# 1. 项目

## （1）项目的基本流程

## （2）项目所涉及的知识点

## （3）难点，如何解决，自己找两个难点

## （4）用到的技术

## （5）再往下做可以到什么程度

# 2. 数据库

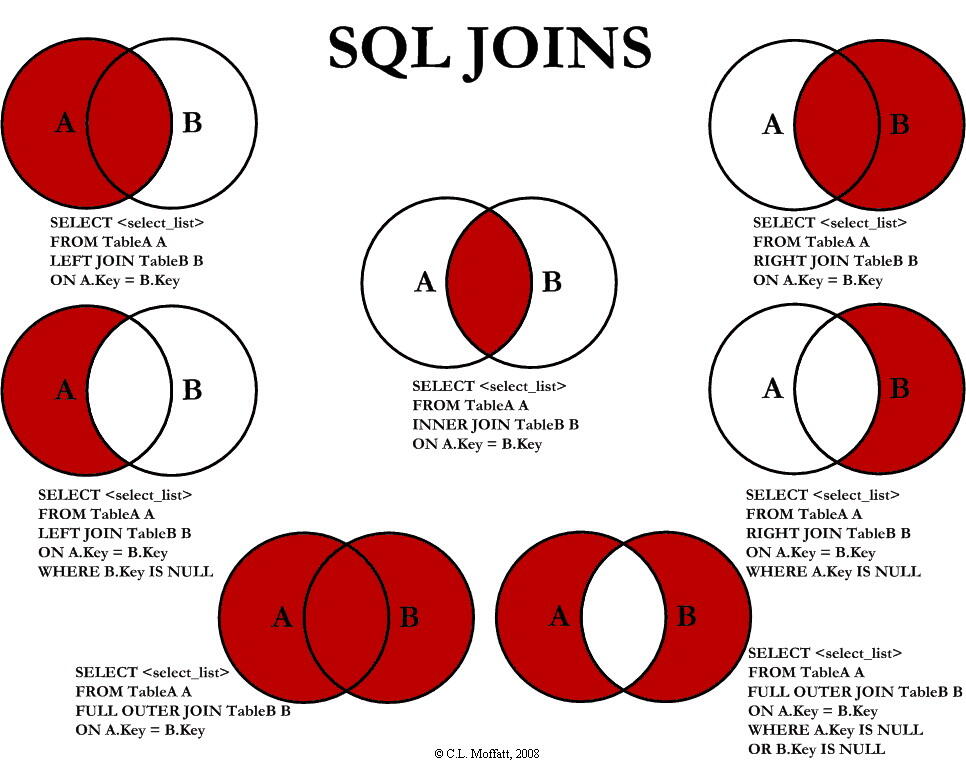
## （1）SQL

SQL 语句有一个让大部分人都感到困惑的特性，就是：SQL 语句的执行顺序跟其语句的语法顺序并不一致。SQL 语句的语法顺序是：

* SELECT [DISTINCT]
* FROM
* WHERE
* GROUP BY
* HAVING
* UNION
* ORDER BY

为了方便理解，上面并没有把所有的 SQL 语法结构都列出来，但是已经足以说明 SQL 语句的语法顺序和其执行顺序完全不一样，就以上述语句为例，其执行顺序为：

* FROM
* WHERE
* GROUP BY
* HAVING
* SELECT
* DISTINCT
* UNION
* ORDER BY



http://blog.jobbole.com/55086/

## （2）MySQL

### 1）索引

#### a如何实现索引？

主要是B+树：http://blog.csdn.net/hguisu/article/details/7786014#t0

#### b 索引为什么不是越多越好？

索引能够极大的提高数据检索效率，也能够改善排序分组操作的性能，但是我们不能忽略的 一个问题就是索引是完全独立于基础数据之外的一部分数据。假设我们在Table ta 中的Column ca 创 建了索引idx\_ta\_ca，那么任何更新Column ca 的操作，MySQL 都需要在更新表中Column ca 的同时， 也更新Column ca 的索引数据，调整因为更新所带来键值变化后的索引信息。而如果我们没有对 Column ca 进行索引的话,MySQL 所需要做的仅仅只是更新表中Column ca 的信息。这样，所带来的最 明显的资源消耗就是增加了更新所带来的IO 量和调整索引所致的计算量。此外，Column ca 的索引 idx\_ta\_ca 是需要占用存储空间的，而且随着Table ta 数据量的增长，idx\_ta\_ca 所占用的空间也会 不断增长。所以索引还会带来存储空间资源消耗的增长。

如果一本书200页，其中150页是索引，50页是内容，你会认为这本书好么，增加删除数据会极其的复杂。

### 2）事务操作

原子性， 一个事务 要么完全提交 要么完全回滚，不会介于2者之间。

一致性，一个查询发起后，不管数据发生了多少变化 多少事务，查询结果应当为发起查询时间一致的数据

数据库应用中完成单一逻辑功能的操作集合，是一个既具有原子性又具有一致性的功能，我们要求事务不违反任何数据库的一致性约束，也就是说，如果事务启动时数据是一致的，那么当这个事务成功结束的时候数据库也应该是一致的

3）隔离性（Isolation）

指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后的状态，**事务不会查看到中间状态的数据**。

4）持久性（Durability）

指的是只要事务成功结束，它对数据库所做的更新就必须**永久保存**下来。即使发生系统崩溃，重新启动数据库系统后，数据库还能恢复到事务成功结束时的状态。

### 3）数据库性能优化

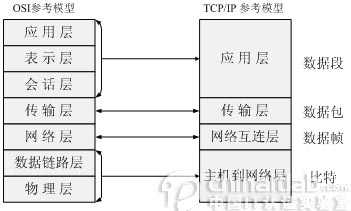
#### A 数据库方面

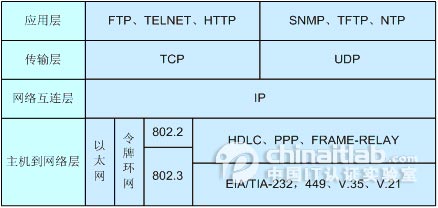
#### B java方面

## （2）NoSQL

# 3. 网络

## 1 TCP/IP四层协议





ARP RARP 网络层的协议,但是它所工作的内容是链路层的

ARP ：IP->MAC

RARP: MAC->IP

## 2 UDP协议

## 3 TCP协议

http://www.cnblogs.com/kzloser/articles/2582957.html

### （1）TCP建立连接后怎么保持连接？

具体是用TCP协议自身的保活定时器实现心跳，还是通讯双方通过自定义协议实现心跳包来保活，还是通过第三方应用来进行保活，要取决于你的项目需求，网络负荷以及那种实现起来更便捷等多方面的因素，根据自身情况来使用。

### （2）短连接和长连接

<http://www.cnblogs.com/liuyong/archive/2011/07/01/2095487.html>

短连接：短连接一般只会在client/server间传递一次读写操作

长连接：Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

首先说一下TCP/IP详解上讲到的TCP保活功能，保活功能主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将永远等待客户端的数据，保活功能就是试图在服务器端检测到这种半开放的连接。

### （3）为什么要三次握手，两次不可以么？

谢希仁版《计算机网络》中的例子是这样的，“已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”

## 4 HTTP协议

### （1）了解http

### （2）http响应状态码

使用ASP.NET/PHP/JSP 或者javascript都会用到http的不同状态，一些常见的状态码为：

200 – 服务器成功返回网页 404 – 请求的网页不存在 503 – 服务不可用

2xx （成功）

表示成功处理了请求的状态代码。

3xx （重定向）

表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向。

4xx（请求错误）

这些状态代码表示请求可能出错，妨碍了服务器的处理。

5xx（服务器错误）

这些状态代码表示服务器在尝试处理请求时发生内部错误。 这些错误可能是服务器本身的错误，而不是请求出错。

### （3）http协议GET和POST的区别

http://www.cnblogs.com/hyddd/archive/2009/03/31/1426026.html

# 4. JAVA

## 1 util包下的数据结构

### 1.ArrayDeque 双端队列

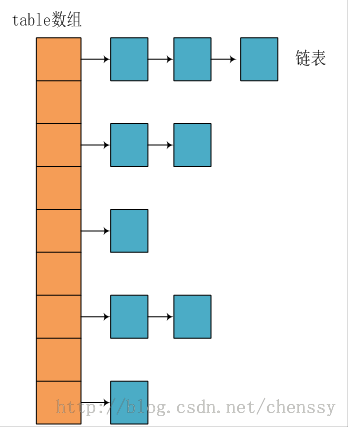
通过数组来实现 elements = (E[]) new Object[16];

### 2.HashMap

加载因子是表示Hsah表中元素的填满的程度.若加载因子越大,填满的元素越多,好处是：空间利用率高了,但冲突的机会加大了.反之,加载因子越小,填满的元素越少,好处是:冲突的机会减小了,但空间浪费多了.

冲突的机会越大,则查找的成本越高.反之,查找的成本越小.因而,查找时间就越小.

因此,必须在 "冲突的机会"与"空间利用率"之间寻找一种平衡与折衷. 这种平衡与折衷本质上是数据结构中有名的"时-空"矛盾的平衡与折衷.



#### HashMap与Hashtable的区别

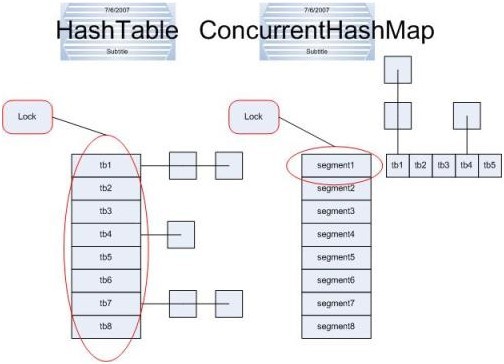
1.构造函数默认的容量不同：hashmap 16 hashtable 11

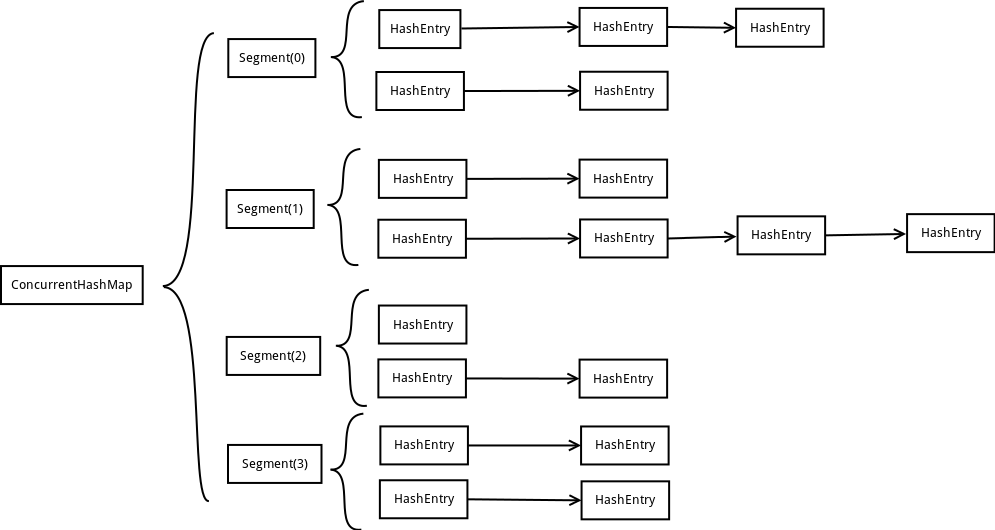
2.put方法不通：hashmap的key可以为null，而hashtable的key不能为null

3.hashtable是线程安全的，方法做了synchronized，而hashmap不是线程安全的。但是效率要比hashtable高。如果要用线程安全的map，我们可以用concurrentHashmap

4.hashtable：rehash长度为原来的2倍+1，hashmap：resize为原来长度的2倍

#### ConcurrentHashMap与Hashtable的区别





虽然它们都是线程安全的，但是Hashtable 锁住的是整张表结构

ConcurrentHashMap锁的方式是稍微细粒度的。 ConcurrentHashMap将hash表分为16个桶（默认值），诸如get,put,remove等常用操作只锁当前需要用到的桶。原来 只能一个线程进入，现在却能同时16个写线程进入（写线程才需要锁定，而读线程几乎不受限制，之后会提到），并发性的提升是显而易见的。

更令人惊讶的是ConcurrentHashMap的读取并发，因为在读取的大多数时候都没有用到锁定，所以读取操作几乎是完全的并发操作，而写操作锁定的粒度又非常细，比起之前又更加快速（这一点在桶更多时表现得更明显些）。只有在求size等操作时才需要锁定整个表。

#### ConcurrentHashMap

##### 线程不安全的HashMap

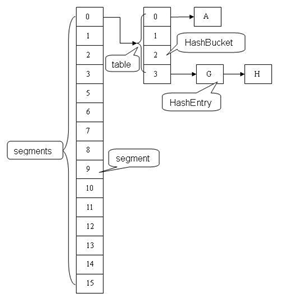
因为多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

##### 效率低下的HashTable容器

HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

##### ConcurrentHashMap的结构

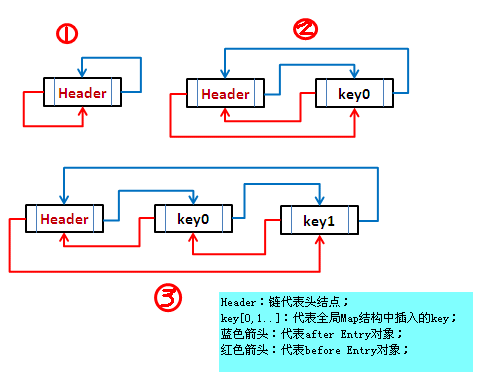
ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。



http://www.cnblogs.com/ITtangtang/p/3948786.html

### 3 LinkedHashMap

LinkedHashMap保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的。



### 4 IdentityHashMap

1.简单说IdentityHashMap与常用的HashMap的区别是：前者比较key时是“引用相等”而后者是“对象相等”，即对于k1和k2，当k1==k2时，IdentityHashMap认为两个key相等，而HashMap只有在k1.equals(k2) == true 时才会认为两个key相等。IdentityHashMap 允许使用null作为key和value. 不保证任何Key-value对的之间的顺序, 更不能保证他们的顺序随时间的推移不会发生变化.

2.IdentityHashMap有其特殊用途，比如序列化或者深度复制。或者记录对象代理。

3.举个例子，jvm中的所有对象都是独一无二的，哪怕两个对象是同一个class的对象，而且两个对象的数据完全相同，对于jvm来说，他们也是完全不同的，如果要用一个map来记录这样jvm中的对象，你就需要用IdentityHashMap，而不能使用其他Map实现。

### 5 TreeMap

TreeMap 通过红黑树实现，红黑树是一棵二叉平衡树。它通过颜色的约束来维持着二叉树的平衡。对于一棵有效的红黑树二叉树而言我们必须增加如下规则：

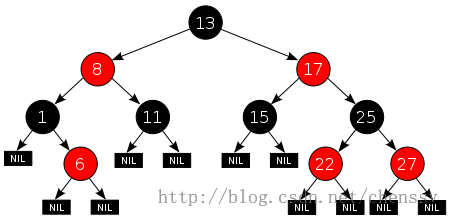
1、每个节点都只能是红色或者黑色

2、根节点是黑色

3、每个叶节点（NIL节点，空节点）是黑色的。

4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。



在TreeMap的put()的实现方法中主要分为两个步骤，第一：构建排序二叉树，第二：平衡二叉树fixAfterInsertion(Entry<K,V> x)

删除节点：

针对于红黑树的增加节点而言，删除显得更加复杂，使原本就复杂的红黑树变得更加复杂。同时删除节点和增加节点一样，同样是找到删除的节点，删除之后调整红黑树。但是这里的删除节点并不是直接删除，而是通过走了“弯路”通过一种捷径来删除的：找到被删除的节点D的子节点C，用C来替代D，不是直接删除D，因为D被C替代了，直接删除C即可。所以这里就将删除父节点D的事情转变为了删除子节点C的事情，这样处理就将复杂的删除事件简单化了。子节点C的规则是：右分支最左边，或者 左分支最右边的。

TreeMap和HashMap  
HashMap：适用于在Map中插入、删除和定位元素。   
Treemap：适用于按自然顺序或自定义顺序遍历键(key)。   
HashMap通常比TreeMap快一点(树和哈希表的数据结构使然)，建议多使用HashMap，在需要排序的Map时候才用TreeMap。

### 6 WeakHashMap

强引用：平时我们编程的时候例如：Object object=new Object（）；那object就是一个强引用了。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

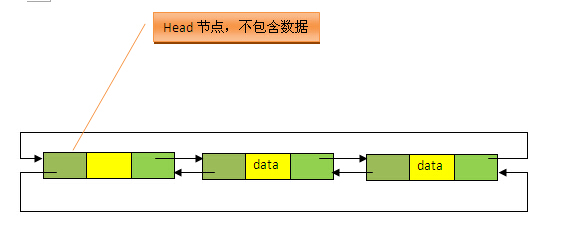
软引用（SoftReference）：如果一个对象只具有软引用，那就类似于可有可物的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存 空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。 软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java虚拟机就会把这个软引用加入到与之关联 的引用队列中。

弱引用（WeakReference）：如果一个对象只具有弱引用，那就类似于可有可物的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更 短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。 弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联 的引用队列中。

虚引用（PhantomReference）：“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象 仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。 虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队 列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

WeakHashMap，此种Map的特点是，当除了自身有对key的引用外，此key没有其他引用那么此map会自动丢弃此值。

### 7 LinkedList



### 8 ArrayList

ArrayList无参构造函数默认构造new Object[10]，所以我们构造ArrayList时尽量对大小有一个预判，设置一个初始值，这可以减少递增式再分配的数量

int newCapacity = (oldCapacity \* 3) / 2 + 1;

### 9 Vector

Vector和ArrayList的区别

1. 同步性。Vector是线程安全的，ArrayList不是线程安全的。从源码中可以得知。
2. 数据增长。内部都是使用一个数组来保存数据的。当内部数组空间不够时，Vector在默认情况下是增长一倍的大小，而ArrayList增加50%的大小。Vector确实有一点点优势，因为你可以指定增加值
3. 查询，插入，删除对象的效率。ArrayList和Vector在从指定位置取得元素，从容器的末尾增加和删除元素都非常的有效，所有的这些操作都能在一个常数级的时间O(1)内完成。但是从一个其他的位置增加和删除一个元素就显得颇为费时，差不多需要的时间为O(n-i)，这里的n代表元素个数，i代表要增加和删除的元素所在的位置。这些操作需花费更多的时间，因为你需要挨个移动i和更高位置的元素。

### 10 Stack

Stack内部是由Vector实现的。

### 11 Properties

Properties由Hashtable实现。

它提供了几个主要的方法：

1． getProperty ( String key)，用指定的键在此属性列表中搜索属性。也就是通过参数 key ，得到 key 所对应的 value。

2． load ( InputStream inStream)，从输入流中读取属性列表（键和元素对）。通过对指定的文件（比如说上面的 test.properties 文件）进行装载来获取该文件中的所有键 - 值对。以供 getProperty ( String key) 来搜索。

3． setProperty ( String key, String value) ，调用 Hashtable 的方法 put() 。他通过调用基类的put方法来设置 键 - 值对。

4． store ( OutputStream out, String comments)，以适合使用 load 方法加载到 Properties 表中的格式，将此 Properties 表中的属性列表（键和元素对）写入输出流。与 load 方法相反，该方法将键 - 值对写入到指定的文件中去。

5． clear ()，清除所有装载的 键 - 值对。该方法在基类中提供。

### 12 Arrays

1. static <T> List<T> asList（T…a）

Java中Arrays的asList()方法 可以将 数组转为List 但是，这个数组类型必须是引用类型的，如果是基本数据类型就不可以.

返回值是一个ArrayList，但是此ArrayList内部是由一个final的数组实现的，所以不能添加或者删除元素，如果想添加或者删除元素，可以重新创建一个ArrayList，然后addAll（）。

1. binarySearch()

实用二分搜索法搜索指定关键字

1. copyOf(int[] original, int newLength)

复制original数组到一个新数组，新数组的长度为newLength，用0填充。

1. copyOfRange（int[]original, int from ,int to）

将指定数组的指定范围复制到一个新数组里

1. deepEquals()和equals()

如果两个数组使用equals返回true，则使用deepEquals也返回true，也就是说在比较的两个数组均为一维数组的前提下，equals和deepEquals的比较结果没有差别；如果要比较多为数组，则需要使用deepEquals方法.

1. fill()

填充数组

1. sort(int [] a)

对指定数组排序。该排序算法是一个经过调优的快速排序算法

### 13 Collections

http://blog.sina.com.cn/s/blog\_a46817ff01017hqr.html

## 2 Java源码阅读

### String

String 深入解读：<http://www.cnblogs.com/ITtangtang/p/3976820.html>

1. String类初始化后是不可变的（immutable）
2. 实用String不一定创建对象
3. 实用new String,一定创建对象。
4. String.inter()

### Object

Object的方法摘要

1. clone（）创建并返回此对象的一个副本。“副本”的准确含义可能依赖于对象的类。

x.clone() != x为 true，表达式：

x.clone().getClass() == x.getClass()也为 true，但这些并非必须要满足的要求。

一般情况下：

x.clone().equals(x)为 true，但这并非必须要满足的要求。

2.equals()

3. finalize()当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

4. getClass()

5. hashCode()

6. notify()唤醒在此对象监视器上等待的单个线程。

7.wait()在其他线程调用此对象的 [notify()](mk:@MSITStore:C:\Users\Hailong\Desktop\JDK_API_1_6.CHM::/java/lang/Object.html#notify()) 方法或 [notifyAll()](mk:@MSITStore:C:\Users\Hailong\Desktop\JDK_API_1_6.CHM::/java/lang/Object.html#notifyAll()) 方法前，导致当前线程等待。

8. notifyAll() 唤醒在此对象监视器上等待的所有线程。

9. toString()

10. wait(long timeout) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。

11. wait(long timeout, int nanos) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

### java基本类型和包装类

基本数据类型：byte，int， short， long， boolean，char， float，double等

包装类型 ： Byte，Integer，Short，Long，Boolean，Character，Float,Double等

基本类型 　 大小 　 最小值 　 最大值 　 包装器类型

boolean 　　 - 　　　 - 　　　 - 　　　 Boolean

char 　　　 16bit Unicode 0 Unicode 2(16)-1 Character

byte 8bit -128 +127 Byte

short 16bit -2(15) 2(15)-1 Short

int 32bit -2(31) 2(31)-1 Integer

long 64bit -2(63) 2(63)-1 Long

float 32bit IEEE754 IEEE754 Float

double 64bit IEEE754 IEEE754 Double

void - - - Void

在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配

而实际的对象是在存储堆内存中

#### 包装类(Wrapper Class)共同的方法

值得说明的是,java是可以直接处理基本类型的,但是在有些情况下我们需要将其作为对象来处理,这时就需要将其转化为包装类了.所有的包装类(Wrapper Class)都有共同的方法,他们是:

(1) 带有基本值参数并创建包装类对象的构造函数.如可以利用Integer包装类创建对象,Integer obj=new Integer(145);

(2)带有字符串参数并创建包装类对象的构造函数.如new Integer("-45");

(3)可生成对象基本值的typeValue方法,如obj.intValue();

(4)将字符串转换为基本值的 parseType方法,如Integer.parseInt(args[0]);

(5)生成哈稀表代码的hashCode方法,如obj.hasCode();

(6)对同一个类的两个对象进行比较的equals()方法,如obj1.eauqls(obj2);

(7)生成字符串表示法的toString()方法,如obj.toString().

#### 转换关系

基本类型------>包装器类

Integer obj=new Integer(145);

包装器类------>基本类型

int num=obj.intValue();

字符串------>包装器类

Integer obj=new Integer("-45");

包装器类------>字符串包装器类

String str=obj.toString();

字符串------>基本类型

int num=Integer.parseInt("-45.36");

基本类型------>字符串包装器类

String str=String.valueOf(5);

### Class

Class.forName(String className)方法实现动态加载类。

Class 的静态 forName() 方法有两个版本，上面的代码是只指定类名称的版本，而另一个版本可以让你指定类名称、加载时是否运行静态区块、指定类加载器：

Static Class forName(String name, boolean initialize, ClassLoader loader)

Class.forName(String className)实际上调用的就是Class forName(className, true, currentLoader)

将 initialize 设定为 false，这样在加载类时并不会立即运行静态区块，而会在使用类建立对象时才运行静态区块。

### ClassLoader

http://www.codeceo.com/article/java-classloader.html

### System

http://www.cnblogs.com/xuhuan/articles/1522212.html

### Runtime

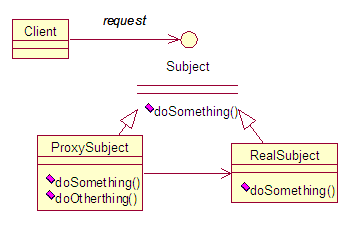
### StringBuffer和StringBuilder

jdk的实现中StringBuffer与StringBuilder都继承自AbstractStringBuilder，对于多线程的安全与非安全看到StringBuffer中方法前面的一堆synchronized就大概了解了。

这里随便讲讲AbstractStringBuilder的实现原理：我们知道使用StringBuffer等无非就是为了提高java中字符串连接的效率，因为直接使用+进行字符串连接的话，jvm会创建多个String对象，因此造成一定的开销。AbstractStringBuilder中采用一个char数组来保存需要append的字符串，char数组有一个初始大小，当append的字符串长度超过当前char数组容量时，则对char数组进行动态扩展，也即重新申请一段更大的内存空间，然后将当前char数组拷贝到新的位置，因为重新分配内存并拷贝的开销比较大，所以每次重新申请内存空间都是采用申请大于当前需要的内存空间的方式，这里是2倍。

## 3 反射

Java反射机制：<http://www.cnblogs.com/hxsyl/archive/2013/03/23/2977593.html>



首先让我们来了解一下如何使用 Java 动态代理。具体有如下四步骤：

1.通过实现 InvocationHandler 接口创建自己的调用处理器；

// InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理类到委托类的分派转发

// 其内部通常包含指向委托类实例的引用，用于真正执行分派转发过来的方法调用

InvocationHandler handler = new InvocationHandlerImpl(..);

2.通过为 Proxy 类指定 ClassLoader 对象和一组 interface 来创建动态代理类；

// 通过 Proxy 为包括 Interface 接口在内的一组接口动态创建代理类的类对象

Class clazz = Proxy.getProxyClass(classLoader, new Class[] { Interface.class, ... });

3.通过反射机制获得动态代理类的构造函数，其唯一参数类型是调用处理器接口类型；

// 通过反射从生成的类对象获得构造函数对象

Constructor constructor = clazz.getConstructor(new Class[] { InvocationHandler.class });

4.通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数被传入。

// 通过构造函数对象创建动态代理类实例

Interface Proxy = (Interface)constructor.newInstance(new Object[] { handler });

实际使用过程更加简单，因为 Proxy 的静态方法 newProxyInstance 已经为我们封装了步骤 2 到步骤 4 的过程，所以简化后的过程如下

// InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理类到委托类的分派转发

InvocationHandler handler = new InvocationHandlerImpl(..); //在这个函数中具体写方法的实现。

// 通过 Proxy 直接创建动态代理类实例

Interface proxy = (Interface)Proxy.newProxyInstance( classLoader, new Class[] { Interface.class }, handler );

例子可查看：<http://www.cnblogs.com/flyoung2008/archive/2013/08/11/3251148.html>

### 面向切面编程

Spring的面向切面编程就是运用动态代理来实现的。

## Java访问权限

Java有四种访问权限， 其中三种有访问权限修饰符，分别为private，public和protected，还有一种不带任何修饰符。

1. private: Java语言中对访问权限限制的最窄的修饰符，一般称之为“私有的”。被其修饰的类、属性以及方法只能被该类的对象访问，其子类不能访问，更不能允许跨包访问。
2. default：即不加任何访问修饰符，通常称为“默认访问模式“。该模式下，只允许在同一个包中进行访问。
3. protected: 介于public 和 private 之间的一种访问修饰符，一般称之为“保护形”。被其修饰的类、属性以及方法只能被类本身的方法及子类访问，即使子类在不同的包中也可以访问。
4. public： Java语言中访问限制最宽的修饰符，一般称之为“公共的”。被其修饰的类、属性以及方法不仅可以跨类访问，而且允许跨包（package）访问。

下面用表格的形式来展示四种访问权限之间的异同点，这样会更加形象。表格如下所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **同一个类** | **同一个包** | **不同包的子类** | **不同包的非子类** |
| Private | √ |  |  |  |
| Default | √ | √ |  |  |
| Protected | √ | √ | √ |  |
| Public | √ | √ | √ | √ |

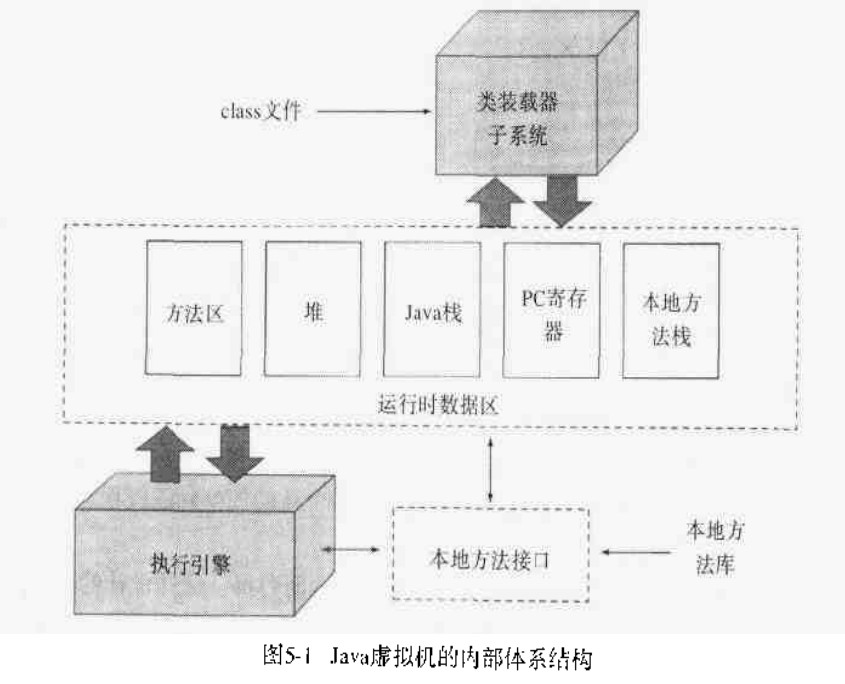
构造方法可以处于public、protected、private和默认四种访问级别之一。

1. 当构造方法为private级别时，意味着只能在当前类访问它，不能被继承，不能被其他程序用new创建实例对象。

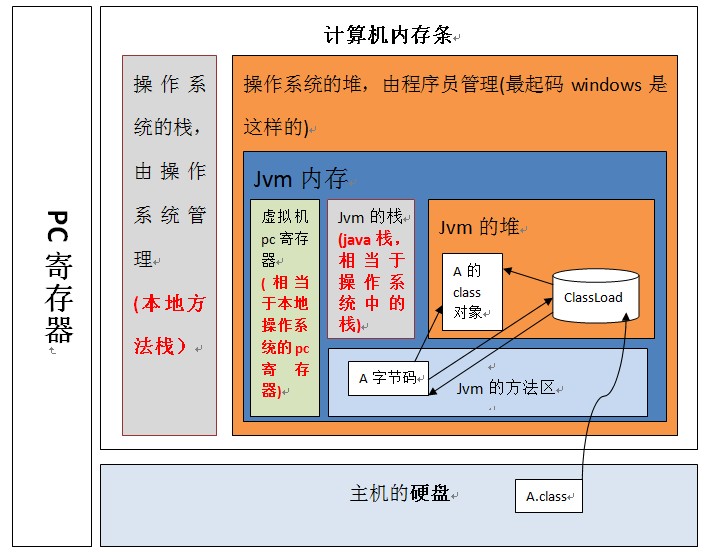
可以对比其他几种修饰符的作用：abstract修饰的类，不允许被实例化，这点和private修饰构造方法相同，但abstract修饰的类可以被继承，拥有子类，可以创建子类的实例；final类禁止被继承，这点和private修饰构造方法相同，但是final类可以用new创建实例对象。

# 5 . JAVA内存模型

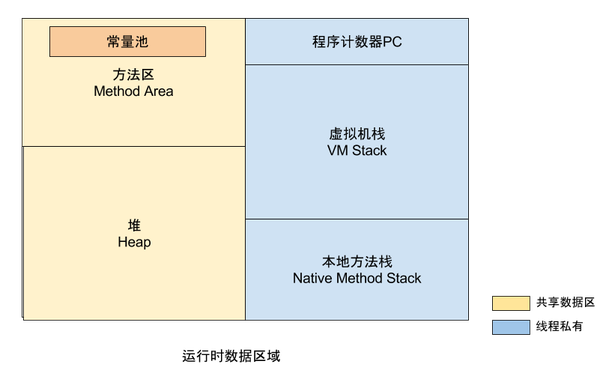
## Java虚拟机的体系结构



## Java虚拟机内存模型



## 运行时数据区域



### 程序计数器

此内存区域是java虚拟机中唯一一个没有规定任何OutOfMemoryError情况的区域

### Java虚拟机栈

线程私有

虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会穿件一个栈帧用于存储局部变量表、操作数栈、动态连接、方法出入口等信息。每一个方法从调入直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

在java虚拟中规范中，对这个区域规定了两种异常情况：

如果线程请求的栈的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；

当虚拟机栈可以动态扩展，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

### 本地方法栈

与虚拟机栈类似，本地方法栈为虚拟机使用到的Native方法服务。与虚拟机栈一样，本地展区域也会抛出StackOverfowError和OutOfMemoryError异常。

### Java堆

1.Java堆是被所有线程贡献的一块内存区域，在虚拟机启动时创建。

2.Java堆是垃圾回收器管理的主要区域。垃圾回收机制，下一章我会详细讲解。

3.从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Alocation Buffer, TLAB）。

4.通过-Xmx和-Xms控制。

5.如果在堆中没有内存完成实例分配，并且堆也无法在扩展时，将会抛出OutOfMemoryError异常。

### 方法区

方法区与Java堆一样，是各个线程共享的内存区域。被用于存取已被虚拟机加载的类信息、常量、静态变量、即时编译器变异后的代码等数据。

当方法去无法满足内存分配需求时，将会抛出OutOfMemoryError。

运行时常量池是方法区的一部分。用于存放编译期产生的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量区存放。

### 直接内存

直接内存不是虚拟机运行数据的一部分，但是这部分内存也被频繁使用，而且也可能导致OOM异常出现。

### 对象的创建

1.虚拟机遇到一条new指令时，首先将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的引用，并检查这个符号代表的类是否已经被加载、解析和初始化过。如果没有，那么必须先执行相应的类加载过程。

2.通过类加载检查通过后，接下来虚拟机将为新生对象分配内存。

3.内存分配完成后，虚拟机将需要对分配到的内存空间都初始化为零值。

4.接下来虚拟机要对对象进行必要的设置，如对象是那个类的实例，如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄。这些存放在对象的对象头中。

5.上述工作完成以后，从虚拟机的角度，一个新的对象已经产生，但是从java程序的角度上来看，对象才刚刚开始，init方法还没有执行，所有的字段为零.故执行new指令后接着执行init方法，把对象按照程序员的意愿初始化，这样一个真正的可用的对象才算完全生产出来。

### OutOfMemoryError

1. Java堆溢出

Java堆用于存储对象实例，只要不断创建新对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么对象数量达到最大堆容量后就会产生内存溢出。

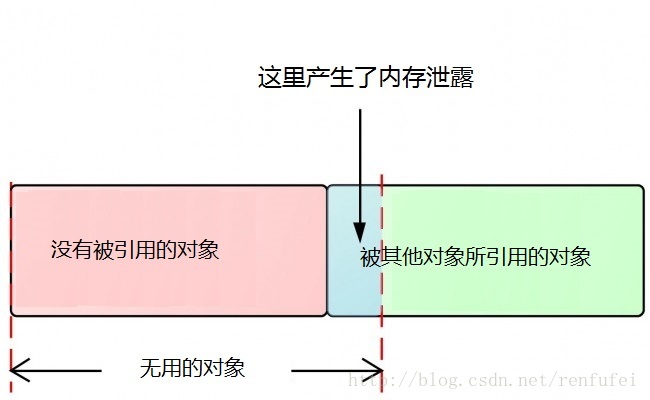
1. 虚拟机栈和本地方法栈溢出
2. 方法区和运行时常量池溢出

String.intern()在JDK1.6会把首次遇到的字符串实例复制到永久代，返回的也是永久代中这个字符串实例的引用。而在JDK1.7的intern()的实现不会在复制实例，只是在常量池中记录首次出现的实例的引用，因此intern()返回的引用即为java堆中的引用。

### 内存泄漏

内存泄露的定义: 当某些对象不再被应用程序所使用,但是由于仍然被引用而导致垃圾收集器不能释放(Remove,移除)他们.

用白话来说就是: 该回收的内存没被回收,最后因为内存不够用而导致程序报错。



从图中可以看出,内存中存在着 有引用的对象,和无引用的对象. 无引用的对象将被垃圾收集器所回收,而有引用的对象则不会被当做垃圾收集. 因为没有任何其他对象所引用,所以无引用对象一定是不再使用的。 但是有一部分无用对象仍然被(无意中)引用着。这就是发生内存泄露的根源.

怎么防止内存泄露?

要防止内存泄露,下面是一些快速上手的实用技巧:

1. 当心集合类,比如 HashMap,ArrayList等,因为这是最容易发生内存泄露的地方.当集合对象被声明为static时,他们的生命周期一般和整个应用程序一样长。
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。
3. 注意事件监听和回调.当注册的监听器不再使用以后,如果没有被注销,那么很可能会发生内存泄露.
4. 各种连接。比如数据库连接，网络连接，IO连接，除非调用起close()方法将期关闭，否则是不会自动被GC回收的。
5. 内部类和外部模块的引用。
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，

# 垃圾回收机制

上一章介绍了内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈3个区域为线程私有，随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈入栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的，因此这几个区域的内存分配和回收都具备确定性，在这几个区域就不需要过多的考虑回收的问题，因为方法结束或者线程结束时，内存就跟随着回收了。本章考虑的内存回收主要在Java堆和方法区。

## 判断对象是否还活着。

1. 引用计数法
2. 可达性分析算法

通过一系列的成为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

GC Roots的对象：

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（Native方法）的引用的对象

## 引用

1. 强引用，平常普通引用
2. 软引用，在系统将要发生内存溢出之前，将会把这些对象列入回收范围之内进行第二次回收，如果还没有足够的内存，才会抛出内存溢出异常。
3. 弱引用，被若引用关联的对象只能生存到下一次垃圾回收之前。在垃圾回收工作时，无论内存是否够用，都会回收掉被若引用关联的对象。
4. 虚引用，一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。

## 生存还是死亡

一个对象实例被回收要经过两次标记：

1. 当对象可达性分析后发现没有与GC Roots相连的链路，那么他将会被第一次标记并且进行第一次筛选，筛选条件是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者该方法已经被虚拟机调用过，那么两种情况视为没有必要执行。

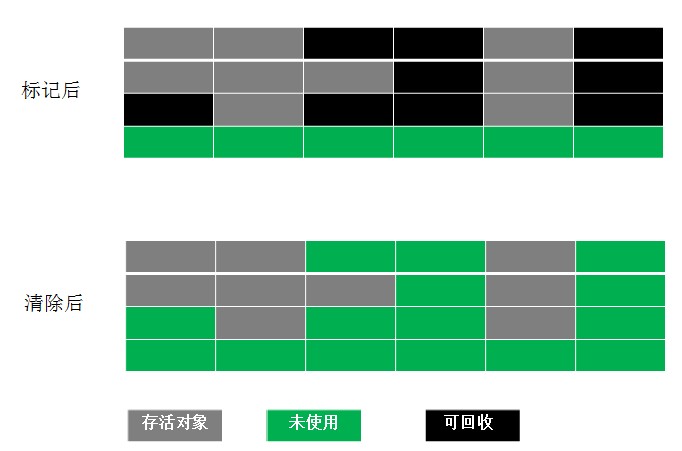
（当对象没有到大GC Roots的可达性同路，它将会被第一次标记）

1. 如果对象被判定有必要执行finalize（）方法，那么这个对象会被放置在F-Queue队列中，Finalize()方法是对象逃脱死亡最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模的标记。回收。

## 垃圾回收算法

### 1.Mark-Sweep（标记-清除）算法

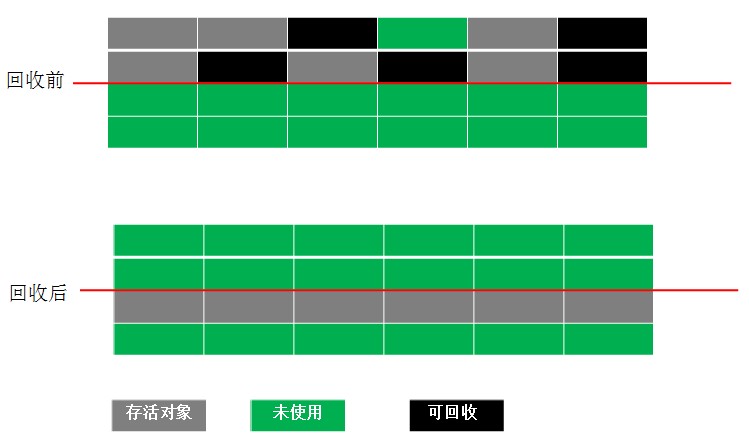
　　这是最基础的垃圾回收算法，之所以说它是最基础的是因为它最容易实现，思想也是最简单的。标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。具体过程如下图所示：



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

### 2.Copying（复制）算法

为了解决Mark-Sweep算法的缺陷，Copying算法就被提了出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。具体过程如下图所示：

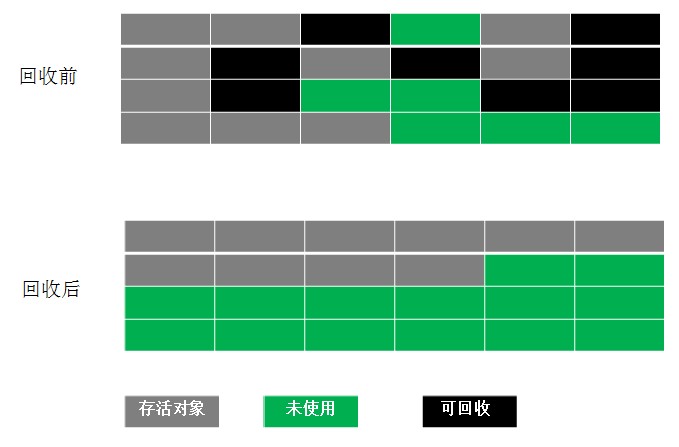


这种算法虽然实现简单，运行高效且不容易产生内存碎片，但是却对内存空间的使用做出了高昂的代价，因为能够使用的内存缩减到原来的一半。

　　很显然，Copying算法的效率跟存活对象的数目多少有很大的关系，如果存活对象很多，那么Copying算法的效率将会大大降低。

### 3.Mark-Compact（标记-整理）算法

　　为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。具体过程如下图所示：



### 4.Generational Collection（分代收集）算法

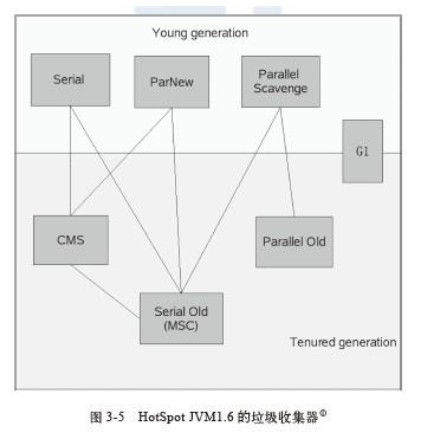
　　分代收集算法是目前大部分JVM的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。

　　目前大部分垃圾收集器对于新生代都采取Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照1：1的比例来划分新生代的空间的，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。

　　而由于老年代的特点是每次回收都只回收少量对象，一般使用的是Mark-Compact算法。

　　注意，在堆区之外还有一个代就是永久代（Permanet Generation），它用来存储class类、常量、方法描述等。对永久代的回收主要回收两部分内容：废弃常量和无用的类。

## 垃圾收集器



主要讲解CMS收集器，由上图可知，可以和CMS搭配实用的收集器有Serial和ParNew

Serial收集器是一个单线程的收集器，在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。（Stop the World）,新生代采用复制算法，老年代采用标记整理算法。

优点：简单高效。Serial收集器对于运行在Client模式下的虚拟机来说是一个很好的选择。

ParNew收集器起始就是Serial收集器的多线程版本。Server模式下的虚拟机中首选的新生代收集器。多CPU的环境下有较好的性能。

CMS收集器

CMS收集器是一种以获取最短回收暂停时间为目标的收集器。（Concurrent Mark Sweep ）从名字上可见CMS收集器是基于标记-清除算法实现的。整个过程分为4个步骤：

1. 初始标记 。这个步骤需要“Stop The World”，仅仅是标记GC Roots能直接关联的对象，速度非常快。
2. 并发标记。进行GC Roots Tracing的过程。
3. 重新标记。这个步骤需要“Stop The World”，为了修正并发标记期间用户程序继续运作而导致标记产生变动的那一部分对象的标记记录。这一阶段停顿时间比初始标记时间长，但远比并发标记时间短。
4. 并发清除。

耗时最长的并发标记和并发清除阶段都可以和用户线程一起工作，所以，从整体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

缺点：

1. CMS收集器对CPU资源非常敏感。在并发阶段，他虽然不会用户线程停顿，但是会因为占用了一部分线程导致应用程序变慢，总体吞吐量会降低。
2. CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败导致另一次Full GC的产生。浮动垃圾就是伴随这程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只能留待下次GC时清理。同时，垃圾收集阶段用户线程还要进行，还要预留足够的内存空间给用户线程实用，因此CMS不能像其他的收集器那样等到老年代几乎被填满在进行收集，需要预留一部分空间来提供并发收集时程序运作使用。如果CMS运行期间预留的内存无法满足程序的需要，就会出现一次“Concurrent Mode Failure”失败，这时，虚拟机紧急预案，临时启动Serial Old收集器收集，这样停顿时间就会很长。
3. CMS是一款基于“标记-清除”算法实现的收集器，那就意味着收集结束时会有大量的空间碎片产生。往往会出现老年带还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不出发一次Full GC。CMS提供机制，用于在CMS收集器顶不住要进行Full GC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片没有了，但是停顿时间不长。

## Minor GC、Major GC和Full GC之间的区别

### Minor GC

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC。这一定义既清晰又易于理解。但是，当发生Minor GC事件的时候，有一些有趣的地方需要注意到：

1. 当 JVM 无法为一个新的对象分配空间时会触发 Minor GC，比如当 Eden 区满了。所以分配率越高，越频繁执行 Minor GC。
2. 内存池被填满的时候，其中的内容全部会被复制，指针会从0开始跟踪空闲内存。Eden 和 Survivor 区进行了标记和复制操作，取代了经典的标记、扫描、压缩、清理操作。所以 Eden 和 Survivor 区不存在内存碎片。写指针总是停留在所使用内存池的顶部。
3. 执行 Minor GC 操作时，不会影响到永久代。从永久代到年轻代的引用被当成 GC roots，从年轻代到永久代的引用在标记阶段被直接忽略掉。
4. **质疑常规的认知，所有的 Minor GC 都会**触发**“全世界的暂停（stop-the-world）”**，停止应用程序的线程。对于大部分应用程序，停顿导致的延迟都是可以忽略不计的。其中的真相就 是，大部分 Eden 区中的对象都能被认为是垃圾，永远也不会被复制到 Survivor 区或者老年代空间。如果正好相反，Eden 区大部分新生对象不符合 GC 条件，Minor GC 执行时暂停的时间将会长很多。

所以 Minor GC 的情况就相当清楚了——**每次 Minor GC 会清理年轻代的内存**。

### Major GC vs Full GC

大家应该注意到，目前，这些术语无论是在 JVM 规范还是在垃圾收集研究论文中都没有正式的定义。但是我们一看就知道这些在我们已经知道的基础之上做出的定义是正确的，Minor GC 清理年轻带内存应该被设计得简单：

* **Major GC** 是清理老年代。
* **Full GC** 是清理整个堆空间—包括年轻代和老年代。

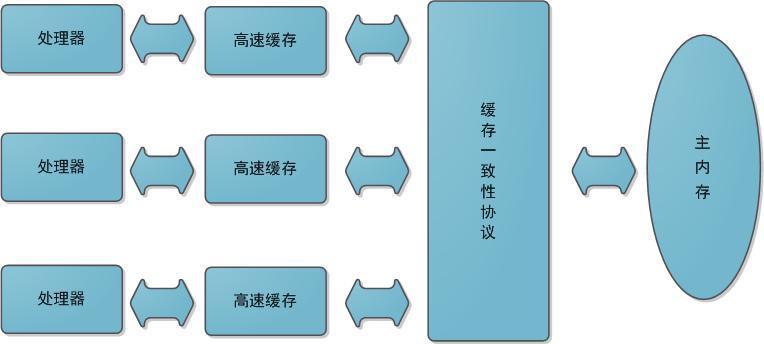
很不幸，实际上它还有点复杂且令人困惑。首先，许多 Major GC 是由 Minor GC 触发的，所以很多情况下将这两种 GC 分离是不太可能的。另一方面，许多现代垃圾收集机制会清理部分老年代空间，所以使用“cleaning”一词只是部分正确。

这使得我们不用去关心**到底是叫 Major GC 还是 Full GC，大家应该关注当前的 GC 是否停止了所有应用程序的线程，还是能够并发的处理而不用停掉应用程序的线程**。

# Java并发编程

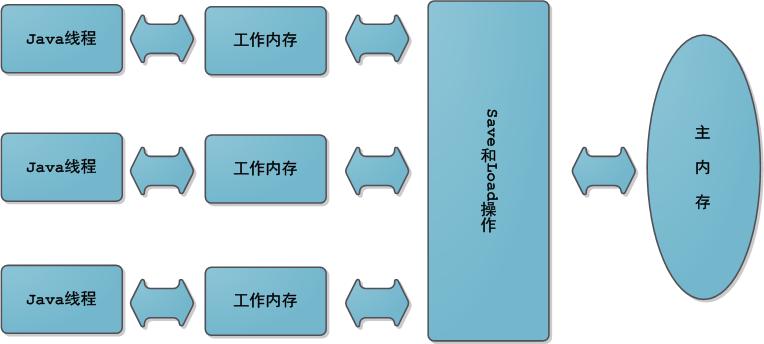
由于cpu速度和IO速度差几个数量级，所以现代计算即系统不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存来作为内存和处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束再从缓存中同步到内存之中，这样处理起就无需等待缓慢的内存读写。

基于高速缓存的存储交互很好的解决了处理器与内存的速度矛盾，但也带来了更高的复杂度，每个处理器都有自己的高速缓存，而他们又共享同一块主内存，当多个处理器的运算任务都设计同一块主内存区域时，将可能导致各自的缓存数据不一致。



## JVM主内存与工作内存

Java内存模型中规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存（可以与前面将的处理器的高速缓存类比），线程的工作内存中保存了该线程使用到的变量到主内存副本拷贝，线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间无法直接访问对方工作内存中的变量，线程间变量值的传递均需要在主内存来完成，线程、主内存和工作内存的交互关系如下图所示，和上图很类似。



## 同步机制

### Volatile

关键字volatile可以说是Java虚拟机提供的最轻量级的同步机制。

##### 当一个变量被定义为volatile之后，它将具备两种特性：

1. 保证此变量对所有线程的可见性。即当一条线程修改这个变量的值，新值对于其他线程来说是可以立即得知的。
2. 禁止指令重排序优化。

##### Volatile关键字进行读写时必须满足如下规则：

1. 在工作内存中，每次使用volatile变量都必须先从主存刷新最新的值，用于保证能看到其他线程对变量所做的修改后的值。
2. 在工作内存中，每次修改volatile变量后都必须立刻同步回主存中，用于保证其他线程可以看到自己对变量所做的修改。
3. Volatile修饰的变量不会被指令重排序优化，保证代码的执行顺序和程序的顺序相同。

##### 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。

#### 为什么要使用Volatile

Volatile变量修饰符如果使用恰当的话，它比synchronized的使用和执行成本会更低，因为它不会引起线程上下文的切换和调度。

### Synchronized

#### 同步的基础

Java中的每一个对象都可以作为锁。

对于同步方法，锁是当前实例对象。

对于静态同步方法，锁是当前对象的Class对象。

对于同步方法块，锁是Synchonized括号里配置的对象。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。那么锁存在哪里呢？锁里面会存储什么信息呢？

同步的原理

JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，但两者的实现细节不一样。

代码块同步是使用monitorenter和monitorexit指令实现，monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处， JVM要保证每个monitorenter必须有对应的monitorexit与之配对。线程执行到monitorenter 指令时，将会尝试获取对象所对应的 monitor 的所有权，即尝试获得对象的锁。

而方法同步是使用另外一种方式实现的，细节在JVM规范里并没有详细说明，但是方法的同步同样可以使用这两个指令来实现。任何对象都有一个 monitor 与之关联，当且一个monitor 被持有后，它将处于锁定状态。

### Final

#### final关键字的好处：

1. final关键字提高了性能。JVM和Java应用都会缓存final变量。
2. final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。
3. 使用final关键字，JVM会对方法、变量及类进行优化。

#### 不可变类

创建不可变类要使用final关键字。不可变类是指它的对象一旦被创建了就不能被更改了。String是不可变类的代表。不可变类有很多好处，譬如它们的对象是只读的，可以在多线程环境下安全的共享，不用额外的同步开销等等。

#### 关于final的重要知识点

1. final关键字可以用于成员变量、本地变量、方法以及类。
2. final成员变量必须在声明的时候初始化或者在构造器中初始化，否则就会报编译错误。
3. final变量不能再次赋值。
4. final本地变量必须在声明时赋值。
5. 在匿名类中所有变量都必须是final变量。
6. final方法不能被重写。
7. final类不能被继承。
8. final关键字不同于finally关键字，后者用于异常处理。
9. final关键字容易与finalize()方法搞混，后者是在Object类中定义的方法，是在垃圾回收之前被JVM调用的方法。
10. 接口中声明的所有变量本身是final的。
11. final和abstract这两个关键字是反相关的，final类就不可能是abstract的。
12. final方法在编译阶段绑定，称为静态绑定(static binding)。
13. 没有在声明时初始化final变量的称为空白final变量(blank final variable)，它们必须在构造器中初始化，或者调用this()初始化。不这么做的话，编译器会报错“final变量(变量名)需要进行初始化”。
14. 将类、方法、变量声明为final能够提高性能，这样JVM就有机会进行估计，然后优化。
15. 按照Java代码惯例，final变量就是常量，而且通常常量名要大写：

### 原子性、可见性、有序性

#### ####原子性

1. 我们基本上可以认为基本数据类型的访问读写是具备原子性的。
2. 如果我们需要更大范围的原子性保证，Java内存提供了lock和unclock操作来满足这种需求，尽管虚拟机未把lock和unclock操作开放给用户，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐式地实用这两个操作，这两个字节码指令反映到java代码中就是同步块----synchronized,因此synchronized块之间的操作也是具备原子性的。
3. CAS

CAS:Compare and Swap, 翻译成比较并交换。

比较并设置。用于在硬件层面上提供原子性操作。在 Intel 处理器中，比较并交换通过指令cmpxchg实现。比较是否和给定的数值一致，如果一致则修改，不一致则不修改。

CAS有3个操作数，内存值V，预期值A，新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

*compareAndSet(current, next) current为预期值，next为新值*

##### concurrent包的实现

由于java的CAS同时具有 volatile 读和volatile写的内存语义，因此Java线程之间的通信现在有了下面四种方式：

1. A线程写volatile变量，随后B线程读这个volatile变量。
2. A线程写volatile变量，随后B线程用CAS更新这个volatile变量。
3. A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量。
4. A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量。

Java的CAS会使用现代处理器上提供的高效机器级别原子指令，这些原子指令以原子方式对内存执行读-改-写操作，这是在多处理器中实现同步的关键（从本质上来说，能够支持原子性读-改-写指令的计算机器，是顺序计算图灵机的异步等价机器，因此任何现代的多处理器都会去支持某种能对内存执行原子性读-改-写操作的原子指令）。同时，volatile变量的读/写和CAS可以实现线程之间的通信。把这些特性整合在一起，就形成了整个concurrent包得以实现的基石。如果我们仔细分析concurrent包的源代码实现，会发现一个通用化的实现模式：

1. 首先，声明共享变量为volatile；
2. 然后，使用CAS的原子条件更新来实现线程之间的同步；
3. 同时，配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

AQS，非阻塞数据结构和原子变量类（java.util.concurrent.atomic包中的类），这些concurrent包中的基础类都是使用这种模式来实现的，而concurrent包中的高层类又是依赖于这些基础类来实现的。从整体来看，concurrent包的实现示意图如下：



#### 可见性

可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。

除了volatile之外，synchronized和final也可以实现可见性。同步块的可见性是由“对一个变量执行unclock操作之前，必须先把此变量同步回主内存中”实现的。

#### 有序性

在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

下面就来具体介绍下happens-before原则（先行发生原则）：

* 1. 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
  2. 锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作
  3. volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
  4. 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C
  5. 线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作
  6. 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生
  7. 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行
  8. 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

## 并发

### 互斥并发（阻塞同步）

最基本的互斥并发就是synchronized。

Java的线程是映射到操作系统的原生线程之上的，如果要阻塞或者唤醒一个线程，都需要操作系统帮忙完成，这就需要从用户态转换成核心态，因此状态转换需要耗费很多的处理器时间。

除了synchronized外java.util.concurrent包下的重入锁（ReentrantLock）也可以实现同步。

### 非阻塞同步

互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题。

基于冲突检测的乐观并发策略，先进行操作，如果没有其他线程争用共享数据，那么操作就成功了；如果有争用，产生冲突，就再采取补偿操作（最常见的补偿措施就是不断尝试，知道成功）。

CAS就是非阻塞同步。（ABA问题）

## 锁优化

### 自旋锁和自适应自旋

解决线程阻塞和唤醒所带来的性能问题。在许多应用上，共享数据的锁定只会持续很短的时间，为这段时间去挂起和恢复线程不值得。我们可以让后面请求锁的线程“稍等一下”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。

自适应自旋意味着自旋的时间不固定。

### 轻量级锁

轻量级锁并不是用来代替重量级锁的，本意是在没有多线程竞争的前提下减少传统的重量级锁使用操作系统互斥量产生的性能消耗。

如果没有竞争，轻量级锁使用CAS操作避免了使用互斥量的开销，但是如果存在竞争，出来互斥量的开销外，还额外发生了CAS操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢。

### 偏向锁

偏向锁的目的是消除数据在无竞争情况下的同步原语，进一步提高性能。如果说轻量级锁是在无竞争的情况下使用CAS操作去消除同步实用的互斥量，那偏向锁就是在无竞争的情况下把整个同步都消除掉，连CAS操作都不要。

## 线程池

合理利用线程池能够带来三个好处。

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

第三：提高线程的可管理性。

线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

### 线程池的创建

我们可以通过ThreadPoolExecutor来创建一个线程池。

new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, milliseconds,runnableTaskQueue, threadFactory,handler);

创建一个线程池需要输入几个参数：

corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。

maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。

TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。

runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。

ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。

SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。

PriorityBlockingQueue：一个具有优先级得无限阻塞队列。

ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字，Debug和定位问题时非常又帮助。

RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。以下是JDK1.5提供的四种策略。n AbortPolicy：直接抛出异常。

CallerRunsPolicy：只用调用者所在线程来运行任务。

DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。

DiscardPolicy：不处理，丢弃掉。

当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务。

### 线程池的分析

流程分析：线程池的主要工作流程如下图：



从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

1. 首先线程池判断基本线程池是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
2. 其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
3. 最后线程池判断整个线程池是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

### 合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

1. 任务的性质：CPU密集型任务，IO密集型任务和混合型任务。
2. 任务的优先级：高，中和低。
3. 任务的执行时间：长，中和短。
4. 任务的依赖性：是否依赖其他系统资源，如数据库连接。

任务性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务配置尽可能少的线程数量，如配置Ncpu+1个线程的线程池。IO密集型任务则由于需要等待IO操作，线程并不是一直在执行任务，则配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

优先级不同的任务可以使用优先级队列PriorityBlockingQueue来处理。它可以让优先级高的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交SQL后需要等待数据库返回结果，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用CPU。

建议使用有界队列，有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。

## 相关问题

1. sleep()和wait()分别是哪个类的方法，有什么区别？

对于sleep()方法，我们首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

不同点：

1.Thread类的方法：sleep(),yield()等 ，Object的方法：wait()和notify()等

2.每个对象都有一个锁来控制同步访问。Synchronized关键字可以和对象的锁交互，来实现线程的同步。

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

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

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

1. sleep()方法：sleep()允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。

2.yield() 方法：yield() 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。

1. wait() 和 notify() 方法：两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用。

1 sleep() 使当前线程进入停滞状态，所以执行sleep()的线程在指定的时间内肯定不会执行, 同时sleep函数不会释放锁资源.

2 yield() 只是使当前线程重新回到可执行状态，所以执行yield()线程有可能在进入到可执行状态后马上又被执行. 只能使同优先级的线程有执行的机会。同样, yield()也不会释放锁资源.

　　sleep和yield的区别在于, sleep可以使优先级低的线程得到执行的机会, 而yield只能使同优先级的线程有执行的机会.

1. synchronized与lock的区别

主要相同点：Lock能完成Synchronized所实现的所有功能。

主要不同点：代码写法上有区别，lock表现为API层面的互斥锁，synchronized表现为原生语法层面的互斥。相对比synchronized，ReentratLock增加了一些高级功能，主要有以下三个：等待可中断、可实现公平锁以及锁可以绑定多个条件。

Synchronized会自动释放锁，但是Lock一定要求程序员手工释放，并且必须在finally从句中释放。在JDK1.6以后两者的性能基本完全持平。

1. 线程同步与阻塞的关系？同步一定阻塞吗？阻塞一定同步吗？

同步和阻塞没有无关系。

同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)

所谓同步，就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。

而异步则是相反，\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果，等待“回调”。

阻塞与非阻塞

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。

非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

1. 进程间的通信方式

文件、管道、共享内存空间：

1.文件方式的进程通信就是通过读写同一个文件来完成共享数据。一个进程将数据写入文件，另一个进程再将数据从文件中读出。

客户端能够读取文件，且能够多个客户端同时读取，系统允许同时打开一个文件。

服务器端通过清空重写的方式对文件内容进行修改

问题：不能避免在文件尚未进行完写操作时就有进程读取文件内容，这时会造成读取的文件内容为空或者是读出文件中的部分数据。

解决方案：可以通过文件锁的方式进行控制。

2.命名管道：常规的管道只能连接相关进程，由进程创建并由最后一个进程关闭。而命名管道可以连接不相关的进程。它是个FIFO先进先出队列，即使没有进程，命名管道依然可以存在，它不依赖于进程。服务器将字节写入队列，客户端从队列头部移出字节。服务器必须重写数据。它没有竞争的问题，在没有超过管道最大长度的时候，read和write都是原子操作，先将管道清空，然后再将管道写满，在读者和写者连通之前系统内核将进程挂起。

3.共享内存段：不依赖于进程的存在而存在。共享内存段拥有权限系统来控制进程各自的访问权限。多个客户端都可以同时从内存中读数据。

问题：如果读取是在服务器向内存中写数据的时候进行的，有可能既读到旧数据又读到新数据。

解决方案：信号量机制。

比较：

1.文件和命名管道消耗操作多。

2.文件和共享内存是无连接的

3.命名管道和共享内存只能在本机中使用

4.使用共享内存和文件要比使用管道麻烦，要处理竞态

http://my.oschina.net/u/248570/blog/53226

<http://blog.chinaunix.net/uid-23381466-id-295750.html>

1. Java进程通信方式
2. 通过访问共享变量的方式(注:需要处理同步问题) 加锁，多线程编程
3. 通过管道流
4. wait()方法

wait()方法使得当前线程必须要等待，等到另外一个线程调用notify()或者notifyAll()方法。

当前的线程必须拥有当前对象的monitor，也即lock，就是锁。

1. notify()方法

notify()方法会唤醒一个等待当前对象的锁的线程。

　　如果多个线程在等待，它们中的一个将会选择被唤醒。这种选择是随意的，和具体实现有关。（线程等待一个对象的锁是由于调用了wait方法中的一个）。

　　被唤醒的线程是不能被执行的，需要等到当前线程放弃这个对象的锁。

1. 什么是死锁？JVM线程死锁，你该如何判断是因为什么？如果用VisualVM，dump线程信息出来，会有哪些信息？
2. 查看jvm虚拟机里面堆、线程的信息，你用过什么命令？我只用过图形界面VisualVM。。。
3. 对多线程有了解吗？在什么项目里用了多线程？在什么情况下用了多线程？java用过多线程吗？
4. 如何创建单例模式？说了双重检查，他说不是线程安全的。如何高效的创建一个线程安全的单例？
5. 问：程序中是如何实现并发的？
6. 问：为什么要实现内存模型？
7. ConcurrentHashMap有哪些问题？
8. 如何理解分布式锁？
9. 现在JVM中有一个线程挂起了，如何用工具查出原因？

可以使用jstack 打印堆信息

1. https://zebinlin.github.io/tags/#java

# 算法

## 排序算法

### 插入排序

•思想：每步将一个待排序的记录，按其顺序码大小插入到前面已经排序的字序列的合适位置，直到全部插入排序完为止。

•关键问题：在前面已经排好序的序列中找到合适的插入位置。

•方法：

–直接插入排序

–二分插入排序

–希尔排序

#### 直接插入排序

1. 基本思想：每步将一个待排序的记录，按其顺序码大小插入已经排好序的序列中（从后向前找到合适位置后）直到全部插入排序完为止。
2. 实例



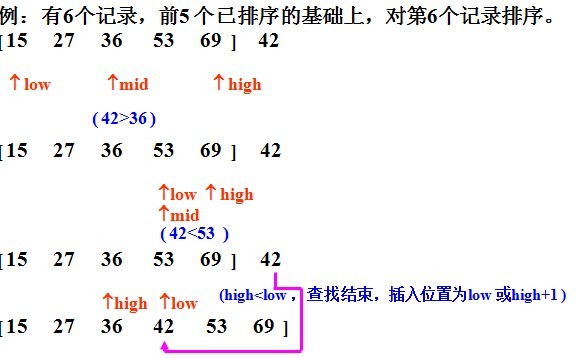
1. 代码：<http://www.cnblogs.com/liuling/p/2013-7-24-01.html>
2. 分析：

直接插入排序是稳定的排序。关于各种算法的稳定性分析可以参考http://www.cnblogs.com/Braveliu/archive/2013/01/15/2861201.html

文件初态不同时，直接插入排序所耗费的时间有很大差异。若文件初态为正序，则每个待插入的记录只需要比较一次就能够找到合适的位置插入，故算法的时间复杂度为O(n)，这时最好的情况。若初态为反序，则第i个待插入记录需要比较i+1次才能找到合适位置插入，故时间复杂度为O(n2)，这时最坏的情况。直接插入排序的平均时间复杂度为O(n2)。

#### 二分法插入排序（按二分法找到合适位置插入）

1. 二分法插入排序的思想和直接插入一样，只是找合适的插入位置的方式不同，这里是按二分法找到合适的位置，可以减少比较的次数。
2. 实例



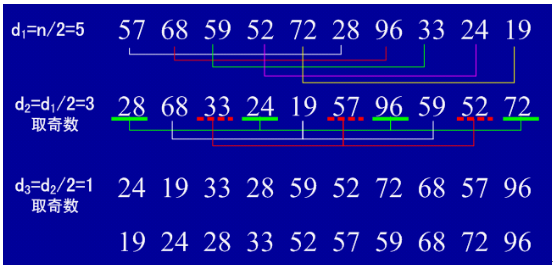
1. 分析

当然，二分法插入排序也是稳定的。

二分插入排序的比较次数与待排序记录的初始状态无关，仅依赖于记录的个数。当n较大时，比直接插入排序的最大比较次数少得多。但大于直接插入排序的最小比较次数。算法的移动次数与直接插入排序算法的相同，最坏的情况为n2/2，最好的情况为n，平均移动次数为O(n2)。

#### 希尔排序

1. 基本思想：先取一个小于n的整数d1作为第一个增量，把文件的全部记录分成d1个组。所有距离为d1的倍数的记录放在同一个组中。先在各组内进行直接插入排序；然后，取第二个增量d2<d1重复上述的分组和排序，直至所取的增量dt=1(dt<dt-l<…<d2<d1)，即所有记录放在同一组中进行直接插入排序为止。该方法实质上是一种分组插入方法。
2. 实例



1. 分析

　我们知道一次插入排序是稳定的，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以希尔排序是不稳定的。

　希尔排序的时间性能优于直接插入排序，原因如下：

（1）当文件初态基本有序时直接插入排序所需的比较和移动次数均较少。

　（2）当n值较小时，n和n2的差别也较小，即直接插入排序的最好时间复杂度O(n)和最坏时间复杂度0(n2)差别不大。

　（3）在希尔排序开始时增量较大，分组较多，每组的记录数目少，故各组内直接插入较快，后来增量di逐渐缩小，分组数逐渐减少，而各组的记录数目逐渐增多，但由于已经按di-1作为距离排过序，使文件较接近于有序状态，所以新的一趟排序过程也较快。

　　因此，希尔排序在效率上较直接插人排序有较大的改进。

希尔排序的平均时间复杂度为O(nlogn)。

### 选择排序

•思想：每趟从待排序的记录序列中选择关键字最小的记录放置到已排序表的最前位置，直到全部排完。

•关键问题：在剩余的待排序记录序列中找到最小关键码记录。

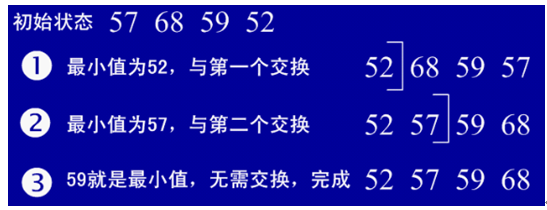
•方法：

–直接选择排序

–堆排序

#### 简单选择排序

1. 基本思想：在要排序的一组数中，选出最小的一个数与第一个位置的数交换；然后在剩下的数当中再找最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。
2. 实例



1. 分析

简单选择排序是不稳定排序

时间复杂度：O(n2)

#### 堆排序

1. 基本思想

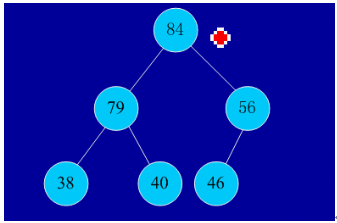
堆排序是一种树形选择排序，是对直接选择排序的有效改进。

　　堆的定义下：具有n个元素的序列 （h1,h2,...,hn),当且仅当满足（hi>=h2i,hi>=2i+1）或（hi<=h2i,hi<=2i+1） (i=1,2,...,n/2)时称之为堆。在这里只讨论满足前者条件的堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项（大顶堆）。完全二叉树可以很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。

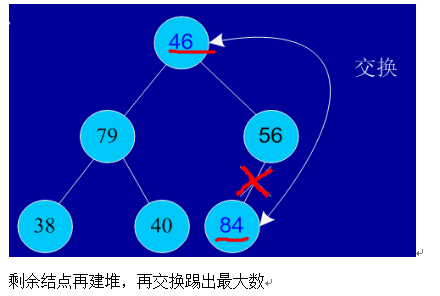
思想:初始时把要排序的数的序列看作是一棵顺序存储的二叉树，调整它们的存储序，使之成为一个 堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点交换。然后对前面(n-1)个数重新调整使之成为堆。依此类推，直到只有两个节点的堆，并对 它们作交换，最后得到有n个节点的有序序列。从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

1. 实例

初始序列：46,79,56,38,40,84



交换，从堆中踢出最大数



依次类推：最后堆中剩余的最后两个结点交换，踢出一个，排序完成。

1. 分析

堆排序也是一种不稳定的排序算法。

堆排序优于简单选择排序的原因：

直接选择排序中，为了从R[1..n]中选出关键字最小的记录，必须进行n-1次比较，然后在R[2..n]中选出关键字最小的记录，又需要做n-2次比较。事实上，后面的n-2次比较中，有许多比较可能在前面的n-1次比较中已经做过，但由于前一趟排序时未保留这些比较结果，所以后一趟排序时又重复执行了这些比较操作。

堆排序可通过树形结构保存部分比较结果，可减少比较次数。

堆排序的最坏时间复杂度为O(nlogn)。堆序的平均性能较接近于最坏性能。由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

### 交换排序

#### 冒泡排序

1. 基本思想：在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。
2. 冒泡排序是一种稳定的排序方法。

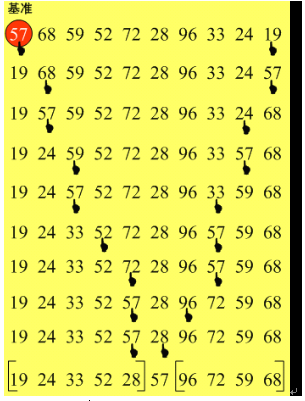
•若文件初状为正序，则一趟起泡就可完成排序，排序码的比较次数为n-1，且没有记录移动，时间复杂度是O(n)

•若文件初态为逆序，则需要n-1趟起泡，每趟进行n-i次排序码的比较，且每次比较都移动三次，比较和移动次数均达到最大值∶O(n2)

•起泡排序平均时间复杂度为O(n2)

#### 快速排序

1. 基本思想：选择一个基准元素,通常选择第一个元素或者最后一个元素,通过一趟扫描，将待排序列分成两部分,一部分比基准元素小,一部分大于等于基准元素,此时基准元素在其排好序后的正确位置,然后再用同样的方法递归地排序划分的两部分。
2. 实例



1. 分析

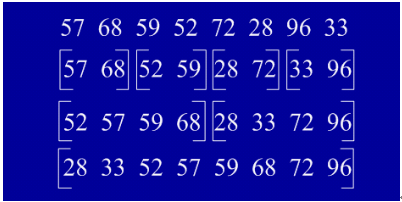
快速排序是不稳定的排序。

快速排序的时间复杂度为O(nlogn)。

当n较大时使用快排比较好，当序列基本有序时用快排反而不好。

### 归并排序

1. 基本思想:归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。
2. 实例



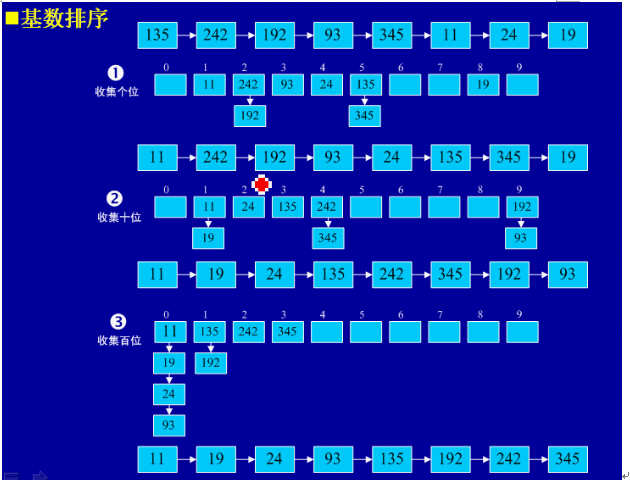
1. 归并排序是稳定的排序方法

归并排序的复杂度为O(nlogn)

速度仅次于快速排序，为稳定的排序算法，一般用于对总体无序，但是各子项相对有序的数列。

### 基数排序

1. 基本思想：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后,数列就变成一个有序序列。
2. 实例



1. 分析

基数排序是稳定的排序算法。

基数排序的时间复杂度为O(d(n+r)),d为位数，r为基数。

### 总结

#### 一、稳定性:

稳定：冒泡排序、插入排序、归并排序和基数排序

不稳定：选择排序、快速排序、希尔排序、堆排序

#### 二、平均时间复杂度

O(n^2):直接插入排序，简单选择排序，冒泡排序。

　　在数据规模较小时（9W内），直接插入排序，简单选择排序差不多。当数据较大时，冒泡排序算法的时间代价最高。性能为O(n^2)的算法基本上是相邻元素进行比较，基本上都是稳定的。

O(nlogn):快速排序，归并排序，希尔排序，堆排序。

其中，快排是最好的， 其次是归并和希尔，堆排序在数据量很大时效果明显。

#### 三、排序算法的选择

##### 1.数据规模较

##### 1）待排序列基本序的情况下，可以选择直接插入排序

##### 2）对稳定性不作要求宜用简单选择排序，对稳定性有要求宜用插入或冒泡

##### 2.数据规模不是很大

##### （1）完全可以用内存空间，序列杂乱无序，对稳定性没有要求，快速排序，此时要付出log（N）的额外空间。

##### （2）序列本身可能有序，对稳定性有要求，空间允许下，宜用归并排序

##### 3.数据规模很大

##### （1）对稳定性有求，则可考虑归并排序

##### （2）对稳定性没要求，宜用堆排序

##### 4.序列初始基本有序（正序），宜用直接插入，冒泡