总的来说，LVS-DR适合搭建可扩展的负载均衡系统，不论是Web服务器还是文件服务器，以及视频服务器，它都拥有出色的性能。前提是你必须为实际器购买一系列的合法IP地址。

六、IP隧道(LVS-TUN)

基于IP隧道的请求转发机制：将调度器收到的IP数据包封装在一个新的IP数据包中，转交给实际服务器，然后实际服务器的响应数据包可以直接到达用户端。目前Linux大多支持，可以用LVS来实现，称为LVS-TUN，与LVS-DR不同的是，实际服务器可以和调度器不在同一个WANt网段，调度器通过IP隧道技术来转发请求到实际服务器，所以实际服务器也必须拥有合法的IP地址。

总体来说，LVS-DR和LVS-TUN都适合响应和请求不对称的Web服务器，如何从它们中做出选择，取决于你的网络部署需要，因为LVS-TUN可以将实际服务器根据需要部署在不同的地域，并且根据就近访问的原则来转移请求，所以有类似这种需求的，就应该选择LVS-TUN。

### 61高并发情况下，我们系统是如何支撑大量的请求的?

1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。

2用jprofiler等工具找出性能瓶颈，减少额外的开销。

3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）。

4优化数据库结构，多做索引，提高查询效率。

5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。

6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）。

7解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

# 六、架构设计

### 1 进程间的通信方式和线程通信方式有什么不同？

几种进程间的通信方式

（1） 管道（pipe）：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有血缘关系的进程间使用。进程的血缘关系通常指父子进程关系。

（2）有名管道（named pipe）：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间通信。

（3）信号量（semophore）：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它通常作为一种锁机制，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。

（4）消息队列（message queue）：消息队列是由消息组成的链表，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。

（5）信号（signal）：信号是一种比较复杂的通信方式，用于通知接收进程某一事件已经发生。

（6）共享内存（shared memory）：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，但多个进程都可以访问，共享内存是最快的IPC方式，它是针对其他进程间的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，来实现进程间的同步和通信。

（7）套接字（socket）：套接口也是一种进程间的通信机制，与其他通信机制不同的是它可以用于不同及其间的进程通信。

几种线程间的通信机制

1、锁机制

1.1 互斥锁：提供了以排它方式阻止数据结构被并发修改的方法。

1.2 读写锁：允许多个线程同时读共享数据，而对写操作互斥。

1.3 条件变量：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

2、信号量机制：包括无名线程信号量与有名线程信号量

3、信号机制：类似于进程间的信号处理。

线程间通信的主要目的是用于线程同步，所以线程没有象进程通信中用于数据交换的通信机制。

进程和线程的区别？

定义：

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

.关系

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

3.区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

4.优缺点

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

### 2 ArrayList如何实现自动扩容？

注意：

不同的JDK版本的扩容机制可能有差异

实验环境：JDK1.8

扩容机制：

当向ArrayList中添加元素的时候，ArrayList如果要满足新元素的存储超过ArrayList存储新元素前的存储能力，ArrayList会增强自身的存储能力，已达到存储新元素的要求

ArrayList:本质通过内部维护的数组对象进行数据存储

①：分析ArrayList的add(E)方法

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;

return true;

}

分析：add方法首先通过ensureCapacityInternal()方法确保当前ArrayList维护的数组具有存储新元素的能力，经过处理之后将元素存储在数组elementData的尾部

elementData：ArrayList真正用于存储元素的数组

②：分析ensureCapacityInternal方法

private void ensureCapacityInternal(int minCapacity) {

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);

}

ensureExplicitCapacity(minCapacity);

}

分析：ensureCapacityInternal判断ArrayList默认的元素存储数据是否为空，为空则设置最小要求的存储能力为必要存储的元素和默认存储元素个数的两个数据之间的最大值，然后调用ensureExplicitCapacity方法实现这种最低要求的存储能力

注意：ArrayList的存储空间并不是需要一个创建一个，而是分阶段性的创建，一般会预留存储空间。

例如，如果ArrayList需要存储10个元素，恰好ArrayList只能存储6个元素，剩余4个元素无法存储，ArrayList可能会一次性扩展10个元素，这种ArrayList就有20个元素的存储能力，在存储能力范围内，下次再存放元素，就不需要再次扩容

③：分析ensureExplicitCapacity方法：

private void ensureExplicitCapacity(int minCapacity) {

modCount++;

// overflow-conscious code

if (minCapacity - elementData.length > 0)

grow(minCapacity);

}

分析：如果最低要求的存储能力>ArrayList已有的存储能力，这就表示ArrayList的存储能力不足，因此需要调用 grow();方法进行扩容

④：分析grow()方法

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

分析：当ArrayList扩容的时候，首先会设置新的存储能力为原来的1.5倍

int newCapacity = oldCapacity + (oldCapacity >> 1);

1

如果扩容之后还是不能满足要求则MAX\_ARRAY\_SIZE比较，求取最大值，

如果MAX\_ARRAY\_SIZE大小的能力还是不能满足则通过hugeCapacity()方法获取ArrayList能允许的最大值：

private static int hugeCapacity(int minCapacity) {

if (minCapacity < 0) // overflow

throw new OutOfMemoryError();

return (minCapacity > MAX\_ARRAY\_SIZE) ?

Integer.MAX\_VALUE :

MAX\_ARRAY\_SIZE;

}

从hugeCapacity方法看出，ArrayList最大的存储能力：存储元素的个数为整型的范围。

确定ArrayList扩容之后最新的可存储元素个数时，调用

elementData = Arrays.copyOf(elementData, newCapacity);

实现elementData数组的扩容，整个流程就是ArrayList的自动扩容机制工作流程

扩展：

ArrayList的自动扩容机制底层借助于System实现

public static native void arraycopy

(Object src, int srcPos,

Object dest, int destPos,

int length);

arraycopy标识为native意味JDK的本地库，不可避免的会进行IO操作，如果频繁的对ArrayList进行扩容，毫不疑问会降低ArrayList的使用性能，因此当我们确定添加元素的个数的时候，我们可以事先知道并指定ArrayList的可存储元素的个数，这样当我们向ArrayList中加入元素的时候，就可以避免ArrayList的自动扩容，从而提高ArrayList的性能

ArrayList含参构造函数:初始化时指定存储元素的能力：

public ArrayList(int initialCapacity) {

if (initialCapacity > 0) {

this.elementData = new Object[initialCapacity];

} else if (initialCapacity == 0) {

this.elementData = EMPTY\_ELEMENTDATA;

} else {

throw new IllegalArgumentException(

"Illegal Capacity: "+initialCapacity);

}

}

### 3 线程状态中BLOCKED和WAITING有什么区别？

BLOCKED和WAITING有什么区别呢？

答复在JDK源码中可以找到，如下是Java.lang.Thread.State类的一部分注释。

/\*\*

\* Thread state for a thread blocked waiting for a monitor lock.

\* A thread in the blocked state is waiting for a monitor lock

\* to enter a synchronized block/method or

\* reenter a synchronized block/method after calling

\* {@link Object#wait() Object.wait}.

\*/

BLOCKED,

/\*\*

\* Thread state for a waiting thread.

\* A thread is in the waiting state due to calling one of the

\* following methods:

\* {@link Object#wait() Object.wait} with no timeout

\* {@link #join() Thread.join} with no timeout

\* {@link LockSupport#park() LockSupport.park}

\*

\*

\* A thread in the waiting state is waiting for another thread to

\* perform a particular action.

\*

\* For example, a thread that has called Object.wait()

\* on an object is waiting for another thread to call

\* Object.notify() or Object.notifyAll() on

\* that object. A thread that has called Thread.join()

\* is waiting for a specified thread to terminate.

\*/

WAITING,

从中可以清晰的得到线程处于BLOCKED和WAITING状态的场景。

BLOCKED状态

线程处于BLOCKED状态的场景。

当前线程在等待一个monitor lock，比如等待执行synchronized代码块或者使用synchronized标记的方法。

在synchronized块中循环调用Object类型的wait方法，如下是样例

synchronized(this)

{

while (flag)

{

obj.wait();

}

// some other code

}

WAITING状态

线程处于WAITING状态的场景。

调用Object对象的wait方法，但没有指定超时值。

调用Thread对象的join方法，但没有指定超时值。

调用LockSupport对象的park方法。

提到WAITING状态，顺便提一下TIMED\_WAITING状态的场景。

TIMED\_WAITING状态

线程处于TIMED\_WAITING状态的场景。

调用Thread.sleep方法。

调用Object对象的wait方法，指定超时值。

调用Thread对象的join方法，指定超时值。

调用LockSupport对象的parkNanos方法。

调用LockSupport对象的parkUntil方法。

### 4 JVM如何加载字节码文件？

**见12题，和12题一样**

### 5 常见GC算法有哪些？

**引用计数法 Reference Counting**

给对象添加一个引用计数器，每过一个引用计数器值就+1，少一个引用就-1。当它的引用变为0时，该对象就不能再被使用。它的实现简单，但是不能解决互相循环引用的问题。

**根搜索算法 GC Roots Tracing**

以一系列叫“GC Roots”的对象为起点开始向下搜索，走过的路径称为引用链（Reference Chain），当一个对象没有和任何引用链相连时，证明此对象是不可用的，用图论的说法是不可达的。那么它就会被判定为是可回收的对象。

JAVA里可作为GC Roots的对象   
虚拟机栈（栈帧中的本地变量表）中引用的对象   
方法区中的类静态属性引用的对象   
方法区中的常量引用的对象   
本地方法栈中JNI（即Native方法）的引用的对象

**标记-清除算法 Mark-Sweep**

这是一个非常基本的GC算法，它是现代GC算法的思想基础，分为标记和清除两个阶段：先把所有活动的对象标记出来，然后把没有被标记的对象统一清除掉。但是它有两个问题，一是效率问题，两个过程的效率都不高。二是空间问题，清除之后会产生大量不连续的内存。

**复制算法 Copying**

复制算法是将原有的内存空间分成两块，每次只使用其中的一块。在GC时，将正在使用的内存块中的存活对象复制到未使用的那一块中，然后清除正在使用的内存块中的所有对象，并交换两块内存的角色，完成一次垃圾回收。它比标记-清除算法要高效，但不适用于存活对象较多的内存，因为复制的时候会有较多的时间消耗。它的致命缺点是会有一半的内存浪费。

**标记整理算法 Mark-Compact**

标记整理算法适用于存活对象较多的场合，它的标记阶段和标记-清除算法中的一样。整理阶段是将所有存活的对象压缩到内存的一端，之后清理边界外所有的空间。它的效率也不高。

### 6 [Major GC和Full GC的区别是什么？触发条件呢？](http://blog.csdn.net/z69183787/article/details/51606410)

GC，即就是Java垃圾回收机制。目前主流的JVM（HotSpot）采用的是分代收集算法。与C++不同的是，Java采用的是类似于树形结构的可达性分析法来判断对象是否还存在引用。即：从gcroot开始，把所有可以搜索得到的对象标记为存活对象。

GC机制

要准确理解Java的垃圾回收机制，就要从：“什么时候”，“对什么东西”，“做了什么”三个方面来具体分析。

第一：“什么时候”即就是GC触发的条件。GC触发的条件有两种。（1）程序调用System.gc时可以触发；（2）系统自身来决定GC触发的时机。

系统判断GC触发的依据：根据Eden区和From Space区的内存大小来决定。当内存大小不足时，则会启动GC线程并停止应用线程。

第二：“对什么东西”笼统的认为是Java对象并没有错。但是准确来讲，GC操作的对象分为：通过可达性分析法无法搜索到的对象和可以搜索到的对象。对于搜索不到的方法进行标记。

第三：“做了什么”最浅显的理解为释放对象。但是从GC的底层机制可以看出，对于可以搜索到的对象进行复制操作，对于搜索不到的对象，调用finalize()方法进行释放。

具体过程：当GC线程启动时，会通过可达性分析法把Eden区和From Space区的存活对象复制到To Space区，然后把Eden Space和From Space区的对象释放掉。当GC轮训扫描To Space区一定次数后，把依然存活的对象复制到老年代，然后释放To Space区的对象。

对于用可达性分析法搜索不到的对象，GC并不一定会回收该对象。要完全回收一个对象，至少需要经过两次标记的过程。

第一次标记：对于一个没有其他引用的对象，筛选该对象是否有必要执行finalize()方法，如果没有执行必要，则意味可直接回收。（筛选依据：是否复写或执行过finalize()方法；因为finalize方法只能被执行一次）。

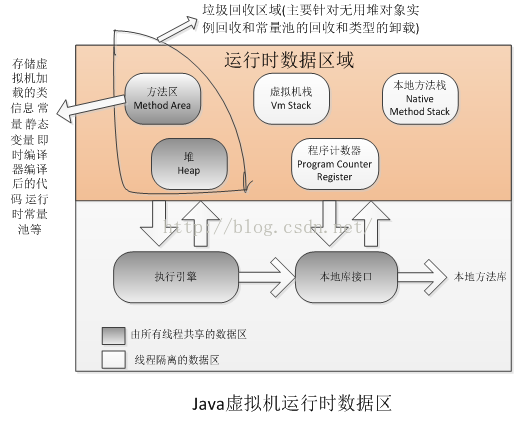
第二次标记：如果被筛选判定位有必要执行，则会放入FQueue队列，并自动创建一个低优先级的finalize线程来执行释放操作。如果在一个对象释放前被其他对象引用，则该对象会被移除FQueue队列。

GC过程中用到的回收算法：

通过上面的GC过程不难看出，Java堆中的年轻代和老年代采用了不同的回收算法。年轻代采用了复制法；而老年代采用了标记-整理法

具体各种回收算法的详解参考：[http://www.cnblogs.com/dolphin0520/p/3783345.html](http://www.cnblogs.com/dolphin0520/p/3783345.html" \t "http://blog.csdn.net/yhyr_ycy/article/details/_blank)

JVM内存空间图解



程序计数器：线程私有。是一块较小的内存，是当前线程所执行的字节码的行号指示器。是Java虚拟机规范中唯一没有规定OOM（OutOfMemoryError）的区域。

Java栈：线程私有。生命周期和线程相同。是Java方法执行的内存模型。执行每个方法都会创建一个栈帧，用于存储局部变量和操作数（对象引用）。局部变量所需要的内存空间大小在编译期间完成分配。所以栈帧的大小不会改变。存在两种异常情况：若线程请求深度大于栈的深度，抛StackOverflowError。若栈在动态扩展时无法请求足够内存，抛OOM。

Java堆：所有线程共享。虚拟机启动时创建。存放对象实力和数组。所占内存最大。分为新生代（Young区），老年代（Old区）。新生代分Eden区，Servior区。Servior区又分为From space区和To Space区。Eden区和Servior区的内存比为8:1。 当扩展内存大于可用内存，抛OOM。

方法区：所有线程共享。用于存储已被虚拟机加载的类信息、常量、静态变量等数据。又称为非堆（Non – Heap）。方法区又称“永久代”。GC很少在这个区域进行，但不代表不会回收。这个区域回收目标主要是针对常量池的回收和对类型的卸载。当内存申请大于实际可用内存，抛OOM。

本地方法栈：线程私有。与Java栈类似，但是不是为Java方法（字节码）服务，而是为本地非Java方法服务。也会抛StackOverflowError和OOM。

 Minor GC ，Full GC 触发条件

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

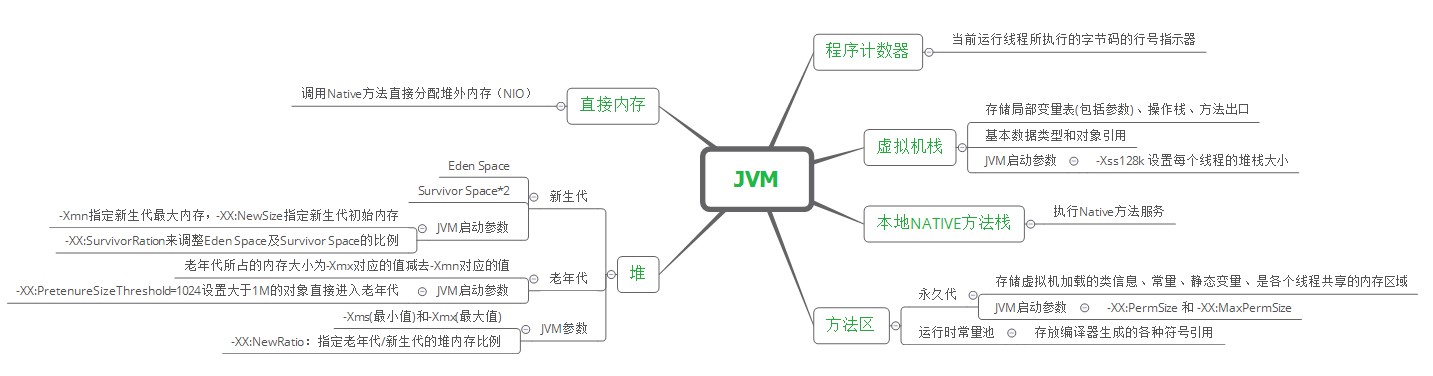
（3）方法去空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

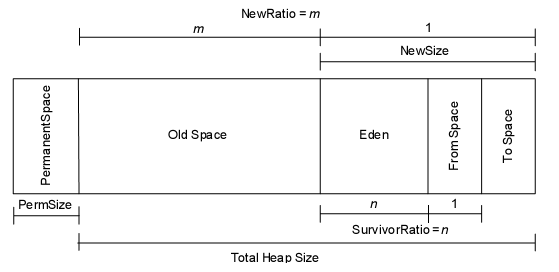
（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

### 7 JVM内存模型是什么？

下图说明JVM内存模型和JVM参数的关系：

[](http://static.haonanji.cn/wp-content/uploads/2016/04/JVM%E5%86%85%E5%AD%98%E6%A8%A1%E5%9E%8B%E5%92%8C%E5%90%AF%E5%8A%A8%E5%8F%82%E6%95%B0%E7%9A%84%E5%85%B3%E7%B3%BB.jpg)

JAVA堆的描述如下：



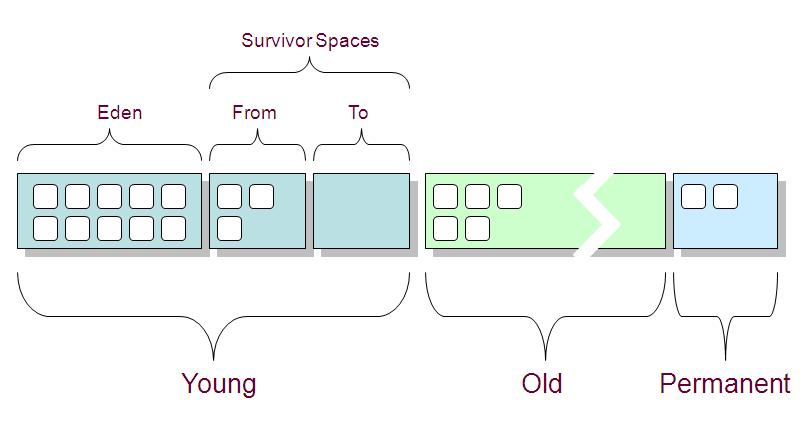
 　　内存由Perm和Heap组成。其中Heap = {Old + NEW = { Eden , from, to } }

　　JVM内存模型中分两大块:

　　NEW Generation：程序新创建的对象都是从新生代分配内存，新生代由Eden Space和两块相同大小的Survivor Space(通常又称S0和S1或From和To)构成，可通过-Xmn参数来指定新生代的大小，也可以通过-XX:SurvivorRation来调整Eden Space及Survivor Space的大小。垃圾回收一般用Copying的算法，速度快。每次GC的时候，存活下来的对象首先由Eden拷贝到某个Survivor Space, 当Survivor Space空间满了后, 剩下的live对象就被直接拷贝到Old Generation中去。因此，每次GC后，Eden内存块会被清空。

　　Old Generation：用于存放经过多次新生代GC任然存活的对象和应用程序中生命周期长的内存对象，例如缓存对象，新建的对象也有可能直接进入老年代，主要有两种情况：①大对象，可通过启动参数设置-XX:PretenureSizeThreshold=1024(单位为字节，默认为0)来代表超过多大时就不在新生代分配，而是直接在老年代分配。②大的数组对象，切数组中无引用外部对象。老年代所占的内存大小为-Xmx对应的值减去-Xmn对应的值。在Old Generation块中，垃圾回收一般用mark-compact的算法，速度慢些，但减少内存要求。

　　PS：还有个Permanent Generation，主要用来放JVM自己的反射对象，比如类对象和方法对象等。



　　垃圾回收描述：

　　现在收集器都是采用分代收集算法，堆被划分为新生代和老年代。新生代主要存储新创建的对象和尚未进入老年代的对象。老年代存储经过多次新生代GC(Minor GC)任然存活的对象。

　　Minor GC：即新生代GC，指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。垃圾回收分多级是1级或以上为部分垃圾回收，只会回收NEW中的垃圾。

　　Major GC  / Full GC：老年代GC，指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC的策略选择过程） 。MajorGC 的速度一般会比Minor GC慢10倍以上。垃圾回收分多级是0级为全部(Full)的垃圾回收，会回收OLD段中的垃圾。

　　PS：内存溢出通常发生于OLD段或Perm段垃圾回收后，Eden区仍然无内存空间容纳新的Java对象的情况。

　　当一个URL被访问时，内存申请过程如下：

　　A. JVM会试图为相关Java对象在Eden中初始化一块内存区域

　　B. 当Eden空间足够时，内存申请结束。否则到下一步

　　C. JVM试图释放在Eden中所有不活跃的对象（这属于1或更高级的垃圾回收）, 释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区

　　D. Survivor区被用来作为Eden及OLD的中间交换区域，当OLD区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区

　　E. 当OLD区空间不够时，JVM会在OLD区进行完全的垃圾收集（0级）

　　F. 完全垃圾收集后，若Survivor及OLD区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现”out of memory错误”

　　为什么一些程序频繁发生GC？有如下原因：

　　（1）程序内调用了System.gc()或Runtime.gc()。

　　 （2）一些中间件软件调用自己的GC方法，此时需要设置参数禁止这些GC。

　　（3）Java的Heap太小，一般默认的Heap值都很小。

　　（4）频繁实例化对象，Release对象。此时尽量保存并重用对象，例如使用StringBuffer()和String()。

　　如果你发现每次GC后，Heap的剩余空间会是总空间的50%，这表示你的Heap处于健康状态。许多Server端的Java程序每次GC后最好能有65%的剩余空间。

　　JVM 使用的GC算法是什么？

　　分代收集。即将内存分为几个区域，将不同生命周期的对象放在不同区域里；

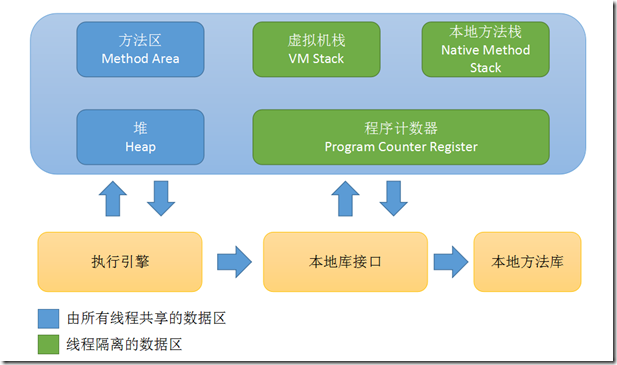
　　在GC收集的时候，频繁收集生命周期短的区域(Young area)；

　　比较少的收集生命周期比较长的区域(Old area)；

　　基本不收集的永久区(Perm area)。

　　GC何时会被触发？  
　　（1）系统空闲：GC线程的优先级低于系统应用线程，当系统中没有应用线程执行时，GC会被触发。  
　　（2）堆空间内存不足：当堆空间的内存不足以创建新对象时，GC会被触发。如果第一GC仍不能获得足够的空间，第二次GC将被触发，如果这一次仍无法获取足够的空间，“Out of memory”将被抛出。

### 8 [JAVA运行时数据区域](http://www.cnblogs.com/fengbs/p/7029013.html)有哪些？

[](http://images2015.cnblogs.com/blog/1182497/201706/1182497-20170616192739978-1176032049.png)

1.程序计数器

    程序计数器（Program Counter Register） 是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条执行字节码指令。

    每条线程都有一个独立的程序计数器。

如果执行的是java方法，这个计数器记录的是正在执行的虚拟机字节码指令地址。如果是native方法，计数器为空。此内存区域是唯一一个在java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

2.Java虚拟机栈

    同样是线程私有，描述Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。一个方法对应一个栈帧。

    局部变量表存放了各种基本类型、对象引用和returnAddress类型（指向了一条字节码指令地址）。其中64位长度long 和 double占两个局部变量空间，其他只占一个。

规定的异常情况有两种：1.线程请求的栈的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；2.如果虚拟机可以动态扩展，如果扩展时无法申请到足够的内存，就抛出OutOfMemoryError异常。

3.本地方法栈

和Java虚拟机栈很类似，不同的是本地方法栈为Native方法服务。

4.Java堆

    是Java虚拟机所管理的内存中最大的一块。由所有线程共享，在虚拟机启动时创建。堆区唯一目的就是存放对象实例。

    堆中可细分为新生代和老年代，再细分可分为Eden空间、From Survivor空间、To Survivor空间。

堆无法扩展时，抛出OutOfMemoryError异常

5.方法区

   所有线程共享，存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

当方法区无法满足内存分配需求时，抛出OutOfMemoryError

6.运行时常量池

   它是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项是常量池（Const Pool Table），用于存放编译期生成的各种字面量和符号引用。并非预置入Class文件中常量池的内容才进入方法运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。

当方法区无法满足内存分配需求时，抛出OutOfMemoryError

7.直接内存

    并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域。

    JDK1.4加入了NIO，引入一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。因为避免了在Java堆和Native堆中来回复制数据，提高了性能。

    当各个内存区域总和大于物理内存限制，抛出OutOfMemoryError异常。

### 9 事务实现过程及原理是什么？

1、Spring中事务处理的作用：

Spring事务处理，是将事务处理的工作统一起来，并为事务处理提供通用的支持。

2、工作原理及实现

a、划分处理单元——IOC

由于spring解决的问题是对单个数据库进行局部事务处理的，具体的实现首相用spring中的IOC划分了事务处理单元。并且将对事务的各种配置放到了ioc容器中（设置事务管理器，设置事务的传播特性及隔离机制）。

b、AOP拦截需要进行事务处理的类

Spring事务处理模块是通过AOP功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用TransactionProxyFactoryBean接口来使用AOP功能，生成proxy代理对象，通过TransactionInterceptor完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。

读取ioc容器事务配置属性，转化为spring事务处理需要的内部数据结构（TransactionAttributeSourceAdvisor），转化为TransactionAttribute表示的数据对象。

c、对事物处理实现（事务的生成、提交、回滚、挂起）

spring委托给具体的事务处理器实现。实现了一个抽象和适配。适配的具体事务处理器：DataSource数据源支持、hibernate数据源事务处理支持、JDO数据源事务处理支持，JPA、JTA数据源事务处理支持。这些支持都是通过设计PlatformTransactionManager、AbstractPlatforTransaction一系列事务处理的支持。

为常用数据源支持提供了一系列的TransactionManager。

d、结合

PlatformTransactionManager实现了TransactionInterception接口，让其与TransactionProxyFactoryBean结合起来，形成一个Spring声明式事务处理的设计体系。

3、应用场景

支持不同数据源，在底层进行封装，可以做到事务即开即用，这样的好处是：即使有其他的数据源事务处理需要，Spring也提供了一种一致的方式。

### 10 如何理解http协议和tcp协议？

TCP协议对应于传输层，而HTTP协议对应于应用层，从本质上来说，二者没有可比性。Http协议是建立在TCP协议基础之上的，当浏览器需要从服务器获取网页数据的时候，会发出一次Http请求。Http会通过TCP建立起一个到服务器的连接通道，当本次请求需要的数据完毕后，Http会立即将TCP连接断开，这个过程是很短的。所以Http连接是一种短连接，是一种无状态的连接。所谓的无状态，是指浏览器每次向服务器发起请求的时候，不是通过一个连接，而是每次都建立一个新的连接。如果是一个连接的话，服务器进程中就能保持住这个连接并且在内存中记住一些信息状态。而每次请求结束后，连接就关闭，相关的内容就释放了，所以记不住任何状态，成为无状态连接。  
  
随着时间的推移，html页面变得复杂了，里面可能嵌入了很多图片，这时候每次访问图片都需要建立一次tcp连接就显得低效了。因此Keep-Alive被提出用来解决效率低的问题。从HTTP/1.1起，默认都开启了Keep-Alive，保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。虽然这里使用TCP连接保持了一段时间，但是这个时间是有限范围的，到了时间点依然是会关闭的，所以我们还把其看做是每次连接完成后就会关闭。后来，通过Session, Cookie等相关技术，也能保持一些用户的状态。但是还是每次都使用一个连接，依然是无状态连接。  
  
以前有个概念很容忍搞不清楚。就是为什么Http是无状态的短连接，而TCP是有状态的长连接？Http不是建立在TCP的基础上吗，为什么还能是短连接？现在明白了，Http就是在每次请求完成后就把TCP连接关了，所以是短连接。而我们直接通过Socket编程使用TCP协议的时候，因为我们自己可以通过代码区控制什么时候打开连接什么时候关闭连接，只要我们不通过代码把连接关闭，这个连接就会在客户端和服务端的进程中一直存在，相关状态数据会一直保存着。

在C#中会有Socket，实际上socket是对TCP/IP协议的封装，Socket本身并不是协议，而是一个调用接口(API)。Socket的出现只是使得程序员更方便地使用TCP/IP协议栈而已，是对TCP/IP协议的抽象，从而形成了我们知道的一些最基本的函数接口，比如create、listen、connect、accept、send、read和write等等。

比较形象的描述：HTTP是轿车，提供了封装或者显示数据的具体形式;Socket是发动机，提供了网络通信的能力。对于从C#编程的角度来讲，为了方便，你可以直接选择已经制造好的轿车Http来与服务器交互。但是有时候往往因为环境因素或者其他的一些定制的请求，必须要使用TCP协议，这时就需要使用Socket编程，然后自己去处理获取的数据。就像是你用已有的发动机，自己造了一辆卡车，去从服务器交互。

HTTP/1.0和HTTP/1.1都把TCP作为底层的传输协议。HTTP客户首先发起建立与服务器TCP连接。一旦建立连接，浏览器进程和服务器进程就可以通过各自的套接字来访问TCP。如前所述，客户端套接字是客户进程和TCP连接之间的“门”，服务器端套接字是服务器进程和同一TCP连接之间的“门”。客户往自己的套接字发送HTTP请求消息，也从自己的套接字接收HTTP响应消息。类似地，服务器从自己的套接字接收HTTP请求消息，也往自己的套接字发送HTTP响应消息。客户或服务器一旦把某个消息送入各自的套接字，这个消息就完全落入TCP的控制之中。TCP给HTTP提供一个可靠的数据传输服务;这意味着由客户发出的每个HTTP请求消息最终将无损地到达服务器，由服务器发出的每个HTTP响应消息最终也将无损地到达客户。  
  
C#代码连接远程数据库用的是TCP协议。每次new 一个connection的时候，connection.open就打开了这个TCP连接。connection.Close的时候就关闭了这个连接。FTP的底层也是TCP， 不过是长连接的。传输大文件比较快。 需要看具体场景。在服务器端，如果程序是采取的长连接的方式，那么就能控制同时连接到这个服务器的连接个数，防止同时有多个连接。但是采取短连接的方式，那么就不能控制同时连接到这个服务器上的连接的个数，这也是一个优点，可以同时处理大量连接请求。但是如果连接请求量太大的话，可能造成服务器停止工作。

WebService不需要连接，一秒中至少可以支持上万/十万的请求，每次请求然后释放，没有空余的内存消耗。一般不会限制同时连接的个数，这是优势。Message Queue需要建立连接， 支持上千的连接就很吃力了。因为每个连接即使没有在请求数据，也会在内存中占用一定的空间存储。会限制，比如SQL Server数据库服务器，一般最多同时连接16个。

Http协议一定通过指定的端口，80，所以一般计算机上不会限制这个端口，所以Http协议能够顺利通过所有机器上的防火墙。而使用Socket编程的话，就需要自己指定特定的端口，那么很可能这个端口是在某个环境中禁用的，那么就无法穿透防火墙。IIS使用的是80端口，也就是这个程序一直在监听着这个端口。一旦发现有人要建立到这个端口的连接，他就会响应，然后建立连接。这里说的连接都是短连接。所以你对服务器上的网址的请求，都是通过80端口送到网站程序的。然后通过这个端口发送的客户端浏览器。

### 11 [一致性哈希算法原理](http://www.cnblogs.com/lpfuture/p/5796398.html)是什么？

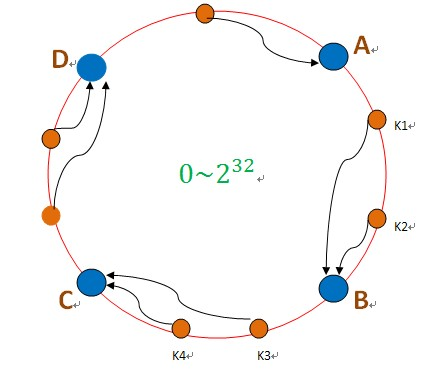
使用场景

现在我们假设有100台redis data服务器，一份数据101进来的时候，以散列公式hash(i)&100，计算所存放的服务器，假设hash(i) = i,那么数据被散列到标号为1的服务器,然后这个时候服务器新增了一台，然后散列公式为hash(i)%101，这个时候请求访问数据101的时候，被分配至0号服务器，但是其实这个时候数据是在1号服务器的。

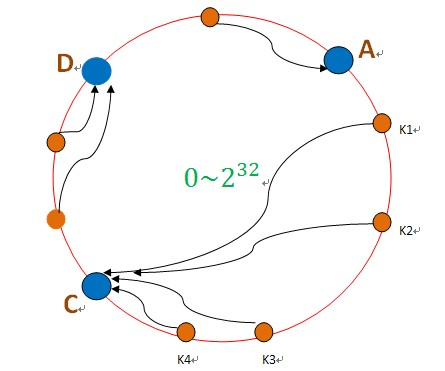
所以这个时候大量的数据失效了（访问不到了）。

所以这个时候，我们假设是新增了服务器，如果是持久化存储的，我们可以让服务器集群对数据进行重新散列，进行数据迁移，然后进行恢复，但是这个时候就意味着每次增减服务器的时候，集群就需要大量的通信，进行数据迁移，这个开销是非常大的。如果只是缓存，那么缓存就都失效了。所以这个时候怎么办？

我们可以看到，关键问题在于，服务器数量变动的时候，要能够保证旧的数据能够按照老的算法，计算到数据所在的服务器，而新的数据能够按照新的散列算法，计算出数据所在的服务器。



如上图，我们有ABCD四台服务器，这四台服务器被分配至0~232 的一个环上，比如0~230的存储在A服务器，230 +1~231 存储到B服务器上.....CD按照这样的进行均分。将我们的散列空间也划为0~232 ，然后数据进来后对232 取模，得到一个值K1，我们根据K1在环上所处的位置，得到所分配到的服务器，如图，K1被分配到B服务器。 这个时候，我们有一台服务器B失效了。



我们可以看到，如果是B失效了，那么如果有持久化存储的，需要做数据恢复，将B的数据迁移至C即可，对于原本散列在A和D的数据，不需要做任何改变。 同理，如果我们是新增了服务器，那么只需要对一台服务器的数据迁移一部分至新加的服务器即可。

一致性hash算法，减少了数据映射关系的变动，不会像hash(i)%N那样带来全局的变动

而且这样还有个好处，假设我们使用UID作为散列范围（即上面的232 ）,那么假设有部分UID的访问很频繁，而且这部分UID集中在B服务器上，那么就造成了B的负载远远高于其他服务器。这就是热点数据的问题。这个时候我们可以向B所在的UID空间添加服务器，减少B的压力。

其实还有个更好的解决办法：虚拟节点。

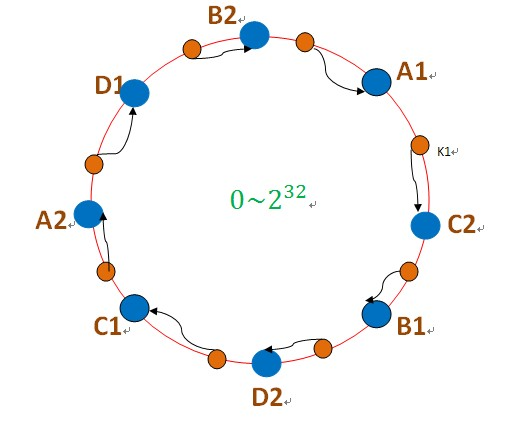
上面说的情况是，使用真实的服务器作为节点散列在232 上。 我们假设，只有4台服务器（如上图），然后A上面有热点数据，结果A挂掉了，然后做数据恢复，A的数据迁移至B，然后B需要承受A+B的数据，也承受不住，也挂了。。。。然后继续CD都挂了。这就造成了

雪崩效应。

上面会造成雪崩效应的原因分析：  
如果不存在热点数据的时候，每台机器的承受的压力是M/2(假设每台机器的最高负载能力为M)，原本是不会有问题的，但是，这个时候A服务器由于有热点数据挂了，然后A的数据迁移至B，导致B所需要承受的压力变为M（还不考虑热点数据访问的压力），所以这个失败B是必挂的，然后C至少需要承受1.5M的压力。。。。然后大家一起挂。。。  
所以我们通过上面可以看到，之所以会大家一起挂，原因在于如果一台机器挂了，那么它的压力全部被分配到一台机器上，导致雪崩。

如果我们A挂了以后，数据被平均分配到BCD上，每台机器多承受M/6的压力，然后大家就都不会挂啦（不考虑热点数据）。

这里引入虚拟节点，如图：



环上的空间被划分为8份，然后A存储A1和A2。。。  
这个时候，如果A服务器挂了，访问压力会分配至C2和D1，也就是C和D服务器，而不是像前面，全部被分配到B上。

引入虚拟节点，主要在于，如果一台服务器挂了，能够将压力引流至不同的服务器。

总结：一致性hash算法（DHT）通过减少影响范围的方式解决了增减服务器导致的数据散列问题，从而解决了分布式环境下负载均衡问题，如果存在热点数据，那么通过增添节点的方式，对热点区间进行划分，将压力分配至其他服务器。重新达到负载均衡的状态。

tair的负载均衡就是采用的一致性hash算法啦~~~  
一致性hash算法在分布式环境中应用的很广，只要是涉及到分布式存储的负载均衡问题，一致性hash都是很好的解决的方案。

### 12 [jvm字节码的加载与卸载](http://www.cnblogs.com/macgradyjames/p/7404054.html)过程怎样的？

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，转换分析和初始化，最终形成可以被虚拟节直接使用的JAVA类型，这就是虚拟机的类加载机制。

类从被加载到虚拟机内存到卸载出内存的生命周期包括：加载->连接(验证->准备->解析)->初始化->使用->卸载

初始化的5种情况：

1.使用new关键字实例化对象时，读取或设置一个类的静态字段，除被final修饰经编译结果放在常量池的静态字段，调用类的静态方法时。 2.使用java.lang.reflect包方法对类进行反射调用时。（Class.forName()）。 3.初始化子类时，如果父类没有初始化。 4.虚拟机启动时main方法所在的类。 5.当使用JDK1.7动态语言支持时，java.lang.invoke.MethodHandle实例解析结果为REF\_getStatic,REF\_putStatic,REF\_invokeStatic的方法句柄，且对应类没有进行初始化。

加载 加载是类加载的第一个阶段，虚拟机要完成以下三个过程：

1.通过类的全限定名获取定义此类的二进制字节流。 2.将字节流的存储结构转化为方法区的运行时结构。 3.在内存中生成一个代表该类的Class对象，作为方法区各种数据的访问入口。

验证 目的是确保class文件字节流信息符合虚拟机的要求。

准备 为static修饰的变量赋初值，例如int型默认为0，boolean默认为false。

解析 虚拟机将常量池内的符号引用替换成直接引用。

初始化 初始化是类加载的最后一个阶段，将执行类构造器< init>()方法，注意这里的方法不是构造方法。该方法将会显式调用父类构造器，接下来按照java语句顺序为类变量和静态语句块赋值。

方法调用

Java是一门面向对象的语言，它具有多态性。那么虚拟机又是如何知道运行时该调用哪一个方法？

静态分派是在编译期就决定了该调用哪一个方法而不是由虚拟机来确定，方法重载就是典型的静态分派。 动态分派是在虚拟机运行阶段才能决定调用哪一个方法，方法重写就是典型的动态分派。

动态分派的实现：当调用一个对象的方法时，会将该对象的引用压栈到操作数栈，然后字节码指令invokevirtual会去寻找该引用实际类型。如果在实际类型中找对应的方法，且访问权限足够，则直接返回该方法引用，否则会依照继承关系对父类进行查找。实际上，如果子类没有重写父类方法，则子类方法的引用会直接指向父类方法。

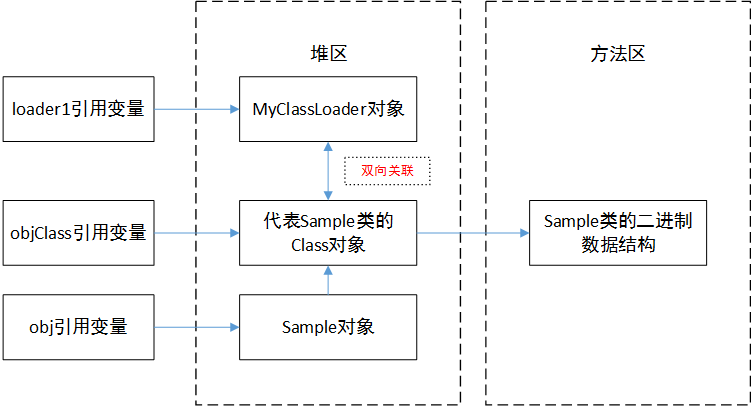
**由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。**

　　前面介绍过，Java虚拟机**自带的类加载器**包括**根类加载器**、**扩展类加载器**和**系统类加载器**。

　　Java虚拟机本身会**始终引用**这些类加载器，而这些类加载器则会**始终引用**它们所加载的类的Class对象，因此这些Class对象**始终是可触及的**。

**由用户自定义的类加载器加载的类是可以被卸载的。**

**具体的例子为：**

****

       loader1变量和obj变量间接应用代表Sample类的Class对象，而objClass变量则直接引用它。

　　如果程序运行过程中，将上图左侧三个引用变量都置为null，此时Sample对象结束生命周期，MyClassLoader对象结束生命周期，代表Sample类的Class对象也结束生命周期，Sample类在方法区内的二进制数据**被卸载**。

　　当再次有需要时，会检查Sample类的Class对象是否存在，**如果存在会直接使用，不再重新加载**；如果不存在Sample类会被重新加载，在Java虚拟机的堆区会生成一个新的代表Sample类的Class实例(可以通过哈希码查看是否是同一个实例)。

### 13 Http连接池作用是什么？实现原理？

随着微服务的流行，服务之间的http调用越来越多，遇到的问题也比较多，写这边文章的目的也是将自己遇到的坑和解决方案跟大家分享

一、为什么要用Http连接池

1、降低延迟：如果不采用连接池，每次连接发起Http请求的时候都会重新建立TCP连接(经历3次握手)，用完就会关闭连接(4次挥手)，如果采用连接池则减少了这部分时间损耗，别小看这几次握手，本人经过测试发现，基本上3倍的时间延迟

2、支持更大的并发：如果不采用连接池，每次连接都会打开一个端口，在大并发的情况下系统的端口资源很快就会被用完，导致无法建立新的连接

二、代码

1、HttpConnectionManager.java连接池管理类，支持https协议

@Component

public class HttpConnectionManager {

PoolingHttpClientConnectionManager cm = null;

@PostConstruct

public void init() {

LayeredConnectionSocketFactory sslsf = null;

try {

sslsf = new SSLConnectionSocketFactory(SSLContext.getDefault());

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

}

Registry<ConnectionSocketFactory> socketFactoryRegistry = RegistryBuilder.<ConnectionSocketFactory> create()

.register("https", sslsf)

.register("http", new PlainConnectionSocketFactory())

.build();

cm =new PoolingHttpClientConnectionManager(socketFactoryRegistry);

cm.setMaxTotal(200);

cm.setDefaultMaxPerRoute(20);

}

public CloseableHttpClient getHttpClient() {

CloseableHttpClient httpClient = HttpClients.custom()

.setConnectionManager(cm)

.build();

/\*CloseableHttpClient httpClient = HttpClients.createDefault();//如果不采用连接池就是这种方式获取连接\*/

return httpClient;

}

}

2、连接池消费类：HaoMaiClient.java

@Component

public class HaoMaiClient {

@Autowired

HttpConnectionManager connManager;

public <T> T get(String path,Class<T> clazz){

CloseableHttpClient httpClient=connManager.getHttpClient();

HttpGet httpget = new HttpGet(path);

String json=null;

CloseableHttpResponse response=null;

try {

response = httpClient.execute(httpget);

InputStream in=response.getEntity().getContent();

json=IOUtils.toString(in);

in.close();

} catch (UnsupportedOperationException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}finally {

if(response!=null){

try {

response.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return JSON.parseObject(json, clazz);

}

}

三、原理及注意事项

连接池中连接都是在发起请求的时候建立，并且都是长连接

HaoMaiClient.java中的in.close();作用就是将用完的连接释放，下次请求可以复用，这里特别注意的是，如果不使用in.close();而仅仅使用response.close();结果就是连接会被关闭，并且不能被复用，这样就失去了采用连接池的意义。

连接池释放连接的时候，并不会直接对TCP连接的状态有任何改变，只是维护了两个Set，leased和avaliabled，leased代表被占用的连接集合，avaliabled代表可用的连接的集合，释放连接的时候仅仅是将连接从leased中remove掉了，并把连接放到avaliabled集合中

### 14 Redis如何实现扩容？

team中的一个同学在其项目中使用了Redis作为缓存，将热点数据存放在Redis中。为了提升性能，写Redis时采用了管道的方式，平时使用时，Redis的性能、资源使用都能符合项目需求，但当访问量增加的时候，Redis的QPS还能满足要求，但CPU使用率高的时候已经达到90%+，平时只有30%+，而众所周知，Redis是单进程的，只能占用1个CPU核，跑满了也就100%，无法利用机器的多核，而当CPU跑到100%时，必然会造成性能瓶颈。怎么解决？

方案一：

首先想到的是，增加Redis服务器的数量，在客户端对存储的key进行hash运算，存入不同的Redis服务器中，读取时，也进行相同的hash运算，找到对应的Redis服务器，可以解决问题，但是不好的地方：

第一，客户端要改动代码；

第二、需要客户端记住所有的Redis服务器的地址；

这个方案可以使用，但能不能不用改动代码就能实现扩容呢？

方案二：

搭建一个集群，由于Redis服务器使用的版本低于3.0，不支持集群，只能通过使用代理，就想到了有名的Redis代理twemproxy。

twemproxy的性能也是杠杠滴，虽然是代理，但它对访问性能的影响非常小，连Redis作者都推荐它。

twemproxy使用方便，对于一个新手来说，不到一个小时就能学会使用，而且关键是不用改动客户端代码，几乎支持所有的Redis命令和管道操作，只需要改下客户端的配置文件中配置的Redis的IP和PORT，由原来的Redis的IP和Port改成twemproxy服务的IP和PORT。

客户端不需要考虑hash的问题，这些twemproxy会做，客户端就像操作一台Redis一样。

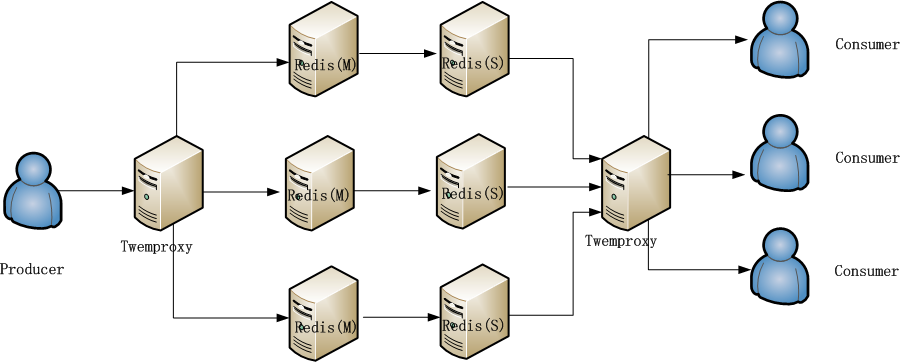
上面用了“几乎”这个词，因为有些命令，比如"keys \*"就不支持

很快部署了好了twemproxy和后面跟着的四个Redis机器，压测发现，后面的四台Redis的CPU使用率降下来了，但新问题来了，twemproxy也是单进程的！性能瓶颈又跑到twemproxy上来了！

方案三：

对Redis的访问分为写和读，类似生产者和消费者， 再仔细分析，发现写的少，读的相对多些，这就可以将读写分离，写的往主的写，读的从备的读，遇到的情况恰好是读和写是两个服务，做到读写分离通过改下配置信息就可以很简单的做到，，这样分散了主Redis的压力。

这里对Redis的访问压力有好转，但不是长久之计，比如遇到举办活动， 数据量增大时，还是会有性能的风险。

最终采用的方法是综合方案二和三，如下图所示：  


这种方法对现有的服务改动最小，可以有效缓解redis压力的问题

producer端和consumer端的twemproxy使用的hash算法要求一致，不然找不到key了。

如果把方案一也加进来，会比较复杂，暂时用不到。

### 15 Netty是如何使用线程池的，为什么这么使用？

线程模型是Netty的核心设计，设计地很巧妙，之前项目中有一块处理并发的设计和Netty的Eventloop单线程设计类似，效果得到了实证。

Netty5的类层次结构和之前的版本变化很大，网上也有很多文章写Netty的线程模型，Reactor模式，比如这篇http://blog.csdn.net/xiaolang85/article/details/37873059， 应该是引自《Netty权威指南》，写得比较全面，但是有几个关键的概念没讲清楚。

这篇文章只讲Netty5线程模型最重要的几个关键点

第一个概念是如何理解NioEventLoop和NioEventLoopGroup：NioEventLoop实际上就是工作线程，可以直接理解为一个线程。NioEventLoopGroup是一个线程池，线程池中的线程就是NioEventLoop。Netty设计这几个类的时候，层次结构挺复杂，反而让人迷惑。

还有一个让人迷惑的地方是，创建ServerBootstrap时，要传递两个NioEventLoopGroup线程池，一个叫bossGroup,一个叫workGroup。《Netty权威指南》里只说了bossGroup是用来处理TCP连接请求的，workGroup是来处理IO事件的。

这么说是没错，但是没说清楚bossGroup具体如何处理TCP请求的。实际上bossGroup中有多个NioEventLoop线程，每个NioEventLoop绑定一个端口，也就是说，如果程序只需要监听1个端口的话，bossGroup里面只需要有一个NioEventLoop线程就行了。

在上一篇文章介绍服务器端绑定的过程中，我们看到最后是NioServerSocketChannel封装的Java的ServerSocketChannel执行了绑定，并且执行accept()方法来创建客户端SocketChannel的连接。一个端口只需要一个NioServerSocketChannel即可。

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

try {

ServerBootstrap b = new ServerBootstrap();

b.group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG, 1024)

.childHandler(new ChildChannelHandler());

ChannelFuture f = b.bind(port).sync();

f.channel().closeFuture().sync();

} finally {

bossGroup.shutdownGracefully();

workerGroup.shutdownGracefully();

}

protected MultithreadEventExecutorGroup(int nThreads, Executor executor, Object... args) {

if (nThreads <= 0) {

throw new IllegalArgumentException(String.format("nThreads: %d (expected: > 0)", nThreads));

}

if (executor == null) {

executor = new ThreadPerTaskExecutor(newDefaultThreadFactory());

}

children = new EventExecutor[nThreads];

for (int i = 0; i < nThreads; i ++) {

boolean success = false;

try {

children[i] = newChild(executor, args);

success = true;

} catch (Exception e) {

// TODO: Think about if this is a good exception type

throw new IllegalStateException("failed to create a child event loop", e);

} finally {

第二个概念是每个NioEventLoop都绑定了一个Selector，所以在Netty5的线程模型中，是由多个Selecotr在监听IO就绪事件。而Channel注册到Selector。

举个例子，比如有100万个连接连到服务器端。平时的写法可能是1个Selector线程监听所有的IO就绪事件，1个Selector面对100万个连接(Channel)。

而如果使用了1000个NioEventLoop的线程池来说，1000个Selector面对100万个连接，每个Selector只需要关注1000个连接(Channel)

[java] view plaincopy

public final class NioEventLoop extends SingleThreadEventLoop {

/\*\*

\* The NIO {@link Selector}.

\*/

Selector selector;

private SelectedSelectionKeySet selectedKeys;

private final SelectorProvider provider;

第三个概念是一个Channel绑定一个NioEventLoop，相当于一个连接绑定一个线程，这个连接所有的ChannelHandler都是在一个线程中执行的，避免的多线程干扰。更重要的是ChannelPipline链表必须严格按照顺序执行的。单线程的设计能够保证ChannelHandler的顺序执行。

[java] view plaincopy

public interface Channel extends AttributeMap, Comparable<Channel> {

/\*\*

\* Return the {@link EventLoop} this {@link Channel} was registered too.

\*/

EventLoop eventLoop();

第四个概念是一个NioEventLoop的selector可以被多个Channel注册，也就是说多个Channel共享一个EventLoop。EventLoop的Selecctor对这些Channel进行检查。

这段代码展示了线程池如何给Channel分配EventLoop,是根据Channel个数取模

[java] view plaincopy

public EventExecutor next() {

return children[Math.abs(childIndex.getAndIncrement() % children.length)];

}

private void processSelectedKeysOptimized(SelectionKey[] selectedKeys) {

for (int i = 0;; i ++) {

// 逐个处理注册的Channel

final SelectionKey k = selectedKeys[i];

if (k == null) {

break;

}

final Object a = k.attachment();

if (a instanceof AbstractNioChannel) {

processSelectedKey(k, (AbstractNioChannel) a);

} else {

@SuppressWarnings("unchecked")

NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;

processSelectedKey(k, task);

}

if (needsToSelectAgain) {

selectAgain();

// Need to flip the optimized selectedKeys to get the right reference to the array

// and reset the index to -1 which will then set to 0 on the for loop

// to start over again.

//

// See https://github.com/netty/netty/issues/1523

selectedKeys = this.selectedKeys.flip();

i = -1;

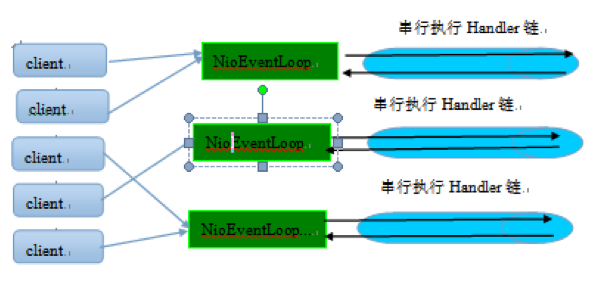
}

}

}

理解了这4个概念之后就对Netty5的线程模型有了清楚的认识：

在监听一个端口的情况下，一个NioEventLoop通过一个NioServerSocketChannel监听端口，处理TCP连接。后端多个工作线程NioEventLoop处理IO事件。每个Channel绑定一个NioEventLoop线程，1个NioEventLoop线程关联一个selector来为多个注册到它的Channel监听IO就绪事件。NioEventLoop是单线程执行，保证Channel的pipline在单线程中执行，保证了ChannelHandler的执行顺序。



### 16 介绍[Spring的IOC容器初始化过程](http://www.cnblogs.com/myadmin/p/5838795.html)？

Ioc容器的初始化是由refresh（）方法来启动的，这个方法标志着Ioc容器的正式启动。

具体来说这个启动过程包括三个基本过程：

1.BeanDifinition的Resource定位

2.BeanDifinition的载入与解析

3.BeanDifinition在Ioc容器中的注册

需要注意的是，Spring把这三个过程分开，并使用不同的模块来完成，如使用相应的ResourceLoader、BeanDifinitionReader等模块，通过这样的实际方式，可以让用户更加灵活的对这三个过程进行剪裁和扩展。

定义出最适合自己的Ioc容器的初始化过程。

第一个过程：BeanDifinition的Resource定位

这个Resource定位指的是BeanDifinition的资源定位，它由ResourceLoader通过统一的Resource接口来完成，这个Resource对各种形式的BeanDifinition的使用都提供了统一的接口。

对于这些BeanDifinition的存在形式，相信大家都不会感到陌生。比如，

在文件系统中的Bean定义信息可以使用FileSystemResource来进行抽象。

在类路劲中的Bean定义信息可以使用ClassPathResource。

这个定位过程类似于容器寻找数据的过程，就想水桶装水先要把水找到一样。

第二个过程：BeanDifinition的载入

这个载入过程是把用户定义好的Bean表示成Ioc容器内部的数据结构，而这个容器内部的数据结构就是BeanDifinition。具体来说，BeanDifinition实际上就是POJO对象在IOC容器中的抽象，通过这个BeanDifinition定义的数据结构，使IOC容器能够方便的对POJO对象也就是Bean进行管理。

第三个过程：BeanDifinition的注册

这个操作是通过调用BeanDifinitionRegistry借口来实现的。这个注册过程把载入过程中解析得到的BeanDifinition向Ioc容器进行注册。在阅读源码中可知，在IOC容器内部将BeanDifinition注入到一个HashMap中去，Ioc容器就是通过这个HashMap来持有这些BeanDifinition数据的。

这里说到的Ioc容器的初始化过程，一般不包含Bean依赖注入的实现。在Ioc的设计中，Bean定义的载入和依赖注入是俩个独立的过程。依赖注入一般发生在应用第一次通过getBean向容器索取Bean的时候。（使用预实例化的配置除外）

### 17 Spring的IOC容器实现原理，为什么可以通过byName和ByType找到Bean？

**1. IoC理论的背景**

我们都知道，在采用面向对象方法设计的软件系统中，它的底层实现都是由N个对象组成的，所有的对象通过彼此的合作，最终实现系统的业务逻辑。

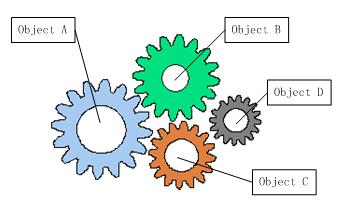


图1：软件系统中耦合的对象

如果我们打开机械式手表的后盖，就会看到与上面类似的情形，各个齿轮分别带动时针、分针和秒针顺时针旋转，从而在表盘上产生正确的时间。图1中描述的就是这样的一个齿轮组，它拥有多个独立的齿轮，这些齿轮相互啮合在一起，协同工作，共同完成某项任务。我们可以看到，在这样的齿轮组中，如果有一个齿轮出了问题，就可能会影响到整个齿轮组的正常运转。

齿轮组中齿轮之间的啮合关系,与软件系统中对象之间的耦合关系非常相似。对象之间的耦合关系是无法避免的，也是必要的，这是协同工作的基础。现在，伴随着工业级应用的规模越来越庞大，对象之间的依赖关系也越来越复杂，经常会出现对象之间的多重依赖性关系，因此，架构师和设计师对于系统的分析和设计，将面临更大的挑战。对象之间耦合度过高的系统，必然会出现牵一发而动全身的情形。

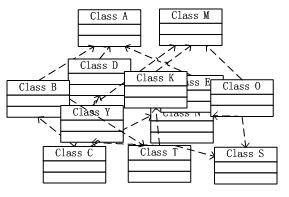
[](http://www.importnew.com/14751.html/attachment/2011052709390013)

图2：对象之间复杂的依赖关系

耦合关系不仅会出现在对象与对象之间，也会出现在软件系统的各模块之间，以及软件系统和硬件系统之间。如何降低系统之间、模块之间和对象之间的耦合度，是软件工程永远追求的目标之一。**为了解决对象之间的耦合度过高的问题**，软件专家Michael Mattson提出了IOC理论，用来实现对象之间的“解耦”，目前这个理论已经被成功地应用到实践当中，很多的J2EE项目均采用了IOC框架产品Spring。

**2. 什么是控制反转(IoC)**

IOC是Inversion of Control的缩写，多数书籍翻译成“控制反转”，还有些书籍翻译成为“控制反向”或者“控制倒置”。  
1996年，Michael Mattson在一篇有关探讨面向对象框架的文章中，首先提出了IOC 这个概念。对于面向对象设计及编程的基本思想，前面我们已经讲了很多了，不再赘述，简单来说就是把复杂系统分解成相互合作的对象，这些对象类通过封装以后，内部实现对外部是透明的，从而降低了解决问题的复杂度，而且可以灵活地被重用和扩展。IOC理论提出的观点大体是这样的：借助于“第三方”实现具有依赖关系的对象之间的解耦，如下图：

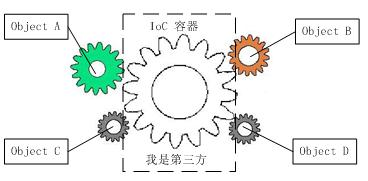
[](http://www.importnew.com/14751.html/attachment/2011052709391014)

图3：IOC解耦过程

大家看到了吧，由于引进了中间位置的“第三方”，也就是IOC容器，使得A、B、C、D这4个对象没有了耦合关系，齿轮之间的传动全部依靠“第三方”了，全部对象的控制权全部上缴给“第三方”IOC容器，所以，IOC容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系，这就是有人把IOC容器比喻成“粘合剂”的由来。

我们再来做个试验：把上图中间的IOC容器拿掉，然后再来看看这套系统：

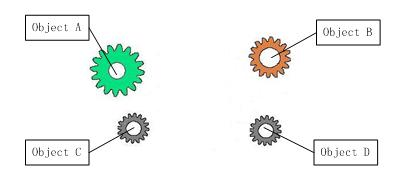
[](http://www.importnew.com/14751.html/attachment/2011052709392670)

图4：拿掉IoC容器后的系统

我们现在看到的画面，就是我们要实现整个系统所需要完成的全部内容。这时候，A、B、C、D这4个对象之间已经没有了耦合关系，彼此毫无联系，这样的话，当你在实现A的时候，根本无须再去考虑B、C和D了，对象之间的依赖关系已经降低到了最低程度。所以，如果真能实现IOC容器，对于系统开发而言，这将是一件多么美好的事情，参与开发的每一成员只要实现自己的类就可以了，跟别人没有任何关系！

我们再来看看，控制反转(IOC)到底为什么要起这么个名字？我们来对比一下：

软件系统在没有引入IOC容器之前，如图1所示，对象A依赖于对象B，那么对象A在初始化或者运行到某一点的时候，自己必须主动去创建对象B或者使用已经创建的对象B。无论是创建还是使用对象B，控制权都在自己手上。

软件系统在引入IOC容器之后，这种情形就完全改变了，如图3所示，由于IOC容器的加入，对象A与对象B之间失去了直接联系，所以，当对象A运行到需要对象B的时候，IOC容器会主动创建一个对象B注入到对象A需要的地方。

通过前后的对比，我们不难看出来：对象A获得依赖对象B的过程,由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

**3.  IOC的别名：依赖注入(DI)**

2004年，Martin Fowler探讨了同一个问题，既然IOC是控制反转，那么到底是“哪些方面的控制被反转了呢？”，经过详细地分析和论证后，他得出了答案：“获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由IOC容器主动注入。于是，他给“控制反转”取了一个更合适的名字叫做“依赖注入（Dependency Injection）”。他的这个答案，实际上给出了实现IOC的方法：注入。所谓依赖注入，就是由IOC容器在运行期间，动态地将某种依赖关系注入到对象之中。

所以，依赖注入(DI)和控制反转(IOC)是从不同的角度的描述的同一件事情，就是指**通过引入IOC容器，利用依赖关系注入的方式，实现对象之间的解耦**。

我们举一个生活中的例子，来帮助理解依赖注入的过程。大家对USB接口和USB设备应该都很熟悉吧，USB为我们使用电脑提供了很大的方便，现在有很多的外部设备都支持USB接口。

[](http://www.importnew.com/14751.html/attachment/2011052709393897)

图5：USB接口和USB设备

现在，我们利用电脑主机和USB接口来实现一个任务：从外部USB设备读取一个文件。

电脑主机读取文件的时候，它一点也不会关心USB接口上连接的是什么外部设备，而且它确实也无须知道。它的任务就是读取USB接口，挂接的外部设备只要符合USB接口标准即可。所以，如果我给电脑主机连接上一个U盘，那么主机就从U盘上读取文件；如果我给电脑主机连接上一个外置硬盘，那么电脑主机就从外置硬盘上读取文件。挂接外部设备的权力由我作主，即控制权归我，至于USB接口挂接的是什么设备，电脑主机是决定不了，它只能被动的接受。电脑主机需要外部设备的时候，根本不用它告诉我，我就会主动帮它挂上它想要的外部设备，你看我的服务是多么的到位。这就是我们生活中常见的一个依赖注入的例子。在这个过程中，**我就起到了IOC容器的作用**。

通过这个例子,依赖注入的思路已经非常清楚：当电脑主机读取文件的时候，我就把它所要依赖的外部设备，帮他挂接上。整个外部设备注入的过程和一个被依赖的对象在系统运行时被注入另外一个对象内部的过程完全一样。

我们把依赖注入应用到软件系统中，再来描述一下这个过程：

对象A依赖于对象B,当对象 A需要用到对象B的时候，IOC容器就会立即创建一个对象B送给对象A。IOC容器就是一个对象制造工厂，你需要什么，它会给你送去，你直接使用就行了，而再也不用去关心你所用的东西是如何制成的，也不用关心最后是怎么被销毁的，这一切全部由IOC容器包办。

在传统的实现中，由程序内部代码来控制组件之间的关系。我们经常使用new关键字来实现两个组件之间关系的组合，这种实现方式会造成组件之间耦合。IOC很好地解决了该问题，它将实现组件间关系从程序内部提到外部容器，也就是说由容器在运行期将组件间的某种依赖关系动态注入组件中。

**4.  IOC为我们带来了什么好处**

我们还是从USB的例子说起，使用USB外部设备比使用内置硬盘，到底带来什么好处？

第一、USB设备作为电脑主机的外部设备，在插入主机之前，与电脑主机没有任何的关系，只有被我们连接在一起之后，两者才发生联系，具有相关性。所以，无论两者中的任何一方出现什么的问题，都不会影响另一方的运行。这种特性体现在软件工程中，就是可维护性比较好，非常便于进行单元测试，便于调试程序和诊断故障。代码中的每一个Class都可以单独测试，彼此之间互不影响，只要保证自身的功能无误即可，这就是组件之间低耦合或者无耦合带来的好处。

第二、USB设备和电脑主机的之间无关性，还带来了另外一个好处，生产USB设备的厂商和生产电脑主机的厂商完全可以是互不相干的人，各干各事，他们之间唯一需要遵守的就是USB接口标准。这种特性体现在软件开发过程中，好处可是太大了。每个开发团队的成员都只需要关心实现自身的业务逻辑，完全不用去关心其它的人工作进展，因为你的任务跟别人没有任何关系，你的任务可以单独测试，你的任务也不用依赖于别人的组件，再也不用扯不清责任了。所以，在一个大中型项目中，团队成员分工明确、责任明晰，很容易将一个大的任务划分为细小的任务，开发效率和产品质量必将得到大幅度的提高。

第三、同一个USB外部设备可以插接到任何支持USB的设备，可以插接到电脑主机，也可以插接到DV机，USB外部设备可以被反复利用。在软件工程中，这种特性就是可复用性好，我们可以把具有普遍性的常用组件独立出来，反复利用到项目中的其它部分，或者是其它项目，当然这也是面向对象的基本特征。显然，IOC不仅更好地贯彻了这个原则，提高了模块的可复用性。符合接口标准的实现，都可以插接到支持此标准的模块中。

第四、同USB外部设备一样，模块具有热插拔特性。IOC生成对象的方式转为外置方式，也就是把对象生成放在配置文件里进行定义，这样，当我们更换一个实现子类将会变得很简单，只要修改配置文件就可以了，完全具有热插拨的特性。  
以上几点好处，难道还不足以打动我们，让我们在项目开发过程中使用IOC框架吗？

**5.  IOC容器的技术剖析**

IOC中最基本的技术就是“反射(Reflection)”编程，目前.Net C#、Java和PHP5等语言均支持，其中PHP5的技术书籍中，有时候也被翻译成“映射”。**有关反射的概念和用法，大家应该都很清楚，通俗来讲就是根据给出的类名（字符串方式）来动态地生成对象**。这种编程方式可以让对象在生成时才决定到底是哪一种对象。反射的应用是很广泛的，很多的成熟的框架，比如象Java中的Hibernate、Spring框架，.Net中 NHibernate、Spring.Net框架都是把“反射”做为最基本的技术手段。

反射技术其实很早就出现了，但一直被忽略，没有被进一步的利用。当时的反射编程方式相对于正常的对象生成方式要慢至少得10倍。现在的反射技术经过改良优化，已经非常成熟，反射方式生成对象和通常对象生成方式，速度已经相差不大了，大约为1-2倍的差距。

**我们可以把IOC容器的工作模式看做是工厂模式的升华，可以把IOC容器看作是一个工厂，这个工厂里要生产的对象都在配置文件中给出定义，然后利用编程语言的的反射编程，根据配置文件中给出的类名生成相应的对象。从实现来看，IOC是把以前在工厂方法里写死的对象生成代码，改变为由配置文件来定义，也就是把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性。**

**6.  IOC容器的一些产品**

Sun ONE技术体系下的IOC容器有：轻量级的有Spring、Guice、Pico Container、Avalon、HiveMind；重量级的有EJB；不轻不重的有JBoss，Jdon等等。Spring框架作为Java开发中SSH(Struts、Spring、Hibernate)三剑客之一，大中小项目中都有使用，非常成熟，应用广泛，EJB在关键性的工业级项目中也被使用，比如某些电信业务。

.Net技术体系下的IOC容器有：Spring.Net、Castle等等。Spring.Net是从Java的Spring移植过来的IOC容器，Castle的IOC容器就是Windsor部分。它们均是轻量级的框架，比较成熟，其中Spring.Net已经被逐渐应用于各种项目中。

**7. 使用IOC框架应该注意什么**

使用IOC框架产品能够给我们的开发过程带来很大的好处，但是也要充分认识引入IOC框架的缺点，做到心中有数，杜绝滥用框架。

第一、软件系统中由于引入了第三方IOC容器，生成对象的步骤变得有些复杂，本来是两者之间的事情，又凭空多出一道手续，所以，我们在刚开始使用IOC框架的时候，会感觉系统变得不太直观。所以，引入了一个全新的框架，就会增加团队成员学习和认识的培训成本，并且在以后的运行维护中，还得让新加入者具备同样的知识体系。

第二、由于IOC容器生成对象是通过反射方式，在运行效率上有一定的损耗。如果你要追求运行效率的话，就必须对此进行权衡。

第三、具体到IOC框架产品(比如：Spring)来讲，需要进行大量的配制工作，比较繁琐，对于一些小的项目而言，客观上也可能加大一些工作成本。

第四、IOC框架产品本身的成熟度需要进行评估，如果引入一个不成熟的IOC框架产品，那么会影响到整个项目，所以这也是一个隐性的风险。  
我们大体可以得出这样的结论：一些工作量不大的项目或者产品，不太适合使用IOC框架产品。另外，如果团队成员的知识能力欠缺，对于IOC框架产品缺乏深入的理解，也不要贸然引入。最后，特别强调运行效率的项目或者产品，也不太适合引入IOC框架产品，象WEB2.0网站就是这种情况

**1.首先，区分清楚什么是byType，什么是byName。**

**<bean id="userServiceImpl" class="cn.com.bochy.service.impl.UserServiceImpl" autowire="byName">**

**</bean>**

**<bean id="userDao" class="cn.com.bochy.dao.impl.UserDaoImpl">**

**</bean>**

**比如说如上这段代码，byName就是通过Bean的id或者name，byType就是按Bean的Class的类型。**

**若autowire="byType"意思是通过 class="cn.com.bochy.dao.impl.UserDaoImpl"来查找UserDaoImpl下所有的对象。**

**代码autowire="byName"意思是通过id="userDao"来查找Bean中的userDao对象**

**建议看 《Spring in Action》 第三章第一节，“自动装配 Bean 属性”**

**在spring中@Autowired注入规则：**

**1.@Autowired默认是按照byType进行注入的**

**二.spring注入的基本语法如下：**

**xml中语法如下：<bean id="beanId" class="包名.类名">**

**如果是属性注入，需要为每一个依赖类创建相应的getter和setter方法。**

**如果是构造方法注入，需要为依赖类创建相应的构造方法。**

**属性注入的语法如下：**

**<bean id="被注入的类的beanId" class="包名.类名" />**

**<bean id="beanId" class="包名.类名">**

**<property name="被注入的bean的名字" ref="被注入的类的beanId"></property>**

**</bean>**

**如例子：**

**<bean id="userService" class="com.uni2uni.spring.service.impl.UserService"><property name="userDao" ref="userDao"></property></bean>**

**<bean id="userDao" class="com.uni2uni.spring.dao.impl.UserDao" />**

**因为UserService依赖于UserDao，因此需要在userService中创建相应的getter和setter方法。**

### 18 消息中间件是如何实现的，技术难点有哪些？

以分布对象技术为基础，不仅能够支持应用集成框架的建立，满足协同工作的需求，而且建立了多层次的软构件框架，更加分布对象中间件便于应用领域框架及领域构件的开发。它也支持以构件形式实现集成平台的系统管理和公共服务，使系统具有良好的开放性和扩展性。基于CORBA标准的对分布对象的透明访问，允许应用对远程对象和本地对象使用相同的访问模式，从而屏蔽了操作系统平台和通信机制，使应用开发者更加关注于应用逻辑的开发。

### 19 如何搭建一个高可用系统？

今天又温习了一下《分布式java应用》，好多名词看了都知道，但记不住，是不习惯记这些不易理解的专业术语呀。因为和客户说的时候他肯定不懂。但和懂技术的客户或者专家进行沟通的时候都是用这些专业术语，这时候我知道但往往想不起来，看来老了啊。看来以后还得记些，不然显得本架构师不专业呀。

好了，言归正传，如何构建高可用的系统呢？

首先什么是高可用？“高可用性”（High Availability）通常来描述一个系统经过专门的设计，从而减少停工时间，而保持其服务的高度可用性。

1.ha

    1.1避免单点

        。负载均衡技术

        。热备

        。使用多机房

    1.2提高应用可用性

        1.2.1尽可能的避免故障

        1.2.2及时发现故障

              。报警系统

              。日志记录和分析系统

         1.2.3访问量和数据量不断上涨的应对策略

              。水平伸缩

              。拆分--1.应用拆分；2.拆分数据库；拆分表。

              。读写分离

              。垂直伸缩

              。其他

以上高级知识点看了两遍觉得还是得继续修炼，毕竟实战经验很少。

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

计算机系统的可靠性用平均无故障时间(MTTF)来度量，即计算机系统平均能够正常运行多长时间，才会发生一次故障。系统的可靠性能越高，平均无故障时间越长。可维护性用平均维修时间(MTTR)来度量，即系统发生故障后维修和重新恢复正常运行平均花费时间。系统的可维护性越好，平均维修时间越短。计算机系统的可用性定义为：MTTF/(MTTF+MTTR)\*100%。

举例来说，淘宝网在2010年成交额为300亿，则每分钟成交额为5—10万，那么对淘宝来说，其后台系统的高可用，对企业运营非常重要。淘宝数据负责人宁海元指出，淘宝系统，可用性至少需要99.999%。那么对于taobao.com系统，在一年365天，系统停止服务时间为5分15秒。

**高可用性的衡量指标**

可用性的计算公式： 　　%availability=（Total Elapsed Time－Sum of Inoperative Times）/ Total Elapsed Time 　  
　elapsed time为operating time+downtime。

TotalElapsed Time 为系统总时间，包括可提供服务时间+停止服务时间。

Sumof Inoperative Times 为停止服务时间，包括宕机时间+维护时间。 　　  
可用性和系统组件的失败率相关。衡量系统设备失败率的一个指标是“失败间隔平均时间”MTBF（mean time between failures）。  
通常这个指标衡量系统的组件，如磁盘。  
　　MTBF=Total Operating Time / Total No. of Failures 　　  
Operating time为系统在使用的时间（不包含停机情况）。

**高可用性系统的设计**

计系统的可用性，最重要的是满足用户的需求。系统的失败只有当其导致服务的失效性足以影响到系统用户的需求时才会影响其可用性的指标。用户的敏感性决定于系统提供的应用。例如，在一个能在1秒钟之内被修复的失败在一些联机事务处理系统中并不会被感知到，但如果是对于一个实时的科学计算应用系统，则是不可被接受的。

　　系统的高可用性设计决定于您的应用。例如，如果几个小时的计划停机时间是可接受的，也许存储系统就不用设计为磁盘可热插拔的。反之，你可能就应该采用可热插拔、热交换和镜像的磁盘系统。

　　所以涉及高可用系统需要考虑：

　　决定业务中断的持续时间。根据公式计算出的衡量HA的指标，可以得到一段时间内可以中断的时间。但可能很大量的短时间中断是可以忍受的，而少量长时间的中断却是不可忍受的。

　　在统计中表明，造成非计划的宕机因素并非都是硬件问题。硬件问题只占40%，软件问题占30%，人为因素占20%，环境因素占10%。您的高可用性系统应该能尽可能地考虑到上述所有因素。

　　当出现业务中断时，尽快恢复的手段。

**导致计划内的停机因素有：**

　　周期性的备份

　　软件升级

　　硬件扩充或维修

　　系统配置更改

　　数据更改

**导致计划外停机的因素有：**

　　硬件失败

　　文件系统满错误

　　内存溢出

　　备份失败

　　磁盘满

　　供电失败

　　网络失败

　　应用失败

　　自然灾害

　　操作或管理失误

　　通过有针对性的设计，可以避免上述全部或部分因素带来的损失。当然，100%的高可用系统是不存在的。

**创建高可用性的计算机系统**

在UNIX系统上创建高可用性计算机系统，业界的通行做法，也是非常有效的做法，就是采用群集系统（Cluster），将各个主机系统通过网络或其他手段有机地组成一个群体，共同对外提供服务。创建群集系统，通过实现高可用性的软件将冗余的高可用性的硬件组件和软件组件组合起来，消除单点故障：

　　消除供电的单点故障

　　消除磁盘的单点故障

　　消除SPU（System Process Unit）单点故障

　　消除网络单点故障

　　消除软件单点故障

　　尽量消除单系统运行时的单点故障

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

**1.2.1如何确保高可用（转载）**

可用性越高越好，提高可用性主要从一下几个方面入手：

(1)系统架构

(2)容灾性

(3)监控报警

(4)故障转移

**1.2.1.1 系统架构**

系统架构，指整个网站后台系统的架构。好的系统架构，主要从下面几个方面考虑：

(1)操作系统的选择，从稳定性、安全性和可维护性考虑，unix和linux性能远远好于windows，从成本考虑，Linux远远低于windows 和unix。

(2)负载均衡器的选择，硬件负载均衡器性能和稳定性高于软件负载均衡器。但成本上，软件比如haproxy、LVS优于硬件(比如F5、Netscaler)。

(3)web server的选择，Nginx优于传统的Apache。

(4)各级缓存的选择与应用，varnish、squid、memcached。

(5)网站开发语言的选择，与开发有关，www.linuxidc.com主要分为需要编译性的语言和不需要编译性的语言。

(6)数据库的选择，传统的关系数据库中，Oracle优于MySQL，但Oracle收费远远高于MySQL，实际上，Oracle有两种收费模式，一种是按用户数，一种是按主机处理器个数。而MySQL有免费的版本。

(7)底层存储设备的选择，比如机械磁盘和固态硬盘的选择。

(8)避免单点故障问题，在逻辑架构上，避免单点故障，避免出现割点。

**1.2.1.2 容灾性**

容灾性能对系统非常重要，比如服务器因为断电，导致数据文件的不一致，因为发生自然或者非自然灾害比如火灾导致的磁盘损坏，发生数据丢失等。所以容灾很重要，主要从以下几个方面提高容灾性能：

(1)服务器热备机的部署，当发生故障后，热备机能马上使用，提供服务。这里的服务器主要指web server 、应用服务器、数据库服务器等。

(2) 数据备份，比如做定期备份、热备份、增量备份，甚至需要做主从备份，来提高抗灾性能。并且从底层存储设备上进行备份，比如做RAID。

(3) 做双线网络交换，尽量优化设计网络，避免因为核心交换机故障，而影响服务。网络上避免单点故障。

**1.2.1.3 监控报警**

监控是指对在线服务和非服务的在线服务器和相应的进程进行状态检测，当出现宕机或者某项服务进程僵死之后，能够在尽量短的时间获得该信息，然后通过报警系统将信息发送到一线运维人员。所以，监控报警，直接影响宕机时间。监控报警，主要从以下几个方面展开：

(1)  监控主机CPU使用情况，负载情况。

(2)  监控主机内存使用情况。

(3)  监控主机IO外设，主要以磁盘为主。如磁盘的读写、磁盘使用量等。

(4)  监控主机网卡使用情况。网卡是否损坏，是否招到DDOS攻击。

(5)  监控应用进程，包括web server ，应用服务器等。

(6)  监控数据库使用情况。包括用户的请求数、缓存使用量等。

(7)  监控交换设备的使用情况。网络入、出的流量。

(8)  监控IDC机房温度、湿度等。

(9)  防火墙、入侵检测等安全检测、监控等。

通过上面的各项监控、得到相应数值，应用监控绘图软件，把相应的数值绘画出来，现有监控绘图软件有mrtg、cacti、nagios等。然后设置一个报警阈值，如果超过该阈值，那么通过报警系统，[www.linuxidc.com](http://www.linuxidc.com)比如短信、msn、邮件、甚至是声音完成报警功能。典型的报警系统如图3-2-1-3所示。

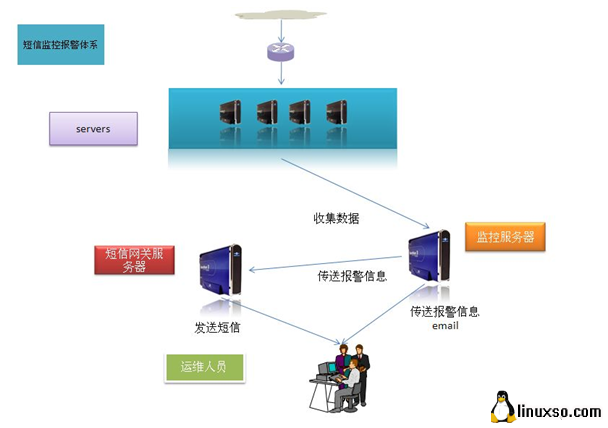
[](http://photo.blog.sina.com.cn/showpic.html#blogid=866b16a60100yh0u&url=http://s10.sinaimg.cn/orignal/866b16a6gbdf419ca3c89)

                                            图3-2-1-3

如图3-2-1-3所示，监控服务器从servers上收集系统信息，如果发现系统的某项状态指数超过预设的阈值，则发送邮件到运维人员。同时，把相应的报警信息发送到短信运营商的短信网关服务器，然后短信网关服务器发送短信到运维人员手机中，完成短信报警。上述报警过程，传送邮件报警信息，是基于TCP/IP协议，而传送短信报警信息，是基于gprs网络。

**1.2.1.4 故障转移**

故障转移是指，当对用户提供服务的服务器或者相应的应用进程发生故障后，比如服务器宕机、进程僵死之后，备用服务器能够在尽量短的时间内启用，提供服务。这样能够最大限度减少损失，保证用户的正常服务。所以，做好故障转移，要解决以下两个问题：

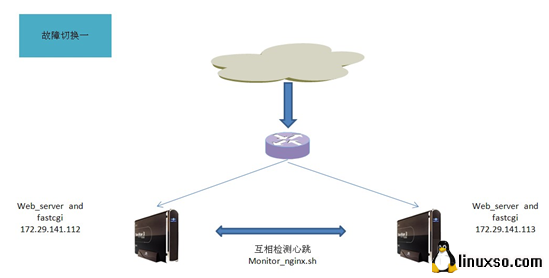
(1)  实时监测故障问题。

(2)  准确快速切换服务器问题。

针对不同层次的服务，监测机制也不同，详细情况，在3.2.1.3已经阐述。下面主要论述一下故障切换问题。

故障切换包括负载均衡器的故障切换、主机os的故障切换、web server的故障切换、应用进程的故障切换、数据库的故障切换、存储系统的故障切换、DNS的故障切换、交换设备的故障切换等。下面主要分析进程僵死的故障转移和服务器宕机的故障转移。

进程僵死故障转移案例，常见的web server僵死故障转移如图3-2-1-4所示。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=866b16a60100yh0u&url=http://s12.sinaimg.cn/orignal/866b16a6gbdf422981f5b)

如图3-2-1-4-1所示，当主机172.29.141.112的web server 对外提供服务时，通过在主机172.29.141.113上部署监控程序Monitor\_nginx.sh来监控主机172.29.141.112上面的web server进程运行情况，一旦发现172.29.141.112上web server停止服务，马上报警，先更改172.29.141.113的ip地址为172.29.141.112，再启用其自身的web server,完成故障转移。此外，也可以在两服务器上同时部署监控程序Monitor\_nginx.sh，完成互相监控。

服务器宕机故障转移案例，常见的服务器宕机故障转移，如图3-2-1-4-2所示。

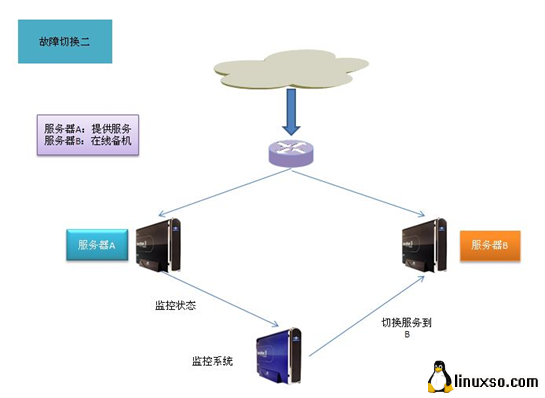
[](http://photo.blog.sina.com.cn/showpic.html#blogid=866b16a60100yh0u&url=http://s9.sinaimg.cn/orignal/866b16a6gbdf426d24458)

   图3-2-1-4-2

如图3-2-1-4-2所示，服务器A和服务器B同时部署，但服务器A提供服务，而服务器B作为热备机。监控系统单独部署。当服务器A宕机之后，监控系统会检测到这一信息，然后通过浮动更改服务器B的ip地址，完成故障切换。

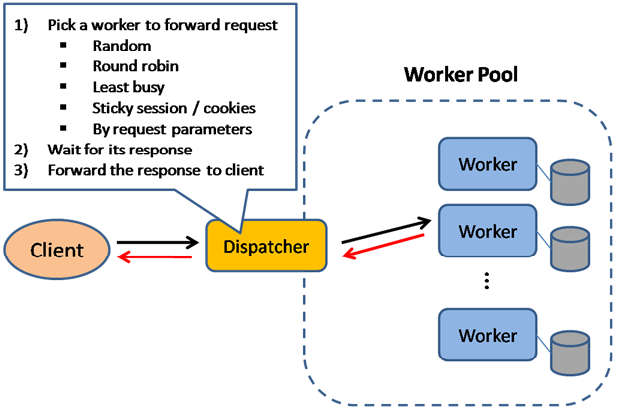
1.3 本文小结

本文主要阐述了网站后台系统的高可用性，分析了高可用性的定义和应用需求，重点阐述了如何做到高可用。通过从不同应用级别，如主机、存储、网络、外设、数据库、安全等各个级别进行分析，最后详细论述了web server的故障转移和主机系统的故障转移。

### 20 哪些设计模式可以增加系统的可扩展性？

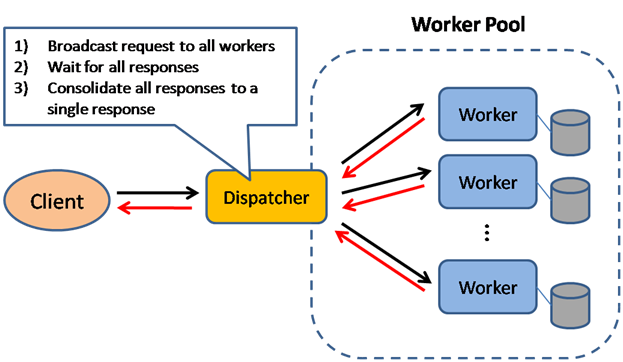
1、Load Balancer

该模式中，一个分发器基于某种策略确定由哪个worker实例处理请求。应用最好是无状态的，以使任何一个worker实例都能同等处理请求。大量的网站都会用到负载均衡器这个模式的。



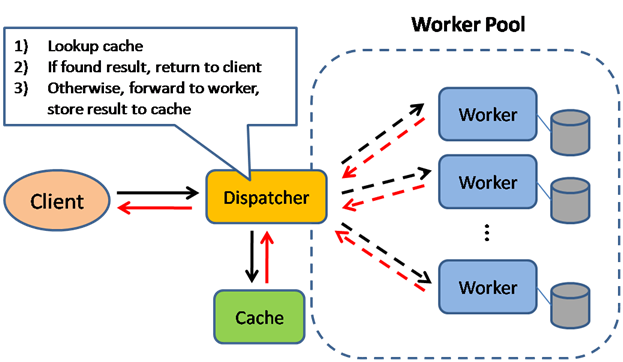
2、Scatter and Gather

该模式中，分发器将请求转发给多个worker实例，每个worker实例处理完返回给分发器，分发器将worker们返回的结果再加工后再返回给客户端。以搜索为例，通常得AS、BS架构就是这种典型模式。



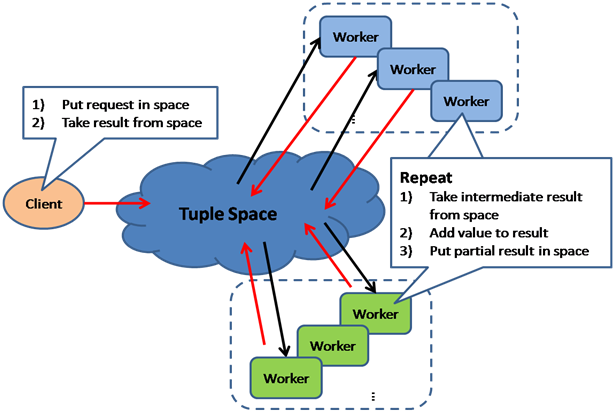
3、Result Cache

承接上个模式，这个模式只是在分发器处理时加了一步查询结果缓存，这都能算是模式！



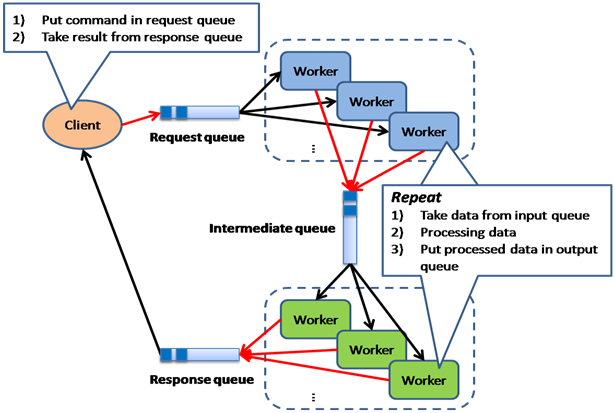
4、Shared Space

这个模式还有个更广泛的名字–“黑板模式”。实现来说，就是在处理流程中，存在一个全局传递的对象，它可能包含了请求参数、中间状态、响应结果等各种信息，供流程中的各个组件对其进行操作。在一些web框架和应用框架中，都可见这个模式的使用。



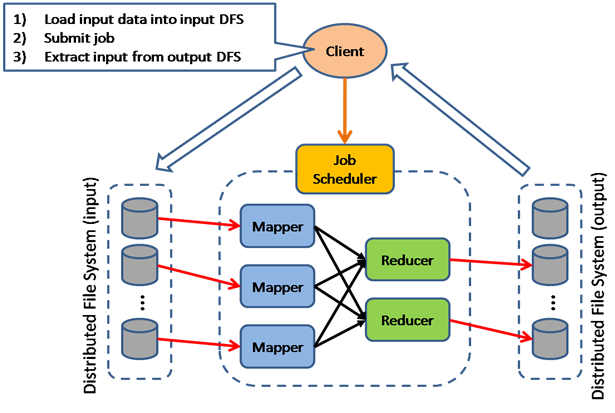
5、Pipe and Filter

这个模式又以“面向数据流编程”知名，是很通用的企业集成模式。



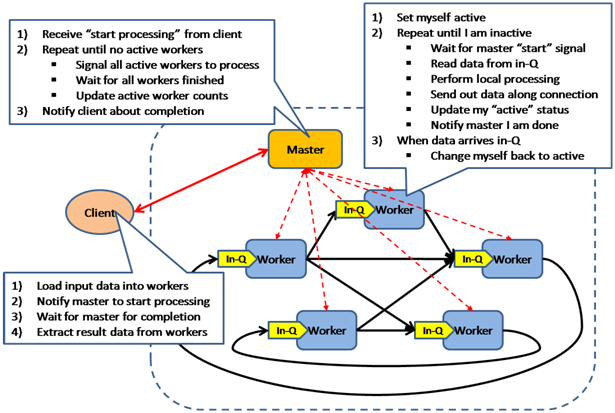
6、Map Reduce

因为google和hadoop，这个模式几乎都了解些，尽管多数人都没亲身应用过。



 7、Bulk Synchronous Parellel

该模型基于一个master协调，所有的worker同步（lock-step）执行。  
该模式被用于Google Pregel Graph Processing  [google-pregel-graph-processing](http://horicky.blogspot.com/2010/07/google-pregel-graph-processing.html) 和[Hama](http://incubator.apache.org/hama/)。



8、Execution Orchestrator

又一个不很了解的模式，似乎是一个和map reduce有一拼的分布式计算模型，似乎是微软的创造：[Microsoft’s Dryad project](http://research.microsoft.com/en-us/projects/dryad/ " \t "http://duanple.blog.163.com/blog/static/7097176720114741728720/_blank)。

### 21 介绍设计模式，如模板，命令，策略，适配器、桥接、装饰，观察者，状态，访问者？

创建型模式共5种：

工厂模式（Factory Pattern）

抽象工厂模式（Abstract Factory Pattern）

单例模式（Singleton Pattern）

建造者模式（Builder Pattern）

原型模式（Prototype Pattern）

结构型模式共8种：

适配器模式（Adapter Pattern）

桥接模式（Bridge Pattern）

过滤器模式（Filter、Criteria Pattern）

组合模式（Composite Pattern）

装饰器模式（Decorator Pattern）

外观模式（Facade Pattern）

享元模式（Flyweight Pattern）

代理模式（Proxy Pattern）。

行为型模式共12种：

责任链模式（Chain of Responsibility Pattern）

命令模式（Command Pattern）

解释器模式（Interpreter Pattern）

迭代器模式（Iterator Pattern）

中介者模式（Mediator Pattern）

备忘录模式（Memento Pattern）

观察者模式（Observer Pattern）

状态模式（State Pattern）

空对象模式（Null Object Pattern）

策略模式（Strategy Pattern）

模板模式（Template Pattern）

访问者模式（Visitor Pattern）

JAVA EE型模式共8种：

MVC 模式（MVC Pattern）

业务代表模式（Business Delegate Pattern）

组合实体模式（Composite Entity Pattern）

数据访问对象模式（Data Access Object Pattern）

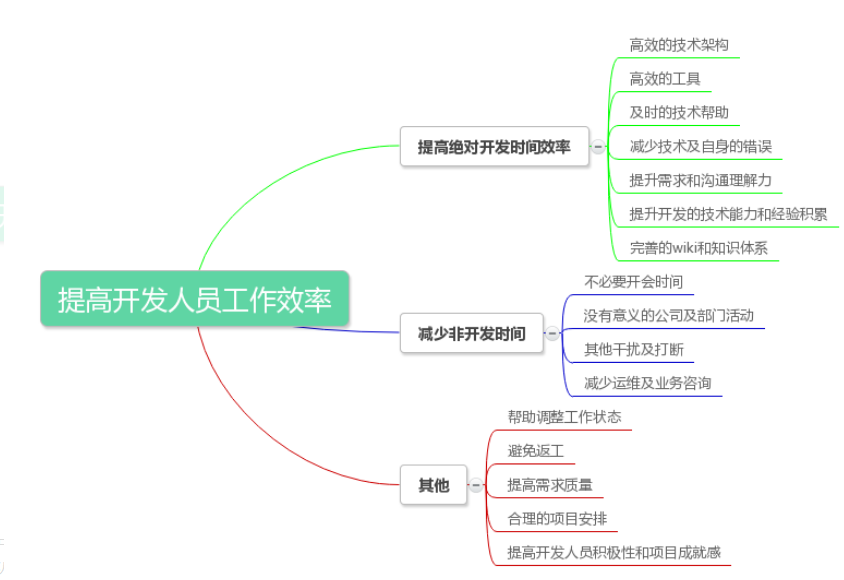
前端控制器模式（Front Controller Pattern）

拦截过滤器模式（Intercepting Filter Pattern）

服务定位器模式（Service Locator Pattern）

传输对象模式（Transfer Object Pattern）

### 22 怎么提高研发效率？



### 23 什么是高内聚低耦合，请举例子如何实现？

高内聚就是说相关度比较高的部分尽可能的集中，不要分散

低耦合就是说两个相关的模块尽可以能把依赖的部分降低到最小，不要让两个系统产生强依赖。

例子：

比如订单系统，订单是跟库存息息相关的，没有库存就没有订单，订单强依赖库存

如果我们把如何扣减库存的逻辑做在订单系统中，那么订单系统和库存系统就耦合了

同样，订单系统中有扣减库存的逻辑会造成订单系统的功能比较分散，订单系统的功能就不够集中。

所以我们把订单中的库存逻辑给拆分出来一个独立的库存系统，对外暴露扣减库存的接口，订单系统可以通过这个暴露的库存接口来扣减库存

将订单逻辑与库存逻辑的依赖降低到最小，减掉了订单系统与库存的耦合

同事订单系统只包含订单处理的逻辑，库存系统只包含了库存的处理逻辑，两个系统的业务上更加内聚。

### 24 什么情况用接口，什么情况用消息？

抽象类：只有一个或几个方法需要定义成抽象方法，从而让子类来实现；其他方法有清晰、明确、共同的逻辑需要在本类中实现，由子类“共享”和“复用”。

接口：纯粹定义方法接口，所有方法都由实现该接口的类来实现。没有共享的公共逻辑，此时用接口比较合适。

消息：一个信息包含两个因素：消息描述（用于定义诸如消息传输目标等）和数据信息（如应用程序数据或数据库查询等）。程序之间的通讯通过传递消息而非直接调用程序。如简单对象访问协议（SOAP）、电子数据交换（EDI）、C、COBOL和XML都是常见的消息类型实现。

队列：一个安全的信息存储区。因为消息存放在队列中，所以应用程序可以相互独立的运行，以不同的速度，在不同的时间，在不同的地点。

消息传输系统：用于确保队列之间的信息提供，包括网络中不同系统上的的远程队列之间的信息提供。并保证网络故障或关闭后的恢复。

应用程序接口：应用程序和消息系统之间通过WebSphere MQ API实现交互操作的接口。WebSphere MQ API在所有WebSphere MQ平台上是一致的。API只有十几个调用，2个关键动词：发送（PUT）和接收（GET）。

发布/订阅模式进一步提升了消息队列的扩展性灵活性，双方无需关注具体发送单位、接收单位，只需要关注主题级即可

### 25 如果AB两个系统互相依赖，如何解除依赖？

spring IOC 模式，降低耦合性，中介者模式

### 26 如何写一篇设计文档，目录是什么？

需求、分析设计

1.需求

1.1 项目背景

1.2 需求描述

2.分析设计

2.1 需求分析

三件事：

1. 了解用户的需求，并用软件工程师的语言描述出来。

2. 让用户了解我们能做出来的是什么。

3. 提出合理化的建议。

2.2 模块划分

模块划分的目的：

1 分类后的功能是比较清晰；

2 分类后可以对开发进行计划起指导作用；

3 分类后有利于面向对象的分析与设计；

模块分类的方式：

1 按角色分 ；按角色分类结果决定了界面操作：

2 按功能分；按功能分类结果决定了操作类内容

3 按信息分；按信息分决定了数据库架构

2.3 概要设计

面向对象的概要设计目的有以下几点：

以需求分析为基础，形成程序的功能流程。

以功能流程为基础，设计流程中传递的数据。

设计类与包，用于实现功能。

那么，如何完成概要设计，实现概要设计的目的呢？首先要从需求分析入手。在需求分

析阶段分析出了所有的功能，然后划分模块。其中有一种模块中按信息来划分的，方便我们

找出系统中数据，也就是数据库架构。

### 27 什么场景应该拆分系统，什么场景应该合并系统？

拆分系统：

当系统通过集群的方式已经无法解决性能问题的时候，或者业务扩展到很大的时候，需要把拆分系统

按照业务的方式垂直拆分：将业务功能结合比较紧密的部分拆分成独立的系统，独立维护

按照性能瓶颈点拆分：将系统性能瓶颈点拆分出一个独立的系统，可以针对这个独立的系统集群部署，增加可伸缩性，提高系统整体的性能

合并系统：

或者系统间通过跨进程访问的性能损耗过高，可以将系统合并成一个系统，减少跨进程访问的消耗

### 28 [系统和模块的区别，分别在什么场景下使用](http://www.cnblogs.com/vindia/p/7967685.html)？

系统是一个完整功能的系统，拥有独立的访问方式，和部署方式，拥有完整的生命周期，系统由模块组成

模块是系统的组成部分，不能单独工作，需要依附于系统才能发挥作用，通常是解决一定场景下的问题

### 29 分布式锁的实现方式有哪几种？

在进行大型网站技术架构设计以及业务实现的过程中，多少都会遇到需要使用分布式锁的情况。那么问题也就接踵而至。分布式锁zk和memcached以及redis三者都能实现，同样是分布式锁，三者的区别何在？各自适用什么场景？

一、Zookeeper

1、实现原理：

基于zookeeper瞬时有序节点实现的分布式锁，其主要逻辑如下（该图来自于IBM网站）。大致思想即为：每个客户端对某个功能加锁时，在zookeeper上的与该功能对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

2、优点

锁安全性高，zk可持久化，且能实时监听获取锁的客户端状态。一旦客户端宕机，则瞬时节点随之消失，zk因而能第一时间释放锁。这也省去了用分布式缓存实现锁的过程中需要加入超时时间判断的这一逻辑。

3、缺点

性能开销比较高。因为其需要动态产生、销毁瞬时节点来实现锁功能。所以不太适合直接提供给高并发的场景使用。

4、实现

可以直接采用zookeeper第三方库curator即可方便地实现分布式锁。

5、适用场景

对可靠性要求非常高，且并发程度不高的场景下使用。如核心数据的定时全量/增量同步等。

二、memcached分布式锁

1、实现原理

memcached带有add函数，利用add函数的特性即可实现分布式锁。add和set的区别在于：如果多线程并发set，则每个set都会成功，但最后存储的值以最后的set的线程为准。而add的话则相反，add会添加第一个到达的值，并返回true，后续的添加则都会返回false。利用该点即可很轻松地实现分布式锁。

2、优点

并发高效

3、缺点

memcached采用列入LRU置换策略，所以如果内存不够，可能导致缓存中的锁信息丢失。

memcached无法持久化，一旦重启，将导致信息丢失。

4、使用场景

高并发场景。需要1)加上超时时间避免死锁;2)提供足够支撑锁服务的内存空间;3)稳定的集群化管理。

三、redis分布式锁

redis分布式锁即可以结合zk分布式锁锁高度安全和memcached并发场景下效率很好的优点，其实现方式和memcached类似，采用setnx即可实现。需要注意的是，这里的redis也需要设置超时时间。以避免死锁。可以利用jedis客户端实现。参考资料Redis实现分布式锁。

### 30 分布式session的实现方式有哪几种？

一、分布式Session的几种实现方式

1.基于数据库的Session共享

2.基于NFS共享文件系统

3.基于memcached 的session，如何保证 memcached 本身的高可用性？

4.基于resin/tomcat web容器本身的session复制机制

5.基于TT/Redis 或 jbosscache 进行 session 共享。

6.基于cookie 进行session共享

或者是：

一、Session Replication 方式管理 (即session复制)

简介：将一台机器上的Session数据广播复制到集群中其余机器上

使用场景：机器较少，网络流量较小

优点：实现简单、配置较少、当网络中有机器Down掉时不影响用户访问

缺点：广播式复制到其余机器有一定廷时，带来一定网络开销

二、Session Sticky 方式管理

简介：即粘性Session、当用户访问集群中某台机器后，强制指定后续所有请求均落到此机器上

使用场景：机器数适中、对稳定性要求不是非常苛刻

优点：实现简单、配置方便、没有额外网络开销

缺点：网络中有机器Down掉时、用户Session会丢失、容易造成单点故障

三、缓存集中式管理

简介：将Session存入分布式缓存集群中的某台机器上，当用户访问不同节点时先从缓存中拿Session信息

使用场景：集群中机器数多、网络环境复杂

优点：可靠性好

缺点：实现复杂、稳定性依赖于缓存的稳定性、Session信息放入缓存时要有合理的策略写入

### 31 消息发送一致性解决方案有哪些？

消息发送一致性

是指产生消息的业务动作与消息发送的一致。（如果业务操作成功，那么由这个业务操作所产生的消息一定要成功投递出去，否则就丢消息）

消息发送一致性如何保障：

场景：

1.业务处理成功，执行发送消息的时候 应用故障，导致没有发送消息（后续服务没有收到消息处理业务，结果数据不一致）

2.业务处理成功，执行发送消息的时候，消息系统（MQ）故障，导致消息发送失败（后续服务没有收到消息处理业务，结果数据不一致）

解决方案：

1.主动方应用先把消息发给消息中间件，消息状态标记为“待确认”

2.消息中间件收到消息后，把消息持久化到消息存储中，但并不向被动方应用投递消息

3.消息中间件返回消息持久化结果（成功/失败），主动方应用根据返回结果进行判断如何进行业务操作处理

a：失败，放弃业务操作处理，结束（必要时向上层返回失败结果）

b：成功，执行业务操作处理

4.业务操作完成后，把业务操作结果（成功/失败）发送给消息中间件

5.消息中间件收到业务操作结果后，根据业务结果进行处理

a：失败：删除消息存储的消息，结束

b：成功：更新消息存储的消息状态为“待发送（可发送）”，紧接着执行消息投递

6.确保了主动方应用业务处理成功就一定会发送消息

7.被动方应用监听并接收“待发送状态的消息”执行业务处理

8.业务处理完成后向消息中间件发送ACK确认，确认消息已经收到（消息中间件将从队列中删除该消息）

消息发送一致性流程中的异常点

异常情况：

主动方

1.预发送消息失败：

消息未进行存储，业务操作未执行（可能的原因：主动方应用、网络、消息中间件、消息存储）【一致】

2.预发送消息后，主动方应用没有收到返回消息存储的结果：

a：消息未进存储，业务操作未执行【一致】

b：消息已进存储（待确认），业务操作未执行【不一致：待确认消息已存储，但是业务却没执行】

3.收到消息存储成功的返回结果，但未执行业务操作就失败

消息已进存储（待确认），业务操作未执行【不一致：待确认消息已存储，但是业务却没执行】

消息中间件

1.消息中间件没有收到主动方应用的业务操作处理结果

a：消息已进存储（待确认），业务操作未执行成功（或业务操作出错回滚了）【不一致：待确认消息已存储，但是业务失败】

b：消息已进存储（待确认），业务操作成功（主动方网络中断）【不一致：待确认消息已存储，但是业务操作成功，但是没有修改消息状态为待发送】

2.消息中间件收到业务操作结果（成功/失败），但处理消息存储中的消息状态失败

a：消息已进存储（待确认），业务操作未执行成功（或业务操作出错回滚了）,告知消息中间件失败(MQ操作消息状态失败)【不一致：业务失败，未删除消息】

b：消息已进存储（待确认），业务操作成功,告知消息中间件成功(MQ更新消息状态失败)【不一致：业务成功，消息状态未更新成待发送】

总结：

1.消息未进存储，业务操作未执行【一致】

2.消息已进存储（状态待确认）业务操作未执行或失败【不一致】

异常解决方案：删除消息

3.消息已进存储（状态待确认）业务操作成功【不一致】

异常解决方案：更新消息状态

异常处理还是会有异常

解决方案：消息持久化，创建定时任务查询消息中间件查询状态为“待确认”的消息调用主动方应用的业务操作结果查询接口，返回业务处理结果（成功/失败）去更新消息状态

被动方

消息重复发送的原因

1.被动方应用接收到消息，业务处理完成后应用出问题，消息中间件不知道消息处理结果，会重新投递消息

2.被动方应用接收到消息，业务处理完成后网络出问题，消息中间件不知道消息处理结果，会重新投递消息

3.被动方应用接收到消息，业务处理时间过长，消息中间件因消息超时未确认，会再次投递消息

4.被动方应用接收到消息，业务处理完成，消息中间件问题导致收不到消息处理结果，消息会重新投递

5.被动方应用接收到消息，业务处理完成，消息中间件收到了消息处理结果，但由于消息存储故障导致消息没能确认，消息会再次投递

总结：消息消费过程中产生消息重复发送主要是因为消息接收者成功处理完消息后，消息中间件没能及时更新消息投递状态（也就是消息没能及时ACK确认）导致的

解决方案：

消息重复发送无法解决，通过被动方应用实现幂等性设计（任意多次执行所产生的影响均与一次执行的影响相同）

1.通过业务操作本身实现幂等性

2.系统缓存所有请求与处理结果 检测到重复请求后，自动返回之前的处理结果

极端情况：

消息重发也得有次数限制，要不然就变成了死循环，对于超过重发限制的消息，进入DLQ（死亡队列），等待人工干预或延后定期处理

实现方式：

1.本地消息服务

1.在主动方应用业务操作中，用一个本地事务将业务处理和消息确认数据存放在本地库中，确保业务完成 本地一定有一条“待确认的消息数据”

2.主动方应用业务处理成功后，发送消息至MQ中

3.被动方应用采用自动ACK确认方式，处理业务（需加幂等性设计）

4.业务处理成功后调用主动方消息确认接口（或再加一个消息确认的MQ，只是业务会相对更复杂），修改消息数据状态，完成整体流程

5.消息恢复系统定时轮询一段时间未确认的消息数据，从新放置再MQ中继续消费

优点：

1.消息数据的可靠性不依赖于MQ，弱化了多MQ的依赖

2.方案轻量级，容易实现

缺点：

1.业务和消息耦合性高，不可共用

2.消息数据与业务数据同库，占用资源

3.业务系统在使用关系型数据库的情况下，消息服务性能会受到关系型数据库并发性能的局限

2.独立消息服务

1.主动方应用系统调用消息服务系统预发送消息

2.获得返回结果后处理业务操作，发送结果告知消息服务系统

3.消息服务系统根据业务处理结果（成功/失败）修改消息状态为“待发送”

异常情况：

业务处理成功，但是与消息系统确认消息状态失败

通过消息状态确认子系统定时轮询未确认的消息去主动请求主动方应用获得业务处理结果更新消息状态

4.被动方系统接收消息，ACK确认删除MQ中的消息，处理业务逻辑完成后

5.调用消息系统确认消息已被成功消费（更新消息状态或删除消息记录）

异常情况：

业务处理成功后没有修改成功消息系统此条消息被成功消费状态

通过消息恢复子系统定时轮询成功消费的消息从新放到MQ队列中去重做直到成功

被动方采用幂等性设计保证多次消费的结果一致性

优点：

1.消息服务独立部署，独立维护、独立伸缩

2.消息存储可以按需选择不同的数据库来集成实现（关系型数据库或者nosql）

3.可复用，可以被相同的使用场景共用

4.消息数据的可靠性不依赖于MQ，弱化了多MQ的依赖

5.降低了业务系统和消息系统间的耦合，有利于系统的扩展维护

缺点：

1.一次消息发送需要两次请求，时效性偏低一点

2.主动方应用系统需要实现业务操作状态校验查询接口

消息状态确认子系统设计

1.系统集成了消息系统查询“未确认”数据接口服务，主动方查询业务状态接口服务

2.消息系统查询“未确认“数据接口，查询条件：

状态“未确认“

查询升序（ASC，先处理最早的数据）

指定固定数量（防止数据量过多一次查询出来）

并且消息存放时间大于一定时间（防止拿到新数据）

3.遍历拿到的“未确认“消息数据，调用主动方查询业务状态接口查询业务状态或者未生成业务数据

a)如果业务状态为成功，则修改消息状态为“待发送”

b)如果业务状态为失败或者就没执行业务，则删除消息

消息恢复子系统设计

1.系统集成了消息系统查询“未消费“数据接口服务

2.消息系统查询“未消费“数据接口，查询条件：

状态“未消费“

查询升序（ASC，先处理最早的数据）

指定固定数量（防止数据量过多一次查询出来）

并且消息存放时间大于一定时间（防止拿到新数据）

未死亡的消息（重试多次后还失败的消息会放入死亡队列）

3.遍历拿到的“未消费“消息数据

a)判断消息重试次数是否达到最多重试次数，如果达到则标记此消息死亡（防止消息一直重试）

b)根据消息重试次数判断是否达到了消息间隔

如第1次失败1分钟后就可以尝试重试，第5次失败则可以30分钟后再尝试，第6次就标记为死亡

c)重发消息

消息服务子系统设计

1.存储预发送消息

2.确定并发送消息

3.查询确认超时的消息（一直处于“未发送”状态的消息）

4.确认已被成功消费消息

5.查询消费确认超时的消息（一直处于“未消费”状态的消息）

6.删除消息

实时消息服务子系统设计（操作MQ）

1.修改mq重试次数为1或者0，将重试的机制交给消息恢复子系统去处理，减少频繁短时间内多次重试带来的效率问题

### 32 如何理解正向代理（客户端代理）和反向代理（服务器端代理）？

       反向代理（Reverse Proxy）方式是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个服务器。

概述：

       通常的代理服务器，只用于代理内部网络对Internet的连接请求，客户机必须指定代理服务器,并将本来要直接发送到Web服务器上的http请求发 送到代理服务器中。由于外部网络上的主机并不会配置并使用这个代理服务器，普通代理服务器也被设计为在Internet上搜寻多个不确定的服务器,而不是 针对Internet上多个客户机的请求访问某一个固定的服务器，因此普通的Web代理服务器不支持外部对内部网络的访问请求。当一个代理服务器能够代理 外部网络上的主机，访问内部网络时，这种代理服务的方式称为反向代理服务。此时代理服务器对外就表现为一个Web服务器，外部网络就可以简单把它当作一个 标准的Web服务器而不需要特定的配置。不同之处在于，这个服务器没有保存任何网页的真实数据，所有的静态网页或者CGI程序，都保存在内部的Web服务 器上。因此对反向代理服务器的攻击并不会使得网页信息遭到破坏，这样就增强了Web服务器的安全性。

　   反向代理方式和包过滤方式或普通代理方式并无冲突，因此可以在防火墙设备中同时使用这两种方式，其中反向代理用于外部网络访问内部网络时使用，正向代理或 包过滤方式用于拒绝其他外部访问方式并提供内部网络对外部网络的访问能力。因此可以结合这些方式提供最佳的安全访问方式。

为何叫反向：

        但是这也没法解释为何他叫反向，从原理上来说.代理服务器都是处理来自客户端的请求，并将其转发到目的服务器上，所以代理服务器的工作并没有任何反向的意味，而下面这张图就能说明为何反向代理叫反向

对了，从结构上来看，正向代理和反向代理的左右两边换了一下，原本代理服务器的客户端来自内网.其和代理服务器组成一个LAN，而反向代理之后.代 理服务器和服务器组成了一组.所以从结构来看,是reverse的，从英文的角度来看，reverse这个词包含翻转的意思,其意思就是指结构图上的这种 反转，但是由于翻译的关系.到了中文变成反向代理.其实proxy并没有做和以前有什么不同的事情.它仍旧是将来自客户端的请求转发给实际端.，而时至今 日,由于CDN的大量使用,反向代理后面的服务器为了适应这种跨网络的架构，而均采用实际IP地址.这样就往往更不容易使得大家对”反向代理这个名词中的反向到底指什么”这个问题有个清晰的答案了。

### 33 CDN实现原理是什么？

CDN发展背景

像我们所熟知的腾讯、新浪、网易这样的大网站，每天都有成千上百万的流量，他们要保持全国每个地方打开网站的速度都很快，这要怎么做到呢？这其中有CDN不小的功劳！

什么是CDN

我们先来看下CDN的含义，CDN即内容分发网络，加速的意思。那么网站CND服务是网站加速服务。我们可以通俗的了解成CDN可以加速网站的打开速度。

CDN加速原理

CDN是怎么做到加速的呢？其实这是CDN服务商在全国各个省份部署计算节点， CDN加速将网站的内容缓存在网络边缘，然后在用户访问网站内容的时候，通过调度系统将用户的请求路由或者引导到离用户接入网络最近或者访问效果的缓存服务器上，有该缓存服务器为用户提供内容服务；相对于直接访问源站，这种方式缩短了用户和内容之间的网络距离，从而达到加速的效果。

CDN具有哪些特点

1、本地加速，提高了企业站点(尤其含有大量图片和静态页面站点)的访问速度，并大大提高以上性质站点的稳定性。

2、镜像服务 消除了不同运营商之间互联的瓶颈造成的影响，实现了跨运营商的网络加速，保证不同网络中的用户都能得到良好的访问质量。

3、远程加速 远程访问用户根据DNS负载均衡技术 智能自动选择Cache服务器，选择最快的Cache服务器，加快远程访问的速度。

4、带宽优化 自动生成服务器的远程Mirror(镜像)cache服务器，远程用户访问时从cache服务器上读取数据，减少远程访问的带宽、分担网络流量、减轻原站点WEB服务器负载等功能。

5、集群抗攻击 广泛分布的CDN节点加上节点之间的智能冗于机制，可以有效地预防黑客入侵以及降低各种D.D.o.S攻击对网站的影响，同时保证较好的服务质量。

### 34 [怎么提升系统的QPS和吞吐量](http://www.cnblogs.com/vindia/p/7545630.html)？

简单而言通过增加集群来提升qps和吞吐量

实际上要比这个要复杂

首先我们需要知道系统的瓶颈

我们所知道的系统拓扑架构

对于rest接口而言

系统设施依次是：

dns

nginx

tomcat

db/soa

首先我们可以通过增加集群来增加qps和吞吐量

其次考虑到负载均衡的问题，我们可以通过其他设施来保证集群节点的负载均衡，进一步提高系统qps

于是就有nginx集群+负载均衡

tomcat集群+负载均衡

到db/soa这一层的时候，同样也可以通过增加集群+负载均衡的方式来解决

我们还可以在每一层增加缓存来应对热点数据

然而另外一个方面，可以系统拆分，服务拆分，分别针对瓶颈的系统单独增加集群和负载均衡来解决

同样db也可以分库分表，

因为单表超过1000万条数据时就很慢了，所以这个时候就需要库拆分，于是就有垂直拆分，水平拆分。

异步化，可以不同调用的异步化，使用mq，比如发送短信，发送邮件等

综上所述：

集群+负载均衡

增加缓存

系统拆分

分库分表

垂直拆分+水平拆分

异步化+MQ

### 35 线上问题出现内存泄露，CPU利用率标高，应用无响应时如何处理的？

java程序 cup使用率过高，会导致程序运行速度变慢，导致系统崩溃等

原因是多向的。跟内存泄漏,数据库等都有关

检查是否有死循环。

频繁的GC.或者有大量的线程。

算法太复杂或者太多

数据库连接的资源未释放或未关闭，

数据库connection过于频繁。

### 36 开发中有没有遇到什么技术问题？如何解决的？

axis服务连接被重置；activiti异步任务；springboot 引入es; 报表查询缓慢

### 37 有几十亿的名单，每天白天需要高并发查询，晚上需要更新一次，如何设计这个功能？

看下高并发网站更新数据的方式是如何做的？如下是收集cnblogs博主的文章。

很多Web系统的瓶颈在网络IO，所以很多系统都采用多Web服务器负载均衡，双DB做双机热备（其实就是只有一个DB，两台只有一台真正工作，死掉一台另一台顶上）的方式部署，在这个时候很多原本不是问题的系统也会产生很多的问题。

这里我们假设有表Product，其定义如下：

|  |  |  |
| --- | --- | --- |
| 列明 | 类型 | 说明 |
| Id | Int | 自增字段，实例的ID |
| ProductName | Varchar(100) | 商品的名称 |
| StoreCount | int | 库存数量 |
| 。。。 | 。。。 | 。。。 |

假设很不凑巧的，3个管理员P1,P2,P3同时操作了这个表，且P1 update StoreCount=50,P2 update StoreCount=49,P3 update StoreCount=48。这个时候问题就来了，如果是让他们都同时提交进去，当然没问题，但是如果这个时候N个Web程序在读的时候就会产生每台服务器上读出来的数据都可能不一样，A服务器读出来是48，B服务器读出来是50，C服务器读出来是49。

如果我们采用数据库锁可以避免这个问题，但是随之而来的是系统效率降低和无可避免的异常，而hibernate等实现的乐观锁呢，呵呵，对不起了，在多Web服务器的时候还能起作用吗？

由此产生了以下的解决方案：

和乐观锁的实现相反，我们不反对任何一个客户端的提交，乐观锁对读取的数据增加版本号，那么这个解决方案中对提交的数据增加“版本号”其实也就是时间戳。针对上面的Product表作为例子，为了实现无锁的提交，我们需要增加一个表Product\_Dirty，以后我们将称其为脏表，Product表我们称之为主表。脏表的结构和主表几乎完全一致，只是增加了一个时间戳字段用于记录详细的插入时间：

|  |  |  |
| --- | --- | --- |
| 列明 | 类型 | 说明 |
| Timespan | Int | 时间戳，精确到毫秒（能到纳秒更好） |
| Id | Int | 实例的ID(这里就不是自增字段了) |
| ProductName | Varchar(100) | 商品的名称 |
| StoreCount | int | 库存数量 |
| 。。。 | 。。。 | 。。。 |

在发生任何update的时候都将数据直接插入这个表，不要犹疑，没锁，所以可以快速的，尽情的插入数据。这里还是保持最初的假设，P1，P2，P3同时修改，所以插入了三条数据。所谓的同时插入其实在毫秒这个级别还是有差距的，所以三条记录的时间戳是不同的。好了这个时候数据进来了，但是主表的列数据还是没有改变，先在假设A服务器和B，C服务器都同时开始读数据了。在主表的时候，如果发现脏表有数据则表明主表数据为脏（已经修改过了）这个时候我们就开始合并数据，当然这个操作是需要在一个事务里实现。合并的操作其实很简单，就是取时间戳最大的（也就是最近一次修改）更新主表的数据，同时删掉脏表里的和主表ID相等的所有数据。如果发现主表关联的脏表没数据，那么就说明主表数据正常，就直接读取主表的内容。

此解决方案来自电信营帐系统的设计，由于省电信众多系统都是由分布很广的地市州电信业务人员操作，所以修改的时候经常存在本文要解决的问题，由于操作的人多，锁表的话会造成严重的拥塞，故产生了这个解决方案，由于电信的业务需要后台跑了一个服务来合并数据，并且每秒定时运行，故每秒为一个业务周期。我将其修改成在读取的时候合并，更加灵活一些。

好处：不用锁表，乐观锁也不用，可以在N多服务器操作的时候使用，且大家都不会报错，简化了异常处理。

坏处：增加了表，结构复杂，如果是用于修改原有业务如果只是几个关键表的话还好，全部都采用这个方式工作量巨大（好在电信不缺钱）。

弱点：和乐观锁类似，在某些场景下仍然可能脏读，所以如果对这方面有很高的要求，还是用悲观锁吧。

### 38 新浪微博中的推送消息给客户端在技术上如何实现的？

通常情况下，无论是web浏览器还是移动app，我们与服务器之间的交互都是主动的，客户端向服务器端发出请求，然后服务器端返回数据给客户端，客户端浏览器再将信息呈现，客户端与服务端对应的模式是: 客户端请求--服务端响应，这种机制对于信息变化不是特别频繁的应用尚可，但对于实时要求高、海量并发的应用来说显得捉襟见肘，尤其在当前业界移动互联网蓬勃发展的趋势下，高并发与用户实时响应是 Web 应用经常面临的问题，比如金融证券的实时信息，Web 导航应用中的地理位置获取，社交网络的实时消息推送，新闻的订阅,天气的提醒等。这些情况下，需要服务器主动推送消息给客户端。

那么在这样的模式下,会有几个问题需要我们思考下:

1.应用服务器如何确定每一个应用所在的设备

2.服务器端是如何将消息推送到客户端的，客户端又不像服务器有一个固定的地址

带着这些疑问我们来研究一下目前有哪些技术可以解决该问题：

一、Ajax轮询

所谓的Ajax轮询，其实就是定时的通过Ajax查询服务端，客户端按规定时间定时像服务端发送ajax请求，服务器接到请求后马上返回响应信息并关闭连接。

这种技术方式实现起来非常简单，但是这种方式会有非常严重的问题，就是需要不断的向服务器发送消息询问，这种方式会对服务器造成极大的性能浪费。

还有一个类似的轮询是使用JSONP跨域请求的方式轮询，在实现起来有差别，但基本原理都是相同的，都是客户端不断的向服务器发起请求。

优点

实现简单。

缺点

这是通过模拟服务器发起的通信，不是实时通信，不顾及应用的状态改变而盲目检查更新，导致服务器资源的浪费，且会加重网络负载，拖累服务器。

二、Comet

Comet，基于 HTTP 长连接的 "服务器推" 技术，能使服务器端主动以异步的方式向客户端程序推送数据，而不需要客户端显式的发出请求，目前有两种实现方式：

1. 基于 AJAX 的长轮询（long-polling）方式

Ajax 的出现使得 JavaScript 可以调用 XMLHttpRequest 对象发出 HTTP 请求，JavaScript 响应处理函数根据服务器返回的信息对 HTML 页面的显示进行更新。使用 AJAX 实现 "服务器推" 与传统的 AJAX 应用不同之处在于：

服务器端会阻塞请求直到有数据传递或超时才返回。

客户端 JavaScript 响应处理函数会在处理完服务器返回的信息后，再次发出请求，重新建立连接。

当客户端处理接收的数据、重新建立连接时，服务器端可能有新的数据到达；这些信息会被服务器端保存直到客户端重新建立连接，客户端会一次把当前服务器端所有的信息取回。

基于长轮询的服务器推模型

相对于"轮询"（poll），这种长轮询方式也可以称为"拉"（pull）。因为这种方案基于 AJAX，具有以下一些优点：请求异步发出；无须安装插件；IE、Mozilla FireFox 都支持 AJAX。

长轮询 (long polling) 是在打开一条连接以后保持并等待服务器推送来数据再关闭，可以采用HTTP长轮询和XHR长轮询两种方式:

(1). HTTP 和JSONP方式的长轮询

把 script 标签附加到页面上以让脚本执行。服务器会挂起连接直到有事件发生，接着把脚本内容发送回浏览器，然后重新打开另一个 script 标签来获取下一个事件，从而实现长轮询的模型。

(2).XHR长轮询

这种方式是使用比较多的长轮询模式。

客户端打开一个到服务器端的 AJAX 请求然后等待响应；服务器端需要一些特定的功能来允许请求被挂起，只要一有事件发生，服务器端就会在挂起的请求中送回响应并关闭该请求。客户端 JavaScript 响应处理函数会在处理完服务器返回的信息后，再次发出请求，重新建立连接；如此循环。

现在浏览器已经支持CROS的跨域方式请求，因此HTTP和JSONP的长轮询方式是慢慢被淘汰的一种技术，建议采用XHR长轮询。

优点

客户端很容易实现良好的错误处理系统和超时管理，实现成本与Ajax轮询的方式类似。

缺点

需要服务器端有特殊的功能来临时挂起连接。当客户端发起的连接较多时，服务器端会长期保持多个连接，具有一定的风险。

>>在这里简单的说明下长轮询，长连接的概念

轮询：客户端定时向服务器发送Ajax请求，服务器接到请求后马上返回响应信息并关闭连接。优点：后端程序编写比较容易。

缺点：请求中有大半是无用，浪费带宽和服务器资源。

实例：适于小型应用。

长轮询：客户端向服务器发送Ajax请求，服务器接到请求后hold住连接，直到有新消息才返回响应信息并关闭连接，客户端处理完响应信息后再向服务器发送新的请求。

优点：在无消息的情况下不会频繁的请求。

缺点：服务器hold连接会消耗资源。

实例：WebQQ、Hi网页版、Facebook IM。

另外，对于长连接和socket连接也有区分：

长连接：在页面里嵌入一个隐蔵iframe，将这个隐蔵iframe的src属性设为对一个长连接的请求，服务器端就能源源不断地往客户端输入数据。

优点：消息即时到达，不发无用请求。

缺点：服务器维护一个长连接会增加开销。

实例：Gmail聊天

Flash Socket：在页面中内嵌入一个使用了Socket类的 Flash 程序JavaScript通过调用此Flash程序提供的Socket接口与服务器端的Socket接口进行通信，JavaScript在收到服务器端传送的信息后控制页面的显示。

优点：实现真正的即时通信，而不是伪即时。

缺点：客户端必须安装Flash插件；非HTTP协议，无法自动穿越防火墙。

实例：网络互动游戏。

2. 基于 Iframe 及 htmlfile 的流（streaming）方式

iframe 是很早就存在的一种 HTML 标记， 通过在 HTML 页面里嵌入一个隐蔵帧，然后将这个隐蔵帧的 SRC 属性设为对一个长连接的请求，服务器端就能源源不断地往客户端输入数据。

基于流方式的服务器推模型

Comet的优缺点

优点： 实时性好（消息延时小）；性能好（能支持大量用户）

缺点： 长期占用连接，丧失了无状态高并发的特点。

Comet实现框架

1. Dojo CometD —— http://cometdproject.dojotoolkit.org/

2. DWR —— http://directwebremoting.org/dwr/index.html

3. ICEfaces —— http://www.icefaces.org/main/home/

4. GlassFish Grizzly —— https://grizzly.dev.java.net/

CometD 目前实现 Comet 比较成熟， DWR 弱一些。 ICEfaces 更商业化，实现得很成熟。 Grizzly 是基于GlassFish ，也很成熟。CometD, DWR 开源性好。

Comet实现要点

不要在同一客户端同时使用超过两个的 HTTP 长连接

我们使用 IE 下载文件时会有这样的体验，从同一个 Web 服务器下载文件，最多只能有两个文件同时被下载。第三个文件的下载会被阻塞，直到前面下载的文件下载完毕。这是因为 HTTP 1.1 规范中规定，客户端不应该与服务器端建立超过两个的 HTTP 连接， 新的连接会被阻塞。而 IE 在实现中严格遵守了这种规定。

HTTP 1.1 对两个长连接的限制，会对使用了长连接的 Web 应用带来如下现象：在客户端如果打开超过两个的 IE 窗口去访问同一个使用了长连接的 Web 服务器，第三个 IE 窗口的 HTTP 请求被前两个窗口的长连接阻塞。

所以在开发长连接的应用时， 必须注意在使用了多个 frame 的页面中，不要为每个 frame 的页面都建立一个 HTTP 长连接，这样会阻塞其它的 HTTP 请求，在设计上考虑让多个 frame 的更新共用一个长连接。

服务器端的性能和可扩展性

一般 Web 服务器会为每个连接创建一个线程，如果在大型的商业应用中使用 Comet，服务器端需要维护大量并发的长连接。在这种应用背景下，服务器端需要考虑负载均衡和集群技术；或是在服务器端为长连接作一些改进。

应用和技术的发展总是带来新的需求，从而推动新技术的发展。HTTP 1.1 与 1.0 规范有一个很大的不同：1.0 规范下服务器在处理完每个 Get/Post 请求后会关闭套接口连接； 而 1.1 规范下服务器会保持这个连接，在处理两个请求的间隔时间里，这个连接处于空闲状态。 Java 1.4 引入了支持异步 IO 的 java.nio 包。当连接处于空闲时，为这个连接分配的线程资源会返还到线程池，可以供新的连接使用；当原来处于空闲的连接的客户发出新的请求，会从线程池里分配一个线程资源处理这个请求。 这种技术在连接处于空闲的机率较高、并发连接数目很多的场景下对于降低服务器的资源负载非常有效。

但是 AJAX 的应用使请求的出现变得频繁，而 Comet 则会长时间占用一个连接，上述的服务器模型在新的应用背景下会变得非常低效，线程池里有限的线程数甚至可能会阻塞新的连接。Jetty 6 Web 服务器针对 AJAX、Comet 应用的特点进行了很多创新的改进。

控制信息与数据信息使用不同的 HTTP 连接

使用长连接时，存在一个很常见的场景：客户端网页需要关闭，而服务器端还处在读取数据的堵塞状态，客户端需要及时通知服务器端关闭数据连接。服务器在收到关闭请求后首先要从读取数据的阻塞状态唤醒，然后释放为这个客户端分配的资源，再关闭连接。

所以在设计上，我们需要使客户端的控制请求和数据请求使用不同的 HTTP 连接，才能使控制请求不会被阻塞。

在实现上，如果是基于 iframe 流方式的长连接，客户端页面需要使用两个 iframe，一个是控制帧，用于往服务器端发送控制请求，控制请求能很快收到响应，不会被堵塞；一个是显示帧，用于往服务器端发送长连接请求。如果是基于 AJAX 的长轮询方式，客户端可以异步地发出一个 XMLHttpRequest 请求，通知服务器端关闭数据连接。

在客户和服务器之间保持“心跳”信息

在浏览器与服务器之间维持一个长连接会为通信带来一些不确定性：因为数据传输是随机的，客户端不知道何时服务器才有数据传送。服务器端需要确保当客户端不再工作时，释放为这个客户端分配的资源，防止内存泄漏。因此需要一种机制使双方知道大家都在正常运行。在实现上：

服务器端在阻塞读时会设置一个时限，超时后阻塞读调用会返回，同时发给客户端没有新数据到达的心跳信息。此时如果客户端已经关闭，服务器往通道写数据会出现异常，服务器端就会及时释放为这个客户端分配的资源。

如果客户端使用的是基于 AJAX 的长轮询方式；服务器端返回数据、关闭连接后，经过某个时限没有收到客户端的再次请求，会认为客户端不能正常工作，会释放为这个客户端分配、维护的资源。

当服务器处理信息出现异常情况，需要发送错误信息通知客户端，同时释放资源、关闭连接。

三，websocket方式

WebSocket是HTML5开始提供的一种在单个 TCP 连接上进行全双工通讯的协议。WebSocket通讯协议于2011年被IETF定为标准RFC 6455，WebSocketAPI被W3C定为标准。在WebSocket API中，浏览器和服务器只需要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送。

由于websocket技术要说明白的话所需要的篇幅不小，所以会在之后的单独文章中介绍下websocket的使用方式，这里就不做详细的说明了。

四，总结

根据以上技术的优缺点和具体业务需要，可以选择合适的技术进行应用。

### 39 Google是如何在一秒内把搜索结果返回给用户的？

在你点击了 Google 搜索按钮之后到看到结果这不足1秒钟的时间内，它做了什么?互联网上的内容如何被谷歌找到?什么样的内容会被收录?想必大家一定都想知道谷歌搜索按钮背后的秘密吧。别急，开始之前我们先来看一下神秘的谷歌数据中心……

谷歌的数据中心高度机密，我们所能得到的信息十分有限。我们先来看几个数据：谷歌在美国本土的数据中心有19个以上，另有17个分布在美国以外的世界各地;每个数据中心有50万平方英尺(46450平方米)，建造一个数据中心要花费约6亿美元;谷歌的数据中心是世界上最高效的设施之一，非常环保;数据中心使用50-100兆瓦的电力，考虑到冷却问题，通常建在便于用水的地方;谷歌的服务器被安置在标准的海运集装箱中，每个集装箱可容纳1160台服务器。关于谷歌的数据中心，我们就只知道这么多了。

谷歌拥有的数十万台服务器都是自己设计的，它们认为这是公司的核心技术之一。每台服务器都配有一颗12伏电池，确保万一主电源断电时还可持续供电。

至于为什么为每台服务器配备电池，谷歌的回答是成本。一般数据中心多依赖 UPS(不间断电源系统)，这基本上算是大电池，会在主电力失效而发电机还来不及启动时暂时协助供电。而谷歌认为直接把电力内建到服务器更便宜，而且成本能直接跟服务器数量相符合，如此便不会浪费多余的容量。另一个原因是效率，大型UPS可达92-95%的效率，这意味着许多电力还是被浪费掉了，但谷歌采用的内建电池作法效率超过99.9%。

Google 搜索谷歌的服务器被安装在集装箱中，每个集装箱容纳1160台

Google 搜索谷歌如何找到并收录你上传的内容?

Google 搜索发生在用户搜索之前

谷歌使用它的”爬虫”工具在一刻不停地周游互联网世界的每一个角落。上图中间的6个步骤依次描绘了从内容出现在互联网上到内容被收录进谷歌的数据库供用户检索这一过程，其中第2、3、5步又有许多分支，所有这些都是为了建立一个信息”集汇池”，这是第一个阶段的工作，第二个阶段才是从这个”池”中为用户筛选他们所需要的内容。接下来我们一步步看谷歌是如何搜集并整合信息的。

1、网友上传内容，比如博客、微博或其它类型的WEB内容被更新到网上。

2、Google的”爬虫”发现了这一更新。在这一步，谷歌加入了许多判断机制，主要包括以下几点：

2.1、Google的”爬虫”沿着链接路径(URL)周游互联网，但如果没有URL指向某一站点，则这一站点将不会被索引。

2.2、如果你在 robots.txt 中设置了不许索引(部分或全部)，Google 的”爬虫”将不会抓取你站点上的相应内容。

2.3、如果指向你站点的连接上有 nofollow 标签，Google 的”爬虫”将不会从这些URL路径来到你的站点。

URL 就像是 Google “爬虫”周游互联网时的路标，谷歌当然希望收录有价值的网页，所以必须采取一种机制分辨哪些 URL 是垃圾信息，nofollow 标签正是谷歌所倡导的方法之一。网站的合法更新人员几乎不会上传垃圾 URL，但它们往往大量出现在评论跟帖和论坛中，就像上图中的例子，这些 URL 对于谷歌来讲是没有意义的，为了防止”爬虫”经由这些 URL 到达某一站点，在源代码中它们都会自动被加上 nofollow 标签。

2.4、Google 也能通过 blog 软件或 xml 站点地图找到你的网站。

2.5、从权威性越高的网站链接到你网站的 URL 越多，你的网站的权威性也就越高，但 Google “爬虫”始终会忽略被加上了 nofollow 标签的 URL。

上面这几点大概就是谷歌在收录信息时对内容提出的”准入”要求，看来在一些开放的地方(比如论坛)大量发布 URL 以求让谷歌关注，这一小伎俩是没有什么效果的。以上是信息被谷歌收录之前所发生的事，一旦信息被谷歌收录了会发生什么呢?请往下看：

Google 搜索信息”素材”的存储

3、信息被谷歌收录之后当然也要进行加工处理，主要包括两个步骤，一是信息”素材”的存储，二是对收录的信息按要求进行优化，上图描绘了”素材”的存储方式，主要包括两部分：网页标题和链接数据被保存在一个索引中，用于广度优先搜索(可见文章标题是多么的重要，做编辑的一定要有驾驭标题党的觉悟);网页内容保存在另一个索引中，以用于检索频率不高的长尾、个性化、深度优先搜索。

此时可能你已经明白了，当你用谷歌搜索时，你并没有在检索时时更新的互联网，而是在检索谷歌的缓存，只是谷歌更新的速度非常快，以让其缓存尽量与互联网上的内容同步。

Google 搜索优化已收录的信息

4、谷歌基于 URL 评估域名和网页的总体权威性。

5、检查网页以防止作弊行为，包括以下几点：

5.1、谷歌的搜索质量和反垃圾信息审查。

5.2、1万多远程测试用户评价搜索结果的质量。

5.3、谷歌征请用户对有 PageRank 讹诈嫌疑的垃圾信息进行举报。

5.4、谷歌根据数字千年版权法( DMCA )去除盗版内容。

6、在对页面进行分析之后，每个页面都被附加上很多用于辅助用户搜索的数据片。

从信息出现在互联网上到被谷歌收录，然后谷歌对这些数据进行分析优化，至此，一个实时更新的互联网信息”集汇池”就建立起来了，可以说谷歌存储着整个互联网的快照。以上就是我们在按谷歌搜索按钮之前它所做的事情，接下来我们看一下谷歌如何响应用户的搜索请求，另外谷歌的广告是如何来到我们面前的，不要忘记，谷歌可是靠广告营生的。

只要有人用谷歌的服务它就能从中赚钱，就怕像安卓 (Android )手机系统那样，有些流氓厂家把安卓装在了自己的智能手机中，但是把其上谷歌的各种服务全部抹掉，改用自己的服务，这样谷歌当然不干了，所以安卓一更新，这些流氓手机厂家就紧张。

谷歌如何帮助用户进行搜索?

从用户开始检索到生成初步结果(这时的结果并不会直接呈现给用户)，经历了4个步骤：

1、用户发出搜索请求。谷歌搜索质量工程师 PatrickRiley 说：在大多数搜索中，你的搜索处于多个并行的控制过程或谷歌实验室的创新项目组过程中，可以说每一个查询请求都会参与一些谷歌的创意实验。我们都是小白鼠?

2、谷歌会对用户输入的关键词提供一些建议。

3、谷歌会用同义词匹配与你的搜索关键词语义相近的查询结果。

4、生成初步的查询结果，虽然谷歌宣称可以找到成千上万的相关结果，但一般只显示不到1000条，同时查询结果将被进行本地化处理，本土站点在查询结果中优先出现。

搜索结果将如何被优化?

1、对查询结果按权威性和 PageRank 进行排序，重复的查询结果被剔除。此时的查询结果已接近最终形态，在这一基础上，有两个进程将分开进行–查询结果优化和为其匹配相应的广告。我们先来看查询结果优化。

对查询结果进行过滤处理

2、对查询结果进行过滤处理。包括以下几点：

2.1、对通常的查询，谷歌会把相关的专题性垂直搜索结果(比如新闻、购物、视频、书籍、地图等)也加到返回的查询结果中。

2.2、个性化，用户访问过的网站在查询结果列表中会更靠上

2.3、大量使用锚点的网站有可能被从查询结果中删除

2.4、如果网页被其他高 PageRank 的网站引用，则网页的重要性会大大提高。

2.5、趋势分析：对搜索流量爆增或有大量新闻的搜索关键词，谷歌会在新的查询结果中增加额外的 PageRank 权值。

2.6、同一个域名下的多个网页如果具有相同的 PageRank 会被归为一组。

2.7、查询结果最终形成(将与广告一同显示)

用户所搜索的内容如何与广告相结合?

1、Google 根据关键词、广告类型、用户所处位置找出相关的被竞价拍卖的关键词广告

1.1、关键词广告必须遵守当地法律条文。广告业主的非法广告将被取缔，如果关键词的搜索流量过低或关键词广告点击量偏低，则会被自动禁用，出于商业策略，像亚马逊这样的客户会给予优惠折扣。

1.2、关键词相关广告按收益潜力排序。

1.3、对广告业主来说广告内容一般是固定的，但有时使用动态关键词使关键词广告与搜索关键词相关度更高。一些广告允许增加附属信息，比如网站链接、电话号码、产品链接、地址等。

1.4、如果广告拥有相当高的点击率，则会显示在搜索结果列表的上方，以使其更显眼。

1.5、其余的广告依序显示在页面右侧

经过上述一系列复杂的信息处理过程，最终返回给用户的是一个个性化的、具备地理位置特征的、布局简洁的查询结果页面，当然还精确匹配了广告，所有这些步骤在总共不到1秒的时间内完成，每天3亿次的点击量给 Google 带来了超过200亿美元的年收入

我们搜索技术的后端软件会在服务器侧触发一系列执行时间不到1秒的并行计算，谷歌问世前的传统搜索引擎的搜索结果严重依赖于关键词在页面上出现的频度，我们使用了200多个指标信号(其中包括我们拥有专利的 PageRank 页面等级加权算法)用来检查万维网的链接结构并决定网页的重要程度。

我们假定一个网页的重要程度取决于别的页面对它的引用，就像学术论文中的引用指数一样，重要的论文总是会被很多其他论文引用。然后我们再根据搜索条件进行超文本匹配分析(对”爬虫”抓取的页面内容进行关键词倒排索引检索)确定跟搜索请求最相关的网页。综合最重要的网页和跟搜索请求最相关的网页两个方面，我们就能按重要程度和用户搜索请求相关程度把查询结果排序后呈现给我们的用户。

### 40 12306网站的订票系统如何实现，如何保证不会票不被超卖？

如果是以前，我肯定会回答：是，因为我们始终认为，一个年投入超3亿元的卖火车票网站，没有什么理由做不好吧？其实不光是马海祥这样认为，包括众多媒体，甚至专业人士都曾喷过12306，但近日一名前淘宝工程师通过科普的方式给了我们一记响亮的耳光：做12306网站的难度要远高于淘宝，而且目前12306网站所面临的问题，也不是那么容易解决的！

**1、人们对12306网站的误解**

这位名为“代码狗”的前淘宝工程师，后来在一些论坛上发文表达了他自己对12306系统的看法，值得注意的是，“代码狗”在12306系统刚上线时也有过不少微词，为了证明12306系统很容易搭建，“代码狗”甚至曾经发起过一个名为“替12306设计系统”的开源项目，通过工作中的实践，“代码狗”对于12306系统也有了新的认识。

本人淘宝技术专家，曾在淘宝写过一段时间代码，2012年在一家百强民企做电商副总，当时在极为艰苦的条件下带队开发了一个B2C网站，走支付宝和银联支付通道，年营业额千万级。

也就在那个时候，我对12306嗤之以鼻，觉得他们做得太烂了，认为自己能带队花几百万半年时间做个好的出来，于是我狂妄地想做一个开源的订票系统给他们，我花了一个星期时间思考建立数据模型，思考到库存这一步的时候，我才发现，12306的库存复杂性比淘宝、京东高很多倍，运算量也大很多倍，传统的分布式数据库、缓存、负载均衡技术并不能恰好满足12306的需求。

在平时，12306也就是个正常的电商网站，但一到黄金周，12306就是一个全站所有商品都秒杀，所有SKU(库存量单位，物流管理术语，编者注)都是动态库存的变态。

即使不考虑线下既有的电话、代售点等渠道，要实现一个12306，最少最少也是千万级别的硬件投入(这是当时的估算，没有精算，可能与实际相差较大，总之，我说得不一定对，12306的业务也许没我说的那么复杂，但也绝不是某些人喷的那么简单)，软件和人力另算。

那些叫嚣只要40台服务器、只要2个架构师4个程序员、大谈分库分表和前端CDN的人们，只是纸上谈兵罢了，所谓初生牛犊不怕虎，做了三年CMS和BBS，就以这个经验来喷12306，未免太天真了。

媒体人喷12306，是他们不懂技术，没有能力和耐心来分析背后的难度；技术人员喷，则是因为大部分的技术人员在短时间思考时，容易陷入过于乐观的误区，经典的例子就是估算工作量；程序员们往往容易估算出一个超短的工期，把写程序的工作乐观地想象成了打字员照稿敲键盘的工作（具体可查看马海祥博客《史上最牛网站12306订票网的十宗罪及改善开发建议》的相关介绍）。

淘宝技术是比12306强大很多倍，淘宝现在的系统也是花了10倍于12306的钱、时间和人才做起来的，根本原因还是铁路运力不能满足春运需求，淘宝也解决不了这个问题。

12306这一年来进步非常大，从前段动画验证码、分时段抢票，到后端去小型机、虚拟化、内存数据库的运用，可以说，12306是中国政府机关做的最强大的网站(电商系统)，能在短短一两年内做出这样的改变，几乎是个奇迹，就连一些市场化的民企都望尘莫及，甚至一些上市公司都比不上它。

事非经过不知难，在网上批判12306的人，大部分还是形成了“国企=垄断+腐败+低效”的思维定势，小部分是真的轻视了它的难度。

至于12306一期工程3个亿(含硬件)贵不贵我不评价，我只提供一个数字供参考，百度一年的研发费用(不含硬件)是10亿，这个数字来自百度财报，网上能查到，3亿看起来好大一个数字，真用到超大型的电商系统、搜索引擎系统里面，其实也不算什么天文数字了。

再解释一下，为什么秒杀压力大，以及为什么12306的动态库存很复杂。

**2、关于“秒杀”**

2013年12月25日前后，天猫搞了一个圣诞季积分兑换活动，持续几天，25号上午10点12分，放出了15000个天猫魔盒(淘宝集市有人卖，大概190-230块)，从成交记录上看，是19秒内全部抢完。

实际上，我也参加秒杀了，那天的题目特别简单(请输入xxx汉字的拼音首字母)，我应该是5秒内答题完成并提交订单，结果告诉我排队的人太多，挤不进去，并提示14秒以后重试。

人太多就是因为题目太简单了，门槛越低，5秒内挤进去的人也越多嘛，如果题目换成“2克浓度为3%的U235在大亚湾核电站能发多少KW的电？”，5分钟之内也不会有1万5千人跟我竞争。

我想，14秒以后哪还有我的事情呀，于是重新答题秒杀，结果出现了服务器错误的页面，反复刷新几次，就告诉秒杀结束了。

在群里问了一下同事，有不到10个人回答我，都说没秒到（也可能秒到的人闷声发大财，不回复我）。

淘宝是什么技术水平呢？淘宝有至少4000技术人员，至少4万台服务器(这都是两年前的公开数据了，按规定可以谈论)，2013年11月11日成交额351亿，2012年全年成交额超过1万亿。

淘宝拥有各种自主研发团队：服务器、交换机(网上可以搜索到淘宝公开的绿色服务器开放标准);操作系统(LinuxKerneltaobao版，yunos手机操作系统是阿里云的，暂时不计入)、Web服务器(Tengine)、Java语言虚拟机(JVMtaobao版)、数据库(MySQL内核taobao版，google和facebook也有自己的版本，HBase淘宝版、还有自己全部从头开发的OceanBase)、负载均衡器(LVS，LVS始创人就在淘宝，担任研究员)、Java运行容器(Jboss，其创始人之一，王文彬，也在淘宝，担任副总裁)。

淘宝还有数不清的开源项目和中间件，如高性能Java通信中间件HSF、分布式数据库中间件TDDL、异步消息系统notify等等等等。

以淘宝这样的技术水平，也不能做到秒杀时让每个用户都没有拥挤感，为什么呢？

一是要尊重物理原理，一台服务器一秒钟能承受的计算量是有极限的，任你怎么优化，采用多高效的算法和编程语言，都突破不了某个极限，比方说汽车发动机驱动的F1赛车至今也不能突破400公里的时速(超音速推进号那个1千多公里的时速不能算，那是飞机引擎驱动的)，再往深了说，就不容易懂了，感兴趣的可以从著名的C10K问题开始看起。

二是要考虑经济效益，十一黄金周的时候，北京主城区到八达岭长城的路堵得严严实实，但不能因为黄金周的高峰，就把这段路修成长安街那样10车道的高速公路，否则的话，花费天文数字(真的是天文数字，12306那3个亿大概只够修1-3公里)，修了一段路，黄金周是可以飙到80公里/小时了，可平时呢？难不成拿来给两边的居民晒谷子？

淘宝目前的硬件和带宽数量，已经超出日常运营的需求了，就是留了相当大的余量给大促销(众所周知的是双十一，双十二，其实基本每个季度都有大促销，每个月都有促销，甚至天天都在促销——聚划算)，amazon当年就是为了应对黑色星期五的大促销购置了大量的服务器，平时订单量没那么大了，amazon就把富余的服务器拿来搞云计算了。

顺便说一下，阿里云是当今中国第一世界数一数二的云计算服务商，和amazon走的路也有点像。

**3、关于动态库存**

淘宝秒杀天猫魔盒的时候，只有一个商品(行话叫做SKU)，它的库存是15000个。有一个人秒杀到了，库存就减1，19秒卖完的，一秒要成功产生789个订单(下订单的请求可能是8万个，只是可能啊，非实际数字，也可能是1万个，用于说明一下壮观程度)，想象一下，你在广场上卖火车票，一秒钟有8万人举着钱对你喊：卖给我！

大家都知道，比秒小的时间单位还有毫秒、皮秒、飞秒，但交易系统登记一个交易可不像原子绕着原子核跑一圈那么简单，它要做这些事：检查是否恶意访问、取到系统时间、取到顾客默认收货地址、核对顾客秒杀资格(当时的规定是天猫T2.T3达人)、生成订单号、把顾客ID系统时间订单号收货地址写入订单系统、扣除顾客天猫积分、商品库存减一、给顾客打标记(每人只能秒一个，下次不能秒了)等等。

这每一件事都要花费毫秒级别的时间，这些操作加起来的时间可能是接近1秒级别的，但由于淘宝的服务器比较强悍，而且采用了分布式和集群技术，结果比1秒理想一点，但即使有1万台服务器，也不能把这个时间稀释成万分之一秒，因为，商品只有一种，它有15000个库存，对应的数据库记录只有一行，所有的交易请求都要到这里来处理。

能不能把这15000个拆分成5000个商品并分配到5000台服务器上呢？那样不就可以5000台服务器同时处理了吗？答案是不能，首先，5000个商品，意味着有5000个商品详情页，5000个购买按钮，这对前期的营销、引流是个灾难，基本上就没法做引流入口了，显然这违背了商业管理原则，人为增加了信息混乱程度。

其次，天猫魔盒秒杀也不是啥大事，即使按官方标价399元来计算，也就6百万的交易，如果6百万的交易要花费那么大的配套成本，那就太不划算了。

再次，淘宝有十几亿商品，这十几亿商品的展示交易和管理，本来就是分布到上万台服务器上去了，没有必要再把每个商品按库存拆成多个商品了。

这789人抢到了，还不一定会付款(99积分换天猫魔盒还好一点，不需要去网银，成本也极低，大部分是会付款的，3999秒杀iPhone5S就不一定，有人可能网银有问题，有人可能改变主意不想要了)，所以就又带来订单取消重新恢复库存的问题，还有想要的消费者们，会认为还有机会，继续在前台刷一会儿，最终这个秒杀会被热情的消费者们猛刷30秒到1分钟。

一分钟过去了，服务器终于可以喘口气了吧？等等，还有超卖，原来，某两台服务器在同一毫秒都拿到了锁，都去减了库存，15000个库存，被下了15500个订单，又得取消一部分订单，如果采用单线程独占锁，是可以做到同时只有一个服务器线程减库存的，但那样就对并发高峰的能力就差了好多了。

8万人举着钱，可能只有8个人能下单成功，这个拥挤狂热的抢购就要持续10分钟以上，平时秒个天猫魔盒，10分钟也就10分钟吧，双十一就惨了，收银台一下子减少了90%，还想做到350亿，要么做梦，要么再加10倍服务器和带宽。

所以，商业是不完美的，要在绝对正确和绝对的快速之间做个取舍，保证相对快速又极为正确，允许一定的库存错误和超卖(具体允许多少我也不知道)。

**4、关于12306网站的数据库设计**

好了，讲了这半天淘宝，我们再来说说12306了吧！

我以北京西到深圳北的G71次高铁为例(这里只考虑南下的方向，不考虑深圳北到北京西的，那是另外一个车次，叫G72)，它有17个站(北京西是01号站，深圳北是17号站)，3种座位(商务、一等、二等)。

表面看起来，这不就是3个商品吗？G71商务座、G71一等座、G71二等座，大部分轻易喷12306的技术人员(包括某些中等规模公司的专家、CTO)就是在这里栽第一个跟头的。

实际上，G71有136\*3=408种商品(408个SKU)，怎么算来的呢？

如果卖北京西始发的，有16种卖法(因为后面有16个站)，北京西到：保定、石家庄、郑州、武汉、长沙、广州、虎门、深圳……，都是一个独立的商品，同理，石家庄上车的，有15种下车的可能，以此类推，单以上下车的站来计算，有136种票：16+15+14…+2+1=136，每种票都有3种座位，一共是408个商品。

好了，再看出票时怎么减库存，由于商务、一等、二等三种座位数是独立的，库存操作也是一样的，下文我就不再提座位的差别的，只讨论出发与到达站。

另外，马海祥跟大家提升一下，下文说的是理论世界的模型，不是说12306的数据库就是这么设计的。

旅客A买了一张北京西(01号站)到保定东(02号站)的，那“北京西到保定东”这个商品的库存就要减一，同时，北京西到石家庄、郑州、武汉、长沙、广州、虎门、深圳等15个站台的商品库存也要减一，也就是说，出一张北京到保定东的票，实际上要减16个商品的库存。

这还不是最复杂的，如果旅客B买了一张北京西(01号站)到深圳北(17号站)的票，除了“北京西到深圳北”这个商品的库存要减一，北京西到保定东、石家庄、郑州、武汉、长沙、广州、虎门等15个站台的商品库存也要减1，保定东到石家庄、郑州、武汉、长沙、广州、虎门、深圳北等15个站台的商品库存要减1，总计要减库存的商品数是16+15+14+……+1=120个。

当然，也不是每一张票都的库存都完全这样实时计算，可以根据往年的运营情况，在黄金周这样的高峰时段，预先对票做一些分配，比如北京到武汉的长途多一点，保定到石家庄的短途少一点，我没有证据证实铁道部这样做了，但马海祥相信，在还没有12306网站的时候，铁道部就有这种人工预分配的策略了。

想象一下，8万人举着钱对你高喊：卖给我！你好不容易在钱堆里找到一只手，拿了他的钱，转身找120个同事，告诉他们减库存，而这120个同事也和你一样被8万人围着，也和你一样，每卖出一个商品要找几十个人减库存……，这就是12306动态库存的变态之处，比你平时买东西的任何网站的库存机制都复杂几十上百倍。

**5、关于抢票插件对12306网站的影响**

再说一下抢票插件（此前，我也曾在马海祥博客《从12306网站改版来看浏览器产品差异化发展》的一文中跟大家说个抢票插件与浏览器的共生的平衡状态），机器永远比人快，当你好不容易从8万人里突出重围，来到了柜台前，你发现，我操，来了10万根绑着钱的竹竿，而且当有退票出来的时候，你要闯过3层人肉才能接近柜台，竹竿在8个人身后一伸，钱就到了柜台前，你低头看了一眼手机，票就没了，竹竿却永远在那里伸着，永不低头，永不眨眼，如果没有这10万根竹竿，虽然你很可能还是抢不到票，但不至于沮丧成这样：我TM为什么总是手最慢的一个！

防机器人抢票，也不是加个图片验证码那么简单，我写过文章系统性分析过，图片验证码有6种机器暴力破解的办法，抢票插件用的是我说的第三种，OCR识别(光学字符识别)。

Google采用的Wave波形字母已经能比较好地防住机器OCR了，ems上的验证码就是反面教材，机器OCR成功率接近100%，12306的比ems的图片验证码强一点，不过，验证码设置得复杂一点吧！

接下来，人们就要喷了：这只是便宜大学生和办公室白领，农民工连26个字母都认不齐，怎么搞？搞动画验证码吧，也有人喷，视力不好的人怎么办？最后验证码搞得太简单了，皆大欢喜了，其实最高兴的是开发抢票插件的公司。

就算采用了机器完全不可能识别的验证码，也防不住社会工程学的破解办法，招募一堆网吧打游戏的青少年朋友，每成功输入50个验证码给1块钱，或者等值的虚拟货币、游戏装备，我保证想赚这个钱的人数不胜数，这点钱对转卖车票的利润而言，是可以接受的成本。

以上讨论只是把12306当成和淘宝一样没有历史包袱从零起步的交易系统，实际上，它不是，它后面的票池，还有电话售票、火车站售票、代售点售票等多个传统渠道要服务。

除了客运服务，12306还有全国最大(很可能也是全球最大)的大宗物资货运系统。

### 41 如何实现一个秒杀系统，保证只有几位用户能买到某件商品？

**什么是秒杀**

秒杀场景一般会在电商网站举行一些活动或者节假日在12306网站上抢票时遇到。对于电商网站中一些稀缺或者特价商品，电商网站一般会在约定时间点对其进行限量销售，因为这些商品的特殊性，会吸引大量用户前来抢购，并且会在约定的时间点同时在秒杀页面进行抢购。

**秒杀系统场景特点**

秒杀时大量用户会在同一时间同时进行抢购，网站瞬时访问流量激增。

秒杀一般是访问请求数量远远大于库存数量，只有少部分用户能够秒杀成功。

秒杀业务流程比较简单，一般就是下订单减库存。

**秒杀架构设计理念**

限流： 鉴于只有少部分用户能够秒杀成功，所以要限制大部分流量，只允许少部分流量进入服务后端。

削峰：对于秒杀系统瞬时会有大量用户涌入，所以在抢购一开始会有很高的瞬间峰值。高峰值流量是压垮系统很重要的原因，所以如何把瞬间的高流量变成一段时间平稳的流量也是设计秒杀系统很重要的思路。实现削峰的常用的方法有利用缓存和消息中间件等技术。

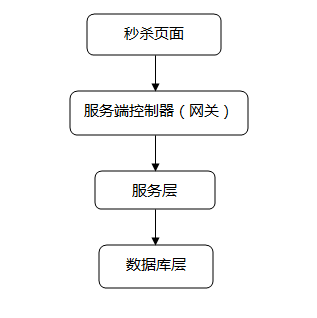
异步处理：秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式。

内存缓存：秒杀系统最大的瓶颈一般都是数据库读写，由于数据库读写属于磁盘IO，性能很低，如果能够把部分数据或业务逻辑转移到内存缓存，效率会有极大地提升。

可拓展：当然如果我们想支持更多用户，更大的并发，最好就将系统设计成弹性可拓展的，如果流量来了，拓展机器就好了。像淘宝、京东等双十一活动时会增加大量机器应对交易高峰。

**架构方案**

一般秒杀系统架构



**设计思路**

将请求拦截在系统上游，降低下游压力：秒杀系统特点是并发量极大，但实际秒杀成功的请求数量却很少，所以如果不在前端拦截很可能造成数据库读写锁冲突，甚至导致死锁，最终请求超时。   
充分利用缓存：利用缓存可极大提高系统读写速度。   
消息队列：消息队列可以削峰，将拦截大量并发请求，这也是一个异步处理过程，后台业务根据自己的处理能力，从消息队列中主动的拉取请求消息进行业务处理。

**前端方案**

浏览器端(js)：

页面静态化：将活动页面上的所有可以静态的元素全部静态化，并尽量减少动态元素。通过CDN来抗峰值。   
禁止重复提交：用户提交之后按钮置灰，禁止重复提交   
用户限流：在某一时间段内只允许用户提交一次请求，比如可以采取IP限流

**后端方案**

服务端控制器层(网关层)

限制uid（UserID）访问频率：我们上面拦截了浏览器访问的请求，但针对某些恶意攻击或其它插件，在服务端控制层需要针对同一个访问uid，限制访问频率。

**服务层**

上面只拦截了一部分访问请求，当秒杀的用户量很大时，即使每个用户只有一个请求，到服务层的请求数量还是很大。比如我们有100W用户同时抢100台手机，服务层并发请求压力至少为100W。

采用消息队列缓存请求：既然服务层知道库存只有100台手机，那完全没有必要把100W个请求都传递到数据库啊，那么可以先把这些请求都写到消息队列缓存一下，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。

利用缓存应对读请求：对类似于12306等购票业务，是典型的读多写少业务，大部分请求是查询请求，所以可以利用缓存分担数据库压力。

利用缓存应对写请求：缓存也是可以应对写请求的，比如我们就可以把数据库中的库存数据转移到Redis缓存中，所有减库存操作都在Redis中进行，然后再通过后台进程把Redis中的用户秒杀请求同步到数据库中。

**数据库层**

数据库层是最脆弱的一层，一般在应用设计时在上游就需要把请求拦截掉，数据库层只承担“能力范围内”的访问请求。所以，上面通过在服务层引入队列和缓存，让最底层的数据库高枕无忧。

案例：利用消息中间件和缓存实现简单的秒杀系统

Redis是一个分布式缓存系统，支持多种数据结构，我们可以利用Redis轻松实现一个强大的秒杀系统。

我们可以采用Redis 最简单的key-value数据结构，用一个原子类型的变量值(AtomicInteger)作为key，把用户id作为value，库存数量便是原子变量的最大值。对于每个用户的秒杀，我们使用 RPUSH key value插入秒杀请求， 当插入的秒杀请求数达到上限时，停止所有后续插入。

然后我们可以在台启动多个工作线程，使用 LPOP key 读取秒杀成功者的用户id，然后再操作数据库做最终的下订单减库存操作。

当然，上面Redis也可以替换成消息中间件如ActiveMQ、RabbitMQ等，也可以将缓存和消息中间件 组合起来，缓存系统负责接收记录用户请求，消息中间件负责将缓存中的请求同步到数据库。

### 42 有关注哪些新的技术？

**云计算，人工智能，区块链，大数据**

### 43 hash算法的实现原理，hashcode的实现原理？

**Hash**，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入，通过散列算法，变换成固定长度的输出，该输出就是散列值。

散列表,它是基于快速存取的角度设计的，也是一种典型的“空间换时间”的做法。顾名思义，该数据结构可以理解为一个线性表，但是其中的元素不是紧密排列的，而是可能存在空隙。

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

hash函数的选择

哈稀函数按照定义可以实现一个伪随机数生成器(PRNG)，从这个角度可以得到一个公认的结论：哈希函数之间性能的比较可以通过比较其在伪随机生成方面的比较来衡量。

一般来说，对任意一类的数据存在一个理论上完美的哈希函数。这个完美的哈希函数定义是没有发生任何碰撞，这意味着没有出现重复的散列值。在现实中它很难找到一个完美的哈希散列函数，而且这种完美函数的趋近变种在实际应用中的作用是相当有限的。在实践中人们普遍认识到，一个完美哈希的哈希函数，就是在一个特定的数据集上产生的的碰撞最少哈希的函数。

我们所能做的就是通过试错方法来找到满足我们要求的哈希函数。可以从下面两个角度来选择哈希函数：

1.数据分布

一个衡量的措施是考虑一个哈希函数是否能将一组数据的哈希值进行很好的分布。要进行这种分析，需要知道碰撞的哈希值的个数，如果用链表来处理碰撞，则可以分析链表的平均长度，也可以分析散列值的分组数目。

2.哈希函数的效率

另个一个衡量的标准是哈希函数得到哈希值的效率。通常，包含哈希函数的算法的算法复杂度都假设为O(1)，这就是为什么在哈希表中搜索数据的时间复杂度会被认为是"平均为O(1)的复杂度"，而在另外一些常用的数据结构，比如图(通常被实现为红黑树)，则被认为是O(logn)的复杂度。

一个好的哈希函数必须在理论上非常的快、稳定并且是可确定的。通常哈希函数不可能达到O(1)的复杂度，但是哈希函数在字符串哈希的线性的搜索中确实是非常快的，并且通常哈希函数的对象是较小的主键标识符，这样整个过程应该是非常快的，并且在某种程度上是稳定的。

在这篇文章中介绍的哈希函数被称为简单的哈希函数。它们通常用于散列（哈希字符串）数据。它们被用来产生一种在诸如哈希表的关联容器使用的key。这些哈希函数不是密码安全的，很容易通过颠倒和组合不同数据的方式产生完全相同的哈希值。

hash方法学

1.基于加法和乘法的散列

这种方式是通过遍历数据中的元素然后每次对某个初始值进行加操作，其中加的值和这个数据的一个元素相关。通常这对某个元素值的计算要乘以一个素数。

2.基于移位的散列

和加法散列类似，基于移位的散列也要利用字符串数据中的每个元素，但是和加法不同的是，后者更多的而是进行位的移位操作。通常是结合了左移和右移，移的位数的也是一个素数。每个移位过程的结果只是增加了一些积累计算，最后移位的结果作为最终结果。

hash构造方法

1. 直接寻址法：取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a?key + b，其中a和b为常数（这种散列函数叫做自身函数）

　　2. 数字分析法：分析一组数据，比如一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体相同，这样的话，出现冲突的几率就会很大，但是我们发现年月日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成散列地址，则冲突的几率会明显降低。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址。

　　3. 平方取中法：取关键字平方后的中间几位作为散列地址。

　　4. 折叠法：将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。

　　5. 随机数法：选择一随机函数，取关键字的随机值作为散列地址，通常用于关键字长度不同的场合。

　　6. 除留余数法：取关键字被某个不大于散列表表长m的数p除后所得的余数为散列地址。即 H(key) = key MOD p, p<=m。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对p的选择很重要，一般取素数或m，若p选的不好，容易产生同义词。

hash冲突及解决

hash冲突在所难免，解决冲突是一个复杂问题。冲突主要取决于：

（1）与散列函数有关，一个好的散列函数的值应尽可能平均分布。

（2）与解决冲突的哈希冲突函数有关。

（3）与负载因子的大小。太大不一定就好，而且浪费空间严重，负载因子和散列函数是联动的。

解决冲突的办法：

（1）开放定址法：线性探查法、平方探查法、伪随机序列法、双哈希函数法。

（2）拉链法：把所有同义词，即hash值相同的记录，用单链表连接起来。

哈希函数和素数

没有人可以证明素数和伪随机数生成器之间的关系，但是目前来说最好的结果使用了素数。伪随机数生成器现在是一个统计学上的东西，不是一个确定的实体，所以对其的分析只能对整个的结果有一些认识，而不能知道这些结果是怎么产生的。

围绕着哈希函数中的素数的使用的基本的概念是，利用一个素数来改变处理的哈希函数的状态值，而不是使用其他类型的数。处理这个词的意思就是对哈希值进行一些简单的操作，比如乘法和加法。这样得到的一个新的哈希值一定要在统计学上具有更高的熵，也就是说不能有为偏向。简单的说，当你用一个素数去乘一堆随机数的时候，得到的数在bit这个层次上是1的概率应该接近0.5。没有具体的证明这种不便向的现象只出现在使用素数的情况下，这看上去只是一个自我宣称的直觉上的理论，并被一些业内人士所遵循。

决定什么是正确的，甚至更好的方法和对散列素数的使用最好的组合仍然是一个很有黑色艺术。没有单一的方法可以宣称自己是最终的通用散列函数。最好的一所能做的就是通过试错演进和获得适当的散列算法，以满足其需要的统计分析方法。

应用领域

哈希是一个在现实世界中将数据映射到一个标识符的工具，下面是哈希函数的一些常用领域：

1.字符串哈希

在数据存储领域，主要是数据的索引和对容器的结构化支持，比如哈希表。

2.加密哈希

用于数据/用户核查和验证。一个强大的加密哈希函数很难从结果再得到原始数据。加密哈希函数用于哈希用户的密码，用来代替密码本身存在某个服务器撒很难过。加密哈希函数也被视为不可逆的压缩功能，能够代表一个信号标识的大量数据，可以非常有用的判断当前的数据是否已经被篡改(比如MD5)，也可以作为一个数据标志使用，以证明了通过其他手段加密文件的真实性。

3.几何哈希

这个哈希表用于在计算机视觉领域，为在任意场景分类物体的探测。最初选择的过程涉及一个地区或感兴趣的对象。几何散列包括各种汽车分类的重新检测中任意场景的目的，典型的例子。检测水平可以多种多样，从刚检测是否是车辆，到特定型号的车辆，在特定的某个车辆。

4.布隆过滤器

布隆过滤器允许一个非常大范围内的值被一个小很多的内存锁代表。在计算机科学，这是众所周知的关联查询，并在关联容器的核心理念。

Bloom Filter的实现通过多种不同的hash函数使用，也可通过允许一个特定值的存在有一定的误差概率会员查询结果的。布隆过滤器的保证提供的是，对于任何会员国的查询就永远不会再有假阴性，但有可能是假阳性。假阳性的概率可以通过改变控制为布隆过滤器，并通过不同的hash函数的数量所使用的表的大小。

随后的研究工作集中在的散列函数和哈希表以及Mitzenmacher的布隆过滤器等领域。建议对这种结构，在数据被散列熵最实用的用法有助于哈希函数熵，这是理论成果上缔结一项最佳的布隆过滤器（一个提供给定一个最低的进一步导致假阳性的可能性表的大小或反之亦然）提供假阳性的概率定义用户可以建造最多也作为两种截然不同的两两独立的哈希散列函数已知功能，大大提高了查询效率的成员。

布隆过滤器通常存在于诸如拼写检查器，字符串匹配算法，网络数据包分析工具和网络/ Internet缓存的应用程序。

5.Hash算法在信息安全方面的应用主要体现在以下的3个方面：

　　（1) 文件校验

　　我们比较熟悉的校验算法有奇偶校验和CRC校验，这2种校验并没有抗数据篡改的能力，它们一定程度上能检测并纠正数据传输中的信道误码，但却不能防止对数据的恶意破坏。

　　MD5 Hash算法的"数字指纹"特性，使它成为目前应用最广泛的一种文件完整性校验和(Checksum)算法，不少Unix系统有提供计算md5 checksum的命令。

　　（2) 数字签名

　　Hash 算法也是现代密码体系中的一个重要组成部分。由于非对称算法的运算速度较慢，所以在数字签名协议中，单向散列函数扮演了一个重要的角色。 对 Hash 值，又称"数字摘要"进行数字签名，在统计上可以认为与对文件本身进行数字签名是等效的。而且这样的协议还有其他的优点。

　　（3) 鉴权协议

　　如下的鉴权协议又被称作挑战--认证模式：在传输信道是可被侦听，但不可被篡改的情况下，这是一种简单而安全的方法。

**HashCode定义**

（1）HashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，HashCode是用来在散列存储结构中确定对象的存储地址的；

（2）如果两个对象相同， equals方法一定返回true，并且这两个对象的HashCode一定相同；

（3）如果对象的equals方法被重写，那么对象的HashCode也尽量重写，并且产生HashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点；

（4）两个对象的HashCode相同，并不一定表示两个对象就相同，也就是equals方法不一定返回true，只能够说明这两个对象在散列存储结构中，如Hashtable，他们存放在同一个篮子里。

HashCode作用

Java中的集合（Collection）有两类，一类是List，再有一类是Set。前者集合内的元素是有序的，元素可以重复；后者元素无序，但元素不可重复。 equals方法可用于保证元素不重复，但是，如果每增加一个元素就检查一次，如果集合中现在已经有1000个元素，那么第1001个元素加入集合时，就要调用1000次equals方法。这显然会大大降低效率。

于是，Java采用了哈希表的原理。

哈希算法也称为散列算法，是将数据依特定算法直接指定到一个地址上。

这样一来，当集合要添加新的元素时，先调用这个元素的HashCode方法，就一下子能定位到它应该放置的物理位置上。

（1）如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；

（2）如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了；

（3）不相同的话，也就是发生了Hash key相同导致冲突的情况，那么就在这个Hash key的地方产生一个链表，将所有产生相同HashCode的对象放到这个单链表上去，串在一起（很少出现）。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。

如何理解HashCode的作用：

从Object角度看，JVM每new一个Object，它都会将这个Object丢到一个Hash表中去，这样的话，下次做Object的比较或者取这个对象的时候（读取过程），它会根据对象的HashCode再从Hash表中取这个对象。这样做的目的是提高取对象的效率。若HashCode相同再去调用equal。

### 44 继承与组合的区别，使用场景？

首先它们都是实现系统功能重用，代码复用的最常用的有效的设计技巧，都是在设计模式中的基础结构。相信大家已了解的，类继承允许我们根据自己的实现来覆盖重写父类的实现细节，父类的实现对于子类是可见的，所以我们一般称之为白盒复用。对象持有（其实就是组合）要求建立一个号的接口，但是整体类和部分类之间不会去关心各自的实现细节，即它们之间的实现细节是不可见的，故成为黑盒复用。

     继承是在编译时刻静态定义的，即是静态复用，在编译后子类和父类的关系就已经确定了。而组合这是运用于复杂的设计，它们之间的关系是在运行时候才确定的，即在对对象没有创建运行前，整体类是不会知道自己将持有特定接口下的那个实现类。在扩展方面组合比集成更具有广泛性。

    继承中父类定义了子类的部分实现，而子类中又会重写这些实现，修改父类的实现，设计模式中认为这是一种破坏了父类的封装性的表现。这个结构导致结果是父类实现的任何变化，必然导致子类的改变。然而组合这不会出现这种现象。

    对象的组合还有一个优点就是有助于保持每个类被封装，并被集中在单个任务上（类设计的单一原则）。这样类的层次结构不会扩大，一般不会出现不可控的庞然大类。而累的继承就可能出来这些问题，所以一般编码规范都要求类的层次结构不要超过3层。组合是大型系统软件实现即插即用时的首选方式。

 最后还说一句，“优先使用对象组合，而不是继承”是面向对象设计的第二原则。但并不是说什么都设计都用组合，只是优先考虑组合，更不是说继承即使不好的设计，应该用组合，应为他们之间也有各自的优势。下面是他们之间的优缺点比比较表：

|  |  |
| --- | --- |
| 组 合 关 系 | 继 承 关 系 |
| 优点：不破坏封装，整体类与局部类之间松耦合，彼此相对独立 | 缺点：破坏封装，子类与父类之间紧密耦合，子类依赖于父类的实现，子类缺乏独立性 |
| 优点：具有较好的可扩展性 | 缺点：支持扩展，但是往往以增加系统结构的复杂度为代价 |
| 优点：支持动态组合。在运行时，整体对象可以选择不同类型的局部对象 | 缺点：不支持动态继承。在运行时，子类无法选择不同的父类 |
| 优点：整体类可以对局部类进行包装，封装局部类的接口，提供新的接口 | 缺点：子类不能改变父类的接口 |
| 缺点：整体类不能自动获得和局部类同样的接口 | 优点：子类能自动继承父类的接口 |
| 缺点：创建整体类的对象时，需要创建所有局部类的对象 | 优点：创建子类的对象时，无须创建父类的对象 |

### 45 使用静态工厂方法的好处和坏处？

静态工厂方法的优缺点分析

先看一个Boolean(基本类型boolean的包装类)的简单示例

public static Boolean valueOf(boolean b){

return b?Boolean.TRUE :Boolean.FALSE;

}

静态工厂方法的优点：

静态方法有名称具有适当名称的静态工厂方法更易于阅读，而构造器的参数本身没有确切描述被返回的对象。如：BigInteger(int, int, Random)返回的BigInteger可能为素数，如果使用BigInteger.probablePrime的静态方法来表示，就更为清楚。

不必在每次调用类时都创建一个新对象对于不可变类可以使用预先构建好的实例，或则将构建好的实例缓存起来，进行重新复用。

可以返回原返回类型的任何子类型的对象

我们在选择返回对象的类时，使用静态工厂方法就更有灵活性。这种灵活性的一种应用是API可以返回对象，同时又不会使对象的类变成公有的。这种方式隐藏实现类会使API变得非常简洁。

在Java的集合框架的接口实现中，几乎所有这些实现都通过静态工厂方法在一个不可实例化的类中导出，所有返回对象的类都是非公有的。

公有的静态工厂方法所返回的对象的类不仅可以是非公有的，还可以随着参数值的不同而每次调用也发生变化。只要是已声明的返回类型的子类型，都是允许的。

在创建参数化类型实例时，代码更加简洁

调用参数化类的构造器时，即使类型参数很明显，也必须指明，这通常要求你接连两次提供类型参数。例如创建HashMap的实例。

随着类型参数变得越来越长，冗长的说明也很快变得痛苦起来，但是使用静态工厂方法，编译器就可以替你找到类型参数，这被称作类型推导.

Google的Guava集合框架中提供了类似的Lists.newArrayList的方法。可以试着比较下面两个代码：

Map<String, List<String>> map = new HashMap<String, List<String>>();

Map<String, List<String> map =Maps.newHashMap();

下面的写法显然更简便。

静态工厂方法的缺点

一个类如果不含公有的或者受保护的构造器，就不能被子类化。其次静态工厂方法与其他静态方法实际上没有任何区别。Javadoc工具不会区别出静态工厂方法，所有静态工厂方法命名应当遵守标准的命名习惯来弥补这一劣势。惯用名称有：valeuOf，of，getInstance，getType，newType等。

### 46 排序算法有哪些？他们是时间复杂度是多少？



1、归并排序每次递归都要用到一个辅助表，长度与待排序的表长度相同，虽然递归次数是O(log2n)，但每次递归都会释放掉所占的辅助空间，

2、快速排序空间复杂度只是在通常情况下才为O(log2n)，如果是最坏情况的话，很显然就要O(n)的空间了。当然，可以通过随机化选择pivot来将空间复杂度降低到O(log2n)。

相关概念：

1、时间复杂度

     时间复杂度可以认为是对排序数据的总的操作次数。反映当n变化时，操作次数呈现什么规律。

     常见的时间复杂度有：常数阶O(1),对数阶O(log2n),线性阶O(n), 线性对数阶O(nlog2n),平方阶O(n2)

     时间复杂度O(1)：算法中语句执行次数为一个常数，则时间复杂度为O(1),

2、空间复杂度

    空间复杂度是指算法在计算机内执行时所需存储空间的度量，它也是问题规模n的函数

    空间复杂度O(1)：当一个算法的空间复杂度为一个常量，即不随被处理数据量n的大小而改变时，可表示为O(1)

    空间复杂度O(log2N)：当一个算法的空间复杂度与以2为底的n的对数成正比时，可表示为O(log2n)

                                 ax=N，则x=logaN，

    空间复杂度O(n)：当一个算法的空间复杂度与n成线性比例关系时，可表示为0(n).

### 47 spring的IOC、AOP的使用场景？

（1）AOP用来封装横切关注点，具体可以在下面的场景中使用：

Authentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading 懒加载

Debugging 调试

logging, tracing, profiling and monitoring 记录跟踪 优化 校准

Performance optimization 性能优化

Persistence 持久化

Resource pooling 资源池

Synchronization 同步

Transactions 事务

（2）IOC使用场景

在 Java EE企业应用开发中，前面介绍的IoC（控制反转）设计模式，是解耦组件之间复杂关系的利器，Spring IoC模块就是这个模式的一种实现。

在EJB模式中，应用开发人员需要编写EJB组件，而这种组件需要满足EJB容器的规范，才能运行在EJB容器中，从而获取事务管理、生命周期管理这些组件开发的基本服务。

从获取的基本服务上看，Spring提供服务和EJB容器提供的服务并没有太大的差别，只是在具体怎样获取服务的方式上，两者的设计有很大的不同：在Spring中，Spring IoC提供了一个基本的JavaBean容器，通过IoC模式管理依赖关系，并通过依赖注入和AOP切面增强了为JavaBean这样的POJO对象赋予事务管理、生命周期管理等基本功能；而对于EJB，一个简单的EJB组件需要编写远程/本地接口、Home接口以及Bean的实现类，而且EJB运行是不能脱离EJB容器的，查找其他EJB组件也需要通过诸如JNDI这样的方式，从而造成对EJB容器和技术规范的依赖。也就是说Spring把EJB组件还原成了POJO对象或者JavaBean对象，降低了应用开发对传统J2EE技术规范的依赖。

同时，在应用开发中，以应用开发人员的身份设计组件时，往往需要引用和调用其它组件的服务，这种依赖关系如果固化在组件设计中，就会造成依赖关系的僵化和维护难度的增加，这个时候，如果使用IoC容器，把资源获取的方向反转，让IoC容器主动管理这些依赖关系，将这些依赖关系注入到组件中，那么会让这些依赖关系的适配和管理更加灵活。

在具体的注入实现中，接口注入（Type 1 IoC）,setter注入（Type 2 IoC）,构造器注入（Type 3 IoC）是主要的注入方式。在Spring的IoC设计中，setter注入和构造器注入是主要的注入方式；相对而言，使用Spring时setter注入是常见的注入方式，而且为了防止注入异常，Spring IoC容器还提供了对特定依赖的检查。

另一方面，在应用管理依赖关系时，可以通过IoC容器将控制进行反转，在反转的实现中，如果能通过文本来完成配置，并且还能通过工具对这些配置信息进行可视化的管理和浏览，那么肯定是能够提高对组件关系的管理水平，并且如果耦合关系需要变动，并不需要重新修改和编译Java源码，这符合在面向对象设计中的开闭准则，并且能够提高组件系统设计的灵活性，同时，如果结合OSGi的使用特性，还可以提高应用的动态部署能力。

在具体使用Spring IoC容器的时候，我们可以看到，Spring IoC容器已经是一个产品实现。作为产品实现，它对多种应用场景的适配是通过Spring设计的IoC容器系列来实现的，比如在某个容器系列中可以看到各种带有不同容器特性的实现，可以读取不同配置信息的各种容器，从不同I/O源读取读取配置信息的各种容器设计，更加面向框架的容器应用上下文的容器设计等。

### 48 宕机的原因有哪几种，怎么样分析宕机的原因？

首先我们要对服务器宕机事件按表现方式而非导致的原因进行分类。一般来说，“运行环境”是排名第一的服务器宕机类别，大约35%的时间属于这一类。运行环境可以看作是支持数据库服务器运行的系统和资源集合，包括操作系统、硬件以及网络等。性能问题紧随其后，也是约占35%；然后是复制；最后剩下的10%包含各种类型的数据丢失或损坏，以及其他问题。

我们对服务器宕机事件按类型进行分类后，才能够确定了导致这些事件的原因。以下是一些可能引发服务器宕机的地方。

1、服务器环境的客观原因

比较常见的是机房突然断电，或者是温度过高，服务器就会出现死机、关机的情况，不过这种情况一般是不会发生的，正规的idc商会做好预防措施，备用电路和发电机，以及智能恒温系统都可以预防这种情况的发生。

2、服务器不堪负重

这是一种比较常见的情况，由于网站的流量突然大量增加，或者是受攻击、程序中毒等，都可以发生这种故障。因为流量的突增会让服务器资源耗尽，造成死机的情况。

3、不合理的应用

这种一般是公司为了减少成本的投入，租用一些价格比较低的主机，这些服务器的配置一般是很低的，但是在主机上安装一些与网站建设没有关系的大型软件，这样就会造成服务器超负荷，就会发生宕机的情况。

服务器宕机不仅仅上上面我们所说的几个原因，还有很多的细节也能导致这种情况，比如环境配置、错误程序、数据库丢失等也是其中的因素。

4、在运行环境的问题中，最普遍的问题时磁盘空间耗尽。

5、在性能问题中，最普通的服务器宕机原因确实是运行很糟糕的SQL，但也不一定都是这个原因，比如也有很多问题时由于服务器Bug或错误的行为导致的。

6、糟糕的Schema和索引设计是第二大影响性能的问题。

7、复制问题通常由于主备数据不一致导致。

8、数据丢失问题通常由于drop table的错误操作导致，并总是便随着缺少可用备份的问题。

### 49 支付宝怎么样保证他的安全？

支付宝采用了HTTPS加密传输,是无法抓包解密的。

攻击者修改了本地系统订单状态是你自己的业务逻辑与支付宝无关,为保证业务安全,你可以在点击发货时向支付宝接口查询交易状态,若未付款则拒绝。

第二种,第二天下载交易数据进行比对。

另外不要用同步通知URL来修改订单状态。

对称加密/非对称加密

### 50 谈谈对OOP、IOC、AOP的设计理念的理解？

AOP，他是一种思想，是OOP的延展。

说OOP就来说下之前的OPP，最早开始的程序，一个函数（方法）或者说是功能做一个事情，这种思想划分的是一个功能一个功能的来组合解决问题。

后来人们发现，功能解决问题是解决了，但是功能太多了，人脑实在是有限，太劳财伤命了，而且跟人们看世界的方法有点差异，人们都习惯把某一类的习性划分一起，作为一个整体研究，而不是把某一个习性放一起，很多个种类的东西都拥有这个习性。

所以就搞了一种OOP来解决问题，这个的好处就是在于把某一类或者说某一种特性的功能组合在一起，赋给抽象的对象，这样跟你我他这种人们的世界观很接近，而且不用研究你啊我啊能干什么，只要知道大家都是人类，人类会有一些通用的什么方法类之类的，而你我具体有些什么嗜好差异，就不管了，这样一来，就能把人力集合起来，各司其职了。脉络就清晰了。

再就是AOP，其实这几个都是人类生产力方式的转变有关，有一定的相通处。这个就更加细化了，他把抽象中的某些方面做了合集，就好比大公司，如果一个经理管理具体细致到人，而且管吃喝住行之类的啥都插一脚，他也很累，而且管得也不够多，但是如果他只管某一个方面的东西，比如大家吃，那样就可以多管很多人，他也能轻松很多，这样就可以找不同的经理来管不同方面，效率又会更加高，所以AOP就出来了，之前的OOP，很多种类，很多抽象的东西，你要调用某个类别的方法，至少要知道他是谁，能干嘛直接去调用，而此时你根本不想知道这些，（比如你要找个洗碗的，你肯定不想了解他能不能吃饭，个子高不高之类的，你只要他能给你干好活就可以了）因为你只关心你要的方法，其余的都是多余的。所以AOP的话，有一个专门的管理者帮你管理你要的方法，一种方面的东西，比如add什么，这样是不更省心省力。

所以说， OPP，OOP，AOP，区别就是"字母不同"！是什么意思，就是什么区别。

然后就是DI和IoC。 这个其实就某人觉得IoC不够响亮 ，所以取了个新名字而已。但是其实也是有区别的

IoC他其实也是个抽象的东西，可能不同的语言有不同的方法来实现这种抽象概念（类似设计模式的一种，现在应该算是一种设计模式了） ，而DI就是实现它的一种比较具体的做法，（例如假如某个语言没有inteface的概念，那么肯定就不会有inteface injection这中注入方式了不）。比如除了DI，工厂模式 也是用来实现IoC的一个办法，只是java有反射机制，那么用DI来实现，耦合性更低更优美。（如果某个语言没有反射，那么工厂模式应该就是用来实现的一个办法了）。

就像AOP可以用 代理模式来 实现一样，差不多的意思。

反正我感觉这些东西就像我刚开始理解 策略和工厂，代理和装饰 之类的设计模式差异一样，比较容易没有清晰的概念，一团团的，要经常琢磨，经常把看到的和写的代码实际套上去论证，慢慢就有点概念了。

总之就是一个容器，或者说一个系统里面命名出来的东西，多看看，先硬记住，然后碰到了又想想，慢慢的思想里面有这个概念了，就差不多开始掌握了，和学数学碰到新符号一样的

发现还忘了理解 AOP和IoC的关系了， 这两个其实是两方面的东西。

IoC就是把使用者要使用的东西的控制权拿到具体使用对象的外面来控制，有一个统一的系统或者说容器控制它，那么当你掌控了对象的时候，自然就有能力来控制它的方法和生死，这样你就可以把某一个方面上的方法对象行成一个面从而控制他，也就可以很好的实现AOP这种思想了。

### 51 谈谈对主流的J2EE框架（Spring、Struts、Ibatis、Hibernate等），这些框架的局限性在哪儿？在何种情况下会不适合用这些框架？

1.Spring架构图

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许您选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。Spring 框架的功能可以用在任何 J2EE 服务器中，大多数功能也适用于不受管理的环境。Spring 的核心要点是：支持不绑定到特定 J2EE 服务的可重用业务和数据访问对象。这样的对象可以在不同 J2EE 环境 （Web 或 EJB）、独立应用程序、测试环境之间重用。

常见Struts、Hibernate、Spring、J2EE、ibatis、Oracle等开发框架架构图及其简介

组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

核心容器：核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转 （IOC） 模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

Spring 上下文：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。

Spring AOP： 通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。

Spring DAO：JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写 的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

Spring ORM：Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。

2.ibatis架构图

ibatis是一个基于Java的持久层框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects（DAO），同时还提供一个利用这个框架开发的JPetStore实例。

常见Struts、Hibernate、Spring、J2EE、ibatis、Oracle等开发框架架构图及其简介

IBATIS：最大的优点是可以有效的控制sql发送的数目，提高数据层的执行效率！它需要程序员自己去写sql语句，不象hibernate那样是完全面向对象的，自动化的，ibatis是半自动化的，通过表和对象的映射以及手工书写的sql语句，能够实现比hibernate等更高的查询效率。

Ibatis只是封装了数据访问层，替我们做了部分的对象关系映射。但代价是必须要写xml配置文件，相对于Hibernate还要写很多sql。Hibernate通过工具直接从数据库模式生成实体类和基本的配置文件，而且大部分情况下不需要我们写sql，会较大的提升开发效率。但这些也有很多的局限性，尤其是对环境的要求较高（数据库设计，对象设计，团队的协作等）。 个人感觉Ibatis对项目比较有意义的地方在于它小巧灵活，可扩展，封装了数据访问层（事务，缓存，异常，日志），并提供了DAO框架支持。

利用Ibatis我们可以做到代码和sql的分离，只要sql能够解决的问题，Ibatis就能帮我们较容易的解决，同时也使我们的项目对某一框架的依赖性变小（因为Ibatis是非侵入性的）。这将极大的降低项目风险，减少解决复杂问题的时间，使项目的维护变得简单。

Ibatis对于应用的修改，调试，扩充和维护将会变得容易自然。修改时，我们主要修改的是代表模型的实体对象，xml配置文件中的sql，和/或配置文件的ResultMap（很多时候是不需要的）。同时，sql和代码分离，我们不用在代码的StringBuffer的append方法之间寻找需要修改的sql。配置文件中的sql便利了我们的调试和对sql的评审及以后的sql重用。

3.structs1架构图

Struts是Apache 基金会Jakarta 项目组的一个Open Source 项目，它采用MVC模式，能够很好地帮助java 开发者利用J2EE开发Web应用。和其他的java架构一样，Struts 也是面向对象设计，将MVC模式"分离显示逻辑和业务逻辑"的能力发挥得淋漓尽致。Structs 框架的核心是一个弹性的控制层，基于如 Java Servlets，JavaBeans，ResourceBundles与XML等标准技术，以及 Jakarta Commons 的一些类库。Struts有一组相互协作的类（组件）、Serlvet以及jsp tag lib组成。基于struts构架的web应用程序基本上符合JSP Model2的设计标准，可以说是一个传统 MVC设计模式的一种变化类型。

Struts有其自己的控制器（Controller），同时整合了其他的一些技术去实现模型层（Model）和视图层（View）。在模型层，Struts可以很容易的与数据访问技术相结合，如 JDBC / EJB ，以及其它第三方类库，如 Hibernate / iBATIS ，或者 Object Relational Bridge(对象关系桥)。在视图层，Struts能够与JSP，包括 JSTL 与 JSF，以及 Velocity 模板，XSLT 与其它表示层技术。

Struts 为每个专业的 Web 应用程序做背后的支撑，帮助为你的应用创建一个扩展的开发环境。

常见Struts、Hibernate、Spring、J2EE、ibatis、Oracle等开发框架架构图及其简介

 Client browser（客户浏览器）

来自客户浏览器的每个 HTTP 请求创建一个事件。Web 容器将用一个 HTTP 响应作出响应。

 Controller（控制器）

控制器接收来自浏览器的请求，并决定将这个请求发往何处。就 Struts 而言，控制器是以 servlet 实现的一个命令设计模式。 struts-config.xml 文件配置控制器。

 业务逻辑

业务逻辑更新模型的状态，并帮助控制应用程序的流程。就 Struts 而言，这是通过作为实际业务逻辑“瘦”包装的 Action 类完成的。

 Model（模型）的状态

模型表示应用程序的状态。业务对象更新应用程序的状态。ActionForm. bean 在会话级或请求级表示模型的状态，而不是在持久级。JSP 文件使用 JSP 标记读取来自 ActionForm. bean 的信息。

 View（视图）

视图就是一个 JSP 文件。其中没有流程逻辑，没有业务逻辑，也没有模型信息 -- 只有标记。标记是使 Struts 有别于其他框架（如 Velocity）的因素之一

4.structs2架构图

Struts 2相对于Struts 1.X，将实现用户业务逻辑（Action）同Servlet API分离开，这种分离机制，是采用了拦截器或者拦截器栈（拦截器链）。拦截器是Struts 2的核心内容之一。

Struts 2内建了多个拦截器和拦截器栈（由多个拦截器形成的拦截器链），将用户的Web请求进行拦截处理，从而提供了更加丰富的功能，例如数据类型转换、国际化、文件上传等。

常见Struts、Hibernate、Spring、J2EE、ibatis、Oracle等开发框架架构图及其简介

5.Hibernate架构图

Hibernate是一个开放源代码的对象关系映射框架，它对JDBC进行了非常轻量级的对象封装，使得Java程序员可以随心所欲的使用对象编程思维来 操纵数据库。 Hibernate可以应用在任何使用JDBC的场合，既可以在Java的客户端程序使用，也可以在Servlet/JSP的Web应用中使用，最具革命 意义的是，Hibernate可以在应用EJB的J2EE架构中取代CMP，完成数据持久化的重任。

常见Struts、Hibernate、Spring、J2EE、ibatis、Oracle等开发框架架构图及其简介

　　Hibernate的核心接口一共有5个，分别为:Session、SessionFactory、Transaction、Query和Configuration。这5个核心接口在任何开发中都会用到。通过这些接口，不仅可以对持久化对象进行存取，还能够进行事务控制。下面对这五个核心接口分别加以介绍。

·Session接口:Session接口负责执行被持久化对象的CRUD操作 (CRUD的任务是完成与数据库的交流，包含了很多常见的SQL语句。)。但需要注意的是Session对象是非线程安全的。同时，Hibernate的 session不同于JSP应用中的HttpSession。这里当使用session这个术语时，其实指的是Hibernate中的session，而 以后会将HttpSesion对象称为用户session。

·SessionFactory接口:SessionFactory接口负责初 始化Hibernate。它充当数据存储源的代理，并负责创建Session对象。这里用到了工厂模式。需要注意的是SessionFactory并不是 轻量级的，因为一般情况下，一个项目通常只需要一个SessionFactory就够，当需要操作多个数据库时，可以为每个数据库指定一个 SessionFactory。

·Configuration接口:Configuration接口负责配置并启动Hibernate，创建SessionFactory对象。在Hibernate的启动的过程中，Configuration类的实例首先定位映射文档位置、读取配置，然后创建SessionFactory对象。

·Transaction接口:Transaction接口负责事务相关的操作。它是可选的，开发人员也可以设计编写自己的底层事务处理代码。

·Query和Criteria接口:Query和Criteria接口负责执行各种数据库查询。它可以使用HQL语言或SQL语句两种表达方式。

6.J2EE架构图

J2EE是一套全然不同于传统应用开发的技术架构，包含许多组件，主要可简化且规范应用系统的开发与部署，进而提高可移植性、安全与再用价值。

J2EE核心是一组技术规范与指南，其中所包含的各类组件、服务架构及技术层次，均有共通的标准及规格，让各种依循J2EE架构的不同平台之间，存在良好的兼容性，解决过去企业后端使用的信息产品彼此之间无法兼容，导致企业内部或外部难以互通的窘境。

### 52 列举三种以上垃圾回收算法，并比较其优缺点？

在引用计数中，每一块动态分配的内存都与一个引用计数有关。这个计数在每次对内存的引用增加的时候增加1，在取消对内存的引用是减1.用C++的术语来说，这意味着每次将一个指针指向一块已分配内存的时候，

与内存相关的引用计数增加1.当这个指针指向其他位置的时候，引用计数减1.当引用计数下降为0的时候，内存不再被使用，从而可以释放。

（1）引用计数，的最大优点是其简单性----易于理解并实现。另外，它的位置不受堆结构的影响，因为引用计数不依赖于对象的物理位置。引用计数增加了每个指针操作的开销，但是回收阶段的开销相对较低。其主要的缺点是循环的引用阻止了其他不再使用的内存的释放。当两个对象互相指向对方的时候(无论是直接的还是间接的)，就会发生循环引用。在此情况下，对象的引用计数永不为0.为了解决循环引用的问题，设计了一些解决方案，但是这些方案都会增加复杂程度和开销。

（2）标记并清除：标记并清除涉及到两个阶段。在第一个阶段，堆中的所有对象都被设置为未标记状态。然后，可以由程序变量直接或者间接访问的所有对象都被标记为“正在使用”。在第二个阶段，扫描所有已分配的内存(也就是说，进行了内存的清除)，会释放所有未标记的元素。

标记并清除有两个主要优点：

首先，他很容易处理循环引用；

其次，在回收之前，他实际上没有增加运行时开销；

标记并清除也有两个主要的缺点：

首先，由于在回收的时候必须扫面整个堆，因此回收垃圾可能会花费较多的时间。因此，对于某次额程序，垃圾回收可能会导致程序运行效率低下；

其次，尽管标记并清除在概念上很简单，但是要有效地实现它并非易事；

（3）复制：复制算法将自由内存分到两个空间中。一个是活动空间(持有当前的堆)，一个是空闲空间。在垃圾回收期间，活动空间中正在使用的对象被确认，并复制到空闲空间中。然后，两个空间的角色反转，空闲空间变为活动空间，活动空间变为空闲空间。提供了复制过程中压缩堆的优点。它的缺点是在某个时刻只允许使用一个的自由内存。

采用哪种算法：

三种垃圾回收的经典算法都各自的优缺点，好像很难做出选择。然而，考虑前面列举出的限制，就会得出明显的选择：引用计数。最重要的是，引用计数可以很容易地应用与现有的C++动态分配系统上。其次，他可以以一种直接的方式来实现，而不会影响代码。第三，它不需要堆的任何特定的组织或者结构，从而不会影响C++提供的标准分配系统。

使用引用计数的一个缺点是很难处理循环引用。这对于许多程序而言，并不是一个问题，因为有意的循环引用并不常用，并且可以避免。(即使我们所说的循环，如循环队列，也不一定要用到循环指针引用)。当然，某些情况下需要使用循环引用。也可能建立了循环引用，而您并不知道，特别是使用第三方库的时候。因此，垃圾回收器必须提供某种方法来适度地处理循环引用。

为了处理循环引用问题，释放任何已经分配的内存。这将确保涉及到循环引用的对象被释放，并且调用他们的析构函数。通常在程序结束的时候，不应该再有已分配的对象，理解这一点很重要。对于涉及到循环引用而不能被释放的对象，这种机制是显示的。

实现垃圾回收器

为了实现引用计数的垃圾回收器，必须有某种方法来跟踪指向每块动态分配的内存的指针的数量。问题在于，C++没有内建的机制来确保一个对象知道其他的对象何时指向他。

幸运的是，在此有一个解决方案：可以建立一个新的支持垃圾回收的指针类型。

为了支持垃圾回收，新的指针类型必须做三件事情：

它必须为使用中的动态分配的对象维护一个引用计数的链表；

它必须跟踪所有的指针运算符，每次某个指向一个对象时，都要使这个对象的引用计数增加1，每次某个指针重新指向其他对象的时候，都要使这个对象的引用计数减1；

它必须回收那些引用计数为0的对象。除了支持垃圾回收之外，这个新指针类型与普通的指针看起来一样；

垃圾回收指针类型的建立不仅以一种简单的方法实现垃圾回收器，而且还满足不影响原始的C++动态分配系统的限制。当需要垃圾回收的时候，使用支持垃圾回收的指针。当不需要垃圾回收的时候，使用普通的C++指针。因此，这两种类型的指针在同一个程序内都可以使用。

是否使用多线程

再设计C++的垃圾回收器时，另一个考虑是应该使用单线程还是多线程。也就是，是否应该吧垃圾回收器设计为一个后台进程，在它自己的线程内运行，并且在CPU时间允许是回收垃圾。或者，这个垃圾回收器在使用它的进程的相同线程中运行，当满足某程序条件的时候回收垃圾。这两种方法各有优缺点。

建立多线程垃圾回收器的主要优点是效率。垃圾可以在CPU空闲时被回收。其缺点是，C++没有提供内建的多线程支持，这意味着威力支持多任务，任何多线程方法都依赖于操作系统是否支持多任务，这使得代码不可移植。

使用单线程垃圾回收器的主要优点是代码可以移植。在不支持多线程或者支持多线程的代价很高的情况下使用。主要缺点是当垃圾回收发生时，程序的其他部分会停止运行。

何时回收垃圾

在实现垃圾回收器之前，需要回答一个问题是：什么时候开始垃圾回收？对于多线程的垃圾回收器这不成问题，因为他可以作为后台任务连续运行，并且在CPU空闲的时候回收垃圾。然而对于单线程的垃圾回收器，为了回收垃圾，必须停止运行程序的其他部分。

实际上，只有在有足够的理由(如内存持续降低)的时候，才会进行垃圾回收。有两个原因使得这样做有意义。首先，通过某种垃圾回收算法，如标记并清除，如果不实际执行回收，就没有办法知道不在使用那块内存。其次，回收垃圾是一个耗时的过程，在不需要的时候不应该执行它。

### 53 说说JVM原理，内存泄露与溢出的区别，何时产生内存泄露？

1、什么是JVM ?

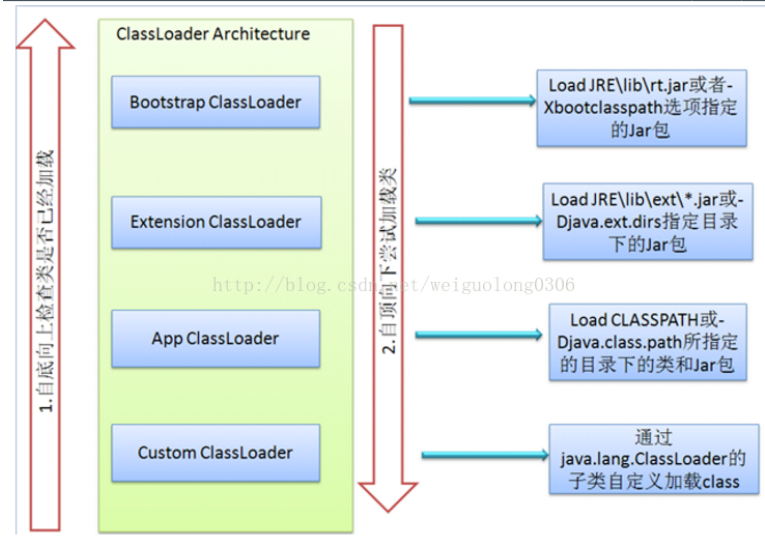
JVM， 中文名是Java虚拟机， 正如它的名字， 是一个虚拟机器，来模拟通用的物理机。 JVM是一个标准，一套规范，  规定了.class文件在其内部运行的相关标准和规范。 及其相关的内部构成。 比如：所有的JVM都是基于栈结构的运行方式。那么不符合这种要求的，不算是JVM， 如Android中所使用的Dalvik 虚拟机就不能称作是JAVA 虚拟机， 因为它是基于寄存器（最新的Android系统据说已经放弃了Dalvik VM, 而是使用ART）。

JVM相关的产品有很多， 通常最有名的莫过于现在Oracle公司所有的HotSpot 虚拟机。因此， 这里讨论的都是HotSpot虚拟机， 如果没有特别说明。

2、类加载？

类加载， 是通过JVM的类加载器从JVM外部以二进制字节流的方式加载到JVM中。但JVM本身有至少三种类加载器：BootStrap（根类加载器，C++实现， 加载位于jre/lib/rt.jar）、Extension(扩展类加载器， 主要用于加载jre/lib/ext/下的jar)、System（加载classpath环境变量所指定的class）；当然还有，自定义的类加载器（用于实现自己的类加载器， 如Tomcat中就实现多个类加载器，用来管理不同的jar）。

如果， 我有一个HelloWorld的类需要加载， 首先类加载器会去从最底层的类加载器去验证这个类是否被加载， 如果没有， 则委托给上一次的类加载器验证是否被加载， 如果到BootStrap类加载器都没有发现HelloWorld类被加载， 那么类加载器将执行加载任务， 如果根类加载器没有加载， 则委托给下一级的Extension类加载器去尝试加载，直到这个类被加载成功。 参考下图：



需要注意的是：如果一个类被不同的类加载器加载， 那么就是两个不同的类。

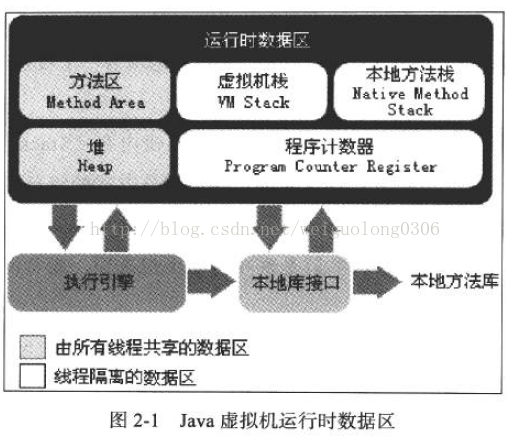
3、类加载的具体过程？

被java编译器（不仅限于， 还有其他任何的可以编辑成为.class的编译器）编译过的.class文件（可能是以jar、war、jsp等形式）， 经过类加载器加载 、 验证、准备、解析、初始化之后， 才可以被使用。基本的过程如下：

* 加载： 首先，通过一个类的全类名来获取此类的二进制字节流。其次，将类中所代表的静态存储结构转换为运行时数据结构， 最后，生成一个代表加载的类的java.lang.Class对象， 作为方法区这个类的所有数据的访问入口。加载完成之后， 虚拟机外部的二进制静态数据结构就转换成了虚拟机所需要的结构存储在方法区中（至于如何转换， 则由具体虚拟机自己定义实现）， 而所生成的Class对象， 则存放在方法区中， 用来作为程序访问方法区中数据的外部接口。
* 验证：其目的就是保证加载进来的.class文件不会危害到虚拟机本身， 且内容符合当前虚拟机规范要求。主要验证的内容大致有：文件格式、元数据验证、字节码验证、符号引用验证。其中文件格式验证， 主要确保符合class文件格式规范（如文本后缀为.class的文件将验证不通过）， 以及主次版本号， 验证是否当前JVM可以处理等。元数据验证，主要验证编译后的字节码描述信息是否符合java语法规范。字节码验证， 其最为复杂， 主要通过控制流和数据流确定语义是否合法、符合逻辑。符号引用验证，可以看做是除自身以外（常量池中各种引用符号）的信息匹配校验，如通过持有的引用能否找到对应的实例。
* 准备：正式为类变量分配内存，并设置类变量的初始值。这些变量都会在方法区中进行分配。
* 解析：将常量池内的符号引用替换为直接引用的过程。主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄等。
* 初始化：加载的最后阶段， 程序真正运行的开始。

4、java运行时数据区 ？

既然类以及加载到JVM中， 那么数据如何真正的运行？如下图：



类加载进来， JVM是通过上图所示的区域来运行和管理这些加载进来的CLASS。即程序运行的是时候， 由上面逻辑单元来运行程序， 包括：方法区、堆、本地方法栈、栈、程序计数器（PC）五大部分组成（有些VM说常量池也是其中的一个单元， 但是HotSpot VM中的常量池是方法区中的一部分）。（注意线程共享）

* 程序计数器 （PC）：可以看做是当前线程执行字节码的行号指示器。字节码解释器工作的时候就是通过这个计数器的值来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、线程恢复等基础功能依赖计数器完成。
* 虚拟机栈：和计数器一样， 也是线程私有的，生命周期同线程一致。每个方法在执行时，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、方法出口等信息。方法调入则入栈， 方法执行完则出站。局部变量表存储各种基本类型数据（java的8种，其中long，double占用2个局部变量控件，其余数据占用1个）、对象引用（reference类型）。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时， 这个方法需要在帧中分配多大的局部变量空间是完全确定的。在方法运行期间是不会改变局部变量表的大小的。
* 本地方法栈：此栈和JVM栈作用非常类似， 不同在于本地方法栈为虚拟机使用到的Native方法服务， 而JVM栈则是为Java执行的方法服务。Sun HotSpot虚拟机， 直接把本地方法栈和虚拟机栈合二为一。本地方法栈也会抛出StackOverFlowError和OutOfMemoryError异常。
* Java堆：是JVM管理内存中最大的一块。被所有线程共享一块区域。堆是GC垃圾收集器管理的主要区域。从内存回收角度看， java堆被分为新生代、老年代， 再细致一点有其他的划分。这些目的主要就是更快的分配和回收内存。

方法区：和java堆相同， 线程共享区域， 用来存储已被虚拟机加载的类信息， 常量、静态变量、即时编译器编译后的代码等数据。有人称作此方法区为“永久带”， 本质上不等价，只是HotSpot VM将GC分代收集扩展到了方法区，这样HotSpot的垃圾收集器管理方法区和管理java堆一样（优点：不用专门为方法区写一套垃圾收集器， 缺点：容易导致内存溢出）。官方现在拥也有放弃永久带并改为采用Native Memory来实现方法区的计划，目前已经发布的JDK7中的HotSpot中， 已经将原本放在方法区中的字符串常量池移出了。

运行时常量池：是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述外，还有一项就是常量池， 用于存放编译期间生成的各种字面量和符号引用 ，这部分内容在类加载后进入方法区的运行时常量池中存放。

5、垃圾收集？

在java运行时区域中， 程序计数器、虚拟机栈、本地方法栈3个区域随线程而生，随程而灭。因为这几个区域

的内存分配和回收都是具有确定性，这几个区域不需要过多考虑回收的问题。因为方法结束之后或线程结束之后，

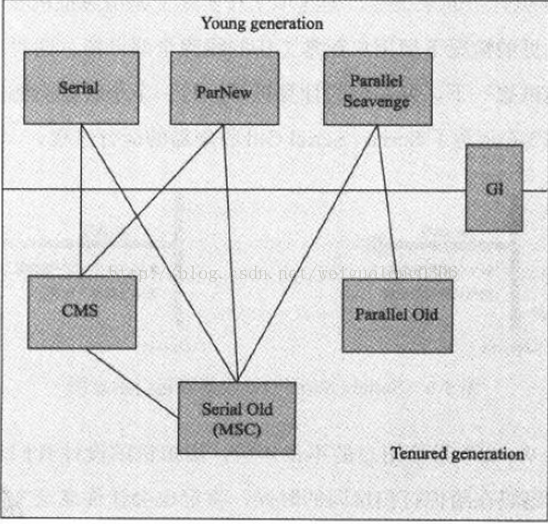
内存自然就跟着回收了（这不是绝对的， 因为如果当栈内存中的引用很消耗内存的时候， 需要手动将引用置为null，

以便垃圾收集器回收大对象）。而java堆和方法区不一样，一个接口中的多个实现类需要的内存可能不一样， 一个

方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时，才知道会创建哪些对象， 垃圾收集

器关注的就是这部分内存。其也是动态的。

垃圾收集器的区域如下图：



垃圾收集本是有一套非常复杂的算法， 如果在方法区中（HotSpot VM中的永久带）进行垃圾收集， 那么其性价比极

底的，因为垃圾回收主要收集永久带中的两部分内容：废弃的常量和无用的类。回收永久带中的常量和方法区非常相

似。但是在堆中， 尤其是在新生代中，常规应用进行一次垃圾收集， 一般可以回收70%——95%的空间。而永久带

的垃圾收集要远地与此。

如上图所示， 每一个黑框中都是一个垃圾收集器， 对应特定的垃圾收集算法， 来挺高整体的垃圾收集效率。

**内存溢出和内存泄漏的区别、产生原因以及解决方案**

内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。

内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

memory leak会最终会导致out of memory！

内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

内存泄漏是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。一个盘子用尽各种方法只能装4个果子，你装了5个，结果掉倒地上不能吃了。这就是溢出！比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。就是分配的内存不足以放下数据项序列,称为内存溢出.

以发生的方式来分类，内存泄漏可以分为4类：

1. 常发性内存泄漏。发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。

2. 偶发性内存泄漏。发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

3. 一次性内存泄漏。发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块仅且一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，所以内存泄漏只会发生一次。

4. 隐式内存泄漏。程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般的用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会最终消耗尽系统所有的内存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害性则非常大，因为较之于常发性和偶发性内存泄漏它更难被检测到

内存溢出的原因以及解决方法

引起内存溢出的原因有很多种，小编列举一下常见的有以下几种：

1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；

2.集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；

3.代码中存在死循环或循环产生过多重复的对象实体；

4.使用的第三方软件中的BUG；

5.启动参数内存值设定的过小

内存溢出的解决方案：

第一步，修改JVM启动参数，直接增加内存。(-Xms，-Xmx参数一定不要忘记加。)

第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。

第三步，对代码进行走查和分析，找出可能发生内存溢出的位置。

重点排查以下几点：

1.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

2.检查代码中是否有死循环或递归调用。

3.检查是否有大循环重复产生新对象实体。

4.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

5.检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

第四步，使用内存查看工具动态查看内存使用情况

### 54 谈谈架构师的职责有哪些？

1. 负责后台系统的架构设计和开发；

2. 构建符合业务需求、高可用、高并发、可伸缩的分布式系统，满足业务需求和用户规模；

3. 管理后台技术团队、开发流程以及后台的运维，和产品以及运营沟通，按时高质量的发布后台系统；

### 55 如果要设计一个搜索引擎，像Google那样只有两个页面，要求性能最大化，Web方面应该如何设计？

性能:

1客户端:js的写法,数据排列,不同浏览器区别.

2服务器:逻辑,计算,缓存,减少I/O,提高命中

3传输:带宽,缓存,异步,进度条,并发,集群,数据压缩.

我认为最主要的性能是人的体验,其它都是可以放到第二位去的.

### 56 描述一个你感觉最成功的一次架构案例?

**运营erp：**

**单机--集群扩容--分布式架构（流程引擎，公共组件及服务，聚引客，客常来）**

**工作流前端采用模板语言**

**后端采用 模板方法模式+策略模式**

**Maven+spring+mybatis+activci+Jenkins+分布式消息队列+mongodb++分布式缓存+**

### 57 怎么做到系统整合?

1. **通过代码的整合方式，使用相同的数据库。B、通过SSO方式，可以是异构数据库.）**

### 58 浅谈一下负载均衡的原理?

负载均衡(Load Balance，简称LB)是一种服务器或网络设备的集群技术。负载均衡将特定的业务(网络服务、网络流量等)分担给多个服务器或网络设备，从而提高了业务处理能力，保证了业务的高可用性。负载均衡基本概念有：实服务、实服务组、虚服务、调度算法、持续性等，其常用应用场景主要是服务器负载均衡，链路负载均衡。......

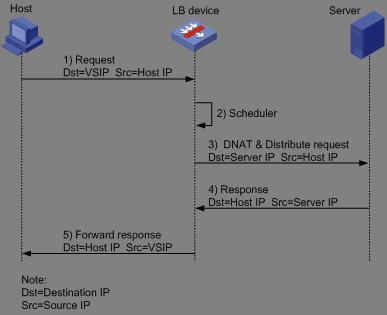
负载均衡(Load Balance，简称LB)是一种服务器或网络设备的集群技术。负载均衡将特定的业务(网络服务、网络流量等)分担给多个服务器或网络设备，从而提高了业务处理能力，保证了业务的高可用性。负载均衡基本概念有：实服务、实服务组、虚服务、调度算法、持续性等，其常用应用场景主要是服务器负载均衡，链路负载均衡。

**一 服务器负载均衡**

服务器负载均衡根据LB设备处理到的报文层次，分为四层服务器负载均衡和七层负载均衡，四层处理到IP包的IP头，不解析报文四层以上载荷(L4 SLB);七层处理到报文载荷部分，比如HTTP，RTSP，SIP报文头，有时也包括报文内容部分(L7 SLB)。

**1.四层服务器负载均衡技术**

客户端将请求发送给服务器群前端的负载均衡设备，负载均衡设备上的虚服务接收客户端请求，通过调度算法，选择真实服务器，再通过网络地址转换，用真实服务器地址重写请求报文的目标地址后，将请求发送给选定的真实服务器;真实服务器的响应报文通过负载均衡设备时，报文的源地址被还原为虚服务的VSIP，再返回给客户，完成整个负载调度过程。报文交互流程如下：



NAT方式的服务器负载均衡报文交互流程图报文交互流程说明：

(1)Host发送服务请求报文，源IP为Host IP、目的IP为VSIP

(2)LB Device接收到请求报文后，借助调度算法计算出应该将请求分发给哪台Server

(3)LB Device使用DNAT技术分发报文，源IP为Host IP、目的IP为Server IP

(4)Server接收并处理请求报文，返回响应报文，源IP为Server IP、目的IP为Host IP

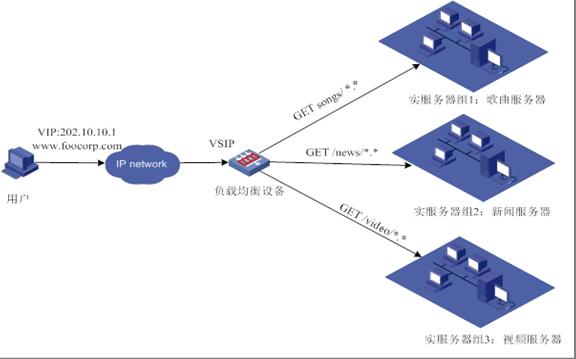
(5)LB Device接收响应报文，转换源IP后转发，源IP为VSIP、目的IP为Host IP

**2.七层服务器负载均衡技术**

七层负载均衡和四层负载均衡相比，只是进行负载均衡的依据不同，而选择确定的实服务器后，所做的处理基本相同，下面以HTTP应用的负载均衡为例来说明。

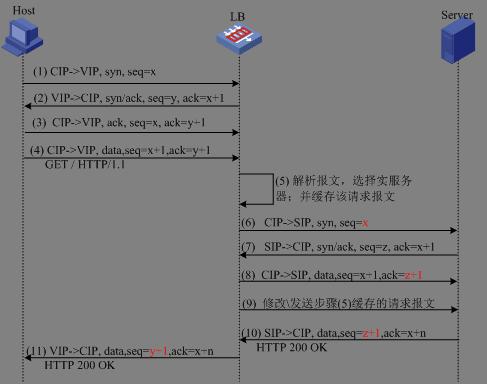
由于在TCP握手阶段，无法获得HTTP真正的请求内容，因此也就无法将客户的TCP握手报文直接转发给服务器， 必须由负载均衡设备先和客户完成TCP握手，等收到足够的七层内容后，再选择服务器，由负载均衡设备和所选服务器建立TCP连接。

七层负载均衡组网和四层负载均衡组网有一个显著的区别：四层负载均衡每个虚服务对应一个实服务组，实服务组内的所有实服务器提供相同的服务;七层负载均衡每个虚服务对应多个实服务组，每组实服务器提供相同的服务。根据报文内容选择对应的实服务组，然后根据实服务组调度算法选择某一个实服务器。



七层负载均衡组网图

上图中描述了基于HTTP的URI目录信息进行的七层负载均衡部署，报文交互流程图如下：



七层负载均衡报文交互流程图报文交互流程说明：

(1)-(3)：Client和LB建立TCP连接;

(4)：Client发送HTTP请求，目的IP为虚IP;

(5)：LB设备分析报文，根据调度算法选择实服务器，注意此时会缓存该报文;

(6)：LB设备向实服务器发Syn报文，序列号为Client的Syn报文序列号

(7)：Server发送Syn/Ack报文，目的IP为Client;

(8)：LB接收Server的Syn/Ack报文后，回应ACK报文

(9)：修改步骤(5)中缓存的报文目的IP和TCP序列号，然后发给Server;

(10)：Server发送响应报文到LB;

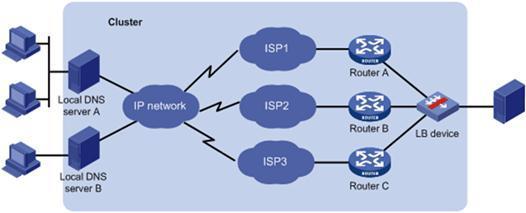
(11)：LB修改步骤(9)中的报文的源地址和TCP序列号后转发给Client。

**二 链路负载均衡**

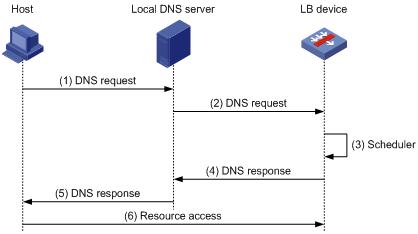
在企业网、运营商链路出口需要部署LB设备以优化链路选择，提升访问体验，链路负载均衡按照流量发起方向分为Inbound负载均衡和Outbound负载均衡

**1.Inbound入方向负载均衡**

Inbound负载均衡技术是DNS智能解析的一种，外网用户通过域名访问内部服务器时，Local DNS的地址解析请求到达LB设备，LB根据对Local DNS的就近性探测结果响应一个最优的IP地址，外网用户根据这个最优的IP响应进行对内部服务器的访问。



Inbound链路负载均衡组网图



 入方向负载均衡

流程简述如下：

(1)外部用户进行资源访问前先进行DNS解析，向其本地DNS服务器发送DNS请求。

(2)本地DNS服务器将DNS请求的源IP地址替换为自己的IP地址，并转发给域名对应的权威服务器——LB device。

(3)LB device根据DNS请求的域名和配置的Inbound链路负载均衡规则进行域名解析。

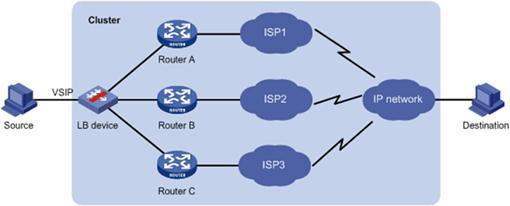
(4)LB device按照域名解析的结果，将DNS应答发送给本地DNS服务器。

(5)本地DNS服务器将解析结果转发给用户。

(6)用户使用解析结果选择的链路，直接对LB device进行资源访问。

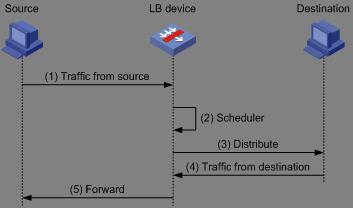
**2.Outbound出方向负载均衡**

内网用户访问Internet上其他服务器。 Outbound链路负载均衡中VSIP为内网用户发送报文的目的网段。用户将访问VSIP的报文发送到负载均衡设备后，负载均衡设备依次根据策略、持续性功能、就近性算法、调度算法选择最佳的链路，并将内网访问外网的业务流量分发到该链路。



Outbound链路负载均衡组网图

Outbound负载均衡报文交互流程如下：



 Outbound 链路负载均衡流程图

Outbound负载均衡报文交互流程说明：

(1)LB Device接收内网用户流量 -

(2)LB Device依次根据策略、持续性功能、就近性算法、调度算法进行链路选择 在Outbound链路负载均衡组网中，通常使用就近性算法或带宽调度算法实现流量分发

(3)LB device按照链路选择的结果将流量转发给选定的链路 -

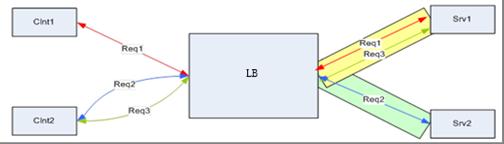
(4)LB Device接收外网用户流量 -

(5)LB Device将流量转发给内网用户

**三 负载均衡优化及应用**

**1.TCP连接复用**

连接复用功能通过使用连接池技术，可以将前端大量的客户的HTTP请求复用到后端与服务器建立的少量的TCP长连接上，大大减小服务器的性能负载，减小与服务器之间新建TCP连接所带来的延时，并最大限度减少后端服务器的并发连接数，降低服务器的资源占用。

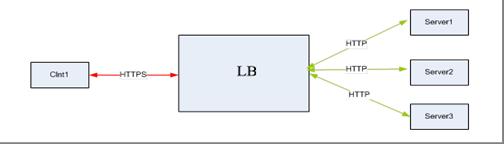


TCP连接复用示意图上图给出了TCP连接复用的简单过程描述。由Client端发送的Req1/ Req2/ Req3三个HTTP请求，经过LB设备后，复用了LB设备和Server端已经建立好的连接，将Client端的三个请求通过两个TCP连接发送给了服务器端。

**2.SSL卸载**

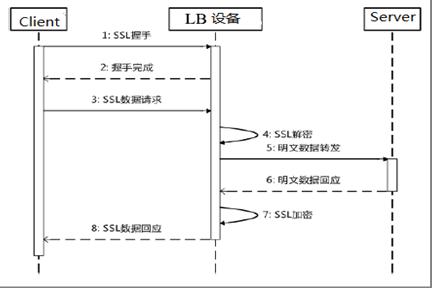
为了避免明文传输出现的安全问题，对于敏感信息，一般采用SSL协议，如HTTPS，对HTTP协议进行加密，以保证整个HTTP传输过程的安全性。SSL是需要耗费大量CPU资源的一种安全技术，如果由后端的服务器来承担，则会消耗很大的处理能力。应用交付设备为了提升用户的体验，分担服务器的处理压力，将SSL加解密集中在自身的处理上，相对于服务器来说LB能提供更高的SSL处理性能，还能够简化对证书的管理，减少日常管理的工作量，LB的该功能又称为SSL卸载。

下图中Client端发送给Server的所有的HTTPS流量都被LB设备终结，LB设备将SSL终结后，与Server之间可采用HTTP或者弱加密的HTTPS进行通讯。LB设备承担了SSL的卸载工作，从而极大的减小了服务器端对SSL处理的压力，将服务器的处理能力释放出来，更加专注于处理服务器本身承担的业务逻辑。



SSL卸载示意图

SSL卸载的处理流程如下：



SSL卸载过程

(1)客户端向服务器端发送SSL握手请求。

(2)LB设备作为中间的卸载设备，代替服务器端和客户端交互，完成SSL握手过程。

(3)客户端发送SSL加密后的请求数据。

(4)LB设备解密数据。

(5)LB设备将解密后的明文发送给Server。

(6)服务器返回给LB设备回应报文。

(7)LB设备将返回的应答报文加密。

(8)LB设备将加密后的应答报文传给客户端。

**3.DRX云环境应用交付**

业务负载监控平台通过H3C负载均衡设备的参数设定和监控可以动态感知业务负载变化，并通知云管理平台动态调整业务资源。由此实现用户业务资源的实时动态调整、业务资源优化调配。

当业务负载监控平台发现业务资源需要调整时：业务负载超限—增加资源;业务资源过剩——回收资源，云管理平台通过自动创建、启动或者删除停止虚拟机的方式为业务进行资源动态调整。

**四 结束语**

负载均衡技术不管应用于用户访问服务器资源，还是应用于多链路出口，均大大提高了对资源的高效利用，显著降低了用户的网络布署成本，提升了用户的网络使用体验。随着云计算的发展，负载均衡的技术实现还将与云计算相结合，在虚拟化和NFV软件定义网关等方面持续发展。

### 59 用JAVA如何实现每天1亿条记录的数据存储，数据库方面怎么设计？

一天秒数：60\*60\*24=86,400秒

每天写入数据量：100,000,000条

平均每秒写入数据量：100,000,000/86,400=1157.5条

峰值每秒估算写入数：1157.5\*10=11575条

因此建议从以下几个层面处理

1、数据库服务器磁盘采用高速SSD磁盘

2、数据库采用2个节点的集群方式部署，每个集群节点3台服务器，1主2备，主数据库为写数据库，备数据库为读数据，采用读写分离，单集群节点内主备库数据实时同步，集群节点主库数据实时同步

3、数据表设计采用分区、分表方式设计表结构

4、数据写入采用单事务批量写入的方法新增数据

5、在关键字段建立索引，提高查询效率

6、第一次查询后将数据缓存到radis中，方便下次查询

7、大的应用集群，插入时使用缓冲，比如每1000条插入一次

8、横向分割，分成不同的数据库（不是表）

### 60 对应大表数据是如何处理；以及数据库性能调优策略？

（**索引，SQL语句效率(切忌全表扫描)，数据迁移，水平切面等**）

1、索引优化和SQL语句优化是必须的，避免模糊查询和非索引查询，删改操作根据聚集索引进行，删改操作太频繁的话还是需要考虑分表

2、看需求，如果需求不限制，那就分表

分区会增加管理复杂度和成本这个很难理解，分区增加不了多少工作，如果需求要求必须单表，分区是解决在千万到几亿数据量的比较合适的方法

可能更大数据量还是要回到分的路上，但是可能更多考虑分布式

3、我们一般都是把历史数据定期转存其他表（一样的表名后加年月例如TABLE201205）归档~

这样该表本年度的查询的压力也小点（90%查询量集中在本年度）,即使查询历史数据也不影响性能，强力推荐！

4、

（1）从结构上来说，分区很有必要，我听过一些培训，微软给客户的建议是一个表如果大小超过50M，那就建议分区了。而且分区几乎是“一次性”的事情，不会增加什么管理成本。

（2）可以使用归档方式管理历史数据。其实你的数据量不大啦，我以前做银行系统，单表就2亿多，40G的大小。

（3）优化你的语句和设计。

（4）结合你的业务去晚上结构。有时候可以考虑用空间去换时间。

有数据表明：用户可以承受的最大等待时间为8秒。

之前曾见过某个产品的一个列表页，40秒左右才能加载出来，几乎没有进行任何优化措施。

没有索引，没有缓存机制，没有进行sql优化(sql语句很长，并且各种left join表关联)。

数据库优化策略有很多，设计初期，建立好的数据结构对于后期性能优化至关重要。因为数据库结构是系统的基石，基础打不好，使用各种优化策略，也不能达到很完美的效果。

一：规范化与反规范化

大家都听说过：数据库设计三大范式.

1．第一范式(确保每列保持原子性)

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。

2．第二范式(确保表中的每列都和主键相关)

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

3．第三范式(确保每列都和主键列直接相关,而不是间接相关)

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

没有最好的设计，只有最合适的设计，所以不要过分注重理论。三范式可以作为一个基本依据，不要生搬硬套。

数据库操作中最为耗时的操作就是 IO 处理，大部分数据库操作 90% 以上的时间都花在了 IO 读写上面。所以尽可能减少 IO 读写量，可以在很大程度上提高数据库操作的性能。

二:优化策略：

在设计表时应同时考虑对某些表进行反规范化，方法有以下几种:

一是分割表。

分割表可分为水平分割表和垂直分割表两种:

水平分割是按照行将一个表分割为多个表，这可以提高每个表的查询速度，但查询、更新时要选择不同的表，统计时要汇总多个表，因此应用程序会更复杂。

垂直分割是对于一个列很多的表，若某些列的访问频率远远高于其它列，就可以将主键和这些列作为一个表，将主键和其它列作为另外一个表。通过减少列的宽度，增加了每个数据页的行数，一次I/O就可以扫描更多的行，从而提高了访问每一个表的速度。但是由于造成了多表连接，所以应该在同时查询或更新不同分割表中的列的情况比较少的情况下使用。

二是保留冗余列。当两个或多个表在查询中经常需要连接时，可以在其中一个表上增加若干冗余的列，以避免表之间的连接过于频繁，一般在冗余列的数据不经常变动的情况下使用。

三是增加派生列。派生列是由表中的其它多个列的计算所得，增加派生列可以减少统计运算，在数据汇总时可以大大缩短运算时间。

在数据库的设计中，数据应当按两种类别进行组织:频繁访问的数据和频繁修改的数据。

对于频繁访问但是不频繁修改的数据，内部设计应当物理不规范化。

对于频繁修改但并不频繁访问的数据，内部设计应当物理规范化。

有时还需将规范化的表作为逻辑数据库设计的基础，然后再根据整个应用系统的需要，物理地非规范化数据。

规范与反规范都是建立在实际的操作基础之上的约束，脱离了实际两者都没有意义。只有把两者合理地结合在一起，才能相互补充，发挥各自的优点。

适当拆分

有些时候，我们可能会希望将一个完整的对象对应于一张数据库表，这对于应用程序开发来说是很有好的，但是有些时候可能会在性能上带来较大的问题。

当我们的表中存在类似于 TEXT 或者是很大的 VARCHAR类型的大字段的时候，如果我们大部分访问这张表的时候都不需要这个字段，我们就该义无反顾的将其拆分到另外的独立表中，以减少常用数据所占用的存储空间。这样做的一个明显好处就是每个数据块中可以存储的数据条数可以大大增加，既减少物理 IO 次数，也能大大提高内存中的缓存命中率。

适度冗余

为什么我们要冗余?这不是增加了每条数据的大小，减少了每个数据块可存放记录条数吗?

确实，这样做是会增大每条记录的大小，降低每条记录中可存放数据的条数，但是在有些场景下我们仍然还是不得不这样做：

1.被频繁引用且只能通过 Join 2张(或者更多)大表的方式才能得到的独立小字段。

2.这样的场景由于每次Join仅仅只是为了取得某个小字段的值，Join到的记录又大，会造成大量不必要的 IO，完全可以通过空间换取时间的方式来优化。不过，冗余的同时需要确保数据的一致性不会遭到破坏，确保更新的同时冗余字段也被更新。

三：其他技巧：

1:字段类型优化

下面的这些关于字段类型的优化建议主要适用于记录条数较多，数据量较大的场景，因为精细化的数据类型设置可能带来维护成本的提高，过度优化也可能会带来其他的问题：

(1)数字类型

非万不得已不要使用DOUBLE，不仅仅只是存储长度的问题，同时还会存在精确性的问题。同样，固定精度的小数，也不建议使用DECIMAL。

非万不得已不要使用DOUBLE，不仅仅只是存储长度的问题，同时还会存在精确性的问题。同样，固定精度的小数，也不建议使用DECIMAL

(2)字符类型

非万不得已不要使用 TEXT 数据类型，其处理方式决定了他的性能要低于char或者是varchar类型的处理。定长字段，建议使用 CHAR 类型，不定长字段尽量使用 VARCHAR，且仅仅设定适当的最大长度，而不是非常随意的给一个很大的最大长度限定，因为不同的长度范围，MySQL也会有不一样的存储处理。

(3)时间类

尽量使用TIMESTAMP类型，因为其存储空间只需要 DATETIME 类型的一半。对于只需要精确到某一天的数据类型，建议使用DATE类型，因为他的存储空间只需要3个字节，比TIMESTAMP还少。不建议通过INT类型类存储一个unix timestamp 的值，因为这太不直观，会给维护带来不必要的麻烦，同时还不会带来任何好处。

2:合理使用索引

3：缓存机制

4：用EXPLAIN使你的SELECT查询更加清晰

5：利用LIMIT 1取得唯一行

6： 尽量避免SELECT \*命令

7：使用ENUM而不是VARCHAR

8:尽可能的使用NOT NULL

　NULL 类型比较特殊，SQL 难优化。虽然 MySQL NULL类型和 Oracle 的NULL 有差异，会进入索引中，但如果是一个组合索引，那么这个NULL 类型的字段会极大影响整个索引的效率。此外，NULL 在索引中的处理也是特殊的，也会占用额外的存放空间。

　很多人觉得 NULL 会节省一些空间，所以尽量让NULL来达到节省IO的目的，但是大部分时候这会适得其反，虽然空间上可能确实有一定节省，倒是带来了很多其他的优化问题，不但没有将IO量省下来，反而加大了SQL的IO量。所以尽量确保 DEFAULT 值不是 NULL，也是一个很好的表结构设计优化习惯。

### 61 分布式系统，数据库设计方面，应注意哪些方面？

**(权限设计、图片存储、服务器集群设计等)**

**分布式是分布式，集群是集群。分布式比如有ABCD用户，将AB提交的数据写到数据库1去，CD用户群组的数据提交到数据库2去优点：**

**降低单个数据库的压力（数据量少，单个数据库不会被频繁操作到，如提交事务，少几个人少去抢占资源），不然，像上面单个数据库同时要处理四个人的数据，肯定比处理两个的压力小得多了（PS：1、2两个数据库的内容是不一致的，但是结构是一致的）。**

**响应速度快，一般用于大客户的解决方案缺点：用的硬件较多，开发成本较高（不像单台机器那样，不管如何都写到一台机器上）。**

**集群：构建多个相同数据库（内容一样），类似备份(一个数据一生成一条数据，立马将该数据同步到另一台上面去，查询时可以多台任选一台，单台压力低)，一台挂了，别一台上面还是有备份的，也有分布式的优点，不过，需要集群的几台机器网络环境较好，数据推送才能及时。**

最近在做系统升级，由于当时设计的局限，导致系统不停服，保证服务的做法非常麻烦。当时再定方案的时候，由于自己在这方面没有经验，导致有些乐观。到了实际做的时候，预期时间至少比预想的多了一周的时间，要知道，在

互联网公司，一周的时间是个非常长的时间。而这一周，还包括了OT。

在这里总结一下分布式系统设计的大忌，本来想试着分一下级，但是还是算了，一来标准太多，无法制定一个合适的规则来界定；二来自己的经验也在增长，低调一下是自己也没详细的研究过超过5个分布式系统；三来做事情还是要严谨，不做没有十足把握的事情。

1. 服务接口的设计至关重要

虽然大家口口声声说对于一个集群来说，每台机器都可能出故障。但是做方案设计的时候，某些资源却向用户直接暴漏了服务的实际地址。对于一个服务几年的服务器来说，故障的可能性非常大，尤其是如果这个服务器的平时负载比较高的话。我不清楚一台服务器的平均保修时间是多少，但是绝对不可能是几个小时能搞定的，这个时间少则一天，多则半个月甚至更长。对于一些高级的用户，它会使用本地的cache，或者其他的策略来屏蔽调用服务不可用带来的影响，但是，几天的停服对于用户方的影响是无论如何不可能忽略的。

这种问题发现后，可能简单的发布一个新版本的api，或者一个简单的配置文件就可以纠正。但是对于线上用户来说，他们运行的是一个一直都在running状态的服务。这个简单的改正可能需要他们服务重启，这对于一个大型的集群来说，带来的成本非常高。如果是因为这个服务的不可用导致了线上事故，那么应用方肯定会非常主动的去修正这个错误。但是如果使用架构方发现了这个问题，而主动推动应用方去修改，可能应用方会因为各种原因而推脱。

因此，设计服务的接口一定要注意，这个接口一定要是稳定的，而且后台服务的故障，升级等操作绝对对于用户要是透明的。不要将服务的实际地址暴漏给用户方:这台服务器终有一天会挂掉。尤其是对于C++等需要编译的api来说，这个接口就更加重要了。毕竟api的修改对于应用方来说意味着要重新编译；重新编译意味着要重新走一下发布流程：至少要提测吧。

2. 后台升级要做到对用户透明

这实际上是又是一句大家都知道的。但是设计时确实有时候会忽略。对于弹性计算系统来说，服务的伸缩是必须的，这个也是设计的目标之一。但是对于一些小规模的计算集群来说，可能大家认为伸缩不是最重要的feature。最重要的feature就是能够快速的完成系统设计和实现，为用户服务。但是实际上，这个通过一些简单的修改，就可以完成：Worker上带一个agent和master或者meta server通信，保持心跳。心跳超时的Worker会被下线，以后的服务都不会发送到这个Worker上来。而新加入的Worker则会加入集群接收计算任务。这个不单是应对服务的伸缩，也是为了应对机器的故障。因此不用太大的改动，就可以将一个系统从山寨提升到真正的可用。

一个系统的服务质量，不是说在一般情况下的服务是可靠的，除了网络丢包、网络传输造成的问题外，服务质量可以做到10000个请求至多有1个失败就是说这个系统是可用的。评价服务质量的另外一个重要指标是全年可服务时间。这个要将机器故障，机房故障考虑在内。如果依赖于运行环境没有问题，才能达到99.99，那么这个服务就有点山寨，对于重要的应用方来说，这种服务不可接受。

3. 应用方设计时候需要衡量后台服务失败的影响

如果服务的可靠性要求非常高，比如是直接面向互联网用户的，要求任何时间都能够对互联网用户提供服务，那么就需要在调用服务时做下服务不可用的预案。甚至做下超时机制：如果服务调用指定时间不返回，那么需要有其余的逻辑来替代。

当然了本次还遇到很多其他的痛点，每个都是设计上得小瑕疵，当时注意的话不会增加工作量，或者增加很少的工作量就可以做到可用。互联网强调快，那么底线应该是可用吧。易用可能是更要的要求。当然了这个可能可以一种互联网风格，就是一个事情可以快速做完，快速上线。当时上线时候也做了二期需要做的改进，但是后台发现上线效果好，符合预期。又去做其它高优先级的事情去了。导致原来设计的局限就永远的停留在那里了，这就是为后来人埋下一个坑。。

本次升级的时候，由于信息的不一致导致一台服务器停服，导致大面积的失败。后来为了避免其它的集群出现类似的问题，因此所有的信息都重新确认了一遍。而这带来了半天的枯燥工作。因此，自己做设计的时候，一定要注意，不求最好，但求可用，在机器故障，服务升级，对于用户来说，服务都可用。

### 62 当用户反映，平台访问变的很慢的时候，怎样处理这个问题的?

1. **数据库端；B、后端应用平台端；C、前端Web端；D、负载均衡；E、网络设置；F、机器性能的优化；G、考虑是否有病毒、木马等干扰等等）**

问题场景：某个用户向你反映说你开发的网站访问速度很慢，但是该用户访问其他问题很正常，分析下原因、有哪些工具分析原因、怎么解决问题？

最近面试两次碰到了这个题目，回来请教了一位做运维的师兄，听他讲解下发现确实这个问题可以牵涉到很多知识面，很具综合型，是个好题目，不过其实这个问题偏运维，但开发人员自然也是越懂越好。结合这位师兄的详细讲解，我梳理了下这个问题牵涉到的一些点。

一、针对这个题目我们可以简单理解成是server端出现的问题，而不是client端出现了问题（用户网络不好包括域名服务器解析等可能），当然面试官要考你用户端的知识，例如域名解析，也是有挺多可以考到的知识点，但单就这个问题，更强调的是server端的知识点。下面逐一来剖析可能的原因：

（1）可能的原因一：服务器出口带宽不够用。这是一个很常见的瓶颈。一方面，可能是本身购买的服务器出口带宽就很小（企业购买带宽相当昂贵），一旦用户访问量上来了，并发量大了，自然均分给用户的出口带宽就更小了，所以某些用户的访问速度就会下降了很多。另一个，就是跨运营商网络导致带宽缩减，例如很多公司的网站（服务器）是放在电信的网络上的，而如果用户这边对接的是长城或者说联通的宽带，运营商之间网络传输在对接时是会有限制的，这就可能导致带宽的缩减。

（2）可能原因二：服务器负载过大忙不过来，比如说CPU和内存消耗完了，这个容易理解，不展开。

（3）可能原因三：网站的开发代码没写好，例如mysql语句没有进行优化，导致数据库的读写相当耗费时间。

（4）可能原因四：数据库的瓶颈，也是很常见的一个瓶颈，这点跟上面第三个原因可以一起来说。当我们的数据库变得愈发庞大，比如好多G好多T这么大，那对于数据库的读写就会变得相当缓慢了，索引优化固然能提升一些效率，但数据库已经如此庞大的话，如果每次查询都对这么大的数据库进行全局查询，自然会很慢。这个学过数据库的话也是挺容易理解的。

二、针对上面可能的原因，有哪些方法和工具去检测呢：

（1）某个用户反馈网站访问变慢，怎么去定位问题。首先你自己也打开下网站，看是否会出现用户反映的问题，如果你这边访问没问题，那就可能是用户那边的问题了，这块就是要先确定是用户那一方的问题还是自身比如说服务器或者网站的问题。

（2）发现确实是自己服务器或者网站的问题，那么可以利用浏览器的调试功能（一般浏览器都会有），调试网络看看各种数据加载的速度，哪一项消耗了多少时间都可以看到，是哪块数据耗时过多，是图片加载太慢，还是某些数据加载老半天都查不出来。

（3）然后针对服务器的负载情况，可以去查看下服务器硬件（网络带宽、CPU、内存）的消耗状况。带宽方面查看流量监控看是不是已经到了峰值，带宽不够用了，如果是公司自己买服务器搭的网站服务器的话，需要自己搭建监控环境；如果用的是阿里云腾讯云这些的，那这些平台那边会提供各方面的监控比如CPU、带宽等等，在后台就可以看到了。

（4）如果发现硬件资源消耗都不高，都比较充裕，那要去看看是不是程序的问题了。这个可以通过查日志来找，比如PHP日志、Apache日志、mysql日志等等的错误日志，特别如mysql有个慢查询的日志功能，可以看到是不是某条mysql语句特别慢，如果某条语句花的时间太长，那这条语句很有可能有问题。

（5）至于说到的数据库太庞大，这个直接看就看得到了，比如一个表的文件大小变得特别大了。

三、针对上面的这些问题，有哪些解决和优化的办法呢：

（1）出口带宽的问题，这个很简单，加带宽，有钱就多买带宽，很简单。

（2）mysql语句优化，开发人员职责。

（3）数据库太庞大，为了读写速度，进行“拆表”、“拆库”，就是把数据表或者数据库进行拆分。

（4）上面的拆库拆表都是针对数据库实在太庞大才会这样做，一般在此之前会有其他优化方法，比如mysql的主从复制，一台主服务器专门用于写，然后其他从服务器用来读，写完之后会同步更新到其他读的服务器中。例如阿里的双十一活动，都不知道用了多少万台服务器一起在扛着。

（6）还有这几年用得比较多的非关系型数据库，它使用了缓存机制，它把数据缓存到了内存，用户访问数据直接从内存读取，读取速度就比在磁盘中读取快了很多，还有它的一个key-value读取机制，这个听师兄说之后没听懂。

（7）CDN（content-delivery-network：内容分发网络），鸡蛋放在多个篮子里，把数据放在离用户更近的位置（例如网站的一些静态文件比如图片或者js脚本），用户访问时判断IP来源是广州，那就通过智能DNS解析到广州的服务器上，直接从广州的篮子里去获取数据，速度就快了。这里有个静态数据和动态数据的概念，例如图片和一些js文件一般是不变的，那就可以把它们的映像分布到全国各地，加快速度，而一些需要在网站后台动态产生的一些数据，则需要去到网站所在的服务器去产生并得到。这个涉及到两种数据的显示的问题，这就交由浏览器处理了。同时异步加载的技术例如前端的Ajax技术，异步请求数据，可以使这些动态数据延迟加载，这块自己不怎么了解，可能表述不好。前端开发的人员应该更懂一些。

（8）上面都没有说到架构的优化，如果网站扛不住，是不是网站架构已经不能适应了，比如做个小博客把数据库服务器和web服务器都用同一台服务器，那所有负载都在同一台服务器上了。但是访问量上来扛不住了，就得加服务器了，就得在架构上优化了，比如在数据库上做集群，在web服务器上也做集群，比如web服务器集群，在服务器前面加一个负载均衡，负载均衡就是专门负责分发，把用户的请求均匀分布到各个服务器上。

### 63 介绍一下你主导过的项目；

设计开发思想

技术实现

任务安排

进度控制

版本控制

需求控制

资源管理

### 64 你认为负责一个大型集成系统的系统分析与架构设计，需要具备哪些能力？

**运营erp：**

**单机--集群扩容--分布式架构（流程引擎，公共组件及服务，聚引客，客常来）**

**工作流前端采用模板语言**

**后端采用 模板方法模式+策略模式**

**Maven+spring+mybatis+activci+Jenkins+分布式消息队列+mongodb++分布式缓存+**

系统架构师负责设计系统整体架构，从需求到设计的每个细节都要考虑到，把握整个项目，使设计的项目尽量效率高，开发容易，维护方便，升级简单，等等

系统架构师的职责：

一、理解系统的业务需求，制定系统的整体框架（包括：技术框架和业务框架）

二、对系统框架相关技术和业务进行培训，指导开发人员开发。并解决系统开发、运行中出现的各种问题。 系统架构师的目的：

对系统的重用、扩展、安全、性能、伸缩性、简洁等做系统级的把握。 系统架构师能力要求：

一、系统架构相关的知识和经验。

二、很强的自学能力、分析能力、解决问题的能力。

三、写作、沟通表达、培训。

角色

软件架构师Software Architect 定义

主导系统全局分析设计和实施、负责软件构架和关键技术决策的角色 职责

l、领导与协调整个项目中的技术活动（分析、设计和实施等） 2、推动主要的技术决策，并最终表达为软件构架 3、确定和文档化系统的相对构架而言意义重大的方面，包括系统的需求、设计、实施和部署等“视图” 4、确定设计元素的分组以及这些主要分组之间的接口 5、为技术决策提供规则，平衡各类涉众的不同关注点，化解技术风险，并保证相关决定被有效的传达和贯彻 6、理解、评价并接收系统需求 7、评价和确认软件架构的实现 专业技能

l、技术全面、成熟练达、洞察力强、经验丰富，具备在缺乏完整信息、众多问题交织一团、模糊和矛盾的情况下，迅速抓住问题要害，并做出合理的关键决定的能力 l、具备战略性和前瞻性思维能力，善于把握全局，能够在更高抽象级别上进行思考； l、对项目开发涉及的所有问题领域都有经验，包括彻底地理解项目需求，开展分析设计之类软件工程活动等 2、具备领导素质，以在各小组之间推进技术工作，并在项目压力下做出牢靠的关键决策 3、拥有优秀的沟通能力，用以进行说服、鼓励和指导等活动，并赢得项目成员的信任； 4、以目标导向和主动的方式来不带任何感情色彩地关注项目结果，构架师应当是项目背后的技术推动力，而非构想者或梦想家（追求完美） 5、精通构架设计的理论、实践和工具，并掌握多种参考构架、主要的可重用构架机制和模式（例如J2EE架构等）； 6、具备系统设计员的所有技能，但涉及面更广、抽象级别更高； 活动

确定用例或需求的优先级、进行构架分析、创建构架的概念验证原型、评估构架的概念验证原型的可行性、组织系统实施模型、描述系统分布结构、描述运行时刻构架、确定设计机制、确定设计元素、合并已有设计元素 工件

软件构架文档、参考构架、分析模型、设计模型、实施模型、部署模型、构架概念验证原型、接口、事件、信号与协议 系统架构师

### 65 tomcat的链接模式，是单线程还是多线程？

tomcat是怎样多线程处理http请求并将代码执行到controller里的的

1.线程池，thread = threadPool.getThread(),thread.executeHttp(htttpRequest),thread的start方法执行里面调用：每个thread里再获取所有的controller，根据传进入thread的httprequest找到相应的controllerer对象获取出来，controller对象就开始执行了嘛。

2.轨迹：线程池-》线程-》传request->线程找到对应的controller，执行

3.Main线程负责向子线程传入参数，任何线程的启动都是由主线程来启动加载的

1. tomcat中，并发的请求是采用多线程处理：并发过程中，每个请求带来的处理开始会从线程池取一个线程，如果并发量高于阈值则会等待线程池有空余

2.tomcat 本身对于每个请求接收处理到调用相应servlet中间这段过程，是线程安全的，放心容器肯定考虑到这个问题

3.tomcat中servlet采用的单例模式，所以servlet需要注意尽量写成线程安全类

### 66 tomcat启动加载java项目的过程？

基于Java的Web 应用程序是 servlet、JSP 页面、静态页面、类和其他资源的集合，它们可以用标准方式打包，并运行在来自多个供应商的多个容器。Web 应用程序存在于结构化层次结构的目录中，该层次结构是由 Java Servlet 规范定义的。Web 应用程序的根目录包含直接存储或存储在子文件夹中的所有公共资源，比如图像、HTML 页面等。构成：Web应用由Web组件(一组Java类库)、html文件，静态资源文件（如图像）、帮助类和库组成。

1 – Tomcat Server的组成部分

1.1 – Server

A Server element represents the entire Catalina servlet container. (Singleton)

1.2 – Service

A Service element represents the combination of one or more Connector components that share a single Engine

Service是这样一个集合：它由一个或者多个Connector组成，以及一个Engine，负责处理所有Connector所获得的客户请求

1.3 – Connector

一个Connector将在某个指定端口上侦听客户请求，并将获得的请求交给Engine来处理，从Engine处获得回应并返回客户

TOMCAT有两个典型的Connector，一个直接侦听来自browser的http请求，一个侦听来自其它WebServer的请求

Coyote Http/1.1 Connector 在端口8080处侦听来自客户browser的http请求

Coyote JK2 Connector 在端口8009处侦听来自其它WebServer(Apache)的servlet/jsp代理请求

1.4 – Engine

The Engine element represents the entire request processing machinery associated with a particular Service

It receives and processes all requests from one or more Connectors

and returns the completed response to the Connector for ultimate transmission back to the client

Engine下可以配置多个虚拟主机Virtual Host，每个虚拟主机都有一个域名

当Engine获得一个请求时，它把该请求匹配到某个Host上，然后把该请求交给该Host来处理

Engine有一个默认虚拟主机，当请求无法匹配到任何一个Host上的时候，将交给该默认Host来处理

1.5 – Host

代表一个Virtual Host，虚拟主机，每个虚拟主机和某个网络域名Domain Name相匹配

每个虚拟主机下都可以部署(deploy)一个或者多个Web App，每个Web App对应于一个Context，有一个Context path

当Host获得一个请求时，将把该请求匹配到某个Context上，然后把该请求交给该Context来处理

匹配的方法是“最长匹配”，所以一个path==””的Context将成为该Host的默认Context

所有无法和其它Context的路径名匹配的请求都将最终和该默认Context匹配

1.6 – Context

一个Context对应于一个Web Application，一个Web Application由一个或者多个Servlet组成

Context在创建的时候将根据配置文件$CATALINA\_HOME/conf/web.xml和$WEBAPP\_HOME/WEB-INF/web.xml载入Servlet类

当Context获得请求时，将在自己的映射表(mapping table)中寻找相匹配的Servlet类

如果找到，则执行该类，获得请求的回应，并返回

2 – Tomcat Server的结构图

3 – 配置文件$CATALINA\_HOME/conf/server.xml的说明

该文件描述了如何启动Tomcat Server

<!----------------------------------------------------------------------------------------------->

<!-- 启动Server 在端口8005处等待关闭命令 如果接受到"SHUTDOWN"字符串则关闭服务器 -->

<Server port="8005" shutdown="SHUTDOWN" debug="0">

<!-- Listener ??? 目前没有看到这里 -->

<Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" debug="0"/>

<Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" debug="0"/>

<!-- Global JNDI resources ??? 目前没有看到这里，先略去 -->

<GlobalNamingResources>

... ... ... ...

</GlobalNamingResources>

<!-- Tomcat的Standalone Service Service是一组Connector的集合 它们共用一个Engine来处理所有Connector收到的请求 -->

<Service name="Tomcat-Standalone">

<!-- Coyote HTTP/1.1 Connector className : 该Connector的实现类是org.apache.coyote.tomcat4.CoyoteConnector port :

在端口号8080处侦听来自客户browser的HTTP1.1请求 minProcessors : 该Connector先创建5个线程等待客户请求，

每个请求由一个线程负责 maxProcessors : 当现有的线程不够服务客户请求时，若线程总数不足75个，则创建新线程来处理请求

acceptCount : 当现有线程已经达到最大数75时，为客户请求排队 当队列中请求数超过100时，后来的请求返回Connection refused

错误 redirectport : 当客户请求是https时，把该请求转发到端口8443去 其它属性略 -->

<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"

port="8080"

minProcessors="5" maxProcessors="75" acceptCount="100"

enableLookups="true"

redirectPort="8443"

debug="0"

connectionTimeout="20000"

useURIValidationHack="false"

disableUploadTimeout="true" />

<!-- Engine用来处理Connector收到的Http请求 它将匹配请求和自己的虚拟主机，

并把请求转交给对应的Host来处理默认虚拟主机是localhost -->

<Engine name="Standalone" defaultHost="localhost" debug="0">

<!-- 日志类，目前没有看到，略去先 -->

<Logger className="org.apache.catalina.logger.FileLogger" .../>

<!-- Realm，目前没有看到，略去先 -->

<Realm className="org.apache.catalina.realm.UserDatabaseRealm" .../>

<!-- 虚拟主机localhost appBase : 该虚拟主机的根目录是webapps/ 它将匹配请求和

自己的Context的路径，并把请求转交给对应的Context来处理 -->

<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true" autoDeploy="true">

<!-- 日志类，目前没有看到，略去先 -->

<Logger className="org.apache.catalina.logger.FileLogger" .../>

<!-- Context，对应于一个Web App path : 该Context的路径名是""，故该Context是该Host的

默认Context docBase : 该Context的根目录是webapps/mycontext/ -->

<Context path="" docBase="mycontext" debug="0"/>

<!-- 另外一个Context，路径名是/wsota -->

<Context path="/wsota" docBase="wsotaProject" debug="0"/>

</Host>

</Engine>

</Service>

</Server>

<!----------------------------------------------------------------------------------------------->

4 – Context的部署配置文件web.xml的说明

一个Context对应于一个Web App，每个Web App是由一个或者多个servlet组成的

当一个Web App被初始化的时候，它将用自己的ClassLoader对象载入“部署配置文件web.xml”中定义的每个servlet类

它首先载入在$CATALINA\_HOME/conf/web.xml中部署的servlet类

然后载入在自己的Web App根目录下的WEB-INF/web.xml中部署的servlet类

web.xml文件有两部分：servlet类定义和servlet映射定义

每个被载入的servlet类都有一个名字，且被填入该Context的映射表(mapping table)中，和某种URL PATTERN对应

当该Context获得请求时，将查询mapping table，找到被请求的servlet，并执行以获得请求回应

分析一下所有的Context共享的web.xml文件，在其中定义的servlet被所有的Web App载入

<!----------------------------------------------------------------------------------------------->

<web-app>

<!-- 概述： 该文件是所有的WEB APP共用的部署配置文件， 每当一个WEB APP

被DEPLOY，该文件都将先被处理，然后才是WEB APP自己的/WEB-INF/web.xml -->

<!-- +-------------------------+ -->

<!-- | servlet类定义部分 | -->

<!-- +-------------------------+ -->

<!-- DefaultServlet

当用户的HTTP请求无法匹配任何一个servlet的时候，该servlet被执行

URL PATTERN MAPPING : / -->

<servlet>

<servlet-name>default</servlet-name>

<servlet-class>

org.apache.catalina.servlets.DefaultServlet

</servlet-class>

<init-param>

<param-name>debug</param-name>

<param-value>0</param-value>

</init-param>

<init-param>

<param-name>listings</param-name>

<param-value>true</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- InvokerServlet

处理一个WEB APP中的匿名servlet 当一个servlet被编写并编译放入

/WEB-INF/classes/中，却没有在/WEB-INF/web.xml中定义的时候

该servlet被调用，把匿名servlet映射成/servlet/ClassName的形式

URL PATTERN MAPPING : /servlet/\* -->

<servlet>

<servlet-name>invoker</servlet-name>

<servlet-class>org.apache.catalina.servlets.InvokerServlet </servlet-class>

<init-param>

<param-name>debug</param-name>

<param-value>0</param-value>

</init-param>

<load-on-startup>2</load-on-startup>

</servlet>

<!-- JspServlet

当请求的是一个JSP页面的时候（\*.jsp）该servlet被调用

它是一个JSP编译器，将请求的JSP页面编译成为servlet再执行

URL PATTERN MAPPING : \*.jsp -->

<servlet>

<servlet-name>jsp</servlet-name>

<servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>

<init-param>

<param-name>logVerbosityLevel</param-name>

<param-value>WARNING</param-value>

</init-param>

<load-on-startup>3</load-on-startup>

</servlet>

<!-- +-

<!-- | servlet映射定义部分 | -->

<!-- +---------------------------+ -->

<servlet-mapping>

<servlet-name>default</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>invoker</servlet-name>

<url-pattern>/servlet/\*</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>jsp</servlet-name>

<url-pattern>\*.jsp</url-pattern>

</servlet-mapping>

<!-- +------------------------+ -->

<!-- | 其它部分，略去先 | -->

<!-- +------------------------+ -->

... ... ... ...

</web-app>

<!----------------------------------------------------------------------------------------------->

5 – Tomcat Server处理一个http请求的过程

假设来自客户的请求为：

http://localhost:8080/wsota/wsota\_index.jsp

1) 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得

2) Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应

3) Engine获得请求localhost/wsota/wsota\_index.jsp，匹配它所拥有的所有虚拟主机Host

4) Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）

5) localhost Host获得请求/wsota/wsota\_index.jsp，匹配它所拥有的所有Context

6) Host匹配到路径为/wsota的Context（如果匹配不到就把该请求交给路径名为””的Context去处理）

7) path=”/wsota”的Context获得请求/wsota\_index.jsp，在它的mapping table中寻找对应的servlet

8) Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类

9) 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法

10)Context把执行完了之后的HttpServletResponse对象返回给Host

11)Host把HttpServletResponse对象返回给Engine

12)Engine把HttpServletResponse对象返回给Connector

13)Connector把HttpServletResponse对象返回给客户browser

### 67 垃圾回收的最佳做法？

垃圾收集算法：

1、标记-清除算法：

首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。标记过程中 实际上即时上面说的finaLize()的过程。主要缺点一个是效率问题。另外一个是空间问题，标记清除后会产生大量不连续的内存碎片。

2、复制算法：

这种算法将可用内存按容量划分为大小相等的两块，每次只使用其中的一块，当这一块的内存用完了。就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次清理掉。

3、标记-整理算法：

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会遍低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以对应被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

标记过程仍然与标记-清除算法一样，但是后续步骤不是直接将对可回收对象进行清理，而是让所有存活的对象都向领一端移动，然后直接清理掉端边界以外的内存。

4、分代收集算法：

当代商业虚拟机的垃圾收集都采用的是“分代收集算法” ，根据对象的存活周期的不同，将内存化为几块，一般是把java堆分为新生代和老年代。这样就可以根据各个年代的特点采用最合适的收集算法。

新生代选用复制算法，老年代使用标记-清理算法 或者 标记-整理算法。

### 68 简述redis提供的6种数据淘汰策略？

在 redis 中，允许用户设置最大使用内存大小 server.maxmemory，在内存限定的情况下是很有用的。譬如，在一台 8G 机子上部署了 4 个 redis 服务点，每一个服务点分配 1.5G 的内存大小，减少内存紧张的情况，由此获取更为稳健的服务。

　　redis中当内存超过限制时，按照配置的策略，淘汰掉相应的kv，使得内存可以继续留有足够的空间保存新的数据。redis 确定驱逐某个键值对后，会删除这个数据并，并将这个数据变更消息发布到本地（AOF 持久化）和从机（主从连接）。

　　redis的conf文件中有对该机制的一份很好的解释：

注意：在redis按照master-slave使用时，其maxmeory应设置的比实际物理内存稍小一些，给slave output buffer留有足够的空间。

　　redis 提供 6种数据淘汰策略：

1 volatile-lru:从设置了过期时间的数据集中，选择最近最久未使用的数据释放；

2 allkeys-lru:从数据集中(包括设置过期时间以及未设置过期时间的数据集中)，选择最近最久未使用的数据释放；

3 volatile-random:从设置了过期时间的数据集中，随机选择一个数据进行释放；

4 allkeys-random:从数据集中(包括了设置过期时间以及未设置过期时间)随机选择一个数据进行入释放；

5 volatile-ttl：从设置了过期时间的数据集中，选择马上就要过期的数据进行释放操作；

6 noeviction：不删除任意数据(但redis还会根据引用计数器进行释放),这时如果内存不够时，会直接返回错误。

默认的内存策略是noeviction，在Redis中LRU算法是一个近似算法，默认情况下，Redis随机挑选5个键，并且从中选取一个最近最久未使用的key进行淘汰，在配置文件中可以通过maxmemory-samples的值来设置redis需要检查key的个数,但是栓查的越多，耗费的时间也就越久,但是结构越精确(也就是Redis从内存中淘汰的对象未使用的时间也就越久~),设置多少，综合权衡。

### 69 Dubbo源码使用了哪些设计模式？

1、工厂模式

ServiceConfig中有个字段，代码是这样的：

private static final Protocol protocol =ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();

Dubbo里有很多这种代码。这也是一种工厂模式，只是实现类的获取采用了jdkspi的机

制。这么实现的优点是可扩展性强，想要扩展实现，只需要在classpath下增加个文件就可

以了，代码零侵入。另外，像上面的Adaptive实现，可以做到调用时动态决定调用哪个实

现，但是由于这种实现采用了动态代理，会造成代码调试比较麻烦，需要分析出实际调用的

实现类。

2、装饰器模式

Dubbo在启动和调用阶段都大量使用了装饰器模式。以Provider提供的调用链为例，具

体的调用链代码是在ProtocolFilterWrapper的buildInvokerChain完成的，具体是将注解中含

有group=provider的Filter实现，按照order排序，最后的调用顺序是

查看文本打印

EchoFilter-》ClassLoaderFilter-》GenericFilter-》ContextFilter-》ExceptionFilter-》

TimeoutFilter-》MonitorFilter-》TraceFilter。

更确切地说，这里是装饰器和责任链模式的混合使用。例如，EchoFilter的作用是判断

是否是回声测试请求，是的话直接返回内容，这是一种责任链的体现。而像ClassLoaderFilter

则只是在主功能上添加了功能，更改当前线程的ClassLoader，这是典型的装饰器模式。

3、观察者模式

Dubbo的provider启动时，需要与注册中心交互，先注册自己的服务，再订阅自己的服

务，订阅时，采用了观察者模式，开启一个listener。注册中心会每5秒定时检查是否有服

务更新，如果有更新，向该服务的提供者发送一个notify消息，provider接受到notify消息

后，即运行NotifyListener的notify方法，执行监听器方法。

4、动态代理模式

Dubbo扩展jdkspi的类ExtensionLoader的Adaptive实现是典型的动态代理实现。Dubbo

需要灵活地控制实现类，即在调用阶段动态地根据参数决定调用哪个实现类，所以采用先生

成代理类的方法，能够做到灵活的调用。生成代理类的代码是ExtensionLoader的

createAdaptiveExtensionClassCode方法。代理类的主要逻辑是，获取URL参数中指定参数的

值作为获取实现类的key。

### 70 多线程同步锁有哪些？

A，RentrantLock，可重入的互斥锁，可中断可限时，公平锁，必须在finally释放锁，而synchronize由JVM释放。可重入但是要重复退出，普通的lock()不能响应中断，lock.lockInterruptbly()可响应中断，可以限时tryLock()，超时返回false，不会永久等待构成死锁。

B，Confition条件变量，signal唤醒其中1个在等待的线程，signalall唤醒所有在等待的线程await()等待并释放锁，与lock结合使用。

C，semaphore信号量，多个线程比（额度=10）进入临界区，其他则阻塞在临界区外。

D，ReadWriteLock，读读不互斥，读写互斥，写写互斥。

E，CountDownLantch倒数计时器，countdown()和await()

F，CyCliBarrier

G，LockSupport，方法park和unpark

java允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查），

将会导致数据不准确，相互之间产生冲突，因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，

从而保证了该变量的唯一性和准确性。

同步的方式

1.同步方法

即有synchronized关键字修饰的方法。

由于java的每个对象都有一个内置锁，当用此关键字修饰方法时，

内置锁会保护整个方法。在调用该方法前，需要获得内置锁，否则就处于阻塞状态。

代码如：

public synchronized void save(){}

注： synchronized关键字也可以修饰静态方法，此时如果调用该静态方法，将会锁住整个类

2.同步代码块

即有synchronized关键字修饰的语句块。

被该关键字修饰的语句块会自动被加上内置锁，从而实现同步

代码如：

synchronized(object){

}

注：同步是一种高开销的操作，因此应该尽量减少同步的内容。

通常没有必要同步整个方法，使用synchronized代码块同步关键代码即可。

package com.xhj.thread;

/\*\*

\* 线程同步的运用

\*

\* @author XIEHEJUN

\*

\*/

public class SynchronizedThread {

class Bank {

private int account = 100;

public int getAccount() {

return account;

}

/\*\*

\* 用同步方法实现

\*

\* @param money

\*/

public synchronized void save(int money) {

account += money;

}

/\*\*

\* 用同步代码块实现

\*

\* @param money

\*/

public void save1(int money) {

synchronized (this) {

account += money;

}

}

}

class NewThread implements Runnable {

private Bank bank;

public NewThread(Bank bank) {

this.bank = bank;

}

@Override

public void run() {

for (int i = 0; i < 10; i++) {

// bank.save1(10);

bank.save(10);

System.out.println(i + "账户余额为：" + bank.getAccount());

}

}

}

/\*\*

\* 建立线程，调用内部类

\*/

public void useThread() {

Bank bank = new Bank();

NewThread new\_thread = new NewThread(bank);

System.out.println("线程1");

Thread thread1 = new Thread(new\_thread);

thread1.start();

System.out.println("线程2");

Thread thread2 = new Thread(new\_thread);

thread2.start();

}

public static void main(String[] args) {

SynchronizedThread st = new SynchronizedThread();

st.useThread();

}

}

3.使用特殊域变量(volatile)实现线程同步

a.volatile关键字为域变量的访问提供了一种免锁机制，

b.使用volatile修饰域相当于告诉虚拟机该域可能会被其他线程更新，

c.因此每次使用该域就要重新计算，而不是使用寄存器中的值

d.volatile不会提供任何原子操作，它也不能用来修饰final类型的变量

例如：

在上面的例子当中，只需在account前面加上volatile修饰，即可实现线程同步。

代码实例：

1

2

class Bank {

//需要同步的变量加上volatile

private volatile int account = 100;

public int getAccount() {

return account;

}

//这里不再需要synchronized

public void save(int money) {

account += money;

}

｝

多线程中的非同步问题主要出现在对域的读写上，如果让域自身避免这个问题，则就不需要修改操作该域的方法。

4.使用重入锁实现线程同步

在JavaSE5.0中新增了一个java.util.concurrent包来支持同步。

ReentrantLock类是可重入、互斥、实现了Lock接口的锁，

它与使用synchronized方法和快具有相同的基本行为和语义，并且扩展了其能力

ReenreantLock类的常用方法有：

ReentrantLock() : 创建一个ReentrantLock实例

lock() : 获得锁

unlock() : 释放锁

class Bank {

private int account = 100;

//需要声明这个锁

private Lock lock = new ReentrantLock();

public int getAccount() {

return account;

}

//这里不再需要synchronized

public void save(int money) {

lock.lock();

try{

account += money;

}finally{

lock.unlock();

}

}

｝

注：关于Lock对象和synchronized关键字的选择：

a.最好两个都不用，使用一种java.util.concurrent包提供的机制，

能够帮助用户处理所有与锁相关的代码。

b.如果synchronized关键字能满足用户的需求，就用synchronized，因为它能简化代码

c.如果需要更高级的功能，就用ReentrantLock类，此时要注意及时释放锁，否则会出现死锁，通常在finally代码释放锁

5.使用局部变量实现线程同步

如果使用ThreadLocal管理变量，则每一个使用该变量的线程都获得该变量的副本，

副本之间相互独立，这样每一个线程都可以随意修改自己的变量副本，而不会对其他线程产生影响。

ThreadLocal 类的常用方法

ThreadLocal() : 创建一个线程本地变量

get() : 返回此线程局部变量的当前线程副本中的值

initialValue() : 返回此线程局部变量的当前线程的"初始值"

set(T value) : 将此线程局部变量的当前线程副本中的值设置为value

例如：

在上面例子基础上，修改后的代码为：

public class Bank{

//使用ThreadLocal类管理共享变量account

private static ThreadLocal<Integer> account = new ThreadLocal<Integer>(){

@Override

protected Integer initialValue(){

return 100;

}

};

public void save(int money){

account.set(account.get()+money);

}

public int getAccount(){

return account.get();

}

}

注：ThreadLocal与同步机制

a.ThreadLocal与同步机制都是为了解决多线程中相同变量的访问冲突问题。

b.前者采用以”空间换时间”的方法，后者采用以”时间换空间”的方式

### 71 栈溢出的原因有哪些？

是否递归的调用；大量循环；全局变量是否过多；数组，List，Map数据是否过大；用DDMS工具检查地方。

一、局部数组过大。当函数内部的数组过大时，有可能导致堆栈溢出。

二、递归调用层次太多。递归函数在运行时会执行压栈操作，当压栈次数太多时，也会导致堆栈溢出。

三、指针或数组越界。这种情况最常见，例如进行字符串拷贝，或处理用户输入等等。

### 72 内存溢出的原因

过多使用了static；static最好只用int和string等基本类型；大量的递归或者死循环；大数据项的查询，如返回表的所有记录，应该采用分页查询。检查是否有数组、List、map中存放的是对象的引用而不是对象，这些引用会让对应对象不能被释放。

栈过大会导致内存占用过多，频繁页交换阻碍效率。

ava.lang.OutOfMemoryError这个错误我相信大部分开发人员都有遇到过，产生该错误的原因大都出于以下原因：JVM内存过小、程序不严密，产生了过多的垃圾。

导致OutOfMemoryError异常的常见原因有以下几种：

内存中加载的数据量过于庞大，如一次从数据库取出过多数据；

集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；

代码中存在死循环或循环产生过多重复的对象实体；

使用的第三方软件中的BUG；

启动参数内存值设定的过小；

此错误常见的错误提示：

tomcat:java.lang.OutOfMemoryError: PermGen space

tomcat:java.lang.OutOfMemoryError: Java heap space

weblogic:Root cause of ServletException java.lang.OutOfMemoryError

resin:java.lang.OutOfMemoryError

java:java.lang.OutOfMemoryError

解决java.lang.OutOfMemoryError的方法有如下几种：

一、增加jvm的内存大小。方法有： 1）在执行某个class文件时候，可以使用java -Xmx256M aa.class来设置运行aa.class时jvm所允许占用的最大内存为256M。 2）对tomcat容器，可以在启动时对jvm设置内存限度。对tomcat，可以在catalina.bat中添加：

set CATALINA\_OPTS=-Xms128M -Xmx256M

set JAVA\_OPTS=-Xms128M -Xmx256M

或者把%CATALINA\_OPTS%和%JAVA\_OPTS%代替为-Xms128M -Xmx256M

3）对resin容器，同样可以在启动时对jvm设置内存限度。在bin文件夹下创建一个startup.bat文件，内容如下：

@echo off

call "httpd.exe" "-Xms128M" "-Xmx256M"

:end

其中"-Xms128M"为最小内存，"-Xmx256M"为最大内存。

二、 优化程序，释垃圾。

主要包括避免死循环，应该及时释放种资源：内存, 数据库的各种连接，防止一次载入太多的数据。导致java.lang.OutOfMemoryError的根本原因是程序不健壮。因此，从根本上解决Java内存溢出的唯一方法就是修改程序，及时地释放没用的对象，释放内存空间。 遇到该错误的时候要仔细检查程序，嘿嘿，遇多一次这种问题之后，以后写程序就会小心多了。

Java代码导致OutOfMemoryError错误的解决：

需要重点排查以下几点：

检查代码中是否有死循环或递归调用。

检查是否有大循环重复产生新对象实体。

检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

tomcat中java.lang.OutOfMemoryError: PermGen space异常处理

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Loader时就会被放到PermGen space中, 它和存放类实例(Instance)的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的应用中有很多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。如果你的WEB APP下都用了大量的第三方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了。 解决方法： 手动设置MaxPermSize大小修改TOMCAT\_HOME/bin/catalina.sh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，这样可以达到减少jar 文档重复占用内存的目的。

weblogic中java.lang.OutOfMemoryError异常处理

错误提示： Root cause of ervletException java.lang.OutOfMemoryError 解决办法：调整bea/weblogic/common中CommEnv中参数

:sun

　　if "%PRODUCTION\_MODE%" == "true" goto sun\_prod\_mode

　　set JAVA\_VM=-client

　　set MEM\_ARGS=-Xms256m -Xmx512m -XX:MaxPermSize=256m

　　set JAVA\_OPTIONS=%JAVA\_OPTIONS% -Xverify:none

　　goto continue

　　:sun\_prod\_mode

　　set JAVA\_VM=-server

　　set MEM\_ARGS=-Xms256m -Xmx512m -XX:MaxPermSize=256m

　　goto continue

Resin下java.lang.OutOfMemoryError异常处理

产生内存溢出的原因：

出现这个错误，一般是因为JVM物理内存过小。默认的Java虚拟机最大内存仅为64兆，这在开发调试过程中可能没有问题，但在实际的应用环境中是远远不能满足需要的，除非你的应用非常小，也没什么访问量。否则你可能会发现程序运行一段时间后包java.lang.OutOfMemoryError的错误。因此我们需要提升resin可用的虚拟机内存的大小。

解决方法：

修改/usr/local/resin/bin/httpd.sh中的args选项 添加参数-Xms（初始内存）和-Xmx（最大能够使用内存大小）可以用来限制JVM的物理内存使用量。例如：

args="-Xms128m -Xmx256m"

设置后，JVM初始物理内存是128m，最大能使用物理内存为256m。

这两个值应该由系统管理员根据服务器的实际情况进行设置。

### 73 单例模式的7种写法？

懒汉2种，枚举，饿汉2种，静态内部类，双重校验锁（推荐）。

第一种（懒汉，线程不安全）：

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

这种写法lazy loading很明显，但是致命的是在多线程不能正常工作。

第二种（懒汉，线程安全）：

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

第三种（饿汉）：

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() {

return instance;

}

}

这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用getInstance方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化instance显然没有达到lazy loading的效果。

第四种（饿汉，变种）：

public class Singleton {

private Singleton instance = null;

static {

instance = new Singleton();

}

private Singleton (){}

public static Singleton getInstance() {

return this.instance;

}

}

表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化instance。

第五种（静态内部类）：

public class Singleton {

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

private Singleton (){}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别）：第三种和第四种方式是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式相比第三和第四种方式就显得很合理。

第六种（枚举）：

public enum Singleton {

INSTANCE;

public void whateverMethod() {

}

}

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

第七种（双重校验锁）：

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

这个是第二种方式的升级版，俗称双重检查锁定，详细介绍请查看：http://www.ibm.com/developerworks/cn/java/j-dcl.html

在JDK1.5之后，双重检查锁定才能够正常达到单例效果。

总结

有两个问题需要注意：

1.如果单例由不同的类装载器装入，那便有可能存在多个单例类的实例。假定不是远端存取，例如一些servlet容器对每个servlet使用完全不同的类装载器，这样的话如果有两个servlet访问一个单例类，它们就都会有各自的实例。

2.如果Singleton实现了java.io.Serializable接口，那么这个类的实例就可能被序列化和复原。不管怎样，如果你序列化一个单例类的对象，接下来复原多个那个对象，那你就会有多个单例类的实例。

对第一个问题修复的办法是：

private static Class getClass(String classname)

throws ClassNotFoundException {

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

if(classLoader == null)

classLoader = Singleton.class.getClassLoader();

return (classLoader.loadClass(classname));

}

}

对第二个问题修复的办法是：

public class Singleton implements java.io.Serializable {

public static Singleton INSTANCE = new Singleton();

protected Singleton() {

}

private Object readResolve() {

return INSTANCE;

}

}

对我来说，我比较喜欢第三种和第五种方式，简单易懂，而且在JVM层实现了线程安全（如果不是多个类加载器环境），一般的情况下，我会使用第三种方式，只有在要明确实现lazy loading效果时才会使用第五种方式，另外，如果涉及到反序列化创建对象时我会试着使用枚举的方式来实现单例，不过，我一直会保证我的程序是线程安全的，而且我永远不会使用第一种和第二种方式，如果有其他特殊的需求，我可能会使用第七种方式，毕竟，JDK1.5已经没有双重检查锁定的问题了。

========================================================================

superheizai同学总结的很到位：

不过一般来说，第一种不算单例，第四种和第三种就是一种，如果算的话，第五种也可以分开写了。所以说，一般单例都是五种写法。懒汉，恶汉，双重校验锁，枚举和静态内部类。

我很高兴有这样的读者，一起共勉。

### 74 lucence倒排索引？

三个文件：字典文件，频率文件，位置文件。词典文件不仅保存有每个关键词，还保留了指向频率文件和位置文件的指针，通过指针可以找到该关键字的频率信息和位置信息。

field的概念，用于表达信息所在位置（如标题中，文章中，url中），在建索引中，该field信息也记录在词典文件中，每个关键词都有一个field信息(因为每个关键字一定属于一个或多个field)。

关键字是按字符顺序排列的（lucene没有使用B树结构），因此lucene可以用二元搜索算法快速定位关键词。

假设要查询单词 “live”，lucene先对词典二元查找、找到该词，通过指向频率文件的指针读出所有文章号，然后返回结果。词典通常非常小，因而，整个过程的时间是毫秒级的。

对词典文件中的关键词进行了压缩，关键词压缩为<前缀长度，后缀>，例如：当前词为“阿拉伯语”，上一个词为“阿拉伯”，那么“阿拉伯语”压缩为<3，语>。对数字的压缩，数字只保存与上一个值的差值。

关系数据库不适合做全文搜索:

like '%xxx%'效率很慢，建的索引将无效，查询的时候会像翻书一样一页一页的翻

返回的结果没有匹配度的概念，比如可能希望搜索的关键词在文章中出现的次数越多越是我想要的文章

当搜索live的时候，也想把LIVE/lives/living搜出来，但是数据库很难做到

一、简介

倒排索引源于实际应用中需要根据属性的值来查找记录。

这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址、出现频次等相关信息。

由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。

搜索引擎的关键步骤就是建立倒排索引.

当用户搜索的时候先找到关键词，也就找到了含有该关键词的相关文章，然后计算出最相关的前几十篇文章返回给用户。

二、Lucene倒排索引原理

(0)设有两篇文章1和2

文章1的内容为：Tom lives in Guangzhou,I live in Guangzhou too

文章2的内容为：He once lived in Shanghai.

(1)获取关键字

全文分析：由于lucene是基于关键词索引和查询的，首先我们要取得这两篇文章的关键词，

通常我们需要如下处理措施:

a.我们现在有的是文章内容，即一个字符串，我们先要找出字符串中的所有单词，即分词。

英文单词由于用空格分隔，比较好处理。中文单词间是连在一起的需要特殊的分词处理。

b.文章中的”in”, “once” “too”等词没有什么实际意义，中文中的“的”“是”等字通常也无具体含义，

这些不代表概念的词可以过滤掉

c.用户通常希望查“He”时能把含“he”，“HE”的文章也找出来，所以所有单词需要统一大小写。

d.用户通常希望查“live”时能把含“lives”，“lived”的文章也找出来，所以需要把“lives”，“lived”还原成“live”

e.文章中的标点符号通常不表示某种概念，也可以过滤掉

经过上面处理后:

　　文章1的所有关键词为：[tom] [live] [guangzhou] [i] [live] [guangzhou]

　　文章2的所有关键词为：[he] [live] [shanghai]

(2) 建立倒排索引

有了关键词后，我们就可以建立倒排索引了。

上面的对应关系是：“文章号”对“文章中所有关键词”。

倒排索引把这个关系倒过来，变成：“关键词”对“拥有该关键词的所有文章号”。

文章1，2经过倒排后变成:

关键词 文章号

　　guangzhou 1

　　he 2

　　i 1

　　live 1,2

　　shanghai 2

　　tom 1

通常仅知道关键词在哪些文章中出现还不够，我们还需要知道关键词在文章中出现次数和出现的位置，

通常有两种位置：

a)字符位置，即记录该词是文章中第几个字符（优点是关键词亮显时定位快）；

b)关键词位置，即记录该词是文章中第几个关键词（优点是节约索引空间、词组（phase）查询快），lucene中记录的就是这种位置。

加上“出现频率”和“出现位置”信息后，我们的索引结构变为：

关键词 文章号 [出现频率] 出现位置

guangzhou 1 [2] 3，6

he 2 [1] 1

i 1 [1] 4

live 1 [2] 2，5

2 [1] 2

shanghai 2 [1] 3

tom 1 [1] 1

以live这行为例我们说明一下该结构：

live在文章1中出现了2次，文章2中出现了一次，它的出现位置为“2,5,2”这表示什么呢？

我们需要结合文章号和出现频率来分析，

文章1中出现了2次，那么“2,5”就表示live在文章1的关键词中出现的两个位置，

文章2中出现了1次，剩下的“2”就表示live是文章2的关键词中第2个关键字。

以上就是lucene索引结构中最核心的部分。

我们注意到关键字是按字符顺序排列的（lucene没有使用B树结构），

因此lucene可以用二元搜索算法(或叫二分查找/折半查找)快速定位关键词。

(3) 实现

实现时，lucene将上面三列分别作为:

词典文件（Term Dictionary）、频率文件(frequencies)、位置文件 (positions)保存。

其中词典文件不仅保存有每个关键词，还保留了指向频率文件和位置文件的指针，

通过指针可以找到该关键字的频率信息和位置信息。

Lucene中使用了field的概念，用于表达信息所在位置（如标题中，文章中，url中），

在建索引中，该field信息也记录在词典文件中，每个关键词都有一个field信息(因为每个关键字一定属于一个或多个field)。

(4) 压缩算法

为了减小索引文件的大小，Lucene对索引还使用了压缩技术。

首先，对词典文件中的关键词进行了压缩，关键词压缩为<前缀长度，后缀>，

例如：当前词为“阿拉伯语”，上一个词为“阿拉伯”，那么“阿拉伯语”压缩为<3，语>。

其次大量用到的是对数字的压缩，数字只保存与上一个值的差值

（这样可以减小数字的长度，进而减少保存该数字需要的字节数）。

例如当前文章号是16389（不压缩要用3个字节保存），上一文章号是16382，压缩后保存7（只用一个字节）。

<5>应用原因

下面我们可以通过对该索引的查询来解释一下为什么要建立索引。

假设要查询单词 “live”，lucene先对词典二元查找、找到该词，通过指向频率文件的指针读出所有文章号，然后返回结果。

词典通常非常小，因而，整个过程的时间是毫秒级的。

而用普通的顺序匹配算法，不建索引，而是对所有文章的内容进行字符串匹配，

这个过程将会相当缓慢，当文章数目很大时，时间往往是无法忍受的。

### 75 ZooKeeper分布式高可用？

ZooKeeper 运行期间，集群中至少有过半的机器保存了最新数据。集群超过半数的机器能够正常工作，集群就能够对外提供服务。

zookeeper可以选出N台机器作主机，它可以实现M:N的备份；keepalive只能选出1台机器作主机，所以keepalive只能实现M:1的备份。

通常有以下两种部署方案：双机房部署（一个稳定性更好、设备更可靠的机房，这个机房就是主要机房，而另外一个机房则更加廉价一些，例如，对于一个由 7 台机器组成的 ZooKeeper 集群，通常在主要机房中部署 4 台机器，剩下的 3 台机器部署到另外一个机房中）；三机房部署（无论哪个机房发生了故障，剩下两个机房的机器数量都超过半数。在三个机房中都部署若干个机器来组成一个 ZooKeeper 集群。假设机器总数为 N，各机房机器数：N1 = (N-1)/2 ，N2=1~(N-N1)/2 ，N3 = N - N1 - N2 ）。

水平扩容就是向集群中添加更多机器，Zookeeper2种方式（不完美），一种是集群整体重启，另外一种是逐台进行服务器的重启。

我们通常部署zookeeper集群来实现高可用性，那么zookeeper是如何实现高可用性的呢？

集群组成

要搭建一个高可用的 ZooKeeper 集群，我们首先需要确定好集群的规模。关于 ZooKeeper 集群的服务器组成，相信很多对 ZooKeeper 了解但是理解不够深入的读者，都存在或曾经存在过这样一个错误的认识：为了使得 ZooKeeper 集群能够顺利地选举出 Leader，必须将 ZooKeeper 集群的服务器数部署成奇数。这里我们需要澄清的一点是：任意台 ZooKeeper 服务器都能部署且能正常运行。

其实关于 ZooKeeper 集群服务器数，ZooKeeper 官方确实给出了关于奇数的建议，但绝大部分 ZooKeeper 用户对于这个建议认识有偏差。在本书前面提到的“过半存活即可用”特性中，我们已经了解了，一个 ZooKeeper 集群如果要对外提供可用的服务，那么集群中必须要有过半的机器正常工作并且彼此之间能够正常通信。基于这个特性，如果想搭建一个能够允许 N 台机器 down 掉的集群，那么就要部署一个由 2\*N+1 台服务器构成的 ZooKeeper 集群。因此，一个由 3 台机器构成的 ZooKeeper 集群，能够在挂掉 1 台机器后依然正常工作，而对于一个由 5 台服务器构成的 ZooKeeper 集群，能够对 2 台机器挂掉的情况进行容灾。注意，如果是一个由6台服务器构成的 ZooKeeper 集群，同样只能够挂掉 2 台机器，因为如果挂掉 3 台，剩下的机器就无法实现过半了。

因此，从上面的讲解中，我们其实可以看出，对于一个由 6 台机器构成的 ZooKeeper 集群来说，和一个由 5 台机器构成的 ZooKeeper 集群，其在容灾能力上并没有任何显著的优势，反而多占用了一个服务器资源。基于这个原因，ZooKeeper 集群通常设计部署成奇数台服务器即可。

容灾

所谓容灾，在 IT 行业通常是指我们的计算机信息系统具有的一种在遭受诸如火灾、地震、断电和其他基础网络设备故障等毁灭性灾难的时候，依然能够对外提供可用服务的能力。

对于一些普通的应用，为了达到容灾标准，通常我们会选择在多台机器上进行部署来组成一个集群，这样即使在集群的一台或是若干台机器出现故障的情况下，整个集群依然能够对外提供可用的服务。

而对于一些核心应用，不仅要通过使用多台机器构建集群的方式来提供服务，而且还要将集群中的机器部署在两个机房，这样的话，即使其中一个机房遭遇灾难，依然能够对外提供可用的服务。

上面讲到的都是应用层面的容灾模式，那么对于 ZooKeeper 这种底层组件来说，如何进行容灾呢？讲到这里，可能多少读者会有疑问，ZooKeeper 既然已经解决了单点问题，那为什么还要进行容灾呢？

单点问题

单点问题是分布式环境中最常见也是最经典的问题之一，在很多分布式系统中都会存在这样的单点问题。具体地说，单点问题是指在一个分布式系统中，如果某一个组件出现故障就会引起整个系统的可用性大大下降甚至是处于瘫痪状态，那么我们就认为该组件存在单点问题。

ZooKeeper 确实已经很好地解决了单点问题。我们已经了解到，基于“过半”设计原则，ZooKeeper 在运行期间，集群中至少有过半的机器保存了最新的数据。因此，只要集群中超过半数的机器还能够正常工作，整个集群就能够对外提供服务。

容灾

解决了单点问题，是不是该考虑容灾了呢？答案是肯定的，在搭建一个高可用的集群的时候依然需要考虑容灾问题。正如上面讲到的，如果集群中超过半数的机器还在正常工作，集群就能够对外提供正常的服务。那么，如果整个机房出现灾难性的事故，这时显然已经不是单点问题的范畴了。

在进行 ZooKeeper 的容灾方案设计过程中，我们要充分考虑到“过半原则”。也就是说，无论发生什么情况，我们必须保证 ZooKeeper 集群中有超过半数的机器能够正常工作。因此，通常有以下两种部署方案。

双机房部署

在进行容灾方案的设计时，我们通常是以机房为单位来考虑问题。在现实中，很多公司的机房规模并不大，因此双机房部署是个比较常见的方案。但是遗憾的是，在目前版本的 ZooKeeper 中，还没有办法能够在双机房条件下实现比较好的容灾效果——因为无论哪个机房发生异常情况，都有可能使得 ZooKeeper 集群中可用的机器无法超过半数。当然，在拥有两个机房的场景下，通常有一个机房是主要机房（一般而言，公司会花费更多的钱去租用一个稳定性更好、设备更可靠的机房，这个机房就是主要机房，而另外一个机房则更加廉价一些）。我们唯一能做的，就是尽量在主要机房部署更多的机器。例如，对于一个由 7 台机器组成的 ZooKeeper 集群，通常在主要机房中部署 4 台机器，剩下的 3 台机器部署到另外一个机房中。

三机房部署

既然在双机房部署模式下并不能实现好的容灾效果，那么对于有条件的公司，选择三机房部署无疑是个更好的选择，无论哪个机房发生了故障，剩下两个机房的机器数量都超过半数。假如我们有三个机房可以部署服务，并且这三个机房间的网络状况良好，那么就可以在三个机房中都部署若干个机器来组成一个 ZooKeeper 集群。

我们假定构成 ZooKeeper 集群的机器总数为 N，在三个机房中部署的 ZooKeeper 服务器数分别为 N1、N2 和 N3，如果要使该 ZooKeeper 集群具有较好的容灾能力，我们可以根据如下算法来计算 ZooKeeper 集群的机器部署方案。

1.）计算 N1

如果 ZooKeeper 集群的服务器总数是 N，那么：

N1 = (N-1)/2

在 Java 中，“/” 运算符会自动对计算结果向下取整操作。举个例子，如果 N=8，那么 N1=3；如果 N=7，那么 N1 也等于 3。

2.）计算 N2 的可选值

N2 的计算规则和 N1 非常类似，只是 N2的取值是在一个取值范围内：

N2 的取值范围是 1~(N-N1)/2

即如果 N=8，那么 N1=3，则 N2 的取值范围就是 1~2，分别是 1 和 2。注意，1 和 2 仅仅是 N2 的可选值，并非最终值——如果 N2 为某个可选值的时候，无法计算出 N3 的值，那么该可选值也无效。

3.） 计算 N3，同时确定 N2 的值

很显然，现在只剩下 N3 了，可以简单的认为 N3 的取值就是剩下的机器数，即：

N3 = N - N1 - N2

只是 N3 的取值必须满足 N3 < N1+N2。在满足这个条件的基础下，我们遍历步骤 2 中计算得到的 N2 的可选值，即可得到三机房部署时每个机房的服务器数量了。

现在我们以 7 台机器为例，来看看如何分配三机房的机器分布。根据算法的步骤 1，我们首先确定 N1 的取值为 3。根据算法的步骤 2，我们确定了 N2 的可选值为 1 和 2。最后根据步骤 3，我们遍历 N2 的可选值，即可得到两种部署方案，分别是 (3,1,3) 和 (3,2,2)。以下是 Java 程序代码对以上算法的一种简单实现：

public class Allocation {

static final int n = 7;

public static void main(String[] args){

int n1,n2,n3;

n1 = (n-1) / 2;

int n2\_max = (n-n1) / 2;

for(int i=1; i<=n2\_max; i++){

n2 = i;

n3 = n - n1 -n2;

if(n3 >= (n1+n2)){

continue;

}

System.out.println("("+n1+","+n2+","+n3+")");

}

}

}

水平扩容

水平可扩容可以说是对一个分布式系统在高可用性方面提出的基本的，也是非常重要的一个要求，通过水平扩容能够帮助系统在不进行或进行极少改进工作的前提下，快速提高系统对外的服务支撑能力。简单地讲，水平扩容就是向集群中添加更多的机器，以提高系统的服务质量。

很遗憾的是，ZooKeeper 在水平扩容扩容方面做得并不十分完美，需要进行整个集群的重启。通常有两种重启方式，一种是集群整体重启，另外一种是逐台进行服务器的重启。

（1）整体重启

所谓集群整体重启，就是先将整个集群停止，然后更新 ZooKeeper 的配置，然后再次启动。如果在你的系统中，ZooKeeper 并不是个非常核心的组件，并且能够允许短暂的服务停止（通常是几秒钟的时间间隔），那么不妨选择这种方式。在整体重启的过程中，所有该集群的客户端都无法连接上集群。等到集群再次启动，这些客户端就能够自动连接上——注意，整体启动前建立起的客户端会话，并不会因为此次整体重启而失效。也就是说，在整体重启期间花费的时间将不计入会话超时时间的计算中。

（2）逐台重启

这种方式更适合绝大多数的实际场景。在这种方式中，每次仅仅重启集群中的一台机器，然后逐台对整个集群中的机器进行重启操作。这种方式可以在重启期间依然保证集群对外的正常服务。

### 76 讲讲mybatis的连接池？

对于ORM框架而言，数据源的组织是一个非常重要的一部分，这直接影响到框架的性能问题。本文将通过对MyBatis框架的数据源结构进行详尽的分析，并且深入解析MyBatis的连接池。

本文首先会讲述MyBatis的数据源的分类，然后会介绍数据源是如何加载和使用的。紧接着将分类介绍UNPOOLED、POOLED和JNDI类型的数据源组织；期间我们会重点讲解POOLED类型的数据源和其实现的连接池原理。

以下是本章的组织结构：

一、MyBatis数据源DataSource分类

二、数据源DataSource的创建过程

三、 DataSource什么时候创建Connection对象

四、不使用连接池的UnpooledDataSource

五、为什么要使用连接池？

六、使用了连接池的PooledDataSource

一、MyBatis数据源DataSource分类

MyBatis数据源实现是在以下四个包中：

MyBatis把数据源DataSource分为三种：

 UNPOOLED 不使用连接池的数据源

 POOLED 使用连接池的数据源

 JNDI 使用JNDI实现的数据源

即：

相应地，MyBatis内部分别定义了实现了java.sql.DataSource接口的UnpooledDataSource，PooledDataSource类来表示UNPOOLED、POOLED类型的数据源。 如下图所示：

对于JNDI类型的数据源DataSource，则是通过JNDI上下文中取值。

二、数据源DataSource的创建过程

MyBatis 数据源 DataSource 对象的创建发生在 MyBatis 初始化的过程中。 下面让我们一步步地了解MyBatis是如何创建数据源DataSource的。

在mybatis的XML配置文件中，使用<dataSource>元素来配置数据源：

1. MyBatis在初始化时，解析此文件，根据<dataSource>的type属性来创建相应类型的的数据源DataSource，即：

type=”POOLED” ：MyBatis会创建PooledDataSource实例

type=”UNPOOLED” ：MyBatis会创建UnpooledDataSource实例

type=”JNDI” ：MyBatis会从JNDI服务上查找DataSource实例，然后返回使用

2. 顺便说一下，MyBatis是通过工厂模式来创建数据源DataSource对象的，MyBatis定义了抽象的工厂接口 : org.apache.ibatis.datasource.DataSourceFactory ,通过其getDataSource()方法返回数据源DataSource：

定义如下：

public interface DataSourceFactory {

void setProperties(Properties props);

//生产DataSource

DataSource getDataSource();

}

上述三种不同类型的type，则有对应的以下dataSource工厂：

POOLED PooledDataSourceFactory

UNPOOLED UnpooledDataSourceFactory

JNDI JndiDataSourceFactory

其类图如下所示：

3. MyBatis创建了DataSource实例后，会将其放到Configuration对象内的Environment对象中， 供以后使用。

三、 DataSource什么时候创建Connection对象

当我们需要创建SqlSession对象并需要执行SQL语句时，这时候MyBatis才会去调用dataSource对象来创建java.sql.Connection对象。也就是说，java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。

比如，我们有如下方法执行一个简单的SQL语句：

String resource = "mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession sqlSession = sqlSessionFactory.openSession();

sqlSession.selectList("SELECT \* FROM STUDENTS");

前4句都不会导致java.sql.Connection对象的创建，只有当第5句sqlSession.selectList("SELECT \* FROM STUDENTS ")，才会触发MyBatis在底层执行下面这个方法来创建java.sql.Connection对象：

protected void openConnection() throws SQLException {

if (log.isDebugEnabled()) {

log.debug("Opening JDBC Connection");

}

connection = dataSource.getConnection();

if (level != null) {

connection.setTransactionIsolation(level.getLevel());

}

setDesiredAutoCommit(autoCommmit);

}

而对于DataSource的UNPOOLED的类型的实现-UnpooledDataSource是怎样实现getConnection()方法的呢？请看下一节。

四、不使用连接池的UnpooledDataSource

当 <dataSource>的type属性被配置成了”UNPOOLED”，MyBatis首先会实例化一个UnpooledDataSourceFactory工厂实例，然后通过.getDataSource()方法返回一个UnpooledDataSource实例对象引用，我们假定为dataSource。

使用 UnpooledDataSource 的 getConnection(), 每调用一次就会产生一个新的 Connection 实例对象。

UnPooledDataSource的getConnection()方法实现如下：

/\*

UnpooledDataSource的getConnection()实现

\*/

public Connection getConnection() throws SQLException

{

return doGetConnection(username, password);

}

private Connection doGetConnection(String username, String password) throws SQLException

{

//封装username和password成properties

Properties props = new Properties();

if (driverProperties != null)

{

props.putAll(driverProperties);

}

if (username != null)

{

props.setProperty("user", username);

}

if (password != null)

{

props.setProperty("password", password);

}

return doGetConnection(props);

}

/\*

\* 获取数据连接

\*/

private Connection doGetConnection(Properties properties) throws SQLException

{

//1.初始化驱动

initializeDriver();

//2.从DriverManager中获取连接，获取新的Connection对象

Connection connection = DriverManager.getConnection(url, properties);

//3.配置connection属性

configureConnection(connection);

return connection;

}

如上代码所示，UnpooledDataSource会做以下事情：

1. 初始化驱动： 判断driver驱动是否已经加载到内存中，如果还没有加载，则会动态地加载driver类，并实例化一个Driver对象，使用DriverManager.registerDriver()方法将其注册到内存中，以供后续使用。

2. 创建Connection对象： 使用DriverManager.getConnection()方法创建连接。

3. 配置Connection对象： 设置是否自动提交autoCommit和隔离级别isolationLevel。

4. 返回Connection对象。

上述的序列图如下所示：

总结：从上述的代码中可以看到， 我们每调用一次 getConnection() 方法，都会通过 DriverManager.getConnection() 返回新的 java.sql.Connection 实例。

五、为什么要使用连接池？

1. 创建一个java.sql.Connection实例对象的代价

首先让我们来看一下创建一个java.sql.Connection对象的资源消耗。我们通过连接Oracle数据库，创建创建Connection对象，来看创建一个Connection对象、执行SQL语句各消耗多长时间。代码如下：

public static void main(String[] args) throws Exception

{

String sql = "select \* from hr.employees where employee\_id < ? and employee\_id >= ?";

PreparedStatement st = null;

ResultSet rs = null;

long beforeTimeOffset = -1L; //创建Connection对象前时间

long afterTimeOffset = -1L; //创建Connection对象后时间

long executeTimeOffset = -1L; //创建Connection对象后时间

Connection con = null;

Class.forName("oracle.jdbc.driver.OracleDriver");

beforeTimeOffset = new Date().getTime();

System.out.println("before:\t" + beforeTimeOffset);

con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");

afterTimeOffset = new Date().getTime();

System.out.println("after:\t\t" + afterTimeOffset);

System.out.println("Create Costs:\t\t" + (afterTimeOffset - beforeTimeOffset) + " ms");

st = con.prepareStatement(sql);

//设置参数

st.setInt(1, 101);

st.setInt(2, 0);

//查询，得出结果集

rs = st.executeQuery();

executeTimeOffset = new Date().getTime();

System.out.println("Exec Costs:\t\t" + (executeTimeOffset - afterTimeOffset) + " ms");

}

上述程序在我笔记本上的执行结果为：

从此结果可以清楚地看出，创建一个Connection对象，用了 250 毫秒 ；而执行SQL的时间用了 170毫秒 。

创建一个Connection对象用了250毫秒！这个时间对计算机来说可以说是一个 非常奢侈 的！

这仅仅是一个Connection对象就有这么大的代价，设想一下另外一种情况：如果我们在Web应用程序中，为用户的每一个请求就操作一次数据库，当有10000个在线用户并发操作的话，对计算机而言，仅仅创建Connection对象不包括做业务的时间就要损耗10000×250ms= 250 0000 ms = 2500 s = 41.6667 min,竟然要 41 分钟！！！如果对高用户群体使用这样的系统，简直就是开玩笑！

2. 问题分析：

创建一个java.sql.Connection对象的代价是如此巨大，是因为创建一个Connection对象的过程，在底层就相当于和数据库建立的通信连接，在建立通信连接的过程，消耗了这么多的时间，而往往我们建立连接后（即创建Connection对象后），就执行一个简单的SQL语句，然后就要抛弃掉，这是一个非常大的资源浪费！

3.解决方案:

对于需要频繁地跟数据库交互的应用程序，可以在创建了Connection对象，并操作完数据库后，可以不释放掉资源，而是将它放到内存中，当下次需要操作数据库时，可以直接从内存中取出Connection对象，不需要再创建了，这样就极大地节省了创建Connection对象的资源消耗。由于内存也是有限和宝贵的，这又对我们对内存中的Connection对象怎么有效地维护提出了很高的要求。我们将在内存中存放Connection对象的容器称之为 连接池（Connection Pool）。下面让我们来看一下MyBatis的线程池是怎样实现的。

六、使用了连接池的PooledDataSource

同样地，我们也是使用PooledDataSource的getConnection()方法来返回Connection对象。现在让我们看一下它的基本原理：

PooledDataSource将java.sql.Connection对象包裹成PooledConnection对象放到了PoolState类型的容器中维护。 MyBatis将连接池中的PooledConnection分为两种状态： 空闲状态（idle）和活动状态(active)，这两种状态的PooledConnection对象分别被存储到PoolState容器内的 idleConnections 和 activeConnections 两个List集合中：

idleConnections :空闲(idle)状态PooledConnection对象被放置到此集合中，表示当前闲置的没有被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从此集合中取PooledConnection对象。当用完一个java.sql.Connection对象时，MyBatis会将其包裹成PooledConnection对象放到此集合中。

activeConnections :活动(active)状态的PooledConnection对象被放置到名为activeConnections的ArrayList中，表示当前正在被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从idleConnections集合中取PooledConnection对象,如果没有，则看此集合是否已满，如果未满，PooledDataSource会创建出一个PooledConnection，添加到此集合中，并返回。

PoolState连接池的大致结构如下所示：

6.1 获取java.sql.Connection对象的过程

下面让我们看一下PooledDataSource 的getConnection()方法获取Connection对象的实现：

public Connection getConnection() throws SQLException {

return popConnection(dataSource.getUsername(), dataSource.getPassword()).getProxyConnection();

}

public Connection getConnection(String username, String password) throws SQLException {

return popConnection(username, password).getProxyConnection();

}

上述的popConnection()方法，会从连接池中返回一个可用的PooledConnection对象，然后再调用getProxyConnection()方法最终返回Conection对象。（至于为什么会有getProxyConnection(),请关注下一节）

现在让我们看一下popConnection()方法到底做了什么：

1. 先看是否有空闲(idle)状态下的PooledConnection对象，如果有，就直接返回一个可用的PooledConnection对象；否则进行第2步。

2. 查看活动状态的PooledConnection池activeConnections是否已满；如果没有满，则创建一个新的PooledConnection对象，然后放到activeConnections池中，然后返回此PooledConnection对象；否则进行第三步；

3. 看最先进入activeConnections池中的PooledConnection对象是否已经过期：如果已经过期，从activeConnections池中移除此对象，然后创建一个新的PooledConnection对象，添加到activeConnections中，然后将此对象返回；否则进行第4步。

4. 线程等待，循环2步

/\*

\* 传递一个用户名和密码，从连接池中返回可用的PooledConnection

\*/

private PooledConnection popConnection(String username, String password) throws SQLException

{

boolean countedWait = false;

PooledConnection conn = null;

long t = System.currentTimeMillis();

int localBadConnectionCount = 0;

while (conn == null)

{

synchronized (state)

{

if (state.idleConnections.size() > 0)

{

// 连接池中有空闲连接，取出第一个

conn = state.idleConnections.remove(0);

if (log.isDebugEnabled())

{

log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");

}

}

else

{

// 连接池中没有空闲连接，则取当前正在使用的连接数小于最大限定值，

if (state.activeConnections.size() < poolMaximumActiveConnections)

{

// 创建一个新的connection对象

conn = new PooledConnection(dataSource.getConnection(), this);

@SuppressWarnings("unused")

//used in logging, if enabled

Connection realConn = conn.getRealConnection();

if (log.isDebugEnabled())

{

log.debug("Created connection " + conn.getRealHashCode() + ".");

}

}

else

{

// Cannot create new connection 当活动连接池已满，不能创建时，取出活动连接池的第一个，即最先进入连接池的PooledConnection对象

// 计算它的校验时间，如果校验时间大于连接池规定的最大校验时间，则认为它已经过期了，利用这个PoolConnection内部的realConnection重新生成一个PooledConnection

//

PooledConnection oldestActiveConnection = state.activeConnections.get(0);

long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();

if (longestCheckoutTime > poolMaximumCheckoutTime)

{

// Can claim overdue connection

state.claimedOverdueConnectionCount++;

state.accumulatedCheckoutTimeOfOverdueConnections += longestCheckoutTime;

state.accumulatedCheckoutTime += longestCheckoutTime;

state.activeConnections.remove(oldestActiveConnection);

if (!oldestActiveConnection.getRealConnection().getAutoCommit())

{

oldestActiveConnection.getRealConnection().rollback();

}

conn = new PooledConnection(oldestActiveConnection.getRealConnection(), this);

oldestActiveConnection.invalidate();

if (log.isDebugEnabled())

{

log.debug("Claimed overdue connection " + conn.getRealHashCode() + ".");

}

}

else

{

//如果不能释放，则必须等待有

// Must wait

try

{

if (!countedWait)

{

state.hadToWaitCount++;

countedWait = true;

}

if (log.isDebugEnabled())

{

log.debug("Waiting as long as " + poolTimeToWait + " milliseconds for connection.");

}

long wt = System.currentTimeMillis();

state.wait(poolTimeToWait);

state.accumulatedWaitTime += System.currentTimeMillis() - wt;

}

catch (InterruptedException e)

{

break;

}

}

}

}

//如果获取PooledConnection成功，则更新其信息

if (conn != null)

{

if (conn.isValid())

{

if (!conn.getRealConnection().getAutoCommit())

{

conn.getRealConnection().rollback();

}

conn.setConnectionTypeCode(assembleConnectionTypeCode(dataSource.getUrl(), username, password));

conn.setCheckoutTimestamp(System.currentTimeMillis());

conn.setLastUsedTimestamp(System.currentTimeMillis());

state.activeConnections.add(conn);

state.requestCount++;

state.accumulatedRequestTime += System.currentTimeMillis() - t;

}

else

{

if (log.isDebugEnabled())

{

log.debug("A bad connection (" + conn.getRealHashCode() + ") was returned from the pool, getting another connection.");

}

state.badConnectionCount++;

localBadConnectionCount++;

conn = null;

if (localBadConnectionCount > (poolMaximumIdleConnections + 3))

{

if (log.isDebugEnabled())

{

log.debug("PooledDataSource: Could not get a good connection to the database.");

}

throw new SQLException("PooledDataSource: Could not get a good connection to the database.");

}

}

}

}

}

if (conn == null)

{

if (log.isDebugEnabled())

{

log.debug("PooledDataSource: Unknown severe error condition. The connection pool returned a null connection.");

}

throw new SQLException("PooledDataSource: Unknown severe error condition. The connection pool returned a null connection.");

}

return conn;

}

对应的处理流程图如下所示：

如上所示,对于PooledDataSource的getConnection()方法内，先是调用类PooledDataSource的popConnection()方法返回了一个PooledConnection对象，然后调用了PooledConnection的getProxyConnection()来返回Connection对象。

6.2java.sql.Connection对象的回收

当我们的程序中使用完Connection对象时，如果不使用数据库连接池，我们一般会调用 connection.close()方法，关闭connection连接，释放资源。如下所示：

private void test() throws ClassNotFoundException, SQLException

{

String sql = "select \* from hr.employees where employee\_id < ? and employee\_id >= ?";

PreparedStatement st = null;

ResultSet rs = null;

Connection con = null;

Class.forName("oracle.jdbc.driver.OracleDriver");

try

{

con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");

st = con.prepareStatement(sql);

//设置参数

st.setInt(1, 101);

st.setInt(2, 0);

//查询，得出结果集

rs = st.executeQuery();

//取数据，省略

//关闭，释放资源

con.close();

}

catch (SQLException e)

{

con.close();

e.printStackTrace();

}

}

调用过close()方法的Connection对象所持有的资源会被全部释放掉，Connection对象也就不能再使用。

那么，如果我们使用了连接池，我们在用完了Connection对象时，需要将它放在连接池中，该怎样做呢？

可能大家第一个在脑海里闪现出来的想法就是：我在应该调用con.close()方法的时候，不调用close()f方法，将其换成将Connection对象放到连接池容器中的代码！

好，我们将上述的想法实现，首先定义一个简易连接池Pool，然后将上面的代码改写：

package com.foo.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.util.Vector;

/\*\*

\*

\* 一个线程安全的简易连接池实现，此连接池是单例的

\* putConnection()将Connection添加到连接池中

\* getConnection()返回一个Connection对象

\*/

public class Pool {

private static Vector<Connection> pool = new Vector<Connection>();

private static int MAX\_CONNECTION =100;

private static String DRIVER="oracle.jdbc.driver.OracleDriver";

private static String URL = "jdbc:oracle:thin:@127.0.0.1:1521:xe";

private static String USERNAME = "louluan";

private static String PASSWROD = "123456";

static {

try {

Class.forName(DRIVER);

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

/\*\*

\* 将一个Connection对象放置到连接池中

\*/

public static void putConnection(Connection connection){

synchronized(pool)

{

if(pool.size()<MAX\_CONNECTION)

{

pool.add(connection);

}

}

}

/\*\*

\* 返回一个Connection对象，如果连接池内有元素，则pop出第一个元素；

\* 如果连接池Pool中没有元素，则创建一个connection对象，然后添加到pool中

\* @return Connection

\*/

public static Connection getConnection(){

Connection connection = null;

synchronized(pool)

{

if(pool.size()>0)

{

connection = pool.get(0);

pool.remove(0);

}

else

{

connection = createConnection();

pool.add(connection);

}

}

return connection;

}

/\*\*

\* 创建一个新的Connection对象

\*/

private static Connection createConnection()

{

Connection connection = null;

try {

connection = DriverManager.getConnection(URL, USERNAME,PASSWROD);

} catch (SQLException e) {

e.printStackTrace();

}

return connection;

}

}

package com.foo.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.Vector;

public class PoolTest

{

private void test() throws ClassNotFoundException, SQLException

{

String sql = "select \* from hr.employees where employee\_id < ? and employee\_id >= ?";

PreparedStatement st = null;

ResultSet rs = null;

Connection con = null;

Class.forName("oracle.jdbc.driver.OracleDriver");

try

{

con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");

st = con.prepareStatement(sql);

//设置参数

st.setInt(1, 101);

st.setInt(2, 0);

//查询，得出结果集

rs = st.executeQuery();

//取数据，省略

//将不再使用的Connection对象放到连接池中，供以后使用

Pool.putConnection(con);

}

catch (SQLException e)

{

e.printStackTrace();

}

}

}

上述的代码就是将我们使用过的Connection对象放到Pool连接池中，我们需要Connection对象的话，只需要使用Pool.getConnection()方法从里面取即可。

是的,上述的代码完全可以实现此能力，不过有一个很不优雅的实现： 就是我们需要手动地将Connection对象放到Pool连接池中，这是一个很傻的实现方式。这也和一般使用Connection对象的方式不一样：一般使用Connection的方式是使用完后，然后调用.close()方法释放资源。

为了和一般的使用Conneciton对象的方式保持一致，我们希望当Connection使用完后，调用.close()方法，而实际上Connection资源并没有被释放，而实际上被添加到了连接池中。这样可以做到吗？答案是可以。上述的要求从另外一个角度来描述就是：能否提供一种机制，让我们知道Connection对象调用了什么方法，从而根据不同的方法自定义相应的处理机制。恰好代理机制就可以完成上述要求.

怎样实现Connection对象调用了close()方法，而实际是将其添加到连接池中

这是要使用代理模式，为真正的Connection对象创建一个代理对象，代理对象所有的方法都是调用相应的真正Connection对象的方法实现。当代理对象执行close()方法时，要特殊处理，不调用真正Connection对象的close()方法，而是将Connection对象添加到连接池中。

MyBatis的PooledDataSource的PoolState内部维护的对象是PooledConnection类型的对象，而PooledConnection则是对真正的数据库连接java.sql.Connection实例对象的包裹器。

PooledConnection对象内持有一个真正的数据库连接java.sql.Connection实例对象和一个java.sql.Connection的代理：

其部分定义如下：

class PooledConnection implements InvocationHandler {

//......

//所创建它的datasource引用

private PooledDataSource dataSource;

//真正的Connection对象

private Connection realConnection;

//代理自己的代理Connection

private Connection proxyConnection;

//......

}

PooledConenction 实现了 InvocationHandler 接口，并且， proxyConnection 对象也是根据这个它来生成的代理对象：

public PooledConnection(Connection connection, PooledDataSource dataSource) {

this.hashCode = connection.hashCode();

this.realConnection = connection;

this.dataSource = dataSource;

this.createdTimestamp = System.currentTimeMillis();

this.lastUsedTimestamp = System.currentTimeMillis();

this.valid = true;

this.proxyConnection = (Connection) Proxy.newProxyInstance(Connection.class.getClassLoader(), IFACES, this);

}

实际上，我们调用PooledDataSource的getConnection()方法返回的就是这个proxyConnection对象。

当我们调用此proxyConnection对象上的任何方法时，都会调用PooledConnection对象内invoke()方法。

让我们看一下PooledConnection类中的invoke()方法定义：

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

String methodName = method.getName();

//当调用关闭的时候，回收此Connection到PooledDataSource中

if (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) {

dataSource.pushConnection(this);

return null;

} else {

try {

if (!Object.class.equals(method.getDeclaringClass())) {

checkConnection();

}

return method.invoke(realConnection, args);

} catch (Throwable t) {

throw ExceptionUtil.unwrapThrowable(t);

}

}

}

从上述代码可以看到，当我们使用了pooledDataSource.getConnection()返回的Connection对象的close()方法时，不会调用真正Connection的close()方法，而是将此Connection对象放到连接池中。

七、JNDI类型的数据源DataSource

对于JNDI类型的数据源DataSource的获取就比较简单，MyBatis定义了一个JndiDataSourceFactory工厂来创建通过JNDI形式生成的DataSource。

下面让我们看一下JndiDataSourceFactory的关键代码：

if (properties.containsKey(INITIAL\_CONTEXT)

&& properties.containsKey(DATA\_SOURCE))

{

//从JNDI上下文中找到DataSource并返回

Context ctx = (Context) initCtx.lookup(properties.getProperty(INITIAL\_CONTEXT));

dataSource = (DataSource) ctx.lookup(properties.getProperty(DATA\_SOURCE));

}

else if (properties.containsKey(DATA\_SOURCE))

{

// //从JNDI上下文中找到DataSource并返回

dataSource = (DataSource) initCtx.lookup(properties.getProperty(DATA\_SOURCE));

}

### 77 spring中beanFactory和ApplicationContext的联系和区别？

ApplicationContext 是 BeanFactory接口的子接口

其实两个在代码看来就是在获取配置文件的时候 的差异，他们还有其他的差异：

1）BeanFactory 采用的是延迟加载，第一次getBean的时候才会初始化Bean

2）ApplicationContext是对BeanFactory的扩展，提供了更多的功能

国际化处理

事件传递

Bean自动装配

各种不同应用层的Context实现

结论：开发中尽量使用ApplicationContext 就可以了

## 七、项目管理

### 1 讲一下你们整个实施过程包括哪些？哪些环节风险比较大的？

需求分析，概要设计，编码，调试，测试，预发布，上线

风险比较大：需求分析，概要设计

### 2 在团队建设上面有什么手段？

团队建设的三个层面

1、团队的凝聚力。

团队的凝聚力是针对团队和成员之间的关系而言的。团队精神表现为团队成员强烈的归属感和一体性，每个团队成员都能感受到自己是团队当中的一分子，把个人工作和团队目标联系在一起，对团队忠诚，对团队的成功感到自豪，对团队的困境感到忧虑。

2、团队的合作意识。

团队的合作意识是指团队和团队成员表现为协作和共为一体的特点。团队成员间相互依存、同舟共济、互相敬重、彼此宽容和尊重个性的差异；彼此间形成一种信任的关系，待人真诚、遵守承诺；相互帮助和共同提高；共享利益和成就、共担责任。

3、团队士气。

团队士气是团队精神的一个重要方面。拿破仑曾说过：“一支军队的实力四分之三靠的是士气”。将这句话的含义延伸到现代企业管理，为团队目标而奋斗的精神状态对团队的业绩非常重要。

### 3 作为项目经理应该具备哪些能力和素质？

1.1 能适应变化

1.2 有实践能力

1.3 能进行创造性的思考

1.4 乐于学习

1.5 有带团队的能力

1.6 具有大局观与组织能力

1.7 适应各类企业的文化与价值观

1.8 善于与人沟通

1.9 善于描绘愿景

### 4 在绩效考核上如何做使下属都信服？主要指标有哪些？

（1）主要方法：

自评，相互评，兄弟部门评，面谈，找出客观事实说出突出点和不足的地方

（2）主要指标：

完成本职工作，责任心，协作能力，积极主动，态度，有新的技术突破，加班情况

（3）公平才能信服：

a，绩效考核制度是不区分职位高低和亲疏远近的，对于所有需要参与考核的部门和考核人都应该有效。

b，绩效考核制度的制定是取决于部门、岗位职责的。并不是所有的工作都会有明确的衡量指标，所以就要求绩效考核的方法存在一定程度上的差异，以适用于不同的方位和指标。

c，绩效考核的指标必须是明确且可以衡量的，需要符合被考核人的岗位且在正常的工作时间能够完成的。

d，绩效考核必须要有明确的截止时间，并且需要验证的遵守，给出合理的评价。

### 5怎么着手制定项目计划？

在适当的活动和阶段或其他的概括的标准说明下，输入确定的任务。将适当的可交付产品及里程碑和特定的任务联系起来。连接全部需要依赖关联的任务。把资源角色或资源名字加到每个任务上。应用度量结果确定事先的任务工作量，把更多的时间用于需求收集，设计和测试。考虑所有已知的节假日，培训，休假或其他的资源停工时间。计划草案将同支持团体，管理层和商务用户一起复查，做为补充性的输入和最终的批准。

### 6你认为项目中最重要的是哪些过程？

分析、设计阶段、测试阶段

### 7如果给你一个4-6个人的Team，那么你怎么分配管理他们？

挑选一个技术过硬的人作为我的替补和项目的轻骑兵，是的团队中必须有机动人员，否则你的项目十有八九会夭折。其他的人会被平均的分配任务。

我们会在每周进行全面的任务分配，每个人获取一周的大概工作，然后每天的工作由他自己完成并汇报。掌握每个人都长处和短板，充分发挥他们的价值。

### 8你认为你应聘我们公司的项目经理，你自身的优势在哪？

带领团队经验丰富，技术能力扎实，工作积极主动，责任心强

### 9你的管理的思路有哪些？

（1）注重成果，管理重在追求或取得成果。检验管理的一个原则是：是否达到了目标，是否完成了任务。当然，这个原则并不是在所有情况下都适用，管理者应该把精力和注意力放在“行得通”的事情上。

（2）把握整体，管理者之所以成为管理者，是因为他们眼观全局，着眼于整体，把整体发展视为己任。管理者应该理解自己的任务，不应从自己的职位出发，而应着眼于如何运用源于职位的知识、能力和经验来为整体效力。

（3）专注要点，专注要点的关键在于专注少数真正重要的东西。许多管理者热衷于寻找所谓的“秘方”，其实这是一种冒险行为。倘若真的有什么“秘方”，那就是专注要点应该是最重要的。要具备专注要点的能力、技巧和纪律性，是效率高的典型表现。

（4）利用优点，利用优点是指利用现有的优点，而不是那些需要重新建立和开发的优点。但现实中，很多管理者总是致力于与之相反的方面，即开发新的优点，而不是发挥现有的优点。如果这样，即使管理方法很有技巧，看上去也很科学，但造成的管理失误却是无法弥补的。

（5）相互信任，怎样在自己的部门或组织内部创造和谐、完美的工作氛围呢？有些管理者一板一眼地按照教科书上说的来做，但效果却不是很好。其实，只要管理者能够赢得周围其他人的信任，那么他所管理的部门或组织的工作气氛就会是和谐的。

（6）正面思维，正面思维的关键在于运用正确的或创造性的方式思考。正面思维的原则能让管理者把注意力放在机会上。事实上，发现和抓住机会要比解决问题更重要，但这并不是说管理者可以忽视存在的问题。有效率的管理者能够清楚地看到问题和困难，并不加以回避，而是先去寻找可能的办法和机会。

### 10 项目中代码质量如何控制的？

1. 进行相关的培训，邀请有经验的人对新员工或者junior的开发人员进行培训，以提高代码质量。

2. 制定相关的coding standard和coding pattern, 要求开发人员严格follow。

3. 使用相关的工具对代码进行静态分析和动态分析，及早发现代码中的问题。

4. 严格执行代码审查，每次代码提交前都要求进行review, 具体可以有CCR(持续代码审查) 和 FCR(正式代码审查)两种code review

### 11 项目管理的五大过程组与九大领域是什么？

5个过程组指的是：启动、规划、执行、监控、收尾

9大知识领域指的是：整体、范围、进度、成本、质量、人力资源、沟通、风险、采购

### 12 工作任务非常多非常杂时如何处理？

首先把所有的工作都列出来，免得被遗漏。

其次，把所有工作按时间的紧迫和任务的轻重进行分类，先做比较急而且比较重要的，

再做不那么急又不那么重要的。把相同类别的任务放在一起做。一件一件地做，做一件就画一个对勾，你会很有成就感，工作效率也会很快的。

### 13 项目出现延迟如何处理？

很多企业的项目管理人员遇到项目拖期的情形，首先采取的策略是加班，管理水平稍好一点的企业则会修正计划。加班，可能使拖期现象暂时得到缓解，但是产生拖期的根本原因依然存在，一段时间之后，这些问题会继续暴露出来。

所以大多数情况下，如果不采取针对性的措施，而是单纯用加班赶进度的策略解决不了问题。管理人员应该慎用加班这个策略。

改计划，起码实事求是地承认了问题，但是对解决问题于事无补。项目经理应该是走向成功之路的引领者，而不是分析员、记录员这样被动的角色。

其实无论加班还是改计划，都没有解决问题。项目管理者发现项目拖期，应当分析问题的真正原因，然后对症下药。该返工的返工，该培训的培训，该招人的时候招人，甚至必要的情况下终止这个项目，以免更大的损失。

项目管理者的真正挑战，不是发现问题和记录问题，而是预见问题、控制问题和解决问题。

### 14 和同事的设计思路不一样怎么处理？

先认真倾听对方的想法，衡量下两边的优缺点，结合当时的业务场景选择最适合的方案

### 15 如何保证开发质量？

1. Java代码规范学习

2. CodeReview，CodeReview是在开发阶段发现代码缺陷的一种方式，可以有效提高代码质量，降低维护成本；是团队成员互相学习、分享编程经验的提高过程。

3. 重构，在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构。

4. 工具查bug：findbugs（idea插件）

### 16 团队的规划是什么？

1树立核心形象与威信，把你的工作经验传授给你的手下，尤其是那些业务新手。在工作中承担更多的责任，有利于你树立威信。

2、创造一个良好的沟通环境，对于沟通的力量，是不容置疑的。有意见、有矛盾，不说出来会积怨；出现问题相互推诿，可能出现更大的问题，这些都是沟通不够的表现。

3合理分工各尽其才，在营销行业里流行着这么一句话：只有优秀的团队，没有优秀的个人。而我的理解是：优秀的团队里，每个一人都优秀。

### 17 能介绍下从工作到现在自己的成长在那里？

团队管理能力，技术能力，沟通能力，协调能力

### 18 [敏捷开发的价值是什么?](http://www.jobui.com/mianshiti/it/xiangmuguanli/4005/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

个人和团队之间的协作比流程和工具更重要。可以交付使用的系统产品比繁琐的文档更重要跟客户的协作比合同上的斤斤计较更重要快速响应需求变更而不是一味的严格执行计划敏捷开发不代表没有流程，没有文档或者没有计划，这些东西都很重要，但是不如团队之间和团队和客户之间的协作更重要

### 19 [你们公司都采取了什么措施来提高代码质量?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3950/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

1. 进行相关的培训，邀请有经验的人对新员工或者junior的开发人员进行培训，以提高代码质量。

2. 制定相关的coding standard和coding pattern, 要求开发人员严格follow。

3. 使用相关的工具对代码进行静态分析和动态分析，及早发现代码中的问题。

4. 严格执行代码审查，每次代码提交前都要求进行review, 具体可以有CCR(持续代码审查) 和 FCR(正式代码审查)两种code review

### 20 [软件项目团队的特征有哪些?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3811/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

–是一个临时性的团队

–是跨职能的

–在软件项目不同阶段中团队成员具有不稳定性

–成员具有极大的流动性

–年轻化程度高

–软件项目团队属于高度集中的知识型团队

–员工业绩难以量化考核

–软件项目团队非常注重自我

高效的软件开发团队是建立在合理的开发流程及团队成员密切合作的基础之上，团队成员需共同迎接挑战、有效的计划、协调和管理各自的工作直至成功完成项目目标。

### 21 [常见的软件风险管理模式都有哪些?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3960/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

（1）危机管理-这种模式类似于救火模式，其特点是听任软件风险的发生，及至软件风险给软件项目开发造成麻烦后才着手进行处理。

（2）失败处理-在这种模式中，项目组人员和负责人察觉到了潜在的风险，但听任软件风险的发生和演化，只是在风险发生之后才采取应对措施。

（3）风险缓解-在风险缓解模式中，项目组人员和负责人在软件开发过程中有意识地识别各种软件风险，并且针对这些软件风险事先制定好风险发生后的补救措施，但是不做任何防范措施。

（4）风险预防-风险预防模式将风险识别和风险防范作为软件项目的一部分加以规划和执行。

（5）消灭根源-在该模式中，项目组人员和负责人不仅要识别软件开发过程中各种潜在的软件风险，而且还要分析导致这些软件风险发生的主要因素，并采取积极的措施消除软件风险产生的根源。

### 22 [项目管理过程中经常出现过哪些conflicts?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3843/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

1. 进度（Schedule）在项目任务的时间安排、先后顺序及安排方面存在不一致的意见

2. 项目优先级（Priorities）项目参与者在活动和任务的优先级上观点不同

3. 资源冲突（Resource）项目团队成员安排与其他领域人员安排方面的冲突

4. 技术意见与执行情况的冲突。在技术问题，执行规范和技术权衡上的不一致

5. 管理程序（Administration Procedures）在项目如何管理问题上发生的管理导向和行政导向的冲突

6. 成本（Cost）在设计工作分解结构上，来自支持部门的成本估算上的冲突

7. 个性(Personality)人际关系方面的冲突

### 23 你认为什么样的团队是高效团队，什么样的团队是糟糕团队?

高效团队具有如下特征：

1. 团队成员对团队遵守承诺和高度忠诚

2. 团队成员对其他成员及项目经理遵守承诺

3. 团队成员具备各种不同的专业技能及专业素养。

4. 团队成员具有较高的工作满意度

5. 团队成员之间不断加强和改善沟通

6. 团队成员对隶属管理和目标的感知

7. 具有较高的团队精神及团队士气

8. 具有合理的冲突化解机制

糟糕团队的特征：

1. 团队缺乏动力或者项目组成员抱着“事不关己、高高挂起”的态度

2. 挫折感，团队成员普遍对项目工作不满意，对所从事的工作充满挫折感

3. 冲突和不良竞争，项目组成员之间经常发生冲突和不良竞争

4. 没有效率的会议，如状态会议变成牢骚会议

5. 沟通太差。团队成员之间不交流项目信息，不能很好的配合工作

6. 对项目经理缺乏尊重和信任。

### 24 如果你明天开始正式接任这个职位，你希望能够获取哪方面的信息呢?

1.岗位相关的部门内部组织架构关系以及与其他部门接口人信息。

2.所负责产品的以前，现在和将来。

3.岗位的绩效考核状况和公司级别的制度

### 25 项目管理部门在项目开发中的质量管理主要包括哪些内容？

(1)决策阶段的质量管理

主要内容是在广泛搜集资料、调查研究的基础上研究、分析、比较，决定项目的可行性和最佳方案。

(2)项目实施前的质量管理

①对项目组的能力重新审查，包括各个成员资质的审查。如果发现实际情况有所变化，必须采取有效措施予以纠正。

②对所有的合同和技术文件、报告进行详细的审阅。如图纸是否完备，有无错漏空缺，各个设计文件之间有无矛盾之处，技术标准是否齐全，等等。

③审阅进度计划和实施方案。

(3)项目实施中的质量管理

①参与项目的阶段性评审。该评审从保证评审过程有效性方面入手，如参与评审的人是否具备一定资格，是否规定的人员都参与了评审，是否对评审对象每个部分都进行了评审，是否给出了明确的结论等。

⑤跟踪问题的解决情况。在项目组内一可以解决的问题就在项目组内部解决，对于在项目组内部无法解决的一问题，或是在项目组中催多次也没有得到解决的问题，可以利用其独立汇报的渠道报告给高层经理。

⑥收集新方法，提供过程改进的依据，便于下一步对规程进行修改和完善。

(4)项目完成后的质量管理

①监督检查项目测试情况。

④进行项目实施后审计。

⑤总结项目实施的经验和教训。

### 26 [成功的团队有哪些特点?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3800/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

（1）明确的目标。团队的每个成员可以有不同的目的、不同的个性，但作为一个整体，必须有共同的奋斗目标。

　　（2）清晰的角色。有效团队的成员必须在清楚的组织架构中有清晰的角色定位和分工，团队成员应清楚了解自己的定位与责任。

　　（3）相互的技能。团队成员要具备为实现共同目标的基本技能，并能够有良好的合作。

　　（4）相互间信任。相互信任是一个成功团队最显著的特征。

　　（5）良好的沟通。团队成员间拥有畅通的信息交流，才会使成员的情感得到交流，才能协调成员的行为，使团队形成凝聚力和战斗力。

（6）合适的领导。团队的领导往往起到教练或后盾作用，他们对团队提供指导和支持，而不是企图控制下属。

### 27 在你以前的项目管理工作中，你碰到最大的困难是什么？

(1) 自身知识包括专业和管理的不足，这个需要利用各种时间进行充电。

(2) 各方不配合，相互推诿，特别是甲方内部，各种关系错综复杂包括甲乙不分。

(3) 工程进度不好控制，无论天气原因还是人为因素，很少有工期提前结束的，虽然工期早就做出预留。大会小会都要做好挨批的准备。

(4) 后期验收调试责任不明确，都想推脱责任。

### 28 [软件开发面临的问题都有哪些?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3895/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

1) 软件开发是高风险、高投入的项目

2) 开发时间长、成本高

3) 无法证明正确性

4) 维护代价高

5) 开发、维护难于度量 等等

6) 偏激看法：凡是软件开发项目就不可能按时完成。

### 29 [软件项目失败的主要原因有哪些?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3916/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

1) 项目经理实战经验不足

项目经理都是从企业高级软件工程师和高级构架师提升来的项目经理职位，因此不可避免的造就了这样的项目经理过分集中于技术上，从而忽视用户的角度去分析。以技术为主的应该将重心放在IT项目交付上，以商业为主的要满足用户的商业目标的需求。

2) 项目计划不够合理

项目计划过程中存在不合理假设和风险预测不足导致项目失败的主要因素，在计划过程中必须留时间处理不可预知的问题，因此在任何项目过程中各种无法预知的都应该预留10%的时间来应对。

3) 需求不确定性

需求分析和开发过中不断的进行修改与变更，需求在初期的过程中对需求定义阶段出现很多不确定的因素，本可以避免的错误在后期过程中影响进度和预算等情况，从而导致项目失败，还有就是开发过程中需求变更处理机制的不合理，麻木进行用户满意度追求，导致附加功能不断增加，原有设计不停变更等，以致于超出预算成本导致项目失败。

4) 项目测试不足

项目开发过程中项目经理经常牺牲项目测试时间压缩几天内完成项目测试，从而导致项目测试结果不到位，用户也放弃测试与检测的责任，开发人员关注系统的性能从而对结果的商业需求满意度缺失导致致命的错误。

5) 无项目时间节点

项目没有具体的开始时间和结束时间，会导致项目在永远不确定的时间无休止的进行，项目成员疲惫最终项目失败。因此必须明确项目何时开始何时结束。

6) 领导者经验不足

项目成功的关键在于领导而不是过多的管理制度，领导需要卓越技能和丰富的经验以及职能以外的奉献精神，也需要激发和带领团队的领导气质。

### 30 [什么是猴子管理法则？](http://www.jobui.com/mianshiti/it/xiangmuguanli/8308/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

为什么有些领导总是看起来那么忙？这是因为他们不懂得猴子管理法则。猴子管理法则通俗地说，就是每个人背上都有一只猴子，这只猴子代表着对自己承担问题的解决责任，如果下属学会解决自己的工作问题，领导就会有更多属于自己的时间去思考自己的问题。

（1）打断下属负面的“依赖”神经链。

（2）训练了下属分析问题、全面思考问题的能力。

（3）让下属产生信心与成就感。他会觉得自己居然也有解决复杂问题的能力。越来越有能力的下属能越来越胜任更重要的任务。

（4）激发下属的行动力。

（5）你将因此不必照看下属的“猴子”而腾出更多的精力去照看自己的“猴子”。

（6）该下属做决定的事，一定要让他们自己学着做决定；

（7）做决定意味着为自己的决定负责任。不想做决定，常常是潜意识里他不想承担责任；

（8）下属不思考问题、不习惯做决定的根源一般有两个：其一是有“托付思想”，依赖上司或别人，这样的下属不堪大用；其二是上司习惯代替下属做决定或喜欢享受别人听命于自己的成就感，这样的上司以及他所带领的团队难以胜任复杂的任务；

（9）让下属自己想办法，做决定，就是训练下属独立思考问题的能力和勇于承担责任的行事风格。

### 31 [如何对软件项目的规模、工作量和成本进行估算?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3963/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

基于代码行和功能点的估算软件项目的规模是影响软件项目成本和工作量的主要因素。

在基于代码行（LOC，Line Of Code）和功能点（Function Point）的估算方法中，利用代码行和功能点来表示软件系统的规模，并通过对软件项目规模的估算进而来估算软件项目的成本和工作量。

显然,一个软件项目的代码行数目越多，它的规模也就越大。软件代码行的数目易于度量，许多软件开发组织和项目组都保留有以往软件项目代码行数目的记录，这有助于在以往类似软件项目代码行记录的基础上对当前软件项目的规模进行估算。

### 32 [在需求分析阶段都通过哪些方式进行需求沟通?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3791/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

1. 问卷调查法， 开发方就用户需求中的一些个性化的、需要进一步明确的需求，通过采用向用户发问卷调查表的方式，达到彻底弄清项目需求的一种需求获取方法。单的问卷调查方法就能使问题得到较好的解决

2. 会议讨论法 ，开发方和用户方召开若干次需求讨论会议，达到彻底弄清项目需求的一种需求获取方法

3. 界面原型法 ，开发方根据自己所了解的用户需求，描画出应用系统的功能界面后与用户进行交流和沟通，通过“界面原型”这一载体，达到双方逐步明确项目需求的一种需求获取的方法。

### 33 [软件项目主合同至少应包括哪些内容?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3942/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

项目名称

项目的技术内容、范围、形式和要求

项目实施计划、进度、期限

地点和方式

项目合同价款、报酬及其支付方式

项目验收标准和方法

各方当事人义务或协作责任

技术成果归属和分享及后续改进的提供与分享规定

技术保密事项

风险责任的承担

违约金或者损失赔偿额的计算方法、仲裁及其它

### 34 [IT项目管理中的团队沟通](http://www.jobui.com/mianshiti/it/xiangmuguanli/3901/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)？

（1）尽早沟通

尽早沟通要求项目经理要有前瞻性，定期与项目成员及项目干系人建立沟通，这不仅容易发现当前存在的问题，而且很多潜在问题也能暴露出来。在项目中出现问题并不可怕，可怕的是问题没被发现。沟通得越晚，暴露得越迟，带来的损失越大。

（2）主动沟通

主动沟通说到底是对沟通的一种态度。在项目中，应该极力提倡主动沟通，尤其是当已经明确了必须要去沟通的时候。当沟通是项目经理面对项目干系人或上级、团队成员面对项目经理时，主动沟通不仅能建立紧密的联系，更能表明你对项目的重视和参与，会使沟通的另一方满意度大大提高，对整个项目非常有利。

（3）内外有别

不管项目组内部有多大的分歧，当面对项目组外部人员，需要处理与项目有关的问题时，要强调对外的一致性，一个项目团队要一种声音说话，这不是一种形式，而是一种文化。面对不同的对象甚至可以选用特定的发言人，这样能取得意想不到的效果。

（4）采用对方能接受的沟通风格

注意肢体语言、语态给对方的感觉。无论在语言和肢体表达上，都需要传递一种合作和双赢的态度，使双方无论在问题的解决上还是在气氛上都达到“双赢”。

（5）沟通的升级原则

横向沟通有平等的感觉，但合理使用纵向沟通，有助于问题的快速解决。沟通的升级可以通过四个步骤来完成。第一步，和对方沟通；第二步与对方的上级沟通；第三步，和自己的上级沟通；第四步，自己的上级与对方的上级沟通。

### 35 [影响团队有效沟通的因素有哪些?](http://www.jobui.com/mianshiti/it/xiangmuguanli/3814/" \t "http://www.jobui.com/mianshiti/it/xiangmuguanli/_blank)

沟通障碍：指延误或曲解消息的因素。沟通障碍的出现必然导致冲突。信息过载、缺少知识、文化差异、组织气候、情绪、沟通线路、选择性认知、噪声、项目行话和术语、距离等造成沟通障碍。