数据结构

数据库分页SQL SQL优化

再过一遍缓存

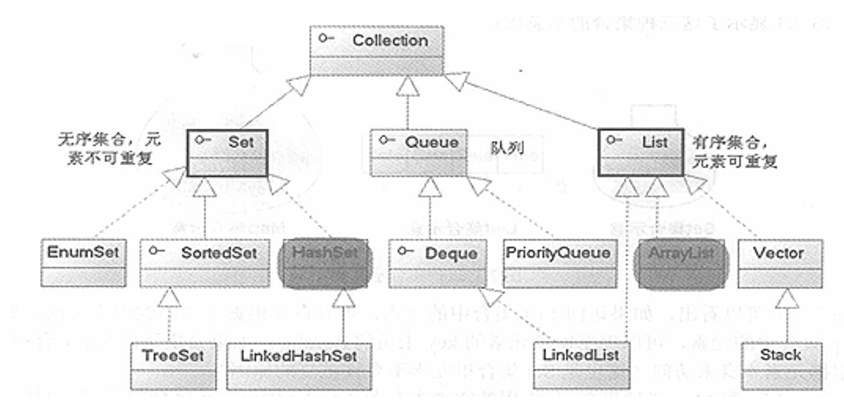
多线程和线程安全

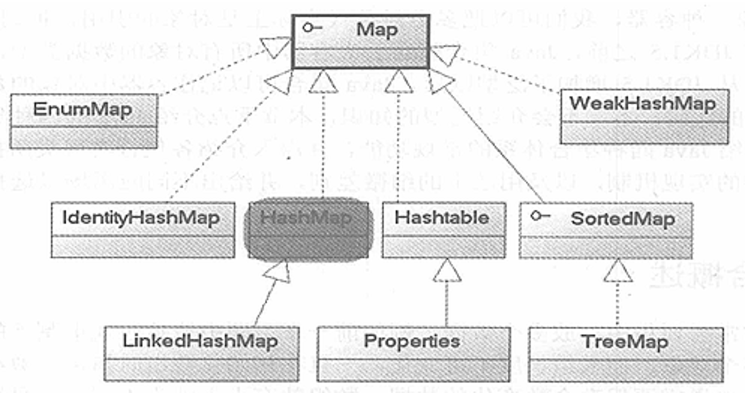
IO流 设计模式

Java的内存原理（后面再刷，不一定会问）

## Java数据集合框架

Collection 和 Map为根接口





## 数组、HashSet、ArrayList、HashMap、LinkedHashMap的比较

不考虑性能的话，键值对用HashMap，要去重的话HashSet，一般都用ArrayList。遍历的话，都可以通过增强for循环实现。

如果要实现带去重 又有顺序的话，可以用 LinkedHashMap，只要把value值设置为NULL或者其他的就可以了。

ArrayList和java数组相比的优势在于，java自动管理 arraylist的大小，而数组必须指定大小。

ArrayList和HashMap相比，一个存放键值对(java中一般封装一个多属性的对象)

ArryList和HashSet相比，HashSet有去重功能，但是没有根据下标获取元素的功能，但是contains方法更快，插入和删除更快。

LinkedHashMap 比HashMap多了一个顺序功能，是通过添加了一个 header(双向链表)的 Entry实现指针的功能，但是get方法不变，插入和删除原理不变只是要维护 header。

## HashMap的数据结构

结构、put、get、遍历、构造函数、resize

### 结构

实现了一个综合数组和单向链表的数据结构，在查询和插入以及内存消耗上面做了一个权衡。

因为内存足够的话，把loadfactor调小(0.75默认)，这样一个节点上面存在多个数据的概率就小，更像一个列表。查询和插入也更快，因为resize的其中一个必要条件是 数量>size\*loadfactor。

### Put

根据key值算 hashcode值（二进制运算算出来一个比较大的数），hashcode值个 数组的size-1 做与运算，算出它应该插入到那个index，然后再检查该位置是否有元素，如果空白，那么他就插入在这个位置，如果有的话，就遍历对比是否有 key equals的情况，有的话就替换，没有的话再考虑是否触发扩容， total>loadfactor\*size

## ArrayList的数据结构

默认大小给的是10，如果扩容的话，大小约为1.5倍，是指向新的一个对象而不是原list变长。

大的list的话，可以通过给定init大小来初始化list的长度。

插入 和删除 会根据下标 移动下标后续的元素。

## HashSet 的数据结构

基于HashMap封装的一个非键值对的集合，特点是除了具有HashMap的增删改查快的特点之外，还兼具去重功能，但是不是键值对。

Add的话，就是把 元素作为 key值，value值为null，插入到hashMap中，相同元素肯定key相同，所以就可以去重了。

比较 ArrayList，查询 因为无序无法根据下标获取对象，但是contains方法的话，ArrayList是全部遍历，而HashSet是根据HashCode值查找，所以HashSet更快。

但是插入删除都更快，而且自带去重功能

## LinkedHashMap的数据结构

对hashmap进一步改进

1. 重写了Entry，添加了 before after的指针，从而无论从哪个元素开始找，都能找到全部的元素。
2. 添加了成员变量 header(Entry，但是 hash -1 key next value都是空的，用来记录当前的 before 和after指针作用。
3. 遍历的时候，只需要从 header里面挨个取数据即可。不需要像hashMap一样取数据。就是链式取数据。
4. 比较HashMap而言，多了一个Header维护顺序的开销，为遍历提供了顺序支持。
5. Get方法还是按照 hashMap的方法

## 数据库分页SQL

### Oracle

select \*

from (select a.id,a.usernam,…, rownum as rn from expense a where rownum <= 40) k

where k.rn > 20

### mysql

select \* from books limit 20,40

### sql Server

2012以上版本 offset 开始且不含的开始index 例如第6条开始取 取10个

select \* from SFA\_PROJECT\_MASTER order by Id offset 5 row fetch next 10 row only

还有TOP函数

### 通用SQL

ROW\_NUMBER() OVER

SELECT J.\*

FROM (SELECT ROW\_NUMBER() OVER(ORDER BY A.ID) AS NUM,A.\* FROM EXPENSE A) J

where j.NUM>5 and J.Num<=15

## 缓存系列

### Mybatis缓存

默认开始一级缓存，一个session里面向数据库中查询相同参数的sql，会直接从缓存中获取。但是session间不共享。

二级缓存就是通过配置缓存策略，使 session间共享缓存。

### Hiernate缓存

一级缓存同 mybatis，二级缓存依赖不同的 缓存插件，默认是 ehcache来实现

### Ehcache

纯java缓存框架，支持数据持久化，可以直接通过插件与mybatis 集成，配置在 mapper当中，对sql查询结果进行缓存维护，hibernate一样，也可以Ehcache直接与 spring整合，通过注解的方式，对整个方法返回结果进行缓存，而不仅仅是对sql查询结果进行缓存。还支持灵活的 keyGenerator，避免缓存错误。默认的key规则是 参数名拼参数值，例如 id1 void get(int id);

优点是纯java，不依赖其他应用，缺点是无法进行分布式缓存。

### Redis

特点同Ehcache，区别是 Redis是独立的第三方应用程序，需要单独起Redis，类似与用内存做存储的Mysql数据库，支持分布式。支持持久化，支持多数据结构。

### Mercache

同Redis, 有持久化需求或者对数据结构和处理有高级要求的应用，选择redis，其他简单的key/value存储，选择memcache。

## 算法系列

思路:算法真的很复杂哎，稍微了解一下原理就可以了，那么认真也没什么必要啊，到时候真要用到的时候再研究也不迟啊

算法考虑的方面:比较的次数、交换的次数和稳定性、内存的使用

主要研究三种排序:冒泡排序、插入排序，快速排序、堆排序，其中冒泡排序和堆排序比较的次数都是一样，数据量较小时，用 冒泡和选择排序比 快速排序和堆排序要好。



### 时间复杂度

相当于 当N趋近无穷大的时候，去掉常量C的简值， 例如 要计算 2n^2+n+1次计算 ，则 O（N^2）复杂维度

当计算次数是常数，跟N无关的时候，O(1) 复杂度就是1了。

嵌套2层for 循环的话，就是 O（n^2） 了

### 冒泡排序:

每一次挨个遍历(N-1次遍历)，标记最大的一个数的下标位置，最后一次调整即可

### 快速排序

选取一个基准值（一般是第一个），进行比较分组，把比基准值小的所有元素分为1组，比基准值大的所有元素分为另外一组，这样 基准值的位置就确定了，再递归，两个分好的小组再进行相同比较。

### 堆排序

把数组构建成一个 二叉树堆模型， 设计一个比较算法，把最大的至于堆顶，然后 取出堆顶，把最后面一个元素再至于堆顶，再进行排序，往复。

Ri 父节点 大于 R2i+1 和 R2i+2 两个子节点

### 插入排序

维护一个递增的 升序数组，如果后面比前面大，则停止搜索，j+1位置就是 i的位置，缺点:比较次数不确定，

比较的次数基于已有的数据的次序，越有序，排序越快

最小比较次数：n-1次，也就是本身就是一个 升序的数组，

最多比较次数： n(n-1)/2次

## 设计模式

参考blog: <http://www.cnblogs.com/jingmoxukong/p/4015675.html> 静默虚空大神

### 单例模式

<http://www.cnblogs.com/jingmoxukong/p/4015675.html>

1、一个类只能有一个实例。

需要定义一个该类的静态私有变量，使这个类的所有对象都共用这个实例。

2、实例必须由类自行创建。

单例模式的类只能提供私有的构造函数。如此，才能保证外部无法实例化这个类的对象。

3、必须提供获取实例的方法。

单例模式的类必须提供一个公共的静态函数用于创建或获取它本身的静态私有对象。

懒汉式、饿汉式、双重锁形式



## JVM相关和JAVA内存空间

权威BLOG，最详细易懂

<http://www.cnblogs.com/therunningfish/archive/2016/05/24/5523995.html#commentform>

一个共享的 堆 和持久带，和多个 线程内存区。NIO的直接内存区

**直接内存**

直接内存并不是虚拟机运行时数据区的一部分。

在NIO中，引入了一种基于通道和缓冲区的I/O方式，它可以使用native函数直接分配堆外内存，然后通过一个存储在java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。



**堆内存分配**

       JVM初始分配的内存由-Xms指定，默认是物理内存的1/64；JVM最大分配的内存由-Xmx指 定，默认是物理内存的1/4。默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制。因此服务器一般设置-Xms、-Xmx相等以避免在每次GC 后调整堆的大小。对象的堆内存由称为垃圾回收器的自动内存管理系统回收。

堆内存是线程共享的。

### Perment Space

**非堆内存分配**  
      JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4。

**(1)持久带**

JVM用持久带（Permanent Space）实现方法区，主要存放所有已加载的类信息，方法信息，常量池等等。

可通过-XX:PermSize和-XX:MaxPermSize来指定持久带初始化值和最大值。

Permanent Space并不等同于方法区，只不过是Hotspot JVM用Permanent Space来实现方法区而已，有些虚拟机没有Permanent Space而用其他机制来实现方法区。

**（2）堆**

堆，主要用来存放类的对象实例信息。

堆分为Old Space（又名，Tenured Generation）和Young Space。

Old Space主要存放应用程序中生命周期长的存活对象；

Eden(伊甸园)主要存放新生的对象；

S0和S1是两个大小相同的内存区域，主要存放每次垃圾回收后Eden存活的对象，作为对象从Eden过渡到Old Space的缓冲地带（S是指英文单词Survivor Space）。

**堆之所以要划分区间，是为了方便对象创建和垃圾回收，后面垃圾回收部分会解释。**

**2).线程内存区**

线程内存区=单个线程内存+单个线程内存+.......

单个线程内存=PC Regster+JVM栈+本地方法栈

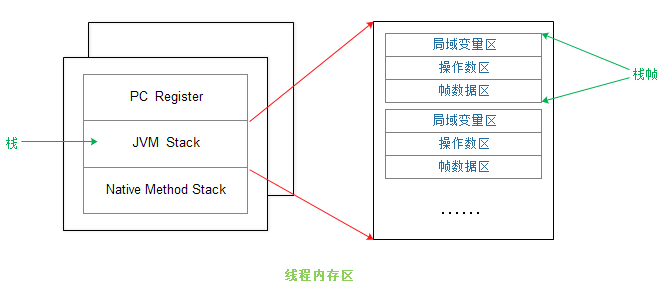
JVM栈=栈帧+栈帧+.....

栈帧=局域变量区+操作数区+帧数据区

在Java中，一个线程会对应一个JVM栈(JVM Stack)，JVM栈里记录了线程的运行状态。

JVM栈以栈帧为单位组成，一个栈帧代表一个方法调用。栈帧由三部分组成：局部变量区、操作数栈、帧数据区。

本地方法栈 ，例如调用native方法。

‘

### **堆 VS 栈**

JVM栈是运行时的单位，而JVM堆是存储的单位。

JVM栈代表了处理逻辑，而JVM堆代表了数据。

JVM堆中存的是对象和数组。JVM栈中存的是基本数据类型和JVM堆中对象的引用。

JVM堆是所有线程共享，JVM栈是线程独有。

### 内存溢出 OOM

参考BLOG：

<http://blog.csdn.net/chaofanwei/article/details/19483101>

堆溢出

原因莫过于对象太多导致，

public class HeapOOM {

static class OOMObject {

}

public static void main(String[] args) {

List<OOMObject> list = new ArrayList<OOMObject>();

while (true) {

list.add(new OOMObject());

}

}

}

1. /\*\*
2. \* java.lang.OutOfMemoryError: Java heap space
3. Dumping heap to java\_pid1820.hprof ...
4. Heap dump file created [24787111 bytes in 0.346 secs]
5. Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
6. at java.util.Arrays.copyOf(Arrays.java:2760)
7. at java.util.Arrays.copyOf(Arrays.java:2734)
8. at java.util.ArrayList.ensureCapacity(ArrayList.java:167)
9. at java.util.ArrayList.add(ArrayList.java:351)
10. at baby.oom.HeapOOM.main(HeapOOM.java:19)
11. \*
12. \*/

栈溢出

栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError

栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是StackOverflowError。

1. **public** **class** JavaVMStackSOF {
2. **private** **int** stackLength = 1;
3. **public** **void** stackLeak() {
4. stackLength++;
5. stackLeak();
6. }
7. // 太深了，无限的循环嵌套，超过默认的深度
8. **public** **static** **void** main(String[] args) **throws** Throwable {
9. JavaVMStackSOF oom = **new** JavaVMStackSOF();
10. **try** {
11. oom.stackLeak();
12. } **catch** (Throwable e) {
13. System.out.println("stack length:" + oom.stackLength);
14. **throw** e;
15. }
16. }
17. }

## 垃圾回收原理

<http://www.cnblogs.com/therunningfish/p/5523995.html>

程序计数器、虚拟机栈、本地方法栈3个区域随线程生，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的执行这出栈和入栈操作。

每一个栈帧中分配多少内存基本上是在类结构确定下来是就已知的，因此这几个区域的内存分配和回收都具备确定性，在这几个区域内不需要过多考虑回收问题，因为方法结束或者线程结束时，内存就跟着回收了。

JVM将堆（heap）分成young区和old区。young区包括eden、s0、s1，并且三个区之间的大小有一定比例。例如，按8:1:1分成一块Eden和两小块Survivor区，每次GC时，young区里，将Eden和S0中存活的对象复制到另一块空闲的S1中。

young区的垃圾回收是经常要发生的，被称为Minor GC（次要回收）。一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Minor GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对Young space的Eden区进行，不会影响到Old space。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

Minor GC主要过程：

a、新生成的对象在Eden区完成内存分配;

b、当Eden区满了，再创建对象，会因为申请不到空间，触发minorGC，进行young(eden+1survivor)区的垃圾回收。（为什么是eden+1survivor：两个survivor中始终有一个survivor是空的，空的那个被标记成To Survivor）;

c、minorGC时，Eden不能被回收的对象被放入到空的survivor（也就是放到To Survivor，同时Eden肯定会被清空），另一个survivor（From Survivor）里不能被GC回收的对象也会被放入这个survivor（To Survivor），始终保证一个survivor是空的。（MinorGC完成之后，To Survivor 和 From Survivor的标记互换）;

d、当做第3步的时候，如果发现存放对象的那个survivor满了，则这些对象被copy到old区，或者survivor区没有满，但是有些对象已经足够Old（通过XX:MaxTenuringThreshold参数来设置），也被放入Old区。(对象在Survivor区中每熬过一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁）时，就会晋升到老年代中)

**（3）标记-整理**

old space也可以标记-复制策略吗？当然不行！

young space中的对象大部分都是生命周期较短的对象，每次GC后，所剩下的活对象数量不是很大。而old space中的对象大部分都是生命周期特别长的对象，即使GC后，仍然会剩下大量的活对象。如果仍然采用复制动作，回收效率会变得非常低。

根据old space的特点，可以采用整理动作。整理时，先清除掉应该清除的对象，然后把存活对象“压缩”到堆的一端，按顺序排放。

动图：

Old space(+Permanent Space)的垃圾回收是偶尔发生的，被称为Full GC（主要回收）。Full GC因为需要对整个堆进行回收，包括Young、Old和Perm，所以比Minor GC要慢，因此应该尽可能减少Full GC的次数。在对JVM调优的过程中，很大一部分工作就是对于FullGC的调节。

有如下原因可能导致Full GC：

* 年老代（Tenured）被写满
* 持久代（Perm）被写满
* System.gc()被显示调用
* 上一次GC之后Heap的各域分配策略动态变化