# 数据结构

## 1.1 HashMap底层实现原理

<http://www.cnblogs.com/chengxiao/p/6059914.html>

<https://www.zhihu.com/question/20733617>

http://blog.csdn.net/richard\_jason/article/details/53887222

哈希冲突的解决方案有多种:开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法，而HashMap即是采用了链地址法，也就是数组+链表的方式。

HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。

其他几个重要字段

//实际存储的key-value键值对的个数

transient int size;

//阈值，当table == {}时，该值为初始容量（初始容量默认为16）；当table被填充了，也就是为table分配内存空间后，threshold一般为 capacity\*loadFactory。HashMap在进行扩容时需要参考threshold，后面会详细谈到

int threshold;

//负载因子，代表了table的填充度有多少，默认是0.75

final float loadFactor;

//用于快速失败，由于HashMap非线程安全，在对HashMap进行迭代时，如果期间其他线程的参与导致HashMap的结构发生变化了（比如put，remove等操作），需要抛出异常ConcurrentModificationException（foreach时）

transient int modCount;

此处对传入的初始容量进行校验，最大不能超过MAXIMUM\_CAPACITY = 1<<30(230)

在常规构造器中，没有为数组table分配内存空间（有一个入参为指定Map的构\*-器例外），而是在执行put操作的时候才真正构建table数组

数组长度一定为2的次幂;

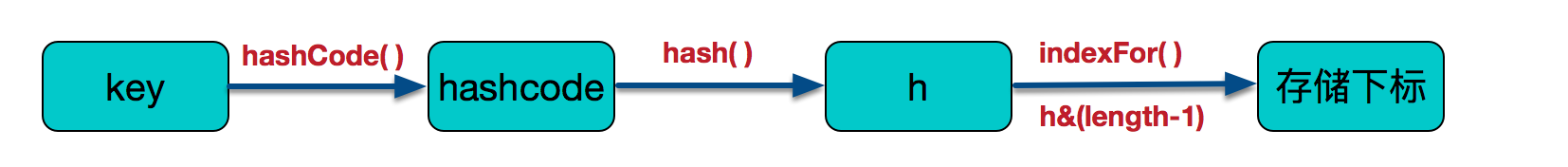
当发生哈希冲突并且size大于阈值（threshold=capacity\*loadFactory）的时候，需要进行数组扩容（resize），扩容时，需要新建一个长度为之前数组2倍的新的数组，然后将当前的Entry数组中的元素全部传输过去（遍历，重新计算索引位置，将老数组数据复制到新数组中去），扩容后的新数组长度为之前的2倍，所以扩容相对来说是个耗资源的操作。

如果负载因子取得太大，threshold与capacity太接近，当容量增大时，冲突会增加，造成同一地址链表过大；如果太小，哈希表太稀疏，浪费存储空间。负载因子可以大于1（即threshold大于数组长度，因为是链地址法）

Put时如果key为null，存储位置为table[0]或table[0]的冲突链上（table为HashMap中存的数组），如果该对应数据已存在，执行覆盖操作。用新value替换旧value，并返回旧value，如果对应数据不存在，则添加到链表的头上（保证插入O（1））

put：首先判断key是否为null，若为null，则直接调用putForNullKey方法。若不为空则先计算key的hash值，然后根据hash值搜索在table数组中的索引位置，如果table数组在该位置处有元素，循环遍历链表，比较是否存在相同的key，若存在则覆盖原来key的value，否则将该元素保存在链头（最先保存的元素放在链尾）。若table在该处没有元素，则直接保存。

最终存储位置的确定流程是这样的：



get方法的实现相对简单，key(hashcode-返回int)-->hash-->indexFor-->最终索引位置，找到对应位置table[i]，再查看是否有链表，遍历链表，通过key的equals方法比对查找对应的记录。（&length-1也将范围较大的hash值缩小到了length内）

我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map

int capacity = 1;

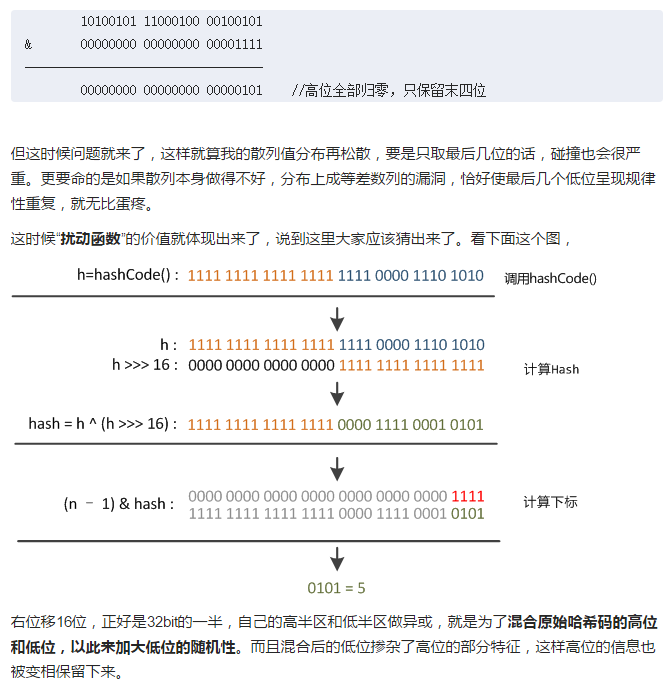
while (capacity < initialCapacity)

capacity <<= 1;

这段代码保证初始化时HashMap的容量总是2的n次方，即底层数组的长度总是为2的n次方。

由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。（这一般是通过将该对象的内部地址转换成一个整数来实现的）

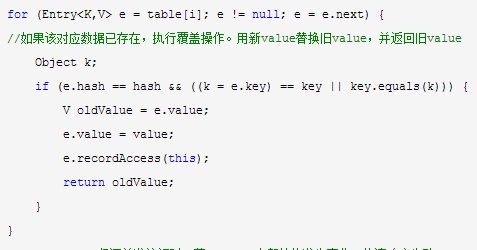




HashMap中得到数组下标是通过低位掩码（与n-1）（这样比取模速度快），但是这样高位信息就会缺失，而计算哈希值右移16位再异或，保留的高位信息，也减小了哈希冲突。

HashMap的存放自定义类时，需要实现自定义类的什么方法？

下图中的hash，都是用hashCode经过hash()函数的



重写hashcode（）和equals（）方法

如果不重写equals（）方法，HashMap没有判断两个对象相等的标准。如果不重写hashcode（），将对用object默认的hashcode方法（根据对象地址生成hashcode），如果new了两个对象，它们的属性均相同，但由于是两个对象，所以object生成的hashcode不同，但在hashmap中这两个key应该当做相同的key，但不重写hashcode则无法实现。

Get和put方法通过key的equals方法比对查找对应的记录。要注意的是，有人觉得上面在定位到数组位置之后然后遍历链表的时候，e.hash == hash这个判断没必要，仅通过equals判断就可以。其实不然，试想一下，如果传入的key对象重写了equals方法却没有重写hashCode，而恰巧此对象定位到这个数组位置，如果仅仅用equals判断可能是相等的，但其hashCode和当前对象不一致，这种情况，根据Object的hashCode的约定，不能返回当前对象，而应该返回null（规定，相等的对象，hashcode必须相同）

HashMap中的final属性，不可变final int hash; final K key;

扩容resize()

如果size大于threshold（capacity\*loadFactory）就进行扩容，原容量乘以2，再进行rehash的过程。如果capacity已经达到最大（2^30），则threshold变为Integer.MAX\_VALUE（没有新建节点，只是新的指针）

多线程同时操作hashmap时

1会产生死循环

如果多个线程同时扩容，产生两个新的table，形成一个闭环

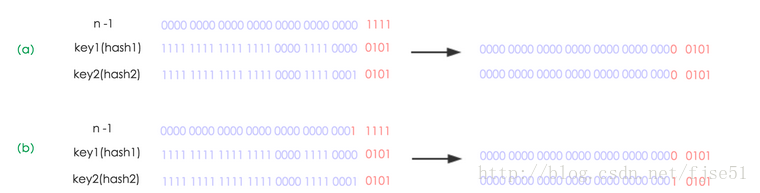
http://ifeve.com/hashmap-infinite-loop/

**HashMap JDK1.8**

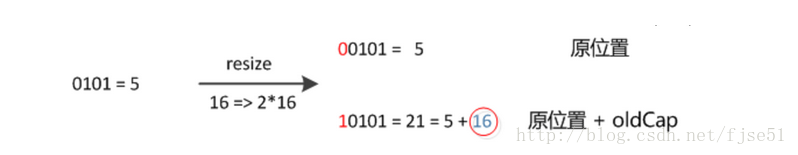
Jdk1.8中没有indexFor函数，直接使用table[index = (n – 1) & hash]（与运算交换左右，结果不变）

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（TREEIFY\_THRESHOLD默认超过8、大于等于）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能（O(logn)）。当长度小于（UNTREEIFY\_THRESHOLD默认为6、小于等于），就会退化成链表。

下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。(每个节点e的hash早就计算好，并保存在final hash中)。通过if ((e.hash & oldCap) == 0)判定前面那个bit是不是1，如果是1则加上oldCap

static class Node<K,V> implements Map.Entry<K,V> {}，jdk1.8中用node替代了entry

## 1.2 HashSet/TreeSet

HashSet内部有一个HashMap，只使用了map的key，value都是同一个object

private static final Object PRESENT = new Object();

TreeSet内部是一个TreeMap，只使用了key，value也是上面这个object

## 1.3 LinkedHashMap

LinkedHashMap是HashMap的子类，与HashMap有着同样的存储结构，但它加入了一个双向链表的头结点（有head和tail指针），将所有put到LinkedHashmap的节点一一串成了一个双向循环链表，因此它保留了节点插入的顺序，可以使节点的输出顺序与输入顺序相同。LinkedHashMap同样是非线程安全的，只在单线程环境下使用。

LinkedHashMap的实现：

对于LinkedHashMap而言，它继承与HashMap、底层使用哈希表与双向链表来保存所有元素。其基本操作与父类HashMap相似，它通过重写父类相关的方法，来实现自己的链接列表特性。

1) Entry元素：

LinkedHashMap采用的hash算法和HashMap相同，但是它重新定义了数组中保存的元素Entry，该Entry除了保存当前对象的引用外，还保存了其上一个元素before和下一个元素after的引用，从而在哈希表的基础上又构成了双向链接列表。

2) 初始化：

通过源代码可以看出，在LinkedHashMap的构造方法中，实际调用了父类HashMap的相关构造方法来构造一个底层存放的table数组。

3) 存储：

LinkedHashMap并未重写父类HashMap的put方法，而是重写了父类HashMap的put方法调用的子方法void addEntry() 和void createEntry(i)，提供了自己特有的双向链接列表的实现。

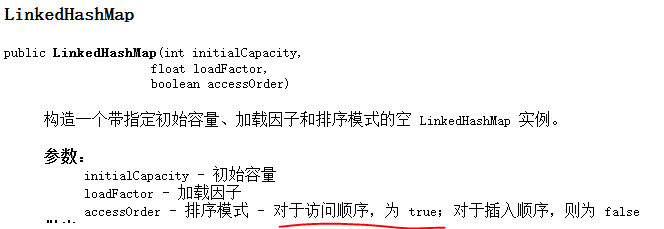
4) 读取：

LinkedHashMap重写了父类HashMap的get方法，实际在调用父类getEntry()方法取得查找的元素后，再判断当排序模式accessOrder为true时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失。

5) 排序模式：

LinkedHashMap定义了排序模式accessOrder，该属性为boolean型变量，对于访问顺序，为true；对于插入顺序，则为false。

一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。这些构造方法都会默认指定排序模式为插入顺序。如果你想构造一个LinkedHashMap，并打算按从近期访问最少到近期访问最多的顺序（即访问顺序）来保存元素，可以实现LRU（最近最少使用页面置换算法）



## 1.4 Hashtable底层实现原理

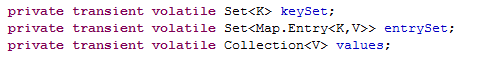
与HashMap十分类似，在put、get、remove等方法上加了同步

public synchronized V put(K key, V value) {}

方法的synchronized使用this锁，把整个对象都锁了，粒度大



计算哈希值，0x7FFFFFFF转换为二进制是1个0,31个1，返回一个符号位为0的数，即丢弃最高位，以免函数外产生影响。



使用volatile关键字

Hashtable 的 key 和 value 都不允许为 null，Hashtable遇到 null，直接返回 NullPointerException。

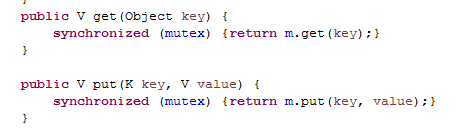
## 1.5 Collections.SynchronizedMap底层实现原理

Collections.synchronizedMap()实现原理是Collections定义了一个SynchronizedMap的内部类，并返回这个类的实例。



SynchronizedMap这个类实现了Map接口，在调用方法时使用synchronized来保证线程同步,当然了实际上操作的还是我们传入的HashMap实例，简单的说就是Collections.synchronizedMap()方法帮我们在操作HashMap时自动添加了synchronized来实现线程同步，类似的其它Collections.synchronizedXX方法也是类似原理）

Mutex在构造时默认赋值为this，即所有方法都用的同一个锁。



## 1.6 ConcurrentHashMap底层实现原理

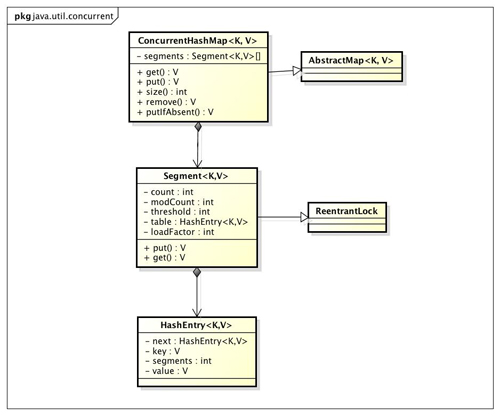
<http://ifeve.com/concurrenthashmap/>

http://blog.csdn.net/dingji\_ping/article/details/51005799

ConcurrentHashMap主要有三大结构：整个Hash表，segment（段），HashEntry（节点）。每个segment就相当于一个HashTable。ConcurrentHashMap将锁加在segment上（每个段上），这样我们在对segment1操作的时候，同时也可以对segment2中的数据操作，这样效率就会高很多。

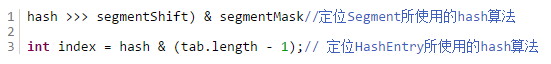
Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。Put和remove方法中有lock()和unlock()（都是使用的this对象，lock()在代码开始，unlock在finally中）。

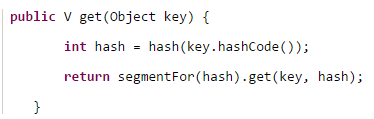
Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素.

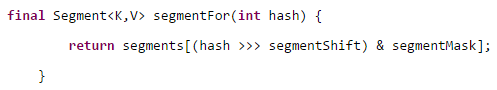


segments数组的长度ssize通过concurrencyLevel(并发等级默认是16)计算得出。为了能通过按位与的哈希算法来定位segments数组的索引，必须保证segments数组的长度是2的N次方（power-of-two size），所以必须计算出一个是大于或等于concurrencyLevel的最小的2的N次方值来作为segments数组的长度。

初始化每个Segment。输入参数initialCapacity是ConcurrentHashMap的初始化容量，loadfactor是每个segment的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个segment。segment里HashEntry数组的长度，它等于initialCapacity除以ssize的倍数（总大小除以segments数组长度），HashEntry的长度也是2的N次方







我们知道HashTable容器的get方法是需要加锁的，那么ConcurrentHashMap的get操作是如何做到不加锁的呢？原因是它的get方法里将要使用的共享变量都定义成volatile，之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值。

如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

其中有一个Segment数组，每个Segment中都有一个锁，因此Segment相当于一个多线程安全的HashMap，采用分段加锁。每个Segment中有一个Entry数组，Entry中成员value是volatile修饰，其他成员通过final修饰。get操作不用加锁，put和remove操作需要加锁，因为value通过volatile保证可见性。

两个hash过程，第一次找到所在的桶，并将桶锁定，第二次执行写操作。而读操作不加锁

Collections.SynchronizedMap和Hashtable都是整个表的锁，与ConcurrentHashMap锁粒度不同

ConcurrentHashMap不允许key或value为null值。

ConcurrentHashMap允许一边更新、一边遍历，也就是说在Iterator对象遍历的时候，ConcurrentHashMap也可以进行remove,put操作，且遍历的数据会随着remove,put操作产出变化，相当于有多个线程在操作同一个map（可以在foreach keyset时remove对象，HashMap不可以）

**java.util.concurrent.ConcurrentHashMap<K,V> JDK1.8**

本文的分析的源码是JDK8的版本，与JDK6的版本有很大的差异。实现线程安全的思想也已经完全变了，它摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现,利用CAS算法。它沿用了与它同时期的HashMap版本的思想，底层依然由“数组”+链表+红黑树的方式思想，大于8个转换为红黑树。默认初始大小16，负载因子也是0.75，定位元素的方法也是先hashCode(),再无符号右移16位异或，再(n-1)&hash

取消segments字段，直接采用transient volatile Node<K,V>[] table;保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

**put函数流程：**

1、判断put进来的key和value是否为null，如果为null抛异常。（ConcurrentHashMap的key、value不能为null）。

2、随后进入无限循环(没有判断条件的for循环)，何时插入成功，何时退出。

3、在无限循环中，若table数组为空（底层数组加链表），则调用initTable()，初始化table；

4、若table不为空，先hashCode(),再无符号右移16位异或，再(n-1)&hash，定位到table中的位置，如果该位置为空（说明还没有发生哈希冲突），则使用CAS将新的节点放入table中。

5、如果该位置不为空，且该节点的hash值为MOVED（即为forward节点，哈希值为-1，其中含有指向nextTable的指针，class ForwardingNode中有nexttable变量），说明此时正在扩容，且该节点已经扩容完毕，如果还有剩余任务（任务没分配完）该线程执行helpTransfer方法，帮助其他线程完成扩容，如果已经没有剩余任务，则该线程可以直接操作新数组nextTable进行put。

6、如果该位置不为空，且该节点不是forward节点。对桶中的第一个结点（即table表中的结点，哈希值相同的链表的第一个节点）进行加锁（锁是该结点，如果此时还有其他线程想来put，会阻塞）（如果不加锁，可能在遍历链表的过程中，又有其他线程放进来一个相同的元素，但此时我已经遍历过，发现没有相同的，这样就会产生两个相同的），对该桶进行遍历，桶中的结点的hash值与key值与给定的hash值和key值相等，则根据标识选择是否进行更新操作（用给定的value值替换该结点的value值），若遍历完桶仍没有找到hash值与key值和指定的hash值与key值相等的结点，则直接新生一个结点并赋值为之前最后一个结点的下一个结点。

7、若binCount值达到红黑树转化的阈值，则将桶中的结构转化为红黑树存储，最后，增加binCount的值。最后调用addcount方法，将concurrenthashmap的size加1，调用size()方法时会用到这个值。

**扩容transfer()函数流程：**

整个扩容操作分为两个部分

第一部分是构建一个nextTable,它的容量是原来的两倍，这个操作是单线程完成的。

第二个部分就是将原来table中的元素复制到nextTable中，这里允许多线程进行操作。

其他线程调用helptransfer方法来协助扩容时，首先拿到nextTable数组，再调用transfer方法。给新来的线程分配任务（默认是16个桶一个任务）。

遍历自己所分到的桶：

1、桶中元素不存在，则通过CAS操作设置桶中第一个元素为ForwardingNode，其Hash值为MOVED（-1）,同时该元素含有新的数组引用

此时若其他线程进行put操作，发现第一个元素的hash值为-1则代表正在进行扩容操作（并且表明该桶已经完成扩容操作了，可以直接在新的数组中重新进行hash和插入操作），该线程就可以去帮助扩容，或者没有任务则不用参与，此时可以去直接操作新的数组了

2、桶中元素存在且hash值为-1，则说明该桶已经被处理了（本不会出现多个线程任务重叠的情况，这里主要是该线程在执行完所有的任务后会再次进行检查，再次核对）

3、桶中为链表或者红黑树结构，则需要获取桶锁，防止其他线程对该桶进行put操作，然后处理方式同HashMap的处理方式一样，对桶中元素分为2类，分别代表当前桶中和要迁移到新桶中的元素。设置完毕后代表桶迁移工作已经完成，旧数组中该桶可以设置成ForwardingNode了，**已经完成从table复制到nextTable的节点，要设置为forward**

**get函数流程：**

1、根据k计算出hash值，找到对应的数组index

2、如果该index位置无元素则直接返回null

3、如果该index位置有元素

如果第一个元素的hash值小于0，则该节点可能为ForwardingNode或者红黑树节点TreeBin

如果是ForwardingNode（表示当前正在进行扩容，且已经扩容完成），使用新的数组来进行查找

如果是红黑树节点TreeBin，使用红黑树的查找方式来进行查找

如果第一个元素的hash大于等于0，则为链表结构，依次遍历即可找到对应的元素，也就是读的时候不会加锁，同时有put，不会阻塞。

读不加锁是因为使用了volatile（用在transient volatile Node<K,V>[] table），happens-before

ConcurrentHashmap和Hashtable不允许key和value为null：

ConcurrentHashmap和Hashtable都是支持并发的，这样会有一个问题，当你通过get(k)获取对应的value时，如果获取到的是null时，你无法判断，它是put（k,v）的时候value为null，还是这个key从来没有做过映射。HashMap是非并发的，可以通过contains(key)来做这个判断。而支持并发的Map在调用m.contains（key）和m.get(key),m可能已经不同了。

ConcurrentHashmap和Hashtable都不允许key和value为null，Collections.synchronizedMap和HashMap的key和value都可以为null（因为就是包装了hashmap），TreeMap的key不可为空（非线程安全，需要排序），value可以

说一下HashMap和TreeMap的区别？

基于红黑树的实现，线程非安全，存入TreeMap的元素应当实现Comparable接口或者实现Comparator接口，会按照排序后的顺序迭代元素，key不能为null，因为null没有实现comparable。线程不同步。有序散列表，实现SortedMap 接口。也有modCount

HashMap采用链地址的哈希表，无序的。

## 1.7 Vector

vector线程安全的，有modCount，在每个方法上加synchronized，初始大小10，底层维护了一个object类型的数组，在构造函数中初始化。实现了list接口。

**Comparable和Comparator**

Comparable和Comparator都是用来实现集合中元素的比较、排序的。

Comparable是在集合内部定义的方法实现的排序，位于java.lang下。

Comparator是在集合外部实现的排序，位于java.util下。

Comparable是一个对象本身就已经支持自比较所需要实现的接口，如String、Integer自己就实现了Comparable接口，可完成比较大小操作。自定义类要在加入list容器中后能够排序，也可以实现Comparable接口，在用Collections类的sort方法排序时若不指定Comparator，那就以自然顺序排序。所谓自然顺序就是实现Comparable接口设定的排序方式。

Comparator是一个专用的比较器，当这个对象不支持自比较或者自比较函数不能满足要求时，可写一个比较器来完成两个对象之间大小的比较。Comparator体现了一种策略模式(strategy design pattern)，就是不改变对象自身，而用一个策略对象(strategy object)来改变它的行为。

总而言之Comparable是自已完成比较，Comparator是外部程序实现比较。

**CAS(UnSafe包)**

java.util.concurrent包完全建立在CAS之上的

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值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。

要实现无锁（lock-free）的非阻塞算法有多种实现方法，其中CAS（比较与交换，Compare and swap）是一种有名的无锁算法。CAS, CPU指令，在大多数处理器架构，包括IA32、Space中采用的都是CAS指令，CAS的语义是“我认为V的值应该为A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少”，CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值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。是一条CPU的原子指令，其实现方式是基于硬件平台的汇编指令，就是说CAS是靠硬件实现的

拿出AtomicInteger来研究在没有锁的情况下是如何做到数据正确性的。

private volatile int value;

在没有锁的机制下可能需要借助volatile原语，保证线程间的数据是可见的（共享的）。

这样才获取变量的值的时候才能直接读取。

public final int get() {

return value;

}

然后来看看++i是怎么做到的。

public final int incrementAndGet() {

for (;;) {//无限循环直到成功

int current = get();

int next = current + 1;

if (compareAndSet(current, next))//相同返回true，不同返回false

return next;

}

}

在这里采用了CAS操作，每次从内存中读取数据然后将此数据和+1后的结果进行CAS操作，如果成功就返回结果，否则重试直到成功为止。

而compareAndSet利用JNI来完成CPU指令的操作。

public final boolean compareAndSet(int expect, int update) {

return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

}

整体的过程就是这样子的，利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法。其它原子操作都是利用类似的特性完成的。

CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作

1. ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2. 循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。**CAS自旋volatile变量**

3. 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

**单例模式**

**public** **class** Singleton {

**private** Singleton(){}

**private** **static** **final** Singleton *instance* = **new** Singleton();

**public** **static** Singleton getInstance(){

**return** *instance*;

}

}

饿汉且保证线程安全（强烈推荐）

**private** **static** **final** Singleton *instance* = **new** Singleton();为什么要用static，因为getInstance方法是static的（不用生成实例就能使用），静态方法只能调用静态成员，在类初始化时就实例化instance

**public** **class** Singleton {//双重校验的懒汉，且线程安全

**private** Singleton(){}

**private** **volatile** **static** Singleton *instance*;（加volatile防止指令重排序）

**public** **static** Singleton getInstance(){

**if**(*instance* == **null**){//加锁效率低，在已经生成实例后，没必要再判断锁

**synchronized**(Singleton.**class**){//加锁，防止多线程时，生成多个实例

**if**(*instance* == **null**){

*instance* = **new** Singleton();**指令重排序，先完成赋值，但构造函数还没执行完**

}

}

}

**return** *instance*;

}

}

可以分解为3行伪代码

1 memory=allocate();// 分配内存 相当于c的malloc

2 ctorInstanc(memory) //初始化对象

3 instance=memory //设置instance**指向刚分配的地址**

上面的代码在编译器运行时，可能会出现重排序 从1-2-3 排序为1-3-2

如此在多线程下就会出现问题

例如现在有2个线程A,B

线程A在执行第5行代码时，B线程进来，而此时A执行了 1和3，没有执行2，此时B线程判断instance不为null 直接返回一个未初始化的对象，就会出现问题

而用了volatile，上面的重排序就会在多线程环境中禁止，不会出现上述问题。

**java.util.ArrayList<E>跟LinkedList的底层实现原理**

常用集合类的继承结构如下：   
Collection<--List<--Vector 底层是数组

Collection<--List<--Vector <--Stack（就是调用vector的方法，线程安全）  
Collection<--List<--ArrayList 底层是数组  
Collection<--List<--LinkedList 底层有node类存储了父节点和子节点  
Collection<--Set<--HashSet 底层是HashMap，只使用key  
Collection<--Set<--HashSet<--LinkedHashSet   
Collection<--Set<--SortedSet<--TreeSet   
Map<--SortedMap<--TreeMap   
Map<--HashMap

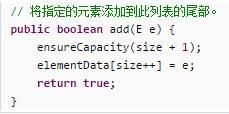
1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

Vector是线程同步的(sychronized)的，这也是Vector和ArrayList（不同步） 的一个的重要区别。

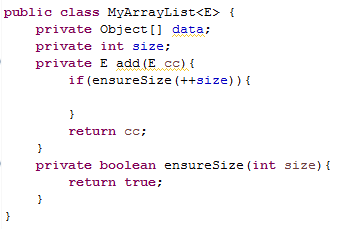
ArrayList底层基于Object类型的数组实现，初始大小为10.使用modCount实现快速失败，非同步



ArrayList如果构造函数不传入初始大小，则object类型数组先赋值为一个空数组，在add方法时才真正构造（使用Arrays.copyOf方法）。如果传入初始大小，则直接new初始大小的数组。

ArrayList删除，使用system.arraycopy，把原数组index+1往后的数据，复制到原数组从开始index的位置，并把最后一个元素赋为null

自己实现arraylist，泛型类，需要一个ensureSize函数判断数组容量是否够（每次扩容1.5倍）



扩容，a原始容量为10，copyOf函数后为20

LinkedList是双向链表，node节点里有next和prev指针以及自身的值，在LinkedList中有First和Last两个节点，用于指向链表的头和尾，如果调用add（）方法，直接在Last节点后面添加，并把Last赋为新加的结点。

在之前版本的jdk中，只有一个header指针，header的next即为头节点，header的prev即为尾节点，header本身值为空。环形的

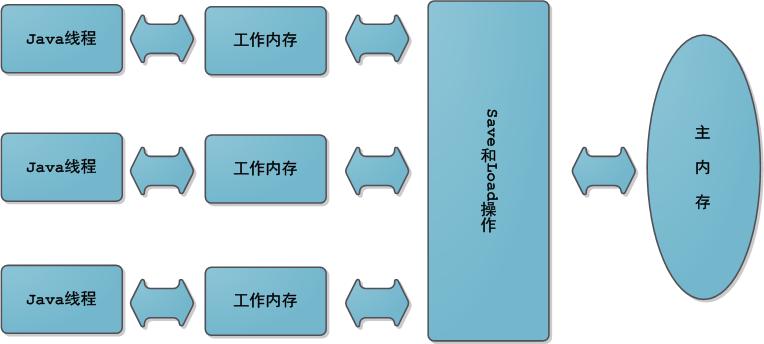
**Java内存模型**

由于计算机的存储设备与处理器的运算能力之间有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（cache）来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中没这样处理器就无需等待缓慢的内存读写了。

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样底层细节。此处的变量与Java编程时所说的变量不一样，指包括了实例字段、静态字段和构成数组对象的元素，但是不包括局部变量与方法参数，后者是线程私有的，不会被共享。

Java内存模型中规定了所有的变量都存储在主内存中（可以与物理硬件的内存类比），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。

这里的主内存、工作内存与Java内存区域的Java堆、栈、方法区不是同一层次内存划分。



关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java内存模型定义了以下八种操作来完成，且这些必须是原子的、不可再分的：

lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。

unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用

load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。

use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。

assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。

write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行read和load操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。Java内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。也就是read和load之间，store和write之间是可以插入其他指令的，如对主内存中的变量a、b进行访问时，可能的顺序是read a，read b，load b， load a。Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

不允许read和load、store和write操作之一单独出现

不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。

不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。

一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。

一个变量在同一时刻只允许一条线程对其进行lock操作，lock和unlock必须成对出现

如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行load或assign操作初始化变量的值

如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。

对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

**happens-before（先行先发生原则）**

先行发生原则（Happens-Before）是判断数据是否存在竞争、线程是否安全的主要依据。

先行发生是Java内存，模型中定义的两项操作之间的偏序关系，如果操作A先行发生于操作B，那么操作A产生的影响能够被操作B观察到。

Java内存模型中存在的天然的先行发生关系：

1. 程序次序规则：同一个线程内，按照代码出现的顺序，前面的代码先行于后面的代码，准确的说是控制流顺序，因为要考虑到分支和循环结构。

2. 管程锁定规则：一个unlock操作先行发生于后面（时间上）对同一个锁的lock操作。

3. volatile变量规则：对一个volatile变量的写操作先行发生于后面（时间上）对这个变量的读操作。

4. 线程启动规则：Thread的start( )方法先行发生于这个线程的每一个操作。

5. 线程终止规则：线程的所有操作都先行于此线程的终止检测。可以通过Thread.join( )方法结束、Thread.isAlive( )的返回值等手段检测线程的终止。

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，可以通过Thread.interrupt( )方法检测线程是否中断

7. 对象终结规则：一个对象的初始化完成先行于发生它的finalize（）方法的开始。

8. 传递性：如果操作A先行于操作B，操作B先行于操作C，那么操作A先行于操作C。

总结：一个操作“时间上的先发生”不代表这个操作先行发生；一个操作先行发生也不代表这个操作在时间上是先发生的（重排序的出现）。

时间上的先后顺序对先行发生没有太大的关系，所以衡量并发安全问题的时候不要受到时间顺序的影响，一切以先行发生原则为准。

**指令重排序**

重排序通常是编译器或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。重排序分为两类：编译期重排序和运行期重排序，分别对应编译时和运行时环境。

编译期重排序的典型就是通过调整指令顺序，在不改变程序语义的前提下，尽可能减少寄存器的读取、存储次数，充分复用寄存器的存储值。

//以下操作在同一个线程中执行

int i = 1;

int j = 2;

"int j = 2"的代码完全可能先被处理器执行，这并不影响先行先发生原则，因为我们在这条线程中没有办法感知到这点。（都是赋值操作，无论先发生哪一个，最终结果都与顺序执行一样）。但如果有其他线程在读取这两个值，重排序就会造成读取错误！

对于单一线程，如果有办法感知到这点，即int a = 1;int b = a;那么就不能重排序

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖。数据依赖分下列三种类型：

|  |  |  |
| --- | --- | --- |
| 名称 | 代码示例 | 说明 |
| 写后读 | a = 1;b = a; | 写一个变量之后，再读这个位置。 |
| 写后写 | a = 1;a = 2; | 写一个变量之后，再写这个变量。 |
| 读后写 | a = b;b = 1; | 读一个变量之后，再写这个变量。 |

上面三种情况，只要重排序两个操作的执行顺序，程序的执行结果将会被改变。所以，编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。也就是说：**在单线程环境下，指令执行的最终效果应当与其在顺序执行下的效果一致，否则这种优化便会失去意义。这句话有个专业术语叫做as-if-serial semantics (as-if-serial语义)**

1. **class** ReorderExample {
2. **int** a = 0;
3. **boolean** flag = **false**;
5. **public** **void** writer() {
6. a = 1;          // 1
7. flag = **true**;    // 2
8. }
10. **public** **void** reader() {
11. **if** (flag) {            // 3
12. **int** i = a \* a; // 4
13. }
14. }
15. }

flag变量是个标记，用来标识变量a是否已被写入。这里假设有两个线程A和B，A首先执行writer()方法，随后B线程接着执行reader()方法。线程B在执行操作4时，能否看到线程A在操作1对共享变量a的写入？

答案是：不一定能看到。

由于操作1和操作2没有数据依赖关系，编译器和处理器可以对这两个操作重排序；同样，操作3和操作4没有数据依赖关系，编译器和处理器也可以对这两个操作重排序。可以使用synchronized修饰这两个方法，或者将flag设为volatile解决这个问题（由先行先发生原则第三条可知）

如果A happens- before B，JMM并不要求A一定要在B之前执行。JMM仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前。这里操作A的执行结果不需要对操作B可见；而且重排序操作A和操作B后的执行结果，与操作A和操作B按happens- before顺序执行的结果一致。在这种情况下，JMM会认为这种重排序并不非法（not illegal），JMM允许这种重排序。

**Java与线程**

内核线程（KLT）就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换。程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口—轻量级进程（LWP），轻量级进程就是我们通常意义上所讲的线程，每个轻量级进程都是由一个内核线程支持（关系为1:1）；

对于Sun JDK来说，它的Window版与Linux版都是使用一对一的线程模型实现的，一条Java线程就映射到一条轻量级进程中。

**Volatile**

Volatile典型用法

Volatile Boolean asleep;

While(!asleep)

countSomeSheep();

Volatile保证可见性，在工作内存中，每次使用volatile变量都必须先从主内存刷新最新值，使用后立刻同步回主内存（volatile变量依然有工作内存的拷贝，但由于它特殊的操作顺序（每次使用前都需要刷新），所以看起来如同直接在主内存中读写访问一般）（将use和load动作相关联，即use前必须是load，load后必须是use；将assign和store相关联，即store前必须是assign，assign后必须是store）

Volatile保证有序性（禁止指令重排序优化，jkd1.5以后重新修复）：先行先发生原则中volatile变量规则：对一个volatile变量的写操作先行发生于（能被后面的读操作感知到）**后面（时间上）**对这个变量的读操作。通过内存屏障（Memory Barrier）实现，指令重排序时不能把后面的指令重排序到内存屏障之前的位置。

Volatile不能保证原子性。如果我们对一个volatile修饰的变量进行多线程下的自增操作，还是会出现线程安全问题。根本原因在于volatile关键字无法对自增进行安全性修饰，因为自增分为三步，读取、+1、写入。中间多个线程同时执行+1操作，还是会出现线程安全性问题。

①volatile轻量级，只能修饰变量。synchronized重量级，还可修饰方法

②volatile只能保证数据的可见性，不能用来同步，因为多个线程并发访问volatile修饰的变量不会阻塞。（常用于循环判断标志）

synchronized不仅保证可见性，而且还保证原子性，因为，只有获得了锁的线程才能进入临界区（操作共享变量的代码段），从而保证临界区中的所有语句都全部执行。多个线程争抢synchronized锁对象时，会出现阻塞。

volatile可见性（use和load、assign和store），有序性（禁止重排序，happens-before，内存屏障）

原子性：synchronized

可见性：volatile、synchronized、final

有序性：volatile、synchronized（相当于保证只有一个线程执行代码段，保证有序性）

**Integer的存储**

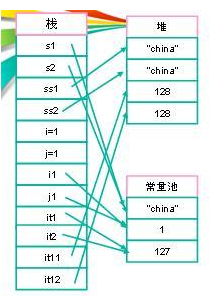
Integer是int的封装类，一般来说基础变量(int)赋值给Integer对象将自动装箱（Auto Boxing）并为Integer对象分配堆空间。因此即使基础变量值一样，封装类对象指向不同地址。所以System.out.println("i4 = i5? " + (i4 == i5)); //false

对JVM为了节省空间， 当Integer的值落在-128~127之间时，如Integer i1 = 2; Integer i2 = 2;此时JVM首先检查是否已存在值为2的Integer对象。如果是，则i2直接是引用已存在对象，即i2 = i1。所以System.out.println("i1 = i2? " + (i1 == i2)); //true

事实上， Integer已经默认创建了数值[-128~127]的Integer缓存数据。所以使用Integer i1=2时，JVM会直接在该在对象池找到该值的引用。

对于显式的new Integer(2),JVM将直接分配新空间。所以System.out.println("i1 = i3? " + (i1 == i3)); //false

int i = 1；如果在类中定义，就存放在堆，在方法中定义，就存放在栈（直接存放i=1）



**自动拆箱与装箱**

Integer、Long、Double这些包装类是final修饰的，不能被继承。Integer对象是不可变的，函数传递参数时，在函数内部修改，不会影响外部。

ArrayList<Integer>（底层是object数组），需要传入对象，而不能直接传入int，所以有了Integer包装类，而在list.add(3)时，会自动装箱成为Integer。int n = list.get(i)会自动拆箱。

Integer.valueOf()底层就是new Integer(num)，判断两个int可以用==，但判断两个Integer不能用==，需要用equals，就算是Integer a = new Integer(1); Integer b = new Integer(1); a==b也会输出false；Integer i1 = 2; Integer i2 = 2;i1==i2会输出true。

自动装箱的弊端

自动装箱有一个问题，那就是在一个循环中进行自动装箱操作的情况，如下面的例子就会创建多余的对象，影响程序的性能。

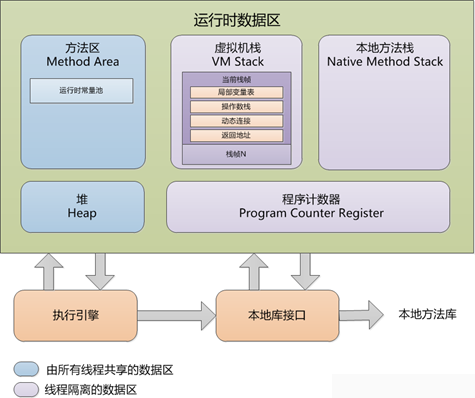
|  |  |
| --- | --- |
| 1  2  3  4 | Integer sum = 0;   for(int i=1000; i<5000; i++){     sum+=i;  } |

上面的代码sum+=i可以看成sum = sum + i，但是+这个操作符不适用于Integer对象，首先sum进行自动拆箱操作，进行数值相加操作，最后发生自动装箱操作转换成Integer对象。其内部变化如下

|  |  |
| --- | --- |
| 1  2 | sum = sum.intValue() + i;  Integer sum = new Integer(result); |

由于我们这里声明的sum为Integer类型，在上面的循环中会创建将近4000个无用的Integer对象，在这样庞大的循环中，会降低程序的性能并且加重了垃圾回收的工作量。因此在我们编程时，需要注意到这一点，正确地声明变量类型，避免因为自动装箱引起的性能问题。

**Java虚拟机运行时内存**



**1、程序计数器（Program Counter Register）**

程序计数器是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的信号指示器。字节码解释器就是通过改变该计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需依赖计数器来完成。线程私有的

此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

**2、Java虚拟机栈（Java Virtual Machine Stack）**

Java虚拟机栈与程序计数器一样，也是线程私有的，其生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型）和returnAddress类型（指向了一条字节码指令的地址）。其中64位长度的long和double会占用2个局部变量空间（Slot，一个32位），其余数据类型只占用1个。局部变量表所需的空间在编译期间完成分配，当进入一个方法时，其需要在帧中分配多大的局部变量空间是确定的，方法运行期间不会改变局部变量表的大小。

Java虚拟机规范中对该区域规定了两种异常情况：

1）如线程请求的深度大于虚拟机所允许的深度，栈溢出，如递归时，抛出StackOverflowError异常。

2）虚拟机栈动态扩展无法申请到足够的内存时，抛出OutOfMemoryError异常。

当方法传递参数时实际上是一个方法将自己栈帧中局部变量表的副本传递给另一个方法栈帧中的局部变量表（注意是副本，而不是其本身），不管数据类型是什么（基本类型，引用类型）

**3、本地方法栈（Native Method Stack）**

Java虚拟机可能会使用到传统的栈来支持native方法（使用Java语言以外的其它语言编写的方法）的执行。线程私有的，如Sun HotSpot虚拟机直接把本地方法栈和虚拟机栈合二为一。Java虚拟机规范中对该区域规定了两种异常情况：

1）如线程请求的深度大于虚拟机所允许的深度，抛出StackOverflowError异常。

2）虚拟机栈动态扩展无法申请到足够的内存时，抛出OutOfMemoryError异常。

**4、Java堆（Java Heap）**

Java堆是Java虚拟机管理内存中最大的一块，是所有线程共享的内存区域，随虚拟机的启动而创建。该区域唯一目的是存放对象实例，几乎所有对象的实例都在堆里面分配。

Java虚拟机规范规定，Java堆可以出于物理上不连续的内存空间中，只要逻辑上连续即可，如同磁盘空间一样，既可以实现成固定大小，也可以是扩展的，当前主流虚拟机都是按照扩展来实现的（通过-Xmx和-Xms控制）。

Java虚拟机规范中对该区域规定了OutOfMemoryError异常：如果堆中没有内存完成实例分配，并且堆无法再扩展则抛出OutOfMemoryError异常。（当Old区被放满的之后，进行Full GC，Full GC后，若Survivor及old区仍然无法存放从Eden复制过来的部分对象，则出现OOM错误/或者直接存放大对象、大数组，导致老年代空间不足）

**5、方法区（Method Area）**

方法区与Java堆一样，是各个线程共享的内存区域，用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。在HotSpot中用永久代来实现方法区，而其他虚拟机（如BEA JRockit、IBM J9等）是不存在永久代的。Java7中已经将运行时常量池从永久代移除，在Java 堆（Heap）中开辟了一块区域存放运行时常量池。而在Java8中，已经彻底没有了永久代，将方法区直接放在一个与堆不相连的本地内存区域，这个区域被叫做元空间。

元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制，但可以通过以下参数来指定元空间的大小：

-XX:MetaspaceSize，初始空间大小。-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

Java虚拟机规范中对方法区规定了OutOfMemoryError异常： 如果方法区的内存空间不能满足内存分配请求，那Java虚拟机将抛出一个OutOfMemoryError异常。

**6、运行时常量池（Runtime Constant Pool）**

运行时常量池是方法区的一部分。线程共享。Class文件中除了有类的版本、字段、方法、接口等信息外，还有一项信息是常量池，用于存放编译期生成的各种字面常量和符号引用，这部分内容在类加载后存放到方法区的常量池中。

Java虚拟机规范中对该区域规定了OutOfMemoryError异常： 当常量池无法申请到内存时抛出OutOfMemoryError异常。

static修饰的静态变量也存放在方法区中，但不是在常量池中（不能修饰局部变量），不能在一个方法内部定义static变量（final可以），只能定义为成员变量。

**7、直接内存**

直接内存并不是虚拟机运行时数据区域的一部分，也非Java虚拟规范中定义的内存区域，但这部分内存也被频繁使用，并且可能导致OutOfMemoryError异常出现。

在JDK1.4中新加入NIO（New Input/Output）类，引入了一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存。

**栈帧**

栈帧是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区的虚拟机栈的栈元素。栈帧存储了方法的局部变量表，操作数栈，动态连接和方法返回地址等信息。第一个方法从调用开始到执行完成，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。在编译代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到了方法表的Code属性中，因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体虚拟机的实现。

一个线程中的方法调用链可能会很长，很多方法都同时处理执行状态。对于执行引擎来讲，活动线程中，只有虚拟机栈顶的栈帧才是有效的，称为当前栈帧(Current Stack Frame)，这个栈帧所关联的方法称为当前方法(Current Method)。

**1、局部变量表**

局部标量表是一组变量值的存储空间，一个以字长为单位，从0开始计数的数组，用于存放方法参数和局部变量。变量槽 （Variable Slot）是局部变量表的最小单位，没有强制规定大小为 32 位，虽然32位足够存放大部分类型的数据。一个 Slot 可以存放 boolean、byte、char、short、int、float、reference 和 returnAddress 8种类型。其中 reference 表示对一个对象实例的引用。returnAddress 则指向了一条字节码指令的地址。 对于64位的 long 和 double 变量而言，虚拟机会为其分配两个连续的 Slot 空间。

虚拟机通过索引定位的方式使用局部变量表。之前我们知道，局部变量表存放的是方法参数和局部变量。当调用方法是非static 方法时，局部变量表中第0位索引的 Slot 默认是用于传递方法所属对象实例的引用，即“this”关键字指向的对象。分配完方法参数后，便会依次分配方法内部定义的局部变量。

为了节省栈帧空间，局部变量表中的 Slot 是可以重用的。当离开了某些变量的作用域之后，这些变量对应的 Slot 就可以交给其他变量使用。

**2、操作数栈**

操作数栈被组织成一个以字长为单位的数组。但不是通过索引来访问，而是通过标准栈操作--压栈和出栈来访问。方法执行中进行算术运算或者是调用其他的方法进行参数传递的时候是通过操作数栈进行的。

在概念模型中，两个栈帧是相互独立的。但是大多数虚拟机的实现都会进行优化，令两个栈帧出现一部分重叠。令下面的部分操作数栈与上面的局部变量表重叠在一块，这样在方法调用的时候可以共用一部分数据，无需进行额外的参数复制传递。

**3、帧数据区**

栈帧需要一些数据来支持常量池解析、正常方法返回和异常处理等。在帧数据区中保存着访问常量池的指针，方便程序访问常量池。此外，当函数返回或者出现异常时，虚拟机必须恢复调用者函数的栈帧，并让调用者函数继续执行下去。对于异常处理，虚拟机必须有一个异常处理表，方便在发生异常的时候找到处理异常的代码，因此异常处理表也是帧数据区中重要的一部分。

**3.1动态连接**

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。符号引用一部分会在类加载阶段或者第一次使用的时候就转化为直接引用，这种转化成为静态解析。另外一部分在每一次运行期间转化为直接引用，这部分称为动态连接。

**3.2方法返回地址**

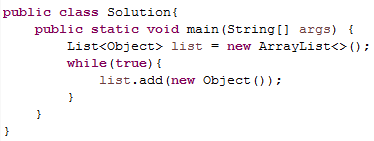
当一个方法开始执行后，只有两种方式可以退出这个方法。第一种是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者。

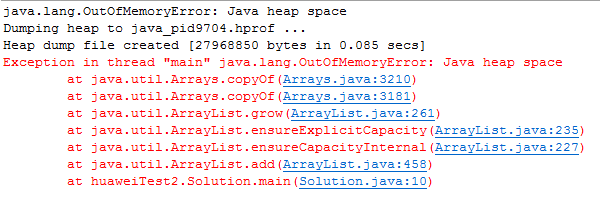
另一种退出方式是，在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理。无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行。

**OutOfMemoryError**

在eclipse中设置-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError（堆最小值、最大值设置成一样为了避免自动扩展，输出内存溢出时信息）

Java堆用于存储对象实例，只要不断创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收，当对象数量达到最大堆容量后就产生内存溢出异常。





**对象存活判断**

1、引用计数算法（JVM中不用）

给对象中添加一个引用计数器，每当有一个地方用它时，计数器值加1,；当引用失效时，计数器值减1；任何时刻计数器为0的对象不能再被使用。缺点：难以解决循环引用问题，objA.instance=objB和objB.instance=objA，除此之外，没有引用。

2、可达性分析算法

这个算法的基本思路就是通过一系列名为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，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，下图对象object5, object6, object7虽然有互相判断，但它们到GC Roots是不可达的，所以它们将会判定为是可回收对象。

在Java语言里，可作为GC Roots对象的包括如下几种：

a.虚拟机栈(栈桢中的本地变量表)中的引用的对象

b.方法区中的类静态属性引用的对象

c.方法区中的常量引用的对象

d.本地方法栈中JNI（即一般说的Native方法）的引用的对象

为什么选择这几个作为GC Roots？

首先要保证被选作GC Roots的对象是存活的，静态变量的声明周期长，而栈中引用的对象肯定是正在使用的对象（是存活的），因为调用方法时才会压栈。

虚拟机并不需要一个不漏地检查完所有全局性应用（如常量或类静态属性）与执行上下文（例如栈帧中的本地变量表），在HotSpot中，是使用一组称为OopMap的数据结构来达到这个目的，在类加载完成的时候，HotSpot就把对象内什么偏移量上是什么类型的数据计算出来，在JIT编译过程中，也会在特定的位置记录下栈和寄存器哪些位置是引用。这些记录信息的特定位置称为安全点（Safepoint），即程序制定时并非所有地方都能停顿下来开始GC，只有在到达安全点才能停顿。

**解释器与编译器**

Java程序最初是仅仅通过解释器解释执行的，即对字节码逐条解释执行，这种方式的执行速度相对会比较慢，尤其当某个方法或代码块运行的特别频繁时，这种方式的执行效率就显得很低。于是后来在虚拟机中引入了JIT编译器（即时编译器），当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为“Hot Spot Code”（热点代码），为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各层次的优化，完成这项任务的正是JIT编译器。

许多主流商用虚拟机（HotSpot和J9）同时包含解释器与编译器。解释器与编译器两者各有优势，当程序需要快速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后（多次调用的方法、多次执行的循环体），可以获得更高的执行效率。当程序运行环境中内存资源限制较大（如嵌入式），可以使用解释执行节约内存，反之可以使用编译执行来提升效率。

**逃逸**

this逃逸：在构造函数返回之前，其他线程就已经取得了该对象的引用，由于构造函数还没有完成，所以，对象也可能是残缺的，所以，取得对象引用的线程使用残缺的对象极有可能发生错误的情况。因为这两个线程是异步的，取得对象引用的线程并不一定会等待构造对象的线程完结后在使用引用。

this逃逸经常发生在构造函数中启动线程或注册监听器时， 如:

public class ThisEscape {

public ThisEscape() {

new Thread(new EscapeRunnable()).start();

// ...其他代码

}

private class EscapeRunnable implements Runnable {

public void run() {

// 在这里通过ThisEscape.this就可以引用外围类对象, 但是此时外围类对象可能还没有构造完成, 即发生了外围类的this引用的逃逸

}

}

}

解决：

public class ThisEscape {

private Thread t;

public ThisEscape() {

t = new Thread(new EscapeRunnable());

// ...其他代码

}

public void init() {

t.start();

}

private class EscapeRunnable implements Runnable {

@Override

public void run() {

// 在这里通过ThisEscape.this就可以引用外围类对象, 此时可以保证外围类对象已经构造完成

}

}

}

**final** List<String> list = **new** ArrayList<>();

list.add("a");

final指向可变的对象,final修饰的方法可以重载，不能重写。static方法重写后也要是static的（也可以重载），非static不能用static重写。

**引用**

1. 强引用：类似“Object o = new Object()”，只要强引用还存在，就不会被回收；

2、软引用：用来描述一些有用但非必须的对象。如果一个对象只具有软引用，则内存空间足够，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（如果内存够，软引用没有被回收，则可以直接使用，如果内存不够，软引用已经被回收，则重新读取数据（如从数据库中））。(java.lang.ref包)

SoftReference<String> softRef = new SoftReference<String>(str);

3、弱引用：也是用来描述非必须对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收只被弱引用关联的对象。如果这个对象是偶尔的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象。

4、虚引用：它是最弱的一种引用关系，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。为一个对象设置虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。PhantomReference

**finalize()**

即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程。标记的前提是对象在进行可达性分析后发现没有与GC Roots相连接的引用链。

1).第一次标记并进行一次筛选。

筛选的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finalize方法已经被虚拟机调用过（finalize只会调用一次），虚拟机将这两种情况都视为“没有必要执行”，对象被回收。

2).第二次标记

如果这个对象被判定为有必要执行finalize（）方法，那么这个对象将会被放置在一个名为：F-Queue的队列之中，并在稍后由一条虚拟机自动建立的、低优先级的Finalizer线程去执行。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这样做的原因是，如果一个对象finalize（）方法中执行缓慢，或者发生死循环（更极端的情况），将很可能会导致F-Queue队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。

Finalize（）方法是对象脱逃死亡命运的最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模标记，如果对象要在finalize（）中成功拯救自己----只要重新与引用链上的任何的一个对象建立关联即可，譬如把自己赋值给某个类变量或对象的成员变量，那在第二次标记时它将移除出“即将回收”的集合。如果对象这时候还没逃脱，那基本上它就真的被回收了。

**垃圾收集算法**

1、标记-清除算法（Mark-Sweep）。首先标记处所有需要回收的对象，在标记完成后统一回收。缺点：标记和清除两个过程都效率低；标记清除后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行中需要分配大对象时，无法找到足够的连续内存而不得不提取触发GC。

2、复制算法。将可用内存按容量划分成大小相等的两块，每次只使用一块。当这一块使用完了，就将还存活着的对象复制到另一块上面，然后再把已使用过的内存一次清理掉。这样不用考虑内存碎片的问题，只要移动堆顶指针，按顺序分配即可，实现简单、运行高效。缺点：内存缩小为原来的一半。

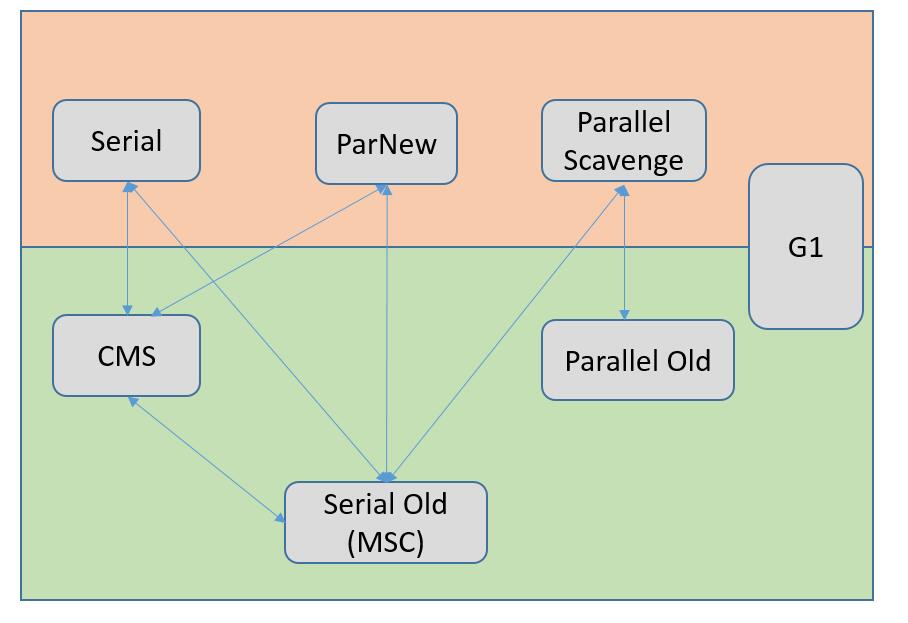
现代商用虚拟机都采用这种算法回收新生代。而新生代中约98%的对象都是“朝生夕死”，所以不需按1:1划分。HotSpot默认Eden和Survivor是8:1，所以每次可用内存为90%。但我们没法保证每次回收只有不多于10%的对象存活，当Survivor空间不够时，需要依赖其他内存（这里指老年代）进行分配担保（直接进入老年代）。

缺点：如果对象存活率太高，要进行较多复制操作，效率低。且需要额外空间担保，老年代不能选用这种算法。

3、标记-整理算法。

过程与“标记-清除”一样，但后续步骤不是直接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。老年代因为对象存活率高、没有额外空间进行分配担保，必须使用“标记-清理”或“标记-整理”算法。

**JVM垃圾收集器**

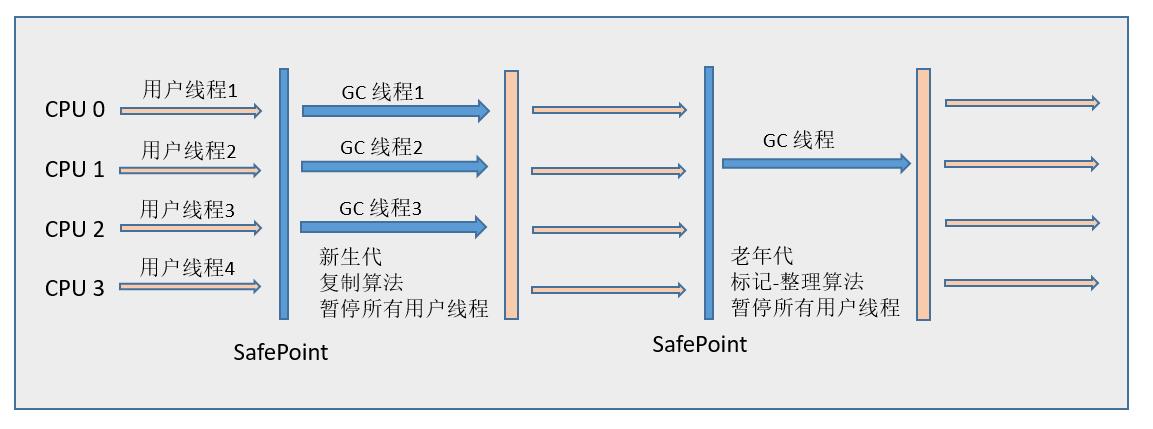


**1、Serial**是一个单线程收集器，在它进行垃圾收集时，必须暂停其他所有工作线程（Stop The World）；简单高效，是虚拟机在Client模式下默认的新生代收集器（复制算法）。停顿时间在几十到一百多毫秒以内，可以接受。



**2、ParNew**其实就是Serial收集器的多线程版本；ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器。除去性能因素，很重要的原因是除了Serial收集器外，目前只有它能与CMS收集器（老年代）配合工作。（复制算法）

但是，在单CPU环境中，ParNew收集器绝对不会有比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百地保证可以超越Serial收集器。然而，随着可以使用的CPU的数量的增加，它对于GC时系统资源的有效利用还是很有好处的。



**3、****Parallel** **Scavenge收集器**是新生代垃圾收集器，使用复制算法，也是并行的多线程收集器。与ParNew收集器相比，很多相似之处，但是Parallel Scavenge收集器更关注可控制的吞吐量（运行用户代码时间/(运行用户代码+垃圾收集时间)）。吞吐量越大，垃圾收集的时间越短，则用户代码则可以充分利用CPU资源，尽快完成程序的运算任务。

直观上，只要最大的垃圾收集停顿时间越小，吞吐量是越高的，但是GC停顿时间的缩短是以牺牲吞吐量和新生代空间作为代价的。比如原来10秒收集一次，每次停顿100毫秒，现在变成5秒收集一次，每次停顿70毫秒。停顿时间下降的同时，吞吐量也下降了。

**4、Serial Old收集器**是Serial收集器的老年代版本，也是一个单线程收集器，采用“标记-整理算法”进行回收。其运行过程与Serial收集器一样。

Serial Old收集器的主要意义也是在于给Client模式下的虚拟机使用。如果在Server模式下，那么它主要还有两大用途：一种用途是在JDK 1.5以及之前的版本中与Parallel Scavenge收集器搭配使用，另一种用途就是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

**5、Parallel Old收集器**是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法进行垃圾回收。其通常与Parallel Scavenge收集器配合使用，“吞吐量优先”收集器是这个组合的特点，在注重吞吐量和CPU资源敏感的场合，都可以使用这个组合。

**6、CMS（Concurrent 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，基于“标记-清除”算法，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的（有的过程也是StopTheWorld）

CMS分为四个步骤：初始标记（GCRoots能直接关联到的对象，速度快，可达性分析，Stop The World），并发标记（可达性分析），重新标记（修正并发标记期间因用户程序继续运作而导致的变动，速度快，Stop The World），并发清除

CMS的优点很明显：并发收集、低停顿。由于进行垃圾收集的时间主要耗在并发标记与并发清除这两个过程，虽然初始标记和重新标记仍然需要暂停用户线程，但是从总体上看，这部分占用的时间相比其他两个步骤很小，所以可以认为是低停顿的。

缺点：

对CPU资源太敏感，这点可以这么理解，虽然在并发标记阶段用户线程没有暂停，但是由于收集器占用了一部分CPU资源，导致程序的响应速度变慢

CMS收集器无法处理浮动垃圾。所谓的“浮动垃圾”，就是在并发标记阶段，由于用户程序在运行，那么自然就会有新的垃圾产生，这部分垃圾被标记过后，CMS无法在当次集中处理它们（为什么？原因在于CMS是以获取最短停顿时间为目标的，自然不可能在一次垃圾处理过程中花费太多时间），只好在下一次GC的时候处理。这部分未处理的垃圾就称为“浮动垃圾”。由于垃圾收集阶段用户线程还需要运行，那就不能等老年代几乎全满了再收集，一般达到92%时就开始收集，而CMS运行期间预留的内存无法满足程序需要，就会出现“Concurrent Mode Failure”，此时将启动备用方案serial old

由于CMS收集器是基于“标记-清除”算法的（可能是为了时间短），前面说过这个算法会导致大量的空间碎片的产生，一旦空间碎片过多，大对象就没办法给其分配内存,那么即使内存还有剩余空间容纳这个大对象，但是却没有连续的足够大的空间放下这个对象，所以虚拟机就会触发一次Full GC。



在使用CMS收集老年代时，新生代只能选用ParNew或者Serial收集器中的一个（CMS与其他不配套，其他的没有使用传统的GC收集器框架）

**7、G1（Garbage-First）收集器**，JDK1.7才开始商用。使用G1收集器时，Java堆内存布局与其他收集器有很大差别，它将整个Java堆分为多个大小相等的独立区域（Region），虽然还保留新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，他们都是Region（不需要连续）的集合。

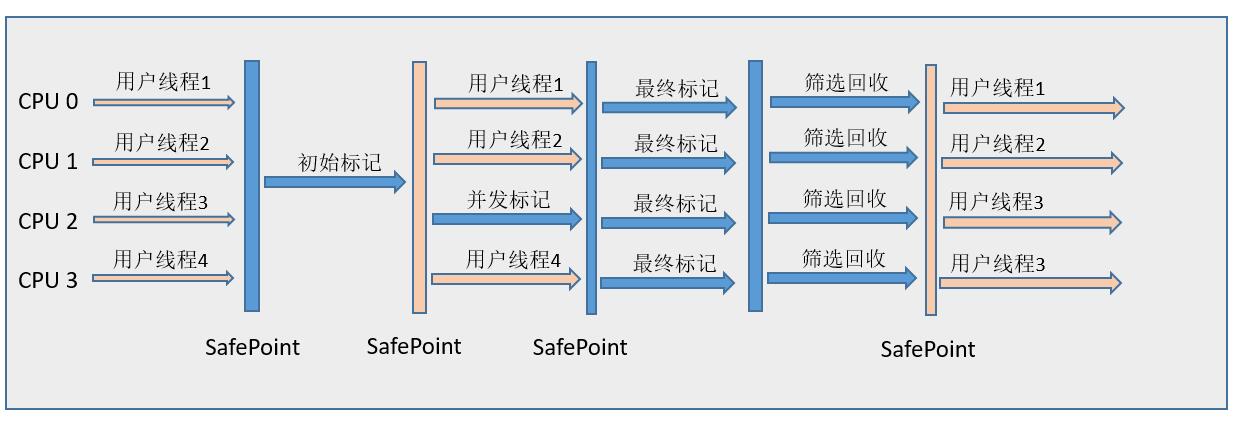
特点：并行与并发。分代收集（不需要其他收集器配合）。空间整合（整体来看采用“标记-整理”，局部（两个Region之间）采用复制）。可预测的停顿。

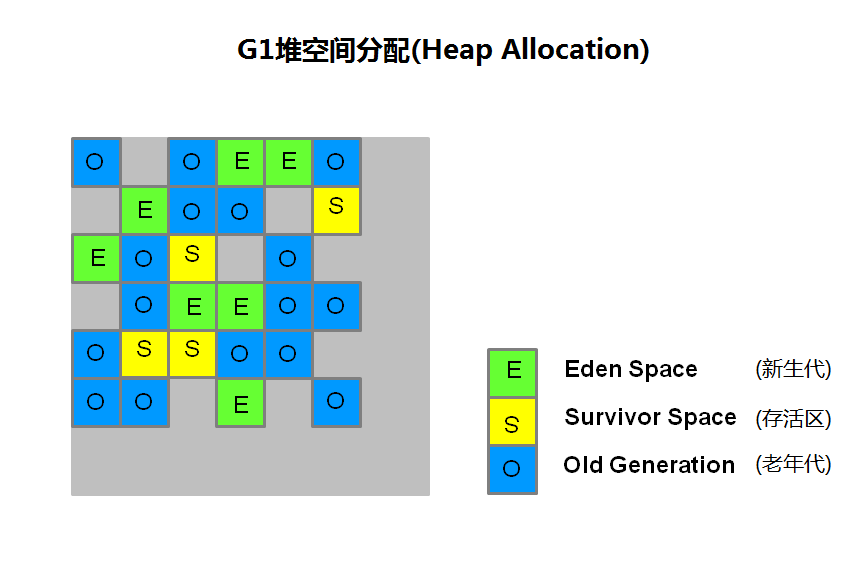
G1跟踪各个Region里面的垃圾堆积价值大小（回收所获得的空间大小以及回收所需的时间），在后台维护一个优先列表，**每次优先收集价值最大的Region**（所以叫Garbage-First），从而保证了G1在有限时间内可以获取尽可能高的收集效率。

（老年代）过程：初始标记（Stop The World）、并发标记、最终标记（Stop The World）、筛选回收（Stop The World）

G1的YoungGC就是将E区和S区复制到灰色的空白区。

G1中有Humongous区（巨大区）用于存放比标准块大50%的对象





**JVM垃圾回收机制**

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集（有eden和survivor供复制，有老年代最分配担保）。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或者“标记-整理”算法来进行回收。

发生Minor GC，采用复制算法，发现

1. 复制对象无法全部放入Survivor，只好通过分配担保机制提前转移到老年代中
2. 大对象（长字符串或长数组等需要大量连续空间的对象）直接进入老年代（防止大对象在eden和Survivor中经常复制）通过-XX:PretenureSizeThreshold参数设置（如3MB），大于这个参数的直接进入老年代
3. 长期存活对象进入老年代（默认15岁）

Minor GC：新对象先放入eden区，当eden满了会触发Minor GC。

Full GC（等于Major GC）：

1、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

2、老年代空间不足时触发Full GC，只有在新生代对象转入或创建为大对象、大数组时才会出现不足的现象（大对象直接进入老年代），分配担保

3、永久代满（永久代JDK8被移除）

优化Full GC本身不会先进行Minor GC，我们可以配置，让Full GC之前先进行一次

Minor GC，因为老年代很多对象都会引用到新生代的对象，先进行一次Minor GC可以提高老年代GC的速度。

在jvm分带垃圾回收机制中，将应用程序可用的堆空间分为年轻代和老年代，又将年轻代分为eden区、from区、to区，新建对象总是在eden区中被创建，当eden区空间已满，就触发一次Minor gc，将还被使用的对象复制到from区，这样整个eden区都是未被使用的空间，可供继续创建对象，当eden区再次用完，再触发一次Minor gc，将eden区和from区还在被使用的对象复制到to区，下一次Minor gc则是将eden区和to区还被使用的对象复制到from区。因此，经过多次Minor gc，某些对象会在from区和to区多次复制，如果超过某个阈值对象还未被释放，则将对象复制到老年代。如果老年代空间也已用完，那么就会触发full gc，即所谓的全量回收。

永久代的垃圾回收主要有两部分：废弃常量和无用的类。如没有任何String对象引用“abc”。在大量使用反射、动态代理、CGlib等ByteCode框架，动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载功能（回收永久代），以保证永久代不会溢出。

**JVM参数**

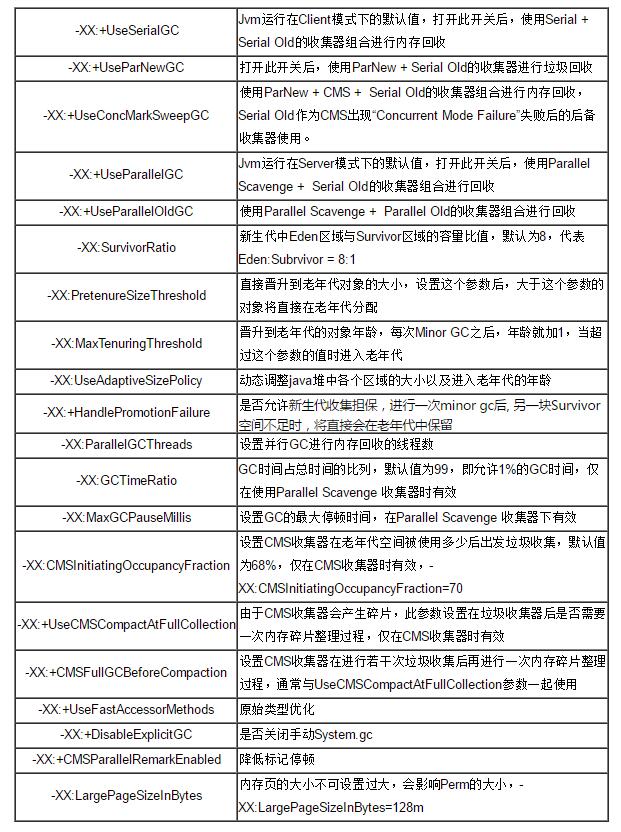
java -Xms256m -Xmx1024m在命令行

eclipse、tomcat都是java写的程序，它们也使用jvm，它们相当于jvm的一部分，我们的程序在它们之上运行，所以可以通过它们设置jvm参数。开两个java程序会启动两个jvm实例。

修改垃圾收集器：-XX：+UseParNewGC（在eclipse或Tomcat修改catalina.bat文件）

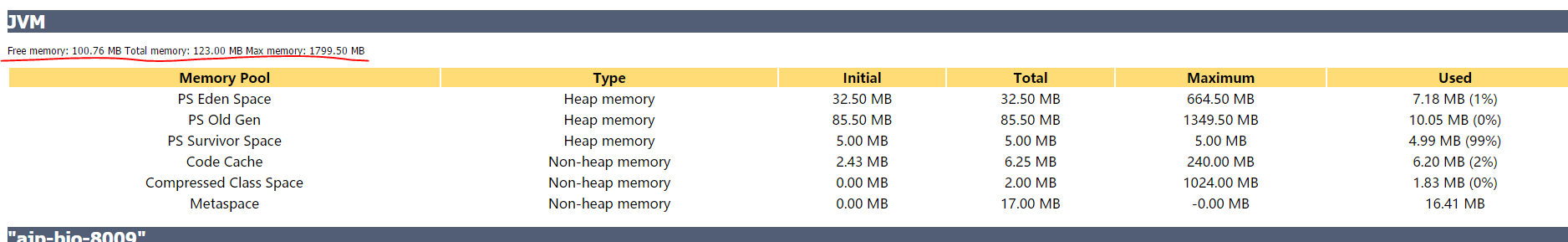
查看使用的垃圾收集器：可以通过命令行输入：jvisualvm，打开Java VisualVm查看，

Eclipse中，-XX:+UseG1GC，默认使用G1



Eclipse默认堆大小256m到1024m

Tomcat初始堆大小

128mb（8GB/64）（开启tomcat进入localhost:8080后，点击server status后查看）

JVM初始分配的堆内存由-Xms指定，默认是物理内存的1/64；JVM最大分配的堆内存由-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

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。

首先JVM内存限制于实际的最大物理内存，假设物理内存无限大的话，JVM内存的最大值跟[操作系统](http://lib.csdn.net/base/operatingsystem)有很大的关系。简单的说就32位处理器虽然可控内存空间有4GB,但是具体的操作系统会给一个限制，  
 这个限制一般是2GB-3GB（一般来说Windows系统下为1.5G-2G，[Linux](http://lib.csdn.net/base/linux)系统下为2G-3G），而64bit以上的处理器就不会有限制了。

Tomcat修改堆大小，修改catalina.bat文件

加上这句



或打开tomcat8w.exe，根据图形化界面修改

-Xms :设置Java堆栈的初始化大小，默认物理内存的1/64(<1GB)  
-Xmx :设置最大的java堆大小，默认物理内存的1/4(<1GB)  
-Xmn :设置年轻代区大小，整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。增大年轻代后会减小老年代  
-Xss :设置java线程堆栈大小，JDK5.0以后每个线程堆栈大小为1M,以前为256K  
-XX:PermSize：设置永久代初始大小，默认物理内存的1/64(<1GB)

-XX:MaxPermSize :设置永久代最大大小，默认物理内存的1/4(<1GB)  
-XX:NewRatio :设置年轻代和老年代的比值，默认4  
-XX:NewSize :设置年轻代的大小（JDK1.3/1.4）

-XX:SurvivorRatio=n :设置年轻代中eden与Survivor比值，默认8

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话,则年轻代对象不经过Survivor区,直接进入年老代. 对于年老代比较多的应用,可以提高效率.

-XX:PretenureSizeThreshold参数设置（如3MB），大于这个参数的直接进入老年代

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

发现虚拟机频繁GC，扩大堆大小，使用jconsole或jvisualvm查看

**java.lang.ThreadLocal<T>**

<http://blog.csdn.net/winwill2012/article/details/71625570>

http://blog.csdn.net/huachao1001/article/details/51734973

概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

ThreadLocal 并不能替代同步机制，两者面向的问题领域不同。

　　1：同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之间进行通信的有效方式；

2：而threadLocal是隔离多个线程的数据共享，从根本上就不在多个线程之间共享变量，这样当然不需要对多个线程进行同步了。（每个线程有单独的数据，在线程内共享，在线程外独立）

最常见的ThreadLocal使用场景为用来解决数据库连接、Session管理等

private static ThreadLocal<Connection> connectionHolder =

new ThreadLocal<Connection>() {

protected Connection initialValue() {

return DriverManager.getConnection(DB\_URL);

}

};

public static Connection getConnection() {

return connectionHolder.get();

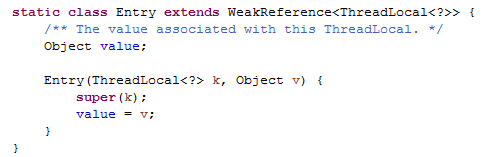
}

JDK中建议ThreadLocal实例通常来说都是private static类型的.

每个ThreadLocal类创建一个Map，然后用线程的ID作为Map的key，实例对象作为Map的value，这样就能达到各个线程的值隔离的效果。JDK最早期的ThreadLocal就是这样设计的。

**ThreadLocal底层实现**

ThreadLocal类中有一个静态内部类ThreadLocalMap，ThreadLocalMap相当于一个哈希表，用private Entry[] table;存储数据，而Entry是一个实现了弱引用（下一次gc时就会被回收）的内部类，它的key弱引用。



ThreadLocalMap的初始大小为16，负载因子为2/3（即超过了长度的三分之二就要扩容），每次扩容为原来的2倍，可以保证大小始终为2的N次方。ThreadLocalMap解决哈希冲突的方法与hashmap不同（数组+链表），ThreadLocalMap如果i位置已经存储了对象，那么就往后挪一个位置依次类推，直到找到空的位置，再将对象存放。另外，在最后还需要判断一下当前的存储的对象个数是否已经超出了阈值（threshold的值）大小，如果超出了，需要重新扩充并将所有的对象重新计算位置（rehash函数来实现）。rehash函数里面先调用了expungeStaleEntries函数，然后再判断当前存储对象的大小是否超出了阈值的3/4。如果超出了，再扩容。看的有点混乱。为什么不直接扩容并重新摆放对象？为啥要搞成这么复杂？

其实，ThreadLocalMap里面存储的Entry对象本质上是一个WeakReference<ThreadLocal>。也就是说，ThreadLocalMap里面存储的对象本质是一个对ThreadLocal对象的弱引用，该ThreadLocal随时可能会被回收！即导致ThreadLocalMap里面对应的Value的Key是null。我们需要把这样的Entry给清除掉，不要让它们占坑。

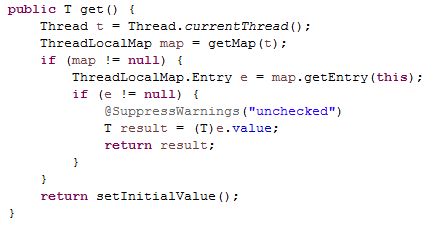
expungeStaleEntries函数就是做这样的清理工作，清理完后，实际存储的对象数量自然会减少。这时候再判断，如果存储对象数量还是过多，才会扩容（resize）

ThreadLocalMap中根据key值获得entry对象的方法是，得到table中的位置i（根据len-1，低位掩码），如果没找到，则有可能发生哈希冲突，所以调用getEntryAfterMiss函数从当前位置继续向后找。



threadLocalHashCode方法就是在ThreadLocal中定义了一个static的atomicInteger，每次调用threadLocalHashCode方法都要给它加上一个固定的值（不知道为什么）

ThreadLocal的get方法。

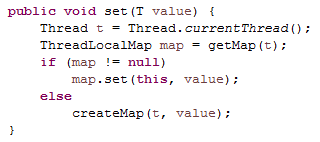


也就是说，每个线程中内部都有一个ThreadLocalMap类型的threadLocals，



在调用ThreadLocal类的get方法时，先获得当前线程中存储的threadLocals（是一个map），如果该map为空，则调用setInitialValue（给当前线程的map new一个ThreadLocalMap， 传入initialValue方法的初值（没重写的话为null）），并返回该初值。如果有map的话，就map.getEntry(this)，注意这里的key是this，也就是该ThreadLocal类。

ThreadLocal的set方法



注意，因为这里set的key是this（ThreadLocal类），所以每一个线程在每一个ThreadLocal中只能对应一个value。若想保存多个value，则需要创建多个ThreadLocal类。

总结：每一个Thread内部都封装了一个ThreadLocalMap，这个map的key是ThreadLocal（map.getEntry(this)），value是具体的变量对象。也就是说，一个thread可以保存多个threadlocal，而正因为threadlocal保存在thread内部，多线程并发时，每次处理的都是自己内部的数据。

流程：新建一个ThreadLocal类（名叫tl），重写它的initialValue方法，当一个线程调用tl的get方法时（此时进入tl类内部），先获得调用get方法线程保存的map，用map.getEntry(this)获得对应的变量对象（因为调用的是tl的get方法，所以this指针为这个tl）

与早期JDK中设计的区别

1这样设计之后每个Map的Entry数量变小了：之前是Thread的数量，现在是ThreadLocal的数量，能提高性能，据说性能的提升不是一点两点(没有亲测)

2当Thread销毁之后对应的ThreadLocalMap也就随之销毁了，能减少内存使用量。

**弱引用**

threadlocal里面使用了一个存在弱引用的map,当释放掉threadlocal的强引用以后,map里面的value却没有被回收.而这块value永远不会被访问到了. 所以存在着内存泄露. 最好的做法是将调用threadlocal的remove方法.

比如ThreadLocal<String> tl = new ThreadLocal<String>()；当tl=null时，即释放了强引用，此时这个ThreadLocal会被gc掉，每个线程中ThreadLocalMap的key（key为threadlocal），如果key为强引用，则这个ThreadLocal不会被gc，就会发生我已经不想要这个ThreadLocal了，但还没被gc。所以为弱引用。value是我threadlocal中initial出来的

当把threadlocal实例置为null以后,没有任何强引用指向threadlocal实例,所以threadlocal将会被gc回收. 但是,我们的value却不能回收,因为存在一条从current thread连接过来的强引用. 只有当前thread结束以后, current thread就不会存在栈中,强引用断开, Current Thread, Map, value将全部被GC回收.

　　所以得出一个结论就是只要这个线程对象被gc回收，就不会出现内存泄露，但在threadLocal设为null和线程结束这段时间不会被回收的，就发生了我们认为的内存泄露。其实这是一个对概念理解的不一致，也没什么好争论的。最要命的是线程对象不被回收的情况，这就发生了真正意义上的内存泄露。比如使用线程池的时候，线程结束是不会销毁的，会再次使用的。就可能出现内存泄露。

　　PS.Java为了最小化减少内存泄露的可能性和影响，在ThreadLocal的get,set的时候都会清除线程Map里所有key为null的value。所以最怕的情况就是，threadLocal对象设null了，开始发生“内存泄露”，然后使用线程池，这个线程结束，线程放回线程池中不销毁，这个线程一直不被使用，或者分配使用了又不再调用get,set方法，那么这个期间就会发生真正的内存泄露。

**AVL树**

http://www.cnblogs.com/skywang12345/p/3576969.html

LL（左左，单旋转），RR（右右，单旋转），LR（左右，先RR在LL），RL（右左，先LL再RR）

**常用排序算法**

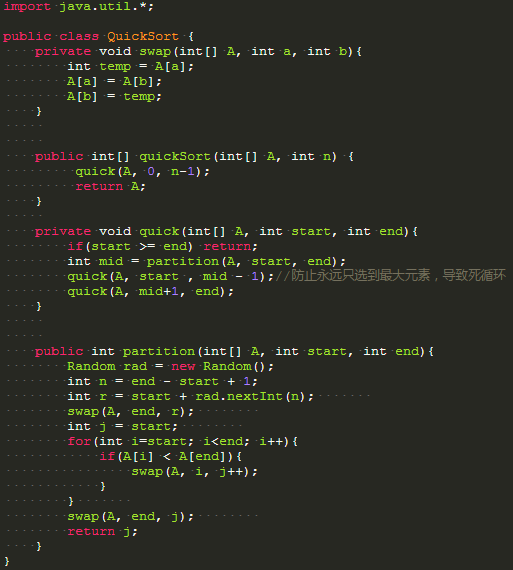


快排的空间复杂度是O(lgn)，归并的额外空间复杂度是O(n)，上面写错了

选择排序：从n个数中选出最小的，放在最前面，再从n-1选出最小的放在第二个。。。

快排：最好的情况是枢纽元选取得当，每次都能均匀的划分序列。 时间复杂度O(nlogn)  
最坏情况是枢纽元为最大或者最小数字，那么所有数都划分到一个序列去了时间复杂度为O(n^2)。空间复杂度是递归

swap交换两个数，如果使用异或，不能自己跟自己交换，因为自己异或自己得0，需要提前判断。



插入排序的时间复杂度分析。在最坏情况下，数组完全逆序，插入第2个元素时要考察前1个元素，插入第3个元素时，要考虑前2个元素，……，插入第N个元素，要考虑前 N - 1 个元素。因此，最坏情况下的比较次数是 1 + 2 + 3 + ... + (N - 1)，等差数列求和，结果为 N^2 / 2，所以最坏情况下的复杂度为 O(N^2)。

最好情况下，数组已经是有序的，每插入一个元素，只需要考查前一个元素，因此最好情况下，插入排序的时间复杂度为O(N)。

冒泡排序最好情况，数组已经有序，虽然还是两层循环，但内层循环每次只用比较一次，不需要交换（A[j]<A[j+1]）。

希尔排序：插入排序的改良版，往前插入的步长不是1，比如是3（步长选择越优，时间复杂度越低，步长选择越差，越趋近于O(n2)，进行完步长为3的调整后，再进行步长为2的，再进行步长为1的，希尔排序最终都会以步长为1的结束）。步长一开始是feet=length/2，然后循环一次feet=feet/2

shell排序的平均复杂度是O(nlogn)~O(n2)，最好的情况O(n1.3)，最坏的情况O(n2)

基数排序：把n个数，按个位放入0-9个桶，再按顺序取出，再按十位。。百位

该算法所花的时间基本是在把元素分配到桶里和把元素从桶里串起来；把元素分配到桶里：循环 length 次；

把元素从桶里串起来：这个计算有点麻烦，看似两个循环，其实第二循环是根据桶里面的元素而定的，可以表示为：k×buckerCount；其中 k 表示某个桶中的元素个数，buckerCount 则表示存放元素的桶个数；

有几种特殊情况：

第一、所有的元素都存放在一个桶内：k = length，buckerCount = 1；

第二、所有的元素平均分配到每个桶中：k = length/ bukerCount，buckerCount = 10；（这里已经固定了10个桶（0-9））

所以平均情况下收集部分所花的时间为：length （k×buckerCount）（也就是元素长度 n）

综上所述：时间复杂度为：posCount \* (length + length) ；其中 posCount 为数组中最大元素的最高位数（位数）；简化下得：O( k\*n ) ；其中k为常数，n为元素个数；

计数排序：排列员工身高，时间复杂度O（n+k），k为桶的个数（一个桶的一次性倒出）



堆排序：

http://www.cnblogs.com/mengdd/archive/2012/11/30/2796845.html

堆排序是一种选择排序，其时间复杂度为O(nlogn)。堆是一个完全二叉树，父节点均大于等于或小于等于其两个子节点（只有堆顶元素是最大或最小，其他无序），一般采用数组存储堆，如果下表从0开始，i的子节点为2i+1和2i+2，i的父节点(i-1)/2.

堆排序的步骤：

一般在输出堆顶元素之后，视为将这个元素排除，然后用表中最后一个元素填补它的位置，自上向下进行调整：首先将堆顶元素和它的左右子树的根结点进行比较，把最小的元素交换到堆顶；然后顺着被破坏的路径一路调整下去，直至叶子结点，就得到新的堆。我们称这个自堆顶至叶子的调整过程为“筛选”。

　　从无序序列建立堆的过程就是一个反复“筛选”的过程。

1、构造初始堆(O(n))

　　初始化堆的时候是对所有的非叶子结点进行筛选。

　　最后一个非叶子节点的下标是(n/2-1)，所以筛选只需要从第(n/2-1)个元素开始，从后往前进行调整。

　　然后从最后一个非叶子结点开始，每次都是从父结点、左孩子、右孩子中进行比较交换，交换可能会引起孩子结点不满足堆的性质，所以每次交换之后需要重新对被交换的孩子结点进行调整。

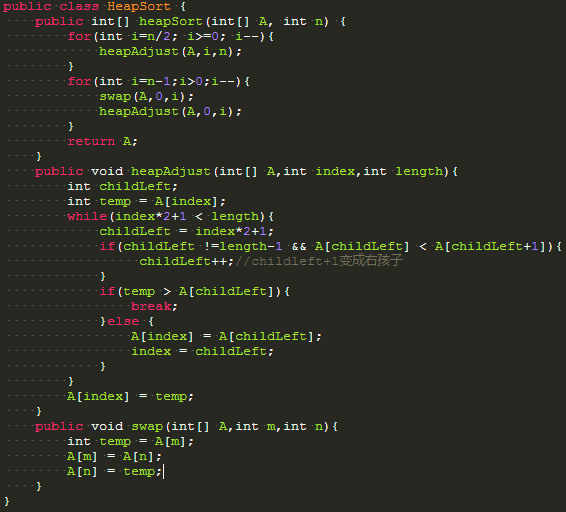
2、进行堆排序(O(nlogn))

　　有了初始堆之后就可以进行排序了。

　　堆排序是一种选择排序。建立的初始堆为初始的无序区。

　　排序开始，首先输出堆顶元素（因为它是最值），将堆顶元素和最后一个元素交换，这样，第n个位置（即最后一个位置）作为有序区，前n-1个位置仍是无序区，对无序区（筛选前n-1个位置）进行调整（此时只有堆顶元素不满足堆的要求），得到堆之后，再交换堆顶和最后一个元素，这样有序区长度变为2。

不断进行此操作，将剩下的元素重新调整为堆，然后输出堆顶元素到有序区。每次交换都导致无序区-1，有序区+1。不断重复此过程直到有序区长度增长为n-1，排序完成。（需要用大根堆实现，因为每次堆顶元素（最大值）交换到最后）



堆的插入与删除

堆是完全二叉树，插入时在最后一个位置添加节点，然后自底向上调整堆。

删除时，只能删除根节点，将堆的最后一个节点放到根节点，然后自顶向下调整堆。

**进程与线程**

（1）一个程序至少有一个进程,一个进程至少有一个线程。

（2）线程(Thread)是进程的一个实体，是CPU调度和分派的基本单位，是进程中负责程序执行的执行单元，也称为执行路径。进程是系统资源分配的基本单位。

（3）进程拥有独立的地址空间，而多个线程共享内存。从而线程效率更高；

（4）线程是轻量化的进程，是程序执行流的最小单位；由线程ID、程序计数器、寄存器集合（cpu只有一组寄存器，这里每个线程拥有自己的寄存器是说切换到自己的时候就把堆栈中保存的之前的值又放到寄存器中，切换时保存, 切回来时恢复）和堆栈组成（堆栈就是栈）；线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。

（5）地址空间和其它资源（如打开的文件、全局变量）：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。

（6）通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量、静态变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

（7）调度和切换：线程上下文切换比进程上下文切换要快得多。由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度，从而显著提高系统资源的利用率和吞吐量。

（8）多进程比多线程程序要健壮。多线程程序一个线程死掉整个进程就死掉了，但是在保护模式下，一个进程死掉对另一个进程没有直接影响。

其中进程包括地址空间，打开的文件，账户信息，预告的报警，子进程。而线程包括堆栈，程序计数器（主要在程序编译成汇编之后用来计数程序运行），状态信息（就绪，运行，阻塞）

线程状态：准备，就绪，运行，阻塞，结束

**线程调度**

一般线程调度模式分为两种——抢占式调度和协同式调度（非抢占式）。抢占式调度（按优先级分配执行时间，一个没执行完，可以换另一个）指的是每条线程执行的时间、线程的切换都由系统控制，系统控制指的是在系统某种运行机制下，可能每条线程都分同样的执行时间片，也可能是某些线程执行的时间片较长，甚至某些线程得不到执行的时间片。在这种机制下，一个线程的堵塞不会导致整个进程堵塞。协同式调度（一个一个执行）指某一线程执行完后主动通知系统切换到另一线程上执行，这种模式就像接力赛一样，一个人跑完自己的路程就把接力棒交接给下一个人，下个人继续往下跑。线程的执行时间由线程本身控制，线程切换可以预知，不存在多线程同步问题，但它有一个致命弱点：如果一个线程编写有问题，运行到一半就一直堵塞，那么可能导致整个系统崩溃。

JVM规范中规定每个线程都有优先级，且优先级越高越优先执行，但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间。Java使用的线程调度是抢占式调度，在JVM中体现为让可运行池中优先级高的线程拥有CPU使用权，如果可运行池中线程优先级一样则随机选择线程，但要注意的是实际上一个绝对时间点只有一个线程在运行（这里是相对于一个CPU来说，如果你的机器是多核的还是可能多个线程同时运行的），直到此线程进入非可运行状态或另一个具有更高优先级的线程进入可运行线程池，才会使之让出CPU的使用权，更高优先级的线程抢占了优先级低的线程的CPU。

Java中线程会按优先级分配CPU时间片运行，那么线程什么时候放弃CPU的使用权？可以归类成三种情况：

1当前运行线程主动放弃CPU，JVM暂时放弃CPU操作（基于时间片轮转调度的JVM操作系统不会让线程永久放弃CPU，或者说放弃本次时间片的执行权），例如调用yield()方法。

2当前运行线程因为某些原因进入阻塞状态，例如阻塞在I/O上，wait()。

3当前运行线程结束，即运行完run()方法里面的任务。（高优先级的线程运行的概率大）

wait(),notify()这也就是用户级别的线程调度，还有一种就是当一个线程进行io操作的时候，会空闲cpu，因为io读取是cpu的几十倍，因此其他线程会对该cpu进行利用，也就是内核级别的线程调度

CPU通过时间片分配[**算法**](http://lib.csdn.net/base/datastructure)来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态（寄存器、堆栈、程序计数器、线程id），以便下次切换回这个任务时，可以再次加载这个任务的状态，**从任务保存到再加载的过程就是一次上下文切换**。

**进程间通信的方法**

IPC（interprocess communication,进程间通信）

1、管道(pipe)：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

2、有名管道 (named pipe)：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

3、高级管道(popen)：将另一个程序当做一个新的进程在当前程序进程中启动，则它算是当前程序的子进程，这种方式我们成为高级管道方式。

4、信号量(semophore)：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

5、消息队列(message queue)：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

6、信号(sinal)：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

7、共享内存(shared memory)：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

8、套接字(socket)：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

**线程间通信的方法**

锁机制：包括互斥锁、条件变量、读写锁

\*互斥锁提供了以排他方式防止数据结构被并发修改的方法。

\*读写锁允许多个线程同时读共享数据，而对写操作是互斥的。

\*条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量

信号机制(Signal)：类似进程间的信号处理

线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。线程间通信可以通过全局变量实现。

多线程不是真的能提高效率，是能够合理使用资源，I/O等操作速度慢。

并发（某一时段，时间片调度）和并行（同一时刻处理多个任务）的区别就是一个处理器同时处理多个任务和多个处理器或者是多核的处理器同时处理多个不同的任务。

并发一条车道，很多车轮着走，并行多条车道。

多核是cpu，操作系统来处理的。效果就是你的多线程不象单核是再一个CPU上，大家轮流调度。多核就可以让多个线程分配到不同CPU，这样可以提高并发能力，性能也能提升

**equals**

Equals与 == 的区别，==是比较值，equals是比较内容。

Equals是object类的一个方法，源码中是比较两者地址是否相同（用==号），string类等重写了这个方法（还要重写hashcode（）），如果地址相同直接返回true，如果地址不同，每一位比较char是否相同

equals()和hashCode()方法是用来在同一类中做比较用的，尤其是在容器里如set存放同一类对象时用来判断放入的对象是否重复。

equals()相等的两个对象，hashcode()一定相等，equals()不相等的两个对象，却并不能证明他们的hashcode()不相等。换句话说，equals()方法不相等的两个对象，hashCode()有可能相等。（我的理解是由于哈希码在生成的时候产生冲突造成的）

当我们重写一个对象的equals方法，就必须重写他的hashCode方法，不过不重写他的hashCode方法的话，Object对象中的hashCode方法始终返回的是一个对象的hash地址，而这个地址是永远不相等的。所以这时候即使是重写了equals方法，也不会有特定的效果的，因为hashCode方法如果都不相等的话，就不会调用equals方法进行比较了，所以没有意义了。

如果一个类的hashCode()方法没有遵循上述要求，那么，当这个类的两个实例对象用equals()方法比较的结果相等时，他们本来应该无法被同时存储进set集合中，但是，如果将他们存储进HashSet集合中时，由于他们的hashCode()方法的返回值不同(Object中的hashCode方法返回值是永远不同的)，第二个对象首先按照哈希码计算可能被放进与第一个对象不同的区域中，这样，它就不可能与第一个对象进行equals方法比较了，也就可能被存储进HashSet集合中了，Object类中的hashCode()方法不能满足对象被存入到HashSet中的要求，因为它的返回值是通过对象的内存地址推算出来的，同一个对象在程序运行期间的任何时候返回的哈希值都是始终不变的，所以，只要是两个不同的实例对象，即使他们的equals方法比较结果相等，他们默认的hashCode方法的返回值是不同的。

**基本数据类型存放**

**class** demo{

**private** **int** a = 3;

**private** **void** fun(){

**int** b = 4;

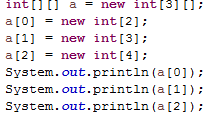
}

}

b存放在栈中，a随实例对象存放在堆中。

Java的8种基本类型(Byte, Short, Integer, Long, Character, Boolean, Float, Double), 除Float和Double以外, 其它六种都实现了常量池, 但是它们只在大于等于-128并且小于等于127时才使用常量池。（2的7次方，即一个字节）

**Java中二维数组的存放**



如果a是局部变量，则a存储在栈中，而数组存储在堆中，可以定义列数不相同的数组，二维数组的第一维“行”，存储了第二维的数组，相当于存储了数组的数组，第一维中是第二维中的地址

**父类子类初始化过程**

父类中有无参构造函数，子类构造函数默认调用super()，子类构造函数中可以不写super();若父类中只有有参构造函数，子类中必须在第一行调用super(参数); 如果没有任何构造函数，系统会默认有一个无参构造函数。

初始化过程：

1.初始化父类中的静态成员变量和静态代码块 （先声明的先执行）；

2.初始化子类中的静态成员变量和静态代码块（先声明的先执行）；

3.初始化父类的普通成员变量和构造代码块（先声明的先执行） ，再执行父类的构造方法；

4.初始化子类的普通成员变量和构造代码块（先声明的先执行） ，再执行子类的构造方法；

构造函数和构造代码块每次创建对象都会调用，静态代码块只调用一次。

静态代码块-->(主方法)-->构造代码块-->构造函数

**类加载过程**

虚拟机把描述类的数据从Class文件加载到内存，最终生成可以被虚拟机直接使用的java类型，这就是虚拟机的类加载机制。与那些在编译时需要进行连接工作的语言不同，java类型的加载、连接和初始化都是在**程序运行期间**完成的。（可以实现反射，OSGi等技术）

类的生命周期：加载、连接（验证、准备、解析）、初始化、使用、卸载。

加载、连接（验证、准备、解析）、初始化是类的加载过程。

虚拟机规范中没有强制约束什么时候开始类加载，但以下几种情况必须开始初始化（而加载、验证、准备必须在初始化之前开始）：

第一：生成该类对象的时候（new关键字）、读取或设置一个类的静态字段（被final修饰、已在编译期放入常量池的静态字段除外）、调用一个类的静态方法。会初始化该类及该类的所有父类；

第二：初始化一个类时，如果其父类还没有被初始化，则先初始化父类；

第三：class．forName("类名")，使用java.lang.reflect包的反射；

第四：虚拟机启动时，main方法的类；

**加载**

在加载阶段，虚拟机主要完成三件事：

1.通过一个类的全限定名来获取定义此类的二进制字节流。

2.将这个字节流所代表的静态存储结构转化为方法区域的运行时数据结构。

3.在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区域数据的访问入口。

数组类本身不通过类加载器创建，由java虚拟机直接创建。但数组类的元素（指数组去掉维度后的类型）最终是由类加载器创建。

加载阶段完成后，虚拟机外部的二进制字节流（如.class文件、jar、war包）就按照虚拟机所需要的格式存储在方法区之中，然后在内存中实例化一个java.lang.Class类的对象（对于HotSpot，Class对象比较特殊，它虽然是对象，但是存放在方法区里）

**验证**

验证的目的是为了确保Class文件中的字节流包含的信息符合当前虚拟机的要求、规范，而且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：文件格式的验证、元数据的验证、字节码验证和符号引用验证。

**准备**

准备阶段是正式为**类变量**分配内存并设置类变量初始值的阶段，**这些内存都将在方法区中分配**。

1、这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。

2、这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。

假设一个类变量的定义为：public static int value = 3；

那么变量value在准备阶段过后的初始值为0，而不是3，因为这时候尚未开始执行任何Java方法，所以把value赋值为3的动作将在初始化阶段才会执行。

对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。

对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被final修饰的常量则既可以在声明时显式地为其赋值（只被final修饰是常量，存放在方法区），也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。

对于引用数据类型reference来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。

如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

3、如果类字段的字段属性表中存在ConstantValue属性（即同时被final和static修饰，且为基本类型或String），那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值。（因为static的要被赋默认值0，而final又不能变，所以提前赋值）

假设上面的类变量value被定义为： public static final int value = 3；

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为3。

方法区是存放虚拟机加载类的相关信息，如类、静态变量和常量

类加载的时候将所有方法的字节码放到方法区的

**解析**

加载、验证、准备、初始化、卸载这几个阶段的顺序是确定的，而解析阶段不一定，可以延迟执行，在某些情况下可以在初始化阶段之后再开始，这是为了支持java语言的运行时绑定。

解析阶段是虚拟机将**常量池中的**符号引用转化为直接引用的过程。包括类或接口的解析、字段解析、类方法解析、接口方法解析。（比如String s ="aaa",转化为 s的地址指向“aaa”的地址）

常量池中主要存放两大类常量：字面量和符号引用。**字面量比较接近于Java层面的常量概念，如文本字符串、被声明为final的常量值、-128-127的Integer包装类等**。而符号引用总结起来则包括了下面三类常量：

1、类和接口的全限定名（即带有包名的Class名，如：org.lxh.test.TestClass）

2、字段的名称和描述符（private、static等描述符）

3、方法的名称和描述符（private、static等描述符）

虚拟机在加载Class文件时才会进行动态连接，也就是说，Class文件中不会保存各个方法和字段的最终内存布局信息，因此，这些字段和方法的符号引用不经过转换是无法直接被虚拟机使用的。当虚拟机运行时，需要从常量池中获得对应的符号引用，再在类加载过程中的解析阶段将其替换为直接引用，并翻译到具体的内存地址中。

前面说解析阶段可能开始于初始化之前，也可能在初始化之后开始，虚拟机会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析（初始化之前），还是等到一个符号引用将要被使用前才去解析它（初始化之后）。

这里说明下符号引用和直接引用的区别与关联：

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

直接引用：直接引用可以是**直接指向目标的指针**、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那说明引用的目标必定已经存在于内存之中了。

**初始化**

初始化是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的Java程序代码。在准备阶段，类变量已经被赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序指定的主观计划去初始化类变量和其他资源，如赋值。

**卸载**

该类所有的实例都已经被回收，也就是java堆中不存在该类的任何实例。

加载该类的ClassLoader已经被回收。

该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

如果以上三个条件全部满足，jvm就会在方法区垃圾回收的时候对类进行卸载，类的卸载过程其实就是在方法区中清空类信息，java类的整个生命周期就结束了。

由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。前面介绍过，Java虚拟机自带的类加载器包括根类加载器、扩展类加载器和系统类加载器。Java虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的类的Class对象，因此这些Class对象始终是可触及的。

 由用户自定义的类加载器加载的类是可以被卸载的（没有引用时）。当再次有需要时，会检查类的Class对象是否存在，如果存在会直接使用，不再重新加载；如果不存在类会被重新加载

生命周期

实例变量取决于类的实例。每创建一个实例，java虚拟机就会为实例变量分配一次内存，实例变量位于堆区中，其生命周期取决于实例的生命周期。

静态变量不属于某个实例对象，而是属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。

总而言之：静态变量生命周期就是类的开始和销毁（与类相同）

实例变量生命周期就是对象的开始和销毁（与对象相同）

**类与类加载器**

 类加载器用于实现类的加载动作

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）（c++实现，其他加载器都是java实现）

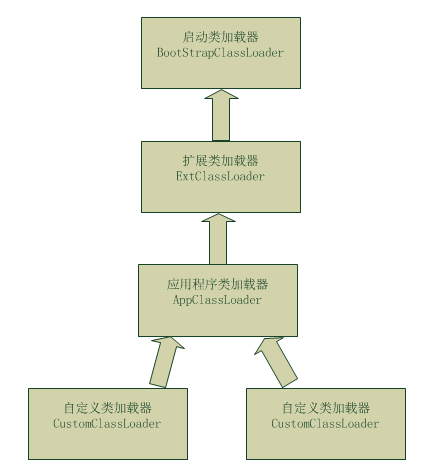
这个类加载器负责将存放在JAVA\_HOME/lib下的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用。

（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。



对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在java虚拟机中的**唯一性**，每一个类加载器，都拥有一个独立的类名称空间。比较两个类是否相等，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。（相等指类的Class对象的equals方法，使用instanceof关键字）。

JVM如何确立每个类在JVM的唯一性？类的全限定名和类加载器。

**双亲****委派模型**

双亲委派模型工作过程：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送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加载请求时，子类才会尝试加载。

如类java.lang.Object，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器，因此Object类在程序的各种类加载器环境中都是同一个类。相反如果没有使用双亲委派模型，各个类加载器自行去加载的话，如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，那系统中将会出现多个不同的Object类，java类体系中最基础的行为也就无法保证，应用程序也会变得一片混乱。

如自己定义一个object类，不会首先加载这个类，只会加载最顶层的object类。

类加载器的父类不是继承关系（子类可以调用父类），而是组合关系

破坏双亲委派模型

一般的场景中使用Java默认的类加载器即可，但有时为了达到某种目的又不得不实现自己的类加载器，例如为了达到类库的互相隔离，例如为了达到**热部署**重加载功能。

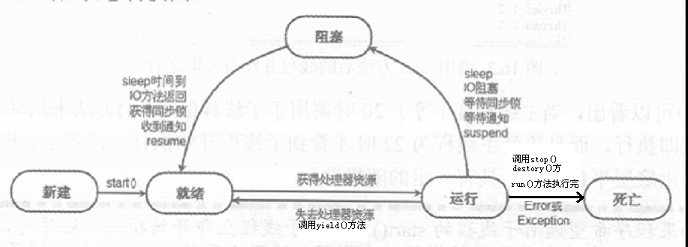
重新定义一个继承ClassLoader的TestClassLoaderN类，这个类与前面的TestClassLoader类很相似，但它除了重写findClass方法外（一般的自定义类加载器只重写findClass），还重写了loadClass方法，默认的loadClass方法是实现了双亲委派机制的逻辑，即会先让父类加载器加载，当无法加载时才由自己加载。这里为了破坏双亲委派机制必须重写loadClass方法，即这里先尝试交由System类加载器加载，加载失败才会由自己加载。它并没有优先交给父类加载器，这就打破了双亲委派机制。

**绑定**

这里简要说明下Java中的绑定：绑定指的是把一个方法的调用与方法所在的类(方法主体)关联起来，对java来说，绑定分为静态绑定和动态绑定：

静态绑定：即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对java，简单的可以理解为程序编译期的绑定。java当中的方法只有final，static，private和构造方法是前期绑定的。

动态绑定：即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在java中，几乎所有的方法都是后期绑定的。

**线程的生命周期**  


**线程的生命周期可以分为四个状态：**

**1.创建状态：**

　　当用new操作符创建一个新的线程对象时，该线程处于创建状态。

　　处于创建状态的线程只是一个空的线程对象，系统不为它分配资源。

**2.可运行状态：**

　　执行线程的**start()**方法将为线程分配必须的系统资源，安排其运行，并调用**线程体**——**run()**方法，这样就使得该线程处于可运行状态（**Runnable**）。

　　这一状态并不是运行中状态（**Running**），因为线程也许实际上并未真正运行。

**3.不可运行状态：**

**当发生下列事件时，处于运行状态的线程会转入到不可运行状态：**

　　调用了sleep()方法；

　　线程调用wait()方法等待特定条件的满足；

　　线程输入/输出阻塞。

**返回可运行状态：**

　　处于睡眠状态的线程在指定的时间过去后；

　　如果线程在等待某一条件，另一个对象必须通过notify()或notifyAll()方法通知等待线程条件的改变；

　　如果线程是因为输入输出阻塞，等待输入输出完成。

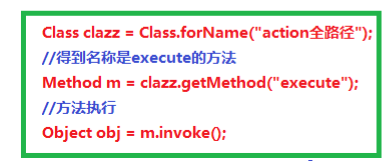
**4.消亡状态：**

　　当线程的run()方法执行结束后，该线程自然消亡。

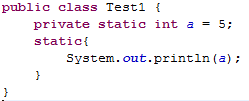
**反射**

反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。（如在运行状态中（正在运行的服务器，不用关了服务器再修改），修改配置文件，切换不同的数据库）

不用类名也可以获得class对象，this.getClass()方法，不能再static方法中使用。

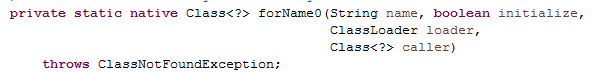


在java虚拟机规范中，必须开始初始化的情况之一就是调用Class.forName()。而加载、验证、准备必须在初始化之前开始。



调用Class.forName(“xxx.Test1”)后，开始初始化。准备阶段会给static类变量赋值0，初始化阶段会赋值5，这里输出5。

Class.forName()会调用forName0方法，使用默认的类加载器，而forName0是native方法。



native关键字说明其修饰的方法是一个原生态方法，方法对应的实现不是在当前文件，而是在用其他语言（如C和C++）实现的文件中。

1. Class.forName()必须开始初始化
2. 可以在Class.forName()传入类加载器classLoader，findClass和loadClass方法，重写loadClass可以破坏双亲委派模型。
3. 调用Class.forName后，完成类的加载过程。即该类被加载进了内存。每个类都有一个Class对象，存放在方法区中。
4. Class clazz = Class.forName(“aaa”);得到Class类型的对象后，可以生成实例，并调用方法。

**反射用途**

当我们在使用IDE(如Eclipse，IDEA)时，当我们输入一个对象或类并想调用它的属性或方法时，一按点号，编译器就会自动列出它的属性或方法，这里就会用到反射。

很多框架（比如Spring）都是配置化的（比如通过XML文件配置JavaBean,Action之类的），为了保证框架的通用性，它们可能需要根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射——运行时动态加载需要加载的对象。

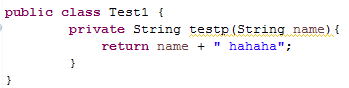
**反射能够使用私有的方法属性**

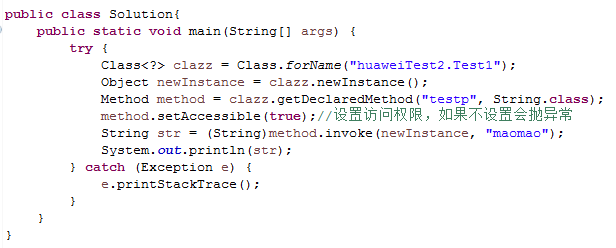
|  |  |
| --- | --- |
| 可以访问私有 | 只能访问公有 |
| Class.getDeclaredField(String name) | Class.getField(String name) |
| Class.getDeclaredFields() | Class.getFields() |
| getDeclaredMethod((String name,  Class<?>... parameterTypes)) | getMethod((String name,  Class<?>... parameterTypes)) |
| getDeclaredMethods() | getMethods() |

访问私有字段，需要调用Declared方法

method.setAccessible(true);//设置访问权限，如果不设置会抛异常

field.setAccessible(true);





Declared方法不可以获得继承的方法（公有继承也不行）。当然也包括它所实现接口的方法。不加Declared可以获得公有的继承方法和实现接口的方法（包括Object中的方法）

通过反射能够获得方法Method的形参类型，返回值类型，方法名称，修饰符，注解。Field可以获得修饰符，注解，字段的类型，字段的值

通过getClass()方法来获取类的定义信息,通过定义信息再调用getFields()方法来获取类的所有公共属性(只有公有，可以获得父类属性),或者调用getDeclaredFields()方法来获取类的所有属性,包括公共,保护,私有,默认的方法,但是这里有一点要注意的是这个方法只能获取当前类里面显示定义的属性,不能获取到父类或者父类的父类及更高层次的属性的，可以使用Class.getSuperclass()得到父类，再获取其字段

没有多继承

之所以没有多继承，是因为多继承容易引起混乱。比如一个相同的方法两个父类都有，子类到底是继承哪个叻？  
java是通过实现多个接口来弥补不能多继承的缺陷的。接口中的方法都是抽象的，子类可以选择实现。

**Java的注解**

<http://www.importnew.com/10294.html>

http://www.importnew.com/23564.html

JDK1.5引入注解（java.lang.annotation包）用一个词就可以描述注解，那就是元数据，即一种描述数据的数据。所以，可以说注解就是源代码的元数据。注解Annotations仅仅是元数据，**和业务逻辑无关。业务逻辑可以通过反射，获得注解中的数据**

元注解（描述注解的注解）

@Documented一个简单的Annotations标记注解，表示是否将注解信息添加在java文档中。

@Retention– 定义该注解的生命周期。

RetentionPolicy.SOURCE – 在编译阶段丢弃。这些注解在编译结束之后就不再有任何意义，所以它们不会写入字节码。@Override, @SuppressWarnings都属于这类注解。

RetentionPolicy.CLASS – 在类加载的时候丢弃。在字节码文件的处理中有用。注解默认使用这种方式。

RetentionPolicy.RUNTIME– 始终不会丢弃，运行期也保留该注解，因此可以使用反射机制读取该注解的信息。我们自定义的注解通常使用这种方式。

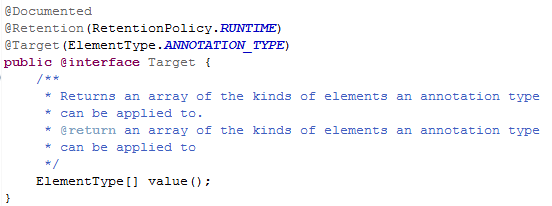
@Target – 表示该注解用于什么地方。如果不明确指出，该注解可以放在任何地方。以下是一些可用的参数。

ElementType.CONSTRUCTOR（构造方法声明）,FIELD（字段声明）,LOCAL VARIABLE（局部变量声明）,METHOD（方法声明）,PACKAGE（包声明）,PARAMETER（参数声明）,TYPE（类接口），ANNOTATION\_TYPE （另一个注释，元注解中用到）

@Inherited – 定义该注释和子类的关系

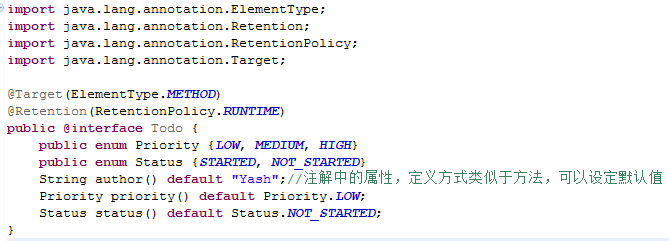
Annotations只支持基本类型、String、枚举、class类型。注释中所有的属性被定义成类似方法的样子，并允许提供默认值。

元注解target的源码

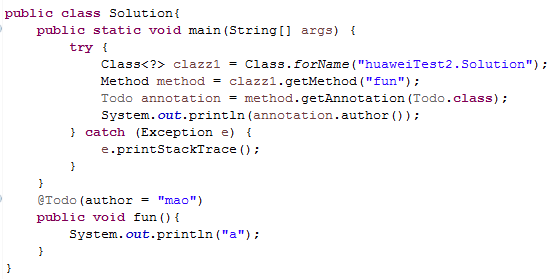


自定义注解实例

定义一个注解与定义一个类相似，可以在eclipse生成（右键包，new，annotation）



这里的@Retention必须为runtime，否则反射时不能用。



method或field都有getAnnotation方法，可以获得method或field上的注解，这里控制台会输出mao。

注解仅仅是元数据（描述数据的数据，只含有数据），**和业务逻辑无关。业务逻辑可以通过反射，获得注解中的数据**

Spring中：

@Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入，而@Resource默认按 byName自动注入罢了。@Resource有两个属性是比较重要的，分是name和type，Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。

**JD****K与JRE**

Jre 是java runtime environment, 是java程序的运行环境。既然是运行，当然要包含jvm，还有所有java类库的class文件（包括java.lang包、java.util包等），都在lib目录下打包成了jar。java命令属于jre

JDK（java development kit）主要包含三部分，

第一部分就是Java运行时环境，JVM。

第二部分就是Java的基础类库，这个类库的数量还是非常可观的。

第三部分就是Java的开发工具，它们都是辅助你更好的使用Java的利器。（javac、其实还有一些比较实用的工具，可以帮助你排查问题。jmap、jconsole、jstack、jvisualvm）

**Spring IoC实现（解耦）**

BeanFacotry是spring中比较原始的Factory。如XMLBeanFactory就是一种典型的BeanFactory。原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。   
  ApplicationContext接口,它由BeanFactory接口派生而来，因而提供BeanFactory所有的功能。ApplicationContext以一种更向面向框架的方式工作以及对上下文进行分层和实现继承

IoC容器BeanFactory（基本）和ApplicationContext（高级），两个都是接口，需要具体实现类。（常用WebApplicationContext）

BeanDefinition是spring IoC容器存储的基本数据结构（xml配置文件等）。是对pojo的抽象。

BeanFactory接口（GenericBeanDefinition实现类）有getBean方法（最主要的方法，依赖注入的入口）；isSingleton；isPrototype；isTypeMatch（是否是指定的class类型）；getType（获取class类型）；getAliases（获取别名）。

回调：A调用B，但B中有A的引用，B中再调用A，可以传入B中处理好的参数

**IoC容器初始化**

在配置文件中配置ContextLoaderListener监听器，并指定spring配置文件位置。



servlet容器启动（tomcat），为应用创建一个“全局上下文环境”：ServletContext ；容器调用web.xml中配置的contextLoaderListener，初始化WebApplicationContext上下文环境（即IoC容器,默认XmlWebApplicationContext），加载context-param指定的配置文件信息到IoC容器中。WebApplicationContext在ServletContext中以键值对的形式保存。此时开始IoC容器的初始化。

IoC容器的初始化由**refresh()**方法启动（这个方法在IoC具体实现类的构造函数中），启动包括BeanDefinition的Resouce定位（获取资源位置的Resource对象）、载入（解析xml配置文件，获取各个Element信息）和注册（将BeanDefinition放入HashMap中）三个基本过程。

**Resource定位**

构造IoC容器时，需要指定BeanDefinition的信息来源（如从xml文件中来），而这个信息来源需要封装成spring中的Resource类。Resource是Spring用来封装I/O操作的类。Resource是一个接口，具体实现类如ClassPathResource。通过配置文件指定的文件位置完成BeanDefinition的定位，但具体的数据还没开始读入。

Resource只是BeanDefinition位置的封装，具体操作由ResourceLoader完成。这个过程即找到BeanDefinition的位置，为其载入创造I/O操作的条件。

1. 构造函数调用refresh()，该方法在synchronized修饰的同步代码块中，保证不能有多个线程初始化。
2. refresh()先调用createBeanFactory()创建容器
3. refresh()再调用loadBeanDefinitions(BeanFactory)方法
4. loadBeanDefinitions()先调用ResourceLoader得到Resource[]数组（定位）
5. loadBeanDefinitions()再调用Reader读取器（如XMLBeanDefinitionReader）读取Resource[]中资源（完成I/O操作，即开始载入），利用回调把结果传入之前建立的BeanFactory中

很多不同的类都有loadBeanDefinitions方法，它们的功能也是不同的。

**BeanDefinition的载入**

把用户定义好的Bean表示成IoC容器内部的数据结构（即BeanDefinition）。BeanDefinition的载入分两部分，首先调用XML解析器得到document对象，但这些document对象没有按照spring的bean规则进行解析。接下来按照spring规则进行解析。

1. 在Reader读取器（如XMLBeanDefinitionReader的loadBeanDefinitions方法）中调用documentLoader（XML解析器）中的方法，获得document对象；
2. 在Reader读取器（如XMLBeanDefinitionReader的loadBeanDefinitions方法）再调用processBeanDefinition（传入document的element）；
3. processBeanDefinition (element,BeanDefinition)方法调用parseBeanDefinitionElement()方法，解析具体的元素标签（如<bean>、<name>、<id>等），将解析得到的结果设置到BeanDefinitionHolder中（BeanDefinitionHolder是BeanDefinition对象的封装类，封装了BeanDefinition、bean的名字等，用它来完成向IoC容器注册）；

经过对document逐层解析，我们在XML中定义的BeanDefinition就被载入到了IoC容器中。但是，重要的依赖注入在这个时候还没有发生，现在IoC容器中BeanDefinition存在的还只是一些静态的配置信息。

**BeanDefinition的注册**

将BeanDefinition放入HashMap中，这些BeanDefinition数据在IoC容器中通过一个HashMap（ConcurrentHashMap<String,BeanDefinition>）来保持和维护,beanName为key，beanDefinition为value。

1. 在第8步processBeanDefinition()方法中调用registerBeanDefinition(BeanDefinitionHolder)开始注册，map.put(beanName,BeanDefinition),registerBeanDefinition中需要synchronized同步代码块，保证数据一致性。以上过程都是在Reader读取器类中完成的。

**IoC容器的依赖注入**

依赖注入的过程是用户第一次向IoC容器索要Bean时触发的（BeanFactory.getBean(name)），当然也有例外，也就是我们可以在BeanDefinition信息中通过控制lazy-init属性来让容器完成对Bean的预实例化（在初始化过程中完成，即在refresh()（初始化的入口）时调用getBean()）。通过依赖注入生成实例。

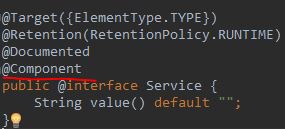
1. getBean()方法中，首先判断是否已经创建，且为单例，这种情况不要创建。然后根据Bean的名字获取BeanDefinition，当前取不到就到双亲BeanFactory中取，如果还取不到就顺着双亲链一直向上。
2. 取到BeanDefinition后，获取当前Bean所依赖的所有Bean，并递归调用getBean。
3. 通过createBean创建Bean的实例，最后返回Bean。
4. createBean()中，调用createBeanInstance()创建Bean（默认使用CGLIB一个常用的字节码生成器类库、或使用JVM的反射）；
5. createBean()中，再调用populateBean()方法，将当前Bean的依赖注入，通过递归调用容器的getBean方法，得到当前Bean的依赖Bean，同时也触发对依赖Bean的创建和注入。

**Bean的其他**

Bean的预实例化：设置了lazy-init属性，在refresh方法中就会调用getBean

Bean对IoC容器的感知：实现ApplicationContextAware接口，可以在Bean中得到Bean所在的应用上下文（ApplicationContext），从而直接在Bean中使用（如getBean）。在IoC对Bean初始化时，会对Bean类型判断，如果是一个ApplicationContextAware的类型，那么IoC会调用其setApplicationContext方法，将ApplicationContext设置进去。

注解：扫描spring容器中指定包，找到包中加注解的类，然后将该类纳入spring容器



service，controller都是component

Spring依赖注入方式

* [1.Set注入](http://www.cnblogs.com/java-class/p/4727775.html#_label0)
* [2.构造器注入](http://www.cnblogs.com/java-class/p/4727775.html#_label1)
* [3.静态工厂的方法注入](http://www.cnblogs.com/java-class/p/4727775.html#_label2)
* [4.实例工厂的方法注入](http://www.cnblogs.com/java-class/p/4727775.html#_label3)
* Spring IOC注入方式用得最多的是(1)(2)种，多写多练就会非常熟练。
* 另外注意：通过Spring创建的对象默认是单例的，如果需要创建多实例对象可以在<bean>标签后面添加一个属性：
* <bean name="..." class="..." scope="prototype">

通过IoC容器实现的依赖反转，把依赖关系的管理从Java对象中解放出来，交给了IoC容器来完成，从而完成了对象之间的关系解耦，利用cglib或反射实现

**java.lang.reflect.Proxy**

代理分为静态代理和动态代理，静态代理是在编译时就将接口、实现类、代理类一股脑儿全部手动完成，但如果我们需要很多的代理，每一个都这么手动的去创建实属浪费时间，而且会有大量的重复代码，此时我们就可以采用动态代理，动态代理可以在程序运行期间根据需要动态的创建代理类及其实例，来完成具体的功能。

Proxy.newProxyInstance()方法利用参数中的classLoader生成字节码文件并动态生成代理类$ProxyN，这个类继承了Proxy并且实现了传入的一组接口，循环遍历传入接口中的每一个方法，将方法设置到$ProxyN中，父类Proxy中有InvocationHandler h属性，利用反射获得传入接口中的方法method，将method作为参数传入h中，调用h的invoke方法利用反射完成代理。

Proxy 类是它的父类，这个规则适用于所有由 Proxy 创建的动态代理类。(也算是java动态代理的一处缺陷，java不支持多继承，所以无法实现对class的动态代理，只能对于Interface的代理)而且该类还实现了其所代理的一组接口，这就是为什么它能够被安全地类型转换到其所代理的某接口的根本原因。

传入一组接口的原因，1、面向接口编程，一种规范；2、java不支持多继承，如果一个类实现了一组接口，对其进行代理时，不同接口的不同方法都会得到增强。

一个典型的动态代理创建对象过程可分为以下四个步骤：

1、通过实现InvocationHandler接口创建自己的调用处理器 IvocationHandler handler = new InvocationHandlerImpl(...);

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

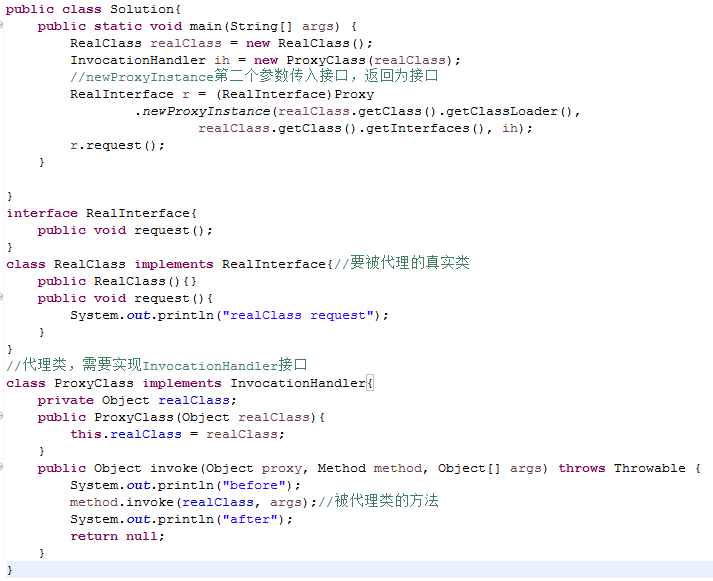
Class clazz = Proxy.getProxyClass(classLoader,new Class[]{...});

3、通过反射机制获取动态代理类的构造函数，其参数类型是调用处理器接口类型

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

4、通过构造函数创建代理类实例，此时需将调用处理器对象作为参数被传入

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

为了简化对象创建过程，Proxy类中的newInstance方法封装了2~4，只需两步即可完成代理对象的创建。

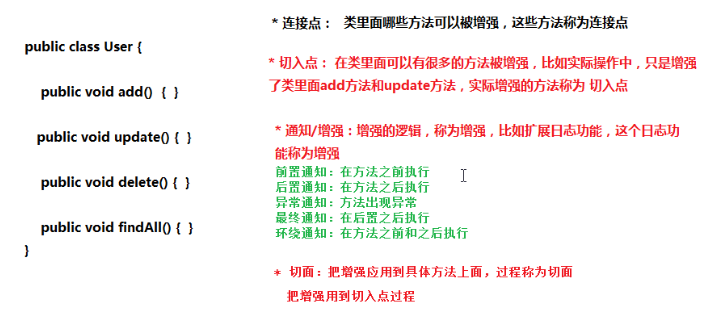
**Spring AOP实现**

AOP（Aspect Oriented Programming），即面向切面编程，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

将重复的代码抽取出来（不同的模块），在需要时统一调用，可以不同的模块按不同的顺序调用，解耦。

AOP适合于那些具有横切逻辑的应用：如性能监测，访问控制，事务管理、缓存、对象池管理以及日志记录。



advice：切面增强设计；pointcut：切入点

advisor（通知器）：把advice和pointcut结合起来，可以定义应该使用哪个advice并在哪个pointcut使用。advisor中有advice和pointcut两个属性，pointcut使用单例模式（饿汉，static final）

Spring AOP实现中，使用的核心技术是动态代理，而这种动态代理实际上是JDK的一个特性（Proxy模式，在Proxy的调用过程中，如果客户调用Proxy的request方法，先调用Proxy的前处理方法，再调用目标对象的request方法，再调用Proxy的后处理方法（通过反射、拦截器链等实现））

实现：

ProxyFactoryBean是AOP中最重要的一个类，在xml文件中配置，并放入IoC中（<bean>），对于ProxyFactoryBean把需要对target目标对象增加的增强处理都通过getObject()方法封装了（工厂模式）。以getObject方法作为入口，spring AOP调用AopProxyFactory作为AopProxy代理对象的生产工厂，由它来负责产生相应的AopProxy代理对象（默认使用JDK的Proxy，或CGLIB）。

在AopProxy代理的接口方法被调用执行时（调用原对象的方法即调用代理对象的方法），首先会触发对这些方法调用进行拦截，这些拦截对目标调用的功能增强提供了工作空间，拦截过程在JDK的proxy代理对象中是通过invoke方法来完成的。

在Spring AOP通过JDK的Proxy方法生成代理对象时，相关的拦截器（一个通知器链）已经配置到代理对象中了。

在InvocationHandler的invoke回调中，首先会根据配置来对拦截器是否与当前调用方法相匹配进行判断，如果匹配则相应的拦截器开始发挥作用。这个过程是一个遍历的过程，它会遍历AopProxy代理对象中设置的拦截器链中的所有拦截器。拦截器逐一被调用，在拦截器都调用完后，才对目标对象方法调用。

通知器advisor在xml中配置，放在IoC容器中，拦截器、通知器等一些信息是存放在AdvisorSupport类中，通过getBean获取。在ProxyFactoryBean中获取IoC容器是实现了BeanFactoryAware接口。

为了完成AOP应用需要的对目标对象增强，对于每种advice通知，spring设计了对应的AdviceAdapter通知适配器，这些通知适配器实现了advice通知对目标对象的不同增强方式（before、after等）。对于这些适配器，在AopProxy回调方法中有一个注册机制。完成注册后，拦截器链中运行的拦截器已经是经过适配的拦截器（实际上是通知器）了。

如果是适配器支持的adapter.supportsAdvice(advice)（advice根据配置文件有before、after等类型），就将生成的包装后的拦截器，放入拦截器链，如MethodBeforeAdviceAdapter，会产生MethodBeforeAdviceInterceptor，这个包装后的拦截器会先调用advice.before方法，在继续遍历拦截器链（methodInvocation.proceed()）。如果是after类型，就先继续遍历拦截器链（methodInvocation.proceed()），再调用after方法。Throws是通过catch中捕获异常实现的。相当于拦截器链中有很多adapter包装后的拦截器，依次遍历每个拦截器，调用其方法，从而实现了增强

电源适配器，它是用于电流变换（整流）的设备。适配器的存在，就是为了将已存在的东西（接口）转换成适合我们的需要、能被我们所利用。

**Spring MVC实现**

在启动过程中，spring会使用一个默认的WebApplicationContext实现作为IoC容器。这个默认使用的IoC容器就是XMLWebApplicationContext。对于spring承载的web应用而言，可以指定在web应用程序启动时载入IoC容器（或者称为WebApplicationContext）。这个功能是由ContextLoaderListener这样的类来完成的，它是在web容器中配置的监听器（配置在web.xml中）。这个ContextLoader就像spring应用程序在web容器的启动器。该IoC容器会被存储到SevletContext中。

在完成对ContextLoaderListener初始化后，web容器（tomcat）开始初始化DispatcherServlet，DispatcherServlet会建立自己的上下文来持有Spring MVC的Bean对象，在建立这个自己持有的IoC容器时，会从ServletContext中得到根上下文（WebApplicationContext）作为DispatcherServlet持有上下文的双亲上下文。有了这个根上下文，再对自己持有的上下文进行初始化，最后把自己持有的这个上下文保存到ServletContext中，供以后检索和使用。

作为servlet，DispatcherServlet的启动与servlet启动过程是相联系的，servlet的init方法会被调用，以进行初始化，接着会初始化DispatcherServlet持有的IoC容器。

Spring MVC的实现大致由以下几个步骤完成：

1、根据controller和HTTP请求之间的映射关系，将url和handle（controller）作为键值对放到HandlerMapping中的handlerMap（HashMap）中。

2、DispatcherServlet调用doDispatch方法，分发请求。

DispatcherServlet持有IoC容器，里面装有Controller、HandlerMapping、HandlerAdapter、ViewResolver等这些特殊的**bean**

HandlerAdapter：HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；如SimpleControllerHandlerAdapter将对实现了Controller接口的Bean进行适配，并且掉处理器的handleRequest方法进行功能处理；

ViewResolver：ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；如InternalResourceViewResolver将逻辑视图名映射为jsp视图；

**Spring MVC的运行流程**

1. 用户向服务器发送请求，请求被Spring 前端控制器DispatcherServlet捕获；

2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。然后根据该URI，调用处理器映射器HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回；

3. DispatcherServlet 根据获得的Handler，选择一个合适的处理器适配器HandlerAdapter。（附注：如果成功获得HandlerAdapter后，此时将开始执行拦截器的preHandler(...)方法）

4.提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。 在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

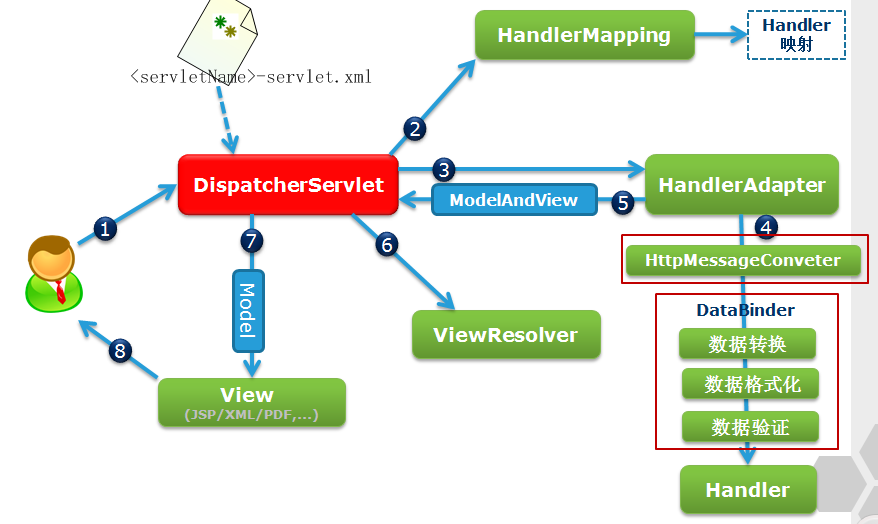
数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

5. Handler执行完成后，向HandlerAdapter返回ModelAndView，HandlerAdapter向DispatcherServlet 返回一个ModelAndView对象；

6. 根据返回的ModelAndView，选择一个适合的视图解析器ViewResolver（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet ；

7. ViewResolver 结合Model和View，来渲染视图

8. 将渲染结果返回给客户端。



HandlerMapping这个组件，它负责的是定位请求处理器Handler。 在HandlerMapping返回处理请求的Controller实例后，需要一个帮助定位具体请求**方法**的处理类，这个类就是HandlerAdapter，HandlerAdapter是处理器适配器，Spring MVC通过HandlerAdapter来实际调用处理函数。

**Controller的线程安全问题**

action是多实例的，所以不存在线程安全的问题，而Controller默认是单例的（效率高）有可能存在线程安全问题，但Controller是基于方法的，所以只要不使用类成员变量，方法中都是局部变量，所以不会出现线程安全问题，如果使用成员变量，可以使用Threadlocal

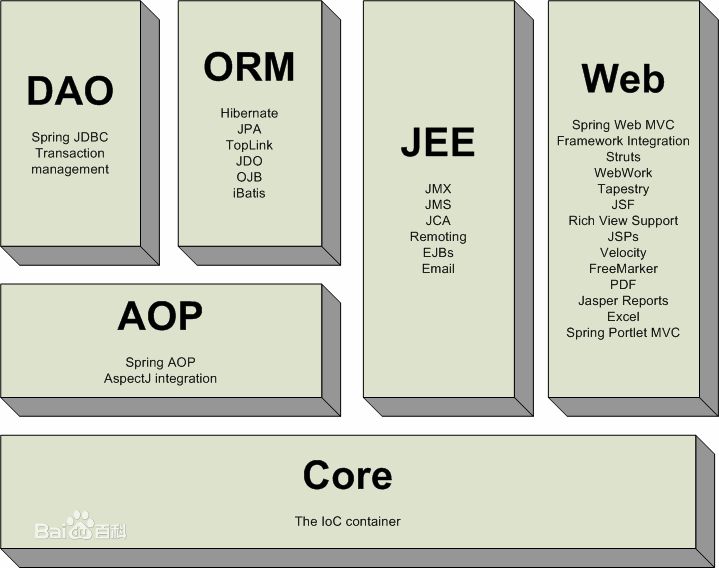
**Spring 事务实现**

声明式事务处理完全可以看出是一个具体的Spring AOP应用。

声明式事务处理的大致过程：在为事务处理配置好AOP的基础设施（比如，对应的Proxy代理对象和事务处理Interceptor拦截器对象）之后，首先需要完成对这些事务属性配置的读取，这些属性的读取处理是在TransactionInterceptor中实现的；

TransactionInterceptor是使用AOP实现声明式事务处理的拦截器，封装了Spring对声明式事务处理实现的基本过程；TransactionAttributeSource和TransactionAttribute这两个类，他们封装了对声明式事务处理属性的识别，以及信息读入和配置。

TransactionInfo和TransactionStatus这两个对象，它们存放事务处理信息的主要数据对象，它们通过与线程的绑定（ThreadLocal）来实现事务的隔离性。TransactionInfo持有TransactionStatus，TransactionStatus掌管事务执行的详细信息，包括具体的事务对象、事务执行状态、事务设置状态。在事务的创建、启动、提交和回滚过程中，都与TransactionStatus对象中的数据打交道。具体的事务处理是由事务处理器TransactionManager来完成。



**Spring 中用到的设计模式**

1. 工厂模式：BeanFactory的getBean方法
2. 单例模式：Spring中的Bean默认都是单例模式的，但没有通过构造器级别去实现，而是通过BeanFactory管理（重复的bean不会创建）
3. 适配器模式：AOP中MethodBeforeAdviceAdapter、MethodAfterAdviceAdapter，将适配器支持的通知器包装成MethodBeforeAdviceAdapter等类型，放入拦截器链中。MVC中handleAdapter
4. 责任链模式：HandlerExecutionChain,AOP中InvocationHandler的invoke遍历拦截器
5. 装饰器模式：BeanWrapper，包装了bean，提供bean更强大的操作功能。BeanWrapperImpl类是对BeanWrapper接口的默认实现，它包装了一个bean对象，缓存了bean的内省结果，并可以访问bean的属性、设置bean的属性值。
6. 代理模式：Proxy在AOP中用到
7. 观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如监听器ApplicationListener，事件ApplicationEvent。ApplicationContextAware使用观察者模式
8. 模板方法：如jdbcTemplate。Template Method模式一般是需要继承的。这里想要探讨另一种对Template Method的理解。spring中的JdbcTemplate，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到JdbcTemplate已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入JdbcTemplate的方法中。
9. 策略模式：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。（原来可能使用多个if。。else判断）本模式使得算法可独立于使用它的客户而变化。 spring中在实例化对象的时候用到Strategy模式。在SimpleInstantiationStrategy中。类图中提供了一个 Resouce 接口，这个接口就是 Spring 为资源访问所提供的策略接口，该接口下的大量实现类：UrlResource、ClassPathResource、 FileSystemResource都实现了该策略接口，用于实现不同的资源访问策略(classpath是项目编译后src目录。 每次编译完在项目的web-inf 目录下有个classes的文件夹//url访问网络资源)。Arrays.sort排序，传入不同的比较器也是策略模式

装饰器模式：能动态的新增或组合对象的行为。

代理模式：为其他对象提供一种代理以控制对这个对象的访问.

适配器模式：是对其他对象接口的一种转换行为，将原接口转换为目标接口，达到适配的效果。

外观模式：外观对象提供对子系统各元件功能的简化为共同层次的调用接口，它主要起到"简化作用"。

装饰模式是“新增行为”，代理模式是“控制访问行为”，适配器模式是"转换行为"，外观模式是一种"简化行为"。

模板模式：

　　定义一个操作中的算法的骨架，而将步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义算法的某些特定步骤。

1抽象类（AbstractClass）：实现了模板方法，定义了算法的骨架。

2具体类（ConcreteClass)：实现抽象类中的抽象方法，已完成完整的算法。

优点：

1模板方法模式通过把不变的行为搬移到超类，去除了子类中的重复代码。

2子类实现算法的某些细节，有助于算法的扩展。

3通过一个父类调用子类实现的操作，通过子类扩展增加新的行为，符合“开放-封闭原则”。

**Spring MVC与Struts**

1. 机制：spring mvc的入口是servlet，而struts2是filter（这里要指出，filter和servlet是不同的。以前认为filter是servlet的一种特殊），这样就导致了二者的机制不同，这里就牵涉到servlet和filter的区别了。

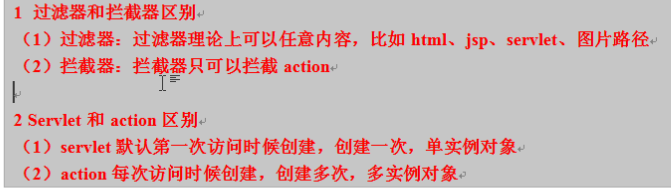
2. 性能：spring会稍微比struts快。spring mvc是基于方法的设计，而sturts是基于类，每次发一次请求都会实例一个action，每个action都会被注入属性.

而spring基于方法，粒度更细，但要小心把握像在servlet控制数据一样。spring mvc是方法级别的拦截，拦截到方法后根据参数上的注解，把request数据注入进去，在spring mvc中，一个方法对应一个request上下文。

而struts2框架是类级别的拦截，每次来了请求就创建一个Action，然后调用setter getter方法把request中的数据注入；struts2实际上是通过setter getter方法与request打交道的；struts2中，一个Action对象对应一个request上下文。

3、springmvc可以进行单例开发，并且建议使用单例开发，struts2通过类的成员变量接收参数，无法使用单例，只能使用多例。

4、经过实际测试，struts2速度慢，在于使用struts标签，如果使用struts建议使用jstl。



**MyBat****is与Hibernate**

hibernate：是一个标准ORM框架（对象关系映射）。入门门槛较高的，不需要程序写sql，sql语句自动生成了。

对sql语句进行优化、修改比较困难的。

应用场景：适用于需求变化不多的中小型项目，比如：后台管理系统，erp、orm、oa。。

mybatis：专注是sql本身，需要程序员自己编写sql语句，sql修改、优化比较方便。mybatis是一个不完全 的ORM框架，虽然程序员自己写sql，mybatis 也可以实现映射（输入映射、输出映射）。

应用场景：适用与需求变化较多的项目，比如：互联网项目。

Mybatis #与$区别

1 #是将传入的值当做字符串的形式，eg:select id,name,age from student where id =#{id},当前端把id值1，传入到后台的时候，就相当于 select id,name,age from student where id ='1'.

2 $是将传入的数据直接显示生成sql语句，eg:select id,name,age from student where id =${id},当前端把id值1，传入到后台的时候，就相当于 select id,name,age from student where id = 1.

3 使用#可以很大程度上防止sql注入。(语句的拼接)

4 但是如果使用在order by 中就需要使用 $.或传入表名，因为表名不能带引号（可以带`tablename`，小撇）

5 在大多数情况下还是经常使用#，但在不同情况下必须使用$.

我觉得#与的区别最大在于：#{} 传入值时，sql解析时，参数是带引号的，而的区别最大在于：#{} 传入值时，sql解析时，参数是带**引号**的，而{}穿入值，sql解析时，参数是不带引号的。

#{ }：解析为一个 JDBC 预编译语句（prepared statement）的参数标记符。预编译，重复利用，效率高

**AJAX**

ajax发送异步请求（四步操作）

1. 第一步（得到XMLHttpRequest）

\* ajax其实只需要学习一个对象：XMLHttpRequest，如果掌握了它，就掌握了ajax！！！

\* 得到XMLHttpRequest

> 大多数浏览器都支持：var xmlHttp = new XMLHttpRequest();

> IE6.0：var xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");

> IE5.5以更早版本的IE：var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");

\* 编写创建XMLHttpRequest对象的函数

function createXMLHttpRequest() {

try {

return new XMLHttpRequest();

} catch(e) {

try {

return new ActiveXObject("Msxml2.XMLHTTP");

} catch(e) {

try {

return new ActiveXObject("Microsoft.XMLHTTP");

} catch(e) {

alert("哥们儿，你用的是什么浏览器啊？");

throw e;

}

}

}

}

2. 第二步（打开与服务器的连接）

\* xmlHttp.open()：用来打开与服务器的连接，它需要三个参数：

> 请求方式：可以是GET或POST

> 请求的URL：指定服务器端资源，例如；/day23\_1/AServlet

> 请求是否为异步：如果为true表示发送异步请求，否则同步请求！

\* xmlHttp.open("GET", "/day23\_1/AServlet", true);

3. 第三步（发送请求）

\* xmlHttp.send(null)：如果不给可能会造成部份浏览器无法发送！

> 参数：就是请求体内容！如果是GET请求，必须给出null。

4. 第四步（）

\* 在xmlHttp对象的一个事件上注册监听器：onreadystatechange

\* xmlHttp对象一共有5个状态：

> 0状态：刚创建，还没有调用open()方法;

> 1状态：请求开始：调用了open()方法，但还没有调用send()方法

> 2状态：调用完了send()方法了；

> 3状态：服务器已经开始响应，但不表示响应结束了！

> 4状态：服务器响应结束！（通常我们只关心这个状态！！！）

\* 得到xmlHttp对象的状态：

> var state = xmlHttp.readyState;//可能是0、1、2、3、4

\* 得到服务器响应的状态码

> var status = xmlHttp.status;//例如为200、404、500

\* 得到服务器响应的内容1

> var content = xmlHttp.responseText;//得到服务器的响应的文本格式的内容

> var content = xmlHttp.responseXML;//得到服务器的响应的xml响应的内容，它是Document对象了！

xmlHttp.onreadystatechange = function() {//xmlHttp的5种状态都会调用本方法

if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {//双重判断：判断是否为4状态，而且还要判断是否为200

// 获取服务器的响应内容

var text = xmlHttp.responseText;

}

};

Ajax 的全称是Asynchronous JavaScript and XML

Ajax的工作原理相当于在用户和服务器之间加了一个中间层(AJAX引擎),使用户操作与服务器响应异步化。并不是所有的用户请求都提交给服务器,像一些数据验证和数据处理等都交给Ajax引擎自己来做, 只有确定需要从服务器读取新数据时再由Ajax引擎代为向服务器提交请求。

Ajax其核心只有JavaScript、XMLHTTPRequest和DOM，在旧的交互方式中,由用户触发一个HTTP请求到服务器,服务器对其进行处理后再返回一个新的HTML页到客户端, 每当服务器处理客户端提交的请求时,客户都只能空闲等待,并且哪怕只是一次很小的交互、只需从服务器端得到很简单的一个数据,都要返回一个完整的HTML页,而用户每次都要浪费时间和带宽去重新读取整个页面。

Ajax的一个最大的特点是无需刷新页面便可向服务器传输或读写数据(又称无刷新更新页面),这一特点主要得益于XMLHTTP组件XMLHTTPRequest对象。

XMLHttpRequest是ajax的核心机制，它是在IE5中首先引入的，是一种支持异步请求的技术。简单的说，也就是javascript可以及时向服务器提出请求和处理响应，而不阻塞用户。达到无刷新的效果

我们可以把服务器端看成一个数据接口，它返回的是一个纯文本流，当然，这个文本流可以是XML格式，可以是Html，可以是Javascript代码，也可以只是一个字符串。这时候，XMLHttpRequest向服务器端请求这个页面，服务器端将文本的结果写入页面，这和普通的web开发流程是一样的，**不同的是，客户端在异步获取这个结果后，不是直接显示在页面，而是先由javascript来处理，然后再显示在页面。**

ajax干掉了back按钮，即对浏览器后退机制的破坏。后退按钮是一个标准的web站点的重要功能，但是它没法和js进行很好的合作。这是ajax所带来的一个比较严重的问题，因为用户往往是希望能够通过后退来取消前一次操作的。

同步传输的比特分组要大得多。它不是独立地发送每个字符，每个字符都有自己的开始位和停止位，而是把它们组合起来一起发送。我们将这些组合称为数据帧，或简称为帧。

　　数据帧的第一部分包含一组同步字符，它是一个独特的比特组合，类似于前面提到的起始位，用于通知接收方一个帧已经到达，但它同时还能确保接收方的采样速度和比特的到达速度保持一致，使收发双方进入同步。

　　帧的最后一部分是一个帧结束标记。与同步字符一样，它也是一个独特的比特串，类似于前面提到的停止位，用于表示在下一帧开始之前没有别的即将到达的数据了。

**Java并发包**

创建线程的方法：

继承thread类（缺点：如果一个类已经有父类）、实现runnable接口（可以解耦）、使用线程池

**线程池**

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。第四：Java1.5中引入的Executor框架把任务的提交和执行进行解耦，只需要定义好任务，然后提交给线程池，而不用关心该任务是如何执行、被哪个线程执行，以及什么时候执行。

Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService，Executors是一个框架，ThreadPoolExecutor是线程池实现的核心类。

可以使用Executors框架中的静态工厂方法之一来创建一个线程池：（如果线程池中抛出异常，内部会被try，catch掉。可以通过future.get()捕获异常）

**newFixedThreadPool**：创建一个固定长度的线程池，每当提交一个任务时就创建一个线程，直到达到线程池的最大数量，这时线程池的规模将不再变化（如果某个线程由于发生了未预期Exception而结束，那么线程池会补充一个新的线程）。

**newCachedThreadPool**：创建一个可缓存的线程池，如果线程池当前规模超过了处理需求，那么将回收空间线程，而当需求增加时，可以添加新线程，线程池规模不存在任何限制（不推荐使用，线程池规模不可控）大小为Integer.MAX\_VALUE，2的31次方-1。

**newSingleThreadExecutor**：是一个单线程的Executor，如果这个线程异常结束，会创建另一个线程替代，它能保证依照任务在队列中的顺序串行执行(FIFO, LIFO, 优先级)。

**newScheduledThreadPool**：创建一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

ExecutorService threadPool = Executors.newFixedThreadPool(3);

ExecutorService threadPool = Executors.newCachedThreadPool();

ExecutorService threadPool = Executors.newSingleThreadExecutor();

threadPool.execute(Runnable r);

threadPool.shutdown();将任务执行完毕后关闭

threadPool.shutdownNow();立即关闭，并返回没有执行的任务（List<Runnable>）

对于计算密集型的任务，在拥有N个cpu的系统上，当线程池的大小为N+1时，通常能实现最优利用率。对于I/O密集型，通常线程池大小为2\*N。

**java.util.concurrent.ThreadPoolExecutor**

ThreadPoolExecutor类是线程池中最核心的一个类，Executors中的newFixedThreadPool等方法就是由ThreadPoolExecutor实现。不过在java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是使用Executors类中提供的几个静态方法来创建线程池。ThreadPoolExecutor继承AbstractExecutorService类，即ThreadPoolExecutor中能够调用ExecutorService的方法

ThreadPoolExecutor中主要方法，execute(Runnable command)、submit(Callable<T> task)、shutdown()、shutdownNow()；submit底层还是调用execute

ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 200, TimeUnit.MILLISECONDS,new ArrayBlockingQueue<Runnable>(5));

构造函数中的参数：

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit, BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);

1、corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

2、maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

3、keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

4、unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.DAYS; //天 TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟 TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒 TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

5、workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

1）ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

2）LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

3）synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

一般使用LinkedBlockingQueue和Synchronous。

6、threadFactory：线程工厂，主要用来创建线程；

7、handler：表示当拒绝处理任务时的策略（大于maximumPoolSize），有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

* 如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；
* 如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；
* 如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；
* 如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

实现原理：ThreadPoolExecutor中有一个实现了Runnable接口的Worker内部类，可以new Worker类，构造函数中传入任务，在Worker类中创建新线程，Worker重写Runnable的run方法，while (task != null || (task = getTask()) != null)，getTask方法从workQueue中取出任务。即循环不断取出任务，while循环中run任务时要lock。当不满足while循环中的条件即没有任务，getTask调用queue的take方法，会阻塞。在这个while循环中，会调用task（即传递进来的runnable）的run方法，而不是start开启新线程，因为while循环在worker类的run方法中，已经是一个线程了。

在addWorker方法中new的Worker类。

在execute()方法中，当小于corePoolSize时，调用addWorker方法并传入任务。

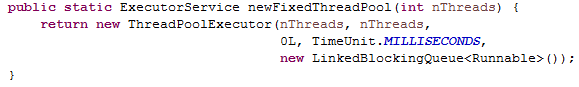
Executors框架：

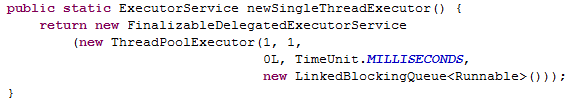
newFixedThreadPool创建的线程池corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue；

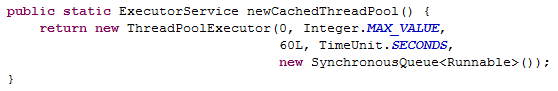
newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；

newCachedThreadPool将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。

ScheduledThreadPoolExecutor继承了ThreadPoolExecutor，传入队列是DelayedWorkQueue()。







对于任务耗时较短的情况下，线程数不宜过多。 对于任务耗时长的情况，分为I/O密集和计算密集。 I/O密集的时候，相对来说CPU空闲的时间比较多，那可以适当的增加线程数，增加CPU的利用率。 计算密集的时候，CPU一直被占用进行计算，线程数不宜太多。

**java.util.concurrent.atomic包**

AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference以原子方式进行操作

AtomicInteger构造函数无参-初始化为0；有参-传入初始值；方法addAndGet(int num)、incrementAndGet()、getAndIncrement()、decrementAndGet()、getAndDecrement()、compareAndSet(int expect, int update) 如果当前值 == 预期值，则以原子方式将该值设置为给定的更新值。线程安全的

相比于以前使用synchronized(obj){增加}，提升了性能，因为AtomicInteger内部通过JNI（Java Native Interface）的方式使用了硬件支持的CAS指令（compare and swap）

**java.util.concurrent.CountDownLatch**

CountDownLatch主要提供的机制是当多个（具体数量为初始化传入的个数）线程都达到了预期状态或完成预期工作时触发事件，其他线程可以等待这个事件来触发自己的后续工作（等待的线程可以是多个）。到达自己预期状态的线程会调用CountDownLatch的countDown方法，而等待线程会调用await方法。

比如CountDownLatch latch = new CountDownLatch(10)，开10个线程计算数据，run方法中，计算完成后调用latch.countDown();比如还有另外两个线程，执行到某个地方，调用latch.await()，只有当10个线程都countDown，才能唤醒await。注意：10个线程执行countDown()后不会停顿，还会继续执行自己的任务。

**java.util.concurrent.Cyclic****Barrier**

CyclicBarrier可以协同多个线程，让多个线程在这个屏障前等待，直到所有线程都到达了这个屏障时，再一起继续执行后面的动作。

比如CyclicBarrier barrier = new CyclicBarrier(3);3个线程run方法中均有barrier.await()方法，任何一个线程到了await()会进入阻塞等待状态，直到3个线程都到了await才会同时从await返回，继续后续的工作。如果在构造CyclicBarrier时设置了一个Runnable实现（构造函数第二个参数），那么最后一个到await的线程会执行这个runnable的run方法。

CyclicBarrier和CountDownLatch都是用于多个线程间的协调。区别：1、CyclicBarrier线程到了await()都会阻塞，然后一起返回；CountDownLatch线程到了countDown()后不会停顿，还会继续执行自己的任务，到await()的会阻塞。2、CountDownLatch不能重复使用，用完了还需要重新初始化一个，CyclicBarrier可以重复使用。

**java.util.concurrent.Semaphore**

Semaphore是用于管理信号量的，构造时传入可供管理的信号量数值。主要用于控制并发数，如果Semaphore管理的信号量只有1个，就退化到互斥锁了。与通过控制线程数来控制并发数的方法相比，通过Semaphore来控制并发数可以控制得更加细粒度，因为真正需要有并发数限制的代码只需要放到acquire和release之间即可（其他仍可以由更多线程执行）。

semaphore.acquire();//acquire、release可以有参数，获取/返还的信号量个数

try{//比如调用远程通信方法}

finally{semaphore.release();}

acquire一个，信号量的个数就减少一个，如果没了acquire就会阻塞，等release之后才会返回。

**java.util.concurrent.Exchanger**

Exchanger用于在两个线程之间进行数据交换，线程会阻塞在Exchanger的exchange方法上，直到另外一个线程也到了同一个Exchanger的exchange方法时，二者进行交换，然后两个线程会继续执行后续代码。

Exchanger定义时有泛型，泛型即为需要交换的数据类型。exchange()方法，括号中传入需要交换的数据（符合之前定义的泛型）。

**java.util.concurrent.Future/FutureTask**

Future是一个接口，FutureTask是一个具体实现类。有一个方法从远程获取一些计算结果，需要很长时间，可以用异步获取，等到需要使用该数据的时候，在get出来（或使用回调的方式，计算完成后再回调）。

Future<HashMap> future = getDataFromRemote2();

//do something

HashMap data = (HashMap) future.get();

private Future<HashMap> getDataFromRemote2(){

return threadPool.submit(new Callable<HashMap>(){

public HashMap call() throws Exception{

return getDataFromRemote();

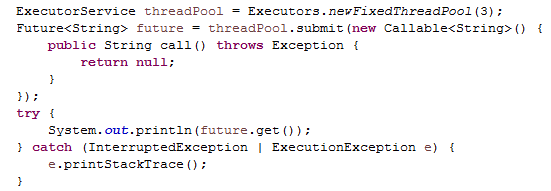
}

});

}

FutureTask是一个具体实现类，在前面例子中，ThreadPoolExecutor的submit方法返回的是一个Future的实现，这个实现就是FutureTask的一个具体实例。

Future的cancel方法（ExecutorService、Executors、ThreadPoolExecutor中没有），试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。



**java.util.concurrent.****ArrayBlockingQueue<E>/LinkedBlockingQueue**

并发库提供的阻塞队列

add();如果满了就抛异常

offer();如果满了返回false

put();如果满了就阻塞

Remove();如果没有了就抛异常

Poll();如果没有返回null

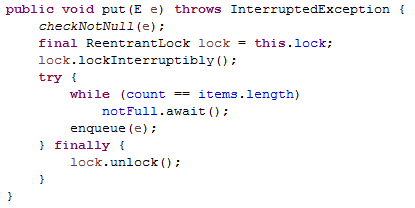
Take();如果没有就阻塞，等放进去了再取；

peek()；查看头，但不移除，如果没有就返回null

put和take方法需要try，catch

两者都是线程安全的，添加、删除方法中都有reentrantlock

实现阻塞的方法，就是生产者消费者模型原始的方法，满了之后await等待signal，等待唤醒；



**线程中断**

Thread的interrupt方法是中断线程。其实，Java的中断是一种协作机制。也就是说调用线程对象的interrupt方法并不一定就中断了正在运行的线程，它只是要求线程自己在合适的时机中断自己。每个线程都有一个boolean的中断状态（中断位），interrupt方法仅仅只是将该状态置为true。

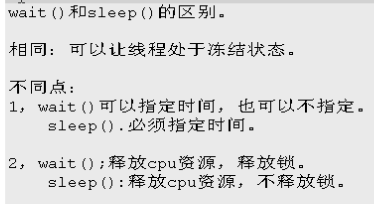
（1）可以通过Thread的类方法interrupted检查当前线程是否中断，但是会重置标志位，比如由true置为false（true表示中断）

（2）通过调用Thread的普通方法isInterrupted来检查中断位，不会重置标志位。

**中断线程方法：**（1）通过检查标志位用break退出循环（2）通过return退出run方法（3）通过对有些状态中断抛异常退出。（4）当循环判断的条件不满足时，run方法执行完毕线程自然会退出；

Thread.yield( )方法：使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。cpu会从众多的可执行态里选择，也就是说，当前也就是刚刚的那个线程还是有可能会被再次执行到的，并不是说一定会执行其他线程而该线程在下一次中不会执行到了。

thread.join（）调用该方法的线程等待被调用的线程执行完毕后才执行

wait是object的方法，sleep是thread的**静态**方法,在线程wait或sleep时，使用thread的interrupt方法（不是静态方法）都会响应中断。

此时，**执行interrupt()时,并不需要获取Thread实例的锁定。**任何线程在任何时刻,都可以调用其他线程的interrupt()方法.**当正在sleeping的线程被调用interrupt()方法**时,就会放弃暂停的状态，并**抛出InterruptedException异常**，这个异常是A线程抛出的。   
wait() & interrupt()   
    线程A调用了wait()进入了等待状态,也可以用interrupt()取消。  
    不过这时候要**小心锁定的问题**。线程进入等待区,会把锁定解除,当对等待中的线程调用interrupt()时(注意是等待的线程调用其自己的interrupt()),会先重新获取锁定,再抛出异常。在获取锁定之前,是无法抛出异常的.

**Runnable和Callable的区别**

(1)Callable规定的方法是call(),Runnable规定的方法是run().

(2)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得，call方法有返回值，run方法没有。返回值通过Future.get()方法得到

(3)call方法可以抛出异常，run方法不可以

(4)运行Callable任务可以拿到一个Future对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

**CopyOnWriteArrayList（线程安全的）**

CopyOnWriteArrayList使用了一种叫写时复制的方法，当有新元素添加到CopyOnWriteArrayList时，先从原有的数组中拷贝一份出来，然后在新的数组做写操作，写完之后，再将原来的数组引用指向到新数组。

CopyOnWriteArrayList的整个add操作都是在锁（ReentrantLock）的保护下进行的。 remove也一样（加锁，复制）

这样做是为了避免在多线程并发add的时候，复制出多个副本出来,把数据搞乱了，导致最终的数组数据不是我们期望的。

CopyOnWriteArrayList 合适读多写少的场景，比如说缓存

同步代码块synchronized（obj）{}

或同步函数public synchronized void add（int n）{}（使用this锁）

静态同步函数public static synchronized void add（int n）{}（静态没创建对象没有this，类加载后，堆内存中会创建 类.class，此时锁即为类.class）

**CGLIB动态代理**

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

**synchronized与ReentrantLock**

**synchronized和Lock区别**

Lock能够实现synchronized的所有功能，出现异常时synchronized会由JVM自动释放，而Lock必须手动释放，因此我们需要把unLock()方法放在finally{}语句块中。

**可中断锁**

lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。（终止线程？）

　　由于lockInterruptibly()的声明中抛出了异常，所以lock.lockInterruptibly()必须放在try块中或者在调用lockInterruptibly()的方法外声明抛出InterruptedException。

　　因此lockInterruptibly()一般的使用形式如下：

public void method() throws InterruptedException {

lock.lockInterruptibly();//放在try外面，可以向外抛出异常

try {

//.....

}

finally {

lock.unlock();

}

}

　　注意，当一个线程获取了锁之后，是不会被interrupt()方法中断的。因为本身在前面的文章中讲过单独调用interrupt()方法不能中断正在运行过程中的线程，只能中断阻塞过程中的线程。

　　因此当通过lockInterruptibly()方法获取某个锁时，如果不能获取到，只有进行等待的情况下，是可以响应中断的。

　　而用synchronized修饰的话，当一个线程处于等待某个锁的状态，是无法被中断的，只有一直等待下去。

**公平锁**

公平锁以请求锁的顺序来获取锁，非公平锁则是无法保证按照请求的顺序执行。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。

构造函数参数为true时表示公平锁，不传或者false都是为非公平锁。

**非阻塞加锁**

　tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

　　tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

Lock lock = ...;

if(lock.tryLock()) {

try{

//处理任务

}catch(Exception ex){

}finally{

lock.unlock(); //释放锁，如果if判断失败，这里的finally不会执行

}

}else {

//如果不能获取锁，则直接做其他事情

}

**可重入锁**

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（或递归调用），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。

**读写锁**

ReentrantReadWriteLock

readLock()获取读锁，通过writeLock()获取写锁

ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

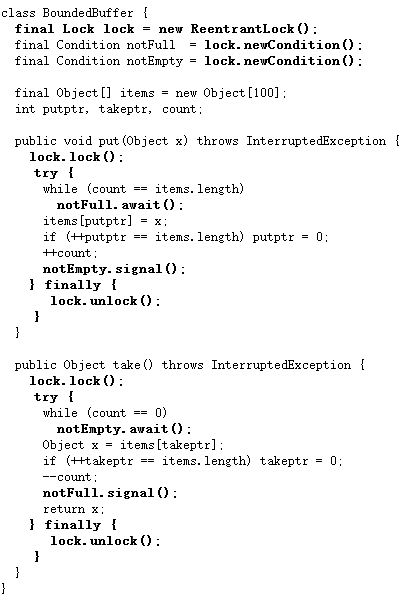
rwl.readLock().lock();rwl.readLock().unlock();

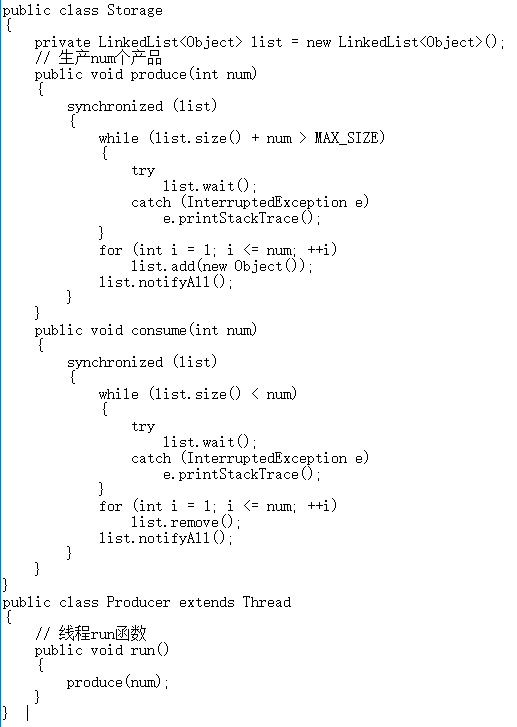
rwl.writeLock().lock();rwl.writeLock().unlock();

synchronized不能实现读写锁，因为它是互斥的

**Condition**

可以直接signal监视器，synchronized使用obj，只能有一个obj





wait被唤醒后，还需要重新去竞争获取锁，如果获得锁，则从原来wait的地方向后执行。

**synchronized 实现原理**

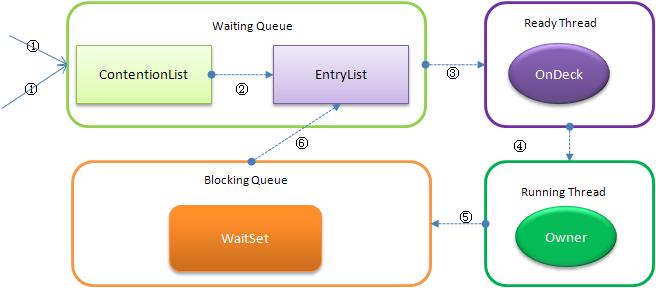
http://blog.csdn.net/tingfeng96/article/details/52219649

JVM通过monitor监视器

通过monitorenter获取锁、通过monitorexit释放锁

底层利用了几个虚拟队列，新请求锁的线程将首先被加入到ContentionList中（通过CAS实现），ContentionList会被线程并发访问，为了降低对 ContentionList队尾的争用，而建立EntryList。Owner线程在unlock时会从ContentionList中迁移线程到 EntryList，并会指定EntryList中的某个线程（一般为队头）为Ready（OnDeck）线程。Owner线程并不是把锁传递给 Ready（OnDeck）线程，只是把竞争锁的权利交给Ready（OnDeck），Ready（OnDeck）线程需要重新竞争锁。这样做虽然牺牲了一定的公平性，但极大的提高了整体吞吐量，在 Hotspot中把OnDeck的选择行为称之为“竞争切换”。

OnDeck线程获得锁后即变为owner线程，无法获得锁则会依然留在EntryList中，考虑到公平性，在EntryList中的位置不 发生变化（依然在队头）。如果Owner线程被wait方法阻塞，则转移到WaitSet队列；如果在某个时刻被notify/notifyAll唤醒， 则再次转移到EntryList。还要重新获取锁



那synchronized实现何时使用了自旋锁？答案是在线程进入ContentionList时，也即第一步操作前。线程在进入等待队列时 首先进行自旋尝试获得锁，如果不成功再进入等待队列。这对那些已经在等待队列中的线程来说，稍微显得不公平。还有一个不公平的地方是自旋线程可能会抢占了 Ready线程的锁。自旋锁由每个监视对象维护，每个监视对象一个。

**Lock实现**

java.util.concurrent.AbstractQueuedSynchronizer

基于AQS实现

AQS实现了一个先进先出**fifo的等待队列**，此类支持默认的独占模式和共享模式（CountDownLatch，可以获得多个资源）之一，或者二者都支持。处于独占模式下时，其他线程试图获取该锁将无法取得成功。在共享模式下，多个线程获取某个锁可能（但不是一定）会获得成功。ReentrantLock只支持独占模式

简单说来，AbstractQueuedSynchronizer会把所有的**请求线程**构成一个CLH队列，当一个线程执行完毕（lock.unlock()）时会激活自己的后继节点

与synchronized相同的是，这也是一个虚拟队列，不存在队列实例，仅存在节点之间的前后关系。令人疑惑的是为什么采用CLH队列呢？原生的CLH队列是用于自旋锁，但Doug Lea把其改造为阻塞锁。

当有线程竞争锁时，该线程会首先尝试获得锁，这对于那些已经在队列中排队的线程来说显得不公平，这也是非公平锁的由来，与synchronized实现类似，这样会极大提高吞吐量。 如果已经存在Running线程，则新的竞争线程会被追加到队尾，具体是采用**基于CAS**的Lock-Free算法，因为线程并发对Tail调用CAS可能会导致其他线程CAS失败，解决办法是循环CAS直至成功。

AbstractQueuedSynchronizer通过构造一个基于阻塞的CLH队列容纳所有的阻塞线程，而对该队列的操作均通过Lock-Free（无锁，CAS）操作，但对已经获得锁的线程而言，ReentrantLock实现了偏向锁的功能。

AQS的acquire方法，尝试获取资源（CAS），如果成功则返回，如果失败则加入fifo的队列中并阻塞。等到前面的节点都出去了，轮到自己的时候再循环尝试获取资源。//整个过程忽略中断

AQS的release方法，尝试释放资源，然后再唤醒等待队列中的下一个节点。

aqs实现的公平锁和非公平锁的最主要的区别是：**非公平锁中，那些尝试获取锁且尚未进入等待队列的线程会和等待队列head结点的线程发生竞争。公平锁中，在获取锁时，增加了isFirst(current)判断，当且仅当，等待队列为空或当前线程是等待队列的头结点时，才可尝试获取锁。**

synchronized 的底层也是一个基于CAS操作的等待队列，但JVM实现的更精细，把等待队列分为ContentionList和EntryList，目的是为了降低线程的出列速度；当然也实现了偏向锁，从数据结构来说二者设计没有本质区别。但synchronized还实现了自旋锁，并针对不同的系统和硬件体系进行了优化，而Lock则完全依靠系统阻塞挂起等待线程。

用户态、内核态

内核态: CPU可以访问内存所有数据, 包括外围设备, 例如硬盘, 网卡. CPU也可以将自己从一个程序切换到另一个程序

用户态: 只能受限的访问内存, 且不允许访问外围设备. 占用CPU的能力被剥夺, CPU资源可以被其他程序获取

为什么要有用户态和内核态？

由于需要限制不同的程序之间的访问能力, 防止他们获取别的程序的内存数据, 或者获取外围设备的数据, 并发送到网络, CPU划分出两个权限等级 -- 用户态 和 内核态

**锁优化（JDK1.6）synchronized**

**自旋锁**

线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能。

缓解上述问题的办法便是自旋，其原理是：当发生争用时，那些正在争用线程可以稍微等一等（自旋）， 在释放锁后，争用线程可能会立即得到锁，从而避免了系统阻塞。如果争用线程自旋一段时间后还是无法获得锁，这时争用线程则会停止自旋进入阻塞状态（后退）。基本思路就是自旋，不成功再阻塞，尽量降低阻塞的可能性，这对那些执行时间很短的代码块来说有非常重要的性能提高。自旋锁有个更贴切的名字：自旋-指数后退锁，也即复合锁。很显然，自旋在多处理器上才有意义。（单cpu，自旋占着cpu，持有锁的线程无法执行任务）

还有个问题是，线程自旋时做些啥？其实啥都不做，可以执行几次for循环，可以执行几条空的汇编指令，目的是占着CPU不放，等待获取锁的机会。所以说，自旋是把双刃剑，如果旋的时间过长会影响整体性能，时间过短又达不到延迟阻塞的目的。

**锁消除**

是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判断依据来源于逃逸分析的数据支持，如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其它线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁自然就无需进行。

public String concatString(String s1,String s2,String s3){

StringBuffer sb = new StringBuffer();//局部变量

sb.append(s1);

sb.append(s2);

sb.append(s3);

return sb.toString();

}

public synchronized StringBuffer append(String str) {

super.append(str);

return this;

}

消除锁，不用每一个append都加锁

**锁粗化**

如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部，以上面代码为例，就是扩展到第一个append()操作之前直到最后一个append()操作之后，这样只需加锁一次就可以了。



对象头的markword，用于存储对象自身的运行时数据，如：哈希吗（HashCode）、GC分代年龄(Generational GC Age)等，这部分数据的长度在32位和64位的虚拟机中分别为32bit和64bit，简称“Mark Word”

**偏向锁**

偏向锁主要解决无竞争下的锁性能问题，首先我们看下无竞争下锁存在什么问题：

现在几乎所有的锁都是可重入的，也即已经获得锁的线程可以多次锁住/解锁监视对象，按照之前的HotSpot设计，每次加锁/解锁都会涉及到一些CAS操作（比如对等待队列的CAS操作），CAS操作会延迟本地调用，因此偏向锁的想法是一旦线程第一次获得了监视对象，之后让监视对象“偏向”这个 线程，之后的多次调用则可以避免CAS操作，说白了就是置个变量，如果发现为true则无需再走各种加锁/解锁流程。

首先JVM要设置为可用偏向锁。然后当一个进程访问同步块并且获得锁的时候，会在对象头和栈帧的锁记录里面储存取得偏向锁的线程ID。

下一次有线程尝试获取锁的时候，首先检查这个对象头的MarkWord是不是储存着这个线程的ID。如果是，那么直接进去而不需要任何别的操作。如果不是，那么分为两种情况。1、对象的偏向锁标志位为0（当前不是偏向锁），说明发生了竞争，已经膨胀为轻量级锁，这时使用CAS操作尝试获得锁。2、偏向锁标志位为1，说明还是偏向锁不过请求的线程不是原来那个了。这时只需要使用CAS尝试把对象头偏向锁从原来那个线程指向目前求锁的线程。这种情况举个例子就是老王准备退休了，他儿子接替他来拿钥匙，于是仓库管理员认识了他儿子，他儿子每次来也不用登记注册了。

**轻量级锁（是指锁对象，如lock，obj）**

“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的，因此传统的锁机制就称为“重量级”锁。

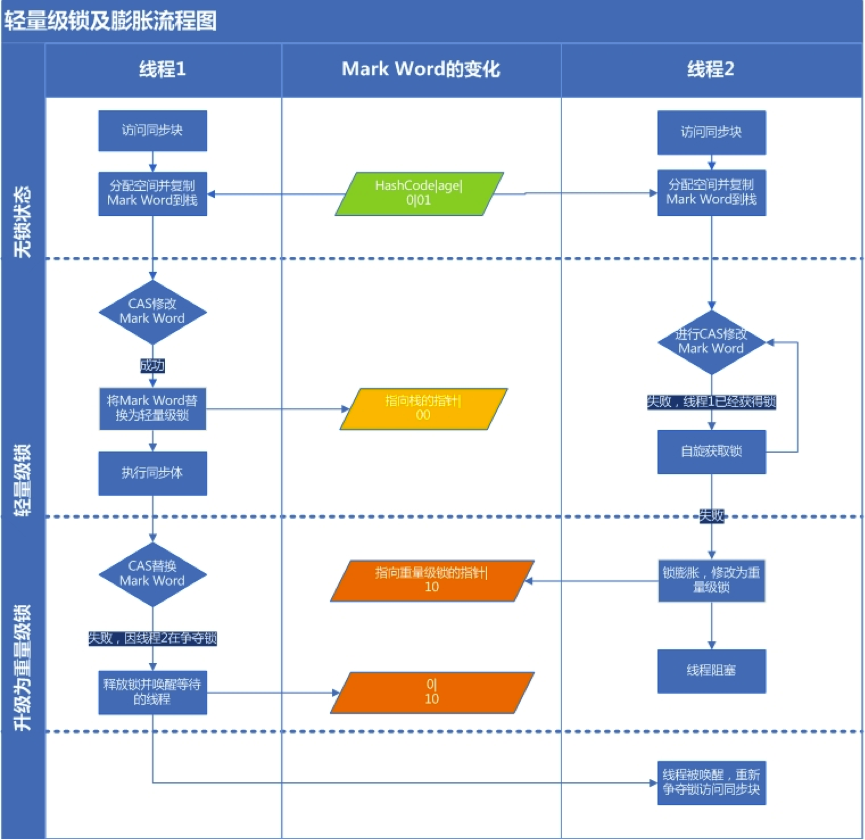
轻量级锁加锁：线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录（Lock Record）中，官方称为Displaced Mark Word。然后线程尝试使用 CAS 将对象头（锁的头）中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

轻量级锁解锁：轻量级解锁时，会使用原子的 CAS 操作来将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。

轻量级锁能提升程序同步性能的依据是“对于绝大部分的锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。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，但如果存在锁竞争（一旦有第二个线程竞争，变成重量级锁），除了互斥量的开销外，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。。

因为自旋会消耗 CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。当锁处于这个状态下，其他线程试图获取锁时，都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程，被唤醒的线程就会进行新一轮的夺锁之争。

偏向锁膨胀为轻量级锁，再膨胀为重量级锁



**JDK 1.8新特性**

1、Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法。可以有多个default方法（实现该接口的类可以用）；允许给接口添加静态方法，用static修饰，可以有多个。可以提高代码的复用性，如果实现多个接口中有方法名重复，则子类必须重写。

2、移除了永久代，改用元空间

JDK8 HotSpot JVM 将移除永久区，使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）。这意味着不会再有java.lang.OutOfMemoryError: PermGen问题，也不再需要你进行调优及监控内存空间的使用。减少发送OOM的情况

默认情况下，类元数据只受可用的本地内存限制（容量取决于是32位或是64位操作系统的可用虚拟内存大小）。新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要**动态调整**。

之前不管是不是需要，JVM都会吃掉那块空间（永久代），如果设置得太小，JVM会死掉；如果设置得太大，这块内存就被JVM浪费了。理论上说，现在你完全可以不关注这个，因为JVM会在运行时自动调校为“合适的大小”；

元空间的垃圾回收：

如果类元数据的空间占用达到参数“MaxMetaspaceSize”设置的值，将会触发对死亡对象和类加载器的垃圾回收。

为了限制垃圾回收的频率和延迟，适当的监控和调优元空间是非常有必要的。元空间过多的垃圾收集可能表示类，类加载器内存泄漏或对你的应用程序来说空间太小了。

3、lambda表达式

lambda表达式采用一种简洁的语法定义代码块。

想按照自己的方式排序，需要传入实现了comparator接口的类，如Arrays.sort(array,new MyComparator());或定时执行任务，将需要执行的任务封装成类传入Timer中。

在java中传递一个代码段并不容易，必须新建一个类，lambda表达式就是为了实现简洁的传递代码块。

(String first,String second) ->{

first.lenghth() – second.length()

}

lambda语法，参数，箭头，一个表达式

对于只有一个抽象方法的接口，需要这种接口的对象时，就可以提供一个lambda表达式，这种接口称为函数式接口。

如Arrays.sort(words,(first,second)->first.length() – second.length());

传递方法引用Math::pow等价于(x,y)->Math.pow(x,y),System.out::println等价于x->System.out.println(x)

传递构造器引用，与方法引用类似，只不过方法名为new，如Person::new是person构造器的一个引用

4、java.util.stream

Stream 就如同一个迭代器（Iterator），单向，不可往复，数据只能遍历一次，遍历过一次后即用尽了，就好比流水从面前流过，一去不复返。

而和迭代器又不同的是，Stream 可以并行化操作，迭代器只能命令式地、串行化操作。顾名思义，当使用串行方式去遍历时，每个 item 读完后再读下一个 item。而使用并行去遍历时，数据会被分成多个段，其中每一个都在不同的线程中处理，然后将结果一起输出。Stream 的并行操作依赖于 Java7 中引入的 Fork/Join 框架（JSR166y）来拆分任务和加速处理过程。

构造流的几种常见方法

// 1. Individual values

Stream stream = Stream.of("a", "b", "c");

// 2. Arrays

String [] strArray = new String[] {"a", "b", "c"};

stream = Stream.of(strArray);

stream = Arrays.stream(strArray);

// 3. Collections

String [] strArray = new String[] {"a", "b", "c"};

List<String> list = Arrays.asList(strArray);

stream = list.stream();

常见的对stream操作有filter方法，forEach方法等

roster.stream()

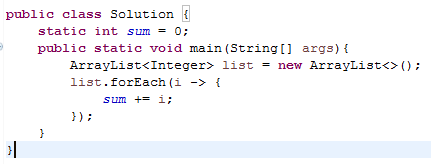
.filter(p -> p.getGender() == Person.Sex.MALE)

.forEach(p -> System.out.println(p.getName()));

Stream API 借助于同样新出现的 Lambda 表达式，极大的提高编程效率和程序可读性。同时它提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势，使用 fork/join 并行方式来拆分任务和加速处理过程。通常编写并行代码很难而且容易出错, 但使用 Stream API 无需编写一行多线程的代码，就可以很方便地写出高性能的并发程序。所以说，Java 8 中首次出现的 java.util.stream 是一个函数式语言+多核时代综合影响的产物。

**迭代遍历的几种方法**

**1**



sum在成员变量且为static（使用lambda表达式）

**2**

for(int i = 0; i < friends.size(); i++) {//++i比i++快,

System.out.println(friends.get(i));

}

前置的自增运算符（++i）速度快，因为前置的自增后直接返回引用，而后置的先定义一个临时变量，把值赋给临时变量，然后自增，返回临时变量

但是仔细想想，以上的代码似乎出现了过多的细节，比如循环变量i的出现。在做简单的遍历操作时，循环变量实际上是不必要的，只有在对某个特定位置的元素执行某个特殊操作时，循环变量的使用才有意义。所以，在Java中引入了增强的for循环，在这种循环方式中，循环变量是不必要的：**所以速度快**

**3**

for(String name : friends) {

System.out.println(name);

}

**java.nio(JDK1.4)**

Java NIO(New IO)是一个可以替代标准Java IO API的IO API（从Java 1.4开始)，Java NIO提供了与标准IO不同的IO工作方式。

Java NIO: Channels and Buffers（通道和缓冲区）

标准的IO基于字节流和字符流进行操作的，而NIO是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

Java NIO: Non-blocking IO（**非阻塞IO**）

Java NIO可以让你非阻塞的使用IO，例如：当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

Java NIO: Selectors（选择器）

Java NIO引入了选择器的概念，选择器用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。

Channel(FileChannel、SocketChannel)实现类

Buffer(ByteBuffer、CharBuffer、DoubleBuffer、IntBuffer) 实现类

ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buffer);

注意第二行，从通道读取字节到ByteBuffer。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。读到的数据可能不是完整的。

public class Program {

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

FileInputStream fin = new FileInputStream("c:\\test.txt");

// 获取通道

FileChannel fc = fin.getChannel();

// 创建缓冲区

ByteBuffer buffer = ByteBuffer.allocate(1024);

// 读取数据到缓冲区

fc.read(buffer);

buffer.flip();

while (buffer.remaining()>0) {

byte b = buffer.get();

System.out.print(((char)b));

}

fin.close();

}

}

从channel中读取数据到buffer中，用buffer的get方法获取数据（put方法可以存入数据）

NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

通道（channel）中的数据总是要先读到一个Buffer，或者总是要从一个Buffer中写入。

使用Buffer读写数据一般遵循以下四个步骤：

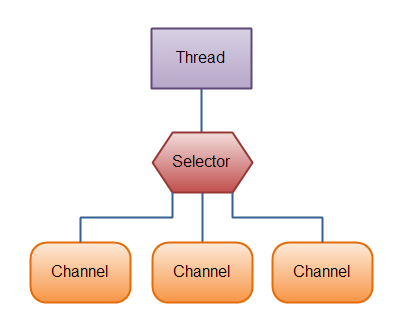
写入数据到Buffer

调用flip()方法//将写变成读

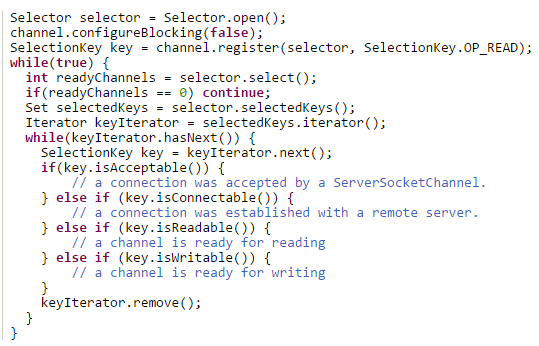
从Buffer中读取数据

调用clear()方法或者compact()方法

Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。   
  
这是在一个单线程中使用一个Selector处理3个Channel的图示：

  
要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

一旦向Selector注册了一或多个通道，就可以调用几个重载的select()方法。这些方法返回你所感兴趣的事件（如连接、接受、读或写）已经准备就绪的那些通道。换句话说，如果你对“读就绪”的通道感兴趣，select()方法会返回读事件已经就绪的那些通道。



其中Channel对应以前的流，Buffer不是什么新东西，Selector是因为nio可以使用异步的非堵塞模式才加入的东西。

以前的流总是堵塞的，一个线程只要对它进行操作，其它操作就会被堵塞，也就相当于水管没有阀门，你伸手接水的时候，不管水到了没有，你就都只能耗在接水（流）上。

nio的Channel的加入，相当于增加了水龙头（有阀门），虽然一个时刻也只能接一个水管的水，但依赖轮换策略，在水量不大的时候，各个水管里流出来的水，都可以得到妥善接纳，这个关键之处就是增加了一个接水工，也就是Selector，他负责协调，也就是看哪根水管有水了的话，在当前水管的水接到一定程度的时候，就切换一下：临时关上当前水龙头，试着打开另一个水龙头（看看有没有水）。

当其他人需要用水的时候，不是直接去接水，而是事前提了一个水桶给接水工，这个水桶就是Buffer。也就是，其他人虽然也可能要等，但不会在现场等，而是回家等，可以做其它事去，水接满了，接水工会通知他们。就绪”的通道感兴趣，select()方法会返回读事件已经就绪的那些通道。

pipe：两个线程间通信。管道由一对通道组成：一个可写入的 sink 通道和一个可读取的 source 通道。一旦将某些字节写入接收器通道，就可以按照与写入时完全相同的顺序从源通道中读取这些字节。

IO与NIO区别

面向流与面向缓冲

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

阻塞与非阻塞IO

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

选择器（Selectors）

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

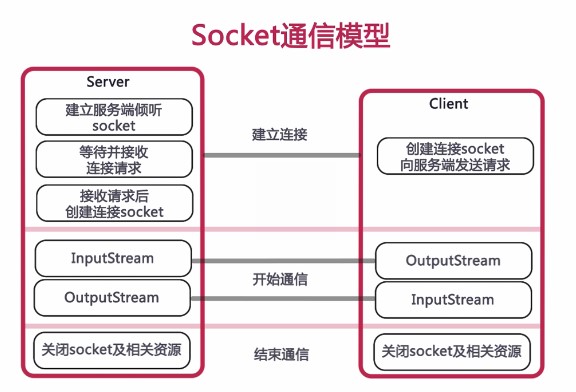
**Socket网络编程**

NIO中selector用epoll实现

Socket套接字：网络上具有唯一标识的IP地址和端口组合在一起才能构成唯一能识别的标识符套接字。

Socket原理机制：通信的两端都有Socket、网络通信其实就是Socket间的通信、数据在两个Socket间通过IO传输

Java中基于TCP实现网络通信的类：客户端的Socket类、服务器端的ServerSocket类



select，poll，epoll都是IO多路复用的机制（一个监控好多个）。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。Epoll 在 Linux2.6 内核中正式引入。

select 模型

1. 最大并发数限制，因为一个进程所打开的 FD （文件描述符）是有限制的，由 FD\_SETSIZE 设置，默认值是 1024/2048 ，因此 Select 模型的最大并发数就被相应限制了。自己改改这个 FD\_SETSIZE ？想法虽好，可是先看看下面吧 …

2. 效率问题， select 每次调用都会线性扫描全部的 FD 集合（遍历），这样效率就会呈现线性下降，把 FD\_SETSIZE 改大的后果就是，大家都慢慢来，什么？都超时了。

3. 内核 / 用户空间 内存拷贝问题，如何让内核把 FD 消息通知给用户空间呢？在这个问题上 select 采取了内存拷贝方法。

总结为：1.数据由内核拷贝到用户态2.遍历导致查找配对速度慢3.连接数受限

poll模型

poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd\_set结构，其他的都差不多。poll打开fd集合的数量没有限制，但也是遍历fd集合

epoll模型

　　epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

　　对于第一个缺点，epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。

　　对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表。epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd

　　对于第三个缺点，epoll没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

总结：

（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll\_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll\_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

（2）select，poll**每次调用**都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll\_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

Level\_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll\_wait()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你！！！如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

Edge\_triggered(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll\_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！！

select(),poll()模型都是水平触发模式，信号驱动IO是边缘触发模式，epoll()模型即支持水平触发，也支持边缘触发，默认是水平触发。

**java.io中用到****的设计模式**

在java语言 I/O库的设计中，使用了两个结构模式，即装饰模式和适配器模式

byte（即字节）占有1个字节，只能存1个字节如’a’

char（即字符）占有2个字节，可以存’a’,’哈’

装饰模式（提供更强的功能）

BufferedInputStream(InputStream in)

BufferInputStream：用来从硬盘将数据读入到一个内存缓冲区中，并从此缓冲区提供数据。

BufferedInputStream “装饰”了InputStream的内部工作方式，使得流的读入操作使用缓冲机制。在使用了缓冲机制后，不会对每一次的流读入操作都产生一个物理的读盘动作，从而提高了程序的效率。在涉及到物理流的读入时，都应当使用这个装饰流类。（InputStream的read方法：从输入流中读取数据的下一个字节）

BufferedReader、BufferedWriter、BufferedOutputStream

BufferInputStream:内部数组byte类型，protected volatile byte[] buf = new byte[8192];（即缓存数组默认8m）内部read等方法都是用synchronized修饰（在方法名上）

BufferedReader：内部数组char类型，private char[] cb = new char[8192]; 内部readLine等方法是synchronized(lock){}代码块

适配器模式（接口转换）

ByteArrayInputStream是一个适配器类 。ByteArrayInputStream继承了InputStream的接口，而封装了一个byte数组。换而言之，它将一个byte数组的接口适配成了InputStream流处理器的接口。

ByteArrayInputStream:从数组中读取数据；

FileInputStream：从文件中读取数据；

ByteArrayOutputStream、FileOutputStream、CharArrayReader、StringReader、CharArrayWriter

InputStreamReader是从byte流到char流的一个桥梁，它读入byte数据并根据指定的编码将之翻译成char数据。从byte输入流到char输入流的一个适配器（将字节转换为字符，即可以输出中文）

**Stri****ng、StringBuilder、StringBuffer**

底层都是使用一个char类型的数组保存元素（叫value）

StringBuffer是线程安全的，因为它的许多方法如append都是使用synchronized修饰的。

String是final类型的，所以也是安全的

速度StringBuilder>StringBuffer>String

简要的说， String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象, 因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后， JVM 的 GC 就会开始工作，那速度是一定会相当慢的。

而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作（对char数组操作），而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。而在某些特别情况下， String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，而特别是以下的字符串对象生成中， String 效率是远要比 StringBuffer 快的：

String S1 = “This is only a” + “ simple” + “ test”;

StringBuffer Sb = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);

String a = "helloworld";

String c = "hello" + "world";

a==b是true

你会很惊讶的发现，生成 String S1 对象的速度简直太快了，而这个时候 StringBuffer 居然速度上根本一点都不占优势。其实这是 JVM 的一个把戏，在 JVM 眼里，这个

String S1 = “This is only a” + “ simple” + “test”; 其实就是：

String S1 = “This is only a simple test”;（在编译时期，直接编译为） 所以当然不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的 String 对象的话，速度就没那么快了，譬如：

String S2 = “This is only a”;

String S3 = “ simple”;

String S4 = “ test”;

String S1 = S2 +S3 + S4;

这时候 JVM 会规规矩矩的按照原来的方式去做

StringBuffer扩容，利用arrays.copyof从新复制数组，arrays.copyof底层先新建一个长度的数组，再用system.arraycopy复制过来。

StringBuilder和StringBuffer的父类都是AbstractStringBuilder，两者的append方法都是使用父类的append方法，append方法中有扩容，所以两者的扩容机制也一样。append也是使用system.arraycopy(native方法)复制数组

**Arrays.sort**

　　Java Arrays中提供了对所有类型的排序。其中主要分为Primitive(8种基本类型)和Object两大类。

基本类型：采用调优的快速排序；对象类型：采用改进的归并排序(Timsort)。

sort(T[] a, Comparator<? super T> c)也是采用归并（Timsort）

String数组也是当做object传入

　　Java对Primitive（int，float等原型数据）数组采用快速排序，对Object对象数组采用归并排序。

　　也就是说，优化的归并排序既快速（nlog(n)）又稳定。

　　对于对象的排序，稳定性很重要。比如成绩单，一开始可能是按人员的学号顺序排好了的，现在让我们用成绩排，那么你应该保证，本来张三在李四前面，即使他们成绩相同，张三不能跑到李四的后面去。

　　而快速排序是不稳定的，而且最坏情况下的时间复杂度是O(n^2)。

　　另外，对象数组中保存的只是对象的引用，这样多次移位并不会造成额外的开销，但是，对象数组对比较次数一般比较敏感，有可能对象的比较比单纯数的比较开销大很多。归并排序在这方面比快速排序做得更好，这也是选择它作为对象排序的一个重要原因之一。

　　排序优化：实现中快排和归并都采用递归方式，而在递归的底层，也就是待排序的数组长度小于7时，直接使用冒泡排序，而不再递归下去。

　　分析：长度为6的数组冒泡排序总比较次数最多也就1+2+3+4+5+6=21次，最好情况下只有6次比较。而快排或归并涉及到递归调用等的开销，其时间效率在n较小时劣势就凸显了，因此这里采用了冒泡排序，这也是对快速排序极重要的优化。

　　源码中的快速排序，主要做了以下几个方面的优化：

　　1）当待排序的数组中的元素个数较少时，源码中的阀值为7，采用的是插入排序。尽管插入排序的时间复杂度为0(n^2)，但是当数组元素较少时，插入排序优于快速排序，因为这时快速排序的递归操作影响性能。

　　2）较好的选择了划分元（基准元素）。能够将数组分成大致两个相等的部分，避免出现最坏的情况。例如当数组有序的的情况下，选择第一个元素作为划分元，将使得算法的时间复杂度达到O(n^2).

　　源码中选择划分元的方法:

　　　　当数组大小为 size=7 时 ，取数组中间元素作为划分元。int n=m>>1;(此方法值得借鉴)

　　　　当数组大小 7<size<=40时，取首、中、末三个元素中间大小的元素作为划分元。

　　　　当数组大小 size>40 时 ，从待排数组中较均匀的选择9个元素，选出一个伪中数做为划分元。

**Arrays.copyOf**

先对需要copy的数组类型进行判断，然后在调用system.arraycopy方法，这个方法是个native

**对象数组**

A[] a = new A[5];//不加小括号

a[0] = new A();//这里再创建对象

**Exeception**

Throwable是所有Java程序中错误处理的父类，有两种资类：Error和Exception。  
   Error：表示由JVM所侦测到的无法预期的错误，由于这是属于JVM层次的严重错误，导致JVM无法继续执行，因此，这是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息。比如：内存资源不足（outofmemoryerror、stackoverflowerror）等。对于这种错误，程序基本无能为力，除了退出运行外[别无选择](https://www.baidu.com/s?wd=%E5%88%AB%E6%97%A0%E9%80%89%E6%8B%A9&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YduHckPjc3uWuWnj9buHfv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWbzrjRznjns)，它是由Java虚拟机抛出的。  
Exception：表示可恢复的例外，这是可捕捉到的。  
Java提供了两类主要的异常:runtime exception和checked exception。checked 异常也就是我们经常遇到的IOException异常，FileNotFoundException,以及SQLException异常、InterruptedException（线程sleep、wait）都是这种异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆catch块去处理可能的异常。  
    但是另外一种异常：runtime exception，也称运行时异常，我们可以不处理。当出现这样的异常时，总是由虚拟机接管。比如：我们从来没有人去处理过NullPointerException异常，它就是运行时异常，并且这种异常还是最常见的异常之一。  
    出现运行时异常后，系统会把异常一直往上层抛，一直遇到处理代码。如果没有处理块，到最上层，如果是多线程就由Thread.run()抛出，如果是单线程就被main()抛出。抛出之后，如果是线程，这个线程也就退出了。如果是主程序抛出的异常，那么这整个程序也就退出了。运行时异常是Exception的子类，也有一般异常的特点，是可以被Catch块处理的。只不过往往我们不对他处理罢了。也就是说，你如果不对运行时异常进行处理，那么出现运行时异常之后，要么是线程中止，要么是主程序终止。  
    如果不想终止，则必须扑捉所有的运行时异常，决不让这个处理线程退出。队列里面出现异常数据了，正常的处理应该是把异常数据舍弃，然后记录日志。不应该由于异常数据而影响下面对正常数据的处理。在这个场景这样处理可能是一个比较好的应用，但并不代表在所有的场景你都应该如此。如果在其它场景，遇到了一些错误，如果退出程序比较好，这时你就可以不太理会运行时异常，或者是通过对异常的处理显式的控制程序退出。  
异常处理的目标之一就是为了把程序从异常中恢复出来

常见runtimeException：

NullPointerException - 空指针引用异常  
ClassCastException - 类型强制转换异常。  
IllegalArgumentException - 传递非法参数异常。  
ArithmeticException - 算术运算异常，除数为0  
ArrayStoreException - 向数组中存放与声明类型不兼容对象异常  
IndexOutOfBoundsException - 下标越界异常  
NegativeArraySizeException - 创建一个大小为负数的数组错误异常  
NumberFormatException - 数字格式异常  
SecurityException - 安全异常  
UnsupportedOperationException - 不支持的操作异常

异常的分类：  
① 异常的继承结构：基类为Throwable，Error和Exception继承Throwable，RuntimeException和IOException等继承Exception，具体的RuntimeException继承RuntimeException。   
② Error和RuntimeException及其子类成为未检查异常（unchecked），其它异常成为已检查异常（checked）。

每个类型的异常的特点   
Error体系 ：  
      Error类体系描述了[Java](http://lib.csdn.net/base/javase)运行系统中的内部错误以及资源耗尽的情形。应用程序不应该抛出这种类型的对象（一般是由虚拟机抛出）。如果出现这种错误，除了尽力使程序安全退出外，在其他方面是无能为力的。所以，在进行程序设计时，应该更关注Exception体系。

Exception体系包括RuntimeException体系和其他非RuntimeException的体系 ：  
① RuntimeException：RuntimeException体系包括错误的类型转换、数组越界访问和试图访问空指针等等。处理RuntimeException的原则是：如果出现RuntimeException，那么一定是程序员的错误。例如，可以通过检查数组下标和数组边界来避免数组越界访问异常。   
②其他非RuntimeException（IOException等等）：这类异常一般是外部错误，例如试图从文件尾后读取数据等，这并不是程序本身的错误，而是在应用环境中出现的外部错误。

3 异常的使用方法   
声明方法抛出异常   
① 语法：throws（略）   
② 为什么要声明方法抛出异常？   
      方法是否抛出异常与方法返回值的类型一样重要。假设方法抛出异常却没有声明该方法将抛出异常，那么客户程序员可以调用这个方法而且不用编写处理异常的代码。那么，一旦出现异常，那么这个异常就没有合适的异常控制器来解决。   
③ 为什么抛出的异常一定是已检查异常？   
      RuntimeException与Error可以在任何代码中产生，它们不需要由程序员显示的抛出，一旦出现错误，那么相应的异常会被自动抛出。而已检查异常是由程序员抛出的，这分为两种情况：客户程序员调用会抛出异常的库函数（库函数的异常由库程序员抛出）；客户程序员自己使用throw语句抛出异常。遇到Error，程序员一般是无能为力的；遇到RuntimeException，那么一定是程序存在逻辑错误，要对程序进行修改（相当于调试的一种方法）；只有已检查异常才是程序员所关心的，程序应该且仅应该抛出或处理已检查异常。   
      注意：覆盖父类某方法的子类方法不能抛出比父类方法更多的异常，所以，有时设计父类的方法时会声明抛出异常，但实际的实现方法的代码却并不抛出异常，这样做的目的就是为了方便子类方法覆盖父类方法时可以抛出异常。

如何抛出异常   
① 语法：throw（略）   
② 抛出什么异常？对于一个异常对象，真正有用的信息时异常的对象类型，而异常对象本身毫无意义。比如一个异常对象的类型是ClassCastException，那么这个类名就是唯一有用的信息。所以，在选择抛出什么异常时，最关键的就是选择异常的类名能够明确说明异常情况的类。   
③ 异常对象通常有两种构造函数：一种是无参数的构造函数；另一种是带一个字符串的构造函数，这个字符串将作为这个异常对象除了类型名以外的额外说明。   
④ 创建自己的异常：当Java内置的异常都不能明确的说明异常情况的时候，需要创建自己的异常。需要注意的是，唯一有用的就是类型名这个信息，所以不要在异常类的设计上花费精力。

捕获异常   
      如果一个异常没有被处理，那么，对于一个非图形界面的程序而言，该程序会被中止并输出异常信息；对于一个图形界面程序，也会输出异常的信息，但是程序并不中止，而是返回用错误页面。  
      语法：try、catch和finally（略），控制器模块必须紧接在try块后面。若掷出一个异常，异常控制机制会搜寻参数与异常类型相符的第一个控制器随后它会进入那个catch 从句，并认为异常已得到控制。一旦catch 从句结束对控制器的搜索也会停止。   
      捕获多个异常（注意语法与捕获的顺序）（略）   
      finally的用法与异常处理流程（略）   
      异常处理做什么？对于Java来说，由于有了垃圾收集，所以异常处理并不需要回收内存。但是依然有一些资源需要程序员来收集，比如文件、网络连接和图片等资源。   
      应该声明方法抛出异常还是在方法中捕获异常？原则：捕捉并处理哪些知道如何处理的异常，而传递哪些不知道如何处理的异常。  
再次抛出异常   
①为什么要再次抛出异常？ 在本级中，只能处理一部分内容，有些处理需要在更高一级的环境中完成，所以应该再次抛出异常。这样可以使每级的异常处理器处理它能够处理的异常。   
②异常处理流程 ：对应与同一try块的catch块将被忽略，抛出的异常将进入更高的一级。

Try{

1

2

3

}catch(Exception e){

}finally{

}

4

2抛出异常后，3不会执行，但4会执行

for循环

在for循环里面定义的变量，不能在外面获得值，这是for循环的作用域；

循环内的话，每次循环内部的局部变量在每次进for循环的时候都要重新定义一遍变量（栈中），也就是执行申请内存空间，变量压入堆栈的过程。   
 循环外定义的话，for循环一直用的是同一块内存空间，效率比较高，但是变量的作用域就大了，耗内存

**Java内存泄漏**

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。无法被GC

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

1、静态集合类引起内存泄露：

像HashMap、Vector（被定义为static）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。

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。

Person p3 = new Person("猪八戒","pwd3",27);

set.add(p3);

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

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。

每个线程中ThreadlocalMap的key为弱引用的Threadlocal对象，value为初始化值，Threadlocal的强引用被释放后，弱引用也会释放，但value有一条从当前线程的引用，只有当前线程结束后，才会释放，如果使用连接池，不释放线程，会导致泄漏

Java为了最小化减少内存泄露的可能性和影响，在ThreadLocal的get,set的时候都会清除线程Map里所有key为null的value。所以最怕的情况就是，threadLocal对象设null了，开始发生“内存泄露”，然后使用线程池，这个线程结束，线程放回线程池中不销毁，这个线程一直不被使用，或者分配使用了又不再调用get,set方法，那么这个期间就会发生真正的内存泄露。

**内存溢出**

java.lang.OutOfMemoryError内存不够用了

1) 内存泄露是导致内存溢出的原因之一；内存泄露积累起来将导致内存溢出。

2) 内存泄露可以通过完善代码来避免；内存溢出可以通过调整配置来减少发生频率，但无法彻底避免。

分类：

永久代溢出：OutOfMemoryError：PermGen space

增加java虚拟机中的XX:PermSize和XX:MaxPermSize参数的大小，其中XX:PermSize是初始永久保存区域大小，XX:MaxPermSize是最大永久保存区域大小。

堆溢出：OutOfMemoryError：Java heap space

1. 检查程序，看是否有死循环或不必要地重复创建大量对象。找到原因后，修改程序和算法。

2. 增加Java虚拟机中Xms（初始堆大小）和Xmx（最大堆大小）参数的大小。如：set JAVA\_OPTS= -Xms256m -Xmx1024m

栈溢出：

1.StackOverflowError(方法调用层次太深，内存不够新建栈帧)

2.OutOfMemoryError（线程太多，内存不够新建线程）

**Java多态**

多态存在的三个必要条件（屏蔽子类对象之间的差异，提高代码的扩展性）

一、要有继承；

二、要有重写；

三、父类引用指向子类对象。

面向对象的三个基本特征：封装、继承、多态

**Java泛型**

好处：

1. 泛型程序设计意味着编写的代码可以被很多不同类型的对象所重用（如ArrayList可以传入String、Integer等多种类型）。
2. ArrayList<String>具有更好的可读性，人们一看就知道里面包含的是String。
3. 编译器可以进行检查，避免插入错误类型的对象。编译器自动、隐式进行类型转换
4. 没有泛型前，要做显式强制类型转换，容易造成运行时异常，安全隐患。

总之，可以提高代码重用率，使程序具有更好的可读性和安全性。

实现原理：

类型擦除

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。（如List<String>编译后变成List，JVM只看到List）。

类型擦除后保留的原始类型（字节码中）：

1. 无限定的用Object（如public class A<T>，类中的T会用Object替代）
2. 有限定的用限定类型（如public class A<T extends B>，类中的T会用B替代）

假设参数类型的占位符为T，擦除规则如下：

<T>擦除后变为Obecjt

<? extends A>擦除后变为A

<？ super A>擦除后变为Object

上述擦除规则叫做保留上界。

**重写与重载**

1、重写发生在父类与子类之间，重写必须有相同的方法名，参数列表和返回类型。重写的方法作用域必须大于等于原方法。重写方法不能抛出新的异常或者比被重写方法声明的检查异常更广的检查异常。但是可以抛出更少，更有限或者不抛出异常。（原来调用父类的地方，再调用子类抛出异常不能更大）静态方法可以重写和重载（都是static的）。

例子： 在父类中是public的方法，如果子类中将其降低访问权限为private， 那么父类在实现多态时如果调用子类的这个重写方法，但是这个方法已经是private，没有办法调用，所以就无法实现多态了。

2、方法重载发生在同一个类里面两个或者是多个方法的方法名相同但是参数不同的情况。重载方法参数必须不同，返回值可相同可不同，作用域可相同可不同，参数相同返回值不同也不行，参数相同作用域不同也不行；

静态绑定：

在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。例如：C。

针对Java简单的可以理解为程序编译期的绑定；这里特别说明一点，**java当中的方法只有final，static，private和构造方法是前期绑定（private只有自己调用，静态方法与类相关）。**

方法重载有静态方法重载和普通方法重载。静态方法重载是静态绑定，方法调用是通过：类名.方法。普通方法重载是动态绑定，方法调用是通过：实例对象引用.方法。构造器能够重载，但是不能够被重写。

**静态方法能够被重写、重载**，但是没有实现多态效果。

**接口与抽象类**

Java提供和支持创建抽象类和接口。它们的实现有共同点，不同点在于：

接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。

类可以实现很多个接口，但是只能继承一个抽象类

类可以不实现抽象类和接口声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

抽象类可以在不提供接口方法实现的情况下实现接口。

Java接口中声明的变量默认、必须都是public static final。抽象类可以包含非final的变量。

Java接口中的成员函数默认、必须是public abstract的。抽象类的成员函数可以是private，protected或者是public。

接口是绝对抽象的，不可以被实例化。抽象类也不可以被实例化，但是，如果它包含main方法的话是可以被调用的。

jdk1.8 default、 static

枚举类可以实现接口，但不能继承类，因为所有枚举类都继承自java.lang.Enum（由编译器添加），同时java不支持多继承（已经自动继承了一个）。枚举类也无法被继承。枚举类中可以有自己的构造函数，可以实现接口的方法。

**数据库连接池原理**

有一个LinkedList，先初始化几个数据库连接，从头get，用完后再放到尾部，并发时可以加synchronized关键字，超时过期机制，代码复用。不能都从头get或put，若不然两者都会加锁互斥。

**遍历**

**遍历list**

方法一：  
超级for循环遍历  
for(String attribute : list) {  
 System.out.println(attribute);  
}  
方法二：  
对于ArrayList来说速度比较快, 用for循环, 以size为条件遍历:  
for(int i = 0 ; i < list.size() ; i++) {  
 system.out.println(list.get(i));  
}  
方法三：  
集合类的通用遍历方式, 从很早的版本就有, 用迭代器迭代  
Iterator it = list.iterator();  
while(it.hasNext()) {  
 System.ou.println(it.next);  
}

**遍历set**

1.迭代遍历：  
Set<String> set = new HashSet<String>();  
Iterator<String> it = set.iterator();  
while (it.hasNext()) {  
  String str = it.next();  
  System.out.println(str);  
}  
  
2.for循环遍历：  
for (String str : set) {  
      System.out.println(str);  
}

**遍历map**

public static void main(String[] args) {

  Map<String, String> map = new HashMap<String, String>();  
  map.put("1", "value1");  
  map.put("2", "value2");  
  map.put("3", "value3");  
    
  //第一种：普遍使用，二次取值(keyset也有iterator)  
  System.out.println("通过Map.keySet遍历key和value：");  
  for (String key : map.keySet()) {  
   System.out.println("key= "+ key + " and value= " + map.get(key));  
  }  
    
  //第二种  
  System.out.println("通过Map.entrySet使用iterator遍历key和value：");  
  Iterator<Map.Entry<String, String>> it = map.entrySet().iterator();  
  while (it.hasNext()) {  
   Map.Entry<String, String> entry = it.next();  
   System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());  
  }  
    
  //第三种：推荐，尤其是容量大时  
  System.out.println("通过Map.entrySet遍历key和value");  
  for (Map.Entry<String, String> entry : map.entrySet()) {  
   System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());  
  }

  //第四种  
  System.out.println("通过Map.values()遍历所有的value，但不能遍历key");  
  for (String v : map.values()) {  
   System.out.println("value= " + v);  
  }  
 }

**Servlet生命周期**

所谓生命周期，指的是servlet容器如何创建servlet实例、分配其资源、调用其方法、并销毁其实例的整个过程。（init初始化、service每个请求、destroy容器销毁）Servlet容器的作用是负责处理客户请求，当客户请求来到时，Servlet容器获取请求，然后调用某个Servlet，并把Servlet的执行结果返回给客户。

阶段一: 实例化（就是创建servlet对象,调用构造器）

在如下两种情况下会进行对象实例化。

第一种情况：

当请求到达容器时，容器查找该servlet对象是否存在，如果不存在，才会创建实例。

第二种情况：

容器在启动时，或者新部署了某个应用时，会检查web.xml当中，servlet是否有 load-on-starup配置。如果有，则会创建该servlet实例。

load-on-starup参数值越小，优先级越高（最小值为0，优先级最高）。

Servlet被装载后，Servlet容器创建一个Servlet实例并且调用Servlet的init()方法进行初始化

阶段二： 初始化

为servlet分配资源，调用init(ServletConfig config);方法

config对象可以用来访问servlet的初始化参数。

初始化参数是使用init-param配置的参数。

init可以override。

阶段三： 就绪/调用

对于用户到达Servlet的请求，Servlet容器会创建特定于这个请求的ServletRequest对象和ServletResponse对象，然后调用Servlet的service方法。service方法从ServletRequest对象获得客户请求信息，处理该请求，并通过ServletResponse对象向客户返回响应信息。

HttpServlet的service()方法，会依据请求方式来调用doGet()或者doPost()方法。但是，这两个do方法默认情况下，会抛出异常，需要子类去override。

GenericServlet是一个通用的，不特定于任何协议的Servlet,它实现了Servlet接口。而HttpServlet继承于GenericServlet，因此HttpServlet也实现了Servlet接口。所以我们定义Servlet的时候只需要继承HttpServlet即可。

阶段四： 销毁

容器依据自身的算法，将不再需要的servlet对象删除掉。

在删除之前，会调用servlet对象的destroy()方法。

destroy()方法用于释放资源。可以在这个方法中执行一些清理动作，比如释放数据库连接，关闭打开的文件等。

在servlet的整个生命周期当中，init,destroy只会执行一次，而service方法会执行多次。

servlet是单例模式的，不是线程安全的，当有多个线程访问时，全局变量或静态变量会发生线程安全问题。

**Servlet与JSP区别**

1.jsp经编译后就变成了Servlet.(JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码,Web容器将JSP的代码编译成JVM能够识别的java类)

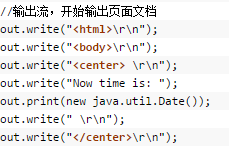
2.jsp更擅长表现于页面显示,servlet更擅长于逻辑控制.

3.Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到.

Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。

Servlet和JSP都是java中的类，需要创建实例

tomcat将jsp生成对应的Servlet（.java文件），html标签等使用out.write输出



**转发与重定向**

重定向和转发有一个重要的不同：当使用转发时，JSP容器将使用一个内部的方法来调用目标页面，新的页面继续处理同一个请求，而浏览器将不会知道这个过程。 与之相反，重定向方式的含义是第一个页面通知浏览器发送一个新的页面请求。因为，当你使用重定向时，浏览器中所显示的URL会变成新页面的URL, 而当使用转发时，该URL会保持不变。重定向的速度比转发慢，因为浏览器还得发出一个新的请求。同时，由于重定向方式产生了一个新的请求，所以经过一次重 定向后，request内的对象将无法使用。

HttpServletResponse.sendRedirect 方法实现的请求重定向与RequestDispatcher.forward 方法实现的请求转发

**如何解决死锁？**

有三个方法:

1.进程在申请资源时，一次性得请求他所需要的所有资源。若无法满足则不能执行。

2.进程在申请新的资源时，释放已占有的资源。后面若还需要它们，则需要重新申请。

3.将系统中的资源顺序编号，规定进程只能依次申请资源。

死锁的条件：互斥、持有并等待、资源不可被剥夺、循环等待（p0等待p1资源，p1等P2，p2等p0）

预防死锁-破坏四个必要条件之一（但代价大（影响程序效率），不太可行）

Dijkstra的[银行家算法](http://baike.baidu.com/item/%E9%93%B6%E8%A1%8C%E5%AE%B6%E7%AE%97%E6%B3%95)（Banker's Algorithm）（动态检测资源分配，确保“循环等待”条件永远不成立）是一个避免死锁（Deadlock）的著名算法

每个进程必须事先预报它将用到的最大资源；算法确保系统处于safty状态

**过滤器、监听器、拦截器**

1.过滤器：所谓过滤器顾名思义是用来过滤的，在java web中，你传入的request,response提前过滤掉一些信息，或者提前设置一些参数，然后再传入servlet或者struts的action进行业务逻辑，比如过滤掉非法url（不是login.do的地址请求，如果用户没有登陆都过滤掉）,或者在传入servlet或者struts的action前统一设置字符集，或者去除掉一些非法字符（聊天室经常用到的，一些骂人的话）。filter 流程是线性的， url传来之后，检查之后，可保持原来的流程继续向下执行，被下一个filter, servlet接收等.

过滤器有四种拦截方式分别是：REQUEST（默认，请求）、FORWARD（转发）、INCLUDE（包含）、ERROR（错误）。

2.监听器：这个东西在c/s模式里面经常用到，他会对特定的事件产生产生一个处理。监听在很多模式下用到。比如说观察者模式，就是一个监听来的。又比如struts可以用监听来启动。Servlet监听器用于监听一些重要事件的发生，监听器对象可以在事情发生前、发生后可以做一些必要的处理。

3. 拦截器是在面向切面编程中应用的，就是在你的service或者一个方法前调用一个方法，或者在方法后调用一个方法。是基于JAVA的反射机制。拦截器不是在web.xml，比如struts在struts.xml中配置，

**数据库**

**Inn****oDB与MyISAM区别**

Oracle收购sun（sun之前收购了mySQL），发布的首个版本5.5，默认使用了InnoDB作为存储引擎，而之前的版本使用MyISAM作为默认。

**MyISAM 和 InnoDB的适用场景**

MyISAM适合：读多写少(1)做很多count 的计算；(2)插入不频繁，查询非常频繁；(3)没有事务。

InnoDB适合：写多读少(1)可靠性要求比较高，或者要求事务；(2)表更新和查询都相当的频繁，并且表锁定的机会比较大的情况。

**为什么MyISAM会比Innodb 的查询速度快**

INNODB在做SELECT的时候，要维护的东西比MYISAM引擎多很多；

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

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

3）INNODB还需要维护MVCC一致；虽然你的场景没有，但他还是需要去检查和维护MVCC ( Multi-Version Concurrency Control )多版本并发控制（读不加锁，读写不冲突。）

**两种类型最主要的差别就是Innodb 支持事务处理与外键和行级锁。**

1、存储结构

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：.ibd的文件，存储与该表相关的数据、索引、表的内部数据字典（表缓存）信息；.frm表结构文件。

2、数据导出：MyISAM只要发给他们对应那表的frm.MYD,MYI的文件，让他们自己在对应版本的数据库启动就行，而Innodb就需要导出xxx.sql了，因为光给别人文件，受字典数据文件的影响，对方是无法使用的。

3、MyISAM的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而Innodb是索引和数据是紧密捆绑的，没有使用压缩从而会造成Innodb占用空间较大

4、MyISAM存储空间几乎没有限制，最多可到64PB，InnoDB最多64TB

5、InnoDB支持事务（每条sql都是事务，默认配置事务自动提交）和外键，MyISAM不支持

6、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。

7、表锁差异

MyISAM：只支持表级锁，select，update，delete，insert语句都会给表自动加锁  
InnoDB：支持行级锁，但是InnoDB的行锁，只是在WHERE的主键（唯一索引）是有效的，非主键的WHERE都会锁全表的。对索引加锁，而不是对数据行加锁，只有当查询条件能使用索引的时候才会使用行级锁

两者锁读的时候共享锁、写的时候排它锁（可以一起读，不能一起写或读写）

8、全文索引（通过关键字匹配来查询）

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。（从5.6开始支持，但不支持中文索引）

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

11、MyISAM B树，InnoDB B+树



**MySQL其他存储引擎**

Blackhole引擎，它会丢弃所有插入的数据，不做任何保存。但会记录日志。

CSV引擎，可以将CSV文件作为MySQL表来处理，可以作为一种数据交换机制。

Memory引擎，数据存在内存中，访问速度快，重启后数据会丢失。

Merge引擎，是MyISAM的变种，由多个MyISAM表合并而来的虚拟表。

**事务四大特性(简称ACID)**

1、原子性(Atomicity)：事务中的全部操作在数据库中是不可分割的，要么全部完成，要么均不执行。

2、一致性(Consistency)：几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。事务必须是使数据库从一个一致性状态变到另一个一致性状态。多个账户钱的总和不变

3、隔离性(Isolation)：事务的执行不受其他事务的干扰，事务执行的**中间结果**对其他事务必须是透明的。（即事务隔离级别）（转钱时，从其他用户角度看）

4、持久性(Durability)：对于任意已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

**事务一致性**

强一致性：读操作可以立即读到提交的更新操作。

弱一致性：提交的更新操作，不一定立即会被读操作读到，此种情况会存在一个不一致窗口，指的是读操作可以读到最新值的一段时间。

最终一致性：是弱一致性的特例。事务更新一份数据，最终一致性保证在没有其他事务更新同样的值的话，最终所有的事务都会读到之前事务更新的最新值。如果没有错误发生，不一致窗口的大小依赖于：通信延迟，系统负载等。

其他一致性变体还有：

单调一致性：如果一个进程已经读到一个值，那么后续不会读到更早的值。

会话一致性：保证客户端和服务器交互的会话过程中，读操作可以读到更新操作后的最新值。

**事务隔离级别**

1，脏读

　　脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

　　当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元

　　当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

2，不可重复读

　　不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。不可重复读出现的原因就是事务并发修改记录，要避免这种情况，最简单的方法就是对要修改的记录加锁，这回导致锁竞争加剧，影响性能。另一种方法是通过MVCC可以在无锁的情况下，避免不可重复读。

　　例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

　　不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

3，虚读(幻读)

　　在同一个事务中，同一个查询多次返回的结果不一致。事务A新增了一条记录，事务B在事务A提交前后各执行了一次查询操作，发现后一次比前一次多了一条记录。就好像产生幻觉一样，这就是发生了幻读。幻读是由于并发事务增加记录导致的，这个不能像不可重复读通过记录加锁解决，因为对于新增的记录根本无法加锁。需要将事务串行化，才能避免幻读。

　　幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

　　MySQL数据库的四种隔离级别（从低到高、为解决并发事务问题）：

①Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

②Read committed (读已提交)：只有在事务提交后，其更新结果才会被其他事务看见。可避免脏读的发生。

③Repeatable read (可重复读)：在一个事务中，对于同一份数据的读取结果总是相同的，无论是否有其他事务对这份数据进行操作，以及这个事务是否提交。可避免脏读、不可重复读的发生。

④Serializable (串行化)：事务串行化执行，隔离级别最高，牺牲了系统的并发性。可避免脏读、不可重复读、幻读的发生。

　　在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

**为什么mysql事务能保证失败回滚**

进行事务处理的时候，MySQL 在开始事务时会切换到一个延缓操作的状态，这个状态下操作并不都是立即执行的（通常情况下语句是立即执行的）。而在 commit 时，会将延缓执行的操作都执行进去，并将状态回归到及时写入状态。同样的， rollback 时会把延缓写入的操作抛弃掉，此间申请的锁释放掉，并将状态回归到及时写入状态。

执行 rollback 的关键在于释放 申请的锁 和 回归及时写入状态，而并不是放弃未写入的操作（你关心的点在未写入的操作，然而执行与不执行 rollback 都没有操作写进去，所有你感觉执行或不执行都没什么区别）。

或者是显示地使用savepoint，rollback到之前设置的savepoint

**spring的事务**

使用spring声明式事务，spring使用AOP来支持声明式事务，会根据事务属性，自动在方法调用之前决定是否开启一个事务，并在方法执行之后决定事务提交或回滚事务。

简单地说，编程式事务侵入到了业务代码里面，但是提供了更加详细的事务管理；而声明式事务由于基于AOP，所以既能起到事务管理的作用，又可以不影响业务代码的具体实现。

在xml中配置transactionManager这个bin，在DAO中方法上加@Transactional注解（可以设置事务隔离级别），从而完成声明式事务。

TransactionInfo和TransactionStatus这两个对象，它们存放事务处理信息的主要数据对象，它们通过与线程的绑定（ThreadLocal）来实现事务的隔离性。

**数据库完整性约束**

数据的完整性

约束是用来确保数据的准确性和一致性。数据的完整性就是对数据的准确性和一致性的一种保证。

数据完整性(Data Integrity)是指数据的精确(Accuracy)和可靠性(Reliability)。

分为以下四类：

1) 实体完整性：规定表的每一行在表中是惟一的实体。

2) 域完整性：是指表中的列必须满足某种特定的数据类型约束，其中约束又包括取值范围、精度等规定。

3) 参照完整性：是指两个表的主关键字和外关键字的数据应一致，保证了表之间的数据的一致性，防止了数据丢失或无意义的数据在数据库中扩散。

4) 用户定义的完整性：不同的关系数据库系统根据其应用环境的不同，往往还需要一些特殊的约束条件。用户定义的完整性即是针对某个特定关系数据库的约束条件，它反映某一具体应用必须满足的语义要求。

与表有关的约束

主键约束(PK)primary key constraint 唯一且不为空

唯一约束(UQ)unique constraint唯一，允许为空，即可以再其中出现null值，但只能出现一次

默认约束(DF)default constraint默认值

检查约束(CK)check constraint范围以及格式限制

外键约束(FK)foreign key constraint表关系

**数据库调优**

MySQL调优分析explain；show status查看服务器状态信息

**SQL语句的优化**

分析：确认程序是否存在查询不需要的记录；mysql是否在扫描额外记录

1、查询不需要的记录：使用select语句查询大量结果，然后再获取前N行（如新闻网站，取100条记录，只显示前面的10条），这时可以使用limit（limit 1，10；从1开始10行）

2、总是使用SELECT \*，对I/O、内存消耗较大，不必要时不要这样。

3、子查询的性能又比外连接性能慢，尽量用外连接来替换子查询。

Select\* from A where exists (select \* from B where id>=3000 and A.uuid=B.uuid);

一种简单的优化就是用innerjoin的方法来代替子查询，查询语句改为：

Select\* from A inner join B using(uuid) where b.uuid>=3000;

4、尽量少排序，排序操作会消耗较多的CPU资源（可以使用索引）

5、对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

6、切分查询，将大查询切分成小查询，每个查询功能一样，只完成一小部分，如果用一个大的语句一次性完成的话，则可能需要一次锁住很多数据、耗尽系统资源、阻塞很多小的但重要的查询。

**索引的优化**

1、建立索引加快查询性能，优先在经常搜索的字段上建立索引（where）；WHERE子句的查询条件里使用了比较操作符LIKE前置通配符%（如：LIKE "%ABC"），因为‘%’代表任何字符，%xxx不知道怎么去索引的，所以使用不了索引。只要列中包含有NULL值都将不会被包含在索引中

2、应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：   
select id from t where num=10 or num=20   
可以这样查询：   
select id from t where num=10   
union all   
select id from t where num=20

UNION 操作符用于合并两个或多个 SELECT 语句的结果集。请注意，UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。不是同一个表也可以union只要列数相同，UNION 结果集中的列名总是等于 UNION 中第一个 SELECT 语句中的列名。UNION ALL允许重复

3、in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

4、有函数的参数，不使用索引

select \* from user where age + 1 =20;

5、应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描

6、在多个列上建立独立的单列索引大部分情况下不能提高查询性能，可以使用复合索引（多列索引），key(name,age,sex)

多列索引有一个特点，即最左前缀（Leftmost Prefixing）。假如有一个多列索引为key(firstname lastname age)，当搜索条件是以下各种列的组合和顺序时，MySQL将使用该多列索引：

firstname，lastname，age

firstname，lastname

firstname

也就是说，相当于还建立了key(firstname lastname)和key(firstname)。但是只搜lastname用不到索引

7、由于复合索引最左前缀匹配，将搜索次数多的列放到最前列（建立B+树时从最左字段开始排序，第一个字段相同才排第二个字段，因为查找树需要一定的顺序）

8、尽量选择小而简单的数据类型做索引，减少磁盘空间

9、有时候需要索引很长的字符列，这会让索引变得大且慢。一个方式是使用哈希索引，另一个是使用前缀索引，即索引开始的部分字符串，这样可以节约索引空间，提高效率。但这样会降低索引的选择性（不重复的索引值/记录总数）。索引的选择性越高则查询效率越高，唯一索引的选择性是1，性能是最好的。一般情况，某个前缀的选择性也是足够高的。

**数据库表结构****的****优化**

选择合适的数据类型

数据类型越小越好：尽量使用可以正确存储数据的最小数据类型（tinyint，占用更少的磁盘、内存、cpu缓存）；tinyint占1字节，int占4字节

简单就好：如整型比字符操作代价更低（用整型存储ip地址）

避免NULL：最好指定列为NOT NULL，因为NULL更难优化，使用的索引更复杂

不要设计太多列：减少不必要的列

适度冗余，减少join关联查询

适当拆分，水平拆分、垂直拆分

选择合适的字符编码：如果我们可以确定不需要存放多种语言，就没必要非得使用UTF8或者其他UNICODE字符类型，这回造成大量的存储空间浪费

**MySQL配置文件优化**

mysql的配置文件名为my.cnf（window为my.ini），不同情况下配置文件参数设置也不相同，应该根据具体场景调优。

InnoDB中最重要的选项是：

innodb\_buffer\_pool\_size：缓存用户表（实际数据row）及索引数据的最主要缓存空间，对 Innodb 整体性能影响也最大，默认为8MB，建议设为内存的70%~80%（MyISAM只缓存索引）

innodb\_log\_file\_size：日志文件大小，默认为48MB，应该调大，至少有几百MB

**硬件的优化**

**CPU的选择**

最好的选择是核心数多并且主频高的。但是有时考虑成本问题，可以参考以下情况：

如果不是密集型的查询，优先选频率高的，而不是数量多的

如果是密集型的、高并发的查询，比如秒杀等活动，优先选更多的cpu

因为一条sql语句只能在一个cpu上执行

**内存的选择**

内存并不是容量越大，性能提升越明显。如果内存大小已经超过了总数据量的大小，那么即使再增加内存，系能提升也不会特别明显。

内存频率选择cpu支持的最高的频率，品牌、型号、规格等要一致。

**磁盘配置和选择**

各种磁盘性能比较：

PCIe > SSD > Raid10 > 磁盘 > 网络存储

各种磁盘的特点和应用：

传统硬盘：需要考虑存储容量、传输速度、访问时间、主轴转速、物理尺寸等参数

raid增加传统硬盘性能：主服务器建议用raid10，从服务器可以raid0（raid0：数据等量放置在2块磁盘中，raid1：让同一份数据完整保存在两块磁盘）

SSD或者PCIe卡（FusionIO）：缺点比传统硬盘更容易坏

SSD应用场景：适用于存在大量随机I/O场景（SSD随机I/O快）、适用于解决单线程负载的I/O瓶颈（用在从服务器上，适用于读的场景，频繁写会减少使用寿命）

网络存储场景（NAS、SAN）：数据库备份

**操作系统**

合理配置操作系统参数，选择合适的文件系统

**加缓存redis**

**先读写分离、再垂直拆分、再水平拆分**

分库：根据业务逻辑垂直拆分

分表：

纵向分表（常见为忙闲分表）

单数据表字段过多，可将频繁更新的整数数据与非频繁更新的字符串数据切分  范例 user表 ，个人简介，地址，QQ号，联系方式，头像 这些字段为字符 串类型，更新请求少； 最后登录时间，在线时常，访问次数，信件数这些字段为整数型字段，更新频繁，可以将后面这些更新频繁的字段独立拆出一张数据表，表内容变少，索引结构变少，读写请求变快。

横向切表

等分切表，如哈希切表或其他基于对某数字取余的切表（USERID奇数、偶数）。等分切表的优点是负 载很方便的分布到不同服务器；缺点是当容量继续增加时无法方便的扩容，需要重新进行数据的切分或转表。而且一些关键主键不易处理。

递增切表，比如每1kw用户开一个新表，优点是可以适应数据的自增趋势； 缺点是往往新数据负载高，压力分配不平均。

**索引的好处**

建立数据库索引的好处  
创建索引可以大大提高数据库系统的查询性能、将随机I/O变为顺序I/O（磁盘顺序）。  
a) 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。  
b) 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。  
c) 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。  
d) 在使用分组和排序子句进行[数据检索](https://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E6%A3%80%E7%B4%A2&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3Pvu9PjK9PWT1ryfsP1Rv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWf4P10dn1f4)时，同样可以 显著减少查询中分组和排序的时间。  
e) 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。  
建立索引需要付出的代价  
建立索引的目的是加快对表中记录的查找或排序。但是为表设置索引是要付出代价的：这个代价有几个个方面  
a) 索引需要占物理空间  
除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。  
b) 创建索引和维护索引要耗费时间  
这种时间随着数据量的增加而增加。  
c) 降低维护速度  
当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度，同样降低了效率。

**判断是否应该建索引的条件**

1、较频繁的作为查询条件的字段应该创建索引

2、唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件

3、增、删、改操作较多的数据库字段不适合建索引（加了索引后update会增加耗时）

4、表记录比较少，例如一两千条甚至只有几百条记录的表，没必要建索引，让查询做全表扫描就好了。

5、性别字段（只有男、女）不适合建索引，性别字段的选择性低，使用索引查找还不如遍历表的效率高。最多只能少检索一半，但索引的额外开销更大，得不偿失

**索引的类型**

**从数据结构角度：B-Tree索引、hash索引（memory引擎支持）、R-Tree索引（空间数据索引，MyISAM支持，用作地理数据存储）、FULLTEXT（全文）索引（MyISAM、InnoDB支持）**

**从物理存储角度：聚集索引、非聚集索引**

**从逻辑角度：普通索引、唯一索引、主键索引、空间索引（只有MyISAM支持且支持的不好）**

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](http://baike.baidu.com/view/68068.htm)或[唯一约束](http://baike.baidu.com/view/1550298.htm)。

**普通索引** 索引值可以不唯一

**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。

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](http://baike.baidu.com/view/68068.htm)。

在数据库关系图中为表定义主键将自动创建主键[索引](http://baike.baidu.com/view/262241.htm)，主键索引是[唯一索引](http://baike.baidu.com/view/709651.htm)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](http://baike.baidu.com/view/692530.htm)

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。

聚集索引和非聚集索引的区别，如字典默认按字母顺序排序，读者如知道某个字的读音可根据字母顺序快速定位。因此聚集索引和表的内容是在一起的。如读者需查询某个生僻字，则需按字典前面的索引，举例按偏旁进行定位，找到该字对应的页数，再打开对应页数找到该字。这种通过两个地方而查询到某个字的方式就如非聚集索引。

主键一唯一索引的区别：  
   1 一个表的主键只能有一个，而唯一索引可以建多个。  
   2 主键可以作为其它表的外键。（**外键不一定是主键，只要唯一就行**；SQL的主键和外键就是起约束作用，插入非空值时，如果主键表中没有这个值，则不能插入。可以有复合主键，但是不能有多个主键。）  
   3 主键不可为null，唯一索引可以为null。  
聚集索引：将表内的数据按照一定的规则进行排列的目录。正因为如此，一个表中的聚焦索引只有一个。对此我们要注意“主键就是聚焦索引”这是极端错误的，是对聚焦索引的一种浪费。

主键并不一定是聚集索引，只是在SQL SERVER中，未明确指出的情况下，默认将主键定义为聚集，而ORACLE中则默认是非聚集

哈希索引：基于哈希表实现，只有memory引擎显示支持哈希索引（使用链地址法解决哈希冲突），InnoDB有一个特殊的功能“自适应哈希索引”，当某些索引值被引用非常频繁，它会在内存中基于B-Tree索引之上再创建一个哈希索引。

哈希索引查找速度很快，但哈希索引数据不是按照索引顺序存储的，所以无法用于排序；不支持范围查找（如大于多少）；如果哈希冲突很多（如选择性低的），当查找或删除一行，需要遍历对应哈希值的链表的每一行，冲突越多，代价越大

**索引的创建**

在表上创建一个简单的索引。允许使用重复的值：

CREATE INDEX index\_name

ON table\_name (column\_name)

在表上创建一个唯一的索引。唯一的索引意味着两个行不能拥有相同的索引值。

CREATE UNIQUE INDEX index\_name

ON table\_name (column\_name)

假如您希望索引不止一个列，您可以在括号中列出这些列的名称，用逗号隔开：

CREATE INDEX PersonIndex

ON Person (LastName, FirstName)

创建索引还可以用alter实现

InnoDB按照主键进行聚集，如果没有定义主键，InnoDB会试着使用唯一的非空索引来代替。如果没有这种索引，InnoDB就会定义隐藏的主键（6个字节）然后在上面进行聚集。mysql不能手动创建聚集索引。

主键索引是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引

**索引的实现**

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，而B-Tree的高度低（多叉树），可以减少I/O次数

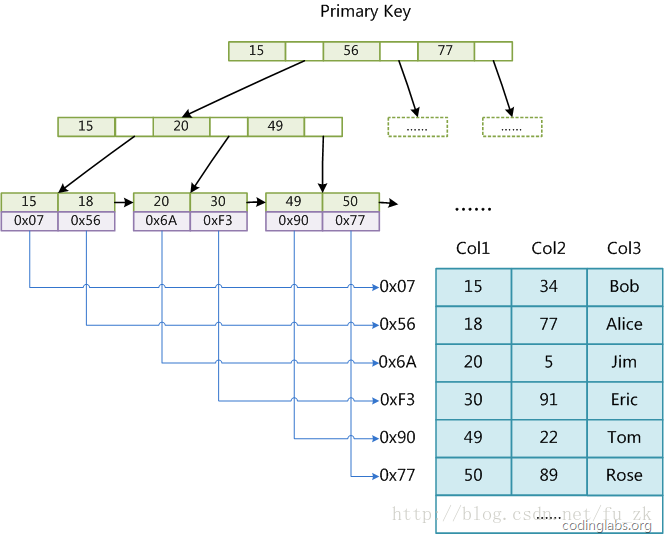
InnoDB（聚簇索引）的数据文件本身就是索引文件（索引和数据存放在一个文件idb）。从上文知道，MyISAM（非聚簇索引）索引文件（MYI）和数据文件（MYD）是分离的，索引文件仅保存数据记录的地址。

mysql中每个表都有一个聚簇索引（clustered index ），除此之外的表上的每个非聚簇索引都是二级索引（普通索引、唯一索引），又叫辅助索引（secondary indexes）。

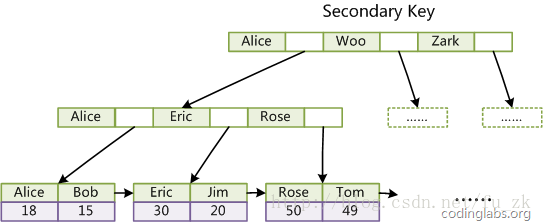
**实现区别**

MyISAM引擎使用B+Tree作为索引结构，叶结点的data域存放的是数据记录的地址。MyISAM的索引方式也叫做“非聚集”的。

MyISAM左图为主索引，右图为辅助索引（二级索引），两者在结构上没什么区别，都是B+树。



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。



InnoDB左图为主索引、右图为辅助索引，辅助索引结构也是B+树

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，这棵树的叶结点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。

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。

这里以英文字符的ASCII码作为比较准则（排序）。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。

由于实际的数据页只能按照一颗B+树进行排序，因此每张表只能有一个聚集索引。

**联合索引与覆盖索引**

联合索引又叫复合索引。对于复合索引:Mysql从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。例如索引是key index (a,b,c). 可以支持a | a,b| a,b,c 3种组合进行查找，但不支持 b,c进行查找 .当最左侧字段是常量引用时，索引就十分有效。符合最左原则

联合索引实现：每个节点含有多个关键字，排序时按照多个关键字的顺序进行排序。而这个顺序就是你创建索引时候的顺序

如果你经常要用到多个字段的多条件查询，可以考虑建立联合索引，建立了一个联合索引就相当于建立了多个索引

联合索引sql会先过滤出last\_name符合条件的记录，在其基础上再过滤first\_name符合条件的记录。那如果我们分别在last\_name和first\_name上创建两个列索引，mysql的处理方式就不一样了，它会选择一个最严格的索引来进行检索，可以理解为检索能力最强的那个索引来检索，另外一个利用不上了，这样效果就不如多列索引了。虽然此时有了两个单列索引，但 MySQL 只能用到其中的那个它认为似乎是最有效率的单列索引。如果经常使用单独一列作为查询条件，那么应该使用单列索引。（如有两个单列索引a、b，查询的时候只用a或只用b）。

多列建索引比对每个列分别建索引更有优势，因为索引建立得越多就越占磁盘空间，在更新数据的时候速度会更慢。另外建立多列索引时，顺序也是需要注意的，应该将严格的索引放在前面，这样筛选的力度会更大，效率更高。

覆盖索引一定是这个联合索引覆盖了你所有select中需要的数据！就是select的数据列只用从索引中就能够取得，不必从数据表中读取，换句话说查询列要被所使用的索引覆盖。

覆盖索引对于innodb表特别的有用，因为innodb是聚集缓存。innodb的secondary index在叶子节点保存了主键的值，因此，覆盖了查询的第二索引，在主键中避免了另外一次索引查找。

覆盖索引和联合索引基本没有区别，覆盖索引只是特定于具体select语录而言的联合索引。也就是说一个联合索引对于某个select语句，通过索引可以直接获取查询结果，而不再需要回表查询啦，就称该联合索引覆盖了这条select语句。

索引覆盖是指建索引的字段正好是覆盖查询条件中所涉及的字段,这里需要注意的是,必须是从第一个开始覆盖,比如:

| 索引字段 | 条件字段 | 有没有覆盖 |
| --- | --- | --- |
| a,b,c | a,b | 覆盖了 |
| a,b,c | b,c | 没有被覆盖 |

第一行满足，第二行不满足

**SQL执行顺序**

from--where--group by--having--select--order by

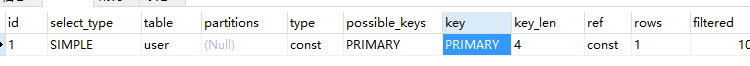
**MySQL中查看索引**

SHOW INDEX FROM tb\_user或SHOW KEYS FROM tb\_user

查看SQL语句的索引使用情况

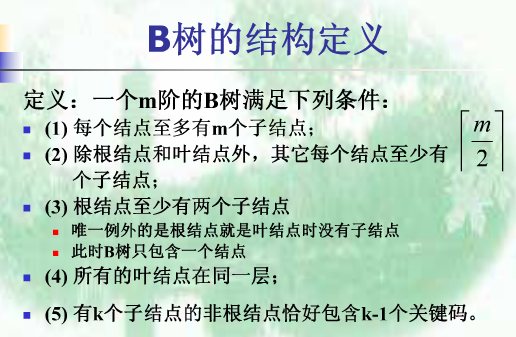
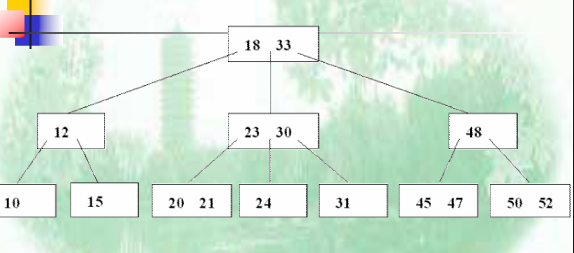
使用“执行过程”EXPLAIN,查看索引使用情况，查询的覆盖行数等

EXPLAIN SELECT \* FROM user WHERE id = 10



**B树和红****黑树**

B树主要是保证只有少数的磁盘访问（io次数少），解决数据结构不在主存中的数据存储问题。高度低。某一个节点可以看做一个磁盘块，里面含有指向下一个磁盘块的指针。

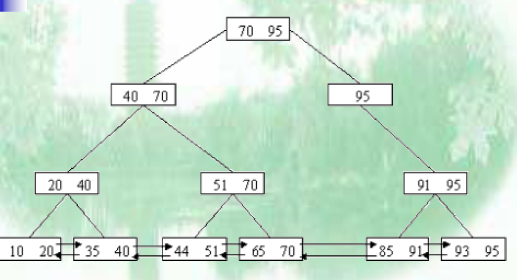
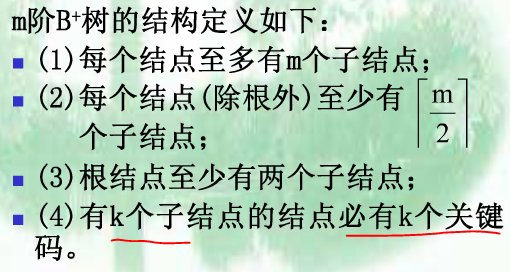
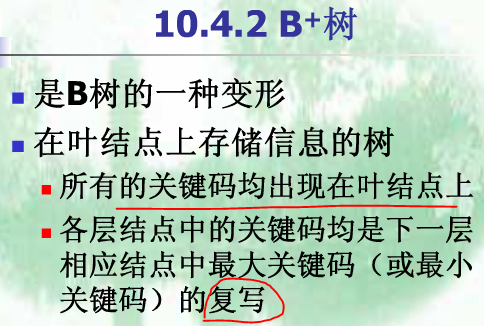


B树从最后一层开始插入，涉及到节点的分裂

一棵含n个结点的B树的高度也为O（logn），但可能比一棵红黑树的高度小许多，应为它的分支因子比较大。所以，B树可以在O（logn）时间内，实现各种如插入（insert），删除（delete）等动态集合操作。

“阶”定义为一个节点的子节点数目的最大值（非根节点关键字个数m/2向上取整-m-1个）

**B+树**



只有叶节点存数据，非叶节点都只是下层节点最大值的复写。叶子节点间多了指针，使得范围查找变得高效（如上图查找20到65）



性质1. 节点是红色或黑色。

性质2. 根是黑色。

性质3. 所有叶子都是黑色（叶子是NIL节点，空节点）。

性质4. 每个红色节点的两个子节点都是黑色。

性质5. 从任一节点到其每个叶子的所有简单路径 都包含相同数目的黑色节点。

在进行红黑树的构造的时候，为了满足第5点，则必须每次插入的节点颜色预设为红色，插入后，有可能会导致4不满足，然后进行节点调整。所以如果是构造出来的，一般来说，不会有节点全黑的红黑树

查找、插入、删除等操作的时间复杂度为O（logn）, 且最多旋转三次

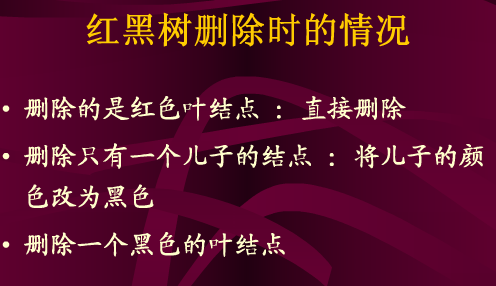
红黑是用非严格的平衡来换取增删节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。所以红黑树的插入效率更高！！！

不能有连续2个的红节点，红黑树也是二叉查找树

红黑树的平衡性：最差情况，一棵子树全是黑色，一棵子树一红一黑····，高度最多差一倍（保证黑色数量相同）

红黑树插入时，插入节点的叔叔是黑节点：也是类似于avl树中的单旋转，双旋转，并改变着色；插入节点的叔叔是红节点：直接重新着色，并再继续向上调整

红黑树某一结点，如果只有一个儿子（如左），那么该结点为黑结点且儿子为红结点，若该结点为红结点，则朝右结点（没有）方向没有黑结点，朝左结点方向有一个黑结点，导致不平衡



删除一个黑色叶结点时需要较复杂的调整，（删除非叶结点情况，可以从叶结点找一个跟该结点替换，从而变成删除叶结点的情况）

**数据库的悲观锁和乐观锁**

一、悲观锁

1、排它锁，当事务在操作数据时把这部分数据进行锁定，直到操作完毕后再解锁，其他事务操作才可操作该部分数据。这将防止其他进程读取或修改表中的数据。

2、实现：大多数情况下依靠数据库的锁机制实现

二、乐观锁

1、如果有人在你之前更新了，你的更新应当是被拒绝的，可以让用户重新操作。

2、实现：大多数基于数据版本（Version）记录机制实现

具体可通过给表加一个版本号或时间戳字段实现，当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断当前版本信息与第一次取出来的版本值大小，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据，拒绝更新，让用户重新操作。

并发量很高的话，建议使用悲观锁，否则的话就使用乐观锁

**MySQL页锁**

页级:引擎 BDB。

表级:引擎 MyISAM ， 理解为锁住整个表，可以同时读，写不行

行级:引擎 INNODB ， 单独的一行记录加锁

表级，直接锁定整张表，在你锁定期间，其它进程无法对该表进行写操作。如果你是写锁，则其它进程则读也不允许

行级，仅对指定的记录进行加锁，这样其它进程还是可以对同一个表中的其它记录进行操作。

页级，表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

**MySQL意向锁**

①在MySQL中有表锁，

LOCK TABLE my\_tabl\_name READ; 用读锁锁表，会阻塞其他事务修改表数据。

LOCK TABLE my\_table\_name WRITe; 用写锁锁表，会阻塞其他事务读和写。

②Innodb引擎又支持行锁，行锁分为

共享锁（读锁），一个事务对一行的共享只读锁。

排它锁，一个事务对一行的排他读写锁。

③这两中类型的锁共存的问题

考虑这个例子：

事务A锁住了表中的一行，让这一行只能读，不能写。

之后，事务B申请整个表的写锁。

如果事务B申请成功，那么理论上它就能修改表中的任意一行，这与A持有的行锁是冲突的。

数据库需要避免这种冲突，就是说要让B的申请被阻塞，直到A释放了行锁。

数据库要怎么判断这个冲突呢？

step1：判断表是否已被其他事务用表锁锁表

step2：判断表中的每一行是否已被行锁锁住。

注意step2，这样的判断方法效率实在不高，因为需要遍历整个表。

于是就有了意向锁。

在意向锁存在的情况下，事务A必须先申请表的意向共享锁，成功后再申请一行的行锁。

在意向锁存在的情况下，上面的判断可以改成

step1：不变

step2：发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务B申请表的写锁会被阻塞。

意向锁是为了提高封锁子系统的效率。

注意：申请意向锁的动作是数据库完成的，就是说，事务A申请一行的行锁的时候，数据库会自动先开始申请表的意向锁，不需要我们程序员使用代码来申请。

**MySQL中InnoDB的MVCC**

MVCC的原理与copyonwrite类似，全称是Multi-Version Concurrent Control，即多版本并发控制。MVCC允许数据具有多个版本，这个版本可以是时间戳或者是全局递增的事务ID，在同一个时间点，不同的事务看到的数据是不同的。

由于在update操作提交之前，不能影响已有数据的一致性，所以不会改变旧的数据，update操作会被拆分成insert + delete。需要标记删除旧的数据，insert新的数据。只有update提交之后，才会影响后续的读操作。而对于读操作而且，只能读到在其之前的所有的写操作，正在执行中的写操作对其是不可见的。（读不加锁，读写不冲突）

innodb会为每一行添加两个字段，分别表示该行创建的版本和删除的版本，填入的是事务的版本号，这个版本号随着事务的创建不断递增。在repeated read的隔离级别下，具体各种数据库操作的实现：

select：满足以下两个条件能够查询成功：（1）该行的创建版本号小于等于当前版本号，用于保证在select操作之前所有的操作已经执行落地。（2）该行的删除版本号大于当前版本或者为空。删除版本号大于当前版本意味着有一个并发事务将该行删除了。

insert：将新插入的行的创建版本号设置为当前系统的版本号。

delete：将要删除的行的删除版本号设置为当前系统的版本号。

update：不执行原地update（相当于写操作），而是转换成insert + delete。将旧行的删除版本号设置为当前版本号，并将新行insert同时设置创建版本号为当前版本号。

其中，写操作（insert、delete和update）执行时，需要将系统版本号递增。

由于旧数据并不真正的删除，所以必须对这些数据进行清理，innodb会开启一个后台线程执行清理工作，具体的规则是将删除版本号小于当前系统版本的行删除，这个过程叫做purge。

通过MVCC很好的实现了事务的隔离性，可以达到repeated read级别，要实现serializable还必须加锁。

**MySQL 5.7新特性**

MySQL 5.7版本提供了更为简单SSL安全访问配置，并且默认连接就采用SSL的加密方式

MySQL数据库从5.7.8版本开始，也提供了对JSON的支持。

generated column是MySQL 5.7引入的新特性，所谓generated column，就是数据库中这一列由其他列计算而得。

不支持子查询中排序

**Redis**

redis单条命令是原子性的，多条命令也可以采用事务（incr、decr自增、自减操作）

基本数据结构

STRING字符串、LIST列表、SET集合、HASH散列、ZSET有序集合

STRING字符串

redis的字符串和其他编程语言中的字符串非常相似，也是key-value的。但是一个key只对应一个value，重复set相同的key会替换。可以存储字节串、整数、浮点数。

GET 获取

SET 存储

DEL 删除

APPEND 将value追加到原value后

GETRANGE 获取在value中截取一段

LIST列表

一个key可以对应多个value，是有序的。像链表一样

RPUSH/LPUSH 将一个或多个值推入列表右端/左端

RPOP/LPOP 移除并返回列表最右/左的元素（从左端push进去，再从左端取出，就会逆序），删除元素只能从一端删

LRANGE 返回从start到end间的value，包含start和end

BLPOP/BRPOP 从第一个非空列表中弹出位于最左/右端的元素，或者在timeout秒之内阻塞并等待可弹出的元素出现。如果timeout是0，则一直阻塞。

SET集合

同一key的多个value不能重复，通过散列表实现。set是无序的方式存储的，不能像列表那样，将元素推入集合的某一端。

SADD 将一个或多个元素添加到集合里

SREM 删除一个或多个元素(srem key value)

SISMEMBER 检查元素item是否存在于集合中

SCARD 返回集合包含的元素数量

SMEMBERS 返回集合包含的所有元素

SPOP 从集合中随机弹出一个元素

SDIFF 返回存在第一个集合，但不存在另外集合中的元素（差集运算）

SINTER 交集

SUNION 并集

HASH散列（应用场景：购物车，一个用户key，里面有购买的物品名称key，数量value）

在一个key里面，value中可以包含许多个子key、value；各个subkey是无序的。重复设置同一个subkey会覆盖

HSET/HMGET 设置/设置多个

HGET/HMSET 获取/获取多个

HDEL 删除

HLEN 返回一个key里面包含的subkey、value数量

HKEYS 获取一个key里包含的所有键subkey

HVALS 获取一个key里包含的所有值

HEXISTS 检查key里的subkey是否存在

ZSET有序集合

一个key中存储了多个member，每个member都有score

ZADD key-name score member 添加一个或多个

ZREM key-name member 删除一个或多个

ZCARD key-name 返回包含的成员数量

ZINCRBY key-name increment member 将成员的分值加上increment

ZCOUNT key-name min max 返回分值介于min和max之间的成员数量，包含min、max

ZRANK/ZREVRANK key-name member 返回member的排名，最高为0名;rev为从大到小

ZSCORE key-name member 返回分值

ZRANGE/ZREVRANGE key-name start stop 返回排名介于start和stop之间的成员，包含start、stop；;rev为从大到小

ZRANGEBYSCORE/ZREVRANGEBYSCORE key min max获取分值介于min和max之间的成员

ZREMRANGEBYRANK key-name start stop移除排名介于start和stop

ZREMRANGEBYSCORE key-name min max 移除分数介于min到max

ZINTERSTORE/ZUNIONSTORE 交集，并集

持久化

redis的持久化有rdb和aof两种。

rdb是记录一段时间内的操作，一般的配置是一段时间内操作超过多少次就持久化。

aof可以实现每次操作都持久化。

1、 快照的方式持久化到磁盘

自动持久化规则配置

save 900 1

save 300 10

save 60 10000

上面的配置规则意思 如下:

# In the example below the behaviour will be to save:

# after 900 sec (15 min) if at least 1 key changed

# after 300 sec (5 min) if at least 10 keys changed

# after 60 sec if at least 10000 keys changed

• RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。如果系统在某时刻崩溃，将丢失在上次生成快照到崩溃时之间的数据。（快照持久化只适用于那些即使丢失一部分数据也不会造成问题的应用程序）

• AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。

• Redis 还可以同时使用 AOF 持久化和 RDB 持久化。 在这种情况下， 当 Redis 重启时， 它会优先使用 AOF 文件来还原数据集， 因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。

• 你甚至可以关闭持久化功能，让数据只在服务器运行时存在。

aof：每个redis的命令都会被写入硬盘，从而将发生系统崩溃时出现的数据丢失减到最少，但是需要对磁盘进行大量写入。aof将数据丢失时间窗口降低至1秒（甚至不丢失任何数据），但重启时，如果aof文件体积太大，还原操作会执行很长时间。（可以压缩aof文件）。aof配置参数（always：每个写命令都同步写入硬盘，everysec（常用）：每秒执行一次同步，显示将多个写命令同步到硬盘，no：让操作系统决定应该何时进行同步）

AOF的Rewrite(重写) ：

　　定义：AOF采用文件追加的方式持久化数据，所以文件会越来越大，为了避免这种情况发生，增加了重写机制。当AOF文件的大小超过了配置所设置的阙值时，Redis就会启动AOF文件压缩，只保留可以恢复数据的最小指令集，可以使用命令bgrewriteaof

RDB与AOF的选择：

　　做备份：当数据量大，且对恢复速度有要求，并且数据的一致性要求不高的话，可以只使用RDB

　　做缓存：不用开启任何的持久化方式

　　两者都开启的建议：RDB数据不实时，同时使用两者时服务器只会找AOF文件，可不可以只使用AOF?作者建议不要，因为RDB更适合备份数据库(AOF在不断变化，不好备份)

事务

redis的事务和传统关系数据库的事务并不相同。在关系数据库中，用户首先向数据库服务发送begin，然后执行各个相互一致的写操作和读操作，最后，用户可以选择发送commit来确认修改，或者发送rollback来放弃那些修改。

redis的事务以multi为开始，之后跟着用户传入的多个命令，最后以exec为结束。但是由于这种简单的事务在exec命令被调用之前不会执行任何实际操作，所以用户将没办法根据读取到的数据来做决定。（在一个事务执行过程中的数据）

并发问题

redis本身是单进程单线程模型，命令是一条接一条执行的，本身不会发生并发问题。但如果有多个链接连到同一个redis上，就会产生问题。

redis命令incr/incrby是原子性的（相对于先GET在+1，再SET来说）。

当redis从一个客户端那里接收到multi命令时，redis会将这个客户端之后发送的所有命令都放入到一个队列里面，直到这个客户端发送exec命令为止（接收到exec后才开始执行），然后redis就会在不被打断的情况下，一个接一个地执行存储在队列里面的命令。这种做法称为流水线，它可以通过减少客户端与redis服务器之间的网络通信次数来提升redis在执行多个命令时的性能。

Redis 在事务失败时不进行回滚，而是继续执行余下的命令，当一个事务执行完毕后，redis才会处理其他客户端的命令。

Redis 命令只会因为错误的语法而失败（并且正确的命令也不会执行，在事务中），如set写成sett不会执行事务中其他代码，而如果get一个散列表的值（是运行时的错误），还会执行事务中的其他代码。

为什么不支持回滚？这种错误可能会导致Redis命令执行失败，是由于程序员自身问题导致。正因为这些程序错误不大可能会进入生产环境，所以我们在开发Redis时选用更加简单和快速的方法，没有实现错误回滚的功能。

Redis 本身提供的所有 API 都是原子操作，那么 Redis 事务其实是要保证批量操作的原子性。Redis 实现批量操作的原理是在一个事务上下文中（通过 MULTI命令开启），所有提交的操作请求都先被放入队列中缓存，在 EXEC 命令提交时一次性批量执行。这样保证了批量操作的一次性执行过程，但 Redis 在事务执行过程的错误情况做出了权衡取舍，那就是放弃了回滚（DISCARD可以人为取消事务）。放弃了原子性（原子性：要么全部执行，要么全部不执行；redis没有回滚操作）

可以通过watch实现原子性（watch只会在数据被其他客户端抢先修改了的情况下通知执行了这个命令的客户端，而不会阻止其他客户端对数据进行修改，所以是乐观锁）

在Redis的事务中，WATCH命令（相当于乐观锁，传统数据库执行加锁的方式称为悲观锁）可用于提供CAS(check-and-set)功能。假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，EXEC命令执行的事务都将被放弃，同时返回Null multi-bulk应答以通知调用者事务执行失败。例如，我们再次假设Redis中并未提供incr命令来完成键值的原子性递增，如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：

val = GET mykey

val = val + 1

SET mykey $val

以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景--竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，这样就会导致mykey的结果为11，而不是我们认为的12。为了解决类似的问题，我们需要借助WATCH命令的帮助，见如下代码：

WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC

和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

隔离性，redis是单进程，开启事务之后，会执行完当前连接的所有命令直到遇到exec命令，才处理其他连接的命令。

1. 单独的隔离操作：事务中的所有命令会被序列化、按顺序执行，在执行的过程中不会被其他客户端发送来的命令打断

2. 没有隔离级别的概念：队列中的命令在事务没有被提交之前不会被实际执行

3. 不保证原子性：redis中的一个事务中如果存在命令执行失败，那么其他命令依然会被执行，没有回滚机制

在客户端将连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。

服务器角度，利用setnx变向实现锁机制。

**范式**

**第一范式1NF的定义为：符合1NF的关系中的每个属性都不可再分，1NF是所有关系型数据库的最基本要。**

**2NF在1NF的基础之上，消除了非主属性对于码的****部分函数依赖**。

**码**：设 K 为某表中的一个属性或属性组，若除 K 之外的所有属性都完全函数依赖于 K（这个“完全”不要漏了），那么我们称 K 为**候选码**，简称为**码**。在实际中我们通常可以理解为：**假如当 K 确定的情况下，该表除 K 之外的所有属性的值也就随之确定，那么 K 就是码。**一张表中可以有超过一个码。

部分函数依赖：依赖于码的一部分（其中的某几个）  
**非主属性**：包含在任何一个码中的属性成为主属性。

**3NF在2NF的基础之上，消除了非主属性对于码的传递函数依赖**。

BCNF范式：在 3NF 的基础上消除**主属性**对于码的部分与传递函数依赖

<https://www.zhihu.com/question/24696366>

反范式是通过增加冗余数据或数据分组来提高数据库读性能的过程。在某些情况下， 反范式有助于掩盖关系型数据库软件的低效。关系型的范式数据库即使做过优化， 也常常会带来沉重的访问负载。

**jdbc参数**



加载驱动，创建连接

**数据库系统的设计步骤**

**数据库设计的过程(六个阶段)**

　 1.需求分析阶段

　　　准确了解与分析用户需求（包括数据与处理）

　　　是整个设计过程的基础，是最困难、最耗费时间的一步

　　2.概念结构设计阶段

　　　是整个数据库设计的关键

　　　通过对用户需求进行综合、归纳与抽象，形成一个独立于具体DBMS的概念模型（DBMS数据库管理系统，如MySQL）

　　3.逻辑结构设计阶段

　　　将概念结构转换为某个DBMS所支持的数据模型，对其进行优化

　　4.数据库物理设计阶段

　　　为逻辑数据模型选取一个最适合应用环境的物理结构（包括存储结构和存取方法）

　　5.数据库实施阶段

　　　运用DBMS提供的数据语言、工具及宿主语言，根据逻辑设计和物理设计的结果

　　　建立数据库，编制与调试应用程序，组织数据入库，并进行试运行

　　6.数据库运行和维护阶段

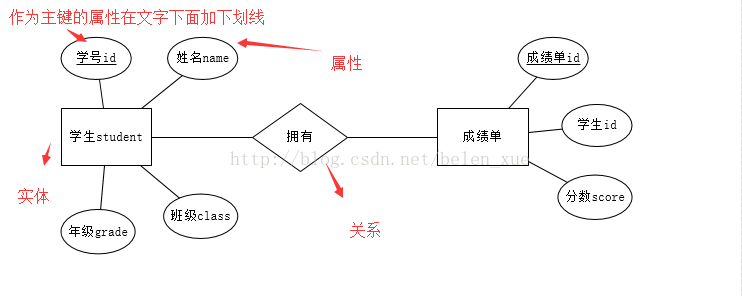
　　　数据库应用系统经过试运行后即可投入正式运行。

　　　在数据库系统运行过程中必须不断地对其进行评价、调整与修改

**数据库各级模式的形成过程**

　　1.需求分析阶段：综合各个用户的应用需求

2.概念设计阶段：形成独立于机器特点，独立于各个DBMS产品的概念模式(E-R图，实体-联系图)



　　3.逻辑设计阶段：首先将E-R图转换成具体的数据库产品支持的数据模型，如关系模型，形成数据库逻辑模式；然后根据用户处理的要求、安全性的考虑，在基本表的基础上再建立必要的视图(View)，形成数据的外模式（将E-R转换成表，根据范式设计表，表结构设计尽量简单、小）

　　4.物理设计阶段：根据DBMS特点和处理的需要，进行物理存储安排，建立索引，形成数据库内模式

**网络**

**T****CP/UDP**

TCP（Transmission Control Protocol，传输控制协议）是面向连接的协议，也就是说，在收发数据前，必须和对方建立可靠的连接。UDP是一个非连接的协议，传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快地把它扔到网络上。在发送端，UDP传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制；

小结TCP与UDP的区别：

1.tcp面向连接与udp面向无连接；

2.对系统资源的要求（TCP较多，UDP少）；

3.UDP程序结构较简单；

4.流模式与数据报模式 ；

5.TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证。

TCP提供面向连接的、可靠的数据流传输，而UDP提供的是非面向连接的、不可靠的数据流传输。

TCP传输单位称为TCP报文段，UDP传输单位称为用户数据报。

TCP注重数据安全性，UDP数据传输快，因为不需要连接等待，少了许多操作，但是其安全性却一般。

滑动窗口（流量控制），拥塞控制，超时重传，序号确认机制

TCP三次握手和四次挥手的全过程

答:三次握手：

第一次握手：客户端发送syn包(syn=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syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（syn=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(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。

四次挥手（客户端或服务器均可主动发起挥手动作）

第一次挥手：主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭方告诉被动关闭方：我已经不 会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可 以接受数据。

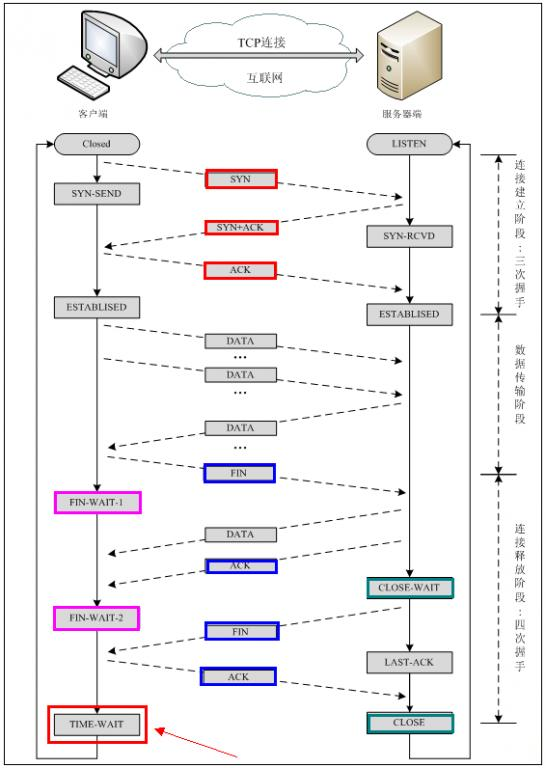
第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号）。

第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了。

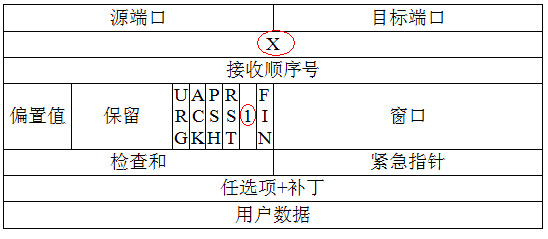
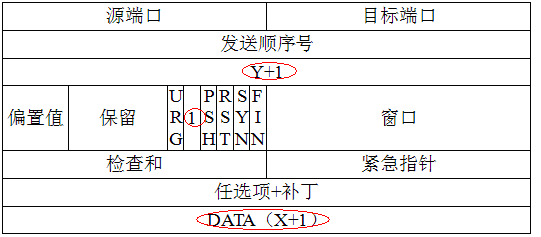
第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手。

请求建立连接syn置1，Syn消耗一个序列号，ack不需要消耗序列号

Fin请求断开连接，Ack确认号，期望接受的下一个序号



图中established写错了

1．为什么建立连接协议是三次握手，而关闭连接却是四次握手呢？   
这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起应答作用，而SYN起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的FIN报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的。 服务器如果SYN\_RECV后没有收到ACK，会重发SYN+ACK  
2．为什么TIME\_WAIT状态还需要等2MSL（报文最长存活时间）后才能返回到CLOSED状态？   
这是因为虽然双方都同意关闭连接了，而且握手的4个报文也都协调和发送完毕，按理可以直接回到CLOSED状态（就好比从SYN\_SEND状态到 ESTABLISH状态那样）；但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的ACK报文会一定被对方收到，因此对方处于 LAST\_ACK状态下的SOCKET可能会因为超时未收到ACK报文，而重发FIN报文，所以这个TIME\_WAIT状态的作用就是用来重发可能丢失的 ACK报文  
第一次握手:  
客户端发送一个TCP的SYN标志位置1的包指明客户打算连接的服务器的端口，以及初始序号X,保存在包头的序列号(Sequence Number)字段里。  
  
第三次握手.  
客户端再次发送确认包(ACK) SYN标志位为0,ACK标志位为1.并且把服务器发来ACK的序号字段+1,放在确定字段中发送给对方.并且在数据段放写ISN的+1  


**SYN攻击**

在三次握手过程中，服务器发送SYN-ACK之后，收到客户端的ACK之前的TCP连接称为半连接(half-open connect).此时服务器处于Syn\_RECV状态.当收到ACK后，服务器转入ESTABLISHED状态.

Syn攻击就是 攻击客户端 在短时间内伪造大量不存在的IP地址，向服务器不断地发送syn包，服务器回复确认包，并等待客户的确认，由于源地址是不存在的，服务器需要不断的重发直 至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，目标系统运行缓慢，严重者引起网络堵塞甚至系统瘫痪。

Syn攻击是一个典型的DDOS攻击。检测SYN攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击.在Linux下可以如下命令检测是否被Syn攻击

**netstat -n -p TCP | grep SYN\_RECV**

一般较新的TCP/IP协议栈都对这一过程进行修正来防范Syn攻击，修改tcp协议实现。主要方法有SynAttackProtect保护机制、SYN cookies技术、增加最大半连接和缩短超时时间等.但是不能完全防范syn攻击。

**Tcp可靠传输**

1. 序号/确认机制
2. 超时重传机制
3. 定时器
4. 重传定时器
5. 持续定时器
6. 保活定时器

4、检查校验和

**Tcp流量控制**

Tcp采用大小可变滑动窗口的方式进行流量控制。窗口大小单位是字节。根据接收方接收能力，通过接收窗口rwnd可以实现端到端的流量控制，接收端将接收窗口rwnd的值放在tcp报文首部的“窗口”字段，传送给发送端。在通信过程中，接收端可以根据自己资源情况，随时动态调整接收窗口（在tcp报文首部的“窗口”字段，如果win为0，则不能发送），然后告诉发送方，使发送方的发送窗口和自己的接收窗口一致。这是由接收端控制发送端的做法。tcp累计确认，ack一次确认之前发送的好几段

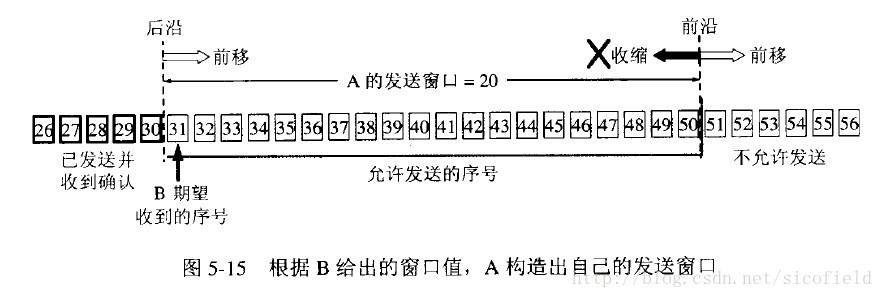
一般说来，我们总是希望数据传输的更快一些。但如果发送方吧数据发送的过快，接收方就可能来不及接收，就会造成数据的丢失。所谓的流量控制就是让发送方的发送速率不要太快，要让接收方来的及接收。

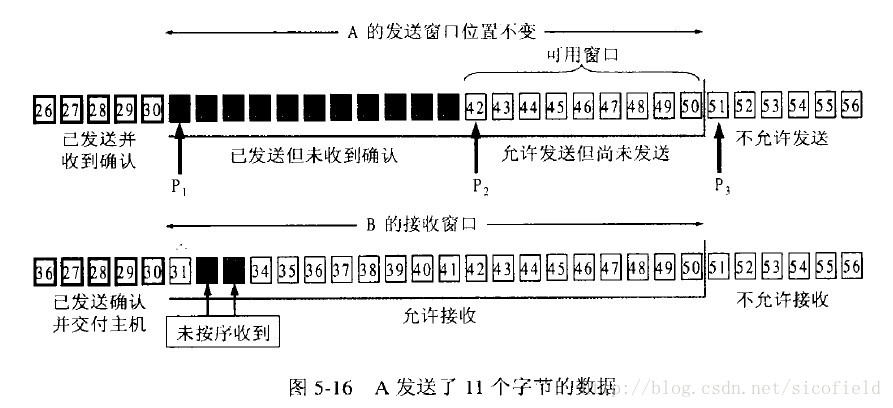


设A向B发送数据。在连接建立是，B告诉了A：“我的接收窗口wnd=400”。因此，发送方的发送窗口不能超过接收方的给出的接收窗口的数值。请注意，TCP的窗口单位是字节，不是报文段。我们应注意，接收方的主机B进行了三次流量控制。第一次把窗口减小到rwnd=300，第二次又减到rwnd=100，最后减到了rwnd=0，即不允许发送方再发送数据了。这种使发送方暂停发送的状态将持续到主机B重新发出一个新的窗口值为止。我们还应注意到，B向A发送的三个报文段都设置了ACK=1，只有在ACK=1时确认号字段才有意义。

现在我们考虑一种情况，B向A发送了零窗口的报文段后不久，B的接收缓存又有了一些存储空间。于是B向A发送了rwnd=400的报文段。然而这个报文段在传送过程中丢失了。A一直等待收到B发送的非零窗口的通知，而B也一直等待A发送的数据，如果没有其他措施，这种互相等待的死锁局面将一直延续下去。

为了解决这个问题，TCP为每一个连接设有一个持续计时器，只要TCP连接的一方收到对方的零窗口通知，就启动持续计时器。若持续计时器设置的时间到期，就发送一个零窗口探测报文段，仅携带一个字节的数据，而对方就在确认这个探测报文段时会给出现在的窗口值。如果窗口值仍为0，那么收到这个报文段的一方就重新设置持续计时器。如果窗口值不为0，那么死锁的僵局就可以打破了。





**Tcp拥塞控制**

拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制是一个全局性的过程，和流量控制不同，流量控制指点对点通信量的控制。

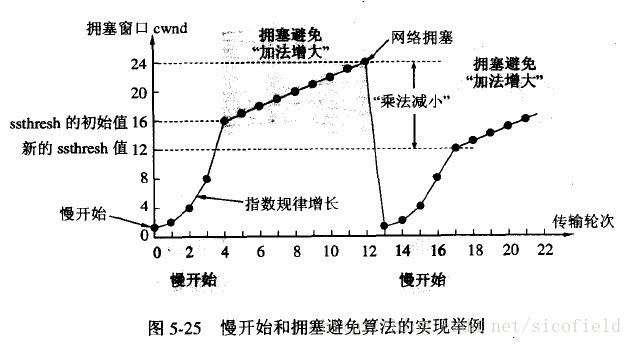
TCP为了进行有效的拥塞控制，需要通过拥塞窗口cwnd来进行衡量网络的拥塞程度。发送窗口的取值是依据拥塞窗口和接收窗口中的较小值min(rwnd,cwnd)

tcp拥塞控制四种算法：

1. 慢启动：在TCP刚建立连接或网络发送拥塞时，将拥塞窗口设置成一个报文段大小，当cwnd小于等于ssthresh（门限值）时，以指数方式增大cwnd（乘以2）；
2. 拥塞避免：当cwnd大于等于ssthresh时，为避免网络发送拥塞，以线性方式增大cwnd（cwnd加1）；
3. 快速重传：如果发送方连续收到3个重复确认的ack，则立即重传该报文段，不必等待重传定时器超时后重传；
4. 快速恢复：当采用快速重传算法时，直接执行拥塞避免算法。以提高传输效率

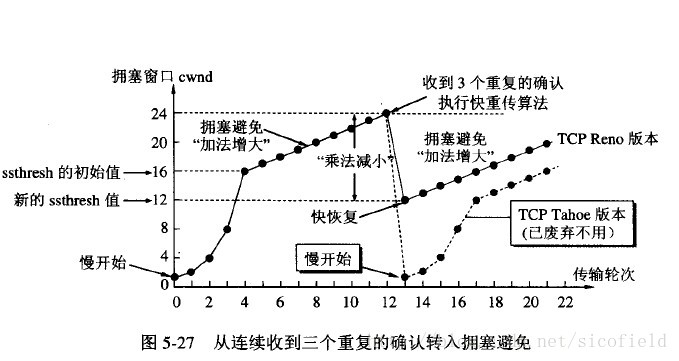
无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把门限设置为**出现拥塞时的发送窗口**大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。

再次提醒这里只是为了讨论方便而将拥塞窗口大小的单位改为数据报的个数，实际上应当是字节。



1. 当发送方连续收到三个重复确认时，把ssthresh门限减半。但是接下去并不执行慢开始算法。

②考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，不将cwnd置为1，而是将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。



**Http请求与响应**

**get请求头**

GET /books/?name=Professional%20Ajax HTTP/1.1

Host: www.baidu.com

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)

Connection: Keep-Alive

请求体是空行

**post请求头**

POST / HTTP/1.1

Host: www.baidu.com

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)

Gecko/20050225 Firefox/1.0.1

Content-Type: application/x-www-form-urlencoded

Content-Length: 40

Connection: Keep-Alive

name=Professional%20Ajax&publisher=Wiley

Keep-Alive方式，即数据传递完并不立即关闭连接。可以设置keepalive\_timeout，时间超限后关闭连接

头中还有cookie

**响应**

HTTP/1.1 200 OK

Date: Sat, 31 Dec 2005 23:59:59 GMT

Content-Type: text/html;charset=ISO-8859-1

Content-Length: 122

<html>

<head>

<title>Wrox Homepage</title>

</head>

<body>

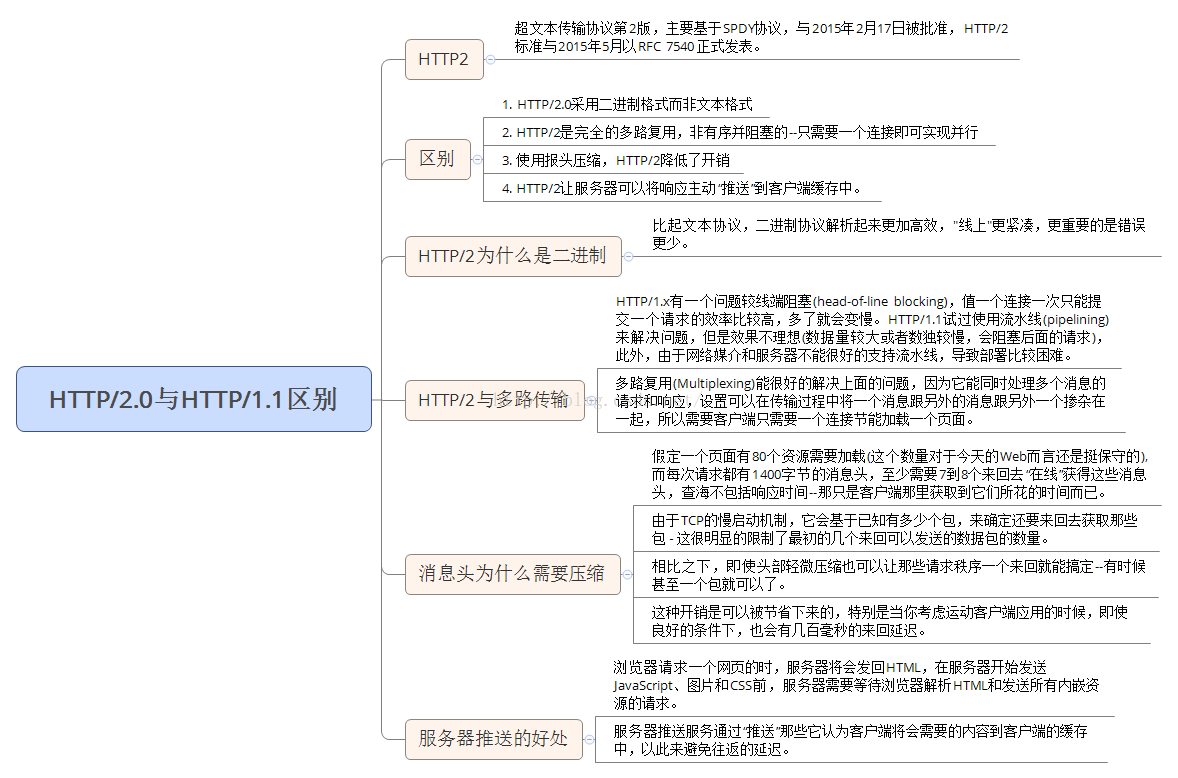
<!-- body goes here -->

</body>

</html>

**HTTP2与1.1区别**

1. HTTP2.0采用二进制格式而非文本格式（二进制解析更加高效，错误更少）
2. HTTP2是完全的多路复用（能同时传输多个请求和响应）
3. 使用报头压缩，降低了开销
4. 可以让服务器主动把响应推送到客户端



**HTTP1.0与1.1区别**

HTTP 1.0规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求。但是，这也造成了一些性能上的缺陷，例如，一个包含有许多图像的网页文件中并没有包含真正的图像数据内容，而只是指明了这些图像的URL地址，当WEB浏览器访问这个网页文件时，浏览器首先要发出针对该网页文件的请求，当浏览器解析WEB服务器返回的该网页文档中的HTML内容时，发现其中的<img>图像标签后，浏览器将根据<img>标签中的src属性所指定的URL地址再次向服务器发出下载图像数据的请求

HTTP 1.1支持持久连接（connection默认keep-alive），在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟。一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。 在HTTP1.0和HTTP1.1协议中都有对长连接的支持。其中HTTP1.0需要在request中增加”Connection： keep-alive“ header才能够支持，而HTTP1.1默认支持.

HTTP 1.1还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求（流水线处理），但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间。

HTTP 1.0不支持Host请求头字段，WEB浏览器无法使用主机头名来明确表示要访问服务器上的哪个WEB站点，这样就无法使用WEB服务器在同一个IP地址和端口号上配置多个虚拟WEB站点。在HTTP 1.1中增加Host请求头字段后，WEB浏览器可以使用主机头名来明确表示要访问服务器上的哪个WEB站点，这才实现了在一台WEB服务器上可以在同一个IP地址和端口号上使用不同的主机名来创建多个虚拟WEB站点。

HTTP 1.1还提供了与身份认证、状态管理和Cache缓存等机制相关的请求头和响应头。

smtp、pop3属于应用层，基于TCP

**当我们在浏览器的地址栏输入“www.baidu.com”然后按回车，这之后发生了什么事，我们直接看到的是打开了对应的网页，那么内部客户端和服务端是如何通信的呢？**

1、DNS地址解析，获取IP地址（属于应用层协议，基于UDP）

浏览器地址栏中输入"HTTP://www.xxx.com/"并提交之后，首先它会在DNS本地缓存表中查找，如果有则直接告诉IP地址。如果没有则要求网关DNS进行查找，如此下去，找到对应的IP后，则返回会给浏览器。

主机向本地域名服务器的查询一般都是采用递归查询。所谓递归查询就是：如果主机所询问的本地域名服务器不知道被查询的域名的IP地址，那么本地域名服务器就以DNS客户的身份，向其它根域名服务器继续发出查询请求报文(即替主机继续查询)，而不是让主机自己进行下一步查询。因此，递归查询返回的查询结果或者是所要查询的IP地址，或者是报错，表示无法查询到所需的IP地址。

2、获取IP，建立TCP连接

当获取IP之后，就开始与所请求的Tcp建立三次握手连接，连接建立后，就向服务器发出HTTP请求。

3、客户端浏览器向服务器发出HTTP请求

一旦建立了TCP连接，Web浏览器就会向Web服务器发送请求命令，接着以头信息的形式向Web服务器发送一些别的信息，之后浏览器发送了一空白行来通知服务器，它已经结束了该头信息的发送。

4、Web服务器应答，并向浏览器发送数据

客户机向服务器发出请求后，服务器内部处理（如DispatcherServlet分发请求），服务器会客户机回送应答。

5、Web服务器关闭TCP连接

一般情况下，一旦Web服务器向浏览器发送了请求数据，它就要关闭TCP连接，然后如果浏览器或者服务器在其头信息加入了这行代码

Connection:keep-alive

TCP连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

**服务端监听80端口，如何处理http请求**

处理http请求分为7个步骤

**1 Tcp连接**

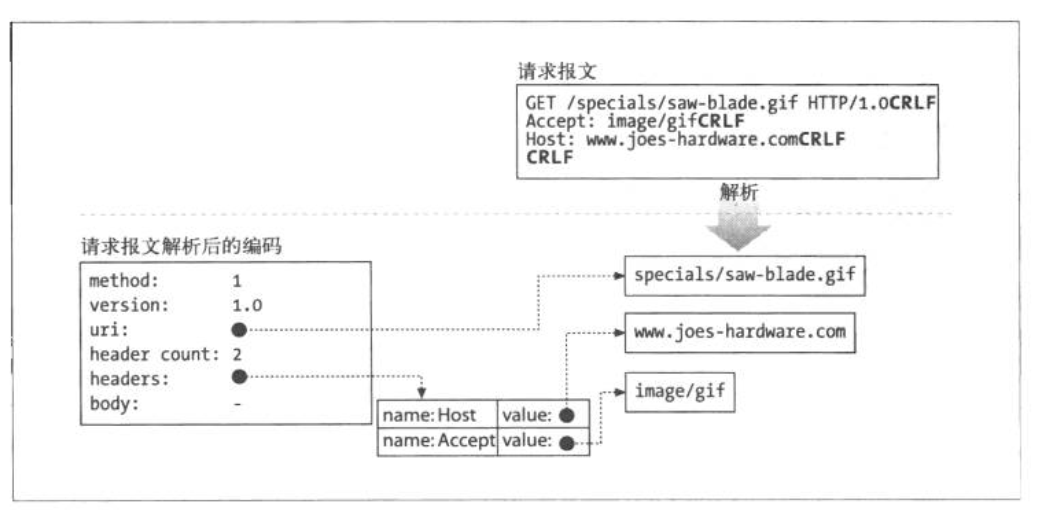
建立一条tcp链接，（若之前不存在持久链接keep-alive），把客户端的ip和port，服务端的ip和port数据放到web服务器连接表中。服务器随时监听链接表中的链接，看有没有数据变化

**2 接收http请求**

一旦我们发送http请求了，这条tcp链接就开始工作了。因为web服务器链接表中有许多链接需要被处理，处理的方式有单线程，多线程这些

**3 处理http请求**

处理的过程大致是把请求的信息解析出来，如下图



**4 访问资源**

访问资源可以是访问静态资源，这个就直接根据url地址去服务器里找就好了。

访问动态资源的话要经过一个叫cgi的东西，再用服务端脚本处理，再返回给前端。

**5 构建响应**

要是找到资源，则构建响应信息，包括响应的对象类型，长度，状态码。

另一个情况是重定向响应，就是直接返回一个重定向，客户端看到之后，立刻再向重定向的地址发起请求。重定向的响应的状态码一般是3xx。

**6 发送响应**

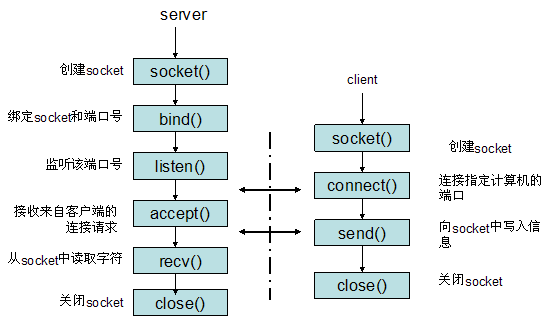
把构建的响应发送给客户端

**7 记录日志**

服务端对这个请求响应过程进行记录。

**socket**

socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。利用ip地址＋协议＋端口号唯一标示网络中的一个进程。服务器socket与客户端socket建立连接的部分其实就是三次握手



**POST与GET**

1. get是从服务器上获取数据，post是向服务器传送数据。

2. get是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。post是通过HTTP post机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址。用户看不到这个过程。post不会作为url的一部分，不会被缓存、保存在服务器日志、以及浏览器浏览记录中.

3. 对于get方式，服务器端用Request.QueryString获取变量的值，对于post方式，服务器端用Request.Form获取提交的数据。

4. get传送的数据量较小，不能大于2KB。post传送的数据量较大，一般被默认为不受限制。但理论上，IIS4中最大量为80KB，IIS5中为100KB。

5. get安全性非常低，post安全性较高。但是执行效率却比Post方法好。

建议：

1、get方式的安全性较Post方式要差些，包含机密信息的话，建议用Post数据提交方式；

2、在做数据查询时，建议用Get方式；而在做数据添加、修改或删除时，建议用Post方式；

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。

**post为什么比get快？**

1.post请求包含更多的请求头

　　因为post需要在请求的body部分包含数据，所以会多了几个数据描述部分的首部字段（如content-type），这其实是微乎其微的

**2.最重要的一条，post在真正接受数据之前会先将请求头发送给服务器进行确认，然后才真正发送数据**

**post请求的过程：**

1.浏览器请求tcp连接（第一次握手）

　　2.服务器答应进行tcp连接（第二次握手）

　　3.浏览器确认，并发送post请求头（第三次握手，这个报文比较小，所以http会在此时进行第一次数据发送）

　　4.服务器返回100 continue响应（先发送头，再发送体）

　　5.浏览器开始发送数据

　　6.服务器返回200 ok响应

**get请求的过程**

1.浏览器请求tcp连接（第一次握手）

　　2.服务器答应进行tcp连接（第二次握手）

　　3.浏览器确认，并发送get请求头和数据（第三次握手，这个报文比较小，所以http会在此时进行第一次数据发送）

　　4.服务器返回200 ok响应

**HTTP与HTTPS**

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。HTTPS使用不同于HTTP协议的默认端口及一个加密、身份验证层（HTTP与TCP之间）

　　HTTPS和HTTP的区别主要如下：

　　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

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

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

　　4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

虽然说HTTPS有很大的优势，但其相对来说，还是存在不足之处的：

　　（1）HTTPS协议握手阶段比较费时，会使页面的加载时间延长近50%，增加10%到20%的耗电；

　　（2）HTTPS连接缓存不如HTTP高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；

　　（3）SSL证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。

　 （4）SSL证书通常需要绑定IP，不能在同一IP上绑定多个域名，IPv4资源不可能支撑这个消耗。

　　（5）HTTPS协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL证书的信用链体系并不安全，特别是在某些国家可以控制CA根证书的情况下，中间人攻击一样可行。

客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤，如图所示。（先非对称，在对称）

　　（1）客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。

　　（2）Web服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

　　（3）客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。

　　（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

　　（5）Web服务器利用自己的私钥解密出会话密钥。

（6）Web服务器利用会话密钥加密与客户端之间的通信。

**HTTP状态码**

**1xx-**信息提示：这些状态代码表示临时的响应。客户端在收到常规响应之前，应准备接收一个或多个1xx响应。

**200**:访问成功（表示一切正常，返回的是正常请求结果）

**201**(已创建)请求成功并且服务器创建了新的资源。

**301** redirect: 301 代表永久性转移(Permanently Moved)

**302**：临时重定向（指出被请求的文档已被临时移动到别处，此文档的新的URL在Location响应头中给出）

301表示旧地址A的资源已经被永久地移除了（这个资源不可访问了），搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址；302表示旧地址A的资源还在（仍然可以访问），这个重定向只是临时地从旧地址A跳转到地址B，搜索引擎会抓取新的内容而保存旧的网址。

**304**：未修改（表示客户机缓存的版本是最新的，客户机应该继续使用它。）

**403** Forbidden(禁止)，Web客户端发送的请求被Web服务器拒绝了（如不允许写入，不允许读取文件）， 如果服务器想说明为什么拒绝请求，可以包含实体的主体部分来对原因进行描述。但这个状态码通常是服务器不想说明拒绝原因。

**404**：访问的文件不存在（服务器上不存在客户机所请求的资源）

**405** method not allowed

问题原因： 请求的方式（get、post、delete）方法与后台规定的方式不符合。

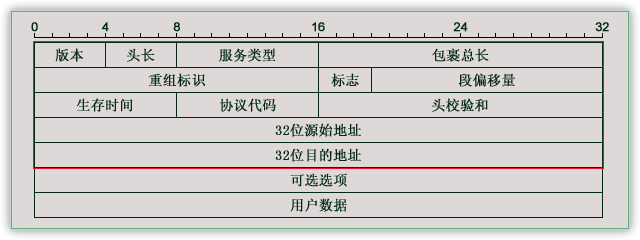
比如： 后台方法规定的请求方式只接受get，如果用post请求，就会出现 405 method not allowed的提示

**500**：内部服务器错误（服务器端的CGI、ASP、JSP等程序发生错误）

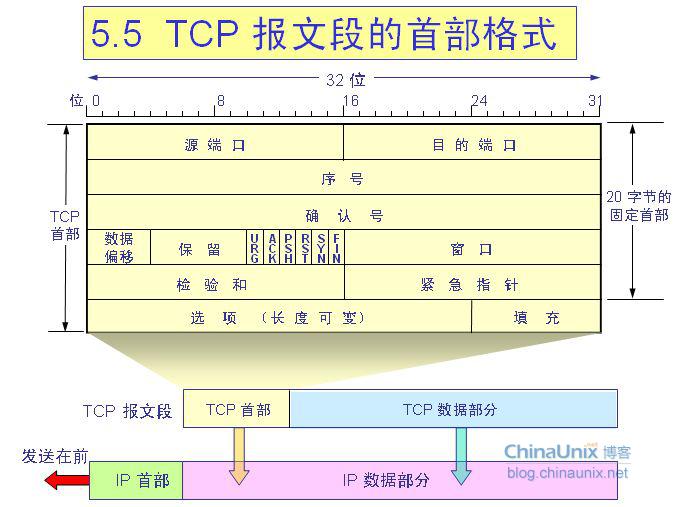
**503**：由于临时的服务器维护或者过载，服务器当前无法处理请求。这个状况是临时的，并且将在一段时间以后恢复。

**IP包、TCP包、HTTP包**

ip包：



tcp包：在ip包（网络层）的数据部分包含tcp包（传输层），tcp包中不含有ip，只有端口



http包就是请求头（体），响应头（体html）

交换机

交换机（数据链路层）独占带宽，指的是交换机的各个端口之间，如某一速率为100M的交换机有20个端口，每个端口间速率为100M，但如果只有一个端口连接互联网，那么网速就不是100M了。一个端口一个冲突域

路由器（网络层）共享带宽，别人下载东西会占用你的网速。

集线器（物理层）共享带宽，只有一个冲突域

在以太网中，如果某个CSMA/CD网络上的两台计算机在同时通信时会发生冲突，那么这个CSMA/CD网络就是一个冲突域

**Linux**

查看服务的命令（你这个命令会出现很多服务，那么怎么找到我要搜索的服务名称）

service --status-all

chkconfig --list（ubuntu中没有）

创建一个文件用什么命令，修改权限使用什么命令，修改所有者使用什么命令

mkdir -p递归创建；rm -rf，-r是递归删除，-f强制删除

touch是创建文件

chmod

chown改变所有者

chgrp改变组

linux的显示文件夹大小 ls -al（l是详细信息，a是列出全部，包括隐藏文件）

查看进程、线程

ps命令查看进程pid等，ps -T -p <pid> “-T”选项可以开启线程查看。不加参数，查看线程

使用top命令，具体用法是 top -H，加上这个选项，top的每一行就不是显示一个进程，而是一个线程。不加参数，查看线程，还可以查看进程占用资源；-p <pid>查看单一进程下的线程；查看cpu使用率

杀死进程kill -9 pid；-9是一个信号参数，表示杀死进程

查看文件100到150行

显示文件头部

head -n 100 filename 显示开始100行

显示文件尾部

tail -n 100 filename 显示最后100行

head -n 5 123.txt | tail -n 2管道连接

查看网络连接状态，查看端口情况

netstat

-a 显示所有socket，包括正在监听的。

-n 以网络IP地址代替名称，显示出网络连接情形。

-t 显示TCP协议的连接情况。

-u 显示UDP协议的连接情况。

统计80端口连接数

netstat -nat|grep -i “80”|wc -l

wc -c 统计字节数。-l 统计行数。

grep -i不区分大小写， grep ‘aaa’ filename//匹配有aaa的行

－c：只输出匹配行的计数。  
－I：不区分大 小写(只适用于单字符)。

sed ‘s/要被替换/新字符串/g’

sed ‘2,5d’ d表示删除

sed ‘2,$d’ $表示最后一行

cat/more/less filename 显示文件内容

cat不能翻页

more和less都是按页显示文档，但less不会一上来就加载所有的文档，都可以使用空格向下翻，b向上翻。不同的是less可以使用上下键一行一行的翻（卷动）

查看磁盘读写吞吐量

iostat

在Linux 中 用 Kill -2 和 Kill -9 都能够结束进程，他们之间的区别为：

Kill -2 ：功能类似于Ctrl + C 是程序在结束之前，能够保存相关数据，然后再退出。

Kill -9 ：直接强制结束程序。

ctrl + z：可以将一个正在前台执行的命令放到后台，并且暂停

vim四种模式

正常模式

启动vim后默认处于正常模式。不论位于什么模式，按下<Esc>键(有时需要按两下）都会进入正常模式。

插入模式

在正常模式中按下i, I, a, A等键，会进入插入模式。现在只用记住按i键会进行插入模式。在插入模式中，击键时会写入相应的字符。

命令模式

在正常模式中，按下:（英文冒号）键，会进入命令模式。在命令模式中可以执行一些输入并执行一些vim或插件提供的指令，就像在shell里一样。这些指令包括设置环境、文件操作、调用某个功能等等。

常用的命令有：q（退出）、q!（强制退出）、w（保存）、wq（保存并退出）。

可视模式

在正常模式中按下v, V, <Ctrl>+v，可以进入可视模式。可视模式中的操作有点像拿鼠标进行操作，选择文本的时候有一种鼠标选择的即视感，有时候会很方便。

awk命令是一个强大的文本分析工具，相对于grep的查找，sed的编辑，awk在其对数据分析并生成报告时，显得尤为强大。简单来说awk就是把文件逐行的读入，以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。

1、打印文件的第一列(域)

awk '{print $1}' filename

2、打印文件的前两列(域)

awk '{print $1,$2}' filename

有一个文件ip.txt，每行一条ip记录，共若干行，下面哪个命令可以实现“统计出现次数最多的前3个ip及其次数”？

sort ip.txt│uniq -c│sort -rn│head -n 3

sort 是按ASCII码排序

-n   依照数值的大小排序。

-r   以相反的顺序来排序。

uniq -c 是去重并显示个数

head -n 3 为取前3行

sort将文件的每一行作为一个单位，相互比较，比较原则是从首字符向后，依次按ASCII码值进行比较，最后将他们按升序输出。

中断

linux进程切换

1.正在运行的用户态进程X

2.发生中断

3.SAVE\_ALL//保存现场

4.中断处理过程中或中断返回前调用了schedule(),其中的switch\_to做了关键的进程上下文切换

5.标号1之后开始运行用户态进程Y(这里Y曾经通过以上步骤被切换出去过因此可以从标号1继续执行）

6.restore-all//恢复现场

7.iret -pop cs:eip/ss:eip/eflags from kernel stack

8.继续运行用户态进程Y

进程切换只发生在内核态

为了能保证不同的进程在CPU上运行，内核必须做到挂起正在CPU上运行的进程，唤醒其他进程，并且使其在CPU上正常运行。这个过程叫进程切换，或者上下文切换，任务切换

内核与硬件协调引发的，进程不会主动释放执行权的，是中断引起的

中断处理过程（包括时钟中断（设置好一个时间）、I\O中断（io接口有需求）、系统调用和异常）

**算法**

KMP算法

<http://kenby.iteye.com/blog/1025599>

http://blog.csdn.net/yutianzuijin/article/details/11954939/

kmp算法完成的任务是：给定两个字符串O和f，长度分别为n和m，判断f是否在O中出现，如果出现则返回出现的位置。常规方法是遍历a的每一个位置，然后从该位置开始和b进行匹配，但是这种方法的复杂度是O(nm)。kmp算法通过一个O(m)的预处理，使匹配的复杂度降为O(n+m)。

asdfbaaahiicl

fbaac//从这一为开始不匹配了，原来的算法只会向后移动一位，但后面这几位，在之前已经匹配过一遍了。

#########000xxxx000######                       文本T

|<--- s ---->|000xxxx000~~~                          模式P

**为什么能跳过中间的x向后匹配呢？**

#########000xxxx000######

|<--- s --------->|000xxxx000~~~

如果从这里开始能够完成匹配，说明000xxxx与xxxx000相等，而xxxx000又与原来的模式P的后部分相等，而此时最长前后缀数大于原来求得next数组中的值，所以矛盾了。

假设位移为s时，T和P匹配了红色部分的字符，即匹配到了模式P的前10个字符，如果按照传统的匹配方法，下一步就是从位移s+1开始比较，而kmp算法则直接从位移s+7开始比较，而且断定：位移s+7对应的串和模式P的前3个字符是相同的，可以不用比较，直接从第4个字符开始比较，这种跳跃式的匹配是不是比传统匹配方法快很多，如下图所示：

#########000xxxx000######                       文本T

|<-------- s+7------->|000xxxx000~~~               模式P

那么kmp是如何实现这种跳跃的呢？注意到红色部分的字符，即模式P的前10个字符，有一个特点：它的开始3个字符和末尾3个字符是一样的，又已知文本T也存在红色部分的字符，我们把位移移动 10-3 = 7个位置(已匹配长度-最大公共长度)，让模式P的开始3个字符对准文本T红色部分的末尾3个字符，那么它们的前3个字符必然可以匹配。

数组next就是维护模式P中每个一个位置前缀的最大公共长度。

规定：

next[1] = next[0] = 0//next[1]表示字符串的第一个字符

next[i]就是前缀数组，下面通过1个例子来看如何构造前缀数组。以i位置结尾

例子1：cacca有5个前缀，求出其对应的next数组。

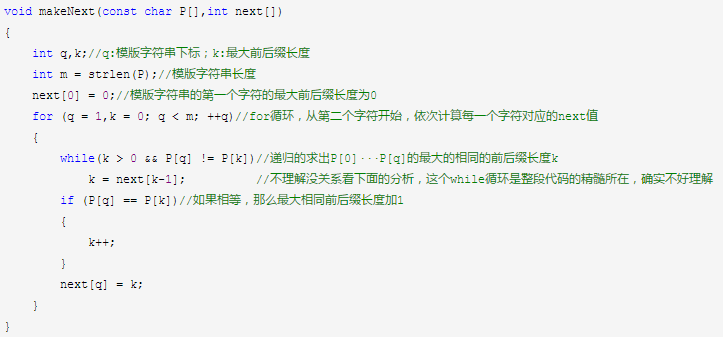
前缀2为ca，显然首尾没有相同的字符，next[2] = 0

前缀3为cac，显然首尾有共同的字符c，故next[3] = 1

前缀4为cacc，首尾有共同的字符c，故next[4] = 1

前缀5为cacca，首尾有共同的字符ca，故next[5] = 2

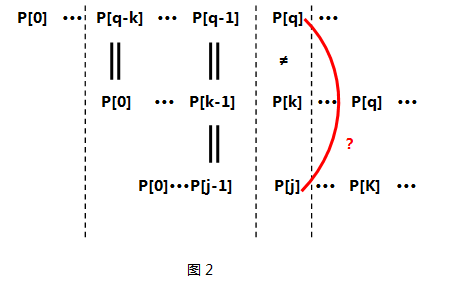
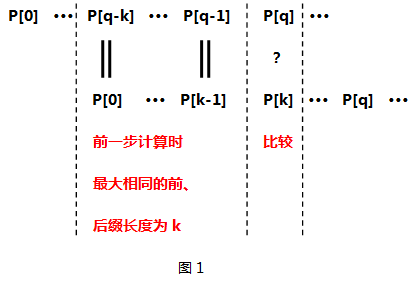
kmp算法next数组求解



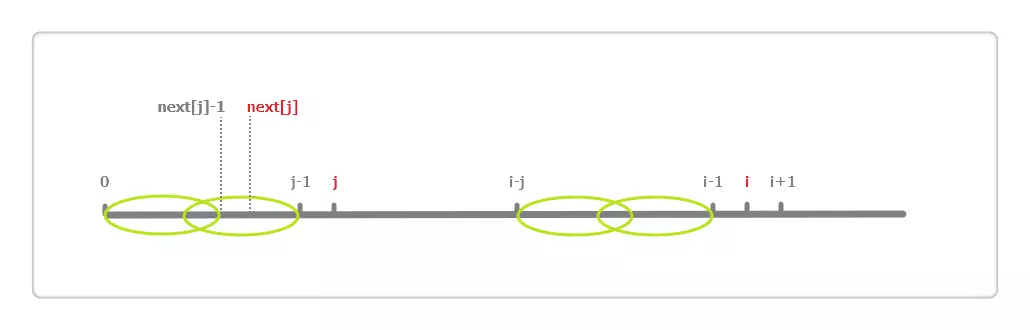
使用进行匹配前，先要计算next数组（即只通过模式字符串计算next数组，当待匹配字符串和模式串某个字符不匹配时，使用next数组跳跃）

假设模式串P[0]到P[q]的最长前缀数组都已经求出，下一个字符P[q]==P[k]（k为最长前缀），则next[q]=next[q-1]+1。

如果不相等，那么我们应该利用已经得到的next[0]···next[k-1]来**求P[0]···P[k-1]这个子串中最大相同前后缀**，可能有同学要问了——为什么要求P[0]···P[k-1]的最大相同前后缀呢？？？是啊！为什么呢？ **原因**在于P[k]已经和P[q]失配了，而且P[q-k] ··· P[q-1]又与P[0] ···P[k-1]相同，看来P[0]···P[k-1]这么长的子串是用不了了，那么我要找个同样也是P[0]打头、P[k-1]结尾的子串即P[0]···P[j-1](j=next[k-1])，看看它的下一项P[j]是否能和P[q]匹配。如图2所示



j = next[j]，见下图：

  
next[j]代表**0 到 j - 1** 区段中最长相同真前后缀的长度。如图，用左侧两个椭圆来表示这个最长相同真前后缀，即这两个椭圆代表的区段内容相同；同理，右侧也有相同的两个椭圆。又因为0到j-1和i-j到i-1是匹配的、相同的（且next[i-1]已知），所以第一个椭圆和第四个椭圆是相同的，即都表示从0开始的字符串

Floyd算法（多源最短路径）可以求有向图、无向图、权值为负

http://www.cnblogs.com/chenying99/p/3932877.html

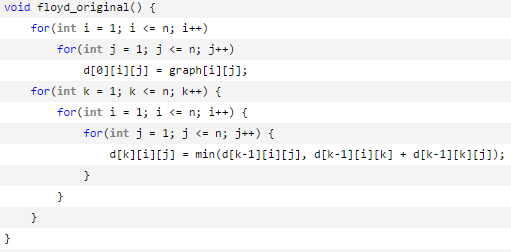
在动态规划算法中，处于首要位置、且也是核心理念之一的就是状态的定义。在这里，把**d[k][i][j]**定义成：

“只能使用第1号到第k号点作为中间媒介时，点i到点j之间的最短路径长度。”

d[1][i][j]表示只使用1号点作为中间媒介时，点i到点j之间的最短路径长度；d[2][i][j]表示使用1号点到2号点中的所有点作为中间媒介时，点i到点j之间的最短路径长度；d[n-1][i][j]表示使用1号点到(n-1)号点中的所有点作为中间媒介时，点i到点j之间的最短路径长度d[n][i][j]表示使用1号到n号点时，点i到点j之间的最短路径长度。有了状态的定义之后，就可以根据动态规划思想来构建动态转移方程。

       动态转移的基本思想可以认为是建立起**某一状态**和**之前状态**的一种**转移表示**。按照前面的定义，d[k][i][j]是一种使用1号到k号点的状态，可以想办法把这个状态通过动态转移，规约到使用1号到(k-1)号的状态，即d[k-1][i][j]。对于d[k][i][j]（即使用1号到k号点中的所有点作为中间媒介时，i和j之间的最短路径），可以分为两种情况：（1）i到j的最短路不经过k；（2）i到j的最短路经过了k。不经过点k的最短路情况下，d[k][i][j]=d[k-1][i][j]。经过点k的最短路情况下，d[k][i][j]=d[k-1][i][k]+d[k-1][k][j]。因此，综合上述两种情况，便可以得到Floyd算法的动态转移方程：

d[k][i][j] = min(d[k-1][i][j], d[k-1][i][k]+d[k-1][k][j])（k,i,j∈[1,n]）



利用滚动数组减小空间复杂度：

d[i][j] = min(d[i][j], d[i][k]+d[k][j])（k,i,j∈[1,n]）

计算d[k][i][j]时，d[k-1][i][j]还没被覆盖，所以可以用d[i][j]表示d[k-1][i][j]。

又有一条结论“dp[k-1][i][k]和dp[k-1][k][j]是不会在第k阶段改变大小的。也就是说，凡是和k节点相连的边，在第k阶段的值都不会变”，所以可以用d[i][k]表示d[k-1][i][k]。

如何简单证明呢？我们可以把j=k代入之前的d[k][i][j]=min(d[k-1][i][j], d[k-1][i][k]+d[k-1][k][j])方程中，即：

d[k][i][k]

= min(d[k-1][i][k], d[k-1][i][k]+d[k-1][k][k])

= min(d[k-1][i][k], d[k-1][i][k]+0)

= d[k-1][i][k]

dijkstra算法（单源最短路径）

带负权值的图，Dijkstra算法无法工作，既可以求有向图，又可以求无向图

1)算法思想：设G=(V,E)是一个带权有向图，把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 , 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了），第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

2)算法步骤：

a.初始时，S只包含源点，即S＝{v}，v的距离为0。U包含除v外的其他顶点，即:U={其余顶点}，若v与U中顶点u有边，则<u,v>正常有权值，若u不是v的出边邻接点，则<u,v>权值为∞。

b.从U中选取一个距离v最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。

c.以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（**经过顶点k**）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值的顶点k的距离加上边上的权。

d.重复步骤b和c直到所有顶点都包含在S中。

步骤c中修改的权值是从源点出发，到U中各点的距离。b中每次都是选取权值最小的点，也就是距离源点最近的点。

一开始时，各个点的距离值都是无穷，然后遍历源点的邻接点，将其距离值改为边的权值

动图：<http://www.cnblogs.com/biyeymyhjob/archive/2012/07/31/2615833.html>

自定义数据类型排序

public int getHeight(int[] w, int[] l, int[] h, int n) {

if(w == null || l == null || h == null)

return 0;

for(int i = n - 1;i > 0;--i){

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

if(w[j] < w[j+1]){

swap(w,j+1,j);

swap(l,j+1,j);

swap(h,j+1,j);

}

}

}

}

w,l,h分别表示宽长高，以w为标准排序

海量数据处理

1. 分而治之，通过哈希函数将大任务分流到机器，或分流成小文件。
2. 常用hashMap或bitmap（bitset）

Map-Reduce

用map-reduce方法统计一篇文章中每个单词出现的个数。

map阶段：

1、先对文章预处理，如去掉标点符号，处理I’m这种缩写

2、map阶段：对只包含单词的文本，对每个单词生成词频为1的记录（如（dog，1）、（pig，1）一个单词可能有许多个词频为1的记录，此时还没合并）

3、对单一词频记录通过哈希函数得到每个单词的哈希值，并根据该值分词若干组子任务。

reduce阶段：

4、因为通过哈希函数，所以同一单词都被分到同一个子任务中，但一个子任务可能包含多种不同的单词（哈希冲突）

5、对单个子任务中同一个单词的词频进行合并（在分布式系统中，每个子任务可能被分配到一台机器上，子任务并行计算，这是map-reduce的优势）

6、最后所有子任务进行合并，得到每种单词的词频统计

对10亿个IPV4的ip地址排序，已知每个ip地址只会出现一次

常规做法：

IPV4的ip数量约42亿个，将ip地址转为无符号整数（ip地址有四个部分，每个部分0~255，即28，总共有32位bit，所以最大能表示的数也就是232，如192168001001），再对整数排序，再转回ip地址。10亿个整数需要4G内存。

更好的做法：

申请长度为232的bit类型数组，每个位置上是一个bit，只可表示0或1两种状态，长度为232的bit数组，空间约为512m（29\*210\*210\*23）.

一个int是4个字节，即232种排列，bitmap的下标即为对应的整数值，遍历10亿个ip地址，将对应的bitmap位置涂黑。如一个ip地址为192168112110，将bit[192168112110]置为1.然后再从头开始遍历bitmap，将涂黑的位置依次取出，在转换成ip地址格式即可。（已经根据bitmap下标顺序排序）

因为为32位无符号的整数，所以范围为0到42亿多，因为原数字已经是无符号，且为32位（ip地址32位）。

有一个包含20亿个全是32位整数的大文件，找出其中出现次数最多的数，但内存限制为2G

常规方法：

使用hashmap记录所有数出现的次数。key为具体某一个数（整型4字节），value为这种数出现的次数（整型4字节，就算一个数出现20亿次也不会超过int的范围）。

所以一条记录（key,value）占有8字节，记录数为2亿时，大约占1.6G内存。也就是2\*8=16亿字节，也就是1.6\*109字节。也就是说如果20亿个数中，不重复的数超过2亿个，内存空间就很危险。所以光用这种方法，肯定内存不够。

解决办法：

用哈希函数将大文件分流成小文件。同一个数不可能被分配到不同的小文件。比如分配到16个小文件，因为哈希函数分配比较均匀，所以每个小文件的记录数不会超过2亿，再使用上面的方法在每个小文件中找出最大数，然后在这16个小文件中得到的16个最大的数中，再找出最大的，即为所求

32位无符号整数的范围是0到42亿多，现在有一个正好包含40亿个无符号整数的文件，所以在整个范围内，必然有没出现过的数，最多可以使用10MB的内存，只用找出一个没出现过的数即可，该如何找？

如果只使用bitmap，需要232长度的bitmap，需要内存512m，不符合要求。所以可以先将42亿分成64个小区间（512m内存除以64等于8m），统计每个区间上的数出现了多少次（如遍历40亿个数，每个区间有一个计数遍历count），肯定有出现次数小于区间长度的区间，找到这个出现次数小于区间长度的区间，再在这个区间上使用bitmap，bitmap的下标即为对应的整数值。

假设某搜索公司用户一天的搜索词汇量有百亿，请设计一种求出每天最热的100词的方法

先利用哈希函数分流，把包含百亿数据量的词汇分流到n台机器上，如果每台机器上分到的文件依然很大，可以再用哈希函数，将大文件分为小文件，然后利用hashmap统计每个小文件里的词频，再建立大根堆，找到每个小文件中的top100.再利用大根堆将每个小文件的top100选出每台机器上的top100，最终选出整个文件的top100；

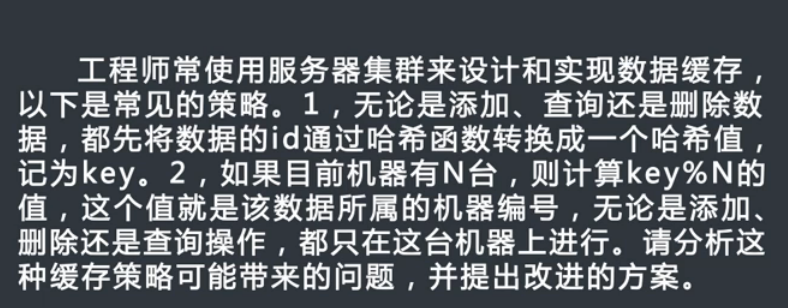
如果有100亿个黑名单网页，请设计一种网页过滤系统，可以根据url判断该网页是否在黑名单上。

如果黑名单数量少，可以采用hashmap，或前缀树

如果黑名单数量多，采用布隆过滤器，假设有k个哈希函数，它们的输出域均为n，建立长度为m的bitmap，对每个黑名单应用k个哈希函数，将得到的k个值取余m，然后将对应的bitmap涂黑。这样只要在黑名单上的一定会被过滤，但可能有误判，不在黑名单的也被过滤。

几亿数据量的大数组排序

步骤：   
1、从大数据中抽取样本，将需要排序的数据切分为多个样本数大致相等的区间，例如：1-100，101-300…   
2、将大数据文件切分为多个小数据文件，这里要考虑IO次数和硬件资源问题，例如可将小数据文件数设定为1G（要预留内存给执行时的程序使用）   
3、使用最优的算法对小数据文件的数据进行排序，将排序结果按照步骤1划分的区间进行存储   
4、对各个数据区间内的排序结果文件进行处理，最终每个区间得到一个排序结果的文件   
5、将各个区间的排序结果合并.   
通过分治将大数据变成小数据进行处理，再合并。



潜在问题：增加或删除机器时，代价高。需要重新计算每个key的哈希值，并进行大规模数据迁移。

可以使用一致性哈希算法，如果key的范围在0~232内，将这个范围组成一个环，每台机器只负责环中的一部分（如从另一台机器顺时针到本机器的范围），如果一台机器坏了数据需要修改的只有这个小范围。添加机器时，根据机器id，找到机器在环中的位置，只要把data1从m2复制到m3即可。为了保证均匀可以利用虚拟结点。

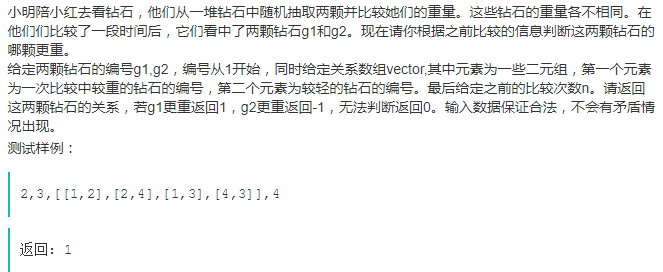


SQL

一个班级某次考试，对于相同分数，我只取学号最大的那个，输出所有不同的分数以及学号

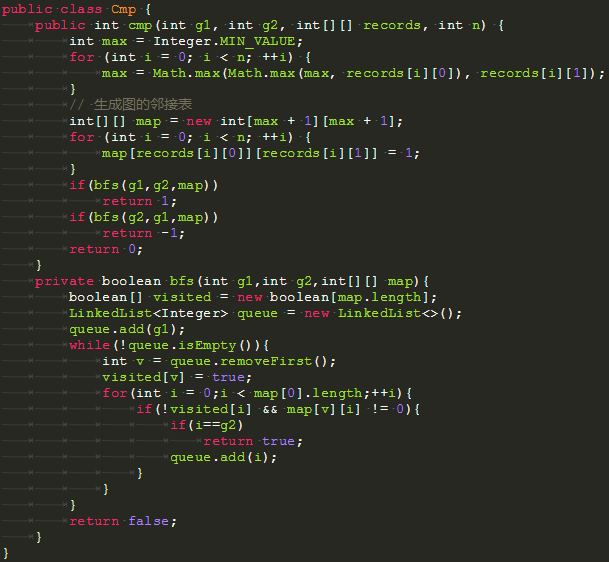
SELECT MAX(id),chengji FROM `user` GROUP BY chengji;

关于图的算法题



当做一个有向图来处理，生成邻接矩阵，可以采用dfs或bfs的方法。

BFS：



DFS：



**其他**

JSON

JSON 语法是 JavaScript 对象表示语法的子集。

* 数据在键值对中
* 数据由逗号分隔
* 花括号保存对象
* 方括号保存数组

{

"people":[

{"firstName":"Brett","lastName":"McLaughlin","email":"aaaa"},

{"firstName":"Jason","lastName":"Hunter","email":"bbbb"},

{"firstName":"Elliotte","lastName":"Harold","email":"cccc"}

]

}

常用设计模式

工厂模式

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

有多个类，实现同一个接口，但是提供的功能是不同的，将他们封装在工厂中，根据条件选择需要生成的类。

简单工厂模式：一个模块仅需要一个工厂类，没有必要把它生产出来（new），使用静态的方法就可以了，即只使用静态方法，工厂模式的缩小版，不利于扩展。

单例模式

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

适配器模式

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

如何解决：继承或依赖（推荐）。

关键代码：适配器继承或依赖已有的对象，实现想要的目标接口。

装饰器模式

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。将原类传入装饰类，装饰类与原类实现相同的接口，在接口的方法中调用传入的原类的方法，并添加一些自己的方法

享元模式

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。

当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象。一提到共享池，我们很容易联想到Java里面的JDBC连接池、线程池，想想每个连接的特点，我们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password及dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。

代理模式

代理模式就是多一个代理类出来，替原对象进行一些操作

责任链模式

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。

观察者模式

当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

策略模式

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数

MVC模式（好像不是）

MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

Model（模型） - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。

View（视图） - 视图代表模型包含的数据的可视化。

Controller（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

迭代器模式

迭代器模式就是顺序访问聚集中的对象，一般来说，集合中非常常见

模板方法模式

解释一下模板方法模式，就是指：一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用

抽象类就相当于一个模板

http://www.cnblogs.com/maowang1991/archive/2013/04/15/3023236.html

二叉树结点

完全二叉树：叶节点只能出现在最下层和次下层，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树

满二叉树：除最后一层无任何子[节点](http://baike.baidu.com/item/%E8%8A%82%E7%82%B9)外，每一层上的所有结点都有两个子结点二叉树

设度为0,1,2的节点分别为n0,n1,n2个，那么节点总数n=n0+n1+n2。

n0=n2+1，也就是说叶子节点要比度为二的节点多一个

深度为k的二叉树，最多有2^k-1个节点（满二叉树）

二叉树的第i层至多有2^{i-1}个结点

solr分词

为什么要分词?

因为搜索这种东西，不是像字符串查询那样，完全按照关键字去对比。比如有一往篇文章，我们要通过一个关键字来搜索这篇文章的内容。首先，我们要对这个文章建立索引，怎么建立的，就是通过分词来建立的，分好词组以后。再通过关键字与每个分词的索引进行比对，分词后检索速度快

3、解释一下Lucene还有 IK 要解决一个什么问题，以及大致的原理？

我们传统的数据库是把文本整段存在库里面，当是搜索一个词的时候，它使用这个词去大文本里面去匹配，这个匹配过程速度很慢。Lucene是一个全文搜索引擎，它的思路是倒过来的。它做的事情分成两个阶段。第一阶段是把文章，也就是长文本在存储的时候建索引（反向索引、倒排索引：单词是索引，记录的内容是谁引用到我了）。建索引的过程要把文本按照词来切分。切分的时候是按词源切分，再把每个词都作为索引，指出词在文章中的位置以及跟哪些文章相关联。当有大量文章进来的时候，你可以想象一下有一个字典，这个字典是按照词来排序，每个词都会记录有哪些文章引用到我了，以及文章的哪些部分。这个索引的过程就是Lucene做的一个存储过程。那在这个存储过程中，对于英文而言基本上按照空格来切分就可以了，但是中文是连续性的，所以计算机就需要一个能够识别中文词汇的程序和算法，这就是中文分词器。分词器在文章进入到数据库的时候，对它进行分词。比如用户输入了“咖啡牛排”，那分词器就要把这个切分成“咖啡”和“牛排”。默认一个词一个词分。

6、分词的基本原理？

首先必须有一个词典

IK分词器采用“正向迭代最细粒度切分算法”，属于“正向最大匹配法”的变种，从正向开始匹配，在词典中找，是否存在



tomcat部署参数

1、jvm的设置：

tomcat 的jvm建议调整到1024M 即在tomcat/bin/catalina.bat文件头部添加

set JAVA\_OPTS=-Xms1024m -Xmx1024m 默认tomcat 最大为64M

2、调整tomcat的server.xml 其中的参数配置，例

将其中的maxThreads=”150″ 修改为maxThreads=”1500″ ，即可以最大承载1500个并发事务。

1. 不要遗漏URIEncoding=”GBK”，能使页面url传递中文参数时保证正确。
2. 如果使用tomcat自带的dbcp数据库链接池，数据库连接池最大连接数
3. 日志功能

Tomcat热部署机制

对于Java应用程序来说，热部署就是在运行时更新Java类文件。在基于Java的应用服务器实现热部署的过程中，类装入器扮演着重要的角色。大多数基于Java的应用服务器，包括EJB服务器和Servlet容器，都支持热部署。类装入器不能重新装入一个已经装入的类，但只要使用一个新的类装入器实例，就可以将类再次装入一个正在运行的应用程序。

既然在类加载器中，java类只能被加载一次，并且无法卸载。那是不是可以直接把类加载器给换了？答案是可以的，我们可以自定义类加载器，并重写ClassLoader的findClass方法。想要实现热部署可以分以下三个步骤：

1、销毁该自定义ClassLoader

2、更新class类文件

3、创建新的ClassLoader去加载更新后的class类文件。

我们在使用Tomcat或者Jboss等应用服务器开发应用时，我们经常会开启热部署功能。热部署，，就是我们将打包好的应用直接替换掉原有的应用，不用关闭或者重启服务器，替换webapps文件（或webapps\wenda\WEB-INF\classes）。

在Java中，要实现热部署，首先，你得明白，Java中类的加载方式。每一个应用程序的类都会被ClassLoader加载，所以，要实现一个支持热部署的应用，我们可以对每一个用户自定义的应用程序使用一个单独的ClassLoader进行加载。然后，当某个用户自定义的应用程序发生变化的时候，我们首先销毁原来的应用，然后使用一个新的ClassLoader来加载改变之后的应用。而所有其他的应用程序不会受到一点干扰。

负载均衡

负载均衡算法：轮询（依次分发到每台服务器）、加权轮询、随机、最少连接（记录正在处理的请求数）、源地址散列（对请求来源的ip进行hash）

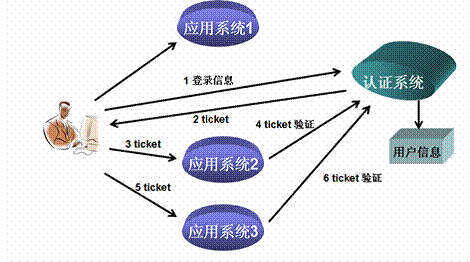
负载均衡分类：http重定向负载均衡（有一个http重定向服务器）、DNS域名解析负载均衡（在DNS服务器中配置多个记录）、反向代理负载均衡（需要反向代理服务器）、ip负载均衡

反向代理（Reverse Proxy）方式是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个服务器。

浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间，深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

session共享：session复制（每个服务器复制一份（负载均衡时）），session绑定（同一个ip请求同一个服务器），cookie记录session，建立session服务器

单点登录的技术实现机制，统一的认证服务平台



当用户第一次访问应用系统1的时候，因为还没有登录，会被引导到认证系统中进行登录；根据用户提供的登录信息，认证系统进行身份效验，如果通过效验，应该返回给用户一个认证的凭据－－ticket；用户再访问别的应用的时候，就会将这个ticket带上，作为自己认证的凭据，应用系统接受到请求之后会把ticket送到认证系统进行效验，检查ticket的合法性（4,6）。如果通过效验，用户就可以在不用再次登录的情况下访问应用系统2和应用系统3了。

**单点登录或者是将用户的登录信息加密后，放入cookie中**

i=i++

i++ 相比 ++i 哪个更高效?为什么？

前置的自增运算符（++i）速度快，因为前置的自增后直接返回引用，而后置的先定义一个临时变量，把值赋给临时变量，然后自增，返回临时变量

++i的效率高些，++i在运算过程中不产生临时对象，返回的就是i,是个左值，类似++i=1这样的表达式是合法的，而i++在运算的过程中会产生临时对象，返回的是临时对象的值，是个右值,像i++=1这样的表达式是非法的

对于内置类型，单独的i++和++i语句，现在的编译器基本上都会优化成++i,所以就没什么区别了

int i = 0;

i = i++;//此时i值还是0,i=i++永远也加不成功

i = ++i;//此时i值为1

i = i+1;//i值会增加

可以理解为i++和++i为表达式，有返回值，i++返回自增前的值，++i返回自增后的值。

i=i++首先把i的值（注意是值，不是引用）拷贝到一个临时变量区（操作数栈），然后对i变量加1，最后返回临时变量区的值

int i = 0;

i = i++;

通过javap大致可以将上面的两行代码翻译成如下的JVM指令执行代码。

0: iconst\_0

1: istore\_1

2: iload\_1

3: iinc          1, 1

6: istore\_1

7: iload\_1

接下来分析一下JVM是如何执行的:

第0：将int类型的0入栈，就是放到操作数栈的栈顶

第1：将操作数栈栈顶的值0弹出，保存到局部变量表 index （索引）值为1的位置。（局部变量表也是从0开始的，0位置一般保存当前实例的this引用，当然静态方法例外，因为静态方法是类方法而不是实例方法）

第2：将局部变量表index 1位置的值的副本入栈。（这时局部变量表index为1的值是0，操作数栈顶的值也是0）

第3：iinc是对int类型的值进行自增操作，后面第一个数值1表示，局部变量表的index值，说明要对此值执行iinc操作，第二个数值1表示要增加的数值。（这时局部变量表index为1的值因为执行了自增操作变为1了，但是操作数栈中栈顶的值仍然是0）

第6：将操作数栈顶的值弹出（值0），放到局部变量表index为1的位置（旧值：1，新值：0），覆盖了上一步局部变量表的计算结果。

第7：将局部变量表index 1位置的值的副本入栈。（这时局部变量表index为1的值是0，操作数栈顶的值也是0）

虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。比如，iadd指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中，看看下面的示例，它演示了虚拟机是如何把两个int类型的局部变量相加，再把结果保存到第三个局部变量的：

**Java代码**

1. begin
2. iload\_0    // push the int in local variable 0 onto the stack
3. iload\_1    // push the int in local variable 1 onto the stack
4. iadd       // pop two ints, add them, push result
5. istore\_2   // pop int, store into local variable 2
6. end

      在这个字节码序列里，前两个指令iload\_0和iload\_1将存储在局部变量中索引为0和1的整数压入操作数栈中，其后iadd指令从操作数栈中弹出那两个整数相加，再将结果压入操作数栈。第四条指令istore\_2则从操作数栈中弹出结果，并把它存储到局部变量区索引为2的位置。

扫码登录原理

微信手机客户端从网页二维码里面得到一些信息，然后发送给网页微信的服务器，网页服务器验证信息并响应。

1.每次打开微信网页版的时候，都会生成一个含有唯一uid的二维码，而且每次刷新后都会改变。这样可以保证一个uid只可以绑定一个账号和密码，确定登录用户的唯一性（类似于一个token）。

2.除了返回唯一的uid，实际上打开这个页面的时候，浏览器跟服务器还创建了一个长连接，请求uid的扫描记录。如果没有，在特定时长后（目前是27秒左右）会接到状态码408（请求超时），表示应该继续下一次请求；如果接到状态码201（服务器创建新资源成功），表示客户端扫描了该二维码。

3.当用户使用登录后的微信扫描二维码的时候，会将uid和手机微信产生的token进行绑定，并上传到服务器。这个时候，浏览器通过长轮询查询到uid扫描记录，立即得到201响应码，然后通知服务器，客户端由此也进入一个新的页面（就是那个要你点确认的按钮）。在客户端点击确认后，获得服务器授信的令牌，进行随后的信息交互过程。

秒杀系统设计

秒杀系统设计的第一个原则就是将这种热点数据隔离出来，不要让1%的请求影响到另外的99%，隔离出来后也更方便对这1%的请求做针对性优化。

1、秒杀系统独立部署，如果需要还可以使用独立的域名，使其与网站完全隔离，目的也是让请求落到不同的集群中；

2、商品页面静态化，不需要访问数据库；

3、租借秒杀活动网络带宽，将秒杀商品页面缓存在CDN；将90%的数据缓存在客户端浏览器；

（CDN是构建在网络之上的内容分发网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。

）

4、动态生成随机下单页面URL，避免用户直接访问下单页面URL，在下单页面URL加入由服务器端生成的随机数作为参数，在秒杀开始的时候才能得到。

控制秒杀商品页面购买按钮点亮：由于是静态页面，缓存在CDN或反向代理服务器或用户浏览器上，用户刷新页面请求不会到达应用服务器，解决方法是使用JS控制，在JS中加入秒杀是否开始标志和下单URL随机参数（相当于不是同一个请求，不会查看缓存），该JS使用随机版本号，不会被浏览器、CDN等缓存，且该JS十分小，不会对网络带宽造成太大压力。

5、减小压力，已经有若干个用户下单后，其他用户直接跳转到秒杀结束页面。有一个队列（使用redis）。

防止表单重复提交

用户多次刷新页面

由于网速较慢，导致用户没有得到及时的反馈，从而多次点击提交按钮

解决方案：1.使用js控制按钮，点击后变灰；2.如果是用户注册的功能，可以在数据库中设计一个字段，如果已经写入，则拒绝请求；3. 首先客户端请求服务器中的表单，服务器将客户机所请求的表单发给客户机同时发送一个特殊的随机数(Token)作为表单号存在表单的隐藏域中(type=hidden)，并且存入服务器端的session中。在客户端填写完表单内容向服务器提交时，同时也将隐藏域中的表单号发给服务器端，服务器端此时会检测服务器端的表单号是否存在，如果存在，则进行提交操作，并删除此表单号，否则，服务器视为客户机端重复提交表单，不予操作。

如何设计一个高并发的系统

① 数据库的优化，包括合理的事务隔离级别、SQL语句优化、索引的优化

② 使用缓存，尽量减少数据库 IO

③ 分布式数据库、分布式缓存

1. 服务器的负载均衡

多用户同时登录的session

jsp四大域：page<request<session<application

servlet三大域：request<session<application

> ServletContext：整个应用程序

> session：整个会话(一个会话中只有一个用户)

> request：一个请求链！

> pageContext：一个jsp页面！这个域是在当前jsp页面和当前jsp页面中使用的标签之间共享数据！一个顶九个

<> 域对象

<> 代理其他域：pageContext.setAttribute("xxx", "XXX", PageContext.SESSION\_SCOPE);//从session域中取数据

<> 全域查找：pageContext.findAttribute("xxx");从小到大，依赖查找！

<> 获取其他8个内置对象：

session是整个会话共用（一个浏览器），不同的用户使用的是不同的session，如cookie中携带sessionID，不会引起冲突。如果有一个浏览器登录多个用户，后台判断的时候可以根据sessionID+USERID判断

redis做缓存保证与mysql的一致性（不要去强一致性时可以加缓存）

方式1：数据库保存数据，redis不persist

redis启动后，从数据库加载数据

不要求强一致实时性的读请求，都由redis处理

要求强一致实时性的读请求，由数据库处理

写请求有2种处理方式，由数据库处理

- 应用先写道数据库，然后更新redis

- 应用先写道数据库，然后其它daemon同步到redis

优点：redis启动不用处理redis数据和数据库不一致

缺点：redis启动给数据库很大的读压力

方式2：数据库和redis分别处理不同的数据类型

数据库处理要求强一致实时性的数据，例如金融数据、交易数据

redis处理不要求强一致实时性的数据，例如网站最热贴排行榜

redis和mysql数据的同步，代码级别大致可以这样做：

读: 读redis->没有，读mysql->把mysql数据写回redis

写: 写mysql->成功，写redis

并发不高的情况：

读: 读redis->没有，读mysql->把mysql数据写回redis，有的话直接从redis中取；

写: 写mysql->成功，再写redis；

并发高的情况：

读: 读redis->没有，读mysql->把mysql数据写回redis，有的话直接从redis中取；

写：异步话，先写入redis的缓存，就直接返回；定期或特定动作将数据保存到mysql，可以做到多次更新，一次保存；

视频传输传输层用的到底是tcp还是udp

TCP 和 UDP 是质量和实时性的权衡。

拿视频网站来说，你完全可以缓冲 20s 再播放，不会带来什么影响，但如果画面有马赛克之类的东西出现肯定是不好的，所以用 TCP。（如使用http，http2.1,2.2是视频播放协议）

而对于视频聊天，如果缓冲 5s，相信整个聊天已经没法愉快的进行了，而这时出现一些画面质量的损失也可以被接受，所以用 UDP。

数据库宕机后的可用性

数据备份（热备份）:异步热备方式和同步热备方式

1. 异步热备方式是指多份数据副本的写入操作异步完成。在异步写入方式下，存储服务器分为主存储服务器（Master）和从存储服务器（Slave），应用程序正常情况下只连接主存储服务器，数据写入时，由主存储服务器写操作代理模块将数据写入本机存储系统后立即返回写操作成功响应，然后通过异步线程将写操作数据同步到从存储服务器。
2. 同步方式是指多份数据副本的写入操作同步完成，即应用程序收到数据服务系统的写成功响应时，多份数据都已经写操作成功（总写操作延迟是响应最慢的那台存储服务器的响应延迟）。但是当应用程序收到数据写操作失败的响应时，可能有部分副本或者全部副本都已经写成功了（因为网络或者系统故障，无法返回操作成功的响应）这种情况下，存储服务器没有主从之分，完全对等，更便于管理和维护。

关系数据库热备机制就是通常所说的Master-Slave同步机制。Master-Slave机制不但解决了数据备份问题，还改善了数据库系统的性能。通常使用读写分离的方法，写操作只访问Master数据库，读操作只访问Slave数据库。

失效确认：用心跳检测判断某台服务器是否宕机，如果宕机，将该台机器的读写访问重新路由到其他服务器上。

redis主从复制（建立从库的时候从主库生成一个快照，将数据同步过来）

1、redis的复制功能是支持多个数据库之间的数据同步。一类是主数据库（master）一类是从数据库（slave），主数据库可以进行读写操作，当发生写操作的时候自动将数据同步到从数据库（每条写都同步，异步写，是一种弱一致性），而从数据库一般是只读的，并接收主数据库同步过来的数据，一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。（从库也可以有自己的从库，成为主从链）主从同步就是 RDB 文件的上传下载；主机有小部分的数据修改，就把修改记录传播给每个从机。

2、通过redis的复制功能可以很好的实现数据库的读写分离，提高服务器的负载能力。主数据库主要进行写操作，而从数据库负责读操作。

在主从架构中出现了宕机的情况（人工操作）

① Slave 宕机

在Redis中,从库重新启动会自动加入到主从架构中,自动完成同步数据

Redis 2.8之后,在从库有做持久化的前提下,如果从库在断开的期间,主库的变化不大,从库再次启动后,主库不会将或有的数据进行RDB操作,而是进行增量复制

② Master 宕机

一 : 在Slave中执行SLAVEOF NO ONE 命令,断开主从关系并且提升为主库继续服务;

二 : 将主库重新启动后,执行 SLAVEOF命令,将其设置为其他Redis的从库,这个时候数据就同步回来了;

Redis 哨兵模式（Sentinel ）实现主从故障互切换，当主宕机了从接替主成为新的主，宕机的主启动后自动变成了从。

Mysql

将Mysql的数据分布到多个系统上去，这种分布的机制，是通过将Mysql的某一台主机的数据复制到其它主机（slaves）上，并重新执行一遍来实现的。复制过程中一个服务器充当主服务器，而一个或多个其它服务器充当从服务器。主服务器将更新写入二进制日志文件，并维护文件的一个索引以跟踪日志循环。这些日志可以记录发送到从服务器的更新。当一个从服务器连接主服务器时，它通知主服务器从服务器在日志中读取的最后一次成功更新的位置。从服务器接收从那时起发生的任何更新，然后封锁并等待主服务器通知新的更新。

CPU最大能查找多大范围的地址叫做寻址能力 ，CPU的寻址能力以字节为单位 ，如32位寻址的CPU可以寻址2的32次方大小的地址也就是4G，这也是为什么32位的CPU最大能搭配4G内存的原因 ，再多的话CPU就找不到了。