# Java语言

## 面向对象

1. 四大特征：

1.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。

2.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。因此可以说，继承是为了重用父类代码，同时为实现多态性作准备。

Final无法继承，直接报错

Private可以有相同的函数，但多态向上转型无法访问

3.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。封装隐藏了类的内部实现机制，从而可以在不影响使用者的前提下改变类的内部结构，同时保护了数据。

封装也称为信息隐藏，是指利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体，数据被保护在抽象数据类型的内部，尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。系统的其他部分只有通过包裹在数据外面的被授权的操作来与这个抽象数据类型交流与交互。也就是说，用户无需知道对象内部方法的实现细节，但可以根据对象提供的外部接口(对象名和参数)访问该对象。

4. 多态性：

多态是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

多态性是指允许不同类的对象对同一消息作出响应。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。方法重载实现的是编译时的多态性（也称为前绑定）。方法重写实现的是运行时的多态性（也称为后绑定）。

http://blog.csdn.net/seu\_calvin/article/details/52191321

Java 里对象方法的调用是依靠类信息里的方法表实现的，虽然对象方法引用调用和接口方法引用调用的实现有所不同但大致思想是一样的。总体而言，当调用对象某个方法时，JVM 要查找对象类的方法表以确定该方法的直接引用地址，有了地址后才真正调用该方法。

我们知道java程序运行时，类的相关信息放在方法区，在这些信息中有个叫方法表的区域，该表包含有该类型所定义的所有方法的信息和指向这些方法实际代码的指针，注意这些具体的方法代码可能是被覆写的方法，也可能是继承自基类的方法。

注:优势：较高的开发效率，高可维护性，鲁棒性。

2. 和面向过程的区别：

面向过程”是一种是事件为中心的编程思想。就是分析出解决问题所需的步骤，然后用函数把这些步骤实现，并按顺序调用。面向对象是以“对象”为中心的编程思想。面向过程就像是一个细心的管家，事无具细的都要考虑到。而面向对象就像是个家用电器，你只需要知道他的功能，不需要知道它的工作原理。

简单的举个例子：汽车发动、汽车到站，这对于“面向过程”来说，是两个事件，汽车启动是一个事件，汽车到站是另一个事件，面向过程编程的过程中，我们关心的是事件，而不是汽车本身。针对上述两个事件，形成两个函数，之 后依次调用。然而这对于面向对象来说，我们关心的是汽车这类对象，两个事件只是这类对象所具有的行为。而且对于这两个行为的顺序没有强制要求。

## Java特点

1. 纯面向对象，包括8种基本类型都是对象。
2. 平台无关性，一次编译，到处运行。
3. 强大的类库和许多的开源框架支持。
4. 对Web应用开发的支持。（JSP，Servlet）
5. 与C/C++异同：

1.Java为解释性语言，C/C++为编译性语言。

2.Java位纯面向对象语言，C++兼具面向对象和过程。

3.Java平台无关性，C/C++与平台紧密相关（比如int字节数）。

4. Java自动垃圾回收机制，C++必须显示的申请和回收内存空间。

5.Java不支持指针，引用，运算符重载，预处理，头文件，默认函数参数，多重继承等。

## 基础语法

1. JDK1.5新特性：a.泛型；b.foreach循环；c.自动拆箱装箱；d.枚举；e.静态导入（Static import）；e.Executor线程池；f.变长参数方法；g.注解。

JDK1.6新特性：a. 简化Web Services；b. Scripting，整合脚本语言；c. 绑定开源嵌入式数据库 Derby；d. 更丰富的Desktop APIs；e. 监视和管理。

JDK1.7新特性：a. switch中可以使用字符串 ；b.新增Objects，Random等类；c.泛型自动类型推断（“菱形”语法）；d.多类型异常捕获和自动关闭资源的try-catch语句；e.新增swing功能；e.NIO.2新增Path，Paths，Files等；f.AIO通信。

JDK1.8新特性：a. 接口的默认方法；b. Lambda 表达式；c. 函数式接口；d. 全新的时间日期API；e.多重注解；e.ForkJoinPool线程池；f. 方法与构造函数引用；g. Stream 接口和Predicate谓词接口等。

1. 回调：就是A类中调用B类中的某个方法C，然后B类中反过来调用A类中的方法D，D这个方法就叫回调方法；线程run()也是一个回调方法，当执行Thread的start（）方法就会回调这个run()方法。
2. Object公有方法：getclass、clone、hashcode、equals、tostring、wait、notify、notifyAll。

Objects工具类就是为了解决当对象调用Object基础方法时发生的空指针异常问题，里面包含了和Object一样的方法，只不过都是静态的工具方法，并且参数是一个Object对象。

1. Main方法可以加final，synchronized，static和public可以互换顺序，可以抛出异常。
2. ++i(i += 1；return i)和i++( j = i； i += 1； return j)，多用一个临时变量，效率低。

注：iload\_1 将索引号为1的常量（j）压至栈顶，iinc 1， 1 将索引号位1的常量（j）自增1， istore\_1 将栈顶数据保存到索引号为1的位置。iinc 指令，是对本地变量的操作，并没有对栈顶的数据进行自增。

1. 程序初始化顺序： 父类静态变量和静态代码块——子类静态变量和静态代码块——子类main方法——父类非静态变量和非静态代码块（父类的实例成员和实例初始化块）——父类构造函数-子类非静态变量和非静态代码块——子类构造函数。（静态部分在类加载阶段完成初始化只一次，且代码块和变量的赋值顺序按照程序中出现的先后顺序）。
2. Switch语句后面控制表达式的数据类型只能是：int， short，char， byte， String，枚举等6类。
3. 包装类和原始类型：java为8种原始类型都提供了包装类型，区别：大小，访问速度，位置，缺省时的初始值。建议只有在原始类型不能用用才使用包装类型，比如集合元素。
4. String、StringBuffer、StringBuilder:

都是 final 类， 都不允许被继承；String 不可变的， 使用final字符数组保存字符串，StringBuffer、StringBuilder 可变的； StringBuffer 是线程安全的， StringBuilder 不是线程安全的，但它们两个中的所有方法都是相同的，StringBuffer在StringBuilder的方法之上添加了synchronized修饰，保证线程安全。StringBuilder比StringBuffer拥有更好的性能。StringBuilder与StringBuffer有公共父类AbstractStringBuilder(抽象类)。

如果一个String类型的字符串，在编译时就可以确定是一个字符串常量，则编译完成之后，字符串会自动拼接成一个常量。此时String的速度比StringBuffer和StringBuilder的性能好的多。在用+号链接字符串的时候会创建新的字符串。通过使用 "+" 符号来串联字符串的时候， 实际上底层会转成通过 StringBuilder 实例的 append() 方法来实现。

如果程序不是多线程的，那么使用StringBuilder效率高于StringBuffer。

1. Static final

Final:

final类不能被继承，没有子类，final类中的方法默认是final的。

final方法不能被子类的方法覆盖，但可以被继承。

final成员变量表示常量，只能被赋值一次，赋值后值不再改变。

final不能用于修饰构造方法。

注意：父类的private成员方法是不能被子类方法覆盖的，因此private类型的方法默认是final类型的。

Static:

被static修饰的成员变量和成员方法独立于该类的任何对象。JVM只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问。只要这个类被加载，Java虚拟机就能根据类名在方法区内找到他们

静态方法可以直接通过类名调用，静态方法中不能用this和super关键字，不能直接访问所属类的实例变量和实例方法，只能访问所属类的静态成员变量和成员方法。

JVM加载类时会执行静态的代码块，每个代码块只会被执行一次

1. 反射机制：运行时自我检查和查看类的信息，创建类的实例，同时允许对其内部成员进行操作的机制。有助于观察和操作应用程序的运行时行为。

<http://blog.csdn.net/fu755383249/article/details/40261223>

当一个类或接口被装入JVM时便会产生一个与之关联的java.lang.Class对象，可以通过这个Class对象对被装入类的详细信息进行访问。

Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。

换言之，Java可以加载一个运行时才得知名称的class，获得其完整结构。

首先通过字符串找到该类名称，通过名称找到对应的字节码文件，将字节码文件加载进内存产生对应的Class对象。通过Class对象的成员方法创建对应的类的对象。

编译过的生成的Class文件，在被类加载器加载后，都会对应有一个java.lang.Class<T> 这个类的实例。每个类的自有的方法属性(类结构)自然被包含在了这个对应的实例上

通过堆中的class对象访问到方法区中类型信息，通过class对象可以访问到该类的所有信息。

Java的反射机制是在程序运行时，能够完全知道任何一个类，及其它的属性和方法，并且能够任意调用一个对象的属性和方法。这种运行时的动态获取就是Java的反射机制。其实这也是Java是动态语言的一个象征。

Java的反射机制允许我们动态的调用某个对象的方法/构造函数、获取某个对象的属性等，而无需在编码时确定调用的对象。这种机制在我们常用的框架中也非常常见。

Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。

用一句话来概括反射就是加载一个运行时才知道的类以及它的完整内部结构。

第一种：反射的目的就是为了扩展未知的应用。比如你写了一个程序，这个程序定义了一些接口，只要实现了这些接口的dll都可以作为插件来插入到这个程序中。那么怎么实现呢？就可以通过反射来实现。就是把dll加载进内存，然后通过反射的方式来调用dll中的方法。

第二种：在编码阶段不知道那个类名,要在运行期从配置文件读取类名, 这时候就没有办法硬编码new ClassName(),而必须用到反射才能创建这个对象。

1. 抽象类和接口异同

都不能被实例化

抽象类可以有方法实现

接口中的成员变量默认为public static final修饰，方法默认为public abstract修饰。

接口实现用implement，可以有多个，类的继承用extend，只能继承一个。

接口体现的是标准，规范，通信设计，抽象类体现的是模板，公共类。

注意：没有任何方法的接口为标记接口（克隆，序列化接口）。

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

2）抽象类中的成员变量可以是各种类型的，抽象方法必须为public或者protected，缺省情况下默认public；而接口中的成员变量只能是public static final类型的；

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

4）一个类只能继承一个抽象类，而一个类却可以实现多个接口。

5) 如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。如果子类没有实现父类的抽象方法，则必须将子类也定义为为abstract类。实现接口的非抽象类必须实现接口的所有方法，而抽象类不需要

1. 内部类有四种：静态内部类，成员内部类，局部内部类，匿名内部类（无构造函数，静态成员，方法，类）。

注：优先选择静态内部类，局部和匿名内部类使用外部变量时，需用final关键字修饰该外部变量

@@@ 要知道局部变量的生命周期与局部内部类的对象的生命周期的不一致。局部变量当所处的函数执行结束后就已经死亡了，不存在了，但是局部内部类对象还可能一直存在（只要有人还引用该对象），这是就会出现了一个悲剧的结果，局部内部类对象访问一个已不存在的局部变量。

@@@原因是编译程序实现上的困难：内部类对象的生命周期会超过局部变量的生命周期。局部变量的生命周期：当该方法被调用时，该方法中的局部变量在栈中被创建，当方法调用结束时，退栈，这些局部变量全部死亡。而内部类对象生命周期与其它类一样：自创建一个匿名内部类对象，系统为该对象分配内存，直到没有引用变量指向分配给该对象的内存，它才会死亡(被JVM垃圾回收)。所以完全可能出现的一种情况是：成员方法已调用结束，局部变量已死亡，但匿名内部类的对象仍然活着。

当我们创建匿名内部类的那个方法调用运行完毕之后，因为局部变量的生命周期和方法的生命周期是一样的，当方法弹栈，这个局部变量就会消亡了，但内部类对象可能还存在。 此时就会出现一种情况，就是我们调用这个内部类对象去访问一个不存在的局部变量，就可能会出现空指针异常

1. 对象的4种引用：

强引用

软引用

如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。

弱引用

只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

虚引用

虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。 虚引用主要用来跟踪对象被垃圾回收的活动。这个对象被收集器回收时得到一个系统通知

附：WeakHashMap:key只保留对实际对象的弱引用，如果key没有被其他对象强引用，则可能被垃圾回收。

1. Equals、hashcode、tostring方法：

Object类提供的equals方法只是比较对象的地址，因此，一般要求重写equals方法，相等条件由业务要求决定。覆盖equals通用约定：自反性、对称性、传递性、一致性。覆盖equal时必须总是覆盖hashcode（相等对象必须具有相等的散列码）。

Object类提供的Hashcode方法是一个本地方法，针对不同的对象肯定返回不同的整数值（通过将该对象的内部地址转换成一个整数来实现的），通常要求覆盖，使用31的原因是奇素数，使用移位来代替乘法具有很好地性能，信息不会丢失。

注：hashMap的hashcode就是每个entry的hashcode的求和，每个entry的hashcode又是key和value的hashcode的异或；字符串的hashcode就是综合考虑字符串里面每一位的字符asci码，乘积再求和算出来的。

Object类提供的tostring方法返回一个字符串，用于描述对象本身，该字符串由类名（对象是该类的一个实例）、at 标记符“@”和此对象哈希码的无符号十六进制表示组成。建议覆盖。

1. 取得父类类名：getclass.getsupercalss.getname(getclass由final和native修饰，最终调用都是返回运行时类)。
2. clone方法:返回对象的一个浅复制，返回的是新对象而不是原来对象。（建议不要覆盖此方法，也不要调用，除非拷贝数组）

需要调用super.clone方法，产生正确大小的空间，并且逐位复制

1. Comparable和Comparator

两个都是接口，Comparable是排序接口；内部有一个compareTo方法，一个类实现了Comparable接口则表明这个类的对象之间是可以相互比较的，这个类对象组成的集合就可以直接使用sort方法排序。

而Comparator是比较器；内部有一个compare方法，我们若需要控制某个类的次序，可以建立一个“该类的比较器”来进行排序。Comparator可以看成一种算法的实现，将算法和数据分离。  
Comparable相当于“内部比较器”，而Comparator相当于“外部比较器”。

1. 快速失败(fail-fast)和安全失败(fail-safe)的区别是什么？

Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util.concurrent包下面的所有的类都是安全失败的，java.util包下面的所有的集合类都是快速失败的。快速失败的迭代器会抛出ConcurrentModificationException异常，而安全失败的迭代器永远不会抛出这样的异常。

Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。

Iterator.remove() 方法会在删除当前迭代对象的同时维护索引的一致性

Iterator本身remove不会造成快速失败，但是一个线程Iterator遍历，另一个线程list.remove，则有可能快速失败

快速失败”也就是fail-fast，它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast机制。

在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。

快速失败：对于非并发集合来说，在其进行迭代时，例如iterator迭代时，iterator是另起一个线程，若有其他线程（如Collection）（另一个iterator修改也会报错，有一定几率，亲测）进行结构修改（修改了增减了集合中的内容），这个迭代会马上感知到，并且立即抛出 ConcurrentModificationException 异常，而不是迭代完成后才告诉你出错了，引起快速失败。若用iterator进行修改则不会出现这个问题（单线程），如iterator.remove();也就是说涉及到了多个线程间的同步问题

Iterator都是对底层集合的拷贝

1. throw和throws有什么区别？

throw关键字用来在程序中明确的抛出异常，相反，throws语句用来表明方法不能处理的异常。

注：error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。 exception 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况。

只有对外部不能确定和预知的运行错误才使用异常，不要使用 catch all，不要忽略捕获的异常。Finally不执行的情况：程序在try之前遇到异常，在try中强制退出（exit）。

1. finally执行顺序

1. finally语句在return语句执行之后return返回之前执行的。

2. finally块中的return语句会覆盖try块中的return返回。

3. 如果finally语句中没有return语句覆盖返回值，那么原来的返回值可能因为finally里的修改而改变（引用类型）也可能不变（基本类型）。

4.除非在try块或者catch块中调用了退出虚拟机的方法

1. 标识符规则：必须以字母（中文，日文都行）、下划线、$、开头，后跟数字、字母、下划线、$。
2. 浮点型：（浮点数除0得到正或者负无穷大）所有的正无穷大相等，所有的负无穷大相等，（0.0除0.0或者对负数开方）NaN不与任何数相等。
3. 方法重载：同一个类中，方法名相同，参数类型、顺序、个数至少一个不同，返回值和修饰符无法影响是否重载。

方法重写：子类与父类之间，方法名和形参列表相同，返回值类型和抛出异常信息更小，访问权限更大。即父类是public，子类不能使private，父类throws IOException，子类不能throws Exception

只有实例方法可以被重写！重写后的方法必须仍为实例方法！(实例方法------重写------>实例方法)

成员变量和静态方法都不能被重写，只能被隐藏。

注：成员变量无法实现多态，只取决于所定义变量的类型。

1. strictfp：修饰类、接口和方法，用于精确浮点数运算
2. instanceof：判断一个引用类型的变量所指向的对象是否为一个类或者接口的实例。System.out.println(rl instanceof ReentrantLock)；//输出是”true”
3. null不是一个合法的Object实例，仅仅表明该引用目前没有指向任何对象。
4. System类：代表当前java运行平台，提供了标准输入、输出、错误输出和一些静态方法用于访问环境变量、系统属性等。

Runtime类：代表Java运行时环境，用于访问JVM相关信息，处理器数量，内存等。

1. 正则表达式：字符串处理工具，可以对字符串进行查找、提取、分割、替换操作。
2. 泛型：让代码更加简洁、程序更加健壮、只在编译阶段有效。Java7新增“菱形”语法。

使用类型参数指示元素的类型

Java泛型是通过擦除实现的，类定义中的类型参数如T会被替换为Object，在程序运行过程中，不知道泛型的实际类型参数

1.安全性：开发环境发现问题，否则，编译时是没有任何问题的。但，运行时，程序抛出了类型转换异常ClassCastException

2.可读性：避免频繁的类型转换，提高可读性

需要特别注意的是，List<?>与List<Object>并不相同，无论A是什么类型，List<A>是List<?>的子类，但是，List<A>不是List<Object>的子类。

例如：

List<Number> lb = new ArrayList<>();

List<Integer> la = lb; // 会报编译错误，尽管Integer是Number的子类，但是List<Integer>不是List<Number>的子类

List<Integer>与List<Number>的关系如下：

List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList;

// 不会报错， List<? extends Integer> 是 List<? extends Number>的子类

让我们测试一下我们对泛型的理解。下面的代码片断合法么？

List<String> ls = new ArrayList<String>(); //1

List<Object> lo = ls; //2

第1行当然合法，但是这个问题的狡猾之处在于第2行。

这产生一个问题：

一个String的List是一个Object的List么？大多数人的直觉是回答：“当然！”。

好，在看下面的几行：

lo.add(new Object()); // 3

String s = ls.get(0); // 4: 试图把Object赋值给String

这里，我们使用lo指向ls。我们通过lo来访问ls,一个String的list。我们可以插入任意对象进去。结果是ls中保存的不再是String。当我们试图从中取出元素的时候，会得到意外的结果。

java编译器当然会阻止这种情况的发生。第2行会导致一个编译错误。

总之，如果Foo是Bar的一个子类型(子类或者子接口)，而G是某种泛型声明，那么G<Foo>是G<Bar>的子类型并不成立!!

1. AWT：抽象窗口工具集，JVM调用操作系统本地的图形界面实现，不可移植。

Swing:独立于本地平台，100%Java实现，采用MVC设计模式。

注：一般swing组件是从awt组件扩展的，swing是轻量组件 而awt是重量组件， AWT是比较依赖操作系统组件，功能少，Swing不依赖，组件多，功能多 窗口风格更多样化 Swing是由纯JAVA CODE所写的，解决了JAVA因窗口类而无法跨平台的问题，使窗口功能也具有跨平台与延展性的特性，而且SWING不需占有太多系统资源，并且一些组件也采用一些更开放式的开发方式，支持更多的扩展。

1. 流：对不同的输入输出的抽象（键盘、文件、网络连接等）。
2. Socket: 在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。socket则是对TCP/IP协议的封装和应用(程序员层面上)。TCP/IP提供可供程序员做网络开发所用的接口，这就是Socket编程接口。
3. 枚举类：解决类型不安全，无命名空间，打印意义不明确的问题，适用于有限且固定的对象。

注：枚举继承java.lang.enum类，而不是Object类，默认使用final修饰，且枚举值必须在第一行显式给出，构造器只能为private。（为什么枚举类的构造器只能用private修饰：其实很容易想明白，所谓枚举类就是有包含有固定数量实例（并且实例的值也固定）的特殊类，如果其含有public构造器，那么在类的外部就可以通过这个构造器来新建实例，显然这时实例的数量和值就不固定了，这与定义枚举类的初衷相矛盾，为了避免这种形象，就对枚举类的构造器默认使用private修饰。如果为枚举类的构造器显式指定其它访问控制符，则会编译出错。）

1. Lamda表达式: lamda表达式（java8最重要的更新）：使用简洁的代码创建函数式接口实例

背景：Java是一门面向对象编程语言。面向对象编程语言和函数式编程语言中的基本元素都可以动态封装程序行为：面向对象编程语言使用带有方法的对象封装行为，函数式编程语言使用函数封装行为。

作用：匿名类型最大的问题就在于其冗余的语法，lambda表达式是匿名方法，它提供了轻量级的语法，是匿名内部类的简化。

组成:形参列表，->，代码块（不需要new，方法名，返回值）。用@FunctionalInterface标注

1. 创建不可变类：（基本类型包装类，String）。

1.类中所有成员变量都用private修饰。

2.类中没有写或者修改成员变量的方法。

3.确保类中所有方法不会被子类覆盖。Final修饰

4.如果存在一个类成员不是不可变量，那么在get方法中获取该成员时，需要用clone方法。

1. 序列化与反序列化

定义：序列化是指把对象转换为字节序列的过程；而反序列化是指把字节序列恢复为对象的过程。为程序员进行对象的传输和保存提供的机制。

用途：把对象的字节序列永久地保存到硬盘上或者数据库中，在网络上传送对象的字节序列。

序列化步骤：创建一个对象输出流；通过对象输出流的writeObject()方法写对象。  
  
反序列化步骤： 创建一个对象输入流；通过对象输入流的readObject()方法读取对象。

注：方法不会被序列化，使用static和transient修饰的成员不会被序列化，序列化一个对象，尽量指定serialVersionUID(提高程序运行效率，提高不同平台兼容性，增加后续版本的兼容性)。常见序列化协议：XML，SOAP，JSON， Protobuf.

1. 内存泄露

内存泄露是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成的内存空间的浪费。原因在于长生命周期的对象持有短生命周期对象的引用。

主要有：静态集合类，监听器，单例模式，物理连接（数据库、网络）是独立于JVM的，必须显式关闭。

1. Class.forname:将指定字符串名的类或者接口加载到JVM中，创建和返回class对象。(获取class对象的三种方式：Class.forname，类名.class，对象.getclass)
2. assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。在实现中，assertion就是在程序中的一条语句，它对一个boolean表达式进行检查，一个正确程序必须保证这个boolean表达式的值为true；如果该值为false，说明程序已经处于不正确的状态下，assert将给出警告或退出。一般来说，assertion用于保证程序最基本、关键的正确性。assertion检查通常在开发和测试时开启。为了提高性能，在软件发布后，assertion检查通常是关闭的。
3. 异常

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

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

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

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

Exception 这种异常分两大类运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

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

运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。

非运行时异常 （编译异常）：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

1. Cglib

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。

其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

1. 动态代理

最后由Proxy.newProxyInstance生成代理类，继承了Proxy类，而java不支持多继承，因此只能实现被代理类的接口，所以 jdk动态代理局限于接口

通过目标类的接口生成代理类的字节码，最后生成代理类



## JDBC

1. 使用步骤：a.加载JDBC驱动器jar包，b. Class.forname加载JDBC驱动，c.建立数据库连接，取得Connection对象。d.建立Statement对象或者是prepareStatement对象。e.执行SQL，f.访问结果集。g.依此关闭资源。
2. JDBC事务：5种，比mysql多了一种：不支持事务。
3. Statement用于执行不带参数的简单SQL，并返回结果集，每次执行都要编译SQL

prepareStatement用于执行带参数的预编译SQL

CallableStatement提供了调用数据库存储过程的接口。

prepareStatement比Statement的优势：效率更高，代码可读性，可维护性更好，更安全（防止SQL注入式攻击）。

注：一定要保证释放不再使用的连接，数据库连接池：客户端调用数据库提供的服务，需要建立到数据库的tcp链接，然而建立链接是很耗时的，所以，我们可以使用一个池子（容器）来保存这些连接，客户端需要连接的时候，从连接池里取出可用连接即可，用完的时候，把连接放回池子就可以了。数据库连接池：DBCP，C3P0。（实现DataSoure）

## 集合类

1. 概述：Collection和Map 是根接口，Collection接口有三个子接口：Set（无序、不可重复）、List（有序、重复）、Queue（队列）。

注：Collections是处理集合的工具类，包含一系列静态方法，如排序，搜索，线程安全化。

1. Set集合类（4个，相对于Collection没有提供任何额外方法，都是线程不安全的）：

HashSet：元素值可以是null；基于HashMap来实现的，有一个虚拟的Object

LinkedHashSet：内部使用一个链表来维护元素次数，即按插入顺序或者最近访问时间遍历，性能不如HashSet；基于LinkedHashMap实现

TreeSet：继承SortedSet，采用红黑树存储元素，支持自然排序和定制排序（Comparator）

EnumSet:为枚举设计，同一枚举类的值，元素值是有序的，不可以是null，性能最好

注意：所有Set实现只是对对应Map的封装，都委托给Map。

1. List集合(Object数组实现，java8增加了ListIterator接口。支持向前遍历和修改元素)

ArrayList:默认为10线程不安全，每次增加原来1.5倍，效率很高，初始时能指定容量参数效率会更高

Vector:线程安全，每次增加原来2倍，尽量少用(还有一个子类Stack)

1. Queue集合(子接口Deque:双端队列)

PriorityQueue:按元素大小值，元素值不能是null，不是线程安全的，入队和出队的时间复杂度是O(log(n))。

ArrayDeque:实现Deque，数组实现，双端队列，栈

LinkedList: 实现Deque和List，循环双链表，双端队列（插入head之前），栈（插入head之后）。

1.7 1.8不是循环链表，add是addLast

注：ArrayList和LinkedList性能比较：add()，增加元素到尾端，ArrayList性能更好(新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据)；删除任意位置的元素，LinkedList性能更好，随机访问ArrayList更好(随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针)，对LinkedList的遍历要采用迭代器，不要用for循环。

LinkedList迭代器是listiterator，和arraylist中的不一样，并且LinkedList在某一位置添加元素需要遍历到那个元素的位置，不过还是要比arraylist要快。

1. Map集合:

Resize：将原来的entry[]数组赋值为null，将原来的值放到新的entry[]数组上，头插法

Remove：若只有一个，那么直接e.next 为null，若为链表，则直接指向就行

HashMap实现原理（一个好的散列函数通常倾向于“为不相等的对象产生不相同的散列码”）：底层使用数组+双链表+红黑树结构，数组元素是Node对象（内含key，value，next，hash等）

一个长度为16的数组中，每个元素存储的是一个链表的头结 点

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

Table代表entry<K，V> []，bucket代表entry中的元素

在put时，如果table不存在，先创建，分别调用hashcode和内部hash函数（对key的hashCode的进一步优化，加入了高位计算，就使得只有相同的hash值的两个值才会被放到数组中的同一个位置上形成链表。）对key做hash，取得hash值以后和数组长度按位与直接得到数组索引，如果当前位置没有元素，直接存即可，如果有元素，比较其hash值和equals方法是否相等，如果都相等，则说明为同一对象，更新value即可，如果不等则采用链地址法解决冲突，数组中总是最新插入的数据（插入到链表头部，头插法1.8尾插）。当table达到阀值(大于等于TREEIFY\_THRESHOLD = 8)时，bucket就会由链表转换为红黑树的方式进行存储（注意：如果两个对象的equals方法相等而hashcode不等，则hashmap会将其认为是不同的对象）。如果table满了(超过load factor\*current capacity)，就要resize（rehash），扩充为原来的2倍

①.判断键值对数组table[i]是否为空或为null，如果是执行resize()进行扩容；

②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

如果发现目前的table占用程度已经超过了Load Factor（0.75）所希望的比例，那么就会发生resize。在resize的过程，简单的说就是把bucket扩充为2倍。

在get时，根据对象的hash值先找到数组索引位置，然后在此链表上依此比较对象的hash值和equals相等，如果相等，返回对象，否则返回null。

在初始时，可以指定容量参数和负载因子，默认容量为16，负载因子为0.75。同时，内部维护一个threshold变量。超过阈值时，便会扩容，进行rehash，实际填充率不会超过负载因子。

Hash冲突：链地址法，新的Entry依然会被放到对应的索引下标内，替换原有值。同时将新的entry的next指向旧值

LinkedHashMap：在内部维护一个双向循环链表，存放元素顺序，通过为Entry属性增加before和after引用实现 （前驱和后继）。LRU实现（两种算法），访问过的放到最后。

三个函数：访问后、插入后、删除后；基本上都是为了保证双向链表中的节点次序或者双向链表容量所做的一些额外的事情，目的就是保持双向链表中节点的顺序要从eldest到youngest

TreeMap：最坏查找时间为O(logn)，通过传入Comparator，或者实现Comparable接口实现有序。例如，若对person类进行排序，若person类实现comparable接口，内部重写compareTo，直接调用treemap即可；若person类没有实现comparable接口，则treeMap需要实现comparator接口，重写compare（person1，person2）函数

IdentityHashMap:当两个key严格相等时才相等。

HashMap和Hashtable（子类Properties）比较：HashMap允许key和value空，不是线程安全，hash算法不同（Hashtable直接利用对象的hashcode值）；HashMap去掉了Hashtable的contains方法。Hashtable中hash数组默认大小是11，增加的方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。HashTable继承Dictionary类，而HashMap继承AbstractMap类。

volatile修饰的变量，jvm虚拟机只是保证从主内存加载到线程工作内存的值是最新的，而修改i=i+1并无法保证其他线程正在等待，若其他线程在修改后在获取，则ok，若在修改前获取，则出错。可见性只能使i=i+1后其他线程立即看到i的最新值而已

http://ifeve.com/java-concurrent-hashmap-2/

一个线程new这个对象的时候还没有执行完构造函数就被另一个线程得到这个对象引用（还没有new完成就调用，为null）

HashMap和ConcurrentHashMap比较: （1）ConcurrentHashMap对整个桶数组进行了分段，在每一个分段上都用锁进行保护，不允许key和value空。

使用注意：必须实现 equal方法和hashcode方法，初始时指定容量效率会比较高。多线程的情况下使用concurrenthashmap. HashMap中的key不能是可变对象，如果是可变对象的话，对象中的属性改变，则对象HashCode也进行相应的改变，导致下次无法查找到已存在Map中的数据。

TREEIFY\_THRESHOLD

UNTREEIFY\_THRESHOLD

MIN\_TREEIFY\_CAPACITY

值及作用如下：

//一个桶的树化阈值

//当桶中元素个数超过这个值时，需要使用红黑树节点替换链表节点

//这个值必须为 8，要不然频繁转换效率也不高

static final int TREEIFY\_THRESHOLD = 8;

//一个树的链表还原阈值

//当扩容时，桶中元素个数小于这个值，就会把树形的桶元素 还原（切分）为链表结构

//这个值应该比上面那个小，至少为 6，避免频繁转换

static final int UNTREEIFY\_THRESHOLD = 6;

//哈希表的最小树形化容量

//当哈希表中的容量大于这个值时，表中的桶才能进行树形化

//否则桶内元素太多时会扩容，而不是树形化

//为了避免进行扩容、树形化选择的冲突，这个值不能小于 4 \* TREEIFY\_THRESHOLD

static final int MIN\_TREEIFY\_CAPACITY = 64;

## String

String s = new String("Hello world")； 可能创建两个对象也可能创建一个对象。如果静态区中有“Hello world”字符串

常量对象的话，则仅仅在堆中创建一个对象。如果静态区中没有“Hello world”对象，则堆上和静态区中都需要创建对象。

在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配

堆内存用来存放由new创建的对象和数组。在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理

## 注解

源程序中的元素关联任何信息和任何元数据的方法。即可以为程序提供一些元数据，这些元数据可以在编译、运行时被读取、从而提供更多的额外处理信息。

1. JDK自带注解：

@Override:覆盖父类方法

@Deprecated：修饰过时的方法。如果程序员不小心使用了它的元素，那么编译器会发出警告信息。

@Suppvisewarnings:忽略特定警告，如@Suppvisewarnings("Deprecation")

1. 按来源：1.JDK注解 2.第三方注解 3.自定义注解
2. 解析注解：通过反射获取类、方法或变量上的运行时注解信息，从而实现动态控制程序运行的逻辑。

## JAVA与XML

1. 一.XML简介（文档类型定义DTD和Schema，Schema本身是XML的，可以本XML解析器解析）：

1、xml文件以.xml为扩展名

2、存储：树形结构

3、XML的主要用途：不同平台不同软件之间通信的媒介。用于不同平台、不同设备间的数据共享通信，主要缺点是占用更大的存储空间，当数据量过大时存储效率会很低。

注：Json：键值对存储，独立于语言，相对于XML的优势：长度更小，数据小，速度快，JS内建方法支持直接解析。

1. 二.XML四大解析方式比较：

DOM解析：最基础，与平台和语言无关，一次性全部加载到内存中进行解析。形成DOM树，直观，便于理解，便于修改，浪费时间，占用内存，造成内存溢出，效率低，不需要导入jar包；w3c组织提供的一个官方解析方式。

SAX解析：基础，是基于事件的解析，不需要导入jar包；逐步解析，耗费内存小，适用于只处理数据，不易编码，不能同时访问。

JDOM解析：只能在Java中使用，大量使用集合类，不适用接口，开源。

DOM4J解析：只能在Java中使用，JDOM的智能分支，使用接口和抽象基类，性能优异，灵活性好，功能强大，易使用，开源，善于处理大XML文件，Hibernate用于解析XML配置文件.

## JAVA编码问题

1. ASCII码：单字节，总共128个；
2. IOS-8859-1:单字节，总共256个字符；
3. GB2312：双字节编码
4. GBK：扩展GB2312，加了很多的汉字
5. UTF-16：定义了Unicode字符在计算机中的存取方法，编码效率高，双字节编码（Unicode只是一个符号集，它只规定了符号的二进制代码）。
6. UTF-8：变长编码，适合于网络传输
7. 在IO中字节流和字符流之间的转化需要编码，可以使用Charset 指定编码字符集，否则默认本地环境字符集。

## NIO与AIO

1. 概述：对于一个network IO (这里我们以read举例)，它会涉及到两个系统对象，一个是调用这个IO的process (or thread)，另一个就是系统内核(kernel)。当一个read操作发生时，它会经历两个阶段：  
    1 等待数据准备  
    2 将数据从内核拷贝到进程中

注：以银行取钱为例介绍同步和阻塞的概念；以钓鱼为例介绍4种IO模型。

1. IO模型（4种，以Linux为例）

blocking IO （两个阶段都被阻塞）。

non-blocking IO（需要不断询问，如果数据没有准备好，立刻返回，但是在数据复制的阶段是阻塞的）

IO multiplexing（需要两个系统调用：select，recvfrom；select调用会阻塞线程，优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接）

Asynchronous I/O（kernel会等待数据准备完成，然后将数据拷贝到用户内存）

1. 各区别：注：阻塞和非阻塞区别：blocking IO会一直block住对应的进程直到操作完成，而non-blocking IO在kernel还准备数据的情况下会立刻返回。

注：同步和异步区别：同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪；异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。

同步和异步是相对于应用和内核的交互方式而言的，同步 需要主动去询问，而异步的时候内核在IO事件发生的时候通知应用程序，而阻塞和非阻塞仅仅是在调用系统调用的时候函数的实现方式而已。

1. BIO

同步并阻塞，传统的IO流方式， 服务器实现模式为一个连接一个线程， 即客户端有连接请求时服务器端就需要启动一个线程进行处理， 如果这个连接不做任何事情会造成不必要的线程开销，

1. NIO（java4 Reactor模式）

同步非阻塞，服务器实现模式为一个请求一个线程， 即客户端发送的连接请求都会注册到多路复用器上， 多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。NIO弥补了原来的I/O的不足，它提供了高速和面向块的I/O，原来的I/O库与NIO最重要的区别是数据打包和传输方式的不同，原来的I/O以流的方式处理数据，而NIO以通道channel和缓冲区Buffer为基础来实现面向块的IO数据处理。

Channel ：双向的非阻塞通道，两边都可以进行读写，只能和Buffer交互。Buffer：缓冲区，用于保存数据（容量、界限、位置）；

直接内存访问：DirectBuffer，直接分配在物理内存上，不占用堆空间，不需要从内核到进程的数据复制过程，但是创建和销毁的代价很昂贵。

Selector：检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的响应处理。这样一来，只是用一个单线程就可以管理多个通道，也就是管理多个连接（一个SelectionKey表示一个到达的事件）。

NIO框架：mina，netty.

1. AIO（java7 Proactor模式）

异步非阻塞， 服务器实现模式为一个有效请求一个线程， 客户端的I/O请求都是由OS先完成了再通知服务器，应用去启动线程进行处理.需要底层操作系统支持，调用read()方法，直接返回，通过注册动作完成处理器，重写completed方法接收I/O完成的通知。

注：Reactor模式：通过Reactor对所有客户端事件做处理，然后派发到不同的线程中，用于同步IO，即IO多路复用，通知在动作发生之前，而Proactor在完成时通知，异步IO；

注：select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的。select的几大缺点：（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大（3）select支持的文件描述符数量太小了，默认是1024。poll的实现和select非常相似，只是描述fd集合的方式不同。epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll\_create，epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。总结：（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll\_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll\_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll\_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。select也有相对于epoll的优点：在大部分连接都是活跃的情况下，epoll的维护队列的开销要大一些。

注：系统调用是属于操作系统内核的一部分的，必须以某种方式提供给进程让它们去调用。大多数系统交互式操作需求在内核态执行。如设备IO操作或者进程间通信。用户程序只在用户态下运行，有时需要访问系统核心功能，这时通过系统调用接口使用系统调用。系统调用很耗时，要尽量少用。第一，系统调用通过中断实现，需要完成栈切换。第二，使用寄存器传参，这需要额外的保存和恢复的过程。操作系统一般是通过中断从用户态切换到内核态。中断就是一个硬件或软件请求，要求CPU暂停当前的工作，去处理更重要的事情。比如，在x86机器上可以通过int指令进行软件中断，而在磁盘完成读写操作后会向CPU发起硬件中断。

同步阻塞IO（JAVA BIO）：

同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

同步非阻塞IO(Java NIO) ：

同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。用户进程也需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问。

异步阻塞IO（Java NIO）：

此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄（如果从UNP的角度看，select属于同步操作。因为select之后，进程还需要读写数据），从而提高系统的并发性！

（Java AIO(NIO.2)）异步非阻塞IO:

在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

同步阻塞：

在此种方式下，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式。

同步非阻塞：

在此种方式下，用户进程发起一个IO操作以后边可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。其中目前JAVA的NIO就属于同步非阻塞IO。

异步：

此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序。

## Lock 和Synchronized区别

**1.lock在获取锁的过程可以被中断。**

**2.lock可以尝试获取锁，如果锁被其他线程持有，则返回 false，不会使当前线程休眠。**

**3.lock在尝试获取锁的时候，传入一个时间参数，如果在这个时间范围内，没有获得锁，那么就是终止请求。**

**4.synchronized 会自动释放锁，lock 则不会自动释放锁。**

interrupted 是作用于当前线程（static），isInterrupted 是作用于调用该方法的线程对象所对应的线程。（线程对象对应的线程不一定是当前运行的线程。例如我们可以在A线程中去调用B线程对象的isInterrupted方法。）

可重入锁：指同一个线程，外层函数获得锁之后，内层递归函数仍有获得该锁的代码，但是不受影响。

Synchronized 独占锁

## JAVA并发编程

1. 线程调度：系统为线程分配处理器使用权的过程

协同式线程调度：线程的执行时间由线程本身来控制；

抢占式线程调度：线程由系统来分配执行时间。

1. 创建线程的三种方式：

1.继承Thread类

2.实现Runnable接口或者Callable接口（函数式接口，可以用lamda表达式创建）

注：Callable接口的call()方法可以有返回值、可以声明抛出异常，Future代表方法返回值，用FutureTask包装Callable。Future<V>代表一个异步执行的操作，通过get()方法可以获得操作的结果，如果异步操作还没有完成，则，get()会使当前线程阻塞。FutureTask<V>实现了Future<V>和Runable<V>。Callable代表一个有返回值得操作。

3.使用ExecutorService线程池

比较：实现Runnable接口这种方式更受欢迎（将任务的执行和任务本身分离），因为这不需要继承Thread类。在应用设计中已经继承了别的对象的情况下，这需要多继承（而Java不支持多继承），只能实现接口。同时，线程池也是非常高效的，很容易实现和使用。

1. 线程生命周期和传统不一样（JAVA线程是抢占式的）：

1.新建：new之后，分配内存，TCB，初始化成员变量，

2.运行：调用start方法后(不要调用run方法)，等待线程调度器调度执行，包括running和ready。

3.无限期等待：不会被分配CPU时间，需要被唤醒(不带时间参数的wait，join)。

4.限期等待：无须被唤醒（sleep，带时间参数的wait，join）

5.阻塞：等待获取一个排它锁

6.死亡：正常执行完，抛出异常，stop(不安全，不要使用，应该通过轮询一个boolean值)，回收TCB。此外，suspend不会释放锁，容易造成死锁。

1. 线程控制相关方法：

Join：阻塞当前调用线程，直到join线程执行完为止。

Yield：将当前线程转入就绪状态，将控制权返回给调用者，优先级比当前线程低的线程一样得不到执行。

注：sleep和wait的比较(都是本地方法，都会让出CPU)：a.原理不同，sleep是Thread类的静态方法，用于控制自身流程，wait是Object方法，用于线程通信；b.对锁的处理机制不同，sleep不会释放锁，c.使用区域不同，wait必须在同步代码块中使用，d.sleep必须捕获异常。

注：Sleep和yield的比较:Sleep和优先级无关，转阻塞状态，抛出中断异常，可移植性更高，永远不用yield，因为依赖于线程调度器。线程优先级：1-10，默认为6，避免直接指定优先级（因为操作系统的原因），最不可移植特征。

1. 后台线程：为其他线程提供服务的线程（如垃圾回收线程），前台线程死亡后，后台线程自动死亡，必须在启动之前设置。
2. Executor线程池（JAVA5）：

使用步骤

1．使用Executor工厂类产生线程池对象ExecutorService（缓存、固定数目等，注意不要使用newCached线程池，而应该使用带有固定上限数量的线程池）

2．使用Runnable或者Callable实现类创建执行任务

3．调用线程池的submit提交任务

4．结束时调用shutdown关闭线程池

注：JAVA8带了增强的ForkJoinPool支持将任务拆分成多个小任务进行并行计算（fork）。

private final ReentrantLock mainLock = new ReentrantLock(); //线程池的主要状态锁，对线程池状态（比如线程池大小、runState等）的改变都要使用这个锁

1. 线程同步：

Synchronized：锁非公平，可重入，锁定对象的同步监视器，保证块中变量可见性，修饰方法和代码块，但不能修饰构造器和成员变量，无法中断正在等待获取锁的线程，必须在该锁的代码块中释放。

同步锁：Lock（需要在finally中显式unlock，），为高级工具，Lock提供了可轮询，可定时，可中断的，公平锁获取操作，读写锁等

比较：synchronized是在JVM层面上实现的，不但可以通过一些监控工具监控synchronized的锁定，而且在代码执行时出现异常，JVM会自动释放锁定，但是使用Lock则不行，lock是通过代码实现的，要保证锁定一定会被释放，就必须将unLock()放到finally{}中。只有在内置锁不能满足要求时，才考虑Lock，就性能而言，在java6之前，Lock性能更好，但是Synchronized是JVM内置属性，JVM能执行一些优化(自旋锁，锁消除，锁粗化，锁偏向等)，将来性能肯定会高于Lock。 相同点：都是可重入的，在同步块的前后插入monitorenter和monitorexit字节码，重量级的锁。

原子类（Atomics）：非阻塞的无锁，内部通过JNI方式使用硬件支持指令CAS（基于比较并交换），性能大大提升。无锁并行框架：Amino

1. 线程通信（已经没有必要使用1和2.应该使用高级同步工具）：

1，Object类：在Synchronized 同步方法或者代码块内使用，wait阻塞当前线程，释放同步监视器，notify唤醒在此同步监视器上等待的任一线程，notifyall唤醒此同步监视器上等待的所有线程，让他们竞争（总是使用notifyall）；

2.condition:绑定在Lock对象上，使用await，signal，signalall

3.阻塞队列：BlockingQueue：支持列表，链表，优先级，同步队列（交替执行）等方式（适用于生产者消费者模式）

1. 高级同步工具（4个）：

CountDownLatch（倒计数）：线程到达预期状态调用countDown方法触发事件，其他线程调用await等待此事件开始自己后续工作(CountDownLatch.await() 方法在倒计数为0之前会阻塞当前线程)。

CyclicBarrier(循环屏障):可以协同多个线程，让多个线程在此屏障前等待，直到所有线程到达，再一起执行后面的动作，await() 方法没被调用一次，计数便会减少1，并阻塞住当前线程。当计数减至0时，阻塞解除，所有在此 CyclicBarrier 上面阻塞的线程开始运行。在这之后，如果再次调用 await() 方法，计数就又会变成 N-1，新一轮重新开始，这便是 Cyclic 的含义所在。

区别：CountDownLatch 适用于一组线程和另一个主线程之间的工作协作。一个主线程等待一组工作线程的任务完毕才继续它的执行是使用 CountDownLatch 的主要场景；CyclicBarrier 用于一组或几组线程，比如一组线程需要在一个时间点上达成一致，例如同时开始一个工作。CyclicBarrier 的循环特性是 CountDownLatch 所不具备的。

Semaphore：通过构造函数设定一个数量的许可，然后通过 acquire 方法获得许可，release 方法释放许可。Semaphore 是只允许一定数量的线程同时执行一段任务。

Exchanger:用于两个线程之间交换数据，调用exchange方法。

1. 并发容器（不要使用外部同步的集合）：

ConcurrentHashMap： 在HashMap的基础上，将数据分段存储，由多个Segment组成，每个Segment都有把锁。Segment下包含很多HashEntry，也就是键值对。

注：Java8中，使用了CAS算法，与Java8的HashMap有相通之处，底层依然由数组+链表+红黑树结构。底层结构存放的是TreeBin对象，而不是TreeNode对象；Segment虽保留，但已经简化属性，仅仅是为了兼容旧版本。

sizeCtl是控制标识符，不同的值表示不同的意义。负数代表正在进行初始化或扩容操作 ，-1代表正在初始化 ，-N 表示有N-1个线程正在进行扩容操作 ；正数或0代表hash表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小，类似于扩容阈值。它的值始终是当前ConcurrentHashMap容量的0.75倍，这与loadfactor是对应的。实际容量>=sizeCtl，则扩容。

Node节点：value和next增加了volatile属性，保证变量可见性。

TreeNode节点：继承上述Node节点，相当简洁；ConcurrentHashMap链表转树时，并不会直接转，只是把这些节点包装成TreeNode放到TreeBin中，再由TreeBin来转化红黑树。

TreeBin节点：当链表转树时，用于封装TreeNode。

ForwardingNode节点：不是我们传统的包含key-value的节点，只是一个标志节点，用于连接两个table，指向nextTable，仅存活于扩容操作，一定会出现在每个bin的首位。

ConcurrentHashMap构造函数可以指定初始容量，负载因子；初始化table表的操作在第一次put时进行，特别说明 concurrencyLevel指能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，在Java8之前实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。在Java8里，仅仅是为了兼容旧版本而保留。唯一的作用就是保证构造map时初始容量不小于concurrencyLevel。

扩容操作：无锁多线程扩容，减少扩容时的时间消耗。单线程构建两倍容量的nextTable；允许多线程复制原table元素到nextTable。若nextTab为null，则初始化其为原table的2倍；死循环遍历，直到结束。finishing时，nextTab赋给table，更新sizeCtl为新容量的0.75倍 ，完成扩容。多线程下，当前线程检测到其他线程正进行扩容操作，则协助其一起扩容；遍历到ForwardingNode节点((fh = f.hash) == MOVED)，说明此节点被处理过了，直接跳过。这是控制并发扩容的核心 。由于给节点上了锁，只允许当前线程完成此节点的操作，处理完毕后，将对应值设为ForwardingNode（fwd），其他线程看到forward，直接向后遍历。如此便完成了多线程的复制工作，也解决了线程安全问题。

Put操作：①判空：null直接抛空指针异常；②hash：计算h=key.hashcode；调用spread计算hash=(h ^(h >>>16))& HASH\_BITS；③遍历table，若table为空，则初始化，仅设置相关参数；计算当前key存放位置，即table的下标i=(n - 1) & hash；若待存放位置为null，casTabAt无锁插入；若是检测到正在扩容，则帮助其扩容；待插入位置非空且不在扩容，将头节点上锁（保证了线程安全）：区分链表节点和树节点，分别插入（遇到hash值与key值都与新节点一致的情况，只需要更新value值即可。否则依次向后遍历，直到链表尾插入这个结点）；若链表长度>8，则转红黑树。④调用addCount(1L， binCount)： 利用CAS快速更新baseCount的值；check>=0.则检验是否需要扩容；if sizeCtl<0（正在进行初始化或扩容操作）。

Get操作：整个get过程不需要加锁，get方法里将要使用的共享变量，都定义成volatile①spread计算hash值；②table不为空；③tabAt(i)处桶位不为空；④check first，是则返回当前Node的value；否则分别根据树、链表查询。

CopyOnWrite开头（ArrayList，ArraySet）：更改容器的时候采用底层数组方式复制容器的数据，保证读不受影响，适用于读多写少的场景，比如缓存。

Put时同时get，即使get 同时put（正在new这个对象）的值，也不会影响（readValueUnderLock）；更新由于volatile，也不会影响；get删除（remove）的值，会得到，但并不影响

Clear前面一半，前面的一半put，在clear后面一半，则会造成map里面没有clear干净，还有前面的一半；迭代的时候，前面的一半迭代过，再删除，那么不会再迭代器中反应出来（迭代显示存在），体现了弱一致性

ConcurrentHashMap中的迭代器主要包括entrySet、keySet、values方法。它们大同小异，这里选择entrySet解释。当我们调用entrySet返回值的iterator方法时，返回的是EntryIterator，在EntryIterator上调用next方法时，最终实际调用到了HashIterator.advance()方法。这个方法在遍历底层数组。在遍历过程中，如果已经遍历的数组上的内容变化了，迭代器不会抛出ConcurrentModificationException异常。如果未遍历的数组上的内容发生了变化，则有可能反映到迭代过程中。这就是ConcurrentHashMap迭代器弱一致的表现。ConcurrentHashMap的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。

主要是modCount改变，modCount代表改变次数

Size操作：先是是不锁表的，但是在多线程的情况下，就会造成数据的不一致，这里就用到了Segment中的modCount来做比较，如果modCount有变化就说明被其他线程污染了，就需要重新做统计，这个时候也是不带锁的。但是这样的循环不可能无限进行下去，所以做了限制，在不带锁的情况下允许进行2次尝试，如果还是受到其他线程的污染，那就要加锁统计了。

a） 在内存中分配空间，并将引用指向该内存空间。

b） 执行对象的初始化的逻辑(和操作)，完成对象的构建。

加锁操作：（put remove clear modcount改变）replace

不加锁操作：get containskey contains

全局加锁：size containsValue

遍历1增强for（底层还是iterator） 2iterator

Iterator：一段一段的遍历，当获取到下一个段的table，也就意味着这个段的头结点在迭代过程中就确定了，在迭代过程中就不能反映对这个段节点并发的删除和添加，对于节点的更新是能够反映的，因为节点的值是一个volatile变量；

已经遍历过的修改不会反应出来，未遍历过的修改会反应出来，

Remove：由于next是final的，因此需要重新new一个hashEntry，颠倒，即原本的顺序：E3-->E2-->E1-->E0，删除E1节点后的顺序：E2-->E3-->E0

Rehash：需要new hashEntry，最后table指向newTable，oldtable自动释放

1. ThreadLocal：线程局部变量，为每个使用该变量的线程都提供一个副本，每一个线程可以独立的改变自己的副本，而不会和其他线程冲突。

注意：不能代替同步机制，面向的问题领域不同，同步是为了资源的并发访问，是线程通信的有效方式，而ThreadLocal仅仅是为了隔离多个线程的数据共享，从根本上避免了资源竞争，线程之间是独立的，不需要通信。

1. Thread类中有一个成员变量叫做ThreadLocalMap，它是一个Map，他的Key是ThreadLocal类

2. 每个线程拥有自己的申明为ThreadLocal类型的变量，所以这个类的名字叫ThreadLocal：线程自己的（变量）。

3. 此变量生命周期是由该线程决定的，开始于第一次初始化（get或者set方法）。

4. 由ThreadLocal的工作原理决定了：每个线程独自拥有一个变量，并非共享或者拷贝。

static class Entry extends WeakReference<ThreadLocal<?>>弱引用

1. 线程安全级别（5种）：

1.不可变

不变的对象绝对是线程安全的，不需要线程同步，如String、Long、BigInteger

2.无条件的线程安全

对象自身做了 足够的内部同步，也不需要外部同步，如 Random 、ConcurrentHashMap、Concurrent集合、atomic

3.有条件的线程安全

对象的部分方法可以无条件安全使用，但是有些方法需要外部同步，需要Collections.synchronized；有条件线程安全的最常见的例子是遍历由 Hashtable 或者 Vector 或者返回的迭代器

4.非线程安全(线程兼容)

对象本身不提供线程安全机制，但是通过外部同步，可以在并发环境使用， 如ArrayList HashMap

5.线程对立

即使外部进行了同步调用，也不能保证线程安全，这种情况非常少

1. 线程的三种实现方式：

1.使用内核线程：直接由内核完成线程切换，内核通过操纵调度器完成线程调度，将线程任务映射到各个处理器，程序使用高级接口-轻量级进程获得内核线程（1:1）

2.使用用户级线程：完全建立在用户控件线程库上，，不需要内核的支援，所有操作都需要自己处理，只是把处理器资源分配给进程。（1:N）

3.混合实现：将内核线程与用户线程一起使用（N:M）

注：JAVA线程实现：Windows和linux都是一对一模型，Solaris可配置。

## Lock源码

1）首先基于CAS将state（锁数量）从0设置为1，如果设置成功，设置当前线程为独占锁的线程；-->请求成功-->第一次插队

2）如果设置失败(即当前的锁数量可能已经为1了，即在尝试的过程中，已经被其他线程先一步占有了锁)，这个时候当前线程执行acquire(1)方法

2.1）acquire(1)方法首先调用下边的tryAcquire(1)方法，在该方法中，首先获取锁数量状态，

2.1.1）如果为0(证明该独占锁已被释放，当下没有线程在使用)，这个时候我们继续使用CAS将state（锁数量）从0设置为1，如果设置成功，当前线程独占锁；-->请求成功-->第二次插队；当然，如果设置不成功，直接返回false

2.2.2）如果不为0，就去判断当前的线程是不是就是当下独占锁的线程，如果是，就将当前的锁数量状态值+1（这也就是可重入锁的名称的来源）-->请求成功

下边的流程一句话：请求失败后，将当前线程链入队尾并挂起，之后等待被唤醒。

公平锁会判断是否有线程等待时间超过当前线程

# Java虚拟机

## JAVA内存模型

1. 背景：硬件的效率与一致性问题：

由于计算机的存储设备与处理器的运算能力之间有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（cache）来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中没这样处理器就无需等待缓慢的内存读写了。  
　　基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是引入了一个新的问题：缓存一致性（Cache Coherence）。在多处理器系统中，每个处理器都有自己的高速缓存，而他们又共享同一主存，如下图所示：多个处理器运算任务都涉及同一块主存，需要一种协议可以保障数据的一致性。

Java虚拟机内存模型中定义的内存访问操作与硬件的缓存访问操作是具有可比性的。

1. 重排序问题：

1.编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。

2.指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，可以改变语句对应机器指令的执行顺序。

3.内存系统的重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

1. 定义：

目的是用于屏蔽掉各种硬件和操作系统的内存访问差异，以实现让java程序在各个平台下都能达到一致的内存访问效果。定义程序中各个变量的访问规则，在虚拟机中将变量（不包括局部变量）存储到内存和从内存中取出这样的底层细节。规定了JVM必须遵循的一组最小保证，即对变量的写入操作在何时将对其他线程可见。（变量值的传递均需要通过主内存完成）围绕并发过程中如何处理原子性，可见性，有序性这3个特征建立。

原子性：基本变量的访问读写都是原子的，提供lock和unlock满足更大范围的原子性保证（monitorenter和monitorexit）

可见性：定义了一个线程在哪种情况下可以访问另外一个线程或者影响另外一个线程（volatile，synchronized，final能保证可见性）当一个线程修改了共享变量的值，其他线程能够立即得知这个修改

有序性：线程内所有操作都是有序的，但是从一个线程观察另一个线程都是无序的（volatile，synchronized保证，因为volatile禁止指令重排序，synchronized一个变量一个时刻只允许一条线程进行lock）

根据JMM的设计，系统存在一个主内存(Main Memory)，Java中所有实例变量都储存在主存中，对于所有线程都是共享的。每条线程都有自己的工作内存(Working Memory)，工作内存由缓存和堆栈两部分组成，缓存中保存的是主存中变量的拷贝，缓存可能并不总和主存同步，也就是缓存中变量的修改可能没有立刻写到主存中；堆栈中保存的是线程的局部变量，线程之间无法相互直接访问堆栈中的变量。

1. 内存间交互操作（保证都是原子的）

Lock，unlock，read， load，use，assign，store，write

1. 先行发生（Happens-Before规则）共8种（不代表时间上会先行发生）

程序次序规则，监视器锁定规则，volatile规则，线程启动规则，线程终止规则，线程中断规则，终结器规则，传递性规则，如果不满足上述规则，就有可能会被重排序。；

Volatile的特殊规则：保证可见性，插入内存屏障指令来禁止重排序，每次都从主存中读取，写完以后立刻写到主存，并不保证原子性。也就是说对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值才能使用volatile。由于使用volatile屏蔽掉了JVM中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

volatile和synchronized区别

volatile不会进行加锁操作：volatile变量是一种稍弱的同步机制在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比synchronized关键字更轻量级的同步机制。

volatile 变量作用类似于同步变量读写操作：从内存可见性的角度看，写入volatile变量相当于退出同步代码块，而读取volatile变量相当于进入同步代码块。

volatile 不如 synchronized安全：在代码中如果过度依赖volatile变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当volatile变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。

volatile 无法保证原子性：加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而volatile变量只能确保可见性，原因是声明为volatile的简单变量如果当前值与该变量以前的值相关，那么volatile关键字不起作用。

volatile变量法则：对volatile域的写入操作happens-before于每一个后续对同一个域的读写操作。

处理器为了提高处理速度，不直接和内存进行通讯，而是先将系统内存的数据读到内部缓存（L1,L2或其他）后再进行操作，但操作完之后不知道何时会写到内存，如果对声明了Volatile变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

线程在回读取主内存中的值到线程本地内存中，进行操作，操作完后并不会立即写入主内存中。

内存语义：当写一个volatile变量时，JMM（java内存模型）会把该线程对应的本地内存中的共享变量值刷新到主内存；当读一个volatile变量时，JMM会把该线程对应的本地内存只为无效。线程接下来将从主内存中读取共享变量。

\*\*\*\*\*\*原理

有volatile变量修饰的共享变量在进行写操作的时候会多出一行带有lock前缀的汇编代码，lock前缀的指令在多核处理器下会引发两件事情：

1、将当前处理器缓存行的数据写回到系统内存；

2、这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效；

1. lock前缀指令会引起处理器缓存写回到内存

2.一个处理器的缓存写回到内存会导致其他处理器的缓存无效

处理器为了提高处理速度，不直接和内存进行通信，而是先将系统内存的数据读到线程内部缓存后再进行操作，但操作完成后不知道什么时候会写回到内存，如果对声明了volatile的变量进行写操作，jvm就会向处理器发送一条lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是即使写回到内存中，如果其他处理器缓存的值还是旧的，再执行计算操作还是会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置为无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存中把数据读到当前处理器缓存里。

增强语义：volatile的写-读没有监视器的释放-获所具有的内存语义。为了提供一种比监视器锁更轻量级的线程之间通信的机制，JSR-133专家组决定增强volatile的内存语义：严格限制编译器和处理器对volatile变量与普通变量的重排序，确保volatile的写-读和监视器的释放-获取一样，具有相同的内存语义。从编译器重排序规则和处理器内存屏障插入策略来看，只要volatile变量与普通变量之间的重排序可能会破坏volatile的内存语义，这种重排序就会被编译器重排序规则和处理器内存屏障插入策略禁止。

## Javac编译过程（4个步骤）

1. 词法分析：读取源代码，一个字节一个字节的读进来，找出这些词法中哪些是我们定义的语言关键词如：if  else while等。

词法分析的结果：就是从源代码中找出一些规范化的token流，就像人类语言中，给你一句话你要分辨出哪些是一个词语，标点符号，动词，名词。

1. 语法分析：就是在词法分析中得到的token流进行语言分析，这一步就是检查这些关键词组合在一起是不是符合java语言规范。如if的后面是不是紧跟着一个布尔型判断表达式。

语法分析的结果：就是形成一个符合java语言规定的抽象语法树，抽象语法树是一个结构化的语法表达形式，它的作用是把语言的主要词法用一个结构话的形式组织在一起。这棵语法树可以被我们后面按照新的规则再重新组织。

1. 语义分析：把一些难懂的，复杂的语法转化成更简单的语法。就如难懂的文言文转化为大家都懂的百话文。给类添加默认构造函数，检查受检异常是否捕获或者抛出，解除Java语法糖，处理注解，泛型信息擦除，字符串常量合并，自动类型转化等。

语义分析结果：就是将复杂的语法转化为简单的语法。最后生成一棵抽象的语法树，这棵语法树也就更接近目标语言的语法规则。

备注：语法糖：在计算机中添加的某种语法，这种语法对语言功能没有影响，但是能够增加程序可读性，减少出错机会，方便程序员使用（泛型、自动装箱拆箱、foreach，变长参数，内部类、枚举类、断言等等）

1. 代码生成： 将会根据经过注释的抽象语法树生成字节码，也就是将一个数据结构转化为另外一个数据结构。就像将所有的中文词语翻译成英文单词后按照英文语法组装文英文语句。代码生成器的结果就是生成符合java虚拟机规范的字节码。

## Class文件结构

1. 概述：Class文件是一组以8位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在Class文件之中，中间没有添加任何分隔符，这使得整个Class文件中存储的内容几乎全部都是程序运行的必要数据，没有空隙存在。当遇到需要占用8位字节以上的空间的数据项时，则会按照高位在前的方式分割成若干个8位字节进行存储。
2. 两种数据类型：无符号数和表。

无符号数属于基本的数据类型，以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节、8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值，或者按照UTF-8编码构成字符串值。

表是由多个无符号数或者其它表作为数据项构成的复合数据类型，所有表都习惯性地以"\_info"结尾。表用于描述有层次关系的复合结构的数据，整个Class文件本质上就是一张表。 它由下表所示的数据项构成。

1. 内容

1. 魔数(u4)： 0xCAFEBABE，用于确定这个文件是否为一个能被虚拟机接受的Class文件。

2.版本号(u4)：（次版本，主版本）。Java的版本号是人45开始的

3. 常量池：开头是常量数量，主要存放两大类常量：字面量和符号引用。字面量如文本字符串、被声明为final的常量值等。符号引用包括了下面三类常量: 类和接口的全限定名字段的名称和描述符， 方法，字段的名称和描述符。

4.访问标志（u4）：用于识别一些类或者接口的访问信息。

5.类索引（u2），父类索引(u2)，接口索引集合(u2集合)

6.字段数量和表集合：用于描述类变量和实例变量（不含局部变量）

7.方法数量和表集合：（不含方法体）

8.属性表集合（Class文件，字段，方法的附加描述），比如字段的初始值，方法体，方法异常表等，内部类等信息。

注意：对于非静态实例变量在实例构造器中初始化，对于静态变量，使用类构造器方法或者ConstantValue属性。

## 字节码指令

1. Java 虚拟机的指令由一个字节长度的、代表着某种特定操作含义的操作码（Opcode）以及跟随其后的零至多个代表此操作所需参数的操作数（Operands）所构成。虚拟机中许多指令并不包含操作数，只有一个操作码。
2. 对于大部分为与数据类型相关的字节码指令，他们的操作码助记符中都有特殊的字符来表明专门为哪种数据类型服务：i 代表对 int 类型的数据操作，l 代表 long，s 代表 short，b 代表 byte，c 代表 char，f 代表 float，d 代表 double，a 代表 reference。
3. 主要有:加载存储指令，运算指令，类型转换指指令，对象创建于访问指令，操作数栈管理指令，控制转移指令，方法调用和返回指令，异常处理指令，同步指令。

## 类加载机制

1. 概念：类加载机制是虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型的过程就是虚拟机的类加载机制。
2. 类的生命周期：加载，验证，准备，解析，初始化，使用，卸载。
3. 类的初始化时机（5种）

1）创建类的实例，也就是new一个对象

2）访问某个类或接口的静态变量，或者对该静态变量赋值，静态方法

3）反射（Class.forName("com.lyj.load")）

4）初始化一个类的子类（会首先初始化子类的父类）

5）JVM启动时标明的启动类，即文件名和类名相同的那个类

注意：class对象只由JVM创建，new 创建对象时静态加载类，因为类必须在编译阶段提供，forname动态加载类，编译阶段不必提供；

1. 加载：

1.获取类的.class文件中的二进制数据，读入到内存中，

2.将其代表的静态存储结构转化为方法区的运行时数据结构，

3.创建一个这个类的Class对象，作为方法区此类数据的访问入口。

注意：Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口，hotspot虚拟机的class对象在方法区。

1. 验证：确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。（文件格式，元数据，字节码，符号引用验证）。
2. 准备：为类变量分配内存并设置类变量初始值(各数据类型的零值)的阶段，这些内存将在方法区中进行分配。但是如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量值就会初始化为ConstantValue属性指定的值。
3. 解析：

将常量池内的符号引用替换为直接引用（类或接口，字段，方法等）。

符号引用：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。

直接引用：直接引用可以是直接指向目标的指针、相对偏移量或者一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

1. 初始化和实例化：类的初始化是指类加载过程中的初始化阶段对类变量按照程序猿的意图进行赋值的过程；而类的实例化是指在类完全加载到内存中后创建对象的过程。
2. 实例初始化方法<init>
3. New 一个对象的流程：检测类是否被加载没有加载的先加载→为新生对象分配内存→将分配到的内存空间都初始化为零值→对对象进行必要的设置→执行<init>方法把对象进行初始化→返回堆中这个对象的引用
4. 初始化: 类初始化方法

执行类构造器<clinit>()方法的过程；

<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static{}块)中的语句合并产生的，不需要先初始化父类构造器，也非必须。

1. 类加载器(双亲委派模型)

1）Bootstrap ClassLoader

负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类，由于引导类加载器涉及到虚拟机本地实现细节， 开发者无法直接获取到启动类加载器的引用， 所以不允许直接通过引用进行操作。

2）Extension ClassLoader

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

3）App ClassLoader

负责记载classpath中指定的jar包及目录中class

4）Custom ClassLoader

属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

注：加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

## 虚拟机执行引擎

1. 栈帧结构:用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈的栈元素。存储方法的局部变量表、操作数栈、动态链接和方法返回地址等信息。每一个方法从调用开始到执行完成的过程，就对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

局部变量表：一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。局部变量表容量以Slot为最小单位，一个Slot可以存放一个32位以内的数据类型。

操作数栈：即用来存放操作数的栈结构，当一个方法刚开始执行的时候，这个方法的操作数栈是空的

动态连接：每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转化为直接引用，这种转化为称为静态解析，另外一部分将在每一次的运行期间转化为直接引用，这部分称为动态连接。

1. 确定方法调用的过程（解析调用和分派调用）

解析调用：所有方法在class文件里面都是一个常量池中的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析能成立的前提是，方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。

注意：符合这个条件的有静态方法，私有方法，实例构造器和父类方法四类，它们在类加载的时候会把符号引用解析为该方法的直接引用。解析调用一定是一个静态的过程，编译期间就完全确定，在类装载的解析阶段就会把涉及到的符号引用全部转化为可确定的直接引用，不会延迟到运行期间再去完成。

分派调用：分派调用可能是静态的也可能是动态的，根据分派依据的宗量数又可分为单分派和多分派。分派机制与java的多态机制关系密切。

1. 静态分派：依赖静态类型来定位方法执行版本的分派动作，称为静态分派。静态分派的最典型的应用就是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。

 2.动态分派：在运行期间根据实际类型来确定方法执行版本的分派调用过程称为动态分派。这跟多态性的另一个体现——重写有着很密切的关联。

注：方法的接收者与方法的参数统称为方法的宗量。单分派：根据一个宗量对目标方法进行选择，多分派：根据多于一个的总量对目标方法进行选择。jdk1.6时期的java语言是一种静态多分派、动态单分派的语言。

## 运行时数据区

1. 程序计数器

程序计数器是一块较小的区域，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的模型里，字节码指示器就是通过改变程序计数器的值来指定下一条需要执行的指令。分支，循环等基础功能就是依赖程序计数器来完成的。

由于java虚拟机的多线程是通过轮流切换并分配处理器执行时间来完成，一个处理器同一时间只会执行一条线程中的指令。为了线程恢复后能够恢复正确的执行位置，每条线程都需要一个独立的程序计数器，以确保线程之间互不影响。所以程序计数器是“线程私有”的内存。

如果虚拟机正在执行的是一个Java方法，则计数器指定的是字节码指令对应的地址，如果正在执行的是一个本地方法，则计数器指定问空undefined。程序计数器区域是Java虚拟机中唯一没有定义OutOfMemory异常的区域。

1. 虚拟机栈

和程序计数器一样也是线程私有的，生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会创建一个栈帧用于存储局部变量表，操作栈，动态链接，方法出口等信息。每一个方法被调用的过程就对应一个栈帧在虚拟机栈中从入栈到出栈的过程。如果线程请求的栈深度超出了虚拟机所允许的深度，就会出现StackOverFlowError。-Xss规定了栈的最大空间虚拟机栈可以动态扩展，如果扩展到无法申请到足够的内存，会出现OOM。

1. 堆

堆是Java虚拟机所管理的内存中最大的一块。堆是所有线程共享的一块区域，在虚拟机启动时创建。堆的唯一目的是存放对象实例，几乎所有的对象实例都在这里分配，不过随着JIT编译器的发展和逃逸技术的成熟，栈上分配和标量替换技术使得这种情况发生着微妙的变化，堆上分配正变得不那么绝对。

1. 方法区

方法区也是线程共享的区域，用于存储已经被虚拟机加载的类信息，常量，静态变量和即时编译器（JIT）编译后的代码等数据。Java虚拟机把方法区描述为堆的一个逻辑分区，不过方法区有一个别名Non-Heap(非堆)，用于区别于Java堆区。

1. 运行时常量池

运行时常量池是方法区的一部分，Class文件中除了有类的版本，字段，方法，接口等信息以外，还有一项信息是常量池用于存储编译器生成的各种字面量和符号引用，这部分信息将在类加载后存放到方法区的运行时常量池中。运行期间也可将新的常量放入常量池中。

1. 此外，还有本地方法栈，直接内存等。（对象的访问定位目前有使用句柄和直接指针两种方式）

## 垃圾回收

1. 垃圾回收回收的是无任何引用的对象占据的内存空间而不是对象本身。

注：静态内存分配：局部变量在java栈上进行分配。

Java对象占用内存分析：局部变量以Slot为最小单位（4个字节），堆上对象分配：

规则一：new Object()占用8个字节（锁标志位，经历GC次数，指向类对象的引用）。

规则二：为类属性分配存储空间时不是按照类中定义的属性顺序。

            1、double\long；----8bytes

            2、int\float；----4bytes

            3、char\short；----2bytes

            4、boolean\byte；----1bytes

            5、reference；----4bytes

规则三：对于继承时，要按照规则二先计算父类的类属性占用情况，再按照规则二计算子类的类属性占用情况，不能将父类和子类的属性混合在一起按规则二分配。

注：每个对象被按照8bytes粒度对齐，不足的话就要填充padding，数组不一样， 对于数组，与普通对象不同的是在头部，头部多了4个字节用于存储长度信息。因此数组的head是12bytes而不是8bytes。

1. 堆内存划分：

年轻代：1个Eden区，2个Survivor区（必须有一个为空）；

老年代：存放年轻代的Survivor区满后触发minorGC后仍然存活的对象，如果老年代也满了，触发FullGC。

永久代：也称方法区，主要回收废弃类和无用常量，当满了之后同样触发FullGC。

注：JAVA8中已经移除了永久代，改用元空间。元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。

注：Minor GC 会清理年轻代（满或者无法分配大对象），Major GC 是清理永久代。Full GC 是清理整个堆空间—包括年轻代和永久代。

1. 确定一个对象为垃圾：

可达性分析法。该方法的基本思想是通过一系列的“GC Roots”对象（局部变量，栈等）作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。

1. 垃圾回收算法：

标记-清除算法：两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。

标记-清除算法实现起来比较容易，但是有一个比较严重的问题就是容易产生内存碎片，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。

复制算法：它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。

标记-整理算法：该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。

分代收集算法（目前常用）：根据对象存活的生命周期将内存划分为若干个不同的区域，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。

1. 典型的垃圾收集器：

1.Serial/Serial Old

Serial/Serial Old收集器是最基本最古老的收集器，它是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial收集器是针对新生代的收集器，采用的是Copying算法，Serial Old收集器是针对老年代的收集器，采用的是Mark-Compact算法。它的优点是实现简单高效，但是缺点是会给用户带来停顿。

2.ParNew

ParNew收集器是Serial收集器的多线程版本，使用多个线程进行垃圾收集。

3.Parallel Scavenge

Parallel Scavenge收集器是一个新生代的多线程收集器（并行收集器），它在回收期间不需要暂停其他用户线程，其采用的是Copying算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

4.Parallel Old

Parallel Old是Parallel Scavenge收集器的老年代版本（并行收集器），使用多线程和Mark-Compact算法。

5.CMS

CMS（Current Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是Mark-Sweep算法。

6.G1

G1收集器是面向服务端应用的收集器，它能充分利用多CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。对垃圾回收进行了划分优先级的操作，这种有优先级的区域回收方式保证了它的高效率；最大的优点是结合了空间整合，不会产生大量的碎片，也降低了进行gc的频率让使用者明确指定指定停顿时间。

## JIT即时编译器

1. 如果有些动态极度频繁的被执行，就会被自动编译成机器码，跳过其中的部分环节， 当开启JIT时，JVM会分析Java应用程序的函数调用并且（达到内部一些阀值后）编译字节码为本地更高效的机器码。JIT流程通常为最繁忙的函数调用提供更高的优先级，一旦函数调用被转为机器码，JVM会直接执行而不是“解释执行”。
2. 常用编译优化技术：公共子表达式消除，数组边界检查消除、方法内联、逃逸分析技术（分析对象作用域的方式，如果确实不会逃逸出方法，则优化：栈上分配，同步消除，标量替换等技术）

# Web相关

## Web概述

注重业务繁琐的做做OA项目，学学工作流，学学activity；注重数据及时性的做做redis的项目，比如抢红包，抢订单，学学非关系型内存数据库；注重线程同步和异步性的，比如匹配问题，做做线程池，多线程的项目，学学线程同步的思想；想了解嵌入型数据库，可以看看H2数据库；想了解一门脚本解释型语言，看看groovy，python；你说你做过ssm基础项目。用过mybatis拦截器吗？用过缓存吗？用过4张表以上的查询吗？用过spring的aop吗？我说spring的监听器吗？....说的挺杂的，因为java这技术范围就是又杂又广~既然选择了远方，就只顾风雨兼程吧~

## SearchEngine

* Spring管理的对象成为beans
* Spring支持setter和构造器方式的依赖注入，2.5后支持Autowired注解注入（基于field方式）
* Spring支持XML或注解两种配置方式
* Spring MVC 好处 p34
* 传统的Spring做法是使用.xml文件来对bean进行注入或者是配置aop、事物，这么做有两个缺点：

1、如果所有的内容都配置在.xml文件中，那么.xml文件将会十分庞大；如果按需求分开.xml文件，那么.xml文件又会非常多。总之这将导致配置文件的可读性与可维护性变得很低

2、在开发中在.java文件和.xml文件之间不断切换，是一件麻烦的事，同时这种思维上的不连贯也会降低开发的效率

为了解决这两个问题，Spring引入了注解，通过"@XXX"的方式，让注解与Java Bean紧密结合，既大大减少了配置文件的体积，又增加了Java Bean的可读性与内聚性。

* @Autowired注解的意思就是(完成自动装配)，当Spring发现@Autowired注解时，将自动在代码上下文中找到和其匹配（默认是类型匹配）的Bean，并自动注入到相应的地方去。
* 虽然我们可以通过 @Autowired 或 @Resource 在 Bean 类中使用自动注入功能，但是 Bean 还是在 XML 文件中通过 <bean> 进行定义 —— 也就是说，在 XML 配置文件中定义 Bean，通过@Autowired 或 @Resource 为 Bean 的成员变量、方法入参或构造函数入参提供自动注入的功能。能否也通过注释定义 Bean，从 XML 配置文件中完全移除 Bean 定义的配置呢？答案是肯定的，我们通过 Spring 2.5 提供的@Component 注释就可以达到这个目标了。通过在类上使用 @Repository、@Component、@Service 和 @Constroller 注解，Spring 会自动创建相应的 BeanDefinition 对象，并注册到 ApplicationContext 中。这些类就成了 Spring 受管组件。这三个注解除了作用于不同软件层次的类，其使用方式与 @Repository 是完全相同的。
* @Component和xml配置是等效的，@Component注解在类上面进行声明后，可以被spring容器识别，Spring容器自动将POJO转换为容器管理的bean

## Servlet&JSP

1. BS和CS的区别以及优缺点   
   C/S称客户/服务器模式。客户端需要安装专用的客户端软件。   
   B/S是Brower/Server的缩写，客户机上只要安装一个浏览器，浏览器通过Web Server 同数据库进行数据交互。   
   C/S的优点是能充分发挥客户端PC的处理能力，很多工作可以在客户端处理后再提交给服务器。对应的优点就是客户端响应速度快。缺点主要有以下几个：   
   只适用于局域网。而随着互联网的飞速发展，移动办公和分布式办公越来越普及，这需要我们的系统具有扩展性。这种方式远程访问需要专门的技术，同时要对系统进行专门的设计来处理分布式的数据。   
   客户端需要安装专用的客户端软件。首先涉及到安装的工作量，其次任何一台电脑出问题，如病毒、硬件损坏，都需要进行安装或维护。特别是有很多分部或专卖店的情况，不是工作量的问题，而是路程的问题。还有，系统软件升级时，每一台客户机需要重新安装，其维护和升级成本非常高。   
   对客户端的操作系统一般也会有限制。可能适应于Win98， 但不能用于win2000或Windows XP。或者不适用于微软新的操作系统等等，更不用说Linux、Unix等。   
   B/S最大的优点就是可以在任何地方进行操作而不用安装任何专门的软件。只要有一台能上网的电脑就能使用，客户端零维护。系统的扩展非常容易，只要能上网，再由系统管理员分配一个用户名和密码，就可以使用了。甚至可以在线申请，通过公司内部的安全认证（如CA证书）后，不需要人的参与，系统可以自动分配给用户一个账号进入系统。
2. Get 和Post区别：a.GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。GET 请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。b.get是把参数数据加到提交URL中，值和表单内各个字段一一对应，在URL中可以看到。post将表单内各个字段与其内容放置在http协议（请求报文）包体内。用户看不到这个过程。C.get传送的数据量较小，不能大于2KB。post传送的数据量较大，一般被默认为不受限制。D.get安全性非常低，post安全性较高。但是执行效率却比Post方法好。
3. Servlet：采用java语言编写的服务端程序，运行于Web服务端的servlet容器中，主要功能是提供请求/响应的Web服务模式。相当于CGI的优点：较好的可移植性、执行效率高（CGI为每个请求创建进程，而Servlet是线程，每个servlet在容器中只有一个实例），功能强大，可以直接与Web服务器交互（运行在服务器进程中），使用方便，可扩展性强。
4. Servlet生命周期：由容器控制（无main方法），可以分为加载，创建实例，初始化init，处理客户端请求service和卸载destroy。
5. Servlet线程安全问题: 实例变量不正确的使用是造成Servlet线程不安全的主要原因。解决办法：a.实现 SingleThreadModel 接口， 确保不会有两个线程同时执行servlet的service方法。 servlet容器通过同步化访问servlet的单实例来保证，也可以通过维持servlet的实例池，对于新的请求会分配给一个空闲的servlet。b.同步对共享数据的操作； c.避免使用实例变量。

为了开发线程安全的Servlet，我们应该尽可能地做到：

（1）尽可能地在Servlet中只使用本地变量。

（2）应该只使用只读的实例变量和静态变量。

（3）不要在Servlet中创建自己的线程。

（4）修改共享对象时，一定要使用同步，尽可能地缩小同步代码的范围，不要直接在service()方法或doXXX()方法上进行同步，以免影响性能。

（5）如果在多个不同的Servlet中，要对外部对象（例如，文件）进行修改操作，一定要加锁，做到互斥的访问。

1. JSP:java服务器页面，一种动态技术标准，模板引擎，本质是签入了Java代码的HTML代码，最终由容器编译成servlet响应用户请求。解决了业务逻辑和视图分离的问题。（servlet可扩展性，维护性，可读性都很差。），其他模板引擎还有velocity，freemaker。

注： JSP只是Servlet模板，JSP可以完成的工作，Servlet都能完成，Serlvet是在Java中嵌入HTML代码，Servlt没有内置对象。JSP侧重于视图， Servlet更侧重于控制逻辑。

注：JSP的渲染机制：由JspServlet负责，将请求的JSP文件包装成Wrapper；创建编译环境类，动态生成java文件，再编译成class文件，加载，调用其service方法渲染。

1. JSP内置对象（9种）：request，response，pageContext(访问本页面的其他对象)，session，application，out，config（服务器配置信息），page（当前页面），exception。
2. JSP基本动作（6种）：include(被请求时引入一个对象)，useBean（寻找并实例化一个Bean），setProperty(设置实例化的Bean属性)，tProperty，forward，plugin。
3. Include指令和include动作区别：都是用于在当前文件中引入另外一个文件，根本差异在于被调用的时间，include指令是编译时指令，也就是在编译时插入到JSP文件中，最终运行时只有一个文件，适合包含静态页面。而include动作是运行时调用，涉及两个文件，类似于方法调用，适合包含动态页面

## Tomcat体系架构

1. 概述:核心组件：Connector 和 Container（以夫妻为例介绍）。多个 Connector 和一个 Container 就形成了一个 Service。Service 只是在 Connector 和 Container 外面多包一层，把它们组装在一起，向外面提供服务，一个 Service 可以设置多个 Connector，但是只能有一个 Container 容器。

注：整个 Tomcat 的生命周期（通过实现Lifecycle实现）由 Server 控制，作用就是要能够提供一个接口让其它程序能够访问到这个 Service 集合、同时要维护它所包含的所有 Service 的生命周期，包括如何初始化、如何结束服务、如何找到别人要访问的 Service。

1. 其他重要组件：安全组件 security、logger 日志组件、session、mbeans、naming 等。这些组件共同为 Connector 和 Container 提供必要的服务。
2. Connector：

负责接收浏览器的发过来的 tcp 连接请求，创建一个 Request 和 Response 对象分别用于和请求端交换数据，然后从线程池中取出一个线程来处理这个请求，并把产生的 Request 和 Response 对象传给处理这个请求的线程。

注：启动过程：初始化SeverSocket， 初始化线程池，构建Request 和 Response，请求到来时，激活线程，解析http协议，把头写到Request 和 Response中，传给Container，完成之后返回Request 和 Response对象，关闭当前Socket，回收线程。

1. Container：

Container 是容器的父接口，所有子容器都必须实现这个接口，Container 容器的设计用的是典型的责任链的设计模式，它有四个子容器组件构成，分别是：Engine、Host、Context、Wrapper，这四个组件不是平行的，而是父子关系，Engine 包含 Host，Host 包含 Context，Context 包含 Wrapper。（子容器的路由放在request中）

Wrapper 代表一个 Servlet，它负责管理一个 Servlet，包括的 Servlet 的装载、初始化、执行以及资源回收。Wrapper 是最底层的容器，Context是 Servlet 运行的基本环境，对应Web工程，定义在父容器 Host 中，Host 不是必须的，但是要运行 war 程序，就必须要 Host，因为 war 中必有 web.xml 文件，这个文件的解析需要 Host ，如果要有多个 Host 就要定义一个 top 容器 Engine 了。而 Engine 没有父容器了，一个 Engine 代表一个完整的 Servlet 引擎。

1. 责任链模式：很多对象由每个对象对其下家的引用连接起来形成，请求在链上传递，这样可以不影响客户端而能够在链上增加任意处理节点。包含抽象处理者和具体处理者，整个容器就是通过一个链连接在一起，这个链一直将请求正确传递给最终处理的Servlet。（其中Pipeline和Value扩展了链的功能，使得可以接收外界干预）。

注：此外，还有命令模式，观察者模式，门面模式等。

## Spring IoC容器

1. 概述：为了解决企业应用程序开发复杂性而创建的，为开发者提供了一站式的轻量级应用开发框架。包含一系列Ioc 容器的设计，提供依赖反转实现，集成了AOP功能，此外，还有MVC、JDBC、事物处理等实现。设计理念：支持POJO（提供一致的编程模型，与具体运行环境隔离）和使用JavaBean的开发方式，使应用面向接口开发，充分支持OO设计方法。

和操作系统类比，操作系统关心对存储、计算、通信、外围设备等物理资源的管理，并在此基础上为用户提供一个统一的接口，而Spring关心的是企业应用资源的使用，如数据持久化、集成、事务处理、消息中间件、分布式计算等高效可靠处理企业数据访问方法的技术抽象。而IoC和AOP为kernel，代表了最为基础的底层抽象，同时也是Spring其他模块实现的基础。

注：POJO：不包含业务逻辑的普通java对象，只用于传输数据。

此外，Spring生态系统还有很多子项目：

Spring Web Flow：构建在SpringMVC上的Web工作流引擎

Spring Security：Spring认证和安全工具，为Spring应用提供安全服务。

Spring Batch:提供构建批处理应用和自动化操作的框架

Spring .NET、Spring Andriod、Spring Mobile、Spring Data等等。

1. IoC叫控制反转，是Inversion of Control的缩写，DI（Dependency Injection）叫依赖注入，是对IoC更简单的诠释。控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反转"就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。DI是对IoC更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。
2. IoC 容器：最主要是完成了完成对象的创建和依赖的管理注入等等。所谓控制反转，就是把原先我们代码里面需要实现的对象创建、依赖的代码，反转给容器来帮忙实现。那么必然的我们需要创建一个容器，同时需要一种描述来让容器知道需要创建的对象与对象的关系。这个描述最具体表现就是我们可配置的文件。

a. BeanFactory作为最顶层的一个接口类，它定义了IOC容器的基本功能规范；只对IOC容器的基本行为作了定义(factoryBean返回的是对象实例。如果一个类继承FactoryBean，用户可以自己定义产生实例对象的方法)

b. Bean对象在Spring实现中是以BeanDefinition来描述的，当spring成功解析定义的一个<bean>节点后，在spring内部就被转化成BeanDefinition对象，以后所有的操作都是对这个对象进行的。

c. IOC初始化 通过XmlBeanFactory或者FileSystemXmlApplicationContext

1. IoC容器初始化：

1、创建IOC配置文件的抽象资源，也就是Resource接口。

2、创建BeanFactory，DefaultListtableBeanFactory是BeanFactory模式实现类。

3、创建一个BeanDefinitionReader对象，该对象为BeanDefinition的读取器。xml文件就使用XMLBeanDefinitionReader。

4、使用Reader来装载配置文件。loadBeanDefinitions就包括了资源文件的解析和注入过程。

通过上面四个步骤我们就可以轻松地使用IOC容器了，在整个过程可以剖析为三个步骤，这三个步骤也是IOC容器的初始化过程：Resource定位、载入、注册

Resource定位（设置资源加载器和资源定位）XmlBeanDefinitionReader

我们一般使用外部资源来描述Bean对象，所以IOC容器第一步就是需要定位Resource外部资源。Resource的定位其实就是BeanDefinition的资源定位，它是由ResourceLoader通过统一的Resource接口来完成的，这个Resource对各种形式的BeanDefinition的使用都提供了统一接口。

载入（AbstractApplicationContext的refresh函数）

第二个过程就是BeanDefinition的载入。BeanDefinitionReader读取、解析Resource定位的资源，也就是将用户定义好的Bean表示成IOC容器的内部数据结构也就是BeanDefinition。在IOC容器内部维护着一个BeanDefinition Map的数据结构，通过这样的数据结构，IOC容器能够对Bean进行更好的管理。

1.通过调用XML解析器将Bean定义资源文件转换得到Document对象（通过io流将bean定义资源转换为document）

2. 完成通用的XML解析之后，然后按照Spring的Bean规则对Document对象进行解析

在配置文件中每一个<bean>都对应着一个BeanDefinition对象。

注册（注册的过程中需要线程同步，以保证数据的一致性，覆盖问题）

第三个过程则是注册，即向IOC容器注册这些BeanDefinition，这个过程是通过BeanDefinitionRegistery接口来实现的（真正完成注册功能的是DefaultListableBeanFactory）。在IOC容器内部其实是将第二个过程解析得到的BeanDefinition注入到一个HashMap（BeanName，BeanDefinition）容器中，IOC容器就是通过这个HashMap来维护这些BeanDefinition的。在这里需要注意的一点是这个过程并没有完成依赖注入，依赖注册是发生在应用第一次调用getBean向容器所要Bean时。当然我们可以通过设置预处理，即对某个Bean设置lazyinit属性，那么这个Bean的依赖注入就会在容器初始化的时候完成。

控制反转是把传统上由程序代码直接操控的对象的调用权交给容器， 通过容器来实现对象组件的装配和管理。 所谓的"控制反转"就是对组件对象控制权的转移，从程序代码本身转移到了外部容器， 由容器来创建对象并管理对象之间的依赖关系。DI是对IoC更准确的描述， 即组件之间的依赖关系由容器在运行期决定，形象的来说， 即由容器动态的将某种依赖关系注入到组件之中。通过IOC容器实现依赖反转，把依赖关系的管理从JAVA对象中解放出来，从而完成对象之间的关系解耦，使原来的对象-对象关系转化为对象-IOC容器-对象关系。（控制反转是关于一个对象如何获取它所应用对象的引用）。

两个主要的容器系列：BeanFactory接口的简单容器系列（只提供容器的最基本功能）和ApplicationContext应用上下文（增加许多面向框架的功能：支持不同的信息源、直接访问资源、国际化、支持应用事件等）

注意：BeanDefinition：用于管理各种对象以及它们之间相互依赖关系的核心数据结构。Resource：用来封装IO操作的类。

初始化过程：通过refresh方法启动，包括BeanDefinition和Resource的定位、载入、注册三个基本过程。

定位：即寻找数据的过程，指对BeanDefinition的资源定位，由ResourceLoader通过统一的Resource接口完成。

载入：把用户定义好的Bean表示成Ioc容器的内部数据结构，即BeanDefinition（POJO对象在IOC容器中的抽象）

注册：将BeanDefinition注入到HashMap中，Ioc容器就是通过这个HahMap来管理数据。

1. 依赖注入过程：一般发生在应用第一次getBean时，除非配置了预实例化属性。在doGetBean方法中完成，先从缓存中取出单例Bean，如果没有，就对当前容器的BeanDefinition是否存在进行检查，根据名称取得BeanDefinition，如果当前容器没有，就顺着双亲往上找，直到BeanFactory。取得Bean之后，再获取其所有依赖Bean，此处触发递归的调用getBean方法。完成之后创建Bean示例，进行类型检查，返回Bean。

class A{  
 B b;  
}

A依赖于B，B是被依赖者

1. (1).用户第一次通过getBean方法向IoC容索要Bean时，IoC容器触发依赖注入。

(2).当用户在Bean定义资源中为<Bean>元素配置了lazy-init属性，即让容器在解析注册Bean定义时进行预实例化，触发依赖注入。

\*\*\*\*\*\*\*

如果Bean定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果Bean定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。

创建bean流程：

1.生成Bean所包含的java对象实例。

2.对Bean属性的依赖注入进行处理。

2.1对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

2.2对于非集合类型的属性，大量使用了JDK的反射和内省机制，通过属性的getter方法(reader method)获取指定属性注入以前的值，同时调用属性的setter方法(writer method)为属性设置注入后的值。看到这里相信很多人都明白了Spring的setter注入原理。

a. 尝试从缓存中加载单例，如果失败再创建单态模式Bean的实例对象（从singletonFactories中加载）

b. bean的实例化

如果从缓存中得到了bean的原始状态，则需要对bean进行实例化。

setter循环依赖(对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入）的Bean来完成的，而且只能解决单例作用域的Bean循环依赖)具体处理步骤如下：

（1） Spring容器创建单例“circularityA” Bean，首先根据无参构造器创建“circularityA” Bean， 并暴露一个exposedObject用于返回提前暴露的Bean，并将“circularityA”Bean放到Catch中；然后进行setter注入“circularityB”;

（2） Spring容器创建单例“circularityB" Bean，首先根据无参构造器创建“circularityB" Bean，并暴露一个exposedObject用于返回提前暴露的Bean，并将“circularityB” Bean放到Catch中，然后进行setter注入“circularityC”;

（3） Spring容器创建单例“circularityC” Bean，首先根据无参构造器创建“circularityC” Bean，并暴露一个exposedObject用于返回暴露的Bean，并将“circularityC” Bean放入Catch中， 然后进行setter注入“circularityA”；进行注入“circularityA”时由于步骤1提前暴露了exposedObject所以从之前的catch里面拿Bean不用重复创建。

（4） 最后在依赖注入“circularityB”和“circularityA”也是从catch里面拿提前暴露的bean， 完成setter注入。

1. Bean生命周期（singleton）：1. Bean实例的创建，2.为Bean注入依赖关系，3.调用Bean的初始化方法，4.使用Bean，5.关闭容器是销毁Bean。
2. 后处理器（Bean后处理器 、容器后处理器）：实质是一个监听器，可以监听容器触发的事件，注册后，bean就具备了接收IOC容器回调的能力（在Bean的初始化前或者后提供回调入口）

## Spring AOP

1. 概述:面向切面的编程，切面用于描述分散在对象、类或者函数中的横向关注点，通过分离这些关注点，把解决特定领域问题的代码从业务逻辑中独立出来，他们之间的关系通过切面来维护。效果是保证程序员在不修改源代码的情况下，为系统中业务组件的多个业务方法添加某种通用功能。

将切面所定义的横切逻辑织入到切面所指定的连接点中

1. 术语：

AOP术语

1）连接点（Joinpoint）

程序执行的某个特定位置：如类开始初始化前、类初始化后、类某个方法调用前、调用后、方法抛出异常后。一个类或一段程序代码拥有一些具有边界性质的特定点，这些点中的特定点就称为“连接点”。Spring仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时以及方法调用前后这些程序执行点织入增强。连接点由两个信息确定：第一是用方法表示的程序执行点；第二是用相对点表示的方位。

2）切点（Pointcut）

每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点，即连接点是程序类中客观存在的事物。AOP通过“切点”定位特定的连接点。连接点相当于数据库中的记录，而切点相当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。在Spring中，切点通过org.springframework.aop.Pointcut接口进行描述，它使用类和方法作为连接点的查询条件，Spring AOP的规则解析引擎负责切点所设定的查询条件，找到对应的连接点。其实确切地说，不能称之为查询连接点，因为连接点是方法执行前、执行后等包括方位信息的具体程序执行点，而切点只定位到某个方法上，所以如果希望定位到具体连接点上，还需要提供方位信息。

3）增强（Advice）

增强是织入到目标类连接点上的一段程序代码，在Spring中，增强除用于描述一段程序代码外，还拥有另一个和连接点相关的信息，这便是执行点的方位。结合执行点方位信息和切点信息，我们就可以找到特定的连接点。

4）目标对象（Target）

增强逻辑的织入目标类。如果没有AOP，目标业务类需要自己实现所有逻辑，而在AOP的帮助下，目标业务类只实现那些非横切逻辑的程序逻辑，而性能监视和事务管理等这些横切逻辑则可以使用AOP动态织入到特定的连接点上。

5）引介（Introduction）

引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过AOP的引介功能，我们可以动态地为该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。

6）织入（Weaving）

织入是将增强添加对目标类具体连接点上的过程。AOP像一台织布机，将目标类、增强或引介通过AOP这台织布机天衣无缝地编织到一起。根据不同的实现技术，AOP有三种织入的方式：

a、编译期织入，这要求使用特殊的Java编译器。

b、类装载期织入，这要求使用特殊的类装载器。

c、动态代理织入，在运行期为目标类添加增强生成子类的方式。

Spring采用动态代理织入，而AspectJ采用编译期织入和类装载期织入。

7）代理（Proxy）

一个类被AOP织入增强后，就产出了一个结果类，它是融合了原类和增强逻辑的代理类。根据不同的代理方式，代理类既可能是和原类具有相同接口的类，也可能就是原类的子类，所以我们可以采用调用原类相同的方式调用代理类。

8）切面（Aspect）

切面由切点和增强（引介）组成，它既包括了横切逻辑的定义，也包括了连接点的定义，Spring AOP就是负责实施切面的框架，它将切面所定义的横切逻辑织入到切面所指定的连接点中。

1. 组成：基础：待增加对象或者说目标对象；切面：对于基础的增加应用；配置（编织）：提供配置环境，把基础和切面结合起来，完成对目标对象的编织实现。
2. AOP实现：

1.静态实现，在编译阶段对程序进行修改，生成代理类，AspectJ

2.动态实现，运行阶段动态生成，Spring AOP（JDK动态代理，与目标类实现相同的接口，cglib生成目标类的子类，策略模式）。

## Spring缓存

可以在任何层次上进行，内置的实现是SimpleCacheManager，内存缓冲区，底层使用ConcurrentHashMap实现。只能用于简单测试环境（EhCache，OSCache）

## Spring 事务

1. Spring事物管理常用的3个接口：

事物管理器PlatformTransactionManager：策略模式，与具体事务策略分离，Spring为不同的持久层框架提供不同的实现。（JDBC访问和Mybatis：DataSourceTransactionManager ）

注：transactionDefinition接口除了定义了隔离级别和传播行为常量，TransactionStatus接口定义了获得和判断事物状态的方法，spring事务默认是数据库的级别。

1. 事务传播行为：解决业务层方法之间的相互调用的问题（7种分3类）

1.PROPAGATION\_REQUIRED 支持当前事务，如果不存在就新建一个

2.PROPAGATION\_SUPPORTS 支持当前事务，如果不存在，就不使用事务

3.PROPAGATION\_MANDATORY 支持当前食物，如果不存在，抛出异常

1.PROPAGATION\_REQUIRES\_NEW 如果有事务存在，挂起当前事务，创建一个新事物

2.PROPAGATION\_NOT\_SUPPORTED 以非事务方式运行，如果有事物，挂起当前事物

3.PROPAGATION\_NEVER 以非事物方式运行，如果有事务存在，抛出异常

1.PROPAGATION\_NESTED 如果当前事务存在，则嵌套事务执行

注意：Spring支持两种事务管理方式（编程式、XML配置声明式），开发中推荐声明式（代码侵入性最小，通过AOP实现）

## Spring MVC

1. MVC概念：mvc的核心思想是业务数据的抽取和业务数据的呈现相分离

Controller：控制层，调用业务逻辑产生合适的数据（Model）；传递数据给视图层用于呈现

View：视图层，为用户提供UI，重点关注的数据的呈现

Model：模型层，业务数据的信息表示。

注：调用initStrategies完成初始化：定义URL映射关系，实现业务逻辑的Handler实例对象，渲染模板资源。

MVC优势：低耦合性（视图和业务分离），高重用性，可维护性，部署快速等。

1. 请求流程：

1、用户发送请求至前端控制器DispatcherServlet

2、DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、DispatcherServlet通过HandlerAdapter（让Handler实现更加灵活）处理器适配器调用处理器

5、执行处理器(Controller，也叫后端控制器)。

6、Controller执行完成返回ModelAndView（连接业务逻辑层和展示层的桥梁，持有一个ModelMap对象和一个View对象）。

7、HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

8、DispatcherServlet将ModelAndView传给ViewReslover视图解析器

9、ViewReslover解析后返回具体View

10、DispatcherServlet对View进行渲染视图（将ModelMap模型数据填充至视图中）。

11、DispatcherServlet响应用户

① 客户端的所有请求都交给前端控制器DispatcherServlet来处理，它会负责调用系统的其他模块来真正处理用户的请求。

② DispatcherServlet收到请求后，将根据请求的信息（包括URL、HTTP协议方法、请求头、请求参数、Cookie等）以及

HandlerMapping的配置找到处理该请求的Handler（任何一个对象都可以作为请求的Handler）。

③在这个地方Spring会通过HandlerAdapter对该处理进行封装。

④ HandlerAdapter是一个适配器，它用统一的接口对各种Handler中的方法进行调用。

⑤ Handler完成对用户请求的处理后，会返回一个ModelAndView对象给DispatcherServlet，ModelAndView顾名思

义，包含了数据模型以及相应的视图的信息。

⑥ ModelAndView的视图是逻辑视图，DispatcherServlet还要借助ViewResolver完成从逻辑视图到真实视图对象的解析

工作。

⑦ 当得到真正的视图对象后，DispatcherServlet会利用视图对象对模型数据进行渲染。

⑧ 客户端得到响应，可能是一个普通的HTML页面，也可以是XML或JSON字符串，还可以是一张图片或者一个PDF文件。

## Spring MVC拦截器

1. 概述：拦截器是一段代码，这段代码和过滤器如出一辙，能够拦截请求和响应的信息，然后能做一些公共的事情，比如：验证用户是否登录、权限控制、解决乱码问题、防止特殊字符的安全注入等。关键在于，特能够拦截请求和响应，所以，我们才能在请求和响应上做一些手脚。
2. 主要方法：

preHandle()方法是否将当前请求拦截下来。（返回true请求继续运行，返回false请求终止。）

postHandle()方法的ModelAndView arg可以改变发往的视图或修改发往视图的信息。

afterCompletion()方法表示视图显示之后在执行该方法。（一般用于资源的销毁）

注：多个拉截器的执行顺序，按照配置文件的写入顺序，实现springmvc拦截器不只可以实现HandleInterceptor接口，还可以实现WebRequestInterceptor接口，实现的方法大同小异，但是WebRequestInterceptor提供的preHandle方法没有返回值，故不提供终止这次请求的功能

1. 拦截器和过滤器Filter的区别

①拦截器是基于java的反射机制的，而过滤器是基于函数回调。

②拦截器不依赖servlet容器，过滤器依赖与servlet容器。

③拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用。

⑥拦截器可以获取IOC容器中的各个bean，而过滤器就不行，这点很重要，在拦截器里注入一个service，可以调用业务逻辑。

注意：@ModelAttribute:用于修饰方法参数（将输入参数对象添加到Model对象中）或非请求方法（在请求方法前被调用，如果返回对象，则被放到Model中）。

1. EL表达式：在MVC体系结构中，JSP页面只是用来显示数据，但JSP脚本中的表达式功能不够强大，它不能直接对隐式对象中某对象的属性进行显示， EL表达式将大大简化JSP页面中的Java代码，语法：${表达式}
2. JSTL: 自定义标签很容易造成重复定义和非标准的实现， JSP标准标签库（JSTL）是一个JSP标签集合，它封装了JSP应用的通用核心功能。JSTL支持通用的、结构化的任务，比如迭代，条件判断，XML文档操作，国际化标签，SQL标签。

## MyBatis

1. ORM模型就是数据库的表和POJO的映射关系模型，主要解决数据库数据和POJO对象的相互映射。
2. 概述：通过 SQL Map 将 Java 对象映射成 SQL 语句和将结果集再转化成 Java 对象，与其他 ORM 框架相比，既解决了 Java 对象与输入参数和结果集的映射，又能够让用户方便的手写使用 SQL 语句。

主要完成两件事情：根据 JDBC 规范建立与数据库的连接；通过反射打通 Java 对象与数据库参数交互之间相互转化关系。

1. 对SQL 的解析：1.从SQL中获取初始化生成的parameterMap；2.根据传进来的对象调用getter方法获取属性值，组成参数值数组（对应SQL中的参数顺序）；3.把参数值数组中的参数设置到prepareStatement中（调用setXXX）。
2. 数据库映射到Java对象：1.获取当前resultMap，2.获取返回对象的所有可写属性数组；3.根据返回结果集ResultSet中的列名去匹配属性数组，将匹配结构构成一个resultMappingList集合；4.根据此集合从ResultSet中取出列对应值，构成数组；5.调用对象的setter方法把值设置到对象中，并返回目标对象（如果有多条记录则继续映射目标对象）
3. Mybatis优势

1.MyBatis可以进行更为细致的SQL优化，可以减少查询字段。

2.MyBatis简单易学，本身就很小且简单。没有任何第三方依赖。

3.解除sql与程序代码的耦合，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。

1. Hibernate优势(将表信息映射到XML文件中，再从XML文件映射到持久化类中)

1.Hibernate数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。

2.Hibernate有更好的二级缓存机制，可以使用第三方缓存。MyBatis本身提供的缓存机制不佳。

1. JDBC流程
2. 加载JDBC驱动，建立并获取数据库连接
3. 操作Connection，创建 JDBC Statements对象
4. 通过statement执行SQL，返回结果到ResultSet对象
5. 使用ResultSet读取数据，然后通过代码转化为具体的POJO对象
6. 释放相关资源（关闭Connection，关闭Statement，关闭ResultSet）
7. 缓存机制

mybatis一级缓存是指在内存中开辟一块区域，用来保存用户对数据库的操作信息（sql）和数据库返回的数据，如果下一次用户再执行相同的请求，那么直接从内存中读数数据而不是从数据库读取。

一级缓存是SqlSession级别的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个(内存区域)数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

一级缓存的作用域是同一个SqlSession，在同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。Mybatis默认开启一级缓存。

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession去操作数据库得到数据会存在二级缓存区域，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

二级缓存是多个SqlSession共享的，其作用域是mapper的同一个namespace，不同的sqlSession两次执行相同namespace下的sql语句且向sql中传递参数也相同即最终执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis默认没有开启二级缓存需要在setting全局参数中配置开启二级缓存。

如果sqlSession去执行commit操作（执行插入、更新、删除），清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。

如果SqlSession3去执行相同 mapper下sql，执行commit提交，清空该 mapper下的二级缓存区域的数据。

如果缓存中有数据就不用从数据库中获取，大大提高系统性能。

1. 插件原理

1） 所有可能被拦截的处理类都会生成一个代理

2) 处理类代理在执行对应方法时，判断要不要执行插件中的拦截方法

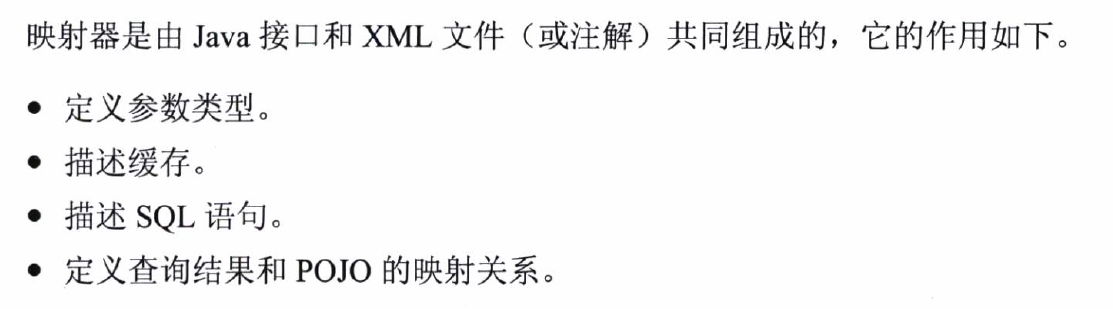
3) 执行插接中的拦截方法后，推进目标的执行

如果有N个插件，就有N个代理，每个代理都要执行上面的逻辑。这里面的层层代理要多次生成动态代理，是比较影响性能的。虽然能指定插件拦截的位置，但这个是在执行方法时动态判断，初始化的时候就是简单的把插件包装到了所有可以拦截的地方。

因为包括sql等其他属性在内的多个属性都没有对应的方法可以直接取到，它们对外部都是封闭的，是对象的私有属性，所以这里就需要引入反射机制来获取或者更改对象的私有属性的值了。

1. Mybatis原理

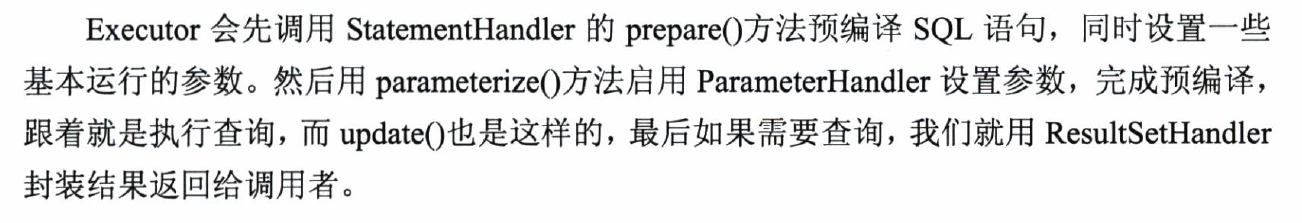
MyBatis应用程序根据XML配置文件创建SqlSessionFactory，SqlSessionFactory在根据配置，配置来源于两个地方，一处是配置文件，一处是Java代码的注解，获取一个SqlSession。SqlSession包含了执行sql所需要的所有方法，可以通过SqlSession实例直接运行映射的sql语句，完成对数据的增删改查和事务提交等，用完之后关闭SqlSession。

配置SqlSessionFactory中最重要的是配置configuration，mybatis中的配置信息全部来源于此。

Boundsql是建立sql和参数的地方。

Mapper映射通过动态代理实现。映射器的xml文件的命名空间对应的是mapper接口的全路径，它根据全路径和方法名便能够绑定起来。然后通过动态代理，使接口跑起来。即调用mapperproxy中的mapperdMethod对的execute方法，将参数传入sqlsession去执行。

MapperMethod



1. PreparedStatement和$ #

每一种数据库都会尽最大努力对预编译语句提供最大的性能优化。因为预编译语句有可能被重复调用。所以语句在被DB的编译器编译后的执行代码被缓存下来,那么下次调用时只要是相同的预编译语句就不需要编译,只要将参数直接传入编译过的语句执行代码中(相当于一个涵数)就会得到执行。只要预编译的语句语法和缓存中匹配。那么在任何时候就可以不需要再次编译而可以直接执行。

PreparedStatement在conn.prepareStatement(sql)时就把sql语句传给它，这样它会在数据库端进行预编译（包含占位符），下次execute或者executeQuery时只要是相同的预编译语句就不需要编译,只要将参数直接传入编译过的语句执行代码中(相当于一个函数)就会得到执行。

#{ } 解析为一个 JDBC 预编译语句（prepared statement）的参数标记符。

${ } 仅仅为一个纯碎的 string 替换，在动态 SQL 解析阶段将会进行变量替换

${ } 的变量的替换阶段是在动态 SQL 解析阶段，而 #{ }的变量的替换是在 DBMS 中。即占位符在预编译后进行替换

## Redis

Reids的特点

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。

Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种数据结构，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。

Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

Redis支持的数据类型

Redis通过Key-Value的单值不同类型来区分, 以下是支持的类型:

Strings

Lists

Sets 求交集、并集

Sorted Set

Hashes

为什么redis需要把所有数据放到内存中？

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。

如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

Redis是单进程单线程的

redis利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销

## Maven

1. 概述：用于项目构建的工具，采用“约定优于配置”的策略，通过它便捷的管理项目的生命周期。即项目的jar包依赖，开发，编译，测试，发布，打包，生成站点等。

Pom。xml:项目对象模型描述文件，描述项目的类别，名称，版本，所需的jar包和插件等。

1. 坐标：为项目指定的一个唯一标识符，Maven世界中唯一标识一个构件的标识，一般由groupId/artifactId/packaging/version三部分组成。
2. 仓库：保存Maven插件，第三方框架，jar包等（搜索顺序：本地->远程->中央）
3. Maven生命周期：指构建项目包含多个有序的阶段，包含三个独立的生命周期（每一个生命周期有分为若干个阶段，其中defaul是最核心的一个）：

clean：清理项目（pre-clean、 clean、post-clean）

default：构建项目（ compile、 test、 package、 install）

site：生成项目站点pre-site 、site、 post-site、 site-deploy）

1. 依赖冲突：是指间接依赖关系中依赖同一个依赖，或者同一个依赖的不同版本的情况，此时我们就需要判断到底依赖那一个依赖（最短路径优先、先声明先优先）

## SVN

1. 一个自由/开源的集中式版本控制系统，一组文件存放在中心版本库，记录每一次文件和目录的修改，允许把数据恢复到早期版本，或是检查数据修改的历史，可以通过网络访问它的版本库，从而使用户在不同的电脑上进行操作。
2. 和Git区别：SVN 是集中式的，需要联网，使用界面统一，环境友好，适合团队各个角色，更加安全，学习成本低等。

## Jekins

1. 一个开源的持续集成服务器，助我们自动构建各类项目。通过插件式集成很多软件，帮助我们持续集成我们的工程项目。
2. 持续集成是持续地编译、测试、检查和部署源代码的过程。

注：各种项目的"自动化"编译、打包、分发部署。jenkins可以很好的支持各种语言（比如：java， c#， php等）的项目构建，也完全兼容ant、maven、gradle等多种第三方构建工具，同时跟svn、git能无缝集成，也支持直接与知名源代码托管网站，比如github、bitbucket直接集成。

## 前端（Html，CSS，JS，Ajax）

1. Html5:

概述：HTML最新的修订版本，2014年10月由万维网联盟（W3C）完成标准制定。设计目的是为了在移动设备上支持多媒体。简单易学。

新特性：

用于绘画的 canvas 元素

用于媒介回放的 video 和 audio 元素

对本地离线存储的更好的支持

新的特殊内容元素，比如 article、footer、header、nav、section

新的表单控件，比如 calendar、date、time、email、url、search

1. CSS：层叠样式表是一种用来表现[HTML](http://baike.baidu.com/view/692.htm)文件样式的计算机语言。

CSS目前最新版本为[CSS3](http://baike.baidu.com/view/1713027.htm)，是能够真正做到[网页](http://baike.baidu.com/view/828.htm)表现与内容分离的一种样式设计语言。相对于传统HTML的表现而言，CSS能够对网页中的对象的位置排版进行像素级的精确控制，支持几乎所有的字体字号样式，拥有对网页[对象](http://baike.baidu.com/subview/2387/10666278.htm)和模型样式编辑的能力，并能够进行初步[交互设计](http://baike.baidu.com/view/426920.htm)。CSS能够根据不同使用者的理解能力，简化或者优化写法，针对各类人群，有较强的易读性。

1. JavaScript一种[直译](http://baike.baidu.com/view/295412.htm)式[脚本语言](http://baike.baidu.com/view/76320.htm)，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的[解释器](http://baike.baidu.com/view/592974.htm)被称为JavaScript引擎，为[浏览器](http://baike.baidu.com/view/7718.htm)的一部分，广泛用于[客户端](http://baike.baidu.com/view/930.htm)的脚本语言，最早是在[HTML](http://baike.baidu.com/view/692.htm)网页上使用，用来给[HTML](http://baike.baidu.com/view/692.htm)网页增加动态功能。

Javascript闭包：[Java](http://www.2cto.com/kf/ware/Java/)script允许使用内部函数---即函数定义和函数表达式位于另一个函数的函数体内。而且，这些内部函数可以访问它们所在的外部函数中声明的所有局部变量、参数和声明的其他内部函数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。

1. jQuery是一个兼容多浏览器的[javascript](http://baike.baidu.com/view/16168.htm)库，免费、开源。jQuery的语法设计可以使开发更加便捷，例如操作[文档](http://baike.baidu.com/view/55621.htm)对象、选择[DOM](http://baike.baidu.com/view/14806.htm)元素、制作动画效果、事件处理、使用[Ajax](http://baike.baidu.com/view/1641.htm)以及其他功能。
2. Ajax原理：

一种用于创建快速动态网页的技术。通过在后台与服务器进行少量数据交换，Ajax可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。通过XmlHttpRequest对象来向服务器发异步请求，从服务器获得数据，然后用javascript来操作DOM而更新页面。

注：Ajax是客户端技术，核心是XmlHttpRequest，该对象支持异步请求技术。

Ajax的优点：

1。最大的一点是页面无刷新，给用户的体验非常好。

2。使用异步方式与服务器通信，不需要打断用户的操作，具有更加迅速的响应能力。

3。ajax的原则是“按需取数据”，可以最大程度的减少冗余请求，减少服务器的负荷。

## Junit4

1. 概述：xUnit是一套基于测试驱动开发的测试框架，JUnit是xUnit的一套子集，只要加上@Test注解即可，不需要继承任何类，命名没有限制，基于注解和断言的自动化测试框架。状态条（绿色——测试成功；红色——测试失败）。

注意：1。测试方法上必须使用@Test进行修饰，2。测试方法必须使用public void 进行修饰，不能带任何的参数

错误类别：1。Failure一般由单元测试使用的断言方法判断失败所引起的，这将表示测试点发现了问题，就是说程序输出的结果和我们预期的不一样。2。error是由代码异常引起的，它可以产生于测试代码本身的错误，也可以是被测试代码中的一个隐藏的bug。测试用例不是用来证明你是对的，而是用来证明你没有错。

# 大型网站与Java中间件

## 大型网站架构演进过程

1。初始阶段：只有一台服务器（应用程序、数据库、文件等）

2。 数据服务与应用服务分离：应用服务器、文件服务器、数据库服务器

3。缓存：应用服务器缓存和分布式缓存

4。应用服务器集群：需要使用[负载均衡](http://www.codeceo.com/article/balanced-algorithm.html)，考虑Session 的管理问题

5。数据库读写分离：解决读压力大的问题（通过数据库主从热备功能）

6。使用反向代理和CDN:原理是缓存，减轻后端服务器负载压力和加快访问速度。

7。使用分布式文件系统和分布式数据库系统

8。使用NoSQL和搜索引擎

9。业务拆分：将网站拆分成许多不同的应用，每个应用独立部署（通过超链接或者消息队列进行数据分发）

10。分布式服务：提取相同的业务操作，独立部署成服务中心，应用系统只需要管理用户界面，通过远程调用完成具体业务操作，而数据库交互也放到服务中心。

注：网站核心要素：性能（响应时间、TPS），可用性（冗余），伸缩性（指不断向集群中加入服务器的手段来缓解不断上身的用户并发访问压力和不断增长的数据存储需求），扩展性（增加新的业务产品，是否对现有产品透明无影响，主要手段是事件驱动架构和分布式服务），安全性。

## CDN工作机制

一种网络缓存技术，内容分发网络，将用户需要的内容分发到离用户近的地方，部署在网络运营商机房，节省广域网带宽和提升用户访问速度，一般存放静态资源。

注：CDN动态加速是指DNS解析过程中通过动态的链路探测来寻找回源最好的路径。

## DNS域名解析过程

1。浏览器缓存；

2。操作系统缓存（hosts文件）；

3。本地DNS服务器；

4。如果没有找到，请求Root Server域名服务器；

5。 Root Server域名服务器返回主域名服务器；

6。请求主域名服务器；

7。主域名服务器查找此域名对应的域名服务器（注册域名服务器），并返回；8。 注册域名服务器查询域名和IP映射关系，返回IP地址给本地域名服务器；9。本地域名服务器做缓存，同时把结果返回给用户。

注：传输层使用TCP或者UDP协议，53端口。

## 高性能优化

1. 测试指标：响应时间，并发数，吞吐量（以高速公路举例）
2. Web前端性能优化：

浏览器优化（a。减少http请求，比如合并CSS，JS，图片等资源；b。使用浏览器缓存，通过设置HTTP头部的信息缓存静态资源；c。压缩；d。CSS上面，JS下面；e。减少Cookie传输）

CDN加速

反向代理（安全，缓存，负载均衡）

注：传统代理位于浏览器一侧，而反向代理位于网站机房一侧。二者区别：

从用途 上来讲：

正向代理的典型用途是为在防火墙内的局域网客户端提供访问Internet的途径。反向代理的典型用途是将防火墙后面的服务器提供给Internet用户访问。反向代理还可以为后端的多台服务器提供负载平衡，或为后端较慢的服务器提供缓冲服务。

从安全性 来讲：

正向代理允许客户端通过它访问任意网站并且隐藏客户端自身

反向代理对外都是透明的，访问者并不知道自己访问的是一个代理。

1. 分布式缓存：
2. 使用消息队列将直接调用异步化（任何可以晚点做的事都应该晚一点做）
3. 使用服务器集群
4. 代码优化：多线程，无状态对象，局部对象，资源服用（单例，对象池），垃圾回收
5. 存储性能优化：

RAID（0、1、10、3、5、6）

## 高可用架构（应用层，服务层，数据层）

主要两种：不同功能进行物理分离实现伸缩：纵向分离（业务流程上的不同部分分离部署）和横向分离（将不同的业务模块分离部署）和单一功能通过集群实现伸缩。

1. 分层：

应用层：无状态性（不保存业务的上下文信息，完全对等），通过负载均衡进行失效转移，Session管理。

服务层：为业务产品提供基础公共服务，无状态性，主要手段有：分级管理（核心应用和服务使用更好的硬件），设置应用调用超时时间，异步调用，服务降级（拒绝低优先级应用调用，随机拒绝部分请求调用，关闭不重要功能），幂等性设计。

数据层（持久性，可访问性，一致性）

1. CAP原理：一个提供数据服务的存储系统无法同时满足数据一致性，数据可用性，分区耐受性（跨网络分区线性伸缩），一般做法是放弃一致性，只要保证最终一致即可。

注：发布控制：预验证发布，代码控制（分支开发，主干发布），灰度发布（将集群服务器分为若干部分，每天发布一部分）

## 安全架构

1. XSS攻击：跨站点脚本攻击，指通过篡改网页，注入恶意HTML代码，在用户浏览网页时，控制用户浏览器进行恶意操作的一种攻击方式（反射性，持久性）。
2. 注入攻击：SQL注入和OS注入。
3. 信息加密技术：单向散列加密（通过对不同输入长度的信息进行散列计算，得到固定长度的输出md5，sha），对称加密，非对称加密（信息传输和数字签名）

## 负载均衡（5种）

1. HTTP重定向：重定向服务器根据用户HTTP请求计算真实的Web服务器地址，写入重定向响应中，返回给浏览器（简单，两次请求，性能差，重定向服务器瓶颈）。
2. DNS域名解析：DNS服务器中配置多个记录，根据算法返回不同的IP地址（将工作转交给DNS，省去管理维护麻烦，支持基于位置解析，但是无控制权，无法做更多改善和管理）。
3. 反向代理：管理一组Web服务器，根据算法转发到不同的Web服务器，返回也需要通过反向代理，Web服务器不直接对外，反向代理需要双网卡。（部署简单，第7层解析，开销大，代理服务器瓶颈）。
4. IP：在操作系统内核进程中获取网络数据包，根据算法计算得到真实Web服务器地址，修改目标地址和源地址（SNAT），不需要用户进程处理，响应同样要经过负载均衡服务器（比反向代理好，代理服务器瓶颈）。
5. 数据链路层：三角传输模式，Web服务器地址和负载均衡服务器地址IP一致，只修改MAC地址，响应不经过负载均衡服务器，使用最广。（LVS）

注：负载均衡算法有轮询、加权轮询（根据服务器性能）、随机、加权随机、最小连接、源地址散列等。

## 分布式缓存集群（memcached）

1. 概述：指缓存部署在多个服务器组成的集群中，以集群方式提供缓存服务。Memcached采用集中式的缓存集群管理，缓存与应用分离部署，应用程序通过一致性Hash算法选择缓存服务器访问缓存数据，缓存服务器之间不通信。易于扩容，可伸缩性很好。
2. 注：Memcached特点：高性能的分布式内存对象缓存系统，简单的通信协议（基于文本的自定义协议），丰富的客户端程序，高性能的网络通信（基于Libevent，支持事件触发的网络通讯程序库），使用固定空间分配（slab->chunk，LRU），互不通信的服务器集群架构。
3. 分布式缓存集群的伸缩性设计：不同于应用服务器，应用程序通过客户端访问Memcached服务器集群，客户端主要有API，路由模块，Memcached服务器集群列表及通信模块构成。
4. 一致性Hash算法：先构造一个长度为2的32次方的整数环，根据节点名称的Hash值将缓存服务器节点放置在环上，顺时针查到即可通常用二次查找树实现（最右边和最左边叶子相连接），查找不小于查找树的最小数值。（通过虚拟层解决负载压力不一致问题）

## 分布式系统

1. 概述：组件分布在网络计算机上，组件之间仅仅通过消息传递来通信并协调行动。意义：升级单机处理能力性价比越来越低，单机存在瓶颈，稳定性和可用性考虑例如：通过负载均衡，名称服务器，规则服务器
2. 难点:缺乏全局时钟(交给单独的集群来完成)，故障独立性(出现一部分节点/模块有问题，另一部分正常运)， 单点故障(某个功能只有在某台单机在支撑，单点备份)，分布式事务等。

## Java中间件

1. 为软件应用提供操作系统所提供的服务之外的服务。让开发者能够方便的处理通信，输入和输出，能够专注于应用本身。既不是最上层应用，也不是最底层的支撑系统，处于中间位置，是应用和应用、服务之间的桥梁。主要有三类：
2. 消息中间件：解决应用之间的消息传递、解耦、异步等问题；

消息生产者（消息源）通过远程访问接口将消息推送给消息队列服务器，消息队列服务器将消息写入本地内存队列后立即返回成功响应给消息生产者，然后根据消息订阅列表查找订阅该消息的消费者，按照先进先出的顺序将消息通过远程通信接口发送给消费者（订阅-发布模式）。产品如：JMS规范和ActiveMQ实现。

淘宝的Notify是一个高性能、高可靠、可扩展组件，轻量级，支持最终一致性和订阅者集群的消息中间件。所谓订阅者集群，即将订阅消息的客户端分为 多个集群，集群之间采用Topic方式，让每个集群都能收到消息，集群之中再按照Queue的方式，仅由一个客户端来处理消息。

1. 远程过程调用和对象访问中间件：解决分布式环境下应用的互相访问的问题；

客户端：接口调用-寻址路由-编码-通信；服务端：通信-解码-示例定位-服务调用-返回结果。(淘宝的HSF，dubbo)

分布式服务:通过接口解耦。WebService，产品如Dubbo (消费者通过接口使用服务，通过注册中心，服务接口通过代理加载具体服务，支持多种通信协议和数据序列化协议，使用NIO通信框架)。

1. 数据访问中间件：解决应用访问数据库的共性问题。

设计：SQL解析->规则处理->SQL改写->数据源选择->SQL执行->返回结果集合并处理。

1. 关系数据库集群伸缩性设计：分库分表（Cobar）和TDDL

Cobar是一个分布式数据库访问代理，应用程序通过JDBC驱动访问Cobar集群，集群根据SQL的分库规则分解SQL，分发到MySql集群不同的数据库实例上执行。

淘宝[分布式数据层](http://www.searchsoa.com.cn/showcontent_33235.htm" \t "_blank)TDDL：支持分库分表以及读写分离，进行了一定的封装。对上层应用而言仍旧操作JDBC，实则是在使用淘宝分布式数据层（TDDL），它能实现SQL解析、规则路由、数据合并；既可以用jar的方式在客户端直接连接数据库，也可以让客户端通过DBProxy服务器访问数据库；具备三层的数据源结构，还支持非对称数据复制。

1. 数据拆分：

1。垂直拆分：不同业务单元数据分到不同的数据库中

2。水平拆分：同一业务单元的数据分到不同的数据库中

带来问题：ACID保证被打破（引入分布式事务）；Join操作困难（分解为多次数据库操作或者数据冗余）；外键约束受到影响；单库自增序列生成唯一ID被打破等（把所有ID集中在一个地方管理，对每个ID序列独立管理，每台机器使用ID时都从这个ID生成器上取）。

## 分布式事务

1. 指事务参与者，支持事务的服务器，资源服务器以及事务管理器分别位于分布式系统的不太节点上，在分布式事务中，会有多个节点参与。
2. 两阶段提交

两阶段提交的过程涉及到协调者和参与者。协调者可以看做成事务的发起者，同时也是事务的一个参与者。对于一个分布式事务来说，一个事务是涉及到多个参与者的。具体的两阶段提交的过程如下：   
第一阶段：   
　　首先，协调者在自身节点的日志中写入一条的日志记录，然后所有参与者发送消息prepare T，询问这些参与者（包括自身），是否能够提交这个事务；   
　　参与者在接受到这个prepare T 消息以后，会根据自身的情况，进行事务的预处理，如果参与者能够提交该事务，则会将日志写入磁盘，并返回给协调者一个ready T信息，同时自身进入可提交状态；如果不能提交该事务，则记录日志，并返回一个not commit T信息给协调者，同时撤销在自身上所做的数据库改；   
第二阶段：   
　　协调者会收集所有参与者的意见。（1）如果收到参与者发来的not commit T信息，则标识着该事务不能提交，协调者会将Abort T 记录到日志中，并向所有参与者发送一个Abort T 信息，让所有参与者撤销在自身上所有的预操作；（2）如果协调者收到所有参与者发来prepare T信息，那么协调者会将Commit T日志写入磁盘，并向所有参与者发送一个Commit T信息，提交该事务。（3）若协调者迟迟未收到某个参与者发来的信息，则认为该参与者发送了一个VOTE\_ABORT信息，从而取消该事务的执行。   
　　参与者接收到协调者发来的Abort T信息以后，参与者会终止提交，并将Abort T 记录到日志中；如果参与者收到的是Commit T信息，则会将事务进行提交，并写入记录。

1. 三阶段提交

面对协调者宕机，参与者也宕机的问题， ，三阶段提交的方法将二阶段提交的第二阶段再分为待定阶段（或预提交阶段）和确定阶段，从而变为三阶段；

注意：另一种知名分布式事务算法pasox（前提是不存在拜占庭将军问题）能解决多个协调者的情况，核心思想是少数服从多数，并提出了多数派的概念。

1. 悲观锁和乐观锁区别：

悲观锁：假定会发生并发冲突，每次在拿数据的时候都会上锁，屏蔽一切可能违反数据完整性的操作。

乐观锁：假设不会发生并发冲突，只在提交操作时使用版本号等机制检查是否违反数据完整性。不能解决脏读的问题。

注：乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

## 分布式文件系统GFS

1. 主要解决了单机文件存储容量和安全性的问题，把多台廉价PC组成一个大的分布式文件系统集群，并对外提供服务。主要分为三个部分：
2. Client：使用GFS的入口，从Master上获取要操作的文件在chunkServer中的地址，然后直接和chunkServer通信（数据的读写）。
3. Master：系统大脑，维护所有文件系统元数据（名字空间、访问控制信息、文件与chunk映射信息、chunk当前位置等）。
4. ChunkServer:文件存储的地方，使用固定大小的chunk存储文件。

## 离线计算和MapReduce

1. 数据离开生产环境后进行的计算，就是把业务数据从在线存储移动到离线存储中，然后进行处理的过程，延迟比较大。
2. MapReduce模型：主要分为两个阶段

Map阶段：根据设定的规则把整体数据映射给不同的Work线程处理，并且生成各自的处理结果。

Reduce阶段：对上一步处理过的数据进行聚合，形成最终结果。

注：一次任务的处理可能不止一次MapReduce过程。

1. Hadoop是MapReduce的一个开源实现，使用HDFS进行数据存储，强调批处理，常用于数据挖掘、分析。

注：HDFS是一个文件被分割成若干块，一个块写完时自动复制到另外两台机器上。

1. Spark为集群中特定类型的工作设计，基于内存的方式，除了能够提供交互式查询外，它还可以优化迭代工作负载。
2. 在线实时计算：Storm框架，实时处理大数据流。

## OSGi

1. Java动态化模块化系统的一系列规范，OSGi可以认为是Java平台的模块层。主要优点：

在不重启容器的情况下，动态地安装、卸载、启动和停止您的应用程序中的不同模块；

应用程序中的某一特定模块，容器可以同时运行该模块的多个版本；  
为开发嵌入式应用、移动应用、富互联网应用(RIA)提供了非常优秀的基础架构。

注：[SpringSource](http://www.springsource.org/)由[Maven转向使用Gradle](https://jira.springsource.org/browse/SPR-8116)来构建系统，[Gradle](http://www.gradle.org/)是Groovy的一款开发工具，已经渐渐抛弃使用OSGi框架和模块化技术。

## NoSql和Redis

1. NoSql:主要指非关系、分布式的数据库设计模式，作为关系数据库的补充，放弃了两大基础：以关系代数为基础的结构化查询语言和事务一致性保证，强化高可用性和可伸缩性。（HBase）
2. Redis是一个开源，内存存储的数据结构服务器，可用作数据库，高速缓存和消息队列代理。它支持[字符串](http://www.redis.net.cn/tutorial/3508.html)、[哈希表](http://www.redis.net.cn/tutorial/3509.html)、[列表](http://www.redis.net.cn/tutorial/3510.html)、[集合](http://www.redis.net.cn/tutorial/3511.html)、[有序集合](http://www.redis.net.cn/tutorial/3512.html)，[位图](http://www.redis.net.cn/tutorial/3508.html)等数据类型。内置复制、LRU收回、[事务](http://www.redis.net.cn/tutorial/3515.html)以及不同级别磁盘持久化功能。优点（异常快速 : Redis是非常快的，每秒可以执行大约110000设置操作，81000个/每秒的读取操作；支持丰富的数据类型；操作都是原子的。 ）
3. Redis 有别于memcache的特点（4点）：

完全在内存中保存数据的数据库，使用磁盘只是为了持久性目的；

相比许多键值数据存储系统有相对丰富的数据类型；

可以将数据复制到任意数量的从服务器中；

Redis既能用作主数据库，由可以作为其他存储系统的辅助数据库。

在Redis中，并不是所有的数据都一直存储在内存中的。Redis只会缓存所有的key的信息，如果Redis发现内存的使用量超过了某一个阀值，将触发swap的操作。

1. 应用场景:   
   redis：数据量较小的更性能操作和运算上  
   memcache：用于在动态系统中减少数据库负载，提升性能；做缓存（适合读多写少，对于数据量比较大，可以采用sharding）。  
   MongoDB: 基于分布式文件存储的数据库，主要解决海量数据的访问效率问题。

## 秒杀活动架构设计

1. 策略：减少http请求，将js，css文件打包成一个文件，系统独立部署，CDN，页面静态化，Web缓存（nginx），租借更多带宽，随机生成下单URL，使用短连接，数据库优化，
2. 设计：使用JS脚本控制“购买”按钮被点亮。每台服务器只接受最多10个下单请求。控制其进入下单页面。

# 设计模式和原则

## 单例模式

原理：

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。通常我们可以让一个全局变量使得一个对象被访问，但他不能防止你实例化多个对象，一个最好的办法就是，让这个类自身保存它的唯一实例。这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问改实例的方法。

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。主要有以下几类：

1。懒汉式：用到时再初始化，线程不安全，多线程情况下不能正常工作，会创建多个实例。

2。懒汉式变种：线程安全，效率很低，99%情况下不需要加锁、解锁的同步操作。

3。恶汉式及其变种：类加载时就初始化，有可能导致在调用其他静态方法的情况下加载。

4。双重检查加锁：先检查实例是否存在，如果不存在才进入下面的同步块，这是第一重检查。进入同步块后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。需要同步一次，从而减少了多次在同步情况下进行判断所浪费的时间。既实现线程安全，又能够使性能不受到很大的影响。

注意：必须用volatile修饰引用变量，否则会导致对象处于不一致的状态。而且volatile禁止指令重排序，效率不高，目前已经被废弃了。

5。延迟初始化占位类模式：通过创建静态内部私有类来达到延迟初始化，且不需要同步。只有在调用单例类的getInstance方法是才会加载内部类。

6。枚举：编写一个包含单个元素的枚举类型，更加简洁，无偿提供序列化机制，绝对防止多次实例化，可以面对复杂序列化和反射攻击。

注意：序列化时调用readObject方法，此方法会返回新的对象，通过readResolve方法可以替换readObject返回的对象。

和原型模式的区别：单例模式将类的构造方法私有化，之后在类的内部产生实例化对象，并通过静态方法返回实例化对象的应用。原型模式是在已指定对象的基础上，然后通过拷贝这些原型对象创建新的对象。当要实例化的类是在运行时刻指定，当一个类的实例只能有几个不同状态组合中的一种时 —— 建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

## 工厂相关模式

1。 普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

2。多个工厂方法模式，是对普通工厂模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法。

3。静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

注意：工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。

4。抽象工厂模式， 工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了开闭原则。设计一个负责创建一组产品的抽象工厂接口，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

原理：

抽象工厂模式存在多个产品族，而简单工厂模式只存在一个产品族。

抽象工厂模式的一个主要目的是把所生成的具体类相分离，这些类的实际名称被隐藏在工厂中，在客户级不必了解

## 适配器模式

1. 原理：

将一个类的接口转换成用户希望的另一个接口，使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。

1. 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。
2. 包含：目标接口，源角色， 适配器（继承目标接口，持有源角色的引用）。
3. 例子：将字符串数据转变成字节数据保存到文件中或者将字节数据转变成流数据等。（InputStreamReader，OutputStreamWriter）

## 迭代器模式

原理

提供一种一致的方法来顺序遍历一个容器中的所有元素

## 装饰器模式

1. 原理：

动态地给一个对象添加一些额外的职责，比单纯的继承更为灵活合理

1. 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。
2. 包含：抽象组件、抽象组件实现者、装饰器接口（持有抽象组件的引用）、具体装饰者。
3. 例子：FilterInputStream为装饰器接口，BufferedInputStream为具体装饰者，其持有InputStream对象的引用，将读取的数据先保存在内存中。

注意：适配器模式和装饰器模式区别：都是包装模式，持有对象引用，但是目的不同，适配器的意义在于将一个接口转变成另一个接口，达到复用。而装饰器模式不改变原来的接口，只是增强原有功能，提升性能等。

装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。换句话 说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模 式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

我们可以用另外一句话来总结这些差别：使用代理模式，代理和真实对象之间的的关系通常在编译时就已经确定了，而装饰者能够在运行时递归地被构造。

## 责任链模式

1. 原理：

使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系。将这些对象连成一条链。

1. 并沿着这条链传递该请求，直到有一个对象处理它为止。在这个链上的所有的对象有相同的接口（抽象类）但却有不同的实现。
2. 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。
3. 包含：客户端（请求发送者）、抽象处理者、具体处理者（处理或传给下家）。
4. 例子：Tomcat的Container容器实现。

## 门面/外观模式

1. 为子系统中的一组接口提供一个一致的界面，使得子系统更加容易使用。主要用在一个大系统中有很多子系统时。（相互通信的同时不能暴露更多数据），例如外交部。
2. 例子：Tomcat 中很多；比如从Request和Response对象封装数据。（HttpRequestFacade，StandardWrapperFacade）

## 观察者模式

1. 原理：

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自动更新。

1. 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。
2. 包含：抽象主题（负责管理所有观察者引用，定义事件主要操作），具体主题，观察者。
3. 例子：邮件订阅和RSS订阅，Listener，Lifecycle。

## 命令模式

1. 将一个请求封装为一个对象，把发出命令的责任和执行命令的责任分离。
2. 包含：命令发起者，命令接收者，命令本身。
3. 例子：Tomcat中的Connecter到Container的请求。

## 代理模式

1. 原理：

为其他对象提供一种代理以控制对这个对象的访问，简单的来说调用这个类的同时其实里面是别的类在干事。

1. 为其他对象提供一种替身以控制对这个对象的访问。
2. 包含：抽象主题，代理类，目标类
3. 例子：远程代理，保护代理JDK动态代理等。

## 策略模式

1. 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。使得算法可独立于使用它的客户而变化，而使用者可以根据需要选择合适的策略。
2. 包含：抽象策略、具体策略、使用环境（根据自身条件选择不同的策略实现类）。
3. 例子：Spring AOP（JDK动态代理和CGLIB代理），Spring事务。

简单工厂是传递相应的条件就能得到想要的一个对象，然后通过这个对象实现算法的操作；而策略模式，使用时必须首先创建一个想使用的类对象（使用context封装），然后将该对象最为参数传递进去，通过调用context中的方法来调用不同的算法。

## 模板模式

1. 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。大的逻辑已经定义好，只需要实现一些具体步骤。
2. 包含：抽象模板和具体模板实现类。
3. 例子：框架的作用就是模板。自定义ClassLoader类时只需要重写findclass方法等。

## MVC模式（策略、观察者、组合）

1. 视图：呈现模型数据。

控制器：取得用户（和视图交互）输入并解读其对模型的意思。

模型：持有所有的数据，状态和程序逻辑。

1. 策略：视图只关心系统中可视的部分，对于任何界面行为，都委托给控制器，对于视图来讲，控制器就是策略，通过策略实现解耦。
2. 观察者：模型为可观察者，视图和控制器为观察者，模型完全独立于视图和控制器。 当模型状态改变时，所有观察者都会被通知。
3. 组合：视图是GUI组件的组合，顶层组件包含其他组件。

## 其他设计模式

1. 访问者模式，目的是将稳定的数据结构和对数据结构的变化多端的操作解耦。
2. 组合模式: 将对象组合成树形结构以表示“部分-整体”的层次结构。
3. 迭代器模式：提供一种方法顺序访问一个聚合对象中各个元素， 而又不需暴露该对象的内部表示。
4. 享元模式：运用共享技术有效地支持大量细粒度的对象。主要目的是实现对象的共享，减少创建对象的内存开销。
5. 中介者模式：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。
6. 状态模式：当对象的状态改变时，同时改变其行为，两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。（QQ）
7. 桥接模式：就是把事物和其具体实现分开，使他们可以各自独立的变化。

例子：JDBC提供统一接口，每个数据库提供各自的实现，数据库驱动的程序进行桥接。

## 6大设计原则

1. 单一职责原则

类T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。

1. 里氏替换原则

子类可以扩展父类的功能，但不能改变父类原有的功能。子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。子类中可以增加自己特有的方法。

1. 依赖倒置原则

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。即面向接口编程。

1. 接口隔离原则

客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。接口必须尽量小。

1. 迪米特法则

一个类对自己所依赖的类应该知道的越少越好，即只与直接朋友通信。（陌生类不要作为局部变量出现在类内部），降低耦合。

1. 开闭原则

对扩展开放，对修改关闭，即通过扩展来实现变化。用抽象构建框架，用实现扩展细节。

1. 总结：：单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。

注意：对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。

# 数据库和SQL

## SQL语句（数据查询和DDL，DML，DCL）

1. 主键：一列（一组列），其值能够唯一区分表中每一行。
2. DISTINCT:只返回唯一的行；（用于所有行，而不仅仅是前置行）；
3. NULL：与字段包含0，空字符串，空格不同。检查某列是否为空：IS NULL；
4. AND在计算次序中优先级更高。
5. IN用来指定条件范围，范围内每一个条件都能匹配，与OR 类似。优点如下：

语法更清楚，更直观，更简洁。事计算次序更容易管理。执行更快。可以包含其他SECLECT语句，动态创建WHERE子句。

1. NOT可以对IN，BETWEEN，EXISTS取反。
2. BINARY：加在搜索模式前表示区分大小写。
3. LIKE：指示后跟的搜索模式利用通配符匹配而不是直接相等匹配进行比较。（通配符：用来匹配一部分的特殊字符）。“%”：任意字符出现任意次数（不匹配NULL），“\_”：匹配单个字符
4. REGEXP：指示后跟的搜索模式用正则表达式匹配。（转义用\\）

注意：LIKE和REGEXP的重要区别在于：LIKE匹配整个列，而REGEXP在列值内匹配。而REGEXP的通配符更加丰富。

1. Concat()函数：用于拼接多个列；
2. Trim，RTrim，LTrim:用于去掉空格；

注意：函数没有SQL可移植性强；

1. 聚集函数：运行在行组上，计算和返回单个值的函数：AVG，COUNT，MAX，MIN，SUM等。
2. GROUP BY：指示分组数据，然后对每个组而不是整个结果进行聚集。（WHERE之后，GROUP之前）。
3. HAVING：过滤分组数据，支持所有Where 操作符。（Where过滤行，或者说Where在分组前过滤，HAVING在分组后过滤）
4. 查询顺序：SELECT、FROM、WHERE、GROUP BY、HAVING、ORDER BY、LIMIT。
5. INSERT INTO SELECT：用于插入被检索出的数据。
6. 删除数据：DELETE，删除符合条件的行，而不是表本身，TRUNCATE删除表，然后重新建一个（所以清空表应该使用TRUNCATE TABLE）

## 内外连接

1. 外键：某个表中的一列，关联到了另外一个表中的主键值。定义了两个表的关系。
2. 内部联结（等值联结，自然联结）：FROM，表名，INNER JOIN，表名，ON。

列出与连接条件匹配的数据行，它使用比较运算符比较被连接列的列值（自联结：自己联结自己）

1. 外部联结：包含了在相关表中没有关联行的行，FROM ，表名 （LEFT，RIGHT，FULL(MySQL不支持全连接)）OUTER JOIN，表名，ON。

注意：左连接以左表为基准进行查询，左表数据会全部显示出来，右表如果和左表匹配的数据则显示相应字段的数据，如果不匹配，则显示为NULL；右连接刚好相反。全连接就是先以左表进行左外连接，然后以右表进行右外连接。

注意：应该总是提供联结条件，否则将返回不正确的数据。

## 视图

1. 从数据库中的基本表中选取出来的逻辑窗口，虚表，本身并不存在。将表与表之间的复杂操作和搜索条件对用户不可见，用户只需要对视图进行查询即可。但是不能提高查询效率。

优点：重用SQL，简化数据库查询，提高数据库的安全性和逻辑独立性。

## 存储过程

1. 一组为了完成特定功能的SQL 语句集，存储在数据库中，经过第一次编译后再次调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。
2. 优点：增加SQL语言的功能，灵活性和安全性。执行速度快，减少网络传输。缺点在于可移植性差。
3. 和函数区别：存储过程是独立的部分，而函数作为查询语句的一部分，嵌入在SQL中，执行速度更快。
4. 游标：用于定位结果集的行，一种能够从包含多条数据记录的结果集中每次提取一条记录的机制。
5. 和函数区别：存储过程实现的过程要复杂一些，而函数的针对性较强；存储过程可以有多个返回值，而自定义函数只有一个返回值；存储过程一般独立的来执行，而函数往往是作为其他SQL语句的一部分来使用；

## 触发器

1. 一种特殊的存储过程，由事件触发（insert，delete，update）。分为事前触发和事后触发。而语句级触发可以在语句执行前或者后执行，行级触发发生在触发器所影响的每一行触发一次。
2. 和存储过程区别：触发器隐时调用，不能接受参数输入。

## 范式

1. 为建立冗余较小，结构合理的数据库时必须遵守的一定规则。
2. 1NF: 所有字段值都是不可分解的原子值，同一列中不能有重复值。（地址）
3. 2NF: 确保数据库表中的每一列都和主键相关，而不能只与主键的一部分相关。（学分）

订单id，商品id——>折扣、数量 符合2NF

订单id，商品id——>商品名称不符合2NF

1. 3NF：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。（订单表客户ID）

订单id——>客户名 符合3NF

订单id——>客户名 客户地址 不符合3NF，存在传递

1. BCNF：一个关系达到3NF，且候选码都是单属性，即主属性之间不存在相互依赖。
2. 4NF：表中不存在多对多关系。（职工）

2NF：非主键列是否完全依赖于主键，还是依赖于主键的一部分；3NF：非主键列是直接依赖于主键，还是直接依赖于非主键列。

在设计中更好的解决数据冗余，数据有效性检查，提高存储效率

第一范式（1NF）：强调的是列的原子性，即列不能够再分成其他几列。

考虑这样一个表：【联系人】（姓名，性别，电话）

如果在实际场景中，一个联系人有家庭电话和公司电话，那么这种表结构设计就没有达到 1NF。要符合 1NF 我们只需把列（电话）拆分，即：【联系人】（姓名，性别，家庭电话，公司电话）。1NF 很好辨别，但是 2NF 和 3NF 就容易搞混淆。

第二范式（2NF）：首先是 1NF，另外包含两部分内容，一是表必须有一个主键；二是没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。

考虑一个订单明细表：【OrderDetail】（OrderID，ProductID，UnitPrice，Discount，Quantity，ProductName）。

因为我们知道在一个订单中可以订购多种产品，所以单单一个 OrderID 是不足以成为主键的，主键应该是（OrderID，ProductID）。显而易见 Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID。所以 OrderDetail 表不符合 2NF。不符合 2NF 的设计容易产生冗余数据。

可以把【OrderDetail】表拆分为【OrderDetail】（OrderID，ProductID，Discount，Quantity）和【Product】（ProductID，UnitPrice，ProductName）来消除原订单表中UnitPrice，ProductName多次重复的情况。

第三范式（3NF）：首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

## 外键

外键具有保持数据完整性和一致性的机制 ，级联操作

## 索引

1. 为了提高表的搜索效率而对某些字段中的值建立的目录 。主要有：[唯一索引](http://baike.baidu.com/view/709651.htm)、主键索引、单列索引、多列索引。
2. 聚集索引的顺序就是数据的物理存储顺序，而非聚集索引顺序与数据的物理排列顺序无关。一个表最多只能有一个聚集索引。其对于那些经常要搜索范围值的列特别有效。聚集索引不是一种单独的索引类型，而是一种存储数据方式。其具体细节依赖于实现方式。
3. 多列索引是指建立一个针对多个列的索引，单列索引是指分别为每个列单独建立的索引；当我们执行查询的时候，MySQL只能使用一个索引。如果你有三个单列的索引，MySQL会试图选择一个限制最严格的索引。但是，即使是限制最严格的单列索引，它的限制能力也肯定远远低于firstname、lastname、age这三个列上的多列索引。  多列索引有最左前缀匹配的特点。
4. Hash索引：检索效率非常高，索引的检索可以一次定位，不像B-Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问。但是有如下缺点：
5. 只能用于等值过滤，不能用基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和Hash运算前完全一样。
6. 不支持联合索引的最优前缀，联合索引中的字段要么全用要么全不用。
7. 不支持索引排序，索引值和计算出来的hash值大小并不一定一致。
8. 遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。

（1）Hash 索引仅仅能满足"="，"IN"和"<=>"查询，不能使用范围查询。

由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值，所以它只能用于等值的过滤，不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和Hash运算前完全一样。

（2）Hash 索引无法被用来避免数据的排序操作。

由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且Hash值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来避免任何排序运算；

（3）Hash 索引不能利用部分索引键查询。

对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。

（4）Hash 索引在任何时候都不能避免表扫描。

前面已经知道，Hash 索引是将索引键通过 Hash 运算之后，将 Hash运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，由于不同索引键存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查询，还是要通过访问表中的实际数据进行相应的比较，并得到相应的结果。

（5）Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。

对于选择性比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。

1. MySQL索引结构为B+树实现。MyISAM表的索引和数据是分开的，用指针指向数据的物理地址，而InnoDB表中索引和数据是储存在一起。

注意：索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引。

1. 在什么情况下适合建立索引  
   为经常出现在关键字order by、group by、distinct后面的字段， 建立索引。在union等集合操作的结果集字段上， 建立索引。 其建立索引的目的同上。  
   为经常用作查询选择的字段， 建立索引。在经常用作表连接的属性上， 建立索引。  
   考虑使用索引覆盖。 对数据很少被更新的表， 如果用户经常只查询其中的几个字段， 可以考虑在这几个字段上建立索引， 从而将表的扫描改变为索引的扫描。
   * 1. 选择唯一性索引。
     2. 为经常需要排序，分组和联合操作的字段建立索引。
     3. 为常作为查询条件的字段建立索引。
     4. 限制索引的数目。
     5. 尽量使用数据量少的索引。
     6. 尽量使用前缀来索引。如果字段的值很长，最好使用值的前缀来索引，如果只检索子酸的前面的若干字符，可以提高检索的速度。
     7. 删除不再使用或者很少使用的索引。

1．选择唯一性索引

唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。例如，学生表中学号是具有唯一性的字段。为该字段建立唯一性索引可以很快的确定某个学生的信息。如果使用姓名的话，可能存在同名现象，从而降低查询速度。

2．为经常需要排序、分组和联合操作的字段建立索引

经常需要ORDER BY、GROUP BY、DISTINCT和UNION等操作的字段，排序操作会浪费很多时间。如果为其建立索引，可以有效地避免排序操作。

3．为常作为查询条件的字段建立索引

如果某个字段经常用来做查询条件，那么该字段的查询速度会影响整个表的查询速度。因此，为这样的字段建立索引，可以提高整个表的查询速度。

4．限制索引的数目

索引的数目不是越多越好。每个索引都需要占用磁盘空间，索引越多，需要的磁盘空间就越大。修改表时，对索引的重构和更新很麻烦。越多的索引，会使更新表变得很浪费时间。

5．尽量使用数据量少的索引

如果索引的值很长，那么查询的速度会受到影响。例如，对一个CHAR(100)类型的字段进行全文检索需要的时间肯定要比对CHAR(10)类型的字段需要的时间要多。

6．尽量使用前缀来索引

如果索引字段的值很长，最好使用值的前缀来索引。例如，TEXT和BLOG类型的字段，进行全文检索会很浪费时间。如果只检索字段的前面的若干个字符，这样可以提高检索速度。

7．删除不再使用或者很少使用的索引

表中的数据被大量更新，或者数据的使用方式被改变后，原有的一些索引可能不再需要。数据库管理员应当定期找出这些索引，将它们删除，从而减少索引对更新操作的影响。

8 。 最左前缀匹配原则，非常重要的原则。

MySQL会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a 1=”” and=”” b=”2” c=”“> 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

9 。=和in可以乱序。

比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

10 。 尽量选择区分度高的列作为索引。

区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就 是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0。1以上，即平均1条扫描10条 记录

11 。索引列不能参与计算，保持列“干净”。

比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本 太大。所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);

12 。尽量的扩展索引，不要新建索引。

比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可

注意：选择索引的最终目的是为了使查询的速度变快。上面给出的原则是最基本的准则，但不能拘泥于上面的准则。读者要在以后的学习和工作中进行不断的实践。根据应用的实际情况进行分析和判断，选择最合适的索引方式。

MySQL只有对以下操作符才使用索引：<，<=，=，>，>=，BETWEEN，IN，以及某些时候的LIKE。可以在LIKE操作中使用索引的情形是指另一个操作数不是以通配符（%或者\_）开头的情形。

1. 为什么要用B+树结构

1。文件很大，不可能全部存储在内存中，故要存储到磁盘上。

2。索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。

3。局部性原理与磁盘预读，预读的长度一般为页的整倍数，(在许多操作系统中，页得大小通常为4k)

4。数据库系统巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。

5。数据库索引采用B+树的主要原因是 B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。(提高区间访问的性能)正是为了解决这个问题，B+树应运而生。B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树需要遍历整棵树，效率太低。而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

1. 什么情况下索引失效
2. Like以%开头，因为b树的结构
3. 隐式转换，id=1和id=’1’,id是字符串时，二者都会转为浮点型（根据规则），MySQL优化器无法确定索引是否更有效，所以选择了其它方案（全表扫描）
4. 不满足最左前缀
5. Mysql预计索引比全表扫描更慢
6. 使用or
7. 主键自增

1)InnoDB使用聚集索引，数据记录本身被存于主索引的叶子节点上，这就要求同一个叶子节点内的各条数据记录按主键顺序存放，因此每当一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子，则开辟一个新的页（节点）如果表使用自增主键，那么每次插入新的记录时，记录就会顺序添加到当前索引节点后续位置，当一页写满，就会自动开辟一个新的页。这样就就会形成一个紧凑的索引结构，近似顺序填满，由于每次插入时也不需要移动所有数据，因此效率很高，也不会增加很多额外的开销维护索引如果使用非自增主键，由于每次插入主键的值近乎于随机，因此每次新纪录都要被插到现有索引页的中间某个位置，此时MySQL不得不为了将新纪录插到合适位置而移动数据，甚至目标页面可能已经被写到磁盘而从缓存中清除，这增加了很多额外开销，同时频繁的移动，分页造成了大量的碎片，得到不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建并优化填充页面。

2)由于MySQL从磁盘读取数据时一块一块来读取的，同时，根据局部性原理，MySQL引擎会选择预读一部分和你当前读数据所在内存相邻的数据块，这个时候这些相邻数据块的数据已经存在于内存中。由于数据库大部分是查询操作，这个时候，如果主键是自增的话，数据存储都是紧凑地存储在一起的，那么对于局部性原理利用和避免过多地I/O操作都有着巨大的促进作用

## B树，B+树，红黑树

1. B树，即多路平衡查找树，节点存放的是键-值对，查找给定[关键字](http://baike.baidu.com/view/390935.htm)的方法是，首先把根结点取来，在根结点所包含的关键字K1，…，Kn查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查找的关键字在Ki与Ki+1之间，Pi为指向子树根节点的指针，此时取[指针](http://baike.baidu.com/view/159417.htm)Pi所指的结点继续查找，直至找到，或指针Pi为空时查找失败。支持对保存在磁盘或者网络上的符号表进行外部查找。B树是用于存储海量数据的，一般其一个结点就占用磁盘一个块的大小。

插入操作：m-阶B树规定的结点的最大容量是m-1个元素，故当插入操作造成超出容量之后也得分裂，其分裂成两个结点每个结点分m/2个元素。副作用是在其父结点中要插入一个中间元素，用于分隔这两结点。再向父结点插入一个元素也可能会造成父结点的分裂，逐级向上操作，直到不再造成分裂为止。

删除操作：首先查找B树中需删除的元素，如果该元素在B树中存在，则将该元素在其结点中进行删除，如果删除该元素后，首先判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的某相近元素(“左孩子最右边的节点”或“右孩子最左边的节点”)到父节点中。然后判断移动之后的情况；移动相应元素之后，如果某结点中元素数目（即关键字数）小于ceil(m/2)-1，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于ceil(m/2)-1），如果丰满，则向父节点借一个元素来满足条件；如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于ceil(m/2)-1，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点，以此来满足条件。

1. B+树是应[文件](http://baike.baidu.com/view/345685.htm)系统所需而出的一种[B-树](http://baike.baidu.com/view/2228473.htm)的变型树。差异在于：

1。所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。(而B 树的叶子节点并没有包括全部需要查找的信息)

2。所有的非终端结点可以看成是索引部分，B 树的非终节点也包含需要查找的有效信息。

1. 红黑树（平衡二叉查找树）：O(log n)时间内做查找，插入和删除。

性质1。 节点是红色或黑色。

性质2。 根是黑色。

性质3。 所有叶子都是黑色（叶子是null节点）。

性质4。 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

性质5。 从任一节点到其每个叶子的所有简单路径 都包含相同数目的黑色节点。

注：红黑树只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。红黑树能够以O(log2 n) 的时间复杂度进行搜索、插入、删除操作，任何不平衡都会在三次旋转之内解决。算法时间复杂度和AVL相同，但统计性能比AVL树更高。

## MySQL数据库引擎比较

1. MyISAM和InnoDB引擎的区别  
   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不支持外键，而InnoDB支持外键。  
MyISAM相对简单， 所以在效率上要优于InnoDB，小型应用可以考虑使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。

1. 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
2. INNODB在做SELECT的时候，要维护的东西比MYISAM引擎多很多；

1）数据块，INNODB要缓存，MYISAM只缓存索引块， 这中间还有换进换出的减少；

2）innodb寻址要映射到块，再到行，MYISAM 记录的直接是文件的OFFSET，定位比INNODB要快

3）INNODB还需要维护MVCC一致；虽然你的场景没有，但他还是需要去检查和维护

## 事务

1. 概念：事务是由一系列对系统中数据进行访问与更新的操作组成的一个程序执行逻辑单元
2. 实现原理：收藏夹里

http://www。hollischuang。com/archives/943

1. 原子性: 事务是一个不可分割的整体，要么全执行，要不全不执行。

一致性: 执行之前和之后，数据库数据必须保持状态的一致性。如转账前后总金额一致。

隔离性: 使事务操作彼此独立的和透明的。一个事务内部的操作不能为其他事务看到。当前事务不能查看由另一个并发事务正在修改的数据

持久性: 保证对事务的修改是永久性的，通过数据库备份和恢复保证。

1. 事务隔离级别：

1。未提交读：查询不会受到任何增删改操作的影响，允许其他事务看到没有提交的数据。不能保证事务一致性，会出现脏读、不可重复读和幻读。

2。提交读：一个事务只能看见已经提交事务所做的改变，会出现不可重复读和幻读。事务读取的时候获取读锁， 但是在读完之后立即释放(不需要等事务结束)， 而写锁则是事务提交之后才释放， 释放读锁之后， 就可能被其他事务修改数据。

3。重复读：确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。会出现幻读。因为这个时候其他事务不能更改所选的数据， 但是可以增加数据，MySQL默认。

4。串行化：隔离级别最高，不允许出现脏读、不可重复读和幻读。即一个事务执行结束了另一个事务才能执行。当然并发性也就最差。强制事务排序

1. 脏读、不可重复读、幻读含义：

脏读：一个事务可以读物另一个事务未提交的数据。事务T1更新了一行记录的内容， 但是并没有提交所做的修改。 事务T2读取更新后的行， 然后T1执行了回滚操作， 取消了刚才所做的修改。 现在T2读取的行就无效。

不可重复读：在一个事务中不同时间段查询出现不同的结果，可能被更新或者删除。事务T1读取了一行记录， 紧接着T2修改了T1刚才读取的那一行记录， 然后T1又再次读取这行记录， 发现与刚才读取的结果不同。侧重于修改

幻读:在一个事务中不同时间段查询，记录数不同。与不可重复读的区别是：在幻读中，已经读取的数据不会改变，只是与以前相比，会有更多的数据满足查询条件。事务T1读取一条指定的Where子句所返回的结果集， 然后T2事务新插入一行记录， 这行记录恰好可以满足T1所使用的查询条件。 然后T1再次对表进行检索，但又看到了T2插入的数据。侧重于增加删除

当A事务执行中，未提交，B事务的任何插入，select操作都会等待，因此不会出现不可重复读

## 分布式事务

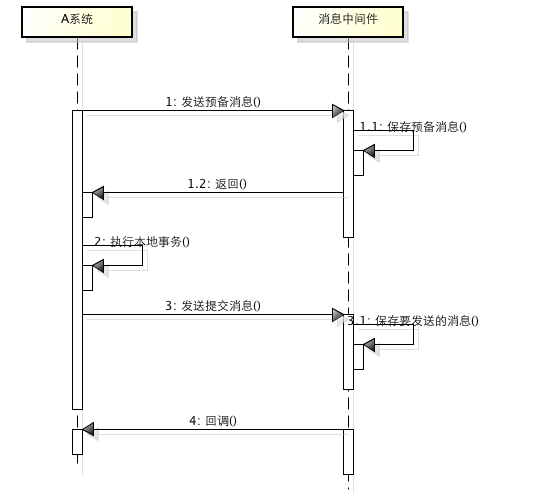
C（一致性）：所有的节点上的数据时刻保持同步

A（可用性）：每个请求都能接受到一个响应，无论响应成功或失败

P（分区容错）：系统应该能持续提供服务，即使系统内部有消息丢失（分区）

AP wihtout C：要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性，具体原理如下：



1、A系统向消息中间件发送一条预备消息

2、消息中间件保存预备消息并返回成功

3、A执行本地事务

4、A发送提交消息给消息中间件

通过以上4步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

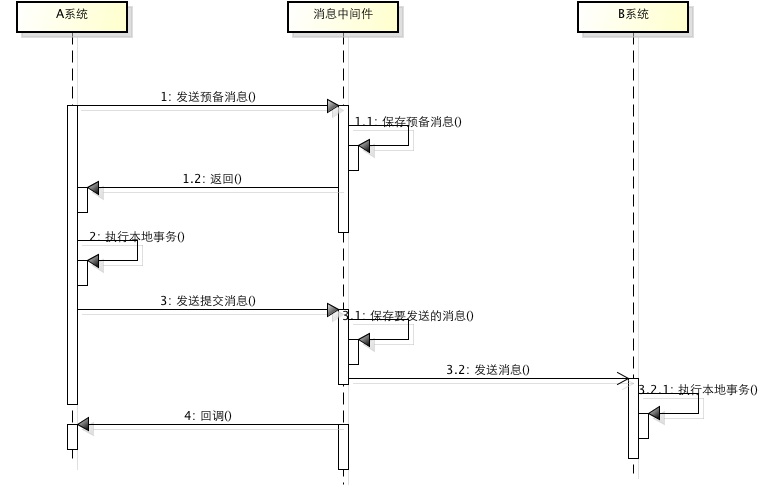
步骤一出错，则整个事务失败，不会执行A的本地操作

步骤二出错，则整个事务失败，不会执行A的本地操作

步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息

步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。原理如下：



虽然上面的方案能够完成A和B的操作，但是A和B并不是严格一致的，而是最终一致的，我们在这里牺牲了一致性，换来了性能的大幅度提升。当然，这种玩法也是有风险的，如果B一直执行不成功，那么一致性会被破坏，具体要不要玩，还是得看业务能够承担多少风险。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1）支付宝在扣款事务提交之前，向实时消息服务请求发送消息，实时消息服务只记录消息数据，而不真正发送，只有消息发送成功后才会提交事务；

2）当支付宝扣款事务被提交成功后，向实时消息服务确认发送。只有在得到确认发送指令后，实时消息服务才真正发送该消息；

3）当支付宝扣款事务提交失败回滚后，向实时消息服务取消发送。在得到取消发送指令后，该消息将不会被发送；

4）对于那些未确认的消息或者取消的消息，需要有一个消息状态确认系统定时去支付宝系统查询这个消息的状态并进行更新。为什么需要这一步骤，举个例子：假设在第2步支付宝扣款事务被成功提交后，系统挂了，此时消息状态并未被更新为“确认发送”，从而导致消息不能被发送。

还有一个很严重的问题就是消息重复投递，以我们支付宝转账到余额宝为例，如果相同的消息被重复投递两次，那么我们余额宝账户将会增加2万而不是1万了。

为什么相同的消息会被重复投递？比如余额宝处理完消息msg后，发送了处理成功的消息给支付宝，正常情况下支付宝应该要删除消息msg，但如果支付宝这时候悲剧的挂了，重启后一看消息msg还在，就会继续发送消息msg。

解决方法很简单，在余额宝这边增加消息应用状态表（message\_apply），通俗来说就是个账本，用于记录消息的消费情况，每次来一个消息，在真正执行之前，先去消息应用状态表中查询一遍，如果找到说明是重复消息，丢弃即可，如果没找到才执行，同时插入到消息应用状态表（同一事务）。

## Oracle和MySQL数据库比较

1. 大体一致
2. 分页查询时oracle用的伪列（rownum），mysql用的是limit   
   oracle对sql语句要求更为严格，而且oracle里变量较mysql更多点，有number型，有大数据类型；  
   oracle不能插入为空列，而mysql是可以的。

两者函数和语法有不同之处，如转日期函数oracle是to\_date，而mysql是str\_to\_date  
oracle不能设置列自动增长，而mysql是可以的，oracle可以用序列加触发器来解决自动增长问题达到与mysql一样的效果。

1. 总之，Oracle格式很严格，很安全。

## SQL优化知识

1. 使用短索引，对字符串列进行索引，指定一个前缀长度，如10，可以节约空间，索引高速缓存块可以容纳更多键值，mysql也可以在内存中容纳更多的值
2. 用PreparedStatement一般来说比Statement性能高。
3. 因为搜索时间更长，不要过度使用LIKE和通配符，如果使用，不要放在搜索模式的开始处。
4. 能小就用小，能用整形就不用浮点型（如IP，价格）：使用能正确的表示和存储数据的最短类型。这样可以减少对磁盘空间、内存、cpu缓存的使用。

注意：Tinyint、smallint、mediumint、int、bigint，分别需要8、16、24、32、64。 int(1) 和int(11)是一样的，唯一区别是mysql客户端显示的时候显示多少位。

1. 避免用NULL，尽可能的使用 NOT NULL，count(列)是不会统计列值为null的行数。
2. char适用于存储很短或固定或长度相似字符，如 MD5加密；varchar适用于当最大长度远大于平均长度并且发生更新的时候。
3. 利用LIMIT 1取得唯一行。
4. 尽量避免SELECT \*命令从表中读取越多的数据，查询会变得更慢。
5. 不要再Where子句中使用不等于操作符，进行NULL判断，对字段进行表达式操作，函数操作。导致引擎放弃使用索引。
6. 为搜索字段建索引。
7. 千万不要 ORDER BY RAND()
8. 使用 ENUM 而不是 VARCHAR，字段的取值是有限而且固定时。
9. 固定长度的表会更快。（ 不包含字段：VARCHAR，TEXT，BLOB。固定长度的表会提高性能，因为MySQL搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量的）。

# Linux

Linux是一套免费使用和自由传播的[类Unix](http://baike.baidu.com/view/3289073.htm)[操作系统](http://baike.baidu.com/subview/880/4940471.htm)，是一个基于[UNIX](http://baike.baidu.com/view/8095.htm)的多用户、[多任务](http://baike.baidu.com/view/757238.htm)、支持[多线程](http://baike.baidu.com/view/65706.htm)和多[CPU](http://baike.baidu.com/view/2089.htm)的操作系统。它能运行主要的UNIX工具软件、应用程序和网络协议。支持[32位](http://baike.baidu.com/subview/125389/10889278.htm)和[64位](http://baike.baidu.com/view/125381.htm)硬件。Linux继承了[Unix](http://baike.baidu.com/view/8095.htm)以[网络](http://baike.baidu.com/view/3487.htm)为核心的设计思想，是一个性能稳定的多用户网络操作系统。

## 问答题

1. 在Linux系统中，以文件方式访问设备。Linux内核引导时，从文件 /etc/fstab中读取要加载的文件系统。
2. Linux文件系统中每个文件用索引节点来标识。Linux文件系统使用索引节点来记录文件信息，索引节点是一个数据结构，它包含了一个文件的文件名，位置，大小，建立或修改时间，访问权限，所属关系等文件控制信息，一个文件系统维护了一个索引节点的数组，每个文件或目录都与索引结点数组中的唯一一个元素对应，系统为每个索引结点分配了一个号码，也就是该结点在数组中的索引号，称为索引结点号。
3. linux硬盘组织方式为:引导区、超级块，索引结点，数据块，目录块。其中超级块中包含了关于该硬盘或分区上的文件系统的整体信息，如文件系统的大小等；超级块后面的数据结构是索引结点，它包含了针对某一个具体文件的几乎全部信息，如文件的存取权限、所有者、大小、建立时间以及对应的目录块和数据块等；数据块是真正存储文件内容的位置。但是索引结点中不包括文件的名字，文件名是放在目录块里的。目录块里包含有文件的名字以及此文件的索引结点编号。
4. 进程查看的命令是ps和top。进程调度的命令有at，crontab，batch，kill。

注：ps看到的是命令执行瞬间的进程信息，而top可以持续的监视，ps只是查看进程，而top还可以监视系统性能，如平均负载，cpu和内存的消耗，另外top还可以操作进程，如改变优先级(命令r)和关闭进程(命令k)。

1. 链接分硬链接和符号链接：符号链接可以建立对于文件和目录的链接。符号链接可以跨文件系统，相当于快捷方式。硬链接不可以跨文件系统。它只能建立对文件的链接。相当于复制文件。
2. NFS网络文件系统是应用层的一种应用服务，它主要应用于Linux和Linux系统、Unix系统之间的文件或目录的共享。对于用户而言可以通过 NFS方便的访问远地文件系统，使之成为本地文件系统的一部分。
3. SNAT是指在数据包从网卡发送出去的时候，把数据包中的源地址部分替换为指定的IP。DNAT，就是指数据包从网卡发送出去的时候，修改数据包中的目的IP 。
4. FTP协议的两种工作模式：主动模式和被动模式；主动模式的FTP是指服务器主动连接客户端的数据端口，被动模式的FTP是指服务器被动地等待客户端连接自己的数据端口。
5. ftp有两种登录方式：匿名登录和授权登录。使用匿名登录时，用户名为：anonymous，密码为：任何合法email地址；使用授权登录时，用户名为用户在远程系统中的用户帐号，密码为用户在远程系统中的用户密码。  
   区别：使用匿名登录只能访问ftp目录下的资源，默认配置下只能下载；而授权登录访问的权限大于匿名登录，上载、下载均可。
6. chmod是Linux下设置文件权限的命令，后面的数字表示不同用户或用户组的权限。一般是三个数字：第一个数字表示文件所有者的权限；第二个数字表示与文件所有者同属一个用户组的其他用户的权限；第三个数字表示其它用户组的权限。权限分为三种：读（r=4），写（w=2），执行（x=1）。
7. 12。Linux运行级别0-6的各自含义

0：关机模式，1：单用户模式，2：无网络支持的多用户模式，3：有网络支持的多用户模式（文本模式，工作中最常用的模式），4：保留，未使用，5：有网络支持的X-windows支持多用户模式（桌面），6: 重新引导系统，即重启。

1. Linux的开机启动流程(8步)

a。加载BIOS硬件信息，自检，比如加载硬盘。

b。读取MBR（主引导分区，在启动设备的第一个扇区）的Boot Loader（引导装载程序）。

注：因为不同操作系统的文件格式不一致，所以每种操作系统都有自己的Boot Loader。

c。依据Boot Loader的设置加载kernel。Kernel开始检测硬件和加载驱动程序。

d。Kernel主动调用init进程，依据inittab文件设定运行级别。

e。init执行rc。sysinit文件准备软件执行的操作环境。

f。init启动各个服务。

g。执行/etc/rc。d/rc。local，自定义开机启动程序。

h。开启终端模拟程序mingetty，启动login进程，进入系统登陆界面。

注：双系统必须先装Windows，再装Linux。Linux的Boot loader有三个功能：提供菜单供用户选择不同的启动项，加载内核文件，转交其他loader。而Windows的Loader不具有控制权转交功能，导致MBR被覆盖。

1. 文件删除原理：Linux系统是通过link的数量来控制文件删除的，只有当一个文件不存在任何link的时候，这个文件才会被删除。一般来说每个文件两个link计数器来控制i\_count和i\_nlink。当一个文件被一个程序占用的时候i\_count就加1。当文件的硬链接多一个的时候i\_nlink也加1。删除一个文件，就是让这个文件，没有进程占用，同时i\_link数量为0。

## 常用命令

1. 组管理命令

groupadd #添加组

groupdel #删除用户组

groupmod #修改用户组

groups  #显示当前用户所属的用户组

1. #用户管理命令

useradd #添加用户

passwd  #为用户设置密码

usermod #修改用户命令，可以通过usermod 来修改登录名、用户的家目录等等

finger #查看用户信息工具（危险命令，一般不用）

id #查看用户的UID、GID及所归属的用户组

su #用户切换工具

who   #显示目前已登录用户信息

last     #列出目前与过去登入系统的用户相关信息

1. 关机和重启服务器的命令

Shutdown，halt，reboot，init(所有进程的祖先，其进程号始终为1)。

1. 安装和登录命令：login、install
2. Mount（加载文件系统）

umount（卸载）

chsh（更改使用者shell设定）

exit（退出系统）

1. 文件处理命令：file（判断文件类型）、mkdir（新建子目录）、grep（指定文件中搜索特定的内容，并将含有这些内容的行标准输出）、dd（复制文件，并根据参数将数据转换和格式化）、find（搜索文件）、mv（文件或目录改名，或者将文件由一个目录移入另一个目录中）、ls（显示目录内容）、diff（用于两个文件之间的比较，并指出两者的不同）、cat（用于连接并显示指定的一个和多个文件的有关信息）、ln（为文件创建链接）、rm删除目录文件；
2. 系统管理相关命令：df（检查文件系统的磁盘空间占用情况）、top（显示执行中的程序进程）、free（显示内存的使用情况）、quota（显示磁盘使用情况和限制情况）、at（在指定时刻执行指定的命令序列）、lp（打印文件）、kill（中止一个进程）、crontab（修改crontab配置文件，然后该配置由cron公用程序在适当的时间执行）、Linux查看IO的工具主要有两个，iostat(集成于sysstat包中)和iotop。前者经常用于查看分区的IO情况，后者常用来查看每个进程的IO占用。
3. 网络操作命令：ifconfig（查看和更改网络接口的地址和参数）、ip（网络配置工具）、ping（检测主机网络接口状态，检测网络是否连通，它使用ICMP协议）、netstat（综合性的网络状态的察看工具，如开启的端口、在为哪些用户服务，以及服务的状态等。此外，它还显示系统路由表、网络接口状态等。）、ftp（远程文件传输）、route（手工产生、修改和查看路由表）、rcp（远程文件拷贝）、finger（查询一台主机上的登录账号的信息，通常会显示用户名、主目录、停滞时间、登录时间、登录Shell等信息，）、mail（发送电子邮件）。
4. 系统安全相关命令：chgrp（修改一个或多个文件或目录所属的组）、chown（更改一个或多个文件或目录的属主和属组）、sudo(以限制配置文件中的命令为基础，在有限时间内给用户使用，并且记录到日志中的命令)、ps（显示瞬间进程）。
5. 其它命令：tar（备份文件）、unzip（将文件压缩成。zip文件）、gunzip（解压文件）、man（提供在线帮助）、uunendcode（把一个二进制文件表编码为一个文本文件）。

# 计算机网络

## OSI参考模型

1. 物理层：利用传输介质为数据链路层提供物理连接，实现比特流的透明传输。
2. 数据链路层：负责建立和管理节点间的链路，实现无差错的、能可靠传输数据帧的数据链路。通过各种控制协议，将有差错的物理信道变为无差错的、能可靠传输数据帧的数据链路。
3. 网络层：地址管理和路由选择。 通过路由选择算法，为报文或分组通过通信子网选择最适当的路径。
4. 传输层：管理两个节点的数据传输。向用户提供可靠的端到端的差错和流量控制，保证报文的正确传输。
5. 会话层：组织和协调两个会话进程之间的通信，并对数据交换进行管理。
6. 表示层：将设备固有的数据格式转化为网络标准传输格式。
7. 应用层：直接向用户提供服务，完成用户希望在网络上完成的各种工作。

注：物理层设备：中继器，集线器；数据链路层设备：网桥，交换机；网路层设备：三层交换机，路由器。

注：TCP/IP协议分为硬件层(a)，网络接口层(b)，互联层(c)，传输层(d)，应用层（efg）。

路由器和交换机，二者区别如下：  
1，路由器工作于OSI模型的网络层，能够识别IP地址，并根据IP地址转发数据包，并维护着路由表，能够基于路由表进行最佳路线选择；  
2，路由器上还能开启ACL访问控制列表、NAT地址转换等功能，扩展网络应用，；  
3，传统交换机工作于OSI模型的数据链路层，能够识别MAC地址，根据MAC地址转发数据帧，并维护着一张桥表，根据桥表上MAC地址和端口的对应关系进行数据帧转发。  
4，交换机能够隔离冲突域，并划分VLAN。

## 数据链路层

1. 概述：细分为介质访问控制层和逻辑链路控制层，同一种数据链路节点之间提供两个设备的直连功能。也就是在互连同一种数据链路的节点之间提供包传递。
2. 以太网帧格式：a。目标MAC地址(6B)，b。源MAC地址(6B)，c。上层协议类型(2B)，d。 帧校验序列FCS(4B)。
3. 注：MAC地址用于标示同一种数据链路中不同计算机的识别码，为48比特。
4. VPN：虚拟专用网络，一般用于加密和远程传输。

## IP协议及相关

1. ping 程序是用来探测主机到主机之间是否可通信，如果不能ping到某台主机，表明不能和这台主机建立连接。ping 使用的是ICMP协议，它发送icmp回送请求消息给目的主机。ICMP协议规定：目的主机必须返回ICMP回送应答消息给源主机。如果源主机在一定时间内收到应答，则认为主机可达。
2. 概述：把数据从源传送到目的地。它不负责保证传送可靠性，流控制，包顺序等。无连接，尽力服务。
3. Ipv4:32bit，由网络标示+主机标示组成。分为A(0)，B(10)，C(110)，D(1110，无主机标示，常用于多播)四类。

Ipv4首部：版本（4b）-首部长度（4b，默认20字节）-区分服务(8b)-总长度（16b）-标示（16b，用于分片重组）-标志（3b，分片信息）-片偏移（13b，标示分片相对于原始数据位置）-生存时间（8b）-上层协议类型（8b）-首部校验和（16b）-源地址（32b）-目的地址（32b）-其他。

1. Ipv6:地址长度128b，性能提升，即插即用，认证与加密，无首部校验和。
2. 广播：同一链路中相互连接的主机之间，无法穿透路由。（将主机部分设置为1）

多播：用于就按数据包发送给特定组内的所有主机，且可以穿透路由。

1. 子网掩码：将原网络分为多个物理网络的一种机制，其网络标示全部为1，主机部分全为0（可变子网掩码）
2. ARP协议:只使用于Ipv4，工作流程：a。ARP缓冲区建立ARP表；b。查找列表，没有则广播；c。目标地址将自己的MAC地址发给源主机，d。源主机添加到自己的列表中，同时进行数据传输。

注：RARP协议完成相反工作，从MAC地址到IP地址的转换。

1. ICMP协议:确认IP包是否成功送达目标地址，通知再发送过程中被废弃的具体原因，从而进行网络问题诊断和改善网络设置等。（消息类型主要有：目标不可达，重定向，超时，回送（ping）等。）
2. DHCP协议：动态主机配置协议，一种局域网的网络协议，使用UDP协议工作，主要有两个用途：给内部网络或网络服务供应商自动分配IP地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段。
3. NAT协议：是一种将私有地址转化为合法IP地址的技术，解决了lP地址不足的问题，而且还能够有效地避免来自网络外部的攻击，隐藏并保护网络内部的计算机。

存在问题:a。无法从外向内建立连接，b。转换表的生成与转换操作会产生一定的开销，c。重启时，TCP连接重置，解决方法：Ipv6或者NAT穿越。

1. 路由协议

RIP协议 ：它选择路由的度量标准是跳数，最大跳数是15跳，如果大于15跳，它就会丢弃数据包。

OSPF协议 ：是链路状态路由选择协议，它选择路由的度量标准是带宽，延迟。

## 传输层

1. TCP保证可靠传输

* 确认和超时重传：接收方收到报文就会确认，发送方发送一段时间后没有收到确认就重传。
* 数据校验。保持首部和数据的检验和，收到的段有差错，则丢弃此报文并不确认接收
* 对失序数据包进行重排序。接收方会缓存未按序到达的数据，重新排序后再交给应用层。
* 流量控制。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。
* 拥塞控制。当网络拥塞时，减少数据的发送。

1. TCP（打电话）和UDP（寄信）的区别：a。TCP面向连接，可靠的传输，用于在传输层有必要实现可靠传输的情况；UDP无连接，不可靠，适用于高速传输和实时性要求高的场合，比如：视频通话、广播等。
2. 端口号：用于标示同一台计算机中进行通信的不同应用程序。

注：标示一个通信有5个要素：源IP地址，目标IP地址，协议号，源端口号，目标端口号。与传输层无关。

1. UDP首部：源端口（16b），目标端口（16b），包长度（16b），校验和（16b）
2. TCP首部：源端口号（16b），目标端口（16b），序列号（32b，发送数据的位置），确认应答号（32b，下一次应该收到的数据序列号），数据偏移（4b，即TCP首部长度），控制位（8b），窗口大小（16b），校验和（16b），紧急指针（16b）。
3. TCP和UDP区别

UDP 是无连接的， 即发送数据之前不需要建立连接。UDP 使用尽最大努力交付， 即不保证可靠交付， 同时也不使用拥塞控制。UDP 是面向报文的。 UDP 没有拥塞控制， 很适合多媒体通信的要求。UDP 支持一对一、 一对多、 多对一和多对多的交互通信。UDP 的首部开销小， 只有 8 个字节。

TCP 是面向连接的运输层协议。每一条 TCP 连接只能有两个端点， 每一条 TCP 连接只能是点对点的（ 一对一） 。TCP 提供可靠交付的服务。TCP 提供全双工通信。TCP是面向字节流。  首部最低20个字节。

1. TCP建立连接的3次握手（双向全双工通信）：客户端发送SYN请求建立连接，服务器回复ACK确认应答，同时发送SYN请求建立连接，客户端回复ACK确认应答。

已失效的连接请求报文段突然传送到了服务端

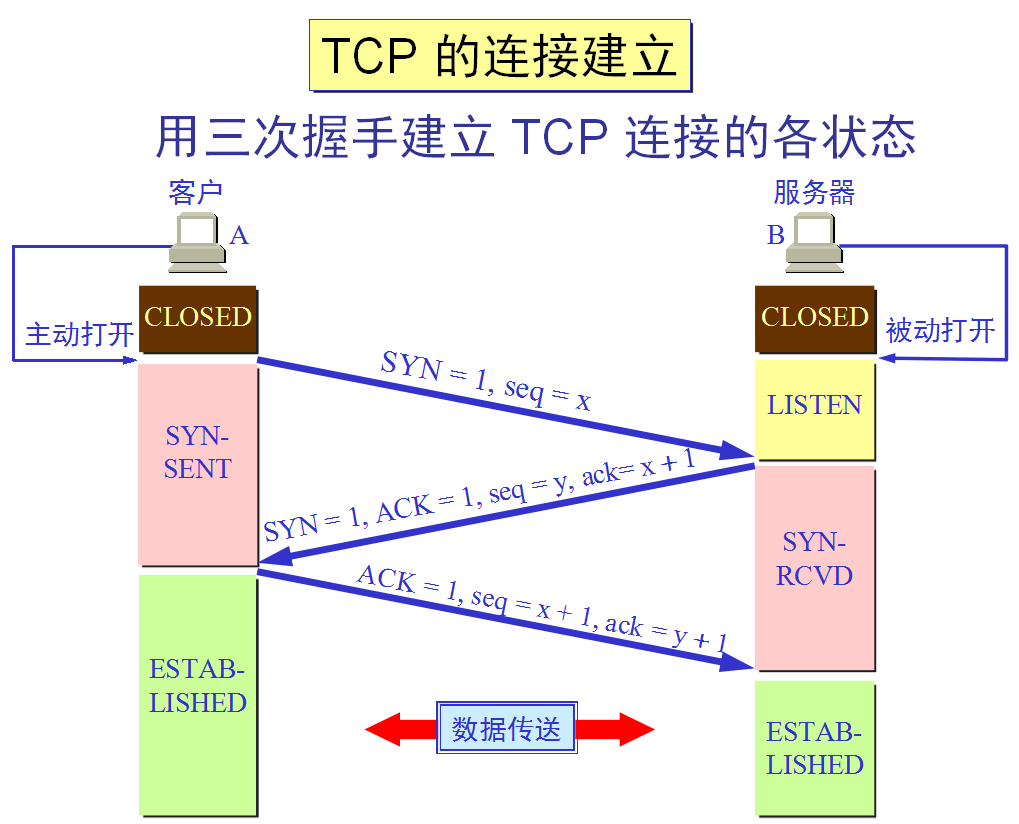
TCP断开连接的4次握手：客户端发送FIN请求断开连接，服务器回复ACK确认应答，服务器发送FIN请求断开连接，客户端回复确认应答。

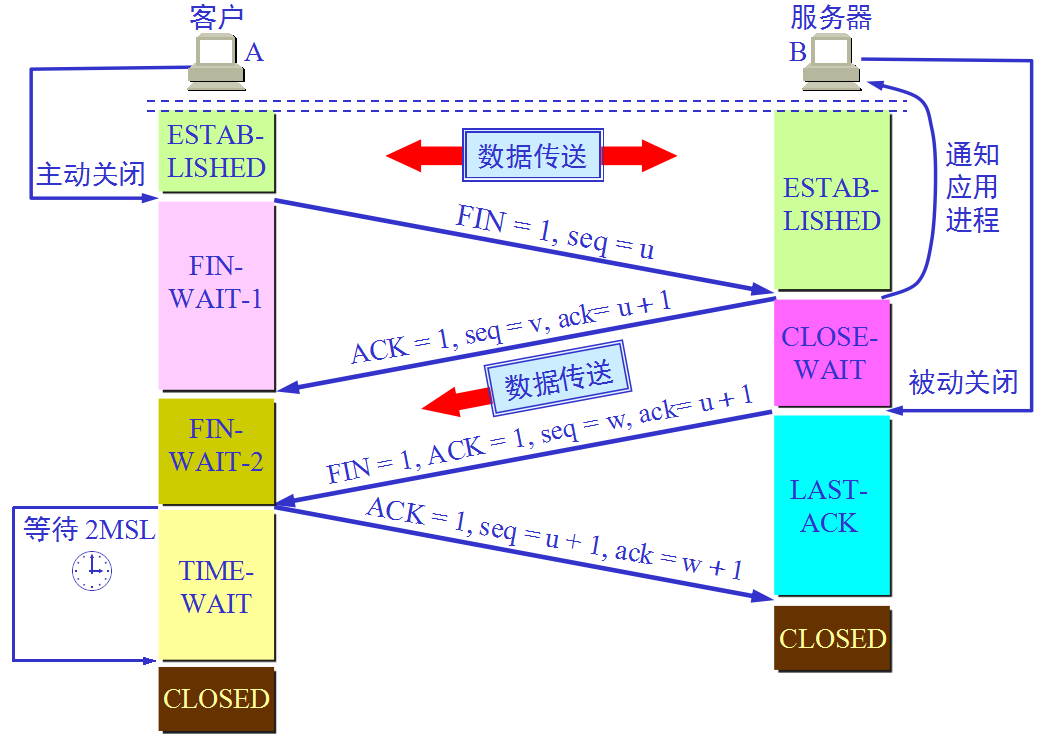
客户端状态变化：CLOSED->SYN\_SENT->ESTABLISHED->FIN\_WAIT1->FIN\_WAIT2->WAIT->CLOSED。

服务端状态变化：CLOSED->LISTEN->SYN\_RECEIVED->ESTABLISHED->CLOSE\_WAIT->LAST\_ACK->CLOSED。

注意:四次握手不是关闭TCP连接的唯一方法。 有时，如果主机需要尽快关闭连接(或连接超时，端口或主机不可达)，RST (Reset)包将被发送。由于RST包不是TCP连接中的必须部分， 可以只发送RST包(即不带ACK标记)。 但在正常的TCP连接中RST包可以带ACK确认标记，请注意RST包是可以不要收到方确认的。

注：保证可靠传输的机制：数据包校验，超时重传机制，应答机制，对失序数据包重排序，TCP还能提供流量控制。





1. TCP漏洞：SYN攻击属于DOS攻击的一种，它利用TCP协议缺陷，通过发送大量的半连接请求，耗费CPU和内存资源。SYN攻击除了能影响主机外，还可以危害路由器、防火墙等网络系统，事实上SYN攻击并不管目标是什么系统，只要这些系统打开TCP服务就可以实施。当服务器未收到客户端的确认包时，重发请求包，一直到超时，才将此条目从未连接队列删除。通常，客户端在短时间内伪造大量不存在的IP地址，向服务器不断地发送syn包，服务器回复确认包，并等待客户的确认，由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，目标系统运行缓慢，严重者引起网络堵塞甚至系统瘫痪。防范措施：过滤网关防护（网关超时设置、SYN网关、SYN代理、增加最大半连接数、缩短超时时间）。
2. TCP粘包问题

指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。出现粘包现象的原因是多方面的，它既可能由发送方造成，也可能由接收方造成。发送方引起的粘包是由TCP协议本身造成的，TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据。若连续几次发送的数据都很少，通常TCP会根据优化算法把这些数据合成一包后一次发送出去，这样接收方就收到了粘包数据。接收方引起的粘包是由于接收方用户进程不及时接收数据，从而导致粘包现象。这是因为接收方先把收到的数据放在系统接收缓冲区，用户进程从该缓冲区取数据，若下一包数据到达时前一包数据尚未被用户进程取走，则下一包数据放到系统接收缓冲区时就接到前一包数据之后，而用户进程根据预先设定的缓冲区大小从系统接收缓冲区取数据，这样就一次取到了多包数据。解决措施：一是对于发送方引起的粘包现象，用户可通过编程设置来避免，TCP提供了强制数据立即传送的操作指令push，TCP软件收到该操作指令后，就立即将本段数据发送出去，而不必等待发送缓冲区满；二是对于接收方引起的粘包，则可通过优化程序设计、精简接收进程工作量、提高接收进程优先级等措施，使其及时接收数据，从而尽量避免出现粘包现象；三是由接收方控制，将一包数据按结构字段，人为控制分多次接收，然后合并，通过这种手段来避免粘包。

注：为什么连接的时候是三次握手，关闭的时候却是四次握手？  
答：三次握手是为了防止已失效的连接请求报文段突然又传送到了服务端， 因而产生错误。四次握手是因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。如果没有这个状态，而是直接就是close状态的话，对面没有收到ack，重发FIN包，这边会发送RST包，导致server端收到rst包报错，影响程序进程； 并且假设没有time\_wait状态，新的连接请求过来，老的关闭报文就会对新的连接产生干扰。

长连接和短连接的区别：

长连接，指一直保持连接，不管当前是否发送或者接收数据。在一个TCP连接上可以连续发送多个数据包，在TCP连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接，短连接就是只有在有数据传输的时候才进行连接，传输数据完毕就关闭连接。短连接是指通信双方有数据交互时，就建立一个TCP连接，数据发送完成后，则断开此TCP连接，即每次TCP连接只完成一次读写操作。

1. 相关控制

重发控制：接收端查询接收数据TCP首部中的序列号和数据长度，将自己下一步应该接收的序号作为确认应答发送出去。收不到确认应答则重发，，重发达到一定次数就判断对端主机异常，强制关闭连接，并且通知应用通信强行终止。重发时间比往返时间及其偏差大一点。

窗口控制：用于提高速度，窗口大小指无须等待确认应答而可以继续发送数据的最大值。即使有少部分ACK丢失也无需重发，接收端会告知下一个应该发送的序列号，如果出现数据丢失，发送端主机连续3次收到同一个确认应答，就会将其相应的数据重发。

流控制：接收端主机向发送端主机通知自己可以接收数据的大小，于是发送端就不会发送超过此限度的数据，该大小成为窗口大小。

拥塞控制与慢启动：启动时拥塞窗口设置为1个MSS，然后以指数增加，与接收端主机通知的窗口大小取较小值，发送数据（超时重发时拥塞窗口将为原拥塞窗口的一半）。MSS：最大消息长度。

1. Nagle算法

任意时刻，最多只能有一个未被确认的小段。 所谓“小段”，指的是小于MSS尺寸的数据块，所谓“未被确认”，是指一个数据块发送出去后，没有收到对方发送的ACK确认该数据已收到。该算法主要用于避免过多小分节报文在网络中传输，从而降低网络容量利用率。比如一个20字节的TCP首部+20字节的IP首部+1个字节的数据组成的TCP数据报，有效传输通道利用率只有将近1 /40。如果网络充斥着这样的小分组数据，则网络资源的利用率是相当低下的。

## 应用层

1. TCP对应的协议：文件传输协议FTP（21），SSH（22），远程登录协议Telnet（23），发送电子邮件协议SMTP（25），接收邮件协议POP3(110)客户端管理邮件，IMAP服务器管理邮件（143），HTTP（80），HTTPS（443）。
2. UDP对应的协议：DNS（53），简单网管协议SNMP（161）；
3. HTTP报文结构
4. 请求报文由请求行、请求头部、空行和请求数据4个部分组成

其中请求行由请求方法字段、URL字段和HTTP协议版本字段3个字段组成，它们用空格分隔；HTTP协议的请求方法有GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT。

请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔。请求头部通知服务器有关于客户端请求的信息。

空行，发送回车符和换行符，通知服务器以下不再有请求头。

请求数据不在GET方法中使用，而是在POST方法中使用。

1. 响应由三个部分组成，分别是：状态行、消息报头、响应正文。

状态行通过提供一个状态码来说明所请求的资源情况。

注：状态代码由三位数字组成，第一个数字定义了响应的类别，且有五种可能取值。

1xx：指示信息--表示请求已接收，继续处理。

2xx：成功--表示请求已被成功接收、理解、接受。

3xx：重定向--要完成请求必须进行更进一步的操作。

4xx：客户端错误--请求有语法错误或请求无法实现。

5xx：服务器端错误--服务器未能实现合法的请求。

1. Http1。1和Http1。0的区别：HTTP/1。0协议使用非持久连接，即在非持久连接下，一个tcp连接只传输一个Web对象，； 2，HTTP/1。1默认使用持久连接。 在持久连接下，不必为每个Web对象的传送建立一个新的连接。HTTP 1。1还提供了与身份认证、状态管理和Cache缓存等机制相关的请求头和响应头。
2. HTTP缓存：减少数据冗余，缓解网络瓶颈，降低原始服务器要求。分为私有缓存（浏览器）和公有缓存（代理）。

处理流程：请求处理-新鲜度检测-服务器在再验证-响应并返回。

1. HTTPS：即HTTP下加入SSL层，用于安全的HTTP数据传输。

和HTTP区别：a。HTTP连接简单，无状态，HTTPS加密传输，身份验证，安全，b。HTTP明文，HTTPS密文（同时用到了对称加密和非对称加密），c。HTTPS需要申请证书和缴费，d。端口不同，HTTPS为443端口。

# 操作系统

1. 一组管理和控制计算机硬件和软件资源，合理的对各类作业进行调度，以及方便用户使用的合集，基本特征是并发性，共享性，虚拟，异步。五大功能为：处理机管理，内存管理，设备管理，文件管理，用户接口。

## 线程调度

一般线程调度模式分为两种——抢占式调度和协同式调度。抢占式调度指的是每条线程执行的时间、线程的切换都由系统控制；协同式调度指某一线程执行完后主动通知系统切换到另一线程上执行。Java使用的线程调度是抢占式调度

## 进程管理

1. 概念：
2. 作业：用户在一次解题或者一个事务处理过程中要求计算机系统所做工作的集合，包括用户程序、所需要的数据和控制命令等，作业由一系列有序的步骤组成。
3. 进程:一个程序在一个数据集合上的运行过程，为了使程序可以并发执行，且对并发程序加以有效控制，最重要的特征是动态性。PCB描述进程当前情况和控制进程运行的全部信息，是进程存在的唯一标志。
4. 线程：进程中的轻型实体，为了减少程序并发执行的时空开销，使OS具有更好的并发性。可并发执行，共享进程资源。
5. 管程：定义了一个数据结构和在该数据结构上的能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据，每次仅允许一个进程在管程内执行某个函数。

注：进程是程序的一次动态执行过程，它对应了从代码加载、执行到执行完毕的一个完整过程，这个过程也是进程本身从产生、发展到消亡的过程 线程是比进程更小的执行单位。一个进程在其执行过程能够产生多个线程，形成多条执行线索。每条线索，即每个线程也有它自身的产生、存在和消亡过程，也是一个动态的概念。

1. 进程和线程的区别：

(1)调度：线程作为独立调度和分派的基本单位，进程作为拥有资源的基本单位。  
(2)并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行  
(3)拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。   
(4)系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。

1. 临界区：　每个进程中访问临界资源的那段程序称为临界区，每次只准许一个进程进入临界区，进入后不允许其他进程进入。

（1）如果有若干进程要求进入空闲的临界区，一次仅允许一个进程进入；

（2）任何时候，处于临界区内的进程不可多于一个。如已有进程进入自己的临界区，则其它所有试图进入临界区的进程必须等待；

（3）进入临界区的进程要在有限时间内退出，以便其它进程能及时进入自己的临界区；

（4）如果进程不能进入自己的临界区，则应让出CPU，避免进程出现“忙等”现象。

1. 进程同步的方式

指多个线程同时访问某资源时，采用一系列的机制以保证同时最多只能一个线程访问该资源。主要有以下几种方式：

a。临界区：通过对多线程的串行化来访问公共资源或者一段代码，速度快，适合控制数据访问。

b。互斥量Mutex：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以可以保证公共资源不会同时被多个线程访问。

注意：互斥对象允许在进程间使用，而临界区只限制与同一进程的各个线程之间使用，但是更节省资源，更有效率。

c。信号量：它允许多个线程同一时刻访问同一资源，但是需要限制同一时刻访问此资源的最大线程数目。信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中PV操作相似。

d。事件（信号）：通过通知操作的方式来保持多线程的同步，还可以方便的实现多线程的优先级比较的操作。使用信号处理器进行控制。

　　注：同步机制遵循的原则：（1）空闲让进；（2）忙则等待（保证对临界区的互斥访问）；（3）有限等待（有限代表有限的时间，避免死等）；（4）让权等待，（当进程不能进入自己的临界区时，应该释放处理机，以免陷入忙等状态）。

1. 进程通信的方式：指进程之间的信息交换

a。信号量， 不能传递复杂消息，只能用来同步，使用PV操作实现。

b。共享存储器系统（存储器中划分的共享存储区）；实际操作中对应的是“剪贴板”（剪贴板实际上是系统维护管理的一块内存区域）的通信方式。

c。 消息传递系统， 首先在一个进程中创建一个消息队列，然后再往消息队列中写数据，而另一个进程则从那个消息队列中取数据。

d。 管道通信系统（管道即：连接读写进程以实现他们之间通信的共享文件（pipe文件，类似先进先出的队列，由一个进程写，另一进程读））。

1. 死锁

一个进程集合中的多个进程因为竞争资源，而造成的互相等待现象。原因在于a。竞争非剥夺性资源和临界资源，b。进程推进顺序非法。

产生死锁的必要条件：

a。互斥条件：资源不能被共享，只能由一个进程使用。

b。请求与保持条件：已经得到资源的进程可以再次申请新的资源。

c。非剥夺条件：已经分配的资源不能从相应的进程中被强制地剥夺。

d。循环等待条件：系统中若干进程组成环路，改环路中每个进程都在等待相邻进程正占用的资源。

1. 处理死锁策略
2. 预防死锁：放弃请求与保持条件，一次性申请和分配全部资源；放弃非剥夺条件，申请不能得到满足时立刻释放已经拥有的资源；放弃循环等待条件，将资源编号，按照申请由低到高的顺序进行。(互斥条件无法被破坏)
3. 避免死锁：将死锁分成安全和不安全状态，实质是避免系统进入不安全状态，如银行家算法。
4. 检测死锁：利用资源分配图进行检测（找到既不阻塞又非独立的进程节点，消去所有边，直到不可化简为止）。
5. 解除死锁：剥夺资源，撤销进程。

死锁定理：当且仅当S状态的资源分配图是不可完全简化的，一定存在死锁。

1. 银行家算法

 最有代表性的避免死锁的算法。银行家算法的基本思想是分配资源之前，判断系统是否是安全的；若是，才分配。在避免死锁方法中允许进程动态地申请资源，但银行家算法在系统在进行资源分配之前(并不是真的不分配，这样就没法做了，只不过是试探性分配，不满足的话再恢复)，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。

注：安全序列是指存在一个进程序列是安全的，不会死锁，安全状态一定是没有死锁发生；不安全状态不存在一个安全序列，不一定导致死锁。

1. 僵尸进程和孤儿进程

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作，孤儿进程没有危害。

僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

僵尸进程的危害：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。 但是仍然为其保留一定的信息(包括进程号，退出状态，运行时间等)。直到父进程通过wait / waitpid来取时才释放。 但这样就导致了问题，如果进程不调用wait / waitpid的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程。

解决办法：通过信号机制，即子进程退出时向父进程发送SIGCHILD信号，父进程处理SIGCHILD信号。在信号处理函数中调用wait进行处理僵尸进程。

1. 生产者消费者模式(JAVA代码)

**class** Producer **implements** Runnable {

**private** **final** BlockingQueue queue；

Producer(BlockingQueue q) { queue = q； }

**public** **void** run() {

**try** {

**while**(**true**) { queue。put(produce())； }

} **catch** (InterruptedException ex) { 。。。 handle 。。。}

}

Object produce() { 。。。 }

}

**class** Consumer **implements** Runnable {

**private** **final** BlockingQueue queue；

Consumer(BlockingQueue q) { queue = q； }

**public** **void** run() {

**try** {

**while**(**true**) { consume(queue。take())； }

} **catch** (InterruptedException ex) { 。。。 handle 。。。}

}

**void** consume(Object x) { 。。。 }

}

**class** Setup {

**void** main() {

BlockingQueue q = **new** ArrayBlockingQueue()；

Producer p = **new** Producer(q)；

Consumer c1 = **new** Consumer(q)；

Consumer c2 = **new** Consumer(q)；

**new** Thread(p)。start()；

**new** Thread(c1)。start()；

**new** Thread(c2)。start()；

}

}

## 内存管理

1. 抖动：在请求分页存储管理中，从主存中刚刚移走某一页面后，根据请求马上又调进该页，这种反复调进调出的现象，称为系统颠簸，也叫系统抖动
2. 抖动解决办法：好的页替换算法；减少运行的进程数；增大内存
3. 分页和分段的区别:

a。页是信息的物理单位。分页的目的是实现离散分配，减少外部碎片，提高内存利用率，页对用户是不可见的。段是信息的逻辑单位。每一段在逻辑上是一组相对完整的信息集合。段对用户是可见的。

b。分页式存储管理的作业地址空间是一维的（页号），而分段式存储管理的作业地址空间是二维的(段号+段内地址)。

c。 页的大小固定且由系统确定，是等长的。而段的长度不定。

注：二者都是离散分配方式，如果分配的基本单位是页，则称为分页存储管理，如果不具备页面对换功能，则称为基本分页存储管理。

1. [逻辑地址、线性地址、物理地址](http://www.cnblogs.com/tureno/articles/3633728.html)

**逻辑地址：**是指由程式产生的和段相关的偏移地址部分。例如，你在进行C语言指针编程中，能读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，是相对于当前进程数据段的地址，不和绝对物理地址相干。只有在Intel实模式下，逻辑地址才和物理地址相等（因为实模式没有分段或分页机制，Cpu不进行自动地址转换）。应用程式员仅需和逻辑地址打交道，而分段和分页机制对你来说是完全透明的，仅由系统编程人员涉及。

**线性地址：**是逻辑地址到物理地址变换之间的中间层。程式代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了线性地址。如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386的线性地址空间容量为4G。

**物理地址：**是CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。

1. 虚拟技术

“虚拟”指通过某种技术把一个物理实体变为若干个逻辑上的对应物，操作系统的虚拟技术主要有时分复用和空分复用：

（1）时分复用

a。 虚拟处理机技术：将一台物理上的处理机虚拟为多台逻辑上的虚拟机，在每台虚拟机上运行一道程序，使每个终端用户或者进程都认为有一个处理机在为自己服务。

b。虚拟设备技术：将一台IO设备虚拟为多台逻辑上的IO设备，变为允许一段时间内同时访问的共享设备。（SPOOLing）。

（2）空分复用

a。虚拟磁盘技术：将一台硬盘虚拟化为多个虚拟磁盘，比如分为1，2，3，4卷，再通过安装程序安装在C，D，E，F四个逻辑驱动器上。

b。虚拟内存技术：指计算机呈现出要比实际拥有的内存大得多的内存量，利用磁盘的空间来存放其他程序，以提高内存利用率。本质上是内存的时分复用，可以使一道程序通过时分复用的方式在远小于它的内存空间中运行。（需要请求调入功能和置换功能）。

基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其余部分留在外存，就可以启动程序执行。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存,然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换出到外存上，从而腾出空间存放将要调入内存的信息。这样，系统好像为用户提供了一个比实际内存大得多的存储器，称为虚拟存储器。

# 数据结构

## 时间复杂度

算法中所有语句的频度之和

## 线性表、栈、队列

1. 基本概念

逻辑结构：对数据之间关系的描述，包括线性结构和非线性结构（图形和树形结构）

物理结构：指数据的逻辑结构在计算机中的标示，包括顺序存储，链式存储，索引存储，散列存储等。

时间复杂度：将算法中基本操作的执行次数作为时间复杂度的度量

关系：O(1)<O(logN)<O(N)<O(NlogN)<O(N2)<O(NK(k>2))<O(2N)<O(N!)<O(NN)

空间复杂度：算法在运行时临时占用的存储空间的大小。

1. 线性表

具有相同特性数据元素的一个有限序列，主要有顺序存储和链式存储两种存储结构

顺序存储：顺序查找次数：（n+1）/2；插入移动元素个数：n/2；删除移动元素个数：(n-1)/2。

链式存储：分为单链表，循环单链表，双链表，循环双链表（带头结点和不带头结点）。在单链表中，删除一个节点必须知道其前驱结点位置，要在某结点后插入一个节点，必须知道此节点位置。

常见题型：

（1）单链表逆置代码：

**Node** reverse(Node head){

Node nextNode；

Node curNode = head；

Node preNode = **null**；

**while** (curNode != **null**) {

nextNode = curNode。next；

curNode。next = preNode；

preNode = curNode；

curNode = nextNode；

}

return preNode；

}

递归版本：

link reverse\_recursive(link list)

{

link head；

if (list==NULL || list->next==NULL)

return list；

head = reverse\_recursive(list->next)；

list->next->next = list；

list->next = NULL；

return head；

}

（2）删除链表中重复元素：（a）设置两个指针，一个内循环，一个外循环。(b)借助hashmap，遍历链表

（3）找出倒数第K个数：设置两个指针，一个先往前走k-1步，然后一起走。

（4）从尾到头输出单链表：先逆置或者利用递归，每访问一个节点，先递归输出后面的节点，再输出自身，或者用栈。

（5）检测一个链表是否有环：设置一个快指针（一次走两步），一个慢指针（一次走一步），检测是否相等。若要找到环入口点，可以在相遇处和起始处分别设一个指针，每次走一步，相遇点就是环入口点。

（6）复杂链表的复制：方法一。先复制原始链表上的每个节点，然后依次遍历设置每个节点的m\_pSibling指针，O(n2)；方法二。空间换时间，第一步不变，将<N，N`>配对信息放到hash表中，然后，根据hash表设置m\_pSibling指针，O(n)，方法三。首先将N`链接在N的后面，然后如果原始链表上的节点N的m\_pSibling指针指向S，则N`的m\_pSibling指针指向S的下一节点S`。

1. 栈

只能在一端进行插入或者删除的线性表，允许进行插入和删除的一端为栈顶。

注：当n个编号元素以某种顺序进栈，且可在任意时刻出栈，所得编号排列数目为N=(1/(n+1))\*(Cn，2n)

1. 队列

只能在一端（队尾）插入，在另一端（队头）删除的线性表。

数组实现（进队和出队都先移动指针）：

队空：qu。rear == qu。front；

队满：(qu。rear + 1) % maxSize = qu。front；

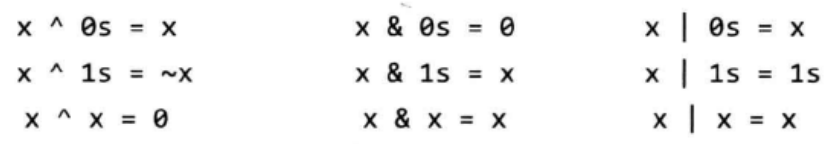
栈和队列的链表实现都可以用单链表。

常见题型：

1. 用两个栈模拟队列操作：入队列则入栈A，出队列时若栈B不为空，则直接弹出栈B元素，若B为空，则弹出栈A中所有元素，压入栈B，再从B弹出。
2. 用两个队列模拟栈操作：入栈:若两个队列都空，入任意一个，若有一个不空，则入其队列；出栈:出非空队列n-1个元素到另一个队列中，然后出队列最后一个元素。（同一时刻至少有一个队列为空）。
3. 汉诺塔问题：先将顶端的N-1个盘子从orgin移到buffer，然后将盘子N移到到destination，最后将N-1个盘子从buffer移动到destination。
4. 排序栈且只允许使用多一个栈结构：将栈元素依次弹出压入另一个栈，如果压入时栈顶元素比其大，则将栈顶元素先移回来，再压入。
5. 队列中取最大值操作：方法一。直接遍历O(n)，方法二。维护一个最大堆，O(1)，但是插入和删除需要O(nlogn)，方法三。用两个栈模拟队列，返回两个栈中较大的最大值即可。而栈中维护一个最大值指针O(1)。

## 位运算

1. 常见位操作



获取第i位的值：



将第i置位1：



将第i位清零：



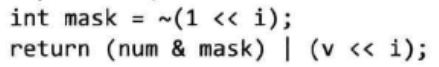
将最高位至i位清0：



将i位至0位清零：



更新第i位的值为v:



1. 判断一个数是否为2的N次方：n>0&&n&(n-1) == 0；
2. 求二进制数中1的个数：给定一个n，每进行一次n = n&(n-1)，count就加1，直到n为0；
3. 给定32位整数M和N，将M插入N之中，从第j位开始，第i位结束：a。将N中j到i之间的值清零，b。将M移位，c。将M和N按位或，合并。
4. 10进制整数转换为2进制，除2取余；10进制小数（0到1之间）转换为2进制，乘2取整；
5. 找出与N的二进制表示中1的个数相同，且大小最接近的两个数：
6. 蛮力法：数出1个数，然后增加或者减小，直到个数相同为止。
7. 位操作法：

取出较大的数：翻转最右边非拖尾的0，该位置为p，数出p右边的1的个数n，先将p右边全部清零，再从最低位开始回填n-1个1；

取出较小的数：翻转最右边非拖尾的1，该位置为p，数出p右边的1的个数n，先将p右边全部清零，再从p-1~0位置开始回填n-1个1；

1. 交换整数x的奇数位和偶数位：



## 数组

1. 找出数组的最大值和最小值：维持两个变量min，max，然后一次取2个元素进行比较，大的和max比，小的和min比，或者分治法。或者先对数组预处理，将较大的数放在奇数上，较小的数放在偶数上，比较1。5N次。或者分治。
2. 找出数组中的第二大数：维持两个变量max，max2，然后依次比较。
3. 数组中出现次数超过一半的数字：基于快速排序思想或者不能改变数组的情况下使用两个值，一个统计次数，一个保存数字，如果次数为0，更新数字，要找的值即是最后一次把值置为1对应的数字。
4. 找出数组中第k小的数或者前k小的数：排序或者利用快速排序的思想，先选一个数作为枢纽，小的放左边，大的放右边，此时，如果此数正好在k-1的位置，找到，否则比较后在前半部分或者后半部分找。
5. 找出连续数组中唯一的重复元素：对数组求和然后减去1到N-1之和。
6. 数组中其他数字都出现偶数次，找出只出现1次的数字：对整个数组挨着异或即可。找出两个只出现1次的数字：先全部异或，然后得到结果之后找到第一个为1的位位置，然后根据该位是否为1进行分组，再对分组进行异或即可。

奇偶都适用的方法：对数组中每个数的各个位相加，统计各个数的各个位的1出现的次数之和，最后对次数取余，若不为0则相应位数为1。

1. 求两个有序整形数组的交集：如果数组长度相当，则同时从两个数组的头部开始遍历；如果相差很大，则依次取小数组中的数在大数组中进行二分查找。且每查找一次就可以缩短二分查找的边界一回。
2. 把数组排成最小的数：先求全排列然后比较O(n!)，或者转化成字符串，然后定义一种新的排序规则，然后对字符串数组排序，依次输出即可O(nlogn)。
3. 求丑数（只包含因子2、3、5的数）：方法1。逐个判断每个数是不是丑数（除以每个2、3、5）；方法2。创建一个数组保存已经找到的丑数，设置3个指针，分别乘2、3、5然后比较大小，将最小的放入数组，更新指针即可。
4. 数字在排序数组中出现的次数：方法1。二分查找，找到之后左右搜O(n)；方法二。二分查找，找到之后确认是不是第一次出现或者最后一次出现，不是的话继续二分查找，直到找到第一次出现和最后一次出现的索引。
5. 最大子数组之和：动态规划的方法，要不在数组左边，要么在数组右边，要么在数组中间（包含数组中间左右起始元素），最佳方法如下：

**int** maxSubArray(**int**[] arr){

**int** maxSum = 0；

**int** nSum = 0；

**for** (**int** i = 0； i < arr。length； i++) {

**if** (nSum < 0) {

nSum = arr[i]；

}

**else** {

nSum += arr[i]；

}

**if** (nSum > maxSum) {

maxSum = nSum；

}

}

**return** maxSum；

}

1. 最大子数组之和（二维）：枚举每一个矩形区域的值或者枚举上下边界，然后用一维情况下的方法确定左右边界。
2. 找出数组重复元素最多的数：空间换时间法，定义int count[max]，然后遍历数组，执行count[arr[i]]++；或者引入一个Map。
3. 求数组中两两相加等于N的组合：蛮力法，双重循环遍历，或者先排序，然后设置两个下标一头一尾靠近，或者借助散列表。
4. 求数组中和为s的连续正数序列：设置两个指针，一个指向数组开头，一个指向开头后一位，然后和s比较，不断移动指针即可，直到small指针到（1+s）/2。
5. 数组循环右移N位：先逆序子数组0到arr。length-N-1；再逆序子数组arr。length-N到arr。length-1；最后整体逆序，时间复杂度为O（n），（循环左移类似）
6. 求反序对的个数:蛮力法或者分治归并（先统计出子数组内部的逆序对数目，再统计出两个相邻子数组之间的逆序对数目）。
7. 打印数组排列情况：如果无特殊要求，用递归实现，若要求某些点不相连，先转化成无向连通图。
8. 打印数组所有排列：

**void** permutation(**char**[] arr， **int** index， **int** size) {

**if** (index == size) {

**for** (**int** i = 0； i < arr。length； i++) {

System。***out***。print(arr[i] + " ")；

}

System。***out***。println()；

} **else** {

**for** (**int** i = index； i < size； i++) {

**if**(i == index || arr[i] != arr[index]) {

*swap*(arr， i， index)；

*permutation*(arr， index+1， size)；

*swap*(arr， i， index)；

}

}

}

}

1. 打印数组所有组合情况：

**void** combiantion(**char** chs[]){

**if**(chs。length == 0) **return** ；

Stack<Character> stack = **new** Stack<Character>()；

**for**(**int** i = 1； i <= chs。length； i++){

*combine*(chs， 0， i， stack)；

}

}

**void** combine(**char** []chs， **int** begin， **int** number， Stack<Character> stack){

**if**(number == 0){

System。***out***。println(stack。toString())；

**return** ；

}

**if**(begin == chs。length){

**return**；

}

stack。push(chs[begin])；

*combine*(chs， begin + 1， number - 1， stack)；

stack。pop()；

*combine*(chs， begin + 1， number， stack)；

}

1. 求数组中一个数字减去右边子数组中的一个数字的最大差值：a。蛮力法；b。二分法：三种情况，被减数和减数都在左边，都在右边，或者被减数在是左边的最大值，减数是右边的最小值。c。动态规划:diff=max(diff，max-a[i])；max=max(max，a[i])。
2. 从大小为n的数组中选出m个整数：要求被选中概率相同：从前n-1个元素中随机选出m个元素，然后在0-n之间随机选择一个数k，如果小于m，则将第n个元素插入k的位置。
3. 完美洗牌：先打乱前n-1个元素的次序，然后取出第n个元素，与数组中的元素随机交换。
4. 给定整数数组：找到索引m和n，使得m和n之间的元素排好序之后，整个数组就有序，且n-m最小：找出位于开头和末尾的最长递增子序列。然后找到中间序列的最小值和左边子序列比较，最大值和右边子序列比较。即可确定m和n。
5. 给定有序数组，查找魔术索引a[i]=i：蛮力法，或者二分查找（分为有重复元素和无重复元素两种情况）。
6. 打印n对括号的全部有效组合：从头开始构造字符串，逐一加入左括号和右括号，只要左括号没有用完，就加入，只要没有语法错误，就如右括号。
7. 找出数组中有一个元素的数量超过数组一半的数：先排序，取中位数，或者每次删除不同的两个数，把问题变小。
8. 寻找最近点对：找出两两的差值O(n2)，或者先排序（一维），或者分治。O（nlogn）
9. 将元素为2n的整数数组拆分成个数为n的两个数组，使两个子数组和尽可能接近：排序，拆分S1={a1，a3，。。}， S2={a2，a4，。。}，然后依据差值交换。或者动态规划，将任务分解成2N步，Sk-1 表示前k个元素中任意i个的和，Sk就为Sk-1并上其加a[k]的值，只考虑取值小于SUM/2的情况。（不适用于SUM大）
10. 找到排序后有效的旋转数组的最小数字：采用二分查找，设置两个头指针和尾指针表示查找范围，通过中间值和头尾比较逐步缩小范围。存放重复元素的情况下，如果头尾和中间值都相等，则必须从头尾顺序查找。
11. 调整数组顺序，使奇数位于偶数前面：从头扫描，碰到偶数就拿出放到最后，或者设置头指针和尾指针，然后相向移动，做判断。
12. 顺时针打印二维数组：循环条件：columns>startX\*2；rows>startY\*2；然后需要判断最后一圈是否只有一行，一列等。

## 字符串

1. 实现字符串反转：先整体转一下，再反转每一个单词，或者分割用栈。
2. 判断字符串是否是相同字符构成：先排序，再比较，或者空间换时间，遍历第一个字符串，在相应位置加1，遍历第二个字符串，然后在相应位置减1，看最后数组元素是否都为0；
3. 删除字符串中重复字符：空间换时间，或者借用hashmap。
4. 统计单词个数：设置一个word=0，count=0；当遇到空格表示单词开始，word=0，当word=0时让word=1，count++；
5. 输出字符串组合：递归，遍历字符串，每一个字符取或者不取即可。或者用二进制位来表示。
6. 给定字符串s和较短字符数组T，根据T中的每一个字符串，对s搜索：先创建s的后缀树，然后只需要在这棵后缀树中搜索查找即可。
7. 给定一组单词，找出最长单词，且其必须有其他单词组合而成：将单词按照长度排序，从最长单词到最短单词，将每个单词分成所有可能的两半，然后检查左右部分是否在列表中。其中，检查右半部分是递归的进行。
8. 输出字符串所有排列：

**void** printAllArray(String s) {

*printAllArray*(s， "")；

}

**void** printAllArray(String s， String n) {

**if** (s。length() == 0) {

System。***out***。println(n)；

} **else** {

**for** (**int** i = 0； i < s。length()； ++i) {

*printAllArray*(s。substring(1)， n + s。charAt(0))；

s = s。substring(1) + s。charAt(0)；

}

}

}

## 树和二叉树

1. 基本概念

二叉树的定义：或为空，或只有根节点，或只有左子树和右子树，或者都有（5种基本形态）。

对于一棵非空二叉树，度为0的节点总是比度为2的节点多1个。

给定n个节点，能组成(1/(n+1))\*(Cn，2n)种不同的二叉树。

满二叉树：所有分支节点都有左右孩子，且叶子节点只集中在最下层

完全二叉树：由一个满二叉树从右至左，从下至上，挨个删除节点所得。

注：满二叉树只有度为2和0的节点，完全二叉树至多只有一个度为1的节点。完全二叉树的深度为logn+1，向下取整。

二叉排序树：或者为空，或者若左子树不空，则左子树的所有关键字均小于根关键字的值；若右子树不空，则右子树的所有关键字均大于根关键字的值；

注：排序二叉树的删除：若叶子节点，直接删除；若只有左子树或右子树，直接删除，再将其子树接到双亲结点之上。若左右子树均有，则以其中序遍历的前驱替代，用左子树中最大的那个值（即最右端S）代替，重接其左子树 。

平衡二叉树（AVL）：左右子树都是平衡二叉树，且左右子树高度之差不超过1；

注：二叉树的平衡：首先找到失去平衡的最小子树，然后调整次子树即可，LL型和RR型找儿子，旋转一次，LR型和RL型找孙子，旋转两次；

其中红黑树就是平衡二叉排序树，而B树和B+树则是多路平衡查找树。

哈夫曼树：又称最优二叉树，是带权路径最短的二叉树（树中所有叶子节点的带权路径长度子和），哈夫曼编码是长度最短的前缀编码。

1. 二叉树的遍历

先序遍历：访问根，先序遍历左子树，先序遍历右子树。

中序遍历：中序遍历左子树，访问根，，中序遍历右子树。

后续遍历：后序遍历左子树，后序遍历右子树，访问根，

层次遍历：先将二叉树头结点入队，循环内，出队访问，若有左子树或者右子树，先入队左子树，再入队右子树。

注：中序遍历和任一其他遍历可完全恢复一棵唯一的二叉树。

//非递归先序遍历

**void** preOrder(TreeNode root){

MyStack<TreeNode> myStack = **new** MyStack<>()；

TreeNode temp = root；

**while** (temp != **null** || !myStack。isEmpty()){

**while** (temp != **null**) {

System。***out***。println(temp。data)；

myStack。push(temp)；

temp = temp。left；

}

temp = myStack。pop()；

temp = temp。right；

}

}

//非递归中序遍历

**void** inOrder(TreeNode root){

MyStack<TreeNode> myStack = **new** MyStack<>()；

TreeNode temp = root；

**while** (**null** != temp || !myStack。isEmpty()) {

**while** (**null** != temp) {

myStack。push(temp)；

temp = temp。left；

}

temp = myStack。pop()；

System。***out***。println(temp。data)；

temp = temp。right；

}

}

//非递归后续遍历

**void** postOrder(TreeNode root){

MyStack<TreeNode> myStack = **new** MyStack<>()；

TreeNode temp = root；

TreeNode pre = **null**；

**while** (**null** != temp || !myStack。isEmpty()) {

**while** (**null** != temp) {

myStack。push(temp)；

temp = temp。left；

}

temp = myStack。peek()；

**if** (**null** != temp。right || temp。right == pre) {

System。***out***。println(temp)；

myStack。pop()；

pre = temp；

temp = **null**；

}

**else** {

temp = temp。right；

}

}

}

1. 常见题型
2. 根据先序遍历和中序遍历确定一颗二叉树：先找到二叉树的根节点（先序遍历第一个节点），然后找到其在中序遍历的位置，左边就是左子树，右边就是右子树，如果根节点左边或右边为空，那么该方向子树为空，如果根节点左边和右边都为空，那么就是叶子节点，递归遍历左右子树即可。
3. 检查一棵树是否平衡：判断root是否为null，是直接返回0，递归向下检查每棵子树的高度；如果为-1，表示不平衡，直接返回-1，否则返回左右子树高度值的最大值+1；
4. [二叉树和最大的子树](http://blog.csdn.net/u010590166/article/details/17658809)：采用自底向上的计算。先计算左右子树总和值，用左右子树的总和加上当前节点值，如果当前总和大于最大值，则更新最大值，同时将最大子树根节点更新为当前根。简单说，就是后序遍历。
5. 检查一棵树是否为二叉排序树：利用一个静态变量记录上次访问的值，判断root是否为null，是直接返回true，否则检查左子树是否为二叉排序树，否则判断当前节点值和上次访问值的大小，否则检查右子树是否为二叉排序树。解法二：最大最小法，设置一个范围，检查节点数据是否在此范围内，如果在，则更新范围值，递归的检查左右子树。
6. 给定有序数组，创建高度最小的二叉查找树：将数组中间元素插入数组，将数组左右半部分元素各插入树中，递归进行。三个参数（arr，start，end）。
7. 查找二叉查找树指定节点的下一个节点：判断该节点是否为空，为空则直接返回，否则若该节点有右子树，则返回右子树的最左端节点，否则向上遍历，直到找到一个父节点且父节点的左子树为该节点（或者父节点）。
8. 查找二叉树中两个节点的共同祖先：首先检查节点是否在二叉树中，否则检查两个节点是否正好有一个是根节点，否则检查两个节点是否在同一边，不在则根节点就是共同祖先，否则往其所在一遍继续递归，直到找到共同祖先。
9. 判断一棵树是否是另一棵树的子树：解法1，创建字符串表示两棵树的前序遍历和中序遍历，如果都是子串则true；解法2，遍历较大的树，当其节点和较小树匹配时，调treeMatch分析比较两棵子树是否相同，具体，先判断较小树是否为null，如果为null则返回True，否则调用比较函数，比较时判断大树是否为空，为空则返回false，否则比较节点如果相等就调用treeMatch比较两棵子树是否相等，否则再递归比较左右子树是否匹配；treeMatch函数若两者都空，则true，若有一个null或者数据不相等，则false，否则返回左右子树继续比较的结果交。
10. 将二叉排序树转换成双向链表，要求顺序不变，方法一。递归的构建左右子树的双向链表，然后合并，返回链表（多一个链表数据结构表头表尾）。方法二。不借助额外的数据结构，只返回表头，然后根据合并时根据表头找表尾。
11. 树中两个节点的公共祖先：如果是二叉排序树，则从根节点开始和两个输入节点进行比较，然后根据结果递归遍历左子树或者右子树；如果是普通的树，且节点有指向父节点的指针时，转化为求链表的第一个公共节点问题。
12. 打印二叉树的所有路径：

**void** printWay(Node root， **int**[] path， **int** size){

**if**(root == **null**){

**return** ；

}

path[size++] = root。data；

**if**(root。left == **null** && root。right == **null**){

**for**(**int** i=0；i<size；i++){

System。***out***。print(path[i] + " ")；

}

System。***out***。println()；

}**else**{

*printWay*(root。left， path， size)；

*printWay*(root。right， path， size)；

}

}

## 图

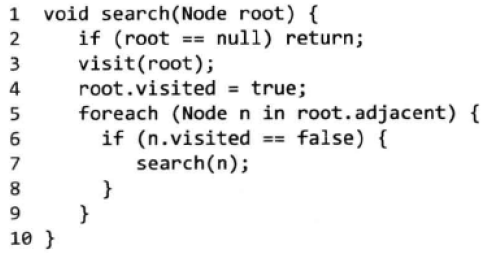
1. 基本概念：
2. 有向图，无向图，路径：相邻顶点序偶所构成的序列，路径长度指路径上边的数目；回路：第一个顶点和最后一个顶点相同的路径。
3. 连通图（无向图），强连通图（有向图）：任意两个顶点之间都连通的图（连通图至少要有n-1条边，强连通图至少需要n条边）。
4. 度：无向图中与顶点V相邻的边的条数；有向图中度为顶点入度（指向V的边数）和出度（V发出的边）之和。
5. 完全图：无向图中，最多有（1/2）\*n\*(n-1)条边的图，有向图中，最多有n\*(n-1)条边的图。
6. 生成树：含有图中全部n个顶点，但只有足以构成一棵树的n-1条边，是极小连通子图（再任意加一条边就会有环）；最小生成树：在带权图中，所有n-1条边的总权值最小的生成树。
7. 图的存储:
8. 邻接矩阵:顺序存储，表示顶点之间相邻关系的矩阵，A[i][j]=1表示顶点i和顶点j相邻，A[i][j]=0表示顶点i和顶点j不相邻（对角元为0）。

对于无向图，邻接矩阵对称，1的个数是图中总边数的2倍，矩阵中第i行（列）的元素之和为顶点i的度。

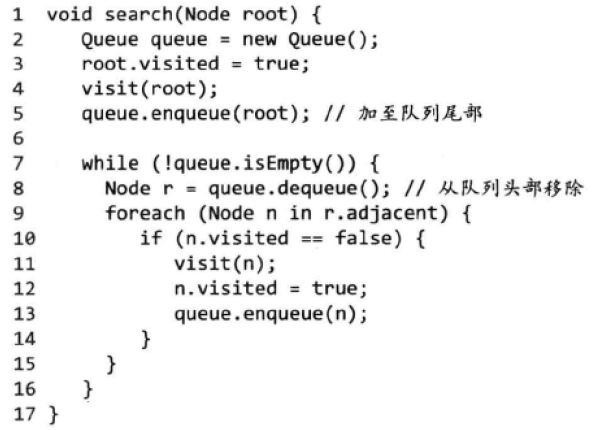
对于有向图，1的个数等于图中总边数，矩阵中第i行的元素之和为顶点i的出度。第i列的元素之和为顶点i的入度。

注意：对于有权图，边的权值不是1，若i和j不邻接，则权值为无穷大。

1. 邻接表：链式存储，为每一个顶点建立一个单链表，用于存储相邻节点。
2. 图的遍历：
3. 深度优先搜索：首先访问出发点V，并标记为已访问，然后选择与V相邻的任一顶点W，访问并标记，再选择与W相邻的点访问，直到遍历完。



1. 广度优先搜索：首先访问出发点V，然后选取与V相邻的全部顶点w1，w2，…访问，再依次访问w1，w2，…相邻的全部顶点，直到遍历完。



注意：DFS和BFS的时间复杂度都是O(n+e)；

1. 最小生成树算法：
2. Prim算法：从任意顶点出发，将其作为根树，选择与树相邻的所有连接边中权值最小的边，然后将其并入树中，再从与树相邻的所有边中选择权值最小且不能构成环的边，直到所有顶点均被并入为止。（选择n-1次）

注：时间复杂度为O（n2），与顶点有关，与边数无关，适用于稠密图；

1. Kruskal算法：每次从候选边中选择权值最小的边且不与树中任意节点构成环的边，将其并入树中，直到所有边都被检测完。（选择n-1次）

注：时间复杂度为O（eloge），与边数有关，与顶点数无关，适用于稀疏图。

注意: Prim算法和Kruskal算法都是以无向图而言，如果图中所有边的权值均不相等，则最小生成树唯一。

1. 最短路径（带权图中总权值最小的路径）：
2. Dijkestra算法：单源最短路径问题，设G=(V，E)是一个带权有向图，把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 ， 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了），第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。
3. Floyd算法：全源最短路径问题，从任意节点i到任意节点j的最短路径不外乎2种可能，1是直接从i到j，2是从i经过若干个节点k到j。所以，我们假设Dis(i，j)为节点u到节点v的最短路径的距离，对于每一个节点k，我们检查Dis(i，k) + Dis(k，j) < Dis(i，j)是否成立，如果成立，证明从i到k再到j的路径比i直接到j的路径短，我们便设置Dis(i，j) = Dis(i，k) + Dis(k，j)，这样一来，当我们遍历完所有节点k，Dis(i，j)中记录的便是i到j的最短路径的距离。
4. 常见题型：
5. 查找两个节点之间是否存在一条路径：利用DFS或者BFS遍历即可；

## 查找与排序

1. 查找方法：
2. 顺序查找，从表的一端开始，顺序扫描线性表，依此与关键字进行比较。算法简单，适用面广，对表结构无要求，同时适用于顺序和链式存储，但是当n很大时查找效率低下，时间复杂度O(n)。
3. 折半查找，先确定查找区间，逐步缩小查找范围，直到找到或者找不到为止。效率高，但是只适用于有序的顺序存储结构表。时间复杂度O(logn)。

**int** binary(**int**[] a， **int** data){

**if** (**null** == a) {

**return** -1；

}

**int** low = 0；

**int** high = a。length - 1；

**int** mid；

**while** (low <= high) {

mid = (low + high) / 2；

**if** (a[mid] == data) {

**return** mid；

}

**else** **if** (a[mid] > data) {

low = mid + 1；

}

**else** {

high = mid - 1；

}

}

**return** -1；

}

1. 索引查找，分块查找，需要建索引表（包含关键字和指针项，有序），先确定待查记录所在的块，再在块中查找。
2. 哈希查找：根据给定的关键字计算其在表中的位置，时间复杂度O(1)。
3. 直接插入：将待排序的每一个记录依次插入到已排好序的序列中，直到全部排好。算法简单，适用于序列基本有序的情况，时间复杂度O(n2)（最好情况O(n)），空间复杂度O(1)，稳定排序。

**void** insertSort(**int**[] a) {

**if** (**null** == a) {

**return**；

}

**for** (**int** i = 0， temp， j； i < a。length； i++) {

temp = a[i]；

j = i；

**while** (j > 0 && a[j-1] > temp) {

a[j] = a[j-1]；

j--；

}

a[j] = temp；

}

}

注：折半插入为直接插入的改进，适用于n很大，时空复杂度同直接插入。

1. 简单选择：从头至尾扫描序列，找出最小的和第一个交换，依次类推，直到有序。比较次数和初始序列无关，时间复杂度为O(n2)，空间复杂度为O(1)，不稳定。

**void** selectSort(**int**[] a){

**if** (**null** == a) {

**return**；

}

**for** (**int** i = 0， temp； i < a。length； i++) {

**for** (**int** j = i + 1； j < a。length； j++) {

**if** (a[i] > a[j]) {

temp = a[i]；

a[i] = a[j]；

a[j] = temp；

}

}

}

}

1. 冒泡排序：依次将相邻各元素进行比较，一趟完成之后，最末端即存在最大元素，直到有序，适用于序列基本有序的情况，时间复杂度O(n2)（最好情况O(n)），空间复杂度O(1)，稳定排序。

**void** bubbleSort(**int**[] a){

**if**(**null** == a){

**return**；

}

**for** (**int** i = a。length， temp； i > 0 ； i--) {

**for** (**int** j = 0； j < i - 1； j++) {

**if** (a[j] > a[j+1]) {

temp = a[j]；

a[j] = a[j+1]；

a[j+1] = temp；

}

}

}

}

1. 希尔排序：缩小增量排序，将待排序序列分割成若干子序列分别进行直接插入排序，一般增量取互质素数，最后一个为1，时间复杂度为O(nlogn)，空间复杂度为O(1)，不稳定排序。

**void** shellSort(**int**[] a){

**if** (**null** == a) {

**return**；

}

**int** temp；

**int** j；

**for** (**int** incre = a。length / 2； incre > 0；incre /= 2) {

**for** (**int** i = incre； i < a。length； i++) {

temp = a[i]；

**for** (j = i； j >= incre； j -= incre) {

**if** (a[j-incre] > temp) {

a[j] = a[j-incre]；

}

**else** {

**break**；

}

}

a[j] = temp；

}

}

}

1. 快速排序：通过一趟排序，将待排序记录分割成两部分，一部分的关键字均比另一部分小，再分别对这两部分进行快速排序。适用于待排序列完全无序的情况，时间复杂度为O(nlogn)（最坏情况为O(n2)），空间复杂度为O(logn)，不稳定排序。

**void** quickSort(**int**[] a， **int** low， **int** high){

**if** (low >= high) {

**return**；

}

**int** i = low；

**int** j = high；

**int** temp = a[i]；

**while** (i < j) {

**while** (i < j && a[j] >= temp) {

j--；

}

**if** (i < j) {

a[i++] = a[j]；

}

**while** (i < j && a[i] <= temp) {

i++；

}

**if** (i < j) {

a[j--] = a[i]；

}

}

a[i] = temp；

*quickSort*(a， low， i - 1)；

*quickSort*(a， i + 1， high)；

}

注意：单链表的快速排序：需要两个指针p和q，这两个指针均往next方向移动，移动的过程中保持p之前的key都小于选定的key，p和q之间的key都大于选定的key，那么当q走到末尾的时候便完成了一次支点的寻找。

1. 二路归并排序：将序列划分成越来越小的半子表，再对半子表排序，最后递归地将排序好的半子表合并成越来越大的有序序列。与初始序列无关，时间复杂度O(nlogn)，空间复杂度O(n)，稳定排序。

**void** mergeSort(**int**[] a) {

**if** (**null** != a) {

*mergeSort*(a， 0， a。length - 1)；

}

}

**void** mergeSort(**int**[] a， **int** low， **int** high) {

**if** (low < high) {

**int** mid = (low + high) / 2；

*mergeSort*(a， low， mid)；

*mergeSort*(a， mid + 1， high)；

*merge*(a， low， mid， high)；

}

}

**void** merge(**int**[] a， **int** low， **int** mid， **int** high){

**int** i，j，k；

**int** n1 = mid - low + 1；

**int** n2 = high - mid；

**int**[] left = **new** **int**[n1]；

**int**[] right = **new** **int**[n2]；

**for** (i = 0； i < n1； i++) {

left[i] = a[low + i]；

}

**for** (i = 0； i < n2； i++) {

right[i] = a[mid + i + 1]；

}

**for** (i = low，j = k = 0； j < n1 && k < n2 ； i++) {

**if** (left[j] > right[k]) {

a[i] = right[k++]；

}

**else** {

a[i] = left[j++]；

}

}

//无需复制数组右半部分的剩余元素

**while** (j < n1) {

a[i++] = left[j++]；

}

}

注：快速排序和归并排序都是基于“分而治之”思想。不同点在于进行的分组策略不同，快速排序以基准值分组，归并排序将数组前一半作为一组，后一半作为一组，合并策略也不同，归并排序还需要对两个有序的数组根据大小合并。

1. 堆排序：分为大堆顶和小堆顶，将记录看做一棵顺序存储的完全二叉树，通过不断调整使得其变成符合堆定义的完全二叉树，最后将根与最后一个元素交换，依此类推，直到调整的堆中只剩一个元素为止。

调整过程：从无序序列确定的第一个非叶结点开始，从右至左，从下至上，对每个结点进行调整，直到得到堆。适用于记录很多的情况，时间复杂度为O(nlogn)，空间复杂度为O(1)，不稳定排序。

**void** heapSort(**int**[] a){

**if** (**null** == a || a。length <= 1) {

**return**；

}

**int** len = a。length；

**for** (**int** i = len/2 - 1； i >= 0； i--) {

*maxHeap*(a， len， i)；

}

**for** (**int** i = len - 1； i > 0； i--) {

*exchangeElements*(a， i， 0)；

*maxHeap*(a， i， 0)；

}

}

**void** maxHeap(**int**[] a， **int** len， **int** pos){

**int** left = 2 \* pos + 1；

**int** right = 2 \* pos + 2；

**int** largest = pos；

**if** (left < len && a[left] > a[pos]) {

largest = left；

}

**if** (right < len && a[right] > a[largest]) {

largest = right；

}

**if** (largest != pos) {

*exchangeElements*(a， pos， largest)；

*maxHeap*(a， len， largest)；

}

}

1. 基数排序

不需要比较关键字的大小，它是根据关键字中各位的值，将要排序的[元素分配](http://baike.baidu.com/view/2914896.htm)至某些“桶”中，通过对排序的N个元素进行若干趟“分配”与“收集”来实现排序。

**void** radixSort(**int**[] number， **int** d) {

**int** k = 0；

**int** n = 1；

**int** m = 1；

**int**[][] temp = **new** **int**[10][number。length]；

**int**[] order = **new** **int**[10]；

**while** (m <= d) {

**for** (**int** i = 0； i < number。length； i++) {

**int** lsd = ((number[i] / n) % 10)；

temp[lsd][order[lsd]] = number[i]；

order[lsd]++；

}

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

**if** (order[i] != 0)

**for** (**int** j = 0； j < order[i]； j++) {

number[k] = temp[i][j]；

k++；

}

order[i] = 0；

}

n \*= 10；

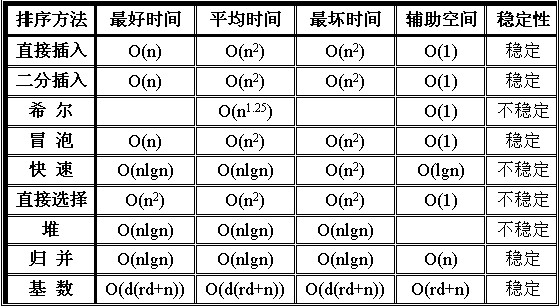
k = 0；

m++；

}

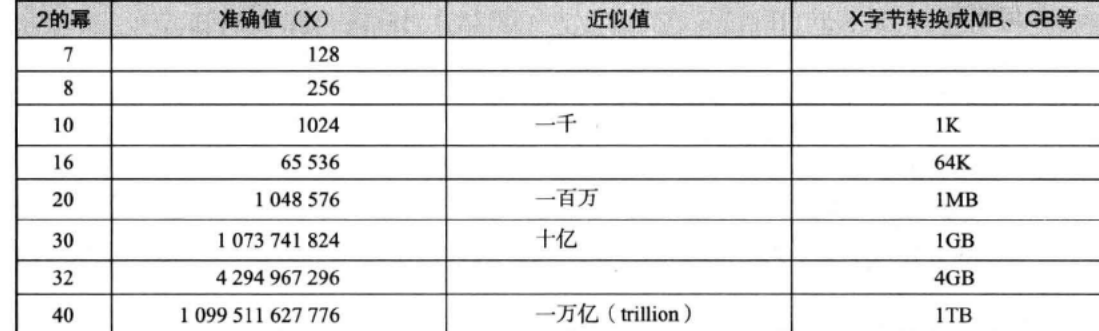
}

注：排序速度：快速排序 > 希尔排序 > 堆排序。先问清楚应用环境和约束条件。



1. 常见题型
2. 合并有序数组，其中A有足够的空间容纳B，从A和B的末端元素开始比较，将最大的元素放到A的末端。
3. 给定MxN矩阵，每一行每一列都是有序的，查找某元素，在每一行上用折半查找，时间复杂度O(mlogn)；从矩阵右上角开始，往左或者往下移动即可。

## 存储限制和海量数据



1. 步骤：

a。 大胆假设没有限制的情况；

b。 回到实际，解决数据的拆分问题；

c。解决问题。

1. 拆分数据的策略 ：
2. 按出现顺序，先存放在当前机器，满了之后再存放在其他机器，不会造成资源浪费，但是查找表复杂，巨大。
3. 随机存储，随机划分数据，再实现一个查找表，好处在于更好的负载均衡。
4. 按散列值，根据数据挑选某种键，利用散列函数得到散列值，将散列值除以机器数量得到余数，将数据存储在对应的机器上；本质是随机的，不用创建查找表，但是会出现存储不均匀的问题；
5. 按实际值，根据数据的信息分类，存放在不同的机器中，可以降低系统延迟。
6. 基本方法：
7. Hash法：用于快速存取、统计某些数据、将大量数据进行分类，如提取某日访问网站次数最多的IP地址。
8. 位图法：使用位向量数据来表示某些元素是否存在，空间换时间，适用于数据状态不多，最大值已知的情况，用于判断数据是否重复和存在。
9. 布隆过滤器：用于检测一个元素是否属于一个集合，与Hash函数结合使用，以牺牲正确性为前提（判断不属于时，一定不属于，判断属于时，不一定属于），来换取空间与时间效率的提高，只能插入元素，不能删除元素，可以用来实现数据字典、进行数据判重、集合求交集等。
10. 倒排索引法：反向索引，被用来存储在全文搜索下某个单词在文档中的存储位置和映射，即单词指向包含他的文档。（正向索引用来存储每个文档的单词列表，文档指向其包含的单词），一般用于查询哪些文件包含某单词。
11. 外排序法：大数据的排序和去重，将待排序记录存储在磁盘上，一般采用归并排序，会消耗大量的IO，效率不高。
12. Trie树：单词查找树（trie）：采用26叉树进行组织，每个结点存储一个字母，根据字典中的单词构建字典树，整棵树的每条路径自上而下表示一个单词。适用于数据量大、重复多、数据种类小的情况。如前缀字符串问题。

在Trie树上进行查找的过程为：从根结点出发，沿给定值相应的指针逐层向下，直至叶子结点，若叶子结点中的关键字和给定值相等，则查找成功。若分支结点中和给定值相应的指针为空，或叶结点中的关键字和给定值不相等，则查找不成功。在Trie树上易于进行插入和删除，只是需要相应地增加和删除一些分支结点。把沿途分支结点中相应的指针域置空，再把其分支结点中的num-1，然后删除叶子结点。当分支结点中num域的值减为1时，便可删除。

1. 双层桶法：借用桶排序的基本思想，适用于关键字取值范围较小的场合，一般用于寻找第k大的数，中位数，重复或者不重复的数等。
2. 经典实例分析：
3. top K 问题：查找最大的k个数（简单排序法、借用容器局部淘汰法、分治法、hash法去重复、最小堆法）；查找出现次数最多的k个数：先利用hash映射将大文件映射为多个小文件，然后在每个小文件提取top k，最后合并统计最终的top k。
4. 重复问题：找出重复元素或者去重，采用位图法或者hash映射先分组数据，然后在分组中寻找。
5. 排序问题：直接排序、数据库排序（将文件导入数据库，利用数据库进行索引排序后提取到文件）、hash映射+分治法、位图法（适合数据不重复）。
6. 常见题型：
7. 查找包含某一组词的文件：首先预处理每个文件，并创建一个散列表索引，散列表将单词映射到含有此单词的一组文件。然后，按字母顺序划分关键字，按关键字划分散列表，按字母顺序遍历所有关键字，将数据尽可能存储在一台机器上，装满之后再放在另一台机器上，最后将字符串排序，给每一台机器发送查找相应数据的请求，最后对返回结果做交集运算。
8. 搭建某种服务，需要给大量客户端提供信息：方案一：将数据保存在纯文本中，客户端用过FTP服务器下载，易于维护，查看和备份，但是需要复杂的文件解析，新增数据时，会打乱客户端解析机制。方案二：使用SQL数据库，为客户端提供一种简单高效的数据查询方式，可以提供数据回滚、备份和安全保障。但是需要客户端多一层数据访问实现。方案三，XML，容易分发和识别，很多支持XML解析的库，添加新节点和数据时不会打乱客户端的解析。但是查询时时必须解析整个文件，且必须向客户端发送所有信息，尽管可能只需要很少一部分。
9. 设计大型社交网站：构造一个无向图，每个人看做一个节点，有连线表示朋友关系。Person节点保存朋友ID的列表，针对ID找出机器所在的位置；找到对应的机器，得到朋友数据，需要一个Server类包含所有机器的列表。寻找朋友时，进行广度优先搜索。
10. 设计网络爬虫程序，避免陷入死循环：创建一个散列表，访问页面v过后，就将hash[v]置为true，使用广度优先搜索抓取网站，每访问一个页面，就收集其所有链接，插入到队列尾部，若已访问过，则忽略。不同页面的定义：根据内容和URL生成页面特征码，只适用于企业内部网。
11. 从海量数据中检测重复数据：方案一。存储在磁盘文件中，第一次扫描按照数据散列值分组数据，使每一组能够加载进内存，第二次扫描，创建数据散列表，找出重复项。方案二：使用多台机器，将数据发往多台机器而不是文件，可以并行执行，但是要考虑机器故障和系统复杂性。
12. 给定输入文件找出其中不存在的数：如果内存放得下，创建位向量数组，第一遍扫描文件，设置相应位为1，第二遍扫描，找出第一个值为0的索引。如果内存放不下，将全部整数划分成同等大小的区块，第一遍扫描时，先将相应区块的数加1，然后数一数每个区块的大小，找到少了数字的区块，第二遍扫描，采取位向量的做法，忽略范围之外的数字。如果内存更小，则增加扫描次数。（Java内置了BitSet位向量类。）

## 算法设计技巧

1. 贪心算法

总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。如单源最短路经问题，最小生成树问题，活动时间安排，区间覆盖等。

1. 分治算法

基本思想是将一个规模为N的问题分解为K个规模较小的子问题，这些子问题**相互独立**且与原问题性质相同。求出子问题的解，就可得到原问题的解。

分治法解题的一般步骤：

（1）分解，将要解决的问题划分成若干规模较小的同类问题；

（2）求解，当子问题划分得足够小时，用较简单的方法解决；

（3）合并，按原问题的要求，将子问题的解逐层合并构成原问题的解。

1. 动态规划

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。如背包问题。

  与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

   使用动态规划求解问题，最重要的就是确定动态规划三要素：（1）问题的阶段 （2）每个阶段的状态（3）从前一个阶段转化到后一个阶段之间的递推关系。

1. 回朔法

在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。如0-1背包问题，八皇后问题，m着色问题，旅行售货员问题，圆排列问题等。

若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。步骤：（1）针对所给问题，确定问题的解空间，（2）确定结点的扩展搜索规则（3）以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

注：分支定界法类似于回溯法，也是一种在问题的解空间树T上搜索问题解的算法。但在一般情况下，分支定界法与回溯法的求解目标不同。回溯法的求解目标是找出T中满足约束条件的所有解，而分支定界的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

1. NP问题理解：

如果一个问题可以找到一个能在多项式的时间里解决它的算法，那么这个问题就属于P问题。NP问题是指可以在多项式的时间里验证一个解的问题。所有的P类问题都是NP问题。也就是说，能多项式地解决一个问题，必然能多项式地验证一个问题的解。

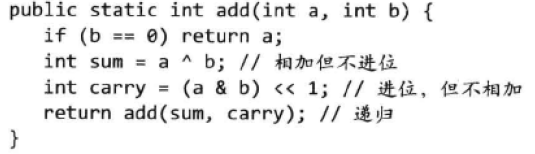
 约化：简单地说，一个问题A可以约化为问题B的含义即是，可以用问题B的解法解决问题A，或者说，问题A可以“变成”问题B。  “问题A可约化为问题B”有一个重要的直观意义：B的时间复杂度高于或者等于A的时间复杂度。也就是说，问题A不比问题B难。“可约化”是指的可“多项式地”约化，即变换输入的方法是能在多项式的时间里完成的。约化的过程只有用多项式的时间完成才有意义。

 NPC问题的定义非常简单。同时满足下面两个条件的问题就是NPC问题。首先，它得是一个NP问题；然后，所有的NP问题都可以约化到它。正是NPC问题的存在，使人们相信P≠NP，NPC问题目前没有多项式的有效算法，只能用指数级甚至阶乘级复杂度的搜索。

NP-Hard问题是这样一种问题，它满足NPC问题定义的第二条但不一定要满足第一条，事实上，由于NP-Hard放宽了限定条件，它将有可能比所有的NPC问题的时间复杂度更高从而更难以解决。

## 常见题型

1. 随机生成数字并传入某个方法，每当收到新数字时找出中位数：使用两个优先级堆，一个大堆顶，存放所有小于中位数的值，一个小堆顶，存放所有大于中位数的值。并且使得大堆顶的元素比小堆顶多一个或者相等。
2. 不用比较运算符求两个数字的最大最小值：max=(a+b+|a-b|)/2。min=(a+b-|a-b|)/2。
3. 两个数字相加，不用+号：（往位运算上想），将+动作分解成求和和进位分开进行。代码如下:



1. 给定rand5()，实现rand7():通过5\* rand5()+rand5()产生范围为0-24的数，然后舍弃21-24，最后除7取余。
2. 找出任一单词在一本书中出现的频率：单次查找直接逐字逐句扫描即可。O(n)；多次查找先对书预处理，构建散列表，将单次映射到出现频率。
3. 给定一个整数，打印其英文表示：分成3位处理，并在合适和地方插入“thousand”，“million”，”billion”。
4. 计算n阶乘的尾随0个数：迭代计算5到n中每个数字有多少个5。或者while(N>0)ret += N/5；N /= 5。计算n阶乘的二进制表示中最低位1的位置（除2）：while(n>0){n>>1；ret += n}。
5. 给定正整数n，求从1到n一个出现的“1”的个数：方法一。遍历1到n，将每个数中1个数加起来。方法二。找出每一个位和高低位的关系。
6. 最大公约数问题：辗转相除法f(x，y)=f(y，x%y)或者相减法f(x，y)=f(x-y，y)。求最小公倍数则用二者乘积除于最小公倍数；
7. 给定长度为N的数组，求子数组N-1的最大乘积：分析数组中“+”，“-”，”0”的个数判断。
8. 区间覆盖问题：一维情况下，方法一。将目标区间逐个投影到坐标轴上，然后考察源区间未被覆盖的部分即可O(n2)。方法二。先对区间进行快速排序并合并，然后二分查找O(nlogn)。二维情况下， 将矩形分别投影到X轴，Y轴上面。对每个方向分别计算包含关系。如果两个方向，都包含。则该图像一定被其他窗口覆盖。如果一个方向包含，就要具体分析另一个方向：半包含，不存在包含关系。

# 中文分词

## 11.1 trie树

Trie树，即字典树，又称单词查找树或键树，是一种树形结构。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是最大限度地减少无谓的字符串比较，查询效率比较高。

Trie的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

它有3个基本性质：

1.根节点不包含字符，除根节点外每一个节点都只包含一个字符。

2.从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。

3.每个节点的所有子节点包含的字符都不相同。

Trie分词，ansj源码；最长匹配（商品和服务）效果差；trie树空间利用率不高；单模式匹配效率高，多模式匹配要一个一个采用KMP算法去匹配，多次遍历，效率低；

AC自动机可完成多模式匹配，主要是trie树加一个failure表，将trie树转为双数组trie树即可。

a.根节点，上图的绿色节点。被称为Forest，没有实际含义，也不含属性。

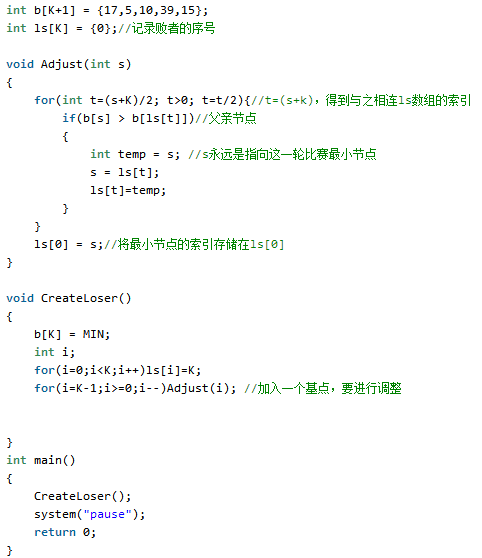
b.起始节点，上图的蓝色节点。是一个单词的开头第一个字，不含属性。

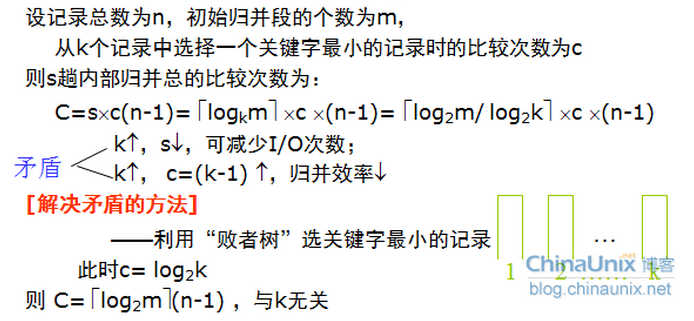
c.中继节点，上图的黄色节点。是一个单词的结尾，含属性；同时是另一个更长的单词的中间某个字，不含属性（节点的属性不代表这个长单词的属性）。

d.结束节点，上图的红色节点。是一个单词的结尾，含属性。

使用正向匹配算法（1：继续 2：是个词语但是还可以继续 3：确定），每次完成一个词的匹配，i++，循环匹配

## 11.2 败者树





## 11.3 ac自动机源码

1.首先构建goto表，当到达一个词的结尾时，添加一个匹配到的模式串

2.构建fail表，设这个节点上的字母为C，沿着他父亲的失败指针走，直到走到一个节点，他的儿子中也有字母为C的节点。然后把当前节点的失败指针指向那个字母也为C的儿子（此时C将此儿子节点的匹配到的字符串也添加进来）。如果一直走到了root都没找到，那就把失败指针指向root。

3.output表用state中的Set<String> emits进行表示，红字添加的就是添加到这个emits中。

4.匹配：匹配时直接按最长匹配，但是每个词的节点都包含了此节点对应的已匹配的字符串（构建fail表时添加），因此只需遍历一次就可以得到所有匹配的字符串

## 11.4双数组ac自动机

State代表节点，包含goto的map，失败后指向的state，深度，模式串等信息。

在建立表的过程中除了构建state之间的关系，还将goto表、fail表以及out表全部构件好；匹配的时候，只需按照ac自动机的流程走一遍即可得到输出结果。

## 11.5 中文分词

1.N最短路径分词：采用贪心算法求得n个最短路径（使用ac自动机），仅粗分；距离采用-log（概率小数）得到，这样概率越大，值越小。此步骤调用了二元模型。

即通过遍历词图来更新二元概率（取大的），然后从最后一个词向前找前驱，即找到最短路径。

歧义识别在此，只有一次机会，若最短路径是错的，那么后面自定义词典中没有这个词，那么结果可能是错的；

得到（N）最短路径后，加载自定义词典，识别自定义词典中的词，然后进行人名识别

2.人名识别（未登录词识别）：基于角色标注。角色标注所需的字词作为不同角色的出现概率和角色间的转移概率,都是在语料库训练过程中自动抽取的。（且人名识别用的是自己特定的角色，如B、C、D）

最终得到p(w|c)，参与下一步计算

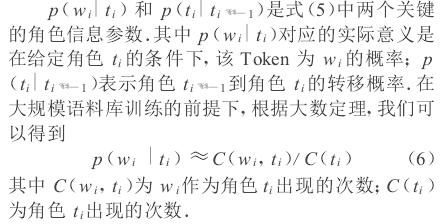
词为观察值，角色为状态值，则有一个观察序列和状态序列，这是一个隐马尔可夫链，通过维特比算法求得。

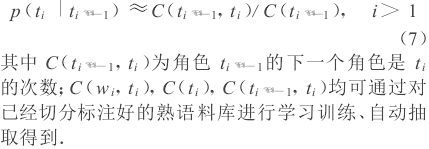
维比特算法看论文，几个公式

W是分词后的观测序列，T是W某个可能的角色标注序列



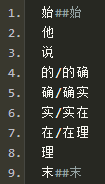






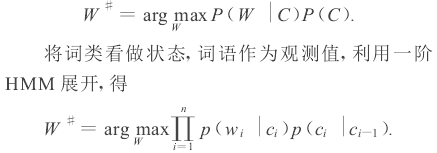
C（w，t）表示w作为角色t出现的次数（观测概率），C（t）代表角色t出现的次数（初始状态概率），C（ti-1, ti）为角色ti-1的下一个角色是ti,的次数（状态转移概率），均可以通过已经切分标注好的语料库进行训练，自动抽取得到。

N最短路径得到的词用一维数组表示



“的确”的行号是4,长度是2,4+2=6，于是第六行的两个词“实/实在”就是“的确”的后续

3.得到p(w|c)后，在词典中的p(w|c)为1，然后通过维特比算法计算p(W|C)；



得到最终的结果

4.隐马尔科夫模型

A:状态转移概率

B：观测概率

Pi：初始概率

## 11.6同义词

分词采取字典匹配方式，分词后会存在一些专业名词，词林中不包括这些专业名词；

Word2vec采用与分词相同的病例数据，所以最终会包含专业名词；

同义词识别，使用word2vec进行处理，得到词向量，进行kmeans聚类，手动查询获取到语义核心词集，进行相似度计算

Word2vec：

1.词向量：用向量表示词。oneone-hot representation，另一种就是 Distributed Representation 余弦相似度

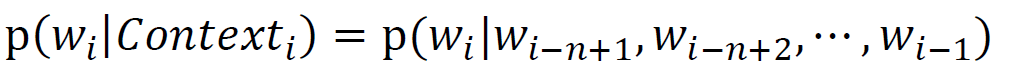
2.语言模型：



当句子的长度很长时，计算的量级和存储信息需要很大的开销。因此使用N-gram模型进行简化。

3.n-gram：N-gram就是只管这个词前面的n-1个词，加上它自己，总共n个词，计算 只考虑用这n个词来算，换成数学的公式来表示，就是





4.算法：

多层softmax：

对一个句子遍历每个词，对一个词划定一个窗口，将窗口内除了这个词之外的词的词向量相加得到v，根据哈弗曼编码求这个词的概率，调整v，使得概率最大化。

负采样：

默认当前词为正样本，其他为负样本，负样本抽取按频次高低抽取。

关键在于负样本的抽取，多层softmax利用哈弗曼树，负采样利用频次

源码预处理将每个区间中的sigmoid值计算好存入到数组expTable中

# 机器学习

## 12.1 逻辑回归

把决策函数限定在某一组条件下

求解参数采用梯度下降和拟牛顿法

## 12.2 决策树

熵：表示随机变量不确定性的度量

信息增益：得知特征x的信息儿时的类y的信息的不确定性减少的程度

做一个映射

与测试显示prob，重写

代码（优化后）：flatmap得到 <key，列号>；distinct去重；maptopair得到<列号，key>；groupByKey得到每一列的key；collect得到全部，然后做映射

## 12.3 FP-grouth

频繁模式： 支持度数，支持度

关联规则：{F,G}=>{A}的置信度为：{A,F,G}的支持度数 除以 {F,G}的支持度数，即3/4

代码：得到[关键字list，freq]，按行读取，遍历list，得到X => Y, generate (X, (Y, freq(X union Y))) Y长度为1；得到所有(X, freq(X))，join得到(X, ((Y, freq(X union Y)), freq(X)));最后得到结果

## 12.4 ROC

横轴：假正类

纵轴：真正类

按区间计算，区间【0。1，0。2，0。3，。。。】，则0。15映射到0。1

代码：使用mapToPair以及reduceByKey分别算出正类负类个数；

将所有正例和负例格式化为<POSITIVETAG+bucket，1>，bucket表示区间位置，然后用一个二维表示，分别为正负以及bucket个数，并进行累加，得到每个区间的正类和负类的个数然后计算即可；从大到小计算，累加即可。

# 分布式

## 13.1 hadoop shuffle

Map：每个map任务都有一个内存缓冲区用于存储任务输出。一旦缓冲内容达到阈值，则把内容溢出（spill）到磁盘。

在写磁盘之前，根据数据最终要传的reducer把数据划分成相应的分区（partition），每个分区对数据进行排序。

最后将溢出文件合并成一个已分区且已排序的输出文件。

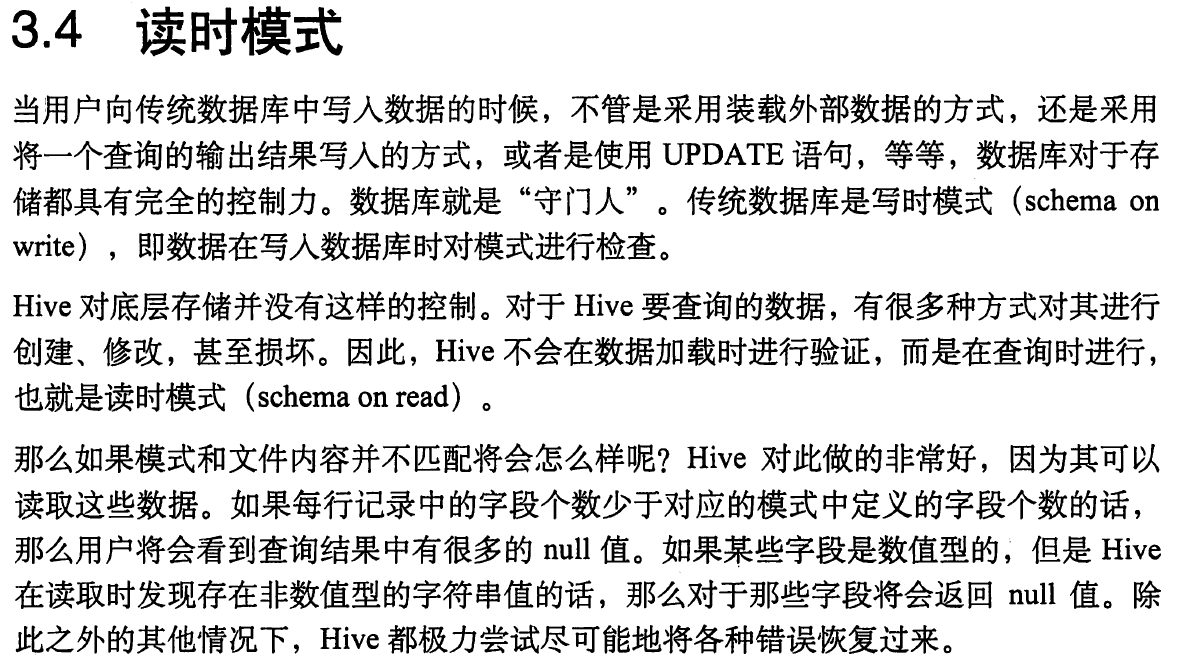
Reduce：将map送来的属于同一划分的输出进行合并，并排序，交给reduce处理。

## 13.2 hive

Hive是基于Hadoop的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供类SQL查询功能。

它提供了一系列的工具，可以用来进行数据提取转化加载（ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制

本质是将SQL转换为MapReduce程序



## 13.3 hdfs

1.处理超大文件 2.一次写入，多次读取 3.运行在廉价集群上

不适合低延迟数据访问，无法高效存储大量小文件（namenode管理文件系统的元数据，文件太多，占内存太多），不支持多用户写入及任意修改

1.将文件划分为块，64M，可以存储任意大的文件（逻辑块）2.简化存储子系统，将文件块分为3份，分别存储在集群的不同节点上。损坏时，通过namenode获取信息，读取副本。

Namenode管理文件系统的命名空间，维护整个文件系统的文件和目录。还记录着每个文件中各个块所在的数据节点信息。

Datanode是文件系统的工作节点，根据需要存储并检索数据块，并定期向namenode发送他么你所存储的块的列表。

通过备份namenode或者运行一个辅助namenode（一个副本）

## 13.4spark

RDD:弹性分布式数据集 支持2中操作：transformation和action

惰性计算：第一次在一个行动操作中用到时，才会进行真正的计算。

持久化：对同一RDD1进行2次action运算，则需要持久化，否则会再次从源头处重新计算一次RD1D的数据

Collect后得到<列，list<类>>，生成hashmap

提交一个spark作业后，作业就会启动一个对应的Driver进程，在每个节点申请Executor进程，根据业务代码以shuffle为界划分stage，并为每个stage创建一批task，然后将这些task分配到各个Executor进程中执行。

根据shuffle类算子来进行stage的划分。shuffle算子执行之前的代码会被划分为一个stage，shuffle算子执行以及之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个task可能都会从上一个stage的task所在的节点，去通过网络传输拉取需要自己处理的所有key，然后对拉取到的所有相同的key使用我们自己编写的算子函数执行聚合操作（比如reduceByKey()算子接收的函数）。这个过程就是shuffle

每个shuffle map task根据key的hash值，计算出每个key需要写入的partition，然后将数据单独写入一个文件，这个partition实际上就对应了下游的一个shuffle map task 或者result task。因此下游的task在计算时会通过网络读取这个文件并进行计算

Map任务的中间输出结果按照key值哈希后分配给某一个reduce任务（把相同的任务传给某个partition）

Spark会根据用户提交的计算逻辑中的RDD的转换和行动来生成RDD之间的依赖关系，也就生成了DAG。RDD之间的依赖关系，DAG可以认为这些RDD之间形成了血统，借助这个血统，能保证一个RDD被计算前，它所依赖的parent RDD都已完成计算；同时业实现了RDD的容错性，即如果一个RDD的部分或者全部计算结果丢失，那么可以重新计算这部分丢失的数据

在Spark里每一个操作生成一个RDD，RDD之间连一条边，最后这些RDD和他们之间的边组成一个有向无环图，这个就是DAG

窄依赖指的是生成的RDD中每个partition只依赖于父RDD(s) 固定的partition。因此可以通过一个task来处理这些partition，这些partition独立，可以多个task并行执行。

宽依赖指的是生成的RDD的每一个partition都依赖于父 RDD(s) 所有partition。宽依赖往往意味着shuffle操作，这也是Spark划分stage的主要边界。

窄依赖典型的操作有map, filter, union等，宽依赖典型的操作有groupByKey, sortByKey等。

Spark的基本概念：

（1）Application：表示你的应用程序

（2）Driver：表示main()函数，创建SparkContext。由SparkContext负责与ClusterManager通信，进行资源的申请，任务的分配和监控等。程序执行完毕后关闭SparkContext

（3）Executor：某个Application运行在Worker节点上的一个进程，该进程负责运行某些task，并且负责将数据存在内存或者磁盘上。在Spark on Yarn模式下，其进程名称为 CoarseGrainedExecutor Backend，一个CoarseGrainedExecutor Backend进程有且仅有一个executor对象，它负责将Task包装成taskRunner，并从线程池中抽取出一个空闲线程运行Task，这样，每个CoarseGrainedExecutorBackend能并行运行Task的数据就取决于分配给它的CPU的个数。

（4）Worker：集群中可以运行Application代码的节点。在Standalone模式中指的是通过slave文件配置的worker节点，在Spark on Yarn模式中指的就是NodeManager节点。

（5）Task：在Executor进程中执行任务的工作单元，多个Task组成一个Stage

（6）Job：包含多个Task组成的并行计算，是由Action行为触发的

（7）Stage：每个Job会被拆分很多组Task，作为一个TaskSet，其名称为Stage

（8）DAGScheduler：根据Job构建基于Stage的DAG，并提交Stage给TaskScheduler，其划分Stage的依据是RDD之间的依赖关系

（9）TaskScheduler：将TaskSet提交给Worker（集群）运行，每个Executor运行什么Task就是在此处分配的。

Spark的执行流程：

Spark在进行transformation计算的时候，不会触发Job ，只有执行action操作的时候，才会触发Job，在Driver中SparkContext根据RDD之间的依赖关系创建出DAG有向无环图，DAGScheduler负责解析这个图，解析时是以Shuffle为边界，反向解析，构建stage。将多个任务根据依赖关系划分为不同的Stage，将每个Stage的Taste Set 提交给TaskScheduler去执行，任务会在Executor进程的多个Task线程上执行，完成Task任务后 将结果信息提交到ExecutorBackend中 他会将信息提交给TaskScheduler。

TaskScheduler接到消息后通知TaskManager，移除该Task任务，开始执行下一个任务。TaskScheduler同时会将信息同步到TaskSet Manager中一份，全部任务执行完毕后TaskSet Manager将结果反馈给DAGScheduler，如果属于ResultTask 会交给JobListener。否则话全部任务执行完毕后写入数据。

# 搜索引擎

## 14.1 流程

点击搜索，进入search，默认是question得到所有id，生成左右id，根据cookie显示左边或者右边，与分页互不影响

左右显示采用了2个重定向，设置不同的cookie值，由search进行判断

## 14.2分页

主要是为了逻辑和显示

## 14.3倒排索引

IDF在得到关键词key的url个数后可以直接求得，因为forward的count是知道的。

TF在forward中有每个url中每个词的tf值（相同）

生成倒排索引的时候，对于每个关键词key，他在每个url的tf和idf值都有，因此直接可以得到tf-idf值，查询的时候进行相加，然后排序即可

## 14.4布尔运算

PageID

TFIDF

QulityandPageID

这三列主要是为了显示不同的排序方法，可保留一列

PageID和QulityandPageID默认按quality排序，直接retainall，TFIDF也是按照quality排序，因此采用hashmap进行求交集