1. **三种多线程创建方法的优缺点：**

**继承Thead类：**

优势：编写简单，如果需要访问当前线程，只需使用this即可，无需使用Thead.currentThread()方法。

劣势：因为这种线程类已经继承了Thead类，所以不能再继承其它类。

**采用实现Runable接口的多线程：**

优势：线程类只是实现了Runable接口，因此还可以继承其他类；

在这种情况下，可以使多个线程共享一个target对象，所以非常适合多个线程用来处理同一份资源的情况，（继承Thread方法要实现共享就需要将变量设为静态的）

劣势：编程略有些复杂，如果要访问当前线程必须使用Thread.currentThread方法。

**使用ExecutorService、Callable、Future实现有返回结果的多线程：**

可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。

**使用线程池的好处:**   
    1.减少在创建和销毁线程上所花的时间以及系统资源的开销   
    2.如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存以及”过度切换”。

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目，尤其是当程序中需要创建大量生存期很短暂的线程时，更应该考虑使用线程池。

ExecutorService exe=Executors.newCachedThreadPool();

exec.execute(run);

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

ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。

也可以直接用callable创建多线程

创建并启动有返回值的线程步骤与实现Runnable接口相似：  
1.创建Callable接口的实现类，并实现call()方法  
2.创建Callable实现类的实例，使用FutureTask类来包装Callable对象  
3.使用FutrueTask对象作为Thread对象的target创建并启动新线程。  
4.调用FutrueTask对象的get()方法来获得子线程执行结束后的返回值。  
例：

**[java]** [view plain](http://blog.csdn.net/zhuangxingrui86/article/details/7865677) [copy](http://blog.csdn.net/zhuangxingrui86/article/details/7865677)

1. **import** java.util.concurrent.Callable;
2. **import** java.util.concurrent.FutureTask;
4. **public** **class** ThirdThread **implements** Callable<Integer>{
5. **public** Integer call(){
6. **int** i = 0;
7. **for**( ; i<60; i ++){
8. System.out.println(Thread.currentThread().getName() + "循环变量i的值是 : " + i);
9. }
10. **return** i;
11. }
13. **public** **static** **void** main(String[] args) {
15. ThirdThread rt = **new** ThirdThread(); //创建Callable对象
16. FutureTask<Integer> task = **new** FutureTask<Integer>(rt);//使用FutrueTask来包装Callable对象
17. **for**(**int** i=0 ; i<60 ; i++){
18. System.out.println(Thread.currentThread().getName() + "-------" + i);
19. **if**(i == 10){
20. **new** Thread(task, "有返回值的线程").start(); //实际上还是以Callable对象创建并启动线程
21. }
22. }
23. **try**{
24. System.out.println("子线程的返回值 ：" + task.get());
25. }
26. **catch**(Exception e){
27. e.printStackTrace();
28. }
29. }
30. }

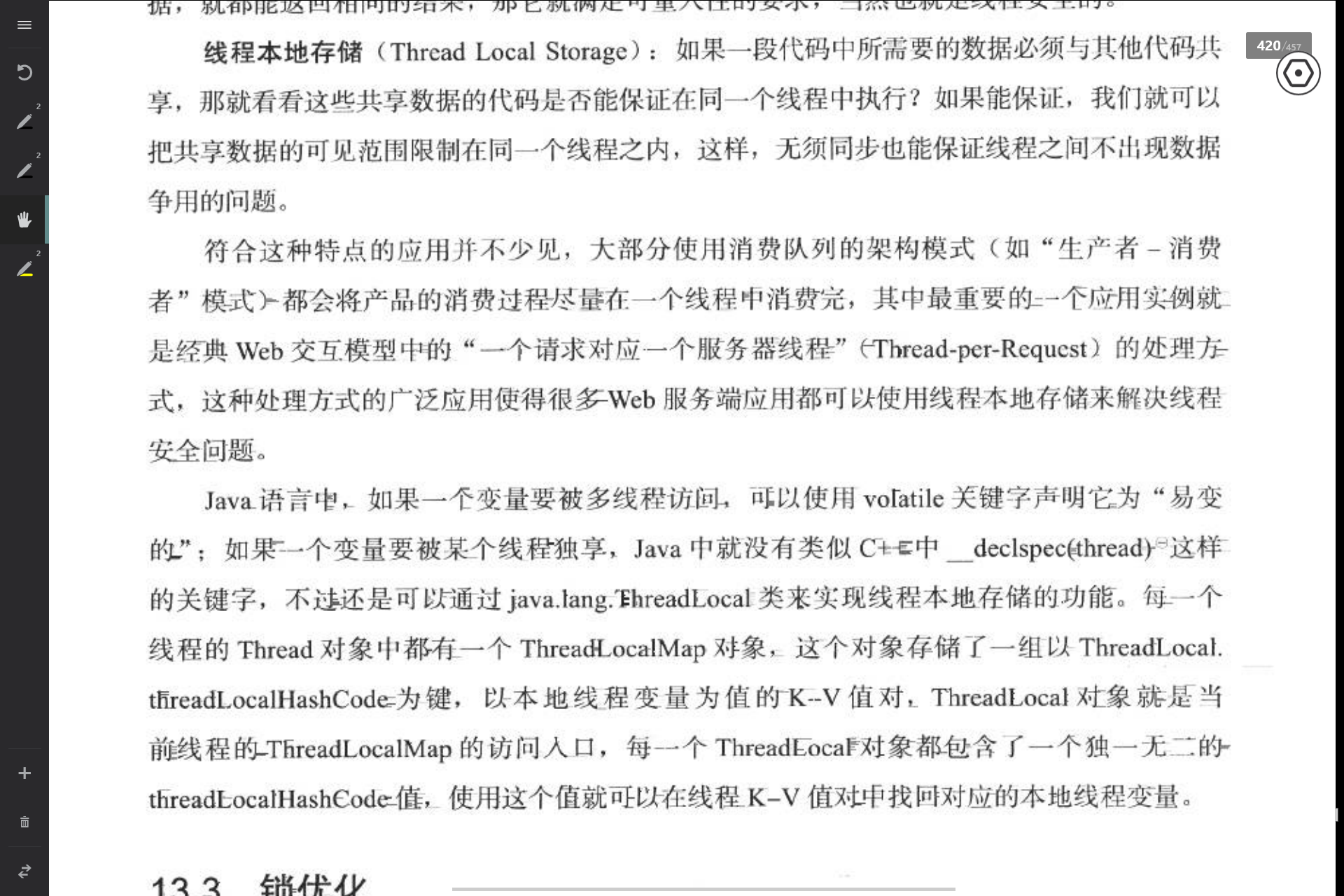
通过继承Thread类或实现Runnable、Callable接口都可以实现多线程，不过实现Runnable接口与实现Callable接口的方式基本相同，只是Callable接口里定义的方法有返回值，可以声明抛出异常而已。采用实现Runnable、Callable接口的方式创建多线程，线程类只是实现了Runnable接口或Callable接口，还可以继承其他类在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况。

 个人理解：可以把Thread类理解成代工厂，而Runnable接口则是原材料，工厂只负责将符合标准的原材料加工成成品，而Callable接口则是另一种材料，他需要先被FutrueTask加工成半成品（即符合Runnable标准的材料），然后在交给Thread工厂加工。

1. **volatile关键字在Java中有什么作用？**

当我们使用volatile关键字去修饰变量的时候，所有线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

### 什么是ThreadLocal?



每个线程都会拥有他们自己的Thread变量，它们可以使用get()\set()方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal实例通常是希望它们同线程状态关联起来是private static属性

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单：在ThreadLocal类中有一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本。

用处：一般用在当多个线程都需要用到一个变量， 最常见的ThreadLocal使用场景为用来解决数据库连接、Session管理等。

　　如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | private static ThreadLocal<Connection> connectionHolder  = new ThreadLocal<Connection>() {  public Connection initialValue() {      return DriverManager.getConnection(DB\_URL);  }  };    public static Connection getConnection() {  return connectionHolder.get();  } |

### HashMap源码分析

### **Hashmap内部的实现就是一个Entry数组，**

**Entry的结构**

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

final K key;

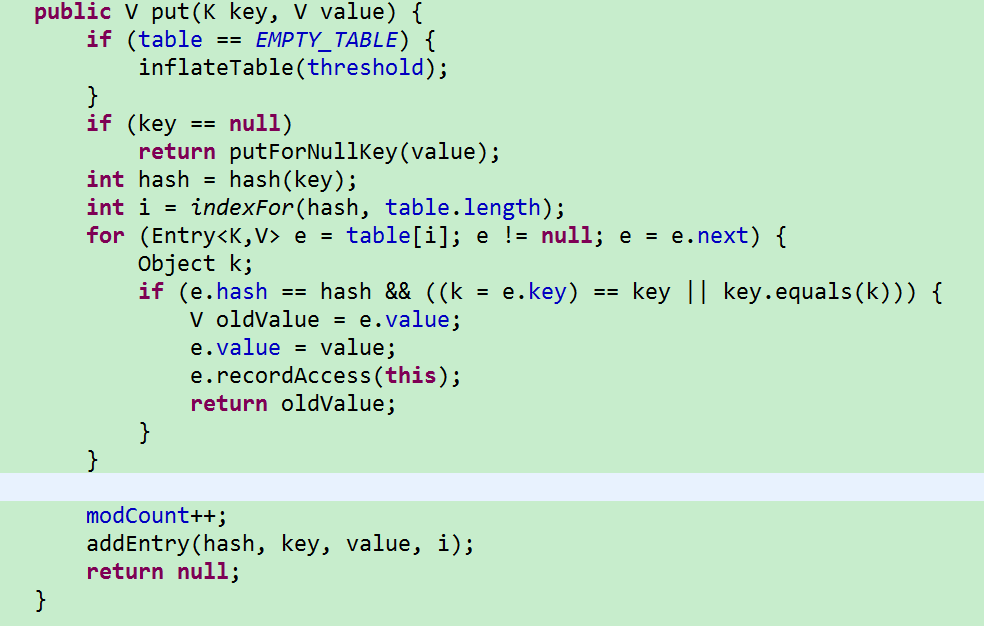
V value;

// 指向下一个节点

Entry<K,V> next;

final int hash;

存储操作



当存入的值为null的时候，操作会找到table[0]，set key为null的值为新值

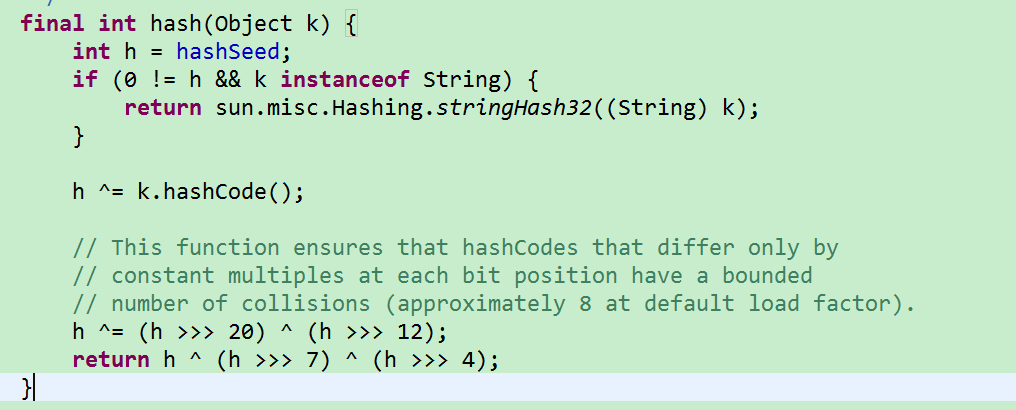
若果不是空值，则进行hash，

当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

当系统决定存储HashMap中的key-value对时，完全没有考虑Entry中的value，仅仅只是根据key来计算并决定每个Entry的存储位置。我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

   hash(int h)--计算hash值的方法根据key的hashCode重新计算一次散列。此算法加入了高位计算，防止低位不变，高位变化时，造成的hash冲突。

hash算法



可以看到，算法进行了二次hash，使高位也参与到计算中，防止低位不变造成的hash冲突

注：这一切的目的实际上都是为了使value尽量分布到不同的

**计算索引的函数如下：**

static int indexFor(int h, int length) { //根据hash值和数组长度算出索引值

return h & (length-1); //这里不能随便算取，用hash&(length-1)是有原因的，这样可以确保算出来的索引是在数组大小范围内，不会超出

}

我们分析下为什么哈希表的容量一定要是2的整数次幂。首先，length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；

其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，

而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，

因此，length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

快速失败

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。

### HashMap和Hashtable的区别

1. **两者最主要的区别在于Hashtable是线程安全，而HashMap则非线程安全**  
   Hashtable的实现方法里面都添加了synchronized关键字来确保线程同步，因此相对而言HashMap性能会高一些，我们平时使用时若无特殊需求建议使用HashMap，在多线程环境下若使用HashMap需要使用Collections.synchronizedMap()方法来获取一个线程安全的集合
2. HashMap可以使用null作为key，而Hashtable则不允许null作为key
3. Hashtable继承自Dictionary类，而HashMap继承自AbstractMap类。但二者都实现了Map接口。
4. Hash的计算方法不同，Hashtable计算hash是直接使用key的hashcode对table数组的长度直接进行取模。而HashMap计算hash对key的hashcode进行了二次hash，以获得更好的散列值，然后对table数组长度取摸
5. 内部实现使用的数组初始化和扩容方式不同。Hashtable和HashMap它们两个内部实现方式的数组的初始大小和扩容的方式。

HashTable中hash数组默认大小是11，增加的方式是 old\*2+1。

HashMap中hash数组的默认大小是16，而且一定是2的指数。

1. HashMap和Hashtable的底层实现都是数组+链表结构实现，链表是单向链表，但在jdk1.8中HashMap采用位桶+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间

具体操作：

当节点冲突数达到8时，就会对hash表进行调整，如果table的长度小于64，那么会进行table扩容，如果不小于64，那么会将因冲突形成的单链表调整为红黑树。

### HashSet和HashMap、Hashtable的区别

HashSet则实现了Set接口，性质类似于集合。

HashSet内部就是使用HashMap实现，只不过HashSet里面的HashMap所有的value都是同一个Object而已，因此HashSet也是非线程安全的。

1. **TreeMap**

TreeMap实现SortMap接口，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。

LinkedHashMap是HashMap的一个子类，它**保留插入的顺序**， LinkedHashMap实现与HashMap的不同之处在于，它维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。

根据链表中元素的顺序可以分为：**按插入顺序的链表，和按访问顺序(调用get方法)的链表。**

默认是按插入顺序排序，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表。

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表。

LinkedHashMap.Entry:

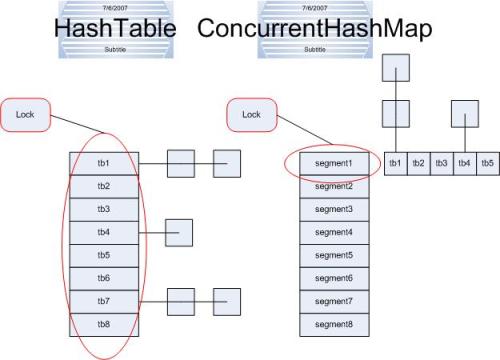
1. /\*\*
2. \* 双向链表的表头元素。
3. \*/
4. **private** **transient** Entry<K,V> header;
6. /\*\*
7. \* LinkedHashMap的Entry元素。
8. \* 继承HashMap的Entry元素，又保存了其上一个元素before和下一个元素after的引用。
9. \*/
10. **private** **static** **class** Entry<K,V> **extends** HashMap.Entry<K,V> {
11. Entry<K,V> before, after;
12. ……
13. }

HashMap.Entry:

1. **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {
2. **final** K key;
3. V value;
4. Entry<K,V> next;
5. **final** **int** hash;
7. Entry(**int** h, K k, V v, Entry<K,V> n) {
8. value = v;
9. next = n;
10. key = k;
11. hash = h;
12. }
13. }

### ConcurrentHashMap和Hashtable的区别

主要区别就是围绕着锁的粒度以及如何锁



左边便是Hashtable的实现方式---锁整个hash表；而右边则是ConcurrentHashMap的实现方式---锁桶（或段）。ConcurrentHashMap将hash表分为16个桶（默认值），诸如get,put,remove等常用操作只锁当前需要用到的桶。试想，原来只能一个线程进入，现在却能同时16个写线程进入（写线程才需要锁定，而读线程几乎不受限制，之后会提到），并发性的提升是显而易见的。

    更令人惊讶的是ConcurrentHashMap的读取并发，因为在读取的大多数时候都没有用到锁定，所以读取操作几乎是完全的并发操作，而写操作锁定的粒度又非常细，比起之前又更加快速

ConcurrentHashMap使用了不同于传统集合的快速失败迭代器的另一种迭代方式，我们称为弱一致迭代器。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据从而不影响原有的数据，iterator完成后再将头指针替换为新的数据，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。

hashMap是使用快速失败的，，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

**hashMap和linkedhashMap都允许null key和null value，treeMap不允许null key，但允许null value，而ConcurrentHashMap既不允许null key也不允许null value，why**

### ConcurrentHashMap源码解析

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



### HashEntry 类

HashEntry 用来封装散列映射表中的键值对。在 HashEntry 类中，key，hash 和 next 域都被声明为 final 型，value 域被声明为 volatile 型。

##### 清单 1.HashEntry 类的定义

static final class HashEntry<K,V> {

final K key; // 声明 key final 型

final int hash; // 声明 hash 值为 final 型

volatile V value; // 声明 value 为 volatile 型

final HashEntry<K,V> next; // 声明 next 为 final 型

HashEntry(K key, int hash, HashEntry<K,V> next, V value) {

this.key = key;

this.hash = hash;

this.next = next;

this.value = value;

}

}

在 ConcurrentHashMap 中，在散列时如果产生“碰撞”，将采用“分离链接法”来处理“碰撞”：把“碰撞”的 HashEntry 对象链接成一个链表。由于 HashEntry 的 next 域为 final 型，所以新节点只能在链表的表头处插入。 下图是在一个空桶中依次插入 A，B，C 三个 HashEntry 对象后的结构图：

图 1. 插入三个节点后桶的结构示意图：

注意：由于只能在表头插入，所以链表中节点的顺序和插入的顺序相反。

### Segment 类

Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个 Segment 对象用来守护其（成员对象 table 中）包含的若干个桶。

table 是一个由 HashEntry 对象组成的数组。table 数组的每一个数组成员就是散列映射表的一个桶。

count 变量是一个计数器，它表示每个 Segment 对象管理的 table 数组（若干个 HashEntry 组成的链表）包含的 HashEntry 对象的个数。每一个 Segment 对象都有一个 count 对象来表示本 Segment 中包含的 HashEntry 对象的总数。注意，之所以在每个 Segment 对象中包含一个计数器，而不是在

ConcurrentHashMap 中使用全局的计数器，是为了避免出现“热点域”而影响 ConcurrentHashMap 的并发性。

##### 清单 2.Segment 类的定义

static final class Segment<K,V> extends ReentrantLock implements Serializable {

/\*\*

\* 在本 segment 范围内，包含的 HashEntry 元素的个数

\* 该变量被声明为 volatile 型

\*/

transient volatile int count;

/\*\*

\* table被更新的次数（相当于ABA问题中的版本 号）

\*/

transient int modCount;

/\*\*

\* 当 table 中包含的 HashEntry 元素的个数超过本变量值时，触发 table 的再散列

\*/

transient int threshold;

/\*\*

\* table 是由 HashEntry 对象组成的数组

\* 如果散列时发生碰撞，碰撞的 HashEntry 对象就以链表的形式链接成一个链表

\* table 数组的数组成员代表散列映射表的一个桶

\* 每个 table 守护整个 ConcurrentHashMap 包含桶总数的一部分

\* 如果并发级别为 16，table 则守护 ConcurrentHashMap 包含的桶总数的 1/16

\*/

transient volatile HashEntry<K,V>[] table;

/\*\*

\* 装载因子

\*/

final float loadFactor;

Segment(int initialCapacity, float lf) {

loadFactor = lf;

setTable(HashEntry.<K,V>newArray(initialCapacity));

}

/\*\*

\* 设置 table 引用到这个新生成的 HashEntry 数组

\* 只能在持有锁或构造函数中调用本方法

\*/

void setTable(HashEntry<K,V>[] newTable) {

// 计算临界阀值为新数组的长度与装载因子的乘积

threshold = (int)(newTable.length \* loadFactor);

table = newTable;

}

/\*\*

\* 根据 key 的散列值，找到 table 中对应的那个桶（table 数组的某个数组成员）

\*/

HashEntry<K,V> getFirst(int hash) {

HashEntry<K,V>[] tab = table;

// 把散列值与 table 数组长度减 1 的值相“与”，

// 得到散列值对应的 table 数组的下标

// 然后返回 table 数组中此下标对应的 HashEntry 元素

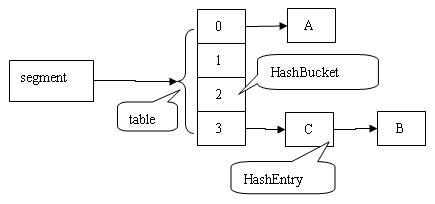
return tab[hash & (tab.length - 1)];

}

}

下图是依次插入 ABC 三个 HashEntry 节点后，Segment 的结构示意图。

##### 插入三个节点后 Segment 的结构示意图：



### ConcurrentHashMap 类

ConcurrentHashMap 在默认并发级别会创建包含 16 个 Segment 对象的数组。每个 Segment 的成员对象 table 包含若干个散列表的桶。每个桶是由 HashEntry 链接起来的一个链表。如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。

##### 清单 3.ConcurrentHashMap 类的定义

public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>

implements ConcurrentMap<K, V>, Serializable {

/\*\*

\* 散列映射表的默认初始容量为 16，即初始默认为 16 个桶

\* 在构造函数中没有指定这个参数时，使用本参数

\*/

static final int DEFAULT\_INITIAL\_CAPACITY= 16;

/\*\*

\* 散列映射表的默认装载因子为 0.75，该值是 table 中包含的 HashEntry 元素的个数与

\* table 数组长度的比值

\* 当 table 中包含的 HashEntry 元素的个数超过了 table 数组的长度与装载因子的乘积时，

\* 将触发 再散列

\* 在构造函数中没有指定这个参数时，使用本参数

\*/

static final float DEFAULT\_LOAD\_FACTOR= 0.75f;

/\*\*

\* 散列表的默认并发级别为 16。该值表示当前更新线程的估计数

\* 在构造函数中没有指定这个参数时，使用本参数

\*/

static final int DEFAULT\_CONCURRENCY\_LEVEL= 16;

/\*\*

\* segments 的掩码值

\* key 的散列码的高位用来选择具体的 segment

\*/

final int segmentMask;

/\*\*

\* 偏移量

\*/

final int segmentShift;

/\*\*

\* 由 Segment 对象组成的数组

\*/

final Segment<K,V>[] segments;

/\*\*

\* 创建一个带有指定初始容量、加载因子和并发级别的新的空映射。

\*/

public ConcurrentHashMap(int initialCapacity,

float loadFactor, int concurrencyLevel) {

if(!(loadFactor > 0) || initialCapacity < 0 ||

concurrencyLevel <= 0)

throw new IllegalArgumentException();

if(concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

// 寻找最佳匹配参数（不小于给定参数的最接近的 2 次幂）

int sshift = 0;

int ssize = 1;

while(ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

segmentShift = 32 - sshift; // 偏移量值

segmentMask = ssize - 1; // 掩码值

this.segments = Segment.newArray(ssize); // 创建数组

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

int c = initialCapacity / ssize;

if(c \* ssize < initialCapacity)

++c;

int cap = 1;

while(cap < c)

cap <<= 1;

// 依次遍历每个数组元素

for(int i = 0; i < this.segments.length; ++i)

// 初始化每个数组元素引用的 Segment 对象

this.segments[i] = new Segment<K,V>(cap, loadFactor);

}

/\*\*

\* 创建一个带有默认初始容量 (16)、默认加载因子 (0.75) 和 默认并发级别 (16)

\* 的空散列映射表。

\*/

public ConcurrentHashMap() {

// 使用三个默认参数，调用上面重载的构造函数来创建空散列映射表

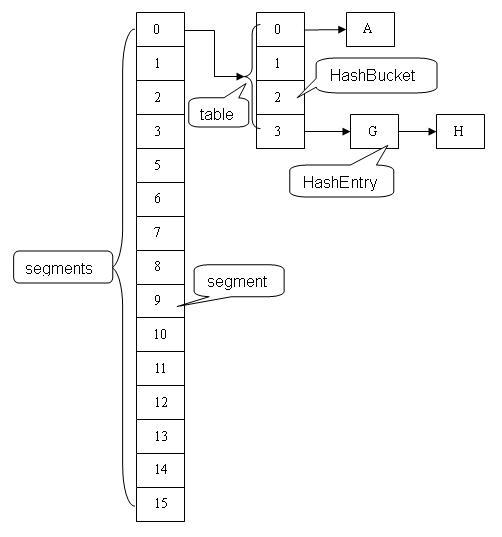
this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_LOAD\_FACTOR, DEFAULT\_CONCURRENCY\_LEVEL);

}

}

下面是 ConcurrentHashMap 的结构示意图。

##### ConcurrentHashMap 的结构示意图：



## 用分离锁实现多个线程间的并发写操作

在 ConcurrentHashMap 中，线程对映射表做读操作时，一般情况下不需要加锁就可以完成，对容器做结构性修改的操作才需要加锁。下面以 put 操作为例说明对 ConcurrentHashMap 做结构性修改的过程。

首先，根据 key 计算出对应的 hash 值：

##### 清单 4.Put 方法的实现

public V put(K key, V value) {

if (value == null) //ConcurrentHashMap 中不允许用 null 作为映射值

throw new NullPointerException();

int hash = hash(key.hashCode()); // 计算键对应的散列码

// 根据散列码找到对应的 Segment

return segmentFor(hash).put(key, hash, value, false);

}

然后，根据 hash 值找到对应的

Segment 对象：

##### 清单 5.根据 hash 值找到对应的 Segment

/\*\*

\* 使用 key 的散列码来得到 segments 数组中对应的 Segment

\*/

final Segment<K,V> segmentFor(int hash) {

// 将散列值右移 segmentShift 个位，并在高位填充 0

// 然后把得到的值与 segmentMask 相“与”

// 从而得到 hash 值对应的 segments 数组的下标值

// 最后根据下标值返回散列码对应的 Segment 对象

return segments[(hash >>> segmentShift) & segmentMask];

}

最后，在这个 Segment 中执行具体的 put 操作：

##### 清单 6.在 Segment 中执行具体的 put 操作

V put(K key, int hash, V value, boolean onlyIfAbsent) {

lock(); // 加锁，这里是锁定某个 Segment 对象而非整个 ConcurrentHashMap

try {

int c = count;

if (c++ > threshold) // 如果超过再散列的阈值

rehash(); // 执行再散列，table 数组的长度将扩充一倍

HashEntry<K,V>[] tab = table;

// 把散列码值与 table 数组的长度减 1 的值相“与”

// 得到该散列码对应的 table 数组的下标值

int index = hash & (tab.length - 1);

// 找到散列码对应的具体的那个桶

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

while (e != null && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue;

if (e != null) { // 如果键 / 值对以经存在

oldValue = e.value;

if (!onlyIfAbsent)

e.value = value; // 设置 value 值

}

else { // 键 / 值对不存在

oldValue = null;

++modCount; // 要添加新节点到链表中，所以 modCont 要加 1

// 创建新节点，并添加到链表的头部

tab[index] = new HashEntry<K,V>(key, hash, first, value);

count = c; // 写 count 变量

}

return oldValue;

} finally {

unlock(); // 解锁

}

}

注意：这里的加锁操作是针对（键的 hash 值对应的）某个具体的 Segment，锁定的是该 Segment 而不是整个 ConcurrentHashMap

。因为插入键 / 值对操作只是在这个 Segment 包含的某个桶中完成，不需要锁定整个

ConcurrentHashMap。

此时，其他写线程对另外 15 个

Segment 的加锁并不会因为当前线程对这个 Segment 的加锁而阻塞。同时，所有读线程几乎不会因本线程的加锁而阻塞（除非读线程刚好读到这个 Segment 中某个

HashEntry 的 value 域的值为 null，此时需要加锁后重新读取该值

）。

相比较于

HashTable 和由同步包装器包装的 HashMap

每次只能有一个线程执行读或写操作，

ConcurrentHashMap 在并发访问性能上有了质的提高。在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设置为 16），及任意数量线程的读操作。

## 用 HashEntery 对象的不变性来降低读操作对加锁的需求

在代码清单“HashEntry 类的定义”中我们可以看到，HashEntry 中的 key，hash，next 都声明为 final 型。这意味着，不能把节点添加到链接的中间和尾部，也不能在链接的中间和尾部删除节点。这个特性可以保证：在访问某个节点时，这个节点之后的链接不会被改变。这个特性可以大大降低处理链表时的复杂性。

同时，HashEntry 类的 value 域被声明为 Volatile 型，Java 的内存模型可以保证：某个写线程对 value 域的写入马上可以被后续的某个读线程“看”到。在 ConcurrentHashMap 中，不允许用 unll 作为键和值，当读线程读到某个 HashEntry 的 value 域的值为 null 时，便知道产生了冲突——发生了重排序现象，需要加锁后重新读入这个 value 值。这些特性互相配合，使得读线程即使在不加锁状态下，也能正确访问 ConcurrentHashMap。

下面我们分别来分析线程写入的两种情形：对散列表做非结构性修改的操作和对散列表做结构性修改的操作。

非结构性修改操作只是更改某个 HashEntry 的 value 域的值。由于对 Volatile 变量的写入操作将与随后对这个变量的读操作进行同步。当一个写线程修改了某个 HashEntry 的 value 域后，另一个读线程读这个值域，Java 内存模型能够保证读线程读取的一定是更新后的值。所以，写线程对链表的非结构性修改能够被后续不加锁的读线程“看到”。

对 ConcurrentHashMap 做结构性修改，实质上是对某个桶指向的链表做结构性修改。如果能够确保：在读线程遍历一个链表期间，写线程对这个链表所做的结构性修改不影响读线程继续正常遍历这个链表。那么读 / 写线程之间就可以安全并发访问这个 ConcurrentHashMap。

结构性修改操作包括 put，remove，clear。下面我们分别分析这三个操作。

clear 操作只是把 ConcurrentHashMap 中所有的桶“置空”，每个桶之前引用的链表依然存在，只是桶不再引用到这些链表（所有链表的结构并没有被修改）。正在遍历某个链表的读线程依然可以正常执行对该链表的遍历。

从上面的代码清单“在 Segment 中执行具体的 put 操作”中，我们可以看出：put 操作如果需要插入一个新节点到链表中时 , 会在链表头部插入这个新节点。此时，链表中的原有节点的链接并没有被修改。也就是说：插入新健 / 值对到链表中的操作不会影响读线程正常遍历这个链表。

下面来分析 remove 操作，先让我们来看看 remove 操作的源代码实现。

##### 清单 7.remove 操作

V remove(Object key, int hash, Object value) {

lock(); // 加锁

try{

int c = count - 1;

HashEntry<K,V>[] tab = table;

// 根据散列码找到 table 的下标值

int index = hash & (tab.length - 1);

// 找到散列码对应的那个桶

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

while(e != null&& (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue = null;

if(e != null) {

V v = e.value;

if(value == null|| value.equals(v)) { // 找到要删除的节点

oldValue = v;

++modCount;

// 所有处于待删除节点之后的节点原样保留在链表中

// 所有处于待删除节点之前的节点被克隆到新链表中

HashEntry<K,V> newFirst = e.next;// 待删节点的后继结点

for(HashEntry<K,V> p = first; p != e; p = p.next)

newFirst = new HashEntry<K,V>(p.key, p.hash,

newFirst, p.value);

// 把桶链接到新的头结点

// 新的头结点是原链表中，删除节点之前的那个节点

tab[index] = newFirst;

count = c; // 写 count 变量

}

}

return oldValue;

} finally{

unlock(); // 解锁

}

}

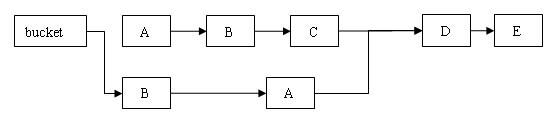
和 get 操作一样，首先根据散列码找到具体的链表；然后遍历这个链表找到要删除的节点；最后把待删除节点之后的所有节点原样保留在新链表中，把待删除节点之前的每个节点克隆到新链表中。下面通过图例来说明 remove 操作。

假设写线程执行 remove 操作，要删除链表的 C 节点，另一个读线程同时正在遍历这个链表。

##### 图 4. 执行删除之前的原链表：

图 4. 执行删除之前的原链表：

##### 图 5. 执行删除之后的新链表



从上图可以看出，删除节点 C 之后的所有节点原样保留到新链表中；删除节点 C 之前的每个节点被克隆到新链表中，注意：它们在新链表中的链接顺序被反转了。

在执行 remove 操作时，原始链表并没有被修改，也就是说：读线程不会受同时执行 remove 操作的并发写线程的干扰。

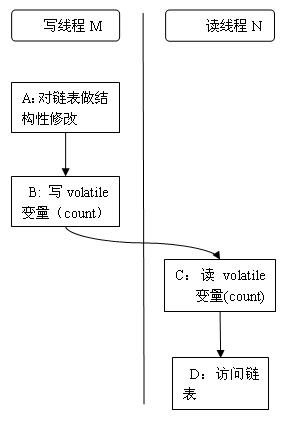
综合上面的分析我们可以看出，写线程对某个链表的结构性修改不会影响其他的并发读线程对这个链表的遍历访问。

## 用 Volatile 变量协调读写线程间的内存可见性

由于内存可见性问题，未正确同步的情况下，写线程写入的值可能并不为后续的读线程可见。

下面以写线程 M 和读线程 N 来说明 ConcurrentHashMap 如何协调读 / 写线程间的内存可见性问题。

##### 图 6. 协调读 – 写线程间的内存可见性的示意图：



假设线程 M 在写入了 volatile 型变量 count 后，线程 N 读取了这个 volatile 型变量 count。

根据 happens-before 关系法则中的程序次序法则，A appens-before 于 B，C happens-before D。

根据 Volatile 变量法则，B happens-before C。

根据传递性，连接上面三个 happens-before 关系得到：A appens-before 于 B； B appens-before C；C happens-before D。也就是说：写线程 M 对链表做的结构性修改，在读线程 N 读取了同一个 volatile 变量后，对线程 N 也是可见的了。

虽然线程 N 是在未加锁的情况下访问链表。Java 的内存模型可以保证：只要之前对链表做结构性修改操作的写线程 M 在退出写方法前写 volatile 型变量 count，读线程 N 在读取这个 volatile 型变量 count 后，就一定能“看到”这些修改。

ConcurrentHashMap 中，每个 Segment 都有一个变量 count。它用来统计 Segment 中的 HashEntry 的个数。这个变量被声明为 volatile。

##### 清单 8.Count 变量的声明

transient volatile int count;

所有不加锁读方法，在进入读方法时，首先都会去读这个 count 变量。比如下面的 get 方法：

##### 清单 9.get 操作

V get(Object key, int hash) {

if(count != 0) { // 首先读 count 变量

HashEntry<K,V> e = getFirst(hash);

while(e != null) {

if(e.hash == hash && key.equals(e.key)) {

V v = e.value;

if(v != null)

return v;

// 如果读到 value 域为 null，说明发生了重排序，加锁后重新读取

return readValueUnderLock(e);

}

e = e.next;

}

}

return null;

}

在 ConcurrentHashMap 中，所有执行写操作的方法（put, remove, clear），在对链表做结构性修改之后，在退出写方法前都会去写这个 count 变量。所有未加锁的读操作（get, contains, containsKey）在读方法中，都会首先去读取这个 count 变量。

根据 Java 内存模型，对 同一个 volatile 变量的写 / 读操作可以确保：写线程写入的值，能够被之后未加锁的读线程“看到”。

这个特性和前面介绍的 HashEntry 对象的不变性相结合，使得在 ConcurrentHashMap 中，读线程在读取散列表时，基本不需要加锁就能成功获得需要的值。这两个特性相配合，不仅减少了请求同一个锁的频率（读操作一般不需要加锁就能够成功获得值），也减少了持有同一个锁的时间（只有读到 value 域的值为 null 时 , 读线程才需要加锁后重读）。

空值的唯一源头就是HashEntry中的默认值，因为 HashEntry中的value不是final的，非同步读取有可能读取到空值。

仔细看下put操作的语句：tab[index] = new HashEntry<K,V>(key, hash, first, value)，在这条语句中，HashEntry构造函数中对value的赋值以及对tab[index]的赋值可能被重新排序（即先对tab[index]赋值，再对value赋值），这就可能导致结点的值为空。

这种情况应当很罕见，一旦发生这种情况，ConcurrentHashMap采取的方式是在持有锁的情况下再读一遍，这能够保证读到最新的值，并且一定不会为空值。  
1. V readValueUnderLock(HashEntry<K,V> e) {   
2. lock();   
3. try {   
4. return e.value;   
5. } finally {   
6. unlock();   
7. }   
8. }

## ConcurrentHashMap 实现高并发的总结

### 基于通常情形而优化

在实际的应用中，散列表一般的应用场景是：除了少数插入操作和删除操作外，绝大多数都是读取操作，而且读操作在大多数时候都是成功的。正是基于这个前提，ConcurrentHashMap 针对读操作做了大量的优化。通过 HashEntry 对象的不变性和用 volatile 型变量协调线程间的内存可见性，使得 大多数时候，读操作不需要加锁就可以正确获得值。这个特性使得 ConcurrentHashMap 的并发性能在分离锁的基础上又有了近一步的提高。

### 总结

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于

HashTable 和

用同步包装器包装的 HashMap（Collections.synchronizedMap(new HashMap())），ConcurrentHashMap 拥有更高的并发性。在

HashTable 和由同步包装器包装的 HashMap 中，使用一个全局的锁来同步不同线程间的并发访问。同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器。这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成

<em>串行化</em>

的了。

在使用锁来协调多线程间并发访问的模式下，减小对锁的竞争可以有效提高并发性。有两种方式可以减小对锁的竞争：

1. 减小请求 同一个锁的 频率。
2. 减少持有锁的 时间。

ConcurrentHashMap 的高并发性主要来自于三个方面：

1. 用分离锁实现多个线程间的更深层次的共享访问。
2. 用 HashEntery 对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。
3. 通过对同一个 Volatile 变量的写 / 读访问，协调不同线程间读 / 写操作的内存可见性。

使用分离锁，减小了请求 同一个锁的频率。

通过 HashEntery 对象的不变性及对同一个 Volatile 变量的读 / 写来协调内存可见性，使得 读操作大多数时候不需要加锁就能成功获取到需要的值。由于散列映射表在实际应用中大多数操作都是成功的 读操作，所以 2 和 3 既可以减少请求同一个锁的频率，也可以有效减少持有锁的时间。

通过减小请求同一个锁的频率和尽量减少持有锁的时间

，使得 ConcurrentHashMap 的并发性相对于 HashTable 和

用同步包装器包装的 HashMap

有了质的提高。

### synchronized 和lock的优缺点

**lock的优点**

* 可以控制线程得到锁的顺序,也就是有公平锁(按照进入顺序得到资源),也可以不按照顺就像.synchronized 一样.
* 可以查看锁的状态, 锁是否被锁上了.
* 可以查看当前有多少线程再等待锁.
* 可以被中断的同步（synchronized的同步是不能Interrupt的）
* 具有读写锁
* 通过多次newCondition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal();而 synchronized 只有一个条件变量，也就是锁对象的 wait 和 notifyAll
* ReentrantLock提供了可轮询的锁请求，他可以尝试的去取得锁，如果取得成功则继续处理，取得不成功，可以等下次运行的时候处理，所以不容易产生死锁，而synchronized则一旦进入锁请求要么成功，要么一直阻塞，所以更容易产生死锁。

**reentrantlock的缺点**

* synchronized可以放在方法的定义里面, 而reentrantlock只能放在块里面. 比较起来, synchronized可以减少嵌套
* ReentrantLock必须在finally中释放锁，否则后果很严重，编码角度来说使用synchronized更加简单，不容易遗漏或者出错。

### Volatile关键词

### 一个变量被定义为volatile之后，具有两个特性:

### 第一个为禁止指令重排序优化

### 第二个为可见性，当一个线程修改了这个变量的值，新值对于其他变量来说是可以立即得知的，但是对变量的操作可能是非原子性，从而不能保证并发下是安全的。

### 如果不是在只有一个线程修改变量的情况或对变量的修改不依赖于变量的原值，比如给变量赋一个新的值，

### 其他情况就仍然要加锁

### Volatile的规则：

### 

### 进程和线程的区别

1. 进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

1. 在进程切换时，耗费资源较大，效率要差一些，开销大

### 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉（我的理解是因为地址空间用的是同一个，所以程序无法继续运行），所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

### 总的来说就是：进程有独立的地址空间，线程没有单独的地址空间（同一进程内的线程共享进程的地址空间）

### 一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。

### 线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。

### 进程间通信的方式

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，管道可用于具有亲缘关系的父子进程间的通信，
2. 有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。
3. 信号（signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一致的。
4. 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。
5. 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，可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。
6. 信号量（semaphore）：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。
7. 套接字（socket）：这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

　进程通信，是指进程之间的信息交换（信息量少则一个状态或数值，多者则是成千上万个字节）。因此，对于用信号量进行的进程间的互斥和同步，由于其所交换的信息量少而被归结为低级通信。

　　所谓高级进程通信指：用户可以利用操作系统所提供的一组通信命令传送大量数据的一种通信方式。操作系统隐藏了进程通信的实现细节。或者说，通信过程对用户是透明的。

　　高级通信机制可归结为三大类：

　　（1）共享存储器系统（存储器中划分的共享存储区）；实际操作中对应的是“剪贴板”（剪贴板实际上是系统维护管理的一块内存区域）的通信方式，比如举例如下：word进程按下ctrl+c，在ppt进程按下ctrl+v，即完成了word进程和ppt进程之间的通信，复制时将数据放入到剪贴板，粘贴时从剪贴板中取出数据，然后显示在ppt窗口上。

　　（2）消息传递系统（进程间的数据交换以消息（message）为单位，当今最流行的微内核操作系统中，微内核与服务器之间的通信，无一例外地都采用了消息传递机制。应用举例：邮槽（MailSlot）是基于广播通信体系设计出来的，它采用无连接的不可靠的数据传输。邮槽是一种单向通信机制，创建邮槽的服务器进程读取数据，打开邮槽的客户机进程写入数据。

　　（3）管道通信系统（管道即：连接读写进程以实现他们之间通信的共享文件（pipe文件，类似先进先出的队列，由一个进程写，另一进程读））。实际操作中，管道分为：匿名管道、命名管道。匿名管道是一个未命名的、单向管道，通过父进程和一个子进程之间传输数据。匿名管道只能实现本地机器上两个进程之间的通信，而不能实现跨网络的通信。命名管道不仅可以在本机上实现两个进程间的通信，还可以跨网络实现两个进程间的通信。

### 产生死锁的必要条件

(1)互斥（mutualexclusion），一个资源每次只能被一个进程使用；

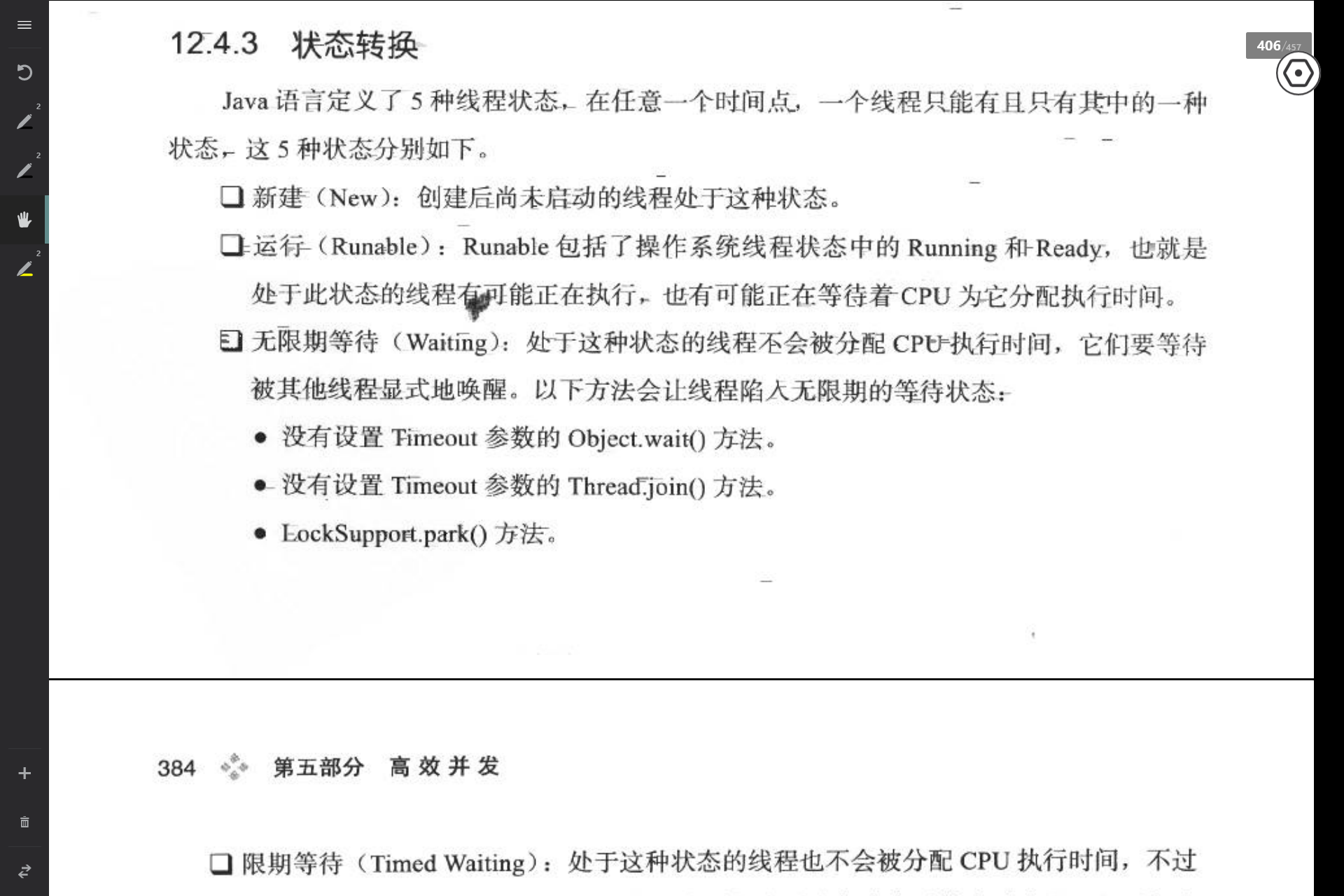
(2)不可抢占（nopreemption），进程已获得的资源，在未使用完之前，不能强行剥夺；

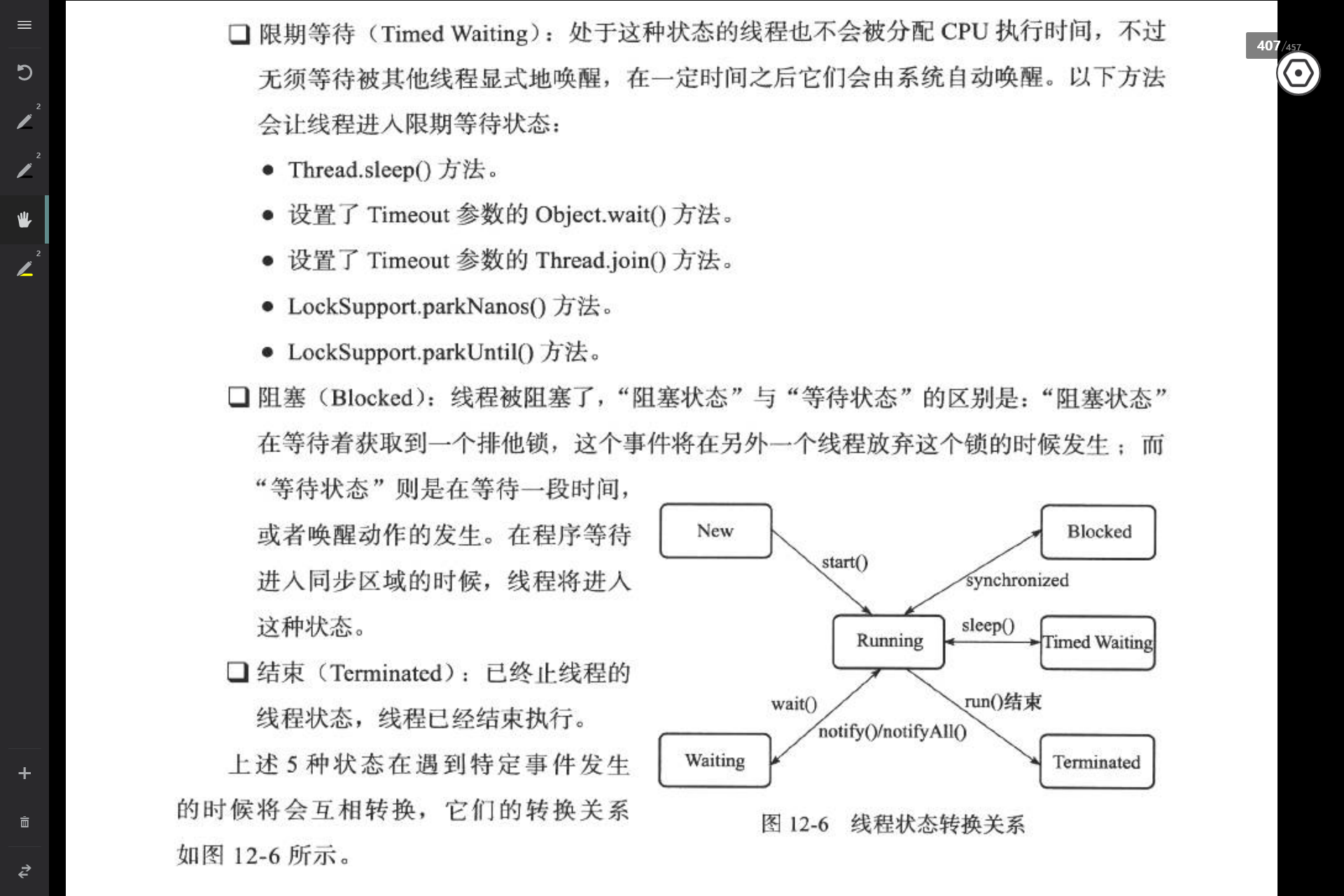
(3)占有并等待（hold andwait），一个进程因请求资源而阻塞时，对已获得的资源保持不放；

(4循环等待（circularwait），若干进程之间形成一种首尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

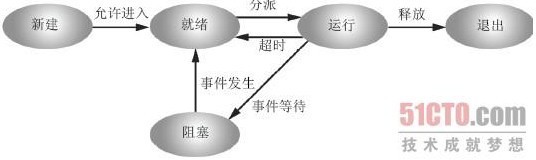
### JAVA五种线程状态





### 操作系统状态转换

**进程五状态**



1)       新状态：进程已经创建，但未被OS接纳为可执行进程。（还没有申请到相应的资源）。

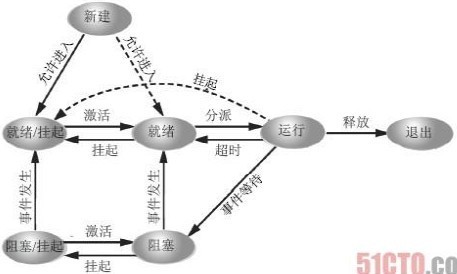
2)       就绪态：进程做好了准备，准备执行（只等待处理机）。

3)       执行状态：该进程正在执行（单处理机，某一时刻仅一个进程占用处理机）。

4)       阻塞状态：等待某事件发生才能执行，如等待I/O完成。

5)       终止状态.

**进程七状态**



l  阻塞 → 阻塞/挂起：OS通常将阻塞进程换出，以腾出内存空间

l  阻塞/挂起→ 就绪/挂起：当阻塞/挂起进程等待的事件发生时，可以将其转换为就绪/挂起。

l  就绪/挂起→ 就绪：OS需要调入一个进程执行。

l  就绪 → 就绪/挂起：一般，OS挂起阻塞进程。但是有时也会挂起就绪进程，释放足够的内存空间。

l  新 → 就绪/挂起（新→ 就绪）：新进程创建后，可以插入到就绪队列或就绪，挂起队列，若无足够的内存分配给新进程，则需要新→ 就绪/挂起。

### 用户级线程和内核级线程

**用户线程**指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。不需要用户态/核心态切换，速度快，操作系统内核不知道多线程的存在，因此一个线程阻塞将使得整个进程（包括它的所有线程）阻塞。由于这里的处理器时间片分配是以进程为基本单位，所以每个线程执行的时间相对减少。

**内核线程**：由操作系统内核创建和撤销。内核维护进程及线程的上下文信息以及线程切换。一个内核线程由于I/O操作而阻塞，不会影响其它线程的运行。

**用户级线程和内核级线程的区别：**

**（1）**内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的。

**（2）**用户级线程的创建、撤消和调度不需要OS内核的支持，是在语言（如[**Java**](http://lib.csdn.net/base/17)）这一级处理的；而内核支持线程的创建、撤消和调度都需OS内核提供支持，而且与进程的创建、撤消和调度大体是相同的。

**（3）**用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。

**（4）**在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内核支持线程的系统内，CPU调度则以线程为单位，由OS的线程调度程序负责线程的调度。

### 分段和分页的区别

### 页是信息的物理单位。

### 段是信息的逻辑单位

1. **页的大小固定且由系统决定；而段的长度却不固定，决定于用户所编写的**[**程序**](http://www.xuebuyuan.com/)**。**
2. 分页的地址空间是一维的，程序员只需利用一个记忆符，即可表示一个地址；而分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。（分段系统，各段长度不相等，整个作业的地址空间分成了多个段，地址由段名和段内地址组成，程序在标识地址时是需给出段名和偏移量（不同的段是不同的）所以说是二维的。  
   分页系统中页号和页内地址是由系统决定好的，所以只用提供一个地址就可计算得出，所以是一维的）
3. 分页会出现内部碎片，分段会出现外部碎片

### 什么是虚拟内存

基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其余部分留在外存，就可以启动程序执行。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存,然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换出到外存上，从而腾出空间存放将要调入内存的信息。这样，系统好像为用户提供了一个比实际内存大得多的存储器，称为虚拟存储器。

### 什么是进程互斥同步

### 互斥：是指散步在不同任务之间的若干程序片断，当某个任务运行其中一个程序片段时，其它任务就不能运行它们之中的任一程序片段，只能等到该任务运行完这个程序片段后才可以运行

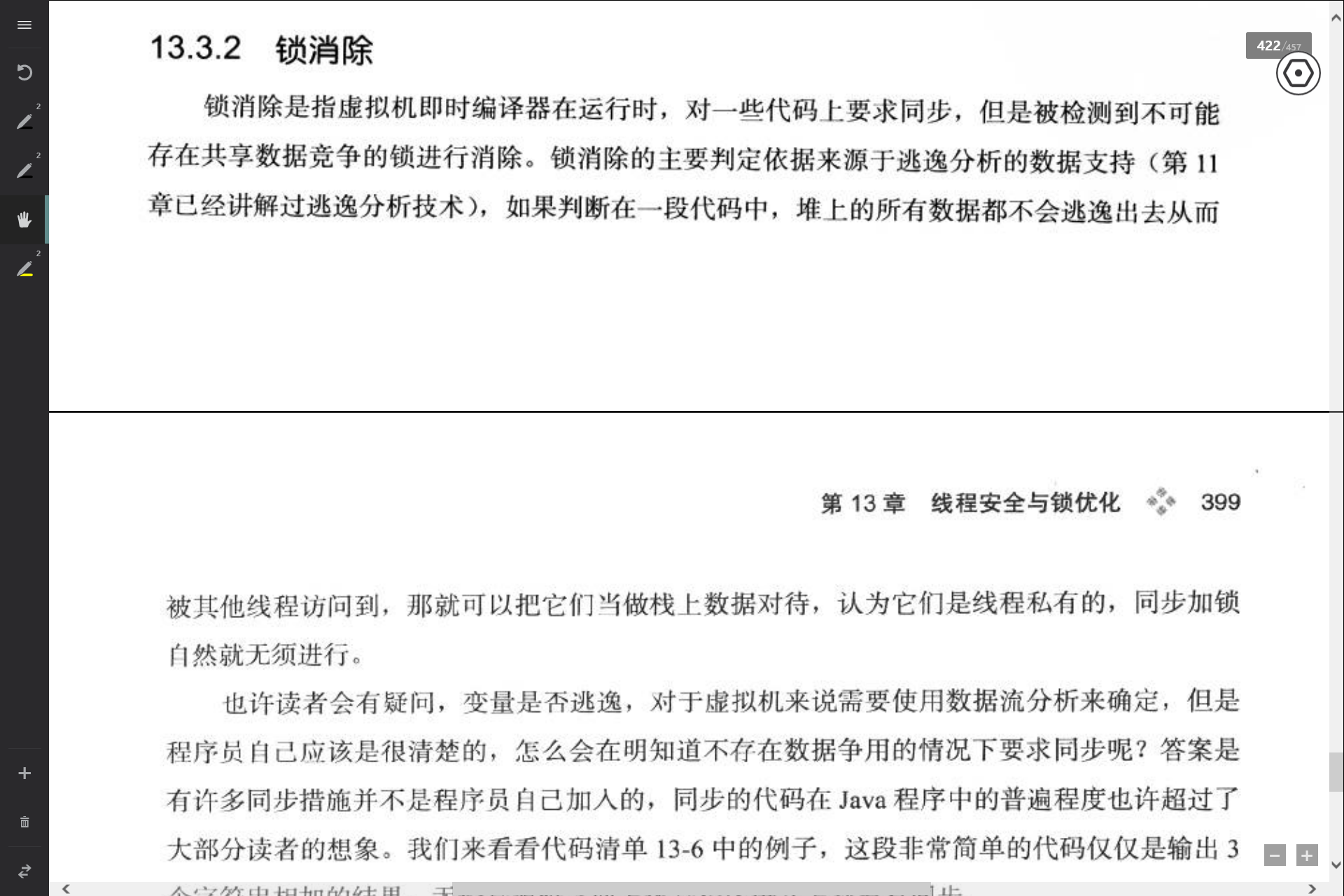
**同步：是指散步在不同任务之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务，最基本的场景就是任务之间的依赖，比如A任务的运行依赖于B任务产生的数据。**

**显然，同步是一种更为复杂的互斥，而互斥是一种特殊的同步。**

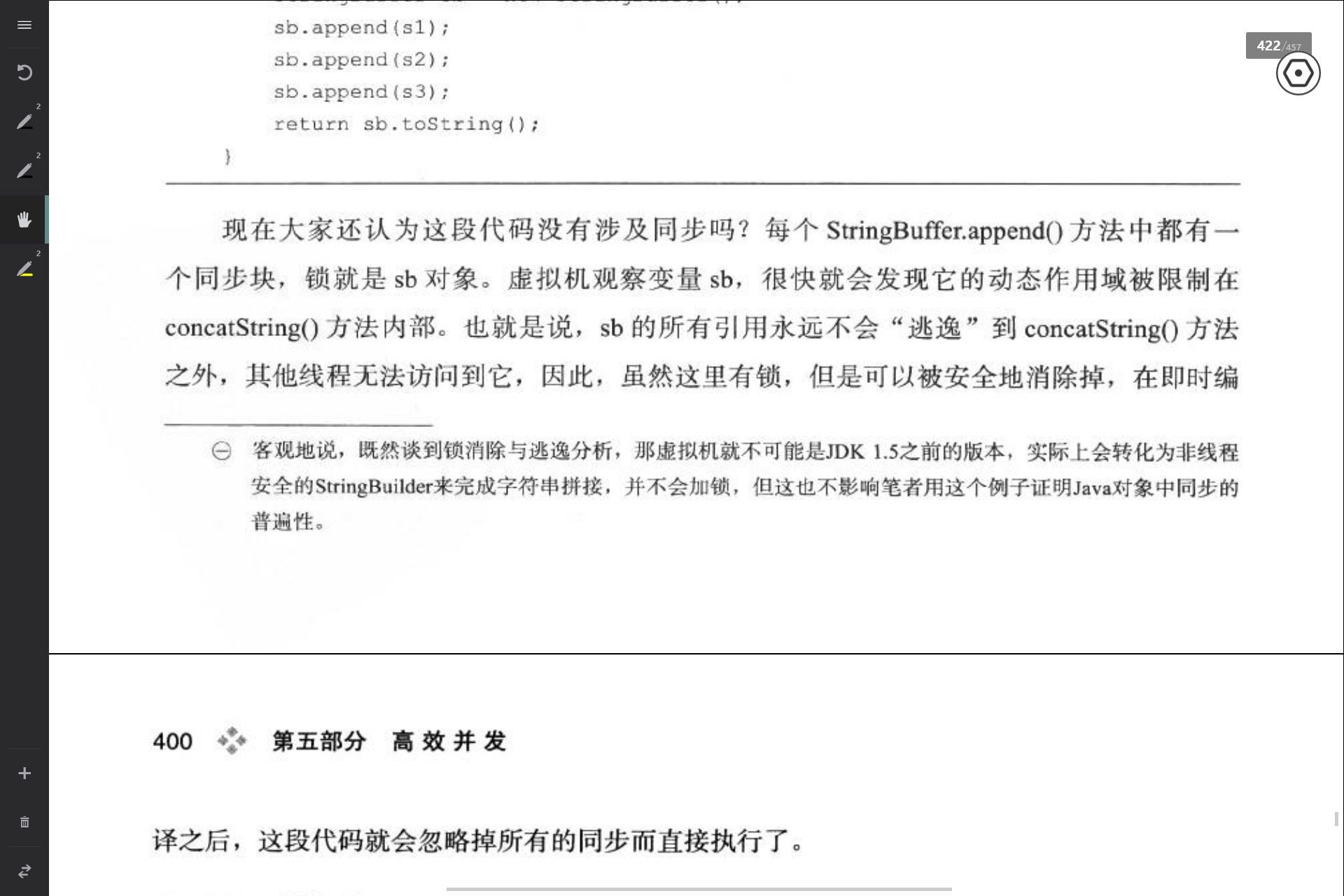
**也就是说互斥是两个任务之间不可以同时运行，他们会相互排斥，必须等待一个线程运行完毕，另一个才能运行，而同步也是不能同时运行，但他是必须要安照某种次序来运行相应的线程（也是一种互斥）！因此互斥具有唯一性和排它性，但互斥并不限制任务的运行顺序，即任务是无序的。而同步的任务之间则有顺序关系。**

### 什么是自旋锁

### 什么是锁消除







### Struts2的工作原理

工作原理

在Struts2框架中的处理大概分为以下几个步骤   
1 客户端初始化一个指向Servlet容器（例如Tomcat）的请求   
2 这个请求经过一系列的过滤器（Filter）   
3 接着FilterDispatcher被调用，FilterDispatcher询问ActionMapper来决定这个请是否需要调用某个Action   
4 如果ActionMapper决定需要调用某个Action，FilterDispatcher把请求的处理交给ActionProxy   
5 ActionProxy通过Configuration Manager询问框架的配置文件，找到需要调用的Action类   
6 ActionProxy创建一个ActionInvocation的实例。   
7 ActionInvocation实例使用代理模式来调用相应的action，   
8 一旦Action执行完毕，ActionInvocation负责根据struts.xml中的配置找到对应的返回结果。返回结果通常是（但不总是，也可 能是另外的一个Action链）一个需要被表示的JSP或者FreeMarker的模版。

### Hibernate操作过程

### 

### Hibernate如何优化

### 

### Hibernate缓存机制

### 

### mysql优化查询的方法

### .对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 的列上建立索引。（并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。）

### 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。

### 避免使用like语句，会进行全表扫描，

### select id from t where name like '%abc%' 若要提高效率，可以考虑全文检索。

MySQL中全文索引的关键字为FULLTEXT，目前可对MyISAM表和InnoDB表的CHAR、VARCHAR、TEXT类型的列创建全文索引。全文索引同其他索引一样，可在创建表是由CREATE TABLE语句创建也可以在表创建之后用ALTER TABLE或者CREATE INDEX命令创建(对于要导入大量数据的表先导入数据再创建FULLTEXT索引比先创建索引后导入数据会更快)。

自然语言全文搜索是MySQL全文搜索的默认搜索方式，实现从一个文本集合中搜索给定的字符串。这里，文本集合指的是指由FULLTEXT索引的一个或者多个列。

**建表，并给title，body字段加FULLTEXT索引**

CREATE TABLE articles (

     id INT UNSIGNED AUTO\_INCREMENT NOT NULL PRIMARY KEY,

     title VARCHAR(200),

     body TEXT,

     FULLTEXT (title,body)

) ENGINE=InnoDB;

**例1：**

SELECT \* FROM articles

WHERE MATCH (title,body)

AGAINST ('database' IN NATURAL LANGUAGE MODE);



可以看到，语句查找到了包含指定内容的行。实际上，返回的行是按与所查找内容的相关度由高到低的顺序排列的。这个相关度的值由WHERE语句中的MATCH (…) AGAINST (…)计算所得，是一个非负浮点数。该值越大表明相应的行与所查找的内容越相关，0值表明不相关。该值基于行中的单词数、行中不重复的单词数、文本集合中总单词数以及含特定单词的行数计算得出。

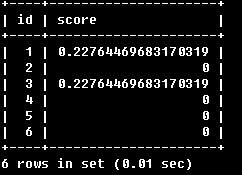
**例2：**

由上例可知MATCH (…) AGAINST (…)实际上会计算一个相关值，可通过下例来验证。

SELECT id, MATCH (title,body)

AGAINST ('Tutorial' IN NATURAL LANGUAGE MODE) AS score

FROM articles;



可以看到，所得结果的第二列即为改行与查找内容的相关度。上例1中所得结果的顺序就是按此相关度排列的。

**例3：**

若想既看到查找到的结果又需要了解具体的相关度，可用下述方法达成。

SELECT id, body, MATCH (title,body) AGAINST

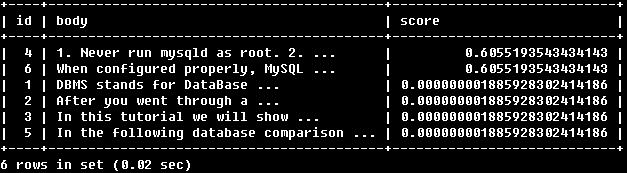
   ('Security implications of running MySQL as root'

   IN NATURAL LANGUAGE MODE) AS score

   FROM articles WHERE MATCH (title,body) AGAINST

   ('Security implications of running MySQL as root'

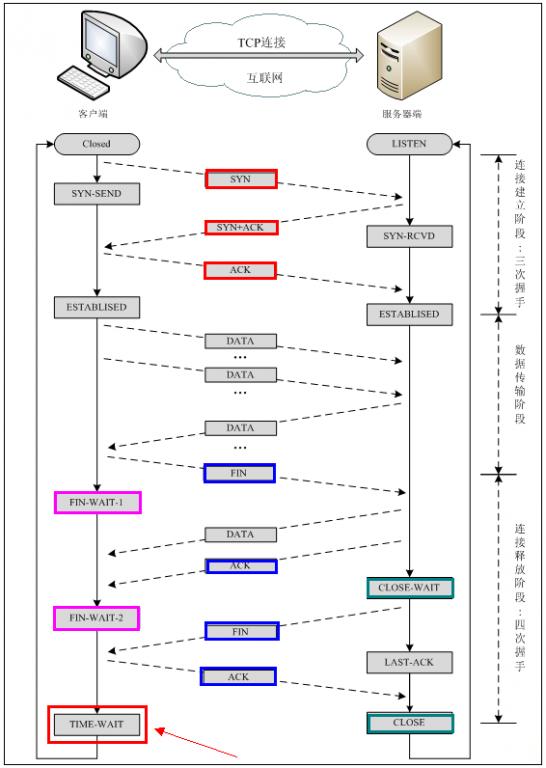
IN NATURAL LANGUAGE MODE);



可以看到，通过在查找部分和条件部分分别使用相同的MATCH(…) AGAINST(…)可以同时获取两方面的内容（不会增加额外开销，优化器知道两个MATCH(…) AGAINST(..)是相同的，只会执行一次该语句）

### TCP三次握手和四次挥手

建立TCP需要三次握手才能建立，而断开连接则需要四次握手。整个过程如下图所示：



**为什么time\_wait需要2\*MSL等待时间？**  
MSL就是maximum segment lifetime(最大分节生命期），这是一个IP数据包能在互联网上生存的最长时间，超过这个时间将在网络中消失。  
假设最终的 ACK 丢失 ， server 将重发 FIN ， client 必须维护 TCP 状态信息以便可以重发最终的 ACK ，否则会发送RST ，结果 server 认为发生错误。   
若要TCP可靠地终止连接的两个方向 ( 全双工关闭 ) ， client 必须进 TIME\_WAIT状态。  
现在我们考虑终止连接时的被动方发送了一个FIN，然后主动方回复了一个ACK，然而这个ACK可能会丢失，这会造成被动方重发FIN，这个FIN可能会在互联网上存活MSL。  
如果没有TIME\_WAIT的话，假设连接1已经断开，然而其被动方最后重发的那个FIN(或者FIN之前发送的任何TCP分段)还在网络上，然而连接2重用了连接1的所有的5元素(源IP，目的IP，TCP，源端口，目的端口)，刚刚将建立好连接，连接1迟到的FIN到达了，这个FIN将以比较低但是确实可能的概率终止掉连接2.

### TCP为什么不能采用两次握手

### 采用三次握手是为了防止失效的连接请求报文段突然又传送到主机B，因而产生错误。失效的连接请求报文段是指：主机A发出的连接请求没有收到主机B的确认，于是经过一段时间后，主机A又重新向主机B发送连接请求，且建立成功，顺序完成数据传输。考虑这样一种特殊情况，主机A第一次发送的连接请求并没有丢失，而是因为网络节点导致延迟达到主机B，主机B以为是主机A又发起的新连接，于是主机B同意连接，并向主机A发回确认，但是此时主机A根本不会理会，主机B就一直在等待主机A发送数据，导致主机B的资源浪费。

### TCP长连接和短连接的区别

**TCP短连接**

我们模拟一下TCP短连接的情况，client向server发起连接请求，server接到请求，然后双方建立连接。client向server发送消息，server回应client，然后一次读写就完成了，这时候双方任何一个都可以发起close操作，不过一般都是client先发起close操作。为什么呢，一般的server不会回复完client后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在client/server间传递一次读写操作

短连接的优点是：管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段

**TCP长连接**

接下来我们再模拟一下长连接的情况，client向server发起连接，server接受client连接，双方建立连接。Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

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

如果一个给定的连接在两小时内没有任何的动作，则服务器就向客户发一个探测报文段，客户主机必须处于以下4个状态之一：

1. 客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。
2. 客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在75秒后超时。服务器总共发送10个这样的探测 ，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。
3. 客户主机崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。
4. 客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探查的响应。

从上面可以看出，TCP保活功能主要为探测长连接的存活状况，不过这里存在一个问题，存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。

在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

长连接和短连接的产生在于client和server采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

### TCP和UDP的区别

TCP：面向连接、传输可靠(保证数据正确性,保证数据顺序)、用于传输大量数据(流模式)、速度慢，建立连接需要开销较多(时间，系统资源)，具有流量控制和拥塞控制

UDP：面向非连接、传输不可靠、用于传输少量数据(数据包模式)、速度快。

### TCP如何保证可靠性

### 顺序号保证不会乱序

### 确认包

### 三次握手

### 流量控制和拥塞控制

### Socket建立过程

### TCP

### Server端

Server端所要做的事情主要是建立一个通信的端点，然后等待客户端发送的请求。典型的处理步骤如下：

1. 构建一个ServerSocket实例，绑定本地的端口。这个socket就是用来监听指定端口的连接请求的。

2.重复如下几个步骤：

a. 调用socket的accept()方法来获得下面客户端的连接请求。通过accept()方法返回的socket实例，建立了一个和客户端的新连接。

b.通过这个返回的socket实例获取InputStream和OutputStream,可以通过这两个stream来分别读和写数据。

c.结束的时候调用socket实例的close()方法关闭socket连接。

1. //1. 构造ServerSocket实例，指定服务端口。
2. ServerSocket servSock = new ServerSocket(servPort);

5. while(true)
6. {
7. // 2.调用accept方法，建立和客户端的连接
8. Socket clntSock = servSock.accept();
9. SocketAddress clientAddress =
10. clntSock.getRemoteSocketAddress();
11. System.out.println("Handling client at " + clientAddress);
13. // 3. 获取连接的InputStream,OutputStream来进行数据读写
14. InputStream in = clntSock.getInputStream();
15. OutputStream out = clntSock.getOutputStream();
17. while((recvMsgSize = in.read(receiveBuf)) != -1)
18. {
19. out.write(receiveBuf, 0, recvMsgSize);
20. }
21. // 4.操作结束，关闭socket.
22. clntSock.close();
23. }

### Client端

客户端的请求过程稍微有点不一样：

1.构建Socket实例，通过指定的远程服务器地址和端口来建立连接。

2.通过Socket实例包含的InputStream和OutputStream来进行数据的读写。

3.操作结束后调用socket实例的close方法，关闭。

示例代码如下；

Java代码  [[收藏代码http://shmilyaw-hotmail-com.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. // 1.根据指定的server地址和端口，建立socket连接。
2. Socket socket = new Socket(server, servPort);
4. // 2. 根据socket实例获取InputStream, OutputStream进行数据读写。
5. InputStream in = socket.getInputStream();
6. OutputStream out = socket.getOutputStream();
7. out.write(data);
9. //3.操作结束，关闭socket.
10. socket.close();

## UDP

UDP和TCP有两个典型的区别，一个就是它不需要建立连接，另外就是它在每次收发的报文都保留了消息的边界。

### server端

因为UDP协议不需要建立连接，它的过程如下：

1. 构造DatagramSocket实例，指定本地端口。

2. 通过DatagramSocket实例的receive方法接收DatagramPacket.DatagramPacket中间就包含了通信的内容。

3. 通过DatagramSocket的send和receive方法来收和发DatagramPacket.

典型的交互流程代码如下：

Java代码  [[收藏代码http://shmilyaw-hotmail-com.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. // 1. 构建DatagramSocket实例，指定本地端口。
2. DatagramSocket socket = new DatagramSocket(servPort);
4. // 2. 构建需要收发的DatagramPacket报文
5. DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX], ECHOMAX);
7. while(true)
8. {
9. // 3. 收报文
10. socket.receive(packet);
11. System.out.println("Handling client at " + packet.getAddress().getHostAddress()
12. + " on port " + packet.getPort());
13. // 4. 发报文
14. socket.send(packet);
15. packet.setLength(ECHOMAX);
16. }

// 1. 构建DatagramSocket实例，指定本地端口。

DatagramSocket socket = new DatagramSocket(servPort);

// 2. 构建需要收发的DatagramPacket报文

DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX], ECHOMAX);

while(true)

{

// 3. 收报文

socket.receive(packet);

System.out.println("Handling client at " + packet.getAddress().getHostAddress()

+ " on port " + packet.getPort());

// 4. 发报文

socket.send(packet);

packet.setLength(ECHOMAX);

}

### client端

UDP客户端的步骤也比较简单，主要包括下面3步：

1. 构造DatagramSocket实例。

2.通过DatagramSocket实例的send和receive方法发送DatagramPacket报文。

3.结束后，调用DatagramSocket的close方法关闭。

因为和TCP不同，UDP发送报文的时候可以在同一个本地端口随意发送给不同的服务器，一般不需要在UDP的DatagramSocket的构造函数中指定目的服务器的地址。

另外，UDP客户端还有一个重要的不同就是，TCP客户端发送echo连接消息之后会在调用read方法的时候进入阻塞状态，而UDP这样却不行。因为UDP中间是可以允许报文丢失的。如果报文丢失了，进程一直在阻塞或者挂起的状态，则进程会永远没法往下走了。所以会一般设置一个setSoTimeout方法，指定在多久的时间内没有收到报文就放弃。也可以通过指定一个数字，循环指定的次数来读取报文，读到就返回，否则就放弃。

一个典型的UDP Client代码示例如下：

Java代码  [[收藏代码http://shmilyaw-hotmail-com.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. // 1. 构造UDP DatagramSocket对象
2. DatagramSocket socket = new DatagramSocket();
4. // 2。指定timeout时间，防止进入无限等待状态
5. socket.setSoTimeout(TIMEOUT);
7. // 3. 构造收发的报文对象
8. DatagramPacket sendPacket = new DatagramPacket(bytesToSend,
9. bytesToSend.length, serverAddress, servPort);
10. DatagramPacket receivePacket =
11. new DatagramPacket(new byte[bytesToSend.length], bytesToSend.length);
13. // 4.指定尝试的次数
14. int tries = 0;
15. boolean receivedResponse = false;
16. do
17. {
18. socket.send(sendPacket);
19. try
20. {
21. socket.receive(receivePacket);
23. if(!receivePacket.getAddress().equals(serverAddress))
24. {
25. throw new IOException("Received packet from an unknown source");
26. }
27. receivedResponse = true;
28. }
29. catch(InterruptedIOException e)
30. {
31. tries += 1;
32. System.out.println("Timed out, " + (MAXTRIES - tries) + "");
33. }
34. }while((!receivedResponse) && (tries < MAXTRIES));
36. // 根据是否接收到报文进行反馈
37. if(receivedResponse)
38. {
39. System.out.println("Received: " + new String(receivePacket.getData()));
40. }
41. else
42. {
43. System.out.println("No response -- giving up.");
44. }
46. // 5. 关闭socket
47. socket.close();

### HTTP状态码

HTTP状态码        摘要说明  
-----------------------------------------------------------------------------------------------------------

成功2XX        成功处理了请求的状态码。  
200          服务器已成功处理了请求并提供了请求的网页。  
204          服务器成功处理了请求，但没有返回任何内容。                           
重定向3XX      每次请求中使用重定向不要超过 5 次。  
301         请求的网页已永久移动到新位置。当URLs发生变化时，使用301代码。搜索引擎索引中保存新的URL。  
302         请求的网页临时移动到新位置。搜索引擎索引中保存原来的URL。  
304         如果网页自请求者上次请求后没有更新，则用304代码告诉搜索引擎机器人，可节省带宽和开销。  
客户端错误4XX  表示请求可能出错，妨碍了服务器的处理。  
400         服务器不理解请求的语法。  
403         服务器拒绝请求。  
404       服务器找不到请求的网页。服务器上不存在的网页经常会返回此代码。  
410        请求的资源永久删除后，服务器返回此响应。该代码与 404（未找到）代码相似，  
但在资源以前存在而现在不存在的情况下，有时用来替代404 代码。如果资源已永久删除，应当使用 301 指定资源的新位置。  
服务器错误5XX  表示服务器在处理请求时发生内部错误。这些错误可能是服务器本身的错误，而不是请求出错。  
500      服务器遇到错误，无法完成请求。  
503      服务器目前无法使用（由于超载或停机维护）。通常，这只是暂时状态。

详细分解：

2XX  成功

200  正常；请求已完成。  
201  正常；紧接 POST 命令。  
202  正常；已接受用于处理，但处理尚未完成。  
203  正常；部分信息 — 返回的信息只是一部分。  
204  正常；无响应 — 已接收请求，但不存在要回送的信息。

3XX  重定向

301  已移动 — 请求的数据具有新的位置且更改是永久的。  
302  已找到 — 请求的数据临时具有不同 URI。  
303  请参阅其它 — 可在另一 URI 下找到对请求的响应，且应使用 GET 方法检索此响应。  
304  未修改 — 未按预期修改文档。  
305  使用代理 — 必须通过位置字段中提供的代理来访问请求的资源。  
306  未使用 — 不再使用；保留此代码以便将来使用。

4XX  客户机中出现的错误

400  错误请求 — 请求中有语法问题，或不能满足请求。  
401  未授权 — 未授权客户机访问数据。  
402  需要付款 — 表示计费系统已有效。  
403  禁止 — 即使有授权也不需要访问。  
404  找不到 — 服务器找不到给定的资源；文档不存在。  
407  代理认证请求 — 客户机首先必须使用代理认证自身。  
415  介质类型不受支持 — 服务器拒绝服务请求，因为不支持请求实体的格式。

5XX  服务器中出现的错误

500  内部错误 — 因为意外情况，服务器不能完成请求。  
501  未执行 — 服务器不支持请求的工具。  
502  错误网关 — 服务器接收到来自上游服务器的无效响应。  
503  无法获得服务 — 由于临时过载或维护，服务器无法处理请求。

504 Gateway Timeout  
作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器（URI标识出的服务器，例如HTTP、FTP、LDAP）或者辅助服务器（例如DNS）收到响应。  
注意：某些代理服务器在DNS查询超时时会返回400或者500错误

505 HTTP Version Not Supported  
服务器不支持，或者拒绝支持在请求中使用的 HTTP 版本。这暗示着服务器不能或不愿使用与客户端相同的版本。响应中应当包含一个描述了为何版本不被支持以及服务器支持哪些协议的实体。

506 Variant Also Negotiates  
由《透明内容协商协议》（RFC 2295）扩展，代表服务器存在内部配置错误：被请求的协商变元资源被配置为在透明内容协商中使用自己，因此在一个协商处理中不是一个合适的重点。

507 Insufficient Storage  
服务器无法存储完成请求所必须的内容。这个状况被认为是临时的。WebDAV (RFC 4918)

508 Loop Detected  
服务器发现请求中出现一个无穷循环

509 Bandwidth Limit Exceeded  
服务器达到带宽限制。这不是一个官方的状态码，但是仍被广泛使用。

510 Not Extended  
获取资源所需要的策略并没有没满足。（RFC 2774）

**常见错误的处理方法**

500错误

500错误是站长经常遇到的问题，就本人的经验，原因及解决方法归纳如下：

1、运行的用户数过多，对服务器造成的压力过大，服务器无法响应，则报HTTP500错误。

这个原因是网站报500错误的最主要原因，很多网站为什么突然会报500错误（service unavailable）呢？90%是由于空间压力过大，超出了空间商设定的上限而造成的。

遇到这个问题，最好是第一时间找到空间商的售后服务，他们一般会帮你回收一下应用程序池，问题便得到解决。如果没有找到售后服务，可以自己在空间管理后台，找到“回收应用程序池”那个项目，自己手动执行一下即可。不过，一些空间管理后台并没有提供该项服务。

2、如果排除了第1的可能性，那么很大原因便是程序上出现了问题。

可以做个简单的测试页面，看是否能运行成功，再检测报错的页面，从而针对性的进行修改。

该做关联的地方没有去做关联，则报HTTP500错误。进行手工或者自动关联，问题得到解决。

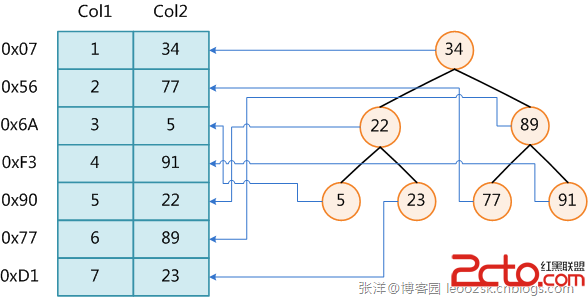
3、如果测试中所进行的操作需要向数据库中插入数据，若大数据量的情况下导致数据库中表空间已满，或者缓冲池较小无法满足数据的存取等，都有可能导致HTTP500错误。

### mysql 索引分类 在数据库表中，对字段建立索引可以大大提高查询速度。通过善用这些索引，可以令 MySQL的查询和运行更加高效。索引是快速搜索的关键。MySQL索引的建立对于MySQL的高效运行是很重要的。下面介绍几种常见的MySQL索引类型。 **1、普通型索引** 这是最基本的索引类型，而且它没有唯一性之类的限制。普通索引可以通过以下几种方式创建： （1）创建索引，例如CREATE INDEX 索引的名字 ON tablename (列名1，列名2,...); （2）修改表，例如ALTER TABLE tablename ADD INDEX 索引的名字 (列名1，列名2,...); （3）创建表的时候指定索引，例如CREATE TABLE tablename ( [...], INDEX 索引的名字 (列名1，列名 2,...) ); **2、唯一索引** 这种索引和前面的“普通索引”基本相同，但有一个区别：索引列的所有值都只能出现一次，即必须唯一。唯一性索引可以用以下几种方式创建： （1）创建索引，例如CREATE UNIQUE INDEX 索引的名字 ON tablename (列的列表); （2）修改表，例如ALTER TABLE tablename ADD UNIQUE 索引的名字 (列的列表); （3）创建表的时候指定索引，例如CREATE TABLE tablename ( [...], UNIQUE 索引的名字 (列的列 表) ); 主键是一种唯一性索引，但它必须指定为“PRIMARY KEY”。如果你曾经用过AUTO\_INCREMENT类型的列，你可能已经熟悉主键之类的概念了。主键一般在创建表的时候指定，例如“CREATE TABLE tablename ( [...], PRIMARY KEY (列的列表) ); ”。但是，我们也可以通过修改表的方式加入主键，例如“ALTER TABLE tablename ADD PRIMARY KEY (列的列表); ”。每个表只能有一个主键。 （主键相当于聚合索引，是查找最快的索引） **4、单列索引和多列索引** 索引可以是单列索引，也可以是多列索引。 （1）单列索引就是常用的一个列字段的索引，常见的索引。 （2）多列索引就是含有多个列字段的索引 alter table student add index sy(name,age，score); 索引sy就为多列索引，多列索引在以下几中情况下才能有效： select \* from student where name='jia' and age>='12' //where条件中含有索引的首列字段和第二个字段 select \* from student where name='jia' //where条件中只含有首列字段 select \* from student where name='jia' and score<60//where条件中含有首列字段和第三个字段 总结：多列索引只有在where条件中含有索引中的首列字段时才有效

**数据库索引**，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B树及其变种B+树**。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



上图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。

**创建索引可以大大提高系统的性能。**

第一，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。

第二，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。

第三，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。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，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。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？**因为，增加索引也有许多不利的方面。**

第一，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。

第二，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。

第三，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。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。**一般来说，应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。**一般来说，不应该创建索引的的这些列具有下列特点：**

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在[数据库设计](http://www.2cto.com/database/)器中创建三种索引：**唯一索引、主键索引和聚集索引**。

**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列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。 **聚集索引** 在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。**一个表只能包含一个聚集索引。**

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。**与非聚集索引相比，聚集索引通常提供更快的数据访问速度。**

1. **什么是事务**

**数据库事务是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行,要么完全地不执行。**

满足4个属性：

原子性（Atomic）、一致性（Consistency）、隔离性（Isolation）和持久性（Durabiliy），简称为ACID。

●   Atomic（原子性）：事务中包含的操作被看做一个逻辑单元，这个逻辑单元中的操作要么全部成功，要么全部失败。

●   Consistency（一致性）：只有合法的数据可以被写入数据库，否则事务应该将其回滚到最初状态。

●   Isolation（隔离性）：事务允许多个用户对同一个数据进行并发访问，而不破坏数据的正确性和完整性。同时，并行事务的修改必须与其他并行事务的修改相互独立。

●   Durability（持久性）：事务结束后，事务处理的结果必须能够得到固化。

· 原子性

事务必须是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行。通常，与某个事务关联的操作具有共同的目标，并且是相互依赖的。如果系统只执行这些操作的一个子集，则可能会破坏事务的总体目标。原子性消除了系统处理操作子集的可能性。

　　· 一致性

事务在完成时，必须使所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。事务结束时，所有的内部数据结构（如 B 树索引或双向链表）都必须是正确的。某些维护一致性的责任由应用程序开发人员承担，他们必须确保应用程序已强制所有已知的完整性约束。例如，当开发用于转帐的应用程序时，应避免在转帐过程中任意移动小数点。

　　· 隔离性

由并发事务所作的修改必须与任何其它并发事务所作的修改隔离。事务查看数据时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看中间状态的数据。这称为可串行性，因为它能够重新装载起始数据，并且重播一系列事务，以使数据结束时的状态与原始事务执行的状态相同。当事务可序列化时将获得最高的隔离级别。在此级别上，从一组可并行执行的事务获得的结果与通过连续运行每个事务所获得的结果相同。由于高度隔离会限制可并行执行的事务数，所以一些应用程序降低隔离级别以换取更大的吞吐量。

　　· 持久性

事务完成之后，它对于系统的影响是永久性的。该修改即使出现致命的系统故障也将一直保持。

设想网上购物的一次交易，其付款过程至少包括以下几步数据库操作：

　　· 更新客户所购商品的库存信息

　　· 保存客户付款信息--可能包括与银行系统的交互

　　· 生成订单并且保存到数据库中

· 更新用户相关信息，例如购物数量等等

1. **Group by和order by 和having的SQL语句举例**

语句顺序  
select 选择的列  
from 表  
where 查询的条件  
group by 分组属性 having 分组过滤的条件  
order by 排序属性  
limit 起始记录位置，取记录的条数   
其中  
select 选择的列  
from 表  
where 查询的条件  
以上是基本的结构  
  
group by 分组属性 having 分组过滤的条件  
这个是按照分组属性进行分组，所有分组属性上值相同的记录被分为一组，作为结果中的一条记录，后面的having是对分组进行过滤的条件，必须和group by一起使用  
  
order by 排序属性 是对结果集合进行排序，可以是升序asc，也可以是降序desc  
  
limit 起始记录位置，取记录的条数   
对记录进行选取，主要用来实现分页功能

select \* from tablename <条件语句> limit 15

SELECT A COUNT(B) FROM TABLE GROUP BY A HAVING COUNT(B)>2

1. **数据库范式**

1NF:每个属性是不可分的。 2NF:若关系R是１NF,且每个非主属性都完全函数依赖于R的键。例SLC(SID#, CourceID#, SNAME,Grade),则不是2NF; 3NF:若R是2NF，且它的任何非键属性都不传递依赖于任何候选键。

1. **触发器的作用**

  答：触发器是一中特殊的存储过程，主要是通过事件来触发而被执行的。它可以强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

1. **索引的作用？和它的优点缺点是什么？**

答：索引就一种特殊的查询表，数据库的搜索引擎可以利用它加速对数据的检索。它很类似与现实生活中书的目录，不需要查询整本书内容就可以找到想要的数据。索引可以是唯一的，创建索引允许指定单个列或者是多个列。缺点是它减慢了数据录入的速度，同时也增加了数据库的尺寸大小。

1. **什么叫视图？**

答：视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，视图通常是有一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比多表查询。

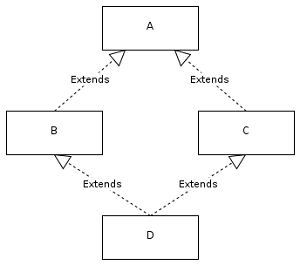
视图可以修改数据，但是有很多限制，比如能够对单表进行修改，多表连接的时候就不行。

1. **查询A(ID,Name)表中第31至40条记录，ID作为主键可能是不是连续增长的列**

完整的查询语句如下：

select top 10 \* from A where ID >(select max(ID) from (select top 30 ID from A order by A ) T) order by A

1. **JAVA为什么不支持多继承**

  
         我们有两个类B和C继承自A.假设B和C都重写继承的方法，他们提供自己的实现。现在B和C从D继承多重继承。D应该继承，重写的方法，将用于重写的方法吗？它从B或C？在这里，我们有歧义。

接口中只存在对方法的声明，所以不会冲突

1. **String和StringBuffer和StringBuilder的区别?**  
     答：String是不可变的对象，每次对String类型进行改变都相当于产生了一个新的对象，StringBuffer和StringBuilder是可变的字符序列，SringBuffer是线程安全的，StringBuilder是线程不安全的
2. **Session和cookie机制之间的区别和联系**

session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。

当程序需要为某个客户端的请求创建一个session的时候，服务器首先检查这个客户端的请求里是否已包含了一个session标识-称为sessionid，如果已包含一个sessionid则说明以前已经为此客户端创建过session，服务器就按照sessionid把这个session检索出来使用（如果检索不到，可能会新建一个），如果客户端请求不包含sessionid，则为此客户端创建一个session并且生成一个与此session相关联的sessionid，sessionid的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个sessionid将被在本次响应中返回给客户端保存。

保存这个sessionid的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器。一般这个cookie的名字都是类似于SEEESIONID，而。比如weblogic对于web应用程序生成的cookie，JSESSIONID=ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764，它的名字就是JSESSIONID。

由于cookie可以被人为的禁止，必须有其他机制以便在cookie被禁止时仍然能够把sessionid传递回服务器。经常被使用的一种技术叫做URL重写，就是把sessionid直接附加在URL路径的后面，附加方式也有两种，一种是作为URL路径的附加信息

在谈论session机制的时候，常常听到这样一种误解“只要关闭浏览器，session就消失了”。其实可以想象一下会员卡的例子，除非顾客主动对店家提出销卡，否则店家绝对不会轻易删除顾客的资料。对session来说也是一样的，除非程序通知服务器删除一个session，否则服务器会一直保留，程序一般都是在用户做logoff的时候发个指令去删除session。然而浏览器从来不会主动在关闭之前通知服务器它将要关闭，因此服务器根本不会有机会知道浏览器已经关闭，之所以会有这种错觉，是大部分session机制都使用会话cookie来保存sessionid，而关闭浏览器后这个sessionid就消失了，再次连接服务器时也就无法找到原来的session。如果服务器设置的cookie被保存到硬盘上，或者使用某种手段改写浏览器发出的HTTP请求头，把原来的sessionid发送给服务器，则再次打开浏览器仍然能够找到原来的session。

　　恰恰是由于关闭浏览器不会导致session被删除，迫使服务器为seesion设置了一个失效时间，当距离客户端上一次使用session的时间超过这个失效时间时，服务器就可以认为客户端已经停止了活动，才会把session删除以节省存储空间。

session就是一种保存上下文信息的机制，它是针对每一个用户的，变量的值保存在服务器端，通过SessionID来区分不同的客户

1. **抽象类和接口的区别**

1，抽象类里可以有构造方法，而接口内不能有构造方法。

   2，抽象类中可以有普通成员变量，而接口中不能有普通成员变量。

   3，抽象类中可以包含非抽象的普通方法，而接口中所有的方法必须是抽象的，不能有非抽象的普通方法。

   4，抽象类中的抽象方法的访问类型可以是public ，protected和默认类型，但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。

   5，抽象类中可以包含静态方法，接口内不能包含静态方法。

   6，抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public  static类型，并且默认为public static类型。

   7，一个类可以实现多个接口，但只能继承一个抽象类。

1. **JAVA内存泄漏**

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

1. 静态集合类像HashMap、Vector等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，所有的对象Object也不能被释放，因为他们也将一直被Vector等应用着。
2. 单例模式   
   不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露。
3. 各种连接   
   比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。
4. **内存泄漏解决方法**
5. 避免内存泄露的小技巧
6. a. 尽量使用直接量,对于Byte、Short、Integer、Long、Float、Double、Bolean、Character程序不应该使用**new** 方式创建对象，而采用直接量创建它们。
7. b. 使用StringBuilder和StringBuffer进行字符串连接
8. Sting和StringBuilder以及StringBuffer等都可以代表字符串，其中String字符串代表的是不可变的字符串，后两者表示可变的字符串。
9. 如果使用多个String对象进行字符串连接运算，在运行时可能产生大量临时字符串，这些字符串会保存在内存中从而导致程序性能下降。
11. c. 尽量少使用静态变量
12. 类的静态变量的生命周期和类同步的。在类不被卸载的情况下，类对应类对象会常驻内存，知道程序运行结束。
13. 如下代码就会存在内存泄露问题：
14. **class**  Person
15. {
16. **static** Object obj = **new** Object();
17. }
18. d.避免在经常调用的方法、循环中创建Java对象。
19. e. 缓存经常使用的对象
20. 如果有些对象需要经常被用到，可以考虑把这些对象用缓存池保存起来，下次用的时候直接从池中拿。典型的就是数据连接池。
21. 如果系统中还有一些常用的基础信息，可以考虑用缓存，实现缓存的方式有如下两种：
22. (1) 使用HashMap进行缓存。
23. (2) 直接使用某些开源的缓存项目。
24. 说明：缓存设计本身就是一直以牺牲系统空间来换取运行时间的技术。
25. f. 考虑使用SoftReference
26. 当程序需要创建长度很大的数组时，可以考虑使用SoftReferene来包装数组元素，而不是直接让将数组元素来引用对象。此时SoftReference是个很好的选择:当内存足够时，它的功能等同于普通引用；当内存不足时，它会牺牲自己，释放软引用所引用的对象。
27. 注意：由于软引用引用对象的不确定性(软引用所获取的对象可能为**null**)，所以程序取出SoftReference所引用的Java之后，应该显式判断该对象是否为**null**； 当该对象为**null**时，应该重建该对象。
28. **为什么需要三次握手**

这主要是为了防止已失效的请求报文段突然又传送到了服务端而产生连接的误判

比如：客户端发送了一个连接请求报文段A到服务端，但是在某些网络节点上长时间滞留了，而后客户端又超时重发了一个连接请求报文段B该服务端，而后 正常建立连接，数据传输完毕，并释放了连接。但是请求报文段A延迟了一段时间后，又到了服务端，这本是一个早已失效的报文段，但是服务端收到后会误以为客户端又发出了一次连接请求，于是向客户端发出确认报文段，并同意建立连接。那么问题来了，假如这里没有三次握手，这时服务端只要发送了确认，新的 连接就建立了，但由于客户端没有发出建立连接的请求，因此不会理会服务端的确认，也不会向服务端发送数据，而服务端却认为新的连接已经建立了，并在 一直等待客户端发送数据，这样服务端就会一直等待下去，直到超出保活计数器的设定值，而将客户端判定为出了问题，才会关闭这个连接。这样就浪费了很多服务 器的资源。而如果采用三次握手，客户端就不会向服务端发出确认，服务端由于收不到确认，就知道客户端没有要求建立连接，从而不建立该连接。

1. **三次握手的缺点**

## SYN Flood 攻击

SYN- Flood攻击是当前网络上最为常见的DDoS攻击，也是最为经典的拒绝服务攻击，它就是利用了TCP协议实现上的一个缺陷，通过向网络服务所在端口发送大量 的伪造源地址的攻击报文，就可能造成目标服务器中的半开连接队列被占满，从而阻止其他合法用户进行访问。这种攻击早在1996年就被发现，但至今仍然显示 出强大的生命力。很多操作系统，甚至防火墙、路由器都无法有效地防御这种攻击，而且由于它可以方便地伪造源地址，追查起来非常困难。它的数据包特征通常 是，源发送了大量的SYN包，并且缺少三次握手的最后一步握手ACK回复。

原理：攻击者首先伪造地址对 服务器发起SYN请求，服务器回应(SYN+ACK)包，而真实的IP会认为，我没有发送请求，不作回应。服务 器没有收到回应，这样的话，服务器不知 道(SYN+ACK)是否发送成功，默认情况下会重试5次（tcp\_syn\_retries）。这样的话，对于服务器的内存，带宽都有很大的消耗。攻击者 如果处于公网，可以伪造IP的话，对于服务器就很难根据IP来判断攻击者，给防护带来很大的困难。

1. **Mysql两种存储引擎的优缺点**

MyISAM引擎是一种非事务性的引擎，提供高速存储和检索，以及全文搜索能力，适合数据仓库等查询频繁的应用。MyISAM中，一个table实际保存为三个文件，.frm存储表定义，.MYD存储数据,.MYI存储索引，数据文件和索引文件可以放置在不同的目录，平均分布io，获得更快的速度。

InnoDB:这种类型是事务安全的，它们还支持外键.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,InnoDB 给 MySQL 提供了具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。InnoDB 在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。 对比Myisam的存储引擎，InnoDB写的处理效率差一些并且会占用更多的磁盘空间以保留数据和索引。Innodb是索引和数据是紧密捆绑的

区别：

1. 是支持事务（TRANSACTION）；

InnoDB支持外键，MyISAM不支持。

|  |  |
| --- | --- |
| MyISAM类型的表强调的是性能，其执行速度更快，但是不提供事务支持 | InnoDB提供事务支持事务，外键等高级数据库功能 |

1. 是采用了行级锁。

InnoDB支持数据行锁定，MyISAM不支持行锁定，只支持锁定整个表。即MyISAM同一个表上的读锁和写锁是互斥的，MyISAM并发读写时如果等待队列中既有读请求又有写请求，默认写请求的优先级高，即使读请求先到，所以MyISAM不适合于有大量查询和修改并存的情况，那样查询进程会长时间阻塞。因为MyISAM是锁表，所以某项读操作比较耗时会使其他写进程饿死。

|  |  |
| --- | --- |
| MyISAM提供 表锁 | InnoDB 提供行锁(locking on row level)，另外，InnoDB表的行锁也不是绝对的，如果在执行一个SQL语句时MySQL不能确定要扫描的范围，InnoDB表同样会锁全表，例如update table set num=1 where name like “%aaa%” |

|  |  |
| --- | --- |
| 如果执行大量的SELECT，MyISAM是更好的选择 | 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 |

1. InnoDB不支持全文索引，而MyISAM支持。

2.InnoDB 中不保存表的具体行数，也就是说，执行select count(\*) from table时，InnoDB要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(\*)语句包含 where条件时，两种表的操作是一样的。  
3.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。

1. **Happens-before**



1. **字符串在java虚拟机中的存储方式**

 String s =  new  String( "xyz" );  在运行时涉及 几个String实例？

     两个，一个是字符串字面量"xyz"所对应的、驻留（intern）在一个全局共享的字符串常量池中的实例，另一个是通过new String(String)创建并初始化的、内容与"xyz"相同的实例。

     String中的final用法和理解：

     final只对引用的"值"(即内存地址)有效，它迫使引用只能指向初始指向的那个对象，改变它的指向会导致编译期错误。至于它所指向的对象的变化，final是不负责的。

**Java代码**

1. **final** StringBuffer a = **new** StringBuffer("111");
2. **final** StringBuffer b = **new** StringBuffer("222");
3. a=b;//此句编译不通过
5. **final** StringBuffer a = **new** StringBuffer("111");
6. a.append("222");//编译通过

**Java代码**

1. String a = "a1";
2. String b = "a" + 1;
3. System.out.println((a == b)); //result = true
4. String a = "atrue";
5. String b = "a" + "true";
6. System.out.println((a == b)); //result = true
7. String a = "a3.4";
8. String b = "a" + 3.4;
9. System.out.println((a == b)); //result = true

    JVM对于字符串常量的"+"号连接，将程序编译期，JVM就将常量字符串的"+"连接优化为连接后的值，拿"a" + 1来说，经编译器优化后在class中就已经是a1。在编译期其字符串常量的值就确定下来，故上面程序最终的结果都为true。

**Java代码**

1. String a = "ab";
2. String bb = "b";
3. String b = "a" + bb;
4. System.out.println((a == b)); //result = false

   JVM对于字符串引用，由于在字符串的"+"连接中，有字符串引用存在，而引用的值在程序编译期是无法确定的，即"a" + bb无法被编译器优化，只有在程序运行期来动态分配并将连接后的新地址赋给b。所以上面程序的结果也就为false。

**Java代码**

1. String a = "ab";
2. **final** String bb = "b";
3. String b = "a" + bb;
4. System.out.println((a == b)); //result = true

     和上面唯一不同的是bb字符串加了final修饰，对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中。所以此时的"a" + bb和"a" + "b"效果是一样的。故上面程序的结果为true。

**Java代码**

1. String a = "ab";
2. **final** String bb = getBB();
3. String b = "a" + bb;
4. System.out.println((a == b)); //result = false
6. **private** **static** String getBB() {
7. **return** "b";
8. }

 JVM对于字符串引用bb，它的值在编译期无法确定，只有在程序运行期调用方法后，将方法的返回值和"a"来动态连接并分配地址为b，故上面程序的结果为false。

1. **JAVA中的finally**

1.无论try里面有没有return语句，finally语句一定都会执行（不愧是finally，名字起的好）。

        2.如果finally中没有return语句，try里面有return，那么在执行try中的return语句之前会先去执行finally中的代码，再去执行try中的return语句；如果在finally中也包含return语句，将会直接返回，不再去执行try中的return语句。

1. **IOC与DI的概念**

### IoC能做什么

　　IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是 松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

　　其实**IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。**

**IoC很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。**

### IoC和DI

**DI—Dependency Injection，即“依赖注入”**：**组件之间依赖关系**由容器在运行期决定，形象的说，即**由容器动态的将某个依赖关系注入到组件之中**。**依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。**通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

　　理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

　　●**谁依赖于谁：**当然是**应用程序依赖于IoC容器**；

　　●**为什么需要依赖：应用程序需要IoC容器来提供对象需要的外部资源**；

　　●**谁注入谁：**很明显是**IoC容器注入应用程序某个对象，应用程序依赖的对象**；

**●注入了什么：**就是**注入某个对象所需要的外部资源（包括对象、资源、常量数据）**。

**IoC和DI**由什么**关系**呢？其实它们**是同一个概念的不同角度描述**，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字：“依赖注入”，相对IoC 而言，**“依赖注入”明确描述了“被注入对象依赖IoC容器配置依赖对象”。**