# Table of Contents

[Introduction](#_bookmark0) [JAVA](#_bookmark1)

[Collection](#_bookmark2)

[HashMap与HashTable有什么区别](#_bookmark3)[关键字](#_bookmark4)

[多线程](#_bookmark5)

[concurrent包](#_bookmark6)[内存模型](#_bookmark7)

[垃圾回收机制](#_bookmark8)[序列化](#_bookmark9)

[对象创建的过程](#_bookmark10)[类加载机制](#_bookmark11)

[NIO](#_bookmark12)

[JAVA反射](#_bookmark13)[动态代理](#_bookmark14)[设计模式](#_bookmark15)

[JAVA内存泄露与OutOfMemory](#_bookmark16)

[for\_each和for的效率差别](#_bookmark17)[CAS](#_bookmark18)

[Iterator与Iterable](#_bookmark19)

[笔试题总结](#_bookmark20)

[C++](#_bookmark21)

[new delete 和 malloc free 的区别](#_bookmark22)

[c++ const提高函数健壮性](#_bookmark23)[多重继承与虚继承](#_bookmark24)

[引用和指针的区别](#_bookmark25)

[const和define](#_bookmark26)

[define 和内联函数](#_bookmark27)

[拷贝构造函数与赋值运算符重载](#_bookmark28)[空指针调用函数](#_bookmark29)

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.2.7

1.2.8

1.2.9

1.2.10

1.2.11

1.2.12

1.2.13

1.2.14

1.2.15

1.2.16

1.2.17

1.2.18

1.2.19

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.3.5

1.3.6

1.3.7

1.3.8

[类的继承与组合](#_bookmark30)[强制转换](#_bookmark31)

[重载 覆盖 隐藏](#_bookmark32)

[Hadoop](#_bookmark33)

[MapReduce](#_bookmark34) [HDFS](#_bookmark35)

[Hadoop2.x 学习笔记](#_bookmark36)

[数据库](#_bookmark37)

[数据库索引](#_bookmark38)

[数据库隔离级别](#_bookmark39)[数据库锁](#_bookmark40)

[Mysql中MyIsam和Innodb的区别](#_bookmark41)[数据库范式](#_bookmark42)

[Nosql](#_bookmark43)

[一致性Hash](#_bookmark44)

[PreparedStatement和Statement的区别](#_bookmark45)[数据库左右连接外连接内连接](#_bookmark46)

[MySQL 数据库性能优化之缓存参数优化](#_bookmark47)[操作系统](#_bookmark48)

[进程与线程的区别](#_bookmark49)

[进程间通信IPC windows端和linux端](#_bookmark50)[死锁](#_bookmark51)

[死锁 java程序](#_bookmark52)

[分页与分段的区别](#_bookmark53) [页面置换算法有哪些](#_bookmark54)

[计算机网络](#_bookmark55)

[TCP三次握手](#_bookmark56)

[Http协议](#_bookmark57)

[TCP数据包首部](#_bookmark58)

[ARP和RARP](#_bookmark59)

[TCP流量控制，拥塞控制和可靠传输](#_bookmark60)[端口](#_bookmark61)

[DNS](#_bookmark62) [ICMP](#_bookmark63)

1.3.9

1.3.10

1.3.11

1.4

1.4.1

1.4.2

1.4.3

1.5

1.5.1

1.5.2

1.5.3

1.5.4

1.5.5

1.5.6

1.5.7

1.5.8

1.5.9

1.5.10

1.6

1.6.1

1.6.2

1.6.3

1.6.4

1.6.5

1.6.6

1.7

1.7.1

1.7.2

1.7.3

1.7.4

1.7.5

1.7.6

1.7.7

1.7.8

[UDP](#_bookmark64)

[IP协议](#_bookmark65)

[TCP如何保持长连接](#_bookmark66)[数据结构与算法](#_bookmark67)

[B-树 B+树](#_bookmark68)[红黑树](#_bookmark69)

[二叉搜索树](#_bookmark70)[平衡二叉树](#_bookmark71)

[B-树 红黑树 二叉搜索树 AVL树 比较](#_bookmark72)[最长递增子序列](#_bookmark73)

[排序算法比较](#_bookmark74)[散列](#_bookmark75)

[各种排序算法代码实现](#_bookmark76)

[UML](#_bookmark77)

[0\/1背包问题](#_bookmark78)[Linux](#_bookmark79)

[Linux常用命令](#_bookmark80)[项目经历](#_bookmark81)

[云计算平台容错能力评估方法的研究](#_bookmark82) [基于HDFS 的云存储系统的设计与实现](#_bookmark83)

[动态可重构穿戴式计算机系统实现（C++）](#_bookmark84)

[无线充电传感器网络点播式充电调度算法设计实现](#_bookmark85)[面试常见问题](#_bookmark86)

[腾讯面试](#_bookmark87)[阿里面试](#_bookmark88)

[安卓签到精灵app](#_bookmark89) [滴滴一面](#_bookmark90)

[去哪面试](#_bookmark91)

[in面试](#_bookmark92)[J2EE](#_bookmark93)

[Structs](#_bookmark94) [Spring](#_bookmark95)

[IOC及Bean容器](#_bookmark96)

1.7.9

1.7.10

1.7.11

1.8

1.8.1

1.8.2

1.8.3

1.8.4

1.8.5

1.8.6

1.8.7

1.8.8

1.8.9

1.8.10

1.8.11

1.9

1.9.1

1.10

1.10.1

1.10.2

1.10.3

1.10.4

1.10.5

1.10.6

1.10.7

1.10.8

1.10.9

1.10.10

1.10.11

1.11

1.11.1

1.11.2

1.11.2.1

[J2EE基础](#_bookmark97)[Hibernate](#_bookmark98)

[转发和重定向的区别](#_bookmark99)

[HTTP 状 态 码](#_bookmark100) [Cookie和Session区别](#_bookmark101)

[URL重写](#_bookmark102)

[HTML CSS JavaScript](#_bookmark103)

[剑指offer](#_bookmark104)

[二叉树镜像](#_bookmark105)

[二维数组中查找](#_bookmark106)[替换空格](#_bookmark107)

[从尾到头打印链表](#_bookmark108)[重建二叉树](#_bookmark109)

[用两个栈实现队列](#_bookmark110)[旋转数组最小值](#_bookmark111) [斐波那契数列](#_bookmark112)

[跳台阶](#_bookmark113)

[变态跳台阶](#_bookmark114)[矩形覆盖](#_bookmark115)

[二进制中一的个数](#_bookmark116)[数值的整数次方](#_bookmark117)

[调整数组顺序是奇数位于偶数前面](#_bookmark118)[链表倒数第K个节点](#_bookmark119)

[合并两个有序链表](#_bookmark120)[树的子结构](#_bookmark121)

[顺时针打印矩阵](#_bookmark122)[包含min函数的栈](#_bookmark123)

[栈的压入与弹出序列](#_bookmark124)[从上到下二叉树遍历](#_bookmark125)

[二叉搜索树的后序遍历序列](#_bookmark126)[二叉树中和为某一值的路径](#_bookmark127)[复杂链表的复制](#_bookmark128)

[二叉搜索树与双向链表](#_bookmark129)

[数组中出现次数超过一半的数字](#_bookmark130)

1.11.3

1.11.4

1.11.5

1.11.6

1.11.7

1.11.8

1.11.9

1.12

1.12.1

1.12.2

1.12.3

1.12.4

1.12.5

1.12.6

1.12.7

1.12.8

1.12.9

1.12.10

1.12.11

1.12.12

1.12.13

1.12.14

1.12.15

1.12.16

1.12.17

1.12.18

1.12.19

1.12.20

1.12.21

1.12.22

1.12.23

1.12.24

1.12.25

1.12.26

[最小的K个数](#_bookmark131)

[连续子数组最大和](#_bookmark132)

[整数中出现1的次数（从1到n整数中1出现的次数）](#_bookmark133) [把数组排成最小的数](#_bookmark134)

[丑数](#_bookmark135)

[找出字符串中第一个只出现一次的字符](#_bookmark136)[数组中的逆序对](#_bookmark137)

[两个链表的第一个公共节点](#_bookmark138) [数字在排序数组中出现的次数](#_bookmark139)[二叉树的深度](#_bookmark140)

[平衡二叉树](#_bookmark141)

[数组中只出现一次的数字](#_bookmark142)[和为S的连续正整数序列](#_bookmark143)[和为S的两个数字](#_bookmark144)

[左旋转字符串](#_bookmark145) [翻转单词顺序列](#_bookmark146)[扑克牌顺子](#_bookmark147)

[求1+2+3+...+n](#_bookmark148)

[不用加减乘除做加法](#_bookmark149)[把字符串转换成整数](#_bookmark150)[数组中重复的数字](#_bookmark151) [构建乘积数组](#_bookmark152)

[矩阵中的路径](#_bookmark153)

1.12.27

1.12.28

1.12.29

1.12.30

1.12.31

1.12.32

1.12.33

1.12.34

1.12.35

1.12.36

1.12.37

1.12.38

1.12.39

1.12.40

1.12.41

1.12.42

1.12.43

1.12.44

1.12.45

1.12.46

1.12.47

1.12.48

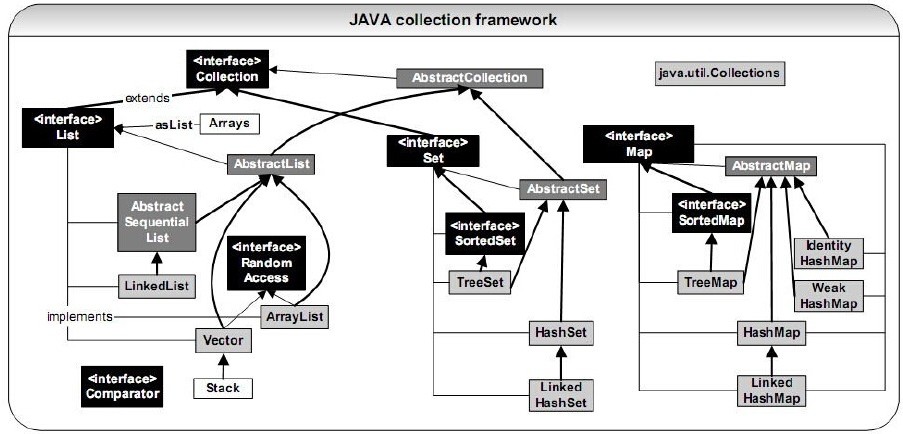
1.12.49

# My Awesome Book

This file file serves as your book's preface, a great place to describe your book's content and ideas.

# JAVA

**JAVA collection**



##### Collection接口

Collection是一个集合，它是一个interface，List和Set都是它的subinterface。

**Set**接口

set是Collection的子接口，它不能有重复元素，不能以自身为set内的元素。

##### HashSet

散列码（hash code）是一个表示对象某些相对位移特性的整数值。散列码由hashcode 方法生成，该方法返回基于内存对象地址返回的一个值。对象经常覆盖hashcode方法， 以获得更合适的表现自身的值。

散列码通过散列法的过程快速找到存储在散列表中的对象键值对。HashSet内部是有

HashMap实现的。

建议添加到集合的对象一定要实现hashcode和equals方法。

##### TreeSet

##### TreeSet是JAVA中集合的一种，TreeSet 是一个有序的集合，它的作用是提供有序的Set集合。它继承于AbstractSet抽象类，实现了NavigableSet<E>,Cloneable,java.io.Serializable接口。TreeSet是基于TreeMap实现的。TreeSet中的元素支持2种排序方式：自然排序 或者 根据创建TreeSet 时提供的 Comparator 进行排序。这取决于使用的构造方法。

**List**接口

List包含了添加、删除和获取集合内部特定位置元素的附加方法。

##### ArrayList 线程不安全

##### LinkedList

##### Vector

**Map**

Map接口并不是一个Collection，他没有对Collection接口进行扩展。Map内有一个方法

entrySet()方法，这个方法返回的是一个set，set里面的元素是Entry内部接口。Entry(K，

V)是一个元组里面放的是Key和Value，把他们变成一个对象。

##### HashMap

HashMap基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

往hashmap里面放键值对的时候先得到key的hashcode，然后重新计算hashcode，（让1 分布均匀因为如果分布不均匀，低位全是0，则后来计算数组下标的时候会冲突），然后与length-1按位与，计算数组出数组下标如果该下标对应的链表为空，则直接把键值对作为链表头结点，如果不为空，则遍历链表看是否有key值相同的，有就把value替换，没 有就把该对象最为链表的第一个节点，原有的节点最为他的后续节点

值得注意的是HashMap不是线程安全的，如果想要线程安全的HashMap，可以通过

Collections类的静态方法synchronizedMap获得线程安全的HashMap。

Map map = Collections.synchronizedMap(new HashMap());

参数:初始容量16，达到阀值扩容，阀值等于最大容量\*负载因子，扩容每次2倍，总是2的

n次方.

**HashMap**的数据结构**:**

HashMap的底层主要是基于数组和链表来实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储的位置。HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。学过数据结构的同学都知道，解决hash冲突的方法有很多，HashMap底层是通过链表来解决hash冲突的。HashMap其实就是一个Entry数组，Entry对象中包含了键和值，其中next也是一个Entry对象，它就是用来处理hash冲突的，形成一个链表。

##### HashTable

**TreeMap**

TreeMap是继承自AbstractMap，AbstractMap实现了Map接口。它的实现是通过红黑树。它自定义了自己的Entry<K,V>。这个Entry里包含的元素有Key,Value,parent,left

child,color等成员变量。根据算法维持红黑树的特性。

可以接受一个Comparator对象作为构造函数。

##### Iterator,ListIterator和Enumeration接口

Iterator接口提供迭代和删除集合内元素的能力。List和Set实现提供了各种返回Iterator实现的方法。这些迭代器是故障快速修复的（fail-fast），这以为这如果在迭代过程中试图从迭代器外部修改底层对象，他们会立刻失效。

ListIterator是一个对Iterator的一个扩展，它提供了向后移动以及添加和替换列表元素的功能。ListIterator也是快速修复的。

**Iterator**，**ListIterator**与**Enumeration**接口的区别：

作为集合框架的一部分，Iterator和ListIterator旨在替换以前的Enumeration接口，该接口最初由Vector和HashTable类支持。他们本质上都具有相同功能。除了下面几项以外：

Iterator提供集合删除元素的功能

Iterator和ListIterator是故障快速修复的，Enumeration实现不是故障快速修复的，因此如果枚举过程中对集合进行修复，可能会产生不可预知的和不想得到的结果。

ListIterator为操作列表提供了一些附加方法。

Iterator和ListIterator提供较短的方法名称。

Enumeration方法包括hasMoreElements()和nextElement()

Iterator提供方法：hasNext()，next()，remove()

ListIterator除了Iterator以外还提供：nextIndex()，previousIndex(),hasPrevious()， previous()，add()， set().

vector和HashTable都它支持在元素总进行枚举（enumeration）的功能。

**Java**中的同步容器类

在Java中，同步容器主要包括2类：

1. Vector、Stack、HashTable
2. Collections类中提供的静态工厂方法创建的类

Vector实现了List接口，Vector实际上就是一个数组，和ArrayList类似，但是Vector中的方法都是synchronized方法，即进行了同步措施。

Stack也是一个同步容器，它的方法也用synchronized进行了同步，它实际上是继承于Vector

类。

HashTable实现了Map接口，它和HashMap很相似，但是HashTable进行了同步处理，而

HashMap没有。

Collections类是一个工具提供类，注意，它和Collection不同，Collection是一个顶层的接口。在Collections类中提供了大量的方法，比如对集合或者容器进行排序、查找等操作。最重要的是，在它里面提供了几个静态工厂方法来创建同步容器类。

# HashMap HashTable ConcurrentHashMap

1. **HashMap HashTable**区别？，**Hashmap key**可以是任何类型吗？
   1. HashTable的方法是同步 的，在方法的前面都有synchronized来同步，HashMap未经同步，所以在多线程场合要手动同步 HashMap这个区别就像Vector和ArrayList一样。可以通过 Map m = Collections.synchronizedMap(new HashMap(...)); 对HashMap进行同步操作。
   2. HashTable不允许null值(key和value都不可以) ,HashMap允许null值(key和value都可以)。
   3. HashTable有一个contains(Object value)功能和containsValue(Object value)功能一样。
   4. HashTable使用Enumeration进行遍历，HashMap使用Iterator进行遍历。以上只是表面的不同，它们的实现也有很大的不同。
   5. 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。

6.哈希值的使用不同，HashTable直接使用对象的hashCode，代码是这样的：

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

而HashMap重新计算hash值，而且用与代替求模：

int hash = hash(k);

int i = indexFor(hash, table.length); static int hash(Object x) {

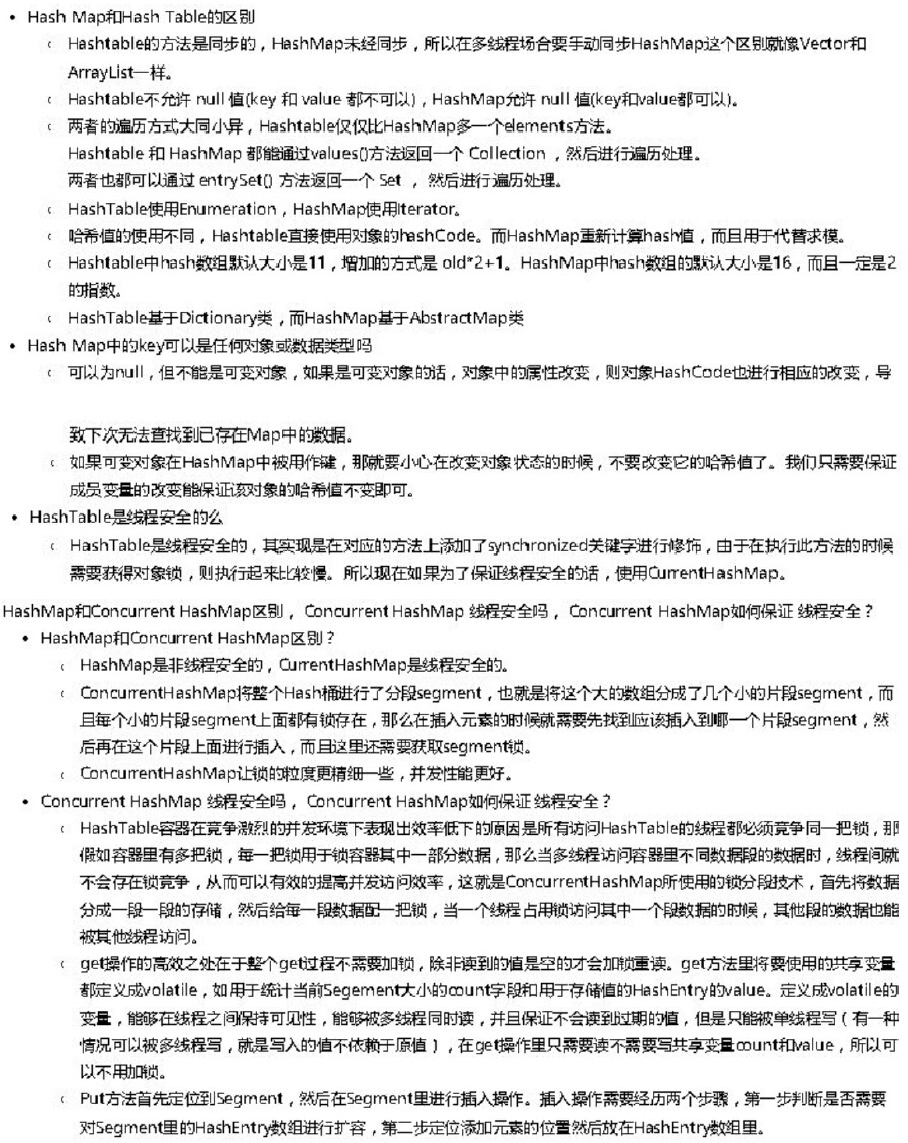
h ^= (h >>> 20) ^ (h >>> 12);

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

}

static int indexFor(int h, int length) { return h & (length-1);

}

以上只是一些比较突出的区别。

**ConcurrentHashMap**使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。通过**lock()**和**unlock().**

##### LinkedList ArrayList区别？

ArrayList基于数组，LinkedList基于链表 ArrayList查找快，LinkedList插入删除快 随机查找频繁用ArrayList,插入删除频繁用LinkedList

##### ArrayList,Vector,HashMap,Hashtable扩容机制？

arraylist,初始容量10，(oldCapacity \* 3)\/2 + 1

vector, 初 始 容 量 10，oldCapacity \* 2 hashmap,初始容量16，达到阀值扩容，为原来的两倍

hashtable，初始容量11，达到阀值扩容，oldCapacity \* 2 + 1

##### Concurrent包

semaphore信号量 用来表示可用资源的数量，如果请求资源数超过这一数量，则就要阻塞这一线程

callable接口，实现call函数，但是它有返回值我们可以得到call函数的返回值判断call函数是否执行完毕什么是Callable和Future?Java 5在concurrency包中引入了

java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结 果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

什么是FutureTask?FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读Java FutureTask例子，学习如何使用它。

**Java Concurrency API**中的**Lock**接口**(Lock interface)**是什么？ 对比同步它有什么优势？

Lock接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

可以使锁更公平

可以使线程在等待锁的时候响应中断可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

可以在不同的范围，以不同的顺序获取和释放锁

**JAVA**关键字

**transient**关键字

transient 关键字可以应用于类的成员变量，以便指出该成员变量不应在包含它的类实例已序列化时被序列化。

当一个对象被串行化的时候，transient型变量的值不包括在串行化的表示中，然而非

transient型的变量是被包括进去的。

Java的serialization提供了一种持久化对象实例的机制。当持久化对象时，可能有一个特殊的对象数据成员，我们不想用serialization机制来保存它。为了在一个特定对象的一个域上关闭serialization，可以在这个域前加上关键字transient。

transient是Java语言的关键字，用来表示一个域不是该对象串行化的一部分。当一个对象被串行化的时候，transient型变量的值不包括在串行化的表示中，然而非transient型的变量是被包括进去的。

**native**关键字

native 关键字可以应用于方法，以指示该方法是用 Java 以外的语言实现的。

##### volatile

volatile 关键字用于表示可以被多个线程异步修改的成员变量。

注意：volatile 关键字在许多 Java 虚拟机中都没有实现。 volatile 的目标用途是为了确保所有线程所看到的指定变量的值都是相同的。

Java 语言中的 volatile 变量可以被看作是一种 “程度较轻的 synchronized”；与synchronized 块相比，volatile 变量所需的编码较少，并且运行时开销也较少，但是它所能实现的功能也仅是 synchronized 的一部分。

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序。

**volatile**的原理和实现机制

前面讲述了源于volatile关键字的一些使用，下面我们来探讨一下volatile到底如何保证可见性和禁止指令重排序的。

下面这段话摘自《深入理解Java虚拟机》：

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

2）它会强制将对缓存的修改操作立即写入主存；

3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

**JAVA**多线程

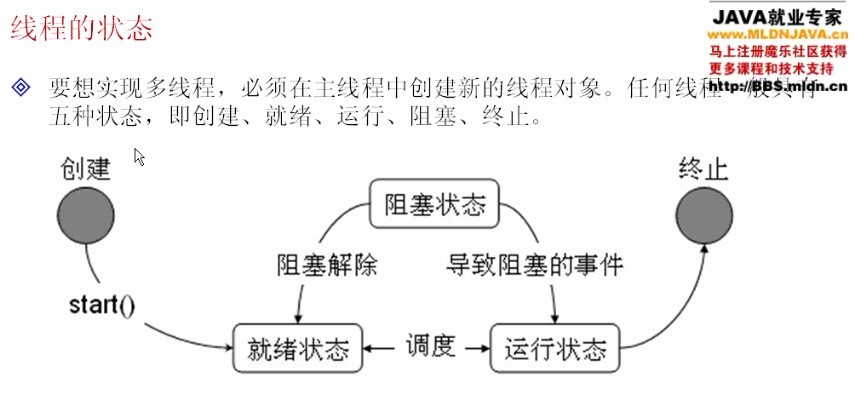
##### Thread与Runnable区别

首先，根据源码可以知道Thread实现了Runnable接口，在执行多线程主体之前，他还干了一些其他的事情，比如说分配CPU资源，检查线程是否启动等事情，然后再执行线程主题。

Runnable适合多个相同程序代码的线程去处理同一个资源，比如买票程序。

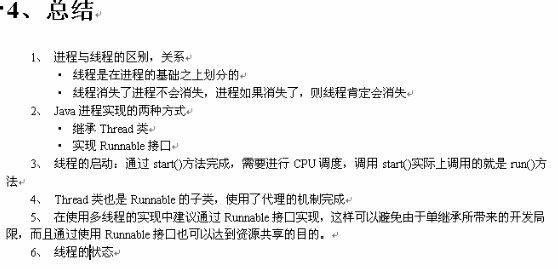
Runnable接口可以避免JAVA单继承的局限。

Runnable增加了程序的健壮性，代码能够被多个线程所共享，数据和代码是独立的。



线程调用**Start()**方法不会立刻启动，而是处于就绪状态。等待

**CPU**调度以后才可以。



###### 多线程操作

isAlive() Thread.currentThread.getName()

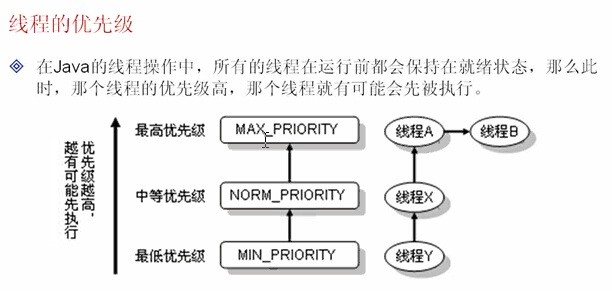
join() :线程强制运行，就是在一个线程中join调用另一个线程的join函数，那么当前线程必

须等到被join的线程执行完才会执行当前线程。

Thread.sleep(): 线程休眠

线程中断：interrupt()，在其他线程中调用t.interrupt()， 该线程抛出interrupt异常，并且继续执行

setDaemon(), 设置后台线程，这样可以在主线程结束的时候它依然在运行优先级：setPriority();主方法优先级是NORM



线程礼让：yield()

###### 线程的终止

* 1. 用stop()方法，这个方法容易产生问题，所以现在已经过时了。
  2. 用一个volatile变量当做控制条件，控制线程执行。
  3. 用interrupt来使处于sleep(),wait()等状态的线程终止
  4. 用Future的cancel方法也可以终止线程，这个跟interrupt用法一样，因为它底层也是用

interrupt实现的。

###### 同步与死锁

资源共享时需要同步，过多同步容易产生死锁。

同步代码块：synchronized(this){}

class MyThread implements Runnable{

private int ticket = 5 ; // 假设一共有5张票

public void run(){

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

synchronized(this){ // 要对当前对象进行同步

if(ticket>0){ // 还有票

try{

Thread.sleep(300) ; // 加入延迟

}catch(InterruptedException e){ e.printStackTrace() ;

}

System.out.println("卖票：ticket = " + ticket-- );

}

}

}

}

};

public class SyncDemo02{

public static void main(String args[]){

MyThread mt = new MyThread() ; // 定 义 线 程 对 象Thread t1 = new Thread(mt) ; // 定 义 Thread 对 象Thread t2 = new Thread(mt) ; // 定 义 Thread 对 象Thread t3 = new Thread(mt) ; // 定 义 Thread 对 象t1.start() ;

t2.start() ;

t3.start() ;

}

};

同步方法：

class MyThread implements Runnable{

private int ticket = 5 ; // 假设一共有5张票

public void run(){

for(int i=0;i<100;i++){ this.sale() ; // 调用同步方法

}

}

public synchronized void sale(){ // 声明同步方法

if(ticket>0){ // 还 有 票

try{

Thread.sleep(300) ; // 加入延迟

}catch(InterruptedException e){ e.printStackTrace() ;

}

System.out.println("卖票：ticket = " + ticket-- );

}

}

};

public class SyncDemo03{

public static void main(String args[]){

MyThread mt = new MyThread() ; // 定 义 线 程 对 象Thread t1 = new Thread(mt) ; // 定 义 Thread 对 象Thread t2 = new Thread(mt) ; // 定 义 Thread 对 象Thread t3 = new Thread(mt) ; // 定 义 Thread 对 象t1.start() ;

t2.start() ;

t3.start() ;

}

};

程序中过多的同步会产生死锁。

# Concurrent包

##### Executor

Executor接口是所有线程执行类的父接口，这个接口可以建立线程池，然后执行线程。

Executor框架的最核心的类是ThreadPoolExecutor，它是线程池的实现类，创建

ThreadPoolExecutor一般使用Executors工厂模式创建，Executors类提供了一系列工厂方法用于创先线程池：

public static ExecutorService newFixedThreadPool(int nThreads)创建固定数目线程的线程池，表示最多创建nThreads个线程，如果传入的任务数大于nThreads时不会创建新的线程，而是阻塞等待有空闲线程执行。

public static ExecutorService newCachedThreadPool()创建一个可缓存的线程池，调用

execute将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor()创建一个单线程化的

Executor。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) 创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

常用方法：

shutDown():关闭执行器，在关闭前允许执行以前提交的任务执行器执行完。调用

shutDown()后，再发送任务给Executor将会被拒绝，抛出RejectExecutionException异常。

shutdownNow() :立即关闭执行器，阻止等待任务启动，并试图停止当前正在执行的任务。返回等待执行的任务列表。

isShutdown():调用shutDown()后，返回true。

isTerminated():调用shutDown()后，并且执行器完成了关闭过程，返回true。

getPoolSize():获取当前线程池的线程数量

getActiveCount():获取线程池中活动线程的数量

getCompleteCount():获取线程池中完成的任务数。

##### ExecutorService

ExecutorService可以帮助我们建立线程池。

##### BlockingQueue

阻塞队列跟信号量差不多，比如生产者消费者问题中，如果不用阻塞队列，采用非阻塞队列存放产品的话，需要用synchronized来对produce()和consume()操作进行同步，而采用阻塞队列就不需要这样了，它自身就带同步功能，当空时自然不能取，满时自然不能继续生产，这就是BlockingQueue的作用。

阻塞队列使用最经典的场景就是socket客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。还有其他类似的场景，只要符合生产者-消费者模型的都可以使用阻塞队列。

##### Semphore

信号量Semaphore是一个计数信号量，用来保护一个或多个共享资源的访问，是Java

Concurrent包下提供的另一种同步方式，就像**synchronized**一样的呢，它就是替代

**synchronized**的。 Semaphore可以控制某个资源可被同时访问的个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

##### Future

可以得到Callable接口与Runnable接口执行的返回值，也可以调用cancel方法取消线程的执行。

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

也就是说Future提供了三种功能：

1）判断任务是否完成；

2）能够中断任务；

3）能够获取任务执行结果。

因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的

FutureTask。

##### Callable

Callable接口与Runnable接口实现的功能都是一样的，不过它有返回值的，我们可以知道线程是否执行完毕呢。通常与Future，FutureTask连着用。

##### ThreadPoolExecutor

CylicBarrier ThreadLocal

##### CountDownLatch

CountDownLatch是Java Concurrent包下提供的同步辅助类。在完成一组正在其他线程中执行的操作之前，它允许线程一直等待。

和Semaphore类似，它使用一个整数进行初始化，Semaphore中的计算表示可以同步访问共享数据的线程数，CountDownLatch中的计算表示线程要等待的操作数数目。

CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用

CountDownLatch来实现这种功能了。

操作方法

构造函数CountDownLatch(int count)，count表示要等待的操作数的数目。

await()方法，阻塞等待，需要其他线程完成期待的操作，直到count为0。

countDown()方法，当某一个操作完成后，调用此方法，count数减一。

下面对上面说的三个辅助类进行一个总结：

1. CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行； 而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行； 另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

1. Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。

##### ThreadFactory

ThreadFactory是一个接口，就是一个工厂模式的典型。线程工厂（**ThreadFactory**接口）

在创建线程的时候，我们当然也能使用工厂模式来生产Thread，ThreadFactory是用来实现创建线程的工厂模式接口，其实它只有一个方法Thread newThread(Runnable r)，所以这个接口没多大用，可以自己编写新接口。

使用ThreadFactory工厂这样就能替代默认的new Thread，而且在自定义工厂里面，我们能创建自定义化的Thread，并且计数，或则限制创建Thread的数量，给每个Thread设置对应的好听的名字，或则其他的很多很多事情。

**ThreadFactory**在并发中的使用

在Java中使用ThreadFactory最多应该就是Executor框架和Fork\/Join框架了，使用

ThreadFactory创建各种线程池中的线程。在Executor框架中，使用Executors创建线程池执行器的时候，也可以传入ThreadFactory对象，执行器将会使用该ThreadFactory对象来创建线程。

如：newCachedThreadPool(ThreadFactory threadFactory)

**Condition**

发包下的提供Lock锁，Lock可以更好的解决线程同步问题，使之更面向对象，并且

ReadWriteLock在处理同步时更强大，那么同样，线程间仅仅互斥是不够的，还需要通信，本篇的内容是基于上篇之上，使用Lock如何处理线程通信。

那么引入本篇的主角，Condition，Condition 将 Object 监视器方法（wait、notify 和

notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set （wait-set）。其中，Lock 替代了 synchronized 方法和语句的使用，

Condition 替代了 Object 监视器方法的使用。

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

Condition的强大之处在于它可以为多个线程间建立不同的Condition， 使用

synchronized\/wait()只有一个阻塞队列，notifyAll会唤起所有阻塞队列下的线程，而使用

lock\/condition，可以实现多个阻塞队列，signalAll只会唤起某个阻塞队列下的阻塞线程。

##### ReentrantLock

可重入锁与synchronized比较？

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，

ReentrantLock类提供了一些高级功能，主要有以下3项：

1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。

2.公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁， 但公平锁表现的性能不是很好。

3.锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象。

##### ThreadLocal

线程私有变量，就是我们在建立类的时候可以把成员变量声明为线程私有的， 最常见的

ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。 ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

准确的说，应该是**ThreadLocal**类型的变量内部的注册表（**Map<Thread,T>**）发生了变化， 但**ThreadLocal**类型的变量本身的确是一个，这才是本质！

虽然**ThreadLocal**变量只有一个，各个线程共享，但是**ThreadLocal**内部维护一个

**Map<Thread,T>**，通过线程**Id**每个线程都维护唯一的一个变量。

一**.synchronized**的缺陷以及与**Lock**的区别

synchronized是java中的一个关键字，也就是说是Java语言内置的特性。那么为什么会出现

Lock呢？

在上面一篇文章中，我们了解到如果一个代码块被synchronized修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2）线程执行发生异常，此时JVM会让线程自动释放锁。

那么如果这个获取锁的线程由于要等待**IO**或者其他原因（比如调用**sleep**方法）被阻塞了，但是又没有释放锁，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

因此就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间或者能够响应中断），通过**Lock**就可以办到。

再举个例子：当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。

但是采用synchronized关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过

Lock就可以办到。

另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

总结一下，也就是说**Lock**提供了比**synchronized**更多的功能。但是要注意以下几点：

1. **Lock**不是**Java**语言内置的，**synchronized**是**Java**语言的关键字，因此是内置特性。

**Lock**是一个类，通过这个类可以实现同步访问；

1. **Lock**和**synchronized**有一点非常大的不同，采用**synchronized**不需要用户去手动释放 锁，当**synchronized**方法或者**synchronized**代码块执行完之后，系统会自动让线程释放对锁的占用；而**Lock**则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象。
2. Lock提供lock(),tryLock(),tryLock(long time, TimeUint uint)，unlock()这些方法，

**tryLock**有返回值，看看返回是否成功。

Lock和synchronized的选择

总结来说，Lock和synchronized有以下几点不同：

1. Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；
2. synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而

Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

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

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

5）Lock可以提高多个线程进行读操作的效率。

在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时

（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

二**.**锁的相关概念介绍

在前面介绍了Lock的基本使用，这一节来介绍一下与锁相关的几个概念。

**1.**可重入锁

如果锁具备可重入性，则称作为可重入锁。像synchronized和ReentrantLock都是可重入锁， 可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个synchronized方法时，比如说method1，而在

method1中会调用另外一个synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2。

看下面这段代码就明白了：

class MyClass {

public synchronized void method1() { method2();

}

public synchronized void method2() {

}

}

上述代码中的两个方法method1和method2都用synchronized修饰了，假如某一时刻，线程A 执行到了method1，此时线程A获取了这个对象的锁，而由于method2也是synchronized方 法，假如synchronized不具备可重入性，此时线程A需要重新申请锁。但是这就会造成一个问题，因为线程A已经持有了该对象的锁，而又在申请获取该对象的锁，这样就会线程A一直等待永远不会获取到的锁。

而由于synchronized和Lock都具备可重入性，所以不会发生上述现象。

**2.**可中断锁

可中断锁：顾名思义，就是可以相应中断的锁。

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

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

在前面演示lockInterruptibly()的用法时已经体现了Lock的可中断性。

**3.**公平锁

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

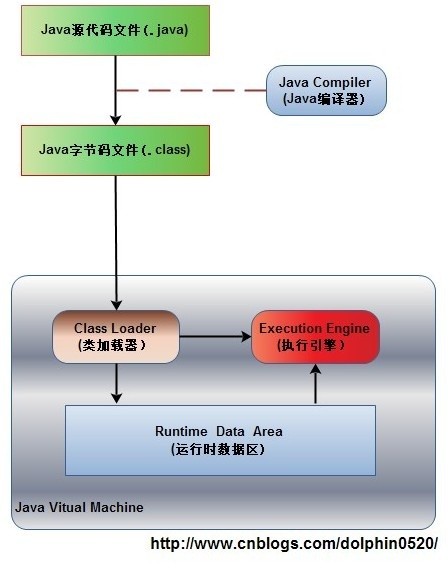
非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

**JAVA** 内存模型

由于Java程序是交由JVM执行的，所以我们在谈Java内存区域划分的时候事实上是指JVM内存区域划分。在讨论JVM内存区域划分之前，先来看一下Java程序具体执行的过程：



如上图所示，首先Java源代码文件(.java后缀)会被Java编译器编译为字节码文件(.class后 缀)，然后由JVM中的类加载器加载各个类的字节码文件，加载完毕之后，交由JVM执行引擎执行。在整个程序执行过程中，JVM会用一段空间来存储程序执行期间需要用到的数据和相关信息，这段空间一般被称作为Runtime Data Area（运行时数据区），也就是我们常说的

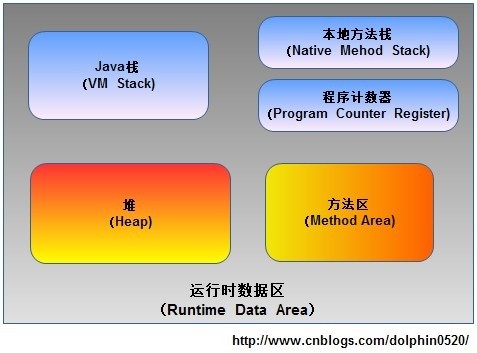
JVM内存。因此，在Java中我们常常说到的内存管理就是针对这段空间进行管理（如何分配和回收内存空间）。

在知道了JVM内存是什么东西之后，下面我们就来讨论一下这段空间具体是如何划分区域的，是不是也像C语言中一样也存在栈和堆呢？

#### 一**.**运行时数据区包括哪几部分？

根据《Java虚拟机规范》的规定，运行时数据区通常包括这几个部分：程序计数器(Program Counter Register)、Java栈(VM Stack)、本地方法栈(Native Method Stack)、方法区(Method

Area)、堆(Heap)。



如上图所示，JVM中的运行时数据区应该包括这些部分。在JVM规范中虽然规定了程序在执行期间运行时数据区应该包括这几部分，但是至于具体如何实现并没有做出规定，不同的虚拟机厂商可以有不同的实现方式。

#### 二**.**运行时数据区的每部分到底存储了哪些数据？

下面我们来了解一下运行时数据区的每部分具体用来存储程序执行过程中的哪些数据。

**1.**程序计数器

程序计数器（Program Counter Register），也有称作为PC寄存器。想必学过汇编语言的朋友对程序计数器这个概念并不陌生，在汇编语言中，程序计数器是指CPU中的寄存器，它保存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当

CPU需要执行指令时，需要从程序计数器中得到当前需要执行的指令所在存储单元的地址， 然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加1或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所有的指令。

虽然JVM中的程序计数器并不像汇编语言中的程序计数器一样是物理概念上的CPU寄存器， 但是JVM中的程序计数器的功能跟汇编语言中的程序计数器的功能在逻辑上是等同的，也就是说是用来指示 执行哪条指令的。

由于在JVM中，多线程是通过线程轮流切换来获得CPU执行时间的，因此，在任一具体时 刻，一个CPU的内核只会执行一条线程中的指令，因此，为了能够使得每个线程都在线程切换后能够恢复在切换之前的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相被干扰，否则就会影响到程序的正常执行次序。因此，可以这么说，程序计数器是每个线程所私有的。

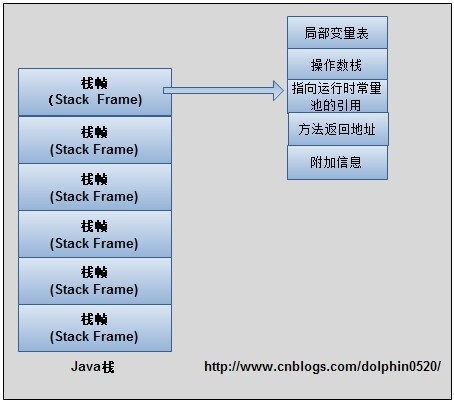
在JVM规范中规定，如果线程执行的是非native方法，则程序计数器中保存的是当前需要执行的指令的地址；如果线程执行的是native方法，则程序计数器中的值是undefined。

由于程序计数器中存储的数据所占空间的大小不会随程序的执行而发生改变，因此，对于程序计数器是不会发生内存溢出现象(OutOfMemory)的。

##### Java栈

Java栈也称作虚拟机栈（Java Vitual Machine Stack），也就是我们常常所说的栈，跟C语言的数据段中的栈类似。事实上，Java栈是Java方法执行的内存模型。为什么这么说呢？下面就来解释一下其中的原因。

Java栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向当前方法所属的类的运行时常量池（运行时常量池的概念在方法区部分会谈到）的引用(Reference to runtime constant pool)、方法返回地址(Return Address)和一些额外的附加信息。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知， 线程当前执行的方法所对应的栈帧必定位于Java栈的顶部。讲到这里，大家就应该会明白为什么 在 使用 递归方法的时候容易导致栈内存溢出的现象了以及为什么栈区的空间不用程序员去管理了（当然在Java中，程序员基本不用关系到内存分配和释放的事情，因为Java有自己的垃圾回收机制），这部分空间的分配和释放都是由系统自动实施的。对于所有的程序设计语言来说，栈这部分空间对程序员来说是不透明的。下图表示了一个Java栈的模型：



局部变量表，顾名思义，想必不用解释大家应该明白它的作用了吧。就是用来存储方法中的局部变量（包括在方法中声明的非静态变量以及函数形参）。对于基本数据类型的变量，则直接存储它的值，对于引用类型的变量，则存的是指向对象的引用。局部变量表的大小在编译器就可以确定其大小了，因此在程序执行期间局部变量表的大小是不会改变的。

操作数栈，想必学过数据结构中的栈的朋友想必对表达式求值问题不会陌生，栈最典型的一个应用就是用来对表达式求值。想想一个线程执行方法的过程中，实际上就是不断执行语句的过程，而归根到底就是进行计算的过程。因此可以这么说，程序中的所有计算过程都是在借助于操作数栈来完成的。

指向运行时常量池的引用，因为在方法执行的过程中有可能需要用到类中的常量，所以必须要有一个引用指向运行时常量。

方法返回地址，当一个方法执行完毕之后，要返回之前调用它的地方，因此在栈帧中必须保存一个方法返回地址。

由于每个线程正在执行的方法可能不同，因此每个线程都会有一个自己的Java栈，互不干扰。

**3.**本地方法栈

本地方法栈与Java栈的作用和原理非常相似。区别只不过是Java栈是为执行Java方法服务 的，而本地方法栈则是为执行本地方法（Native Method）服务的。在JVM规范中，并没有对本地方发展的具体实现方法以及数据结构作强制规定，虚拟机可以自由实现它。在HotSopt虚拟机中直接就把本地方法栈和Java栈合二为一。

**4.**堆

在C语言中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过malloc函数和free函数在堆上申请和释放空间。那么在Java中是怎么样的呢？

Java中的堆是用来存储对象本身的以及数组（当然，数组引用是存放在Java栈中的）。只不过和C语言中的不同，在Java中，程序员基本不用去关心空间释放的问题，Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。

**5.**方法区

方法区在JVM中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区 中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。

在Class文件中除了类的字段、方法、接口等描述信息外，还有一项信息是常量池，用来存储编译期间生成的字面量和符号引用。

在方法区中有一个非常重要的部分就是运行时常量池，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到JVM后，对应的运行时常量池就被创建出来。当然并非

Class文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如String的intern方法。

在JVM规范中，没有强制要求方法区必须实现垃圾回收。很多人习惯将方法区称为“永久代”， 是因为HotSpot虚拟机以永久代来实现方法区，从而JVM的垃圾收集器可以像管理堆区一样管理这部分区域，从而不需要专门为这部分设计垃圾回收机制。不过自从JDK7之后，Hotspot虚拟机便将运行时常量池从永久代移除了。

**JAVA**垃圾回收机制

如何判断一个对象是否已经死去？

* 1. 引用计数法，只要有一个地方引用它，计数器就加1，引用失效的话计数器减1。这样有一个坑爹的地方就是相互循环引用的问题
  2. 可达性分析：以“GC Roots”的对象作为起始点，从这些节点向下搜索，所走过的路径成为引用链（Reference Chain），当一个对象到GC Roots 没有任何引用连，则该对象不可达，可回收。

可作为**GC Roots**的对象包括哪些？

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中JNI（java native interface）引用的对象引用的分类：

强引用：通过new出来的引用，只要强引用还存在，则不会回收。

软引用：通过SoftReference类来实现，用来描述一些有用但非必须的对象。在系统将要发生内存溢出异常之前，会把这些对象回收了，如果这次回收还是内存不够的话，才抛出内存溢出异常。

弱引用：非必须对象，通过WeakReference类来实现，被弱引用引用的对象，只要已发生GC就会把它干掉。

虚引用：通过PhantomReference类来实现，无法通过徐引用获得对象的实例，唯一作用就是在这个对象被GC时会收到一个系统通知。

回收方法区**(**永久代**)**： 来存储class类、常量、方法描述

回收废弃常量，就是没有任何实例叫做这个常量

回收无用类，1.所有实例都被回收 2.加载该类的ClassLoader也被回收 3.该类对应的

java.lang.Class对象没有任何地方引用，无法在任何地方通过反射访问该类的方法。满足以上三个条件才回收该类

###### 垃圾回收算法

垃圾回收算法主要有一下四种：标记-清除（Mark-Sweep），复制算法（copying），标记-整理（Mark-Compact），分代收集算法

标记**-**清除算法

首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

缺点：1.效率问题，标记和清除两个过程效率都不高 2.空间问题，标记清除后会有大量的内存碎片

复制算法

它将内存容量划分为大小相等两块，每次只使用其中的一块。当一块内存用完了，就将还存活的对象复制到另一块上面，然后再把已使用过的内存空间一次清理掉。

现在商业虚拟机都采用这种手机算法来回收新生代，但斌不是1:1的比例来分配内存，而是一块较大的Eden空间，和两块较小的Survivor空间，每次使用Eden和其中一块

Survior。当回收时，将还存活的对象复制到另一块Survivor空间上，然后清理这两块空间，这过程中，当Survior空间不足时，回想老年代分配担保。HotSpot虚拟机采用的是8:1:1的比例。也就是浪费了10%的内存。

标记**-**整理算法

根据老年代的特点，有人提出了标记-整理算法，也是先标记出所有需要回收的对象，但后续步骤不是对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

它一般在老年代采用这种方法。分代收集算法

分代收集算法并没有提出新的思想，只是根据对象存活周期的不同将内存划为几块。一般Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用适当的收集算法。在新生袋中每次垃圾手机时都会由大批对象死去，只有少量存活，那就用复制算法，只需要付出少量存活对象的复制成本就可以。老年代中对象存活率高、没有额外担保，所以必须使用“标记-清理”或者“标记整理算法。

#### 垃圾收集器

垃圾收集算法是 内存回收的理论基础，而垃圾收集器就是内存回收的具体实现。下面介绍一下HotSpot（JDK 7)虚拟机提供的几种垃圾收集器，用户可以根据自己的需求组合出各个年代使用的收集器。

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。

1. 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。除了serial外，只有ParNew能跟CMS配合工作。因此，采用CMS作为老年代收集器的话，那么新生代你就只能选择ParNew和Serial，而不能选择比较新的Parallel Scavenge

1. 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。所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值。

1. 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。

1. 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。

1. G1

G1收集器是当今收集器技术发展最前沿的成果，它是一款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。

**Minor GC** 与**Major GC\/Full GC**

Minor GC：指发生在新生代的垃圾收集动作，因为java对象大多具备朝生夕灭的特性， 所以Minor GC非常频繁，一般回收速度也比较快，当Eden去满了的时候就会发生Minor

GC。

Full GC：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor

GC（但非绝对的，在Parallel Scavange收集器的手机策略中就有直接进行Major GC的策略选择过程）。Major GC的速度一般比Minor GC慢10倍以上。

对象分配规则

1.对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。

2.大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

3.长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1 次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。

4.动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

5.空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Minor GC,如果false则进行Full GC。

#### 一、序列化和反序列化的概念

把对象转换为字节序列的过程称为对象的序列化。 把字节序列恢复为对象的过程称为对象的反序列化。对象的序列化主要有两种用途：

1. 把对象的字节序列永久地保存到硬盘上，通常存放在一个文件中；
2. 在网络上传送对象的字节序列。

在很多应用中，需要对某些对象进行序列化，让它们离开内存空间，入住物理硬盘，以便长期保存。比如最常见的是Web服务器中的Session对象，当有 10万用户并发访问，就有可能出现10万个Session对象，内存可能吃不消，于是Web容器就会把一些seesion先序列化到硬盘中，等要用了，再把保存在硬盘中的对象还原到内存中。

当两个进程在进行远程通信时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象。

主要作用就是对象持久化， 除了在持久化对象时会用到对象序列化之外，当使用RMI(远程方法调用)，或在网络中传递对象时，都会用到对象序列化。

Transient关键字：禁止对该成员进行序列化

二、**JDK**类库中的序列化**API**

java.io.ObjectOutputStream代表对象输出流，它的writeObject(Object obj)方法可对参数指定的obj对象进行序列化，把得到的字节序列写到一个目标输出流中。

java.io.ObjectInputStream代表对象输入流，它的readObject()方法从一个源输入流中读取字节序列，再把它们反序列化为一个对象，并将其返回。

只有实现了Serializable和Externalizable接口的类的对象才能被序列化。Externalizable接口继承自 Serializable接口，实现Externalizable接口的类完全由自身来控制序列化的行为，而仅实现Serializable接口的类可以 采用默认的序列化方式 。

对象序列化包括如下步骤：

1. 创建一个对象输出流，它可以包装一个其他类型的目标输出流，如文件输出流；
2. 通过对象输出流的writeObject()方法写对象。对象反序列化的步骤如下：
3. 创建一个对象输入流，它可以包装一个其他类型的源输入流，如文件输入流；
4. 通过对象输入流的readObject()方法读取对象。

三、指定 **serialVersionUID** 的好处

如果为类增加或者删除对象以后，如果不指定 serialVersionUID 的话，那么系统自动生成serialVersionUID ，当你更改之后，又会生成另外一个 serialVersionUID ，导致你之前的流中的class与新生成的class SerialVersionUID 不符合，因此不让你加载。如果是指定的话就不会这样了。

#### 四、常见序列化协议

##### JSON

JavaScript中的序列化协议 就是键值对。JSONObject和JSONArray

Json的优点：明文结构一目了然，可以跨语言，属性的增加减少对解析端影响较小。缺点：字节数过多，依赖于不同的第三方类库。

##### Object Serializalbe

Object Serializalbe 优点：java原生支持，不需要提供第三方的类库，使用比较简单。缺点： 无法跨语言，字节数占用比较大，某些情况下对于对象属性的变化比较敏感。

对象在进行序列化和反序列化的时候，必须实现Serializable接口，但并不强制声明唯一的

serialVersionUID

是否声明serialVersionUID对于对象序列化的向上向下的兼容性有很大的影响。

##### XML

**ProtoBuf**

protocol buffers 是google内部得一种传输协议，目前项目已经开源

（http:\/\/code.google.com\/p\/protobuf\/）。它定义了一种紧凑得可扩展得二进制协议格式， 适合网络传输，并且针对多个语言有不同得版本可供选择。 google protobuf 优点：字节数很小，适合网络传输节省io，跨语言 。缺点：需要依赖于工具生成代码。

**Avro Hadoop**的序列化协议

三种方式对比传输同样的数据，google protobuf只有53个字节是最少的。结论：

|  |  |  |
| --- | --- | --- |
| 方式 | 优点 | 缺点 |
| JSON | 跨语言、格式清晰一目了然 | 字节数比较大，需要第三方类库 |
| Object Serialize | java原生方法不依赖外部类库 | 字节数比较大，不能跨语言 |
| Google protobuf | 跨语言、字节数比较少 | 编写.proto配置用protoc工具生成对应的代码 |

**JAVA**对象创建的过程

虚拟机遇到new指令时，首先先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没

有，就执行类加载过程。

为对象分配空间，把一块固定大小的内存从java堆里划分出来。如果java堆是那种齐整 的，一边放已经用的内存，另一边放没有用的内存，这样就有一个中间指针隔开两个区域，只要把这个指针移动到相应位置即可，这种方法叫做“指针碰撞”。如果已使用内存和未使用内存交错，就需要一个“空闲列表”方法来决定分配在哪里。

多线程并发时会出现正在给对象A分配内存，还没来得急修改指针，对象B又用这个指针分配内存，这样就出现问题了。解决这种问题有两种方案：1.对分配内存空间的动作进行同步；2.把内存分配动作按照线程划分在不同的空间之中进行，即每个线程在java对中预先分配一小块内存成为本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）。可以通过-XX:+\/-UseTLAB参数决定

内存分配完之后虚拟机将分配到的内存全都初始化为零值。接下来设置对象头，包括这个对象时哪个类的实例，如何才能找到该类的元数据信息，对象的哈希码，对象的GC分代年龄等信息。

执行init方法，初始化

#### 对象的内存布局

对象的内存布局包括三个部分：对象头，实例数据和对齐填充。

对象头：对象头包括两部分信息，第一部分是存储对象自身的运行时数据，如哈希码，

GC分代年龄，锁状态标志，线程持有的锁等等。第二部分是类型指针，即对象指向类元数据的指针。

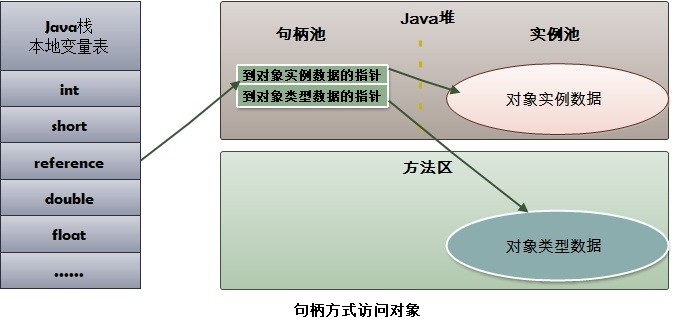
实例数据：就是数据啦

对齐填充：不是必然的存在，就是为了对齐的嘛

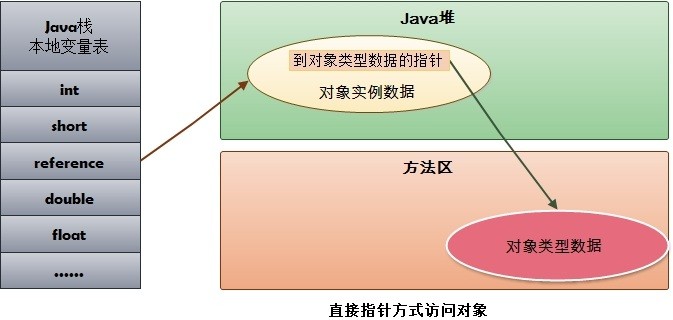
#### 对象定位访问

对象的访问定位有两种：句柄定位和直接指针

句柄定位：java堆会画出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息



直接指针访问：java堆对象的不居中就必须考虑如何放置访问类型数据的相关信息，而

reference中存储的直接就是对象地址

###### 比较：

使用直接指针就是速度快，使用句柄reference指向稳定的句柄，对象被移动改变的也只是句柄中实例数据的指针，而reference本身并不需要修改。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载，连接

（验证，准备，解析），

初始化，使用，卸载。

###### **5**种情况下必须对类进行初始化：

1.遇到new，getstatic,putstatic,invokestatic这4条字节码时候，如果类没有进行初始化，则需要先触发其初始化

2.使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化

3.当初始化一个类的时候，发现它的父类还没有初始化，则先触发父类的初始化

4.当虚拟机启动时，用户需要指定一个要执行的主类（包含main（）方法的类），虚拟机会先初始化这个类

5.当使用JDK1.7的动态语言支持时候，如果一个java.lang.invoke.MethodHandle实例最后解析结果为REF\_getstatic,

REF\_putstatic,REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化

###### 加载器分类

从java虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器（bootstrap classloader），这个类加载器

使用C++语言实现的，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类都是有java语言实现的，独立于

虚拟机外部，并且 全部继承自抽象类java.lang.ClassLoader。

在加载阶段，**java**虚拟机需要完成以下**3**件事：

a.通过一个类的全限定名来获取定义此类的二进制字节流。 b.将定义类的二进制字节流所代表的静态存储结构转换为方法区的运行时数据结构。

c.在java堆中生成一个代表该类的java.lang.Class对象，作为方法区数据的访问入口。从Java开发人员的角度来看分为3种类加载器：

###### 类加载器按照层次，从顶层到底层，分为以下三种：

（1）启动类加载器（Bootstrap ClassLoader）

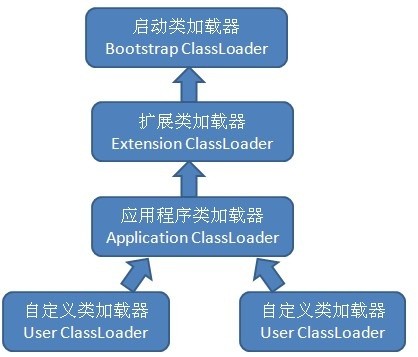
这个类加载器负责将存放在JAVA\_HOME\/lib下的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用。

（2）扩展类加载器（Extension ClassLoader）

这个加载器负责加载JAVA\_HOME\/lib\/ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器

（3）应用程序类加载器（Application ClassLoader）

这个加载器是ClassLoader中getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（Classpath）上所指定的类库，可直接使用这个加载器， 如果应用程序没有自定义自己的类加载器，一般情况下这个就是程序中默认的类加载器



###### 类加载的双亲委派模型

双亲委派模型要求除了顶层的启动类加载器外，其他的类加载器都应当有自己的父类加载 器。这里类加载器之间的父子关系一般不会以继承关系来实现，而是都使用组合关系来复用父加载器的代码

工作过程：

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传递到顶层的启动类加载器中，

只有当父类加载器反馈自己无法完成这个请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载

好处：

Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类Object，它放在

rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给启动类加载器进行加载，因此

Object类在程序的各种类加载器环境中都是同一个类

判断两个类是否相同是通过classloader.class这种方式进行的，所以哪怕是同一个class文件如果被两个classloader加载，那么他们也是不同的类

###### 实现自己的加载器

只需要继承ClassLoader，并覆盖findClass方法。

在调用loadClass方法时，会先根据委派模型在父加载器中加载，如果加载失败，则会调用自己的findClass方法来完成加载

参考：[http:\/\/blog.csdn.net\/altchen\/article\/details\/328698](http://blog.csdn.net/altchen/article/details/328698) 类加载器如何识别

通过findLoadedClass来识别是否已经加载某个类，这个方法是findLoadedClass0的一个包装类，而findLoadedClass0是一个native方法

**Java NIO**：**NIO**概述

在上一篇博文中讲述了几种IO模型，现在我们开始进入Java NIO编程主题。NIO是Java 4里面提供的新的API，目的是用来解决传统IO的问题。

###### 同步与异步的理解：

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系统在调用系统调用的时候函数的实现方式而已。

Selector模式是同步非阻塞IO， 我认为Selector不是异步的。因为它对IO的读写还是同步阻塞的。只是通过线程复用，将IO的准备时间分离出来。真的进行IO时，还等等。

一**.NIO**中的几个基础概念

在NIO中有几个比较关键的概念：Channel（通道），Buffer（缓冲区），Selector（选择器）。

首先从Channel说起吧，通道，顾名思义，就是通向什么的道路，为某个提供了渠道。在传统

IO中，我们要读取一个文件中的内容，通常是像下面这样读取的：

public class Test {

public static void main(String[] args) throws IOException { File file = new File("data.txt");

InputStream inputStream = new FileInputStream(file); byte[] bytes = new byte[1024]; inputStream.read(bytes);

inputStream.close();

}

}

这里的InputStream实际上就是为读取文件提供一个通道的。

因此可以将NIO 中的Channel同传统IO中的Stream来类比，但是要注意，传统IO中，Stream 是单向的，比如InputStream只能进行读取操作，OutputStream只能进行写操作。而**Channel** 是双向的，既可用来进行读操作，又可用来进行写操作。

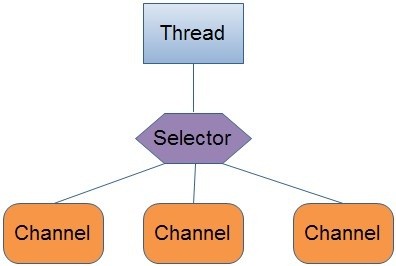
Buffer（缓冲区），是NIO中非常重要的一个东西，在NIO中所有数据的读和写都离不开

Buffer。比如上面的一段代码中，读取的数据时放在byte数组当中，而在**NIO**中，读取的数据只能放在**Buffer**中。同样地，写入数据也是先写入到Buffer中。

下面介绍一下NIO中最核心的一个东西：Selector。可以说它是NIO中最关键的一个部分，

**Selector**的作用就是用来轮询每个注册的**Channel**，一旦发现**Channel**有注册的事件发生，便获取事件然后进行处理。

比如看下面的这个例子：



用单线程处理一个Selector，然后通过Selector.select()方法来获取到达事件，在获取了到达事件之后，就可以逐个地对这些事件进行响应处理。

### 二.Channel

在前面已经提到，Channel和传统IO中的Stream很相似。虽然很相似，但是有很大的区别， 主要区别为：通道是双向的，通过一个Channel既可以进行读，也可以进行写；而Stream只能进行单向操作，通过一个Stream只能进行读或者写；

以下是常用的几种通道：

FileChannel SocketChanel ServerSocketChannel DatagramChannel

通过使用FileChannel可以从文件读或者向文件写入数据；通过SocketChannel，以TCP 来向网络连接的两端读写数据；通过ServerSocketChanel能够监听客户端发起的TCP连接，并为每个TCP连接创建一个新的SocketChannel来进行数据读写；通过

DatagramChannel，以UDP协议来向网络连接的两端读写数据。

下面给出通过FileChannel来向文件中写入数据的一个例子：

public class Test {

public static void main(String[] args) throws IOException { File file = new File("data.txt");

FileOutputStream outputStream = new FileOutputStream(file); FileChannel channel = outputStream.getChannel(); ByteBuffer buffer = ByteBuffer.allocate(1024);

String string = "java nio"; buffer.put(string.getBytes());

buffer.flip(); //此处必须要调用buffer的flip方法

channel.write(buffer); channel.close(); outputStream.close();

}

}

通过上面的程序会向工程目录下的data.txt文件写入字符串"java nio"，注意在调用channel的

write方法之前必须调用buffer的flip方法，否则无法正确写入内容，至于具体原因将在下篇博文中具体讲述Buffer的用法时阐述。

### 三.Buffer

Buffer，故名思意，缓冲区，实际上是一个容器，是一个连续数组。Channel提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由Buffer。具体看下面这张图就理解了：



上面的图描述了从一个客户端向服务端发送数据，然后服务端接收数据的过程。客户端发送数据时，必须先将数据存入Buffer中，然后将Buffer中的内容写入通道。服务端这边接收数据必须通过Channel将数据读入到Buffer中，然后再从Buffer中取出数据来处理。

在NIO中，Buffer是一个顶层父类，它是一个抽象类，常用的Buffer的子类有：

ByteBuffer IntBuffer CharBuffer LongBuffer DoubleBuffer FloatBuffer ShortBuffer

如果是对于文件读写，上面几种Buffer都可能会用到。但是对于网络读写来说，用的最多的是ByteBuffer。

关于Buffer的具体使用以及它的limit、posiion和capacity这几个属性的理解在下一篇文章中讲述。

### 四.Selector

Selector类是NIO的核心类，Selector能够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。这样使得只有在连接真正有读写事件发生时， 才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线 程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销。

与Selector有关的一个关键类是SelectionKey，一个SelectionKey表示一个到达的事件，这2个类构成了服务端处理业务的关键逻辑。

关于Selector类的具体使用将在后续文章中阐述。

#### 五**.**关于阻塞与非阻塞，同步与非同步的理解

我们都知道常见的IO有四种方式，同步阻塞，同步非阻塞，异步阻塞，异步非阻塞。然而对于同步和阻塞的理解却一直很模糊。

我个人的理解：所谓同步，是对于多个进程或者多个线程之间而言的，他们是否能够并发执行，如果能并发执行，就说明这两个线程是同步的，如果不能并发执行，就是非同步的，它的对象是对于多个线程之间而非自身而言的。所谓阻塞，就是指对于单个进程或者线程而言的，如果执行完这个操作，是否会有一直等待的情况。

在于多个任务和事件发生时，一个事件的发生或执行是否会导致整个流程的暂时等待。我觉得可以将同步和异步与Java中的synchronized关键字联系起来进行类比。当多个线程同时访问一个变量时，每个线程访问该变量就是一个事件，对于同步来说，就是这些线程必须逐个地来访问该变量，一个线程在访问该变量的过程中，其他线程必须等待；而对于异步来说， 就是多个线程不必逐个地访问该变量，可以同时进行访问。

阻塞和非阻塞的区别关键在于当发出请求一个操作时，如果条件不满足，是会一直等待还是返回一个标志信息。

六**.**什么是阻塞**IO**？什么是非阻塞**IO**？

在了解阻塞IO和非阻塞IO之前，先看下一个具体的IO操作过程是怎么进行的。

通常来说，IO操作包括：对硬盘的读写、对socket的读写以及外设的读写。

当用户线程发起一个IO请求操作（本文以读请求操作为例），内核会去查看要读取的数据是否就绪，对于阻塞IO来说，如果数据没有就绪，则会一直在那等待，直到数据就绪；对于非阻塞IO来说，如果数据没有就绪，则会返回一个标志信息告知用户线程当前要读的数据没有就绪。当数据就绪之后，便将数据拷贝到用户线程，这样才完成了一个完整的IO读请求操 作，也就是说一个完整的IO读请求操作包括两个阶段：

1）查看数据是否就绪；

2）进行数据拷贝（内核将数据拷贝到用户线程）。

那么阻塞（blocking IO）和非阻塞（non-blocking IO）的区别就在于第一个阶段，如果数据没有就绪，在查看数据是否就绪的过程中是一直等待，还是直接返回一个标志信息。

Java中传统的IO都是阻塞IO，比如通过socket来读数据，调用read()方法之后，如果数据没有就绪，当前线程就会一直阻塞在read方法调用那里，直到有数据才返回；而如果是非阻塞IO 的话，当数据没有就绪，read()方法应该返回一个标志信息，告知当前线程数据没有就绪，而不是一直在那里等待。

七**.**什么是同步**IO**？什么是异步**IO**？

我们先来看一下同步IO和异步IO的定义，在《Unix网络编程》一书中对同步IO和异步IO的定义是这样的：

A synchronous I\/O operation causes the requesting process to be blocked until that I\/O operation completes.

**An asynchronous I\/O operation does not cause the requesting process to be blocked.**

从字面的意思可以看出：同步IO即 如果一个线程请求进行IO操作，在IO操作完成之前，该线程会被阻塞；

而异步IO为 如果一个线程请求进行IO操作，IO操作不会导致请求线程被阻塞。事实上，同步IO和异步IO模型是针对用户线程和内核的交互来说的：

对于同步IO：当用户发出IO请求操作之后，如果数据没有就绪，需要通过用户线程或者内核不断地去轮询数据是否就绪，当数据就绪时，再将数据从内核拷贝到用户线程；

而异步IO：只有IO请求操作的发出是由用户线程来进行的，IO操作的两个阶段都是由内核自动完成，然后发送通知告知用户线程IO操作已经完成。也就是说在异步IO中，不会对用户线程产生任何阻塞。

这是同步IO和异步IO关键区别所在，同步IO和异步IO的关键区别反映在数据拷贝阶段是由用户线程完成还是内核完成。所以说异步IO必须要有操作系统的底层支持。

注意同步IO和异步IO与阻塞IO和非阻塞IO是不同的两组概念。

阻塞IO和非阻塞IO是反映在当用户请求IO操作时，如果数据没有就绪，是用户线程一直等待数据就绪，还是会收到一个标志信息这一点上面的。也就是说，阻塞IO和非阻塞IO是反映在

IO操作的第一个阶段，在查看数据是否就绪时是如何处理的。

八**.**多路复用**IO**模型

多路复用IO模型是目前使用得比较多的模型。Java NIO实际上就是多路复用IO。

在多路复用IO模型中，会有一个线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。因为在多路复用IO模型中，只需要使用一个线程就可以管理多个socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有socket读写事件进行时，才会使用IO资源，所以它大大减少了资源占用。

在Java NIO中，是通过selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。

也许有朋友会说，我可以采用 多线程+ 阻塞IO 达到类似的效果，但是由于在多线程 + 阻塞IO 中，每个socket对应一个线程，这样会造成很大的资源占用，并且尤其是对于长连接来说，线程的资源一直不会释放，如果后面陆续有很多连接的话，就会造成性能上的瓶颈。

而多路复用IO模式，通过一个线程就可以管理多个socket，只有当socket真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用IO比较适合连接数比较多的情况。

另外多路复用IO为何比非阻塞IO模型的效率高是因为在非阻塞IO中，不断地询问socket状态时通过用户线程去进行的，而在多路复用IO中，轮询每个socket状态是内核在进行的，这个效率要比用户线程要高的多。

不过要注意的是，多路复用IO模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用IO模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

自己总结

NIO其实实现的是一个IO的多路复用，用select来同时监听多个channel，本质上还是同步阻塞的，需要select不断监听端口。

但是对于IO各个通道来说就是可以看做是异步

AIO本质上就是异步通信的，发出IO请求之后，立刻返回结果，但是不知道是否成功等信息。

输入流复用通过两种方式：BufferedInputStream 和ByteArrayInputStream

BufferedInputStream 是通过设置标志位，可以利用InputStream本身的mark和reset方法，在用过一次之后将InputStream恢复。

而ByteArrayInputStream把InputStream流内的数据读到数组里，然后根据数组再构造一个

ByteArrayInputStream输入流

文件通道：FileChannel创建方法可以用InputStream，OutputStream对象的getChannel方法获得，也可以用FileChannel.open()方法获取，

在open中可以规定读，写，创建等关键字。Scattering，GatheringByteChannel接口，是代表可以分散和聚集，也就是可以写入，读取一个对象数组。

FileChannel文件传输可以用transferFrom 和transferTo来进行文件间和各种其他通道间的数据传输。

可以使用内存映射文件，将文件映射到内存当中（MappedByteBuffer）,这样读写文件的操作就可以用读写内存来做，这在处理大文件操作的时候就比较方便，当然，这需要更大的内

存。

套接字通道：传统的socket和serversocket在传输数据和建立连接时都是阻塞式的。也就是说如果操作没完成，当前线程会处于等待状态。

处理这种问题方法是单独开一个线程来执行accept方法来监听连接请求，一旦连接一建立就会创建一个新的线程来处理这个请求。

void run(){

Socket socket = ss.accept();

new Thread(new Handle(socket)).start();

}\/\/采用多线程来处理

JAVA NIO中提供了非阻塞式和多路复用的套接字连接。

FileChannel在java 7中有所更新，增加了FileChannel.open()方法，而之前只能通过inputstream或者outputstream的getChannel()方法来创建。

多路复用套接字通道：

NIO.2中的异步套接字：

Java使用Class对象来执行其RTTI。类是程序的一部分，每个类都有一个Class对象，换言 之，每当编写并且编译了一个新类，就会产生一个Class对象（准确地说，是保存在一个同名的.class文件中了）。为了生成这个类的对象，运行这个程序的JVM将使用被称为类加载器的子系统。

无论何时,只要你想在运行时获得类型信息,就必须首先获得对恰当的Class对象的引用，

Class.forName()就是实现此功能的便捷途径,因为你不需要持有该类的对象。但是如果你已经有对象了，就直接用对象调用.getClass()方法就可以了。

比较判断一个对象是否是一个类?

1.a instanceof B 判断a对象是不是B类，或者是B类的派生类，都返回true 2.a.getClass() == B 判断，只有是同一类型的才可以返回true

#### 反射：运行时的类信息

RMI(远程方法调用): 它允许一个Java程序讲对象分布在多台机器上，在跨网络的远程平台上创建和运行对象。

Class类和java.lang.reflect类库对反射的概念进行了支持,给类库包含了Field，Method以及Constructor类（每个类都是先了Member接口）。这些类型的对象时由JVM在运行时创建的， 用以表示未知类里对应的成员。这样你就可以使用Constructor来创建新的对象，用get()和

set()方法读取和修改Field对象关联的字段，用invoke()方法调用与Method对象相关联的方

法。另外，可以使用getFileds(),getMethods(),getConstructors()，等函数来返回表示字段，方法以及构造器的对象数组。

RTTI与反射的区别:

对RTTI来说,编译器在编译时打开和检查.class文件（换句话说，我们可以用普通方式调用对象的所有方法）。而对于反射来说，.class文件在编译时是不可获取的，所以在运行时打开和检查.class文件。

###### 利用反射机制进行远程方法调用？

客户端要远程调用服务器端的某个方法，然后得到返回值。客户端可以通过一个中间对象我们就叫他Call类，里面包含调用类的名称，方法名称，方法参数类型，方法参数等信息。然后把这个对象传送给服务器端。服务器端接收到这些信息，根据这些信息利用反射调用该方

法，得到返回值，把返回值再发给客户端。

#### 网上总结

在Java运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法？答案是肯定的。这种动态获取类的信息，以及动态调用对象的方法的功能来自于Java语言的反射（Reflection）机制。Java反射机制主要提供了以下功能：

* 在运行时判断任意一个对象所属的类；
* 在运行时构造任意一个类的对象；
* 在运行时判断任意一个类所具有的成员变量和方法；
* 在运行时调用任意一个对象的方法；
* 生成动态代理。

本章首先介绍了Java Reflection API的用法，然后介绍了一个远程方法调用的例子，在这个例子中客户端能够远程调用服务器端的一个对象的方法。服务器端采用了反射机制提供的动态调用方法的功能，而客户端则采用了反射机制提供的动态代理功能。

* 1. Java Reflection API简介

在JDK中，主要由以下类来实现Java反射机制，这些类都位于java.lang.reflect包中。

* Class类：代表一个类。
* Field类：代表类的成员变量（成员变量也称为类的属性）。
* Method类：代表类的方法。
* Constructor类：代表类的构造方法。
* Array类：提供了动态创建数组，以及访问数组元素的静态方法。

如例程10-1所示DumpMethods类演示了Reflection API的基本作用，它读取命令行参数指定的类名，然后打印这个类所具有的方法信息：

例程10-1 DumpMethods.java

import java.lang.reflect.\*; public class DumpMethods {

public static void main(String args[]) throws Exception{

//加载并初始化命令行参数指定的类

Class classType = Class.forName(args[0]);

//获得类的所有方法

Method methods[] = classType.getDeclaredMethods(); for(int i = 0; i < methods.length; i++)

System.out.println(methods[i].toString());

}

}

运行命令“java DumpMethods java.util.Stack”，就会显示java.util.Stack类所具有的方法，程序的打印结果如下：

public synchronized java.lang.Object java.util.Stack.pop() public java.lang.Object java.util.Stack.push(java.lang.Object) public boolean java.util.Stack.empty()

public synchronized java.lang.Object java.util.Stack.peek() public synchronized int java.util.Stack.search(java.lang.Object)

如例程10-2所示ReflectTester类进一步演示了Reflection API的基本使用方法。ReflectTester 类有一个copy(Object object)方法，这个方法能够创建一个和参数object同样类型的对象，然后把object对象中的所有属性复制到新建的对象中，并将它返回。

这个例子只能复制简单的JavaBean，假定JavaBean的每个属性都有public类型的getXXX()和

setXXX()方法。

例程10-2 ReflectTester.java

//调用原对象的getXXX()方法

Object value=getMethod.invoke(object,new Object[]{});

System.out.println(fieldName+":"+value);

;

String fieldName=field.getName();

String firstLetter=fieldName.substring(0,1).toUpperCase();

//获得和属性对应的getXXX()方法的名字

String getMethodName="get"+firstLetter+fieldName.substring(1);

//获得和属性对应的setXXX()方法的名字

String setMethodName="set"+firstLetter+fieldName.substring(1);

//获得和属性对应的getXXX()方法

Method getMethod=classType.getMethod(getMethodName,new Class[]{});

//获得和属性对应的setXXX()方法

Method setMethod=classType.getMethod(setMethodName,new Class[]{field.getType()})

Field field=fields[i];

for(int i=0; i

import java.lang.reflect.\*; public class ReflectTester {

public Object copy(Object object) throws Exception{

//获得对象的类型

Class classType=object.getClass(); System.out.println("Class:"+classType.getName());

//通过默认构造方法创建一个新的对象

Object objectCopy=classType.getConstructor(new Class[]{}). newInstance(new Object[]{});

//获得对象的所有属性

Field fields[]=classType.getDeclaredFields();

//调用复制对象的setXXX()方法

etMethod.invoke(objectCopy,new Object[]{value});

}

return objectCopy;

}

public static void main(String[] args) throws Exception{ Customer customer=new Customer("Tom",21); customer.setId(new Long(1));

Customer customerCopy=(Customer)new ReflectTester().copy(customer); System.out.println("Copy information:"+customerCopy.getName()+""+

customerCopy.getAge());

}

}

class Customer{ //Customer类是一个JavaBean private Long id;

private String name; private int age;

public Customer(){}

public Customer(String name,int age){ this.name=name;

this.age=age;

}

public Long getId(){return id;}

public void setId(Long id){this.id=id;}

public String getName(){return name;}

public void setName(String name){this.name=name;}

public int getAge(){return age;}

public void setAge(int age){this.age=age;}

}

#p#

ReflectTester类的copy(Object object)方法依次执行以下步骤。

（1）获得对象的类型：

Class classType=object.getClass(); System.out.println("Class:"+classType.getName());

在java.lang.Object类中定义了getClass()方法，因此对于任意一个Java对象，都可以通过此方法获得对象的类型。Class类是Reflection API中的核心类，它有以下方法。

* getName()：获得类的完整名字。
* getFields()：获得类的public类型的属性。
* getDeclaredFields()：获得类的所有属性。
* getMethods()：获得类的public类型的方法。
* getDeclaredMethods()：获得类的所有方法。
* getMethod(String name, Class[] parameterTypes)：获得类的特定方法，name参数指定方法的名字，parameterTypes参数指定方法的参数类型。
* getConstrutors()：获得类的public类型的构造方法。
* getConstrutor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes参数指定构造方法的参数类型。
* newInstance()：通过类的不带参数的构造方法创建这个类的一个对象。

（2）通过默认构造方法创建一个新的对象：

**Object objectCopy=classType.getConstructor(new Class[]{}).newInstance(new Object[]{});**

以上代码先调用Class类的getConstructor()方法获得一个Constructor对象，它代表默认的构造方法，然后调用Constructor对象的newInstance()方法构造一个实例。

（3）获得对象的所有属性：

**Field fields[]=classType.getDeclaredFields();**

Class类的getDeclaredFields()方法返回类的所有属性，包括public、protected、默认和

private访问级别的属性。

（4）获得每个属性相应的getXXX()和setXXX()方法，然后执行这些方法，把原来对象的属性复制到新的对象中：

for(int i=0; i Field field=fields[i];

String fieldName=field.getName();

String firstLetter=fieldName.substring(0,1).toUpperCase();

//获得和属性对应的getXXX()方法的名字

String getMethodName="get"+firstLetter+fieldName.substring(1);

//获得和属性对应的setXXX()方法的名字

String setMethodName="set"+firstLetter+fieldName.substring(1);

//获得和属性对应的getXXX()方法

Method getMethod=classType.getMethod(getMethodName,new Class[]{});

//获得和属性对应的setXXX()方法

Method setMethod=classType.getMethod(setMethodName,new Class[]{field.getType()});

//调用原对象的getXXX()方法

Object value=getMethod.invoke(object,new Object[]{}); System.out.println(fieldName+":"+value);

//调用复制对象的setXXX()方法

setMethod.invoke(objectCopy,new Object[]{value});

}

以上代码假定每个属性都有相应的getXXX()和setXXX()方法，并且在方法名

中，“get”和“set”的后面一个字母为大写。例如，Customer类的name属性对应getName()和

setName()方法。Method类的invoke(Object obj,Object args[])方法用于动态执行一个对象的特定方法，它的第一个obj参数指定具有该方法的对象，第二个args参数指定向该方法传递的参数。

如例程10-3所示的InvokeTester类的main()方法中，运用反射机制调用一个InvokeTester对象的add()和echo()方法。

例程10-3 InvokeTester.java

import java.lang.reflect.\*; public class InvokeTester {

public int add(int param1,int param2){ return param1+param2;

}

public String echo(String msg){ return "echo:"+msg;

}

public static void main(String[] args) throws Exception{ Class classType=InvokeTester.class;

Object invokeTester=classType.newInstance();

//调用InvokeTester对象的add()方法

Method addMethod=classType.getMethod("add",new Class[]{int.class,int.class}); Object result=addMethod.invoke(invokeTester,

new Object[]{new Integer(100),new Integer(200)}); System.out.println((Integer)result);

//调用InvokeTester对象的echo()方法

Method echoMethod=classType.getMethod("echo",new Class[]{String.class}); result=echoMethod.invoke(invokeTester,new Object[]{"Hello"}); System.out.println((String)result);

}

}

add()方法的两个参数为int类型，获得表示add()方法的Method对象的代码如下：

**Method addMethod=classType.getMethod("add",new Class[]{int.class,int.class});**

Method类的invoke(Object obj,Object args[])方法接收的参数必须为对象，如果参数为基本类型数据，必须转换为相应的包装类型的对象。invoke()方法的返回值总是对象，如果实际被调用的方法的返回类型是基本类型数据，那么invoke()方法会把它转换为相应的包装类型的对 象，再将其返回。

在本例中，尽管InvokeTester类的add()方法的两个参数及返回值都是int类型，调用

addMethod对象的invoke()方法时，只能传递Integer类型的参数，并且invoke()方法的返回类型也是Integer类型，Integer类是int基本类型的包装类：

Object result=addMethod.invoke(invokeTester,

new Object[]{new Integer(100),new Integer(200)}); System.out.println((Integer)result); //result为Integer类型

java.lang.Array类提供了动态创建和访问数组元素的各种静态方法。如例程10-4所示的

ArrayTester1类的main()方法创建了一个长度为10的字符串数组，接着把索引位置为5的元素设为“hello”，然后再读取索引位置为5的元素的值。

#p#

例程10-4 ArrayTester1.java

import java.lang.reflect.\*; public class ArrayTester1 {

public static void main(String args[])throws Exception { Class classType = Class.forName("java.lang.String");

//创建一个长度为10的字符串数组

Object array = Array.newInstance(classType, 10);

//把索引位置为5的元素设为"hello" Array.set(array, 5, "hello");

//读取索引位置为5的元素的值

String s = (String) Array.get(array, 5); System.out.println(s);

}

}

如例程10-5所示的ArrayTester2类的main()方法创建了一个5×10×15的整型数组，并把索引位置为[3][5][10]的元素的值为设37。

例程10-5 ArrayTester2.java

import java.lang.reflect.\*; public class ArrayTester2{

public static void main(String args[]) { int dims[] = new int[]{5, 10, 15};

Object array = Array.newInstance(Integer.TYPE, dims);

//使arrayObj引用array[3]

Object arrayObj = Array.get(array, 3);

Class cls = arrayObj.getClass().getComponentType(); System.out.println(cls);

//使arrayObj引用array[3][5]

arrayObj = Array.get(arrayObj, 5);

//把元素array[3][5][10]设为37 Array.setInt(arrayObj, 10, 37);

int arrayCast[][][] = (int[][][]) array; System.out.println(arrayCast[3][5][10]);

}

}

* 1. 在远程方法调用中运用反射机制

假定在SimpleServer服务器端创建了一个HelloServiceImpl对象，它具有getTime()和echo()方法。HelloServiceImpl类实现了HelloService接口。如例程10-6和例程10-7所示分别是

HelloService接口和HelloServiceImpl类的源程序。

例程10-6 HelloService.java

package remotecall; import java.util.Date;

public interface HelloService{ public String echo(String msg); public Date getTime();

}

例程**10-7 HelloServiceImpl.java**

package remotecall; import java.util.Date;

public class HelloServiceImpl implements HelloService{ public String echo(String msg){

return "echo:"+msg;

}

public Date getTime(){ return new Date();

}

}

SimpleClient客户端如何调用服务器端的HelloServiceImpl对象的getTime()和echo()方法呢？ 显然，SimpleClient客户端需要把调用的方法名、方法参数类型、方法参数值，以及方法所属的类名或接口名发送给SimpleServer，SimpleServer再调用相关对象的方法，然后把方法的返回值发送给SimpleClient。

为了便于按照面向对象的方式来处理客户端与服务器端的通信，可以把它们发送的信息用Call 类（如例程10-8所示）来表示。一个Call对象表示客户端发起的一个远程调用，它包括调用的类名或接口名、方法名、方法参数类型、方法参数值和方法执行结果。

例程10-8 Call.java

package remotecall; import java.io.\*;

public class Call implements Serializable{ private String className; //表示类名或接口名private String methodName; //表示方法名private Class[] paramTypes; //表示方法参数类型

private Object[] params; //表示方法参数值

//表示方法的执行结果

//如果方法正常执行，则result为方法返回值，如果方法抛出异常，那么result为该异常。

private Object result;

public Call(){}

public Call(String className,String methodName,Class[] paramTypes,

Object[] params){ this.className=className; this.methodName=methodName; this.paramTypes=paramTypes; this.params=params;

}

public String getClassName(){return className;}

public void setClassName(String className){this.className=className;}

public String getMethodName(){return methodName;}

public void setMethodName(String methodName){this.methodName=methodName;}

public Class[] getParamTypes(){return paramTypes;}

public void setParamTypes(Class[] paramTypes){this.paramTypes=paramTypes;}

public Object[] getParams(){return params;}

public void setParams(Object[] params){this.params=params;}

public Object getResult(){return result;}

public void setResult(Object result){this.result=result;}

public String toString(){

return "className="+className+" methodName="+methodName;

}

}

SimpleClient调用SimpleServer端的HelloServiceImpl对象的echo()方法的流程如下。

1. SimpleClient创建一个Call对象，它包含了调用HelloService接口的echo()方法的信息。
2. SimpleClient通过对象输出流把Call对象发送给SimpleServer。
3. SimpleServer通过对象输入流读取Call对象，运用反射机制调用HelloServiceImpl对象的

echo()方法，把echo()方法的执行结果保存到Call对象中。

1. SimpleServer通过对象输出流把包含了方法执行结果的Call对象发送给SimpleClient。
2. SimpleClient通过对象输入流读取Call对象，从中获得方法执行结果。

#p#

如例程10-9和例程10-10所示分别是SimpleServer和SimpleClient的源程序。例程**10-9 SimpleServer.java**

package remotecall; import java.io.\*; import java.net.\*; import java.util.\*;

import java.lang.reflect.\*; public class SimpleServer {

private Map remoteObjects=new HashMap(); //存放远程对象的缓存

/\*\* 把一个远程对象放到缓存中 \*/

public void register(String className,Object remoteObject){ remoteObjects.put( className,remoteObject);

}

public void service()throws Exception{

ServerSocket serverSocket = new ServerSocket(8000);

System.out.println("服务器启动."); while(true){

Socket socket=serverSocket.accept(); InputStream in=socket.getInputStream(); ObjectInputStream ois=new ObjectInputStream(in); OutputStream out=socket.getOutputStream();

ObjectOutputStream oos=new ObjectOutputStream(out);

Call call=(Call)ois.readObject(); //接收客户发送的Call对象System.out.println(call);

call=invoke(call); //调用相关对象的方法oos.writeObject(call); //向客户发送包含了执行结果的Call对象

ois.close();

oos.close(); socket.close();

}

}

public Call invoke(Call call){ Object result=null;

try{

String className=call.getClassName(); String methodName=call.getMethodName(); Object[] params=call.getParams();

Class classType=Class.forName(className); Class[] paramTypes=call.getParamTypes();

Method method=classType.getMethod(methodName,paramTypes);

Object remoteObject=remoteObjects.get(className); //从缓存中取出相关的远程对象if(remoteObject==null){

}

}

public static void main(String args[])throws Exception { SimpleServer server=new SimpleServer();

//把事先创建的HelloServiceImpl对象加入到服务器的缓存中

server.register("remotecall.HelloService",new HelloServiceImpl()); server.service();

}

//设置方法执行结果

call.setResult(result);

return call;

throw new Exception(className+"的远程对象不存在");

}else{

result=method.invoke(remoteObject,params);

}

}catch(Exception e){result=e;}

例程10-10 SimpleClient.java

package remotecall; import java.io.\*; import java.net.\*; import java.util.\*;

public class SimpleClient {

public void invoke()throws Exception{

Socket socket = new Socket("localhost",8000); OutputStream out=socket.getOutputStream(); ObjectOutputStream oos=new ObjectOutputStream(out); InputStream in=socket.getInputStream(); ObjectInputStream ois=new ObjectInputStream(in);

//Call call=new Call("remotecall.HelloService","getTime", new Class[]{},new Object[]{});

Call call=new Call("remotecall.HelloService","echo", new Class[]{String.class},new Object[]{"Hello"});

oos.writeObject(call); //向服务器发送Call对象

call=(Call)ois.readObject(); //接收包含了方法执行结果的Call对象

System.out.println(call.getResult());

ois.close();

oos.close(); socket.close();

}

public static void main(String args[])throws Exception { new SimpleClient().invoke();

}

}

先运行命令“java remotecall.SimpleServer”，再运行命令“java remotecall. SimpleClient”，

SimpleClient端将打印“echo:Hello”。该打印结果是服务器端执行HelloServiceImpl对象的

echo()方法的返回值。如图10-1所示显示了SimpleClient与SimpleServer的通信过程。

图10-1 SimpleClient与SimpleServer的通信过程

相关文章链接：[Java网络编程精解之Java语言的反射机制二](http://developer.51cto.com/art/200702/40215.htm)

#### 什么是动态代理**,**为什么要用代理模式？

传给动态代理类一个Object，就可以调用这个Object的类方法。主要用来做方法的增强，让你可以在不修改源码的情况下，增强一些方法，在方法执行前后做任何你想做的事情（甚至根本不去执行这个方法），因为在***InvocationHandler***的invoke方法中，你可以直接获取正在调用方法对应的***Method***对象，具体应用的话，比如可以添加调用日志，做事务控制等。

还有一个有趣的作用是可以用作远程调用，比如现在有Java接口，这个接口的实现部署在其它服务器上，在编写客户端代码的时候，没办法直接调用接口方法，因为接口是不能直接生成对象的，这个时候就可以考虑代理模式（动态代理）了，通过***Proxy.newProxyInstance***代理一个该接口对应的**InvocationHandler**对象，然后在**InvocationHandler**的invoke方法内封装通讯细节就可以了。具体的应用，最经典的当然是Java标准库的RMI，其它比如hessian， 各种webservice框架中的远程调用，大致都是这么实现的。

动态代理的实现就是写一个代理Handler实现InvocationHandler，重写invoke方法，就可以 了。然后在调用的时候，用Proxy.newProxyInstance(Interface.getClass().getClassLoader(), new Class[]{Interface.class}, new DynamicProxyHandler(real));方法来创建代理对象。代码如下：

package org.heihei.learn;

import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class SimpleDynamicProxy {

public static void consumer(Interface inter){ inter.doSomething(); inter.somethingElse("hello");

}

public static void main(String[] args){ RealObject real = new RealObject(); consumer(real);

Interface inter = (Interface) Proxy.newProxyInstance(Interface.class.getClassL oader(),

new Class[]{Interface.class}, new DynamicProxyHandler(real)); consumer(inter);

}

}

interface Interface{ void doSomething();

void somethingElse(String args);

}

class RealObject implements Interface{

@Override

public void doSomething() {

// TODO Auto-generated method stub System.out.println("do something");

}

@Override

public void somethingElse(String args) {

// TODO Auto-generated method stub System.out.println("something else "+args);

}

}

class DynamicProxyHandler implements InvocationHandler{ private Object proxied;

public DynamicProxyHandler(Object proxied){ this.proxied = proxied;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

// TODO Auto-generated method stub

System.out.println("Object "+proxy.getClass()+" method: "+method+" args:"

+ args); if(args !=null){

for(Object arg :args) System.out.println(arg);

}

return method.invoke(proxied, args);

}

}

设计模式

###### 设计模式六大原则

单一职责原则：一个类，或者一个接口，最好只做一件事情，当发生变化时，他只能受到单一的影响；因为职责过多，可能引起变化的原因将会很多，这样导致职责和功能上的依赖，将严重影响其内聚性和耦合度，混乱由此而生。

开闭原则：开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这

里氏代换原则：里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是

对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

依赖倒转原则：这个是开闭原则的基础，具体内容：面向接口编程，依赖于抽象而不依赖于具体。

接口隔离原则：这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖， 降低耦合。

迪米特原则（最少知道原则）：为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。System.out.print()就是不符合这个原则。

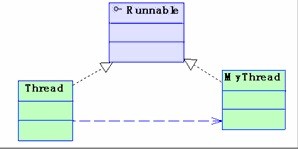
合成复用原则：原则是尽量使用合成\/聚合的方式，而不是使用继承。可有可无

###### 代理模式

代理模式的定义：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下， 一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

举例子：Thread其实实现了Runnable接口，但是它还干了许多Runnable接口没有干的事

情，它先完成比线程主体更多的操作，比如分配CPU资源，线程是否启动等等，从类关系上看是典型的代理模式。



###### 策略模式

策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替 换。策略模式让算法独立于使用它的客户而独立变化。将算法的责任委派给不同的对象处理，在比较和排序时候经常用到。

举例子：java的自定义类型内部新建一个内部类，有name和id两个属性，为了比较两个自定义类型，实现compartor接口，而不是直接重写comparable接口，在调用sort()函数的时候，可以把这个内部类直接传给sort()函数，这样就可以避免重写compareTo()方 法，只要写两个内部类实现comparator接口即可。

Hibernate的二级缓存有不同的缓存提供商，这也是一种策略模式，例如ehcache, hashset, oscache等等

###### 单例模式

单例模式是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例的特殊类。通过单例模式可以保证系统中一个类只有一个实例

应用场景：主要用在资源共享，控制资源之间交流 1.数据库连接池 2.日志应用 3.应用配置 4.线程池

JDKRunTime，Selector类用到了单例模式 Spring IoC控制的bean默认的就是单例的，但是我们可以通过改scope属性来改成prototype的。

Hibernate的SessionFactory可以写成一个单例模式，因为一个SessionFactory相当于一个数据库的内存镜像，存在多个也是没有意义的，所以在Spring IoC中只要把它设置成一个Singleton模式的就可以了，就可以一些不必要的资源浪费了。

单例模式实现：

懒汉模式（懒加载）：首先将构造函数设成private型，定义一个static变量来储存实例， 调用getInstance方法获得实例，方法内部判断实例是否已经建立，没有就建一个，否则直接返回已有实例

饿汉模式：定义变量的时候直接初始化。

线程安全：懒汉模式不是线程安全的，因为可能有两个线程同时判断出对象没有实例 化，于是创建了两个实例，饿汉模式一开始就初始化了实例，所以始终只有一个，不需要再创建

实例化时机不同：懒汉模式获取的时候才实例化，饿汉定义变量的时候就实例化了

延迟加载：懒汉模式需要的时候才创建，体现了延迟加载，（可用于缓存，实例就是缓存，有了就不再往缓存立方，没有才放）

双重加锁：double-check：懒汉模式实现线程安全，每次获取实例的时候都需要加锁同步，影响效率，所以在外面在加一层判断，对象是否实例化，如果已经实例化过了就不需要再进里面去加锁同步了，（变量要定义成volatile类型）这还不是线程安全的？该还有将instance加个关键字volatile，不然的话编译重排序可能造成还没初始化完成就已经完成对象引用的复制，但是此时对象还没有初始化完毕。饱汉模式的区别volatile内存模型防止被优化

枚举实现单例最好

一种是通过枚举，一种是通过静态内部类

###### 装饰模式

在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。就是把想要包装的类，通过作为初始化参数传递给装饰类，然后装饰类给他加了一些功能。

Java中的打印流，PrintStream类PrintWriter类就用到了装饰模式，它本身实现的就是

PrintStream(OutputStream),它所要干的事情就是构造函数里的OutputStream要干的事情，把数据传入到输入流当中，但是如果直接用OutputStream的话只能用write()方法， 并请只能写byte类型，感觉很苦逼，所以它用PrintStream类来装饰（包装）了一下

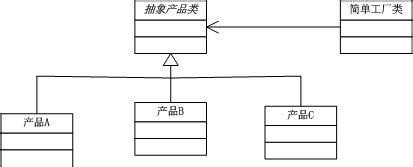
OutputStream类，这样它就可以更加方便的使用了，比如可以直接调用print();println();各种函数往流里面写东西。

ObjectOutputStream类就采用了装饰者模式，并要求通过另一个OutputStream从物理上写入序列化数据。ObjectOutputStream类包含了覆盖所有原始类型的方法，以及一个使用与引用类型的方法，通过这些方法可以将对应类型的值写入流中。

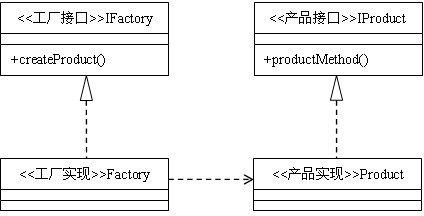
###### 工厂模式

工厂模式又分为:简单工厂模式，工厂模式和抽象工厂模式。目的是解耦合。

简单工厂模式就是一个父类产品抽象类（接口），N个子类产品继承自父类，然后工厂类有一个static方法来根据不同的参数（函数内用switch或者if语句来判断）来生产不同的子类对象。具体的UML图如下图所示：



工厂模式比简单工厂模式更加具体化，工厂模式还是N个子类产品继承自一个父类，然后工厂类这回不是大包干了，而是工厂类也有一个父类，然后派生出不同的子类来对N个产品子类进行生产，这样在应用的时候，我们只要先声明一个工厂类对象，然后再根据不同的工厂类对象生产处不同的产品对象。具体UML图如下图所示：



抽象工厂模式与工厂模式相比它提供了一个产品族的生产方法，也就是一个工厂类里生产了不同的等级结构的产品。所谓产品族是处于不同等级结构的产品之间有关联，然后他们组成了一个产品组。比如Car是一个大类，可以派生出BenzCar和BmwCar，而BenzCar又可以派生出BenzSportsCar，BMWCar又派生出BMWSportsCar，这两个SportsCar就是一个产品族的。

###### 优点：

可以使代码结构清晰，有效地封装变化。在编程中，产品类的实例化有时候是比较复杂和多变的，通过工厂模式，将产品的实例化封装起来，使得调用者根本无需关心产品的实例化过程，只需依赖工厂即可得到自己想要的产品。

对调用者屏蔽具体的产品类。如果使用工厂模式，调用者只关心产品的接口就可以了， 至于具体的实现，调用者根本无需关心。即使变更了具体的实现，对调用者来说没有任何影响。

降低耦合度。产品类的实例化通常来说是很复杂的，它需要依赖很多的类，而这些类对于调用者来说根本无需知道，如果使用了工厂方法，我们需要做的仅仅是实例化好产品类，然后交给调用者使用。对调用者来说，产品所依赖的类都是透明的。

工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。

最主要的作用是解耦和，通过父类引用创建对象。可以利用反射机制实现工厂模式。

###### 原型模式

原型模式(Prototype)，用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

原型模式 其实就是从一个对象再创建另外一个可定制的对象，并且不需要知道任何创建的细节。

JAVA原型模式是实现cloneable接口，并且重写clone（）方法。分为浅复制和深复制。 **PS**： 深拷贝与浅拷贝问题中，会发生深拷贝的有**java**中的**8**中基本类型以及他们的封装类型，另外还有**String**类型。其余的都是浅拷贝。

原型模式的优点及适用场景

使用原型模式创建对象比直接new一个对象在性能上要好的多，因为Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明 显。

使用原型模式的另一个好处是简化对象的创建，使得创建对象就像我们在编辑文档时的复制粘贴一样简单。

因为以上优点，所以在需要重复地创建相似对象时可以考虑使用原型模式。比如需要在一个循环体内创建对象，假如对象创建过程比较复杂或者循环次数很多的话，使用原型模式不但可以简化创建过程，而且可以使系统的整体性能提高很多。

###### 模板方法模式

定义一个操作中的算法的股价，而将一些步骤延迟到子类中。模板方法使得子类可以不改变算法的结构即可重定义该算法的某些特定步骤。子类替换掉父类中的抽象方法。

优点：模板方法模式是通过把不变行为搬到超类，去除子类中的重复代码来体现它的优势， 提供了一个很好的代码复用平台。当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为就会在子类中重复出现。我们通过模板方法模式把这些行为搬移到单一的地方，这样就帮助子类摆脱重复的不变行为的纠缠。

一、Java内存回收机制

不论哪种语言的内存分配方式，都需要返回所分配内存的真实地址，也就是返回一个指针到内存块的首地址。Java中对象是采用new或者反射的方法创建的，这些对象的创建都是在堆

（Heap）中分配的，所有对象的回收都是由Java虚拟机通过垃圾回收机制完成的。GC为了能够正确释放对象，会监控每个对象的运行状况，对他们的申请、引用、被引用、赋值等状况进行监控，Java会使用有向图的方法进行管理内存，实时监控对象是否可以达到，如果不可到达，则就将其回收，这样也可以消除引用循环的问题。在Java语言中，判断一个内存空间是否符合垃圾收集标准有两个：一个是给对象赋予了空值null，以下再没有调用过，另一个是给对象赋予了新值，这样重新分配了内存空间。

二、Java内存泄露引起原因

首先，什么是内存泄露？经常听人谈起内存泄露，但要问什么是内存泄露，没几个说得清 楚。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称为内存泄露。内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你Out of memory。

那么，Java内存泄露根本原因是什么呢？长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景。具体主要有如下几大类：

1、静态集合类引起内存泄露：

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

例:

Static Vector v = new Vector(10); for (int i = 1; i<100; i++)

{

Object o = new Object(); v.add(o);

o = null;

}\/\/

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

2、当集合里面的对象属性被修改后，再调用remove（）方法时不起作用。例：

public static void main(String[] args)

{

Set<Person> set = new HashSet<Person>(); Person p1 = new Person(" 唐 僧 ","pwd1",25); Person p2 = new Person("孙悟空","pwd2",26);

Person p3 = new Person("猪八戒","pwd3",27); set.add(p1);

set.add(p2);

set.add(p3);

System.out.println("总共有:"+set.size()+" 个元素!"); \/\/结果：总共有:3 个元素! p3.setAge(2); \/\/修改p3的年龄,此时p3元素对应的hashcode值发生改变set.remove(p3); \/\/此时remove不掉，造成内存泄漏

set.add(p3); \/\/重新添加，居然添加成功

System.out.println("总共有:"+set.size()+" 个元素!"); \/\/结果：总共有:4 个元素! for (Person person : set)

{

System.out.println(person);

}

}

3、监听器

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

4、各种连接

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

NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的

Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在

finally里面释放连接。

5、内部类和外部模块等的引用

内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如：

public void registerMsg(Object b);

这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B 是否提供相应的操作去除引用。

6、单例模式

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

class A{ public A(){

B.getInstance().setA(this);

}

....

}

\/\/B类采用单例模式class B{

private A a;

private static B instance=new B(); public B(){}

public static B getInstance(){

return instance;

}

public void setA(A a){ this.a=a;

}

\/\/getter...

}

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

**OutOfMemoryError**原因

OutOfMemory主要有几个方面：永久带溢出，java堆溢出。 容易发生内存溢出问题的内存空间包括：Permanent Generation space和Heap space。

1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；也就是永久带的空间不够了，加载了大量的jar，.class文件

2.集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；

3.代码中存在死循环或循环产生过多重复的对象实体；

4.使用的第三方软件中的BUG；

5.启动参数内存值设定的过小； 解决；

1.应用服务器提示错误的解决： 把启动参数内存值设置足够大。

2.Java代码导致错误的解决： 重点排查以下几点：

1)检查代码中是否有死循环或递归调用。

2)检查是否有大循环重复产生新对象实体。

3)检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

4 )检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

5）可以用软引用案例：

1.hibernate查询数据时，一次查询过多的数据，后来调整了该部分的代码，每次只取出指定量的数据，成功的解决该问题。

2.在做压力测试时，出现OutOfMemoryError，发现session的资源一直没有被释放产生的，最好通过session的invalidate()方法将session的资源释放。

3.程序中出现死循环。

1. tomcat部署、运行出现OutOfMemoryError，加大内存参数值，解决此问题。

### StackOverFlow

栈溢出一般就是指JAVA虚拟机栈溢出。每一个线程都有一个虚拟机栈，每个栈会有许多栈帧，每个栈帧是一个方法的调用，因此如果很多层递归的话就容易导致栈溢出。

for和foreach的用法区别for可以不逐个遍历，比如每隔一个遍历一个，也可以从前向后遍历， 从后向前，但是foreach不能从后向前遍历，只能从前向后，并且foreach只能逐个遍历。在

foreach中不能向迭代变量赋值。for循环有条件判断，可以限制执行条件。foreach没有。对集合对象的遍历，foreach更常用。

建议如果对集合对象的从头到位读取操作用foreach，这个foreach有优化。

for通常用于已知次数的循环；

foreach通常用于遍历集合；

CompareAndSwap是JAVA实现乐观锁的一个前提，比如Atomic包里的一些数据类型都是借用处理器的支持的CAS操作。CAS是CPU指令集的一个操作，比较和交换，在实现乐观锁的时候通过调用JNI中的CompareAndSwap方法来实现预期的值与更新的值是否相等，从而确定此次更新是否成功。

### Iterator接口

Iterator接口包含三个方法：

public interface Iterator<E>{ E next();

boolean hasNext(); void remove();

}

以下例子是利用了Iterator接口的着三个方法，实现遍历ArrayList<String>类型。

一开始迭代器在所有元素的左边，调用next()之后，迭代器移到第一个和第二个元素之间，next()方法返回迭代器刚刚经过的元素。

hasNext()若返回True，则表明接下来还有元素，迭代器不在尾部。

remove()方法必须和next方法一起使用，功能是去除刚刚next方法返回的元素。

package com.xujin;

import java.util.ArrayList; import java.util.Collection; import java.util.Iterator;

public class Test{

public static void main(String...arg){ Collection<String> a = new ArrayList<String>(); a.add("Bob");

a.add("Alice");

a.add("Lisy");

Iterator<String> iter = a.iterator(); while(iter.hasNext()){

String ele = iter.next();

System.out.print(ele + " ");//Bob Alice Lisy

}

System.out.println();

System.out.println(a);//[Bob, Alice, Lisy] Iterator<String> iter2 = a.iterator(); iter2.next();

iter2.remove(); System.out.println(a);//[Alice, Lisy]

}

}

### Iterable接口

Iterable接口仅包含一个方法：

public interface Iterable<E>{ Iterator<E> iterator();

}

for-each循环可以与任何实现了Iterable接口的对象一起工作。

而Collection接口扩展了Iterable接口，故标准类库中的任何集合都可以使用for-each循环。

为什么一定要实现**Iterable**接口，为什么不直接实现**Iterator**接口呢？

看一下JDK中的集合类，比如List一族或者Set一族，都是实现了Iterable接口，但并不直接实现Iterator接口。

仔细想一下这么做是有道理的。

因为Iterator接口的核心方法next()或者hasNext() 是依赖于迭代器的当前迭代位置的。

如果Collection直接实现Iterator接口，势必导致集合对象中包含当前迭代位置的数据(指针)。

当集合在不同方法间被传递时，由于当前迭代位置不可预置，那么next()方法的结果会变成不可预知。

除非再为Iterator接口添加一个reset()方法，用来重置当前迭代位置。但即时这样，Collection也只能同时存在一个当前迭代位置。

而Iterable则不然，每次调用都会返回一个从头开始计数的迭代器。多个迭代器是互不干扰的。

##### Spring Ioc

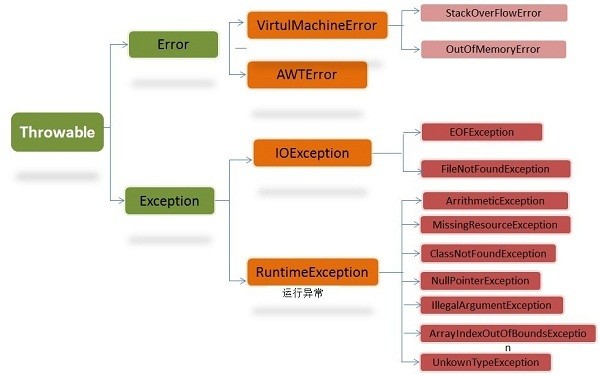
控制反转即IoC (Inversion of Control)，是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。它把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的“控制反转”概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。

##### i = i++

i = i++这个题是java中常见的一道笔试题。它的结果就是i本身。这与C语言和C++是不同的， 因为java中会先开辟一段内存里临时存i的值，在计算完事之后再把这个值赋值给i，所以i还是

i，不会发生大小改变。

###### 异常类



如图所示，Error和Exception都继承自Throwable类，Exception又派生出RuntimeExeption和

IOException。IOException一定要用try,catch进行捕捉。RuntimeException就需要我们自己通过编程注意这些事情了。Error又可以分为虚拟机错误和AWTError，这些错误都是严重错误，

catch到也处理不了，所以就不catch了。不捕捉**RuntimeExeption**是**java**默认的，**java**默认每个方法都会**throws**运行时异常，不需要人为捕捉，因为通常是由于程序员的粗心原因导致

的，所以捕捉到了也没有用，干脆直接往上抛

* 1. 异常分为运行时异常（RuntimeException）、受检异常（Exception）、系统错误error。
  2. RuntimeException，也就是运行时异常，表示代码本身存在BUG，比如

ArrayIndexOutOfBoundsException，[数组下标越界](https://www.baidu.com/s?wd=%E6%95%B0%E7%BB%84%E4%B8%8B%E6%A0%87%E8%B6%8A%E7%95%8C&amp;tn=67012150_cpr&amp;fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1Y4uAnYuHIBP19bPW79nWKW0ZwV5fKWUMw85HmLnjDznHRsgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHDvnWf3Pj03)，数组定义的长度不够实际使用，代码若不调BUG进行处理肯定还会报错，控制台一旦报RuntimeException，就必须在代码中找BUG，因为代码BUG是人为粗心制造的，不是try-catch一下就能解决的。try-catch 用在代码BUG上是毫无意义的，只需要写代码时谨慎点就能减少BUG，而不是try- catch。

非RuntimeException，就是受检异常。比如处理文件流时的I\/O问题，就属于编译时异常，相当于假设有IO异常就利用try-catch对其进行处理，或者 throws即可。

error，通常是系统出现了不可控制的错误，这个通常与程序无关，所以是不需要处理的。

* 1. 下面给出运行时异常与受检异常的清晰定义：

①受检查异常表示程序可以处理的异常，如果抛出异常的方法本身不能处理它，那么方法调用者应该去处理它，从而使程序恢复运行，不至于终止程序。例如，喷墨打印机在打印文件时，如果纸用完或者墨水用完，就会暂停打印，等待用户添加打印纸或更换墨

盒，如果用户添加了打印纸或更换了墨盒，就能继续打印。

②运行时异常表示无法让程序恢复运行的异常，导致这种异常的原因通常是由于执行了错误操作。一旦出现了错误操作，建议终止程序并仔细的debug，因此[Java编译器](https://www.baidu.com/s?wd=Java%E7%BC%96%E8%AF%91%E5%99%A8&amp;tn=67012150_cpr&amp;fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1Y4uAnYuHIBP19bPW79nWKW0ZwV5fKWUMw85HmLnjDznHRsgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHDvnWf3Pj03)不检查这种异常

###### 子类父类加载顺序

先这行父类静态代码块，再 执行子类静态代码块，然后执行父类代码块，子类代码块，最后是父类构造函数，子类构造函数。

###### 值传递与引用传递

只有8中基本数据类型和String是值传递，其他自定义类型和集合类型等等都是引用传递，引用传递可以修改变量，而值传递修改也是白修改，修改的知识副本而已的啦。注意数组传递的是引用的呢。

###### 在服务器的网络编程中，解决会话跟踪的方法有**?**

1.cookie 2.session 3.url重定向 4.使用隐藏的表单域

##### ==与equals

1）对于==，如果作用于基本数据类型的变量，则直接比较其存储的 “值”是否相等；

如果作用于引用类型的变量，则比较的是所指向的对象的地址

2）对于equals方法，注意：equals方法不能作用于基本数据类型的变量

如果没有对equals方法进行重写，则比较的是引用类型的变量所指向的对象的地址； 诸如String、Date等类对equals方法进行了重写的话，比较的是所指向的对象的内容。

# C++

**new delete** 和 **malloc free** 的区别

之前面试的时候面试官问了我一个问题，是C语言中有了malloc和free为什么还要再有new 和

delete。说一下这两者的区别。 其实之前也有看过这个东西，包括

malloc free是C语言的库函数，而new和delete是C++的操作符。

malloc开辟出的是一片空间，返回的是一个void \*的指针，用的时候必须强制转换。而

new直接返回的就是该类型的指针。但是这些都不是最为重点的。

最为关键的就是对于非内部数据类型的对象也就是自定义的类型而言，这两者的区别是最大的。光用malloc free无法满足动态对象的要求。

对象在创建的同时要执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc

free是库函数不是操作符，不在编译器控制权限之内，无法把构造函数和析构函数任务强加给malloc和free。

因此 C++语言有了new 和 delete来完成动态内存分配和初始化的工作。

对了，malloc和free还不能为动态对象初始化，要想初始化必须给类再写一个初始化void Initialize();的函数。

最后，为什么C++不淘汰掉malloc和free呢，这个很简单，因为它还要调用C的函数啥的，兼容一下下。delete可以释放malloc的内存，但不建议这么做，free不能释放new的内存，因为它无法执行析构函数，编译会出错。

**c++ const**提高函数健壮性

**const**提高函数的健壮性参数

const只能修饰输入参数。因为输出参数的话是需要修改的，如果const修饰过后就白扯了。

const作函数的输入参数是，可以采用指针或者引用传递，用const修饰，这样就可以防止意外修改了，但是如果是值传递，即没有指针或者引用时就不需要了，因为参数复制，即使函数改了也是白改，不会影响实参。注意对于费内部数据类型的输入参数，应该将值传递改为

const引用传递即void Func(A a);改为void Func(const A &a)；对于内部数据类型就没有必要改了，直接用void Func(int a);就好了，没必要加const引用了就。

###### 返回值

const char \*GetString();类似的，返回值指针所指向的内容是不能更改的 返回值只能赋值给加了const的同类型指针 const char \*str = GetString()；

如果函数返回值采用“值传递”方式，由于函数会吧返回值赋值到外部临时的存储单元中， 加上Const也没啥意义。所以不要把 int GetInt();改为const int GetInt();

函数返回值是引用传递的场合不多，一半用在类内赋值函数中，目的是为了实现链式表达。例如：

class A{

A& operate= （const A &other）；//赋值函数

}；

A a,b,c;

a = b = c;//可以

const 成员函数 记住哦 const放在尾部，大概是其他地方都给占上了,只能放在函数最后了,不会修改类的数据成员要声明为const类型。const成员函数不能更改数据成员，也不能调用其他非const成员函数。形如:

class Stack{

public：

void Push(int elem); int Pop();

int GetCount() const;//const成员函数private：

int m\_num;

int m\_data[100];

}；

int Stack::GetCount() const

{

++m\_num;//error 修改了数据成员

Pop();//error 调用了非const成员函数

}

多重继承与虚继承

C++中的多重继承和虚继承是一个非常重要的概念，也是看你是不是懂C++的一个重要的标志之一。这中间包括了运行时多态，虚函数表等等相关概念。

多重继承，顾名思义，是一个类继承了多个父类。例如

class C：public A， publicB{};这样就是一个多重继承。

###### 多重继承的构造和析构函数。

多重继承的构造函数是先执行父类中的构造函数在执行自己的构造函数，多个父类按照继承顺序来。上面的例子就是先A再B最后C。析构正好相反。

可以显示的在初始化列表中进行构造，也可以不写隐式调用父类的构造函数。例如

C():A(),B(){}这样就是显示的。C():B(){}这样就是隐式的调用A构造函数，显示调用B的构造函数了。

###### 运行时多态

运行时多态就是用父类的指针指向子类的对象，调用子类中的函数。在运行时决定调用哪一个函数。

运行时多态是与静态多态不同的，静态多态就是函数的重载和模板。

###### 虚继承

虚继承是解决多重继承中的一个非常常见的问题：菱形继承问题。即基类是A，B和C分别继承A，D多重继承B和C。D在构造函数的时候就执行了两个A的构造函数。所以会出现错误的。这个时候就用到了虚继承。

虚继承的用法是这样的：虚继承的类就叫做虚基类。 A就是是虚基类，B和C虚继承A，D

中只有一个A对象。

class A{

};

class B:public virtual A{

}；

class C:virtual public A{//virtual与public的顺序都可以的

}；

class D:public B, public C{

}；

最后注意，虚基类总是先于非虚基类构造的，与继承的顺序是没有关系的。

###### 虚函数表

还要说的就是虚函数表V-Table的问题。虚函数表是运行时多态的根本上的保障。虚函数是一个类如果它包含虚函数，则该类就会在内存中建立这样一个虚函数表。在运行时， 根据虚函数表决定调用哪一个函数。

一个类如果包含虚函数，则它的地址中有4位是虚函数表地址的指针。虚函数表示一块连续的内存。为每一个父类（有虚函数）都会有一个虚函数表。多重继承中，子类的函数放在第一个虚函数表内。详情请看这个链接，他这个将的就非常全面。

<http://blog.csdn.net/haoel/article/details/1948051/>

## 引用和指针的区别

###### 引用和指针

引用其实是变量的另一个别名而已，它用的时候完全与之前的变量一模一样。

引用使用的时候是有规则的：

（1）引用创建时候必须初始化(指针可以在任何时候初始化)

（2）引用一旦创建就不能更改引用关系（指针还可以指向其他的对象）

（3）引用不能为NULL

另外，指针是会单独分配空间的，而引用不会单独分配空间，如果sizeof指针和引用的话就会看到区别，sizeof（指针）是系统分配给指针的内存是4个字节，而sizeof(引用)的话是它所引用对象的大小，如果是int p[10];那么它的大小就是40。

两者可以作为函数的参数 void func(int &a);void func(int \*a);都可以对a所指向或者引用的对象进行更改，但是两者是有区别的，&a完全没有复制的过程直接操作在指向的对象上面，而\*a是将对象的指针复制给它

###### 伟哥版本

指针：一个数据对象的地址与引用：一个数据对象的别名之间的区别(不区分C/C++)

区别：

1. 空值：指针可以为空，悬空指针问题由此而来;引用无法为空，只有存在的数据对象才有别名。
2. 改变：指针可以更改;引用（别名）则不能易主。
3. 大小：指针是实体，大小为4字节（32位系统）;引用仅为别名，大小为其依附数据对象的大小。
4. 安全：指针没有类型检查；引用有类型检查。
5. 访问：指针间接访问对象；引用直接访问对象。

小结：在函数参数传递中，指针与引用均可解决大块数据或对象传递效率低和空间开销大的问题。

# const和define

##### const和#define

（1）const常量有数据类型，而宏定义没有数据类型。编译器可以对前者进行类型安全检查。而对于后者指进行字符的替换（文本上的替换），没有类型安全检查，并且字符替换可能会产生意想不到的错误。

（2）有一些集成化的调试工具可以对const常量进行调试，但是不能对宏常量进行调试

**define** 和内联函数

**#define**和内联函数

内联函数和普通函数相比可以加快程序运行的速度，因为它不需要中断，相当于直接把函数执行的代码直接复制到代码段，而宏只是一个简单的替换，而且宏还有有可能出 错，一会儿为大家举个例子。

与宏相比，内联函数更加安全可靠，但是这是以消耗空间为代价的。如果函数代码量小可以，如果代码量大的话，就会复制过多的代码到代码区，导致过多的空间消耗，得不偿失。因此，内联函数经常用于多次被调用的短小函数，不能有for while switch等语句。

接下来是一个**#define**函数产生错误，而内联函数没问题的例子。

#define MAX(a,b) (a) > (b)?(a):(b) result = MAX(i,j)+2;

由于#define 只是简单的文本替换，所以被预处理器理解为result = (a) > (b)?(a):(b) + 2；

因为加号的优先级高于三目运算符，所以先算b+2会先执行。这样就是错误的。只有

#define MAX(a,b) ((a) > (b)?(a):(b))才可以。

另外，内联函数做成员函数是，如果在类内直接实现的默认为内联函数。也可以在类外显示声明为**inline**。

拷贝构造函数与赋值运算符重载

拷贝构造函数和重载赋值=的函数可以有效防止在浅复制过程中可能对于同一片内存释放两次的问题。然而拷贝函数和重载复制=的函数很容易混淆。拷贝构造函数是在对象创建时调用的，而赋值函数只能被已经存在的对象调用。

一个例子 类String的拷贝构造函数和赋值函数。

//拷贝构造函数

String::String(const String &other)

{

int length = strlen(other.m\_data); m\_data = new char[length + 1]; strcpy(m\_data, other.m\_data);

}

//赋值函数

String &String::operator = (const String &other)//一定要注意4个步骤都不能少，这就是区别拷贝构造函数和赋值函数的区别。

{

//检查自赋值if(this == other)

return \*this;

//释放原有的内存资源delete [] m\_data;

//分配新的内存资源，并复制内容

int length = strlen(other.m\_data); m\_data = new char[length + 1]; strcpy(m\_data, other.m\_data);

//返回本对象的引用

return \*this;

}

应用的例子是这样的； String a("Hello"); String b("World"); String c = a;//

这就时拷贝构造函数，注意不是赋值哦，其实应该写成String c(a);才是标准的，这样考的目的就是让你容易混淆

c = b;这就是赋值函数，先释放原来的内存，开辟新内存。

## 空指针调用函数

首先看一段代码是否知道其正确还是错误。

class A{ public:

void print()

{

cout << "Hello" << endl;

}

};

void main()

{

A \*a = NULL;

a.print();

}

问你程序是否正确执行，或者执行结果是什么。

这就是一个典型的表示C++是一个静态语言的特征。可以阐明“静态绑定”和“动态绑定”的区别。

真正的原因是：因为对于非虚成员函数，Ｃ++这门语言是静态绑定的。这也是Ｃ++语言和其它语言Java, Python的一个显著区别。C++奉行的原则是能够在编译时候搞定的事情绝不拖到运行时搞定。

我们的第一感觉是这个程序应该是错的，然而事实却是相反的。每个对象都有指向自己的this指针，指针的值会因为不同的对象不同而不同，用来区别不同的对象。这里的this 指针就是一个NULL。如果我们调用this指针的时候就会出错。可是这个函数并没有用到

this指针就是简单的输出字符串，这一过程在编译阶段就已经完成了。所以不会报错。这一点与JAVA等动态语言不同。

但是对于C++。为了保证程序的运行时效率，Ｃ++的设计者认为凡是编译时能确定的事情，就不要拖到运行时再查找了。所以C++的编译器看到这句话会这么干：

1：查找a的类型，发现它有一个非虚的成员函数叫print。（编译器干的）

2：找到了，在这里生成一个函数调用，直接调A::print()。

所以到了运行时，由于print函数里面并没有任何需要解引用somenull指针的代码，所以真实情况下也不会引发segment fault。这里对成员函数的解析，和查找其对应的代码的工作都是在编译阶段完成而非运行时完成的，这就是所谓的静态绑定，也叫早绑定。 \* 正确理解C++的静态绑定可以理解一些特殊情况下C++的行为。

## 类的继承与组合

C++中设计孤立的类是非常容易的，难的是如何设计正确的基类和派生类。

继承（**inheritance**） 组合（**composition**）

如果类A和类B没啥太大关系，不要为了让类B多点功能就硬是让它继承类A，这样就好像一个好好地大老爷们你给他各种吃人参，吃甲鱼补身体一样。

还有关键的一点是如果B是A的一种（a kind of）这时候就要用到继承了，比如Man是一种（a kind of）Human。加强一下，如果逻辑上B是A的一种，并且，A的所有功能都对B 有益，则继承。一个特例就是一个圆继承椭圆的话，它的长轴短轴继承的就毫无意义

了。

组合的意义是一部分（a part of）。如果A是B的一部分，则B不能继承A，而是B由A和其他类组合而成的。

## 强制转换

C++中的强制转换类型主要有4种，分别是static\_cast, dynamic\_cast, const\_cast,

reinterpret\_cast。 我们首先来说一下比较简单的两种吧。

##### const\_cast

const\_cast主要是用来修改的const和volatile属性的。

const char \*pc;

char \*p = const\_cast<char\*>(pc);//正确，但是通过p写值是未定义的行为

const\_cast只能修改常量属性，不能修改其他属性。比如

const char \*cp;

char \*q = static\_cast(char\*) cp;//错误 static\_cast 不能修改常量属性

static\_cast<string(cp);//正确：字符串字面值转换成string类型了。

const\_cast<string>(cp);//错误：const\_cast只改变常量属性。

注意，指针也可以改成引用的。举个例子：

class B

{

public:

B() { }

public:

int m\_iNum;

};

void foo()

{

const B b1;

//b1.m\_iNum = 100; //compile error

// 可以做如下转换，体现出转换为指针类型

B \*b2 = const\_cast<B\*>(&b1);

// 或者左侧也可以用引用类型，如果对b2或b3的数据成员做改变，就是对b1的值在做改变

B &b3 = const\_cast<B&>(b1); b2->m\_iNum = 200; //fine b3.m\_iNum = 300; //fine

}

int main( int argc, char \* argv[] )

{

foo(); return 0;

}

使用const\_cast可以返回一个指向非常量的指针（或引用）指向b1，就可以通过该指针（或引用）对它的数据成员任意改变。 注：你不能直接对非指针和非引用的变量使用const\_cast操作符去直接移除它的const、volatile和 unaligned属性。

##### reinterpert\_cast

它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针。操作符修改了操作数类型，单仅仅是重新解释了给出对象的比特模型而没有进行二进制转换。

总结来说reinterpret\_cast用在任意指针（或引用）类型之间的转换；以及指针与足够大的整数类型之间的转换；从整数类型（包括枚举类型）到指针类型，无视大小。虽然看起来很强大，可以把任何一种类型的指针转换成另一种，但是转换完的通常是不能用 的，只有通过转换回原类型的指针才能用的。

比如:

int value = 10; int \*p = &value;

double \*d = reinterpret\_cast<double\*>(p);

虽然编译不会有错误，但是你所得到的double是不可用的，它只有再转回int才可以，也就是再加上，int \*q = reinterpret\_cast(d); 这样就可以了。

最后还有一点，reinterpret\_cast不能去除const和volatile属性。

##### static\_cast

static\_cast应用非常广泛，它可以向我们之前C语言中的强制转换类型那么用，还可以继承类中用于指针的转换。static\_cast主要用于非多态类型之间的转换，不提供运行时的检查来确保安全的检查。

主要在以下几种场合中使用：

1.用于类层次结构中，基类和子类之间指针和引用的转换； 当进行上行转换，也就是把子类的指针或引用转换成父类表示，这种转换是安全的；当进行下行转换，也就是把父类的指针或引用转换成子类表示，这种转换是不安全的，也需要程序员来保证；

2.用于基本数据类型之间的转换，如把int转换成char，把int转换成enum等等，这种转换的安全性需要程序员来保证；

3.把void指针转换成目标类型的指针，是及其不安全的； 注：static\_cast不能转换掉

expression的const、volatile和 unaligned属性。

##### dynamic\_cast

dynamic\_cast是属于RTTI中的一部分，常常与type\_id一起使用，来确定运行时类别，主

要用于类层次之间的上行和下行转换，当然也可以用于类之间的交叉转换。

注意一点是，dynamic\_cast只能转换指针类型和引用类型，不能转换其他类型。

dynamic\_cast用于上行转换的时候，即把子类指针（引用）转换为父类指针（引用）时跟static\_cast是一样的，都是安全的。然而在进行下行转换时，\* dynamic\_cast具有类型检查的功能，比static\_cast更加安全。在多态类型之间的转换主要使用dynamic\_cast,应为类型提供运行时信息。

如果一条dynamic\_cast语句的转换目标是指针类型失败了，则结果为0。如果引用类型失败了，则抛出一个bad\_cast的异常。 \* 最后注意将基类cast到派生类时候，基类必须要有虚函数，没有虚函数不可以的。

## 重载 覆盖 隐藏

重载 覆盖 隐藏是C++中最为常见的几种函数相关的概念，特别是在存在虚函数的类继承中特别容易混淆。因此，区分三个概念是非常重要的。

重载（**overload**）

重载是在一个类中，相同的函数名，不同的参数，可以实现重载。跟返回值无关，返回值不同，不能叫做重载。

不是两个函数的名字相同就能构成重载。

全局函数和类的成员函数同名也不算重载，因为函数的作用域不同。例如

void Print();//全局函数class A{

public:

void Print();

};

不论Print的函数是否不同，都不算重载。如果某个类的成员要调用全局函数时候，必须用::Print();

还有就是C++是C的一个扩展，如果C++要调用C语言编译后的C函数改怎么办？比如void foo(int x,int y);C语言编译的时候产生的库中的名字是\_foo,而C++的编译器会产生

\_foo\_int\_int这样的名字来应付重载。由于编译后的名字不同，C++不能直接调用C函数。

C++提供了一个C连接交换指定符号extern "C"{ void foo(int x, int y );//或者#include ".h"}

来解决

覆盖（**override**）

覆盖发生在虚函数的继承时候发生的，而且发生在不同的范围内，基类和子类。函数前必须有virtual。而重载有没有virtual无所谓。而且函数名字和参数必须相同。如果参数不同，就是隐藏，不是重载哦。

隐藏（**hide**）

隐藏发生在继承时候。如果基类的函数不是virtual，子类又重新定义了该函数，无论参数相同不相同都是隐藏。如果是virtual 函数， 参数相同就是覆盖，参数不同就是隐藏。

举个例子：

class Base{ public:

virtual void f(float x); void g(float x);

void h(float x);

};

class Derived{

void f(float x);//覆盖void f(int x);//隐藏void g(float x);//隐藏void h(int x);//隐藏

};

# First Chapter

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.

# MapReduce

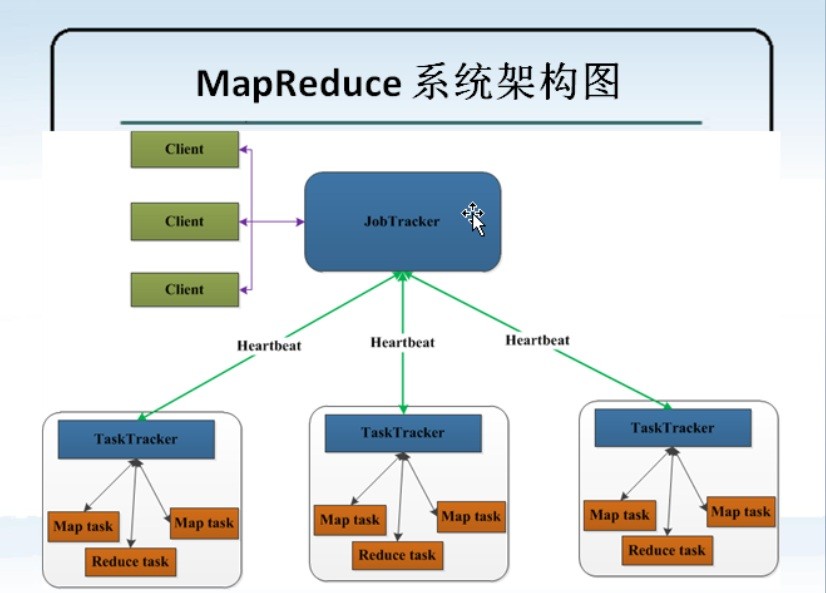
**MapReduce**特点：

易于编程

良好的扩展性高容错性

适合PB级海量数据的离线处理

**MapReduce**系统架构图



**MapReduce**编程模型**:**



**MapReduce**守护进程：

MapReduce框架主要有两个守护进程，jobtracker和tasktracker。jobtraker是管理者，

taskertracker是被管理者。

##### jobtracker:

负责接收用户提交的作业,负责启动跟踪任务执行管理所有作业(job:用户的一个计算请求)

将作业分成一系列任务(task:由job拆分出来的执行单元)进行调度将任务指派给tasktracker

作业\/任务监控,错误处理等

##### tasktracker:

负责执行由jobtracker分配的任务，管理各个任务在每个节点执行情况运行MapTask和ReduceTask

与Jobtracker进行交互，执行命令,并汇报任务状态

**MapReduce**相关概念

##### MapTask

Map引擎

分析每条数据记录,将数据解析传递给用户自定义的map()函数

将map()函数输出写到本地磁盘（如果是map-only情况，直接输出到HDFS中）

##### ReduceTask

Reduce引擎

从MapTask上远程读取输入数据对数据进行排序

将数据按照分组传递给用户编写的reduce()函数

**MapReduce**运行流程

1.在客户端启动一个作业

2.客户端向JobTracker请求作业号

3.客户端向HDFS复制作业的资源文件，这些文件包括打包jar文件，配置文件,以及由客户端计算所得到的输入划分信息。这些文件都存在jobtracker专门为这个job创建的一个文件夹中，以JobID命名。输入划分信息告诉JobTracker应该为这个作业启动多少个map任务等信息

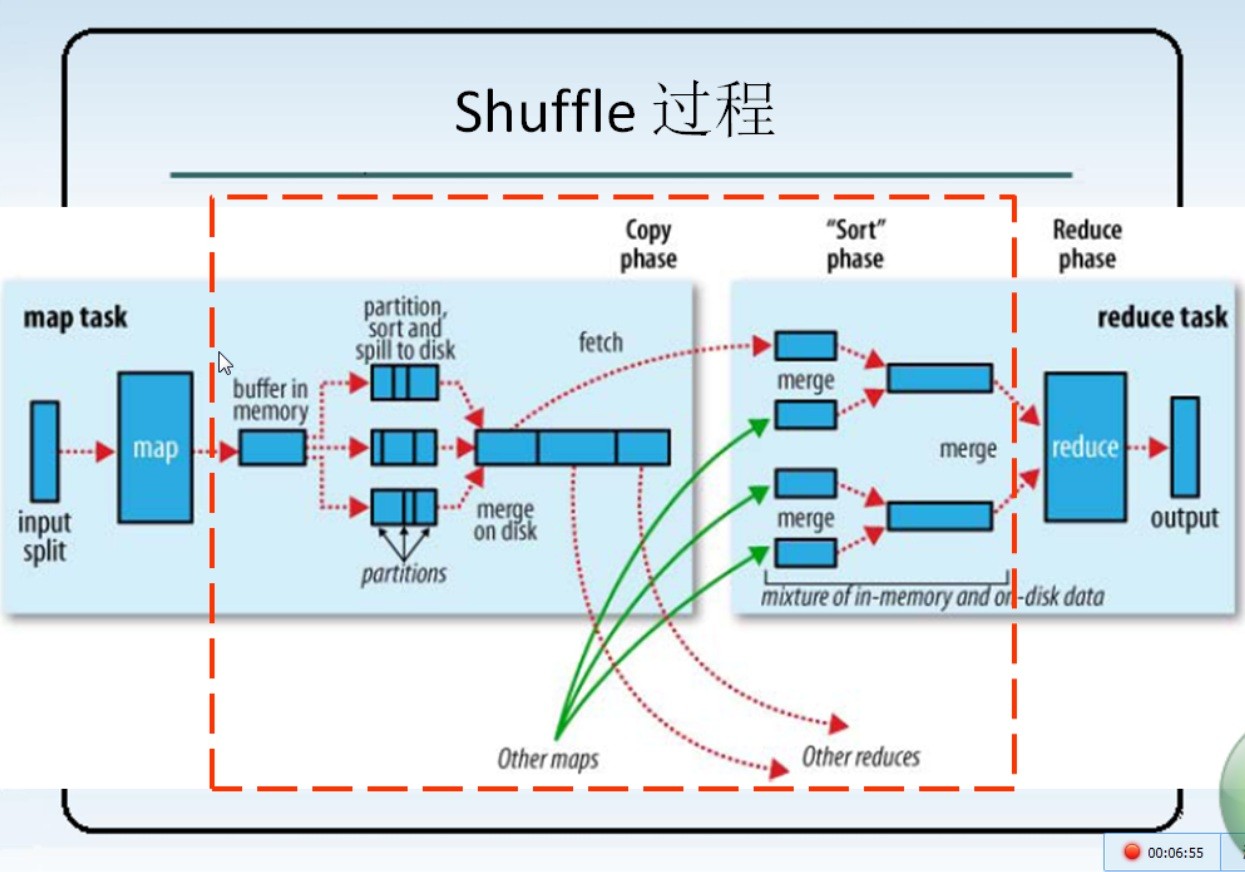
4.客户端向JobTracker提交作业,JobTracker接收到作业以后，把它加入到作业队列，然后JobTracker根据自己的调度算法调度到当前作业时，根据输入划分信息,开始为每个划分新建1个task任务，并把task任务分配给tasktracker执行。这里的分配不是随便分配的, 而是遵循数据本地化原则的。(数据本地化Data-Local, 就是将map任务分配给拥有该map 所要处理数据的DataNode节点，并将jar拷贝到这个节点，这个叫做移动计算，不是移动数据。)

5.TaskTracker每个一段时间向JobTracker发送心跳,告诉他自己仍然在运行。同时心跳中还带着其他的一些信息，比如当前map任务完成的进度。当jobtracker接收到最后一个

map任务发来的信息的时候,便把作业设置为"成功", 当jobclient查询时，将成功信息返回给用户。

**shuffle**过程：

shuffle是洗牌或者弄乱的意思，在MapReduce中是指从map task输出到reduce task输入这段过程。



**HDFS**组成：

HDFS主要由NameNode，DataNode，SecondaryNameNode组成，进程启动顺序也是如上所示。

##### NameNode

NameNode是主节点，负责存储文件的元数据，如文件名，文件目录结构，文件属性（创建时间，副本数等等），以及块列表和所对应的DataNode节点。它是一个主服务器，除了负责存储文件元数据以外，还负责控制client对文件的访问，但是client读取数据不是从client这里读。

##### DataNode

DataNode是数据节点，负责存储文件和文件的校验文件，校验文件包括文件的长度，文件的校验和时间戳。client可以从DataNode中读取数据和写入数据，前提是client从NameNode那里得到允许。

##### SecondaryNameNode

SecondaryNameNode是辅助NameNode节点工作的。

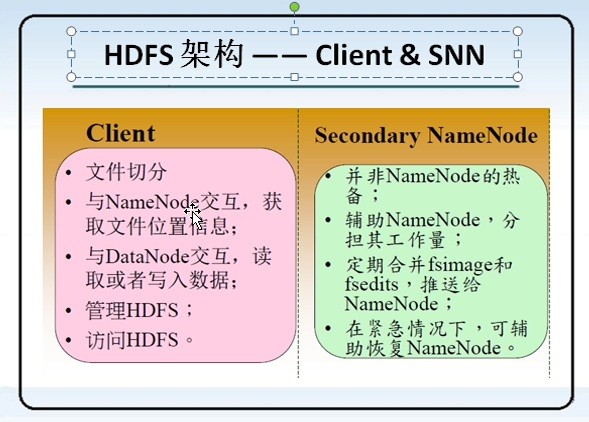
SecondaryNameNode不是NameNode的一个热备份，而是它的冷备份，

SecondaryNode定期从NameNode获得FSImage和Edit，将他们合并成新的

FSIamge.ckpt,再将FSImage发送回给NameNode。

当NameNode宕机时，就可以用SecondaryNameNode记录的信息进行恢复，但是它并不是最新的镜像。

**HDFS**架构之**SecondaryNameNode**与**Client:**



##### Client

client负责文件分块，当文件大小超过系统block的大小的时候，需要对文件进行分块，

client从NameNode获取如何分块，然后由Client进行数据分块（逻辑），然后再将分好的块写入到相对应的DataNode中

##### SecondaryNameNode

**NameNode**两个重要文件：

fsimage 元数据镜像文件 (保存文件系统的目录树)

edits 元数据操作日志(针对目录树的更改操作)

###### 元数据镜像

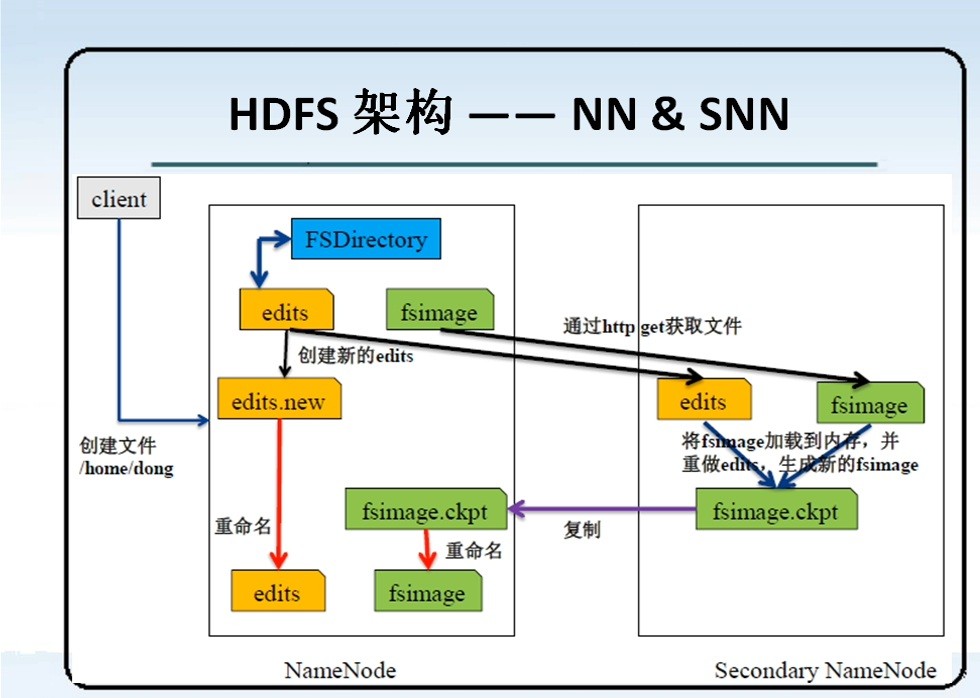
内存中保存一份最新的

内存中的镜像=fsimage+edits

定期合并**fsimage**和**edits**

edits过大将导致NameNode重启过慢，NameNode启动是先将fsimage加载到内存中，然后再根据edits更改目录树，如果edits过大就会导致重启过慢

SencondaryNameNode负责定期合并fsimage和edits



##### SecondaryNameNode合并过程

SNN通知NN切换到editlog

SNN通过http get方式从NN获得fsimage和edits

SNN将fsiamge加入内存，然后开始合并edits

SNN将新的fsimage.ckpt返回给NN

NN用新的fsimage替换旧的fsimage

**HDFS**文件读写流程：

###### 文件读取流程

Client调用FileSystem.open()方法：

1).FileSystem通过RPC与NameNode通信，NameNode返回该文件的部分或全部block列表（含有block拷贝的DataNode地址）

2).选取距离客户端最近的DataNode建立连接，读取block，返回FSDataInputStream

Client调用FSDataInputStream的read()方法

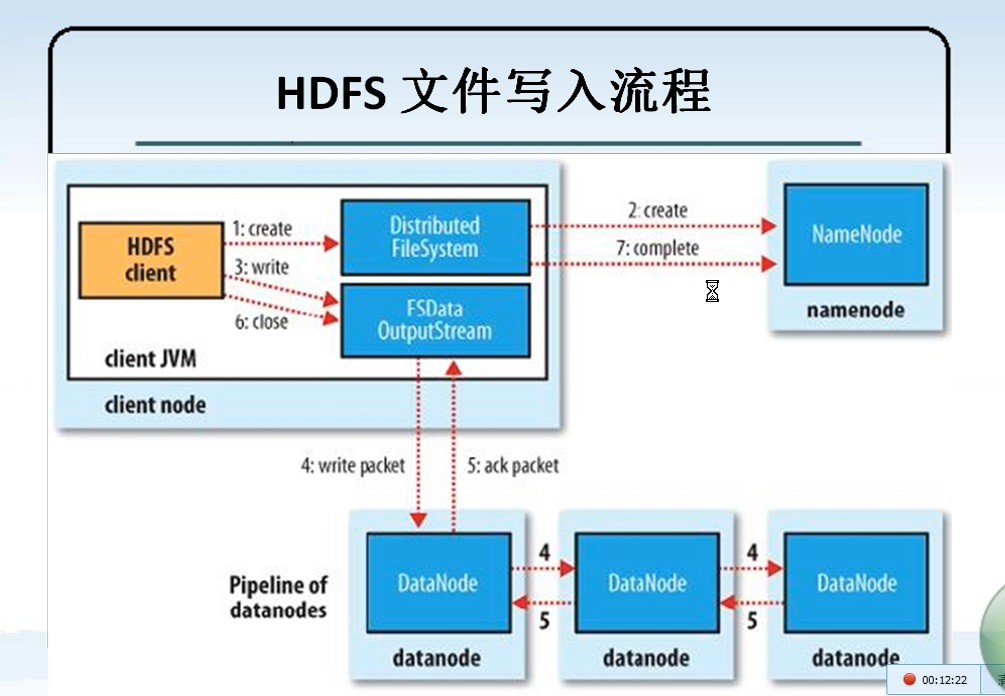
1. 当读取到block结尾时，FSDataInputStream关闭与当前DataNode的连接，并为读取下一个block寻找最近的DataNode
2. 读取完一个Block都会进行checksum验证，如果验证失败，客户端会通知

NameNode，然后读取下一个最近的拥有该block拷贝的DataNode

1. 如果block列表读取完，文件仍然没有结束，那么FileSystem再向NameNode获取下一批block列表

关闭FSDataInputStream.

###### 文件写入流程



Client调用FileSystem的create()方法:

1. FileSystem向NameNode发送请求，在namespace中创建新文件，但并不关联任何块
2. NameNode验证该文件是否已经存在，操作权限。如果验证通过，NameNode记录新文件信息，并且在某一DataNode上创建数据块
3. 返回FSDataOutputStream, 并将Client引导至数据块执行写入操作

Client调用FSDataOutputStream的write()方法：

HDFS默认将每个数据块放置3份。FSDataOutputStream先将数据写到第一个节点，数据包由第一个节点传送并写入到第二个节点，再由第二个节点到三个节点，最后在由3->2-

>1->FSDataOutputStream放回ack包，确认复制成功。

Client调用流的close()方法：

flush缓冲区的数据包，block完成复制份数后，向NN返回成功消息

**HDFS**高可靠性如何实现： **1.**一个**NameNode**节点多个**DataNode**节点

**2.**数据备份机制

对数据进行冗余存储

机架存放策略（2种）

**3.**故障检测机制

心跳机制 DataNode每3秒向NameNode发送心跳包，表示节点还活着，然后从

NameNode接受返回信息，返回信息中可能包括一些数据操作指令。如果NameNode超过10分钟没有接收到某个DataNode节点的心跳包，则表明该节点已经死亡。

块报告 DataNode每1个小时向NameNode发送块报告，快报告包含该节点所有的块信息。

数据完整性检测

##### SecondaryNameNode辅助NameNode

###### **5.**垃圾回收机制安全模式

hadoop安全模式

在分布式文件系统启动的时候，开始的时候会有安全模式，当分布式文件系统处于安全模式的情况下，文件系统中的内容不允许修改也不允许删除，直到安全模式结束。安全模式主要是为了系统启动的时候检查各个DataNode上数据块的有效性，同时根据策略必要的复制或者删除部分数据块。运行期通过命令也可以进入 安全模式。在实践过程中， 系统启动的时候去修改和删除文件也会有安全模式不允许修改的出错提示，只需要等待一会儿即可。

NameNode在启动的时候首先进入安全模式，如果datanode丢失的block达到一定的比例

（1- dfs.safemode.threshold.pct），则系统会一直处于安全模式状态即只读状态。

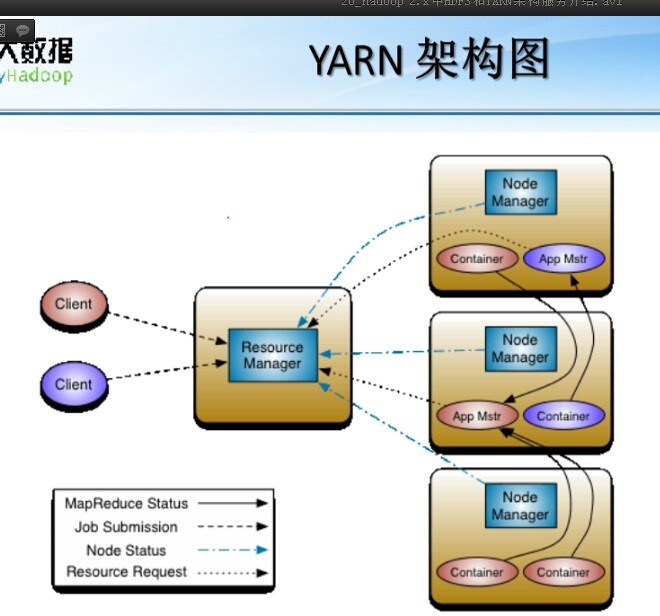
dfs.safemode.threshold.pct（缺省值0.999f）表示HDFS启动的时候，如果DataNode上报的block个数达到了 元数据记录的block个数的0.999倍才可以离开安全模式，否则一直是这种只读模式。如果设为1则HDFS永远是处于SafeMode。

修改dfs.safemode.threshold.pct为一个比较小的值，缺省是0.999。

hadoop dfsadmin -safemode leave命令强制离开

**Hadoop2.x** 学习笔记

Uber模式是hadoop2.x为MR小作业的一个优化机制。如果MapReduce作业足够小，所有的任务在一个JVM中完成。默认情况下为false.



MRv1与MRv2 yarn平台把传统的jobtracker干的事情分成了两个部分，分开来做。传统的

jobtracker既要管理资源分配，又要监听每一个任务的执行情况。这样对于Jobtracker节点来说压力非常大。yarn平台将资源管理与任务执行分开管理。有ResourceManager负责总的资源管理，相当与NameNode，NodeManager负责节点的资源管理。每当client提交任务时，

ResourceManager就会为这个任务创建一个Application Master来监控任务的执行情况,向ResourceManager汇总。App Mst 向ResourceManager申请资源，RM根据情况在

NodeManager分配container给App Mst这个任务使用。

数据库

数据库索引

###### 建立数据库索引的作用

数据库索引是将数据库表中的某一列或几列以特定的数据结构存起来，比如B-Tree，

Hash等，这样查找的时候就可以不用从头插到尾要O(n)，这样可以缩短到O(log)级别甚至O(1)。

建立索引之后查找和修改，排序等操作可以省很多时间。

索引是对数据库表中一个或多个列（例如，employee 表的姓名 (name) 列）的值进行排序的结构。如果想按特定职员的姓来查找他或她，则与在表中搜索所有的行相比，索引有助于更快地获取信息。

例如这样一个查询：select \* from table1 where id=10000。如果没有索引，必须遍历整个表，直到ID等于10000的这一行被找到为止；有了索引之后(必须是在ID这一列上建立的索引)，即可在索引中查找。由于索引是经过某种算法优化过的，因而查找次数要少的多。可见，索引是用来定位的。

数据库索引好比是一本书前面的目录，能加快数据库的查询速度。

###### 聚簇索引与非聚簇索引

索引分为聚簇索引和非聚簇索引两种，聚簇索引 是按照数据存放的物理位置为顺序的， 就像书中的目录，内容是按照页码顺序排列的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

注意一个表只能有一个聚集索引，但是可以由多个非聚集索引。

###### 唯一索引 主键索引 聚集索引

接下来说以下三种不同的索引： 根据数据库的功能，可以在数据库设计器中创建三种索引：唯一索引、主键索引和聚集索引。 提示：尽管唯一索引有助于定位信息，但为获得最佳性能结果，建议改用主键或唯一约束。

唯一索引 唯一索引是不允许其中任何两行具有相同索引值的索引。不是只能建一个索引。 当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在

employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。

主键索引 数据库表经常有一列或多列组合，其值唯一标识表中的每一行。该列称为表的主键。在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引 在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

索引列 可以基于数据库表中的单列或多列创建索引。多列索引可以区分其中一列可能有相同值的行。

如果经常同时搜索两列或多列或按两列或多列排序时，索引也很有帮助。例如，如果经常在同一查询中为姓和名两列设置判据，那么在这两列上创建多列索引将很有意义。

###### 索引建立语法

接下来说一下在数据库中建立索引的语法：

（1）可以用create index创建 create [unique | CLUSTERED | UNCLUSTERED ] index 索引名 [索引类型] on 表名（列名（n） [asc|desc ]，列名，列名...）； 索引名命名最好用tablename\_ColName1\_ColName2 索引类型可以指定为BTREE或者HASH等 n代表列中的几位做索引，可以省空间 asc desc 升序降序。

（2）可以在alter table 语句创建 alter table 表名 add [UNIQUE|FULLTEXT] index 索引名 [using [BTREE|HASH]] (列名)； 例如alter table xs add index xs\_sm using btree(姓名); alter table xs add index mark(出生日期，姓名)

（3）可以在create table时创建create table 表名{

学号 char（6）, 课程号 char(3), 成 绩 tinyint(1), 学分 tinyint(1),

primary key(学号，课程号);

index cj(成绩)//index 索引名 [索引类型] (索引列名)

}

###### 删除索引的方法：

* 1. DROP INDEX 索引名 ON 表名
  2. ALTER TABLE 表名 DROP INDEX 索引名

###### 查看一个表的索引：

show index in 表名

事务

数据库事务（Transaction）是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。一方面，当多个应用程序并发访问数据库时，事务可以在应用程序间提供一个隔离方法，防止互相干扰。另一方面，事务为数据库操作序列提供了一个从失败恢复正常的方法。

#### 事务的四个特性

事务具有四个特性：原子性（Atomicity）、一致性（Consistency）、隔离型（Isolation）、持久性（Durability），简称ACID。

原子性（Atomicity） 事务的原子性是指事务中的操作不可拆分，只允许全部执行或者全部不执行。

一致性（Consistency） 事务的一致性是指事务的执行不能破坏数据库的一致性，一致性也称为完整性。一个事务在执行后，数据库必须从一个一致性状态转变为另一个一致性状态。

隔离型（Isolation） 事务的隔离型是指并发的事务相互隔离，不能互相干扰。

持久性（Durability） 事务的持久性是指事务一旦提交，对数据的状态变更应该被永久保存。

#### 数据库隔离级别

对于同时运行的多个事务,当这些事务访问数据库中相同的数据时,如果没有采取必要的隔离机制,就会导致各种并发问题:

脏读：对于两个事务T1，T2，T1读取了已经被T2更新但还没有提交的字段，之后，若T2 回滚，T1读取到的内容就是临时无效的内容。

不可重复读：对于事务T1，T2，T1需要读取一个字段两次，在第一次和第二次读取之间，T2更新了该字段，导致T1第二次读取到的内容值不同。

幻读： 事务A读取与搜索条件相匹配的若干行。事务B以插入或删除行等方式来修改事务

A的结果集，然后再提交。 幻读与不可重复读之间的区别是幻读强调的是新增或删除,而不可重复读强调的是修改。比如Mary两次查工资，中间有人改过工资，则两次结果不一样，这就是不可重复读。Mary要查工资一千的人数，第一次查到了10个，中间有人增加了一条工资为一千的人，下次查的时候就变成了11个，好像第一次查询的是幻觉一样。

事务的四个隔离级别 实际工作中事务几乎都是并发的，完全做到互相之间不干扰会严重牺牲性能，为了平衡隔离型和性能，SQL92规范定义了四个事务隔离级别：读未提交（Read

Uncommitted）、读已提交（Read Committed）、可重复读（Repeatable Read）、串行化

（Serializable）。四个级别逐渐增强，每个级别解决上个级别的一个问题。

读未提交（Read Uncommitted） 另一个事务修改了数据，但尚未提交，而本事务中的

SELECT会读到这些未被提交的数据（脏读）。 脏读是指另一个事务修改了数据，但尚未提交，而本事务中的SELECT会读到这些未被提交的数据。

读已提交（Read Committed） 本事务读取到的是最新的数据（其他事务提交后的）。问题是，在同一个事务里，前后两次相同的SELECT会读到不同的结果（不可重复读）。

不可重复读是指同一个事务执行过程中，另外一个事务提交了新数据，因此本事务先后两次读到的数据结果会不一致。

可重复读（Repeatable Read） 在同一个事务里，SELECT的结果是事务开始时间点的状态，同样的SELECT操作读到的结果会是一致的。但是，会有幻读现象。

可重复读保证了同一个事务里，查询的结果都是事务开始时的状态（一致性）。但是，如果另一个事务同时提交了新数据，本事务再更新时，就会发现了这些新数据，貌似之前读到的数据是幻觉，这就是幻读。

串行化（Serializable） 所有事务只能一个接一个串行执行，不能并发。

#### 隔离级别的选择

事务隔离级别越高，越能保证数据的一致性，但对并发性能影响越大，一致性和高性能必须有所取舍或折中。

一般情况下，多数应用程序可以选择将数据库的隔离级别设置为读已提交，这样可以避免脏读，也可以得到不错的并发性能。尽管这个隔离级别会导致不可重复度、幻读，但这种个别场合应用程序可以通过主动加锁进行并发控制。

Oracle支持两种隔离级别，READ COMMITED和SERIALIZABLE默认的事务隔离级别是READ COMMITED

MYSQL支持4中隔离界别，默认的是REPEATED READ

###### 数据库中的锁：

锁的种类：

按照锁的内容上分为：表锁，页锁，行锁。InnoDB支持行锁，MyIsam支持表锁，

BerkeleyDBA支持页锁。不同的粒度

按照锁的性质分可以分为：共享锁（读锁或S锁）,独占锁（写锁，排它锁）（X锁），更新锁

（U锁）（意向锁）

当执行select时候，加共享锁，当insert update delete等操作时，加排它锁。更新锁首先对数据对象作更新锁锁定，这样数据将不能被修改，但可以读取。等到SQL Server确定要进行更新数据操作时，他会自动将更新锁换为独占锁，当对象上有其他锁存在时，无法对其加更新锁。

意向锁是为了提高封锁子系统的效率。该封锁子系统支持多种封锁粒度。原因是:在多粒度封锁方法中一个数据对象可能以两种方式加锁 ― 显式封锁和隐式封锁。

数据库引擎使用意向锁来保护锁层次结构的底层资源，以防止其他事务对自己锁住的资源造成伤害，提高锁冲突检测性能。例如，当读取表里的页面时，在请求页共享锁（S锁）之前， 事务在表级请求共享意向锁。这样可以防止其他事务随后在表上获取排他锁（X锁），修改整个表格。意向锁可以提高性能，因为数据库引擎仅在表级检查意向锁，确定事务是否能安全地获取该表上的锁，而不需要检查表中的每行或每页上的锁以确定事务是否可以锁定整个

表。

###### 悲观所与乐观锁

1、悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系

统不会修改数据）。

2、乐观锁（ Optimistic Locking ）

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销， 特别是对长事务而言，这样的开销往往无法承受。

而乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案

中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同

读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

**Myisam**与**InnoDB**的区别：

最主要的一点是MyIsam不支持事务操作，外键以及行级锁，而InnoDB支持这些功能。MyIsam只支持表级锁，当多线程并发操作数据库时，就会为整个表上锁，而InnoDB 则从行上锁。但这也是不一定的，如果一个SQL语句不能确定扫描范围的话，也会为整个表上锁，比如： update table set num=1 where name like “%aaa%”

但是MyIsam的读的速度比InnoDB更加快一些，InnoDB不保存表的行数，比如要执行

count(\*)的话，它就会扫描整个表，但是如果是MyIsam的话，直接返回行数

InnoDB执行Drop from table 时，不会重新建表，而是一行一行删除。如果执行大量的

select操作选择MyIsam，如果有大量的insert和update操作，选择InnoDB。

AUTO\_INCREMENT 对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中，可以和其他字段一起建立联合索引

表的存储方面：InnoDB的存储方式是一个表空间数据文件，一个日志文件，而MyIsam的存储方式是索引文件（.MYI（myindex）），数据文件(.MYD(mydata))，表存储定义文件

(.frm)

**MyISAM**与**InnoDB**的区别是什么？

**1**、 存储结构

MyISAM：每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。

InnoDB：所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。

**2**、 存储空间

MyISAM：可被压缩，存储空间较小。支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。

InnoDB：需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

**3**、 可移植性、备份及恢复

MyISAM：数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。

InnoDB：免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

**4**、 事务支持

MyISAM：强调的是性能，每次查询具有原子性,其执行数度比InnoDB类型更快，但是不提供事务支持。

InnoDB：提供事务支持事务，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback) 和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant)) 型表。

5、 AUTO\_INCREMENT

MyISAM：可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。

InnoDB：InnoDB中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

**6**、 表锁差异

MyISAM：只支持表级锁，用户在操作myisam表时，select，update，delete，insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。

InnoDB：支持事务和行级锁，是innodb的最大特色。行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。

**7**、 全文索引

MyISAM：支持 FULLTEXT类型的全文索引

InnoDB：不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。

**8**、 表主键

MyISAM：允许没有任何索引和主键的表存在，索引都是保存行的地址。

InnoDB：如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。

**9**、 表的具体行数

MyISAM：保存有表的总行数，如果select count(\*) from table;会直接取出出该值。

InnoDB：没有保存表的总行数，如果使用select count(\*) from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，myisam和innodb处理的方式都一样。

**10**、 **CURD**操作

MyISAM：如果执行大量的SELECT，MyISAM是更好的选择。

InnoDB：如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用

InnoDB表。DELETE 从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate

table这个命令。

**11**、 外键

MyISAM：不支持

InnoDB：支持

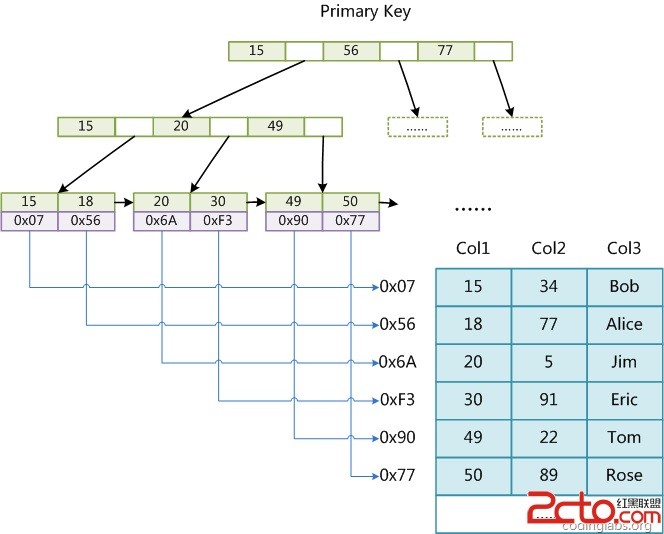
通过上述的分析，基本上可以考虑使用InnoDB来替代MyISAM引擎了，原因是InnoDB自身很多良好的特点，比如事务支持、存储 过程、视图、行级锁定等等，在并发很多的情况下，相信InnoDB的表现肯定要比MyISAM强很多。另外，任何一种表都不是万能的，只用恰当的针对业务类型来选择合适的表类型，才能最大的发挥MySQL的性能优势。如果不是很复杂的

Web应用，非关键应用，还是可以继续考虑MyISAM的，这个具体情况可以自己斟酌。

**myisam**和**innodb**索引实现的不同

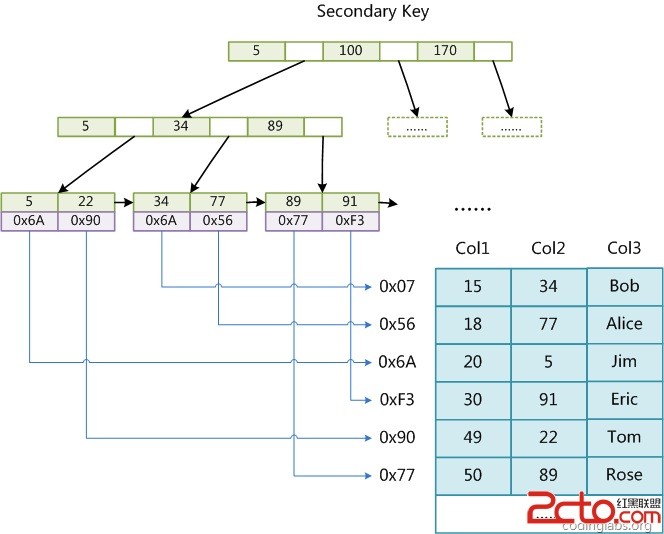
MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是

MyISAM索引的原理图：



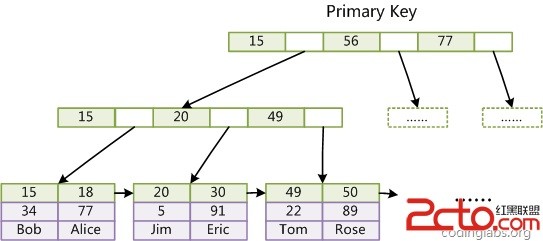
这里设表一共有三列，假设我们以Col1为主键，则上图是一个MyISAM表的主索引（Primary

key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



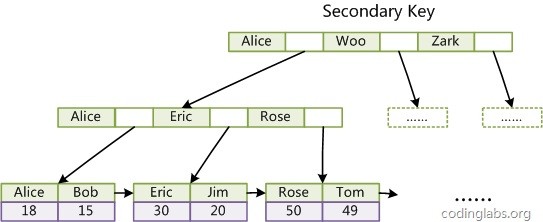
同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

InnoDB索引实现虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL[系统](http://www.2cto.com/os/)会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地 址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了

InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主 键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

* 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。 考虑一个订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，

CustomerCity）主键是（OrderID）。 其中 OrderDate，CustomerID，CustomerName，

CustomerAddr，CustomerCity 等非主键列都完全依赖于主键（OrderID），所以符合 2NF。不过问题是 CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。 通过拆分【Order】为【Order】（OrderID，OrderDate，CustomerID）和【Customer】

（CustomerID，CustomerName，CustomerAddr，CustomerCity）从而达到 3NF。 第二范式（2NF）和第三范式（3NF）的概念很容易混淆，区分它们的关键点在于，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。

# NoSQL

NoSQL，泛指非关系型的数据库。

1. key-value存储

|  |  |
| --- | --- |
| **Examples** | **Tokyo Cabinet\/Tyrant, Redis, Voldemort, Oracle BDB** |
| 典型应用场景 | 内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等等。 |
| 数据模型 | Key 指向 Value 的键值对，通常用hash table来实现 |
| 强项 | 查找速度快 |
| 弱项 | 数据无结构化，通常只被当作字符串或者二进制数据 |

**2.**列式数据库

|  |  |
| --- | --- |
| **Examples** | **Cassandra, HBase, Riak** |
| 典型应用场景 | 分布式的文件系统 |
| 数据模型 | 以列簇式存储，将同一列数据存在一起 |
| 强项 | 查找速度快，可扩展性强，更容易进行分布式扩展 |
| 弱项 | 功能相对局限 |

**3.**文档型数据库

|  |  |
| --- | --- |
| **Examples** | **CouchDB, MongoDb** |
| 典型应用场景 | Web应用（与Key-Value类似，Value是结构化的，不同的是数据库能够了解Value的内容） |
| 数据模型 | Key-Value对应的键值对，Value为结构化数据 |
| 强项 | 数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构 |
| 弱项 | 查询性能不高，而且缺乏统一的查询语法。 |

**4.**图结构数据库

|  |  |
| --- | --- |
| **Examples** | **Neo4J, InfoGrid, Infinite Graph** |
| 典型应用场景 | 社交网络，推荐系统等。专注于构建关系图谱 |
| 数据模型 | 图结构 |
| 强项 | 利用图结构相关算法。比如最短路径寻址，N度关系查找等 |
| 弱项 | 很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式的集群方案。 |

一致性**Hash**算法背景

一致性哈希算法在1997年由麻省理工学院的Karger等人在解决分布式Cache中提出的，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和CARP十分类似。他在Dynamo数据库中有所应用。一致性哈希修正了CARP使用的简单哈希算法带来的问题，使得DHT可以在P2P 环境中真正得到应用。

但现在一致性hash算法在分布式系统中也得到了广泛应用，研究过memcached缓存数据库的人都知道，memcached服务器端本身不提供分布式cache的一致性，而是由客户端来提供， 具体在计算一致性hash时采用如下步骤：

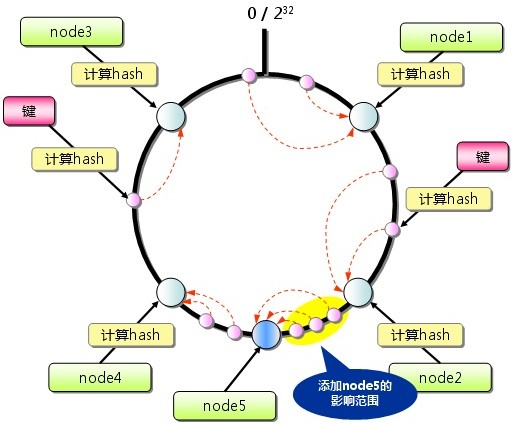
* 1. 首先求出memcached服务器（节点）的哈希值，并将其配置到0～232的圆

（continuum）上。

* 1. 然后采用同样的方法求出存储数据的键的哈希值，并映射到相同的圆上。
  2. 然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过232仍然找不到服务器，就会保存到第一台memcached服务器上。



从上图的状态中添加一台memcached服务器。余数分布式算法由于保存键的服务器会发生巨大变化而影响缓存的命中率，但Consistent Hashing中，只有在园（continuum）上增加服务器的地点逆时针方向的第一台服务器上的键会受到影响，如下图所示：



一致性**Hash**性质

考虑到分布式系统每个节点都有可能失效，并且新的节点很可能动态的增加进来，如何保证当系统的节点数目发生变化时仍然能够对外提供良好的服务，这是值得考虑的，尤其实在设计分布式缓存系统时，如果某台服务器失效，对于整个系统来说如果不采用合适的算法来保证一致性，那么缓存于系统中的所有数据都可能会失效（即由于系统节点数目变少，客户端在请求某一对象时需要重新计算其hash值（通常与系统中的节点数目有关），由于hash值已经改变，所以很可能找不到保存该对象的服务器节点），因此一致性hash就显得至关重要， 良好的分布式cahce系统中的一致性hash算法应该满足以下几个方面：

平衡性**(Balance)**

平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

单调性(Monotonicity)

单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲区加入到系统中，那么哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲区中去，而不会被映射到旧的缓冲集合中的其他缓冲区。简单的哈希算法往往不能满足单调性的要求，如最简单的线性哈希：x = (ax + b) mod (P)，在上式中，P表示全部缓冲的大小。不难看出，当缓冲大小发生变化时(从P1到P2)，原来所有的哈希结果均会发生变化，从而不满足单调性的要

求。哈希结果的变化意味着当缓冲空间发生变化时，所有的映射关系需要在系统内全部更

新。而在P2P系统内，缓冲的变化等价于Peer加入或退出系统，这一情况在P2P系统中会频繁发生，因此会带来极大计算和传输负荷。单调性就是要求哈希算法能够应对这种情况。

分散性**(Spread)**

在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效 率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

负载**(Load)**

负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同的内 容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

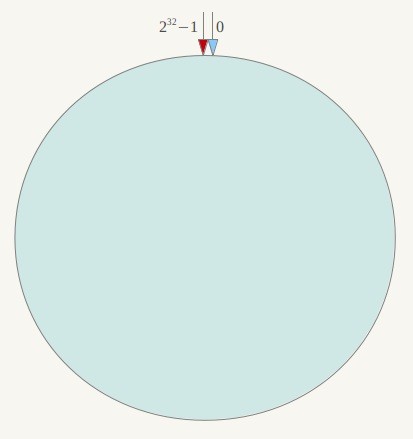
平滑性(Smoothness)

平滑性是指缓存服务器的数目平滑改变和缓存对象的平滑改变是一致的。

原理

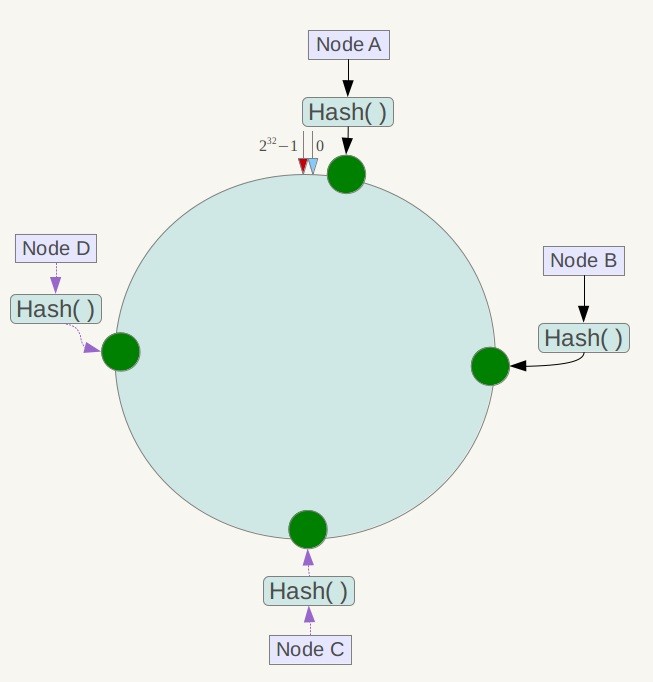
###### 基本概念

[一致性哈希算法（Consistent Hashing）最早在论文《Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web》](http://dl.acm.org/citation.cfm?id=258660)中被提出。简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H 的值空间为0-2^32-1（即哈希值是一个32位无符号整形），整个哈希空间环如下：



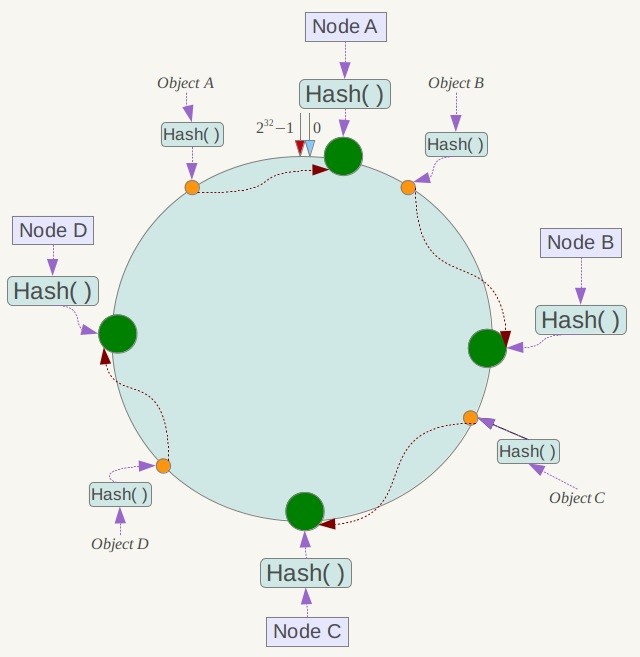
整个空间按顺时针方向组织。0和232-1在零点中方向重合。

下一步将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用ip 地址哈希后在环空间的位置如下：



接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。

例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



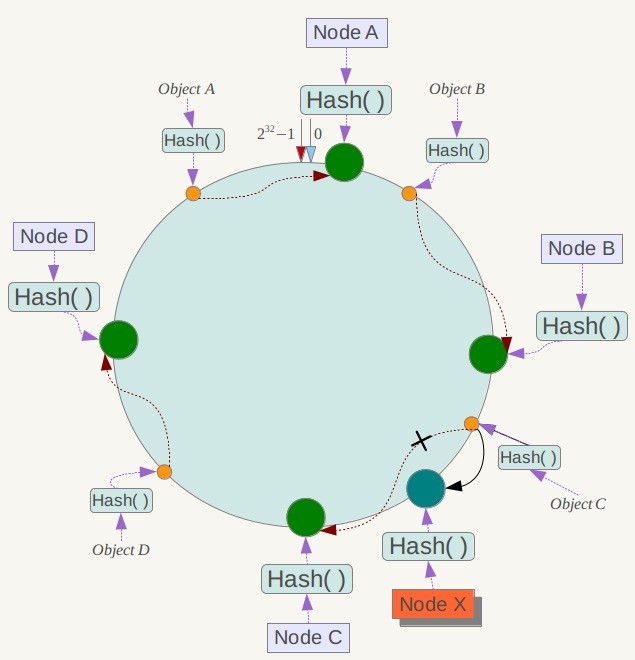
根据一致性哈希算法，数据A会被定为到Node A上，B被定为到Node B上，C被定为到Node

C上，D被定为到Node D上。

下面分析一致性哈希算法的容错性和可扩展性。现假设Node C不幸宕机，可以看到此时对象

A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

下面考虑另外一种情况，如果在系统中增加一台服务器Node X，如下图所示：

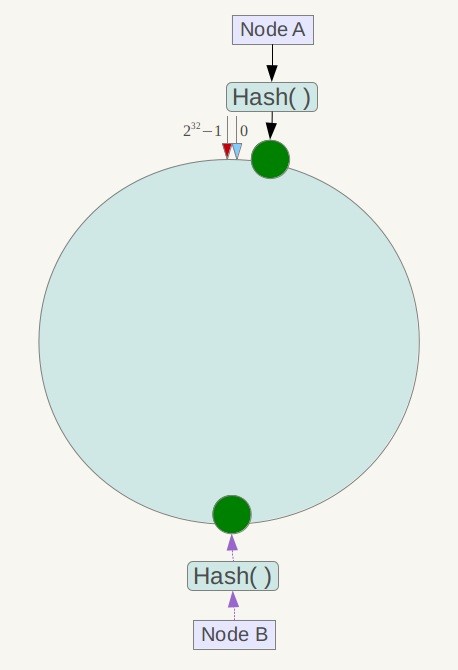


此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X 。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影

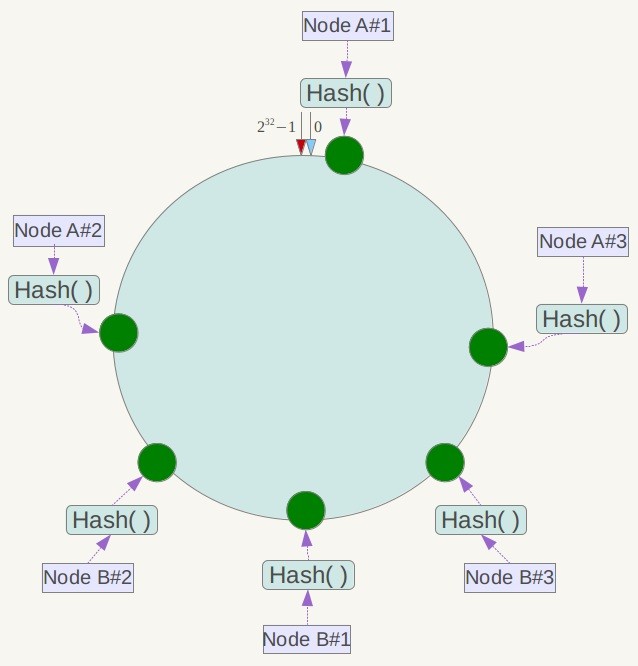
响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

另外，一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题。例如系统中只有两台服务器，其环分布如下，



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器ip或主机名的后面增加编号来实现。例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：



同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node

A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

PrepareStatement与Statement区别

PrepareStatement预编译过后就可以在缓存中，下次执行时就不需要重新编译，而Statement

每次都要重新编译。

PrepareStatement代码可读性更强，虽然多了几行代码，但是一目了然，比Statement的语句强了太多。

PrepareStatement更加安全，避免SQL注入，or '1'='1'

PreparedStatement 相 对 Statement有以下 优 点：

1.防注入攻击

2.多次运行速度快

3.防止数据库缓冲区溢出

4.代 码 的可读性可维护性好

**MySQL** 数据库性能优化之缓存参数优化

数据库属于 IO 密集型的应用程序，其主要职责就是数据的管理及存储工作。而我们知道，从内存中读取一个数据库的时间是微秒级别，而从一块普通硬盘上读取一个IO是在毫秒级别， 二者相差3个数量级。所以，要优化数据库，首先第一步需要优化的就是 IO，尽可能将磁盘

IO转化为内存IO。本文先从 MySQL 数据库IO相关参数\(缓存参数\)的角度来看看可以通过哪些参数进行IO优化。

**query\_cache\_size\/query\_cache\_type (global)** Query cache 作用于整个 MySQL

Instance，主要用来缓存 MySQL 中的 ResultSet，也就是一条SQL语句执行的结果集，所以仅仅只能针对select语句。当我们打开了 Query Cache 功能，MySQL在接受到一条select语句的请求后，如果该语句满足Query Cache的要求(未显式说明不允许使用Query Cache，或者已经显式申明需要使用Query Cache)，MySQL 会直接根据预先设定好的HASH算法将接受到的select语句以字符串方式进行hash，然后到Query Cache 中直接查找是否已经缓存。也就是说，如果已经在缓存中，该select请求就会直接将数据返回，从而省略了后面所有的步骤

(如 SQL语句的解析，优化器优化以及向存储引擎请求数据等)，极大的提高性能。

当然，Query Cache 也有一个致命的缺陷，那就是当某个表的数据有任何任何变化，都会导致所有引用了该表的select语句在Query Cache 中的缓存数据失效。所以，当我们的数据变化非常频繁的情况下，使用Query Cache 可能会得不偿失。

Query Cache的使用需要多个参数配合，其中最为关键的是 query\_cache\_size 和query\_cache\_type ，前者设置用于缓存 ResultSet 的内存大小，后者设置在何场景下使用Query Cache。在以往的经验来看，如果不是用来缓存基本不变的数据的MySQL数据库，

query\_cache\_size 一般 256MB 是一个比较合适的大小。当然，这可以通过计算Query

Cache的命中率(Qcache\_hits\/(Qcache\_hits+Qcache\_inserts)\*100))来进行调整。

query\_cache\_type可以设置为0(OFF)，1(ON)或者2(DEMOND)，分别表示完全不使用query

cache，除显式要求不使用query cache(使用sql\_no\_cache)之外的所有的select都使用query

cache，只有显示要求才使用query cache(使用sql\_cache)。

**binlog\_cache\_size (global)** Binlog Cache 用于在打开了二进制日志(binlog)记录功能的环境，是 MySQL 用来提高binlog的记录效率而设计的一个用于短时间内临时缓存binlog数据的内存区域。

一般来说，如果我们的数据库中没有什么大事务，写入也不是特别频繁，2MB～4MB是一个合适的选择。但是如果我们的数据库大事务较多，写入量比较大，可与适当调高binlog\_cache\_size。同时，我们可以通过binlog\_cache\_use 以及 binlog\_cache\_disk\_use来分析设置的binlog\_cache\_size是否足够，是否有大量的binlog\_cache由于内存大小不够而使用临时文件(binlog\_cache\_disk\_use)来缓存了。

**key\_buffer\_size (global)** Key Buffer 可能是大家最为熟悉的一个 MySQL 缓存参数了，尤其是在 MySQL 没有更换默认存储引擎的时候，很多朋友可能会发现，默认的 MySQL 配置文件中设置最大的一个内存参数就是这个参数了。key\_buffer\_size 参数用来设置用于缓存

MyISAM存储引擎中索引文件的内存区域大小。如果我们有足够的内存，这个缓存区域最好是能够存放下我们所有的 MyISAM 引擎表的所有索引，以尽可能提高性能。

此外，当我们在使用MyISAM 存储的时候有一个及其重要的点需要注意，由于 MyISAM 引擎的特性限制了他仅仅只会缓存索引块到内存中，而不会缓存表数据库块。所以，我们的 SQL 一定要尽可能让过滤条件都在索引中，以便让缓存帮助我们提高查询效率。

bulk\_insert\_buffer\_size (thread) 和key\_buffer\_size一样，这个参数同样也仅作用于使用

MyISAM存储引擎，用来缓存批量插入数据的时候临时缓存写入数据。当我们使用如下几种数据写入语句的时候，会使用这个内存区域来缓存批量结构的数据以帮助批量写入数据文件：

insert … select … insert … values (…) ,(…),(…)… load data infile… into… (非空表)

**innodb\_buffer\_pool\_size(global)** 当我们使用InnoDB存储引擎的时候，

innodb\_buffer\_pool\_size 参数可能是影响我们性能的最为关键的一个参数了，他用来设置用于缓存 InnoDB 索引及数据块的内存区域大小，类似于 MyISAM 存储引擎的 key\_buffer\_size 参数，当然，可能更像是 Oracle 的 db\_cache\_size。简单来说，当我们操作一个 InnoDB 表的时候，返回的所有数据或者去数据过程中用到的任何一个索引块，都会在这个内存区域中走一遭。

和key\_buffer\_size 对于 MyISAM 引擎一样，innodb\_buffer\_pool\_size 设置了 InnoDB 存储引擎需求最大的一块内存区域的大小，直接关系到 InnoDB存储引擎的性能，所以如果我们有足够的内存，尽可将该参数设置到足够打，将尽可能多的 InnoDB 的索引及数据都放入到该缓存区域中，直至全部。

我们可以通过 (Innodb\_buffer\_pool\_read\_requests – Innodb\_buffer\_pool\_reads) \/ Innodb\_buffer\_pool\_read\_requests \* 100% 计算缓存命中率，并根据命中率来调整

innodb\_buffer\_pool\_size 参数大小进行优化。

**innodb\_additional\_mem\_pool\_size(global)** 这个参数我们平时调整的可能不是太多，很多人都使用了默认值，可能很多人都不是太熟悉这个参数的作用。

innodb\_additional\_mem\_pool\_size 设置了InnoDB存储引擎用来存放数据字典信息以及一些内部数据结构的内存空间大小，所以当我们一个MySQL Instance中的数据库对象非常多的时候，是需要适当调整该参数的大小以确保所有数据都能存放在内存中提高访问效率的。

这个参数大小是否足够还是比较容易知道的，因为当过小的时候，MySQL 会记录 Warning 信息到数据库的 error log 中，这时候你就知道该调整这个参数大小了。

**innodb\_log\_buffer\_size (global)** 这是 InnoDB 存储引擎的事务日志所使用的缓冲区。类似于 Binlog Buffer，InnoDB 在写事务日志的时候，为了提高性能，也是先将信息写入Innofb Log Buffer 中，当满足 innodb\_flush\_log\_trx\_commit 参数所设置的相应条件(或者日志

缓冲区写满)之后，才会将日志写到文件(或者同步到磁盘)中。可以通过

innodb\_log\_buffer\_size 参数设置其可以使用的最大内存空间。

注：innodb\_flush\_log\_trx\_commit 参数对 InnoDB Log 的写入性能有非常关键的影响。该参数可以设置为0，1，2，解释如下：

0：log buffer中的数据将以每秒一次的频率写入到log file中，且同时会进行文件系统到磁盘的同步操作，但是每个事务的commit并不会触发任何log buffer 到log file的刷新或者文件系统到磁盘的刷新操作;

1：在每次事务提交的时候将log buffer 中的数据都会写入到log file，同时也会触发文件系统到磁盘的同步;

2：事务提交会触发log buffer 到log file的刷新，但并不会触发磁盘文件系统到磁盘的同步。此外，每秒会有一次文件系统到磁盘同步操作。

此外，MySQL文档中还提到，这几种设置中的每秒同步一次的机制，可能并不会完全确保非常准确的每秒就一定会发生同步，还取决于进程调度的问题。实际上，InnoDB 能否真正满足此参数所设置值代表的意义正常 Recovery 还是受到了不同 OS 下文件系统以及磁盘本身的限制，可能有些时候在并没有真正完成磁盘同步的情况下也会告诉 mysqld 已经完成了磁盘同步。

**innodb\_max\_dirty\_pages\_pct (global)** 这个参数和上面的各个参数不同，他不是用来设置用于缓存某种数据的内存大小的一个参数，而是用来控制在 InnoDB Buffer Pool 中可以不用写入数据文件中的Dirty Page 的比例(已经被修但还没有从内存中写入到数据文件的脏数据)。这个比例值越大，从内存到磁盘的写入操作就会相对减少，所以能够一定程度下减少写入操作的磁盘IO。

但是，如果这个比例值过大，当数据库 Crash 之后重启的时间可能就会很长，因为会有大量的事务数据需要从日志文件恢复出来写入数据文件中。同时，过大的比例值同时可能也会造成在达到比例设定上限后的 flush 操作“过猛”而导致性能波动很大。

上面这几个参数是 MySQL 中为了减少磁盘物理IO而设计的主要参数，对 MySQL 的性能起到了至关重要的作用。

相关参数的建议值 按照 **mcsrainbow** 朋友的要求，这里列一下根据以往经验得到的相关参数的建议值：

query\_cache\_type : 如果全部使用innodb存储引擎，建议为0，如果使用MyISAM 存储引擎， 建议为2，同时在SQL语句中显式控制是否是哟你gquery cache

query\_cache\_size: 根据 命中率(Qcache\_hits\/(Qcache\_hits+Qcache\_inserts)\*100))进行调整，一般不建议太大，256MB可能已经差不多了，大型的配置型静态数据可适当调大

binlog\_cache\_size: 一般环境2MB～4MB是一个合适的选择，事务较大且写入频繁的数据库环境可以适当调大，但不建议超过32MB

key\_buffer\_size: 如果不使用MyISAM存储引擎，16MB足以，用来缓存一些系统表信息等。如果使用 MyISAM存储引擎，在内存允许的情况下，尽可能将所有索引放入内存，简单来说就是“越大越好”

bulk\_insert\_buffer\_size: 如果经常性的需要使用批量插入的特殊语句(上面有说明)来插入数据，可以适当调大该参数至16MB～32MB，不建议继续增大，某人8MB

innodb\_buffer\_pool\_size: 如果不使用InnoDB存储引擎，可以不用调整这个参数，如果需要使用，在内存允许的情况下，尽可能将所有的InnoDB数据文件存放如内存中，同样将但来说也是“越大越好”

innodb\_additional\_mem\_pool\_size: 一般的数据库建议调整到8MB～16MB，如果表特别多， 可以调整到32MB，可以根据error log中的信息判断是否需要增大

innodb\_log\_buffer\_size: 默认是1MB，系的如频繁的系统可适当增大至4MB～8MB。当然如上面介绍所说，这个参数实际上还和另外的flush参数相关。一般来说不建议超过32MB

innodb\_max\_dirty\_pages\_pct: 根据以往的经验，重启恢复的数据如果要超过1GB的话，启动速度会比较慢，几乎难以接受，所以建议不大于 1GB\/innodb\_buffer\_pool\_size(GB)\*100 这个值。当然，如果你能够忍受启动时间比较长，而且希望尽量减少内存至磁盘的flush，可以将这个值调整到90，但不建议超过90

注：以上取值范围仅仅只是我的根据以往遇到的数据库场景所得到的一些优化经验值，并不一定适用于所有场景，所以在实际优化过程中还需要大家自己不断的调整分析，也欢迎大家随时通过 Mail 与我联系沟通交流优化或者是架构方面的技术，一起探讨相互学习。

操作系统

进程与线程的区别

进程是系统分配资源的的基本单位。

线程是CPU调度的基本单位。线程基本不拥有系统资源，只有程序计数器，栈，部分寄存器。

进程间上下文切换消耗非常大，而线程上下文切换相对较小。一个进车更可以有多个线程。

进程有PCB，线程有TCB，TCB内的资源要比PCB的资源少很多，这也是为什么进程上下文切换消耗极大。

进程拥有一个完整的虚拟地址空间，不依赖于线程而独立存在；反之，线程是进程的一部分，没有自己的地址空间，与进程内的其他线程一起共享分配给该进程的所有资源。

进程间通信**IPC windows**端和**linux**端

一**.**在**Windows OS**上，有几种常用的进程间通信方式：

###### 文件映射

文件映射(Memory-Mapped Files)能使进程把文件内容当作进程地址区间一块内存那样来对待。因此，进程不必使用文件I\/O操作，只需简单的指针操作就可读取和修改文件的内容。

Win32 API允许多个进程访问同一文件映射对象，各个进程在它自己的地址空间里接收内存的指针。通过使用这些指针，不同进程就可以读或修改文件的内容，实现了对文件中数据的共享。

应用程序有三种方法来使多个进程共享一个文件映射对象。

(1)继承：第一个进程建立文件映射对象，它的子进程继承该对象的句柄。

(2)命名文件映射：第一个进程在建立文件映射对象时可以给该对象指定一个名字(可与文件名不同)。第二个进程可通过这个名字打开此文件映射对象。另外，第一个进程也可以通过一些其它IPC机制(有名管道、邮件槽等)把名字传给第二个进程。

(3)句柄复制：第一个进程建立文件映射对象，然后通过其它IPC机制(有名管道、邮件槽等)把对象句柄传递给第二个进程。第二个进程复制该句柄就取得对该文件映射对象的访问权限。

文件映射是在多个进程间共享数据的非常有效方法，有较好的安全性。但文件映射只能用于本地机器的进程之间，不能用于网络中，而开发者还必须控制进程间的同步。

###### 共享内存

Win32 API中共享内存(Shared Memory)实际就是文件映射的一种特殊情况。进程在创建文件映射对象时用0xFFFFFFFF来代替文件句柄(HANDLE)，就表示了对应的文件映射对象是从操作系统页面文件访问内存，其它进程打开该文件映射对象就可以访问该内存

块。由于共享内存是用文件映射实现的，所以它也有较好的安全性，也只能运行于同一计算机上的进程之间。

###### 匿名管道

管道(Pipe)是一种具有两个端点的通信通道：有一端句柄的进程可以和有另一端句柄的进程通信。管道可以是单向－一端是只读的，另一端点是只写的；也可以是双向的一管道的两端点既可读也可写。

匿名管道(Anonymous Pipe)是 在父进程和子进程之间，或同一父进程的两个子进程之间传输数据的无名字的单向管道。通常由父进程创建管道，然后由要通信的子进程继承通道的读端点句柄或写 端点句柄，然后实现通信。父进程还可以建立两个或更多个继承匿名管道读和写句柄的子进程。这些子进程可以使用管道直接通信，不需要通过父进程。

匿名管道是单机上实现子进程标准I\/O重定向的有效方法，它不能在网上使用，也不能用于两个不相关的进程之间。

1. 命名管道

命名管道(Named Pipe)是服务器进程和一个或多个客户进程之间通信的单向或双向管道。不同于匿名管道的是命名管道可以在不相关的进程之间和不同计算机之间使用，服务器建立命名管道时给它指定一个名字，任何进程都可以通过该名字打开管道的另一 端，根据给定的权限和服务器进程通信。

命名管道提供了相对简单的编程接口，使通过网络传输数据并不比同一计算机上两进程之间通信更困难，不过如果要同时和多个进程通信它就力不从心了。

###### 动态连接库

Win32动态连接库(DLL)中的全局数据可以被调用DLL的所有进程共享，这就又给进程间通信开辟了一条新的途径，当然访问时要注意同步问题。

虽然可以通过DLL进行进程间数据共享，但从数据安全的角度考虑，我们并不提倡这种方法，使用带有访问权限控制的共享内存的方法更好一些。

###### 远程过程调用

Win32 API提供的远程过程调用(RPC)使应用程序可以使用远程调用函数，这使在网络上用RPC进行进程通信就像函数调用那样简单。RPC既可以在单机不同进程间使用也可以在网络中使用。

由 于 Win32 API 提 供 的 RPC 服 从 OSF-DCE(Open Software Foundation Distributed Computing Environment)标准。所以通过Win32 API编写的RPC应用程序能与其它操作系统上支持DEC的RPC应用程序通信。使用RPC开发者可以建立高性能、紧密耦合的分布式应用程序。

1. **Sockets**

Windows Sockets规范是以U.C.Berkeley大学BSD UNIX中流行的Socket接口为范例定义的一套Windows下的网络编程接口。除了Berkeley Socket原有的库函数以外，还扩展了一组针对Windows的函数，使程序员可以充分利用Windows的消息机制进行编程。

现在通过Sockets实现进程通信的网络应用越来越多，这主要的原因是Sockets的跨平台性要比其它IPC机制好得多，另外WinSock 2.0不仅支持TCP\/IP协议，而且还支持其它协议(如IPX)。Sockets的唯一缺点是它支持的是底层通信操作，这使得在单机的进程间进行简单数据传递不太方便，这时使用下面将介绍的WM\_COPYDATA消息将更合适些。

##### WM\_COPYDATA消息

WM\_COPYDATA是一种非常强大却鲜为人知的消息。当一个应用向另一个应用传送数据时，发送方只需使用调用SendMessage函数，参数是目的窗口的句柄、传递数据的起始地址、WM\_COPYDATA消息。接收方只需像处理其它消息那样处理WM\_COPY DATA消息，这样收发双方就实现了数据共享。

WM\_COPYDATA是一种非常简单的方法，它在底层实际上是通过文件映射来实现的。它的缺点是灵活性不高，并且它只能用于Windows平台的单机环境下。

二**. linux**下进程间通信的几种主要手段简介：

管道（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；

信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于

BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）；

报文（Message）队列（消息队列）：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。

套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：

Linux和System V的变种都支持套接字。

进程间通信就是在不同进程之间传播或交换信息，那么不同进程之间存在着什么双方都可以访问的介质呢？进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享内存区。但是，系统空间却是“公共场所”，所以内核显然可以提供这样的条件。除此以外，那就是双方都可以访问的外设了。在这个意义上，两个进程当然也可以通过磁盘上的普

通文件交换信息，或者通过“注册表”或其它数据库中的某些表项和记录交换信息。广义上这也是进程间通信的手段，但是一般都不把这算作“进程间通信”。因为那些通信手段的效率太低 了，而人们对进程间通信的要求是要有一定的实时性。

进程间通信主要包括管道, 系统IPC(包括消息队列,信号量,共享存储), SOCKET.

管道包括三种:1)普通管道PIPE, 通常有种限制,一是半双工,只能单向传输;二是只能在父子进程间使用. 2)流管道s\_pipe: 去除了第一种限制,可以双向传输. 3)命名管道:name\_pipe, 去除了第二种限制,可以在许多并不相关的进程之间进行通讯.

系统IPC的三种方式类同,都是使用了内核里的标识符来识别.

###### 死锁产生条件：

1.互斥

2.请求和保持（部分分配，占有申请）

3.非抢占

4.环路等待

###### 死锁的四个必要条件

互斥条件（Mutual exclusion）：资源不能被共享，只能由一个进程使用。

请求与保持条件（Hold and wait）：已经得到资源的进程可以再次申请新的资源。

非剥夺条件（No pre-emption）：已经分配的资源不能从相应的进程中被强制地剥夺。

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

###### 处理死锁的策略

1.忽略该问题。例如鸵鸟算法，该算法可以应用在极少发生死锁的的情况下。为什么叫鸵鸟算法呢，因为传说中鸵鸟看到危险就把头埋在地底下，可能鸵鸟觉得看不到危险也就没危险了吧。跟掩耳盗铃有点像。

2.检测死锁并且恢复。

3.仔细地对资源进行动态分配，以避免死锁。

4.通过破除死锁四个必要条件之一，来防止死锁产生。

一. 可以规定该资源获得顺序，如果请求的资源顺序与当前请求资源顺序不符合，则把当前持有的资源释放，再重新得到资源。

二. 尝试定时的锁，使用每个显式Lock类中定时tryLock特性，来替代使用内部锁机制

三. 在程序中尽量使用开放调用。依赖于开放调用的程序，相比于那些在持有锁的时候还调用外部方法的程序，更容易进行死锁自由度的分析。重新构建synchronized使开放调用更加安全。所谓开放调用是指调用的方法本身没有加锁，但是要以对方法操作的内容进行加锁。 一句话，synchronized尽量不要锁定整个函数，而是把向锁的内容给锁了。

四. 可抢占锁，也就是能够响应中断。

写一个死锁程序，首先是有两个共享资源（静态成员变量为类所共有），然后写一个flag，判断run()函数执行哪一段代码，代码段1首先锁住Object1然后sleep3秒，然后请求锁住 Object2。代码段2首先锁住Object2然后sleep3秒，然后请求锁住Object1。具体代码如下：

public class TestLock implements Runnable{

static Object o1 = new Object(),o2 = new Object(); private int flag = 1;

public TestLock(int flag){ this.flag = flag;

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

// TODO Auto-generated method stub TestLock t1 = new TestLock(1); TestLock t2 = new TestLock(0); new Thread(t1).start();

new Thread(t2).start();

}

@Override

public void run() {

// TODO Auto-generated method stub if(flag == 1){

synchronized(o1){

System.out.println("Thread1 is running!"); try {

Thread.sleep(3000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block e.printStackTrace();

}

synchronized(o2){ System.out.println("1");

}

}

}

if(flag == 0){

synchronized(o2){ System.out.println("Thread2 is running"); try {

Thread.sleep(3000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block e.printStackTrace();

}

synchronized(o1){

System.out.println("2");

}

}

}

}

}

页和分段系统有许多相似之处，但在概念上两者完全不同，主要表现在：

1、页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率；或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。

段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了能更好的满足用户的需要。

2、页的大小固定且由系统确定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而一个系统只能有一种大小的页面。

段的长度却不固定，决定于用户所编写的程序，通常由编辑程序在对源程序进行编辑时，根据信息的性质来划分。

3、分页的作业地址空间是维一的，即单一的线性空间，程序员只须利用一个记忆符，即可表示一地址。

分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

1. 先入先出法（**FIFO**）

最简单的页面置换算法是先入先出（FIFO）法。这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存。理由是：最早调入内存的页，其不再被使用的可能性比刚调入内存的可能性大。建立一个FIFO队列，收容所有在内存中的页。被置换页面总是在队列头上进行。当一个页面被放入内存时，就把它插在队尾 上。

这种算法只是在按线性顺序访问地址空间时才是理想的，否则效率不高。因为那些常被访问的页，往往在主存中也停留得最久，结果它们因变“老”而不得不被置换出去。

FIFO的另一个缺点是，它有一种异常现象，即在增加存储块的情况下，反而使缺页中断率增加了。当然，导致这种异常现象的页面走向实际上是很少见的。

1. 最优置换算法（**OPT**）

最优置换（Optimal Replacement）是在理论上提出的一种算法。其实质是：当调入新的一页而必须预先置换某个老页时，所选择的老页应是将来不再被使用，或者是在最远的将来才被访问。采用这种页面置换算法，保证有最少的缺页率。

但是最优页面置换算法的实现是困难的，因为它需要人们预先就知道一个进程整个运行过程中页面走向的全部情况。不过，这个算法可用来衡量（如通过模拟实验分析或理论分析）其他算法的优劣。

1. 最久未使用算法（**LRU**）

FIFO算法和OPT算法之间的主要差别是，FIFO算法利用页面进入内存后的时间长短作为置换依据，而OPT算法的依据是将来使用页面的时间。如果以最近的过去作为不久将来的近似， 那么就可以把过去最长一段时间里不曾被使用的页面置换掉。它的实质是，当需要置换一页时，选择在最近一段时间里最久没有使用过的页面予以置换。这种算法就称为最久未使用算法（Least Recently Used，LRU）。

LRU算法是与每个页面最后使用的时间有关的。当必须置换一个页面时，LRU算法选择过去一段时间里最久未被使用的页面。

LRU算法是经常采用的页面置换算法，并被认为是相当好的，但是存在如何实现它的问题。

LRU算法需要实际硬件的支持。其问题是怎么确定最后使用时间的顺序，对此有两种可行的办法：

（1）计数器。最简单的情况是使每个页表项对应一个使用时间字段，并给CPU增加一个逻辑时钟或计数器。每次存储访问，该时钟都加1。每当访问一个页面时，时钟寄存器的内容就被复制到相应页表项的使用时间字段中。这样我们就可以始终保留着每个页面最后访问的“时 间”。在置换页面时，选择该时间值最小的页面。这样做，不仅要查页表，而且当页表改变时

（因CPU调度）要维护这个页表中的时间，还要考虑到时钟值溢出的问题。

（2）栈。用一个栈保留页号。每当访问一个页面时，就把它从栈中取出放在栈顶上。这样一来，栈顶总是放有目前使用最多的页，而栈底放着目前最少使用的页。由于要从栈的中间移走一项，所以要用具有头尾指针的双向链连起来。在最坏的情况下，移走一页并把它放在栈顶上需要改动6个指针。每次修改都要有开销，但需要置换哪个页面却可直接得到，用不着查找，因为尾指针指向栈底，其中有被置换页。

因实现LRU算法必须有大量硬件支持，还需要一定的软件开销。所以实际实现的都是一种简单有效的LRU近似算法。

一种LRU近似算法是最近未使用算法（Not Recently Used，NUR）。它在存储分块表的每一表项中增加一个引用位，操作系统定期地将它们置为0。当某一页被访问时，由硬件将该位置

1。过一段时间后，通过检查这些位可以确定哪些页使用过，哪些页自上次置0后还未使用过。就可把该位是0的页淘汰出去，因为在最近一段时间里它未被访问过。

1. 第二次机会算法（**SCR**）

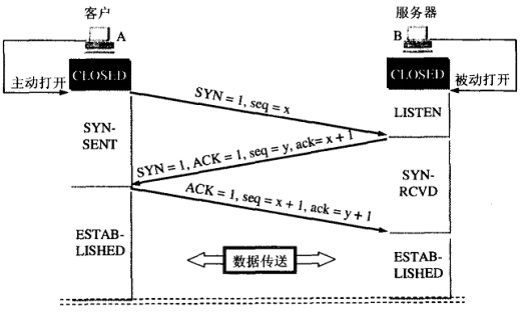
第二次机会算法的基本思想是与FIFO相同的，但是有所改进，避免把经常使用的页面置换出去。当选择置换页面时，检查它的访问位。如果是0，就淘汰这页；如果访问位是1，就给它第二次机会，并选择下一个FIFO页面。当一个页面得到第二次机会时，它的访问位就清为0， 它的到达时间就置为当前时间。如果该页在此期间被访问过，则访问位置1。这样给了第二次机会的页面将不被淘汰，直至所有其他页面被淘汰过（或者也给了第二次机会）。因此，如果一个页面经常使用，它的访问位总保持为1，它就从来不会被淘汰出去。

第二次机会算法可视为一个环形队列。用一个指针指示哪一页是下面要淘汰的。当需要一个存储块时，指针就前进，直至找到访问位是0的页。随着指针的前进，把访问位就清为0。在最坏的情况下，所有的访问位都是1，指针要通过整个队列一周，每个页都给第二次机会。这时就退化成FIFO算法了。

页面置换算法还有很多变种，如考虑到被置换页是否修改过、按FIFO算法选中的页正在使用等情况，都需要硬件、软件协同实现。

## 计算机网络

**TCP**三次握手



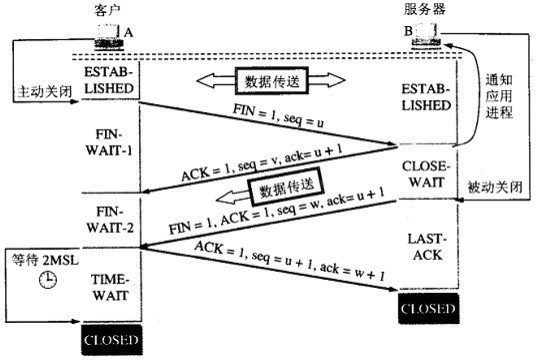
第一次握手：建立连接时发送SYN会选择一个初始序号（ISN），每个连接的ISN都是不同的。客户端发送数据包(SYN=1，seq=x)到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到数据包包，必须确认客户的SYN（ACK=1，ack=x+1），同时自己也发送一个SYN包（SYN=1，seq=y），即SYN+ACK包，此时服务器进入SYN\_RECV状 态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK=1(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数 据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。

**TCP**四次挥手



第一次挥手：主动断开连接一方向被动断开连接发送FIN数据包，FIN=1,seq=x，告诉被动断开连接一方“我要跟你断开连接了，我不会再给你发送数据了”，这是主动断开连接方式可以接受数据的，如果一直没有收到被动连接方的确认包，则可以重新发送这个包。此时，主动断开连接方处于FIN\_WAIT\_1状态

第二次挥手：被动连接方接收到FIN包以后，发送确认包ACK=1,ack=x+1（FIN和SYN一样占用一个序列号），这个动作是告诉主动断开连接方我知道你要断开了，但是我还有数据没有发送完，等发送完了所有的数据就进行第三次挥手，此时被动断开连接方处于CLOSE\_WAIT 状态，主动断开连接方处于FIN\_WAIT\_2状态

第三次挥手：被动断开连接方发送FIN=1，seq=y+1包，用来停止向主动断开连接方发送数据，也就是告诉主动断开连接方，我的数据也发完了，我也不给你发数据了，此时被动断开连接方处于LAST\_ACK状态，主动断开连接方处于TIME\_WAIT 状态

第四次挥手：等过了一定时间（2MSL（报文段最大生存时间）:为了保证最后ACK报文能够到达B，防止已失效连接请求报文段出现在此连接中）过后，主动断开连接方发送确认包ACK=1, ack=y+2，至此，四次挥手已经完成。

**TCP**的三次握手过程？为什么会采用三次握手，若采用二次握手可以吗？

答：建立连接的过程是利用客户服务器模式，假设主机A为客户端，主机B为服务器端。

（1）TCP的三次握手过程：主机A向B发送连接请求；主机B对收到的主机A的报文段进行确认；主机A再次对主机B的确认进行确认。

（2）采用三次握手是为了防止失效的连接请求报文段突然又传送到主机B，因而产生错误。失效的连接请求报文段是指：主机A发出的连接请求没有收到主机B的确认，于是经过一段时间后，主机A又重新向主机B发送连接请求，且建立成功，顺序完成数据传输。考虑这样一种

特殊情况，主机A第一次发送的连接请求并没有丢失，而是因为网络节点导致延迟达到主机

B，主机B以为是主机A又发起的新连接，于是主机B同意连接，并向主机A发回确认，但是此时主机A根本不会理会，主机B就一直在等待主机A发送数据，导致主机B的资源浪费。

（3）采用两次握手不行，原因就是上面说的实效的连接请求的特殊情况。

**TIME\_WAIT**状态存在的理由：

1）可靠地实现TCP全双工连接的终止

在进行关闭连接四路握手协议时，最后的ACK是由主动关闭端发出的，如果这个最终的ACK 丢失，服务器将重发最终的FIN，因此客户端必须维护状态信息允 许它重发最终的ACK。如果不维持这个状态信息，那么客户端将响应RST分节，服务器将此分节解释成一个错误（在

java中会抛出connection reset的SocketException)。因而，要实现TCP全双工连接的正常终止，必须处理终止序列四个分节中任何一个分节的丢失情况，主动关闭 的客户端必须维持状态信息进入TIME\_WAIT状态。

2）允许老的重复分节在网络中消逝

TCP分节可能由于路由器异常而“迷途”，在迷途期间，TCP发送端可能因确认超时而重发这个分节，迷途的分节在路由器修复后也会被送到最终目的地，这个 原来的迷途分节就称为lost

duplicate。在关闭一个TCP连接后，马上又重新建立起一个相同的IP地址和端口之间的TCP 连接，后一个连接被称为前一个连接的化身 （incarnation)，那么有可能出现这种情况，前一个连接的迷途重复分组在前一个连接终止后出现，从而被误解成从属于新的化身。为了避免这个情 况，TCP不允许处于TIME\_WAIT状态的连接启动一个新的化身，因为TIME\_WAIT状态持续2MSL，就可以保证当成功建立一个TCP连接的时 候，来自连接先前化身的重复分组已经在网络中消逝。

Http协议是超文本传输协议。

HTTP协议定义了浏览器怎样向万维网服务器请求万维网文档，以及服务器怎样把文档传送给浏览器。从层次的角度看，HTTP是面向事务的应用层协议。

每一个万维网网点都有一个服务器进程，不断的监听TCP的端口80，一遍发现是否有浏览器向它发送链接请求。一旦监听到链接建立请求并建立了TCP链接以后，浏览器就向万维网服务器发出浏览某个页面的请求，服务器接着就返回所请求的页面作为响应。最后TCP链接就被释放了。在浏览器和服务器之间的请求和响应的交互，必须按照规定的格式和遵循一定的规则。这些格式和规则就是超文本传输协议HTTP。

用户输入网址以后发生的几个时间。

（1）浏览器向DNS请求解析[www.qq.com的IP地址](http://www.qq.com的IP地址/)

（2）域名系统DNS解析出服务器的IP地址为X.X.X.X

（3）浏览器与服务器建立TCP连接

（4）浏览发出请求命令

（5）服务器给出响应，并把相应的文件发送给浏览器

（6）释放TCP连接

（7）浏览器显示该页面的所有文本

HTTP使用了面向连接的TCP作为传输层协议，保证了数据的可靠传输。

HTTP协议本身是无连接的，这个意思是虽然HTTP使用了TCP连接，但通信的双方在交换

HTTP报文之前不需要先建立HTTP连接。

HTTP的协议是无状态的。也就是说同一个客户第二次访问同一个服务器上的页面时，服务器的响应是与第一次被访问时候是相同的。这里就需要引用Cookie来实现记住之前的访问状 态。

HTTP协议是基于请求与响应模式的。就是客户端请求，服务器端响应，就是这样的一个模式。

**HTTP**协议的主要特点可概括如下：

1.支持客户\/服务器模式。

2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、

HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得

HTTP服务器的程序规模小，因而通信速度很快。

3.灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

5.无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

一、**HTTP**协议详解之**URL**篇

http（超文本传输协议）是一个基于请求与响应模式的、无状态的、应用层的协议，常基于

TCP的连接方式，HTTP1.1版本中给出一种持续连接的机制，绝大多数的Web开发，都是构建在HTTP协议之上的Web应用。

HTTP URL (URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息)的格式如下：

[http:\/\/host[":"port][abs\_path]([http://host](http://host/)[%22%22port][abs\_path/)\] http表示要通过HTTP协议来定位网络资源；host表示合法的Internet主机域名或者IP地址；

port指定一个端口号，为空则使用缺省端口80；abs\_path指定请求资源的URI；如果URL中没有给出abs\_path，那么当它作为请求URI时，必须以“\/”的形式给出，通常这个工作浏览器自动帮我们完成。

eg:

1、输入：[www.guet.edu.cn](http://www.guet.edu.cn/)

浏览器自动转换成：[http:\/\/www.guet.edu.cn\/](http://www.guet.edu.cn/)

2、http:192.168.0.116:8080\/index.jsp

二、**HTTP**协议详解之请求篇

http请求由三部分组成，分别是：请求行、消息报头、请求正文

1、请求行以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，格式如下：Method Request-URI HTTP-Version CRLF

其中 Method表示请求方法；Request-URI是一个统一资源标识符；HTTP-Version表示请求的

HTTP协议版本；CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或

LF字符）。

请求方法（所有方法全为大写）有多种，各个方法的解释如下：

GET 请求获取Request-URI所标识的资源

POST 在Request-URI所标识的资源后附加新的数据

HEAD 请求获取由Request-URI所标识的资源的响应消息报头

PUT 请求服务器存储一个资源，并用Request-URI作为其标识

DELETE 请求服务器删除Request-URI所标识的资源

TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断

CONNECT 保留将来使用

OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求应用举例：

GET方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用GET方法向服务器获取资源，eg:GET \/form.html HTTP\/1.1 (CRLF)

POST方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。eg：POST \/reg.jsp HTTP\/ (CRLF)

Accept:image\/gif,image\/x-xbit,... (CRLF)

...

HOST:[www.guet.edu.cn](http://www.guet.edu.cn/) (CRLF) Content-Length:22 (CRLF) Connection:Keep-Alive (CRLF) Cache-Control:no-cache (CRLF)

(CRLF) \/\/该CRLF表示消息报头已经结束，在此之前为消息报头

user=jeffrey&pwd=1234 \/\/此行以下为提交的数据

HEAD方法与GET方法几乎是一样的，对于HEAD请求的回应部分来说，它的HTTP头部中包含的信息与通过GET请求所得到的信息是相同的。利用这个方法，不必传输整个资源内容， 就可以得到Request-URI所标识的资源的信息。该方法常用于测试超链接的有效性，是否可以访问，以及最近是否更新。

2、请求报头后述

3、请求正文(略)

三、**HTTP**协议详解之响应篇

在接收和解释请求消息后，服务器返回一个HTTP响应消息。

HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文

1、状态行格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。

状态代码有三位数字组成，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常见状态代码、状态描述、说明：

200 OK \/\/客户端请求成功

1. Bad Request \/\/客户端请求有语法错误，不能被服务器所理解
2. Unauthorized \/\/请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用

403 Forbidden \/\/服务器收到请求，但是拒绝提供服务

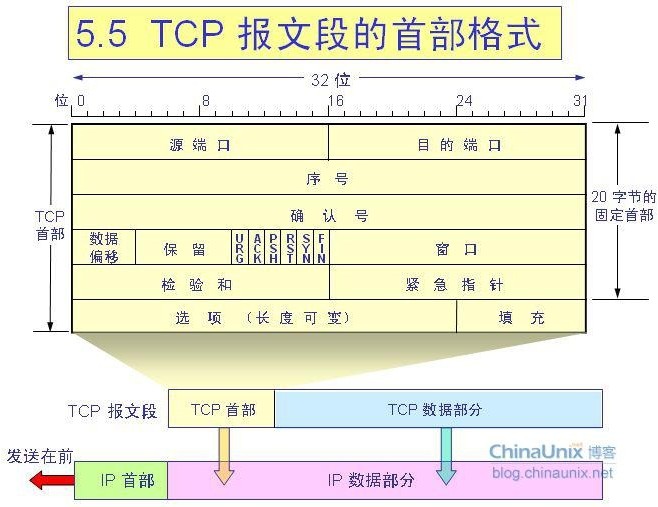
404 Not Found \/\/请求资源不存在，eg：输入了错误的URL 500 Internal Server Error \/\/服务器发生不可预期的错误

503 Server Unavailable \/\/服务器当前不能处理客户端的请求，一段时间后可能恢复正常eg：HTTP\/1.1 200 OK （CRLF）

2、响应报头后述

3、响应正文就是服务器返回的资源的内容

**TCP**数据报首部



TCP提供了一种可靠的面向连接的字节流运输服务。TCP将用户数据打包构成报文段；它发送数据后启动一个定时器；另一端对收到的数据进行确认，对失序的数据重新排序，丢弃重复数据；TCP提供端到端的流量控制，并计算验证强制性的端到端校验和。

源端口（**Source Port**），目标端口（**Destination Port**） 各**2**字节

源端口号，标识主机上发起传送的应用程序；目的端口标识主机上传送要到达的应用程序。源端和目的端的端口号，用于寻找发端和收端应用进程。这两个值加上I P首部中的源端I P地址和目的端I P地址唯一确定一个TCP连接。一个I P地址和一个端口号有时也称为一个插口

（socket），插口对（socket pair）(包含客户I P地址、客户端口号、服务器 I P地址和服务器端口号的四元组 )可唯一确定互联网络中每个TCP连接的双方。IP+TCP端口唯一确定一个TCP连接。

TCP协议通过使用"端口"来标识源端和目标端的应用进程。 端口号可以使用0到65535之间的任何数字。在收到服务请求时，操作系统动态地为客户端的应用程序分配端口号。在服务器端，每种服务在"众所周知的端 口"（Well-Know Port）为用户提供服务。

封包序号（**Sequence Number**） **4**字节

由于 TCP 封包必须要带入 IP 封包当中，所以如果 TCP 数据太大时(大于 IP 封包的容许程度)， 就得要进行分段。这个 Sequence Number 就是记录每个封包的序号，可以让收受端重新将 TCP 的数据组合起来。

序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。确认号（**Acknowledge Number**） **4**字节

为了确认主机端确实有收到我们 client 端所送出的封包数据，我们 client 端当然希望能够收到主机方面的响应，那就是这个 Acknowledge Number 的用途了。 当 client 端收到这个确认码时，就能够确定之前传递的封包已经被正确的收下了。这个号是期望收到对方的下一个报文段的数据的第一个字节的序号。

数据偏移（**Data Offset**）**4**比特

和IP数据包头部一样，也有个Options字段，长度是不固定的，而为了要确认整个TCP封包大小，就需要这个标志来说明整个封包区段的起始位置。

这个字段指出TCP报文段的数据起始处距离 TCP报文段的起始处有多远。“数据偏移”的单位不是字节而是32bit字（4字节为计算单位）。

没有任何选项字段的TCP头部长度为20字节；最多可以有60字节的TCP头部。保留字段（**Reserved**） 占**6**比特

保留为今后使用，但目前应置为0。状态控制码（**Code**，**Control Flag**）

标志位字段（U、A、P、R、S、F）：占**6**比特。各 比特的含义如下:

URG：紧急比特（urgent）,当URG＝1时，表明紧急指针字段有效,代表该封包为紧急封包。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)， 且上图中的 Urgent Pointer 字段也会被启用。

ACK：确认比特（Acknowledge）。只有当ACK＝1时确认号字段才有效,代表这个封包为确认封包。当ACK＝0时，确认号无效。

PSH：（Push function）若为1时，代表要求对方立即传送缓冲区内的其他对应封包，而无需等缓冲满了才送。

RST：复位比特(Reset) ,当RST＝1时，表明TCP连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。

SYN：同步比特(Synchronous)，SYN置为1，就表示这是一个连接请求或连接接受报文,

通常带有 SYN 标志的封包表示『主动』要连接到对方的意思。。

FIN：终止比特(Final)，用来释放一个连接。当FIN＝1时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

滑动窗口（**Window**） 占**2**字节

窗口字段用来控制对方发送的数据量，可以告知对方目前本身有的缓冲器容量(Receive

Buffer) 还可以接收封包。当 Window=0 时，代表缓冲器已经额满，所以应该要暂停传输数据。单位为字节。TCP连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限,这个值是本机期望一次接收的字节数。

**TCP**校验和**(Checksum)** 占**2**字节

当数据要由发送端送出前，会进行一个检验的动作，并将该动作的检验值标注在这个字段 上； 而接收者收到这个封包之后，会再次的对封包进行验证，并且比对原发送的 Checksum 值是否相符，如果相符就接受，若不符就会假设该封包已经损毁，进而要求对方重新发送此封包！在计算检验和时，要在TCP报文段的前面加上12字节的伪首部。

紧急指针**(Urgent Pointer)** 占**2**字节

这个字段是在 Code 字段内的 URG = 1 时才会产生作用。可以告知紧急数据所在的位置(紧急指针指出在本报文段中的紧急数据的最后一个字节的序号)。

选项（**Options**） 长度可变。

TCP首部可以有多达40字节的可选信息，用于把附加信息传递给终点，或用来对齐其它选项。

目前此字段仅应用于表示接收端可以接收的最大数据区段容量，若此字段不使用， 表示可以使用任意数据区段的大小。 这个字段较少使用。

TCP首部的主要选项：

最大报文段长度MSS(Maximum Segment Size)是TCP报文段中的数据字段的最大长度。MSS

告诉对方TCP：“我的缓存所能接收的报文段的数据字段的最大长度是MSS个字节。”

填充字段**(Padding)**

如同 IP 封包需要有固定的 32bits 表头一样， Options 由于字段为非固定， 所以也需要

Padding 字段来加以补齐才行。同样也是 32 bits 的整数。

这是为了使整个首部长度是4字节的整数倍。

**ARP**是地址解析协议，简单语言解释一下工作原理。

注意：**ARP**是解决同一个局域网上的主机和路由器**ip**和**MAC**地址的解析。

1：首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址之间的对应关系。

2：当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址， 如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP数据包，该数据包包括的内容有：源主机 **IP**地址，源主机**MAC**地址，目的主机的**IP** 地址。

3：当本网络的所有主机收到该ARP数据包时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的IP和MAC地址写入到ARP列表中，如果已经存在，则覆盖，然后将自己的MAC地址写入ARP响应包中，告诉源主机自己是它想要找的MAC地址。

4：源主机收到ARP响应包后。将目的主机的IP和MAC地址写入ARP列表，并利用此信息发送数据。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。

广播发送ARP请求，单播发送ARP响应。描述：**RARP**

RARP是逆地址解析协议，作用是完成硬件地址到IP地址的映射，主要用于无盘工作站，因为给无盘工作站配置的IP地址不能保存。工作流程：在网络中配置一台RARP服务器，里面保存着IP地址和MAC地址的映射关系，当无盘工作站启动后，就封装一个RARP数据包，里面有其MAC地址，然后广播到网络上去，当服务器收到请求包后，就查找对应的MAC地址的IP地址装入响应报文中发回给请求者。因为需要广播请求报文，因此RARP只能用于具有广播能力的网络。

#### 可靠重传

对于收到的请求给出确认响应

超时重传

#### 流量控制

让发送方的发送数据的速率不要太快，让接受方来得及接收

通过滑动窗口协议来实现流量控制

TCP的窗口单位是字节，不是报文段，发送方的窗口大小不能大于接收方的窗口大小

#### 滑动窗口

TCP滑动窗口主要有两个作用，一是提高TCP的可靠性，二是提供TCP的流量控制特性。同时，滑动窗口机制还体现了TCP面向字节流的设计思路。

TCP的Window是一个16bit位字段，它代表窗口的字节容量，也就是TCP的标准窗口最大就是

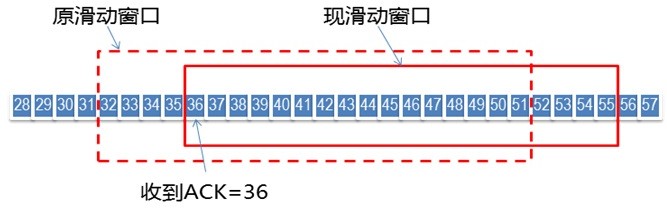
65535个字节。另外TCP的选项字段中还包括一个TCP窗口扩大因子，option-kind为3， option-length为3个字节，option-data取值范围是0-14。窗口扩大因子用来扩大TCP窗口，可把原来16bit的窗口，扩大为31bit。

###### 滑动窗口基本原理

1）对于TCP回话的发送方，任何时候在其发送缓存内的数据都可以分为4类，“已发送并得到对端ACK的”，“已发送未得到对端ACK的”，"未发送但对端允许发送的"，“未发送且对端不允许发送的”。注意：**”**已经发送但未收到对端**ACK“**和**”**未发送单对端允许发送的**“**这两部分数据称之为发送窗口（中间两部分）。



当收到接收方新的ACK对于发送窗口中后续字节的确认时，窗口滑动，滑动原理如下图：



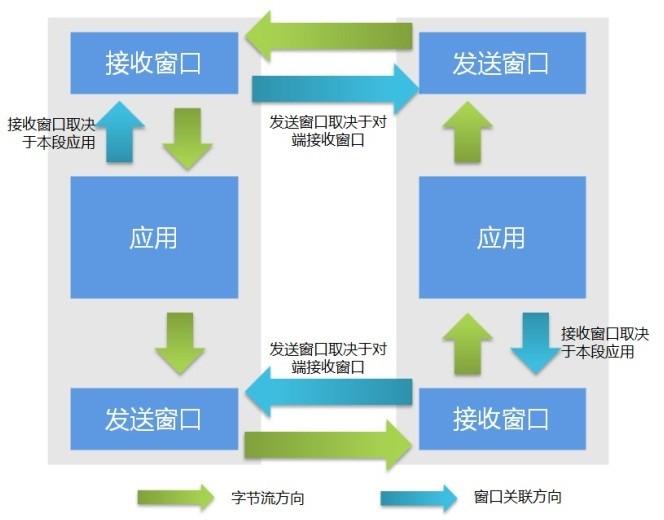
当接收到ACK=36时，发生窗口滑动。

2）对于TCP接收方，在某一时刻接收缓存内存中包括三种数据。“已接收”，“未接受但准备接收”，“未接受且不准备接收”（由于接收端的ACK是有协议栈发送，默认无应用延迟，不存在已接收但未回复）。其中”未接收准备接收“叫做接收窗口。

###### 发送窗口与接收窗口关系

TCP是双工协议，会话的双发可以同时接收、发送数据。TCP会话的双方都各自维护一个“发送窗口”和一个“接收窗口”。其中各自的“接收窗口”的大小取决于应用、系统、硬件的限制

（TCP传输速率不能大于应用的数据处理速率）。各自的“发送窗口”则取决于对端通告的“接收窗口”，要求相同。



###### 滑动窗口实现面向流的可靠性

最基本的传输可靠性来源于“确认重传”基础上的。

**TCP**的滑动窗口的可靠性也是建立在**“**确认重传**”**基础上的。

发送窗口只有接收对端对于本段发送窗口内字节的ACK确认，才会移动发送窗口的左边界。

接收窗口只有再前面所有的段都确认的情况下才会移动左边界。当在前面还有字节未接受但收到后面字节的情况下，窗口不会移动，并不对后续字节确认。一次确保对端会重传这些数据。

###### 滑动窗口的流控特性

TCP的滑动窗口是动态的，我们可以想象成小学常见的一个数学题，一个水池，体积V，每小时进水量V1，出水量V2。当水满了就不允许再注入了，如果有一个液压系统控制水池大小， 那么就可以控制谁的注入速率和量。这样的水池就类似于**TCP**的窗口。应用根据自身的处理能力变化，通过本段**TCP**接收窗口大小控制来对对端的发送窗口流量控制。

应用程序在需要（如内存不足）时，通过API通知TCP协议栈缩小TCP的接收窗口。然后TCP 协议栈在下一个段发送时包含新的窗口大小通知给对端，对端按照通知的窗口来改变发送窗口，一次达到减缓发送速率的目的。

#### 拥塞控制

所谓拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。

慢开始 慢开始的“慢”并不是指增长速率的慢，而是指在TCP开始发送报文段时先设置拥塞窗口为1。

拥塞避免 使拥塞窗口按线性规律。增长。

快重传 发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待为其设置的重传计时器到期。

快恢复(与快重传配合使用） 当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把慢开始门限减半。这是为了预防网络发生拥塞。请注意，接下来不执行慢开始算法。

由于发送方现在认为网络很可能没有发生拥塞(如果网路发生了严重的拥塞，就不会一连有好几个报文段连续达到接收方，就不会导致接收方连续发送重复确认)，因此与慢开始不同之处是现在不执行慢开始算法(即拥塞窗口现在不设置为1)，而是把它设置为慢开始门限减半后的数值，然后开始执行拥塞避免算法(“加法增大”)，使拥塞窗口缓慢地线性增

拥塞控制是防止传输数据的联络层网络出拥塞时数据大量丢失的情况。拥塞避免主要包含以下2个内容：

（1）慢开始，拥塞避免

（2）快重传，快恢复慢开始和拥塞避免

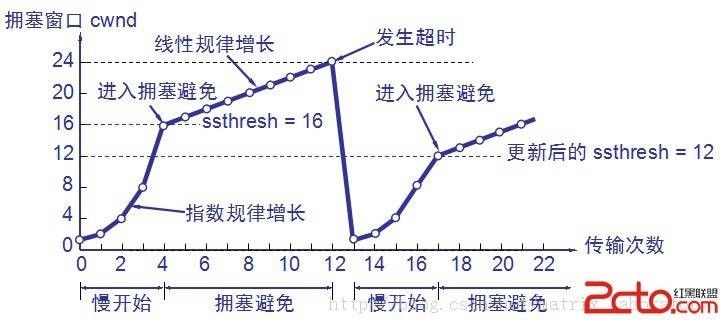
* + 1. 慢开始原理

（1）在主机刚刚开始发送报文段时可先将拥塞窗口 cwnd 设置为一个最大报文段 MSS 的数值。

（2）在每收到一个对新的报文段的确认后，将拥塞窗口增加至多一个 MSS 的数值。

（3）用这样的方法逐步增大发送端的拥塞窗口 cwnd，可以使分组注入到网络的速率更加合理。

* + 1. 实例讲解



注：图中窗口的单位都是报文段

（1）当 TCP 连接进行初始化时： 发送窗口：swnd = 1

慢开始阈值：ssthresh = 16

（2）发送端收到 ACK1 （确认 M0，期望收到 M1）后，将 cwnd 从 1 增大到 2，于是发送端可以接着发送 M1 和 M2 两个报文段（指数增长）

（3）接收端发回 ACK2 和 ACK3。发送端每收到一个对新报文段的确认 ACK，就把发送端的拥塞窗口加 1。现在发送端的 cwnd 从 2 增大到 4，并可发送 M4 ~ M6共 4个报文段。（指数增长）

（4）当swnd >= ssthresh，swnd执行拥塞避免算法，swnd窗口按线性规律增长。 （加法增大）

（5）当发送 超时，此时swnd = 24 ：

ssthresh = swnd\/2 = 12；（乘法减小） swnd = 1

（6）重复地2步。

* 1. 快重传和快恢复
     1. 快重传

发送端只要一连收到三个重复的 ACK 即可断定有分组丢失了，就应立即重传丢失的报文段而不必继续等待为该报文段设置的重传计时器的超时

* + 1. 快恢复

1. 当发送端收到连续三个重复的 ACK 时，就重新设置慢开始门限 ssthresh。
2. 与慢开始不同之处是 swnd 不是设置为 1，而是设置为 ssthresh + 3 \* MSS。
3. 若收到的重复的 ACK 为 n 个（n > 3），则将 cwnd 设置为 ssthresh + n \* MSS。
4. 若发送窗口值还容许发送报文段，就按拥塞避免算法继续发送报文段。
5. 若收到了确认新的报文段的 ACK，就将 swnd 缩小到 ssthresh。

**Nagle**算法

Nagle算法要求一个TCP连接上最多只能有一个未被确认的未完成的小分组，在该分组的确认到达之前不能发送其他的小分组。所谓小分组就是指比如一个Rlogin连接上客户一半每次发送一个字节的服务器，也就是一些41字节长的分组：20字节的IP首部，20字节的TCP首部，1字节的数据。

端口及对应的服务？

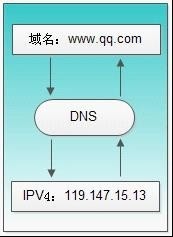
|  |  |  |  |
| --- | --- | --- | --- |
| 服务 | 端口号 | 服务 | 端口号 |
| FTP | 21 | SSH | 22 |
| telnet | 23 | SMTP | 25 |
| Domain(域名服务器) | 53 | HTTP | 80 |
| POP3 | 110 | NTP（网络时间协议） | 123 |
| [**MySQL**](http://lib.csdn.net/base/14)数据库服务 | 3306 | Shell或 cmd | 514 |
| POP-2 | 109 | SQL Server | 1433 |

DNS(Domain Name System)域名系统

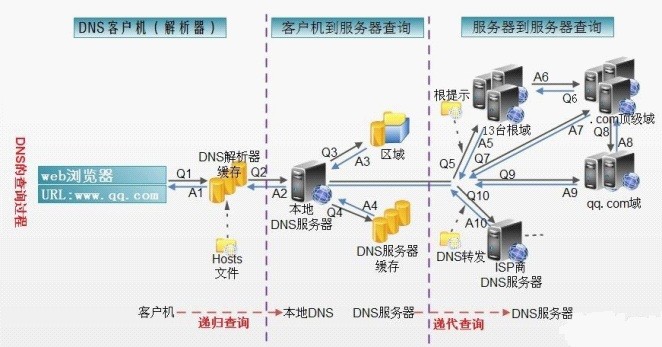
**1**、**DNS**的工作原理

DNS提供的服务是用来将域名转换为IP地址的工作。TCP\/IP协议详解中给的定义是：它是一种用于TCP\/IP应用程序的分布式数据库，它提供主机名字和IP地址之间的转换及有关电子邮件的选路信息。属于应用层协议。

它的基本工作原理如下图所示 ：



**2**、**DNS**的工作过程



在浏览器中输入[www.qq.com域名](http://www.qq.com域名/)，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。

如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

如果hosts与本地DNS解析器缓存都没有相应的网址映射关系，首先会找TCP\/IP参数中设置的首选DNS服务器，在此我们叫它本地DNS服务器，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析， 此解析具有权威性。

如果要查询的域名，不由本地DNS服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。

如果本地DNS服务器本地区域文件与缓存解析都失效，则根据本地DNS服务器的设置(是否设置转发器)进行查询，如果未用转发模式，本地DNS就把请求发至13台根DNS，根

DNS服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管

理.com域的下一级DNS服务器地址(qq.com)给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找qq.com域服务器，重复上面的动作，进行查询，直至找到

[www.qq.com主机](http://www.qq.com主机/)。

如果用的是转发模式，此DNS服务器就会把请求转发至上一级DNS服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地DNS服务器用的是转发，还是根提示，最后都是把结果返回给本地

DNS服务器，由此DNS服务器再返回给客户机。

**ICMP**协议

ICMP是（Internet Control Message Protocol）Internet控制[报文](http://baike.baidu.com/view/175122.htm)协议。它是[TCP\/IP协议族](http://baike.baidu.com/view/2221037.htm)的一个子协议，用于在IP[主机](http://baike.baidu.com/view/23880.htm)、[路由](http://baike.baidu.com/view/18655.htm)器之间传递控制消息。控制消息是指[网络通](http://baike.baidu.com/view/8079702.htm)不通、[主机](http://baike.baidu.com/view/23880.htm)是否可达、[路由](http://baike.baidu.com/view/18655.htm)是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

**Ping**

ping命令主要是用于检测网络的连通性。

Ping命令发送一个ICMP请求报文给目的IP，然后目的IP回复一个ICMP报文。

原理：网络上的机器都有唯一确定的IP地址，我们给目标IP地址发送一个数据包，对方就要返回一个同样大小的数据包，根据返回的数据包我们可以确定目标主机的存在，可以初步判断目标主机的操作系统等。

因为ping命令是使用ICMP协议，所以没有端口号，但是有两个域：类型和代码。

### TraceRoute

TraceRoute协议是可以让我们看到IP数据报从一台主机到另一台主机所经过的路由。

###### 原理：

TraceRoute程序使用ICMP报文和IP首部的TTL字段（声明周期）。TTL字段是由发送端初始设置一个8bit字段。每个处理数据报的路由器都需要把TTL值进行减1或者减去数据包在路由器上停留的描述。TraceRoute程序发送一份TTL字段为1 的IP数据报给目的主机。处理这份数据包的第一个路由器将TTL值减1，丢弃该数据报，并发回一份超时ICMP报文。这样就得到第一个路由器的地址。然后发送一个TTL字段为2的IP数据包给目的主机，这样我们就得到了第二个路由器的地址。继续这个过程直到该数据报到达目的主机。但是目的主机哪怕接收到了

TTL值为1的IP数据包，也不会丢弃该数据报产生超时ICMP报文，这是因为数据包已经到达其最终目的地。我们采用的方法是TraceRoute程序发送一份UDP数据包给目的主机，它选择一个不可能的值作为UDP端口号（大于30000），使目的主机的任何一个应用程序都不可能使用该端口。从而产生一个“端口不可达”错误的ICMP报文。TraceRoute程序只要判断ICMP报文是超时还是不可达就可以判断出来什么时候结束了。

（trcertroute建立一个UDP数据包，不断修改TTL值并发送出去，如果收到"超时错"，表示刚刚到达的是路由器，而如果收到的是"端口不可达"错误，表示刚刚到达的就是目的主机。这样路由跟踪完成，程序结束。）

java的jpcap实现了TraceRoute功能。

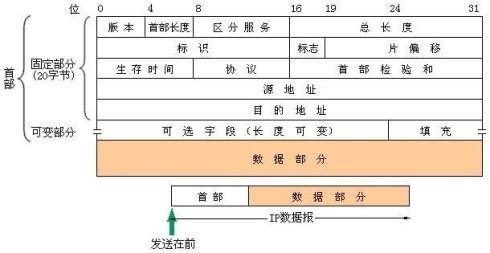
UDP封装

IP首部20字节+UDP首部8个字节+UDP数据

MTU最大传输单元，当IP数据包长度超过网络的MTU时，就要对IP数据包哦进行分片。

UDP首部包括16位源端口号，16位目的端口号，16位UDP长度，16位UDP校验和。一共8个字节。

IP数据报包括首部和数据两个部分。首部20个字节。有可选字段。



IP数据包格式 TCP\/IP协议定义了一个在因特网上传输的包，称为IP数据报(IP Datagram)。这是一个与硬件无关的虚拟包, 由首部和数据两部分组成，其格式如图所示。首部的前一部分是固定长度，共20字节，是所有IP数据报必须具有的。在首部的固定部分的后面是一些可选字段，其长度是可变的。首部中的源地址和目的地址都是IP协议地址 IP数据报格式(转) - 沉的 - 沉的的博客

1、IP数据报首部的固定部分中的各字段

(1)版本 占4位，指IP协议的版本。通信双方使用的IP协议版本必须一致。目前广泛使用的IP 协议版本号为4（即IPv4）。

(2)首部长度 占4位，可表示的最大十进制数值是15。请注意，这个字段所表示数的单位是

32位字长（1个32位字长是4字节），因此，当IP的首部长度为1111时（即十进制的15），首部长度就达到60字节。当IP分组的首部长度不是4字节的整数倍时，必须利用最后的填充字段加以填充。因此数据部分永远在4字节的整数倍开始，这样在实现IP协议时较为方便。首部长度限制为60 字节的缺点是有时可能不够用。但这样做是希望用户尽量减少开销。最常用的首部长度就是20字节（即首部长度为0101），这时不使用任何选项。

（＃我们一般看到的版本和首部长度两个字段是十六进制45，就是版本号

version=4,headlength=5,也就是首部长度是60个字节）

(3)区分服务 占8位，用来获得更好的服务。这个字段在旧标准中叫做服务类型，但实际上一直没有被使用过。1998年IETF把这个字段改名为区分服务DS(Differentiated Services)。只有在使用区分服务时，这个字段才起作用。

(4)总长度 总长度指首部和数据之和的长度，单位为字节。总长度字段为16位，因此数据报的最大长度为216-1=65535字节。

IP数据报格式(转) - 沉的 - 沉的的博客＃可以看这个以太网frame总长为336字节，而IP数据包Total length＝322，336－322＝14正好是Ethernet包头的长度，所以就可以看出这IP数据包总长度一值就是除去Ethernet头的剩余长度，也就是IP包头加数据的长度。

在IP层下面的每一种数据链路层都有自己的帧格式，其中包括帧格式中的数据字段的最大长度，这称为最大传送单元MTU(Maximum Transfer Unit)。当一个数据报封装成链路层的帧时，此数据报的总长度（即首部加上数据部分）一定不能超过下面的数据链路层的MTU值。

(5)标识(identification) 占16位。IP软件在存储器中维持一个计数器，每产生一个数据报，计数器就加1，并将此值赋给标识字段。但这个“标识”并不是序号，因为IP是无连接服务，数据报不存在按序接收的问题。当数据报由于长度超过网络的MTU而必须分片时，这个标识字段的值就被复制到所有的数据报的标识字段中。相同的标识字段的值使分片后的各数据报片最后能正确地重装成为原来的数据报。

(6)标志(flag) 占3位，但目前只有2位有意义。

* 标志字段中的最低位记为MF(More Fragment)。MF=1即表示后面“还有分片”的数据报。

MF=0表示这已是若干数据报片中的最后一个。

* 标志字段中间的一位记为DF(Don’t Fragment)，意思是“不能分片”。只有当DF=0时才允许分片。

(7)片偏移 占13位。片偏移指出：较长的分组在分片后，某片在原分组中的相对位置。也就是说，相对用户数据字段的起点，该片从何处开始。片偏移以8个字节为偏移单位。这就是说，每个分片的长度一定是8字节（64位）的整数倍。

(8)生存时间 占8位，生存时间字段常用的的英文缩写是TTL(Time To Live)，表明是数据报在网络中的寿命。由发出数据报的源点设置这个字段。其目的是防止无法交付的数据报无限制地在因特网中兜圈子，因而白白消耗网络资源。最初的设计是以秒作为TTL的单位。每经过一个路由器时，就把TTL减去数据报在路由器消耗掉的一段时间。若数据报在路由器消耗的时间小于1秒，就把 TTL值减1。当TTL值为0时，就丢弃这个数据报。

＃TTL通常是32或者64，scapy中默认是64

(9)协议 占8位，协议字段指出此数据报携带的数据是使用何种协议，以便使目的主机的IP层知道应将数据部分上交给哪个处理过程。（在scapy中，下层的这个protocol一般可以从上曾继承而来，自动填充，我们一般可以省略不填此项）

(10)首部检验和 占16位。这个字段只检验数据报的首部，但不包括数据部分。这是因为数据报每经过一个路由器，路由器都要重新计算一下首部检验和（一些字段，如生存时间、标

志、片偏移等都可能发生变化）。不检验数据部分可减少计算的工作量。

(11)源地址 占32位。

(12)目的地址 占32位。

2、IP数据报首部的可变部分

IP首部的可变部分就是一个可选字段。选项字段用来支持排错、测量以及安全等措施，内容很丰富。此字段的长度可变，从1个字节到40个字节不等，取决于所选择的项目。某些选项项目只需要1个字节，它只包括1个字节的选项代码。但还有些选项需要多个字节，这些选项一个个拼接起来，中间不需要有分隔符，最后用全0的填充字段补齐成为4字节的整数倍。

增加首部的可变部分是为了增加IP数据报的功能，但这同时也使得IP数据报的首部长度成为可变的。这就增加了每一个路由器处理数据报的开销。实际上这些选项很少被使用。新的IP版 本IPv6就将IP数据报的首部长度做成固定的。

目前，这些任选项定义如下：

（1）安全和处理限制（用于军事领域）

（2）记录路径（让每个路由器都记下它的IP地址）

（3）时间戳（让每个路由器都记下它的IP地址和时间）

（4）宽松的源站路由（为数据报指定一系列必须经过的IP地址）

（5）严格的源站路由（与宽松的源站路由类似，但是要求只能经过指定的这些地址，不能经过其他的地址）

这些选项很少被使用，并非所有主机和路由器都支持这些选项。

附：scapy中的IP层实现：各个参数对应的非常清楚，我们经常关心的可能是ttl,proto,src,dst 等等

一，如何更好的检测TCP连接是否正常

这方面问题，我上网查了很久，一般来说比较成熟的有两种方法：

1是在应用层制定协议，发心跳包，这也是C#，JAVA等高级语言比较常用的方法。客户端和服务端制定一个通讯协议，每隔一定时间（一般15秒左右），由一方发起，向对方发送协议包；对方收到这个包后，按指定好的通讯协议回一个。若没收到回复，则判断网络出现问 题，服务器可及时的断开连接，客户端也可以及时重连。

2通过TCP协议层发送KeepAlive包。这个方法只需设置好你使用的TCP的KeepAlive项就好， 其他的操作系统会帮你完成。操作系统会按时发送KeepAlive包，一发现网络异常，马上断 开。我就是使用这个方法，也是重点向大家介绍的。

使用第二种方法的好处，是我们在应用层不需自己定协议，通信的两端，只要有一端设好这个值，两边都能及时检测出TCP连接情况。而且这些都是操作系统帮你自动完成的。像我们公司的服务端代码就是早写好的，很难改动。以前也没加入心跳机制，后面要改很麻烦，

boss要求检测连接的工作尽量客户端单独完成....

还有一个好处就是节省网络资源。KeepAlive包，只有很简单的一些TCP信息，无论如何也是比你自己设计的心跳包短小的。然后就是它的发送机制，在TCP空闲XXX秒后才开始发送。自己设计心跳机制的话，很难做到这一点。

## 数据结构与算法

**B-**树 **B+**树

在前面专题中讲的BST、AVL、RBT都是典型的二叉查找树结构，其查找的时间复杂度与树高相关。那么降低树高自然对查找效率是有所帮助的。另外还有一个比较实际的问题：就是大量数据存储中，实现查询这样一个实际背景下，平衡二叉树由于树深度过大而造成磁盘IO读写过于频繁，进而导致效率低下。那么如何减少树的深度（当然不能减少查询数据量），一个基本的想法就是：

* 1. 每个节点存储多个元素 （但元素数量不能无限多，否则查找就退化成了节点内部的线性查找了）。
  2. 摒弃二叉树结构，采用多叉树 （由于节点内元素数量不能无限多，自然子树的数量也就不会无限多了）。

这样我们就提出来了一个新的查找树结构 ——多路查找树。 根据AVL给我们的启发，一颗平衡多路查找树**(B~**树**)**自然可以使得数据的查找效率保证在O(logN)这样的对数级别上。

【**2-4**树】

(2,4)树是一棵典型的平衡多路查找树。性质如下：

1. 大小性质：每个结点最多4个子结点。
2. 深度性质：所有外部结点的深度相同。

(2,4)其实是一棵迷你型的B树，其主要应用并不是为了将大数据量存储在外存上，而是通过减少树高来降低二叉查找树的查找代价。在介绍下面的B~树\/B+树之前，请大家首先了解一下

《[外部存储器—磁盘](http://hxraid.iteye.com/blog/630007) 》。

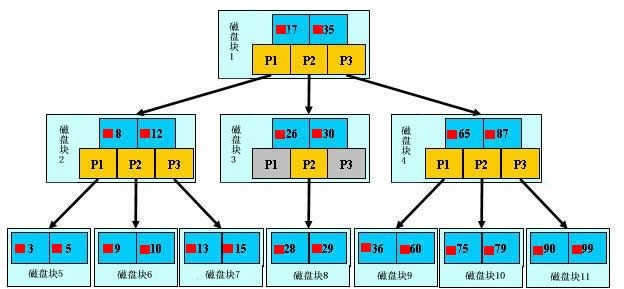
【**B~**树】

**B~**树，又叫平衡多路查找树。一棵**m**阶的**B~**树 **(m**叉树**)**的特性如下：

1. 树中每个结点至多有m个孩子；
2. 除根结点和叶子结点外，其它每个结点至少有[m\/2]个孩子；
3. 若根结点不是叶子结点，则至少有2个孩子；
4. 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息(可以看做是外部接点或查询失败的接点，实际上这些结点不存在，指向这些结点的指针都为null)；
5. 每个非终端结点中包含有n个关键字信息： (n，A0，K1，A1，K2，A2，......，Kn，An)。其中，
   1. Ki (i=1...n)为关键字，且关键字按顺序排序Ki < K(i-1)。
   2. Ai为指向子树根的接点，且指针A(i-1)指向子树种所有结点的关键字均小于Ki，但都大于K(i-1)。
   3. 关键字的个数n必须满足： [m\/2]-1 <= n <= m-1

例如：下面就是一棵3阶B~树

(为了简单，这里用少量数据构造一棵2-4树的形式，其实实际应用中的B树结点中关键字很多的)



**B~**树的建立

由于B~树结点中的关键字个数必须>=[m\/2]-1。因此和平衡二叉树不同，每一次插入一个关键字并不是在树中添加一个结点，而是首先在最低层的某个非终端结点中添加一个关键字，若该结点的关键字个数不超过m-1，则插入完成。否则，要产生结点的"分裂" 。

关于B~树以及后面B+树的插入，删除操作，在《[数据结构算法与应用-搜索树](http://dl.iteye.com/topics/download/59ffea3c-102c-38df-a0dc-6f84d22f590a) 》中有详细讲解。

关于文件目录索引的**B~**树 磁盘 存储结构及其查询过程

既然我们说过B~树相比平衡二叉树的一个巨大的特点就是：海量数据存储时B~树利用磁盘存储的效率会高很多。那么我们现在就来简单的看看操作系统中文件目录的B~树结构是怎样的

(这里只介绍简单的原理，至于想Windows,Linux的文件系统使用的是B+树结构，而且技术远远比这里介绍的复杂)。

首先，我们构造一个B树结点的信息：

**Java**代码

* + 1. **class** BTNode<E **extends** Comparable<E>>{ 2.

1. \/\/ 结点中文件个数
2. **int** filenum;
3. \/\/子树的根结点指针向量
4. BTNode[] ptr=**new** BTNode(filenum+1);
5. \/\/文件名向量
6. E[] filename=**new** E(filenum);
7. \/\/ 指向磁盘中文件存储地址向量
8. FileHardAddress[] recptr=**new** FileHardAddress(filenum); 11.

12. }

上面的图中比如根结点，其中17表示一个磁盘文件的文件名；小红方块表示这个17文件的内容在硬盘中的存储位置；p1表示指向17左子树的指针。

我们现在把整棵树构造在磁盘中，假如每个盘块可以正好存放一个B~树的结点（正好存放2个文件名）。那么一个BTNode结点就代表一个盘块，而子树指针就是存放另外一个盘块 （详细见《[外部存储器—磁盘](http://hxraid.iteye.com/blog/630007) 》）的地址。

现在我们模拟查找文件29的过程：

* 1. 根据根结点指针找到文件目录的根磁盘块1，将其中的信息导入内存。【磁盘IO操作1 次】
  2. 此时内存中有两个文件名17，35和三个存储其他磁盘页面地址的数据。根据算法我们发现17<29<35，因此我们找到指针p2。
  3. 根据p2指针，我们定位到磁盘块3，并将其中的信息导入内存。【磁盘IO操作2次】
  4. 此时内存中有两个文件名26，30和三个存储其他磁盘页面地址的数据。根据算法我们发现26<29<30，因此我们找到指针p2。
  5. 根据p2指针，我们定位到磁盘块8，并将其中的信息导入内存。【磁盘IO操作3次】
  6. 此时内存中有两个文件名28，29。根据算法我们查找到文件29，并定位了该文件内存的磁盘地址。

分析一下上面的过程，我们发现需要3次磁盘IO操作和3次内存查找操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于3次磁盘IO操作时影响整个B~树查找效率的决定因素。

当然，如果我们使用平衡二叉树的磁盘存储结构来进行查找，磁盘IO操作最少4次，最多5 次。而且文件越多，B~树比平衡二叉树所用的磁盘IO操作次数将越少，效率也越高。

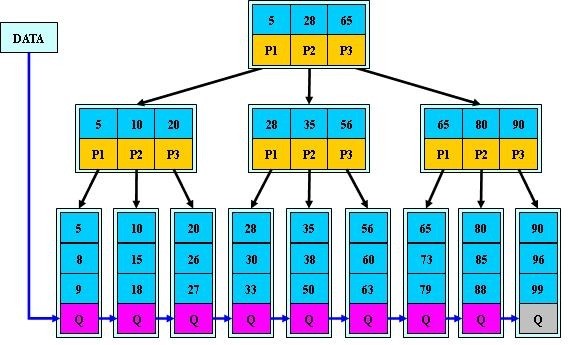
【**B+**树】

**B+**树：是应文件系统所需而产生的一种**B~**树的变形树。 一棵m阶的B+树和m阶的B-树的差异在于：

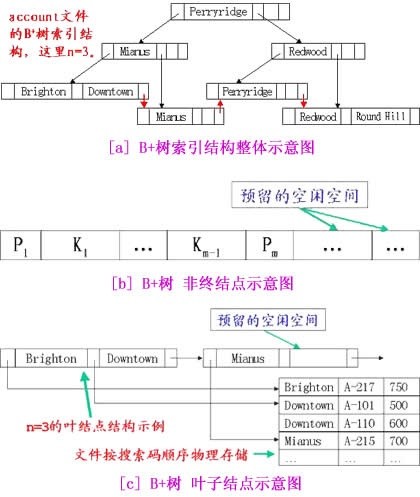
1) 有n棵子树的结点中含有n个关键字； (B~树是n棵子树有n+1个关键字)2) 所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。 (B~树的叶子节点并没有包括全部需要查找的信息)

1. 所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。 (B~树的非终节点也包含需要查找的有效信息)

例如：下面就是一棵3阶B+树。我们可以和B~树做一个明显的对比。



下面我们用另外一个图来看一下B+树叶子节点和非终节点的特点：



上面这个图有一个很重要的性质：B+树的叶子结点包含了所有待查询关键字，而非终节点只是作为叶子结点中最大(最小)关键字的索引。因此B+树的非终结点没有文件内容所在物理存储的地址，而B~树所有结点均有文件内容所在的磁盘物理地址（B~树结构图中结点内部的小红方块）。 这个特点是B+树的一个重要优势所在。这一点我们在下面谈及。

关于**FOXPRO**索引文件的**B+**树磁盘存储结构及其查询

B+树在数据库，文件系统的索引结构中是十分常用的。关于B+树的磁盘存储可以参见上面B~ 树的情况，其一个结点用一个磁盘块存储。在这里我们对FOXPRO索引文件做一个简单的介绍，让大家对B+树的磁盘存储有一个大致的了解。

FOXPRO的索引文件（后缀IDX）由索引文件头和索引文件体组成。

文件头占一个块,相对于索引文件的物理零块号,它描述索引文件的组织信息,包括索引树的根结点位置,索引关键字表达式及索引关键字长度.其有用字节的含义如下表：

|  |  |
| --- | --- |
| 字节 | 内容 |
| 0-3 | 标识根结点所在块号 |
| 4-7 | 保留 |
| 8-11 | 索引文件总块数 |
| 12 | 索引文件的关键字长度 |
| 16 | 索引关键字表达式(以ASCII码存放） |

索引文件体从索引文件的相对物理块号为1的块开始,文件体的每块也就是索引树的一个结点。其中重要的是索引项。索引项=关键字+指针域(4字节)。这就是我们上面常说的B+树结点中的两个重要的信息：待查询关键字和指向另一个结点的指针。文件体每块含义如下表：

|  |  |
| --- | --- |
| 字节 | 内容 |
| 0 | 块属性标记.00H:非叶结点和根结点.01H:根结点,02H:叶结点.03H:既是根结点又是叶结点. |
| 1 | 00H |
| 2,3 | 块内索引项总数 (多个索引项) |
| 4-7 | 同一层前继结点块号或4个FFFF值 |
| 8-  11 | 同一层后继结点块号或4个FFFF值 |
| 12- | 非递减顺序存放的索引项内容 |

B+树的查找与B~树类似。但通常B+树有两个头指针，一个指向根结点。另一个指向关键字最小的叶子节点。此外，所有叶子结点也按照大小顺序链接。因此，B+树有两种查找算法：一种从根结点出发，一种从叶子结点出发的顺序查找。

**B+**树的优势所在

为什么说B+树比B~树更适合实际应用中操作系统的文件索引和数据库索引？

1、B+树的磁盘读写代价更低

我们都知道磁盘时可以块存储的，也就是同一个磁道上同一盘块中的所有数据都可以一次全部读取(详见《 [外部存储器—磁盘](http://hxraid.iteye.com/blog/630007) 》 )。而B+树的内部结点并没有指向关键字具体信息的指针

(比如文件内容的具体地址 ， 比如说不包含B~树结点中的FileHardAddress[filenum]部分） 。因此其内部结点相对B~树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。这样，一次性读入内存中的需要查找的关键字也就越

多。相对来说IO读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B~树(一个结点最多8个关键字)的内部结点需要2个盘快。而B+树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B~树就比B+数多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

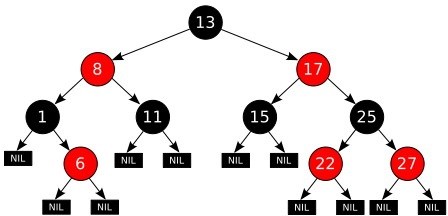
2、B+树的查询效率更加稳定。

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

#### 红黑树的性质与定义

红黑树(red-black tree) 是一棵满足下述性质的二叉查找树：

* 1. 每一个结点要么是红色，要么是黑色。
  2. 根结点是黑色的。
  3. 所有叶子结点都是黑色的（实际上都是Null指针，下图用NIL表示）。叶子结点不包含任何关键字信息，所有查询关键字都在非终结点上。
  4. 每个红色结点的两个子节点必须是黑色的。换句话说：从每个叶子到根的所有路径上不能有两个连续的红色结点
  5. 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点



黑深度 ——从某个结点x出发(不包括结点x本身)到叶结点(包括叶子结点)的路径上的黑结点个数,称为该结点x的黑深度,记为bd(x),根结点的黑深度就是该红黑树的黑深度。叶子结点的黑深度为0。比如：上图bd(13)=2，bd(8)=2，bd(1)=1

内部结点 —— 红黑树的非终结点外部节点 —— 红黑树的叶子结点

#### 红黑树相关定理

1. 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。

根据上面的性质5我们知道上图的红黑树每条路径上都是3个黑结点。因此最短路径长度为

2(没有红结点的路径)。再根据性质4(两个红结点不能相连)和性质1，2(叶子和根必须是黑结点)。那么我们可以得出：一条具有3个黑结点的路径上最多只能有2个红结点(红黑间隔存

在)。也就是说黑深度为2（根结点也是黑色）的红黑树最长路径为4，最短路径为2。从这一点我们可以看出红黑树是 大致平衡的。 (当然比平衡二叉树要差一些，AVL的平衡因子最多为

1)

1. 红黑树的树高**(h)**不大于两倍的红黑树的黑深度**(bd)**，即**h<=2bd**

根据定理1，我们不难说明这一点。bd是红黑树的最短路径长度。而可能的最长路径长度(树高的最大值)就是红黑相间的路径，等于2bd。因此h<=2bd。

1. 一棵拥有**n**个内部结点**(**不包括叶子结点**)**的红黑树的树高**h<=2log(n+1)**

下面我们首先证明一颗有n个内部结点的红黑树满足n>=2^bd-1。这可以用数学归纳法证明， 施归纳于树高h。当h=0时，这相当于是一个叶结点，黑高度bd为0，而内部结点数量n为0， 此时0>=2^0-1成立。假设树高h<=t时，n>=2^bd-1成立，我们记一颗树高 为t+1的红黑树的根结点的左子树的内部结点数量为nl，右子树的内部结点数量为nr，记这两颗子树的黑高度为

bd'（注意这两颗子树的黑高度必然一 样），显然这两颗子树的树高<=t，于是有nl>=2^bd'-1 以及nr>=2^bd'-1，将这两个不等式相加有nl+nr>=2^(bd'+1)-2，将该不等式左右加1，得到n>=2^(bd'+1)-1，很显然bd'+1>=bd，于是前面的不等式可以 变为n>=2^bd-1，这样就证明了一颗有n个内部结点的红黑树满足n>=2^bd-1。

在根据定理2，h<=2bd。即n>=2^(h\/2)-1，那么h<=2log(n+1)

从这里我们能够看出，红黑树的查找长度最多不超过2log(n+1)，因此其查找时间复杂度也是

O(log N)级别的。

#### 红黑树的操作

因为每一个红黑树也是一个特化的二叉查找树，因此红黑树上的查找操作与普通二叉查找树上的查找操作相同。然而，在红黑树上进行插入操作和删除操作会导致不 再符合红黑树的性质。恢复红黑树的属性需要少量(O(log n))的颜色变更(实际是非常快速的)和不超过三次树旋转(对于插入操作是两次)。 虽然插入和删除很复杂，但操作时间仍可以保持为 O(log n) 次 。

#### 红黑树的优势

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。此外,任何不平衡都会在3 次旋转之内解决。这一点是AVL所不具备的。

而且实际应用中，很多语言都实现了红黑树的数据结构。比如 TreeMap, TreeSet(Java )、

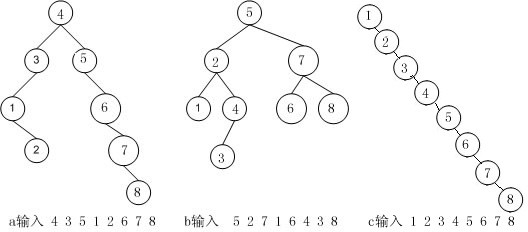
STL(C++)等。

当所有的静态查找结构添加和删除一个数据的时候，整个结构都需要重建。这对于常常需要在查找过程中动态改变数据而言，是灾难性的。因此人们就必须去寻找高效的动态查找结 构，我们在这讨论一个非常常用的动态查找树——二叉查找树 。

二叉查找树的特点

下面的图就是两棵二叉查找树，我们可以总结一下他的特点：

1. 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值
2. 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值
3. 它的左、右子树也分别为二叉查找树



我们中序遍历这两棵树发现一个有序的数据序列： 【1 2 3 4 5 6 7 8 】二叉查找树的操作

插入操作：

现在我们要查找一个数9，如果不存在则，添加进a图。我们看看二叉查找树动态添加的过程：

1. 数9和根节点4比较(9>4)，则9放在节点4的右子树中。
2. 接着，9和节点5比较(9>5)，则9放在节点5的右子树中。
3. 依次类推：直到9和节点8比较(9>8)，则9放在节点8的右子树中，成为节点8的右孩子。

这个过程我们能够发现，动态添加任何一个数据，都会加在原树结构的叶子节点上，而不会重新建树。 由此可见，动态查找结构确实在这方面有巨大的优势。

删除操作：

如果二叉查找树中需要删除的结点左、右子树都存在，则删除的时候需要改变一些子树结构，但所需要付出的代价很小。

具体的插入，删除算法请参加《数据结构算法与应用——搜索树》P5-8。[该章节已经上传到

《[查找结构专题(6)：动态查找树比较](http://hxraid.iteye.com/blog/612108) 》中]。

二叉查找树的效率分析

那么我们再来看看二叉查找树的效率问题

很显然，在a,b两图的二叉查找树结构中查找一个数据，并不需要遍历全部的节点元素，查找效率确实提高了。但是有一个很严重的问题：我们在a图中查找8需要比较5次数据，而在B图中只需要比较3次。更为严重的是：如果按有序序列[1 2 3 4 5 6 7 8]建立一颗二叉查找树，整棵树就退化成了一个线性结构（如c输入图：单支树），此时查找8需要比较8次数据，和顺序查找没有什么不同。

总结一下：最坏情况下，构成的二叉排序树蜕变为单支树，树的深度为n，其查找时间复杂度与顺序查找一样O(N)。最好的情况是二叉排序树的形态和折半查找的判定树相同，其平均查找长度和log2(N)成正比 （O(log2(n))）。

这说明：同样一组数据集合，不同的添加顺序会导致查找树的结构完全不一样，直接影响了查找效率。

那么如何解决这个问题呢? 我们会在下面的专题中：[《平衡二叉树》](http://hxraid.iteye.com/blog/609949) 中来解决。

在上一个专题中，我们在谈论二叉查找树的效率的时候。不同结构的二叉查找树，查找效率有很大的不同（单支树结构的查找效率退化成了顺序查找）。如何解决这个问题呢？关键在于如何最大限度的减小树的深度。正是基于这个想法，平衡二叉树出现了。

平衡二叉树的定义 （**AVL——** 发明者为**Adel'son-Vel'skii** 和 **Landis**）

平衡二叉查找树，又称 AVL树。 它除了具备二叉查找树的基本特征之外，还具有一个非常重要的特点：它 的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值

（平衡因子 ） 不超过1。 也就是说AVL树每个节点的平衡因子只可能是-1、0和1（左子树高度减去右子树高度）。

那么如何是二叉查找树在添加数据的同时保持平衡呢？基本思想就是：当在二叉排序树中插入一个节点时，首先检查是否因插入而破坏了平衡，若 破坏，则找出其中的最小不平衡二叉树，在保持二叉排序树特性的情况下，调整最小不平衡子树中节点之间的关系，以达 到新的平衡。所谓最小不平衡子树 指离插入节点最近且以平衡因子的绝对值大于1的节点作为根的子树。

平衡二叉树的操作

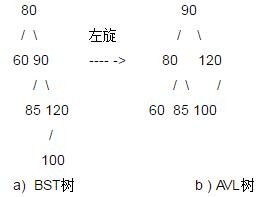
* 1. 查找操作

平衡二叉树的查找基本与二叉查找树相同。

* 1. 插入操作

在平衡二叉树中插入结点与二叉查找树最大的不同在于要随时保证插入后整棵二叉树是平衡的。那么调整不平衡树的基本方法就是： 旋转 。 下面我们归纳一下平衡旋转的4中情况

1. 绕某元素左旋转



分析一下：在插入数据100之前，a图的B ST树只有80节点的平衡因子是-1(左高-右高)，但整棵树还是平衡的。加入100之后，80节点的平衡因子就成为了-2，此时平衡被破坏。需要左旋转成b 图。

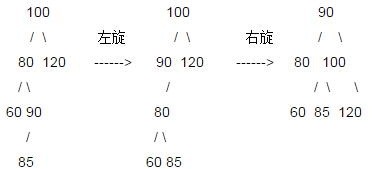
当树中节点X的右孩子的右孩子上插入新元素，且平衡因子从-1变成-2后，就需要绕节点X进行左旋转。

1. 绕某元素右旋转



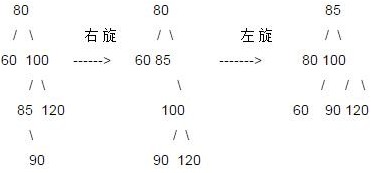
当树中节点X的左孩子的左孩子上插入新元素，且平衡因子从1变成2后，就需要绕节点X进行右旋转。

1. 绕某元素的左子节点左旋转，接着再绕该元素自己右旋转。 此情况下就是左旋与右旋 的结合，具体操作时可以分 解成这两种操作，只是围绕点不一样而已。



当树中节点X的左孩子的右孩子上插入新元素，且 平衡因子从1变成2后，就需要 先绕X的左子节点Y左旋转，接着再绕X右旋转

1. 绕某元素的右子节点右旋转，接着再绕该元素自己左旋转。 此情况下就是 右旋与左旋 的结合，具体操作时可以分解 成这两种操作，只是围绕点不一样而已 。



当树中节点X的右孩子的左孩子上插入新元素，且 平衡因子从-1变成-2后，就需要 先绕X的右子节点Y右旋转，接着再绕X左旋转

平衡二叉树性能分析

平衡二叉树的性能优势：

很显然，平衡二叉树的优势在于不会出现普通二叉查找树的最差情况。其查找的时间复杂度为O(logN)。

平衡二叉树的缺陷：

1. 很遗憾的是，为了保证高度平衡，动态插入和删除的代价也随之增加。因此，我们在下一专题中讲讲[《红黑树》](http://hxraid.iteye.com/blog/611816) 这种更加高效的查找结构。
2. [所有二叉查找树结构的查找代价都与树高是紧密相关的，能否通过减少树高来进一步降低查找代价呢。我们可以通过多路查找树的结构来做到这一点，在后面专题中我们将通过《多路查找树\/B-树\/B+树 》来介绍。](http://hxraid.iteye.com/blog/611105)
3. 在大数据量查找环境下(比如说系统磁盘里的文件目录，数据库中的记录查询 等)，所有的二叉查找树结构(BST、AVL、RBT)都不合适。如此大规模的数据量（几G数据），全部组织成平衡二叉树放在内存中是不可能做到的。那么把这棵树放在磁盘中吧。问题就来了：假如构造的平衡二叉树深度有1W层。那么从根节点出发到叶子节点很可能就需要1W次的硬盘IO 读写。大家都知道，硬盘的机械部件读写数据的速度远远赶不上纯电子媒体的内存。 查找效率在IO读写过程中将会付出巨大的代价。在大规模数据查询这样一个实际应用背景下，平衡二叉树的效率就很成问题了。对这一问题的解决：我们也会在《[多路查找树\/B-树\/B+树](http://hxraid.iteye.com/blog/611105) 》 将详细分析。

上面提到的红黑树和多路查找树都是属于深度有界查找树（**depth-bounded tree —DBT**）

我们这个专题介绍的动态查找树主要有： 二叉查找树(BST)，平衡二叉查找树(AVL)，红黑树

(RBT)，B~\/B+树(B-tree)。这四种树都具备下面几个优势：

1. 都是动态结构。在删除，插入操作的时候，都不需要彻底重建原始的索引树。最多就是执行一定量的旋转，变色操作来有限的改变树的形态。而这些操作所付出的代价都远远小于重建一棵树。这一优势在《[查找结构专题(1)：静态查找结构概论](http://www.iteye.com/blog/608982) 》中讲到过。
2. 查找的时间复杂度大体维持在O(log(N))数量级上。可能有些结构在最差的情况下效率将会下降很快，比如BST。这个会在下面的对比中详细阐述。

下面我们开始概括性描述这四种树，并相互比较一下优劣。

1. 二叉查找树 **(Binary Search Tree)** 详细见《[查找结构专题(2)：二叉查找树 [BST]](http://www.iteye.com/blog/609312) 》

很显然，二叉查找树的发现完全是因为静态查找结构在动态插入，删除结点所表现出来的无能为力(需要付出极大的代价)。

**BST** 的操作代价分析：

1. 查找代价： 任何一个数据的查找过程都需要从根结点出发，沿某一个路径朝叶子结点前进。因此查找中数据比较次数与树的形态密切相关。

当树中每个结点左右子树高度大致相同时，树高为logN。则平均查找长度与logN成正比，查找的平均时间复杂度在O(logN)数量级上。

当先后插入的关键字有序时，BST退化成单支树结构。此时树高n。平均查找长度为

(n+1)\/2，查找的平均时间复杂度在O(N)数量级上。

1. 插入代价： 新结点插入到树的叶子上，完全不需要改变树中原有结点的组织结构。插入一个结点的代价与查找一个不存在的数据的代价完全相同。
2. 删除代价： 当删除一个结点P，首先需要定位到这个结点P，这个过程需要一个查找的代价。然后稍微改变一下树的形态。如果被删除结点的左、右子树只有一个存在，则改变形态的代价仅为O(1)。如果被删除结点的左、右子树均存在，只需要将当P的左孩子的右孩子的右孩子的...的右叶子结点与P互换，在改变一些左右子树即可。因此删除操作的时间复杂度最大不会超过O(logN)。

**BST**效率总结 **:** 查找最好时间复杂度O(logN)，最坏时间复杂度O(N)。插入删除操作算法简单，时间复杂度与查找差不多

1. [平衡二叉查找树 **( Balanced Binary Search Tree )** 详细见《查找结构专题(3)：平衡二叉查找树 [AVL] 》](http://www.iteye.com/blog/609949)

二叉查找树在最差情况下竟然和顺序查找效率相当，这是无法仍受的。事实也证明，当存储数据足够大的时候，树的结构对某些关键字的查找效率影响很大。当然，造成这种情况的主要原因就是BST不够平衡(左右子树高度差太大)。

既然如此，那么我们就需要通过一定的算法，将不平衡树改变成平衡树。因此，AVL树就诞生了。

**AVL** 的操作代价分析：

1. 查找代价： AVL是严格平衡的BST（平衡因子不超过1）。那么查找过程与BST一样，只是AVL不会出现最差情况的BST(单支树)。因此查找效率最好，最坏情况都是O(logN)数量级的。
2. 插入代价： AVL必须要保证严格平衡(|bf|<=1)，那么每一次插入数据使得AVL中某些结点的平衡因子超过1就必须进行旋转操作。事实上，AVL的每一次插入结点操作最多只需要旋转1 次(单旋转或双旋转)。因此，总体上插入操作的代价仍然在O(logN)级别上(插入结点需要首先查找插入的位置)。
3. 删除代价：AVL删除结点的算法可以参见BST的删除结点，但是删除之后必须检查从删除结点开始到根结点路径上的所有结点的平衡因子。因此删除的代价稍微要大一些。每一次删除操作最多需要O(logN)次旋转。因此，删除操作的时间复杂度为O(logN)+O(logN)=O(2logN)

**AVL** 效率总结 **:** 查找的时间复杂度维持在O(logN)，不会出现最差情况

AVL树在执行每个插入操作时最多需要1次旋转，其时间复杂度在O(logN)左右。

AVL树在执行删除时代价稍大，执行每个删除操作的时间复杂度需要O(2logN)。

1. 红黑树 **(Red-Black Tree )** 详细见《[查找结构专题(4)：红黑树 [RBT]](http://www.iteye.com/blog/611816) 》

二叉平衡树的严格平衡策略以牺牲建立查找结构(插入，删除操作)的代价，换来了稳定的

O(logN) 的查找时间复杂度。但是这样做是否值得呢？

能不能找一种折中策略，即不牺牲太大的建立查找结构的代价，也能保证稳定高效的查找效率呢？ 答案就是：红黑树。

**RBT** 的操作代价分析：

1. 查找代价：由于红黑树的性质(最长路径长度不超过最短路径长度的2倍)，可以说明红黑树虽然不像AVL一样是严格平衡的，但平衡性能还是要比BST要好。其查找代价基本维持在

O(logN)左右，但在最差情况下(最长路径是最短路径的2倍少1)，比AVL要略逊色一点。

1. 插入代价：RBT插入结点时，需要旋转操作和变色操作。但由于只需要保证RBT基本平衡就可以了。因此插入结点最多只需要2次旋转，这一点和AVL的插入操作一样。虽然变色操作需要O(logN)，但是变色操作十分简单，代价很小。
2. 删除代价：RBT的删除操作代价要比AVL要好的多，删除一个结点最多只需要3次旋转操作。

**RBT** 效率总结 **:** 查找 效率最好情况下时间复杂度为O(logN)，但在最坏情况下比AVL要差一些，但也远远好于BST。

插入和删除操作改变树的平衡性的概率要远远小于AVL（RBT不是高度平衡的）。因此需要的旋转操作的可能性要小，而且一旦需要旋转，插入一个结点最多只需要旋转2次，删除最多只需要旋转3次(小于AVL的删除操作所需要的旋转次数)。虽然变色操作的时间复杂度在

O(logN)，但是实际上，这种操作由于简单所需要的代价很小。

1. **B~**树**\/B+**树 **(B-Tree )** 详细见《[查找结构专题(5)：B~树\/B+树](http://www.iteye.com/blog/611105) 》

对于在内存中的查找结构而言，红黑树的效率已经非常好了(实际上很多实际应用还对RBT进行了优化)。但是如果是数据量非常大的查找呢？将这些数据全部放入内存组织成RBT结构显然是不实际的。实际上，像OS中的文件目录存储，数据库中的文件索引结构的存储.... 都不可能在内存中建立查找结构。必须在磁盘中建立好这个结构。那么在这个背景下，RBT还是一种好的选择吗？

在磁盘中组织查找结构，从任何一个结点指向其他结点都有可能读取一次磁盘数据，再将数据写入内存进行比较。大家都知道，频繁的磁盘IO操作，效率是很低下的(机械运动比电子运动要慢不知道多少)。显而易见，所有的二叉树的查找结构在磁盘中都是低效的。因此，B树很好的解决了这一个问题。

**B-Tree**的操作代价分析：

(1) 查找代价： B-Tree作为一个平衡多路查找树(m-叉)。B树的查找分成两种：一种是从一个结点查找另一结点的地址的时候，需要定位磁盘地址(查找地址)，查找代价极高。另一种是将结点中的有序关键字序列放入内存，进行优化查找(可以用折半)，相比查找代价极低。而B树的高度很小，因此在这一背景下，B树比任何二叉结构查找树的效率都要高很多。而且B+树作为B树的变种，其查找效率更高。

(2)插入代价： B-Tree的插入会发生结点的分裂操作。当插入操作引起了s个节点的分裂时， 磁盘访问的次数为h(读取搜索路径上的节点)＋2s(回写两个分裂出的新节点)＋1（回写新的根节点或插入后没有导致分裂的节点）。因此，所需要的磁盘访问次数是h+2s+1，最多可达到

3h+1。因此插入的代价是很大的。

(3)删除代价：B-Tree的删除会发生结点合并操作。最坏情况下磁盘访问次数是3h＝（找到包含被删除元素需要h次读访问）+（获取第2至h层的最相邻兄弟需要h-1次读访问）+（在第3至

h层的合并需要h-2次写访问）+（对修改过的根节点和第2层的两个节点进行3次写访问）

**B-Tree**效率总结： 由于考虑磁盘储存结构，B树的查找、删除、插入的代价都远远要小于任何二叉结构树(读写磁盘次数的降低)。

动态查找树结构的对比：

1. 平衡二叉树和红黑树 **[AVL PK RBT]**

AVL 和RBT 都是二叉查找树的优化。其性能要远远好于二叉查找树。他们之间都有自己的优势，其应用上也有不同。

结构对比： AVL的结构高度平衡，RBT的结构基本平衡。平衡度AVL > RBT.

查找对比： AVL 查找时间复杂度最好，最坏情况都是O(logN)。

RBT 查找时间复杂度最好为O(logN)，最坏情况下比AVL略差。

插入删除对比： 1. AVL的插入和删除结点很容易造成树结构的不平衡，而RBT的平衡度要求较低。因此在大量数据插入的情况下，RBT需要通过旋转变色操作来重新达到平衡的频度要小于AVL。

* 1. 如果需要平衡处理时，RBT比AVL多一种变色操作，而且变色的时间复杂度在O(logN)数量级上。但是由于操作简单，所以在实践中这种变色仍然是非常快速的。
  2. 当插入一个结点都引起了树的不平衡，AVL和RBT都最多需要2次旋转操作。但删除一个结点引起不平衡后，AVL最多需要logN 次旋转操作，而RBT最多只需要3次。因此两者插入一个结点的代价差不多，但删除一个结点的代价RBT要低一些。
  3. AVL和RBT的插入删除代价主要还是消耗在查找待操作的结点上。因此时间复杂度基本上都是与O(logN) 成正比的。

总体评价：大量数据实践证明，RBT的总体统计性能要好于平衡二叉树。

1. B~树和B+树 [ B~Tree PK B+Tree]

B+树是B~树的一种变体，在磁盘查找结构中，B+树更适合文件系统的磁盘存储结构。

结构对比： B~树是平衡多路查找树，所有结点中都包含了待查关键字的有效信息(比如文件磁盘指针)。每个结点若有n个关键字，则有n+1个指向其他结点的指针。

B+树严格意义上说已经不是树，它的叶子结点之间也有指针链接。B+树的非终结点中并不含有关键字的信息，需要查找的关键字的全部信息都包含在叶子结点上。非终结点中只作为叶子结点关键字的索引而存在。

查找对比：1. 在相同数量的待查数据下，B+树查找过程中需要调用的磁盘IO操作要少于普通

B~树。由于B树所在的磁盘存储背景下，因此B+树的查找性能要好于B~树。

* 1. B+树的查找效率更加稳定，因为所有叶子结点都处于同一层中，而且查找所有关键字都必须走完从根结点到叶子结点的全部历程。因此同一颗B+树中，任何关键字的查找比较次数都是一样的。而B树就不一定了，可能查找到某一个非终结点就结束了。

插入删除对比： B+树与B~树在插入删除操作中的效率是差不多的。

总体评价：在应用背景下，特别是文件结构存储中。B+树的应用要更多，其效率也要比B~树好。

字符串查找结构

这次专题所讲的BST、AVL、BRT、B~Tree等可以胜任对任何关键字数据进行查找。但对字符串的查找(字符串匹配)结构，有专门的结构和算法。详见：《[KMP算法](http://www.iteye.com/blog/621660) 》，《[Trie Tree](http://www.iteye.com/blog/618962) 》

#### 最长递增子序列

这是一道最经典的动态规划题，它要求求出一串数字当中最长的递增的子序列，首先找到动态规划的状态量，这里的状态量dp[i]是从后往前遍历，指以第i个数字开头的最长递增子序列，然后状态转移方程式dp[i] = max{1 to n dp[j]} 也就是比array[i]大的数字中最大的dp[j] +1

所以代码如下：

import java.util.Scanner;

/\*

\* The longest SubSequence

\*/

public class DPLIS {

public static int find(int array[]){

int len = array.length; System.out.println(len); if(len==0)

return 0;

int dp[] = new int[len];

//From backward to forward dp[len-1] = 1;

for(int i=len-2;i>=0;i--){ int max = 0;

//find the max dp[j]&&(bigger than array[i]) to plus 1 as the dp[i] for(int j=i+1;j<len;j++){

if(array[j]>array[i]){

if(max<dp[j]){ max = dp[j]; continue;

}

}

}

dp[i] = max+1;

System.out.println(array[i] + " " + dp[i]);

}

//find the max dp int max = dp[0];

for(int i=1;i<dp.length;++i){ if(dp[i]>max)

max = dp[i];

}

return max;

}

public static void main(String[] args){

Scanner scan = new Scanner(System.in); while(scan.hasNext()){

int n = scan.nextInt(); int array[] = new int[n]; for(int i=0;i<n;++i){

array[i] = scan.nextInt();

}

System.out.println(find(array));

}

}

}

1. 快速排序（**QuickSort**）

快速排序是一个就地排序，分而治之，大规模递归的算法。从本质上来说，它是归并排序的就地版本。快速排序可以由下面四步组成。

1. 如果不多于1个数据，直接返回。
2. 一般选择序列最左边的值作为支点数据。
3. 将序列分成2部分，一部分都大于支点数据，另外一部分都小于支点数据。
4. 对两边利用递归排序数列。

快速排序比大部分排序算法都要快。尽管我们可以在某些特殊的情况下写出比快速排序快的算法，但是就通常情况而言，没有比它更快的了。快速排序是递归的，对于内存非常有限的机器来说，它不是一个好的选择。

1. 归并排序（**MergeSort**）

归并排序先分解要排序的序列，从1分成2，2分成4，依次分解，当分解到只有1个一组的时候，就可以排序这些分组，然后依次合并回原来的序列中，这样就可以排序所有数据。合并排序比堆排序稍微快一点，但是需要比堆排序多一倍的内存空间，因为它需要一个额外的数组。

1. 堆排序（**HeapSort**）

堆排序适合于数据量非常大的场合（百万数据）。

堆排序不需要大量的递归或者多维的暂存数组。这对于数据量非常巨大的序列是合适的。比如超过数百万条记录，因为快速排序，归并排序都使用递归来设计算法，在数据量非常大的时候，可能会发生堆栈溢出错误。

堆排序会将所有的数据建成一个堆，最大的数据在堆顶，然后将堆顶数据和序列的最后一个数据交换。接下来再次重建堆，交换数据，依次下去，就可以排序所有的数据。

1. **Shell**排序（**ShellSort**）

Shell排序通过将数据分成不同的组，先对每一组进行排序，然后再对所有的元素进行一次插入排序，以减少数据交换和移动的次数。平均效率是O(nlogn)。其中分组的合理性会对算法产生重要的影响。现在多用D.E.Knuth的分组方法。

Shell排序比冒泡排序快5倍，比插入排序大致快2倍。Shell排序比起QuickSort，MergeSort，

HeapSort慢很多。但是它相对比较简单，它适合于数据量在5000以下并且速度并不是特别重要的场合。它对于数据量较小的数列重复排序是非常好的。

1. 插入排序（**InsertSort**）

插入排序通过把序列中的值插入一个已经排序好的序列中，直到该序列的结束。插入排序是对冒泡排序的改进。它比冒泡排序快2倍。一般不用在数据大于1000的场合下使用插入排序， 或者重复排序超过200数据项的序列。

1. 冒泡排序（**BubbleSort**）

冒泡排序是最慢的排序算法。在实际运用中它是效率最低的算法。它通过一趟又一趟地比较数组中的每一个元素，使较大的数据下沉，较小的数据上升。它是O(n^2)的算法。

1. 交换排序（**ExchangeSort**）和选择排序（**SelectSort**）

这两种排序方法都是交换方法的排序算法，效率都是 O(n2)。在实际应用中处于和冒泡排序基本相同的地位。它们只是排序算法发展的初级阶段，在实际中使用较少。

1. 基数排序（**RadixSort**）

基数排序和通常的排序算法并不走同样的路线。它是一种比较新颖的算法，但是它只能用于整数的排序，如果我们要把同样的办法运用到浮点数上，我们必须了解浮点数的存储格式， 并通过特殊的方式将浮点数映射到整数上，然后再映射回去，这是非常麻烦的事情，因此， 它的使用同样也不多。而且，最重要的是，这样算法也需要较多的存储空间。

1. 总结

下面是一个总的表格，大致总结了我们常见的所有的排序算法的特点。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 排序法 | 平均时间 | 最差情形 | 稳定度 | 额外空间 | 备注 |
| 冒泡 | O(n2) | O(n2) | 稳定 | O(1) | n小时较好 |
| 交换 | O(n2) | O(n2) | 不稳定 | O(1) | n小时较好 |
| 选择 | O(n2) | O(n2) | 不稳定 | O(1) | n小时较好 |
| 插入 | O(n2) | O(n2) | 稳定 | O(1) | 大部分已排序时较好 |
| 基数 | O(logRB) | O(logRB) | 稳定 | O(n) | B是真数(0-9)，R是基数(个十百) |
| Shell | O(nlogn) | O(ns) 1<s<2 | 不稳定 | O(1) | s是所选分组 |
| 快速 | O(nlogn) | O(n2) | 不稳定 | O(nlogn) | n大时较好 |
| 归并 | O(nlogn) | O(nlogn) | 稳定 | O(1) | n大时较好 |
| 堆 | O(nlogn) | O(nlogn) | 不稳定 | O(1) | n大时较好 |

散列就是一个Key-Value对，通过Key可以在线性时间内找到Value。Key通过散列函数找到存储位置，一半用数组和链表来存储。

散列函数：

1.直接定址法：就是线性函数值算散列地址，有多大数据，就得多大的散列

2.数据分析法：明确每一个关键码的每一位数值分布情况

3.除留余数发：设散列表中允许地址数为m，取一个不大于m，但是最接近于或等于m的质数p，并构造散列函数：Hash(key)=key%p， p<=m;

4.平法取中法：先平方，再去中间几位，这样比较均匀

5.基数转换法：换进制表示

6.折叠法：

快速排序

public void quickSort(int[] array, int begin, int end){ if(begin<end){

int current = array[begin]; int i=begin,j=end; while(i<j){

while(array[j]>current&&i<j){ j--;

}

if(i<j){

array[i++] = array[j];

}

while(array[i]<current&&i<j){ i++;

}

if(i<j){

array[j--] = array[i];

}

}

array[i] = current; quickSort(array, begin, i-1); quickSort(array, i+1, end);

}

}

二分查找

public boolean binarySearch(int[] array,int n){ int begin = 0;

int end = array.length-1; int mid = (begin+end)/2; while(begin<end){

if(array[mid] == n){ return true;

}else if(array[mid]>n){ end = mid-1;

}else{

begin = mid+1;

}

mid = (begin+end)/2;

}

return false;

}

深入浅出**UML**

#### 类

类(Class)封装了数据和行为，是面向对象的重要组成部分，它是具有相同属性、操作、关系的对象集合的总称。在系统中，每个类都具有一定的职责，职责指的是类要完成什么样的功能，要承担什么样的义务。一个类可以有多种职责，设计得好的类一般只有一种职责。在定义类的时候，将类的职责分解成为类的属性和操作（即方法）。类的属性即类的数据职责， 类的操作即类的行为职责。设计类是面向对象设计中最重要的组成部分，也是最复杂和最耗时的部分。

在软件系统运行时，类将被实例化成对象(Object)，对象对应于某个具体的事物，是类的实例

(Instance)。

类图(Class Diagram)使用出现在系统中的不同类来描述系统的静态结构，它用来描述不同的类以及它们之间的关系。

在系统分析与设计阶段，类通常可以分为三种，分别是实体类(Entity Class)、控制类(Control

Class)和边界类(Boundary Class)，下面对这三种类加以简要说明：

1. 实体类：实体类对应系统需求中的每个实体，它们通常需要保存在永久存储体中，一般使用数据库表或文件来记录，实体类既包括存储和传递数据的类，还包括操作数据的类。实体类来源于需求说明中的名词，如学生、商品等。
2. 控制类：控制类用于体现应用程序的执行逻辑，提供相应的业务操作，将控制类抽象出来可以降低界面和数据库之间的耦合度。控制类一般是由动宾结构的短语（动词+名词）转化来的名词，如增加商品对应有一个商品增加类，注册对应有一个用户注册类等
3. 边界类：边界类用于对外部用户与系统之间的交互对象进行抽象，主要包括界面类，如对话框、窗口、菜单等。

在面向对象分析和设计的初级阶段，通常首先识别出实体类，绘制初始类图，此时的类图也可称为领域模型，包括实体类及其它们之间的相互关系。

* 1. 类的UML图示

在UML中，类使用包含类名、属性和操作且带有分隔线的长方形来表示，如定义一个

Employee类，它包含属性name、age和email，以及操作modifyInfo()，在UML类图中该类如图1所示：

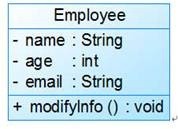


图1 类的UML图示

图1对应的Java代码片段如下：

public class Employee { private String name; private int age; private String email;

public void modifyInfo() {

......

}

}

在UML类图中，类一般由三部分组成：

1. 第一部分是类名：每个类都必须有一个名字，类名是一个字符串。
2. 第二部分是类的属性(Attributes)：属性是指类的性质，即类的成员变量。一个类可以有任意多个属性，也可以没有属性

UML规定属性的表示方式为：

可见性 名称:类型 [ = 缺省值 ]

其中：

“可见性”表示该属性对于类外的元素而言是否可见，包括公有(public)、私有(private)和受保护

(protected)三种，在类图中分别用符号+、-和#表示。 “名称”表示属性名，用一个字符串表示。 “类型”表示属性的数据类型，可以是基本数据类型，也可以是用户自定义类型。 “缺省值”是一个可选项，即属性的初始值。 (3) 第三部分是类的操作(Operations)：操作是类的任意一个实例对象都可以使用的行为，是类的成员方法。

UML规定操作的表示方式为：

可见性 名称(参数列表) [ : 返回类型]

其中：

“可见性”的定义与属性的可见性定义相同。 “名称”即方法名，用一个字符串表示。 “参数列表”表示方法的参数，其语法与属性的定义相似，参数个数是任意的，多个参数之间用逗

号“，”隔开。 “返回类型”是一个可选项，表示方法的返回值类型，依赖于具体的编程语言，可以是基本数据类型，也可以是用户自定义类型，还可以是空类型(void)，如果是构造方法，则无返回类型。 在类图2中，操作method1的可见性为public(+)，带入了一个Object类型的参数

par，返回值为空(void)；操作method2的可见性为protected(#)，无参数，返回值为String类型；操作method3的可见性为private(-)，包含两个参数，其中一个参数为int类型，另一个为

int[]类型，返回值为int类型。

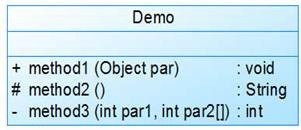


图2 类图操作说明示意图

由于在Java语言中允许出现内部类，因此可能会出现包含四个部分的类图，如图3所示：

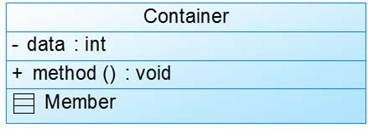


图3 包含内部类的类图

###### 类与类之间的关系（**1**）

在软件系统中，类并不是孤立存在的，类与类之间存在各种关系，对于不同类型的关系，

UML提供了不同的表示方式。

* 1. 关联关系

关联(Association)关系是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系，如汽车和轮胎、师傅和徒弟、班级和学生等等。在UML类图中，用实线连接有关联关系的对象所对应的类，在使用Java、C#和C++等编程语言实现关联关系时，通常将一个类的对象作为另一个类的成员变量。在使用类图表示关联关系时可以在关联线上标注角色名，一般使用一个表示两者之间关系的动词或者名词表示角色名（有时该名词为实例对象名），关系的两端代表两种不同的角色，因此在一个关联关系中可以包含两个角色名，角色名不是必须的，可以根据需要增加，其目的是使类之间的关系更加明确。

如在一个登录界面类LoginForm中包含一个JButton类型的注册按钮loginButton，它们之间可以表示为关联关系，代码实现时可以在LoginForm中定义一个名为loginButton的属性对象，其类型为JButton。如图1所示：



图1 关联关系实例

图1对应的Java代码片段如下：

public class LoginForm { private JButton loginButton; \/\/定义为成员变量 …… } public class JButton { …… } 在UML中，关联关系通常又包含如下几种形式：

1. 双向关联

默认情况下，关联是双向的。例如：顾客(Customer)购买商品(Product)并拥有商品，反之， 卖出的商品总有某个顾客与之相关联。因此，Customer类和Product类之间具有双向关联关系，如图2所示：

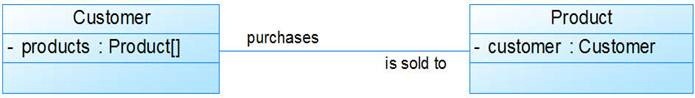


图2 双向关联实例

图2对应的Java代码片段如下：

public class Customer { private Product[] products; …… }

public class Product { private Customer customer; …… } (2) 单向关联

类的关联关系也可以是单向的，单向关联用带箭头的实线表示。例如：顾客(Customer)拥有地址(Address)，则Customer类与Address类具有单向关联关系，如图3所示：



图3 单向关联实例

图3对应的Java代码片段如下：

public class Customer { private Address address; …… }

public class Address { …… } (3) 自关联

在系统中可能会存在一些类的属性对象类型为该类本身，这种特殊的关联关系称为自关联。例如：一个节点类(Node)的成员又是节点Node类型的对象，如图4所示：

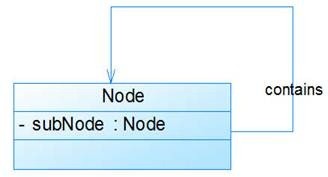


图4 自关联实例

图4对应的Java代码片段如下：

public class Node { private Node subNode; …… } (4) 多重性关联

多重性关联关系又称为重数性(Multiplicity)关联关系，表示两个关联对象在数量上的对应关系。在UML中，对象之间的多重性可以直接在关联直线上用一个数字或一个数字范围表示。

对象之间可以存在多种多重性关联关系，常见的多重性表示方式如表1所示： 表1 多重性表示方式列表

表示方式 多重性说明 1..1 表示另一个类的一个对象只与该类的一个对象有关系 0..\* 表示另一个类的一个对象与该类的零个或多个对象有关系 1..\* 表示另一个类的一个对象与该类的一个或多个对象有关系 0..1 表示另一个类的一个对象没有或只与该类的一个对象有关系 m..n 表示另一个类的一个对象与该类最少m，最多n个对象有关系 (m≤n) 例如：一个界面(Form)可以拥有零个或多个按钮(Button)，但是一个按钮只能属于一个界面，因此，一个Form类的对象可以与零个或多个Button类的对象相关联，但一个Button类的对象只能与一个Form类的对象关

联，如图5所示：



图5 多重性关联实例

图5对应的Java代码片段如下：

public class Form { private Button[] buttons; \/\/定义一个集合对象 …… } public class Button { …… } (5) 聚合关系

聚合(Aggregation)关系表示整体与部分的关系。在聚合关系中，成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在。在UML中，聚合关系用带空心菱形的直线表示。例如：汽车发动机(Engine)是汽车(Car)的组成部分，但是汽车发动机可以独立存在，因此，汽车和发动机是聚合关系，如图6所示：

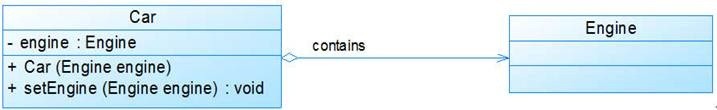


图6 聚合关系实例

在代码实现聚合关系时，成员对象通常作为构造方法、Setter方法或业务方法的参数注入到整体对象中，图6对应的Java代码片段如下：

public class Car { private Engine engine;

\/\/构造注入 public Car(Engine engine) { this.engine = engine; }

设值注入

public void setEngine\(Engine engine\) { t his.engine = engine;

}

…… }

public class Engine { …… } (6) 组合关系

组合(Composition)关系也表示类之间整体和部分的关系，但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在，成员对象与整体对象之间具有同生共死的关系。在UML中，组合关系用带实心菱形的直线表示。例如：人的头

(Head)与嘴巴(Mouth)，嘴巴是头的组成部分之一，而且如果头没了，嘴巴也就没了，因此头和嘴巴是组合关系，如图7所示：



图7 组合关系实例

在代码实现组合关系时，通常在整体类的构造方法中直接实例化成员类，图7对应的Java代码片段如下：

public class Head { private Mouth mouth;

public Head() { mouth = new Mouth(); \/\/实例化成员类 }

…… }

public class Mouth { …… } 类与类之间的关系（2）

* 1. 依赖关系

依赖(Dependency)关系是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系。大多数情况下，依赖关系体现在某个类的方法使用另一个类的对象作为参数。在UML中，依赖关系用带箭头的虚线表示， 由依赖的一方指向被依赖的一方。例如：驾驶员开车，在Driver类的drive()方法中将Car类型的对象car作为一个参数传递，以便在drive()方法中能够调用car的move()方法，且驾驶员的

drive()方法依赖车的move()方法，因此类Driver依赖类Car，如图1所示：

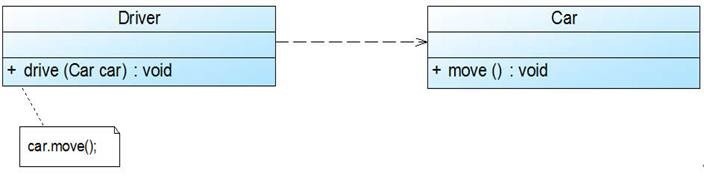


图1 依赖关系实例

在系统实施阶段，依赖关系通常通过三种方式来实现，第一种也是最常用的一种方式是如图1 所示的将一个类的对象作为另一个类中方法的参数，第二种方式是在一个类的方法中将另一个类的对象作为其局部变量，第三种方式是在一个类的方法中调用另一个类的静态方法。图1 对应的Java代码片段如下：

public class Driver { public void drive(Car car) { car.move(); } …… } public class Car { public void move() { ...... } …… }

1. 泛化关系

泛化(Generalization)关系也就是继承关系，用于描述父类与子类之间的关系，父类又称作基类或超类，子类又称作派生类。在UML中，泛化关系用带空心三角形的直线来表示。在代码实现时，我们使用面向对象的继承机制来实现泛化关系，如在Java语言中使用extends关键 字、在C++\/C#中使用冒号“：”来实现。例如：Student类和Teacher类都是Person类的子类，

Student类和Teacher类继承了Person类的属性和方法，Person类的属性包含姓名(name)和年龄(age)，每一个Student和Teacher也都具有这两个属性，另外Student类增加了属性学号(studentNo)，Teacher类增加了属性教师编号(teacherNo)，Person类的方法包括行走move() 和说话say()，Student类和Teacher类继承了这两个方法，而且Student类还新增方法study()，

Teacher类还新增方法teach()。如图2所示：

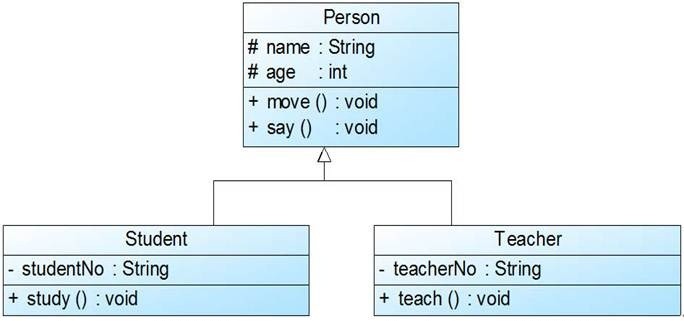


图2 泛化关系实例

图2对应的Java代码片段如下：

\/\/父类 public class Person { protected String name; protected int age; public void move() { …… }

public void say() { …… }

}

\/\/子类 public class Student extends Person { private String studentNo; public void study() { …… } }

\/\/子类 public class Teacher extends Person { private String teacherNo; public void teach() { …… } }

1. 接口与实现关系

在很多面向对象语言中都引入了接口的概念，如Java、C#等，在接口中，通常没有属性，而且所有的操作都是抽象的，只有操作的声明，没有操作的实现。UML中用与类的表示法类似的方式表示接口，如图3所示：



图3 接口的UML图示

接口之间也可以有与类之间关系类似的继承关系和依赖关系，但是接口和类之间还存在一种实现(Realization)关系，在这种关系中，类实现了接口，类中的操作实现了接口中所声明的操作。在UML中，类与接口之间的实现关系用带空心三角形的虚线来表示。例如：定义了一个交通工具接口Vehicle，包含一个抽象操作move()，在类Ship和类Car中都实现了该move()操作，不过具体的实现细节将会不一样，如图4所示：

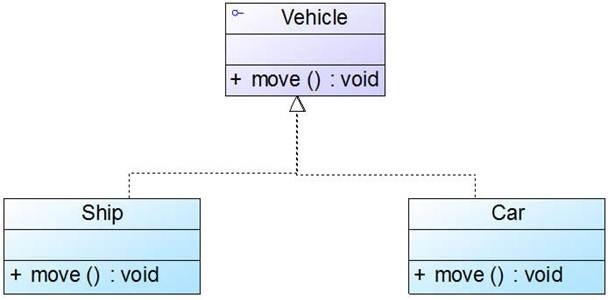


图4 实现关系实例

实现关系在编程实现时，不同的面向对象语言也提供了不同的语法，如在Java语言中使用

implements关键字，而在C++\/C#中使用冒号“：”来实现。图4对应的Java代码片段如下：

public interface Vehicle { public void move(); }

public class Ship implements Vehicle { public void move() { …… } }

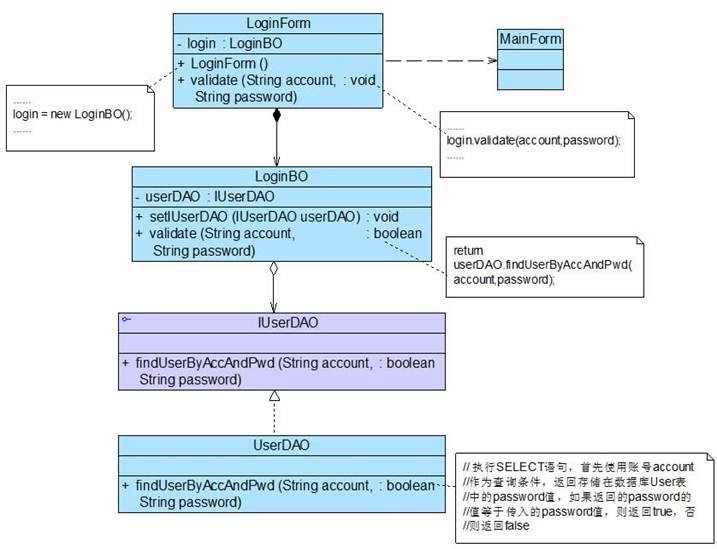
public class Car implements Vehicle { public void move() { …… } } 实例分析1——登录模块某基于C\/S的即时聊天系统登录模块功能描述如下：

用户通过登录界面(LoginForm)输入账号和密码，系统将输入的账号和密码与存储在数据库

(User)表中的用户信息进行比较，验证用户输入是否正确，如果输入正确则进入主界面(MainForm)，否则提示“输入错误”。

根据以上描述绘制初始类图。参考解决方案：

参考类图如下：



考虑到系统扩展性，在本实例中引入了抽象数据访问接口IUserDAO，再将具体数据访问对象注入到业务逻辑对象中，可通过配置文件（如XML文件）等方式来实现，将具体的数据访问类类名存储在配置文件中，如果需要更换新的具体数据访问对象，只需修改配置文件即可， 原有程序代码无须做任何修改。

类说明：

类 名 说 明 LoginForm 登录窗口，省略界面组件和按钮事件处理方法（边界类） LoginBO 登录业务逻辑类，封装实现登录功能的业务逻辑（控制类） IUserDAO 抽象数据访问类接口， 声明对User表的数据操作方法，省略除查询外的其他方法（实体类） UserDAO 具体数据访问类，实现对User表的数据操作方法，省略除查询外的其他方法（实体类） MainForm 主窗口（边界类） 方法说明：

方法名 说 明 LoginForm类的LoginForm()方法 LoginForm构造函数，初始化实例成员

LoginForm类的validate()方法 界面类的验证方法，通过调用业务逻辑类LoginBO的validate() 方法实现对用户输入信息的验证 LoginBO类的validate()方法 业务逻辑类的验证方法，通过调用数据访问类的findUserByAccAndPwd()方法验证用户输入信息的合法性 LoginBO类的

setIUserDAO()方法 Setter方法，在业务逻辑对象中注入数据访问对象（注意：此处针对抽象数据访问类编程 IUserDAO接口的findUserByAccAndPwd()方法 业务方法声明，通过用户账号和密码在数据库中查询用户信息，判断该用户身份的合法性 UserDAO类的

findUserByAccAndPwd()方法 业务方法实现，实现在IUserDAO接口中声明的数据访问方法

**0\/1**背包问题

0\/1背包是在M件物品取出若干件放在空间为W的背包里，每件物品的体积为C1，C2，…，

Cn，与之相对应的价值为W1,W2，…，Wn.求解将那些物品装入背包可使总价值最大。

动态规划（DP）：

1. 子问题定义：F[i][j]表示前i件物品中选取若干件物品放入剩余空间为j的背包中所能得到的最大价值。
2. 根据第i件物品放或不放进行决策

(1-1)

其中F[i-1][j]表示前i-1件物品中选取若干件物品放入剩余空间为j的背包中所能得到的最大价值；

而F[i-1][j-C[i]]+W[i]表示前i-1件物品中选取若干件物品放入剩余空间为j-C[i]的背包中所能取得的最大价值加上第i件物品的价值。

根据第i件物品放或是不放确定遍历到第i件物品时的状态F[i][j]。

设物品件数为N，背包容量为V，第i件物品体积为C[i]，第i件物品价值为W[i]。代码如下：

public class Packet {

public static int DP(int[] val, int[] wei, int m ){ int len = val.length;

//开辟大小为len+1,m+1大小的空间

int[][] V = new int[len+1][m+1];

//前0个物品，容量为多少价值都是0 for(int i=0;i<m;i++){

V[0][i] = 0;

}

//容量为0，前多少个物品的价值都是0 for(int i=0;i<len;i++){

V[i][0] = 0;

}

for(int weight=1; weight<=m;weight++){ for(int n=1;n<=len;n++){

//当前的物品的容量小于等于剩余容量，则判断加入当前物品后总容量与未加入容量的大小比较

if(wei[n-1] <= weight){

V[n][weight] = Math.max(val[n-1]+V[n-1][weight-wei[n-1]], V[n-1][w

eight]);

}else{

//如果当前物品的容量大于剩余容量，则一定不能加入该物品，则最大价值与前n-1个的最大价

值相同

V[n][weight] = V[n-1][weight];

}

}

}

for(int i[]:V){ for(int j:i){

System.out.print(j + " ");

}

System.out.println();

}

return V[len][m];

}

public static void main(String[] args){ int Weight[] = {2,3,1,4,6,5};

int Value[] = {5,6,5,1,19,7};

int nCapacity = 10;

System.out.println(DP(Value, Weight, nCapacity));

}

}

# Linux

**Linux**常用命令

netstat top

df -h dstat ifstat ifconfig

uname -a 查看内核\/操作系统\/CPU信息

head -n 1 \/etc\/issue 查看操作系统版本

cat \/proc\/cpuinfo 查看CPU信息

hostname 查看计算机名lspci -tv 列出所有PCI设备lsusb -tv 列出所有USB设备

lsmod 列出加载的内核模块

env 查看环境变量资源

free -m 查看内存使用量和交换区使用量

df -h 查看各分区使用情况

du -sh <目录名> 查看指定目录的大小

grep MemTotal \/proc\/meminfo 查看内存总量

grep MemFree \/proc\/meminfo 查看空闲内存量

uptime 查看系统运行时间、用户数、负载cat \/proc\/loadavg 查看系统负载磁盘和分区mount | column -t 查看挂接的分区状态fdisk -l 查看所有分区

swapon -s 查看所有交换分区

hdparm -i \/dev\/hda 查看磁盘参数(仅适用于IDE设备) dmesg | grep IDE 查看启动时IDE设备检测状况网络

ifconfig 查看所有网络接口的属性iptables -L 查看防火墙设置route -n 查看路由表

netstat -lntp 查看所有监听端口netstat -antp 查看所有已经建立的连接netstat -s 查看网络统计信息进程

ps -ef 查看所有进程

top 实时显示进程状态用户

w 查看活动用户

id <用户名> 查看指定用户信息

last 查看用户登录日志

cut -d: -f1 \/etc\/passwd 查看系统所有用户cut -d: -f1 \/etc\/group 查看系统所有组crontab -l 查看当前用户的计划任务服务chkconfig –list 列出所有系统服务

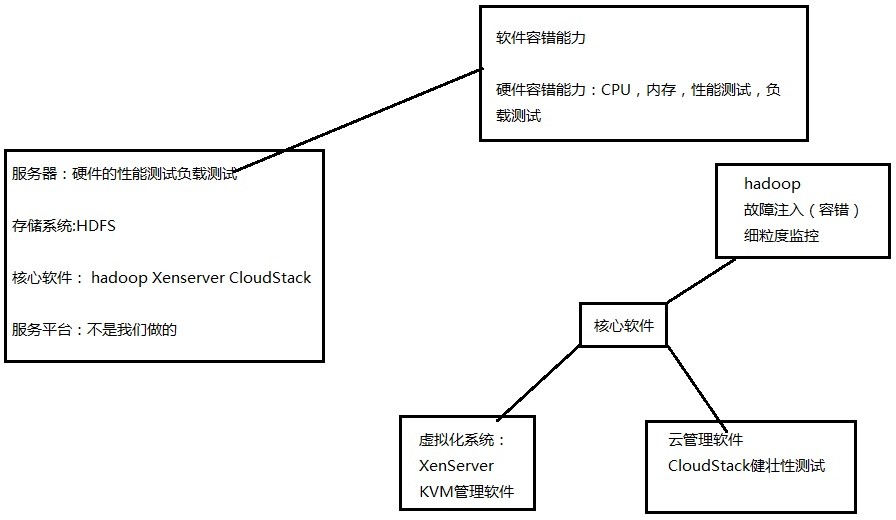
chkconfig –list | grep on 列出所有启动的系统服务程序

rpm -qa 查看所有安装的软件包

## 项目经历

云计算平台容错能力评估方法的研究

系统介绍：



该项目是采用软件技术实现的故障注入方法进行云计算平台容错能力的评估。本项目包括对Hadoop平台容错能力分析，云计算平台虚拟机容错能力分析等等方面，最终形成一个全方位的云平台容错能力评估系统，该系统是用JAVA实现的。

作为参与者，我主要负责的是Hadoop平台下的人为配置故障注入测试，软件健壮性测 试，过载测试以及块校验故障测试和最后评估系统中的平台节点状态监控模块的实现。人为配置故障注入测试，是通过写一个自动化更改hadoop配置的工具，更改hadoop配置文件的参数，主要有三种错误：拼写错误，语义错误（认知层次错误），结构错误（配置文件内容重复，缺失，配置结构错误），主要更改的文件包括，core-default.xml，

hdfs-default.xml, mapred.default.xml

软件健壮性，用的工具是findbugs， findbugs是一种静态的java代码分析工具，寻找真正的缺陷和潜在的性能问题，尽可能减少误报率，谷歌，Sun，Oracle等公司都在用

findbugs来寻找代码的缺陷。 主要检查bytecode中的bug patterns，如NullPoint空指针检查、没有合理关闭资源、字符串相同判断错（==，而不是equals）等。

**Bad practice** 坏的实践 一些不好的实践，下面列举几个：

**HE**： 类定义了equals()，却没有hashCode()；或类定义了equals()，却使用

Object.hashCode()；或类定义了hashCode()，却没有equals()；或类定义了

hashCode()，却使用Object.equals()；类继承了equals()，却使用Object.hashCode()。

**SQL**：Statement 的execute方法调用了非常量的字符串；或Prepared Statement是由一个非常量的字符串产生。

**DE**： 方法终止或不处理异常，一般情况下，异常应该被处理或报告，或被方法抛出。

**Correctness** 一般的正确性问题

可能导致错误的代码，下面列举几个：

**NP**： 空指针被引用；在方法的异常路径里，空指针被引用；方法没有检查参数是否

null；null值产生并被引用；null值产生并在方法的异常路径被引用；传给方法一个声明为

@NonNull的null参数；方法的返回值声明为@NonNull实际是null。

**Nm**： 类定义了hashcode()方法，但实际上并未覆盖父类Object的hashCode()；类定义了tostring()方法，但实际上并未覆盖父类Object的toString()；很明显的方法和构造器混淆；方法名容易混淆。

**SQL**：方法尝试访问一个Prepared Statement的0索引；方法尝试访问一个ResultSet的0

索引。

**UwF**：所有的write都把属性置成null，这样所有的读取都是null，这样这个属性是否有必要存在；或属性从没有被write。

Internationalization 国际化

当对字符串使用upper或lowercase方法，如果是国际的字符串，可能会不恰当的转换。

**Malicious code vulnerability** 可能受到的恶意攻击

如果代码公开，可能受到恶意攻击的代码，下面列举几个：

**FI**： 一个类的finalize()应该是protected，而不是public的。

**MS**：属性是可变的数组；属性是可变的Hashtable；属性应该是package protected的。

**Multithreaded correctness** 多线程的正确性

多线程编程时，可能导致错误的代码，下面列举几个：

**ESync**：空的同步块，很难被正确使用。

**MWN**：错误使用notify()，可能导致IllegalMonitorStateException异常；或错误的使用

wait()。

**No**： 使用notify()而不是notifyAll()，只是唤醒一个线程而不是所有等待的线程。

**SC**： 构造器调用了Thread.start()，当该类被继承可能会导致错误。

**Performance** 性能问题

可能导致性能不佳的代码，下面列举几个：

**DM**：方法调用了低效的Boolean的构造器，而应该用Boolean.valueOf(…)；用类似

Integer.toString(1) 代替new Integer(1).toString()；方法调用了低效的float的构造器，应该用静态的valueOf方法。

**SIC**：如果一个内部类想在更广泛的地方被引用，它应该声明为static。

**SS**： 如果一个实例属性不被读取，考虑声明为static。

**UrF**：如果一个属性从没有被read，考虑从类中去掉。

**UuF**：如果一个属性从没有被使用，考虑从类中去掉。

**Dodgy** 危险的

具有潜在危险的代码，可能运行期产生错误，下面列举几个：

**CI**： 类声明为final但声明了protected的属性。

**DLS**：对一个本地变量赋值，但却没有读取该本地变量；本地变量赋值成null，却没有读取该本地变量。

**ICAST**： 整型数字相乘结果转化为长整型数字，应该将整型先转化为长整型数字再相乘。

**INT**：没必要的整型数字比较，如X <= Integer.MAX\_VALUE。

**NP**： 对readline()的直接引用，而没有判断是否null；对方法调用的直接引用，而方法可能返回null。

**REC**：直接捕获Exception，而实际上可能是RuntimeException。

**ST**： 从实例方法里直接修改类变量，即static属性

资源过载， 资源过载是指使CPU和内存的使用率达到一定的高负荷值，是测试hadoop节点在高负荷的情况下的容错能力。hadoop的mapreduce是用来进行海量数据的处理的， 属于计算资源敏感性任务。将MapReduce子任务所在的JobTracker和TaskTracker所在的节点的CPU和内存资源占用，从而模拟资源过载时对MR任务的执行的影响。MEM占用就是通过不断的malloc内存使内存达到一定阀值，然后通过一定方法保持他不被回收。CPU占用就是先通过sysconf函数来获得CPU的核数，然后每个CPU核心都启动一个线程并使用pthread\_setaffinity\_np函数绑定在对应的CPU核心上，然后线程定期获取CPU的利用率，并通过调整CPU时间片的空闲时间与占用比例来使CPU的使用稳定在阀值附

近。

节点级和进程级故障注入：模拟节点宕机，和进程失效，通过杀死进程，是节点关机等

网络故障注入：流量控制器--tc工具，在流量输入输出端口上建立一个队列进行流量控制，TC在IP层与网卡层之间进行干扰。的netem模块，主要可以实现网络的丢包，延时，乱序，数据包副本等网络错误。

‘hdfs文件系统故障注入包括两部分，包括数据块文件进行故障注入和数据块校验文件进行故障注入。DataNode都有数据块的校验文件，如果校验文件发生错误的话就会发生故障。数据块文件就是通过文件名找到对应的数据块的存储位置，然后更改数据块的权限或者是破坏数据块数据，进而使得该数据开文件不好用。检验文件可以通过两个方面来着手，一个是通过更改校验文件的本身，另一个是从数据方面入手，更改hadoop源码。

Hadoop本身就提供Hadoop-FI接口来对hadoop源码CheckSum类进行故障注入。

节点监控模块实现，我是使用一种ssh的远程登录jar包进行linux系统的ssh远程登录，然后远程执行命令，并获取执行结果。这样实现了节点的信息采集，包括节点的CPU状 态，系统IO状态，MEM状态，硬盘状态，网络IO状态等等，分别用读取mpstat -P -ALL, top,iostat,df -h, netstat等等指令获取响应信息，并通过jfreechart动态画出对应的图像。

项目困难点：首先是要学习的东西比较多，首先得对hadoop整个的工作流程以及源码有一定的了解，整个过程比较繁杂，得分模块一块一块的进行解决。找到哪些地方可以进行故障注入，以及采取什么样的故障注入方法都是非常需要考虑的。因此该项木的困难点就是相关文献，文章的查找，无论是看论文，还是上网查资料都或非了很多的时间。除了故障注入以外，还要对故障注入后系统的反应做出记录，整个过程必须非常繁杂， 只有一步一个脚印方可完成。

基于**HDFS** 的云存储系统的设计与实现

该项目是在Linux 下实现，利用hdfs 存储用户数据，类似百度云盘的操作界面。用Spring 的MVC 架构实现了文件上传、下载、删除、新建等基本功能。

作为负责人和参与者，我主要设计了系统架构，实现了系统后台的大部分功能，以及开发过程中组织开会协调项目工作。

## 动态可重构穿戴式计算机系统实现（**C++**）

实验室自主研发的可穿戴式计算机，在Linux端实现了一个可穿戴式计算机系统，在

Windows端实现了一个指挥中心，实现了可穿戴计算机与PC端进行通信,Linux端用的是QT creator，Windows端用的VC++6.0。

作为项目参与者,我主要负责PC端和穿戴机的语音，文字消息，GPS信息交互代码

项目困难点：主要困难点就是arm板上会出现很多奇奇怪怪的问题，比如：字符编码乱 码，需要在arm板子上一直编码，网络连接不稳定等等，我在语音交互的过程中就发生非常蛋疼的问题，由于我们采用的是udp进行传输数据，从pc端想移动端发送信息没有问 题，一切正常，但是当从移动端向pc端发送语音就会有问题，会出现有时能够收到包， 优势收不到包的问题，这个问题困扰了我很长时间，通过wireshark等工具抓包分析，反正试了各种方法，还是不行，最后博士师兄发现可能是由于arm板本身的网络接口的问 题，它对广播包并不会实时进行获取，只有在被指定时才一定会打开端口，而未指定的话可能打开也可能不打开端口接受数据，所以我们发送的广播包就接收不到，最后我们相出一个办法就是通过心跳包的方式解决这一个问题。

主要工作就是网络通信方面的。

我们的文字信息和GPS信息交互是通过TCP实现的，用的是C++自带的TCP函数，语音包的发送语接收是用UDP实现的。主要困难还是在UDP中丢包的问题，乱洗问题还没怎么遇到。

UDP乱序和丢包解决办法：

给每个包加上序号，如果一定时间内接受到的信号不是希望得到的ID号，则发现丢包，请求客户端重新传输过来。

每接收一个包就回发送一个包，告诉已经接到。解决方法，

一是 客户端降低发送速度，可以等待回包，或者加一些延迟。

二是，服务器部分单独开一个线程，去接收UDP数据，存放在一个缓冲区中，又另外的线程去处理收到的数据，尽量减少因为处理数据延时造成的丢包。

有两种方法解决楼主的问题：

方法一：重新设计一下协议，增加接收确认超时重发。（推荐）

方法二：在接收方，将通信和处理分开，增加个应用缓冲区；如果有需要增加接收socket的系统缓冲区。（本方法不能从根本解决问题，只能改善）

网络丢包，是再正常不过的了。

既然用UDP，就要接受丢包的现实，否则请用TCP。

如果必须使用UDP，而且丢包又是不能接受的，只好自己实现确认和重传，说白了，就是自己实现TCP（当然是部分和有限的简单实现）

## 无线充电传感器网络点播式充电调度算法设计实现

提出一种无线充电传感器网络中的为传感器节点充电调度算法，并通过JAVA模拟实现传感器网络运行过程，验证了结果的有效性。

## 面试时常见问题：

###### 我为什么要聘用你？（你又什么优势）

第一点我想说的是，我能够的专业能力能够胜任这一岗位。我所投递的岗位是JAVA研 发。我本科所学专业是软件工程，研究生就端读的是计算机专业，有着比较扎实的计算机基础。同时，java对我来说也特别熟悉，在校期间参加了许多项目对我的编程能力有一个很大的提升，我自己也跟同学做过一些小的项目，明白如何在一个团队中做好自己的工作。因此我认为我的专业能力足以胜任这一工作。

第二点我想说的是，我学习计算机也已经有6年的时间了，这么多年下来，我对编程有着特殊的情感，我对编程有着很高的热情，如果公司能够聘用我，我会以很高的热情，积极的态度投入到工作中来，我相信一个积极的态度会给公司和我个人带来更大的收获。

第三点我向所的是，我是一个很好的团队型人员。我很善于与大家沟通交流，懂得如何在一个团队中找准自己的位置，为团队营造一个很好的氛围，做好自己的本质工作。我本科阶段有着三年的学生会经历，同时，参加项目的过程中也在很多的项目组中担任职责，所以，当我进入公司后我可以很快适应团队，尽可能的为我们的团队创造出最大的价值。

故障注入的意义：加速故障的产生，模拟系统现实运行时可能出现的错误，对系统可用性容错能力进行评价。benchmark

故障注入与测试的区别与联系：故障注入是通过人为的向系统中注入故障，结果只有两种， 导致或者未导致系统异常，而系统的异常与否是通过测试来实现的。

Hash函数怎样设计?

UDP怎样改造成TCP? NTU

拷贝构造函数？

#define和内联函数? 穿戴机网络传输？ 红黑树具体实现？

阿里面经

排序算法 稳定不稳定

稳定不稳定就是指相同的元素排序前后是否发生了变化。

java数据库事务 在哪个包里了？

在Connection里了，setAutoCommit，commit, rollback,savepoint等等，在java.sql.Connection.

怎样处理高并发

如果服务器端反应过慢，原因是什么

hashmap底层类加载机制 内存模型

为什么要进行并发编程 实际情况中如何避免死锁

如果某一个时间点并发量很大的情况下应该怎么做？

1.利用弹性的原理，当线程池的大小不足以处理当前数量的线程的时候可以动态增大线程池的大小，当线程比较少达到一定阈值的情况下，进行动态减小线程池的大小

2.利用消息队列，用消息队列来存储用户发送过来的数据，而不是直接写入到数据库中，当用户发送过来数据时，我们把它存入的消息队列中，然后另开一个线程写一个死循环来一直读写队列中的第一个数据，这样就可以达到很快的处理速度了。因为数据库的操作的速度内存操作的速度查了几个量级，所以这样可以提高数据的处理速度。

反应慢的瓶颈：

1.数据库的读写慢，这个读的可以用索引来实现。但是索引只加快了读，其他写，改都增加了困难，性能具体是否有提高还有未可知。加快写的方法的是增加一个消息队列，用消息队列来存储部分比如提交签到信息等更新提交数据库的信息，然后单独开一个线程一直读消息队列的头，然后向数据库里写数据。

2.多线程创建线程的过程慢，采用线程池来解决。线程池可以减少线程创建和销毁的时间。使用框架：json-lib框架，Spring框架，Hibernate框架

json-lib框架用来传输数据，里面存放的是对象

Spring框架用了其中的AOP，主要是用来产生日志的，当有响应的更新等操作的时候，会产生日志。

Hibernate框架主要是用来进行对象持久化的，当接收到json穿过来的对象的时候，我们就把这个对象进行持久化，这样就可以了，如果是更新就跟新，如果是insert就insert，如果是

delete就delete

还用到了Hibernate框架的缓存机制，分为一级缓存和二级缓存，我们为它设置了二级缓存， 将现有的所有的用户名信息都缓存到了内存当中，使用的第三方类是ehcache

一：100层楼 两个一样的鸡蛋 问你 最少需要多少次能够试验出哪一层楼是鸡蛋碎了的临界层

这道题最开始的想法是二分查找，但是第一次试验第50个，如果碎了，试验25但是又碎了怎么办。。。所以不可以，如果50层碎了，那么只能从第一层一层一层网上试验50次。这个给了我们启发，我们需要一个临界值来试验。假设这个临界值是K次，第一次试验第K层，如果碎了，那么就需要K-1次就知道了答案，如果没碎，则浪费了一次机会，还有K-1次，如果还没有碎，接着K-2层，则如果一直没碎所试验的楼层是就是K+K-1+K-2+....+K-(K-1) =

K(K+1)\/2 >= 100 K=14次。

二： Linux awk 以及top 里面的load代表什么

load average:0.15, 0.16, 0.15

系统负载，即任务队列的平均长度。三个数值分别为 1分钟、5分钟、15分钟前到现在的平均值。

(就是进程队列的长度，每隔5秒钟检查一次活跃的进程数，显示为1分钟、5分钟、15分钟内的每次检查结果的平均值。

当这个值>cpu核心数的时候就说明有进程在等待处理了，被认为卡，也就是负载过重)

1.判断链表有环？

2.判断2个链表相交？

3.怎么测试程序？sonar

4.中断？ 5.tcp udp

6.3次握手

1. tcp拥塞控制
2. tcp流量控制

9.数据库索引太多？

10.行锁实现？

11.左右连接？

12.死锁？

13.找出出现次数超过一半的数O(n)

14.电梯调度算法，给出测试用例？

15.最有成就的事？外包项目分工，时间分配？

16.怎么分工交流?

1. java泛型实现，好处？好处是进行安全检查，所有的转换都是自动的，提高代码的重用率。

Java中的泛型基本上都是在编译器这个层次来实现的。在生成的Java字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会在编译器在编译的时候去掉。这个过程就称为类型擦除。

1. java反射理解？什么时候用，好处？

Java反射机制主要提供了以下功能： 在运行时判断任意一个对象所属的类；在运行时构造任意一个类的对象；在运行时判断任意一个类所具有的[成员变量](http://baike.baidu.com/view/684821.htm)和方法；在运行时调用任意一个对象的方法；生成动态代理。

19.举个例子体现面向对象封装继承多态特性

20.什么时候会OOM？出现在哪些区域？举不同的内存溢出例子？

21.jvm内存模型？老年代标记清除，标记整理算法怎么觉得用哪个？

22.什么对象被回收？ 23.GC roots引用链哪些？

24.fixsize线程池？线程池原理？

25.16位操作系统内存怎么算？

26.迷宫问题？栈回溯？多条路径比较？

27.封装对象有多种需求，字段不一样，怎么弄？

28.MVC含义，好处？

29.二分插入，关键字相等怎么插入？

1. awk，流量网卡查看？linux命令

31.看过啥源码？

32.最近在看什么？

33.怎么做sql优化？添加索引，explain工具

34.实习收获？

35.怎么提高自己，主动找任务，学习别人的东西？

聚集索引 非聚集索引 怎么查数据 联合索引

JSON框架

json-lib框架，jackson框架线程安全的Collection

递归创建二叉树

设计模式（jdk或者框架中哪些用到了）

ngix服务器（）

Mysql int(5) 的5是干什么的

TCP\/IP是怎样处理应用层的请求的阿里云，亚马逊云提供哪些服务

JVM内存模型，垃圾回收

自己做客户端的时候，除了服务器，还要注意哪些事情。 DNS劫持

Java后端框架有哪些

http协议了解吗

# J2EE

Spring是一个轻量级的控制反转和面向切面的容器框架。

IoC控制反转就是把对象控制权交出去，用的时候直接拿到就行了。

AOP（事务管理、日志等）将业务逻辑与系统逻辑分开。

接口：就是生命

抽象类的方法既可以实现也可以只声明。interface的方法只能声明。在JAVA8中，interface可以有方法体了。

面向接口编程例子：

向外只提供接口，而不提供实现类，所以看不到类的实现，只能知道方法。

**IOC(**控制反转**):**本来由应用程序管理的对象之间的依赖关系，现在交给了容器管理，这就叫控制反转，即交给了**IOC**容器。**Spring**的**IoC**容器主要使用**DI**方式实现的。不需要主动查找，对象的查找、对位和创建全部由容器管理。

1、大量减少了Factory和Singleton的数量，使代码层次更加清晰，主要原因是我们不再查找、定位、创建爱你和管理对象之间的以来关系了，都交给IoC容器管理了

2、Spring的IoC容器是一个轻量级的容器，灭有侵入式，不需要以来容器的API，也不需要实现一些特殊接口

3、一个合理的实际最好尽量避免侵入性

4、会使我们的矿建（Structs和Hibernate)工作的更好

5、提供了AOP声明式服务能力，可以针对POJO对象提供声明式服务能力：如：声明式事务

6、对于资源，如Hibernate Session或JDBC Connection我们不在负责开启和关闭

7、鼓励了我们面向接口编程

8、减少了代码中的耦合（解耦合），将耦合推迟到了配置文件中，发生了变化也更容易控制了

Ioc：控制反转，控制权的转移，应用程序本身不负责以来对象的创建和维护，而是由外部容器负责创建和维护。

DI（依赖注入）:就是由IOC容器在运行期间，动态地将某种依赖关系注入到对象中。注入的是成员变量.

哪些方面的控制被反转了呢？

获得依赖对象的过程被反转了。于是给控制反转取了更合适的名字:依赖注入。实际上是实现

IOC的方法：注入。

IoC容器中所有的对象叫做Bean。

Spring的Bean配置。

有了IOC就不必自己创建对象了。

通过接口隔离的方式实现控制反转

所谓依赖注入，就是指程序运行过程中，如果需要调用另一个对象协作，无需再代码中创建被调用者，而是依赖于外部容器的注入。

**Spring**的属性常用注入方式**:**

* 1. 构造注入---->调用构造方法
  2. 设值注入--->调用setter()方法

Spring可以用一个数学公式来表达：Spring = XML + Bean。

Spring通过BeanFactory或者ApplicationContext来管理Bean。属性**init()**操作发生在设置属性之后。

多个Bean之间存在联系，这种联系就叫做依赖。

**Spring** 容 器 通 常 有 两 种 表 现 形 式 ： BeanFactory和ApplicationContext。ApplicationContext实现了BeanFactory接口。实现BeanFactoryAware接口的Bean拥有访问BeanFactory容器的能力。

创建**Bean**的几种方法：

1. 调用构造器来创建Bean实例
2. 调用静态工厂方法创建Bean
3. 调用实例工厂方法来创建Bean

协调作用域不同步的**Bean**

当singleton作用域的Bean依赖于prototype作用域Bean时，会产生不同步的现象，我们可以采用在xml文件中设置Bean的lookup方法，这样每次想要新的被依赖实例只要调用createAxe方法就可以了——这个方法在被依赖Bean的代码中是一个抽象类，由Spring框架进行具体实

现。

###### 视频学习内容：

了 解 常 见 属 性 的 注 入 ： int，String，List， Set， Map的注入什么事属性编辑器和作用？

属性编辑器：将spring配置文件中的字符串转换成相应的java对象 spring内置了一些属性编辑器，也可以自定义属性编辑器

如何自定义属性编辑器？

1：继承PropertyEditorSupport 2:覆盖setAsTest()方法 3：将自定义的属性编辑器注入到

spring中

了解关于多配置文件的读取方式？

1：可以采用字符串数组 2：可以采用\*匹配模式

如何减少**spring**的配置文件？ 1：通过<bean>标签将公共的配置提取出来，然后指定<bean> 标签中的abstract属性为true，然后再在其他<bean>标签中指定其为parent是这个bean的ID即可 **spring**默认在创建**BeanFactory**时，将配置文件中所有的对象实例化并进行注入，但可以采用相关的属性延迟配置文件的初始化，如：**default-lazy-int="true"**即可 **spring bean**的作用域 scope取指：singleton:默认值，每次调用getBeans()向IOC容器中取得的对象是相同的。prototype：每次调用getBean()向IOC容器中取得对象是不相同的。还用request，session，

global等在javaee中有所应用。 自动装配：

Spring IoC容器可以自动装配（autowire）相互协作bean之间的关联关系。因此，如果可能的话，可以自动让Spring通过检查BeanFactory中的内容，来替我们指定bean的协作者（其他被依赖的bean）。

简而言之,就是对于bean当中引用的其他bean不需要我们自己去配置它改使用哪个类,Spring 的自动装配可以帮助我们完成这些工作。

§2 自动装配的意义?

引用理解自动装配的优缺点是很重要的。其中优点包括：

自动装配能显著减少配置的数量。不过，采用bean模板（见这里）也可以达到同样的目的。

自动装配可以使配置与java代码同步更新。例如，如果你需要给一个java类增加一个依赖，那么该依赖将被自动实现而不需要修改配置。因此强烈推荐在开发过程中采用自动装配，而在系统趋于稳定的时候改为显式装配的方式。

缺点是bean之间的协作关系就不清晰了，所以建议在开发时候用他，如果开发完毕以后尽量补全

**AOP**

AOP叫做面向切面编程，它与OOP面向对象编程是互补的。AOP专门用于处理系统中分布于各个模块（不同方法）中的交叉关注点问题，在JAVA EE应用中，常常通过AOP来处理一些具有切面性质的系统级服务，如事务管理，安全检查，缓存，对象池的管理等。

因为业务处理与这些其他的这些安全检查的这种切入到各个方法中的各个角落里的独立服 务，这些叫做横切的交叉观察点。这些安全检查之类的我们可以用filter, 动态代理还有AOP都能实现。

AspectJ是一个基于JAVA语言的AOP框架。

##### AspectJ

AspectJ包括两个部分：

1. 定义了如何表达、定义AOP编程中的语法规范
2. 工具部分，包括编译器和调试工具等

AOP在不更改某个方法的内容的情况下，为该方法添加了许多功能，而且这些功能在调用的时候的人并不知道，只要跟原来一样调用方法一样，这个跟动态代理有点像呀。

SpringAOP采用基于代理的AOP实现方案（可以是Java动态代理也可以是CGLIB动态代 理），而AspectJ采用的是编译时增强的解决方案，也就是向字节码注入自己想要执行的东西。

AOP编程程序员只需要参与3个部分：

1. 定义普通业务组件（也就是定义一个方法）
2. 定义一个切入点，一个切入点可能横切多个业务组件（定义一个切入点，它可以同时对多个方法进行管理）
3. 定义增强处理，增强处理就是在AOP框架为普通业务组织织入的处理动作（也就是你要做的额外的动作）

Cross Cutting Concern(横切性关注点)

是一种独立服务，它会遍布在系统的处理流程之中

Aspect

对横切性关注点的模块化

Advice

对横切性关注点的具体实现

PointCut

它定义了Advice应用到那些JoinPoint上，对Spring来说JoinPoint就是方法调用

Weave

将Advice应用到Target Object上的过程叫做织入，Spring支持动态织入Target Object

Advice被应用的对象Proxy

SpringAOP默认使用JDK的动态代理，代理是在运行时创建的。也可以使用CGLIB代理Introduction

可以动态的为类添加方法

JoinPoint

Advice在应用程序上执行的点（时机）Spring只支持方法的JoinPoint。这个点也可以使属性修改，如：AspectJ可以支持

**Spring** 对**AOP**的支持

1. 如果目标对象实现了接口，在默认情况下会采用JDK 的默认动态代理来实现AOP
2. 如果目标对象实现了接口，也可以强制使用CGLIB生成代理AOP(但建议不要用CGLIB)
3. 如果目标对象没有实现接口，那么必须引入CGLIB， Spring会在JDK的动态代理和

CGLIB代理之间切换

如何使用强制使用**CGLIB**生成代码？

* 加入CGLIB库， SPRING\_HOME\/lib\/cglib\/\*.jar
* 加入如下配置，强制使用CGLIB代理

<aop:aspectj-autoproxy proxy-target-class="true"\/>

**JDK**动态代理和**CGLIB**代理的区别？

* JDK动态代理对实现了接口的类进行代理
* CGLIB可以对类进行代理，主要对指定的类生成一个子类，因为是继承我们的目标最好不要使用final声明

**J2EE**基础

### Servlet

Servlet： Server+let 就是服务器端的小程序。正规说：Servlet是用Java编写的服务器端的组件（小程序），运行在Servlet容器中，和JSP通称为web层的2大组件，Servlet和客户的通信采用“请求\/响应“模式，即我们平时说的接收和处理客户端请求，并往客户端返回一个响应。J2EE Web层除了JSP和Servlet这2大组件，还有过滤器和监听器组件。

JSP：全称是Java Server Pages；是Web层组件；用来创建动态网页。如网上购物等等，可以用JSP实现。

我们前面学过Servlet，知道Servlet可以实现动态网站。那么为什么不用Servlet？Servlet刚出现时确实是用作表示的组件。

但是Servlet有2个重要缺点：

（1）进行前台--表示层设计的人员主要是美工，美工熟悉html，但是对Java不熟悉；Servlet 要求熟练掌握Java编程；

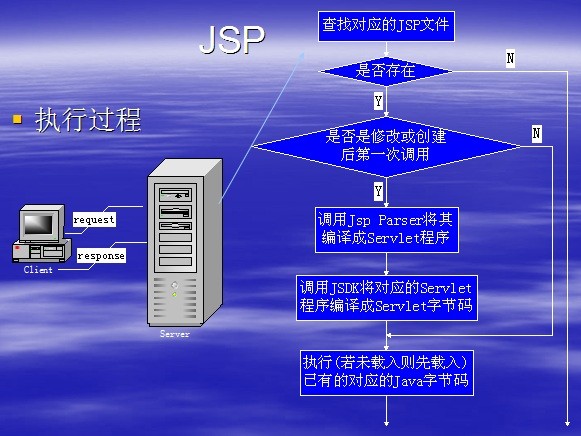
（2）另外，Servlet不适合作为表示组件。为什么？大量的out.println()语句很难维护。

**JSP**执行过程。

客户端浏览器能解释HTML标记，不能直接解释执行JSP文件。那么就存在一个服务器端的从

JSP到HTML的转化。实际上Web服务器如Tomcat是将JSP翻译成Servlet源文件（.java文

件），再将Servlet源文件编译成类文件（.class文件）；最后加载类文件创建Servlet实例为客户提供服务。如果客户访问某个JSP那么Web服务器将会将该JSP最后以Html格式发送到客户端浏览器中。



**JSP**元素分为**4**大类：注释；指令；脚本元素； 动作元素。

**JSP**注释**:**

有3种注释

**JSP**指令**:**

JSP指令分3种：页面指令即page指令；包含指令即include指令；标签库指令即taglib指令。这三种指令格式是：<%@ XXX 指令属性……%>。XXX是page、include、taglib之一。

“page”指令用于定义JSP文件的全局属性。 include指令：向当前页中插入一个静态文件的内容。

taglib指令：使用标签库指令的JSP语法<%@ taglib uri="URIToTagLibrary" prefix="tagPrefix"

%>用于引入定制标签库。

**JSP**脚本：

脚本元素分为3种：脚本段；表达式；声明。

①脚本段就是Java代码片段，不是完整的类文件而是类文件的一部分。语法格式如下：

**<% Java code fragment %>** 或**<%** 代码 **%>**

②表达式：是符合Java语法的合法表达式，作用：在当前位置显示表达式的结果。语法格式如下：

**<%= expression %>** 或**<%=** 表达式 **%>**

③声明：在JSP中，声明是一段Java代码，用来声明变量和方法，声明后的变量和方法可在该

JSP文件的任何地方使用

注意：声明的变量属于实例变量。

JSP 语法格式如下：

**<%! declaration; [ declaration; ] ... %>** 或**<%!** 声明**; [**声明**; ] ... %>**

**JSP**动作元素：

JSP动作元素有很多，常用的有：

<jsp:forward>：跳转到另外一个界面，可以向动态页面传入参数，<jsp:param name="',value="">

<jsp:include>：既可以加动态页面，也可以加静态页面，也可以向动态页面传入参数

<jsp:useBean>

<jsp:getProperty>

<jsp:setProperty>

后3种在JSP使用JavaBean时讲解。下面我们先看看<jsp:forward>和<jsp:include>。

**JSP**内建对象

为了简化页面的快速开发，JSP提供了许多内容对象如request等，这些对象不需要JSP编程者实例化，它们是由容器实现和管理的，我们称为内建对象。常用的内建对象有request对 象、response对象、out对象、pageContext对象、session对象、application对象、config对象、page对象、exception对象。所有的隐含对象只能用于脚本段代码或表达式，不能在JSP 声明中使用。

**out**对象

out对象用于向客户端输出数据。它有多个重载的方法可以输出基本数据类型（boolean、

char、char[]、double、float、int、long）、String类型、Object类型等。使用格式：

out.print(boolean)或out.println(boolean)；这两个方法的区别是第一个只是输出数据，第二个输出数据后会再后面加上一个换行符。但是该换行符不能被浏览器解析，要真正换行需要使用out.println(“<br>”)来实现。

**request**对象

request对象代表请求对象，客户端的请求信息被Web容器封装到该对象中。

request对象可以通过getParameter();来获取表单数据

一般情况下，表单设计时使用的标记包括：<form>、< input>、<Option>、<Select>、

<textarea>和<isindex>。

注意：解决**get**请求中文乱码问题，可以到**tomcat**的**server.xml**文件中去解决，修改编码为

**UTF-8**而默认的是**ISO-8859-1**

解决**post**请求中文乱码问题，用**request.setCharacterEncoding("GBK");**来解决

##### response对象

response对象代表服务器发送到客户端的响应，主要功能是服务器端向客户端发送信息，例如设置HTTP的头信息、设置Cookie信息、重定向。

response.setHeader();设置Http头信息最有用的就是设置refresh，可以一定时间刷新一次，也可以实现一定时间后跳转到另一个界面当中。

response.setRedirect(跳转路径)；设置跳转的路径。

使用**response**设置**cookie**

HTTP协议是无状态的，这表示它在一个请求结束、另一个请求开始时，不会记住前一个请求的所有信息。Web应用通常要处理用户的多个连续的请求，而且还要记住前一个请求的一切信息，这就是常常提到的用户会话跟踪。能实现用户会话跟踪的方式有session与cookie，另

外还有隐藏域和URL重写。

**cookie**保存在客户端的硬盘上，是明码信息，安全性不好。

下面我们编码实现通过response对象和request对象使用cookie来记录用户上次登录时间。

**Session**对象

JSP中的session对象是javax.servlet.http.HttpSession接口的实例化对象。**session**对象是用户所私有的

**session**对象实现用户会话跟踪的原理。

当客户（浏览 器如IE）访问服务器上的一个JSP页面时，JSP引擎（如Tomcat）会产生一个

session对象（此对象在服务器上内存中），该session对象有一个String类型的ID号，能唯一区分该session。JSP引擎同时将这个ID号发送到客户端，如果客户端允许使用cookie就将ID 放在cookie中；客户端不允许使用cookie就将该ID放在URL的结尾。这样session对象与客户之间建立了一对一的关系。当客户再访问该服务器的其他页面时，不再给客户分配新的

session对象，直到客户浏览器关闭或者session过期或者调用了session对象的invalidate方法。当客户重新打开浏览器再次访问该服务器时，服务器会为该客户重新在创建一个新的

session对象。

##### application对象

application对象实现了javax.servlet.ServletContext，；这个对象被某个**Web**应用的所有的客户共享。当Web服务器启动后某个Web应用开始工作时由Web容器自动创建该对象。

该对象常用方法：

public void setAttribute(String key, Object obj); public Object getAttribute(String key);

application对象主要应用：在线人员统计；在线人员名单列表。

如果要释放application占用的资源，只能重启Web服务器。

##### pageContext对象

pageContext对象属于javax.servlet.jsp.pageContext接口类型，主要功能是管理JSP中特殊的可见部分中已命名的对象的访问。该对象是容器创建的，在JSP中可以直接使用。

该对象的常用方法有：

forward(String anotherPage);

getAttribute(String key[,int scope]);\/\/得到某个范围内存放的对象，通过存放时的名字setAttribute(String key,Object obj[,int scope]);\/\/在某个范围内存放一个对象及其名字

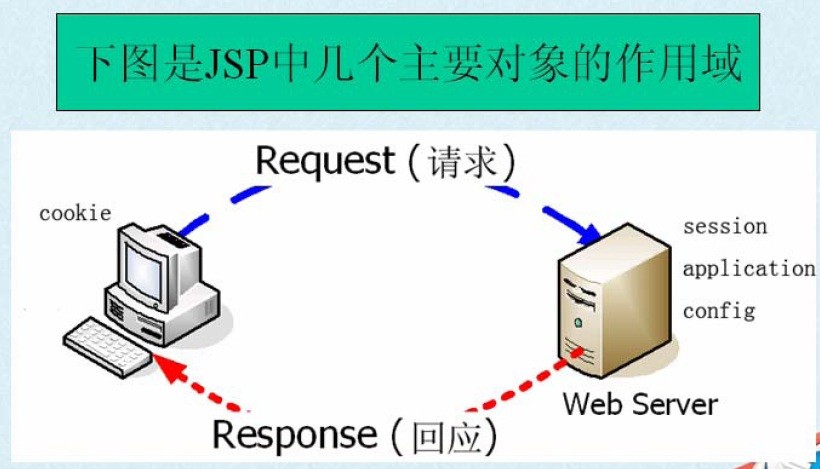
getRequest();\/\/得到当前的request对象

getResponse();\/\/得到当前的response对象

removeAttribute((String key[,int scope]);\/\/删除某个范围内存放的对象，通过存放时的名字

###### 四种对象的作用范围

request、session、application这3个保存属性的对象都需要跨多个页；属性保存是有内存开销的，范围越大开销越大，则服务器性能会降低；所以，能使用request的就使用request来保存属性对象而不用session或application；能使用session保存的就不用application。关键是看待保存的属性对象需要的范围。pageContext只作用于页面，不会跟随跳转跳转。



# JavaBean

JavaBean是Sun公司制定的一种可重用的、跨平台的软件组件。JavaBean有2种：一种有用户界面的JavaBean；另一种是处理业务逻辑（操作数据和数据库）的JavaBean。JSP使用的是后一种JavaBean。

JSP与JavaBean结合使用的好处：

①提高了代码的可维护性：最初Web应用可以使用JSP来单独创建，JSP既负责表示逻辑、流程控制、还有业务逻辑（可能访问数据库），使得代码难以维护；有了JavaBean，可以让

JavaBean负责业务逻辑和数据访问逻辑，而只让JSP负责表示逻辑和流程控制逻辑，提高了程序的可维护性。

②易于角色分工：不熟悉Java的美工人员可以只负责JSP的修饰而不用掌握Java编码。

③有利于代码重用：业务处理由JavaBean来完成，可以充分利用JavaBean的可重用的特点， 提高开发效率。

**JavaBean**与**EJB**的区别：

EJB是J2EE的核心技术之一：创建分布式、服务器端的企业应用程序的组件。

JavaBean组件不属于J2EE规范。那么为什么要讲解它呢？JavaBean技术是J2EE的基础，尤其是在构建web应用时离不开JavaBean。

**JavaBean**的编写规范

**①JavaBean**是一个**public**的类；

**②JavaBean**有一个无参数的构造方法；

**③JavaBean**的每个属性最好是**private**的，每个属性有一对**get**和**set**方法，通过**get**和**set**方法来读取和设置属性。

**④JavaBean**中可以增加业务方法，来进行数据库的访问操作，可以查看后面的

JSP+JavaBean开发模式章节的相关内容。

**JSP**中访问**JavaBean**

Jsp中有2种方式来访问JavaBean

①通过脚本段代码访问JavaBean；

②通过<jsp:useBean>动作元素（建议使用）。

javaBean在<jsp:useBean>访问时可以设置scope即作用范围。作用范围同样包括

session,request,application。

#### 重点总结：

response，session，request这些对象都有setAttribute(key,value);方法

我们可以用session.setAttrabute(key,value)设置用户信息，然后通过getAttrabute(key)判断它是否为空来判断是否用户是在登录的状态。

我们可以给request.setAttrabute(key,value)来存放我们在这个页面中通过数据库或者其他途径得到的数据加到request中，然后把他分发给下一个界面，这样可以在下一个界面中提取出这个数据，比如从数据库中得到所有的用户信息，放到一个List中，然后把List加到request中，然后把request dispatch给下一个界面，我们可以在下一个界面中根据这个

request来把所有的用户信息呈现出来。

分页技术实现

**SmartUpload**实现文件上传下载

这是一个开源工具，下载下来jar包，就可以用SmartUpload类来实现文件的上传与下载，改名等一系列功能。

**Tomcat**数据源配置

本章学习**JNDI**和**Tomcat 6**下数据源的配置。

JNDI：Java命名和目录接口的API。全称是Java Naming and Directory Interface；

借助于JNDI API，为开发人员提供了查找和访问各种命名和目录服务的通用、统一的接口。开发者能通过名字访问用户、机器、网络、对象、服务。比如可以访问公司内网的打印机、访问Java对象、关系数据库等。

J2EE中的EJB、JDBC等常常使用JNDI获取数据源等。这个地方主要就是数据源DataSource的使用。

## 过滤器解决中文乱码及登录验证

前面讲过J2EE的Web层的组件主要是JSP和Servlet，除此之外还有过滤器和监听器。

**Get**和**post**的区别

我们在进行网页开发的过程中经常涉及的form表单的提交的方法，有两个可选项：get，

post。 Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是**GET**，**POST**，

**PUT**，**DELETE** 。然而，web开发中只提供了两种Get和Post。get是获取服务器的资源，

post是向服务器发送资源。

URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资 源，而HTTP中的GET，POST，PUT，DELETE就对应着对这个资源的查 ，改 ，增 ，删 4

个操作。到这里，大家应该有个大概的了解了，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(个人认为这是**GET**和**POST**的本质区别，也是协议设计者的本意，其它区别都是具体表现形式的差异 )。

###### 两种提交方式的区别：

（1）GET提交，请求的数据会附在URL之后（就是把数据放置在HTTP协议头中），以?分割

URL 和 传 输 数 据 ， 多 个 参 数 用 & 连 接 ； 例 如 ：login.action? name=hyddd&password=idontknow&verify=%E4%BD%A0 %E5%A5%BD。如果数据是英文字母\/数字，原样发送，如果是空格，转换为+，如果是中文\/其他字符，则直接把字符串用

BASE64加密，得出如： %E4%BD%A0%E5%A5%BD，其中％XX中的XX为该符号以16进制表示的ASCII。

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

因此，GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变

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

而在实际开发中存在的限制主要有：

GET:特定浏览器和服务器对URL长度有限制，例如 IE对URL长度的限制是2083字节

(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系 统的支持。

因此对于GET提交时，传输数据就会受到URL长度的 限制。

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

(3)安全性：

POST的安全性要比GET的安全性高。注意：这里所说的安全性和上面GET提到的“安全”不是同个概念。上面 “安全”的含义仅仅是不作数据修 改，而这 里安全的含义是真正的Security的含义，比如：通过GET提交数据，用户名和密码将明文出现在URL上，因为(1)登录页面有可

能被浏览器缓存， (2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用GET提交数据还可能会造成Cross-site request forgery攻击

（4）Http get,post,soap协议都是在http上运行的

1. get：请求参数是作为一个key\/value对的序列（查询字符串）附加到URL上的

查询字符串的长度受到web浏览器和web服务器的限制（如IE最多支持2048个字符），不适合传输大型数据集同时，它很不安全

1. post：请求参数是在http标题的一个不同部分（名为entity body）传输的，这一部分用来传输表单信息，因此必须将Content-type设置为:application\/x-www-form- urlencoded。post 设计用来支持web窗体上的用户字段，其参数也是作为key\/value对传输。

但是：它不支持复杂数据类型，因为post没有定义传输数据结构的语义和规则。

1. soap：是http post的一个专用版本，遵循一种特殊的xml消息格式Content-type设置为: text\/xml 任何数据都可以xml化

**EL**语法（表达式语法）

EL语法格式：${XXXScope.varName.property}或${XXXScope.varName[property]}

如果不指定scope的话，它就会从page，request，session，application中依次寻找，找到立刻返回，找不到返回null。

**JSTL**

虽然，SUN公司有了自定义标签可以使得程序员使用自定义标签来替代脚本段代码，但是会导致不同程序员创建重复功能的自定义标签。于是，SUN公司推出了JSTL，一个统一的官方的标准标签库，实现了常用功能，来解决替代脚本段的问题。

**JSTL**的优点：

可移植性好：JSTL规范要求实现了JSTL的容器必须实现其相应的接口，这样，JSTL代码可以在应用服务器之间无缝移植。

最大限度的减少了JSP中的脚本代码，简化了Web程序的开发..

**Sun** 发布的标准 **JSTL1.1** 标签库有以下几个标签：

核心标签库：包含 Web 应用的常见工作，比如：循环、表达式赋值、基本输入输出等。

国际化标签库：用来格式化显示数据的工作，比如：对不同区域的日期格式化等。

数据库标签库：可以做访问数据库的工作。

XML 标签库：用来访问 XML 文件的工作，这是 JSTL 标签库的一个特点。

函数标签库：用来读取已经定义的某个函数。

在使用**Core**标记库前，必须使用**<%@ taglib %>**指令，并指定**prefix**和**uri**的值。通常设定如下：

<%@ taglib uri="http:\/\/java.sun.com\/jsp\/jstl\/core" prefix="c" %>

<c:out>这个是输出标签，就是往网页上输出东西，这里面可以用EL表达式

<c:set>该标签用于在四种属性范围（pageContext,request,session,application）中设置某个值，或者设置某个对象（通常是JavaBean）的属性。

<c:remove>该标签用于删除指定范围内的某个变量或者属性。

<c:choose>相当于switch，<c:when>相当于case，<c:otherwise>相当于default。

### DAO

将数据库有关操作抽象出来，形成一个接口(DAO），然后实现这个接口，在实现类里实现数据库操作，这样讲业务逻辑与数据库操作解耦。

ORM：Object Relationship Model 对象-关系映射。

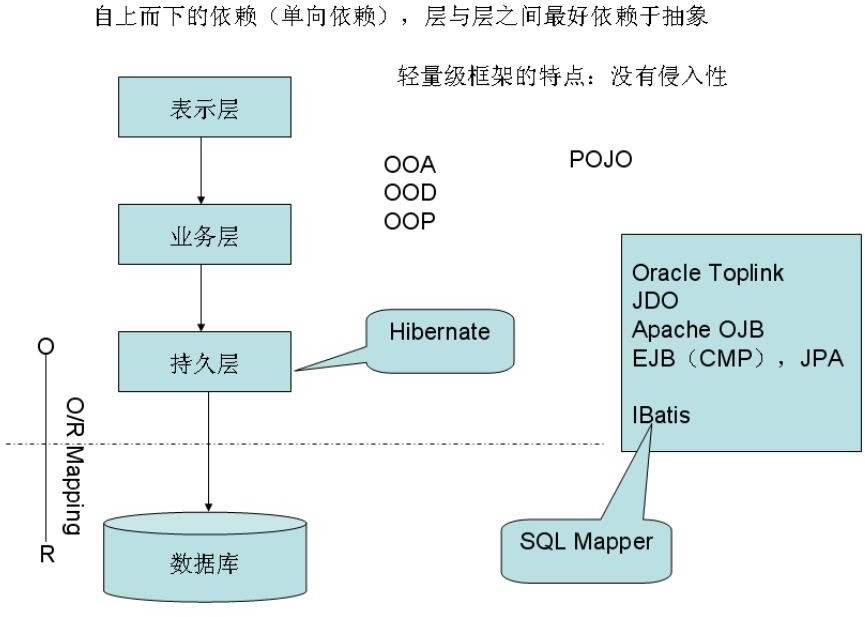
简单说就是把对象映射到数据库中，对数据库的操作用对对象的操作来实现。灰常合理吧。

Hibernate:冬眠的意思，也就是数据持久化

POJO是一个纯粹的类，没有实现任何借口，没有继承任何类。移植方便。轻量级框架的特点：没有侵入性。

JPA：java持久化API 这是一个规范。Hibernate实现了JPA规范。

IBatis不是ORM框架，它没有Object到数据库表的映射，它只是把sql语句的结果与Object相对应，只是省略了Resultset等的取值赋值的过程。



Hibernate优点：

提高生产力（编写的连接操作数据库的代码没有了）

是开发更加对象化（阻抗不匹配）

可移植性（不知道数据库的存在的嘛，所以移植方便许多了）

没有侵入性（不需要继承实现任何类和接口），支持透明持久化缺点：

使用数据库特性的语句，将很难调优

它对大批量数据更新有问题（它要缓存大量数据）

系统中存在大量的统计查询功能

使用**Hibernate**步骤：

1.创建JAVA项目

2.创建User Library, 加入以来包

3.配置数据库基本配置 hibernamte.cfg.xml

4.创建实体类 User.java（回头映射成表）

5.提供User.hbm.xml文件，完成实体类的映射

6.将User.hbm.xml文件加入到hibernate.cfg.xml文件中 7.编写工具类ExportDB.java 把我们的实体类生成对应的表，将hbm生成ddl，也就是hbm2ddl

8.建立客户端类Client，添加用户数据到MySQL

最好加入如下配置项，方便观察hibernate sql的生成

<property name="hibernate.show\_sql">true<\/property>

<property name="format\_sql">true<\/property>

最好加入log4j.properties 将该文件拷贝的src下，便于程序的调试。

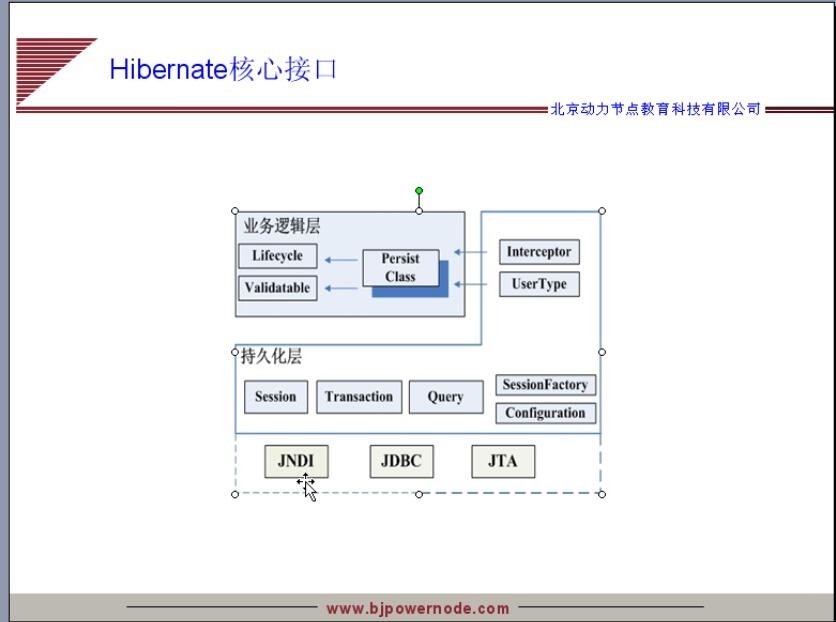
Hibernate开发步骤：

实体类（持久化）类的设计

实体类与关系数据库的映射

应用的开发

PO = POJO+配置文件



JTA：java Transaction API 是一个事务容器

JNDI: Java Naming and Directory Interface,Java命名和目录接口 Datasource

持久化对象的声明周期：

Transient：就是new了一个，还没有进行session.save()，也就是没有进入session，所以随时就给回收了

Persistent：session里有的数据

Detached：数据库里有的数据，但是session里没有了 但是要想操作数据库还得重新获得Session

JUnit可以用来单元测试，自动化测试用assert（）语句来判定预期值与实际值的比对

**junit**简介：（**extents TestCase**类）

* 编写测试XXTest, 需要继承TestCase类
* 方法名必须定义为public void testXXX()
* 最要好采用单独的目录存放测试程序
* 建议使用断言

掌握**Hibernate**的**CRUD**（增删改查）操作

必须掌握**get**和**load**方法的区别？

* get不支持lazy, load在默认情况下是lazy
* get加载数据，如果不存在放回null， 而load返回ObjectNotFoundException异常

Transeint

* 没有被session管理
* 在数据库中 没有与之匹配的记录

Persistent

* 纳入session管理
* 在数据库中有与之匹配的记录
* 当属性发生改变时，在清理缓存（提交事务）时，会自动和数据库同步（脏数据检查）

Detached

* 没有被session管理
* 在数据库中存在与之匹配的记录

**Session**是一个持久化管理器（不仅仅是一个**Connection**，还有缓存区等一系列操作功能）

**Hibernate**缓存

分成一级缓存和二级缓存。一级缓存是Session级别的缓存，只在一个session中好用，如果出了这个session就不好用了，得到的数据（通过load，get或者sql语句得到的数据）都会存储到缓存当中，下一次访问的时候，可以直接从一级缓存中得到。

一级缓存是用来缓存实体对象的，iterator查询普通属性，一级缓存不会缓存，所以会发出查询语句。

一级缓存声明周期很短和session的生命周期一直，一级缓存也叫session级的缓存或事务级缓存

那些方法支持一级缓存：

loac\/get\/iterate查询实体对象

一级缓存是缓存实体对象的 了解大批量数据更新的做法

hibernate二级缓存

二级缓存也成为进程级的缓存或SessionFactory级的缓存，二级缓存可以被所有的session共享

二级缓存的声明周期和SessionFactory的声明周期一致，SessionFactory可以管理二级缓存二级缓存的配置和使用：

将ehcache.xml文件拷贝到src下

在hibernate.cfg.xml文件中加入缓存产品提供商

启用二级缓存，这也是它的默认配置

指定那些实体类使用耳机缓存

可以在映射文件中采用<cache>标签指定或者在hibernate.cfg.xml文件中指定之一使用的策略，通常采用read-only和read-write

缓存原则：通常读远远大于写的数据进行缓存

二级缓存主要是缓存实体对象的 了解一级缓存和二级缓存的交互

注意大批量数据更新时，如果配置了二级缓存，建议禁用一级缓存和二级缓存的交互

**[HTTP](http://www.cnblogs.com/yqin/archive/2010/06/07/1810454.html)**[中的重定向和请求转发的区别](http://www.cnblogs.com/yqin/archive/2010/06/07/1810454.html)

一、调用方式

我 们 知 道 ， 在 servlet 中 调 用 转 发 、 重 定 向 的 语 句 如 下 ： request.getRequestDispatcher("new.jsp").forward(request, response); \/\/转发到new.jsp response.sendRedirect("new.jsp"); \/\/重定向到new.jsp

在jsp页面中你也会看到通过下面的方式实现转发：

<jsp:forward page="apage.jsp" \/>

当然也可以在jsp页面中实现重定向：

<%response.sendRedirect("new.jsp"); %> \/\/重定向到new.jsp

二、本质区别解释一

一句话，转发是服务器行为，重定向是客户端行为。为什么这样说呢，这就要看两个动作的工作流程：

转发过程：客户浏览器发送http请求——》web服务器接受此请求——》调用内部的一个方法在容器内部完成请求处理和转发动作——》将目标资源发送给客户；在这里，转发的路径必须是同一个web容器下的url，其不能转向到其他的web路径上去，中间传递的是自己的容器内的request。在客户浏览器路径栏显示的仍然是其第一次访问的路径，也就是说客户是感觉不到服务器做了转发的。转发行为是浏览器只做了一次访问请求。

重定向过程：客户浏览器发送http请求——》web服务器接受后发送302状态码响应及对应新的location给客户浏览器——》客户浏览器发现是302响应，则自动再发送一个新的http请求， 请求url是新的location地址——》服务器根据此请求寻找资源并发送给客户。在这里location 可以重定向到任意URL，既然是浏览器重新发出了请求，则就没有什么request传递的概念 了。在客户浏览器路径栏显示的是其重定向的路径，客户可以观察到地址的变化的。重定向行为是浏览器做了至少两次的访问请求的。

解释二

重定向，其实是两次request

第一次，客户端request A,服务器响应，并response回来，告诉浏览器，你应该去B。这个时候IE可以看到地址变了，而且历史的回退按钮也亮了。重定向可以访问自己web应用以外的资源。在重定向的过程中，传输的信息会被丢失。

例子：

response.sendRedirect("loginsuccess.jsp");

请求转发是服务器内部把对一个request\/response的处理权，移交给另外一个

对于客户端而言，它只知道自己最早请求的那个A，而不知道中间的B，甚至C、D。传输的信息不会丢失。

例子：

RequestDispatcher dis=request.getRequestDispatcher(“loginsuccess.jsp”); Dis.forward(request,response);

解释三

假设你去办理某个执照

重定向：你先去了A局，A局的人说：“这个事情不归我们管，去B局”，然后，你就从A退了出来，自己乘车去了B局。

转发：你先去了A局，A局看了以后，知道这个事情其实应该B局来管，但是他没有把你退回来，而是让你坐一会儿，自己到后面办公室联系了B的人，让他们办好后，送了过来。

**[HTTP](http://www.cnblogs.com/lxinxuan/archive/2009/10/22/1588053.html)**[状态码大全](http://www.cnblogs.com/lxinxuan/archive/2009/10/22/1588053.html)

完整的 HTTP 1.1规范说明书来自于RFC 2616，你可以

在 [http:\/\/www.talentdigger.cn\/home\/link.php?url=d3d3LnJmYy1lZGl0b3Iub3JnLw%3D%3D](http://www.talentdigger.cn/home/link.php?url=d3d3LnJmYy1lZGl0b3Iub3JnLw%3D%3D) 在线查阅。HTTP 1.1的状态码被标记为新特性，因为许多浏览器只支持 HTTP 1.0。你应只把状态码发送给支持 HTTP 1.1的客户端，支持协议版本可以通过调用request.getRequestProtocol来检查。

本部分余下的内容会详细地介绍 HTTP 1.1中的状态码。这些状态码被分为五大类：

**100-199** 用于指定客户端应相应的某些动作。

**200-299** 用于表示请求成功。

**300-399** 用于已经移动的文件并且常被包含在定位头信息中指定新的地址信息。

**400-499** 用于指出客户端的错误。

**500-599** 用于支持服务器错误。

HttpServletResponse中的常量代表关联不同标准消息的状态码。在servlet程序中，你会更多地用到这些常量的标识来使用状态码。例如：你一般会使用

response.setStatus(response.SC\_NO\_CONTENT)而不是 response.setStatus(204)，因为后者不易理解而且容易导致错误。但是，你应当注意到服务器允许对消息轻微的改变，而客户端只注意状态码的数字值。所以服务器可能只返回 HTTP\/1.1 200 而不是 HTTP\/1.1 200

OK。

1. (Continue\/继续)

如果服务器收到头信息中带有100-continue的请求，这是指客户端询问是否可以在后续的请求中发送附件。在这种情况下，服务器用100(SC\_CONTINUE)允许客户端继续或用417 (Expectation Failed)告诉客户端不同意接受附件。这个状态码是 HTTP 1.1中新加入的。

1. (Switching Protocols\/转换协议)

101 (SC\_SWITCHING\_PROTOCOLS)状态码是指服务器将按照其上的头信息变为一个不同的协议。这是 HTTP 1.1中新加入的。

200 (OK\/正常)

1. (SC\_OK)的意思是一切正常。一般用于相应GET和POST请求。这个状态码对servlet是缺省的；如果没有调用setStatus方法的话，就会得到200。
2. (Created\/已创建)
3. (SC\_CREATED)表示服务器在请求的响应中建立了新文档；应在定位头信息中给出它的

URL。

1. (Accepted\/接受)
2. (SC\_ACCEPTED)告诉客户端请求正在被执行，但还没有处理完。
3. (Non-Authoritative Information\/非官方信息)

状态码203 (SC\_NON\_AUTHORITATIVE\_INFORMATION)是表示文档被正常的返回，但是由于正在使用的是文档副本所以某些响应头信息可能不正确。这是 HTTP 1.1中新加入的。

1. (No Content\/无内容)

在并没有新文档的情况下，204 (SC\_NO\_CONTENT)确保浏览器继续显示先前的文档。这各状态码对于用户周期性的重载某一页非常有用，并且你可以确定先前的页面是否已经更新。例如，某个servlet可能作如下操作：

int pageVersion =Integer.parseInt(request.getParameter("pageVersion")); if (pageVersion >;= currentVersion) { response.setStatus(response.SC\_NO\_CONTENT);

} else {

\/\/ Create regular page

}

但是，这种方法对通过刷新响应头信息或等价的HTML标记自动重载的页面起作用，因为它会返回一个204状态码停止以后的重载。但基于JavaScript脚本的自动重载在这种情况下仍然需要能够起作用。可以阅读本书7.2 ( HTTP 1.1 Response Headers and Their Meaning\/HTTP

1.1响应头信息以及他们的意义)部分的详细讨论。

1. (Reset Content\/重置内容)

重置内容205 (SC\_RESET\_CONTENT)的意思是虽然没有新文档但浏览器要重置文档显示。这个状态码用于强迫浏览器清除表单域。这是 HTTP 1.1中新加入的。

1. (Partial Content\/局部内容)

206 (SC\_PARTIAL\_CONTENT)是在服务器完成了一个包含Range头信息的局部请求时被发送的。这是 HTTP 1.1中新加入的。

300 (Multiple Choices\/多重选择)

1. (SC\_MULTIPLE\_CHOICES)表示被请求的文档可以在多个地方找到，并将在返回的文档中列出来。如果服务器有首选设置，首选项将会被列于定位响应头信息中。
2. (Moved Permanently)
3. (SC\_MOVED\_PERMANENTLY)状态是指所请求的文档在别的地方；文档新的URL会在定位响应头信息中给出。浏览器会自动连接到新的URL。
4. (Found\/找到)

与301有些类似，只是定位头信息中所给的URL应被理解为临时交换地址而不是永久的。注意：在 HTTP 1.0中，消息是临时移动(Moved Temporarily)的而不是被找到，因此

HttpServletResponse中的常量是SC\_MOVED\_TEMPORARILY不是我们以为的

SC\_FOUND。

注意

代表状态码302的常量是SC\_MOVED\_TEMPORARILY而不是SC\_FOUND。

状态码302是非常有用的因为浏览器自动连接在定为响应头信息中给出的新URL。这非常有用，而且为此有一个专门的方法——sendRedirect。使用response.sendRedirect(url)比调用

response.setStatus(response.SC\_MOVED\_TEMPORARILY)和response.setHeader("Location", url)多几个好处。首先，response.sendRedirect(url)方法明显要简单和容易。第二，servlet自动建立一页保存这一连接以提供给那些不能自动转向的浏览 器显示。最后，在servlet 2.2版本（J2EE中的版本）中，sendRedirect能够处理相对路径，自动转换为绝对路径。但是你只能在2.1版本中使用绝对路径。

如果你将用户转向到站点的另一页中，你要用 HttpServletResponse 中的 encodeURL 方法传送URL。这么做可预防不断使用基于URL重写的会话跟踪的情况。URL重写是一种在你的网站跟踪不使用 cookies 的用户的方法。这是通过在每一个URL尾部附加路径信息实现的，但是 servlet 会话跟踪API会自动的注意这些细节。会话跟踪在第九章讨论，并且养成使用

encodeURL 的习惯会使以后添加会话跟踪的功能更容易很多。

核心技巧

如果你将用户转向到你的站点的其他页面，用

response.sendRedirect(response.encodeURL(url)) 的方式事先计划好会话跟踪(session

tracking)要比只是调用 response.sendRedirect(url) 好的多。

这个状态码有时可以与301交换使用。例如，如果你错误的访问了

[http:\/\/www.talentdigger.cn\/home\/link.php?url=aG9zdC9%2BdXNlcg%3D%3D](http://www.talentdigger.cn/home/link.php?url=aG9zdC9%2BdXNlcg%3D%3D)（路径信息不完整），有些服务器就会回复301状态码而有些则回复302。从技术上说，如果最初的请求是GET浏览器只是被假定自动转向。如果想了解更多细节，请看状态码307的讨论。

1. (See Other\/参见其他信息)

这个状态码和 301、302 相似，只是如果最初的请求是 POST，那么新文档（在定位头信息中给出）药用 GET 找回。这个状态码是新加入 HTTP 1.1中的。

1. (Not Modified\/为修正)

当客户端有一个缓存的文档，通过提供一个 If-Modified-Since 头信息可指出客户端只希望文档在指定日期之后有所修改时才会重载此文档，用这种方式可以进行有条件的请求。304

(SC\_NOT\_MODIFIED)是指缓冲的版本已经被更新并且客户端应刷新文档。另外，服务器将返回请求的文档及状态码 200。servlet一般情况下不会直接设置这个状态码。它们会实现

getLastModified方法并根据修正日期让默认服务方法处理有条件的请求。这个方法的例程已在2.8部分(An Example Using Servlet Initialization and Page Modification Dates\/一个使用

servlet初始化和页面修正日期的例子)给出。

1. (Use Proxy\/使用代理)

305 (SC\_USE\_PROXY)表示所请求的文档要通过定位头信息中的代理服务器获得。这个状态码是新加入 HTTP 1.1中的。

307 (Temporary Redirect\/临时重定向)

浏览器处理307状态的规则与302相同。307状态被加入到 HTTP 1.1中是由于许多浏览器在收到302响应时即使是原始消息为POST的情况下仍然执行了错误的转向。只有在收到303响应时才假定浏览器会在POST请求时重定向。添加这个新的状态码的目的很明确：在响应为303 时按照GET和POST请求转向；而在307响应时则按照GET请求转向而不是POST请求。注 意：由于某些原因在HttpServletResponse中还没有与这个状态对应的常量。该状态码是新加入HTTP 1.1中的。

注意

在 HttpServletResponse 中没有 SC\_TEMPORARY\_REDIRECT 常量，所以你只能显示的使用307状态码。

400 (Bad Request\/错误请求)

1. (SC\_BAD\_REQUEST)指出客户端请求中的语法错误。
2. (Unauthorized\/未授权)

401 (SC\_UNAUTHORIZED)表示客户端在授权头信息中没有有效的身份信息时访问受到密码保护的页面。这个响应必须包含一个WWW-Authenticate的授权信息头。例如，在本书4.5部分中的“Restricting Access to Web Pages.\/限制访问Web页。”

403 (Forbidden\/禁止)

403 (SC\_FORBIDDEN)的意思是除非拥有授权否则服务器拒绝提供所请求的资源。这个状态经常会由于服务器上的损坏文件或目录许可而引起。

404 (Not Found\/未找到)

404 (SC\_NOT\_FOUND)状态每个网络程序员可能都遇到过，他告诉客户端所给的地址无法找到任何资源。它是表示“没有所访问页面”的标准方式。这个状态码是常用的响应并且在

HttpServletResponse类中有专门的方法实现它：sendError("message")。相对于setStatus使用sendError得好处是：服务器会自动生成一个错误页来显示错误信息。但是，Internet Explorer 5浏览器却默认忽略你发挥的错误页面并显示其自定义的错误提示页面，虽然微软这么做违反了 HTTP 规范。要关闭此功能，在工具菜单里，选择Internet选项，进入高级标签页，并确认“显示友好的 HTTP 错误信息”选项（在我的浏览器中是倒数第8各选项）没有被选。但是很少有用户知道此选项，因此这个特性被IE5隐藏了起来使用户无法看到你所返回给用户的信息。而其他主流浏览器及IE4都完全的显示服务器生成的错误提示页面。可以参考图6-3及6-4中的例子。

核心警告

默认情况下，IE5忽略服务端生成的错误提示页面。

405 (Method Not Allowed\/方法未允许)

405 (SC\_METHOD\_NOT\_ALLOWED)指出请求方法(GET, POST, HEAD, PUT, DELETE, 等)

对某些特定的资源不允许使用。该状态码是新加入 HTTP 1.1中的。

406 (Not Acceptable\/无法访问)

406 (SC\_NOT\_ACCEPTABLE)表示请求资源的MIME类型与客户端中Accept头信息中指定的类 型 不 一 致 。 见 本 书 7.2 部 分 中 的 表 7.1(HTTP 1.1 Response Headers and Their Meaning\/HTTP 1.1响应头信息以及他们的意义)中对MIME类型的介绍。406是新加入 HTTP

1.1中的。

407 (Proxy Authentication Required\/代理服务器认证要求)

407 (SC\_PROXY\_AUTHENTICATION\_REQUIRED)与401状态有些相似，只是这个状态用于代理服务器。该状态指出客户端必须通过代理服务器的认证。代理服务器返回一个Proxy-

Authenticate响应头信息给客户端，这会引起客户端使用带有Proxy-Authorization请求的头信息重新连接。该状态码是新加入 HTTP 1.1中的。

408 (Request Timeout\/请求超时)

408 (SC\_REQUEST\_TIMEOUT)是指服务端等待客户端发送请求的时间过长。该状态码是新加入 HTTP 1.1中的。

409 (Conflict\/冲突)

该状态通常与PUT请求一同使用，409 (SC\_CONFLICT)状态常被用于试图上传版本不正确的文件时。该状态码是新加入 HTTP 1.1中的。

410 (Gone\/已经不存在)

410 (SC\_GONE)告诉客户端所请求的文档已经不存在并且没有更新的地址。410状态不同于

404，410是在指导文档已被移走的情况下使用，而404则用于未知原因的无法访问。该状态码是新加入 HTTP 1.1中的。

411 (Length Required\/需要数据长度)

411 (SC\_LENGTH\_REQUIRED)表示服务器不能处理请求（假设为带有附件的POST请

求），除非客户端发送Content-Length头信息指出发送给服务器的数据的大小。该状态是新加入 HTTP 1.1的。

412 (Precondition Failed\/先决条件错误)

412 (SC\_PRECONDITION\_FAILED)状态指出请求头信息中的某些先决条件是错误的。该状态是新加入 HTTP 1.1的。

413 (Request Entity Too Large\/请求实体过大)

413 (SC\_REQUEST\_ENTITY\_TOO\_LARGE)告诉客户端现在所请求的文档比服务器现在想要处理的要大。如果服务器认为能够过一段时间处理，则会包含一个Retry-After的响应头信息。该状态是新加入 HTTP 1.1的。

414 (Request URI Too Long\/请求URI过长)

414 (SC\_REQUEST\_URI\_TOO\_LONG)状态用于在URI过长的情况时。这里所指的“URI”是指URL中主机、域名及端口号之后的内容。例如：在URL--http:\/\/www.y2k-

disaster.com:8080\/we\/look\/silly\/now\/中URI是指\/we\/look\/silly\/now\/。该状态是新加入HTTP 1.1的。

415 (Unsupported Media Type\/不支持的媒体格式)

415 (SC\_UNSUPPORTED\_MEDIA\_TYPE)意味着请求所带的附件的格式类型服务器不知道如何处理。该状态是新加入 HTTP 1.1的。

416 (Requested Range Not Satisfiable\/请求范围无法满足)

416表示客户端包含了一个服务器无法满足的Range头信息的请求。该状态是新加入 HTTP

1.1的。奇怪的是，在servlet 2.1版本API的HttpServletResponse中并没有相应的常量代表该状态。

注意

在servlet 2.1的规范中，类HttpServletResponse并没有

SC\_REQUESTED\_RANGE\_NOT\_SATISFIABLE 这样的常量，所以你只能直接使用416。在

servlet 2.2版本之后都包含了此常量。

417 (Expectation Failed\/期望失败)

如果服务器得到一个带有100-continue值的Expect请求头信息，这是指客户端正在询问是否可以在后面的请求中发送附件。在这种情况下，服务器也会用该状态(417)告诉浏览器服务器不接收该附件或用100 (SC\_CONTINUE)状态告诉客户端可以继续发送附件。该状态是新加入HTTP 1.1的。

500 (Internal Server Error\/内部服务器错误)

500 (SC\_INTERNAL\_SERVER\_ERROR) 是常用的“服务器错误”状态。该状态经常由CGI程序引起也可能（但愿不会如此！）由无法正常运行的或返回头信息格式不正确的servlet引 起。

501 (Not Implemented\/未实现)

501 (SC\_NOT\_IMPLEMENTED)状态告诉客户端服务器不支持请求中要求的功能。例如，客户端执行了如PUT这样的服务器并不支持的命令。

502 (Bad Gateway\/错误的网关)

502 (SC\_BAD\_GATEWAY)被用于充当代理或网关的服务器；该状态指出接收服务器接收到远端服务器的错误响应。

503 (Service Unavailable\/服务无法获得)

状态码503 (SC\_SERVICE\_UNAVAILABLE)表示服务器由于在维护或已经超载而无法响应。例如，如果某些线程或数据库连接池已经没有空闲则servlet会返回这个头信息。服务器可提供一个Retry-After头信息告诉客户端什么时候可以在试一次。

504 (Gateway Timeout\/网关超时)

该状态也用于充当代理或网关的服务器；它指出接收服务器没有从远端服务器得到及时的响应。该状态是新加入 HTTP 1.1的。

505 (HTTP Version Not Supported\/不支持的 HTTP 版本)

505 (SC\_HTTP\_VERSION\_NOT\_SUPPORTED)状态码是说服务器并不支持在请求中所标明

HTTP 版本。该状态是新加入 HTTP 1.1的。

Cookie是存在客户端的，有服务器端进行写入，客户端可以拒绝服务器写入。Cookie里存的是Key-Value对。

Session是存在服务器端的内存当中的，当客户端请求是，服务器端根据请求生成

SessionID，通过session.setAttribute(name,value)来生成SessionID。

理解 **Cookie**

Cookie 的作用我想大家都知道，通俗地说就是当一个用户通过 HTTP 协议访问一个服务器的时候，这个服务器会将一些 Key\/Value 键值对返回给客户端浏览器，并给这些数据加上一些限制条件，在条件符合时这个用户下次访问这个服务器的时候，数据又被完整地带回给服务器。

这个作用就像您去超市购物时，第一次给您办张购物卡，这个购物卡里存放了一些您的个人信息，下次您再来这个连锁超市时，超市会识别您的购物卡，下次直接购物就好了。

当初 W3C 在设计 Cookie 时实际上考虑的是为了记录用户在一段时间内访问 Web 应用的行为路径。由于 HTTP 协议是一种无状态协议，当用户的一次访问请求结束后，后端服务器就无法知道下一次来访问的还是不是上次访问的用户，在设计应用程序时，我们很容易想到两次访问是同一人访问与不同的两个人访问对程序设计和性能来说有很大的不同。例如，在一个很短的时间内，如果与用户相关的数据被频繁访问，可以针对这个数据做缓存，这样可以大大提高数据的访问性能。Cookie 的作用正是在此，由于是同一个客户端发出的请求，每次发出的请求都会带有第一次访问时服务端设置的信息，这样服务端就可以根据 Cookie 值来划分访问的用户了。

用response.setCookie()；来由服务器端向客户端存入Cookie。

理解 **Session**

前面已经介绍了 Cookie 可以让服务端程序跟踪每个客户端的访问，但是每次客户端的访问都必须传回这些 Cookie，如果 Cookie 很多，这无形地增加了客户端与服务端的数据传输量， 而 Session 的出现正是为了解决这个问题。

同一个客户端每次和服务端交互时，不需要每次都传回所有的 Cookie 值，而是只要传回一个

ID，这个 ID 是客户端第一次访问服务器的时候生成的，而且每个客户端是唯一的。这样每个客户端就有了一个唯一的 ID，客户端只要传回这个 ID 就行了，这个 ID 通常是 NANE 为

JSESIONID 的一个 Cookie。

##### Session 与 Cookie

下面详细讲一下 Session 如何基于 Cookie 来工作。实际上有三种方式能可以让 Session 正常工作：

基于 URL Path Parameter，默认支持。

基于 Cookie，如果没有修改 Context 容器的 cookies 标识，默认也是支持的。

基于 SSL，默认不支持，只有 connector.getAttribute("SSLEnabled") 为 TRUE 时才支持。

第一种情况下，当浏览器不支持 Cookie 功能时，浏览器会将用户的 SessionCookieName 重写到用户请求的 URL 参数中，它的传递格式如 \/path\/Servlet;name=value;name2=value2?

Name3=value3，其中“Servlet；”后面的 K-V 就是要传递的 Path Parameters，服务器会从这个 Path Parameters 中拿到用户配置的 SessionCookieName。关于这个

SessionCookieName，如果在 web.xml 中配置 session-config 配置项，其 cookie-config 下的

name 属性就是这个 SessionCookieName 值。如果没有配置了 session-config 配置项，默认的 SessionCookieName 就是大家熟悉的“JSESSIONID”。需要说明的一点是，与 Session 关联的 Cookie 与其他 Cookie 没有什么不同。接着 Request 根据这个 SessionCookieName 到

Parameters 中拿到 Session ID 并设置到 request.setRequestedSessionId 中。

请注意，如果客户端也支持 Cookie，Tomcat 仍然会解析 Cookie 中的 Session ID，并会覆盖

URL 中的 Session ID。

如果是第三种情况，将会根据 javax.servlet.request.ssl\_session 属性值设置 Session ID。

**Cookie** 安全问题

虽然 Cookie 和 Session 都可以跟踪客户端的访问记录，但是它们的工作方式显然是不同的， Cookie 通过把所有要保存的数据通过 HTTP 协议的头部从客户端传递到服务端，又从服务端再传回到客户端，所有的数据都存储在客户端的浏览器里，所以这些 Cookie 数据可以被访问到，就像我们前面通过 Firefox 的插件 HttpFox 可以看到所有的 Cookie 值。不仅可以查看

Cookie，甚至可以通过 Firecookie 插件添加、修改 Cookie，所以 Cookie 的安全性受到了很大的挑战。

相比较而言 Session 的安全性要高很多，因为 Session 是将数据保存在服务端，只是通过

Cookie 传递一个 SessionID 而已，所以 Session 更适合存储用户隐私和重要的数据。

总结

Cookie 和 Session 都是为了保持用户访问的连续状态，之所以要保持这种状态，一方面是为了方便业务实现，另一方面就是简化服务端程序设计，提高访问性能，但是这也带来了另外一些挑战，如安全问题、应用的分布式部署带来的 Session 的同步问题及跨域名 Session 的

同步等一系列问题。本章分析了 Cookie 和 Session 的工作原理，并介绍了一致分布式

Session 的解决方案。

**[URL](http://baike.baidu.com/view/1496.htm)**重写

[URL](http://baike.baidu.com/view/1496.htm)重写就是首先获得一个进入的URL请求然后把它重新写成网站可以处理的另一个URL的过程。举个例子来说，如果通过浏览器进来的URL是“UserProfile.aspx?ID=1”那么它可以被重写成 “UserProfile\/1.aspx”，这样的URL，这样的网址可以更好的被网站所阅读。

如果浏览器不支持Cookie或用户阻止了所有Cookie，可以把会话ID附加在HTML页面中所有的URL上，这些页面作为响应发送给客户。这样，当用户单击URL时，会话ID被自动作为请求行的一部分而不是作为头行发送回服务器。这种方法称为URL重写(URL rewriting)。

一般来说，URL重写是支持会话的非常健壮的方法。在不能确定浏览器是否支持Cookie的情况下应该使用这种方法。然而，使用URL重写应该注意下面几点：

1.如果使用URL重写，应该在应用程序的所有页面中，对所有的URL编码，包括所有的超链接和表单的action属性值。

2.应用程序的所有的页面都应该是动态的。因为不同的用户具有不同的会话ID，因此在静态

HTML页面中无法在URL上附加会话ID。

3.所有静态的HTML页面必须通过Servlet运行，在它将页面发送给客户时会重写URL。

#### 重写方法

一、**Tomcat**的过滤器

最典型的就是用 urlReweite的类库。大家爱如果有经验，自己写也不是很麻烦。修改web.xml增加过滤器，然后配置个过滤的规则就可以了

二、使用**Apache**的**mod**

三、使用系统的**404**错误进行也就是页面没找到的错误

总结

第一种修改麻烦，但对于平稳运行的系统，是个不错的选择

第二种需要配置Apache,但 Apache的重启速度很快，适合于有可能经常变得情况，而且Apache Mod的效率还是不错的

第三种，最灵活，我一般简单的应用都是用这个实现的。

**URL**与**URI**的区别

**URI**属于**URL**更低层次的抽象，一种字符串文本标准。

就是说，**URI**属于父类，而**URL**属于**URI**的子类。**URL**是**URI**的一个子集。

在《HTTP权威指南》一书中，对于URI的定义是：统一资源标识符；对于URL的定义是：统一资源定位符。

二者的区别在于，URI表示请求服务器的路径，定义这么一个资源。而URL同时说明要如何访问这个资源（http:\/\/）。

例如，一个URL通常包括三部分： 方案部分（scheme）：http:\/\/

地址部分：CEALER | 一些瞬间、一些回忆、一些经典、一些原创、一些愤怒、一些感动资源部分：\/1.png

而在C#中，URL类属于System.Security.Policy命名空间，Uri属于System。在MSDN对Url类的备注中，能更好的说明Url与Uri的区别：

Url 证据的存在将在授予集内生成 UrlIdentityPermission。如果有对 UrlIdentityPermission 的

Demand，则与 Url 证据对应的 UrlIdentityPermission 将与请求的权限进行比较。

考虑完整的 URL，包括协议（HTTP、HTTPS、FTP）和文件。例如，Microsoft Home Page

就是一个完整的 URL。

URL 可以精确匹配，也可在最后一个位置使用通配符来匹配。例如，Microsoft Home Page\*

就是一个含通配符的 URL。

而Uri类在实例化的时候，可以指定为绝对路径，相对路径，但可以不指定到具体的某个资源。

那么我理解的二者的区别就是：

**URI**可以表示一个域，也可以表示一个资源。

**URL**只能表示一个资源。

同样的，URN（统一资源名称）也是URI的一个子集，目前没有大规模运用。

*PS*：

*URI*是一个字符串格式规范 并没有指定它的用途

*URL*是资源定位的规范 包括网址 *ftp*服务器 文件路径

**HTML**：**Hyper Text Markup Language** 超文本标记语言

超文本 (文字+图片+音视+链接…) 标记语言 (浏览器根据标记显示内容) 来自于SGML(标准通用标记语言)

专注于在Web上传递信息是写给浏览器的语言

**CSS**：**Cascade Style Sheet** 层叠样式表，可以修饰**HTML**

CSS是一种样式表语言，用于为HTML文档定义布局。例如，CSS涉及字体、颜色、边距、高度、宽度、背景图像、高级定位等方面。

用HTML设置红色作为网页的背景色：

<body bgcolor="#FF0000">

用CSS设置红色作为网页的背景色

body {background-color: #FF0000;}

CSS 语法由三部分构成：选择器、属性和值：

selector {property: value}

选择器 (selector) 通常是你希望定义的 HTML 元素或标签，属性 (property) 是你希望改变的属性，并且每个属性都有一个值。属性和值被冒号分开，并由花括号包围，这样就组成了一个完整的样式声明。

为**HTML**文档应用**CSS**，有三种方法可供选择。下面对这三种方法进行了概括。我们建议你对第三种方法（即外部样式表）予以关注。

方法1：行内样式表（style属性）

为HTML应用CSS的一种方法是使用HTML属性style。我们在上例的基础之上，通过行内样式表将页面背景设为红色：

<html>

<head>

<title>例子<\/title>

<\/head>

<body style="background-color: #FF0000;">

<p>这个页面是红色的<\/p>

<\/body>

<\/html>

方法2：内部样式表（style元素）

为HTML应用CSS的另一种方法是采用HTML元素style。比如像这样：

html>

<head>

<title>例子<\/title>

<style type="text\/css">

body {background-color: #FF0000;}

<\/style>

<\/head>

<body>

<p>这个页面是红色的<\/p>

<\/body>

方法3：外部样式表（引用一个样式表文件），推荐此方法

外部样式表就是一个扩展名为css的文本文件。跟其他文件一样，你可以把样式表文件放在

Web服务器上或者本地硬盘 上。例如，样式表文件名为style.css，它通常被存放于名为style 的目录中。style.css文件内容如下：

body {

background-color: #FF0000;

}

如何在一个HTML文档里引用一个外部样式表文件（style.css）

<html>

<head>

<title>我的文档<\/title>

<link rel="stylesheet" type="text\/css" href=“style\/style.css" \/>

<\/head>

<body>

<h1>我的第一个样式表<\/h1>

<\/body>

<\/html>

这一方法可以令你省去许多工作。例如，假设你要修改某网站的所有网页（比方说有100个网页）的背景

<\/html>

颜色与背景主要属性

color

background-color background-image background-repeat background

字体主要属性

font-family font-style font-size font

文本主要属性

text-align

text-decoration letter-spacing

text-transform

**JavaScript** 用来提高网页的互动能力

JavaScript脚本可以方法页面中的三个位置

放在<script><\/script>标签中间，前面的示例就是这样

可以将脚本单独放到一个文件中，向CSS文件一样，推荐此种用法

如该文件的命名为script.js，位于script目录中，那么引用方式为：<script src=“\/script.js”language=“javascript”\/>

将脚本直接放到html元素中，作为属性值

### DOM（Document Object Model）编程

HTML DOM 是一个W3C标准，它是HTML的文本对象模型(Document Object Model for HTML)的缩写。

HTML DOM 定义了一套HTML的标准对象，以及对HTML文本对象访问和操作的标准方法。

所有HTML 元素，连同它们包含的文本和属性，都可以通过DOM来访问。其内容可以被修改或删除，并可以生成新的元素。

HTML DOM 是独立于平台和编程语言的。它可以被任何编程语言所使用，例如Java,

JavaScript, 和 VBScript。

也就是用**Form**这些对象代替**HTML**中的**form**这些标签，然后操作这些对象来操作显示标签。

### AJAX：Asynchronous Javascript And XML

构建网页的一种综合使用JavaScript和XML的技术不是什么划时代的技术

HTML网页的异步传输技术

在等待网页的传输过程中，用户依然可以和系统进行交互页面不用刷新就可以更新内容

合理的运用可以让用户感觉更好更方便但也不要滥用

剑指offer原题及其解题思路。

#### 题目描述**:**二叉树

操作给定的二叉树，将其变换为源二叉树的镜像。输入描述**:**

二叉树的镜像定义：源二叉树

8

/ \

6 10

/ \ / \ 5 7 9 11

镜像二叉树

8

/ \

10 6

/ \ / \ 11 9 7 5

###### 结题思路：

把一个二叉树变为他的镜像，把他的左右孩子交换。然后再对它的左右孩子做同样的操作。注意的问题是条件的终止：当root为null或者root.left和root.right同时为null条件终止。可以用递归和非递归两种

解题代码：

/\*\*

public class TreeNode { int val = 0; TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) { this.val = val;

}

}

\*/

import java.util.\*; public class Solution {

public void Mirror(TreeNode root) {

if(root == null || (root.left==null&&root.right==null)) return;

//递归

/\*TreeNode tmp = root.left; root.left = root.right; root.right = tmp; Mirror(root.left); Mirror(root.right);\*/

//非递归

Stack<TreeNode> stack = new Stack<TreeNode>(); stack.push(root);

while(!stack.empty()){

TreeNode tree = stack.peek(); stack.pop();

if(tree.left!=null || tree.right!=null){ TreeNode tmp = tree.left;

tree.left = tree.right; tree.right = tmp;

}

if(tree.right!=null) stack.push(tree.right);

if(tree.left!=null) stack.push(tree.left);

}

}

}

#### 题目描述：二维数组中查找

在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

###### 解题思路：

这道题从上到下是递增的，从左到右是递增的，我们可以从左下角或者右上角开始，每次根据判断与target的大小，判断下标是向那个方向移动，这样就可以了。

public class Solution {

public boolean Find(int [][] array,int target) { if(array.length == 0)

return false;

int len = array.length; int wid = array[0].length; if( wid == 0)

return false;

int i=len-1,j=0;//从左下角开始移动while(i>=0&&j<wid){

if( array[i][j] > target ){//比target大就向上移动

i--;

}else if( array[i][j] < target){//比target小就向右移动，以便增大j++;

}else{

return true;

}

}

return false;

}

}

#### 题目描述：替换空格

请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

###### 解题思路：

这一题的结题思路可以是这样，新建一个StringBuffer，然后遍历str，如果str.charAt(index)为空格就替换，这样即可。代码如下：

public class Solution {

public String replaceSpace(StringBuffer str) { StringBuffer tmp = new StringBuffer(); for(int i=0;i<str.length();i++){

if(str.charAt(i)!=' '){ tmp.append(str.charAt(i));

}else{

tmp.append("%20");

}

}

String ret = tmp.toString(); return ret;

}

}

除了此中解法以外，还可以从后向前进行遍历，只要先遍历一遍发现有多少个空格，然后为

StringBuffer扩容，然后开始从右网左移动元素。

#### 题目描述：从尾到头打印链表

输入一个链表，从尾到头打印链表每个节点的值。输入描述**:**

输入为链表的表头

输出描述**:**

输出为需要打印的“新链表”的表头

###### 解题思路：

这个题第一瞬间的想法就是用一个栈来保存，然后再把栈中的数据放入到要返回的ArrayList 中，我们为了方便，直接用一个ArrayList就行，可以完成栈的操作，先添加，后从两边交换数据。代码如下：

import java.util.ArrayList; import java.util.\*;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

ArrayList<Integer> arr = new ArrayList<Integer>(); if(listNode == null)

return arr; while(listNode!=null){

arr.add(listNode.val); listNode = listNode.next;

}

int i=0,j=arr.size()-1; while(i<j){

int tmp = arr.get(i); arr.set(i,arr.get(j)); arr.set(j,tmp);

i++; j--;

}

return arr;

}

}

}

ListNode(int val) { this.val = val;

}

public class ListNode { int val;

ListNode next = null;

/\*\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*/

#### 题目描述：重建二叉树

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列

{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

###### 解题思路：

根据前序遍历可以找到根节点，然后以根节点为index找到在中序遍历中的位置，然后index的下标就是左右子树的分界点，然后对左子树preLeft和inLeft递归，右子树preRight和inRight递归。

/\*\*

* Definition for binary tree
* public class TreeNode {
* int val;
* TreeNode left;
* TreeNode right;
* TreeNode(int x) { val = x; }

\* }

\*/

import java.util.\*; public class Solution {

public static int[] getArr(int[] arr, int begin, int end){ if( end < begin)

return null;

int[] array = new int[end-begin+1]; for(int i=begin,j=0; i<=end; i++){

array[j] = arr[i]; j++;

}

return array;

}

public TreeNode reConstructBinaryTree(int [] pre,int [] in) { if(pre.length == 0 || in.length == 0)

return null;

TreeNode root = new TreeNode(pre[0]); int index = 0; while(in[index]!=pre[0]){

index++;

}

int[] preLeft = getArr(pre, 1, index );

int[] preRight = getArr(pre, index+1, pre.length-1); int[] inLeft = getArr(in, 0,index-1);

int[] inRight = getArr(in, index+1, in.length-1); if(preLeft != null){

root.left = reConstructBinaryTree(preLeft, inLeft);

}

if(preRight!=null){

root.right = reConstructBinaryTree(preRight, inRight);

}

return root;

}

}

#### 题目描述：用两个栈实现队列

用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

###### 解题思路：

stack1负责存储，stack2负责pop()操作时的中间容器，每次pop()操作都是把stack1数据都放到stack2中，然后pop()一个数据，最后再把数据拿回到stack1中。代码如下：

import java.util.Stack;

public class Solution {

Stack<Integer> stack1 = new Stack<Integer>(); Stack<Integer> stack2 = new Stack<Integer>();

public void push(int node) { stack1.push(node);

}

public int pop() { while(!stack1.empty()){

stack2.push(stack1.pop());

}

int ret = stack2.pop(); while(!stack2.empty()){

stack1.push(stack2.pop());

}

return ret;

}

}

#### 题目描述：旋转数组最小数字

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。

例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

###### 解题思路：

由于是有序的，则中间两个有序的交界点就是最小值，可以用二分查找嘛。一下是代码：

import java.util.ArrayList; public class Solution {

public int minNumberInRotateArray(int [] array) { if(array.length == 0)

return 0; int begin = 0;

int end = array.length-1; int mid = (begin+end)/2; while(end - begin>1){

if(array[mid] > array[end]){ begin = mid;

}else{

end = mid;

}

mid = (begin+end)/2;

}

return Math.min(array[begin],array[end]);

}

}

#### 题目描述：斐波那契数列

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项。

n<=39

###### 解题思路：

没什么说的，Fibonacci(n) = Fibonacci(n-1)+Fibnacci(n-2)。使用数组保存结果，自底向上求

n。

public class Solution {

public int Fibonacci(int n) { if(n==0)

return 0; if(n==1)

return 1; if(n==2)

return 1;

int[] arr = new int[n+1]; arr[1] = 1;

arr[2] = 1;

for(int i=3;i<=n;i++){

arr[i] = arr[i-1]+arr[i-2];

}

return arr[n];

}

}

#### 题目描述：跳台阶

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

###### 解题思路：

根据提议，第n级台阶可以有两种方式跳上去，从n-1级跳上去，或者从n-2级跳上去，所以也是Fibonacci数列。代码如下：

public class Solution {

public int JumpFloor(int target) { int a=1,b=2,c=0;

if(target <= 0)

return 0;

if(target ==1 || target ==2) return target;

for(int i=3;i<=target;i++){ c = a+b;

a = b; b = c;

}

return c;

}

}

#### 题目描述：变态跳台阶

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n 级的台阶总共有多少种跳法。

###### 解题思路：

还是跟跳台阶一样，之前是Fibonacci(i) = Fibonacci(i-1)+Fibonacci(i-2)。现在变为Fibonacci(i)=Fibonacci(i-1)+......+Fibonacci(1);

以下是代码：

public class Solution {

public int JumpFloorII(int target) { if(target <= 0 )

return 0;

if(target == 1 || target ==2) return target;

int[] arr = new int[target+1]; arr[1] = 1;

arr[2] = 2;

for(int i=3;i<=target;i++){ int tmp = 0;

for(int j=1;j<i;j++){ tmp += arr[j];

}

tmp++; arr[i] = tmp;

}

return arr[target];

}

}

#### 题目描述：矩形覆盖

我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？

###### 解题思路：

又是一个斐波那契题目，这个题面积必须是2\*n的，所以要是先竖着放一个，那么剩下的n-1 个长方形就是接下来2\*n-1面的解f(n-1)，但是如果这个长方形横着放，就得拿一个把下面的

1\*2的窟窿堵上，然后剩下n-2个解，f(n-2)，f(n) = f(n-1)+f(n-2)代码如下：

public class Solution {

public int RectCover(int target) { if(target <= 0)

return 0;

if(target == 1 || target == 2){ return target;

}

int a=1,b=2,c=0;

for(int i=3;i<=target;i++){ c = a+b;

a = b; b = c;

}

return c;

}

}

#### 题目描述：二进制中一的个数

输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。

###### 解题思路：

1.最直观的想法就是n%2看是否得0，当然也可以用n&1，然后如果得1，count++，否则继续循环。这种方法比较坑爹的是如果输入时负数的话，右移过后由于要保持负数的属性，最高位一定会置1，因此最后就编程了0XFFFFFFFF的死循环的，所以要用第二种方法。代码如下：

public class Solution { public int NumberOf1(int n) {

int count = 0; while(n!=0){

if(n&1) count++;

n= n >> 1;

}

return count;

}

}

2.第二种方法我们从第一种方法的反方向入手，第一种方法是向右移动n，然后与1进行&操作，我们可以变换一下思维，让1向左移，与之前效果一样，同时不会改n，非常好。代码如下：

public class Solution { public int NumberOf1(int n) {

int count = 0; int flag = 1;

while(n!=0){ if(n&flag)

count++;

flag = flag << 1;

}

return count;

}

}

3.第三种方法是更快的算法，我们知道一个整数如果不是0的话，最少有一位是1，我们可以这样想，最右边的1，再往右一定都是0，这个数字减1，过后最右边的这个1变为0，它右边所有的0都变成了1，这样这两个数一进行与操作，便为n减掉了一个1。知道n为零位置。代码如下：

public class Solution { public int NumberOf1(int n) {

int count = 0; while(n!=0){ count++;

n = n&(n-1);

}

return count;

}

}

#### 题目描述：数值的整数次方

给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

###### 解题思路：

这个题的最关键点就是要考虑特殊情况，比如exponent为0，为负数，为整数的三种情况，就可以了。答题时一定要注意全面思考特殊情况。代码如下：

public class Solution {

public double Power(double base, int exponent) { if(base == 0)

return 0;

if(exponent == 0)

return 1; double result = 1.0; if(exponent>0){

for(int i=0;i<exponent;i++){ result \*= base;

}

}else{

for(int i=0;i<(-1\*exponent);i++){ result \*=base;

}

result = 1/result;

}

return result;

}

}

#### 题目描述：调整数组顺序使奇数位于偶数前面

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

###### 解题思路：

可以向一个冒泡排序一样，当遇到相邻的两个数为左偶数又奇数的情况下就交换位置，这样冒泡排序的每轮都会把最右边的在偶数后面的奇数向前调整一位，依旧是最后一个数字肯定是偶数。第二轮倒数第二个是偶数。就酱紫。

public class Solution {

public void reOrderArray(int [] array) { for(int i=0;i<array.length;i++){

for(int j=0;j<array.length-i-1;j++){ if(array[j]%2==0 && array[j+1]%2!=0){

int tmp = array[j]; array[j] = array[j+1]; array[j+1] = tmp;

}

}

}

}

}

还有一个想法就是，如果顺序可以变的话，我们可以向快速排序一样，从两端开始相中间找到左端第一个偶数，右端第一个奇数，然后交换位置。

#### 题目描述：链表倒数第**K**个节点

输入一个链表，输出该链表中倒数第k个结点。

###### 解题思路：

这个题就是先拿一个指针，指向头结点，然后向后面跑K个节点，记住是K个，然后头结点跟着这个节点一起跑，知道这个指针指向空节点，这时候head节点指向的就是倒数第K个节点了。

/\*

public class ListNode { int val;

ListNode next = null;

ListNode(int val) { this.val = val;

}

}\*/

public class Solution {

public ListNode FindKthToTail(ListNode head,int k) { if(head == null || k<=0)

return null; ListNode node = head; for(int i=0;i<k;i++){

if(node == null) return null;

node = node.next;

}

while(node!=null){ node = node.next; head = head.next;

}

return head;

}

}

#### 题目描述：合并两个排序链表

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

解题思路：

就是归并排序的原理，取个相对较小值，然后放到一个链表中，把这个小节点排除，然后继续重复这个过程。这个题注意保存头结点。代码如下：

/\*

public class ListNode { int val;

ListNode next = null;

ListNode(int val) { this.val = val;

}

}\*/

public class Solution {

public static ListNode min(ListNode list1, ListNode list2){ return list1.val<list2.val?list1:list2;

}

public ListNode Merge(ListNode list1,ListNode list2) { if(list1 == null)

return list2; if(list2 == null) return list1;

ListNode list = null;

ListNode head = null;//记录头结点if(list1.val < list2.val){

list = list1; list1 = list1.next;

}else{

list = list2; list2 = list2.next;

}

head = list;

while(list1!=null&& list2!=null){

list.next = min(list1,list2);//取较小节点list = list.next;//list后移

if(list == list1) list1 = list1.next;

else

list2 = list2.next;

}

if(list1 == null){ list.next = list2;

}else{

list.next = list1;

}

return head;

}

}

#### 题目描述：树的子结构

输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

###### 解题思路：

判断一个树A是不是另一个树B的子树，过程应该就是遍历整个B树，如果A树节点的value与B 节点相同，就对他进行进一步判断，如果不同，遍历过程为递归对其左子树和右子树进行判断。具体代码如下：

/\*\*

public class TreeNode { int val = 0; TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) { this.val = val;

}

}

\*/

public class Solution {

public static boolean IsSubTree(TreeNode root1, TreeNode root2){ if(root2 == null)//必须先判断pRoot2

return true; if(root1 == null)

return false;

if( root1.val != root2.val ) return false;

return IsSubTree(root1.left, root2.left)&&IsSubTree(root1.right, root2.right);

}

public boolean HasSubtree(TreeNode root1,TreeNode root2) { boolean result = false;

if(root1 != null && root2 != null ){

if(root1.val == root2.val)//判断根节点是否相等，如果相等进一步判断他们的子树是否也

相等

result = IsSubTree(root1,root2);

if(!result)//如果之前根节点不相等，继续遍历整个树

result = (HasSubtree(root1.left, root2)||HasSubtree(root1.right, root2

));

}

return result;

}

}

#### 题目描述：顺时针打印矩阵

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字

1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

###### 解题思路：

定义上下左右四个边界，然后从左到右，从上到下，从右到左，从下到上。这个顺序，然后依次改变上下左右边界，继续执行，然后在判断循环是否打印是需要注意，让上边界一直小于等于下边界，左边界一直小于等于右边界，这样可以解决边界情况。

代码如下：

import java.util.ArrayList; public class Solution {

public ArrayList<Integer> printMatrix(int [][] matrix) { if(matrix.length==0)

return null; if(matrix[0].length==0)

return null;

ArrayList<Integer> retArr = new ArrayList<Integer>();

int left = 0, right = matrix[0].length-1;//定义上下左右四个边界int up = 0,bottom = matrix.length-1;

while(left<=right&&up<=bottom){

for(int i=left;i<=right&&up<=bottom;i++){//一定要控制边界情况retArr.add(matrix[up][i]);

}

up++;//循环完立刻更改值

for(int i=up;i<=bottom&&left<=right;i++){ retArr.add(matrix[i][right]);

}

right--;

for(int i=right;i>=left&&up<=bottom;i--){ retArr.add(matrix[bottom][i]);

}

bottom--;

for(int i=bottom;i>=up&&left<=right;i--){ retArr.add(matrix[i][left]);

}

left++;

}

return retArr;

}

}

题目描述：包含**min**函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的min函数。

###### 解题思路：

用ArrayList的相关操作实现。代码如下：

import java.util.Stack; import java.util.\*;

public class Solution {

ArrayList<Integer> array = new ArrayList<Integer>(); public void push(int node) {

array.add(node);

}

public void pop() { if(array.size()==0)

return; array.remove(array.size()-1);

}

public int top() { if(array.size()==0){

return -1;

}

return array.get(array.size()-1);

}

public int min() { if(array.size() == 0)

return -1;

int minimum = array.get(0); for(int i=1;i<array.size();i++){

if(minimum>array.get(i)) minimum = array.get(i);

}

return minimum;

}

}

#### 题目描述：栈的压入与弹出序列

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4，5,3,2,1 是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意： 这两个序列的长度是相等的）

解题思路：

用一个栈来存储数据，将pushA数据一次压入栈，如果栈顶的值与popA当前位置的值相等， 则弹出栈。最后如果栈为空则OK。

import java.util.ArrayList; import java.util.Stack; public class Solution {

public boolean IsPopOrder(int [] pushA,int [] popA) { Stack<Integer> stack = new Stack<Integer>();

int pushLen = pushA.length; int popLen = popA.length; if(pushLen == 0 || popLen == 0)

return false; int popIndex = 0;

for(int i=0;i<popLen;i++){ stack.push(pushA[i]);

while(popIndex<popLen && stack.peek()==popA[popIndex]){ stack.pop();

popIndex++;

}

}

return stack.empty()?true:false;

}

}

#### 题目描述：从上到下二叉树遍历

从上往下打印出二叉树的每个节点，同层节点从左至右打印。解题思路：

就是一个广度优先遍历，我们借用一个队列的数据结构实现的。非递归形式。代码如下：

import java.util.ArrayList;

/\*\*

public class TreeNode { int val = 0; TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) { this.val = val;

}

}

\*/

public class Solution {

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) { ArrayList<Integer> ret = new ArrayList<Integer>(); ArrayList<TreeNode> queue = new ArrayList<TreeNode>(); if(root == null)

return ret; queue.add(root); while(!queue.isEmpty()){

if(queue.get(0).left!=null){ queue.add(queue.get(0).left);

}

if(queue.get(0).right!=null){ queue.add(queue.get(0).right);

}

ret.add(queue.get(0).val); queue.remove(0);

}

return ret;

}

}

#### 题目描述：二叉搜索树的后续遍历序列

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。

###### 解题思路：

把数组分成左右两个子树序列，然后递归判断两个序列。判断条件用两个子树的长度是否与整个长度减1相等。代码如下：

public class Solution {

public boolean VerifySquenceOfBST(int [] sequence) {

if(sequence.length == 0)//其实如果是空的话应该返回true的，这样就可以省略掉倒数那几行的判

断了

return false;

int root = sequence[sequence.length-1]; int i=0;

for(;sequence[i]<root;i++){} int[] smaller = new int[i]; for(i=0;i<smaller.length;i++){

smaller[i] = sequence[i];

}

int j=0; for(;sequence[i]>root;i++){j++;} int[] bigger = new int[j];

i = smaller.length; for(j=0;j<bigger.length;j++){

bigger[j] = sequence[i+j];

}

boolean flag = false; if(bigger.length+smaller.length == sequence.length-1)

flag = true;

if(bigger.length==0 && smaller.length!=0) return flag&&VerifySquenceOfBST(smaller);

if(bigger.length!=0 && smaller.length==0) return flag&&VerifySquenceOfBST(bigger);

if(bigger.length==0 && smaller.length==0) return flag;

return flag&&VerifySquenceOfBST(smaller)&&VerifySquenceOfBST(bigger);

}

}

#### 题目描述：二叉树中和为某一值额路径

输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

解题思路：

这一道题的首先读懂题意，注意最后一句，路径的定义是从根节点到叶子节点，而不是从根节点到任意节点。读懂题意过后我们想到的是进行深度遍历中的先序遍历，但是还不一样， 如果只是现需遍历的话只需要把值输出来即可，而这一题目中需要保存路径，还要计算和的值，所以当抛弃一条路径的时候需要把该节点从路径中除去才可以，因此，代码如下：

import java.util.ArrayList;

/\*\*

public class TreeNode { int val = 0; TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) { this.val = val;

}

}

\*/

import java.util.Stack; import java.util.ArrayList; public class Solution {

public static void dfs(TreeNode root, int target, ArrayList<ArrayList<Integer>> re tArr, ArrayList<Integer> arr){

if(root == null) return;

arr.add(root.val);

//如果当前的target值与当前节点的value相等，则把路径添加进来，这里注意的是一定要申请一个新的

list对象，然后把这个对象添加

//retArr中，否则添加到最后添加的都是arr，而arr最后肯定会清空的

if(root.left == null && root.right == null && target == root.val){ ArrayList<Integer> list = new ArrayList<Integer>(arr); retArr.add(list);

}

if(root.left!=null)//其实这里也可以不用target-root.val做参数，添加一个currentSum 即可，只不过多一个参数而已

dfs(root.left, target-root.val, retArr,arr); if(root.right!=null)

dfs(root.right, target-root.val, retArr, arr); arr.remove(arr.size()-1);

}

public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {

ArrayList<ArrayList<Integer>> retArr = new ArrayList<ArrayList<Integer>>(); ArrayList<Integer> arr = new ArrayList<Integer>();

if(root == null){ return retArr;

}

dfs(root, target, retArr, arr); return retArr;

}

}

#### 题目描述**:**复杂链表的复制

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

解题思路：

复杂链表的复制最开始的想法是先把整个链表复制下来，然后根据原链表节点到random节点走的步数决定新链表走的补数，这样是O(n^2)的解法。现在可以用另一个方法，就是把新建的节点恰好接在同一值的老节点的后面，这样next.random = random.next。具体代码如下：

/\*

public class RandomListNode { int label;

RandomListNode next = null; RandomListNode random = null;

RandomListNode(int label) { this.label = label;

}

}

\*/

public class Solution {

public RandomListNode Clone(RandomListNode pHead)

{

if(pHead == null) return null;

RandomListNode current = pHead;

//创建节点 使新建节点恰好在老节点后面

while(current!=null){

RandomListNode node = new RandomListNode(current.label); node.next = current.next;

current.next = node; current = node.next;

}

current = pHead;

//初始化新建节点的random指针while(current!=null){

if(current.random != null)//注意这个地方，前面节点的random可能是null，所以要确定current.random不为null

current.next.random = current.random.next;

current = current.next.next;

}

//打开链表，分成两个current = pHead;

RandomListNode newHead = current.next; RandomListNode tmp = newHead; while(current!=null){

current.next = tmp.next; if(tmp.next == null){

break;

}

tmp.next = current.next.next; current = current.next;

tmp = tmp.next;

}

return newHead;

}

}

#### 题目描述：二叉搜索树与双向链表

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

解题思路：

首先要知道，要形成一个排序的链表，首先，二叉搜索树的中序遍历就是一个排序的，这个题只不过是改成双向链表的了。我们可以通过递归的方式来实现，把递归的左子树看成一个整体，右子树看成一个整体，那么左子树的递归结果就在最左端了，右子树的递归结果就在最右端，所以，把左子树移动到最大的右端，然后令他的p.right=root，root.left=p即可，右子树字节赋值即可，根本不用移动。代码如下：

/\*\*

public class TreeNode { int val = 0; TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) { this.val = val;

}

}

\*/

public class Solution {

public TreeNode Convert(TreeNode pRootOfTree) { if(pRootOfTree == null)

return null;

if(pRootOfTree.left == null && pRootOfTree.right == null){ return pRootOfTree;

}

//将左子树构建成双链表，返回头结点

TreeNode pLeft = Convert(pRootOfTree.left); TreeNode p = pLeft;

//移动到最右端最大的节点

while(p!=null&&p.right!=null){ p = p.right;

}

//将最右端节点与root节点进行连接if(pLeft!=null){

p.right = pRootOfTree; pRootOfTree.left = p;

}

//将右子树构建成双链表，返回头结点

TreeNode pRight = Convert(pRootOfTree.right); if(pRight!=null){

pRootOfTree.right = pRight; pRight.left = pRootOfTree;

}

return (pLeft!=null)?pLeft:pRootOfTree;

}

}

还有一种方法是非递归方法，就是模拟二叉树的非递归中序遍历，就是借用栈这一数据结 构，先把节点一直压入栈中，知道遇到最左侧节点为止，然后把栈顶的元素pop()出来，也就是最左端的左子树，要用一个pre节点保存之前的节点，然后与当前节点进行互相左右子树赋值。代码如下：

/\*\*

public class TreeNode { int val = 0; TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) { this.val = val;

}

}

\*/

import java.util.\*; public class Solution {

public TreeNode Convert(TreeNode pRootOfTree) { if(pRootOfTree == null)

return null;

if(pRootOfTree.left == null && pRootOfTree.right == null){ return pRootOfTree;

}

//就是中序遍历后修改一下前面节点与后面节点的指针指向Stack<TreeNode> stack = new Stack<TreeNode>(); TreeNode p = pRootOfTree;

TreeNode pre = null; boolean isFirst = true;

while(!stack.empty() || p!=null){ while(p!=null){

stack.push(p); p = p.left;

}

p = stack.pop(); if(isFirst){

pRootOfTree = p; pre = p; isFirst = false;

}else{

pre.right = p; p.left = pre; pre = p;

}

p = p.right;

}

return pRootOfTree;

}

}

#### 题目描述 **:**数组中出现次数超过一半的数字

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9 的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出

2。如果不存在则输出0。

###### 解题思路：

这个题可以用HashMap来解决，key是数字，Value是次数。但是这应该不是这个题目的意 思，这里用一个candidate变量，遍历一遍就可以了，如果当前的值与candidate相等则次数加

1，如果一个数字超过一般的话，最后剩的肯定是这个数字。具体代码如下：

public class Solution {

public int MoreThanHalfNum\_Solution(int [] array) { if( array.length == 0 )

return 0;

int candidate = array[0]; int nTimes = 0;

for(int i=1;i<array.length;i++){ if(nTimes == 0){

candidate= array[i]; nTimes++;

}else{

if(candidate == array[i]){ nTimes++;

}else{

nTimes--;

}

}

}

int count = 0;

for(int i=0;i<array.length;i++){ if(array[i] == candidate)

count++;

}

if(count>array.length/2){ return candidate;

}

return 0;

}

}

#### 题目描述：最小的**K**个数

输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。

###### 解题思路：

最基本的思路就是用一个最大堆，堆顶是最大元素，然后添加K个元素，如果下一个元素比K 大，就把最大的顶删除，加入这个元素。具体代码如下：这个是用ArrayList实现的。

import java.util.\*; public class Solution {

public static int findMax(ArrayList<Integer> array){ int max = array.get(0);

for(int i=1;i<array.size();i++){ if(max < array.get(i)){

max = array.get(i);

}

}

return max;

}

public ArrayList<Integer> GetLeastNumbers\_Solution(int [] input, int k) { ArrayList<Integer> retArr = new ArrayList<Integer>();

if(k > input.length || k == 0) return retArr;

for(int i=0;i<input.length;i++){ if(retArr.size()<k){

retArr.add(input[i]);

}else{

int max = findMax(retArr); if(input[i] < max){

retArr.set(retArr.indexOf(max), input[i]);

}

}

}

return retArr;

}

}

#### 题目描述：连续子数组最大和

HZ偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:

{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为8(从第0个开始,到第3个为止)。你会不会被他忽悠住？

###### 解题思路：

这个是一个典型的动态规划题，这个题中的状态函数是以这个数字开头的连续子向量的最大值sum(i)（当然我们也可以以这个数字结尾的最大值，结果是一样的）。我们可以从后往前来看，最后一个肯定是他的本身，然后sum(i) = max(sum(i+1)+h(i), h(i))。所以，代码如下：

public class Solution {

public int FindGreatestSumOfSubArray(int[] array) { if(array.length == 0)

return 0;

int[] sum = new int[array.length]; sum[sum.length-1] = array[array.length-1]; for(int i=sum.length-2;i>=0;i--){

sum[i] = sum[i+1]>0?array[i]+sum[i+1]:array[i];

}

int max = sum[0];

for(int i=0;i<sum.length;i++){ if(max < sum[i])

max = sum[i];

}

return max;

}

}

#### 题目描述**:** 整数中**1**出现的次数（从**1**到**n**整数中**1**出现的次数）

求出1~13的整数中1出现的次数,并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、12、13因此共出现6次,但是对于后面问题他就没辙了。ACMer希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中1出现的次数。

###### 解题思路：

最直观的想法就是从1到n遍历一遍，算出每个数字中包含的1的个数，然后加和即可。代码如下：

public class Solution {

public static int ContainNum(int i){ int count = 0;

while(i!=0){

if(i%10 == 1){

count++;

}

i = i/10;

}

return count;

}

public int NumberOf1Between1AndN\_Solution(int n) { int sum = 0;

for(int i=1;i<=n;i++){//注意这里一定要从1开始

sum += ContainNum(i);

}

return sum;

}

}

然而上面这种算法的时间复杂度是很高的，应该是O(n\*k)，K是指数字的位数。同样也可以用这样的方法，通过观察数字的规律，发现答案的。我们可以把一个数字分成当前的位，和当前位所隔开的高几位和低几位。

如果对1到n每个数都计算其1出现的个数，时间复杂度太高了。这道题需要利用数字特点和规律，计算每一位上1出现的次数：

例如百位上1出现次数，数值n在百位上的值是curNum则：

if(curNum==0)

1出现的次数等于比百位更高位数\*100。例如n=1023，高位数就是1，百位上出现1的次数是1\*100；

if(curNum==1)

1出现的次数等于比百位更高位数\*100，再加上低位上的数，再加1。例如n=1123，高位数就是1，低位数是23

，百位上出现1的次数是1\*100+23+1；

if(curNum>1)

1出现的次数等于比百位更（高位数+1）\*100,例如n=1223，高位数就是1，次数百位上出现1的次数是（1+1

）\*100；具体解析代码如下：

public class Solution {

public int NumberOf1Between1AndN\_Solution(int n) { int count = 0;

int curNum = 0; int lowNum = 0; int highNum = 0; int factor = 1; while(n/factor!=0){

// 12345 变为1234再取余10,也就是把当前位置于个位，然后取余10就可以了

curNum = (n/factor)%10; lowNum = n%factor; highNum = n/(factor\*10);

//如果当前位数字为0的话，那么他完全由高位的数字决定

//比如12013，百位数字为1，那么百位上数字为1的仅仅是12\*100个

if(curNum == 0){

count += factor\*highNum;

}else if(curNum == 1){

count += factor\*highNum + lowNum + 1;//这个1是100,1000整的时候

}else{

count += factor\*(highNum+1);//这个就不用加1了，因为在1\*factor中包括了100,10

00的情况

}

factor \*= 10;

}

return count;

}

}

#### 题目描述：把数组排成最小的数

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3，32，321}，则打印出这三个数字能排成的最小数字为

321323。

解题思路：

这个题我先想用的是分治法，拿出数组中的第一个，与接下来的所有子数组的两个排列进行比较，返回较小的一个。每次交换数组中的一个数字，完成上面的操作。代码如下：

import java.util.ArrayList;

public class Solution {

public String cmp(int [] numbers) { String ret = null;

if( numbers.length == 0 ) return null;

if( numbers.length == 1)

return String.valueOf(numbers[0]); int[] tmp = new int[numbers.length-1]; for(int i=0;i<tmp.length;i++){

tmp[i] = numbers[i+1];

}

String str1 = PrintMinNumber(tmp)+String.valueOf(numbers[0]); String str2 = String.valueOf(numbers[0])+PrintMinNumber(tmp); return str1.compareTo(str2)>0?str2:str1;

}

public String PrintMinNumber(int [] numbers) {

if( numbers.length == 0 ) return "";

if( numbers.length == 1)

return String.valueOf(numbers[0]);

//先找到把第一个数字拿出来的最小值String min = cmp(numbers); for(int i=0;i<numbers.length;i++){

swap(numbers, 0 , i); String tmp = cmp(numbers); if(tmp == null){

continue;

}

if(min.compareTo(tmp) > 0) min = tmp;

}

return min;

}

public void swap(int[] array, int i, int j){ int tmp = array[i];

array[i] = array[j]; array[j] = tmp;

}

}

#### 题目描述：丑数

把只包含因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含因子7。 习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第N个丑数。

###### 解题思路：

这个题的第一瞬间是从1一个一个数网上增加判断一个数是否为丑数，如果是count++，如果不是，跳过。判断丑数的标准是，用他们分别取余2,3,5，然后除以2,3,5。如果最后得0，则为丑数。但是这种方法不好，因为每个数字都要判断，因此可以借用一个数组来完成。我们用一个数组来存储已经数好的丑数，从前面的数组中找到分别与2,3,5，相乘刚好大于数组中最大的丑数，然后比较这三个数值中最小的一个，就是下一个丑数。具体代码如下：

import java.util.\*; public class Solution {

//从当前的数组中找到乘以2,3,5,中刚好大于当前数组中最大的那个值，然后比较这三个数中最小的一个

public int GetUglyNumber\_Solution(int index) { if( index <= 3 )

return index;

ArrayList<Integer> array = new ArrayList<Integer>(); array.add(1);

array.add(2);

array.add(3);

int two = 0, three = 0, five = 0; for(int i=4;i<=index;i++){

for(int j=0;j<array.size();j++){

if(array.get(j)\*2 > array.get(array.size()-1)){ two = array.get(j)\*2;

break;

}

}

for(int j=0;j<array.size();j++){

if(array.get(j)\*3 > array.get(array.size()-1)){ three = array.get(j)\*3;

break;

}

}

for(int j=0;j<array.size();j++){

if(array.get(j)\*5 > array.get(array.size()-1)){ five = array.get(j)\*5;

break;

}

}

array.add(min(two,three,five));

}

return array.get(array.size()-1);

}

public int min(int a, int b, int c){ int tmp = (a<b?a:b);

return tmp < c? tmp:c;

}

}

#### 题目描述

找出字符串中第一个只出现一次的字符详细描述：

接口说明原型：

bool FindChar(char\* pInputString, char\* pChar);

输入参数：

char\* pInputString：字符串

输出参数（指针指向的内存区域保证有效）：

char\* pChar：第一个只出现一次的字符如果无此字符 请输出'.'

输入描述**:**

输入一串字符,由小写字母组成

输出描述**:**

输出一个字符

输入例子**:**

asdfasdfo

输出例子**:**

o

###### 解题思路：

用一个hash来实现，key是字符，value是次数，自己写一个hash。代码如下：

import java.util.Scanner; import java.util.\*; public class Main{

public static char find(String str){ char ret = ' ';

if(str.length() == 0) return ret;

int[] times = new int[26]; for(int i=0;i<str.length();i++){

times[str.charAt(i)-'a']++;

}

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

if( times[str.charAt(i)-'a'] == 1) return str.charAt(i);

}

return ret;

}

public static void main(String[] args){

Scanner scan = new Scanner(System.in); while(scan.hasNext()){

String str = scan.next(); System.out.println(Main.find(str));

}

}

}

#### 题目描述：数组中的逆序对

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P。并将P对1000000007取模的结果输出。 即输出P%1000000007

输入描述**:**

对于%75的数据,size<

对于%50的数据,size<=10^4

题目保证输入的数组中没有的相同的数字数据范围：

=10^5

对于%100的数据,size<=2\*10^5

输入例子**:**

1,2,3,4,5,6,7,0

输出例子**:**

7

###### 解题思路：

这个题目最开始的想法就是直接进行遍历，从前往后比，冒泡排序一样，只要是前面比后面大的就count++，这样时间复杂度是O(n^2)。但是提交过后，提示我复杂度过高，因此需要另一种方式。这个题跟归并排序很像，在归并排序的时候，两个元素之间只比较一次，如果左面的第一个元素比右面的大，则左面数组的所有值都比右边的第一个元素大，这是count += mid - i +1(i是左面的元素的下标)。具体代码如下：

public class Solution { int count =0;

public int InversePairs(int [] array) { if(array.length <= 1)

return 0;

//辅助数组，用来合并数组时候用的，省的每次都要重新申请再释放int[] tmp = new int[array.length]; mergeSort(array, 0, array.length-1,tmp);

return count % 1000000007;

}

public void mergeSort(int[] array, int begin, int end, int[] tmp){ if(begin<end){

int mid = (begin+end)/2;

mergeSort(array, begin, mid, tmp);//左数组有序mergeSort(array, mid+1, end, tmp);//右数组有序mergeArray(array, begin, mid, end, tmp);//合并左右数组

}

}

public void mergeArray(int[] array, int begin, int mid, int end, int[] tmp){ int i=begin,j=mid+1;//左子树开头，右子树开头

int index = 0;//记tmp下标

while(i<=mid&&j<=end){

if(array[i] > array[j]){//如果左边的第一个元素比右边的第一个元素大，则左边所有的元素

都比这个元素大

count += mid - i + 1;//count+=左边元素到mid中间所有的元素

tmp[index++] = array[j]; j++;

if(count >=1000000007 )

count = count % 1000000007 ;

}else{

tmp[index++] = array[i]; i++;

}

}

while(i <= mid){

tmp[index++] = array[i++];

}

while(j <= end){

tmp[index++] = array[j++];

}

for(i=0;i<index;i++){//给数组排序用array[begin+i] = tmp[i];

}

}

}

#### 题目描述：两个链表的第一个公共节点

输入两个链表，找出它们的第一个公共结点。

###### 解题思路：

这个题首先想到的是用两个栈，把两个链表都push()到栈中，由于从第1个公共节点以后所有的节点是同一个节点，所以从栈顶到第一个公共节点的元素都是相等的，当然还有第二种方式，就是两个链表一定有一个长一个短，把长的到短的差算出来，让长的先走这些步，然后两个一起走，当他们相等的时候就可以是第一个公共节点了，所以代码分别如下：

/\*

public class ListNode { int val;

ListNode next = null;

ListNode(int val) { this.val = val;

}

}\*/

import java.util.\*; public class Solution {

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) { if(pHead1 == null || pHead2 == null)

return null;

Stack<ListNode> stack1 = new Stack<ListNode>(); Stack<ListNode> stack2 = new Stack<ListNode>(); ListNode p1 = pHead1;

ListNode p2 = pHead2; while(p1!=null){

stack1.push(p1); p1 = p1.next;

}

while(p2!=null){ stack2.push(p2); p2 = p2.next;

}

ListNode p = null;

//注意空栈的判断

while(!stack1.empty()&&!stack2.empty()&&stack1.peek() == stack2.peek()){ p = stack1.peek();

stack1.pop();

stack2.pop();

}

return p;

}

}

/\*

public class ListNode { int val;

ListNode next = null;

ListNode(int val) { this.val = val;

}

}\*/

//第二种方法，长的比短的先走n步，然后在每步一比较import java.util.\*;

public class Solution {

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) { if(pHead1 == null || pHead2 == null)

return null; ListNode p1 = pHead1;

ListNode p2 = pHead2; int count1 = 0;

int count2 = 0; while(p1 != null){

p1 = p1.next; count1++;

}

while(p2 != null){ p2 = p2.next; count2++;

}

p1 = pHead1; p2 = pHead2;

if( count2 > count1 ){

for(int i=0;i<count2-count1;i++){ p2 = p2.next;

}

}else{

for(int i=0;i<count1-count2;i++){ p1 = p1.next;

}

}

while(p1!=null && p2!=null && p1 != p2){ p1 = p1.next;

p2 = p2.next;

}

return p1;

}

}

#### 题目描述：数字在排序数组中出现的次数

统计一个数字在排序数组中出现的次数。

###### 解题思路：

这个题的思想是找到这个这个数字的index，然后顺着index一个一个加直到不是这个值为止。如果顺序查找的话时间复杂度为O(n)，这个时候我们想到用binary search，时间复杂度为

O(logn)，但是二分查找到的index不一定是最开始的那个，所以还要往前回溯，然后再重头在加。代码如下：

public class Solution {

public int BinarySearch(int[] array, int k){ int begin = 0;

int end = array.length-1;

while(begin<=end){//注意二分查找中的判断条件是begin<=end，和end = mid-1 begin = mi

d+1的情况

int mid = (begin+end)/2; if(array[mid] == k){

return mid;

}else if(array[mid] > k){ end = mid-1;

}else{

begin = mid+1;

}

}

return -1;

}

public int GetNumberOfK(int [] array , int k) { if(array.length == 0)

return 0;

int index = BinarySearch(array, k); if(index == -1)

return 0;

for(;index>=0&& array[index] == k; index--){

}

index++;

int count = 0;

for(;index<array.length && array[index]==k;index++) count++;

return count;

}

}

#### 题目描述：二叉树的深度

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

###### 解题思路：

这个题求二叉树的深度，我们可以用递归来解决，二叉树的深度应该等于左子树的深度与右子树深度的最大值加1，就这么简单。

代码如下：

/\*

public class TreeNode { int val = 0; TreeNode left = null;

TreeNode right = null; public TreeNode(int val) {

this.val = val;

}

};\*/

public class Solution {

public int TreeDepth(TreeNode pRoot)

{

if(pRoot == null) return 0;

return Math.max(TreeDepth(pRoot.left), TreeDepth(pRoot.right)) + 1;

}

}

#### 题目描述：平衡二叉树

输入一棵二叉树，判断该二叉树是否是平衡二叉树。解题思路：

就是先判断深度，然后递归左右子树深度差值。代码如下：

public class Solution {

public int Depth(TreeNode root){ if(root == null)

return 0;

if(root.left == null && root.right == null) return 1;

int l = Depth(root.left); int r = Depth(root.right); return l>r?l+1:r+1;

}

public boolean IsBalanced\_Solution(TreeNode root) { if(root == null)

return true;

if(root.left == null && root.right == null) return true;

boolean flag = false;

if(Math.abs(Depth(root.right) - Depth(root.left))<=1) flag = true;

return flag&&IsBalanced\_Solution(root.left)&&IsBalanced\_Solution(root.right);

}

}

#### 题目描述：数组中只出现一次的数字

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

###### 解题思路：

第一个想法是用HashMap来存储，Key是数字，Value是次数，具体代码如下：

//num1,num2分别为长度为1的数组。传出参数

//将num1[0],num2[0]设置为返回结果

import java.util.\*; public class Solution {

public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) { HashMap<Integer,Integer> hash = new HashMap<Integer,Integer>(); if(array.length<2)

return;

for(int i=0;i<array.length;i++){ if(hash.containsKey(array[i])){

hash.put(array[i],hash.get(array[i])+1);

}else{

hash.put(array[i],1);

}

}

int count = 0;

for(int i=0;i<array.length;i++){ if(hash.get(array[i]) == 1){

if(count == 0){ num1[0] = array[i]; count++;

continue;

}else{

num2[0] = array[i]; break;

}

}

}

}

}

然而，实际上题目的意思并非如此。我们可以看到除了出现一次的数字以外，其他数字都出现了两次，为什么是两次不是三次呢，这个就是有说法的了。我们都知道两个相同的数字亦或^运算过后是等于0的，所以整个序列进行亦或运算以后剩下的就是只出现一次的数字亦或的结果。如果只有一个数字出现一次，那么这个数字就是我们要找的数字，如果两个数字的话，我们可以把序列分成两组，两个数字亦或过后肯定不为0，那么找到从右往左数的第一个

不为0的为，那么两个数字中一个数字的这一位是0，另一位不是零，我们可以把所有数字按照这个特点分成两组，然后再重复第一个过程，找到这个数字。例如{1,2,2,3,3,4}亦或后的结果是0101,我们找到最右端为1的分为一组，为0的分为一组，{1,3,3}和{2,2,4}在求亦或，{1}{4} 完美。具体代码如下：

//num1,num2分别为长度为1的数组。传出参数

//将num1[0],num2[0]设置为返回结果

import java.util.\*; public class Solution {

public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {

int result = 0;

for(int i=0;i<array.length;i++){ result = result^array[i];

}

//找到第一个不为0的位int bit = 0; while(result!=0){

if( (result & 1) != 0 ) break;

result = result >> 1; bit++;

}

//flag左移bit位int flag = 1; flag = 1 << bit; num1[0] = 0;

num2[0] = 0;

for(int i=0;i<array.length;i++){ if( (array[i] & flag) == 0){

num1[0] = num1[0] ^ array[i];

}else{

num2[0] = num2[0] ^ array[i];

}

}

}

}

#### 题目描述：和为**S**的连续正整数序列

小明很喜欢数学,有一天他在做数学作业时,要求计算出9~16的和,他马上就写出了正确答案是

100。但是他并不满足于此,他在想究竟有多少种连续的正数序列的和为100(至少包括两个

数)。没多久,他就得到另一组连续正数和为100的序列:18,19,20,21,22。现在把问题交给你,你能不能也很快的找出所有和为S的连续正数序列? Good Luck!

输出描述**:**

输出所有和为S的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序

###### 解题思路：

这个题的我的解题思路就是序列的值如果比sum小，就加一个，如果比sum大就减一个最小的，如果相等就把整个序列添加进去然后减掉最小的。具体代码如下：

import java.util.ArrayList; public class Solution {

public int sumOfList(ArrayList<Integer> array){ int sum = 0;

for(int i=0;i<array.size();i++){ sum+= array.get(i);

}

return sum;

}

public ArrayList<ArrayList<Integer> > FindContinuousSequence(int sum) { ArrayList<ArrayList<Integer> > retArr = new ArrayList<ArrayList<Integer>>(); ArrayList<Integer> array = new ArrayList<Integer>();

int i=1; while(i<=sum){

//如果比sum小，就接着往下加

if(sumOfList(array) < sum){ array.add(i++);

}//如果比sum大，就减掉一个最小的

else if(sumOfList(array) > sum){ array.remove(0);

}

else{

ArrayList<Integer> add = new ArrayList<Integer>(array); retArr.add(add);

array.remove(0);

}

}

return retArr;

}

}

高端解法，不用计算整个ArrayList的值，而是使用small，big，curSum来标记这些值，具体代码如下：

import java.util.ArrayList; public class Solution {

public ArrayList<ArrayList<Integer> > FindContinuousSequence(int sum) { ArrayList<ArrayList<Integer>> sqList = new ArrayList<ArrayList<Integer>>(); if(sum < 3) return sqList;

int small = 1; int big = 2;

int curSum = small + big;

//设置middle变量的原因在于如果在当前总和小于sum的情况下，

//small在增加到sum的一半的过程中肯定会大于sum

int middle = (1 + sum) / 2; while(small < middle){

ArrayList<Integer> sq = new ArrayList<Integer>(); if(curSum == sum){

for (int i = small; i <= big; i++) { sq.add(i);

}

}

while(curSum > sum && small < middle){ curSum -= small;

small++; if(curSum == sum){

for (int i = small; i <= big; i++) { sq.add(i);

}

}

}

if(sq.size() > 0) sqList.add(sq);; big++;

curSum += big;

}

return sqList;

}

}

#### 题目描述：和为**S**的两个数字

输入一个递增排序的数组和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。

输出描述**:**

对应每个测试案例，输出两个数，小的先输出。

解题思路：

这个题的解题思路就是，由于是递增序列，所以可以从两端向中间进行加和，如果和比S大， 说明这个东西过大，把较大的右端换成它左边相对较小的值。如果和比S小，说明这个和过 小，需要变大，则把较小的左端换成相对较大的右边的值，具体代码如下：

import java.util.ArrayList; public class Solution {

public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) { ArrayList<Integer> retArr = new ArrayList<Integer>(); if(array.length < 2)

return retArr; int begin = 0;

int end = array.length-1; while(begin<end){

if(array[begin]+array[end] == sum){ retArr.add(array[begin]); retArr.add(array[end]);

return retArr;

}else if(array[begin]+array[end] <sum){ begin++;

}else{

end--;

}

}

return retArr;

}

}

#### 题目描述：左旋转字符串

汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的字符序列S，请你把其循环左移K位后的序列输出。例如，字符序列S=”abcXYZdef”,要求输出循环左移3位后的结果，即“XYZdefabc”。是不是很简单？OK，搞定它！

解题思路：

最简单的方法就是利用String的substring()方法，得到两个子串，然后拼接就可以了。要注意的是如果n>str.length()，要n=n%str.length()；具体代码如下：

public class Solution {

public String LeftRotateString(String str,int n) { if(str.length() <=1)

return str;

n = n%str.length();

String str1 = str.substring(n); String str2 = str.substring(0,n); return str1+str2;

}

}

还有另一种解法，就是跟之前的翻转单词顺序列一样，先把整个字符串旋转，然后再进行局部旋转两次。

#### 题目描述**:** 翻转单词顺序列

牛客最近来了一个新员工Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事

Cat对Fish写的内容颇感兴趣，有一天他向Fish借来翻看，但却读不懂它的意思。例

如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了，正确的句子应该是“I am a student.”。Cat对一一的翻转这些单词顺序可不在行，你能帮助他么？

解题思路：这道题可以有两种解法，第一种解法就是先把大的String反过来，然后再把字符串分割开。再把每个字符串反过来就可以了。第二种解法是直接分割，然后对每个翻转，最后再把大的翻转了。

//第一种解法 注意最开始的判断string为null时的情况

public class Solution {

public static void reverse(char ch[], int begin, int end){

while(begin<end){

char tmp = ch[begin]; ch[begin] = ch[end]; ch[end] = tmp; begin++;

end--;

}

}

public String ReverseSentence(String str) { if(str == null || str.length()<=1)

return str;

char ch[] = str.toCharArray(); reverse(ch, 0, ch.length-1); int begin=0, end = 0; while(begin<ch.length){

if(ch[begin]==' '){//如果第一个就遇到空格的话，直接跳过

begin++; end++;

}else if( end==ch.length|| ch[end]==' '){//如果是空格的话，或者遇到最后一个字符

后面的字符的时候

reverse(ch, begin, --end);

begin = ++end;//最后一定要让begin和end相等

}else{

end++;

}

}

String ret = new String(ch); return ret;

}

}

#### 题目描述：扑克牌顺子

LL今天心情特别好,因为他去买了一副扑克牌,发现里面居然有2个大王,2个小王(一副牌原本是54张^\_^)...他随机从中抽出了5张牌,想测测自己的手气,看看能不能抽到顺子,如果抽到的话,他决定去买体育彩票,嘿嘿！！“红心A,黑桃3,小王,大王,方片5”,“Oh My God!”不是顺子.....LL不高兴了,他想了想,决定大\小 王可以看成任何数字,并且A看作1,J为11,Q为12,K为13。上面的5张牌就可以变成“1,2,3,4,5”(大小王分别看作2和4),“So Lucky!”。LL决定去买体育彩票啦。 现在, 要求你使用这幅牌模拟上面的过程,然后告诉我们LL的运气如何。为了方便起见,你可以认为大小王是0。

解题思路：

这个题可以先把整个数组排序，然后由于左端可能是大小王，所以从右端开始往左判断，首尾各一个指针，当尾部指针的值比它左面的值恰好大1的情况下，不需要大小王，如果如果差值比1大的话，左端恰好有一个大小王的话，则把当前的尾部指针的值减1，然后首部指针向后移动一个位置，也就是用掉了一个王，这样知道首尾指针相等的情况下，则该这副牌就是顺子，如果不是顺子的话早就返回了。具体代码如下：

import java.util.\*; public class Solution {

public boolean isContinuous(int [] numbers) { if(numbers.length <=1){

return false;

}

Arrays.sort(numbers); int begin = 0;

int end = numbers.length -1; boolean flag = false;

//从右端开始想左端移动指针

while(begin < end){

if(numbers[end]-numbers[end-1] == 1){ end--;

}else{

if(numbers[begin] == 0){ begin++;

//大小王相当于把这个大牌值减一

numbers[end]--;

}

else{

break;

}

}

}

if(begin == end) return true;

return flag;

}

}

沈斌强算法：就是先算出大小王的数量，然后算出所有其他数字之间的差值，如果差值大于大小王数量则废掉，如果小于等于则可以是顺子

import java.util.Arrays; public class Solution {

public boolean isContinuous(int [] numbers) {

if(numbers == null || numbers.length == 0) return false;

//对数组进行排序Arrays.sort(numbers);

//统计大小王的张数

int numberOfZero = 0; int numberOfGap = 0;

for(int i = 0; i < numbers.length && numbers[i] == 0; i++) numberOfZero++;

int low = numberOfZero; int high = low + 1;

while(high < numbers.length){

//如果出现了对子则不可能是顺子

if(numbers[low] == numbers[high]) return false;

//计算空缺数

numberOfGap += numbers[high] - numbers[low] - 1; low = high;

high++;

}

return numberOfGap <= numberOfZero ? true : false;

}

}

题目描述**:**求 **1+2+3+...+n**

求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

解题思路：

这个题不然用乘除法，不让用for循环那么只能用递归来解决，又因为递归需要有终结条件， 因此需要使用&&操作表示要不要继续执行。具体代码如下：

public class Solution { private int result = 0;

public int Sum\_Solution(int n) { cal(n);

return result;

}

public boolean cal(int n){ result += n;

return n!=0&&cal(n-1);

}

}

#### 题目描述：不用加减乘除做加法

写一个函数，求两个整数之和，要求在函数体内不得使用+、-、\*、\/四则运算符号。

###### 解题思路：

首先这个题想到如果不能用加减乘除做，那么只能用逻辑运算符，由于是加法，因此我们都知道加法无非就是求不进位的值，和进位的值，不进位的加和值我们可以通过异或^操作得 到，进位的话只有在两位都是1的时候才有，因此想到用&操作。只要把&操作结果左移一位然后再与之前亦或的结果再异或一遍就OK了，知道不再有进位为止。具体代码如下：

public class Solution {

public int Add(int num1,int num2) { int sum = 0;

int carry = 0; while(num2!=0){

sum = num1 ^ num2;

carry = (num1&num2) << 1;//进位num1 = sum;

num2 = carry;

}

return num1;

}

}

#### 题目描述：把字符串转换成整数

将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。

###### 解题思路：

这个题目关键的几个重要的边界判断：

1. 第一个字符是否为正负数的判断（包括如果为正负号，但是只有一个符号没有其他数字）
2. 字符串为空的判断
3. 是否有非法字符的判断等等。所以具体代码如下：

public class Solution {

public int StrToInt(String str) { if(str.isEmpty()){

return 0;

}

char[] ch = str.toCharArray();

int positive = 0;//1表示符号位为正，0表示没有符号位，-1表示符号位为- int len = ch.length;

int ret = 0;

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

//判断是否有符号位

if(i==0 && (ch[i]=='+' ||ch[i]=='-') ){ if(ch[i] == '-')

positive = -1; else{

positive = 1;

}

if(len == 1)

return 0; continue;

}

//判断是否为异常字符if(ch[i]<'0' || ch[i]>'9')

return 0;

if(positive == 0)

ret += (ch[i]-'0')\*(int)Math.pow(10,len-1-i); else{

ret += (ch[i]-'0')\*(int)Math.pow(10,len-1-i);

}

}

if(positive==0 || positive == 1) return ret;

else

return -1\* ret;

}

}

#### 题目描述：数组中重复的数字

在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字2或者3。

###### 解题思路：

第一反应使用HashSet存储，遍历整个数组，add到set中，如果add失败，立刻返回该数字。代码如下：

}

public boolean duplicate(int numbers[],int length,int [] duplication) { if(length <=1 )

return false; boolean flag = false;

HashSet<Integer> hash = new HashSet<Integer>(); for(int i = 0;i<length; i++){

if(!hash.add(numbers[i])){

duplication[0] = numbers[i]; return true;

}

}

return false;

}

otherwise false

true if the input is valid, and there are some duplications

// Return value: in the array number

//

plication array is 1,so using duplication[0] = ? in implementation;

// Here duplication like pointor in C/C++, duplication[0] equal \* duplication in C/C++

// 这里要特别注意~返回任意重复的一个，赋值duplication[0]

duplication: (Output) the duplicated number in the array number,length of du

an array of integers

the length of array numbers

numbers:

length:

//

//

//

import java.util.HashSet; public class Solution {

// Parameters:

方法二：让数组的下标与数字相对应，遍历数组，比较当前值与数组下标是否相同。如果不相同，则与数组下标等于当前值得数进行交换，在遍历过程中，若发现需要交换的两个数大小相同，则直接返回给数组，并结束遍历。

}

public boolean duplicate(int numbers[],int length,int [] duplication) {

//异常处理

if(numbers==null||numbers.length<=1||length<=1) return false; boolean result = false;

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

if(i != numbers[i]){ //判断当前数组下标和数组的值是否相等if(numbers[numbers[i]]!=numbers[i]){ //判断两个需要交换的数组的值是否相等

int temp = numbers[numbers[i]]; numbers[numbers[i]] = numbers[i]; numbers[i] = temp;

}else{ //相等则说明找到了重复的数

duplication[0] = numbers[i]; result = true;

}

}

}

return result;

}

otherwise false

true if the input is valid, and there are some duplications

// Return value: in the array number

//

plication array is 1,so using duplication[0] = ? in implementation;

// Here duplication like pointor in C/C++, duplication[0] equal \* duplication in C/C++

// 这里要特别注意~返回任意重复的一个，赋值duplication[0]

duplication: (Output) the duplicated number in the array number,length of du

an array of integers

the length of array numbers

numbers:

length:

//

//

//

import java.util.Map; import java.util.HashMap;

public class Solution {

// Parameters:

方法三：新建一个数组，当做是HashMap，Key是下标，Value是次数，如果次数大于0，则返回，等于零，++。

}

public boolean duplicate(int numbers[],int length,int [] duplication) { if(length <=1 )

return false; boolean flag = false;

int[] tmp = new int[length]; for(int i=0;i<length;i++){

if(tmp[numbers[i]]==0){ tmp[numbers[i]]++;

}else{

duplication[0] = numbers[i]; return true;

}

}

return false;

}

otherwise false

true if the input is valid, and there are some duplications

// Return value: in the array number

//

plication array is 1,so using duplication[0] = ? in implementation;

// Here duplication like pointor in C/C++, duplication[0] equal \* duplication in C/C++

// 这里要特别注意~返回任意重复的一个，赋值duplication[0]

duplication: (Output) the duplicated number in the array number,length of du

an array of integers

the length of array numbers

numbers:

length:

//

//

//

import java.util.HashSet; public class Solution {

// Parameters:

#### 题目描述：构建乘积数组

给定一个数组A[0,1,...,n-1],请构建一个数组B[0,1,...,n-1],其中B中的元素B[i]=A[0]\*A[1]\*...\*A[i- 1]\*A[i+1]\*...\*A[n-1]。不能使用除法。

###### 解题思路：

暴力解法，就是一个一个乘嘛，除了A[i]之外都乘。代码如下：

import java.util.ArrayList; public class Solution {

public int[] multiply(int[] A) {

if(A.length == 0) return A;

int[] B = new int[A.length]; for(int i=0;i<A.length;i++){

int tmp = 1;

for(int j=0;j<A.length;j++){ if(j == i)

continue; tmp \*= A[j];

}

B[i] = tmp;

}

return B;

}

}

第二个方法就比较优雅了，写一个front数组表示A[0]\*....A[i-1]的值，back数组表示A[i]\*....A[n-

1]的值，最后一次循环把两个数组一乘就完事了，具体代码如下：

import java.util.ArrayList; public class Solution {

public int[] multiply(int[] A) { int n = A.length;

//front[i]就是从A[0]...到A[i - 1]的值

int[] front = new int[n]; if(n<=1) return front;

/\* back[i]就是从A[i + 1]...到A[n - 1]的值

\* back数组的第一位从最后一位开始，所以back[n - 1]=1

\*/

int[] back = new int[n]; front[0] = back[n-1] = 1;

//分别计算前半部分和后半部分的值，并分别将结果保存在front与back数组中

for(int i=1;i<n;i++){

front[i] = front[i-1]\*A[i-1];

back[n-i-1] = back[n-i]\*A[n-i];

}

//将两个计算结果两次相乘得到最后的结果for(int i=0;i<n;i++){

front[i] \*=back[i];

}

//返回front数组即为所求return front;

}

#### 题目描述：矩阵中的路径

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如 a b c e s f c s a d e e 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

###### 解题思路：

代码如下：

public class Solution {

/\* @author miracle

\*/

public boolean hasPath(char[] matrix, int rows, int cols, char[] str){

//创建一个布尔数组

boolean visited[]= new boolean[matrix.length]; for(int i = 0;i<rows;i++){

for(int j=0;j<cols;j++){ if(hasPathCore(matrix,i,j,0,rows,cols,str,visited)){

return true;

}

}

}

return false;

}

public boolean hasPathCore(char[] matrix,int i,int j,int k,int rows,int cols,char[] str,boolean[] visited){

// 对于给定行数和列数得到其在矩阵中的下标

int index = i\*cols+j;

//判断合法性if(i<0||i>=rows||j<0||j>=cols||matrix[index]!=str[k]||visited[index]){

return false;

}

visited[index] = true;

// 如果递归到最后一个位置的字符，则表明前面位置的字符都在矩阵中找到了对应的位置

if (k == str.length - 1) return true;

if(hasPathCore(matrix,i-1,j,k+1,rows,cols,str,visited)

||hasPathCore(matrix,i,j-1,k+1,rows,cols,str,visited)

||hasPathCore(matrix,i+1,j,k+1,rows,cols,str,visited)

||hasPathCore(matrix,i,j+1,k+1,rows,cols,str,visited)){ return true;

}else{

//如果相邻格子的字符都没有匹配到下一个字符，则需要回到前一个格子，从而需要把把位置的状态重新设定为未访问

k--;

visited[index] = false;

}

return false;

}

}