# 总结

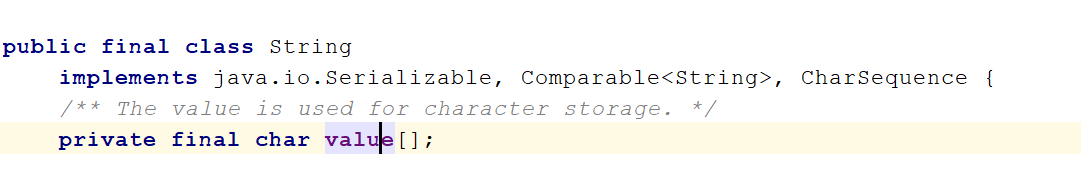
## Java基础：

### 面向过程(c)相对性能高于面向对象(java)的原因：

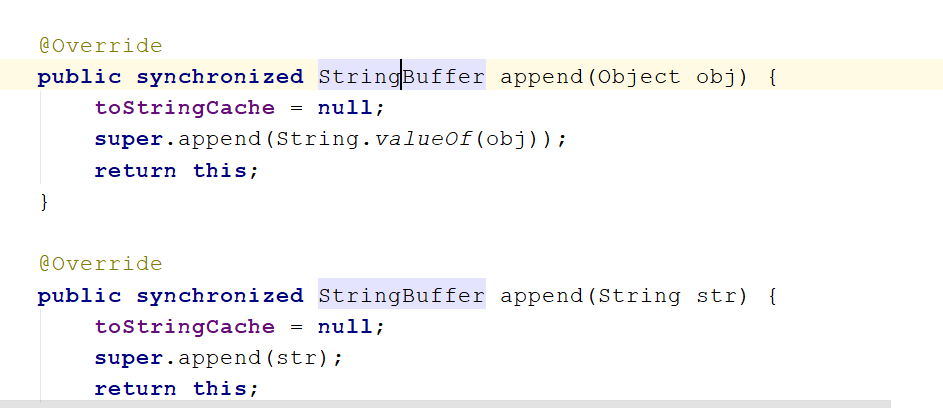
* 1. 面向过程经过一次编译为机器运行的机器码，而面向对象得先编译为字节码(.class)，然后java虚拟机转换为机器码运行。
  2. 面向过程：性能相对较好；
  3. 面向对象：易维护、易复用、易扩展。

### String、StringBuilder、StringBuffer区别：

1. String中value被加了final,具备不可变性，每一次操作都是一个新的字符串:



1. StringBuilder线程不安全，未被final修饰，每次都是操作的同一个字符串。
2. StringBuffer线程安全，被**synchronized** 修饰；未被final修饰，每次都是操作的同一个字符串。
3. StringBuilder和StringBuffer继承公共抽象父类AbstractStringBuilder



StringBuilder比StringBuffer性能高10%-15%，String适合操作少量数据；单线程操作大量数据使用StringBuilder，多线程操作大量数使用StringBuffer。

### hashCode作用：

1. 当把对象放入hashSet时，会先判断对象放入的位置，及hashCode值；
2. 获取哈希码(散列码)，通过哈希码确定对象在哈希表中的索引位置。
3. hashCode相同，两个对象不一定相等；相同的hashCode可能有多个对象，hashCode相当于缩小搜索范围。

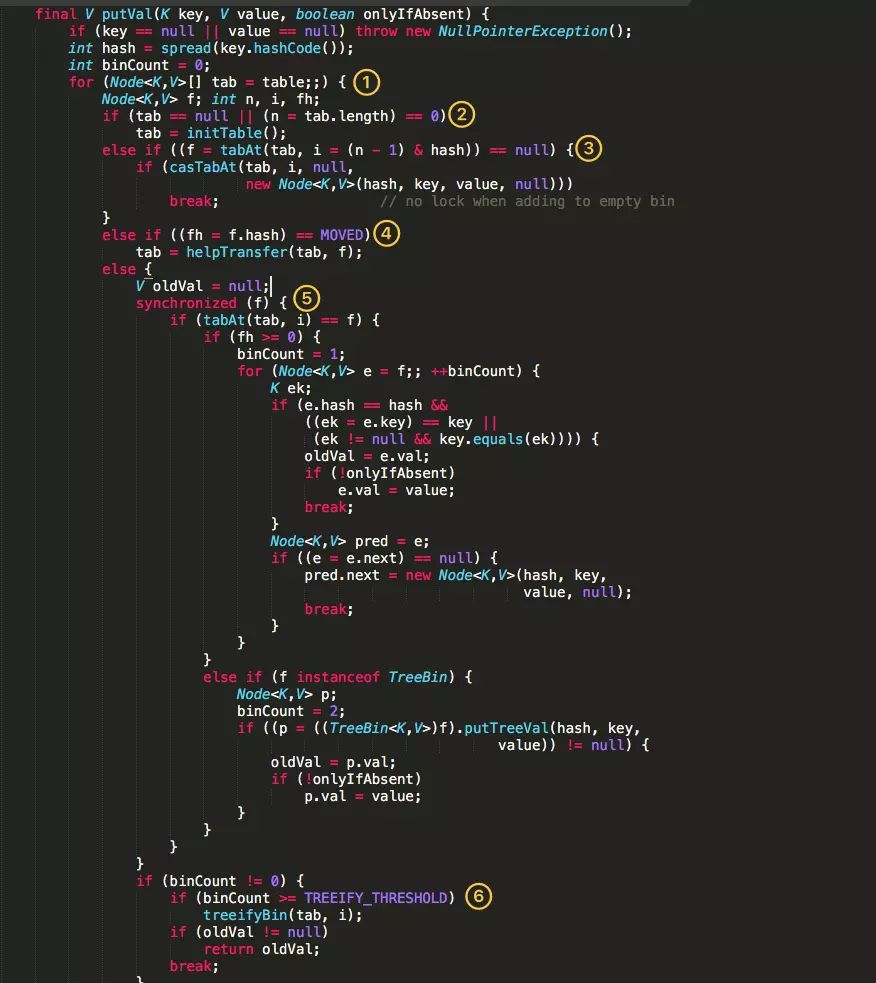
### jdk1.8新特性

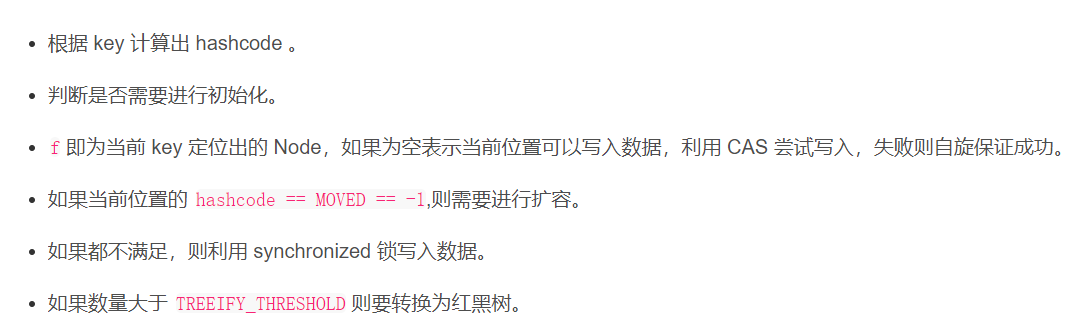
* **Lambda表达式**
* **函数式接口**
* **\*方法引用和构造器调用**
* **Stream API**
* **接口中的默认方法和静态方法**
* **新时间日期API**

详情地址：<https://blog.csdn.net/qq_29411737/article/details/80835658>

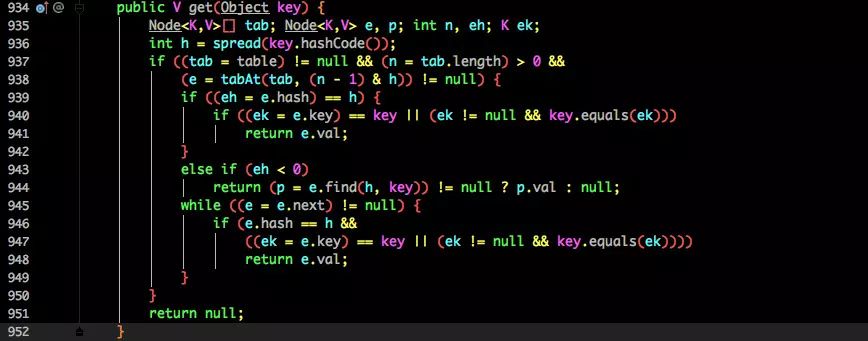
### ConcurrentHashMap原理

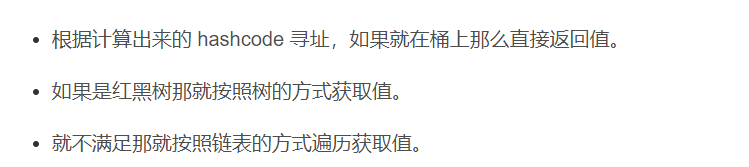
Put方法源码理解





get方法源码理解：



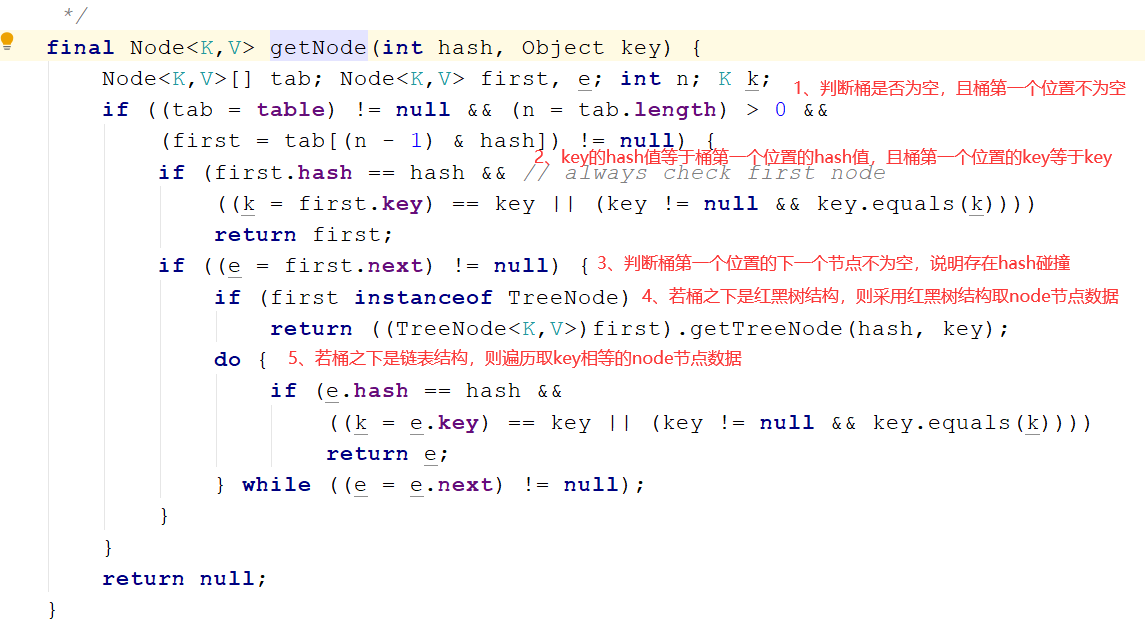


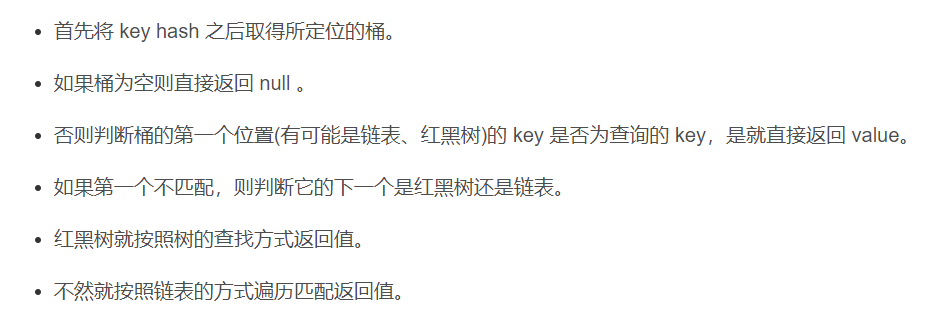
参考：<https://blog.csdn.net/tp7309/article/details/76532366>

https://blog.csdn.net/weixin\_44460333/article/details/86770169

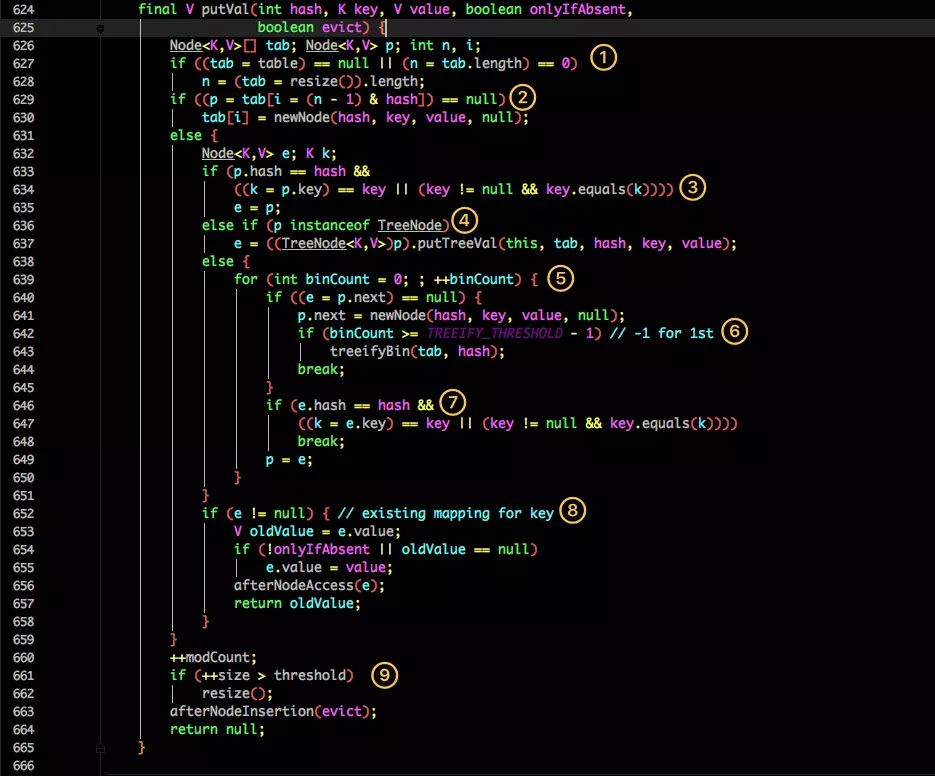
### HashMap原理

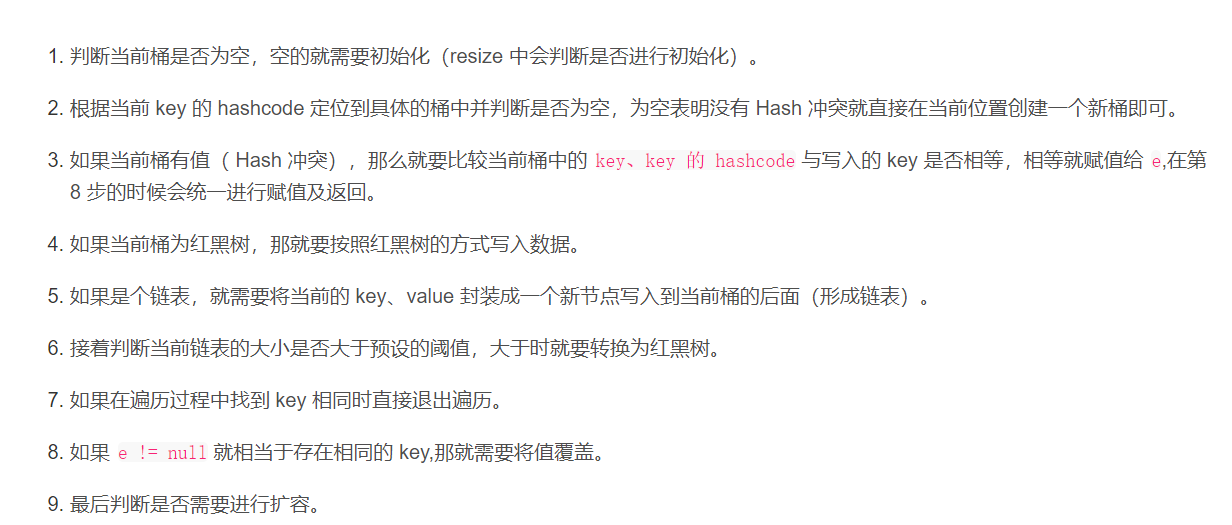
get方法源码理解：





put方法源码理解：





面试题：

#### 1、“你用过HashMap吗？” “什么是HashMap？你为什么用到它？”

答：用过；

hashMap特性：1.HashMap存储键值对实现快速存取，允许为null。key值不可重复，若key值重复则覆盖。2.非同步，线程不安全；3.底层是hash表，不保证有序(比如插入的顺序)；4、底层采用数组及链表结构或者数组及红黑树结构(当联表数量大于等于8时，自动转为红黑树)。

若需要hashMap线程安全可采用以下两个方法：1、Collections.synchronizedMap(map);2、ConcurrentHashMap。

#### 谈一谈hashMap底层原理

答：基于hashing的原理，jdk8后采用数组+链表+红黑树的数据结构。我们通过put和get存储和获取对象。当我们给put()方法传递键和值时，先对键做一个hashCode()的计算来得到它在bucket数组中的位置来存储Entry对象。当获取对象时，通过get获取到bucket的位置，再通过键对象的equals()方法找到正确的键值对，然后在返回值对象。

#### 谈一谈hashMap中put方法的实现

答：1.计算关于key的hashcode值（与Key.hashCode的高16位做异或运算）

2.如果散列表为空时，调用resize()初始化散列表

3.如果没有发生碰撞，直接添加元素到散列表中去

4.如果发生了碰撞(hashCode值相同)，进行三种判断

    4.1:若key地址相同或者equals后内容相同，则替换旧值

    4.2:如果是红黑树结构，就调用树的插入方法

   4.3：链表结构，循环遍历直到链表中某个节点为空，尾插法进行插入，插入之后判断链表个数是否到达变成红黑树的阙值8；也可以遍历到有节点与插入元素的哈希值和内容相同，进行覆盖。

1. 如果桶满了大于阀值，则resize进行扩容

#### 3、1、说⼀下HashMap的Put⽅法

先说HashMap的Put⽅法的⼤体流程：

1. 根据Key通过哈希算法与与运算得出数组下标

2. 如果数组下标位置元素为空，则将key和value封装为Entry对象（JDK1.7中是Entry对象，JDK1.8中是 Node对象）并放⼊该位置

3. 如果数组下标位置元素不为空，则要分情况讨论

a. 如果是JDK1.7，则先判断是否需要扩容，如果要扩容就进⾏扩容，如果不⽤扩容就⽣成Entry对象，并使⽤头插法添加到当前位置的链表中

b. 如果是JDK1.8，则会先判断当前位置上的Node的类型，看是红⿊树Node，还是链表Node

i. 如果是红⿊树Node，则将key和value封装为⼀个红⿊树节点并添加到红⿊树中去，在这个过程中会判断红⿊树中是否存在当前key，如果存在则更新value

ii. 如果此位置上的Node对象是链表节点，则将key和value封装为⼀个链表Node并通过尾插法插⼊到链表的最后位置去，因为是尾插法，所以需要遍历链表，在遍历链表的过程中会判断是否存在当前key，如果存在则更新value，当遍历完链表后，将新链表Node插⼊到链表中，插⼊到链表后，会看当前链表的节点个数，如果⼤于等于8，那么则会将该链表转成红⿊树

iii. 将key和value封装为Node插⼊到链表或红⿊树中后，再判断是否需要进⾏扩容，如果需要就

#### 4.谈一下hashMap中什么时候需要进行扩容，扩容resize()又是如何实现的？

调用场景：

1.初始化数组table

2.当数组table的size达到阙值时即++size > load factor \* capacity 时，也是在putVal函数中

实现过程：(细讲)

1.通过判断旧数组的容量是否大于0来判断数组是否初始化过

否：进行初始化判断是否调用无参构造器，

是:使用默认的大小和阙值

否:使用构造函数中初始化的容量，当然这个容量是经过tableSizefor计算后的2的次幂数

是，进行扩容，扩容成两倍(小于最大值的情况下)，之后在进行将元素重新进行与运算复制到新的散列表中

概括的讲：扩容需要重新分配一个新数组，新数组是老数组的2倍长，然后遍历整个老结构，把所有的元素挨个重新hash分配到新结构中去。

PS：可见底层数据结构用到了数组，到最后会因为容量问题都需要进行扩容操作

#### 5.谈一下hashMap中get是如何实现的？

对key的hashCode进行hashing，与运算计算下标获取bucket位置，如果在桶的首位上就可以找到就直接返回，否则在树中找或者链表中遍历找，如果有hash冲突，则利用equals方法去遍历链表查找节点。

#### 6.谈一下HashMap中hash函数是怎么实现的？还有哪些hash函数的实现方式？

对key的hashCode做hash操作，与高16位做异或运算

还有平方取中法，除留余数法，伪随机数法

#### 7.为什么不直接将key作为哈希值而是与高16位做异或运算？

因为数组位置的确定用的是与运算，仅仅最后四位有效，设计者将key的哈希值与高16为做异或运算使得在做&运算确定数组的插入位置时，此时的低位实际是高位与低位的结合，增加了随机性，减少了哈希碰撞的次数。

HashMap默认初始化长度为16，并且每次自动扩展或者是手动初始化容量时，必须是2的幂。

#### 8、HashMap和HashTable的区别

相同点：都是存储key-value键值对的

不同点：

HashMap允许Key-value为null，hashTable不允许；

hashMap没有考虑同步，是线程不安全的。hashTable是线程安全的，给api套上了一层synchronized修饰;

HashMap继承于AbstractMap类，hashTable继承与Dictionary类。

迭代器(Iterator)。HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException。

容量的初始值和增加方式都不一样：HashMap默认的容量大小是16；增加容量时，每次将容量变为"原始容量x2"。Hashtable默认的容量大小是11；增加容量时，每次将容量变为"原始容量x2 + 1"；

添加key-value时的hash值算法不同：HashMap添加元素时，是使用自定义的哈希算法。Hashtable没有自定义哈希算法，而直接采用的key的hashCode()。

#### 9、平时在使用HashMap时一般使用什么类型的元素作为Key？

使用String或者Integer这样的类。这个时候可以继续追问为什么使用String、Integer呢？这些类有什么特点？如果面试者有很好的思考，可以回答出这些类经final修饰是不可变（Immutable）的，并且这些类已经很规范的覆写了hashCode()以及equals()方法。作为不可变类天生是线程安全的，而且可以很好的优化比如可以缓存hash值，避免重复计算等等，那么基本上这道题算是过关了。

参考资料：<https://blog.csdn.net/weixin_44460333/article/details/86770169>

<https://blog.csdn.net/suifeng629/article/details/82179996>

<https://www.cnblogs.com/zengcongcong/p/11295349.html>

### IO流



nio:

它是一种同步非阻塞的I/O模型，也是I/O多路复用的基础，已经被越来越多地应用到大型应用服务器，成为解决高并发与大量连接、I/O处理问题的有效方式。

传统IO基于字节流和字符流进行操作，而NIO基于Channel和Buffer(缓冲区)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。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。

NIO和IO适用场景

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，这时候用NIO处理数据可能是个很好的选择。

而如果只有少量的连接，而这些连接每次要发送大量的数据，这时候传统的IO更合适。使用哪种处理数据，需要在数据的响应等待时间和检查缓冲区数据的时间上作比较来权衡选择。

参考资料：<https://blog.csdn.net/zengxiantao1994/article/details/88094910>

<https://blog.csdn.net/u013857458/article/details/82424104>

<https://blog.csdn.net/u011381576/article/details/79876754>

https://www.cnblogs.com/dolphin0520/p/3919162.html

### 常见网络攻击

#### (1)分类

1、DDoS攻击

DDoS是最常见的一种攻击方式，攻击者通过向某一个站点服务器反复发送请求，导致服务器无法承载大量的请求而产生“拒绝服务”，这就导致正常的服务无法进行，影响我们服务器的使用。

2、获取账号和密码

这类攻击会存在一定的技术性。一般来说，攻击者会利用程序来抓取数据包，获取口令和数据内容，通过侦听程序在来监视网络数据流，进而通过分析获取用户的登录账号和密码。

3、SQL注入

SQL注入的攻击手段主要是利用后台的漏洞，通过URL将关键SQL语句带入程序，在数据库中进行破坏。许多的攻击者会使用F12或者postman等拼装ajax请求，将非法的数字发送给后台，造成程序的报错，并展现在页面上，这样攻击者就会知道后台使用的语言和框架了。

4、恶意小程序

这类攻击的方式主要存在我们使用的程序上面，它们可以通过入侵修改硬盘上的文件、窃取口令等。

5、木马植入

这种攻击方式主要是通过向服务器植入木马，开启后面，获取服务器的控制权，恶意破坏服务器文件或盗取服务器数据，这类的危害都是比较大的。

#### (2)sql注入

#{}是预编译处理，${}是字符替换（字符串拼接）。 在使用 #{}时，MyBatis 会将 SQL 中的 #{}替换成“?”，配合 PreparedStatement 的 set 方法赋值，这样可以有效的防止 SQL 注入，保证程序的运行安全。

#### (3)简述 tcp（传输控制协议） 和 udp的区别？

tcp 和 udp 是 OSI 模型中的运输层中的协议。tcp 提供可靠的通信传输，而 udp 则常被用于让广播和细节控制交给应用的通信传输。

两者的区别大致如下：

* tcp 面向连接，udp 面向非连接即发送数据前不需要建立链接；
* tcp 提供可靠的服务（数据传输），udp 无法保证；
* tcp 面向字节流，udp 面向报文；
* tcp 数据传输慢，udp 数据传输快；
* tcp占用系统资源多，udp较少；

## ArrayList和LinkedList的区别

1. ⾸先，他们的底层数据结构不同，ArrayList底层是基于数组实现的，LinkedList底层是基于链表实现的;

2. 由于底层数据结构不同，他们所适⽤的场景也不同，ArrayList更适合随机查找，LinkedList更适合删 除和添加，查询、添加、删除的时间复杂度不同 ;

3. 另外ArrayList和LinkedList都实现了List接⼝，但是LinkedList还额外实现了Deque接⼝，所以 LinkedList还可以当做队列来使⽤;

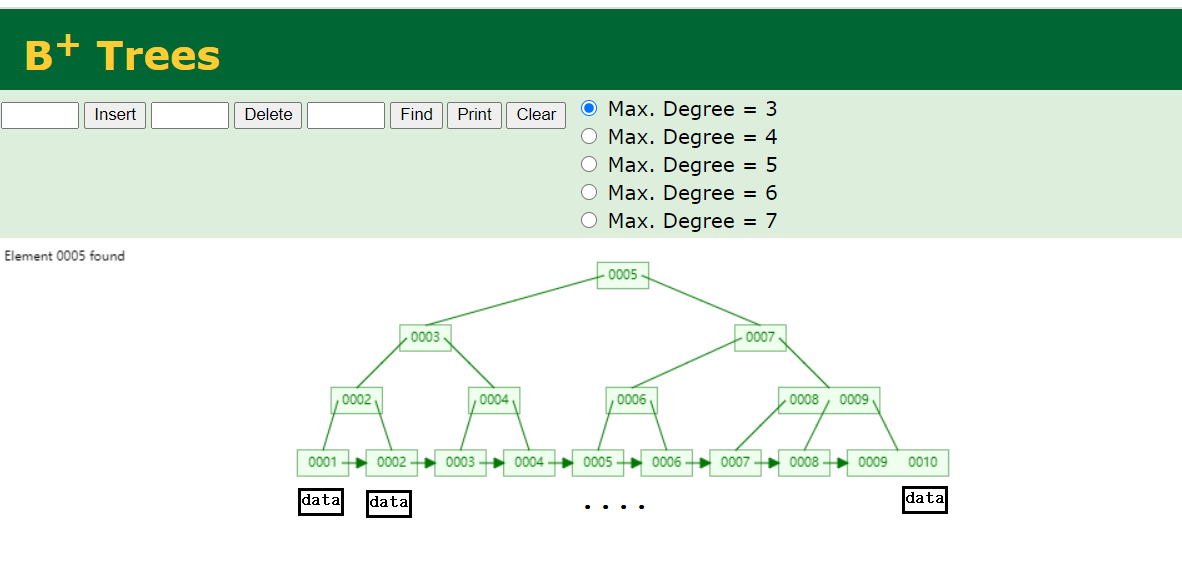
4、遍历LinkedList必须使用iterator不能使用for循环，因为每次for循环体内通过get(i)取得某一元素时都需 要对list重新进行遍历，性能消耗极大。 另外不要试图使用indexOf等返回元素索引，并利用其进行遍历，使用indexlOf对list进行了遍历，当结果为空时会遍历整个列表。

5、ArrayList扩容机制：因为数组长度固定，超出长度存数据时需要新建数组，然后将老数组的数据拷贝到新数组，如果不是尾部插入数据还会涉及到元素的移动（往后复制一份，插入新元素），使用尾插法并指定初始容量可以极大提升性能、甚至超过linkedList（需要创建大量的node对象）

## Mysql

### mysql为什么用b+树作为数据结构。

注：InnoDB存储引擎中（聚簇索引）data存放的是这一行的真实数据；myisam存放的是真实数据的磁盘地址。



使用b+树的优点：非叶子节点只保存了索引的key值，不会保存索引的value值，树结构每层可以保存更多的索引，mysql的InnoDB定义的非叶子节点(数据库中页的大小是固定的，InnoDB 中页的默认大小是 16KB)可以保存16kb的数据，以bigint为列：索引占据8个字节，两个索引之间指针占据6个字节，一层数可存放16kb/14b个索引，约为1143个索引，一般b+数三层可存储2000多万数据。并且，叶子节点为双向链表结构，比起b树而言，支持范围查找。

#### 为什么不用线性表？

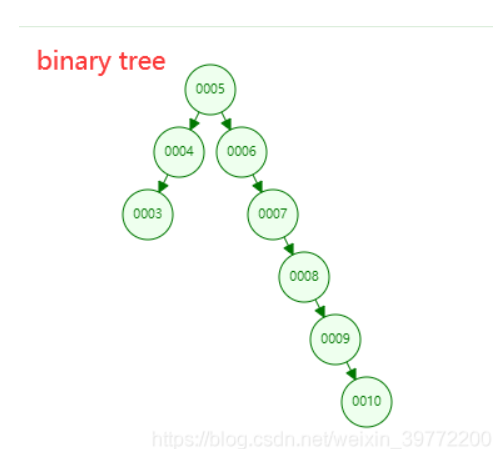
当查找的数据正好在线性表的最后一个索引时，则会进行全表扫描

#### （2）、为什么不用二分查找？

二分查找本身有排序，排序无论用什么算法都会消耗资源，尤其是查找时可能还会有insert、update、delete，这可能会导致数据的索引发生变化，极端情况查询开销也非常大

#### （3）、为什么不用二叉树查找？

二叉树：极端情况单分支变成线性表，查询开销也非常大。eg：右侧分支有几十万甚至上百万条数据，最坏的情况是数据在右侧分支的最后一个节点，此时查询效率和线性表无二。



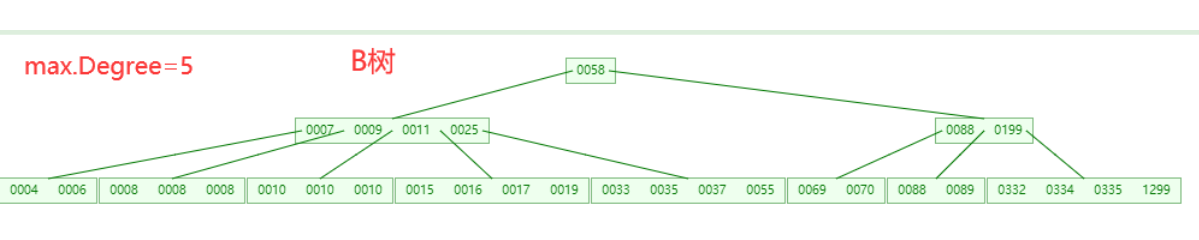
#### （4）为什么不用红黑树

使用红黑树，当数据越多，树的深度越深，查询耗时也越多，性能越差。

#### （5）、为什么不是哈希查找？

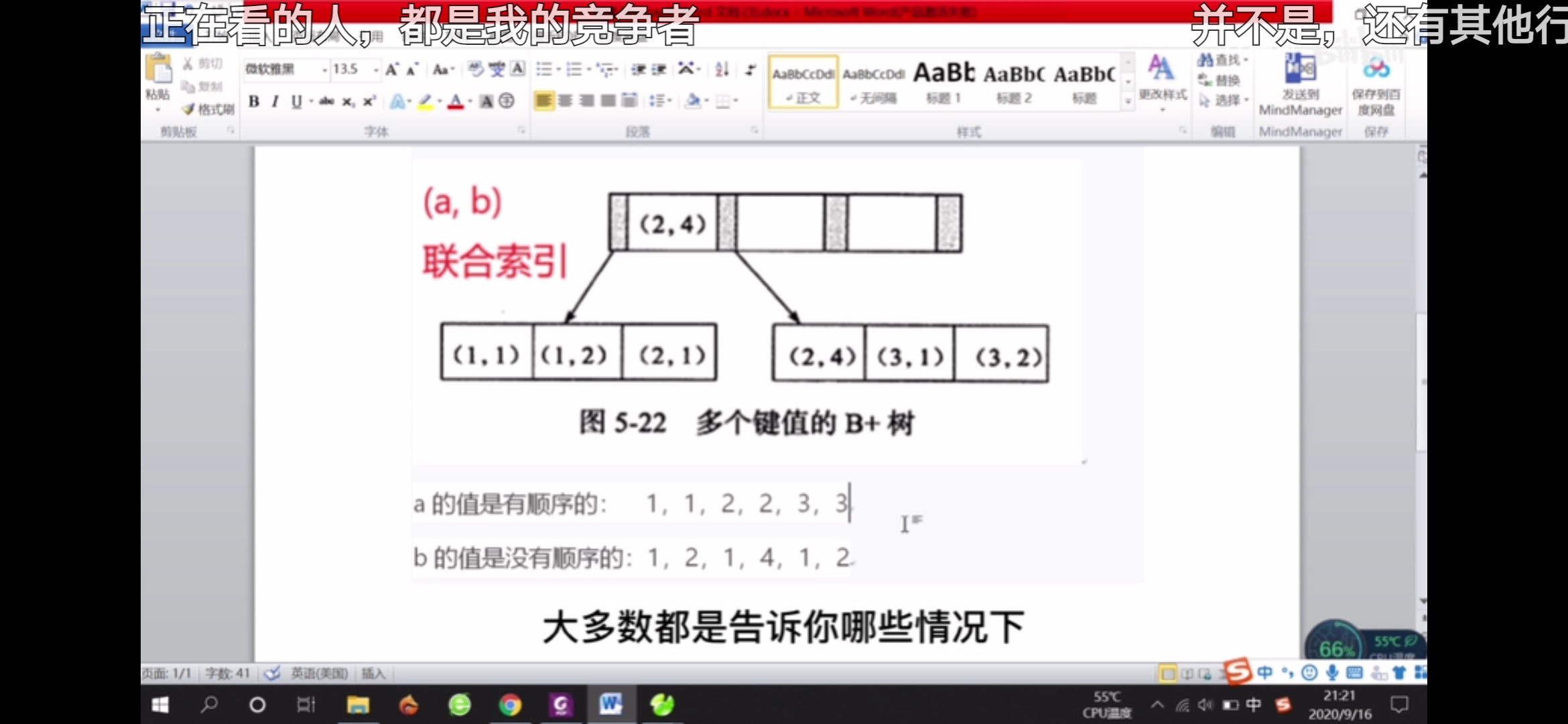
哈希算法虽然高，但也有它的局限性  
 1、 哈希索引只能操作：in、=，不支持范围查询，eg：>,<,!=  
原因：哈希索引比较的是经过哈希算法之后的哈希值，而哈希索引是无序的，用范围值的结果计算出来的哈希值并不一定和哈希索引中的排序是一样的。还有可能发生哈希碰撞。  
 2、 由于哈希索引本身是不排序的，所以任何时候都避免不了索引全表扫描。

#### （6）、为什么不是B树查找？



B树的优点：一个节点可以保存多个索引，通过度解决了红黑树的树深问题  
 缺点：由于非叶子节点也保存数据，导致一次IO读不到更多的索引

### mysql联合索引为什么会失效



如图：

关键原理：a索引是有序的，b索引是无序的。但在a索引相同的情况下，b索引是有序的

原因：遵循最佳左前缀原则。

注意：最佳左前缀法则在全是and=的时候不适用，mysql会自动调整为创建索引的顺序。

例：

EXPLAIN SELECT \* FROM contract\_management WHERE apply\_person ='唐超' AND STATUS=2

EXPLAIN SELECT \* FROM contract\_management WHERE STATUS=2 AND apply\_person ='唐超'

这两者效果一样，但是建议按照索引的顺序来书写

例1：select name,age from user where name=? and age=?;

联合索引name,和age,b+树是先根据第一个索引去查询到满足的数据索引位置，再根据第二个索引去查询最终满足的数据。

例2：select name,age from user where name=? and age>?and gener=?;

此联合索引范围查询之后的索引不会使用，第三个索引变得无序。当范围查找后，第三个索引变得无序，只能进行全表扫描。

例3：select name,age from user where like name “%name%”;

当左边%存在，不能使用索引，当左边%不存在，可以用到索引。



### mysql的索引原理和数据结构功能

Mysql采用B+树存储索引，索引的key存放于树的非叶子节点，数据保存于叶子节点data域中，当执行查询时，通过二分法查找索引key，查询出对应的叶子节点数据。

### B树和B+树的区别

区别：

1. B树每个节点都存放了索引数据和当前所有对应的表数据，而B+树每个非叶子节点只存储索引的key,而在叶子节点存放表数据；
2. B树由于每个节点既存索引数据又存表数据，导致每个节点所存索引更少，索引树的深度更深，查询时相对B+树更慢。
3. B树不支持范围查询，而B+树叶子节点为双向链表结构，支持范围查询。

### 5、mysql聚簇索引和非聚簇索引的区别

存储引擎MyISAM使用非聚簇索引，InolDB主键索引使用聚簇索引。

聚簇索引就是索引和数据存放一个文件，如InnoDB使用的是聚簇索引，将主键组织到一棵B+树中，而行数据就储存在叶子节点data域上，若使用"where id = 14"这样的条件查找主键，则按照B+ 树的检索算法即可查找到对应的叶节点，之后获得行数据。

Innerdb非聚簇索引索引和数据文件分别在不同的表中，非聚簇索引存储的叶子节点不存放数据，存放的是非聚簇索引的列和主键的值，查到主键之后，再使用主键（聚簇索引）查找所在行数据。MyISAM非聚簇索引叶子节点存放的索引列+主键+数据磁盘地址。

MyISAM使用非聚簇索引，在B+树中，每个叶子节点的data域都存放数据磁盘位置的指针，指向另一个文件的具体文件位置。

参考：http://www.liuzk.com/410.html

### 6、使用mysql索引都有什么原则

1、 对于查询频率高的字段创建索引；

2、 对排序、分组、联合查询频率高的字段创建索引；

3、 索引的数目不宜太多 原因：

a、每创建一个索引都会占用相应的物理空间；

b、过多的索引会导致insert、update、delete语句的执行效率降低；

4、选择唯一性索引 唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。

5、尽量使用数据量少的索引 如果索引的值很长，那么查询的速度会受到影响。

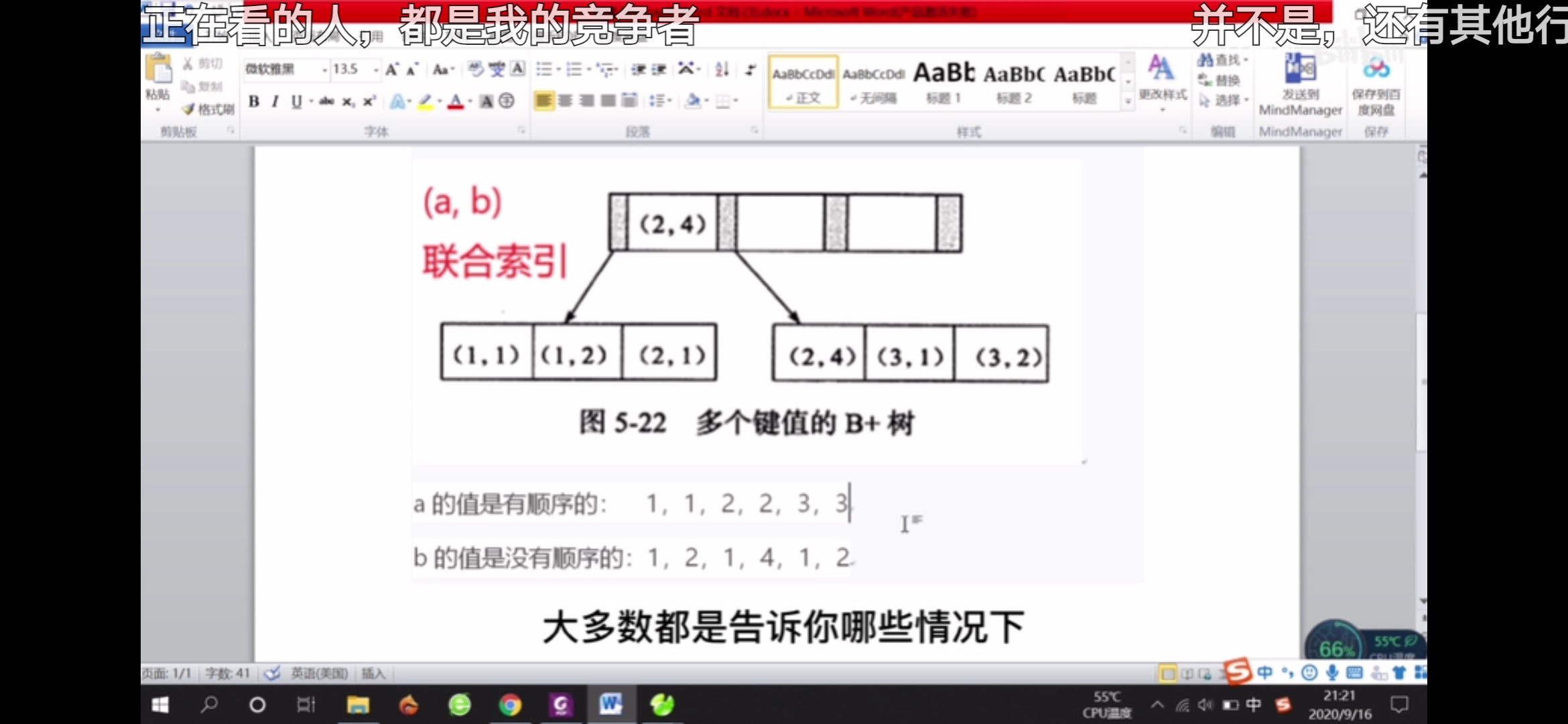
6、尽量使用前缀来索引 如果索引字段的值很长，最好使用值的前缀来索引。

7、删除不再使用或者很少使用的索引. 表中的数据被大量更新，或者数据的使用方式被改变后，原有的一些索引可能不再需要。

### 7、不同的存储引擎是如何进行实际存储的

参考：https://blog.csdn.net/ggh0314/article/details/114663639

### 8、mysql的组合索引结构是什么样的

如图所示：先第一个索引a查找，在找到a位置后，再对b进行查找。

### mysql索引如何进行优化

查询慢的原因：

1.硬件问题。如网络速度慢，内存不足，I/O吞吐量小，磁盘空间满了等。

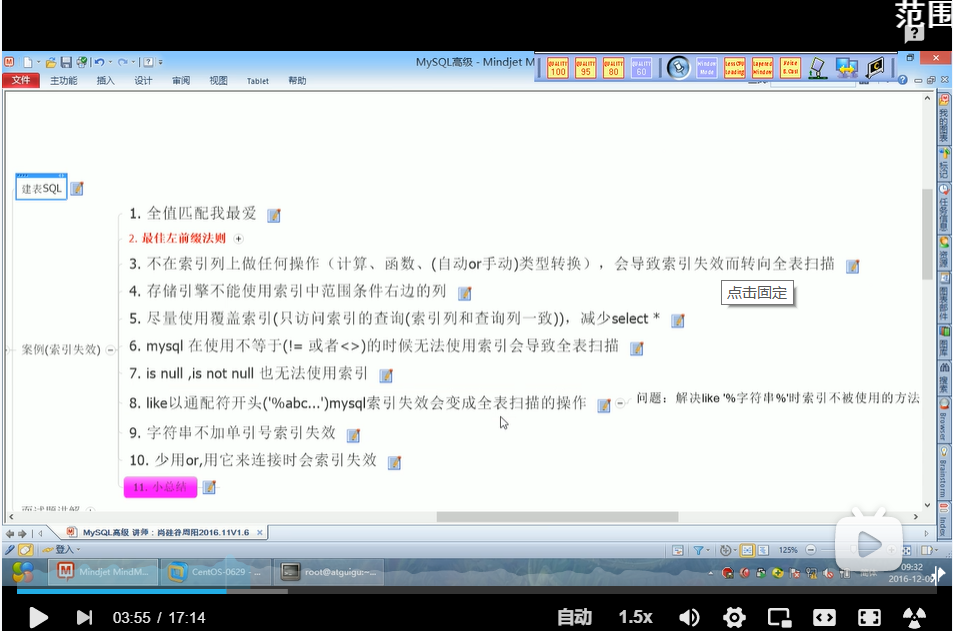
2.没有索引或者索引失效。（一般在互联网公司，DBA会在半夜把表锁了，重新建立一遍索引，因为当你删除某个数据的时候，索引的树结构就不完整了。所以互联网公司的数据做的是假删除.一是为了做数据分析,二是为了不破坏索引 ）

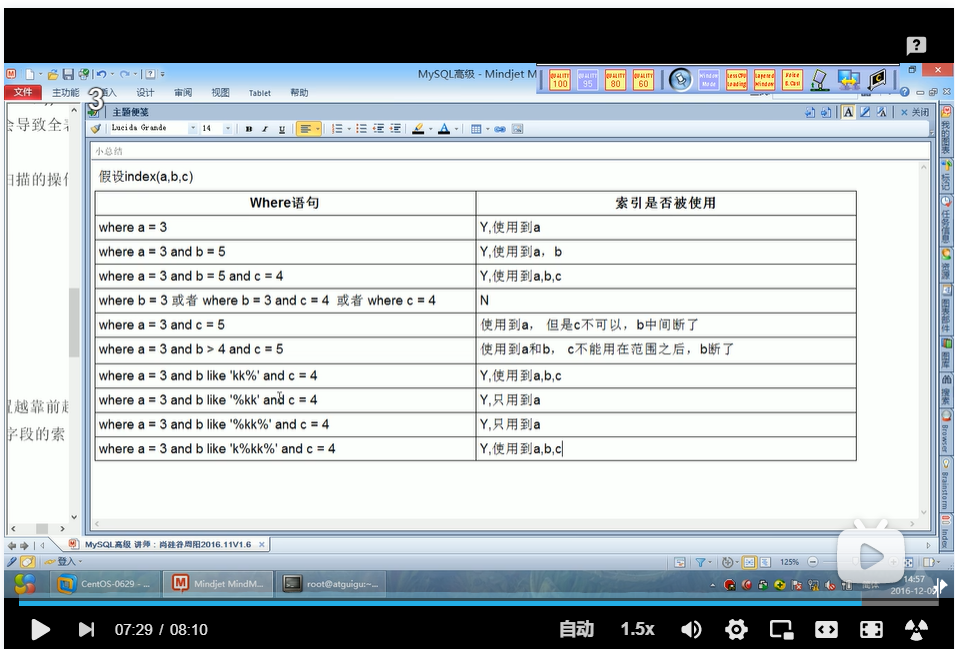
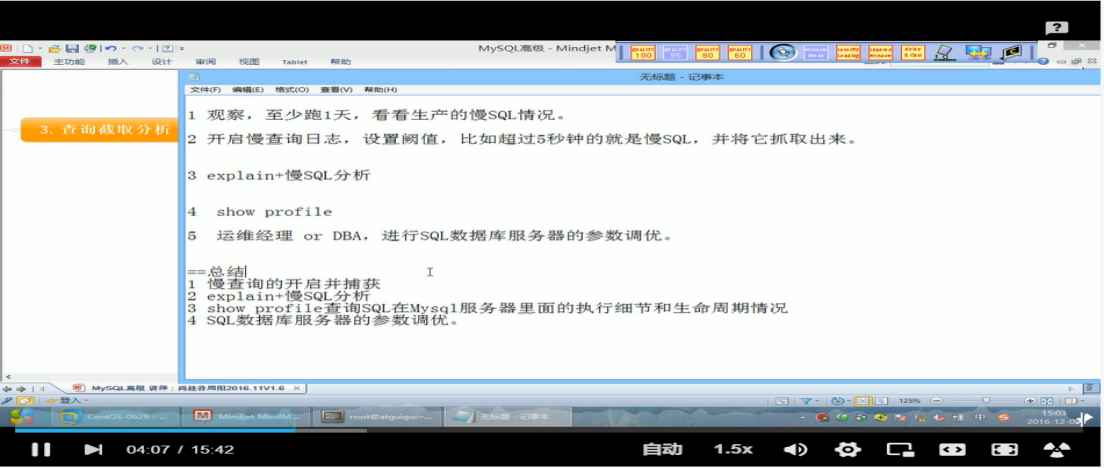
3.数据过多（分库分表）

4.服务器调优及各个参数设置（调整my.cnf）

10、覆盖索引

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。



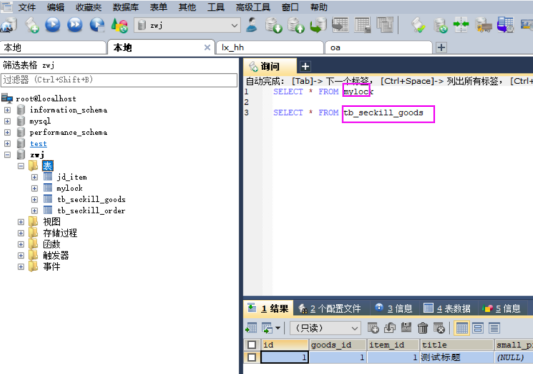
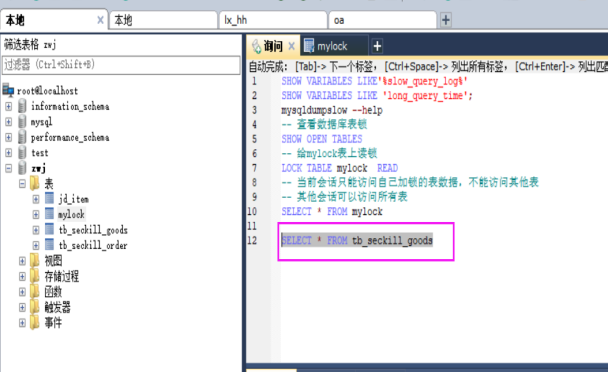


Sql慢查询的开启并捕获：<https://blog.csdn.net/qq_39291929/article/details/81561388>

### mysql读写锁

读锁：

* 1. 当mysql为一个表开启读锁的时候,其他进程包括进程本身没有权利去修改这张表的内容,也没有权利更新,但是可以读取表里面的内容



-- 查看数据库表锁

SHOW OPEN TABLES

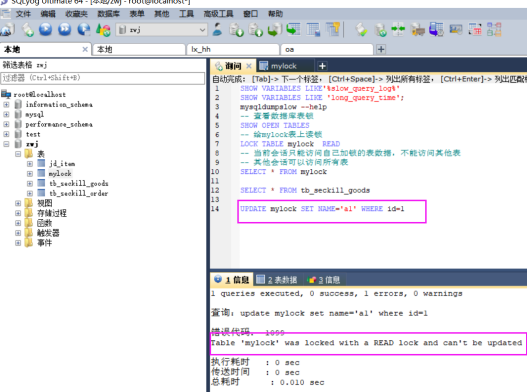
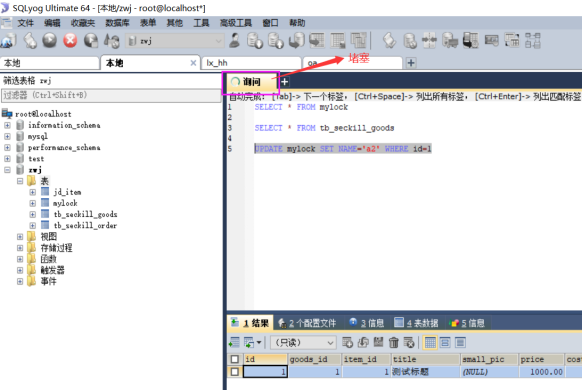
-- 给mylock表上读锁

LOCK TABLE mylock READ

-- 当前会话只能访问自己加锁的表数据，不能访问其他表

-- 其他会话可以访问所有表

SELECT \* FROM mylock



SELECT \* FROM tb\_seckill\_goods

-- 释放读锁

UNLOCK TABLES

本会话不能对加读锁的表进行增删改写操作；其他会话写操作会堵塞，知道加锁的表释放锁。

网上参考：<https://blog.csdn.net/qq_44766883/article/details/105879308>

写锁：

-- mylock加写锁

LOCK TABLE mylock WRITE

只有当前会话能增删改查，其他会话无法任何操作（阻塞，等待加锁会话释放锁）

网上参考：<https://www.cnblogs.com/CyLee/p/5575152.html>

查看锁竞争情况：SHOW STATUS LIKE 'table%'

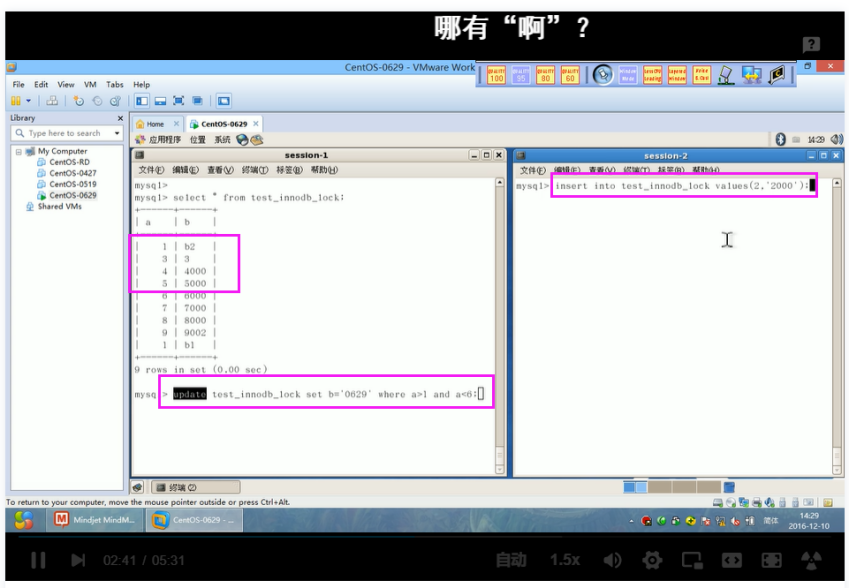
### 行锁

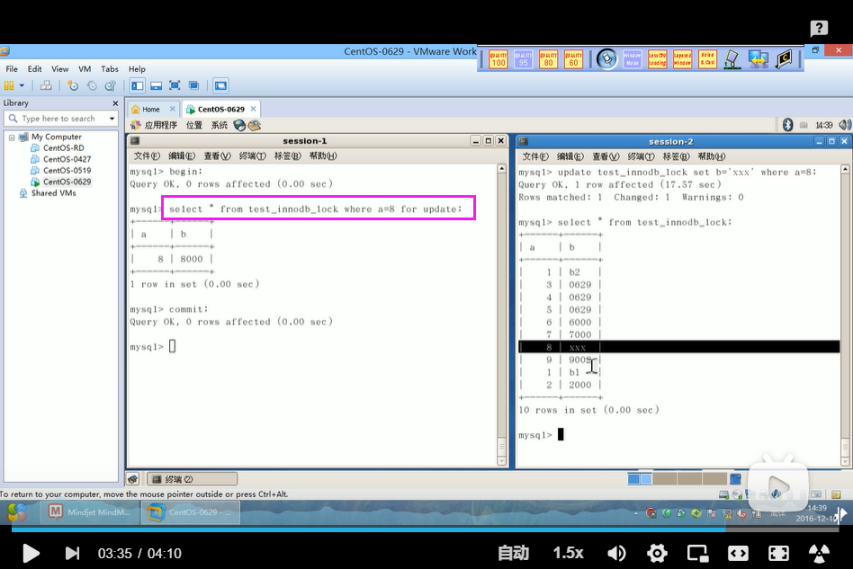
* 1. 索引失效，行锁变表锁

例子：如varchar类型的字段，主动不加’’,致索引失效，当前会话可以增删改，未提交完成的话(可设置手动提交 set commit 0效果更明显)，其他会话会阻塞。



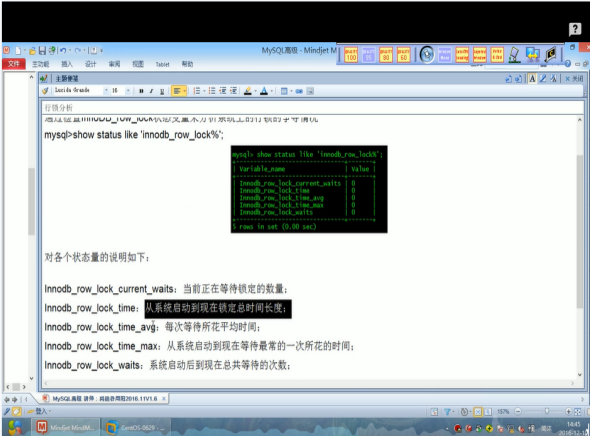
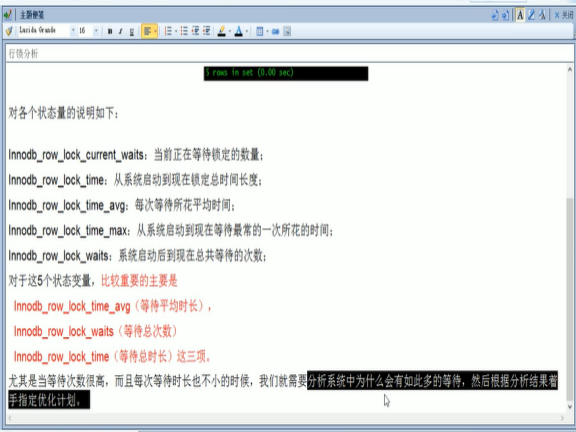
间隙锁的危害



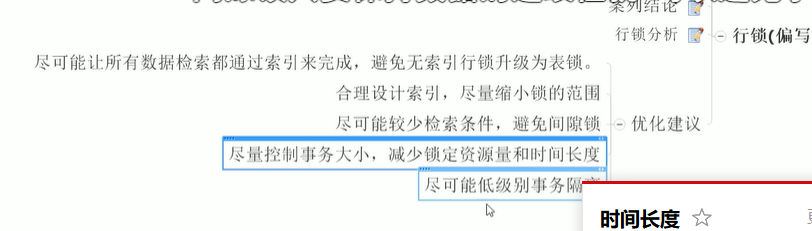


人为给表的某行加锁，select \* from table(表明) where id= 1 for update;

若当前会话未提交完成，其他会话阻塞



查看innodb锁情况

SHOW STATUS LIKE 'innodb\_row\_lock%'

### mysql缓存池

学习地址：一起看更容易懂

<https://mp.weixin.qq.com/s/AbiYMbDUJsMWK-_g0mFNwg>

https://blog.csdn.net/wuhenyouyuyouyu/article/details/93377605

## 消息队列

### RabbitMQ

### RabbitMQ 的使用场景有哪些？

* 抢购活动，削峰填谷，防止系统崩塌。
* 延迟信息处理，比如 10 分钟之后给下单未付款的用户发送邮件提醒。

解耦系统，对于新增的功能可以单独写模块扩展，比如用户确认评价之后，新增了给用户返积分的功能，这个时候不用在业务代码里添加新增积分的功能，只需要把新增积分的接口订阅确认评价的消息队列即可，后面再添加任何功能只需要订阅对应的消息队列即可。

## Spring

### 描述一下Spring Bean的生命周期？

答：

1. 加载类获取BeanDefinition。如图：
2. 确定构造方法，进行实例化得到对象。
3. 回调aware方法，对bean需要的依赖进行注入。如图：
4. 调用BeanPostProcessor的初始化前置方法：主要对bean进行扩展。如图：
5. 调用初始化对象方法，初始化对象。
6. 调用BeanPostProcessor的初始化后置方法：主要对bean进行扩展。
7. 若生成单例对象，会被容器放入单例池。
8. 使用Bean。
9. Spring容器关闭时调用DisposableBean中destory()方法 。

参考：

<https://blog.csdn.net/weixin_40009099/article/details/111019951?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-0.control&spm=1001.2101.3001.4242>

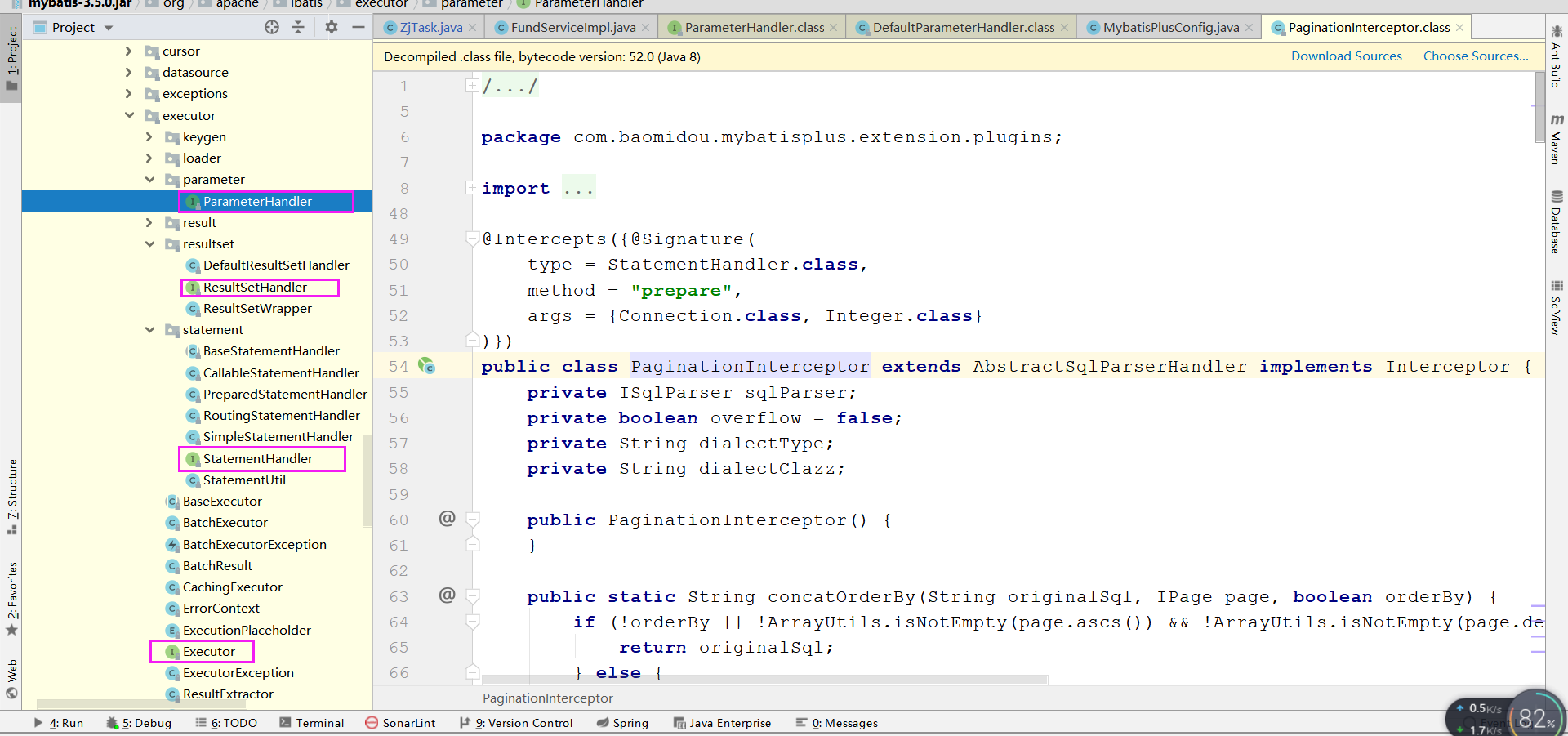
### 2、Spring 框架中都用到了哪些设计模式？

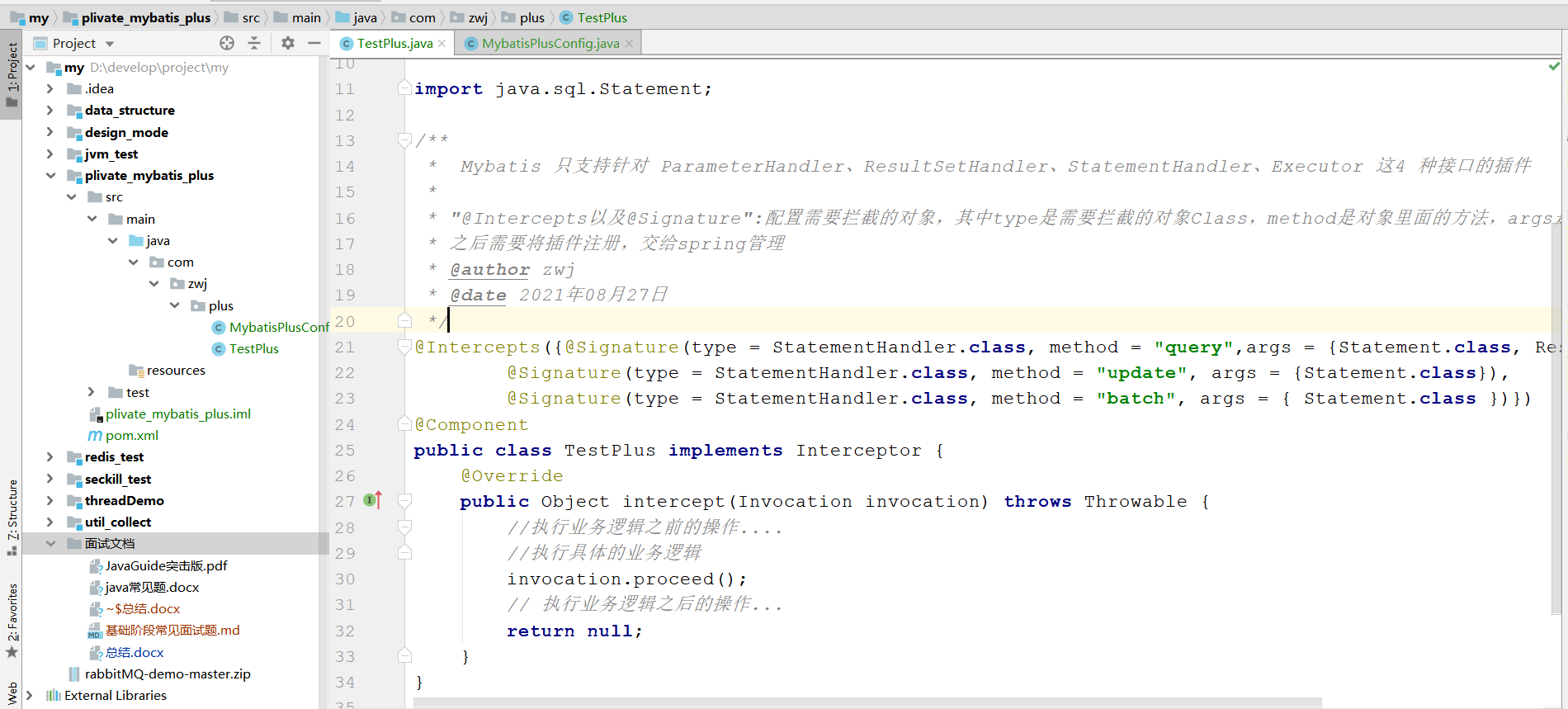
## Mybatis

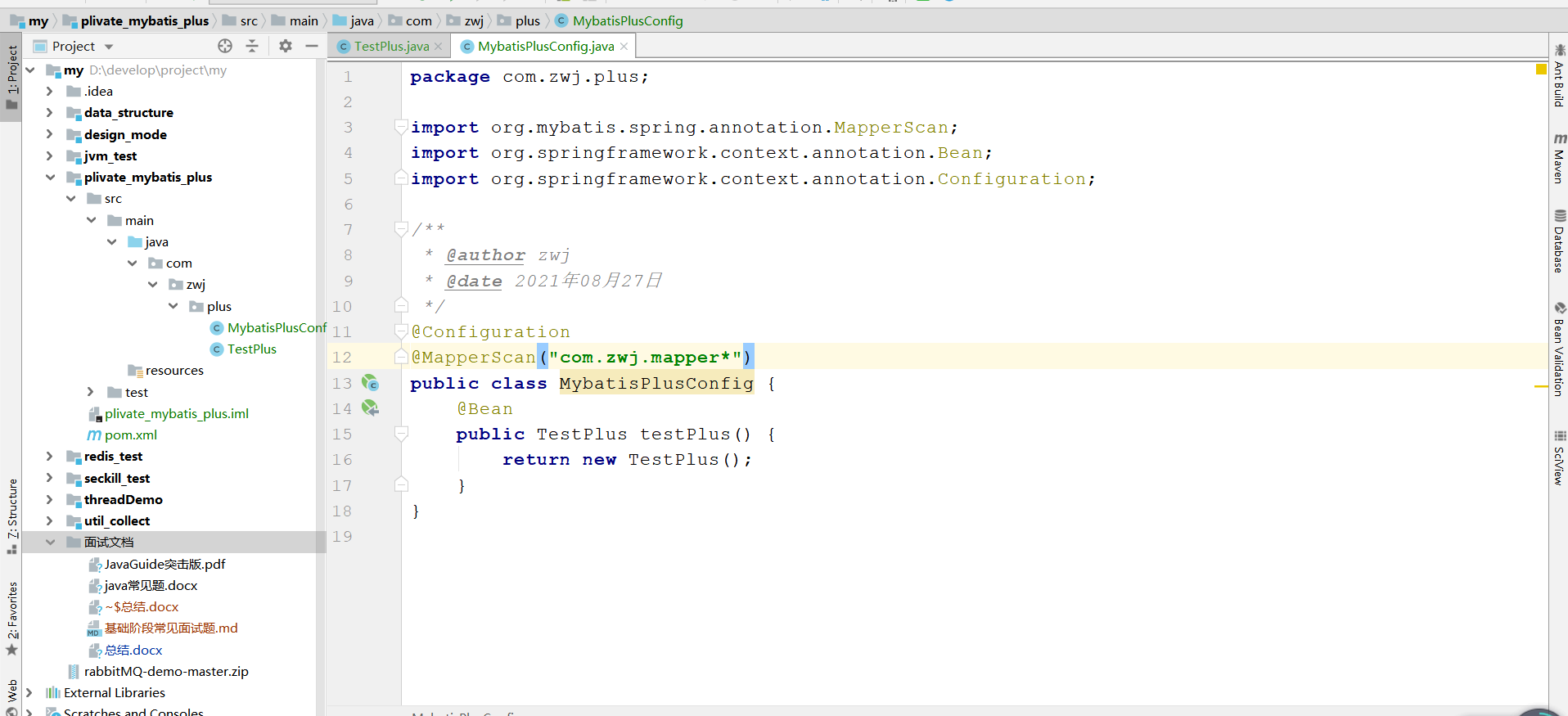
## 1、简述 Mybatis 的插件运行原理，如何编写一个插件。

答： Mybatis 只支持针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这4 种接口的插件， Mybatis 使用 JDK 的动态代理， 为需要拦截的接口生成代理对象以实现接口方法拦截功能， 每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的invoke() 方法， 拦截那些你指定需要拦截的方法。

具体如图所示：

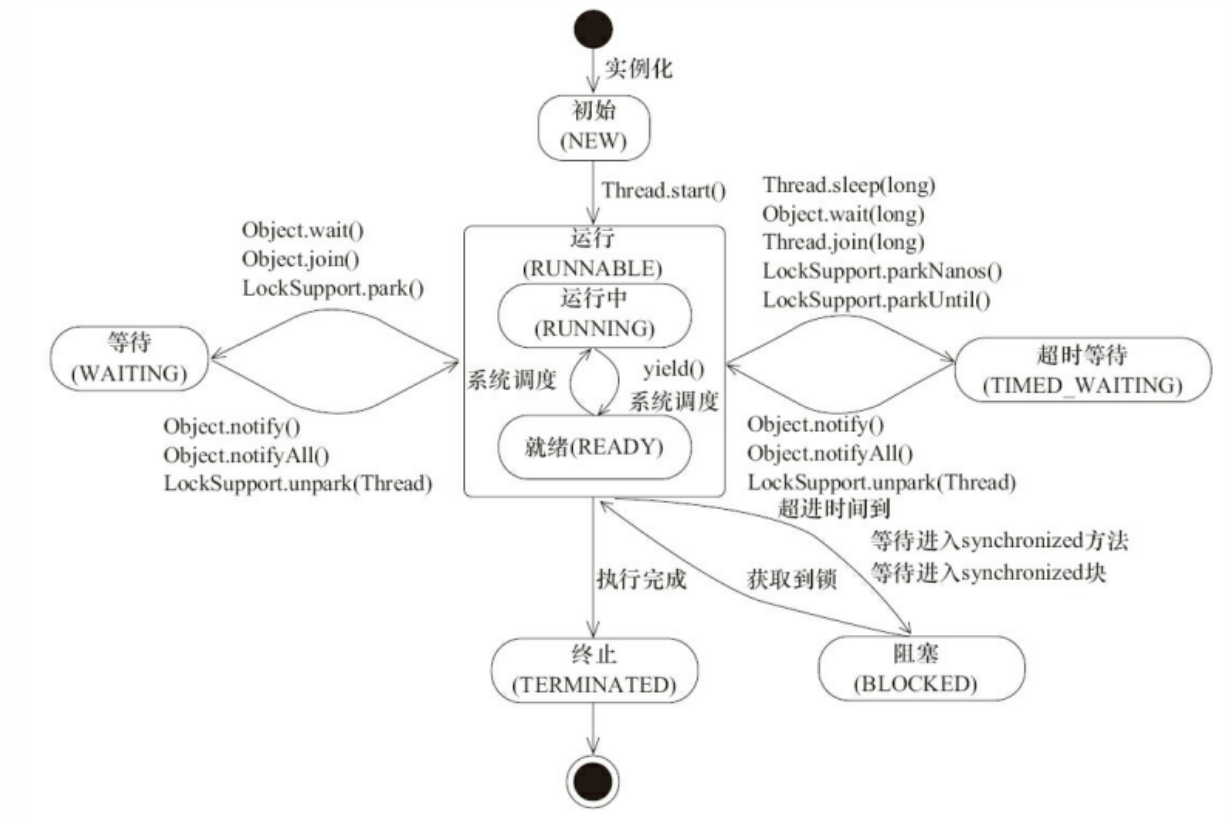






# 多线程

## 1、线程的生命周期



1. 创建：通过new Tread()或者线程池创建线程；
2. 就绪：调用线程得start方法，加入锁池中，等待获取锁；
3. 运行：线程获取锁，运行中；
4. 等待/阻塞：
   1. 等待阻塞：运行的线程执行wait方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify或notifyAll方法才能被唤醒，wait是object类的方法
   2. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。
   3. 其他阻塞：运行的线程执行sleep或join方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状 态。当sleep状态超时、join等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。sleep是Thread类的方法

5、死亡：线程运行结束或者异常结束。

## 2、[偏向锁、轻量级锁及重量级锁synchronized原理](https://www.cnblogs.com/deltadeblog/p/9559035.html)

**Java对象头与Monitor**

java对象头是实现synchronized的锁对象的基础，synchronized使用的锁对象是存储在Java对象头里的。

**对象头包含两部分：Mark Word 和 Class Metadata Address**



其中Mark Word在默认情况下存储着对象的HashCode、分代年龄、锁标记位等以下是32位JVM的Mark Word默认存储结构



由于对象头的信息是与对象自身定义的数据没有关系的额外存储成本，因此考虑到JVM的空间效率，Mark Word 被设计成为一个非固定的数据结构，以便存储更多有效的数据，它会根据对象本身的状态复用自己的存储空间，如32位JVM下，除了上述列出的Mark Word默认存储结构外，还有如下可能变化的结构：



偏向所锁，轻量级锁都是乐观锁，重量级锁是悲观锁。  
 一个对象刚开始实例化的时候，没有任何线程来访问它的时候。它是可偏向的，意味着，它现在认为只可能有一个线程来访问它，所以当第一个线程来访问它的时候，它会偏向这个线程，此时，对象持有偏向锁。偏向第一个线程，这个线程在修改对象头成为偏向锁的时候使用CAS操作，并将对象头中的ThreadID改成自己的ID，之后再次访问这个对象时，只需要对比ID，不需要再使用CAS在进行操作。  
 一旦有第二个线程访问这个对象，因为偏向锁不会主动释放，所以第二个线程可以看到对象时偏向状态，这时表明在这个对象上已经存在竞争了，检查原来持有该对象锁的线程是否依然存活，如果挂了，则可以将对象变为无锁状态，然后重新偏向新的线程，如果原来的线程依然存活，则马上执行那个线程的操作栈，检查该对象的使用情况，如果仍然需要持有偏向锁，则偏向锁升级为轻量级锁，（偏向锁就是这个时候升级为轻量级锁的）。如果不存在使用了，则可以将对象回复成无锁状态，然后重新偏向。  
轻量级锁认为竞争存在，但是竞争的程度很轻，一般两个线程对于同一个锁的操作都会错开，或者说稍微等待一下（自旋），另一个线程就会释放锁。 但是当自旋超过一定的次数（默认最多自旋10次，便会升级为重量级锁），或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁膨胀为重量级锁，重量级锁使除了拥有锁的线程以外的线程都阻塞，防止CPU空转。

参考：https://www.cnblogs.com/deltadeblog/p/9559035.html

## 3、什么是CAS，有哪些应用。

CAS是英文单词Compare And Swap的缩写，翻译过来就是比较并替换。

CAS机制当中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。

更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。

1. 在内存地址V当中，存储着值为10的变量。
2. 此时线程1想要把变量的值增加1。对线程1来说，旧的预期值A=10，要修改的新值B=11。、
3. 在线程1要提交更新之前，另一个线程2抢先一步，把内存地址V中的变量值率先更新成了11。
4. 线程1开始提交更新，首先进行****A和地址V的实际值比较（Compare）****，发现A不等于V的实际值，提交失败。
5. 线程1重新获取内存地址V的当前值，并重新计算想要修改的新值。此时对线程1来说，A=11，B=12。这个重新尝试的过程被称为****自旋****。
6. 这一次比较幸运，没有其他线程改变地址V的值。线程1进行****Compare****，发现A和地址V的实际值是相等的。
7. 线程1进行****SWAP****，把地址V的值替换为B，也就是12。

从思想上来说，Synchronized属于****悲观锁****，悲观地认为程序中的并发情况严重，所以严防死守。CAS属于****乐观锁****，乐观地认为程序中的并发情况不那么严重，所以让线程不断去尝试更新。

CAS的缺点：

1.CPU开销较大

在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性

CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

参考：https://blog.csdn.net/skycto\_com/article/details/100167742

## 4、说一下ThreadLocal

ThreadLocal是每个线程私有的本地变量，可以缓存一些数据，该线程可以在任意时刻、任意⽅法中获取缓存的数据。比如：数据连接、事务信息就存储在ThreadLocal本地缓存中。

每个线程都存在一个ThreadLocalMap,ThreadLocal对象为ThreadLocalMap的key,value值为缓存的值，但是ThreadLocal和ThreadLocalMap是弱引用，也就是说当外部强引用断开ThreadLocal连接时，GC垃圾回收就会回收ThreadLocal对象，ThreadLocalMap的key就为空值，而ThreadLocalMap的key和value是强引用，造成此内存数据回收失败，进而随着时间的推移有可能造成内存泄漏。只有thead线程退出以后,value的强引用链条才会断掉，但如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链

内存泄漏避免办法：

对ThreadLocal对象私有化静态处理：private static ThreadLocal threadLocal=new ThreadLocal();

每次使用完ThreadLocal都调用他的remove()方法。

## 5、为什么用线程池？解释下线程池参数？

1、降低资源消耗；提高线程利用率，降低创建和销毁线程的消耗。

2、提高响应速度；任务来了，直接有线程可用可执行，而不是先创建线程，再执行。

3、提高线程的可管理性；线程是稀缺资源，使用线程池可以统一分配调优监控。

线程池参数：

corePoolSize 代表核心线程数，也就是正常情况下创建工作的线程数，这些线程创建后并不会 消除，而是一种常驻线程

maxinumPoolSize 代表的是最大线程数，它与核心线程数相对应，表示最大允许被创建的线程数，比如当前任务较多，将核心线程数都用完了，还无法满足需求时，此时就会创建新的线程，但是线程池内线程总数不会超过最大线程数

keepAliveTime 、 unit 表示超出核心线程数之外的线程的空闲存活时间，也就是核心线程不会消除，但是超出核心线程数的部分线程如果空闲一定的时间则会被消除,我们可以通过setKeepAliveTime 来设置空闲时间

workQueue 用来存放待执行的任务，假设我们现在核心线程都已被使用，还有任务进来则全部放入队列，直到整个队列被放满但任务还再持续进入则会开始创建新的线程

ThreadFactory 实际上是一个线程工厂，用来生产线程执行任务。我们可以选择使用默认的创建工厂，产生的线程都在同一个组内，拥有相同的优先级，且都不是守护线程。当然我们也可以选择自定义线程工厂，一般我们会根据业务来制定不同的线程工厂

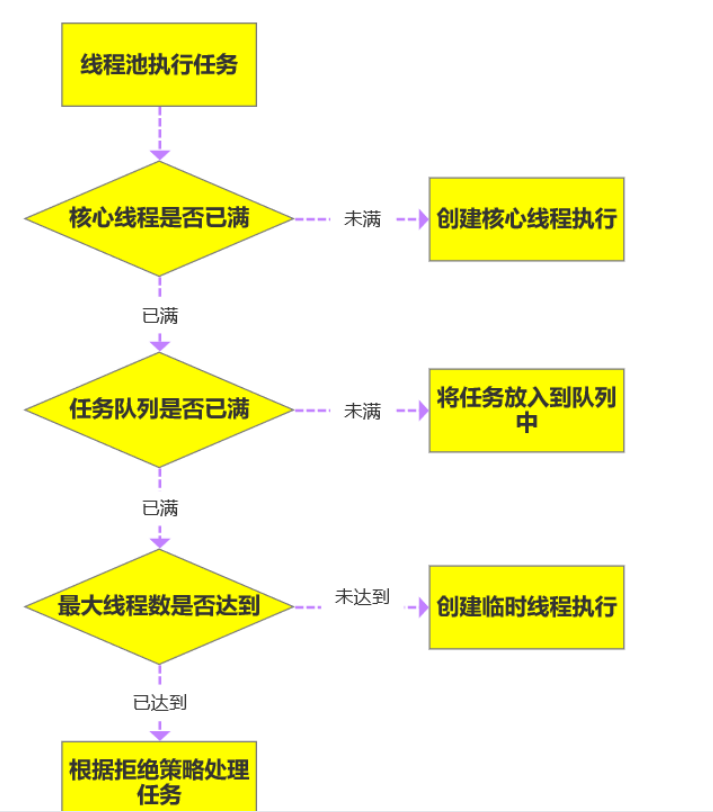
Handler 任务拒绝策略，有两种情况，第一种是当我们调用 shutdown 等方法关闭线程池后，这时候即使线程池内部还有没执行完的任务正在执行，但是由于线程池已经关闭，我们再继续想线程池提交任务就会遭到拒绝。另一种情况就是当达到最大线程数，线程池已经没有能力继续处理新提交的任务时，这是也就拒绝

## 6、简述线程池处理流程

1、线程池执行任务，查看核心线程是否已满，未满创建核心线程执行，已满执行第二步；

2、将任务放入任务队列中，查看任务队列是否已满，未满将任务加人队列，已满执行第三步；

3、查看最大线程数是否已满，未满创建临时线程执行，已满根据拒绝策略处理任务。



## 7、线程池拒绝策略

线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列也已经排满了，再也塞不下新任务了。这时候我们就需要拒绝策略机制合理的处理这个问题。

JDK 内置的拒绝策略如下：

1. AbortPolicy ： 直接抛出异常，阻止系统正常运行。

2. CallerRunsPolicy ： 只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。

3. DiscardOldestPolicy ： 丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。

4. DiscardPolicy ： 该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢失，这是最好的一种方案。

以上内置拒绝策略均实现了 RejectedExecutionHandler 接口，若以上策略仍无法满足实际需要，完全可以自己扩展 RejectedExecutionHandler 接口。

## 8、线程池中阻塞队列的作用？为什么是先添加列队而不是先创建最大线程？

阻塞队列作用：

阻塞队列可以保证任务队列中没有任务时阻塞获取任务的线程，使得线程进入wait状态，释放cpu资源。阻塞队列自带阻塞和唤醒的功能，不需要额外处理，无任务执行时,线程池利用阻塞队列的take方法挂起，从而维持核心线程的存活、不至于一直占用cpu资源。

先队列不是创建最大线程原因：

在创建新线程的时候，是要获取全局锁的，这个时候其它的就得阻塞，影响了整体效率。

## 9、线程池中线程复用原理

在线程池中，同一个线程可以从阻塞队列中不断获取新任务来执行，其核心原理在于线程池对Thread 进行了封装，并不是每次执行任务都会调用 Thread.start() 来创建新线程，而是让每个线程去执行一个“循环任务”，在这个“循环任务”中不停检查是否有任务需要被执行，如果有则直接执行，也就是调用任务中的 run 方法，将 run 方法当成一个普通的方法执行，通过这种方式只使用固定的线程就将所有任务的 run 方法串联起来。

如图所示：



## 10、sleep()、wait()、join()、yield()的区别

1.锁池

所有需要竞争同步锁的线程都会放在锁池当中，比如当前对象的锁已经被其中一个线程得到，则其他线程需要在这个锁池进行等待，当前面的线程释放同步锁后锁池中的线程去竞争同步锁，当某个线程得到后会进入就绪队列进行等待cpu资源分配。

2.等待池

当我们调用wait（）方法后，线程会放到等待池当中，等待池的线程是不会去竞争同步锁。只有调用了notify（）或notifyAll()后等待池的线程才会开始去竞争锁，notify（）是随机从等待池选出一个线程放到锁池，而notifyAll()是将等待池的所有线程放到锁池当中。

1、sleep 是 Thread 类的静态本地方法，wait 则是 Object 类的本地方法。

2、sleep方法不会释放lock，但是wait会释放，而且会加入到等待队列中。

3、sleep方法不依赖于同步器synchronized，但是wait需要依赖synchronized关键字。

4、sleep不需要被唤醒（休眠之后推出阻塞），但是wait需要（不指定时间需要被别人中断）。

5、sleep 一般用于当前线程休眠，或者轮循暂停操作，wait 则多用于多线程之间的通信。

6、sleep 会让出 CPU 执行时间且强制上下文切换，而 wait 则不一定，wait 后可能还是有机会重新竞争到锁继续执行的。

yield（）执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格所以有可能cpu下次进行线程调度还会让这个线程获取到执行权继续执行

join（）执行后线程进入阻塞状态，例如在线程B中调用线程A的join（），那线程B会进入到阻塞队列，直到线程A结束或中断线程

## 11、线程上下文切换

巧妙地利用了时间片轮转的方式, CPU 给每个任务都服务一定的时间，然后把当前任务的状态保存下来，在加载下一任务的状态后，继续服务下一任务，任务的状态保存及再加载, 这段过程就叫做上下文切换。时间片轮转的方式使多个任务在同一颗 CPU 上执行变成了可能。

## 12、对守护线程的理解

为所有非守护线程提供服务的线程；任何一个守护线程都是整个JVM中所有非守护线程的保姆；当其他线程都结束，守护线程才会结束。

注意： 由于守护线程的终止是自身无法控制的，因此千万不要把IO、File等重要操作逻辑分配给它；因为它不靠谱；

此处可以以GC垃圾回收举例。GC垃圾回收线程：就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread,程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。

## 13、并发三大特性

原子性

原子性是指在一个操作中cpu不可以在中途暂停然后再调度，即不被中断操作，要不全部执行完成，要不都不执行。就好比转账，从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，

往账户B加上1000元。2个操作必须全部完成。

如：count++;运算过程

1：将 count 从主存读到工作内存中的副本中

2：+1的运算

3：将结果写入工作内存

4：将工作内存的值刷回主存(什么时候刷入由操作系统决定，不确定的)

那程序中原子性指的是最小的操作单元，比如自增操作，它本身其实并不是原子性操作，分了3步的，包括读取变量的原始值、进行加1操作、写入工作内存。所以在多线程中，有可能一个线程还没自增完，可能才执行到第二部，另一个线程就已经读取了值，导致结果错误。那如果我们能保证自增操作是一个原子性的操作，那么就能保证其他线程读取到的一定是自增后的数据。

可见性

当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

若两个线程在不同的cpu，那么线程1改变了i的值还没刷新到主存，线程2又使用了i，那么这个i值肯定还是之前的，线程1对变量的修改线程没看到这就是可见性问题。

//线程1 boolean stop = false; while(!stop){ doSomething(); }//线程2 stop = true;

如果线程2改变了stop的值，线程1一定会停止吗？不一定。当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

有序性

虚拟机在进行代码编译时，对于那些改变顺序之后不会对最终结果造成影响的代码，虚拟机不一定会按照我们写的代码的顺序来执行，有可能将他们重排序。实际上，对于有些代码进行重排序之后，虽然对变量的值没有造成影响，但有可能会出现线程安全问题。

int a = 0; bool flag = false; public void write() { a = 2; //1 flag = true; //2 }public void multiply() { if (flag) { //3 int ret = a \* a;//4 } }

write方法里的1和2做了重排序，线程1先对flag赋值为true，随后执行到线程2，ret直接计算出结果，再到线程1，这时候a才赋值为2,很明显迟了一步

volatile本身就包含了禁止指令重排序的语义，而synchronized关键字是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则明确的。

synchronized关键字同时满足以上三种特性，但是volatile关键字不满足原子性。

在某些情况下，volatile的同步机制的性能确实要优于锁(使用synchronized关键字或

java.util.concurrent包里面的锁)，因为volatile的总开销要比锁低。

我们判断使用volatile还是加锁的唯一依据就是volatile的语义能否满足使用的场景(原子性)

## Volatile

1、保证被volatile修饰的共享变量对所有线程总是可见的，也就是当一个线程修改了一个被volatile修饰共享变量的值，新值总是可以被其他线程立即得知。

2. 禁止指令重排序优化

## 15、java锁

### 乐观锁

乐观锁是一种乐观思想，即认为读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读-比较-写的操作。

java 中的乐观锁基本都是通过 CAS 操作实现的，CAS 是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。

### 悲观锁

悲观锁是就是悲观思想，即认为写多，遇到并发写的可能性高，每次去拿数据的时候都认为别人会修改，所以每次在读写数据的时候都会上锁，这样别人想读写这个数据就会 block 直到拿到锁。

java中的悲观锁就是Synchronized,AQS框架下的锁则是先尝试cas乐观锁去获取锁，获取不到，才会转换为悲观锁，如 RetreenLock。

### 自旋锁

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。线程自旋是需要消耗 cup 的，说白了就是让 cup 在做无用功，如果一直获取不到锁，那线程也不能一直占用 cup 自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用 cpu 做无用功，占着 XX 不 XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要 cup 的线程又不能获取到 cpu，造成 cpu 的浪费。所以这种情况下我们要关闭自旋锁；

自旋锁时间阈值（1.6 引入了适应性自旋锁）

自旋锁的目的是为了占着 CPU 的资源不释放，等到获取到锁立即进行处理。但是如何去选择 自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用CPU 资源，进而会影响整体系统的性能。因此自旋的周期选的额外重要！

JVM 对于自旋周期的选择，jdk1.5 这个限度是一定的写死的，在 1.6 引入了适应性自旋锁，适应性自旋锁意味着自旋的时间不在是固定的了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定，基本认为一个线程上下文切换的时间是最佳的一个时间，同时 JVM 还针对当前 CPU 的负荷情况做了较多的优化，如果平均负载小于 CPUs 则一直自旋，如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞，如果正在自旋的线程发现 Owner 发生了变化则延迟自旋 时间（自旋计数）或进入阻塞，如果 CPU 处于节电模式则停止自旋，自旋时间的最坏情况是 CPU的存储延迟（CPU A 存储了一个数据，到 CPU B 得知这个数据直接的时间差），自旋时会适当放弃线程优先级之间的差异。

自旋锁的开启

JDK1.6 中-XX:+UseSpinning 开启；

-XX:PreBlockSpin=10 为自旋次数；

JDK1.7 后，去掉此参数，由 jvm 控制；

### Synchronized 同步锁

synchronized 它可以把任意一个非 NULL 的对象当作锁。他属于独占式的悲观锁，同时属于可重入锁。

Synchronized 作用范围

1. 作用于方法时，锁住的是对象的实例(this)；

2. 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带PermGen（jdk1.8 则是 metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；

3. synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

Synchronized 核心组件

1) Wait Set：哪些调用 wait 方法被阻塞的线程被放置在这里；

2) Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中；

3) Entry List：Contention List 中那些有资格成为候选资源的线程被移动到 Entry List 中；

4) OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为 OnDeck；

5) Owner：当前已经获取到所资源的线程被称为 Owner；

6) !Owner：当前释放锁的线程。

# Jvm

图灵诸葛老师讲解

## 1、栈、栈帧

栈：存放多个栈帧。一个main线程中，每个方法都有一个栈帧。

栈帧定义：线程里每个方法分配的内存空间

栈帧结构：存放线程的局部变量，操作数栈，动态链接，方法出口

局部变量：方法类的局部变量；

操作数栈：记录了一个方法执行过程中的字节码指令，他往操作数栈中进行入栈和出栈；

动态链接：每一个栈帧都包含一个指向运行时常量池中该栈帧所属的方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。如果你看了[字节码文件构成](https://github.com/leosanqing/Java-Notes/blob/master/JVM/%E8%99%9A%E6%8B%9F%E6%9C%BA%E6%89%A7%E8%A1%8C%E5%AD%90%E7%B3%BB%E7%BB%9F/%E5%AD%97%E8%8A%82%E7%A0%81%E6%96%87%E4%BB%B6%E7%BB%93%E6%9E%84/%E8%83%BD%E7%9C%8B%E6%87%82%E7%9A%84%E5%AD%97%E8%8A%82%E7%A0%81-%E4%B8%8A.md)和[类加载过程](https://github.com/leosanqing/Java-Notes/blob/master/JVM/%E8%99%9A%E6%8B%9F%E6%9C%BA%E6%89%A7%E8%A1%8C%E5%AD%90%E7%B3%BB%E7%BB%9F/%E7%B1%BB%E5%8A%A0%E8%BD%BD%E8%BF%87%E7%A8%8B/%E7%B1%BB%E5%8A%A0%E8%BD%BD.md)，你应该知道，字节码文件中有很多符号引用。这些符号引用一部分会在类加载的解析阶段或者第一次使用的时候转化为直接引用，这种转化称为静态解析；

方法出口：即返回地址。一个方法执行后，只有两种方法可以退出：

return，正常退出.

异常，并且不在该方法中处理.

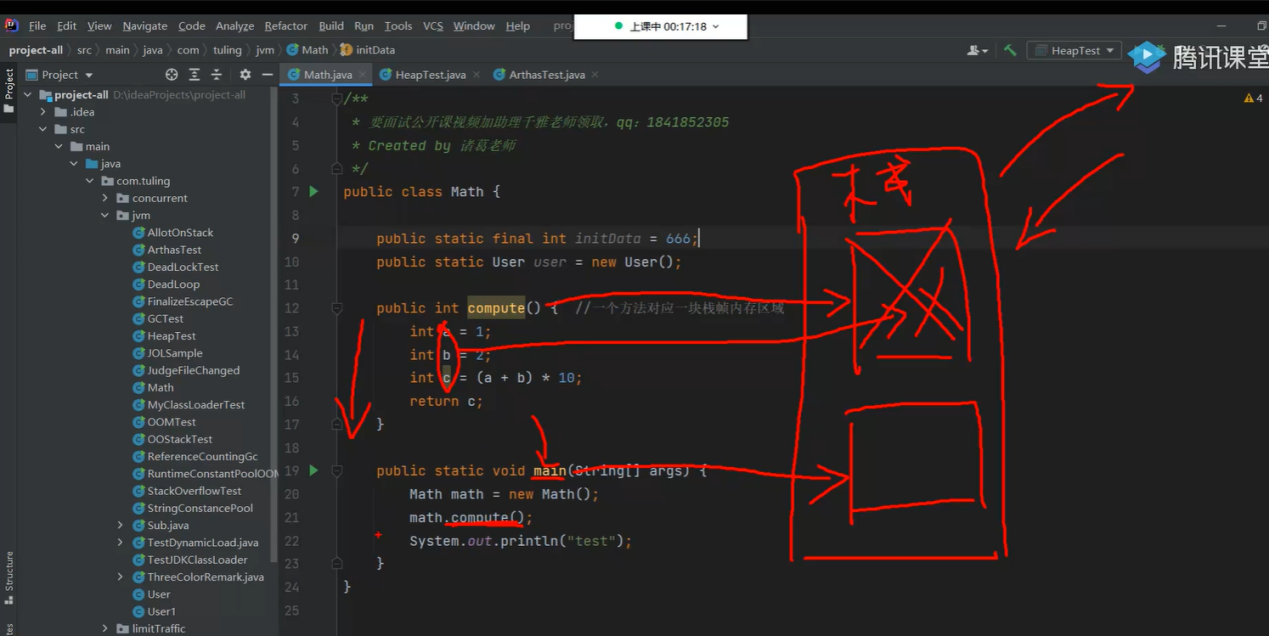
方法退出时可能的操作：

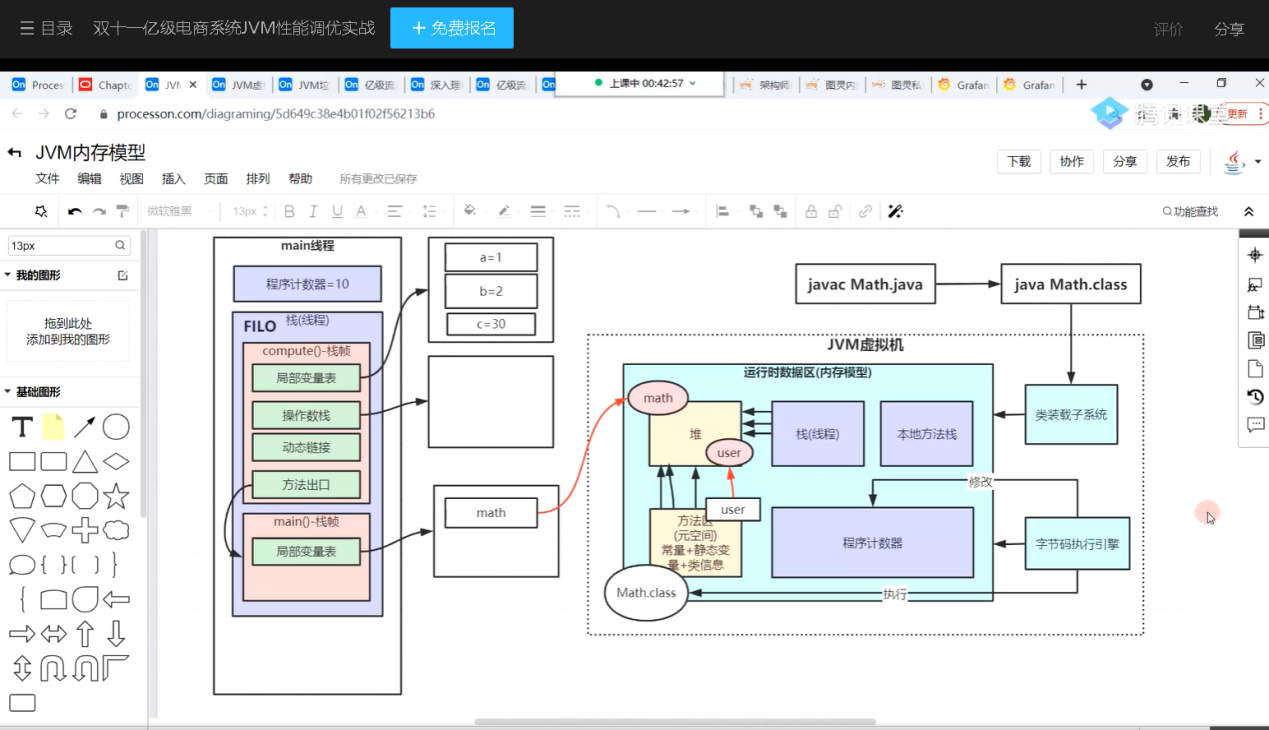
1、恢复上层方法的局部变量表和操作数栈

2、把返回值(如果有的话)压入调用者栈帧的操作数栈中

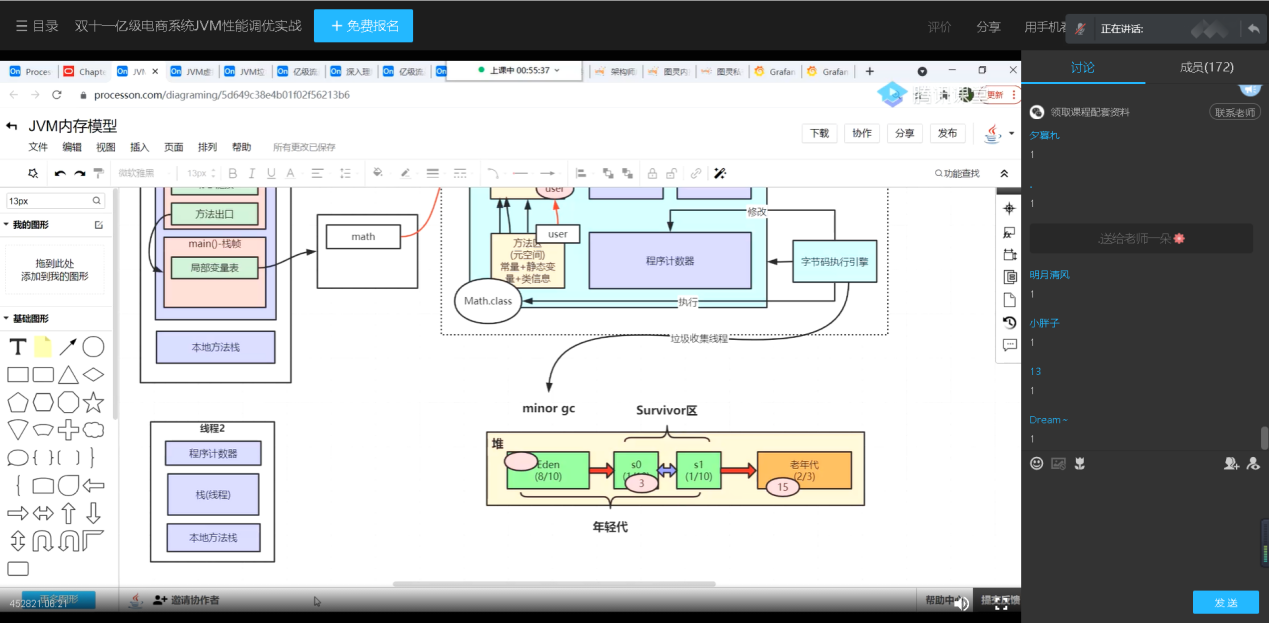
3、恢复PC计数器的值，以指向方法调用指令后面的一条指令

另一部分会在运行期间转化为直接引用，这部分称为动态连接

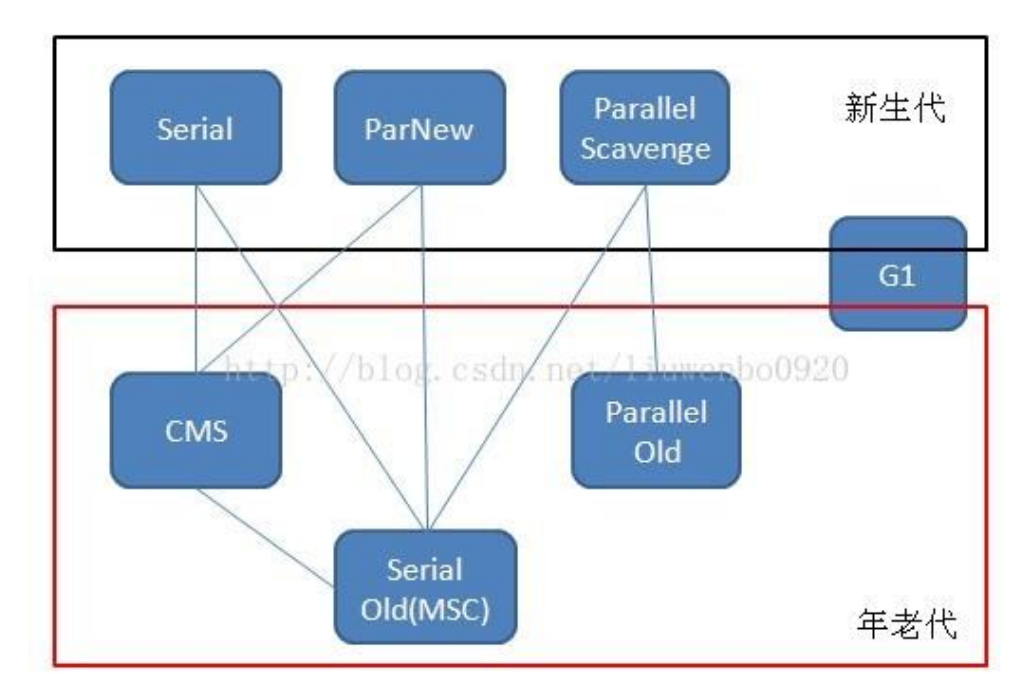




## 2、垃圾回收



## 3、垃圾收集器





Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；

ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU 环境下有着比Serial更好的表现；

Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；

Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；

Parallel Old收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器 的老年代版本；

CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集器，以获取最短回收 停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。

G1(Garbage First)收集器 ( 标记整理 + 复制算法来回收垃圾 )： Java堆并行收集器，G1收集器是 JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。

此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生 代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

## 4、浅拷贝和深拷贝

浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址；

深拷贝（deepCopy）是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存(复制源对象所有属性，并申请新得内存保存，源对象变化时，拷贝对象不会该表)。

浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来的对象也会相应的 改变。

深复制：在计算机中开辟一块新的内存地址用于存放复制的对象

参考：<https://blog.csdn.net/u014727260/article/details/55003402>

## 5、永久代会垃圾回收吗

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确 的永久代大小对避免Full GC是非常重要的原因。

(注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

## 6、元空间

在JDK1.8中元空间区取代了永久代，永久代原本主要存放Class和Meta的信息。而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。

为什么进行替换大致有以下几点原因：

* 1、字符串存在永久代中，容易出现性能问题和内存溢出。
* 2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
* 3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

## 程序计数器为什么是线程私有的？

1、字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制；

2、在多线程情况下，程序计数器记录当前线程执行的位置，从而当线程被切换回来时知道该线程上次被执行的位置。

总结：所以程序计数器私有是为了线程切换后能恢复到正确的执行位置。

