面试宝典

# Java基础知识

## JDK JRE JVM 的区别

### Java 开发工具包 (JDK)

Java开发工具包是Java环境的核心组件，并提供编译、调试和运行一个Java程序所需的所有工具，可执行文件和二进制文件。JDK是一个平台特定的软件，有针对Windows，Mac和Unix系统的不同的安装包。可以说JDK是JRE的超集，它包含了JRE的Java编译器，调试器和核心类。目前JDK的版本号是1.7，也被称为Java 7。

JDK有以下三种版本：

J2SE，standard edition，标准版，是我们通常用的一个版本

J2EE，enterpsise edtion，企业版，使用这种JDK开发J2EE应用程序

J2ME，micro edtion，主要用于移动设备、嵌入式设备上的java应用程序

### Java虚拟机(JVM)

JVM是Java编程语言的核心。当我们运行一个程序时，JVM负责将字节码转换为特定机器代码。JVM也是平台特定的，并提供核心的Java方法，例如内存管理、垃圾回收和安全机制等。JVM 是可定制化的，我们可以通过Java 选项(java options)定制它，比如配置JVM 内存的上下界。JVM之所以被称为虚拟的是因为它提供了一个不依赖于底层操作系统和机器硬件的接口。这种独立于硬件和操作系统的特性正是Java程序可以一次编写多处执行的原因。

### Java运行时环境(JRE)

JRE是JVM的实施实现，它提供了运行Java程序的平台。JRE包含了JVM、Java二进制文件和其它成功执行程序的类文件。JRE不包含任何像Java编译器、调试器之类的开发工具。如果你只是想要执行Java程序，你只需安装JRE即可，没有安装JDK的必要。

### JDK, JRE 和JVM的区别

JDK是用于开发的而JRE是用于运行Java程序的。

JDK和JRE都包含了JVM，从而使得我们可以运行Java程序。

JVM是Java编程语言的核心并且具有平台独立性。

## Jdk1.7新特性

1.自动资源管理

Java中某些资源是需要手动关闭的，如InputStream，Writes，Sockets，Sql classes等。这个新的语言特性允许try语句本身申请更多的资源，

这些资源作用于try代码块，并自动关闭。

2.新增一些取环境信息的工具方法

System.getJavaHomeDir() // JRE的安装目录

File System.getUserHomeDir() // 当前用户目录

File System.getUserDir() // 启动java进程时所在的目录

3.两个char间的equals

4.switch中使用string

## Jdk1.8新特性

1.函数式接口

函数式接口（functional interface 也叫功能性接口，其实是同一个东西）。简单来说，函数式接口是只包含一个方法的接口。比如Java标准库中的java.lang.Runnable和java.util.Comparator都是典型的函数式接口。java 8提供 @FunctionalInterface作为注解,这个注解是非必须的，只要接口符合函数式接口的标准（即只包含一个方法的接口），虚拟机会自动判断，但 最好在接口上使用注解@FunctionalInterface进行声明，以免团队的其他人员错误地往接口中添加新的方法。

2.Lambda语法

包含三个部分

1.一个括号内用逗号分隔的形式参数，参数是函数式接口里面方法的参数

2.一个箭头符号：->

3.方法体，可以是表达式和代码块，方法体函数式接口里面方法的实现，如果是代码块，则必须用{}来包裹起来，且需要一个return 返回值，但有个例外，若函数式接口里面方法返回值是void，则无需{}

(parameters) -> expression 或者 (parameters) -> { statements; }

lambda表达式语法代码简洁，能简化集合上数据的多线程或者多核的处理，提供更快的集合处理速度

3.两个char间的equals

4.switch中使用string

## JDK常用的包

◆java.lang： 这个是系统的基础类，比如String、Math、Integer、System和Thread，提供常用功能。在java.lang包中还有一个子包：java.lang.reflect用于实现java类...

◆java.io: 这里面是所有输入输出有关的类，比如文件操作等

◆java.net: 这里面是与网络有关的类，比如URL,URLConnection等。

◆java.util : 这个是系统辅助类，特别是集合类Collection,List,Map等。

◆java.sql: 这个是数据库操作的类，Connection, Statememt，ResultSet等

## 面向对象的特征

### 抽象

抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用部分细节。抽象包括两个方面，一是过程抽象,二是数据抽象。

### 继承

继承是一种联结类的层次模型，并且允许和鼓励 类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。新类继承了原始类的特性，新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。派生类可以从它的基类那里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。

### 封装

封装是把过程和数据包围起来，对数据的访问只能通过已定义的界面。面向对象计算始于这个基本概念，即现实世界可以被描绘成一系列完全自治、封装的对象,这些对象通过一个受保护的接口访问其他对象。

### 多态性

多态性是指允许不同类的对象对同一消息作出响应。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好的解决了应用程序函数同名问题。

## Java多态的具体体现

面向对象编程有四个特征：抽象，封装，继承，多态。

多态有四种体现形式：

1. 接口和接口的继承。

2. 类和类的继承。

3. 重载。

4. 重写。

## 方法覆盖(Overriding)和方法重载(Overloading)

### 重写（Overriding）

1）父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。在Java中，子类可继承父类中的方法，而不需要重新编写相同的方法。但有时子类并不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。方法重写又称方法覆盖。

2）若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。

3）子类函数的访问修饰权限不能少于父类的；

### 重载(Overloading)

1）方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。重载Overloading是一个类中多态性的一种表现。

2）Java的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法, 这就是多态性。

3）重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

方法的重写Overriding和重载Overloading是Java多态性的不同表现。重写Overriding是父类与子类之间多态性的一种表现，重载Overloading是一个类中多态性的一种表现。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被“屏蔽”了，而且如果子类的方法名和参数类型和个数都和父类相同，那么子类的返回值类型必须和父类的相同；如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载(Overloading)。Overloaded的方法是可以改变返回值的类型。也就是说，重载的返回值类型可以相同也可以不同

## 运行时异常与一般异常有何异同

Throwable 是所有 Java 程序中错误处理的父类 ，有两种资类： Error 和 Exception 。

Error ：表示由 JVM 所侦测到的无法预期的错误，由于这是属于 JVM 层次的严重错误 ，导致 JVM 无法继续执行，因此，这是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息。

Exception ：表示可恢复的例外，这是可捕捉到的。

Java 提供了两类主要的异常 :runtime exception 和 checked exception 。 checked 异常也就是我们经常遇到的 IO 异常，以及 SQL 异常都是这种异常。 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 。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。

但是另外一种异常： runtime exception ，也称运行时异常，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，系统会把异常一直往上层抛，一直遇到处理代码。如果没有处理块，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，则必须扑捉所有的运行时异常，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。 在这个场景这样处理可能是一个比较好的应用，但并不代表在所有的场景你都应该如此。如果在其它场景，遇到了一些错误，如果退出程序比较好，这时你就可以不太理会运行时异常 ，或者是通过对异常的处理显式的控制程序退出

## Integer与int的区别

int是基本类型(Primitive),Integer是引用类型(Warpper).

int是Java的原始数据类型,Integer是java为int提供的封装类.

int的变量不需要实例化,Interger的变量需要实例化.

int的默认值为0,Integer的默认值为null.

int一般做数值参数,Integer一般做类型转换.

int a = 5；

Integer b = 5；

a你只能用来做计算.

b你可以用来做很多事,因为它是一个对象,他有很多静态方法

### Dom4j

dom4j是一个Java的XML API，类似于jdom，用来读写XML文件的。dom4j是一个非常优秀的Java XML API，具有性能优异、功能强大和极端易用使用的特点，同时它也是一个开放源代码的软件。

### SAX

SAX（simple API for XML）是一种XML解析的替代方法。相比于DOM，SAX是一种速度更快，更有效的方法。它逐行扫描文档，一边扫描一边解析。而且相比于DOM，SAX可以在解析文档的任意时刻停止解析，但任何事物都有其相反的一面，对于SAX来说就是操作复杂。

### JAXB

JAXB（Java Architecture for XML Binding) 是一个业界的标准，是一项可以根据XML Schema产生Java类的技术。该过程中，JAXB也提供了将XML实例文档反向生成Java对象树的方法，并能将Java对象树的内容重新写到XML实例文档。从另一方面来讲，JAXB提供了快速而简便的方法将XML模式绑定到Java表示，从而使得Java开发者在Java应用程序中能方便地结合XML数据和处理函数。

### dom4j 与 sax 之间的对比：【注：必须掌握！】

dom4j不适合大文件的解析，因为它是一下子将文件加载到内存中，所以有可能出现内存溢出，

sax是基于事件来对xml进行解析的，所以他可以解析大文件的xml

也正是因为如此，所以dom4j可以对xml进行灵活的增删改查和导航，而sax没有这么强的灵活性

所以sax经常是用来解析大型xml文件，而要对xml文件进行一些灵活（crud）操作就用dom4j

## final,finally,finalize 三者区别

### **Final是一个修饰符**

当final修饰一个变量的时候，变量变成一个常量，它不能被二次赋值

当final修饰的变量为静态变量（即由static修饰）时，必须在声明这个变量的时候给它赋值

当final修饰方法时，该方法不能被重写

当final修饰类时，该类不能被继承

Final不能修饰抽象类，因为抽象类中会有需要子类实现的抽 象方法，（抽象类中可以有抽象方法，也可以有普通方法，当一个抽象类中没有抽象方 法时，这个抽象类也就没有了它存在的必要）

Final不能修饰接口，因为接口中有需要其实现类来实现的方法

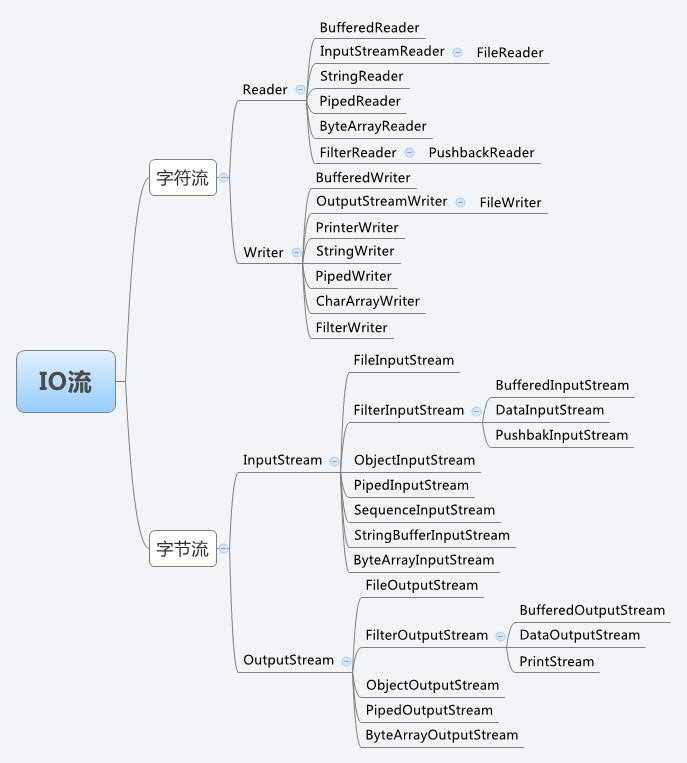
### Finally

Finally只能与try/catch语句结合使用，finally语句块中的语句一定会执行， 并且会在return，continue，break关键字之前执行

### finalize

Finalize是一个方法，属于java.lang.Object类，finalize()方法是GC （garbage collector垃圾回收）运行机制的一部分，finalize()方法是在 GC清理它所从属的对象时被调用的

## IO流



## 多线程

### java中多线程同步是什么

在多线程程序下，同步能控制对共享资源的访问。如果没有同步，当一个java线程在修改一个共享变量时，另外一个线程正在使用或者更新同一个变量，这样容易导致程序出现错误。

### 解释实现多线程的几种方法？区别是什么？

Java线程可以实现Runnable接口或者继承Thread类来实现，当你打算多重继承时， 优先选择实现Runnable

启动方法不一样。Thread1继承，Thread2实现Runnable接口，则启动一

个Thread1线程可以使用new Thread1().start(),而启动Thread2线程则new

Thread(new Thread2()).start()。

### 3.Thread.start()与Thread.run()有什么区别

run()方法:在本线程内调用该Runnable对象的run()方法，可以重复多次调用；

start()方法:启动一个线程，调用该Runnable对象的run()方法，不能多次启动一个线程；

### 为什么需要run（）和start（）方法，我们可以只用run（）方法来完成任务吗

我们需要run（）&start（）这两个方法是因为JVM创建一个单独的线程不同于普通方法的调用，所以这项工作由线程的start方法来完成，start由本地方法实现，需要显式地被调用，使用这两个方法的另外一个好处是任何一个对象都可以作为线程运行，只要实现了Runnable接口，这就避免了因继承了Thread类而造成的java的多继承问题。

### 什么是ThreadLocal类，怎么使用它？

ThreadLocal是一个线程级别的局部变量，并非“本地线程”。ThreadLocal为每个使用该变量的线程提供了一个独立的变量副本，每个线程修改副本时不影响其他线程对象的副本

线程局部变量（ThreadLocal variables)的关键点：

一个线程局部变量为每个线程方便的提供了一个单独的变量。Threadlocal实例通常作为静态的私有的字段出现在一个类中，这个类用来关联一个线程。当多个线程访问ThreadLocal实例时，每个线程维护ThreadLocal提供的独立的变量副本。常用的使用可在Dao模式中见到，当Dao类作为一个单例类时，数据库连接被每一个线程独立的维护，互不影响。

### 什么时候抛出 InvalidMonitorStateException 异常，为什么？

调用 wait ()/notify ()/notifyAll ()中的任何一个方法时，如果当前线程没有获得该对象的锁，那么就会抛出 IllegalMonitorStateException 的异常(也就是说程序在没有执行对象的任何同步块或者同步方法时，仍然尝试调用 wait ()/notify ()/notifyAll ()时)。由于该异常是 RuntimeExcpetion 的子类，所以该异常不一定要捕获(尽管你可以捕获只要你愿意).作为 RuntimeException，此类异常不会在 wait (),notify (),notifyAll ()的方法签名提及。

### Sleep ()、suspend ()和 wait ()之间有什么区别？

Thread.sleep ()使当前线程在指定的时间处于“非运行”（Not Runnable）状态。线程一直持有对象的监视器。比如一个线程当前在一个同步块或同步方法中，其它线程不能进入该块或方法中。如果另一线程调用了 interrupt ()方法，它将唤醒那个“睡眠的”线程。

**注意：**sleep ()是一个静态方法。这意味着只对当前线程有效，一个常见的错误是调用t.sleep ()，（这里的t是一个不同于当前线程的线程）。即便是执行t.sleep ()，也是当前线程进入睡眠，而不是t线程。t.suspend ()是过时的方法，使用 suspend ()导致线程进入停滞状态，该线程会一直持有对象的监视器，suspend ()容易引起死锁问题。

object.wait ()使当前线程出于“不可运行”状态，和 sleep ()不同的是 wait 是 object 的方法而不是 thread。调用 object.wait ()时，线程先要获取这个对象的对象锁，当前线程必须在锁对象保持同步，把当前线程添加到等待队列中，随后另一线程可以同步同一个对象锁来调用 object.notify ()，这样将唤醒原来等待中的线程，然后释放该锁。基本上 wait ()/notify ()与 sleep ()/interrupt ()类似，只是前者需要获取对象锁。

### 在静态方法上使用同步时会发生什么事？

同步静态方法时会获取该类的“Class”对象，所以当一个线程进入同步的静态方法中时，线程监视器获取类本身的对象锁，其它线程不能进入这个类的任何静态同步方法。它不像实例方法，因为多个线程可以同时访问不同实例同步实例方法。

### 当一个同步方法已经执行，线程能够调用对象上的非同步实例方法吗？

可以，一个非同步方法总是可以被调用而不会有任何问题。实际上，Java 没有为非同步方法做任何检查，锁对象仅仅在同步方法或者同步代码块中检查。如果一个方法没有声明为同步，即使你在使用共享数据 Java 照样会调用，而不会做检查是否安全，所以在这种情况下要特别小心。一个方法是否声明为同步取决于临界区访问(critial section access)，如果方法不访问临界区(共享资源或者数据结构)就没必要声明为同步的。

### 在一个对象上两个线程可以调用两个不同的同步实例方法吗？

不能，因为一个对象已经同步了实例方法，线程获取了对象的对象锁。所以只有执行完该方法释放对象锁后才能执行其它同步方法。看下面代码示例非常清晰：Common 类有 synchronizedMethod1()和 synchronizedMethod2()方法，MyThread 调用这两个方法。

### 什么是死锁

死锁就是两个或两个以上的线程被无限的阻塞，线程之间相互等待所需资源。这种情况可能发生在当两个线程尝试获取其它资源的锁，而每个线程又陷入无限等待其它资源锁的释放，除非一个用户进程被终止。就 JavaAPI 而言，线程死锁可能发生在一下情况。

当两个线程相互调用 Thread.join ()

当两个线程使用嵌套的同步块，一个线程占用了另外一个线程必需的锁，互相等待时被阻塞就有可能出现死锁。

### 什么是线程饿死，什么是活锁？

线程饿死和活锁虽然不想是死锁一样的常见问题，但是对于并发编程的设计者来说就像一次邂逅一样。

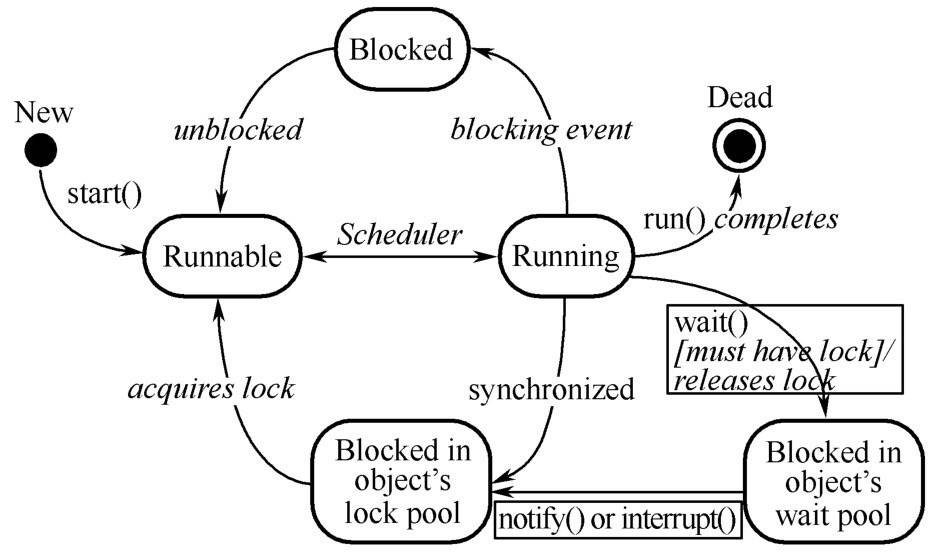
当所有线程阻塞，或者由于需要的资源无效而不能处理，不存在非阻塞线程使资源可用。JavaAPI 中线程活锁可能发生在以下情形：

当所有线程在程序中执行 Object.wait (0)，参数为 0 的 wait 方法。程序将发生活锁直到在相应的对象上有线程调用 Object.notify ()或者 Object.notifyAll ()。

当所有线程卡在无限循环中。Java程序员面试中的多线程问题

### 13.线程的基本概念、线程的基本状态以及状态之间的关系

线程指在程序执行过程中，能够执行程序代码的一个执行单位，每个程序至少都有一个线程，也就是程序本身。Java中的线程有四种状态分别是新建 运行、就绪、挂起、结束



### 简述synchronized和java.util.concurrent.locks.Lock的异同 ？

主要相同点：Lock能完成synchronized所实现的所有功能

主要不同点：Lock有比synchronized更精确的线程语义和更好的性能。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且必须在finally从句中释放。

### wait()，wait(long timeout)和notify()、notifyAll()

这三个方法用于协调多个线程对共享数据的存取，所以必须在Synchronized语句块内使用这三个方法。Synchronized这个关键字用于保护共享数据，阻止其他线程对共享数据的存取。

wait()方法使当前线程暂停执行并释放对象锁标志，让其他线程可以进入Synchronized数据块，当前线程被放入对象等待池中。 wait(timeout)等待timeout长的时间，如果这段时间过了还没有被唤醒，则自动唤醒。

当调用 notify()方法后，将从对象的等待池中移走一个任意的线程并放到锁标志等待池中，只有锁标志等待池中的线程能够获取锁标志；如果锁标志等待池中没有线程，则notify()不起作用。

notifyAll()则从对象等待池中移走所有等待那个对象的线程并放到锁标志等待池中。

注意 这三个方法都是java.lang.Ojbect的方法!

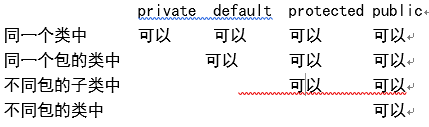
### 请说出你所知道的线程同步的方法

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。S

notifyAll():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。

## 修饰符的作用范围



## ==和equals的区别

java中equals和==的区别 值类型是存储在内存中的堆栈（简称栈），而引用类型的变量在栈中仅仅是存储引用类型变量的地址，而其本身则存储在堆中。

==操作比较的是两个变量的值是否相等，对于引用型变量表示的是两个变量在堆中存储的地址是否相同，即栈中的内容是否相同。

equals操作表示的两个变量是否是对同一个对象的引用，即堆中的内容是否相同。

## Java动态代理

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。

动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。java.lang.reflect 包中的Proxy类和InvocationHandler 接口提供了生成动态代理类的能力。

JDK的动态代理依靠接口实现，如果有些类并没有实现接口，则不能使用JDK代理，这就要使用cglib动态代理了。

## Cglib动态代理

JDK的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现JDK的动态代理，cglib是针对类来实现代理的，他的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。

## 当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递?

是值传递。Java 编程语言只有值传递参数。当一个对象实例作为一个参数

被传递到方法中时，参数的值就是对该对象的引用。对象的内容可以在被调用的方法中改变，但对象的引用是永远不会改变的。

值传递：方法调用时，实际参数把它的值传递给对应的形式参数，方法执行中形式参数值的改变不影响实际参 数的值。

引用传递：也称为传地址。方法调用时，实际参数的引用(地址，而不是参数的值)被传递给方法中相对应的形式参数，在方法执行中，对形式参数的操作实际上就是对实际参数的操作，方法执行中形式参数值的改变将会影响实际参数的值。

## abstract class 和interface 有什么区别

声明方法的存在而不去实现它的类被叫做抽象类（abstract class），它用于要创建一个体现某些基本行为的类，并为该类声明方法，但不能在该类中实现该类的情况。不能创建abstract 类的实例。然而可以创建一个变量，其类型是一个抽象类，并让它指向具体子类的一个实例。不能有抽象构造函数或抽象静态方法。Abstract 类的子类为它们父类中的所有抽象方法提供实现，否则它们也是抽象类。取而代之，在子类中实现该方法。知道其行为的其它类可以在类中实现这些方法。接口（interface）是抽象类的变体。新型多继承性可通过实现这样的接口而获得。接口中的所有方法都是抽象的，所有成员变量都是public static final 的。一个类可以实现多个接口，当类实现特殊接口时，它定义（即将程序体给予）所有这种接口的方法。然后，它可以在实现了该接口的类的任何对象上调用接口的方法。由于有抽象类，它允许使用接口名作为引用变量的类型。通常的动态联编将生效。引用可以转换到接口类型或从接口类型转换，instanceof 运算符可以用来决定某对象的类是否实现了接口。

## 字符串操作：如何实现字符串的反转及替换？

可用字符串构造一个StringBuffer 对象,然后调用StringBuffer 中的reverse

方法即可实现字符串的反转,调用replace 方法即可实现字符串的替换。

# JavaEE

## 网络七层协议

OSI是一个开放性的通信系统互连参考模型，他是一个定义得非常好的协议规范。OSI模型有7层结构，每层都可以有几个子层。 OSI的7层从上到下分别是 7 应用层 6 表示层 5 会话层 4 传输层 3 网络层 2 数据链路层 1 物理层 ；其中高层（即7、6、5、4层）定义了应用程序的功能，下面3层（即3、2、1层）主要面向通过网络的端到端的数据流。

### 分层的优点

（1）人们可以很容易的讨论和学习协议的规范细节。

（2）层间的标准接口方便了工程模块化。

（3）创建了一个更好的互连环境。

（4）降低了复杂度，使程序更容易修改，产品开发的速度更快。

（5）每层利用紧邻的下层服务，更容易记住各层的功能。

大多数的计算机网络都采用层次式结构，即将一个计算机网络分为若干层次， 处在高层次的系统仅是利用较低层次的系统提供的接口和功能，不需了解低层实现该功能所采用的算法和协议；较低层次也仅是使用从高层系统传送来的参数，这就 是层次间的无关性。因为有了这种无关性，层次间的每个模块可以用一个新的模块取代，只要新的模块与旧的模块具有相同的功能和接口，即使它们使用的算法和协 议都不一样。

网络中的计算机与终端间要想正确的传送信息和数据，必须在数据传输的顺序、数据的格式及内容等方面有一个约定或规则，这种约定或规则称做协议。

## HTTPS和HTTP的区别

https协议需要到ca申请证书，一般免费证书很少，需要交费。

http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议

http和https使用的是完全不同的连接方式用的端口也不一样,前者是80,后者是443。

http的连接很简单,是无状态的

HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议 要比http协议安全

## web.xml的作用

web.xml文件是用来初始化配置信息：比如Welcome页面、servlet、servlet-mapping、filter、listener、启动加载级别等。

Web.xml中常用的一些标签元素及其功能：

1、指定欢迎页面

2、命名与定制URL。我们可以为Servlet和JSP文件命名并定制URL,其中定制URL是依赖命名的，命名必须在定制URL前。

3、定制初始化参数：可以定制servlet、JSP、Context的初始化参数，然后可以再servlet、JSP、Context中获取这些参数值。

4、指定错误处理页面，可以通过“异常类型”或“错误码”来指定错误处理页面。

5、设置过滤器：比如设置一个编码过滤器，过滤所有资源 。

6、设置监听器。

7、设置会话(Session)过期时间，其中时间以分钟为单位。

## CS架构、BS架构

C/S=Client/Server

B/S=Brower/Server

一个基于客户端/服务器，一个基于浏览器/服务器

CS架构的优点就是基于客户机和服务器模式,大部分的运算是在客户机上运行,从而提高一定的效率,但是也因为他在客户机上运行,所以要安装客户端,也导致了软件维护和升级时的不方便,需要程序员亲自进行升级配置，然后BS架购的优点在于,软件维护和升级方便,只要能上网就能升级了且在很多地方都能够直接使用软件,而不需要安装,前提是你要能登陆到服务器。

## 重构

重构可以说是覆盖,用覆盖一词来讲就比较清楚。在面向对象的一个特性--继承中就表现出重构的意思。

重构，就是继承于父类方法的子类将子类的方法重写，方法名及参数完全相同，就是有相同的声明，但是实现不一致或完全不同。这样如果对面向对象理解比较深刻的话，就清楚重构正是面向对象编程的一大特性。

## 分层的优缺点

### 优点

1.分离开发人员的关注。由于某一层仅仅调用其相邻下一层所提供的服务，所以，只要本层的API和相邻下一层的API定义完整，开发人员在开发某一层时就可以像关注集中于这一层所用的思想、模式、技术，这样，就等同于将分工带来的生产力提高优势引入软件开发。又如买蛋糕的例子，作为超市，只要知道下层API(如何从蛋糕厂获取蛋糕)和本层需要实现的API(把蛋糕销售给客户)，就可以制定自己的业务模式和策略计划了，而不必关心如何种小麦、如何磨面粉、如何做奶油、如何做蛋糕等。这样，超市只需进行商业运作，而不必进行产业运作，如此专一，必然提高业务水平。

2.无损替换。想象一下，如果某家奶牛场倒闭了，奶油加工厂也要跟着倒闭吗?当然不会，它可以迅速更换一家奶牛场，因为各个奶牛场都可以实现“提供牛奶”这项服务。再譬如，如果某天国家出台政策，要求所有奶油厂必须从审查合格的奶牛场引进原料，恰好某奶油厂的合作牛奶供应商没能通过审查，那么，只要换一家通过审查的合作就行了。而且奶油厂内部的各个环节一动不用动，因为不同的奶牛场都可以提供“供应牛奶”这个服务。而如果奶油厂自己养牛生产牛奶，一旦遇到这个政策，还得自己去有关部门进行审查，调整相应业务流程，牵一发而动全身。程序中同样的道理，最常听说的可能就是迁移数据库了。

3.降低了系统间的依赖。还是蛋糕那个例子，如果某天蛋糕厂内部换机器了，或业务流程调整了，请问顾客需要关心吗?显然不用，因为顾客只调用超市提供的服务。而超市为顾客隐藏了下面所有产业细节。如果每一个顾客买一样商品，都要了解这个商品从原料生产到成型再到销售的一系列细节，岂不累死了。换做程序中，就如表示层只管调用业务层的服务，至于业务层下还有几层?各种数据是怎么来的?怎么存的?是真实的还是捏造的?都不需要了解，这大大降低了系统各职责之间的依赖。

4.复用。例如，你可以去这个超市买东西，我也可以去这个超市买东西。蛋糕厂可以从面粉厂提取面粉，馒头厂也可以。这样，同样的层就可以为不同的上层提供服务，达到了复用的目的。具体到程序中，例如气象局制作发布了一个“Service Layer”，用于提供天气预告信息。这样新浪、搜狐这些网站可以利用这个服务层提供的服务，制作天气预告页面，QQ也可以利用这个服务在它的聊天工具上添加天气预告，你自己做一个软件需要用到天气预告功能，也可以调用气象台的“Service Layer”。

### 缺点

1.级联修改问题。这个问题在现实中不好比喻，但在程序中相信很多朋友都明白。例如，一个人事管理系统，本来查看人员信息只能分页查看，而现在，需要增加一个功能：在分页的同时还能分部门。例如，可以查看“销售部的前50个人”，这样，为了这个功能所有层都需要修改。

2.性能问题。本来直来直去的操作，现在要层层传递，势必造成性能的下降。就如在购买蛋糕的例子中。顾客在享受分工带来的便利时，也要承受由于不同层的部门分布各地而造成的蛋糕价格上升，这是因为分层增加了成本，如运输、不同层间部门的协调管理成本等。

纵观以上分析，分层架构有利有弊。这是一定得，世上任何事物都有利弊，所以，把“分层架构捧上天”和“一棍子打死”这两种做法都是不明智也是不科学的。对待分层架构，我们的态度应当是明晰其本质和利弊，然后根绝具体情况做出理性的分析和抉择。

从上面的分析可以看出，分层架构可以降低层内变化的成本，而对于API的变化非常敏感。如在级联修改中提到的“在分页的同时还能分部门”的新需求，就是对API进行的变动。API的变动对于分层架构是致命的，修改起来难度非常大。所以，一个简单的判断法则就是：如果您的系统层内频繁变动(甚至整层替换)可能性很大，而API变动可能性很小，就使用分层;而如果API可能会频繁变动，那就要谨慎使用分层架构了。

# JavaEE常用设计模式

## 简单工厂

又叫做静态工厂方法（StaticFactory Method）模式，但不属于23种GOF设计模式之一。

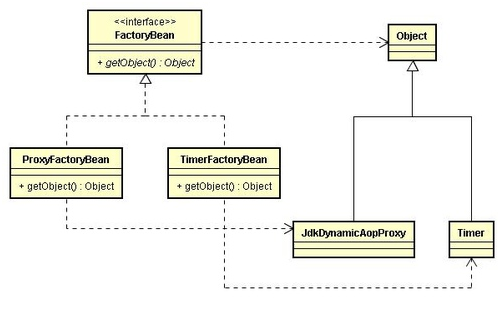
简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

## 工厂方法（Factory Method）

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使一个类的实例化延迟到其子类。

Spring中的FactoryBean就是典型的工厂方法模式。如下图：



## 单例（Singleton）

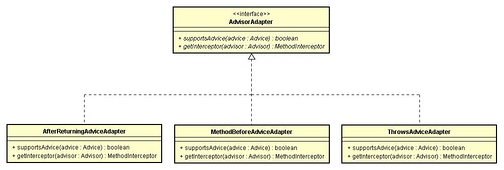
保证一个类仅有一个实例，并提供一个访问它的全局访问点。

Spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为Spring管理的是是任意的Java对象。

## 适配器

将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

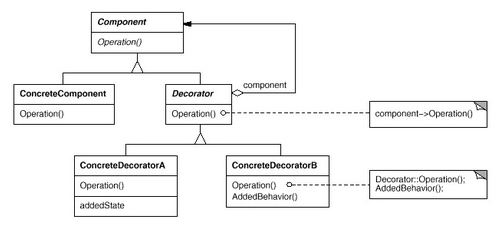
Spring中在对于AOP的处理中有Adapter模式的例子，见如下图：



由于Advisor链需要的是MethodInterceptor对象，所以每一个Advisor中的Advice都要适配成对应的MethodInterceptor对象。

## 包装器（Decorator）

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

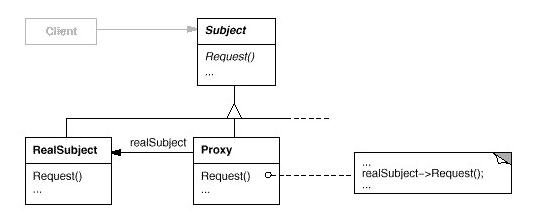


Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。基本上都是动态地给一个对象添加一些额外的职责。

## 代理（Proxy）

为其他对象提供一种代理以控制对这个对象的访问。

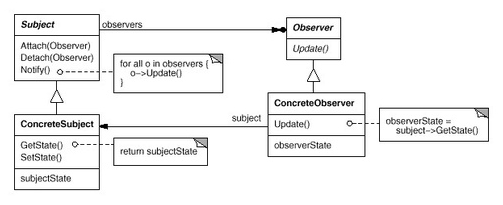
从结构上来看和Decorator模式类似，但Proxy是控制，更像是一种对功能的限制，而Decorator是增加职责。



Spring的Proxy模式在aop中有体现，比如JdkDynamicAopProxy和Cglib2AopProxy。

## 观察者（Observer）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

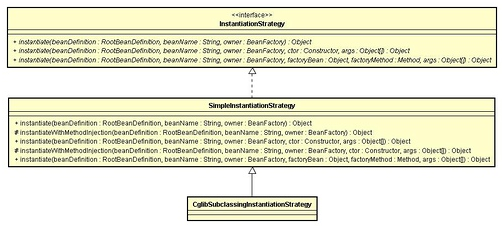


Spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

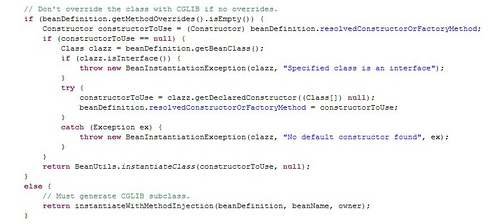
## 策略（Strategy）

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

Spring中在实例化对象的时候用到Strategy模式，见如下图：



在SimpleInstantiationStrategy中有如下代码说明了策略模式的使用情况：



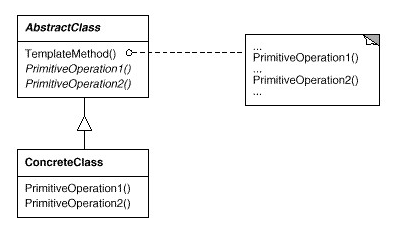
## 模板方法（Template Method）

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

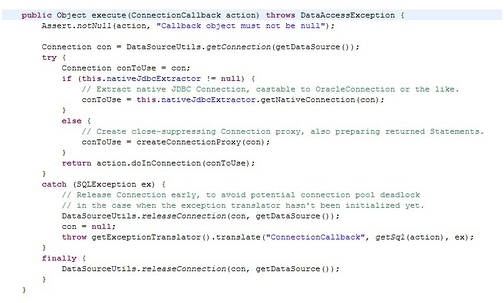
Template Method模式一般是需要继承的。这里想要探讨另一种对Template Method的理解。Spring中的JdbcTemplate，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到JdbcTemplate已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入JdbcTemplate的方法中。但是变化的东西是一段代码，而且这段代码会用到JdbcTemplate中的变量。怎么办？那我们就用回调对象吧。在这个回调对象中定义一个操纵JdbcTemplate中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到JdbcTemplate，从而完成了调用。这可能是Template Method不需要继承的另一种实现方式吧。

以下是一个具体的例子：

JdbcTemplate中的execute方法：



JdbcTemplate执行execute方法：



# 前台问题

## js如何创建对象

### 工厂方法

var lev=function(){

return "啊打";

};

function Parent(){

var Child = new Object();

Child.name="李小龙";

Child.age="30";

Child.lev=lev;

return Child;

};

var x = Parent();

alert(x.name);

alert(x.lev());

**说明：**

1.在函数中定义对象,并定义对象的各种属性，,虽然属性可以为方法，但是建议将属性为方法的属性定义到函数之外，这样可以避免重复创建该方法

2.引用该对象的时候，这里使用的是 var x = Parent()而不是 var x = new Parent();因为后者会可能出现很多问题（前者也成为工厂经典方式,后者称之为混合工厂方式），不推荐使用new的方式使用该对象

3.在函数的最后返回该对象

4.不推荐使用这种方式创建对象，但应该了解

### 构造函数方式

var lev=function(){

return "啊打";

};

function Parent(){

this.name="李小龙";

this.age="30";

this.lev=lev;

};

var x =new Parent();

alert(x.name);

alert(x.lev());

**说明：**

1.与工厂方式相比，使用构造函数方式创建对象，无需再函数内部重建创建对象，而使用this指代，并而函数无需明确return

2.同工厂模式一样，虽然属性的值可以为方法，仍建议将该方法定义在函数之外

3..同样的，不推荐使用这种方式创建对象，但仍需要了解

### 原型模式

var lev=function(){

return "啊打";

};

function Parent(){

};

Parent.prototype.name="李小龙";

Parent.prototype.age="30";

Parent.prototype.lev=lev;

var x =new Parent();

alert(x.name);

alert(x.lev());

**说明：**

1.函数中不对属性进行定义

2.利用prototype属性对属性进行定义

3.同样的，不推荐使用这样方式创建对象

### 混合的构造函数 原型方法（推荐）

function Parent(){

this.name="李小龙";

this.age=32;

};

Parent.prototype.lev=function(){

return this.name;

};;

var x =new Parent();

alert(x.lev());

**说明：**

1.该模式是指混合搭配使用构造函数方式和原型方式

2.将所有属性不是方法的属性定义在函数中（构造函数方式），将所有属性值为方法的属性利用prototype在函数之外定义（原型方式）

3.推荐使用这样方式创建对象,这样做有好处和为什么不单独使用构造函数方式和原型方式，由于篇幅问题这里不予讨论。

### 动态原型方式

function Parent(){

this.name="李小龙";

this.age=32;

if(typeof Parent.\_lev=="undefined"){

Parent.prototype.lev=function(){

return this.name;

}

Parent.\_lev=true;

}

};

var x =new Parent();

alert(x.lev());

**说明：**

1.动态原型方式可以理解为混合构造函数，原型方式的一个特例

2.该模式中,属性为方法的属性直接在函数中进行了定义，但是因为

if(typeof Parent.\_lev=="undefined"){

Parent.\_lev=true;

}

从而保证创建该对象的实例时，属性的方法不会被重复创建

3.推荐使用这种模式

## 撰写CSS样式，将鼠标样式变成手指形

CSS鼠标手型效果 <a href="#" style="cursor:hand">CSS鼠标手型效果</a>

CSS鼠标手型效果 <a href="#" style="cursor:pointer">CSS鼠标手型效果</a>

注：pointer也是小手鼠标，建议大家用pointer，因为它可以兼容多种浏览器

## js隐藏超链接在地址栏里显示的地址

window.status='';

当鼠标在链接上有事件触发时，将状态栏的文本设置为空白或者其它文本就行了

## 页面禁止缓存

**前台：**

<meta http-equiv="pragma" content="no-cache">

<meta http-equiv="cache-control" content="no-cache">

<meta http-equiv="expires" content="0">

**后台：**

response.setHeader("Pragma","No-cache");

response.setHeader("Cache-Control","no-cache");

response.setDateHeader("Expires", 0);

# 框架问题

## MVC

### 什么是MVC？

MVC是Model,View,Controller的缩写,MVC是Application开发的设计模式，

也就是大家所知道的Model2.在MVC的设计模式中，它包括三类对象：

（1）模型（Model）对象：是应用程序的主体部分。

（2）视图（View）对象：是应用程序中负责生成用户界面的部分。

（3）控制器（Control）对象：是根据用户的输入，控制用户界面数据显示及更新Model对象状态的部分。即要求在Application开发中你把商业逻辑，界面显示，数据分离。

### MVC有哪些优缺点？

**优点：**

1）视图控制模型分离， 提高代码重用性。

2）提高开发效率。

3）便于后期维护， 降低维护成本。

4）方便多开发人员间的分工。

**缺点：**

1）清晰的构架以代码的复杂性为代价， 对小项目优可能反而降低开发效率。

2）运行效率相对较低

3）目前没有比较好的rich 客户端的解决方案

4) 控制层和表现层有时会过于紧密，导致没有真正分离和重用

## Struts2

首先在项目启动的时候，项目下的Struts.xml文件会被dom4j工具解析；而当我们发送一个请求时，这个请求被 StrutsPrepareAndExecuteFilter 拦截,而我们配置了struts处理url的拦截表达式，比如在<filter- maping>中我们配置<url -pattern> 为 /\* 或 \*.do的通配符。这样当你发送一个请求的时候就被拦截器给拦截下来了。

而针对这样的请求其实是由 struts的过滤器处理的。这个过滤器我在看源代码的时候，发现它是在生命周期初始化的时候，在一个init()方法中 利用 filterConfig的对象，找到了项目中的struts.xml配置文件，然后通过InitOperations() 和 Dispatcher 把 struts.xml 文件转换成 ActionMapping对象进行的解析。

当我们在客户端发送一个请求的时侯PrepareOperations类里面 的 prepare.createActionContext(request, response)方法会帮我们封装 request和response ，并且创建 ValueStack（值栈对象）；struts用 dispatcher 对象 的 createContextMap 方法 把 上面request和session 还有用户传递过来的所有数据，通过request.getParameterMap()先获取到前台，然后全部封装到了 ActionContext 和ValueStack 中,所以我们在struts2中获取 request和session等对象的时候有一个解耦合的方式 ActionContext.getContext().get("request")就能获取到request和session等的对象;

比如用户发送了一个usermanager的请求,那么在之前封装好的ActionMapping中会有一个与这个请求一一对应的属性值,通过在代码中执行一个 findActionMapping(request, response, true)的方法,来得到一一对应的关系，那是因为在ActionMapping中本身就包含着我们的业务Action的路径。

ActionProxyFactory 会生成一个 ActionProxy 对我们的业务Action进行一个反射的代理操作。所以我们Action并不是一个Servlet，但是却能达到servlet的效果，是因为serlvet的操作都是 StrutsPrepareAndExecuteFilter的dofilter方法执行的。它通过刚才说的 ActionProxy 反射执行我们 Action中的业务方法，然后返回的值string和 ActionMapping中的Result对象进行匹配，最后在让StrutsPrepareAndExecuteFilter 帮我们执行跳转或转发的功能。

### 运行原理

1、tomcat 启动的时候会加载 web.xml 、核心控制器 FilterDispatcher 会加载并解析 struts.xml

2、客户端会发送一个请求到 action 、FilterDispatcher 会根据后缀名进行拦截

3、FilterDispatcher根据 struts.xml 的配置文件信息 找到 某个action 对应的某个类里的指定方法

4、执行相关的业务逻辑最后返回 一个String

5、<action/> 里配置 <result/> name的属性值与返回的String 进行匹配,跳转到指定的jsp 页面

### 体系结构

1、客户端向Servlet容器（例如Tomcat）发送一个请求；

2、这个请求经过一系列的过滤器（Filter）；

3、接着FilterDispatcher被调用，FilterDispatcher询问ActionMapper来决定这个请求是否需要调用某个Action；

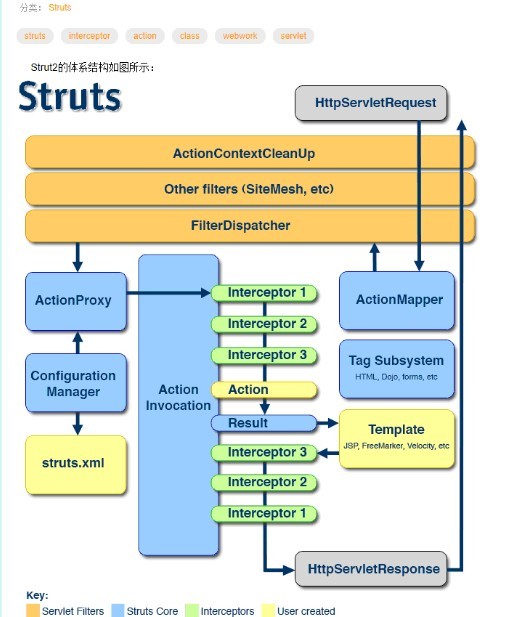
4、如果ActionMapper决定需要调用某个Action，FilterDispatcher把请求的处理交给ActionProxy；

5、ActionProxy通过Configuration Manager询问框架的配置文件，找到需要调用的Action类；

6、ActionProxy创建一个ActionInvocation的实例。

7、ActionInvocation在调用Action的过程前后，涉及到相关拦截器（Intercepter）的调用。

8、一旦Action执行完毕，ActionInvocation负责根据struts.xml中的配置找到对应的返回结果。返回结果通常是jsp或者FreeMarker的模版。（体系结构图见下一页）



### result中的type类型

1.dispatcher：它是默认的，用来转向页面，通常处理JSP

2.redirect：将用户重定向到一个已配置好的URL

3.redirectAction：将用户重定向到一个已定义好的action

4.chain：将action和另外一个action链接起来

5.freemarker：呈现Freemarker模板

6.httpheader：返回一个已配置好的HTTP头信息响应

7.stream：向浏览器发送InputSream对象对下载的内容和图片非常有用

8.velocity：呈现Velocity模板

9.xslt ：该XML可以通过XSL模板进行转换

10.plaintext：显示原始文件内容，例如文件源代码

### 拦截器运行原理

大部分时候，拦截器方法都是通过代理的方式来调用的。Struts 2的拦截器实现相对简单。当请求到达Struts 2的ServletDispatcher时，Struts 2会查找配置文件，并根据其配置实例化相对的拦截器对象，然后串成一个列表（list），最后一个一个地调用列表中的拦截器。事实上，我们之所以能够如此灵活地使用拦截器，完全归功于“动态代理”的使用。动态代理是代理对象根据客户的需求做出不同的处理。对于客户来说，只要知道一个代理对象就行了。那Struts2中，拦截器是如何通过动态代理被调用的呢？当Action请求到来的时候，会由系统的代理生成一个Action的代理对象，由这个代理对象调用Action的execute()或指定的方法，并在struts.xml中查找与该Action对应的拦截器。如果有对应的拦截器，就在Action的方法执行前（后）调用这些拦截器；如果没有对应的拦截器则执行Action的方法

## Struts1.x与Struts2.x的区别

Struts 2以WebWork为核心，

采用拦截器的机制来处理用户的请求，struts1严重依赖于servletAPI,

属于侵入性框架,struts2不严重依赖于servletAPI,属于非侵入型框架。

线程模型方面：

Struts1的Action是单实例的,一个Action的实例处理所有的请求。

Struts2的Action是一个请求对应一个实例（每次请求时都新new出一个对象）,没有线程安全方面的问题

封装请求参数：

Struts1中强制使用ActionForm对象封装请求的参数。

Struts2可以选择使用POJO类来封装请求的参数，或者直接使用Action的属性。

struts1的前端总控制器(核心总控制器)为ActionServlet,

struts2的前端总控制器(核心总控制器)为FilterDispather

## Spring MVC

### Spring MVC运行原理

整个处理过程从一个HTTP请求开始：

1.Tomcat在启动时加载解析web.xml,找到spring mvc的前端总控制器DispatcherServlet,并且通过DispatcherServlet来加载相关的配置文件信息。

2.DispatcherServlet接收到客户端请求，找到对应HandlerMapping，根据映射规则，找到对应的处理器（Handler）。

3.调用相应处理器中的处理方法，处理该请求后，会返回一个ModelAndView。

4.DispatcherServlet根据得到的ModelAndView中的视图对象，找到一个合适的ViewResolver（视图解析器），根据视图解析器的配置，DispatcherServlet将要显示的数据传给对应的视图，最后显示给用户。

## Spring MVC、struts1和struts2区别

1.spring mvc 单例 非线程安全

struts1单例 非线程安全

struts2线程安全对每个请求都产生一个实例

2.spring mvc的入口是servlet，而struts2是filter

spring 的前端总控制器为 DispatcherServlet

struts2 的前端总控制器为 FilterDispatcher

struts1 的前端总控制器为 actionServlet

3. 参数传递：struts是在接受参数的时候，

可以用属性来接受参数，这就说明参数是让多个方法共享的。

springmvc 用方法来接受参数

4.spring mvc是基于方法的设计,而sturts是基于类

## Spring

### Ioc

首先想说说IoC（Inversion of Control，控制倒转）。这是spring的核心，贯穿始终。所谓IoC，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系。举个简单的例子，我们是如何找女朋友的？常见的情况是，我们到处去看哪里有长得漂亮身材又好的mm，然后打听她们的兴趣爱好、qq号、电话号、ip号、iq号………，想办法认识她们，投其所好送其所要，然后嘿嘿……这个过程是复杂深奥的，我们必须自己设计和面对每个环节。传统的程序开发也是如此，在一个对象中，如果要使用另外的对象，就必须得到它（自己new一个，或者从JNDI中查询一个），使用完之后还要将对象销毁（比如Connection等），对象始终会和其他的接口或类藕合起来。 那么IoC是如何做的呢？有点像通过婚介找女朋友，在我和女朋友之间引入了一个第三者：婚姻介绍所。婚介管理了很多男男女女的资料，我可以向婚介提出一个列表，告诉它我想找个什么样的女朋友，比如长得像李嘉欣，身材像林熙雷，唱歌像周杰伦，速度像卡洛斯，技术像齐达内之类的，然后婚介就会按照我们的要求，提供一个mm，我们只需要去和她谈恋爱、结婚就行了。简单明了，如果婚介给我们的人选不符合要求，我们就会抛出异常。整个过程不再由我自己控制，而是有婚介这样一个类似容器的机构来控制。Spring所倡导的开发方式就是如此，所有的类都会在spring容器中登记，告诉spring你是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 spring来控制，也就是说控制对象生存周期的不再是引用它的对象，而是spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转

### Spring事务传播属性和隔离级别

#### **事务的传播属性（Propagation）**



1) REQUIRED ，这个是默认的属性

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

如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

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

2) MANDATORY

Support a current transaction, throw an exception if none exists.支持当前事务，如果当前没有事务，就抛出异常。

3) NEVER

Execute non-transactionally, throw an exception if a transaction exists.

以非事务方式执行，如果当前存在事务，则抛出异常。

4) NOT\_SUPPORTED

Execute non-transactionally, suspend the current transaction if one exists.

以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

5) REQUIRES\_NEW

Create a new transaction, suspend the current transaction if one exists.

新建事务，如果当前存在事务，把当前事务挂起。

6) SUPPORTS

Support a current transaction, execute non-transactionally if none exists.

支持当前事务，如果当前没有事务，就以非事务方式执行。

7) NESTED

Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.

支持当前事务，新增Savepoint点，与当前事务同步提交或回滚。

嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

8) PROPAGATION\_NESTED 与PROPAGATION\_REQUIRES\_NEW的区别

它们非常 类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用PROPAGATION\_REQUIRES\_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA 事务管理器的支持。

使用PROPAGATION\_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

#### 事务的隔离级别（Isolation Level）

首先说明一下事务并发引起的三种情况

i. Dirty Reads 脏读

一个事务正在对数据进行更新操作，但是更新还未提交，另一个事务这时也来操作这组数据，并且读取了前一个事务还未提交的数据，而前一个事务如果操作失败进行了回滚，后一个事务读取的就是错误数据，这样就造成了脏读。

ii. Non-Repeatable Reads 不可重复读

一个事务多次读取同一数据，在该事务还未结束时，另一个事务也对该数据进行了操作，而且在第一个事务两次次读取之间，第二个事务对数据进行了更新，那么第一个事务前后两次读取到的数据是不同的，这样就造成了不可重复读。

iii. Phantom Reads 幻像读

第一个数据正在查询符合某一条件的数据，这时，另一个事务又插入了一条符合条件的数据，第一个事务在第二次查询符合同一条件的数据时，发现多了一条前一次查询时没有的数据，仿佛幻觉一样，这就是幻像读。

iv. 非重复度和幻像读的区别

非重复读是指同一查询在同一事务中多次进行，由于其他提交事务所做的修改或删除，每次返回不同的结果集，此时发生非重复读。(A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data. )

幻像读是指同一查询在同一事务中多次进行，由于其他提交事务所做的插入操作，每次返回不同的结果集，此时发生幻像读。(A transaction reexecutes a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition. )

表面上看，区别就在于非重复读能看见其他事务提交的修改和删除，而幻像能看见其他事务提交的插入。

**Spring总共支持五种事务隔离级别设置**

1) DEFAULT （默认）

使用数据库默认的事务隔离级别.另外四个与JDBC的隔离级别相对应

3) READ\_UNCOMMITTED （读未提交）

这是事务最低的隔离级别，它允许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。

4) READ\_COMMITTED （读已提交）

保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻像读。

5) REPEATABLE\_READ （可重复读）

这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了不可重复读

6) SERIALIZABLE（串行化）

这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻像读。

## Hibernate与mybatis的区别

1.hibernate是全自动，而mybatis是半自动

hibernate完全可以通过对象关系模型实现对数据库的操作，拥有完整的JavaBean对象与数据库的映射结构来自动生成sql。而mybatis仅有基本的字段映射，对象数据以及对象实际关系仍然需要通过手写sql来实现和管理。

2.hibernate数据库移植性远大于mybatis

hibernate通过它强大的映射结构和hql语言，大大降低了对象与数据库（oracle、MySQL等）的耦合性，而mybatis由于需要手写sql，因此与数据库的耦合性直接取决于程序员写sql的方法，如果sql不具通用性而用了很多某数据库特性的sql语句的话，移植性也会随之降低很多，成本很高。

3. hibernate拥有完整的日志系统，mybatis则欠缺一些

hibernate日志系统非常健全，涉及广泛，包括：sql记录、关系异常、优化警告、缓存提示、脏数据警告等；而mybatis则除了基本记录功能外，功能薄弱很多。

4. mybatis相比hibernate需要关心很多细节

hibernate配置要比mybatis复杂的多，学习成本也比mybatis高。但也正因为mybatis使用简单，才导致它要比hibernate关心很多技术细节。mybatis由于不用考虑很多细节，开发模式上与传统jdbc区别很小，因此很容易上手并开发项目，但忽略细节会导致项目前期bug较多，因而开发出相对稳定的软件很慢，而开发出软件却很快。hibernate则正好与之相反。但是如果使用hibernate很熟练的话，实际上开发效率丝毫不差于甚至超越mybatis。

5. sql直接优化上，mybatis要比hibernate方便很多

由于mybatis的sql都是写在xml里，因此优化sql比hibernate方便很多。而hibernate的sql很多都是自动生成的，无法直接维护sql；虽有hql，但功能还是不及sql强大，见到报表等变态需求时，hql也歇菜，也就是说hql是有局限的；hibernate虽然也支持原生sql，但开发模式上却与orm不同，需要转换思维，因此使用上不是非常方便。总之写sql的灵活度上hibernate不及mybatis。

**总结：**

mybatis：小巧、方便、高效、简单、直接、半自动

hibernate：强大、方便、高效、复杂、绕弯子、全自动

## Mybatis

1. 入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

2. 可以进行更为细致的SQL优化，可以减少查询字段。

3. 缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

4. 二级缓存机制不佳。

## Hibernate

1. 功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码会相当简单，需要写的代码很少，开发速度很快，非常爽。

2. 有更好的二级缓存机制，可以使用第三方缓存。

3. 缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好

### 什么是hibernate懒加载

所谓懒加载(lazy)就是延时加载，延迟加载，load方法会导致懒加载出现，如果要数据及时送入数据库的话，使用save方法时，load会延迟至强制访问数据库执行，比如get方法调用的时候

### 为什么要懒加载

使用懒加载是解决了负载问题，也就是说懒加载后不会立即去访问数据库，而是尽量延迟，延迟到用户执行数据库查询命令的时候或者使用initialize方法的时候去和数据库交互，这样就减轻了与数据库交互的压力，如果缓存中有数据的话会优先考虑缓存数据，总之是尽量避免和数据库的直接交互，但有些时候也会出现数据没有得到及时更新的错误数据，所以，在金融等方面的项目开发的时候尽量避免懒加载。比如在统计论坛数据访问量等要求不是很严格的情况并且数据量比较大的情况下可以使用懒加载。

### Query接口的list方法和iterate方法有什么区别

对于Query接口的list()方法与iterate()方法来说，都可以实现获取查询的对象，但是list()方法返回的每个对象都是完整的（对象中的每个属性都被表中的字段填充上了），而iterator()方法所返回的对象中仅包含了主键值（标识符），只有当你对iterator中的对象进行操作时，Hibernate才会向数据库再次发送SQL语句来获取该对象的属性值

### Hibernate中get和load有什么不同之处

get和 load的最大区别是，如果在缓存中没有找到相应的对象，get将会直接访问数据库并返回一个完全初始化好的对象，而这个过程有可能会涉及到多个数据库调 用；而load方法在缓存中没有发现对象的情况下，只会返回一个代理对象，只有在对象getId()之外的其它方法被调用时才会真正去访问数据库，这样就 能在某些情况下大幅度提高性能。

### Hibernate中save、persist和saveOrUpdate这三个方法的不同之处

save()只能INSERT记录，但是saveOrUpdate()可以进行 记录的INSERT和UPDATE。还有，save()的返回值是一个Serializable对象，而persist()方法返回值为void

### Hibernate中的命名SQL查询指的是什么

命名查询指的是用<sql-query>标签在影射文档中定义的SQL查询，可以通过使用 Session.getNamedQuery()方法对它进行调用。命名查询使你可以使用你所指定的一个名字拿到某个特定的查询。 Hibernate中的命名查询可以使用注解来定义，也可以使用我前面提到的xml影射问句来定义。在Hibernate中，@NameQuery用来定 义单个的命名查询，@NameQueries用来定义多个命名查询。

### SessionFactory有什么作用? SessionFactory是线程安全的吗？

SessionFactory就是一个用于创建Hibernate的Session对象的工厂。 SessionFactory通常是在应用启动时创建好的，应用程序中的代码用它来获得Session对象。作为一个单个的数据存储，它也是 线程安全的， 所以多个线程可同时使用同一个SessionFactory。

### Session指的是什么? 可否将单个的Session在多个线程间进行共享？

Session代表着Hibernate所做的一小部分工作，它负责维护者同数据库的链接而且 不是线程安全的，也就是说，Hibernage中的Session不能在多个线程间进行共享。虽然Session会以主动滞后的方式获得数据库连接，但是Session最好还是在用完之后立即将其关闭。

### sorted collection和ordered collection有什么不同

sorted collection是通过使用 Java的Comparator在内存中进行排序的，ordered collection中的排序用的是数据库的order by子句。对于比较大的数据集，为了避免在内存中对它们进行排序而出现 Java中的OutOfMemoryError，最好使用ordered collection。

### 为什么在Hibernate的实体类中要提供一个无参数的构造器这一点非常重要

每个Hibernate实体类必须包含一个 无参数的构造器, 这是因为Hibernate框架要使用Reflection API，通过调用Class.newInstance()来创建这些实体类的实例。如果在实体类中找不到无参数的构造器，这个方法就会抛出一个InstantiationException异常。

### Hibernate与JDBC的区别

1、hibernate和jdbc主要区别就是，hibernate先检索缓存中的映射对象( 即hibernate操作的是对象)，而jdbc则是直接操作数据库.

2、Hibernate是JDBC的轻量级的对象封装，它是一个独立的对象持久层框架。Hibernate可以用在任何JDBC可以使用的场合

3、Hibernate是一个和JDBC密切关联的框架，所以Hibernate的兼容性和JDBC驱动，和数据库都有一定的关系，但是和使用它的Java程序，和App Server没有任何关系，也不存在兼容性问题。

4、如果正确的使用JDBC技术,它的执行效率一定比hibernate要好,因为hibernate是基于jdbc的技术.

5、JDBC使用的是SQL语句，Hibernate使用的是HQL语句，但是HQL语句最终还会隐式转换成SQL语句执行。

### Hibernate的三种状态以及状态的转换

**Transient(临时)**

new 一个初始化对象后，并没有在数据库里保存数据，处于临时状态；

**Persistent(持久化)**

当执行save()方法，调用session.close()方法之前，内存中的对象与数据库有对应关系处于持久化状态；

**Detached(托管/游离)**

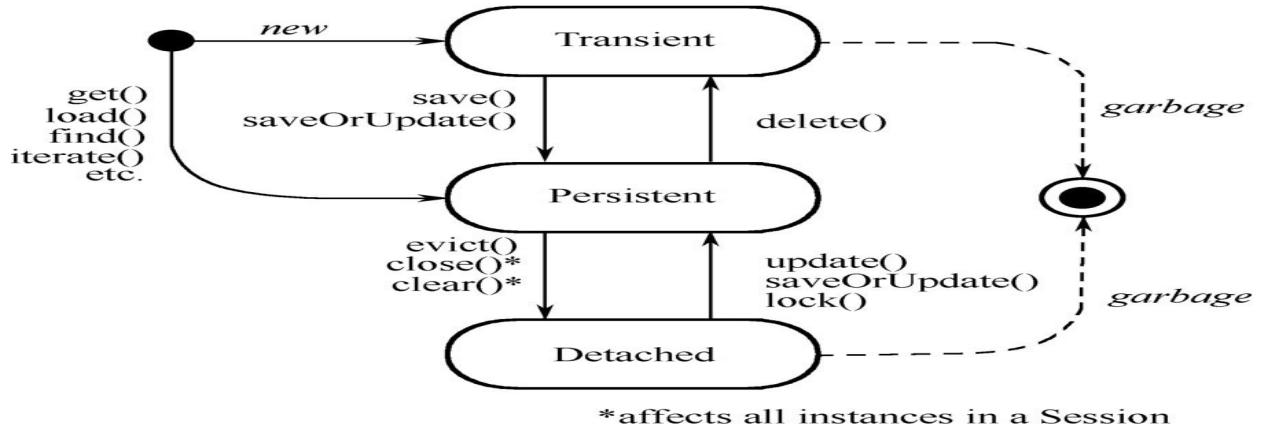
　　　当执行session.close()之后，处于托管状态；

**状态的转换**

处于托管状态下，调用update()方法后，转换为持久化状态；

在持久化状态下，执行delete()方法后，转换为临时状态；

在未初始化对象之前，调用get(),load(),find(),iterate()之后，直接进入持久化状态。



### Hibernate的batch\_size 和 fetch\_size 的区别

Fetch Size 是设定JDBC的Statement读取数据的时候每次从数据库中取出的记录条数。

例如一次查询1万条记录，对于Oracle的JDBC驱动来说，是不会1次性把1万条取出来的，而只会取出Fetch Size条数，当纪录集遍历完了这些记录以后，再去数据库取Fetch Size条数据。

因此大大节省了无谓的内存消耗。当然Fetch Size设的越大，读数据库的次数越少，速度越快；Fetch Size越小，读数据库的次数越多，速度越慢。

并不是所有的数据库都支持Fetch Size特性，例如MySQL就不支持。

Batch Size是设定对数据库进行批量删除，批量更新和批量插入的时候的批次大小，有点相当于设置Buffer缓冲区大小的意思。

## SpringBoot

这个springboot主要特点就是开发速度和启动速度非常快，他们可以使用maven和gradle进行项目的构建，也可以使用命令操作，我们使用的都是maven构建的他们里面有很多钟starter依赖，叫做启动器比如一个非常重要的web-starter,这是一个web项目启动器。还有持久层的jdbc-starter。当我们将他们依赖起来时springboot就会猜测到你想搭建的是什么类型的项目，并且把相关的依赖都给你组织好了。比如web项目就会把spring和springmvc和json转换还有tomcat这些依赖jar包全部准备好，我们启动项目时使用soringboot:run启动。他会在项目里找到唯一的一个main方法并运行。在main方法里面使用springAppcation来启动应用，这里面涉及到几个注解@compomentscan是进行扫描包的，@enableAutoconfig启用自动配置的，@RestController是@controller和@Responsbody的结合，会将返回的数据转成Json@RequestMapping依然是进行路由的2Enableschedule启用调度器注解扫描等等。springboot项目是可以脱离Xml配置的因为框架会根据你的POM依赖帮你准备好所有的东西。并使用了很多默认配置。如果你想改掉默认配置就可以在resources目录下创建application.properties属性文件，根据需要进行修改，这些初始值在springboot官网都有。springboot开发服务非常方便，通常不与页面技术集成，说白了就是非常适合开发接口。

# 数据库

## JDBC

### JDBC连接数据库步骤(以MYSQL为例)

1、加载JDBC驱动程序：

通过Class类的forName方法实现，并将驱动地址放进去，成功加载后，会将Driver类的实例注册到DriverManager类中。

2、提供JDBC连接的URL 、创建数据库的连接

•要连接数据库，需要向java.sql.DriverManager请求并获得Connection对象，该对象就代表一个数据库的连接。

•使用DriverManager的getConnectin()方法传入指定的欲连接的数据库的路径、数 据库的用户名和密码。

Connection con=DriverManager.getConnection(url , username , password);

&&&:"jdbc:mysql://localhost/test?user=root&password=123&useUnicode=true&characterEncoding=utf-8”;

3、创建一个Statement

•要执行SQL语句，必须获得java.sql.Statement实例

•执行静态SQL语句。通常通过Statement实例实现。

•执行动态SQL语句。通常通过PreparedStatement实例实现。

String sql = “”;

Statement st = con.createStatement() ;

PreparedStatement pst = con.prepareStatement(sql) ;

4、执行SQL语句

Statement接口提供了executeQuery、executeUpdate、execute三种方法

executeQuery：执行select语句，返回ResultSet结果集

ResultSet rst = pst.executeQuery();

• executeUpdate：执行insert、update、delete语句

pst.executeUpdate();

5、关闭JDBC对象

操作完成以后要把所有使用的JDBC对象全都关闭，以释放JDBC资源。

### Preparedstatement和statement的区别

用Prepared statement进行开发。Prepared statement是预编译的，而statement不是，在每次执行sql语句的增删改时，如果是一条数据两者没差距，但如果数据量大于1，那么每次执行sql语句statement都要重新编译一次，而Prepared statement不用，Prepared statement的运行效率大于statement；从代码的可维护性和可读性来说，虽然用Prepared statement来代替statement会使代码多出几行，但这样的代码无论从可读性还是可维护性来说，都比直接使用statement的代码高很多档次；最重要的一点，从安全角度来说，使用Prepared statement可以大大提高程序的安全性，因为Prepared statement是用‘？’传参,可以防止sql注入，具有安全性，而statement用的是‘+’字符串拼接，安全性较低。

## 数据库连接池

数据库连接池的优点运行原理:

在我们不使用数据库连接池的时候，每次访问数据库都需要创建连接，

使用完成之后需要释放关闭连接，而这样是很耗费资源的。当我们使用

数据库连接池的时候，在tomcat启动的时候就创建了指定数量的连接，

之后当我们程序使用的时候就直接从连接池里面取，而不需要创建，同理，

当我们使用完的时候也不需要关闭连接，而是将连接返回到连接池中，供

其他请求继续使用。

DBCP：比较稳定。

C3P0: 性能比较高。

## 数据库三范式

第一范式：数据库表中的所有字段值都是不可分解的原子值。

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）

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

## 数据库空值排序

【sqlserver】：

sqlserver 认为 null 最小。

升序排列：null 值默认排在最前。

要想排后面，则：order by case when col is null then 1 else 0 end ,col

降序排列：null 值默认排在最后。

要想排在前面，则：order by case when col is null then 0 else 1 end , col desc

【oracle】：

oracle认为 null 最大。

升序排列，默认情况下，null值排后面。

降序排序，默认情况下，null值排前面。

有几种办法改变这种情况：

（1）用 nvl 函数或decode 函数 将null转换为一特定值

（2）用case语法将null转换为一特定值（oracle9i以后版本支持。和sqlserver类似）：

order by (case mycol when null then '北京漂客' else mycol end)

（3）使用nulls first 或者nulls last 语法。

这是oracle专门用来null值排序的语法。

nulls first ：将null排在最前面。如：select \* from mytb order by mycol nulls first

nulls last ：将null排在最后面。如：select \* from mytb order by mycol nulls last

如果要想让含有null的列按照自己的意愿进行排序，可做如上处理。

【mysql】：

MySQL同sqlserver，null默认最小，解决办法同sqlserver

## Oracle

### Oracle的基本数据类型(常用)

1、字符型

Char 固定长度字符串 占2000个字节

Varchar2 可变长度字符串 占4000个字节

Nvarchar2 占2000个字符（最多能存2000个字母/中文）

2、大对象型（lob）

Blob ：二进制数据 最大长度4G

Blob 用于存一些图片，视频，文件。

比如：当我们在进行文件上传时，我们一般把上传的文件存在硬盘上，可以不占用 数据库，下载时，如果项目迁移时，文件也要跟着迁移。因此我们可以把用blob把它存在数据库中。但这样也增加了数据库的负担。

Clob ：字符数据 最大长度4G，可以存大字符串 varchar2和nvarchar2都具有一定的局限性，它们长度有限，但数据库中无论用varchar2或nvarchar2类型，还是用clob，在java端都使用String接收。

3、数值型

Integer 整数类型，小的整数。

Float 浮点数类型。

Real 实数类型。

Number（p,s）包含小数位的数值类型。P表示精度，s表示小数后的位数。

Eg: number(10,2) 表示小数点之前可有8位数字，小数点后有2位。

4、日期类型

Date 日期（日-月-年） DD-MM-YY(HH-MI-SS)

Timestamp 跟date比 它可以精确到微秒。精确范围0~9 默认为6.

### id、rowid、rownum的区别

rowid物理位置的唯一标识。

而id是逻辑上的唯一标识，所以rowid查找速度要快于id,是目前最快的 定位一条记录的方式

rowid和rownum都是"伪数列"

所谓“伪数列”也就是默认隐藏的一个数列。

rownum用于标记结果集中结果顺序的一个字段，它的特点是按顺序标记，而且是连续的，换句话说就是只有有rownum=1的记录，才可能有rownum=2的记录。

rownum关键字只能和<或者<=直接关联

如果是>或者=则需要给他起个别名

### 主键和唯一索引的区别

在创建主键的同时会生成对应的唯一索引，主键在保证数据唯一性的同时不允许为空，而唯一可以有一个为空数据项，一个表中只能有一个主键，但是一个主键可以有多个字段，一个表中可以有多个唯一索引。

## MySql

### MyISAM和InnoDB的区别

myISAM是MySQL的默认存储引擎，基于传统的ISAM类型，支持全文搜索，但不是事务安全的，而且不支持外键。每张MyISAM表存放在三个文件中：frm 文件存放表格定义；数据文件是MYD (MYData)；索引文件是MYI (MYIndex)。

InnoDB是事务型引擎，支持回滚、崩溃恢复能力、多版本并发控制、ACID事务，支持行级锁定（InnoDB表的行锁不是绝对的，如果在执行一个SQL语句时MySQL不能确定要扫描的范围，InnoDB表同样会锁全表，如like操作时的SQL语句），以及提供与Oracle类型一致的不加锁读取方式。InnoDB存储它的表和索引在一个表空间中，表空间可以包含数个文件。

**主要区别：**

• MyISAM是非事务安全型的，而InnoDB是事务安全型的。

• MyISAM锁的粒度是表级，而InnoDB支持行级锁定。

• MyISAM支持全文类型索引，而InnoDB不支持全文索引。

• MyISAM相对简单，所以在效率上要优于InnoDB，小型应用可以考虑使用MyISAM。

• MyISAM表是保存成文件的形式，在跨平台的数据转移中使用MyISAM存储会省去不少的麻烦。

• InnoDB表比MyISAM表更安全，可以在保证数据不会丢失的情况下，切换非事务表到事务表（alter table tablename type=innodb）。

**应用场景：**

• MyISAM管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的SELECT查询，那么MyISAM是更好的选择。

• InnoDB用于事务处理应用程序，具有众多特性，包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。

### MySQL最大连接数

mysql的最大连接数默认是100, 最大可以达到16384

修改MySQL配置文件my.ini 或my.cnf的参数max\_connections，将其改为max\_connections=1000，然后重启MySQL即可。但是有一点最难的就是my.ini这个文件在哪找。通常有两种可能，一个是在安装目录下（这是比较理想的情况），另一种是在数据文件的目录下，安装的时候如果没有人为改变目录的话，一般就在C:/ProgramData/MySQL往下的目录下;

## 事务

### 简介

事务指的是逻辑上的一组操作，这组操作要么全部发生，要么全部失败。

举例 ： 张三和李四 进行 转账的操作

张三向转账李四 1000元 张三余额-1000元 李四余额+1000元

不应该出现的是 在转账过程中由于一些意外，使张三的余额减去了1000元， 而李四并没有收到这笔钱。 使用事务来进行管理。 必须一起成功或者一起失败

### 事务特性（ACID）

**原子性**：是指事务一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。 不能够单独运行。 把一组操作放入事务中

**一致性**：事务的执行的前后，数据的完整性要得到保证。

**隔离性**：强调的是多个用户并发访问数据库的时候，一个用户事务不能被其他用户的事务所干扰到，多个并发事务之间的数据要相互隔离。 解决多个用户对同一个数据进行操作， 数据库会有 事务的 隔离级别来进行解决。

**持久性**:一个事务一旦被提交，它对数据库中数据的改变就是永久性，即使数据库发生故障也不应该对其有任何影响。

### 什么是脏数据，脏读，不可重复读，幻觉读

**脏读**: 指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据， 那么另外一个事务读到的这个数据是脏数据，依据脏数据所做的操作可能是不正确的。

**不可重复读**: 指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。 那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。

**幻觉读**: 指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，

以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。

### 数据库事务隔离级别

数据库事务的隔离级别有4个，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

√: 可能出现    ×: 不会出现

|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复读 | 幻读 |
| Read uncommitted | √ | √ | √ |
| Read committed | × | √ | √ |
| Repeatable read | × | × | √ |
| Serializable | × | × | × |

#### Read uncommitted 读未提交

公司发工资了，领导把5000元打到singo的账号上，但是该事务并未提交，而 singo正好去查看账户，发现工资已经到账，是5000元整，非常高兴。可是不幸的是，领导发现发给singo的工资金额不对，是2000元，于是迅速 回滚了事务，修改金额后，将事务提交，最后singo实际的工资只有2000元，singo空欢喜一场。

出现上述情况，即我们所说的脏读，两个并发的事务，“事务A：领导给singo发工资”、“事务B：singo查询工资账户”，事务B读取了事务A尚未提交的数据。

当隔离级别设置为Read uncommitted时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

#### Read committed 读提交

singo拿着工资卡去消费，系统读取到卡里确实有2000元，而此时她的老婆也正好 在网上转账，把singo工资卡的2000元转到另一账户，并在singo之前提交了事务，当singo扣款时，系统检查到singo的工资卡已经没有 钱，扣款失败，singo十分纳闷，明明卡里有钱，为何......

出现上述情况，即我们所说的不可重复读，两个并发的事务，“事务A：singo消费”、“事务B：singo的老婆网上转账”，事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

当隔离级别设置为Read committed时，避免了脏读，但是可能会造成不可重复读。

大多数数据库的默认级别就是Read committed，比如Sql Server , Oracle。如何解决不可重复读这一问题，请看下一个隔离级别。

#### Repeatable read 重复读

当隔离级别设置为Repeatable read时，可以避免不可重复读。当singo拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），singo的老婆就不可能对该记录进行修改，也就是singo的老婆不能在此时转账。

虽然Repeatable read避免了不可重复读，但还有可能出现幻读。

singo的老婆工作在银行部门，她时常通过银行内部系统查看singo的信用卡消费 记录。有一天，她正在查询到singo当月信用卡的总消费金额（select sum(amount) from transaction where month = 本月）为80元，而singo此时正好在外面胡吃海塞后在收银台买单，消费1000元，即新增了一条1000元的消费记录（insert transaction ... ），并提交了事务，随后singo的老婆将singo当月信用卡消费的明细打印到A4纸上，却发现消费总额为1080元，singo的老婆很诧异，以为出 现了幻觉，幻读就这样产生了。

#### Serializable 序列化

Serializable是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

# 各种优化

## 编码规范

我们会在编码之前制定统一编码规范，保持项目组内成员编码风格的一致性，同时，也会在规范中制定一些优化策略，比如代码中的sql语句尽量大写，查询时不要用select \* ,而是需要哪些字段，就查询哪些字段的信息。

Java代码的规范：

(1) 类名首字母应该大写。字段、方法以及对象（句柄）的首字母应小写。对于所有标识符，其中包含的所有单词都应紧靠在一起，而且大写中间单词的首字母。例如：

ThisIsAClassName

thisIsMethodOrFieldName

若在定义中出现了常数初始化字符，则大写static final基本类型标识符中的所有字母。这样便可标志出它们属于编译期的常数。

Java包（Package）属于一种特殊情况：它们全都是小写字母，即便中间的单词亦是如此。对于域名扩展名称，如com，org，net或者edu等，全部都应小写（这也是Java 1.1和Java 1.2的区别之一）。

(2) 为了常规用途而创建一个类时，请采取"经典形式"，并包含对下述元素的定义：

equals()

hashCode()

toString()

clone()（implement Cloneable）

implement Serializable

(3) 对于自己创建的每一个类，都考虑置入一个main()，其中包含了用于测试那个类的代码。为使用一个项目中的类，我们没必要删除测试代码。若进行了任何形式的改动，可方便地返回测试。这些代码也可作为如何使用类的一个示例使用。

(4) 应将方法设计成简要的、功能性单元，用它描述和实现一个不连续的类接口部分。理想情况下，方法应简明扼要。若长度很大，可考虑通过某种方式将其分割成较短的几个方法。这样做也便于类内代码的重复使用（有些时候，方法必须非常大，但它们仍应只做同样的一件事情）。

(5) 设计一个类时，请设身处地为客户程序员考虑一下（类的使用方法应该是非常明确的）。然后，再设身处地为管理代码的人考虑一下（预计有可能进行哪些形式的修改，想想用什么方法可把它们变得更简单）。

(6) 使类尽可能短小精悍，而且只解决一个特定的问题。下面是对类设计的一些建议：

■一个复杂的开关语句：考虑采用"多形"机制

■数量众多的方法涉及到类型差别极大的操作：考虑用几个类来分别实现

■许多成员变量在特征上有很大的差别：考虑使用几个类

(7) 让一切东西都尽可能地"私有"--private。可使库的某一部分"公共化"（一个方法、类或者一个字段等等），就永远不能把它拿出。若强行拿出，就可能破坏其他人现有的代码，使他们不得不重新编写和设计。若只公布自己必须公布的，就可放心大胆地改变其他任何东西。在多线程环境中，隐私是特别重要的一个因素--只有private字段才能在非同步使用的情况下受到保护。

(8) 谨惕"巨大对象综合症"。对一些习惯于顺序编程思维、且初涉OOP领域的新手，往往喜欢先写一个顺序执行的程序，再把它嵌入一个或两个巨大的对象里。根据编程原理，对象表达的应该是应用程序的概念，而非应用程序本身。

(9) 若不得已进行一些不太雅观的编程，至少应该把那些代码置于一个类的内部。

(10) 任何时候只要发现类与类之间结合得非常紧密，就需要考虑是否采用内部类，从而改善编码及维护工作（参见第14章14.1.2小节的"用内部类改进代码"）。

(11) 尽可能细致地加上注释，并用javadoc注释文档语法生成自己的程序文档。

(12) 避免使用"魔术数字"，这些数字很难与代码很好地配合。如以后需要修改它，无疑会成为一场噩梦，因为根本不知道"100"到底是指"数组大小"还是"其他全然不同的东西"。所以，我们应创建一个常数，并为其使用具有说服力的描述性名称，并在整个程序中都采用常数标识符。这样可使程序更易理解以及更易维护。

(13) 涉及构建器和异常的时候，通常希望重新丢弃在构建器中捕获的任何异常--如果它造成了那个对象的创建失败。这样一来，调用者就不会以为那个对象已正确地创建，从而盲目地继续。

(14) 当客户程序员用完对象以后，若你的类要求进行任何清除工作，可考虑将清除代码置于一个良好定义的方法里，采用类似于cleanup()这样的名字，明确表明自己的用途。除此以外，可在类内放置一个boolean（布尔）标记，指出对象是否已被清除。在类的finalize()方法里，请确定对象已被清除，并已丢弃了从RuntimeException继承的一个类（如果还没有的话），从而指出一个编程错误。在采取象这样的方案之前，请确定finalize()能够在自己的系统中工作（可能需要调用System.runFinalizersOnExit(true)，从而确保这一行为）。

(15) 在一个特定的作用域内，若一个对象必须清除（非由垃圾收集机制处理），请采用下述方法：初始化对象；若成功，则立即进入一个含有finally从句的try块，开始清除工作。

(16) 若在初始化过程中需要覆盖（取消）finalize()，请记住调用super.finalize()（若Object属于我们的直接超类，则无此必要）。在对finalize()进行覆盖的过程中，对super.finalize()的调用应属于最后一个行动，而不应是第一个行动，这样可确保在需要基础类组件的时候它们依然有效。

(17) 创建大小固定的对象集合时，请将它们传输至一个数组（若准备从一个方法里返回这个集合，更应如此操作）。这样一来，我们就可享受到数组在编译期进行类型检查的好处。此外，为使用它们，数组的接收者也许并不需要将对象"造型"到数组里。

(18) 尽量使用interfaces，不要使用abstract类。若已知某样东西准备成为一个基础类，那么第一个选择应是将其变成一个interface（接口）。只有在不得不使用方法定义或者成员变量的时候，才需要将其变成一个abstract（抽象）类。接口主要描述了客户希望做什么事情，而一个类则致力于（或允许）具体的实施细节。

## 代码优化

所谓代码优化是指对程序代码进行等价(指不改变程序的运行结果)变换。程序代码可以是中间代码(如四元式代码)，也可以是目标代码。等价的含义是使得变换后的代码运行结果与变换前代码运行结果相同。优化的含义是最终生成的目标代码短(运行时间更短、占用空间更小)，时空效率优化。原则上，优化可以在编译的各个阶段进行，但最主要的一类是对中间代码进行优化，这类优化不依赖于具体的计算机。

在不改变程序运行效果的前提下，对被编译的程序进行等价变换，使之能生成更加高效的目标代码。

### 后台代码优化

对字符串操作时尽量用stringBuffer,而不是String；使用某些资源时，要记得及时释放，比如io流；我们会采用数据库连接池，降低与数据库连接的频率。

现在对于代码的优化很多不是性能上的优化，而是可读性方面的优化。但是如果能写出可读性好并且效率高的代码确实是一件令人兴奋的事情。

1、尽最大努力少New新对象，新对象占用的可是宝贵的内存空间。

2、字符串的拼接是一件非常浪费资源的事情，所以如果非要对字符串进行拼接的话，使用stringbuffer而不用string + 这种形式。

3、对于稀缺资源的使用问题，像流、数据库连接这种稀缺资源使用后一定要关闭

4、多看数据结构、和算法方面的书，很多非常复杂的算法，前辈们都已经提供了非常简单的算法实现，如果这些东西自己编写程序实现那是一件非常繁琐并且枯燥的事情，可想效率也不一定很好。

5、如果对于线程安全性没有要求的话，尽量使用线程不安全的类，因为一个类实现线程安全，也就是线程同步，这个是非常耗费系统资源的做法。所以对于线程安全没有要求的情况下，不用线程安全的类。

6、避免无用的空字符串：

如为了避免NullPointerException，新手常把字符串初始化为空字符串，如：

String s = "";

其实这是非常不好的做法，白白创建一个无用的字符串对象，占用资源。

7、及时清除不再需要的会话：

为了清除不再活动的会话，许多应用服务器都有默认的会话超时时间，一般为30分钟。当应用服务器需要保存更多会话时，如果内存容量不足，操作系统会把部分内存数据转移到磁盘，应用服务器也可能根据“最近最频繁使用”（Most Recently Used）算法把部分不活跃的会话转储到磁盘，甚至可能抛出“内存不足”异常。在大规模系统中，串行化会话的代价是很昂贵的。当会话不再需要时，应当及时调用HttpSession.invalidate()方法清除会话。HttpSession.invalidate()方法通常可以在应用的退出页面调用。

8、尽量使用局部变量，调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快。其他变量，如静态变量、实例变量等，都在堆（Heap）中创建，速度较慢。另外，依赖于具体的编译器/JVM，局部变量还可能得到进一步优化。

9、我们会采用数据库连接池，降低与数据库连接的频率。

### 前台代码优化

#### Css文件

1 Css引用应该放在<head>标签中，并保证每个页面引用唯一的css

2 避免1个频道所有页面公用1个css文件，以免发生公用class过多难以修改和删除，造成的css不断增大以及小页面引用超大css的问题

#### Javascript

1 将js引用和执行放在页面尽量靠下的位置

2 Javascript函数、判断、循环要封装到js中，html中只允许简单的函数调用

3 Javascript代码执行效率的优化(足够优化的代码执行速度和CPU使用能有10倍甚至更高的降低)

4 同一页面，特定参数需要运行的js，只在特定参数的情况下运行

#### Html

1 尽可能的减少div嵌套，合理的应用dt dd li等元素

2 使整体的html代码量尽量的小，在和同行业同页面相比无特殊功能和区块的情况下，总字节数不可高于同行业同页面的10%

3 巨大的包含数据的html（如产品下拉框），数据应封装到js中减少html代码量并充分利用缓存

4 虽然html要减少，但为了SEO（搜索引擎优化） 链接仍然要使用绝对路径

5 尽量减少使用iframe

6 禁用ViewState

#### Cookie，由于Cookie会随着每次http请求往返于服务器和客户浏览器之间，增加网络流量开销，降低网友访问速度

1 Cookie要尽量少用。

2 使用的时候要使Cookie尽量的小

3 设定Cookie过期时间要尽量的短且合理

4 将Cookie精准的写在使用它的2级域名下，尽可能的不去写根域的Cookie

5 样式、js、图片等不需要Cookie的文件分离到不携带Cookie的域名上，如

ico.pcpop.com.cn img.pcpop.com.cn

#### 减少前台http请求数量

1 页面http总请求数量不应高于最优竞争对手的同页面的10%

2 最大化的css sprite 合并样式小图，每页面调用样式图片的总数不超过3个

3 尽量将零散的js合并，“页面自用”的js总数控制在5个以内(1个jquery或其他工具引用、1个公用js、1个页面专用js、2个其他功能js)

4 尽量减少页面ajax的请求次数，将功能相近可以合并的ajax请求合并

#### 缓存优化

1 “所有可以缓存的资源”都必须在HttpHead中对过期时间进行设置

2 Ajax要特别注意对是否可缓存的资源区分，可缓存的ajax请求的get参数中要加入正确的时间戳以便CDN可以正确缓存

3 不能缓存的ajax接口应从可被CDN缓存的域名中分离出来。

## 业务优化

我们做项目的时候业务优化这方面最主要是从用户体验度角度进行考虑,减少用户操作的步骤提高工作效率，通常有以下几种：

　　1.可以通过tabindex属性来改变tab键盘的操作顺序

　　2.可以通过回车键来进行搜索或者提交操作

　　3.对于单选按钮和复选按钮可以通过操作后面的文本来选择前面的单选按钮以及复选按钮

　　4.添加的信息要按照id倒序进行排列

　　5.进行搜索操作时加入js loading操作（不仅告诉用户所进行的请求正在被处理，而且防止用户多次点击提交操作）

　　6.当进行删除操作的时候要弹出提示框，警告用户要进行删除操作，是否确认。

　　7.根据returnURL在用户登录成功后直接跳到想要访问的资源。

　　8.进行删除操作时通过confirm提示用户是否确认删除操作，操作完后提示操作是否成功。

　　9.减少用户操作的步骤

10.使用autocomplete插件快速进行搜索

## MySql优化

### 索引优化

#### 什么是主键？什么是索引？

主键是表中的一个或多个字段，它的值用于惟一地标识表中的某一条记录.;使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构，只有当经常查询索引列中的数据时，才需要在表上创建索引。索引占用磁盘空间，并且降低添加、删除和更新行的速度。当然索引也有好处就是查询速度快，它利还是大于弊的所以请慎重使用索引。

#### ****创建索引****

对于查询占主要的应用来说，索引显得尤为重要。很多时候性能问题很简单的就是因为我们忘了添加索 引而造成的，或者说没有添加更为有效的索引导致。如果不加索引的话，那么查找任何哪怕只是一条特定的数据都会进行一次全表扫描，如果一张表的数据量很大而 符合条件的结果又很少，那么不加索引会引起致命的性能下降。但是也不是什么情况都非得建索引不可，比如性别可能就只有两个值，建索引不仅没什么优势，还会 影响到更新速度，这被称为过度索引。

#### ****复合索引****

比如有一条语句是这样的：select \* from users where area='beijing' and age=22;如 果我们是在area和age上分别创建单个索引的话，由于mysql查询每次只能使用一个索引，所以虽然这样已经相对不做索引时全表扫描提高了很多效率， 但是如果在area、age两列上创建复合索引的话将带来更高的效率。如果我们创建了(area, age, salary)的复合索引，那么其实相当于创建了(area,age,salary)、(area,age)、(area)三个索引，这被称为最佳左前缀 特性。因此我们在创建复合索引时应该将最常用作限制条件的列放在最左边，依次递减。

#### ****索引不会包含有NULL值的列****

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

#### ****使用短索引****

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的 列，如果在前10 个或20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

#### ****排序的索引问题****

mysql查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

#### ****like语句操作****

一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 “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

#### ****不要在列上进行运算****

select \* from users where YEAR(adddate)<2007;

where子句使用了Mysql函数的时候，索引将无效，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成

select \* from users where adddate<‘2007-01-01';

#### ****不使用NOT IN和<>操作****

NOT IN和<>操作都不会使用索引将进行全表扫描。NOT IN可以NOT EXISTS代替，id<>3则可使用id>3 or id<3来代替。

#### 聚集索引与非聚集索引有什么不同？

聚集索引本身就包含表，并且表中的数据按照索引字段进行排序 这就说明聚集索引可以直接定位到目标数据 一个表也只能有一个聚集索引 适用场合：选择数据表中最常用 最频繁查询的字段 一般讲表的主键设置为聚集索引

非聚集索引不含表 他是通过记录地址定位到表中的数据（所以聚集索引比非聚集索引更快） 默认建立的索引都是非聚集索引 非聚集索引可以建立多个（一个表可以建立249个 超够用了）

### 函数和存储过程的区别

#### 存储过程

什么是存储过程：存储过程可以说是一个记录集吧，它是由一些T-SQL语句组成的代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。

#### 存储过程的好处

1.由于数据库执行动作时，是先编译后执行的。然而存储过程是一个编译过的代码块，所以执行效率要比T-SQL语句高。

2.一个存储过程在程序在网络中交互时可以替代大堆的T-SQL语句，所以也能降低网络的通信量，提高通信速率。

3.通过存储过程能够使没有权限的用户在控制之下间接地存取数据库，从而确保数据的安全。

#### 存储过程与函数的区别

存储过程和函数目的是为了 可重复地 执行操作数据库的sql语句的集合。

区别是写法和调用上。

写法上：存储过程的参数列表可以有输入参数、输出参数、可输入输出的参数；

函数的参数列表只有输入参数，并且有return <返回值类型，无长度说明>。

返回值上：

　　　　存储过程的返回值，可以有多个值，

函数的返回值，只有一个值。

调用方式上：

　　　　存储过程的调用方式有：

1)、exec <过程名>;

2)、execute <过程名>;

3)、在PL/SQL语句块中直接调用。

函数的调用方式有：

在PL/SQL语句块中直接调用。

### 触发器

触发器在数据库里以独立的对象存储，它与存储过程和函数不同的是，存储过程与函数需要用户显示调用才执行，而触发器是由一个事件来启动运行。即触发器是当某个事件发生时自动地隐式运行。并且，触发器不能接收参数

### View（数据库视图）

#### 视图的作用

视图可以视为“虚拟表”或“存储的查询”

创建视图所依据的表称为“基表”

#### 视图的优点

1.提供了另外一种级别的表安全性:隐藏了一些关键的字段

2.简化的用户的SQL命令

3.隔离基表结构的改变

### 防SQL注入

针对防sql注入，我们通常是这样做的：

首先在前台页面对用户输入信息进行js验证，对一些特殊字符进行屏蔽，比如：or ,单引号，--，= ，还有就是限制用户名输入的长度，我们一般将其限制在6---13位。另外，对于用户的敏感信息我们进行Md5加密，还有为了增加用户体验度和用户友好度，为了不使用户看到一些详细的异常信息我们会进行错误信息页面的定制，像404,500错误。另一个我层面讲，这样做也是为了保护我们的一些重要信息。此外，我们会给特定的人分配定定的权限，而不是给其分配管理员权限！

### 模拟场景

（业务背景，一张实时数据表，存储近百家电厂发电机组的负荷数据（还有其它指标数据），每台机组5分钟生成一条记录，到目前为止已有六百多万条记录。

需求：哪些机组跳机了？（满足条件：前面一次负荷＞＝10％，后面连续三次＜1％（满负荷是100％）

按照条件写出SQL语句，然后一执行---卡成翔了?！

**第一次优化：加索引**

在网上搜索各种优化方案，感觉加索引比较靠谱（索引类似于字典目录，按照字母顺序可以快速定位），于是就加上了，性能有明显改善，但是在查所有机组时，又卡成翔了?！

分析原因：把几百万条数据按照时间排一下序，光这一点就特别耗时，于是想到了下面的优化。

**第二次优化：限定时间段**

只查询出最近10分钟的数据（每5分钟一次，时间有延迟，适当放宽），这样一来按理说，数据量已经大大减少，应该快很快，一测试，果然快多了，基本可接受。

剧情往往要一点波折，否则就没有第三次优化了，周一一来，又卡成翔了。

分析原因：还是数据量太大，第二次虽然减少了排序时间，但依然没有减少过滤时间，但是为什么当时比较快的一下就出来，可能是那台机组数据量小，或者最近一直没有最新数据，取最近10分钟也不太精确。

**第三次优化：空间换时间**

问题的关键是数据量太大，加上算法复杂，需要很多次在六百多万数据中查找数据。回头看看问题，发现只需每个机组的最近4条记录。于是一种思路出来了，把每台机组的最近4条单独存在一个表中，粗算下来，不到1000条数据，于是数据量瞬间减少了6000倍。最后用触发器实现了存储最新记录功能，然后从不到1000条记录中查询，速度立马就上来了）

一、注释使用：在语句中多写注释，注释不影响SQL语句的执行效率。增加代码的可读性；

二、对于事务的使用：尽量使事务处理达到最短，如果事务太长最好按功能将事务分开执行(如：可以让用户在界面上多几步操作)。事务太长很容易造成数据库阻塞，用户操作速度变慢或死机情况；

三、对于与数据库的交互：尽量减少与数据库的交互次数。如果在前端程序写有循球访问数据库操作，最好写成将数据一次读到前端再进行处理或者写成存储过程在数据库端直接处理；

四、对于SELECT \*这样的语句：不要使用SELECT \*这样的语句，而应该使用SELECT table1.column1这样的语句，明确指出要查询的列减少数据的通讯量并且这样的代码可读性好，便于维护；

五、尽量避免使用游标：它占用大量的资源。如果需要row-by-row地执行，尽量采用非光标技术,如：在客户端循环，用临时表，Table变量，用子查询，用Case语句等等。如果使用了游标，就要尽量避免在游标循环中再进行表连接的操作；

六、尽量使用count(1)：count函数只有在统计表中所有行数时使用，而且count(1)比count(\*)更有效率；

七、IN和EXISTS：EXISTS要远比IN的效率高。里面关系到full table scan和range scan。几乎将所有的IN操作符子查询改写为使用EXISTS的子查询；

注意表之间连接的数据类型：避免不同类型数据之间的连接；

八、尽量少用视图：对视图操作比直接对表操作慢,可以用stored procedure来代替她。特别的是不要用视图嵌套,嵌套视图增加了寻找原始资料的难度。我们看视图的本质：它是存放在服务器上的被优化好了的已经产生了查询规划的SQL。对单个表检索数据时，不要使用指向多个表的视图，直接从表检索或者仅仅包含这个表的视图上读，否则增加了不必要的开销,查询受到干扰；

没有必要时不要用DISTINCT和ORDER BY：这些动作可以改在客户端执行，它们增加了额外的开销；

九、避免相关子查询：一个列的标签同时在主查询和where子句中的查询中出现，那么很可能当主查询中的列值改变之后，子查询必须重新查询一次。查询嵌套层次越多，效率越低，因此应当尽量避免子查询。如果子查询不可避免，那么要在子查询中过滤掉尽可能多的行；

十、代码离数据越近越好：所以优先选择Default,依次为Rules,Triggers, Constraint(约束如外健主健CheckUNIQUE……,数据类型的最大长度等等都是约束),Procedure.这样不仅维护工作小，编写程序质量高，并且执行的速度快；

十一、插入大的二进制值到Image列：使用存储过程，千万不要用内嵌Insert来插入。因为这样应用程序首先将二进制值转换成字符串(尺寸是它的两倍)，服务器受到字符后又将他转换成二进制值.存储过程就没有这些动作：方法：

Create procedure p\_insert as insert into table(Fimage) values (@image), 在前台调用这个存储过程传入二进制参数，这样处理速度明显改善；

十二、Between在某些时候比IN 速度更快：Between能够更快地根据索引找到范围。用查询优化器可见到差别。 select \* from chineseresume where title in ('男','女') Select \* from chineseresume where between '男' and '女' 是一样的。由于in会在比较多次，所以有时会慢些；

十三、对Where条件字段修饰字段移到右边：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；

在海量查询时尽量少用格式转换；

十四、IS NULL 与 IS NOT NULL：不能用null作索引，任何包含null值的列都将不会被包含在索引中。即使索引有多列这样的情况下，只要这些列中有一列含有null，该列就会从索引中排除。也就是说如果某列存在空值，即使对该列建索引也不会提高性能。任何在where子句中使用is null或is not null的语句优化器是不允许使用索引的；

十五、建立临时表：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log，提高速度;如果数据量不大，为了缓和系统表的资源，建议先create table，然后insert。临时表是tempdb数据库实际的表，没有主键、索引，应该避免在临时表中存储大量的数据；

Where中索引的使用：WHERE条件顺序尽量把索引字段放在前面（主键的唯一性最高），复合索引字段顺序与where条件顺序保持一致。Sql自动查找使用那个索引；

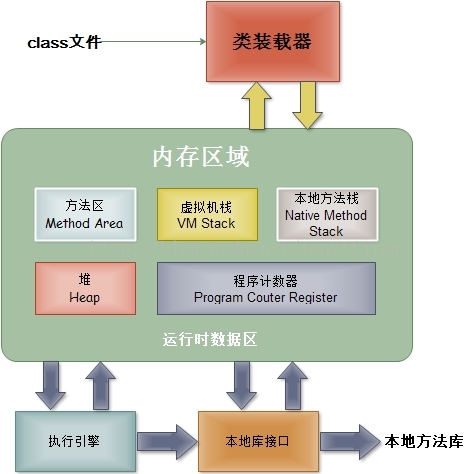
十六、外键关联的列应该建立索引：(如子表id)主子表单据肯定要建视图，2个表的关联以2个表中的MainID为关系，所以，需要给子表的MainID单独建索引，这将很大地提高视图的速度。例如Gy\_InOutSub中的InoutMainid增加索引；

十七、注意UNion和`UNion all 的区别：UNION all执行效率高；

十八、Insert：Insert into 表 values() 应该为 Insert into 表 (字段) values()；

十九、order by语句：ORDER BY语句决定了如何将返回的查询结果排序。Order by语句对要排序的列没有什么特别的限制，也可以将函数加入列中（象联接或者附加等）。任何在Order by语句的非索引项或者有计算表达式都将降低查询速度。 仔细检查order by语句以找出非索引项或者表达式，它们会降低性能。解决这个问题的办法就是重写order by语句以使用索引，也可以为所使用的列建立另外一个索引，同时应绝对避免在order by子句中使用表达式。

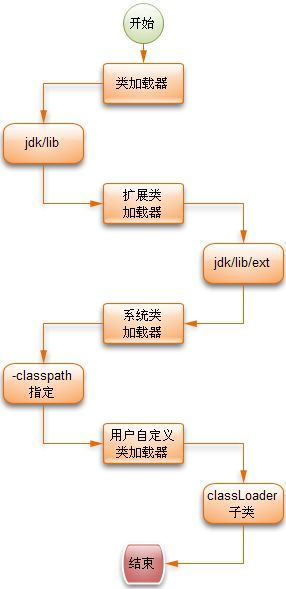
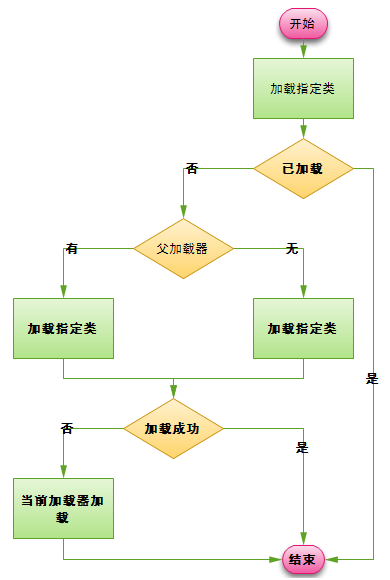
## JVM优化



### 类加载器（ClassLoader）

在JVM启动时或者在类运行时将需要的class加载到JVM中。

#### 类加载顺序

1、启动jvm调用loadClass。classloader加载类的入口，此方法负责加载指定名字的类，ClassLoader的实现方法为先从已经加载的类中寻找，如没有则继续从父ClassLoader中寻找，如仍然没找到，则从BootstrapClassLoader中寻找（BootStrapClassLoader。它是最顶层的类加载器，是由C++编写而成, 已经内嵌到JVM中了。在JVM启动时会初始化该ClassLoader，它主要用来读取Java的核心类库JRE/lib/rt.jar中所有的class文件，这个jar文件中包含了java规范定义的所有接口及实现。）这个过程如下图：

2、ExtensionClassLoader。它是用来读取Java的一些扩展类库，如读取JRE/lib/ext/\*.jar中的包等

3、AppClassLoader。它是用来读取CLASSPATH下指定的所有jar包或目录的类文件，一般情况下这个就是程序中默认的类加载器。

4、CustomClassLoader。它是用户自定义编写的，它用来读取指定类文件 。基于自定义的ClassLoader可用于加载非Classpath中（如从网络上下载的jar或二进制）的jar及目录、还可以在加载前对class文件优一些动作，如解密、编码等。

#### 双亲委派机制描述

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，（每个ClassLoader实例都有一个父类加载器的引用【不是继承的关系，是一个包含的关系】）依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

**java类加载为什么采用双亲委派模型**

如类java.lang.Object，它存放在rt.jar中，无论哪个类加载器要加载这个类，最终都会委派给启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果用户自己写了一个名为java.lang.Object的类，并放在程序的Classpath中，那系统中将会出现多个不同的Object类，java类型体系中最基础的行为也无法保证，应用程序也会变得一片混乱。

### 栈

Java栈是与每一个线程关联的，JVM在创建每一个线程的时候，会分配一定的栈空间给线程。存储局部变量、引用、方法、返回值等。

**StackOverflowError：**如果在线程执行的过程中，栈空间不够用，那么JVM就会抛出此异常，这种情况一般是死递归造成的。

### 堆

JVM堆内存分为2块：Permanent Space 和 Heap Space。

Permanent 即 持久代（Permanent Generation），主要存放的是Java类定义信息，与垃圾收集器要收集的Java对象关系不大。

Heap = { Old + NEW = {Eden, from, to} }，Old 即 年老代（Old Generation），New 即 年轻代（Young Generation）。年老代和年轻代的划分对垃圾收集影响比较大。

#### 年轻代

所有新生成的对象首先都是放在年轻代。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代一般分3个区，1个Eden区，2个Survivor区（from 和 to）。

大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当一个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当另一个Survivor区也满了的时候，从前一个Survivor区复制过来的并且此时还存活的对象，将可能被复制到年老代。

2个Survivor区是对称的，没有先后关系，所以同一个Survivor区中可能同时存在从Eden区复制过来对象，和从另一个Survivor区复制过来的对象；而复制到年老区的只有从另一个Survivor区过来的对象。而且，因为需要交换的原因，Survivor区至少有一个是空的。特殊的情况下，根据程序需要，Survivor区是可以配置为多个的（多于2个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

针对年轻代的垃圾回收即 **Young GC**。

#### 年老代

在年轻代中经历了N次（可配置）垃圾回收后仍然存活的对象，就会被复制到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。一般如果系统中用了application级别的缓存，缓存中的对象往往会被转移到这一区间。

针对年老代的垃圾回收即 **Full GC**。

#### 持久代

用于存放静态类型数据，如 Java Class, Method 等。持久代对垃圾回收没有显著影响。但是有些应用可能动态生成或调用一些Class，例如 hibernate CGLib 等，在这种时候往往需要设置一个比较大的持久代空间来存放这些运行过程中动态增加的类型。

#### 一组对象生成时，内存申请流程

1.JVM会试图为相关Java对象在年轻代的Eden区中初始化一块内存区域。

2.当Eden区空间足够时，内存申请结束。否则执行下一步。

3.JVM试图释放在Eden区中所有不活跃的对象（Young GC）。释放后若Eden空间仍然不足以放入新对象，JVM则试图将部分Eden区中活跃对象放入Survivor区。

4.Survivor区被用来作为Eden区及年老代的中间交换区域。当年老代空间足够时，Survivor区中存活了一定次数的对象会被移到年老代。

5.当年老代空间不够时，JVM会在年老代进行完全的垃圾回收（Full GC）。

6.Full GC后，若Survivor区及年老代仍然无法存放从Eden区复制过来的对象，则会导致JVM无法在Eden区为新生成的对象申请内存，即出现“Out of Memory”。

#### OOM（“Out of Memory”）异常

一般主要有如下2种原因：

**1. 年老代溢出，表现为：java.lang.OutOfMemoryError:Javaheapspace**

这是最常见的情况，产生的原因可能是：设置的内存参数Xmx过小或程序的内存泄露及使用不当问题。

例如循环上万次的字符串处理、创建上千万个对象、在一段代码内申请上百M甚至上G的内存。还有的时候虽然不会报内存溢出，却会使系统不间断的垃圾回收，也无法处理其它请求。这种情况下除了检查程序、打印堆内存等方法排查，还可以借助一些内存分析工具，比如MAT就很不错。

**2. 持久代溢出，表现为：java.lang.OutOfMemoryError:PermGenspace**

通常由于持久代设置过小，动态加载了大量Java类而导致溢出，解决办法唯有将参数 -XX:MaxPermSize 调大（一般256m能满足绝大多数应用程序需求）。将部分Java类放到容器共享区（例如Tomcat share lib）去加载的办法也是一个思路，但前提是容器里部署了多个应用，且这些应用有大量的共享类库。

#### 参数说明

-Xmx3550m：设置JVM最大堆内存为3550M。

-Xms3550m：设置JVM初始堆内存为3550M。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

-Xss128k：设置每个线程的栈大小。JDK5.0以后每个线程栈大小为1M，之前每个线程栈大小为256K。应当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。需要注意的是：当这个值被设置的较大（例如>2MB）时将会在很大程度上降低系统的性能。

-Xmn2g：设置年轻代大小为2G。在整个堆内存大小确定的情况下，增大年轻代将会减小年老代，反之亦然。此值关系到JVM垃圾回收，对系统性能影响较大，官方推荐配置为整个堆大小的3/8。

-XX:NewSize=1024m：设置年轻代初始值为1024M。

-XX:MaxNewSize=1024m：设置年轻代最大值为1024M。

-XX:PermSize=256m：设置持久代初始值为256M。

-XX:MaxPermSize=256m：设置持久代最大值为256M。

-XX:NewRatio=4：设置年轻代（包括1个Eden和2个Survivor区）与年老代的比值。表示年轻代比年老代为1:4。

-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的比值。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2:4，即1个Survivor区占整个年轻代大小的1/6。

-XX:MaxTenuringThreshold=7：表示一个对象如果在Survivor区（救助空间）移动了7次还没有被垃圾回收就进入年老代。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代，对于需要大量常驻内存的应用，这样做可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代存活时间，增加对象在年轻代被垃圾回收的概率，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。

#### 疑问解答

-Xmn，-XX:NewSize/-XX:MaxNewSize，-XX:NewRatio 3组参数都可以影响年轻代的大小，混合使用的情况下，优先级是什么？

**如下：**

高优先级：-XX:NewSize/-XX:MaxNewSize

中优先级：-Xmn（默认等效 -Xmn=-XX:NewSize=-XX:MaxNewSize=?）

低优先级：-XX:NewRatio

推荐使用-Xmn参数，原因是这个参数简洁，相当于一次设定 NewSize/MaxNewSIze，而且两者相等，适用于生产环境。-Xmn 配合 -Xms/-Xmx，即可将堆内存布局完成。

-Xmn参数是在JDK 1.4 开始支持。

### 栈与堆的比较

栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。

对象存放在堆中，对象引用和基本类型存放在栈中；对象的属性存放在堆中，对象的方法存在栈中。

程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

在JVM中，静态属性保存在Stack指令内存区，动态属性保存在Heap数据内存区

### 垃圾回收器选择

JVM给出了3种选择：串行收集器、并行收集器、并发收集器。串行收集器只适用于小数据量的情况，所以生产环境的选择主要是并行收集器和并发收集器。

默认情况下JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置进行智能判断。

#### 串行收集器

-XX:+UseSerialGC：设置串行收集器。

#### 并行收集器（吞吐量优先）

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:ParallelGCThreads=20：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:+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:MaxGCPauseMillis=100：设置每次年轻代垃圾回收的最长时间（单位毫秒）。如果无法满足此时间，JVM会自动调整年轻代大小，以满足此时间。

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动调整年轻代Eden区大小和Survivor区大小的比例，以达成目标系统规定的最低响应时间或者收集频率等指标。此参数建议在使用并行收集器时，一直打开。

#### 并发收集器（响应时间优先）

-XX:+UseConcMarkSweepGC：即CMS收集，设置年老代为并发收集。CMS收集是JDK1.4后期版本开始引入的新GC算法。它的主要适合场景是对响应时间的重要性需求大于对吞吐量的需求，能够承受垃圾回收线程和应用线程共享CPU资源，并且应用中存在比较多的长生命周期对象。CMS收集的目标是尽量减少应用的暂停时间，减少Full GC发生的几率，利用和应用程序线程并发的垃圾回收线程来标记清除年老代内存。

-XX:+UseParNewGC：设置年轻代为并发收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此参数。

-XX:CMSFullGCsBeforeCompaction=0：由于并发收集器不对内存空间进行压缩和整理，所以运行一段时间并行收集以后会产生内存碎片，内存使用效率降低。此参数设置运行0次Full GC后对内存空间进行压缩和整理，即每次Full GC后立刻开始压缩和整理内存。

-XX:+UseCMSCompactAtFullCollection：打开内存空间的压缩和整理，在Full GC后执行。可能会影响性能，但可以消除内存碎片。

-XX:+CMSIncrementalMode：设置为增量收集模式。一般适用于单CPU情况。

-XX:CMSInitiatingOccupancyFraction=70：表示年老代内存空间使用到70%时就开始执行CMS收集，以确保年老代有足够的空间接纳来自年轻代的对象，避免Full GC的发生。

#### 其他垃圾回收参数

-XX:+ScavengeBeforeFullGC：年轻代GC优于Full GC执行。

-XX:-DisableExplicitGC：不响应 System.gc() 代码。

-XX:+UseThreadPriorities：启用本地线程优先级API。即使 java.lang.Thread.setPriority() 生效，不启用则无效。

-XX:SoftRefLRUPolicyMSPerMB=0：软引用对象在最后一次被访问后能存活0毫秒（JVM默认为1000毫秒）。

-XX:TargetSurvivorRatio=90：允许90%的Survivor区被占用（JVM默认为50%）。提高对于Survivor区的使用率。

#### 辅助信息参数设置

-XX:-CITime：打印消耗在JIT编译的时间。

-XX:ErrorFile=./hs\_err\_pid.log：保存错误日志或数据到指定文件中。

-XX:HeapDumpPath=./java\_pid.hprof：指定Dump堆内存时的路径。

-XX:-HeapDumpOnOutOfMemoryError：当首次遭遇内存溢出时Dump出此时的堆内存。

-XX:OnError=";"：出现致命ERROR后运行自定义命令。

-XX:OnOutOfMemoryError=";"：当首次遭遇内存溢出时执行自定义命令。

-XX:-PrintClassHistogram：按下 Ctrl+Break 后打印堆内存中类实例的柱状信息，同JDK的 jmap -histo 命令。

-XX:-PrintConcurrentLocks：按下 Ctrl+Break 后打印线程栈中并发锁的相关信息，同JDK的 jstack -l 命令。

-XX:-PrintCompilation：当一个方法被编译时打印相关信息。

-XX:-PrintGC：每次GC时打印相关信息。

-XX:-PrintGCDetails：每次GC时打印详细信息。

-XX:-PrintGCTimeStamps：打印每次GC的时间戳。

-XX:-TraceClassLoading：跟踪类的加载信息。

-XX:-TraceClassLoadingPreorder：跟踪被引用到的所有类的加载信息。

-XX:-TraceClassResolution：跟踪常量池。

-XX:-TraceClassUnloading：跟踪类的卸载信息。

#### 关于参数名称等

标准参数（-），所有JVM都必须支持这些参数的功能，而且向后兼容；例如：

-client——设置JVM使用Client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——设置JVM使Server模式，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。

非标准参数（-X），默认JVM实现这些参数的功能，但是并不保证所有JVM实现都满足，且不保证向后兼容；

非稳定参数（-XX），此类参数各个JVM实现会有所不同，将来可能会不被支持，需要慎重使用；

### JVM服务参数调优实战（大型网站服务器案例）

承受海量访问的动态Web应用

服务器配置：8 CPU, 8G MEM, JDK 1.6.X

参数方案：

-server -Xmx3550m -Xms3550m -Xmn1256m -Xss128k -XX:SurvivorRatio=6 -XX:MaxPermSize=256m -XX:ParallelGCThreads=8 -XX:MaxTenuringThreshold=0 -XX:+UseConcMarkSweepGC

调优说明：

-Xmx 与 -Xms 相同以避免JVM反复重新申请内存。-Xmx 的大小约等于系统内存大小的一半，即充分利用系统资源，又给予系统安全运行的空间。

-Xmn1256m 设置年轻代大小为1256MB。此值对系统性能影响较大，Sun官方推荐配置年轻代大小为整个堆的3/8。

-Xss128k 设置较小的线程栈以支持创建更多的线程，支持海量访问，并提升系统性能。

-XX:SurvivorRatio=6 设置年轻代中Eden区与Survivor区的比值。系统默认是8，根据经验设置为6，则2个Survivor区与1个Eden区的比值为2:6，一个Survivor区占整个年轻代的1/8。

-XX:ParallelGCThreads=8 配置并行收集器的线程数，即同时8个线程一起进行垃圾回收。此值一般配置为与CPU数目相等。

-XX:MaxTenuringThreshold=0 设置垃圾最大年龄（在年轻代的存活次数）。如果设置为0的话，则年轻代对象不经过Survivor区直接进入年老代。对于年老代比较多的应用，可以提高效率；如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概率。根据被海量访问的动态Web应用之特点，其内存要么被缓存起来以减少直接访问DB，要么被快速回收以支持高并发海量请求，因此其内存对象在年轻代存活多次意义不大，可以直接进入年老代，根据实际应用效果，在这里设置此值为0。

-XX:+UseConcMarkSweepGC 设置年老代为并发收集。CMS（ConcMarkSweepGC）收集的目标是尽量减少应用的暂停时间，减少Full GC发生的几率，利用和应用程序线程并发的垃圾回收线程来标记清除年老代内存，适用于应用中存在比较多的长生命周期对象的情况。

### 内部集成构建服务器案例

高性能数据处理的工具应用

服务器配置：1 CPU, 4G MEM, JDK 1.6.X

参数方案：

-server -XX:PermSize=196m -XX:MaxPermSize=196m -Xmn320m -Xms768m -Xmx1024m

**调优说明：**

-XX:PermSize=196m -XX:MaxPermSize=196m 根据集成构建的特点，大规模的系统编译可能需要加载大量的Java类到内存中，所以预先分配好大量的持久代内存是高效和必要的。

-Xmn320m 遵循年轻代大小为整个堆的3/8原则。

-Xms768m -Xmx1024m 根据系统大致能够承受的堆内存大小设置即可。

在64位服务器上运行应用程序，构建执行时，用 jmap -heap 11540 命令观察**JVM堆内存状况如下：**

Attaching to process ID 11540, please wait...

Debugger attached successfully.

Server compiler detected.

JVM version is 20.12-b01

using thread-local object allocation.

Parallel GC with 4 thread(s)

Heap Configuration:

MinHeapFreeRatio = 40

MaxHeapFreeRatio = 70

MaxHeapSize = 1073741824 (1024.0MB)

NewSize = 335544320 (320.0MB)

MaxNewSize = 335544320 (320.0MB)

OldSize = 5439488 (5.1875MB)

NewRatio = 2

SurvivorRatio = 8

PermSize = 205520896 (196.0MB)

MaxPermSize = 205520896 (196.0MB)

Heap Usage:

PS Young Generation

Eden Space:

capacity = 255852544 (244.0MB)

used = 101395504 (96.69828796386719MB)

free = 154457040 (147.3017120361328MB)

39.63044588683081% used

From Space:

capacity = 34144256 (32.5625MB)

used = 33993968 (32.41917419433594MB)

free = 150288 (0.1433258056640625MB)

99.55984397492803% used

To Space:

capacity = 39845888 (38.0MB)

used = 0 (0.0MB)

free = 39845888 (38.0MB)

0.0% used

PS Old Generation

capacity = 469762048 (448.0MB)

used = 44347696 (42.29325866699219MB)

free = 425414352 (405.7067413330078MB)

9.440459523882184% used

PS Perm Generation

capacity = 205520896 (196.0MB)

used = 85169496 (81.22396087646484MB)

free = 120351400 (114.77603912353516MB)

41.440796365543285% used

结果是比较健康的。

## Tomcat优化

### 内存优化

默认情况下Tomcat的相关内存配置较低，这对于一些大型项目显然是不够用的，这些项目运行就已经耗费了大部分内存空间，何况大规模访问的情况。即使是本文中的这个只有一个页面的超小项目，在并发达到一定程度后也会抛出以下类似异常：

严重: Exception invoking periodic operation: java.lang.OutOfMemoryError: Java heap space

严重: Error processing request java.lang.OutOfMemoryError: GC overhead limit exceeded

说明Tomcat已经无力支持访问处理，内部GC也已经“无能无力”。所以一般情况下我们需要重新配置Tomcat的相关内存大小。

#### 修改内存等JVM相关配置

Linux下修改TOMCAT\_HOME/bin/catalina.sh，在其中加入，可以放在CLASSPATH=下面：

JAVA\_OPTS="-server -XX:PermSize=512M -XX:MaxPermSize=1024m -Xms2048m -Xmx2048m"

windows下修改TOMCAT\_HOME/bin/catalina.bat，在其中加入，可以放在set CLASSPATH=下面：

set JAVA\_OPTS=-server -XX:PermSize=512M -XX:MaxPermSize=1024m -Xms2048m -Xmx2048m

-server：启用 JDK的 server 版本；

-Xms：Java虚拟机初始化时堆的最小内存，一般与 Xmx配置为相同值，这样的好处是GC不必再为扩展内存空间而消耗性能；

-Xmx：Java虚拟机可使用堆的最大内存；

-XX:PermSize：Java虚拟机永久代大小；

-XX:MaxPermSize：Java虚拟机永久代大小最大值；

除了这些参数外您还可以根据具体需要配置其他参数，参数的配置可以参考JVM参数的配置

#### 验证

设置成功后我们可以利用JDK自带的工具进行验证，这些工具都在JAVA\_HOME/bin目录下：

1）jps：用来显示本地的java进程，以及进程号，进程启动的路径等。

2）jmap：观察运行中的JVM 物理内存的占用情况，包括Heap size , Perm size 下载地址 等。

进入命令行模式后，进入JAVA\_HOME/bin目录下，然后输入jps命令；其次jmap -heap 2340

### 配置优化

我们知道TOMCAT\_HOME/conf/server.xml可以配置端口，虚拟路径等等 Tomcat相关主要配置。

#### Connector优化

Connector是连接器，负责接收客户的请求，以及向客户端回送响应的消息。所以 Connector的优化是重要部分。默认情况下 Tomcat只支持200线程访问，超过这个数量的连接将被等待甚至超时放弃，所以我们需要提高这方面的处理能力。

修改这部分配置需要修改TOMCAT\_HOME/conf/server.xml，打开server.xml找到Connector 标签项，默认配置如下：

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

其中port代表服务接口；protocol代表协议类型；connectionTimeout代表连接超时时间，单位为毫秒；redirectPort代表安全通信（https）转发端口，一般配置成443。

可以看到除了这几个基本配置外并无特殊功能，所以我们需要对 Connector 进行扩展。

其中Connector 支持参数属性可以参考Tomcat官方网站（https://tomcat.apache.org/tomcat-8.0-doc/config/http.html），非常多，所以本文就只介绍些常用的。

我们将 Connector 配置修改为如下：

<Connector port="8080"

protocol="HTTP/1.1"

maxThreads="1000"

minSpareThreads="100"

acceptCount="1000"

maxConnections="1000"

connectionTimeout="20000"

maxHttpHeaderSize="8192"

tcpNoDelay="true"

compression="on"

compressionMinSize="2048"

disableUploadTimeout="true"

redirectPort="8443"

enableLookups="false"

URIEncoding="UTF-8" />

1）**port**：代表Tomcat监听端口，也就是网站的访问端口，默认为8080，可以根据需要改成其他。

2）**protocol**：协议类型，可选类型有四种，分别为BIO（阻塞型IO），NIO，NIO2和APR。

（1）BIO：BIO(Blocking I/O)，顾名思义，即阻塞式I/O操作，表示Tomcat使用的是传统的Java I/O操作(即java.io包及其子包)。Tomcat在默认情况下，是以bio模式运行的。遗憾的是，就一般而言，bio模式是三种运行模式中性能最低的一种。BIO配置采用默认即可。

（2）NIO：NIO(New I/O)，是Java SE 1.4及后续版本提供的一种新的I/O操作方式(即java.nio包及其子包)。Java nio是一个基于缓冲区、并能提供非阻塞I/O操作的Java API，因此nio也被看成是non-blocking I/O的缩写。它拥有比传统I/O操作(bio)更好的并发运行性能。要让Tomcat以nio模式来运行也比较简单，我们只需要protocol类型修改为：

//NIO

protocol="org.apache.coyote.http11.Http11NioProtocol"

//NIO2

protocol="org.apache.coyote.http11.Http11Nio2Protocol"

（3）APR：APR(Apache Portable Runtime/Apache可移植运行时)，是Apache HTTP服务器的支持库。你可以简单地理解为:Tomcat将以JNI的形式调用 Apache HTTP服务器的核心动态链接库来处理文件读取或网络传输操作，从而大大地提高 Tomcat对静态文件的处理性能。

与配置 NIO运行模式一样，也需要将对应的 Connector节点的 protocol属性值改为：

protocol="org.apache.coyote.http11.Http11AprProtocol"

相关APR介绍及配置会在下面专门讲。

3）**maxThreads**：由该连接器创建的处理请求线程的最大数目，也就是可以处理的同时请求的最大数目。如果未配置默认值为200。如果一个执行器与此连接器关联，则忽略此属性，因为该属性将被忽略，所以该连接器将使用执行器而不是一个内部线程池来执行任务。

maxThreads是一个重要的配置属性，maxThreads配置的合理直接影响了Tomcat的相关性能，所以这里我们重点讨论下。

maxThreads并不是配置的越大越好，事实上你即使配置成999999也是没有用的，因为这个最大值是受操作系统及相关硬件所制约的，并且最大值并不一定是最优值，所以我们追寻的应该是最优值而不是最大值。

QPS（Query Per Second）：每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。我们常常使用 QPS值来衡量一个服务器的性能。

QPS = 并发数 / 平均响应时间

或者

并发数 = QPS \* 平均响应时间

一个系统吞吐量通常由QPS、并发数两个因素决定，每套系统的这两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其它消耗导致系统性能下降。所谓吞吐量这里可以理解为每秒能处理请求的次数。

所以选择一个合理的 maxThreads值，其实并不是那么容易的事。因为过多的线程只会造成，更多的内存开销，更多的CPU开销，但是对提升QPS确毫无帮助；找到最佳线程数后通过简单的设置，可以让web系统更加稳定，得到最高，最稳定的QPS输出。

我们可以通过以下几种方式来获取 maxThreads的最佳值：

（1）通过线上系统不断使用和用户的不断增长来进行性能测试，观察QPS，响应时间，这种方式会在爆发式增长时系统崩溃，如双12等。

（2）根据公式计算，服务器端最佳线程数量=((线程等待时间+线程cpu时间)/线程cpu时间) \* cpu数量，这种方式有时会被误导，因为某些系统处理环节可能会耗时比较长，从而影响公式的结果。

（3）单、多用户压力测试，查看CPU的消耗，然后直接乘以百分比，再进行压测，一般这个值的附近应该就是最佳线程数量，这种方式理想场景比较适用，实际情况会比这个复杂的多。

（4）根据系统的自身情况调整，如硬件限制，系统限制，程序处理能力限制等。

（5）定期修改为不同的 maxThreads值，看服务器响应结果及用户反应。

QPS和线程数的关系

（1）在最佳线程数量之前，QPS和线程是互相递增的关系，线程数量到了最佳线程之后，QPS持平，不在上升，甚至略有下降，同时相应时间持续上升。

（2）同一个系统而言，支持的线程数越多（最佳线程数越多而不是配置的线程数越多），QPS越高。

QPS和响应时间的关系

（1）对于一般的web系统，响应时间一般有CPU执行时间+IO等待时间组成。

（2）CPU的执行时间减少，对QPS有实质的提升，IO时间的减少，对QPS提升不明显。如果要想明显提升QPS，优化系统的时候要着重优化CPU消耗大户。

所以想要找出 maxThreads的最优值可并不容易，没有最好只有更好，更好的值只能通过时间来显现，如果你不想考虑那么多，一般情况下设置成1000即可。

4）**minSpareThreads**：线程的最小运行数目，这些始终保持运行。如果未指定，默认值为10。

5）**acceptCount**：当所有可能的请求处理线程都在使用时传入连接请求的最大队列长度。如果未指定，默认值为100。一般是设置的跟 maxThreads一样或一半，此值设置的过大会导致排队的请求超时而未被处理。所以这个值应该是主要根据应用的访问峰值与平均值来权衡配置。

6）**maxConnections**：在任何给定的时间内，服务器将接受和处理的最大连接数。当这个数字已经达到时，服务器将接受但不处理，等待进一步连接。NIO与NIO2的默认值为10000，APR默认值为8192。

7）**connectionTimeout**：当请求已经被接受，但未被处理，也就是等待中的超时时间。单位为毫秒，默认值为60000。通常情况下设置为30000。

8）**maxHttpHeaderSize**：请求和响应的HTTP头的最大大小，以字节为单位指定。如果没有指定，这个属性被设置为8192（8 KB）。

9）**tcpNoDelay**：如果为true，服务器socket会设置TCP\_NO\_DELAY选项，在大多数情况下可以提高性能。缺省情况下设为true。

10）**compression**：是否启用gzip压缩，默认为关闭状态。这个参数的可接受值为“off”（不使用压缩），“on”（压缩文本数据），“force”（在所有的情况下强制压缩）。

11）**compressionMinSize**：如果compression="on"，则启用此项。被压缩前数据的最小值，也就是超过这个值后才被压缩。如果没有指定，这个属性默认为“2048”（2K），单位为byte。

12）**disableUploadTimeout**：这个标志允许servlet Container在一个servlet执行的时候，使用一个不同的，更长的连接超时。最终的结果是给servlet更长的时间以便完成其执行，或者在数据上载的时候更长的超时时间。如果没有指定，设为false。

13）**enableLookups**：关闭DNS反向查询。

14）**URIEncoding**：URL编码字符集。

Connector 还有很多其他参数，可以参考Tomcat官网，这里只介绍与性能相关的部分。

#### BIO、NIO、APR

通过配置 protocol的类型可以使用不同的 Connector处理请求。

 //BIO

 protocol="HTTP/1.1"

 //NIO

 protocol="org.apache.coyote.http11.Http11NioProtocol"

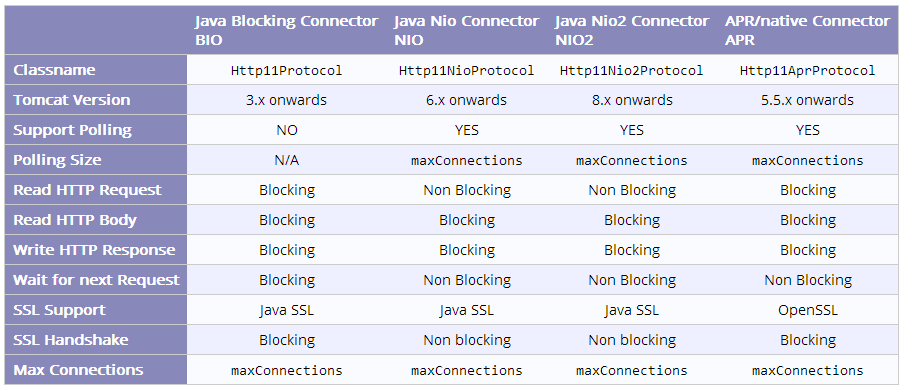
 //NIO2

 protocol="org.apache.coyote.http11.Http11Nio2Protocol"

 //APR

 protocol="org.apache.coyote.http11.Http11AprProtocol"

以下是几种类型 Connector的参数对比：



并不是说 BIO的性能就一定不如 NIO，这几种类型 Connector之间并没有明显的性能区别，它们之间实现流程和原理不同，所以它们的选择是需要根据应用的类型来决定的。

BIO更适合处理简单流程，如程序处理较快可以立即返回结果。简单项目及应用可以采用BIO。

NIO更适合后台需要耗时完成请求的操作，如程序接到了请求后需要比较耗时的处理这已请求，所以无法立即返回结果，这样如果采用BIO就会占用一个连接，而使用NIO后就可以将此连接转让给其他请求，直至程序处理完成返回为止。

APR可以大大提升Tomcat对静态文件的处理性能，同时如果你使用了HTTPS方式传输的话，也可以提升SSL的处理性能。

本文的最后会对几种 Connector进行对比测试。

#### 线程池

Executor代表了一个线程池，可以在Tomcat组件之间共享。使用线程池的好处在于减少了创建销毁线程的相关消耗，而且可以提高线程的使用效率。

要想使用线程池，首先需要在 Service标签中配置 Executor，如下：

<Service name="Catalina">

<Executor name="tomcatThreadPool"

namePrefix="catalina-exec-"

maxThreads="1000"

minSpareThreads="100"

maxIdleTime="60000"

maxQueueSize="Integer.MAX\_VALUE"

prestartminSpareThreads="false"

threadPriority="5"

className="org.apache.catalina.core.StandardThreadExecutor"/>

其中，

**name**：线程池名称，用于 Connector中指定。

**namePrefix**：所创建的每个线程的名称前缀，一个单独的线程名称为 namePrefix+threadNumber。

**maxThreads**：池中最大线程数。

**minSpareThreads**：活跃线程数，也就是核心池线程数，这些线程不会被销毁，会一直存在。

**maxIdleTime**：线程空闲时间，超过该时间后，空闲线程会被销毁，默认值为6000（1分钟），单位毫秒。

**maxQueueSize**：在被执行前最大线程排队数目，默认为Int的最大值，也就是广义的无限。除非特殊情况，这个值不需要更改，否则会有请求不会被处理的情况发生。

**prestartminSpareThreads**：启动线程池时是否启动 minSpareThreads部分线程。默认值为false，即不启动。

**threadPriority**：线程池中线程优先级，默认值为5，值从1到10。

**className**：线程池实现类，未指定情况下，默认实现类为org.apache.catalina.core.StandardThreadExecutor。如果想使用自定义线程池首先需要实现 org.apache.catalina.Executor接口。

线程池配置完成后需要在 Connector中指定：

<Connector executor="tomcatThreadPool"  ………

#### Listener

另一个影响Tomcat 性能的因素是内存泄露。Server标签中可以配置多个Listener，其中 JreMemoryLeakPreventionListener是用来预防JRE内存泄漏。此Listener只需在Server标签中配置即可，默认情况下无需配置，已经添加在 Server中。

<Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />

### 组件优化

#### APR

APR(Apache Portable Runtime)是一个高可移植库，它是Apache HTTP Server 2.x的核心。APR有很多用途，包括访问高级 IO功能(例如sendfile,epoll和OpenSSL)，OS级别功能(随机数生成，系统状态等等)，本地进程管理(共享内存，NT管道和UNIX sockets)。这些功能可以使Tomcat作为一个通常的前台WEB服务器，能更好地和其它本地web技术集成，总体上让Java更有效率作为一个高性能web服务器平台而不是简单作为后台容器。

APR的目的如其名称一样，主要为上层的应用程序提供一个可以跨越多操作系统平台使用的底层支持接口库。在早期的Apache版本中，应用程序本身必须能够处理各种具体操作系统平台的细节，并针对不同的平台调用不同的处理函数。随着Apache的进一步开发，Apache组织决定将这些通用的函数独立出来并发展成为一个新的项目。这样，APR的开发就从Apache中独立出来，Apache仅仅是使用APR而已。目前APR主要还是由Apache使用，不过由于APR的较好的移植性，因此一些需要进行移植的C程序也开始使用APR。

APR使得平台细节的处理进行下移。对于应用程序而言，它们根本就不需要考虑具体的平台，不管是Unix、Linux还是Window，应用程序执行的接口基本都是统一一致的。因此对于APR而言，可移植性和统一的上层接口是其考虑的一个重点。而APR最早的目的并不是如此，它最早只是希望将Apache中用到的所有代码合并为一个通用的代码库，然而这不是一个正确的策略，因此后来APR改变了其目标。有的时候使用公共代码并不是一件好事，比如如何将一个请求映射到线程或者进程是平台相关的，因此仅仅一个公共的代码库并不能完成这种区分。APR的目标则是希望安全合并所有的能够合并的代码而不需要牺牲性能。

#### Tomcat Native

Tomcat Native是 Tomcat可选组件，它可以让 Tomcat使用 Apache 的 APR包来处理包括文件和网络IO操作，从而提升性能及兼容性。

### 性能测试

#### Jmeter

Apache JMeter是Apache组织开发的基于Java的压力测试工具。用于对软件做压力测试，它最初被设计用于Web应用测试，但后来扩展到其他测试领域。 它可以用于测试静态和动态资源，例如静态文件、Java 小服务程序、CGI 脚本、Java 对象、数据库、FTP 服务器， 等等。JMeter 可以用于对服务器、网络或对象模拟巨大的负载，来自不同压力类别下测试它们的强度和分析整体性能。另外，JMeter能够对应用程序做功能/回归测试，通过创建带有断言的脚本来验证你的程序返回了你期望的结果。为了最大限度的灵活性，JMeter允许使用正则表达式创建断言。

Apache jmeter 可以用于对静态的和动态的资源（文件，Servlet，Perl脚本，java 对象，数据库和查询，FTP服务器等等）的性能进行测试。它可以用于对服务器、网络或对象模拟繁重的负载来测试它们的强度或分析不同压力类型下的整体性能。你可以使用它做性能的图形分析或在大并发负载测试你的服务器/脚本/对象。

# 技术

## 缓存

### 缓存穿透

#### 什么是缓存穿透

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

#### 如何避免

1：对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。

2：对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤。【感觉应该用的不多吧】

### 缓存雪崩

#### 什么是缓存雪崩

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

#### 如何避免

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

3：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期（此点为补充）

### 分布式缓存系统面临的问题

#### 缓存一致性问题

1：缓存系统与底层数据的一致性。这点在底层系统是“可读可写”时，写得尤为重要

2：有继承关系的缓存之间的一致性。为了尽量提高缓存命中率，缓存也是分层：全局缓存，二级缓存。他们是存在继承关系的。全局缓存可以有二级缓存来组成。

3：多个缓存副本之间的一致性。为了保证系统的高可用性，缓存系统背后往往会接两套存储系统（如memcache，redis等）

#### 缓存穿透和缓存雪崩

上面有讲过

#### 缓存数据的淘汰

缓存淘汰的策略有两种：

(1) 定时去清理过期的缓存。

(2)当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂，具体用哪种方案，大家可以根据自己的应用场景来权衡。

1. 预估失效时间

2. 版本号（必须单调递增，时间戳是最好的选择）

3. 提供手动清理缓存的接口。

## Redis

### Redis的了解

1.redis是一种内存型、非关系型数据库；

2.redis的数据会存储到硬盘上；

3.Redis（持久化）保存数据一种采用的是rdb(快照的形式进行数据的保存，快照是采用一种机制保存，这种机制你可以自己在redis的配置文件中进行配置，隔多长时间保存多少数据,这种方法是比较快了，数据会被保存到硬盘上，当redis启动的时候，硬盘上的数据会在写到数据库中，这种方法是大大提高了数据的存储和读取的效率。但是性能上还是会有一定的影响，毕竟他还要去占用cpu去做保存快照的操作；

4.redis保存数据的另一种方法就是采用aof集群的方式，它会使每个服务器内都有对方服务器上的数据，这中方式避免了像rdb那种在特定时间内用快照保存数据的机制，减少内存的占用，从而提高性能。而且这种方法还有另一个好处就是，当一个服务器荡掉以后，数据不会因此丢失，而是当你在操作数据的时候会从其他的服务器上读到你想要操作的数据。但是这种方法会使你在硬件设备上付出一定的成本。

### Redis应用

1.它应用在一些数据经常被读取的地方，如电商购物车中的价格、数量等；

2.它应用在一些能大大减轻数据库压力的地方。

### Redis数据类型

#### 字符串类型

字符串类型是Redis中最为基础的数据存储类型，它在Redis中是二进制的，这便意味着该类型可以接受任何格式的数据，如JPEG图像数据或Json对象描述信息等。在Redis中字符串类型的Value最多可以容纳的数据长度是512M。

#### hash（散列）

我们可以将Redis中的Hashes类型看成具有String Key和String Value的map容器。所以该类型非常适合于存储值对象的信息。如Username、Password和Age等。如果Hash中包含很少的字段，那么该类型的数据也将仅占用很少的磁盘空间。每一个Hash可以存储4294967295个键值对。

#### List类型

在Redis中，List类型是按照插入顺序排序的字符串双向链表。和数据结构中的普通链表一样，我们可以在其头部(left)和尾部(right)添加新的元素。在插入时，如果该键并不存在，Redis将为该键创建一个新的链表。与此相反，如果链表中所有的元素均被移除，那么该键也将会被从数据库中删除。List中可以包含的最大元素数量是4294967295。

从元素插入和删除的效率视角来看，如果我们是在链表的两头插入或删除元素，这将会是非常高效的操作，即使链表中已经存储了百万条记录，该操作也可以在常量时间内完成。然而需要说明的是，如果元素插入或删除操作是作用于链表中间，那将会是非常低效的。相信对于有良好数据结构基础的开发者而言，这一点并不难理解。

#### set类型

在Redis中，我们可以将Set类型看作为没有排序的字符集合，和List类型一样，我们也可以在该类型的数据值上执行添加、删除或判断某一元素是否存在等操作。需要说明的是，这些操作的时间复杂度为O(1)，即常量时间内完成次操作。Set可包含的最大元素数量是4294967295。

和List类型不同的是，Set集合中不允许出现重复的元素，这一点和C++标准库中的set容器是完全相同的。换句话说，如果多次添加相同元素，Set中将仅保留该元素的一份拷贝。和List类型相比，Set类型在功能上还存在着一个非常重要的特性，即在服务器端完成多个Sets之间的聚合计算操作，如unions、intersections和differences。由于这些操作均在服务端完成，因此效率极高，而且也节省了大量的网络IO开销。

#### Sorted-Sets类型

Sorted-Sets和Sets类型极为相似，它们都是字符串的集合，都不允许重复的成员出现在一个Set中。它们之间的主要差别是Sorted-Sets中的每一个成员都会有一个分数(score)与之关联，Redis正是通过分数来为集合中的成员进行从小到大的排序。然而需要额外指出的是，尽管Sorted-Sets中的成员必须是唯一的，但是分数(score)却是可以重复的。

在Sorted-Set中添加、删除或更新一个成员都是非常快速的操作，其时间复杂度为集合中成员数量的对数。由于Sorted-Sets中的成员在集合中的位置是有序的，因此，即便是访问位于集合中部的成员也仍然是非常高效的。事实上，Redis所具有的这一特征在很多其它类型的数据库中是很难实现的，换句话说，在该点上要想达到和Redis同样的高效，在其它数据库中进行建模是非常困难的。

### Redis持久化

Redis对持久化的支持主要是通过RDB和AOF文件来进行持久化的。

其中RDB持久化是在指定的时间间隔内生成数据集的时间点快照。而AOF持久化记录的是服务器执行的所有写操作命令，在服务器启动的时候，通过重新执行这些命令来还原数据集的。AOF文件中的命令全部以Redis协议的格式来保存，新的命令会被追加到文件的末尾。Redis还可以在后台对AOF文件进行重写，使得AOF文件的体积不会超出保存数据集状态所需要的实际大小。

Redis可以同时使用AOF持久化和RDB持久化，在这种情况下，Redis重启之后，他会优先使用AOF文件来还原数据集，因为AOF文件保存的数据集通常要比RDB文件中保存的数据集更加的完整。

当然也可以关闭持久化功能，让数据仅仅在服务器运行的时候存在。

#### RDB

RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。也就是说RDB持久化是将当前数据的快照存成一个数据文件的持久化机制。

RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。因此RDB 非常适用于灾难恢复（disaster recovery）。

而一个持续写入的数据库如何生成快照呢？

Redis借助了fork命令的copy on write机制。在生成快照时，将当前进程fork出一个子进程，然后在子进程中循环所有的数据，将数据写成为RDB文件。

我们可以通过Redis的save指令来配置RDB快照生成的时机，比如你可以配置当10分钟以内有100次写入就生成快照，也可以配置当1小时内有1000次写入就生成快照，也可以多个规则一起实施。这些规则的定义就在Redis的配置文件中，你也可以通过Redis的CONFIG SET命令在Redis运行时设置规则，不需要重启Redis。

Redis的RDB文件不会坏掉，因为其写操作是在一个新进程中进行的，当生成一个新的RDB文件时，Redis生成的子进程会先将数据写到一个临时文件中，然后通过原子性rename系统调用将临时文件重命名为RDB文件，这样在任何时候出现故障，Redis的RDB文件都总是可用的。

同时，Redis的RDB文件也是Redis主从同步内部实现中的一环。

但是，我们可以很明显的看到，RDB有他的不足，就是一旦数据库出现问题，那么我们的RDB文件中保存的数据并不是全新的，从上次RDB文件生成到Redis停机这段时间的数据全部丢掉了。在某些业务下，这是可以忍受的，我们也推荐这些业务使用RDB的方式进行持久化，因为开启RDB的代价并不高。但是对于另外一些对数据安全性要求极高的应用，无法容忍数据丢失的应用，RDB就无能为力了，所以Redis引入了另一个重要的持久化机制：AOF日志。

#### AOF

aof日志的全称是append only file，从名字上我们就能看出来，它是一个追加写入的日志文件。与一般数据库的binlog不同的是，AOF文件是可识别的纯文本，它的内容就是一个个的Redis标准命令。

写操作都生成了一条相应的命令作为日志。其中值得注意的是最后一个del命令，它并没有被记录在AOF日志中，这是因为Redis判断出这个命令不会对当前数据集做出修改。所以不需要记录这个无用的写命令。

另外AOF日志也不是完全按客户端的请求来生成日志的，比如命令INCRBYFLOAT在记AOF日志时就被记成一条SET记录，因为浮点数操作可能在不同的系统上会不同，所以为了避免同一份日志在不同的系统上生成不同的数据集，所以这里只将操作后的结果通过SET来记录。

##### AOF重写

你可以会想，每一条写命令都生成一条日志，那么AOF文件是不是会很大？

答案是肯定的，AOF文件会越来越大，所以Redis又提供了一个功能，叫做AOF rewrite。其功能就是重新生成一份AOF文件，新的AOF文件中一条记录的操作只会有一次，而不像一份老文件那样，可能记录了对同一个值的多次操作。

其生成过程和RDB类似，也是fork一个进程，直接遍历数据，写入新的AOF临时文件。在写入新文件的过程中，所有的写操作日志还是会写到原来老的AOF文件中，同时还会记录在内存缓冲区中。当重写操作完成后，会将所有缓冲区中的日志一次性写入到临时文件中。然后调用原子性的rename命令用新的AOF文件取代老的AOF文件。

从上面的流程我们能够看到，RDB和AOF操作都是顺序IO操作，性能都很高。而同时在通过RDB文件或者AOF日志进行数据库恢复的时候，也是顺序的读取数据加载到内存中。所以也不会造成磁盘的随机读。

##### AOF可靠性设置

AOF是一个写文件操作，其目的是将操作日志写到磁盘上，所以它也同样会遇到我们上面说的写操作的5个流程。那么写AOF的操作安全性又有多高呢。实际上这是可以设置的，在Redis中对AOF调用write(2)写入后，何时再调用fsync将其写到磁盘上，通过appendfsync选项来控制，下面appendfsync的三个设置项，安全强度逐渐变强。

appendfsync no 当设置appendfsync为no的时候，Redis不会主动调用fsync去将AOF日志内容同步到磁盘，所以这一切就完全依赖于操作系统的调试了。对大多数Linux操作系统，是每30秒进行一次fsync，将缓冲区中的数据写到磁盘上。

appendfsync everysec 当设置appendfsync为everysec的时候，Redis会默认每隔一秒进行一次fsync调用，将缓冲区中的数据写到磁盘。但是当这一次的fsync调用时长超过1秒时。Redis会采取延迟fsync的策略，再等一秒钟。也就是在两秒后再进行fsync，这一次的fsync就不管会执行多长时间都会进行。这时候由于在fsync时文件描述符会被阻塞，所以当前的写操作就会阻塞。

所以，结论就是，在绝大多数情况下，Redis会每隔一秒进行一次fsync。在最坏的情况下，两秒钟会进行一次fsync操作。

这一操作在大多数数据库系统中被称为group commit，就是组合多次写操作的数据，一次性将日志写到磁盘。

appednfsync always 当设置appendfsync为always时，每一次写操作都会调用一次fsync，这时数据是最安全的，当然，由于每次都会执行fsync，所以其性能也会受到影响。

对于pipelining有什么不同

对于pipelining的操作，其具体过程是客户端一次性发送N个命令，然后等待这N个命令的返回结果被一起返回。通过采用pipilining就意味着放弃了对每一个命令的返回值确认。由于在这种情况下，N个命令是在同一个执行过程中执行的。所以当设置appendfsync为everysec时，可能会有一些偏差，因为这N个命令可能执行时间超过1秒甚至2秒。但是可以保证的是，最长时间不会超过这N个命令的执行时间和。

##### 与postgreSQL和MySQL的比较

这一块就不多说了，由于上面操作系统层面的数据安全已经讲了很多，所以其实不同的数据库在实现上都大同小异。总之最后的结论就是，在Redis开启AOF的情况下，其单机数据安全性并不比这些成熟的SQL数据库弱。

数据导入

这些持久化的数据有什么用，当然是用于重启后的数据恢复。Redis是一个内存数据库，无论是RDB还是AOF，都只是其保证数据恢复的措施。所以Redis在利用RDB和AOF进行恢复的时候，都会读取RDB或AOF文件，重新加载到内存中。相对于MySQL等数据库的启动时间来说，会长很多，因为MySQL本来是不需要将数据加载到内存中的。

但是相对来说，MySQL启动后提供服务时，其被访问的热数据也会慢慢加载到内存中，通常我们称之为预热，而在预热完成前，其性能都不会太高。而Redis的好处是一次性将数据加载到内存中，一次性预热。这样只要Redis启动完成，那么其提供服务的速度都是非常快的。

而在利用RDB和利用AOF启动上，其启动时间有一些差别。RDB的启动时间会更短，原因有两个，一是RDB文件中每一条数据只有一条记录，不会像AOF日志那样可能有一条数据的多次操作记录。所以每条数据只需要写一次就行了。另一个原因是RDB文件的存储格式和Redis数据在内存中的编码格式是一致的，不需要再进行数据编码工作。在CPU消耗上要远小于AOF日志的加载。

### Redis备份数据

Redis 对于数据备份是非常友好的， 因为你可以在服务器运行的时候对 RDB 文件进行复制： RDB 文件一旦被创建， 就不会进行任何修改。 当服务器要创建一个新的 RDB 文件时， 它先将文件的内容保存在一个临时文件里面， 当临时文件写入完毕时， 程序才使用 原子地用临时文件替换原来的 RDB 文件。这也就是说， 无论何时， 复制 RDB 文件都是绝对安全的。

### Redis事务

和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，MULTI/EXEC/DISCARD/WATCH这四个命令是我们实现事务的基石。相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出Redis中事务的实现特征：

1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

2). 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

3). 我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

4). 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行。

5). 当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

### Redis集群

redis cluster在设计的时候，就考虑到了去中心化，去中间件，也就是说，集群中的每个节点都是平等的关系，都是对等的，每个节点都保存各自的数据和整个集群的状态。每个节点都和其他所有节点连接，而且这些连接保持活跃，这样就保证了我们只需要连接集群中的任意一个节点，就可以获取到其他节点的数据。

Redis 集群没有并使用传统的一致性哈希来分配数据，而是采用另外一种叫做哈希槽 (hash slot)的方式来分配的。redis cluster 默认分配了 16384 个slot，当我们set一个key 时，会用CRC16算法来取模得到所属的slot，然后将这个key 分到哈希槽区间的节点上，具体算法就是：CRC16(key) % 16384。

Redis 集群会把数据存在一个 master 节点，然后在这个 master 和其对应的salve 之间进行数据同步。当读取数据时，也根据一致性哈希算法到对应的 master 节点获取数据。只有当一个master 挂掉之后，才会启动一个对应的 salve 节点，充当 master 。

需要注意的是：必须要3个或以上的主节点，否则在创建集群时会失败，并且当存活的主节点数小于总节点数的一半时，整个集群就无法提供服务了。

## JBPM4

### 工作流引擎描述

JPBM是JBOSS旗下的一个开源的基于hibernate的工作流引擎。工作流就是在日常生活中，我们一些常见的如请假流程、采购流程、入职流程，通俗的来讲就是一些在现实生活中的流程以信息化以程序的方式实现。

一个工作流首先需要进行流程定义，流程定义是由节点和跳转组成的，节点又可以称为环节、活动节点、活动环节，并且节点也可以分为两大类型：人工节点和自动节点，人工节点有start开始节点、end结束节点、task任务节点，自动节点有decision判断节点、fork分支节点、join聚合节点和state状态节点，并且一个流程有且只有一个开始节点，但可以有多个结束节点。

流程定义是静止的，它在运行状态时会转换成流程实例，一个流程定义可以对应多个流程实例。流程运行后，会产生两个文件，\*.jdpl.xml文件和\*.png图片文件，也会生成18张数据库表，常用且核心的表有JBPM4\_LOB 存储表，主要存储xml文件和png图片、JBPM4\_TASK 任务表、JBPM4\_EXECUTION 流程实例表、JBPM4\_VARIABLE变量表。

JBPM有五大核心类：

ProcessEngine：主要获取各种的Service

RepositoryService：主要发布流程定义

ExecutionService：主要操作流程实例

TaskService：主要操作人工服务

HistoryService：主要操作历史服务。

核心方法：

读取jbpm定义的文件生成zip包存到lob表中：createDeployment()

获取流程定义列表：createProcessDefinitionQuery

根据定义的key或id来启动流程实例：startProcessInstanceByKey(id)

获取待办任务列表：findPersonalTasks(userName)

完成指定任务列表：completeTask(\*.getActivityId())

获取历史任务列表：createHistoryTaskQuery()

获取流程实例的ID：task.getExecutionId()

(了解的表)

JBPM4\_HIST\_ACTINST 流程活动(节点) 实例表

JBPM4\_HIST\_DETAIL 流程历史详细表

JBPM4\_HIST\_PROCINST 流程实例历史表

JBPM4\_HIST\_TASK 流程任务实例历史表

JBPM4\_HIST\_VAR 流程变量( 上下文) 历史表

### JPBM业务场景

首先进行请假的流程定义，我们流程的定义是（员工提交请假单---》经理审批---》总监审批---》总经理审批---》结束），通过repositoryService将其发布部署到jbpm4\_lob表中，之后获取流程定义列表，选中请假的流程定义，员工开始进行请假单的填写，保存并通过executionService开启流程实例，然后用taskService获取经理的待办任务列表，选中待办任务，进行审批，通过调用taskService.completeTask()进入到总监审批环节，然后用总监进行登录，同样获取待办任务列表，然后调用taskService.completeTask()进入总经理审批环节，总经理审批之后，结束流程。在这个过程中我们还可以根据historyService查看当前登录人已办的任务列表。

## Apache ActiveMQ

### 技术概述

JMS是指java消息服务（Java Message Service) 应用程序接口是一个java平台中关于面向消息中间件（MOM）的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

ActiveMQ是Apache软件基金下的一个开源软件，它遵循JMS1.1规范，是消息驱动中间软件，为企业消息传递提供高可用，出色性能，可扩展，稳定和安全保障。ActiveMQ使用Apache许可协议，因此任何人都可以使用和修改它而不必反馈任何改变。ActiveMQ的目标是在尽可能多的平台和语言上提供一个标准的，消息驱动的应用集成。ActiveMQ实现JMS规范并在此之上提供大量额外的特性。

### 何时使用ActiveMQ

有很多情况ActiveMQ和异步消息能对一个系统的架构产生有意义的作用。下面列举一些场景。

1.不同语言应用集成----ActiveMQ使用java编写，并且提供一个java客户端API。但ActiveMQ也为C/C++，.NET,Perl,PHP,Python,Ruby等提供客户端。当你考虑在不同平台不同语言下的各个应用进行集成时，这将是一个巨大的优势。不同语言的客户端API使各种不同的语言能够通过ActiveMQ发送和接收消息。对于ActiveMQ提供的多语言兼容，还有一个好处是相对于RPC调用，它能帮助系统各应用间的解耦。

2.RPC的替代者----应用广泛的使用RPC模式的同步调用。想一下，现在大量使用RPC调用的客户端服务器模式的应用，它们包括ATMs，大多数web应用，信用卡系统，销售点（point-of-sale）系统等。尽管它们大多数是成功的，但是转换到异步消息模式能够在保证正确响应的情况下带来一些好处。使用同步请求的系统在规模上有较大的限制，因为请求会被阻塞，从而导致整个系统变慢。如果使用异步消息替代，可以很容易增加额外的消息接收者，使得消息能被并发消耗，从而加快请求处理。当然，必须你系统应用间是解耦的。

3.应用间解耦----就是上面讨论过的，紧耦合系统能带来很多问题，特别是在应用是分布式的情况下。松耦合系统，也就是依赖性小的系统，可以更好地适应未知变化。不只是系统某部分的改变不会影响整个系统，而且部件间的交互也更简单。相比使用同步的系统（调用者必须等待被调用者返回信息），异步系统（调用方发送消息后就不管，即fire-and-forget）能够给我们带来事件驱动架构（event-driven architecture EDA）.

4.作为事件驱动架构的骨架----解耦，异步架构的系统允许通过代理器自己配置更多的客户端，内存等（即vertical scalability）来扩大系统，而不是增加更多的代理器（即horizontal scalability）。考虑如亚马逊这样繁忙的电子商务系统。当用户购买物品，事实上系统需要很多步骤去处理，包括下单，创建发票，付款，执行订单，运输等。但是用户下单后，会立即返回“谢谢你下单”的界面。不只是没有延迟，而且用户还会受到一封邮件表明订单已经收到。在亚马逊下单的例子就是一个多步处理的例子。每一步都由单独的服务去处理。当用户下单是，有一个同步的体积表单动作，但整个处理流程并不通过浏览器同步处理。相反地，订单马上被接受和反馈。而剩下的步骤就通过异步处理。如果在处理过程中出错，用户会通过邮件收到通知。这样的异步处理能提供高负载和高可用性。

5.提高系统扩展性。很多使用事件驱动设计的系统是为了获得高可扩展性，例如电子商务，政府，制造业，线上游戏等。通过异步消息分开商业处理步骤给各个应用，能够带来很多可能性。考虑设计一个应用来完成一项特殊的任务。这就是面向服务的架构（service-oriented architecture SOA）。每一个服务完成一个功能并且只有一个功能。应用就通过服务组合起来，服务间使用异步消息和最终一致性。这样的设计便可以引入一个复杂事件处理概念（complex event processing CEP）。使用CEP，部件间的交互可以被记录追踪。在异步消息系统中，可以很容易在部件间增加一层处理。

## 负载均衡

负载均衡是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。均衡负载能够平均分配客户请求到服务器列阵，籍此提供快速获取重要数据，解决大量并发访问服务问题。这种群集技术可以用最少的投资获得接近于大型主机的性能。

网络负载均衡的优点

第一，网络负载均衡能将传入的请求传播到多达32台服务器上，即可以使用最多32台服务器共同分担对外的网络请求服务。网络负载均衡技术保证即使是在负载很重的情况下，服务器也能做出快速响应;

第二，网络负载均衡对外只需提供一个IP地址(或域名);

第三，当网络负载均衡中的一台或几台服务器不可用时，服务不会中断。网络负载均衡自动检测到服务器不可用时，能够迅速在剩余的服务器中重新指派客户机通讯。这项保护措施能够帮助你为关键的业务程序提供不中断的服务，并可以根据网络访问量的增加来相应地增加网络负载均衡服务器的数量;

第四，网络负载均衡可在普通的计算机上实现。

## LVS

### 简介

首先简单介绍一下LVS (Linux Virtual Server)到底是什么东西，其实它是一种集群(Cluster)技术，采用IP负载均衡技术和基于内容请求分发技术。调度器具有很好的吞吐率，将请求均衡地转移到不同的服务器上执行，且调度器自动屏蔽掉服务器的故障，从而将一组服务器构成一个高性能的、高可用的虚拟服务器。整个服务器集群的结构对客户是透明的，而且无需修改客户端和服务器端的程序。

### 组成

负载调度器(load balancer/ Director)，它是整个集群对外面的前端机，负责将客户的请求发送到一组服务器上执行，而客户认为服务是来自一个IP地址(我们可称之为虚拟IP地址)上的。

　　服务器池(server pool/ Realserver)，是一组真正执行客户请求的服务器，执行的服务一般有WEB、MAIL、FTP和DNS等。

　　共享存储(shared storage)，它为服务器池提供一个共享的存储区，这样很容易使得服务器池拥有相同的内容，提供相同的服务。

## WebService

### 用过和WebService相同的技术么？

Hessian是一个轻量级的remoting onhttp工具，使用简单的方法提供了RMI的功能。 相比WebService，Hessian更简单、快捷。采用的是二进制RPC协议，因为采用的是二进制协议，所以它很适合于发送二进制数据。Hessian 是由 caucho 提供的一个基于 binary-RPC 实现的远程通讯 library 。

1 、是基于什么协议实现的？

基于 Binary-RPC 协议实现。

2 、怎么发起请求？

需通过 Hessian 本身提供的 API 来发起请求。

3 、怎么将请求转化为符合协议的格式的？

Hessian 通过其自定义的串行化机制将请求信息进行序列化，产生二进制流。

4 、使用什么传输协议传输？

Hessian 基于 Http 协议进行传输。

5 、响应端基于什么机制来接收请求？

响应端根据 Hessian 提供的 API 来接收请求。

6 、怎么将流还原为传输格式的？

Hessian 根据其私有的串行化机制来将请求信息进行反序列化，传递给使用者时已是相应的请求信息对象了。

7 、处理完毕后怎么回应？

处理完毕后直接返回， hessian 将结果对象进行序列化，传输至调用端。

### WebService数据压缩

在使用WebService技术的过程中，免不了进行数据传输，比如说传输文本、图片、Zip压缩包等。在网络环境比较好、数据量小的情况下，传输的方式可以忽略不计，我们可以选择任意的方式，不需要考虑优化的问题，可以将图片、包变成二进制流进行传输，可以从数据库读出数据来，以DataSet形式进行传输。

但是如果网络条件不好，或者数据量很大的时候，我们就要考虑到对数据进行压缩之后再传输。

例如在传输DataSet的过程中，我们首先要从数据库中读取DataSet，利用DataSetSurrogate类将DataSet序列化，转换成二位数组，这个时候数据的大小基本上变成原来的1/3，然后我们利用SharpZipLib对数据进行压缩，最后使用WebService Enhancements的WS-Attachment进行传输。将包放到SOAP附件里面，而不是在信封里面，不用经过XML序列化，封成DIME消息（Direct Internet Message Encapsulation），这样原来的大数据就可以被封成很小的数据了，传输也变得简单，迅速。

实现DIME文件传输，我们首先要安装Microsoft WSE 2.0 SP3.msi，然后在服务器端配置：

1.添加Microsoft.Web.Service2.dll引用

2.修改WebConfig配置文件

3.WebMethod实现

最后在客户端引用，Reference.cs文件中代理类的基类为Microsoft.Web.Services2.WebServicesClientProtocol;

### WebService数据加密

发布WebService服务的过程中，为了安全，通常需要安全验证。所谓的安全验证就是客户端的每一次请求，都要带着请求头，而服务端就去解析请求头，看里面带的token是否跟预期的一致，如果一致就说明安全了，否则就抛出异常不让调用。

那么我们在实现安全验证的过程中，我们涉及到了这样两个操作：

首先在客户端发送webService调用前，构造一个SOAP消息头，把token带过去

然后在服务器端解析消息头，把指定的那个头字段解析出来，对比两边的token是否相同

## Solr

### 简介

采用Java开发，基于Luncene的全文搜索服务器，同时对其进行了扩展(扩展了面向抽象编程的地方，比如分词器，查询)，提供了比Lucene更为丰富的查询语言（比如，过滤器），同时实现了可配置（跟hadoop整合，之前索引结构写在代码中，现在提前定义好）、可扩展并对查询性能进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的全文搜索引擎。

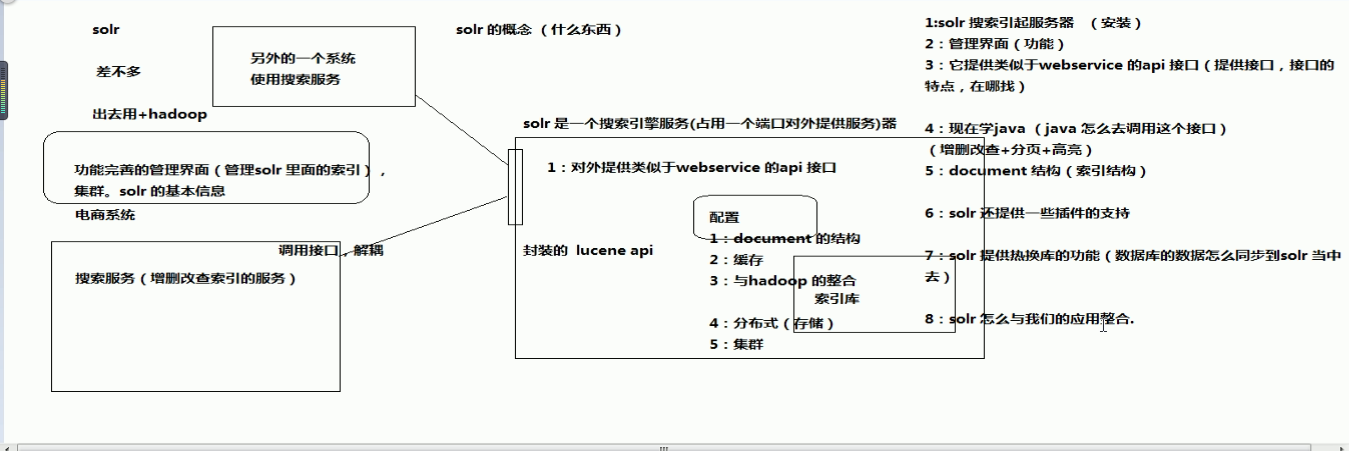
服务器 占用一个端口来提供服务 比如 可以加缓存

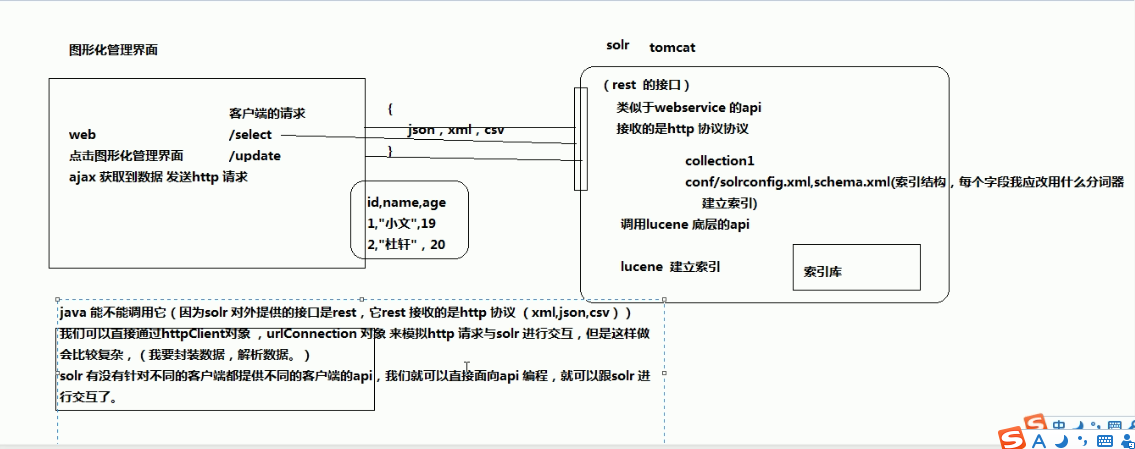
Solr是一个独立的企业级搜索应用服务器，它对外提供类似于Web-service的API接口。用户可以通过http请求，向搜索引擎服务器提交一定格式的XML文件，生成索引；也可以通过Http G SolrJ操作提出查找请求（也可以提交json格式），并得到XML格式的返回结果.

### Solr与lucene的关系

Luncene是一套信息检索工具包，但并不包含搜索引擎系统，它包含了索引结构，读写索引工具、相关性工具（其他的搜索组件）、排序等功能，因此在使用luncene时你扔需要关注搜索引擎系统，例如数据获取、解析、分词等方面的东西。

首先solr是基于luncene做的，solr的目标是打造一款企业级的搜索引擎系统，因此它更接近于我们认识到的搜索引擎系统，它是一个搜索引擎服务，通过各种API可以让你的应用使用搜索服务，而不需要将搜索逻辑耦合在应用中。而且solr可以根据配置文件定义数据解析的方式，更像是一个搜索框架，它也支持主从（集群中的方式）、热换库（索引的数据与数据库的同步）等操作，还添加了高亮、facet（搜索组件）等搜索引擎常见功能的支持。





使用solrj 使用solrJava版的solrj来跟tomcat进行交互

schema.xml中 <field name=”id”(分词器名称) type=”string”(类型) indexed=”true”(是否建立索引) stored=”true”(是否存储) required=”是否是必须存在” multiValued=”false” (是否允许有多个值)/>

<dy>

schema.xml 是用来定义索引数据中的域的，包括域名称，域类型，域是否索引，是否分词，是否存储，是否标准化即 Norms ，是否存储项向量等等。

solrconfig .xml 文件包含了大部分的参数用来配置Solr本身的。

### Solr性能优化

• 将所有只用于搜索的，而不需要作为结果的field（特别是一些比较大的field）的stored设置为false

• 将不需要被用于搜索的，而只是作为结果返回的field的indexed设置为false

• 删除所有不必要的copyField声明

• 为了索引字段的最小化和搜索的效率，将所有的 text fields的index都设置成field，然后使用copyField将他们都复制到一个总的 text field上，然后对他进行搜索。

• 为了最大化搜索效率，使用java编写的客户端与solr交互（使用流通信）

• 在服务器端运行JVM（省去网络通信），使用尽可能高的Log输出等级，减少日志量。

### Solr集群

#### 什么是SolrCloud

SolrCloud(solr 云)是Solr提供的分布式搜索方案，当你需要大规模，容错，分布式索引和检索能力时使用 SolrCloud。当一个系统的索引数据量少的时候是不需要使用SolrCloud的，当索引量很大，搜索请求并发很高，这时需要使用SolrCloud来满足这些需求。

SolrCloud是基于Solr和Zookeeper的分布式搜索方案，它的主要思想是使用Zookeeper作为集群的配置信息中心。

**它有几个特色功能**：

1）集中式的配置信息

2）自动容错

3）近实时搜索

4）查询时自动负载均衡

#### Zookeeper是个什么玩楞

顾名思义zookeeper就是动物园管理员，他是用来管hadoop（大象）、Hive(蜜蜂)、pig(小猪)的管理员， Apache Hbase和 Apache Solr 的分布式集群都用到了zookeeper；Zookeeper:是一个分布式的、开源的程序协调服务，是hadoop项目下的一个子项目。

#### Zookeeper可以干那些事

##### 配置管理

在我们的应用中除了代码外，还有一些就是各种配置。比如数据库连接等。一般我们都是使用配置文件的方式，在代码中引入这些配置文件。但是当我们只有一种配置，只有一台服务器，并且不经常修改的时候，使用配置文件是一个很好的做法，但是如果我们配置非常多，有很多服务器都需要这个配置，而且还可能是动态的话使用配置文件就不是个好主意了。这个时候往往需要寻找一种集中管理配置的方法，我们在这个集中的地方修改了配置，所有对这个配置感兴趣的都可以获得变更。比如我们可以把配置放在数据库里，然后所有需要配置的服务都去这个数据库读取配置。但是，因为很多服务的正常运行都非常依赖这个配置，所以需要这个集中提供配置服务的服务具备很高的可靠性。一般我们可以用一个集群来提供这个配置服务，但是用集群提升可靠性，那如何保证配置在集群中的一致性呢？ 这个时候就需要使用一种实现了一致性协议的服务了。Zookeeper就是这种服务，它使用Zab这种一致性协议来提供一致性。现在有很多开源项目使用Zookeeper来维护配置，比如在HBase中，客户端就是连接一个Zookeeper，获得必要的HBase集群的配置信息，然后才可以进一步操作。还有在开源的消息队列Kafka中，也使用Zookeeper来维护broker的信息。在Alibaba开源的SOA框架Dubbo中也广泛的使用Zookeeper管理一些配置来实现服务治理。

##### 名字服务

名字服务这个就很好理解了。比如为了通过网络访问一个系统，我们得知道对方的IP地址，但是IP地址对人非常不友好，这个时候我们就需要使用域名来访问。但是计算机是不能是别域名的。怎么办呢？如果我们每台机器里都备有一份域名到IP地址的映射，这个倒是能解决一部分问题，但是如果域名对应的IP发生变化了又该怎么办呢？于是我们有了DNS这个东西。我们只需要访问一个大家熟知的(known)的点，它就会告诉你这个域名对应的IP是什么。在我们的应用中也会存在很多这类问题，特别是在我们的服务特别多的时候，如果我们在本地保存服务的地址的时候将非常不方便，但是如果我们只需要访问一个大家都熟知的访问点，这里提供统一的入口，那么维护起来将方便得多了。

##### 分布式锁

其实在第一篇文章中已经介绍了Zookeeper是一个分布式协调服务。这样我们就可以利用Zookeeper来协调多个分布式进程之间的活动。比如在一个分布式环境中，为了提高可靠性，我们的集群的每台服务器上都部署着同样的服务。但是，一件事情如果集群中的每个服务器都进行的话，那相互之间就要协调，编程起来将非常复杂。而如果我们只让一个服务进行操作，那又存在单点。通常还有一种做法就是使用分布式锁，在某个时刻只让一个服务去干活，当这台服务出问题的时候锁释放，立即fail over到另外的服务。这在很多分布式系统中都是这么做，这种设计有一个更好听的名字叫Leader Election(leader选举)。比如HBase的Master就是采用这种机制。但要注意的是分布式锁跟同一个进程的锁还是有区别的，所以使用的时候要比同一个进程里的锁更谨慎的使用。

##### 集群管理

在分布式的集群中，经常会由于各种原因，比如硬件故障，软件故障，网络问题，有些节点会进进出出。有新的节点加入进来，也有老的节点退出集群。这个时候，集群中其他机器需要感知到这种变化，然后根据这种变化做出对应的决策。比如我们是一个分布式存储系统，有一个中央控制节点负责存储的分配，当有新的存储进来的时候我们要根据现在集群目前的状态来分配存储节点。这个时候我们就需要动态感知到集群目前的状态。还有，比如一个分布式的SOA架构中，服务是一个集群提供的，当消费者访问某个服务时，就需要采用某种机制发现现在有哪些节点可以提供该服务(这也称之为服务发现，比如Alibaba开源的SOA框架Dubbo就采用了Zookeeper作为服务发现的底层机制)。还有开源的Kafka队列就采用了Zookeeper作为Cosnumer的上下线管理。

#### Solr集群的结构



#### Solr集群的搭建

SolrCloud结构图如下：

本教程的这套安装是单机版的安装，所以采用伪集群的方式进行安装，如果是真正的生产环境，将伪集群的ip改下就可以了，步骤是一样的。



需要三个zookeeper节点

四个solr节点。

使用伪分布式实现solr集群。需要三个zookeeper实例，4个tomcat实例，可以在一台虚拟机上模拟。建议虚拟机1G以上内存。

## 关于加密

### 加密类型

#### 对称加密

又称作私钥加密。（举例：家里大门开门和锁门用的是同一把钥匙）加密和解密使用相同的密钥。密钥：在加密算法当中，将明文转换为密文，或者密文转换为明文的时候，需要用到的参数。分为公钥和私钥。

常见的对称加密算法有：AES(wifi密码)、DES(wifie密码)、RC5(电视遥控器)

\* DES Data Encryption Standard

\* AES: Advanced Encryption Standard

**特点：**

在对称加密算法中，双方使用的密钥相同，要求解密方事先必须知道加密密钥。其特点是算法公开、计算量小、加密速度快、加密效率高，适合加密比较大的文件。不足之处是，通信双方都使用同样的密钥，安全性得不到保证。此外，用户每次使用该算法，需要保证密钥的唯一性，使得双方所拥有的密钥数量很大，密钥管理较为麻烦。同样，只有一把密钥,如果密钥暴露,文件就会被暴露。

#### 非对称加密

与对称加密算法不同，非对称加密算法需要两个密钥：公开密钥和私有密钥。每个人拥有这两个密钥，公开密钥对外公开，任何人都可以获得；私有密钥不公开，单独给指定用户的。如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；如果用私有密钥对数据进行加密，那么只有对应的公开密钥才能解密。

也就是说，如果某个用户得到了公钥加密后的信息，只能用该用户的解密私钥才能解密。如果知道了其中一个，并不能计算出另外一个，因此如果公开了一对密钥中的一个，并不会危害到另外一个的秘密性质。

RSA加密算法是最常用的非对称加密算法。另外，还有美国国家标准局提出的DSA。

**特点：**

非对称加密算法的保密性比较好，消除了最终用户交换密钥的需要，但加密和解密花费时间长、速度慢，它不适合于对文件加密而只适合于对少量数据进行加密。

#### 单项加密

该算法在加密过程中不需要使用密钥，输入明文后由系统直接经过加密算法处理成密文，密文无法解密。只有重新输入明文，并经过同样的加密算法处理，得到相同的密文并被系统重新识别后，才能真正解密。

**特点：**

该算法计算复杂，通常只在数据量有限的情形下使用，如广泛应用在计算机系统中的口令加密。近年来，单向加密的应用领域正在逐渐增大。

**常见的：**MD5加密（RSA公司发明），SHA(Secure Hash Algorithm)(美国国家安全局设计）

### 加密算法使用

#### RSA

什么是RSA加密？

RSA加密算法是最常用的非对称加密算法。也是目前为止最安全的非对称加密算法。

特点:加密速度比较慢一些,但是安全系数比较高。

秘钥对的话需要程序生成.不能我们自己定义

加密/解密:公钥加密-->私钥解密

> public static byte[] encryptByPublicKey(byte[] data, String publicKey)`//公钥加密

> public static byte[] decryptByPrivateKey(byte[] encryptedData, String privateKey) `//私钥解密

加密/解密:私钥加密-->公钥解密

> public static byte[] encryptByPrivateKey(byte[] data, String privateKey) `//私钥加密

> public static byte[] decryptByPublicKey(byte[] encryptedData, String publicKey)//公钥解密

在实际开发中，都有相应的工具提供RsaUtils，不需要去操心算法的内部实现原理（互质关系、欧拉函数、欧姆定理等）。这些都是数学家们花了几十年搞出来的心血。

**名字的来源**

RSA是1977年由罗纳德?李维斯特(Ron Rivest)、阿迪?萨默尔（Adi Shamir）和伦纳德?奥德曼（Leonard Adleman）一起提出的。RSA就是他们三人姓氏开头字母拼在一起组成的。

#### MD5

全称是Message-Digest Algorithm 5，在90年代初由MIT的计算机科学实验室和RSA Data Security Inc发明，经MD2、MD3和MD4发展而来。

它是哪种机密类型？根据特点而定

md5算法可用于对一段信息进行不可逆的变换运算，产生一个32位的字符串。如果对输入信息作了任何形式的改变，对改变后的信息再次进行MD5运算所产生的数字摘要，与改变前的内容所运算出的MD5数字摘要都是不同的。MD5不是加密，因为结果是不能恢复出原始数据的。

**MD5加密的特点**

MD5加密既不是对称加密算法，也不是非对称加密算法，属于单向加密算法。因为它不可解密，只能通过匹配的方式间接解密。因此安全系数最高。

**开发中如何使用MD5加密？ MD5Utils工具类**

MD5加密的过程：

1. 先将指定的字符串转换为一个16位的byte[]

2. 遍历取出数组中的每个byte元素

3. 将取出的byte值与255(0xff)做与运算(&)后得到一个255以内的数值

4. 将得到的数值转换为16进制的字符串, 如果它只有一位, 在它的前面补0

5. 将生成的16个二位16进制形式的字符串连接起来, 它就是md5加密后的32位字符串

**Dom4j和sax的区别**

**Dom4j和sax的区别**

## Dom4j和sax的区别

这两个都是用来解析xml的

Dom4j简单易用，他是一下子将文件加载到内存中，不适合大文件的解析，但是灵活性好，

SAX相较于dom来说速度更快，更有效。它是逐行扫描文档，一边扫描一边解析，可以在任意时刻停止解析，适合解析大型xml文件，但是操作相对dom来说就要复杂一些。

# 话术

## Ajax请求Session超时问题

我在做项目时有时会遇到session超时问题，如果session超时，平常请求没有什么问题，通过拦截器可以正确跳到登陆页面，可是你如果用ajax请求的话这就出现问题了，因为ajax是异步的，局部刷新，所以登陆界面不会再全页面中显示，他只会显示到页面的一部分当中。所以根据我这几年的经验找到了我认为比较好的一种方法。因为那我用的框架是和struts2集成的，所以就在拦截器中进行设置：

首先判断session是否为空就是判断session是否超时，如果超时就取出请求的head头信息request.getHeader("x-requested-with")，如果不为空就和XMLHttpRequest(Ajax标识)进行比较 (request.getHeader("x-requested-with").equalsIgnoreCase("XMLHttpRequest"))) 如果相等说明此请求是ajax请求。

如果是ajax请求就可以用response.setHeader("键","值")来设置一个标识来告诉用户这是ajax请求并且session超时发出的，这样我就可以在回调函数中取出自己设置的那个唯一标识：XMLHttpRequest.getResponseHeader("");如果取出的值是和自己在后台中设置的值一样的话，就证明session已经超时，这样就可以设置window.location.replace("登陆界面")，来跳转到登陆界面了。

这样做虽然解决了问题，但是，会在每个回调函数中写入那些代码，这样的话代码就会显得特别零散，所以就想能不能定义一个全局的设置所以就找到了jqery的ajaxSetUp方法，通过ajaxSetUp对jqery的ajax进行全局的判断(ajaxSetUp就相当于ajax的拦截器)，通过设置ajaxSetUp里的complete，它就相当于回调函数，这样那就弥补了上一方法的不足。

我做项目时还用到$(document).ajaxStart()，这是ajax请求时的事件；$(document).ajaxSuccess()，这是AJAX请求成功后的事件。我一般用他们来显示遮罩层和隐藏遮罩层用的加遮罩层是为了不让用户重复提交，更提高了用户体验度，让用户知道已经提交了。

## Servlet

Servlet是一个web容器，我们通常用的servlet是httpservlet，而httpservlet又是继承于genericservlet，而genericservlet又实现了servlet接口

servlet的生命周期是 ：先进行实例化，然后是初始化，然后是提高服务，然后销毁，最后不可用，在这五个生命周期，其中，初始化是调用的init方法，这个方法只有一个，而提高服务的时候调用的是service方法，而我们具体在我们所写的这个方法中，因为我们继承了httpservlet，其实就是对应了doGet（），doPost(),这种方法，然后据我了解，servlet是单例的。非线程安全的，我们通常有一下几种方案来解决：

第一种，继承SingleThreadModel但是这样每次都会创建一个新的servlet实例，但这样消耗服务器的内存，降低了性能，并且这个接口现在已经过时了，不推荐使用。

第二种：我们尽量避免使用全局变量，就我个人而言，我比较喜欢使用这种方法。

第三种，我们可以通过使用ThreadLocal， 内部结构是一个Map结构，用当前线程作为key,他会创建多个副本。get,set方法

第四种，我们当然还可以来加锁，进行解决线程问题。

而且我还知道，向我们这种常用的MVC框架，struts1，spring这些MVC框架，都是基于servlet发展而来的，就比如struts1 的核心总控制器是ActionServlet，而springMVC的前端总控制器是dispatchServlet，在项目我们曾经用serlet来生成 图片验证码的，防止用户进行暴力破解

（别人问了，再回答）

servlet的配置文件 web.xml

<servlet>

<servlet-name>ImageCodeServlet</servlet-name>

<servlet-class>org.leopard.code.ImageCodeServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ImageCodeServlet</servlet-name>

<url-pattern>/d</url-pattern>

</servlet-mapping>

**描述：**

我在web.xml中，我首先需要写一个servlet标签，servlet标签中有两个子标签，一个叫servlet-name，这个name可以随便起，但是要保证唯一性，除此之外，在这个servlet-name下有一个servlet-class，这个servlet-class对应的就是我后台提高服务的servlet，除此之外还有一个servlet-mapping，这个里边首先有一个servl-name。，这个servl-name首先要保证和上边的servlet-name保持一致，除此之外还有一个url-pattern，这是一个虚拟路径，是用来发送请求的url地址

## 介绍所过的项目中遇到的特别难解决的问题

我们的项目在测试阶段呢，首先是功能测试，然后是性能测试，最后是仿真测试；在功能测试和性能测试的过程中，项目运行良好，没有出现问题；但到了仿真测试阶段，在系统最初开始运行的一段时间呢，系统没有出现问题，7-8天后，系统开始出现运行缓慢的现象，在过一段时间之后，发现系统崩溃了。我们根据这个现象判断系统是出现内存泄露问题，所以使用了java内存检测工具jprofiler对系统内存进行检测，定位源码，发现内存中有大量的javabean占用资源，没有释放，随着系统运行时间的增长，这样没有释放的资源越来越多，最终导致内存泄露。

发现问题原因之后，我们开始着手修复，通过相应信息定位到代码块，发现我们在spring中定义了多个定时器，用来执行不同的操作，比如备份系统数据、数据转换、发送邮件等功能。通过百度搜索quartz，得知quartz有这样一个bug就是它会导致整个web应用的类加载器不能进行垃圾回收，在web应用关闭之后，会看到这个应用的所有静态类资源。也就是说，系统退出的时候没有释放相应资源。

Spring提供了一个名为org.springframework.web.util.IntrospectorCleanupListener的监听器，主要负责处理由于JavaBeans Introspector使用引起的内存泄露。这个监听器是在web应用关闭的时候。清除JavaBeans Introspector的监听器。Web.xml中注册这个listener，可以保证在web应用关闭的时候释放掉与这个web应用相关的class loader和由他管理的类。

所以，解决上述问题的办法就是，在web.xml中加入

<listener>

<listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-class>

</listener>

修复完成之后，我们重新进行了仿真测试，对测试跟踪一段时间之后，发现内存占用一直保持在一个比较稳定的状态，没有出现内存泄露问题，这个问题得到解决

我们知道servlet标准不允许在web容器内自行做线程管理，quartz的问题确实存在。对于web容器来说，最忌讳应用程序私自启动线程，自行进行线程调度，像quartz这种在web容器内部默认就自己启动了线程进行一步job调度的框架本身就是很危险的事情，很容易造成servlet线程资源回收不掉。

## 是否解决过项目上线的bug

解决过。在项目测试过程中，都是由一些专业的操作人员去进行操作的，思维有一定的模式，在操作系统的时候也会按照一定的规则去操作。而项目上线之后，面对的是普通用户，可能因为普通用户的不正确操作导致项目出现bug，这也是我们在测试过程中没有考虑到的，当用户把这个bug反馈给我们的时候，我们马上进行了排查。比如说由于一些用户的不正确操作，导致系统数据丢失，我们在解决bug的过程中对用户操作进行了验证，在页面提示用户可以进行的正确操作，或者通过页面提示引导用户执行正确的操作，另外，如果使用的是Oracle数据库的话，我们可以使用oracle闪存来修复误操作的数据。

应用Flashback Query查询过去的数据

Flashback Query这一特性，最常被应用的就是修复误操作的数据了。注意，这并不是说Flashback Query能够恢复数据。Flashback Query本身不会恢复任何操作或修改，也不能告诉你做过什么操作或修改，实际上Flashback Query特性实际应用时，是基于标准SELECT的扩展，借助该特性能够让用户查询到指定时间点的表中的记录，相当于拥有了看到过去的能力，至于恢复，SELECT的结果都出来了，难道还不懂如何执行INSERT TABLE SELECT或CREATE TABLE AS SELECT吗？

基于时间的查询

执行查询语句，查询距现在5分钟左右的数据

SELECT \* FROM 表名 AS OF TIMESTAMP SYSDATE-5/1440;

Sysdate是系统函数，用来取得当前系统时间，是以天为单位的，1440=24（小时）\*60（分钟）计算出每天一共有的分钟数，sysdate-5/1440就可以计算出距离当前时间5分钟前的记录，然后再将数据用insert语句新增到数据库中

## 说说对敏捷开发的认识

敏捷开发有如下特征:

1. 工作在小的团队中

2. 团队是跨功能的-包括测试人员，开发人员，文档开发人员等等

3. 短迭代-利用短迭代方法来交付软件

4. 相较于文档，敏捷开发更注重面对面的交流

5. 敏捷不是一个过程，而是一个软件开发的形式或者方法

6. 敏捷可以与软件过程如CMMI等一起实施

## 高并发解决方案

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。

### 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。

### 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。

### 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。

### 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。

### 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。

### 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。

### 最新：CDN加速技术

#### 什么是CDN？

　　 CDN的全称是内容分发网络。其目的是通过在现有的Internet中增加一层新的网络架构，将网站的内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容，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。

附：某CDN服务商的服务说明

#### 采用GCDN加速方式

　　采用了GCDN加速方式以后，系统会在浏览用户和您的服务器之间增加一台GCDN服务器。浏览用户访问您的服务器时，一般静态数据，如图片、多媒体资料等数据将直接从GCDN服务器读取，使得从主服务器上读取静态数据的交换量大大减少。

　　为VIP型虚拟主机而特加的VPN高速压缩通道，使用高速压缩的电信<==>网通、电信<==>国际（HK）、网通<==>国际（HK）等跨网专线通道，智能多线，自动获取最快路径，极速的动态实时并发响应速度，实现了网站的动态脚本实时同步，对动态网站有一个更加明显的加速效果。

每个网络运营商（电信、网通、铁通、教育网）均有您服务器的GCDN服务器，无论浏览用户是来自何处，GCDN都能让您的服务器展现最快的速度！另外，我们将对您的数据进行实时备份，让您的数据更安全！

## FreeMarker

FreeMarker是一个用Java语言编写的模板引擎，它是基于模板来生成文本输出的通用工具。Freemarker可以生成HTML， XML，JSP或Java等多种文本输出。

使用Freemarker的好处是可以提高开发效率，使得在开发的过程中的员工分工更加明确。

在我做的一个门户网项目里面，我利用FreeMarker技术将二级，三级页面静态化，在我做的项目里面，我会在Web.xml里面或者Spring里面配置一个监听器利用监听器将生成页面静态话的方法监听起来，这样就会在项目启动的时候利用监听器，将二级，三级页面生成静态页面也就是HTML页面，这样做也会有弊端就是在更新数据之后，静态页面还会显示旧的数据，这个我们可以在后期使用Spring定时间来解决这个问题,可定定时每天几点的时候重新生成一次静态页面.

**注意：**

在用FreeMarker的时候我们要注意在模板中的数据的变量不可以为空，否则会抛异常

## Maven和ant的区别

我觉得最大的差别就是在于maven的编译以及所有的脚本都有一个基础，就是POM（project object model）。这个模型定义了项目的方方面面，然后各式各样的脚本在这个模型上工作，而ant完全是自己定义，显然maven更胜一筹。

第二：Maven对所依赖的包有明确的定义，如使用那个包，版本是多少，一目了然。而ant则通常是简单的inclde 所有的jar。

第三：Maven是基于中央仓库的编译，即把编译所需要的资源放在一个中央仓库里，如jar，tld，pom，等。当编译的时候，maven会自动在仓库 中找到相应的包，如果本地仓库没有，则从设定好的远程仓库中下载到本地。这一切都是自动的，而ant需要自己定义了。这个好处导致的结果就是，用 maven编译的项目在发布的时候只需要发布源码，小得很，而反之，ant的发布则要把所有的包一起发布，显然maven又胜了一筹。

第四：maven有大量的重用脚本可以利用，如生成网站，生成javadoc，sourcecode reference，等。而ant都需要自己去写。试试 maven site 的效果。

第五：maven目前不足的地方就是没有象ant那样成熟的GUI界面，不过mavengui正在努力中。目前使用maven最好的方法还是命令行，又快又方便。

## 图片及excel的导入导出怎样实现

**导入：**首先导出excel模板，让客户按照模板进行相关的数据信息录入，之后根据提供的导入功能将excel传到服务器端，服务器端对它进行解析。

**解析流程如下：**

首先后台创建一个Workbook对象，因为每个workbook都有与其对应的多个sheet对象，再依次解析每一个sheet对象，同理，每一个sheet对象又对应多个row对象，所以再依次解析每一个row对象，同理，每一个row对象又对应多个cell对象，再依次解析每一个cell对象取出要获取的相应数据值。即完成excel的导入功能。

**导出：**首先从后台数据库获取到要导出的全部数据，之后根据相应的导出功能将数据导到相应的excel表格中。

**流程如下：**先获取数据，之后先创建一个workbook工作簿，然后循环创建出需要的sheet，再根据查询出来的记录条数创建row，之后根据查询出来的每一条记录的属性个数创建出cell，并且给cell赋值，所有循环完成后将文件输出，即完成了excel导出功能。

Excel2003导出一个sheet最多只能65535行

**大数据量操作时的问题：**如果xls文件中数据量比较大时，将读入到的数据封装到对象中，再全部都放进List时，List对其中的对象都是强引用(Strong Reference)【强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。】，大量的数据会引发内存溢出异常，为解决这个问题，所以在工具类中每得到若干个对象后调用Service层的方法去保存这些数据。再重新给List赋值，再继续读取出数据。

POI导出Excel可以利用HSSFComment来生成批注信息。

//创建批注对象

HSSFComment comment =patriarch.createComment(new HSSFClientAnchor(0,0,0,0,(short)3,3,(short)5,8));

//设置批注内容

comment.setString(new HSSFRichTextString(column.getDescription()));

comment.setAuthor("作者名称");

//设置单元格批注信息

cell.setCellComment(comment);

**POI导出Excel生成下拉列表**

我们在导出的excel的时候，如果页面上是下拉框展示的数据，我们在导出excel时会生成相应的下拉列表。

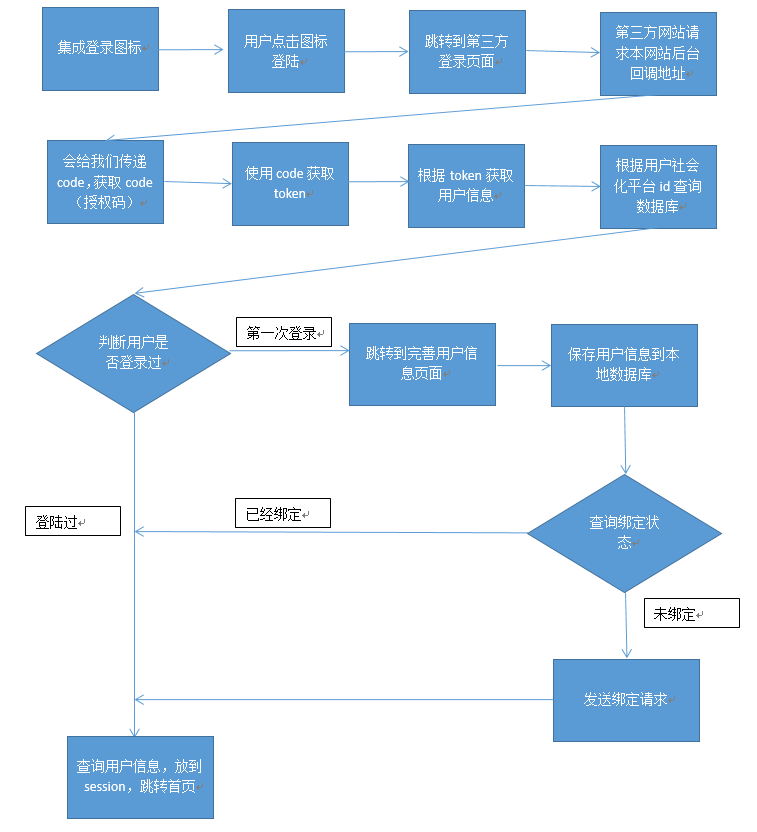
首先我们会生成一个隐藏的sheet，然后将下拉列表中的数据添加到隐藏sheet中，每一列的数据代表一个下拉列表，然后在要生成下拉列表的单元格上创建对隐藏sheet下拉列表的引用

//工作表添加验证数据 dataSheet.addValidationData(data\_validation\_list);

## 单点登录



## 第三方登录



## 支付

### 支付宝流程

1、首先拿公司资质去支付宝申请商铺号和秘钥，成功后支付宝会给你返回一个API，里面有商铺号、秘钥、支付的URL、回调的URL格式、传递参数的规格等。

2、点击“支付”，会跳进自己写的一个controller方法中，从页面传过去的有订单号和价格总和，再把商铺号、秘钥、回调的URL一起add到一个list里。

3、用httpclient把list根据支付宝给的支付URL传递给支付宝。

4、此时支付宝会把你传递过来的商铺号和秘钥进行比对，如果成功，支付宝会以html的方式返回一个选择银行支付的界面。你进行支付就OK。

5、你支付完成后，支付宝会把流水号、订单号、价格返回，此时你controller的回调方法里会把这些数据进行比对，都正确后给支付宝发送一个“SUCCESS”，支付宝就知道支付已经成功，页面就跳转到支付成功页面。反之就跳转到支付失败页面（这里所有支付宝返回给我们的页面都是支付宝自己的页面，我们不需要写）

6、最后把流水号等相关数据拿去数据库进行一个操作。我们是订单表和购物车表为一张表，此时只需要根据订单号把订单状态从0（购物车）改为1（付款后的订单），然后再把流水号存入。此时整个支付就完成了。

流水号主要负责的是商城数据库的价格与第三方支付方价格进行对比！！成功则审核通过并发货！

## Dubbo

首先我们要知道dubbo有两个概念，一个是提供服务方，一个是消费方，我们也叫提供服务者，和消费者。Zookeeper就是dubbo的服务中心，首先服务方要编写出真正的实现，把接口和VO对象打成jar包，为消费者提供依赖，编写完真正的实现后我们需要注册到zookeeper服务中心，然后消费者连接zookeeper注册相应的地址引用相关的interface，这就是使用dubbo和zeekeeper的简易流程，使用dubbo和zookeeper的好处：比如我们有pc端，APP服务端，后台和红包系统都需要进行用户的查询，而且每一个查询都会打开一个数据库连接而且会有重复的代码工作，需要我们进行优化，而且如果不同的模块单独开发，实现的方式实现逻辑不同导致结果不一致。这样我们就可以使用dubbo提供服务，各个模块之间都可以使用，直接操作的是service接口而不是API。

## POI

### 内容

我当时在项目中实现excel导入导出，采用的是POI来实现的，中间也碰到了一些问题，

比如当时我们在excel导入的过程中，碰到了2003版本和2007版本的不兼容问题，我们使用POI进行解析的时候，它对excel 2003版本和excel 2007版本的解析采用了不同的核心类，2003版本都是HSSF开头的类，2007版本都是XSSF开头的类，这样，对于2003和2007就需要写两套代码，后来我研究了一下，发现虽然03和07的类不同，但是都是继承自相同的接口，所以，我们在编码中都使用接口类型的变量，这样保证一套代码就可以兼容。

还有导出的时候，中文乱码的问题，当时我们在excel导出的过程中，碰到了文件名中文乱码的问题，而且这个问题在IE和火狐浏览器下的表现情况都不一样，当时，我们通过获取客户端浏览器类型的方式，针对不同的浏览器进行不同的中文乱码处理，解决了这个问题。

对于一些复杂的excel报表，比如包含跨行，跨列的内容，是先给定一个矩形区域，使用Region对象设置起始行列，结束行列，然后，在这个矩形区域中添加内容，最后，将矩形区域Region对象单独添加到工作表Sheet,

还有样式，是有一个接口对象CellStyle, 可以设置前景色，背景色，对齐，字体，高度宽度等等

还有公式，是使用单元格Cell的setCellFormula方法设置一个公式的字符串

还有数字日期格式化，是通过一个DataFormat来进行设置。

其实导出excel，使用一些框架，比如POI来做，有些时候还是比较麻烦的，一方面代码量比较大，另一方面，维护的时候，比如修改了一个东西，可能需要全部重来，我们当时在项目中，因为需求变化比较大，当时我们采用了另外一种方式来实现，

使用excel模板 + freemarker的方式来做，具体的做法就是，先创建一个excel的模板，另存为xml文件，然后将模板文件中的动态数据，使用freemarker的标签进行替换，最后，使用freemarker的代码导出excel文件。这样，即使excel的内容修改了，只需要修改模板，以及里面的freemarker标签就行了。开发和维护都很方便，只需要使用Freemarker的Template模板类调用process方法，传入Map结构的数据和输出流对象（可以是文件输出流到硬盘，可以是reponse的输出流，到客户端）就OK了。比如当时有一个报表，编写代码可能需要1天的时间，采用这种方式当时只需要半个小时就OK了。

当时，项目中也有需要导出word格式的文件，我们也是采用这样的方式来实现，先创建一个word的模板，另存为xml文件，然后将模板文件中的动态数据，使用freemarker的标签进行替换，最后，使用freemarker的代码导出word文件。

### 说明

#### Excel的核心接口

WorkBook工作簿, Sheet工作表, Row行, Cell单元格

WorkBook可以添加Sheet，获得Sheet,

Sheet可以添加Row，获得Row,

Row可以添加Cell,获得Cell,

Sheet还可以添加Region

Region表示一个矩形区域，用来实现跨行跨列

#### Excel导入兼容03和07版本的问题

**try** {

// HSSFWorkbook->excel文件

Workbook wb = **null**;

**try** {

wb = **new** XSSFWorkbook();

} **catch** (Exception ex) {

wb = **new** HSSFWorkbook(**new** FileInputStream(path));

}

// HSSFSheet->excel文件中的某个工作表

Sheet sheet = wb.getSheetAt(0);

**int** index = sheet.addMergedRegion(**new** CellRangeAddress(1, 4, 3, 3));

sheet.getMergedRegion(index);

// 得到实际的行数

**int** rowCount = sheet.getPhysicalNumberOfRows();

**int** columnCount = 0;// 变量->列数

System.*out*.println("实际的行数:" + rowCount);

**for** (**int** i = 0; i < rowCount; i++) {

Row row = sheet.getRow(i);

**if** (i == 0) {

**continue**;

}

columnCount = row.getPhysicalNumberOfCells();

User u = **new** User();

u.setId(Long.*valueOf*((**int**)row.getCell(0)

.getNumericCellValue()));

//...

users.add(u);

}

System.*out*.println(users);

// 数据库批量增加用户的业务逻辑代码

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

#### Excel导出文件名中文乱码问题

//处理火狐浏览器文件名问题

String browser = request.getHeader("USER-AGENT");

**if** (browser.indexOf("Mozilla") >= 0) {

fileName = javax.mail.internet.MimeUtility.encodeText(

"用户表数据.xls", "UTF-8", "B");

}

**if**(browser.indexOf("MSIE")>=0){

fileName = java.net.URLEncoder.encode("用户表数据.xls",

"UTF-8");

}

//设置content-type,一般设置为流的方式对所有的文件类型通用

response.setContentType("application/octet-stream;charset=UTF-8");// 不同类型的文件对应不同的MIME类型

//设置header,文件名

response.setHeader("Content-Disposition",

"attachment; filename=" + fileName);

#### Excel合并单元格，跨行跨列

~~Region~~ region3 = **new** Region(0, (**short**)2, 1, (**short**)2);

sheet.~~addMergedRegion~~(region1);

#### Excel设置单元格样式

单元格边框分为上、下、左、右四部分，可以设置其边框的宽度、样式及颜色。

边框默认为无边框，若加上边框默认为黑色。

现在为一行10个单元格全部加上边框（细），并设置颜色为红色

CellStyle style = workbook.createCellStyle();

//设置上下左右四个边框宽度

style.setBorderTop(HSSFBorderFormatting.*BORDER\_THIN*);

style.setBorderBottom(HSSFBorderFormatting.*BORDER\_THIN*);

style.setBorderLeft(HSSFBorderFormatting.*BORDER\_THIN*);

style.setBorderRight(HSSFBorderFormatting.*BORDER\_THIN*);

//设置上下左右四个边框颜色

style.setTopBorderColor(HSSFColor.RED.*index*);

style.setBottomBorderColor(HSSFColor.RED.*index*);

style.setLeftBorderColor(HSSFColor.RED.*index*);

style.setRightBorderColor(HSSFColor.RED.*index*);

cell.setCellStyle(style);

#### Excel单元格公式

cell.setCellFormula("F"+(rowIndex+1)+"+G"+(rowIndex+1)+"+H"+(rowIndex+1)+"+I"+(rowIndex+1));

#### Excel模板 + Freemarker导出excel

注意：图片处理不了。

String templateDir = ExcelData.**class**.getResource("/").getPath();

templateDir = templateDir.replaceAll("/build/classes/", "/src/");

templateDir += "demo/j2ee/data/excel";

**try** {

// 得到freemarkder模板

File dir = **new** File(templateDir);

Configuration config = **new** Configuration();

config.setDirectoryForTemplateLoading(dir);

config.setObjectWrapper(**new** DefaultObjectWrapper());

Template template = config

.getTemplate("demo\_excel\_freemarker.xml");

// 构造测试数据

Map<String, Object> dataMap = **new** HashMap<String, Object>();

List<Map<String, Object>> emps =

**new** ArrayList<Map<String, Object>>();

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

Map<String, Object> emp = **new** HashMap<String, Object>();

//...

}

// 指定word文件生成的路径

String wordPath = ExcelData.**class**.getResource("/").getPath();

wordPath = wordPath.replaceAll("/build/classes/", "/src/");

wordPath += "demo/j2ee/data/excel/demo\_excel\_freemarker.xls";

// 写文件之前,需要判断文件及文件所在的文件夹是否存在

// 省略判断文件所在的文件夹是否存在

// 判断文件是否存在

File f = **new** File(wordPath);

//...

// 使用freemarker生成word

FileWriter out = **new** FileWriter(wordPath);

dataMap.put("emps", emps);

template.process(dataMap, out);

} **catch** (Exception e) {

e.printStackTrace();

}

## POI与JXL对比

Poi和jxl相比，如果是基于大数据量而言，poi更加消耗内存，而且相差的不是一星半点，但是功能方面，poi则更加强大，所以如果要实现的功能比较复杂的情况下可以考虑使用POI,但如果只想生成一些大数据量可以考虑使用JXL

就这两者的区别,主要谈下JVM虚拟机内存消耗的情况.

数据量3000条数据,每条60列.JVM虚拟机内存大小64M.

使用POI:运行到2800条左右就报内存溢出.

使用JXL:3000条全部出来,并且内存还有21M的空间.

可想而知,在对内存的消耗方面差距还是挺大的.

也许是由于JXL在对资源回收利用方面做的还挺不错的.

关于两者效率方面,没有研究过,我想这个也是基于大数据量而言的,数据量小的话基本上差别不大,也不难被发觉.但是大的数据量,POI消耗的JVM内存远比JXL消耗的多.但相比提供的功能的话,JXL又相对弱了点.所以如果要实现的功能比较复杂的情况下可以考虑使用POI,但如果只想生成一些大数据量可以考虑使用JXL

## PDF

### 内容

当时在项目中，需要导出PDF格式的文件，使用的是itext这个框架，使用它的时候，需要注意导入语言包，itext-asia，可以处理中文乱码的问题，这个框架也比较好用，对于文本，一般是先设置段落，在增加文本内容，也可以添加表格，图片等等。添加表格可以使用PdfPTable，还可以在表格的单元格中添加图片。

### 说明

// 设置段落对齐,添加表格

@Test

public void testWriteComplex() {

String path = PdfData.class.getResource("/").getPath();

path = path.replaceAll("/build/classes/", "/src/");

path += "demo/j2ee/data/pdf/demo\_pdf\_write3.pdf";

// 模拟拿到数据库数据,封装到实体类对象中

Emp emp = new Emp(new Long(1), "阿狸", new Date(), "1", new Date(),

"3000", "13088888888", "ali@sohu.com", "12345678", "12345678");

OutputStream os = null;

// Step 1—Create a Document.

Document document = null;

// Step 2—Get a PdfWriter instance.

try {

os = new FileOutputStream(path);

document = new Document();

PdfWriter.getInstance(document, os);

// Step 3—Open the Document.

document.open();

// Step 4—Add content.

BaseFont bfChinese = BaseFont.createFont("STSong-Light",

"UniGB-UCS2-H", BaseFont.NOT\_EMBEDDED);

// 页面标题

Font titleFont = new Font(bfChinese, 15, Font.NORMAL);

Paragraph title = new Paragraph(emp.getName(), titleFont);

title.setAlignment(Element.ALIGN\_CENTER);

document.add(title);

document.add(new Paragraph(" "));// 空白行

System.out.println(DateFormatUtils.format(emp.getBirthdate(),

"yyyy-MM-dd"));

// 页面表格

Font tableFont = new Font(bfChinese, 12, Font.NORMAL);

PdfPTable table = new PdfPTable(2);

table.setWidths(new float[] { 0.3f, 0.7f });

table.addCell(new Phrase("出生日期", tableFont));

table.addCell(new Phrase(DateFormatUtils.format(emp.getBirthdate(),

"yyyy年MM月dd日"), tableFont));

table.addCell(new Phrase("性别", tableFont));

table.addCell(new Phrase("" + (emp.getGender() == "1" ? "男" : "女"),

tableFont));

table.addCell(new Phrase("头像", tableFont));

Image image = Image.getInstance(PdfData.class.getResource("/").getPath()

+"demo/j2ee/data/pdf/demo\_pdf\_table\_img.jpg");

PdfPCell imageCell = new PdfPCell(image, true);

imageCell.setFixedHeight(50);

table.addCell(imageCell);

document.add(table);

} catch (Exception e) {

e.printStackTrace();

} finally {

//....

}

}

## 图片

使用jdk自带的javax.imageio包中对象，可以处理图片信息。

## 定时任务+web service同步数据

### 内容

当时，项目中需要将一些业务数据同步到其他的业务系统， 当时我们采用了定时调度+web service的方式。

其实定时调度，有4种做法，

第一种是JDK自带的Timer, 当时它实际上就是新开一个线程去实现的，当执行任务有异常的时候，定时任务都会出现问题。

第二种是JDK并发包中的任务调度的线程池ScheduledExecutorService，它可以捕捉到系统时间的变化，比如现在是10点，启动一个定时任务在11点执行，那任务应该等待1个小时候之后执行，如果现在把系统时间改为11点，这个任务不会再等待1个小时，会立即执行。

第三种是采用quartz框架，可以配置线程池，以及分布式。

第四种，是使用spring自带的task框架，这个是spring3.0之后加入的新功能，它会自己处理异常，当执行定时任务出现异常，不会影响整个定时任务。

我们当时使用的是spring自带的task框架，因为和spring整合比较方便，而且还支持使用cron表达式的方式进行定时任务的配置，我们当时是设置在每天凌晨3点执行同步数据的任务。

当时我们采用了oracle数据库的触发器，在需要同步的数据表上，创建3个触发器，分别是针对插入insert, 修改update，删除delete的操作，将这些操作以及操作的数据都添加到一个临时表中。当执行定时任务的时候，查询临时表中的数据，将这些数据同步到另一个系统中。在同步完成之后，删除临时表中的数据。

同步数据使用的是web service的方式，采用的是CXF框架来实现。另一个系统作为服务端，发布一个web service的服务，我们这边，使用发布的wsdl文件生成客户端java类，调用接口，将数据同步过去。这里需要注意安全性问题，

一个是身份验证，当时我们事先和服务端约定一个用户名和密码，在发送webservice请求时，先用用户名 + 密码 + 时间戳 生成一个字符串，然后对字符串进行MD5得到一个数字签名，在发送请求时，带上用户名， 时间戳，以及数字签名，服务端得到之后，先通过用户名查询到服务端保存的密码，然后使用 用户名 + 密码 + 时间戳，然后MD5，得到另一个数字签名，将这个数字签名和客户端发送的数字签名进行比较，如果相等，表示合法。验证通过。

另一个就是业务数据的安全，我们当时对业务数据采用了JSON的数据格式，因为json格式相对于xml格式来讲，数据量要小。使用事先约定好的秘钥，采用AES的方式进行加密，发送请求的时候，只发送加密后的数据，服务端得到请求之后，先使用AES, 通过秘钥进行解密。

### 可能问到的问题

**数据量比较大的问题?**

当时我们的数量量不是很大，一天几千条，可以采用分页的方式，发送多次请求。

如果数据量特别大，可以采用流的方式传输，也可以通过web service 发送文件，或者，直接采用FTP的方式，将需要发送的数据打包成文件，通过ftp上传到服务端，服务端解析文件之后，进行处理。

**请求端发送重复的数据，服务端怎么解决?**

每条数据的主键采用自增长，临时表的数据不删除，增加一个状态字段，表示，待同步，同步失败，已经同步，服务端接收请求的时候，先判断，同步的数据是否已经同步过，这个判断，通过查询服务端的数据库得知，如果已经存放，服务端返回指定的状态码或者记录ID，告知客户端，已经发送过。

**数据的推与拉？什么叫推数据？什么叫拉数据？**

数据产生方A 数据接收方B

A->B 推数据，特点，快速，及时

B->A 拉数据，特点，需要轮询，有延时，而且很多时候发的请求得不到新数据。

## App应用网站简介（后台）

下面来介绍一下网站的后台，网站后台是网站工作人员运营维护网站的过程，主要是公司内部人员使用，所以不涉及高并发问题，为了提高开发效率我们采用了spring+jdbcTemelate+springmvc框架，页面采用了easyui。

### 用户、角色、权限

后台出于安全考虑，我们首先做的是权限，不同角色（我们网站的角色有：个人开发者，企业开发者，客服人员，推广人员，维护人员，系统管理员）的人登录应该看到不同的权限和内容，权限通常有3,5,7张表甚至更多来完成，我们当时用了五张，包括员工信息表、角色信息表、权限信息表和两张中间关系表：一张员工角色关系表，一张角色权限关系表。用户访问后台页面时，从session中获取用户id，通过用户id获取用户的所有角色id，再通过角色id获取所有权限url。把当前用户的权限信息用ztree以树形结构展示，就实现了不同的人登录展示不同的权限树。（当然，在这里可能有一些安全问题，就是用户直接在浏览器输入非法权限的url执行非法操作的问题，我们采用拦截器在用户执行操作时先判断该用户有无该权限，在有权限的情况下才放行，否则提示非法操作，终止该操作）。

### 第三方登录

第三方账号登陆采用国际通用的OAuth协议，允许各大互联网公司的账号在不泄露密码的前提下在其他网站登录，并获取相关信息。

一个网站要想集成第三方登陆功能，首先，我们需要先去各大第三方账号网站注册，获取唯一性id和secret安全码，然后在我们网站添加第三方账号登陆图标。

当用户点击某个第三方网站账号时，比如点击用QQ登陆，会跳转到QQ页面，让用户登陆。用户登陆后会跳转到我们网站的回调地址。并发送给我们一个code随机码。这样第一次握手通信完成。

然后我们再将接收到的code随机码和之前在腾讯注册的唯一性id和安全码发送给腾讯（利用httpclient发送post请求），然后腾讯给我们返回给一个token令牌,令牌中包含了第三方账号的相关信息。第二次数据交互完成。

然后利用token再给腾讯发送第三次数据请求，获取用户信息（用户名，头像，性别，住址），很多第三方网站为了保护用户隐私，返回的信息都不是很全面，所以我们需要根据用户的绑定状态去进行下一步的跳转。

接下来我们需要往第三方网站发送一个查询账号是否绑定的请求。根据账号的绑定信息判断第三方账号之前是否在我们网站登陆过。

假设是第一次登录，我们会给第三方网站发送一个绑定账号的请求，将我们网站与第三方账号绑定，之后跳转到用户信息完善页面，让用户输入邮箱和电话号码，从而获取用户更详细的个人资料。然后将第三方账号信息保存到我们网站自己的数据库中。

假设之前已经登陆过，我们会根据第三方账号信息去我们网站自己的数据库中查询出之前添加的信息，并保存到session中，然后跳转到首页，这样就完成了第三方账号在我们网站的登陆。

### 应用管理

应用管理包含我的应用，应用列表，应用类型三个功能。开发者可以通过我的应用对上传的应用进行维护和更新。网站工作人员通过应用列表对网站的所有应用进行统一维护。应用类型分为软件和应用两大类。每个大类下面又细化分了很多小类型。整个应用管理模块中主要用到了四张表，包括应用类别表、应用表、应用截图表、应用评论表：应用表与应用类型表和应用截图表是一对多关系，应用表与应用评论表也是一对多关系。

### 专题管理

针对某些特定的人群或者特定节日专门定制的应用专题。专题包含专题标题，专题内容，专题图片等字段信息，一个专题包含多个应用，同一个应用可以属于多个专题。应用与专题之间是多对多的关系，所以也需要一张中间表来建立对应关系。

### 推广管理

运营一个网站是需要投入很多成本，所以在设计一个网站的时候也要考虑盈利模式。我们网站目前是考虑采用广告推广和应用合作的方式来进行盈利。

#### 广告推广

我们网站提供了不同位置的广告位为开发者的应用进行宣传。广告位主要有首页轮动图片宣传，热门下载排行榜宣传，网站侧边栏广告位宣传。当一个开发者上传的应用审核通过时，就可以进行应用推广。开发者点击推广按钮会弹出选择广告位页面，在页面上需要选择广告位置，推广时间，一个应用可以同时选择多个广告位置进行推广，不同的广告位价格是不一样的，（在后台有一个广告位价格设置功能，网站工作人员需要根据竞争对手网站的价格和自身网站以及时间等各方面的因素来随时调整广告位价格）我们系统会根据开发者的选择实时计算出应该支付的广告费用，当开发者点击确认推广按钮后，就需要进行网银结算。网银结算主要用到订单表和订单详情表。订单与订单详情表是一对多关系。订单表主要保存支付信息，订单详情表主要保存单个广告的支付信息。

#### 网银结算

现在市面上主流的第三方支付平台有：支付宝，财付通，快钱等。我们网站目前对接的是财付通，中国银联，以及银行直连接口。选择财付通的一个主要原因是他的门槛低，支付费率低，后期网站还需要对接其他支付平台以及与银行之间的接口直连。（网站要想与第三方支付平台对接，需要先去该平台注册商户号，获取商户秘钥。）开发者点击确认推广按钮后，我们会在后台生成一个订单号，并且跳转到确认订单页面，这个页面显示了用户需要支付的金额以及购买的广告推广信息，用户点击支付按钮后，我们会给财付通支付接口发送一个请求，请求中包含商户号，秘钥，回调地址，加密签名（签名算法：MD5加密（编码+商户号+总价格））等支付信息。并且将支付信息保存到我们的订单表以及订单详情表中。同时页面也跳转到财付通的支付页面，用户支付完成后，财付通会把相关的支付结果发送给我们的回调地址，支付结果中包含一个通知ID，我们需要将通知ID作为参数调用财付通给我们提供的查询支付结果接口，查询支付信息是否正确。主动查询的目的是为了防止我们的回调地址泄露，不法分子恶意给我们发送虚假支付信息。查询支付结果接口会以xml的格式给我们返回相关查询信息，告诉我们该通知ID是否是由支付通发起，并且该支付信息是否有效。我们还需要根据用户的实际支付金额和理应支付金额做判断，如果金额一致并且通知ID有效，那我们就认为该用户支付成功。然后我们根据用户的支付信息跳转到相应的支付结果页面，并且更新数据库中的支付信息。这样整个支付流程就结束了。

延伸扩展：webservice调用物流接口或者库管接口查询库存，调配库存

秒杀（抢购）：用户支付完成后，不会立马送货，会有几个小时的空挡，利用空挡时间将非关系型数据库（redis,mongo）中的支付信息同步到关系型数据库中

#### 应用合作

应用合作的盈利模式我们目前也在探讨中，初步是打算精挑细选一些有潜力的应用进行商务合作，并且通过应用内的用户支付进行盈利分成。具体功能暂时还没有开发。

### 网站报表管理

#### 应用下载统计

网站上线后热门应用下载量可能会很大，如果用户下载一次应用就记录一次数据的话，会对数据库服务器产生很大的压力。所以我们在这里采用的是缓存+日志+定时器进行应用下载统计。

将应用的下载次数保存到缓存中，用户每下载一次应用，实时更新缓存中的数量，页面展示的时候直接从缓存中读取。并且采用log4j将用户的下载信息保存到日志表中，每天生产一个文件。然后在项目中配置quartz定时器，每天晚上1点分析前一天的日志，然后更新到数据库中。这样就能大大减少对数据库的频繁读写。

#### 网站访问统计

网站访问统计采用tomcat日志记录+百度地图，我们在tomcat中配置了日志记录功能，可以记录访问者的相关信息，我们配置的是每个小时生成一个日志文件。定时器每天晚上12点利用多线程分析访问日志，并且将访问信息保存到数据库中的访问日志表中，然后调用百度地图，百度地图可以根据IP地址转换成经纬度在地图上展示出来。

### 网络爬虫

网站测试阶段需要大量基础数据，我们为了丰富网站的内容，提高用户的体验度

### 网站定时任务

1.每天晚上12点分析昨天的用户下载日志，更新到数据库，更新到缓存中

2.每天晚上12点半分析昨天访问日志，保存到数据库

3.每天晚上1点定时根据昨天的下载量以及以往的下载量算出应用排名，然后将top10保存到memcached,前台页面展示时直接从缓存中去取

### 秒杀业务规划

秒杀这一块业务是其他同事负责的，要是让我设计的话，我会按照如下方式规划：

秒杀具有时间短，并发访问量大的特点，所以要是跟现有网站部署到一起的话，必然会对网站的其他业务造成影响，所以我们要规划一个单独的秒杀系统独立部署，甚至是独立域名，使其与本网站完全隔离。

其次在架构规划上要分为业务层，数据层。业务层是用户秒杀的层面，数据层是真实的库存数据。

业务层面采取的优化方案是：

**A：扩容**

　　加机器，这是最简单的方法，通过增加前端池的整体承载量来抗峰值。

**B：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。

大部分用户在秒杀开始前，通过不停刷新浏览器页面以保证不会错过秒杀，这些请求如果按照一般的网站应用架构，访问应用服务器、连接数据库，会对应用服务器和数据库服务器造成负载压力。所以我们要重新设计秒杀商品页面，不使用网站原来的商品详细页面，页面内容静态化，用户请求不需要经过应用服务。在秒杀期间网站流量也会增加很多，所以势必会对网站的整体网络带宽造成影响，所以我们要预先将秒杀页面缓存到CDN服务商（内容分发网络）

**C：限流**

　　一般都会采用IP级别的限流，即针对某一个IP，限制单位时间内发起请求数量。

　　或者活动入口的时候增加游戏或者问题环节进行消峰操作。

**D：有损服务**

最后一招，在接近前端池承载能力的水位上限的时候，随机拒绝部分请求来保护活动整体的可用性。

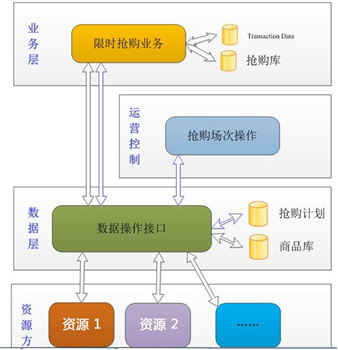
我们还需要在业务层面虚拟一个抢购库，相当于库存库的副本。抢购库可以用redis或者memcache、mongodb来实现，因为在真实的抢购过程中是需要频繁的往数据库查询库存请求，这样设计能大大减轻对数据库的压力。

抢购开始后，用户请求到了后台会统一放到redis队列里面，（有点类似于把多线程强行变成了单线程），然后再更新的时候只是更改抢购库，利用程序一步写入真正的mysql数据库。

在这里我们会采用乐观锁的机制，防止产生超卖的情况。在每次操作的时候都会获取一个版本号，更新的时候只有版本号一致才能更新成功。

（乐观锁实现：使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。）

业务规划图：



系统架构图：



**部分细节问题：**

直接下单秒杀的游戏规则是到了秒杀才能开始对商品下单购买，在此时间点之前，只能浏览商品信息，不能下单。而下单页面也是一个普通的URL，如果得到这个URL，不用等到秒杀开始就可以下单了。解决方案：为了避免用户直接访问下单页面URL，需要将该URL动态化，即使秒杀系统的开发者也无法在秒杀开始前访问下单页面的URL。办法是在下单页面URL加入由服务器端生成的随机数作为参数，在秒杀开始的时候才能得到。

### 微信公众号

目前我们的企业号还没申请下来，所以我们在开发的时候都是采用的测试号开发。

技术点：

微信公众号在开发时主要就是调用微信给我们提供的各种接口。所以我们在开发的时候采用的httpclient技术调用微信接口，发送get或者post请求，进行xml或者json格式数据的交互。

功能点：

创建自定义菜单：按照3\*5的数量创建自定义菜单，（常用按钮类型有超链接按钮、普通点击按钮、扫码按钮，选择图片按钮，发送地理位置按钮）

订阅推送：用户订阅完成以后，向用户被动推送图文消息

（流程描述：用户订阅公众号后，腾讯服务器会给我们的应用程序服务器发送一个关注（订阅）事件，我们应用程序能获得用户的OpenID以及事件类型等信息，然后调用查询素材接口获得要发送图片的MediaID,然后拼凑成图文消息的格式（图文消息个数，标题，描述，图片链接等）返回给微信服务器，然后微信服务器以图文消息推送给用户）

绑定用户：

将微信账号与电商用户进行关联，并且将关联信息（openid,手机号码，邮箱）通过调用Web端电商项目提供的WebService接口保存到电商数据库中

消息推送：

用户在Web端电商消费以后，通过ActiveMQ发送订单消息给微信项目，其中包含了订单金额，消费时间等信息。微信项目收到ActiveMQ消息后，调用客服接口，生成文本消息并主动推送信息到用户微信。

微信支付(红包)：

微信支付接口测试账号是没有权限的，需要认证微信支付商户成功以后才可以。我们公司现在在申请阶段。这个东西也不是很难，根据接口API一步步的调试一下基本上就可以出来

（红包场景：购买返现：我们商城搞活动，购买商品满多少钱就返现百分之1左右，付款成功后会给用户生成一个红包码，用户需要关注我们的微信公众号，然后点击领取红包，跳转到输入红包码页面，用户输入红包码，请求到了微信项目后台，微信项目会调用Web电商项目提供的WebService查询红包接口来判断该红包是否有效，Web电商项目会先去电商数据库查询红包码是否有效，红包码可以是订单表的一个字段，然后计算出该用户的返现金额，然后返回给微信项目。微信项目服务器获取信息后调用微信支付接口，从对应的商务号扣取对应的金额，在扣款之前会先检查一下商户号余额，扣款成功后会返回xml格式字符串，其中有一个return\_msg字段，根据这个字段去判断支付是否成功。通过他交互的字段可以看出，微信发红包就是相当于通过我们的商户号下了一个订单而已。具体技术实现都是通过httpclient发送get或者post请求）

<http://www.cnblogs.com/txw1958/p/weixin-redpackets.html>

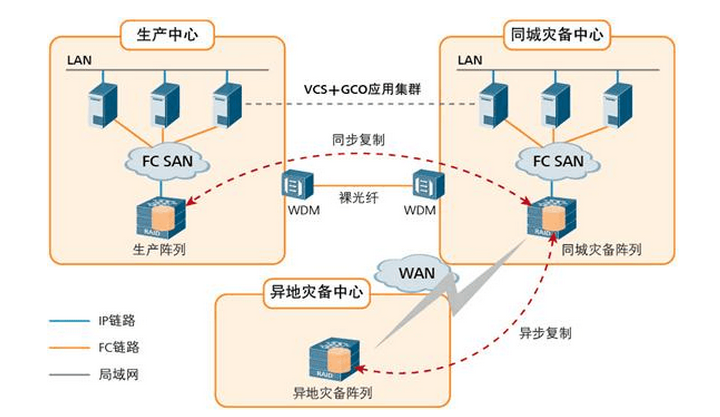
### WebService应用场景

Appweb项目首页有热门排行榜，系统通过quartz定时器定时统计出top10的app信息。然后放到Mongodb中。并且利用cxf对外发布webservice查询接口。

手机端和微信端在展示热门下载时，直接调用web端发布的接口，获取数据展示给用户。用户点击热门下载排行时，请求先发送到微信服务器，然后微信服务器给我们的应用服务器发送post请求，我们根据用户的请求，处理相关业务逻辑，返回数据给微信服务器，然后微信服务器再推送给用户展示。

### 网站架构

（我们网站还没有上线，目前是测试阶段。）



**两地三中心：**

两地指的是本地和异地。三中心:生产中心，同城灾备中心，异地灾备中心

我们借鉴了两地三中心灾备解决方案，我们分别从北京，杭州，深圳三个地域购买了阿里云服务器搭建集群。其中北京和杭州是本地，异地是深圳。生产中心在北京，同城灾备中心在杭州，异地灾备中心在深圳。

我们部署的时候采用的是nginx+tomcat集群+Mysql主从备份。总共用了7台服务器。Nginx两台，三个tomcat,mysql主从各两台。为了保证数据的安全性，我们采用mysql主从备份。找了一台配置比较高的服务器作为主服务器,从服务器在正常情况下只是一个备份服务器，mysql主服务器的数据实时备份到从服务器。

**MySQL主从搭建流程：**

1准备两台服务器，分别安装mysql，因为安装完mysql以后，默认只允许本机用户访问本机的mysql，不能远程访问该mysql，这样就无法实现远程备份。所以需要执行一条赋权的sql GRANT ALL PRIVILEGES ON \*.\* TO '用户名'@'%' IDENTIFIED BY '密码' WITH GRANT OPTION;

2.修改主服务器配置文件my.ini 配置要备份的数据库

3.修改从服务器配置文件my.ini 配置一下主服务器的ip地址和要同步的数据库

4.重启两个服务器，然后测试

Nginx总共有三种转发策略，一种是默认随机分配的，还有一种就是根据IP\_HASH算法将一个ip地址固定的转发到一台tomcat。还有一种就是根据权重比来分配。我们在项目中采用的是ip\_hash转发策略。为了提高用户的体验度，我们也采用javolution+memcached来实现seesion共享策略。（需要在每台tomcat服务器上都要安装memcached服务端）。当一个tomcat宕机以后，nginx会将请求自动转发给其他的服务器。在转发的同时，session信息同步复制，保证用户请求转发时的无登陆访问。

在我们的架构中，nginx和mysql主服务器都存在单机问题（单机问题：一台服务器宕机以后，没有备份服务器立即自动切换）。

为了防止主服务器由于意外原因宕机，我们采用keepalived实现了主从切换。当mysql主服务器意外宕机后，keepalived可以自动的切换到备份服务器，并且我们也配置了邮件发送策略，实时给工作人员发送邮件提醒。

（Keepalived的作用是检测web服务器的状态，如果有一台web服务器死机，或工作出现故障，Keepalived将检测到，并将有故障的web服务器从系统中剔除，当web服务器工作正常后Keepalived自动将web服务器加入到服务器群中，这些工作全部自动完成，不需要人工干涉，需要人工做的只是修复故障的web服务器。

优点：数据同步非常简单，不像负载均衡对数据一致性要求非常高，实现起来相对复杂维护也颇为不便，双机热备用rsync就可以实现了操作和维护非常简单

缺点：服务器有点浪费，始终有一台处于空闲状态 ）

**Keepalived安装过程（Linux）：**

准备两台服务器，ip地址分别是192.168.6.10 192.168.6.11

1.找到安装目录 make install

2.修改配置文件/keepalived.conf 在从服务器中配置主服务器的IP地址

3.配置发送邮件脚本 sendmail.pl 在里面配置邮件的接收人和内容

4.service keepalived start

5.遇见的问题：

当时在搭建成功后测试宕机邮件发送功能，发现邮件一直无法正常发送，最后跟踪了一下发现Keepalived的邮件发送功能是调用的系统的smtp服务，如果系统的smtp端口没有打开的话，邮件就会发不出去。所以我们需要开启25端口和80端口

其他的主要都是一些配置问题。

<http://www.2cto.com/os/201407/317795.html>

## 优惠券话术

来说一下优惠劵模块，在这个项目中我们采用自己定义的规则来实现这个优惠券功能，这样一来后期我们可以通过在后台系统中灵活的设置条件来生成我们业务所需要的优惠劵，具体流程就是当我们在后台设置好我们业务所需要的优惠劵条件之后，例如该优惠券的使用场景（商品类型，会场），使用时限（几月几日至几月几日），该优惠券使用规则类型（如：满x元可以使用；满x元包邮；满x元打y折；满X元送礼品；无消费限制使用），还有使用平台规则例如：只能在PC端使用，平台有（PC端，手机移动端，平板电脑移动端，第三方合作端：如微信，QQ等），还指定的使用的范围的要求，有平台自营店的全类别可用，自营店部分类别可用等。

指定的某个店铺内的全类别可用和店铺内部分类别可用，这是我们的优惠券的特性，然后我在说一说优惠券模块包含哪几个小模块。

我们的优惠券模块分为5个小模块，分别为：首页模块、好券集市、任务集市、专属优惠券、券集市。

### 首页模块

首页模块中主要有每日精选（优惠券秒杀）和好券集市部分优惠券的显示（按照时间和热门类型排序），以及链接到好券集市、任务集市、专属优惠券、券集市。

每日精选的业务流程是：当进入到首页时，以列表的形式显示每日精选的优惠券（暨秒杀优惠券），显示的属性有优惠券的图片，面值，使用条件以及其他使用条件和抢券活开始的倒计时（每种优惠券的倒计时时间不同），并且以百分比的形式显示出该券的剩余量，当剩余量为零时，显示成灰色并提示已抢完。抢券的业务流程是：当点击“立即领取”时，验证用户是否已登录，如果没有登录，将用户引导到登录页面进行登录，当登录后返回到优惠券首页，获取用户的会员等级，在后台根据用户的会员等级进行“摇奖”（概率运算）来获得领券的机会，如果“摇奖”失败，则提示用户抢券失败；如果“摇奖”成功，则在数据库用户与优惠券关系表中添加一条记录，并返回success，此时将“立即领取”按钮改变成“立即使用”，“百分比剩余量”显示为已领取，当点击立即使用时，跳转到优惠券指定使用类别的商品列表页，促使客户进行消费。

当点击下方好券集市中优惠券的“立即领取”按钮时，则显示领取成功，“立即领取”按钮改变成“立即使用”，跳转到相应优惠券指定的商品列表页。

该模块使用到的技术有：ajax异步请求来解决前台页面用户对优惠券秒杀的操作和Jquery对倒计时时间的动态显示，以及按钮的改变事件，后台使用spring定时器对首页商品信息做定时刷新（在秒杀时刷新，秒杀期间每5分钟刷新一次，以获取最新的数据）。

### 好券集市

好券集市客户端主要有各种优惠券的显示，各种类型、条件的优惠券的分页查询和优惠券的搜索功能。

当进入到好券集市时是以每页显示18张优惠券的格式展示。每张优惠券都有领取功能，当点击领取优惠券时会判断用户是否登录，如果没有登录就去登录页面进行登录。如果已登录就提示领取成功，然后已领取过的优惠券，会显示立即使用。当点击立即使用就会转跳到该优惠券凑单页面进行使用该优惠券。

在我们的好券集市还有优惠券搜索功能，可以对用户所需求的优惠券进行搜索。在搜索功能下面，我们还有按使用优惠券的商品范围的分类进行查询分类展示，还可以按三大类进行条件查询：第一类：默认、即将过期、面值最大；第二类：京券、东券、运费券、自营券、商家券；第三类：抢券、京豆换券。

### 任务集市

进入优惠劵首页 点击任务集市 会通过域名进入nginx，进行反向代理返回给Tomcat 服务器进行运行web.xml

解析ApplicationCountex.xml 然后进入controller，调用后台的api获取数据，然后返回到前台，这里呢我们用了dubbo 技术，来解决前台和后台的耦合性,在运行项目之前 我们会先运行注册中心zookeeper来给前台提供服务接口，在这个页面里面，是一个优惠劵的分类：全部商品,评价返卷,满额返卷,其他,点击其中一个会传到后台一个参数，进行处理返回对应类型的list集合数据返回到前台页面显示出来,剩余时间在后台处理的时候用到来到data类的一个data.time()方法把时间转化为long类型 我们把系统时间和创建发优惠劵的日期做了时间差，返回到前台js，进行数据处理返回到前台页面显示出来,当我们点击任何图片的时候根据这个图片的数据id进行了后台处理业务，跳转到了二级页面 任务详情，在详细页面我们可以看到任务活动详情介绍，以及购买活动商品的日期和等待奖励页面的发放时间，跳转到这个页面在后台我们会判断是否为登陆状态 如果为登陆的话，我们会在后台查询该用户优惠劵的信息，用到spring定时器，进行间隔查询数据库，查看是否有过期的劵进行判断，有的话进行页面提示框进行提醒，没有的话，不做任何操作，该页面下面活动商品也是根据数据的依赖进行查询回显出来的，并且做出分页，提高用户的体验。

由于任务详情页涉及到的图片资源过多，所以我们采用Nginx反向代理+文件服务器来解决图片对web服务器性能的过度负载，将图片资源静态化。

### 专属优惠券

用户想要某件商品的优惠券，但是这件商品没有优惠券，他可以通过这个模块获取到想要的优惠券，首先用户通过输入商品编码和选择许愿时间许愿。后台会获得许愿请求，当执行实现愿望就是发券的行为，就会根据用户的电子邮件和手机号分别进行发电子邮件和短信提示的功能，提示用户的已经得到优惠券，若是不实现愿望，通过Spring定时器判断许愿到期时间过期则不实现愿望。短信接口采用互亿无线触发短信接口，支持发送优惠券已发送的信息，具体实现是改的提供的Demo来实现的。电子邮件才用javax.mail包来发送的，采用自己编写的发送邮件的工具类实现的发送邮件的功能，测试时发送邮件的服务器选用的是网易的“smtp.163.com”服务器，由于发送的邮件有超链接和样式所以采用“text/html;charset=GBK”编码格式。后来测试成功换成了企业已备案的邮件服务器。

### 券集市

进入优惠券页面，点击券集市进入该页面，展示出用户要用优惠券兑换京豆的所有优惠券。

**业务流程：**

优惠券如何换京豆？

1.在用户账户中被标记为“可换豆”的优惠券是可以用来兑换成京豆的；

2.点击可被兑换的优惠券，选择“兑换为京豆”,优惠券将会在“领券中心-券集市”上架；

3.用户可根据“券集市”中优惠券显示的“消耗京豆”数量，使用等额京豆兑换“券集市”中的优惠券；

4.若您在“券集市”上架的优惠券被其它用户成功兑换，则您可获得系统提示的京豆数量。您参加此次优惠券兑换京豆活动所使用的京豆数量为该优惠券在“券集市”上架时显示“消耗京豆”数量（即其他用户兑换该优惠券所需京豆数量）的25%；

例如：您上架在券集市中的优惠券显示需要消耗100京豆兑换，一但被其他用户成功兑换，您将获得75京豆，25%为您参与此次活动所使用的京豆数量

5.优惠券及京豆到达相应账号的时间以系统时间为准。

这个模块中最主要的问题就是：高并发。

多人来兑换最后一张优惠券时的操作。我们用的是消息队列（rabbitMQ），当只有最后一张优惠券 这个消息被用户消费那一瞬间将不会再有消息去消费，则拒绝接受后来的所有请求，提示券已被领取完。

为了减少对数据库的压力我把所有的可换豆的券存放到redis中，所有的操作都在redis中进行，再同步到数据库中。点击兑换时先检测用户是否登录，如果登录就跳到实名验证页面进行实名核对，继续操作，如果没有登录就弹出登录框，支持第三方登录（qq、微信等），再进行实名认证继续操作。

# 源码跟踪

## Map

### hashMap（1192行）

#### HashMap概述

HashMap基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

值得注意的是HashMap不是线程安全的，如果想要线程安全的HashMap，可以通过Collections类的静态方法synchronizedMap获得线程安全的HashMap。

Map map = Collections.synchronizedMap(new HashMap());

#### HashMap的数据结构

HashMap的底层主要是基于数组和链表来实现的

HashMap中主要是通过key的hashCode来计算hash值的外加取余最终获取索引，而这个索引可以认定是一种地址，既而把相应的value存储在地址指向内容中。它通过把关键码值（key-value）映射到表中一个位置来访问记录，以加快查找的速度。

为什么又需要对得到的散列码(即hash值)求余呢？---上面的 indexFor(int h, int length)完成的功能

在底层是用数组来存储<key, value>的，而我们得到的散列码可能很大（事实上散列码的范围非常广）

而内存是有限的，不能分配为数组分配一块很大很大的空间，因此，存储<key, value>的数组空间相对较小。

从而需要把 所有的散列码都 “约束” 到这个有效的数组空间中。----这也是导致冲突的根源

但是只要key的 hashCode相同，计算出来的hash值就一样。如果存储的对象对多 了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。学过数据结构的同学都知道，解决hash冲突的方法有很 多，HashMap底层是通过链表（链地址法）来解决hash冲突的。

**开放地址法：**

当冲突发生时，通过查找数组的一个空位，并将数据填入。

**1.线性探测**

处理方式：如果index索引与当前的index有冲突，即把当前的索引index+1。如果在index+1已经存在占位现象（index+1的内容不为NULL）试图接着index+2执行。。。直到找到索引为NULL内容为NULL的为止。这种处理方式也叫：线性地址探测法(offset-of-1)

**2平方探测**：以平方大小来递增下一次待探测的位置

链地址法

链地址法的基本思想是：将所有哈希地址为i 的元素构成一个称为同义词链的链表，并将链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。

在HashMap的实现中，采用的链地址法来解决冲突，它有一个桶的概念:对于Entry数组而言，数组的每个元素处存储的是链表，而不是直接的Value。在链表中的每个元素才是真正的<Key, Value>。而一个链表，就是一个桶！因此HashMap最多可以有Entry.length 个桶。

**简单总结：**就是在冲突的位置上建立一个链表，然后将冲突的元素插入到链表尾端

**负载因子：**尺寸／容量．空表的负载因子是0,而半满的负载因子是0.5.负载轻的表,产生冲突的可能性小,对于插入和查找都是最理想的.

HashMap和HashSet都允许指定负载因子,当达到负载因子的水平时,容器将自动增加其容量(桶位数),实现方式是容量大致加倍,并重新将现有的对象分布到新的桶位中,称之为再散列.

HashMap使用的负载因子是0.75(只有当表达到四分之三时,才进行散列),这个因子在时间和空间代价中达到了平衡,更高的负载因子可以降低表所需的空间,但是会增加查找的代价.

**其他方法：**

1 、再散列法：建立多个hash函数，若是当发生hash冲突的时候，使用下一个hash函数，直到找到可以存放元素的位置。

2、建立公共溢出区：将哈希表分为基本表和溢出表，将与基本表发生冲突的元素放入溢出表中。

3. 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。

### HashTable（345行）

Hashtable出来的比HashMap早，HashMap 1.2才有，而Hashtable在1.0就已经出现了。HashMap和Hashtable实现原理基本一样，都是通过哈希表实现。而且两者处理冲突的方式也一样，都是通过链表法

若干方法都加了synchronized关键字，也就意味着这个Hashtable是个线程安全的类，这也是它和HashMap最大的不同点.

1.Hashtable每次扩容，容量都为原来的2倍加2，而HashMap为原来的2倍。

2.Hasbtable并不允许值和键为空（null），若为空，会抛空指针（HashMap可以）

3.Hashtable的容量为任意正数（最小为1），而HashMap的容量始终为2的n次方。Hashtable默认容量为11，HashMap默认容量为16；

### TreeMap（2442行）

TreeMap基于红黑树实现

## List（600行）

**话术：**List源码比较简单，里面就是定义了一些方法

读取(get)，插入(add)，删除(remove)，修改(set)

也可批量增加(addAll)，删除(removeAll,retainAll)

获取(subList)。

还有一些判定操作：

包含(contains[All]),相等(equals),索引(indexOf,lastIndexOf),大小(size)。

还有获取元素类型数组的操作：toArray()

### ArrayList（1172行）

ArrayList是基于数组实现的，是一个动态数组（长度可变的数据结构，允许null当元素，默认长度是10，一次增长0.5倍，其容量为原来的1.5倍。）

ArrayList提供了三种方式的构造器：

1、可以构造一个默认初始容量为10的空列表

2、构造一个指定初始容量。

3、构造一个包含指定collection的元素的列表

// **ArrayList**带容量大小的构造函数。

public **ArrayList**(int initialCapacity) {

super();

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);

// 新建一个指定容量的数组</span>

this.elementData = new Object[initialCapacity];

}

// **ArrayList**无参构造函数。默认容量是10。

public **ArrayList**() {

super();

this.elementData = EMPTY\_ELEMENTDATA;

}

//构造方法传入一个Collection， 则将Collection里面的值copy到**arrayList**

public **ArrayList**(Collection<? extends E> c) {

elementData = c.toArray();

size = elementData.length;

// c.toArray might (incorrectly) not return Object[] (see 6260652)

if (elementData.getClass() != Object[].class)

elementData = Arrays.copyOf(elementData, size, Object[].class);

}

#### 调整数组容量：

**ensureCapacity(int minCapacity) ：** 从上面介绍的向ArrayList中 存储元素的代码中，我们看到， 每当向数组中添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数 据的需求。数组扩容通过一个公开的方法ensureCapacity(int minCapacity)来实现。在实际添加大量元素前，我也可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。另外扩容一次其容量为原来的1.5倍

public void ensureCapacity(int minCapacity) {

if (minCapacity > 0)

ensureCapacityInternal(minCapacity);

}

**问题：**

数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。所以在初始化ArrayList的时候尽量预算下大致的容量需求，降低频繁调整容量的开销。

**void trimToSize() ：** 将底层数组的容量调整为当前列表保存的实际元素的大小的功能。

由于elementData的长度会被拓展，size标记的是其中包含的元素的个数。所以会出现size很小但 elementData.length很大的情况，将出现空间的浪费。trimToSize将返回一个新的数组给elementData，元素内容保持不 变，length和size相同，节省空间。

public void trimToSize() {

modCount++;

int oldCapacity = elementData.length;

if (size < oldCapacity) {

elementData = Arrays.copyOf(elementData, size);

}

}

**补充方法：**

**System.arraycopy方法**：如果是数组比较大，那么使用System.arraycopy会比较有优势，因为其使用的是内存复制，省去了大量的数组寻址访问等时间

1. 源码：public static native void arraycopy(Object src, int srcPos,

2. Object dest, int destPos,

3. int length);

复制指定源数组src到目标数组dest。复制从src的srcPos索引开始，复制的个数是length，复制到dest的索引从destPos开始。

**Arrays.copyOf()**

1. //基本数据类型（其他类似byte，short···）

2. public static int[] copyOf(int[] original, int newLength) {

3. int[] copy = new int[newLength];

4. System.arraycopy(original, 0, copy, 0,

5. Math.min(original.length, newLength));

6. return copy;

7. }

观察其源代码发现copyOf()，在其内部创建了一个新的数组，然后调用arrayCopy()向其复制内容，返回出去。

**总结：**

1.copyOf()的实现是用的是arrayCopy();

2.arrayCopy()需要目标数组，对两个数组的内容进行可能不完全的合并操作。

3.copyOf()在内部新建一个数组，调用arrayCopy()将original内容复制到copy中去，并且长度为newLength。返回copy;

4.arraycopy 方法会因为新数组大小比旧数组大小 小而报IndexOutOfBoundsException;

copyOf 则不会因此报错，因为copyOf 的返回值是在内部new 好的copy 数组，而该copy 数组new 的大小就等于newLength ，

故即使在客户端指定好了新数组newArray 的大小，接收到返回值后也是指向底层new 出来的数组copy 。换句话说( 也可以因此推出其他的区别) ，在客户端代码中即使不给新数组new 对象，如：String[] newStr = null;那么对于arraycopy 是会报NullPointerException 的错误的，而对于java.util.Arrays 中的copyOf 方法则由于jdk 底层已经new 出了对象而不会报该错误！不过需要特别注意的是：copyOf 方法最后也是调用System.arraycopy 的方法，不过由于前面的准备，异常情况就不会出现了。

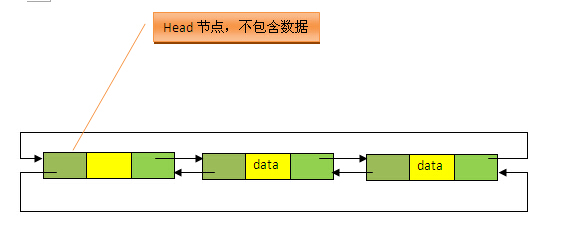
### LinkedList（1138行）

LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。

链表主要包含单向链表，单向循环链表，双向链表，双向循环链表。具体的图我就不在这边画出了，不清楚的可以自行百度。LinkedList是属于双向链表，下图是包含头结点和尾节点的双向链表。

#### LinkedList数据结构原理

LinkedList底层的数据结构是基于双向链表的，且头结点中不存放数据,如下：



既然是双向链表，那么必定存在一种数据结构——我们可以称之为节点，节点实例保存业务数据，前一个节点的位置信息和后一个节点位置信息，如下图所示：

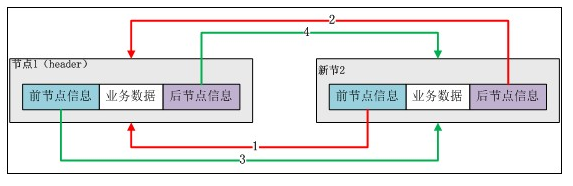


**元素添加步骤：**

第一步：调用无参构造方法，创建LinkedList实例。

第二步：初始化一个预添加的Entry实例（newEntry）。

第三步：调整新加入节点和头结点（header）的前后指针。



#### 构造方法

LinkedList提供了两个构造方法。

1 public LinkedList() {

2 header.next = header.previous = header;

3 }

4 public LinkedList(Collection<? extends E> c) {

5 this();

6 addAll(c);

7 }

**删除数据remove()**

由于删除了某一节点因此调整相应节点的前后指针信息，如下：

e.previous.next = e.next;//预删除节点的前一节点的后指针指向预删除节点的后一个节点。

e.next.previous = e.previous;//预删除节点的后一节点的前指针指向预删除节点的前一个节点。

在LinkedList中查找元素：get()

为了提高效率，需要根据获取的位置判断是从头还是从尾开始遍历。

注意细节：位运算与直接做除法的区别。先将index与长度size的一半比较，如果index<size/2，就只从位置0往后遍历到位置 index处，而如果index>size/2，就只从位置size往前遍历到位置index处。这样可以减少一部分不必要的遍历

在容量不够的时候可以扩容，然而LinkedList容量的说法，每次向其中加入元素时候，容量自动加1。

### Vector（1212行）

Vector:ArrayList的前身，数组结构，线程安全，速度慢

Vector构造函数。默认容量是10。一次增长原来的一倍。

public Vector() {

this(10);

}

其他分析，请参考ArrayList.

# 大数据方向

## Hadoop电梯演讲

Hadoop是一个提供分布式存储和计算的软件框架，它具有无共享、高可用、弹性可扩展的特点，非常适合处理海量数据。

## 云计算

云计算是一种可以通过网络方便地接入共享资源池，按需获取计算资源（这些资源包括网络、服务、存储、应用、服务等）的服务模型。共享资源池中的资源可以通过较少的管理代价和简单业务交互过程而快速部署和发布。

之所以称之为“云”，是因为云计算在某些地方和现实中的云非常符合，云的规模可以动态伸缩，它的边界是模糊的，云在空中飘忽不定，无法也无需确定它的具体位置，但它确实存在于某处。

### 云计算特点

1.按需提供服务：以服务的形式为用户提供应用程序、数据存储、基础设施等资源，并可以根据用户需求自动分配资源，而不需要系统管理员干预。例如亚马逊弹性计算云（Amazon Elastic Compute Cloud， AmazonEC2），用户可以通过填写Web表单将自己所需要的配置，比如CPU核数、内存大小提交给亚马逊，从而动态地获得计算能力。

2.宽带网络访问：用户可以利用各种终端设备（如PC机、笔记本电脑、智能手机等）随时随地通过互联网访问云计算服务。

3.资源池化：资源以共享资源池的方式统一管理。利用虚拟化技术，将资源分享给不同用户，资源的放置、管理与分配策略对用户透明。

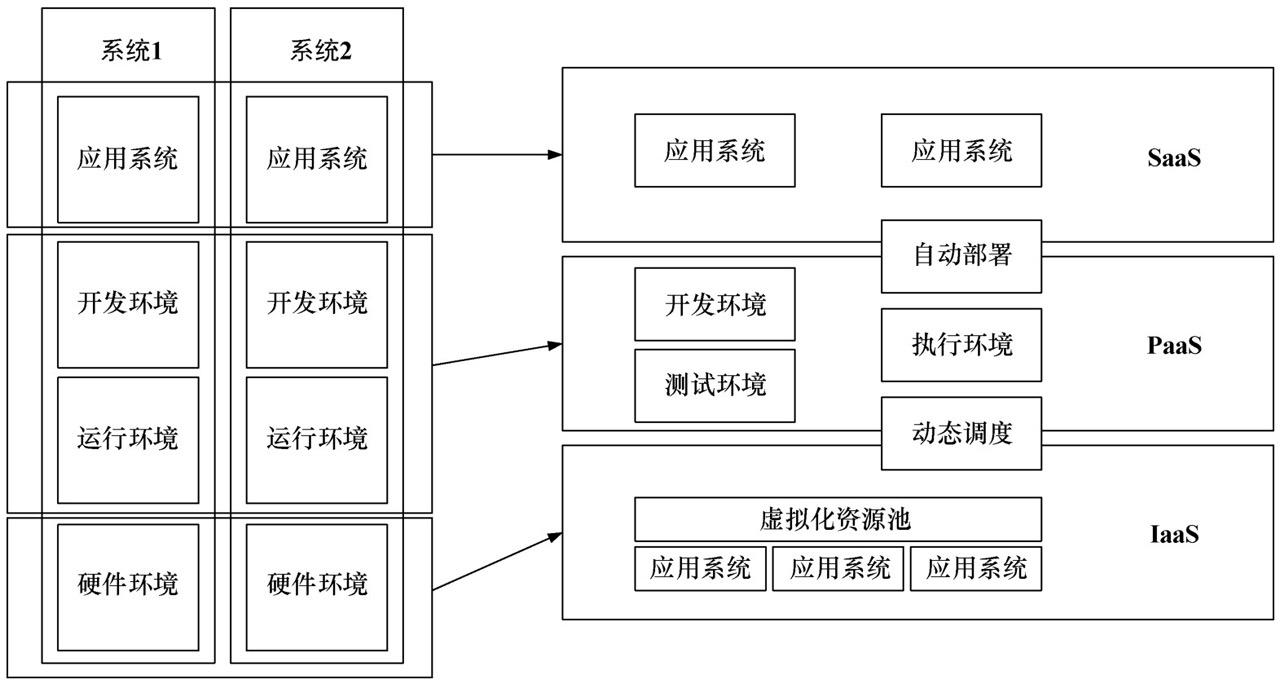
4.高可伸缩性：服务的规模可快速伸缩，以自动适应业务负载的动态变化，用户使用的资源同业务的需求相一致，避免了因为服务器性能过载或冗余而导致的服务质量下降或资源浪费。

5.可量化的服务：云计算中心都可以通过监控软件监控用户的使用情况，并根据资源的使用情况对服务计费。

### 云计算的类型

云计算按照服务类型大致可以分为基础设施即服务（Infrastructure as a Service, IaaS）、平台即服务（Platform as a Service， PaaS）、软件即服务（Software as a Service， SaaS）三类。





小黑留有最终解释权！