Java后台开发面经

written by GeYu

# Java基础

## 语言基础

### static和final的区别、用途

static:

修饰变量：静态变量随类加载时被完成时初始化,内存中只有一个,且jvm也只会为它分配一次内存，所有类共享变量

1. 修饰方法：在类加载时存在，不依赖任何实例，static方法必须实现，不能用abstract
2. 修饰代码块：在类加载完之后执行代码块中内容

在类加载的准备阶段就已经赋值 赋值为0。等到初始化之后才会赋值为所指定的值

Final:

1. 修饰变量：①编译期常量：类加载的过程完成初始化（基本类型）

②运行时常量：基本数据类型或引用数据类型

在类加载的准备阶段就已经赋值，赋值为所指定的值

1. 修饰方法：不能继承、不能被子类继承
2. 修饰类：不能继承
3. 修饰形参：final形参不可变

·加载顺序：

父类静态代码块->子类静态代码块->父类非静态代码块->父类构造方法->子类非静态代码块->子类构造方法

### java中为什么一个类中的一个方法调用另一个方法需要将被调用的方法声明static

static是静态修饰，加上之后可以变为[静态成员](https://www.baidu.com/s?wd=%E9%9D%99%E6%80%81%E6%88%90%E5%91%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)方法，这样编译的时候就可以随着类的加载而加载。然后其他的成员方法才可以使用。如果不声明为static ， 那么没有加载的方法不能被调用。

因为被调用的那个方法不能直接调用，你要new出它所在的类的对象才行。  
而static方法就不用new出对象，可以直接调用

### String知识点

1. String 类是被final进行修饰的，不能被继承
2. 在用+号链接字符串的时候会创建新的字符串。
3. java中,通过使用"+"符号来串联字符串的时候,实际上底层会转成通过 StringBuilder 实例的 append() 方法来实现。
4. String s = new String("Hello world");可能创建两个对象也可能创建一个对象。如果静态区中有“Hello world”字符串常量对象的话，则仅仅在堆中创建一个对象。如果静态区中没有“Hello world”对象，则堆上和静态区中都需要创建对象.
5. String重写了Object类的hashcode和toString方法
6. 当equals方法被重写时，通常有必要重写hashCode方法，在存储散列集合时(如Set类)，如果原对象.equals(新对象)，但没有对hashCode重写，即两个对象拥有不同的 hashCode，则在集合中将会存储两个值相同的对象，从而导致混淆.

**静态区：位于方法区的一块内存。存放类中以static声明的静态成员变量**

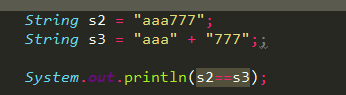
String s=”a”+”b”+”c”+”d”+”e”;改语句只创建了一个对象”abcde”；

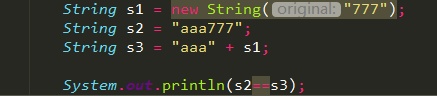
String s1=”aaa111”; String s2=“aaa111”; 对于值相同的常量，在常量池中只会创建一个， 所以在编译好的 class 文件中，我们只能找到一个“aaa111”的表示。另外，对于“ ”内容 为 null 的常量字符串，会创建一个长度为 0，内容为空的字符串放入常量池中。

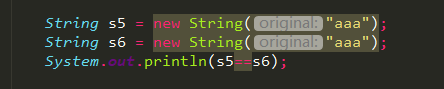
String s1=”abcde”; String s2=”abc”+”de”; s1==s2 是 true

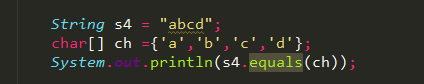
常量池中只会维护一个值相同的 String 对象

几个String例子如下:

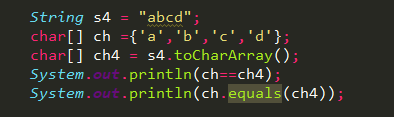
 

因为他们是不同的类

### **Equal和 hashCode**

**①若两个对象相等（equals），那么这两个对象一定有相同的哈希值（hashCode）；②若两个对象的哈希值相同，但这两个对象并不一定相等。**

重写了equals方法的对象必须同时重写hashCode()方法，如果2个对象通过equals调用后返回是true，那么这个2个对象的hashCode方法也必须返回同样的int型散列码

首先equals()和hashcode()这两个方法都是从object类中继承过来的。equals()是对两个对象的地址值进行的比较（即比较引用是否相同）。　hashCode()是一个本地方法，它的实现是根据本地机器相关的。

，String、Math等封装类都对Object中的equals()方法进行了重写。所以再String中equal比较的是内容是否相等

### 序列化

Java序列化是指把Java对象保存为二进制字节码的过程，Java反序列化是指把二进制码重新转换成Java对象的过程。

将Java对象序列化为二进制文件的 Java 序列化技术是 Java 系列技术中一个较为重要的技术点，在大部分情况下，开发人员只需要了解被序列化的类需要实现 Serializable 接口，使用 ObjectInputStream 和 ObjectOutputStream 进行对象的读写。

<http://www.importnew.com/24490.html>

<https://blog.csdn.net/xlgen157387/article/details/79840134>

序列化方式: 原生序列化

Json 序列化

Fastjson序列化

### java中的NIO,BIO,AIO分别是什么

（1）BIO：同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中

（2）NIO：同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接上有I/O请求时才启动一个线程进行处理。NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中。

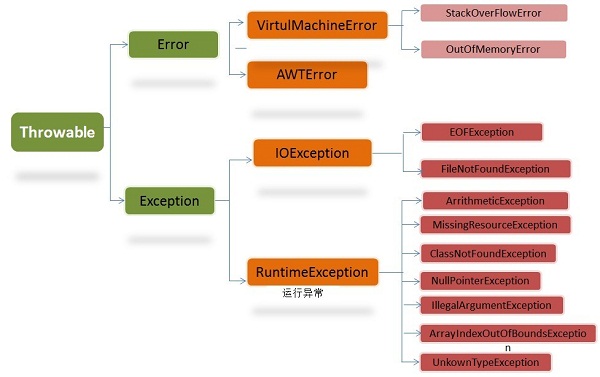
（3）AIO: 异步非阻塞, 服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作（nginx采用了异步非阻塞的方式来处理请求）。

### Java nio

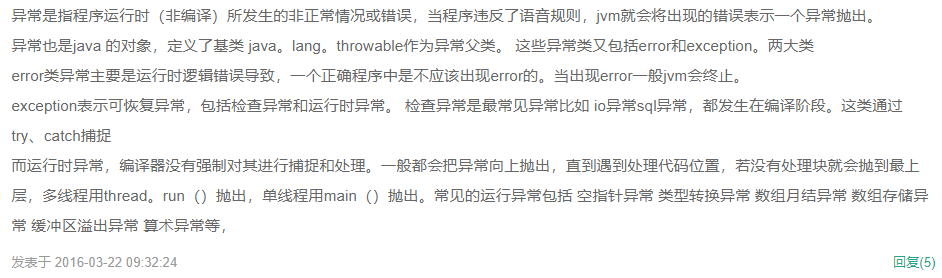
Java NIO的核心组件包括：Channel(通道)，Buffer(缓冲区)，Selector(选择器)，其中Channel和Buffer比较好理解

简单来说 NIO是面向通道和缓冲区的，意思就是：数据总是从通道中读到buffer缓冲区内，或者从buffer写入到通道中

### Error 和 Exception



Error 和 exception 的区别？Error 类一般指与虚拟机相关的问题，比如系统崩溃，虚拟机 错误，内存空间不足，对于这种错误导致的应用程序中断，仅靠程序本身无法恢复和预防， 遇到这样的错误，建议让程序终止。Exception 表示程序可以处理的异常，遇到这类异常， 应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。



### 字节和字符

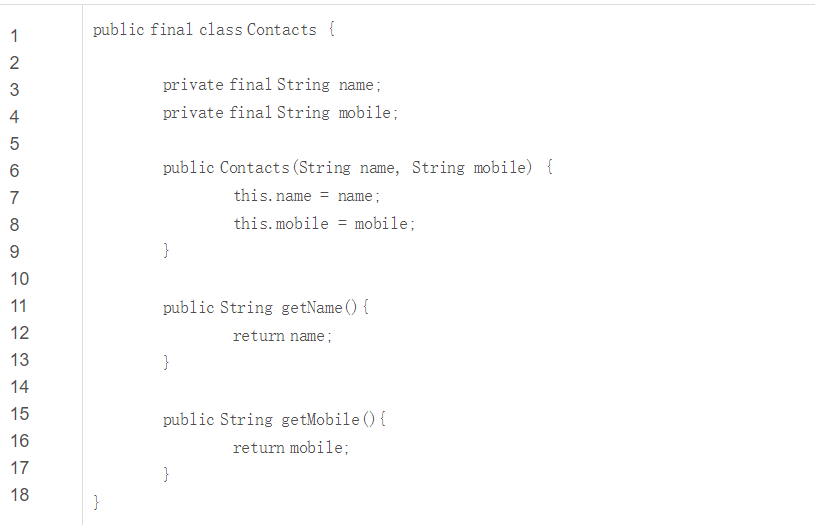
字节：byte即字节的意思，是java中的基本数据类型，用来申明字节型的变量，一个字节包含8个位，所以，byte类型的取值范围是-128~127。

字符：指计算机中使用的文字和符号，机器只只知道字节，而字符却是语义上的单位，它是有编码的，一个字符可能编码成1个 2个 甚至3个4个字节，跟字符集编码有关系。ASCII码，UTF-8编码、Unicode编码、UTF-16编码，UTF-32编码，不同的编码，字符所占的字节不同。

### 创建不可变对象

<https://blog.csdn.net/waeceo/article/details/54377218>

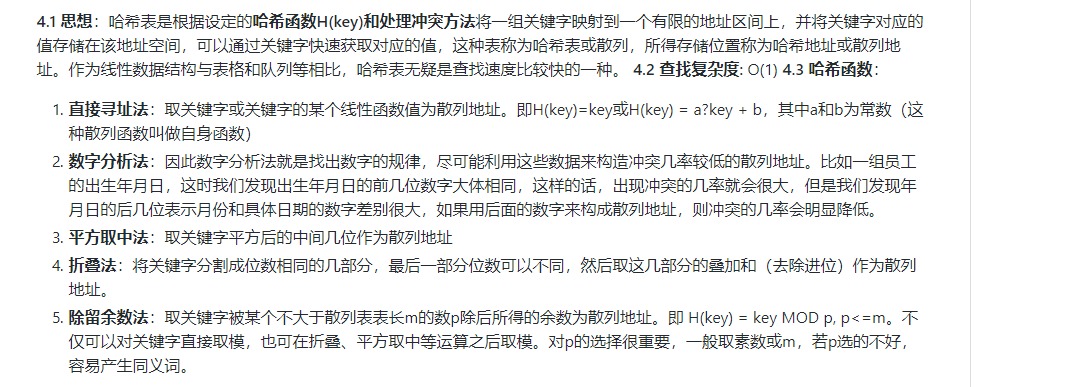




### 抽象类和接口



### hash方法 及冲突解决



HashMap hash方法：

static final int hash(*Object key*) {  
 int h;  
 return (*key* == null) ? 0 : (h = *key*.hashCode()) ^ (h >>> 16);  
}

hashcode方法为 object的hashCode，方法 为native方法 跟机器有关 Object中的hashCode() 返回对象的32位jvm内存地址。也就是说如果对象不重写该方法，则返回相应对象的32为JVM内存地址。

解决hash冲突的办法

开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）

再哈希法

链地址法(Java hashmap就是这么做的)

建立一个公共溢出区

### 静态内部类 和非静态内部类

### OOM 如何避免

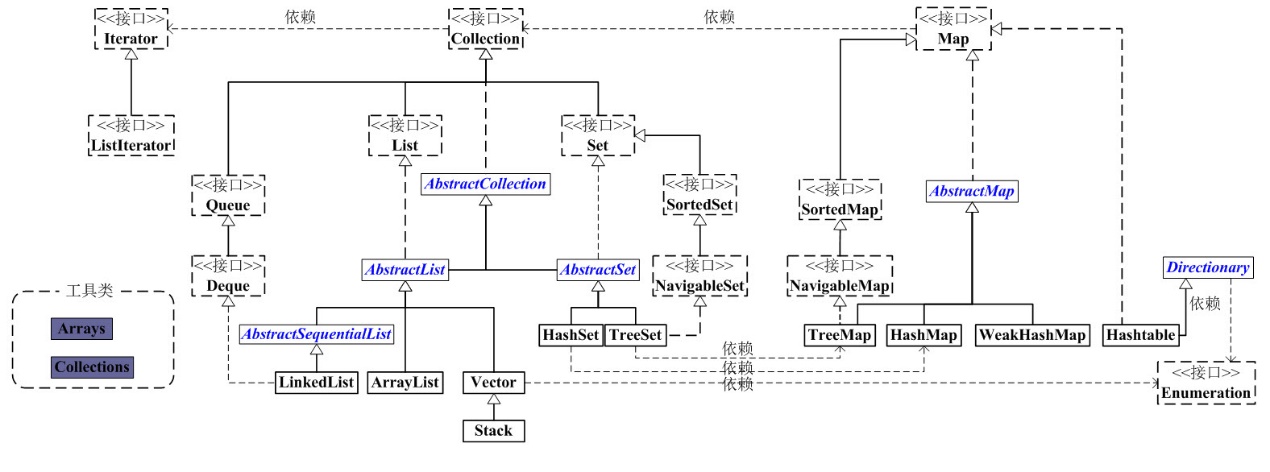
### 反射

### Java1.7与1.8新特性。

<https://www.cnblogs.com/aspirant/p/8617201.html>

## 集合类

### 总体架构



### List set

Set(集合)

Set是最简单的一种集合。集合中的对象不按特定的方式排序，并且没有重复对象。 Set接口主要实现了两个实现类：

HashSet： HashSet类按照哈希算法来存取集合中的对象，存取速度比较快

TreeSet ：TreeSet类实现了SortedSet接口，能够对集合中的对象进行排序。

Set 的用法：存放的是对象的引用，没有重复对象.

Set：1.不允许重复对象

　　 2. 无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或者 Comparable 维护了一个排序顺序。

3. 只允许一个 null 元素

4.Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

List(列表)

List的特征是其元素以线性方式存储，集合中可以存放重复对象。

对于List的随机访问来说，就是只随机来检索位于特定位置的元素。 List 的 get(int index) 方法放回集合中由参数index指定的索引位置的对象，下标从“0” 开始。最基本的两种检索集合中的所有对象的方法：

List：1.可以允许重复的对象。

　　 2.可以插入多个null元素。

3.是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。

4.常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

### ArrayList vs LinkedList vs vector

A、简单的说是ArrayList 和Vector 的底层数据是数组的结构素的位置都是有顺序即连续的，相当于一种动态的数组，我们以后可以按位置索引号取出某个元素，并且其中的数据是允许重复的

B、LinkedList使用双向链表实现存储，存储元素的位置是不连续的。 按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入,修改，删除 速度较快.

C、Vector是同步处理,性能较低；ArrayList是使用异步处理，性能高。.Vector是线程安全的，ArrayList是非线程安全。LinkedList是非线程安全。一般情况都用ArrayList，ArrayList占据内存小

D、ArrayList顺序存取速度快 LinkedList随机存取速度快 arraylist随机查找快，插入删除慢， linkedlist插入 删除快 查找慢

E、ArrayList 初始化大小是 10，扩容机制是原始大小+原始大小/2 + 1，linkedlist不存在扩容，基于双向链表 只需要一直添加就行

### HashMap VS HashTable vs HashSet

1. HashMap允许null值（key和value都可以），HashTable不允许null值（key和value都不可）. HashMap的key不能时可变对象
2. HashMap的方法不是同步的，非线程安全，HashTable是线程安全的，方法添加synchronized关键字进行修饰，执行速度慢
3. HashMap基于AbstactMap类，HashTable基于Dictionary类
4. 哈希值的使用不同。HashTable直接使用对象的hashCode, HashMap重新计算hash值
5. HashMap种hash数组的默认大小是16，而且一定是2的指数，为的是hash过后能够分布的更加均匀。增加方式是数组大小扩大为2倍， HashTable中hash数组默认大小是11，增加的方式是old\*2+1.

现在如果为了保证线程安全的话 使用ConcurrentHashMap

f、HashSet依赖的数据结构是哈希表，因为实现的是Set接口，所以不允许有重复的值，插入到HashSet中的对象不保证与插入的顺序保持一致。对象的插入是根据它的hashcode。HashSet中允许有NULL值

HashMap采用数组+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换位红黑树，这样大大减少了查找时间

**部分源码分析**：

初始化容量 左移4位 16

static final int ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4;

最大容量：

static final int ***MAXIMUM\_CAPACITY*** = 1 << 30;

加载因子：在容量的3/4的时候扩容

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

put操作的时候 达到容量的3/4的时候扩容

底层：必须记住

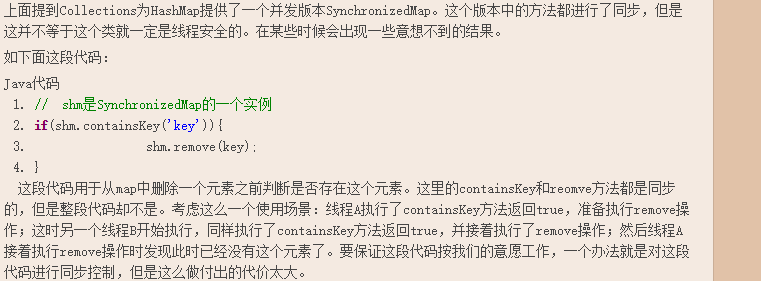
**http://zhangshixi.iteye.com/blog/672697**

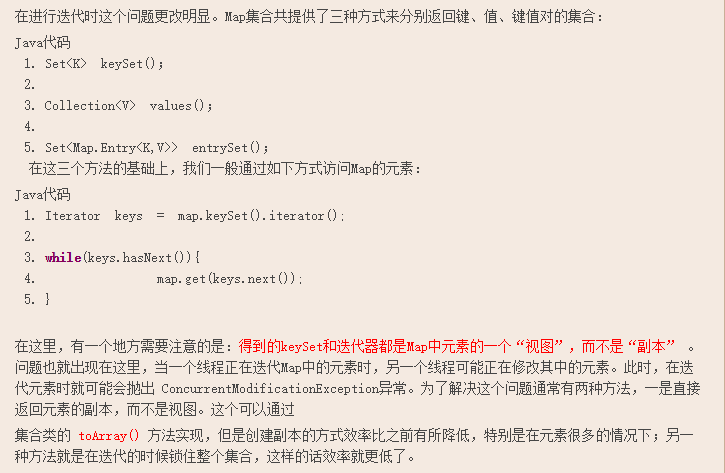
[**http://www.admin10000.com/document/3322.html**](http://www.admin10000.com/document/3322.html)

[**http://www.cnblogs.com/skywang12345/p/3310887.html**](http://www.cnblogs.com/skywang12345/p/3310887.html)

### SynchronizedMap(了解)

HashMap不是线程安全的。在多线程环境中，需要手动实现同步机制。因此，在Collections类中提供了一个方法返回一个同步版本的HashMap用于多线程的环境：SynchronizedMap.





更好的选择 concurrentHashMap

### ConcurrentHashMap(重要，重点)

1. ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作，第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部，因此，这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长，但是带来的好处是写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上），所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。
2. HashMap是非线程安全的，ConcurrentHashMap是线程安全的
3. ConcurrentHashMap将整个Hash桶进行了分段segment，也就是将这个大的数组分成了几个小的片段segment，而且每个小的片段segment上面都有锁存在，那么在插入元素的时候就需要先找到应该插入到哪一个片段segment，然后再在这个片段上面进行插入，而且这里还需要获取segment锁
4. ConcurrentHashMap让锁的粒度更精细一些，并发性能更好
5. 初始化除了initalCapacity,loadfactor参数，还有一个重要的参数concurrency level，决定了segment数组的长度，默认为16(需要2的n次方，与hash算法有关)，每次get/put操作都会通过hash算法定位到一个segment，再hash到某个具体的entry
6. HashEntry 中，除了 value，其余 3 个 key、hash、next 都是 final 的。这意味着不能从中间或者尾部添加或删除节点，因为这需要改变 next 的引用值，所有节点修改只能从头部开始，对于 put 操作，可以一律添加到哈希链头部，对于 remove 操作，可能需要从中间删一个节点，这就需要将要删除节点的前面所有节点整个复制一遍，最后一个节点指向要删除节点的下一个节点。
7. Get操作不需要加锁，get方法将共享变量定义为volatile，能够再线程之间保证可见性，能够被多个线程同时读，并且保证不会读到过期**的**值。但是只能被单线程写。 Put操作必须加锁。
8. Resize 时，只对 Segment 扩容，不对整个 Concurrenthashmap 扩容，先判断是否需要扩容，再 put，传统 hashmap 是先 put 再判断。

### String、StringBuffer、StringBuilder

1. 都是final类，都不允许被继承
2. String长度是不可变的，StringBuffer、StringBuilder可变
3. StringBuffer线程安全，StringBuilder非线程安全，StringBuffer在StringBuilder方法之上添加了synchronized修饰。所以StringBuilder性能更好

### TreeMap(了解)

TreeMap 是一个有序的key-value集合，它是通过红黑树实现的。

继承于AbstactMap, 实现SortMap接口。所以它是一个Map，即一个Key-value集合

默认按键的升序排序，Iterator遍历是排序的，键值都不能为null，TreeMap效率较低。

### TreeSet(了解)

TreeSet也是一个有序的集合，实现Set接口，不能有重复元素

## 线程类

### Java实现多线程的方式及三种方式的区别

1. 继承Thread类，重写run函数
2. 实现Runnable接口
3. 实现Callable接口

区别：

实现Runnable接口可以避免Java单继承特性而带来的局限；

继承Thread类和实现Runnable方法启动线程都是使用start方法，然后JVM虚拟机将此线程放到就绪队列中，如果有 处理机可用，则执行run方法.

实现Callable接口要实现call方法，并且线程执行完毕后会有返回值.

### 线程的状态

1. 新建：创建后尚未启动的线程处于这种状态
2. 运行：包括了 OS 中 Running 和 Ready 状态，也就是处于次状态的线程可能正在运行，也可能正在等待 cpu 为他分配执行时间
3. 无限期等待: 处于这种状态的线程不会被分配 cpu 执行时间，要等待其他线程显示唤醒。以下方法会让线程进入无限期等待:1.没有设置 timeout 的 object.wait()方法 2.没有设置 timeout 参数的 Thread.join()方法 3.LockSupport.park();
4. 有限期等待：处于这种状态的线程也不会被分配 cpu 执行时间，不过无需等待被 其他线程显示唤醒，而是在一定时间后，他们会由 os 自动唤醒 1.设置了 timeout 的 object.wait()2.设置了timeout 参数的 Thread.join() 3.LockSupport.parkNanos() 4.LockSupport.parkUnit()
5. 阻塞：线程被阻塞了与“等待状态”的区别是：阻塞状态在等待获取一个排它锁，这个事件将在另外一个线程放弃这个锁的时候发生。等待状态在等待一段时间或者唤醒动作。
6. 结束：已终止线程的线程状态，线程已经结束执行。

### 线程池相关

1. 什么是线程池？

事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建， 使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

1. 设计线程池应该有哪些方法

(1)、线程管理器(ThreadPool)：用于创建并管理线程池，包括创建线程，销毁线程池，添加新任务

(2)、工作线程(PoolWorker)：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

(3)、任务接口(Task)：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行 完后的收尾工作，任务的执行状态等

(4)、任务队列(TaskQueue)：用于存放没有处理的任务。提供一种缓冲机制

ThreadPoolExecutor

<https://www.cnblogs.com/aspirant/p/6920418.html>

<https://www.jianshu.com/p/210eab345423>

ThreadPoolExector中的参数 BlockingQueue的几种类型：

1. ArrayBlockQueue:基于数组结构的有界队列，此队列按FIFO(first in first out)原则对任务进行排序。如果队列已满，新的任务将会被采取拒绝策略对待。
2. LinkedBlockingQueue: 基于链表的无界队列，按FIFO原则排序。因为是无界的，所以不存在满的情况，此时拒绝策略无效
3. PriorityBlockingQueue：具有优先级的队列的有界队列，可以自定义优先级，默认为自然排序

几种线程池：

https://www.cnblogs.com/aaron911/p/6213808.html

1. FixedThreadPool ：

由Executors的newFixedThreadPool方法创建。它是一种线程数量固定的线程池，当线程处于空闲状态时，他们并不会被回收，除非线程池被关闭。当所有的线程都处于活动状态时，新的任务都会处于等待状态，直到有线程空闲出来。FixedThreadPool只有核心线程，且该核心线程都不会被回收，这意味着它可以更快地响应外界的请求

1. CachedThreadPool

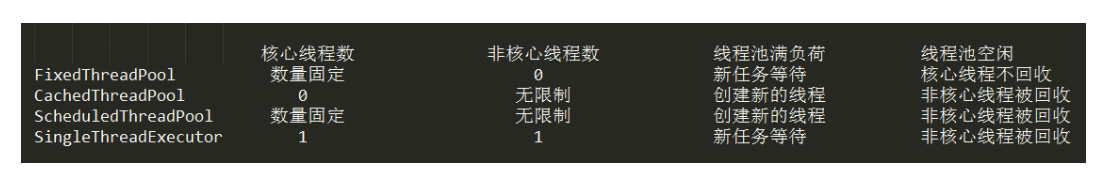
由Executors的newCachedThreadPool方法创建，不存在核心线程，只存在数量不定的非核心线程，而且其数量最大值为Integer.MAX\_VALUE。当线程池中的线程都处于活动时(全满),线程池会创建新的线程来处理新的任务，否则就会利用新的线程来处理新的任务，线程池中的空闲线程都有超时机制，默认超时时长为60s，超过60s的空闲线程就会被回收。从CachedThreadPool的特性看，这类线程比较适合执行大量耗时较小的任务。当整个线程池都处于闲置状态时，线程池中的线程都会因为超时而被停止回收，几乎是不占任何系统资源

1. ScheduledThreadPool

通过Executors的newScheduledThreadPool方式创建，核心线程数量是固定的，而非核心线程是没有限制的，并且当非核心线程闲置时它会被立即回收，ScheduledThreadPool这类线程池主要用于执行定时任务和具有固定时期的重复任务

4、SingleThreadExecutor

通过Executors的newSingleThreadExecutor方法来创建。这类线程池内部只有一个核心线程，它确保所有的任务都在同一个线程中按顺序执行。SingleThreadExecutor的意义在于统一所有外界任务一个线程中，这使得这些任务之间不需要处理线程同步的问题

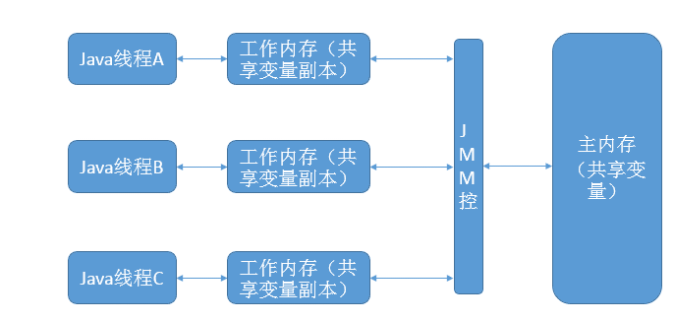


### volatile关键字（重要）

a、保证内存可见性（在每次访问变量时都会进行一次刷新，因此每次访问都是主内存中新的版本。）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

b、防止指令重排

c、volatile并不保证原子性



使用volatile关键字会强制将修改的值立即写入主存，也就是将某个线程的工作内存修改的值写入主内存

volatile使用建议：

（1）、对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。该变量没有包含在具有其他变量的不变式中.

（2）、在两个或者更多的线程需要访问的成员变量上使用volatile。当要访问的变量已在synchronized代码块中，或者为常量时，没必要使用volatile。

（3）、由于使用volatile屏蔽掉了JVM中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

（4）、在访问变量时不需要加锁。

volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的

指令重排序（happens-before）：

在虚拟机层面，为了尽可能减少内存操作速度远慢于CPU运行速度所带来的CPU空置的影响，虚拟机会按照自己的一些规则(这规则后面再叙述)将程序编写顺序打乱——即写在后面的代码在时间顺序上可能会先执行，而写在前面的代码会后执行——以尽可能充分地利用CPU。

在硬件层面，CPU会将接收到的一批指令按照其规则重排序，同样是基于CPU速度比缓存速度快的原因，和上一点的目的类似，只是硬件处理的话，每次只能在接收到的有限指令范围内重排序，而虚拟机可以在更大层面、更多指令范围内重排序。

#### volatile如何防止指令重排序

volatile关键字通过提供“内存屏障”的方式来防止指令被重排序，为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

大多数的处理器都支持内存屏障的指令。

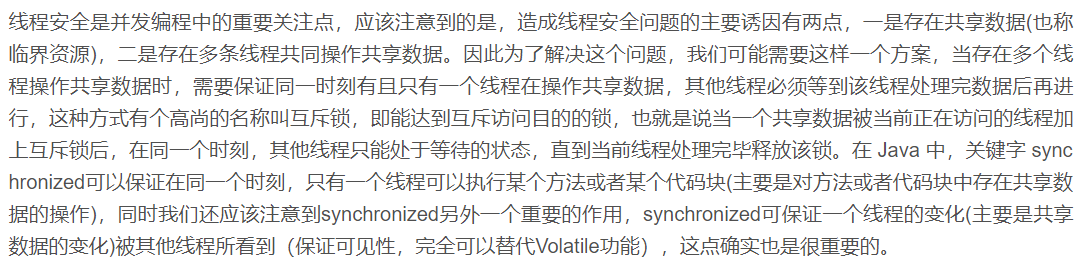
在每个volatile写操作的前面插入一个StoreStore屏障。

在每个volatile写操作的后面插入一个StoreLoad屏障。

在每个volatile读操作的后面插入一个LoadLoad屏障。

在每个volatile读操作的后面插入一个LoadStore屏障。

### synchronized关键字



(1)、synchronized底层:

(2)、synchronized在代码块和方法上的区别:

a、synchronized用在代码块锁的是调用该方法的对象（this），也可以选择锁住任何一个对象。可以减小锁的粒度，从而提高并发性能。

b、synchronized用在方法上锁的是调用该方法的对象。

c、无论用在代码块上还是用在方法上，都是获取对象的锁；每一个对象只有一个锁与之相关联；实现同步需要很大的系统 开销作为代价，甚至可能造成死锁，所以尽量避免无谓的同步控制。

### Synchronized底层实现

<https://blog.csdn.net/javazejian/article/details/72828483>

1. 同步代码块

使用monitorenter和moniterexit指令实现，monitorenter指令插入到同步代码块的开始位置，moniterexit指令插入到同步代码块的结束位置，jvm需要保证每一个monitorenter都有一个moniterexit与之对应。这两个字节码都需要一个reference类型的参数来指明要锁定和解锁的对象。任何对象都有一个monitor与之相关联，当且一个monitor被持有之后，他将处于锁定状态。线程执行到monitor指令前，将会尝试获取对象所对应的monitor所有权，即尝试获取对象的锁。

1. 同步方法

依靠的是方法修饰符上的ACC\_SYNCHRONIZED实现，Synchronized方法则会被翻译为普通的方法调用和返回指令，比如invokevirtual指令，在jvm字节码层面并没有任何特别的指令来实现synchronized修饰的方法，而是在class文件的方法表重将该方法的accless\_flags字段重的synchronized标志位置为1，表示该方法为synchronized方法，且使用调用该方法的对象or该方法所属的class在jvm内部对象表示作为锁对象。

### synchronized vs volatile and synchronized vs lock

Volatile 和 synchronized的区别：

1. volatile是一种稍弱的同步机制，不会进行加锁操作，不会造成线程阻塞，轻量级的同步机制。
2. volatile变量作用类似于同步变量读写操作。写入volatile相当于退出代码块，读取volatile变量相当于进入同步代码块。
3. volatile不如synchronized安全
4. volatile只能保证内存可见性，不能保证原子性

synchronized 和 lock的区别

1. synchronized(隐式锁)：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。
2. lock（显示锁）：需要显示指定起始位置和终止位置。一般使用ReentrantLock类做为锁，多个线程中必须要使用一个ReentrantLock类做为对象才能保证锁的生效。且在加锁和解锁处需要通过lock()和unlock()显示指出。所以 一般会在finally块中写unlock()以防死锁。
3. synchronized是托管给JVM执行的，而lock是java写的控制锁的代码。性能彼此差不多
4. synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。Lock用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作（Compare and Swap）。
5. lock是接口，synchronized是关键字,lock可以提高多个线程进行读写操作的效率
6. 通过Lock可以知道有没有成功获取锁，而synchronized无法办到。

### sleep() vs wait() 都可以设置时间

Sleep是Thread类的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会 （CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态）。

wait是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池 （wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

### wait方法()

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好

### ReenTrantLock

1. 可重入，单线程可以重复进入，但要重复退出，synchroized的也是可重入锁
2. 可中断，可响应中断
3. 可限时：超时不能获得锁，就返回false，不会永久等待构成死锁
4. 非公平锁：默认是非公平锁，可以设置为公平锁。
5. Condition 的 await()与 single()和 object 的 wait()和 notify()类似。

### ReentrantReadWriteLock(juc)

读写锁，读写锁允许同一时刻被多个读线程访问，但是在写线程访问时，所有的读线程和其他的写线程都会被阻塞。

### CopyOnWriteArrayList(juc)

COW通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy,复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的而引用指向新的容器 对CopyOnWrite容器进行并发的读的时候，不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，延时更新的策略是通过在写的时候针对的是不同的数据容器来实现的，放弃数据实时性达到数据的最终一致性。

### BlockingQueue(juc)

### CAS（了解）

Compare and swap 比较并交换。

Java.util.concurrent 包完全建立在 cas 上，没有 cas 就不会有此包。

CAS操作包含三个操作数-----内存位置（V），预期原值（A）和新值（B）。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在CAS指令之前返回该位置的值。CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。

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。而整个J.U.C都是建立在CAS之上的，因此对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。

### ThreadLocal

　ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。ThreadLocal变量类似于全局变量，能降低代码的可重用性。

ThreadLocal与线程同步无关。

所以ThreadLocal与线程同步机制不同，线程同步机制是多个线程共享同一个变量，而ThreadLocal是为每一个线程创建一个单独的变量副本，故而每个线程都可以独立地改变自己所拥有的变量副本，而不会影响其他线程所对应的副本。可以说ThreadLocal为多线程环境下变量问题提供了另外一种解决思路。

## 锁

在java设计中，每一个对象自打娘胎里出来就带了一把看不见的锁,即monitor锁。Monitor是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局可用列表，每一个被锁住对象都会和一个monitor关联。Monitor中有一个owner字段存放拥有该对象的线程的唯一标识，表示该锁被这个线程占有。Owner:初始时为null,表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时，又设为null，Entry Q:关联一个系统互斥锁，阻塞all试图锁住monitor entry失败的线程。Next:用来实现重入锁的计数

锁主要有4种状态：无锁状态、偏向状态、轻量级状态、重量级状态，他们会随着竞争的激烈而逐渐升级，锁可以升级但不可以降级

自旋锁、自适应自旋锁、锁消除。

轻量级锁：传统的锁是重量级锁，他是用系统的互斥量来实现。轻量级锁的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。

偏向锁：目的是消除数据在无竞争情况下的同步原语。如果说轻量级锁是在无竞争的情况下使用CAS操作去消除同步使用的互斥量，那么偏向锁就是在无竞争的情况下把整个同步消除掉。

Synchronized用的锁是存在java对象头里，对象头包含标记字段和类型指针。类型指针是对象指向它类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。标记字段用于存储对象自身运行时数据，他是实现轻量级锁和偏向锁的关键。

Jvm可以通过对象的元数据信息确定对象的大小，但是无法从数组的元数据来确定数组的大小

<https://blog.csdn.net/javazejian/article/details/72828483>

## 同步/公平锁

同步：多个线程并发访问共享数据时，保证共享数据在同一个时刻，只被一个（or 一些，使用信号量）线程使用，而互斥是实现同步的一种手段，临界区，互斥量，信号量都是主要的互斥实现方式。互斥是因，同步是果；互斥是方法，同步是目的。

公平锁：（先申请先得到）多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获取，而非公平锁则不保证这一点。Synchronized不是公平锁，reetrantlock 默认下也是非公平的，但是在构造函数中，可以设置为公平的。

互斥同步对性能最大的影响是阻塞的实现，挂起线程和恢复线程的需要转入内核态中完成。若物理机器上有一个以上的处理器，能让2个or2个以上的线程同时并行执行，我们就 可以让后面请求锁的那个线程“稍等一下”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就释放锁，为了让线程等待，我们只需要让他执行一个忙循环（自旋），这项技术就是所谓的自旋锁

## 线程安全的实现方法

深入理解JAVA虚拟机 p390

## 设计模式

### 设计模式分类

**创建型模式**：这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象

1. 工厂模式
2. 抽象工厂模式
3. 单例模式
4. 建造者模式
5. 原型模式

**结构型模式**：这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

1. 适配器模式
2. 桥接模式
3. 过滤器模式
4. 组合模式
5. 装饰器模式
6. 外观模式
7. 享元模式
8. 代理模式

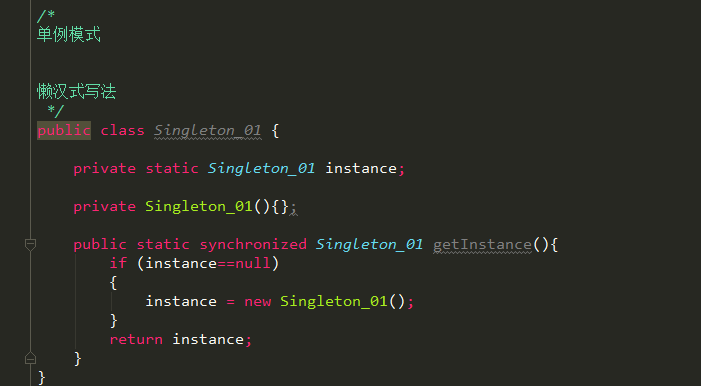
**行为型模式：**这些设计模式特别关注对象之间的通信

1. 责任链模式
2. 命令模式
3. 解释器模式
4. 迭代器模式
5. 中介者模式
6. 备忘录模式
7. 状态模式
8. 观察者模式
9. 空对象模式
10. 策略模式
11. 模板模式
12. 访问者模式

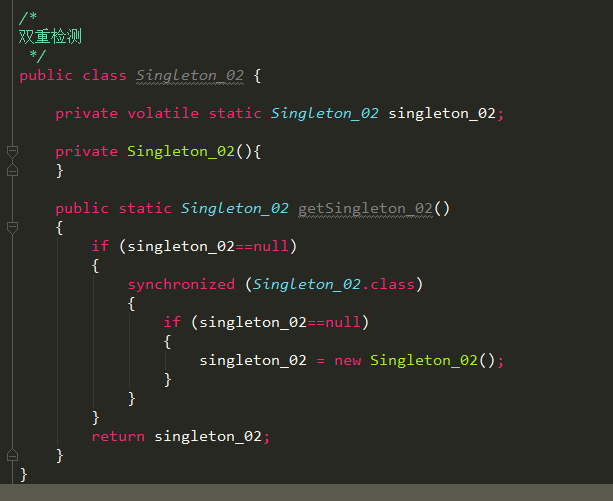
### A、单例模式：

典型例子有一个公司只能有一个CEO。它主要是为了保证一个类仅有一个实例，这个类中自己提供一个返回实例的方法，方法中先判断系统是否已经有这个单例，如果有则返回，如果没有则创建。如果创建多个实例会消耗过多的资源或者某种类型的对象只应该有且只有一个时，应该考虑使用单例模式。

1、懒汉式写法

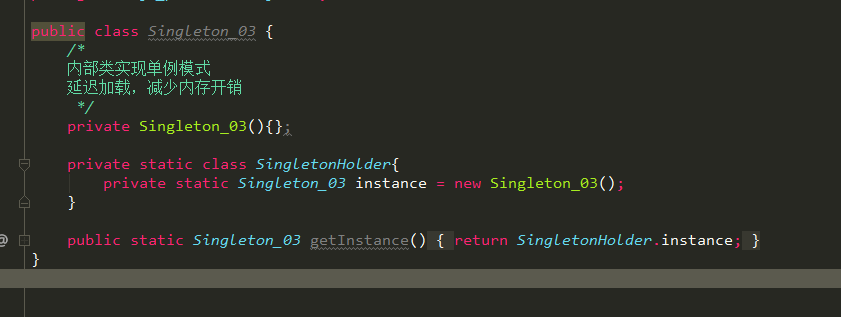


2、双重检测



加入volatile是为了避免重排序 如果没有的话 再判断singleton\_02==null时 可能还没初始化完成

3、静态内部类实现（推荐使用）

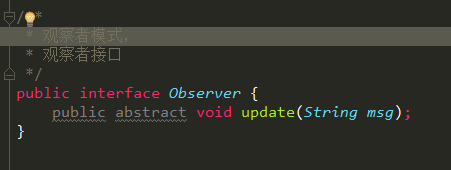


### B、观察者模式：

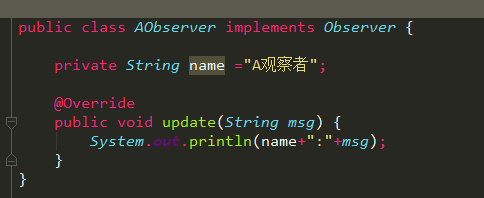
当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

观察者模式，我理解的就是观察者订阅被观察者的状态，当被观察者状态改变的时候会通知所有订阅的观察者的过程

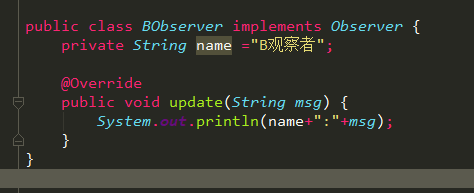
观察者接口:



第一个观察者：



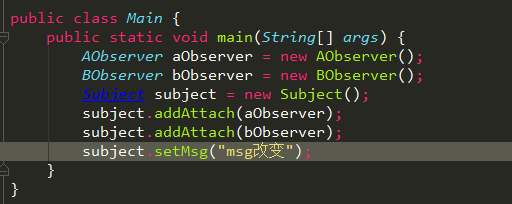
第二个观察者：



被观察者：



测试：



结果：



除此之外，java 可以用 Observer 接口和 Obserable 类实现观察者模式。1）创建被观察者类 observable 类 2）创建观察者类 Observer 接口 3）对于被观察者，添加他的观察者 void add Observer(Observer o)：该方法把观察者对象添加到观察者对象列表中，当被观察事件发生时， 调用 1.setchange()用来设置一个内部标志，注明数据发生了变换；2.notifyObservers()：调用 观察者对象列表中 all Observer 的 update 方法，通知他们数据发生变换，只有在 setchange 被调用以后，notifyObservers 才会调用 update。4）对于观察者类，实现 Observer 接口的唯一方法update

### C、代理模式

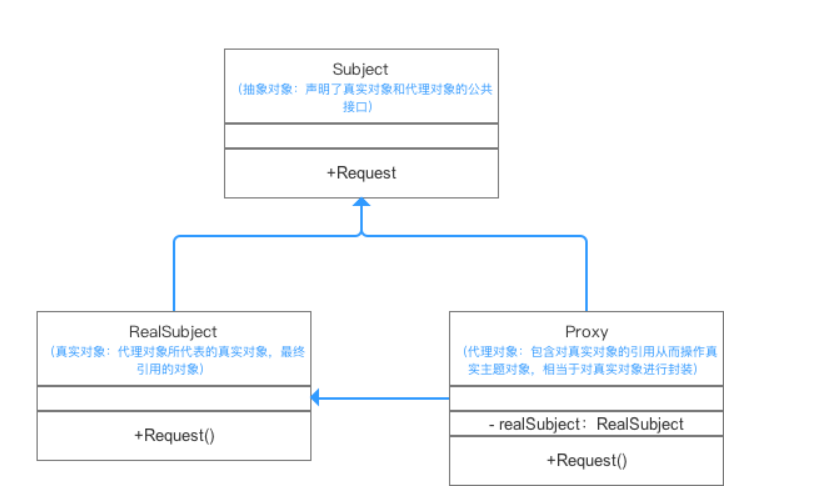
代理类和委托类有相同的接口，一个代理类的对象与一个委托类的对象关联。代理类的独享本身并不真正实现服务，而是通过调用委托类的对象的相关方法来提供特定的服务。JDK动态代理（掌握程度是自己会写，知道每个函数的每个参数的作用）反射 1）proxy 类： 类的静态方法用来生成动态代理的实例 2）innovationhandler 接口有一个 invoke 方法，用来 集中处理在动态代理类对象的方法调用，通常在该方法中实现对委托类的代理访问，每次生 成动态代理对象时都要指定一个对应的调用处理器。CGlib 动态代理：jdk 代理机制只能代理 实现了接口的类，而没有实现接口的类不能用 jdk 动态代理。CGlib 是针对类来实现代理，他的原理是对指定的目标类生成一个子类，并且覆盖其中方法实现增强，因为采用的是继承，所以不能对 final 修饰的类进行代理，methodinterceptor

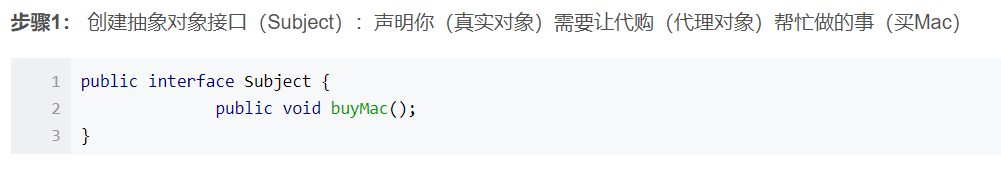
通过使用动态代理，我们可以通过在运行时，动态生成一个持有RealObject、并实现代理接口的Proxy，同时注入我们相同的扩展逻辑。哪怕你要代理的RealObject是不同的对象，甚至代理不同的方法，都可以动过动态代理，来扩展功能。

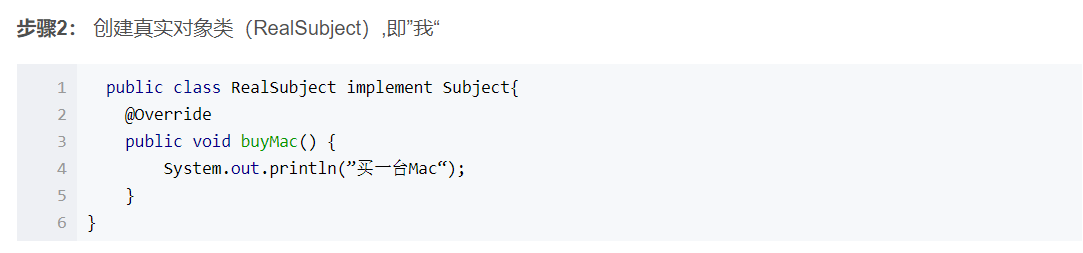
和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。

和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

#### 1、静态代理

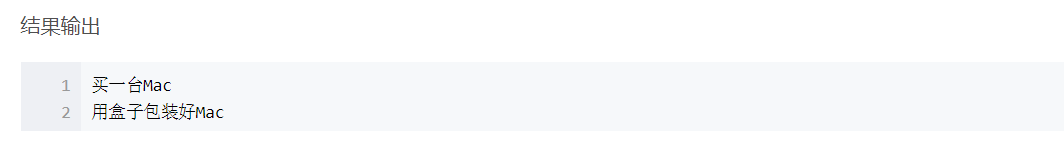












#### 2、动态代理

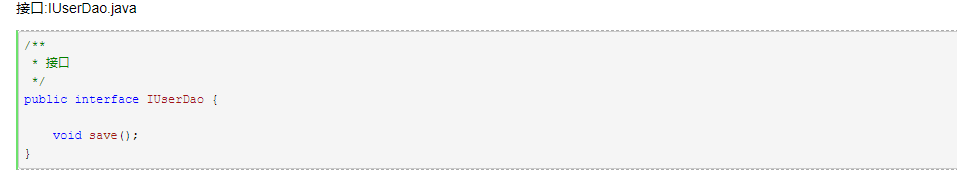
动态代理有以下特点:

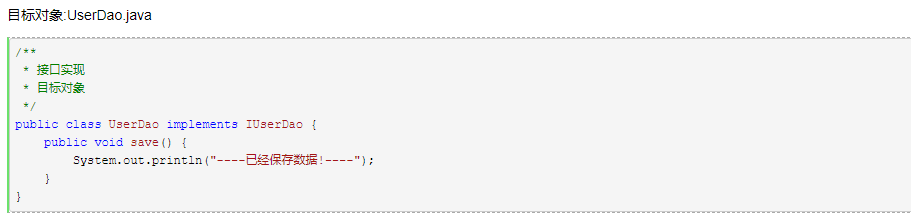
1.代理对象,不需要实现接口

2.代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)

3.动态代理也叫做:JDK代理,接口代理

代理类所在包:java.lang.reflect.Proxy









代理对象不需要实现接口,但是目标对象一定要实现接口,否则不能用动态代理

#### 3、Calib代理

上面的静态代理和动态代理模式都是要求目标对象是实现一个接口的目标对象,但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候就可以使用以目标对象子类的方式类实现代理,这种方法就叫做:Cglib代理

### D、装饰器模式

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

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。

装饰器模式和代理模式的区别：

装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。换句话 说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模 式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

我们可以用另外一句话来总结这些差别：使用代理模式，代理和真实对象之间的的关系通常在编译时就已经确定了，而装饰者能够在运行时递归地被构造。

# Java web && web框架

## Servlet

Servlet是用于java编写的服务器端程序，其使用java servlet API，当客户机发送请求到服务器时，服务器可以将请求信息发送给servlet,并让servlet建立起服务器返回给客户机的响应，当启动web服务器or客户机第一次请求服务时，可以自动装入servlet,装入后，servlet继续运行知道其他客户机发送请求。

servlet 生命周期？Servlet 生命周期分为 3 个阶段

1）初始化阶段：调用 init()方法

2）响应客户请求：调用 service()

3）终止:调用 destory().

A、初始化阶段：在下列时刻 servlet 容器装载 servlet

1.servlet 容器启动时，自动装载某些 servlet

2.在servlet 容器启动后，客户首次向servlet发送请求

3.servlet类文件被更新以后，重新装载servlet。Servlet被装载后，servlet 容器创建一个 servlet 对象并调用 servlet 的 init 方法。在 servlet 生命周期内，init()方法只被调用过一次。Servlet 工作原理：客户端发起一个请求，servlet 调用 service()方法时请求进行响应，service 对请求的方式进行了匹配，选择调用 dopost 或者 doget 等这些方法，然后进 入对应方法中调用逻辑层上的方法，实现对客户的响应。

B、响应客户请求：对于用户到达 servlet 的请求，servlet 容器会创建特定于该请求的 servletRequest 和 servletresponse 对象，然后调用 servlet 的 service 方法，service 方法从 servletrequest 对象中获得客户请求的信息， 处理该请求，并通过 servletresponse 对象向客户返回响应消息

C、终止：当 web应用终止或者servlet容器终止或 servlet 容器重新装载 servlet新实例时，servlet 容器会调用servlet对象的第 destory 方法，在 destory方法中可释放servlet占用的资源。

### 过滤器 拦截器 监听器

## cookie和session的区别

cookie和session http都是无状态的协议。

session在服务器端保持http状态信息，cookie在客户端保持http状态信息，会话跟踪。

session的运行依赖session id，session id是存在cookie中的，也就是说，如果浏览器禁用了cookie，同时session也会失效（但可以通过其他方式实现）。

session可以放在文件，数据库，或内存中都可以。

用户验证这种场合一般会用session。

session会在一定时间内保存在服务器上，当访问增多，比较占用服务器性能的时候考虑使用cookie。

## Spring

### Spring IOC,AOP

1. Spring IOC

IOC叫控制反转，DI叫依赖注入，是对IOC更简单的诠释，控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的“控制反转”就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。DI是对IOC更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

举个例子：一个类A需要用到接口B中的方法，那么就需要为类A和接口B建立关联或依赖关系，原始的方法是在类A 中创建一个接口B的实现类C的实例，但这种方法需要开发人员自行维护二者的依赖关系，也就是说当依赖关系发生变 动的时候需要修改代码并重新构建整个系统。如果通过一个容器来管理这些对象以及对象的依赖关系，则只需要在类A 中定义好用于关联接口B的方法（构造器或setter方法），将类A和接口B的实现类C放入容器中，通过对容器的配置来实现二者的关联。

1. Spring IOC实现原理
2. 通过反射创建实例
3. 获取需要注入的接口实现类并将其赋值给该接口
4. Spring AOP

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的模块，并将其命名为“Aspect”,即方面。

所谓“方面“，简单地说就是将那些与业务无关，却为业务模块所共同调用地逻辑或责任封装起来，便于减少系统地重复代码，降低模块间地耦合度，并有利于未来地可维护性和可操作性

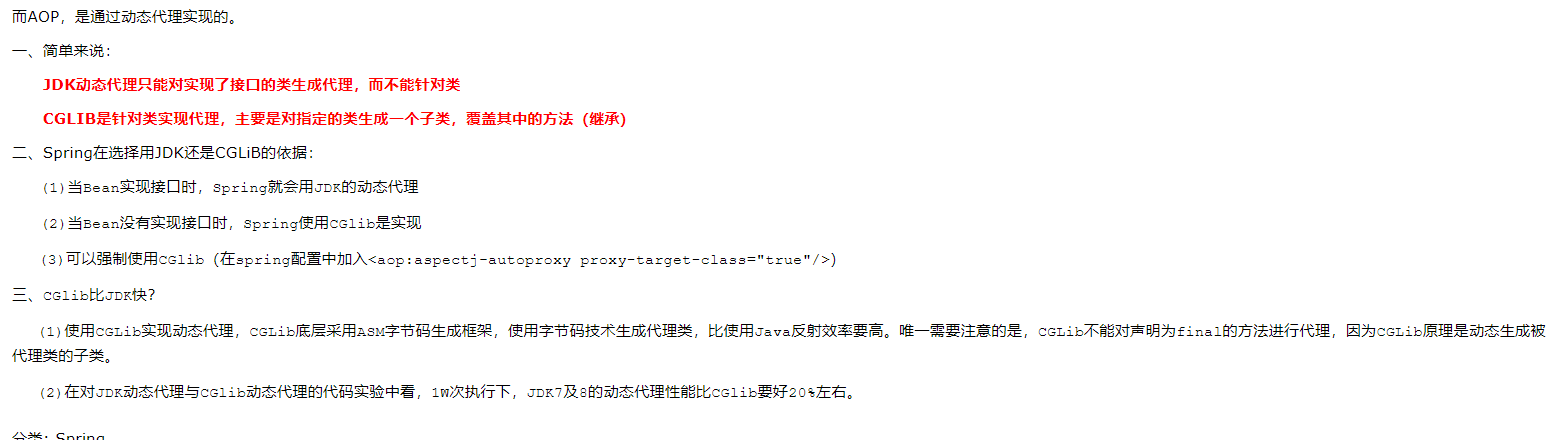
AOP是一种程序设计范型，该范型以一种称为切面的语言构造为基础，切面是一种新的模块化机制。

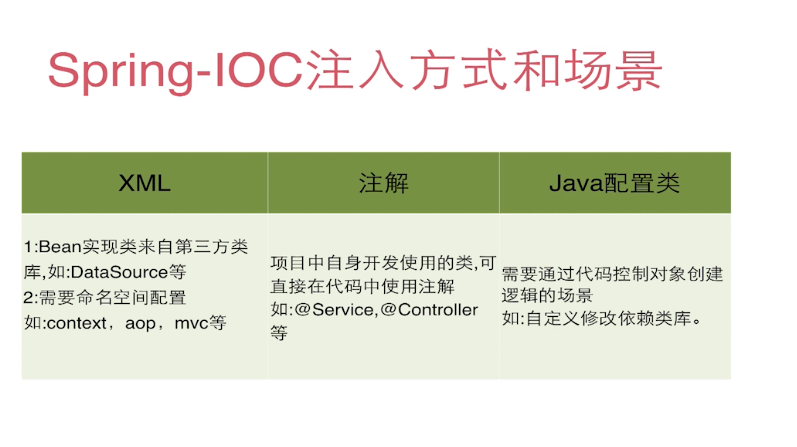
1. Spring AOP的实现原理

动态代理（利用反射和动态编译将代理模式变成动态的）

JDK的动态代理

Cglib动态代理





Java配置类少见

Xml配置注入有 设值方法注入（setter getter）和构造方法注入

主要用设值

### Ioc容器的加载过程

IOC注册bean的过程

Refresh方法中

初始化beanfactory obtainFreshBeanFactory中的refreshBeanFactory

AbstractRefreshableApplicationContext 的refreshBeanFactory方法：

1. 创建IOC配置文件的抽象资源
2. 创建一个BeanFactory
3. 把读取配置信息的BeanDefinitionReader,这里是XmlBeanDefinitionReader来载入xml文件形式的BeanDefinition
4. 然后将上面定义好的resource通过一个回调配置给BeanFactory
5. 从定义好的资源位置读入配置信息，具体的解析过程由XmlBeanDefinitionReader来完成，这样完成整个载入bean的过程
6. Resource定位（Bean的定义文件定位）
7. 将Resource定位好的资源载入到BeanDefinition
8. 将BeanDefiniton注册到容器中

<https://www.jianshu.com/p/38d8eecbf4b7>

### IOC 容器依赖注入

也就是getBean()方法

大牛博客

<https://www.cnblogs.com/ITtangtang/p/3978349.html>

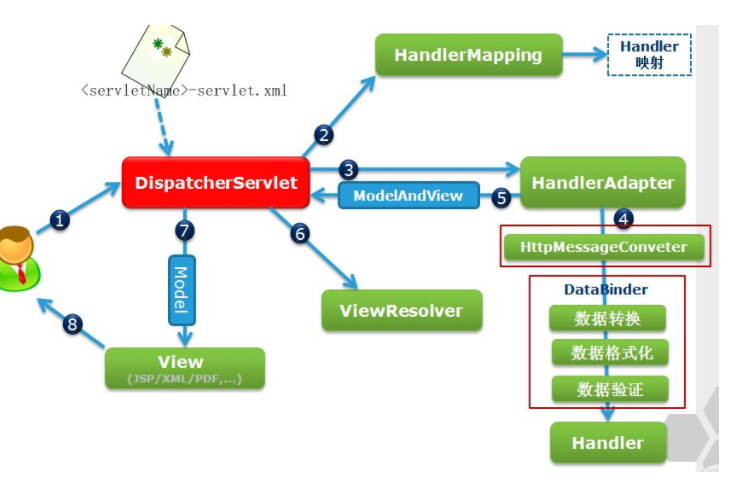
## Hibernate一级缓存与二级缓存的区别(看下就行)

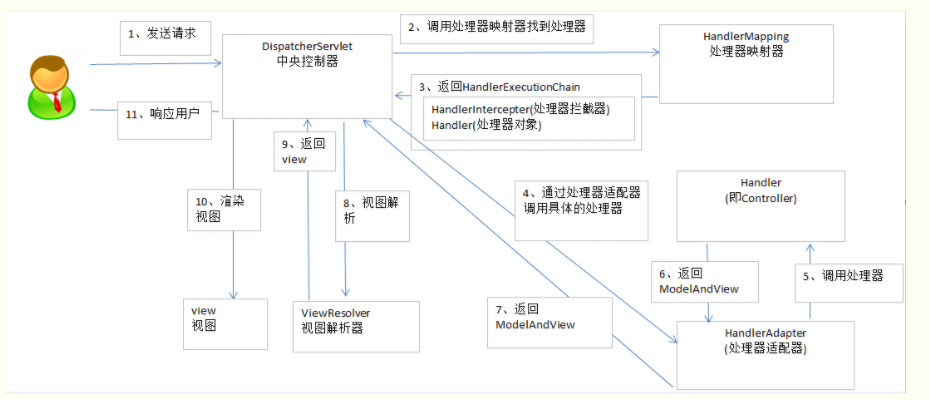
Hibernate的Session提供了一级缓存的功能，默认总是有效，当应用程序保存持久化实体、修改持久化实体时，Session并不会立即把这种改变提交到数据库，而是缓存在当前的Session中，除非显示调用了Session的flush（）方法或通过close()方 法关闭Session。通过一级缓存，可以减少程序与数据库的交互，从而提高数据库访问性能。

SessionFactory级别的二级缓存是全局性的，所有的Session可以共享这个二级缓存。不过二级缓存默认是关闭的，需要显 示开启并指定需要使用哪种二级缓存实现类（可以使用第三方提供的实现）。一旦开启了二级缓存并设置了需要使用二级缓 存的实体类，SessionFactory就会缓存访问过的该实体类的每个对象，除非缓存的数据超出了指定的缓存空间。

一级缓存和二级缓存都是对整个实体进行缓存，不会缓存普通属性，如果希望对普通属性进行缓存，可以使用查询缓存。查询缓存是将HQL或SQL语句以及它们的查询结果作为键值对进行缓存，对于同样的查询可以直接从缓存中获取数据。查询缓存默认也是关闭的，需要显示开启。

## Springmvc的原理





A、客户端的所有请求都交给前端控制器DispatcherServlet来处理，它会负

调用系统的其他模块来真正处理用户的请求。

B、DispatcherServlet收到请求后，将根据请求的信息（包括URL、HTTP协议方法，请求头、请求参数、Cookie等）以及HandlerMapping的配置找到处理该请求的Handler（任何一个对象都可以作为请求的Handler）;

C、在这个地方Spring会通过HandleAdapter对该处理进行封装。

D、HandlerAdapter是一个适配器，它用统一的接口对各种Handler中的方法进行调用。

E、Handler完成对用户请求的处理后，会返回一个ModelAndView对象给DispatcherServlet,ModelAndView顾名思义，包含了数据模型以及相应的视图的信息。

F、ModelAndView的视图是逻辑视图，DispatcherServlet还要借助ViewResolver完成从逻辑视图到真实视图对象的解析工作

G、当得到真正的视图对象后，DispatcherServlet会利用视图对象对模型数据进行渲染。

H、客户端得到响应，可能是一个普通的HTML页面，也可以是XML或JSON字符串，还可以是以长图片或者一个PDF文件

## 解决超卖的方法

1）乐观锁

2）队列

建立一条先进先出的队列，每个请求加入到队列中，然后异步获取队列数据进行处理，把多线程变为单线程，处理完一个就从队列中移出一个，因为高并发，可能一下将内存撑爆，然后系统又陷入异常状态。或者设计一个极大的队列，但是系统处理速度和涌入速度根本没办法比，平均响应时间小。

## web应用服务器

(1)Tomcat:应用十分广泛的 web 服务器，支持部分 j2ee，免费，支持 servlet、

Jsp

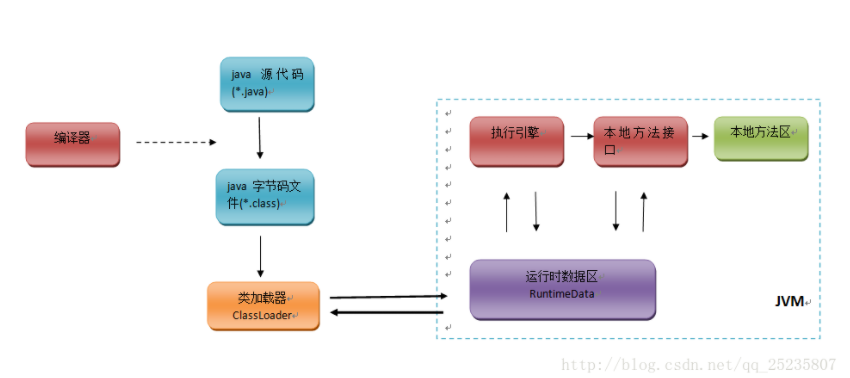
(2)Apache: 全球应用最广泛的 http 服务器，但只支持静态网页，比如 asp、PHP 等动态网页不支持

(3)Nginx:性能优异的web服务器，也可作为反向代理服务器实现负载均衡

(4)Jetty: Jetty 是一个开源的servlet容器，它为基于Java的web容器，例如JSP和servlet提供运行环境。Jetty是使用[Java语言](https://baike.baidu.com/item/Java%E8%AF%AD%E8%A8%80)编写的，它的API以一组JAR包的形式发布。开发人员可以将Jetty容器实例化成一个对象，可以迅速为一些独立运行（stand-alone）的Java应用提供网络和web连接。

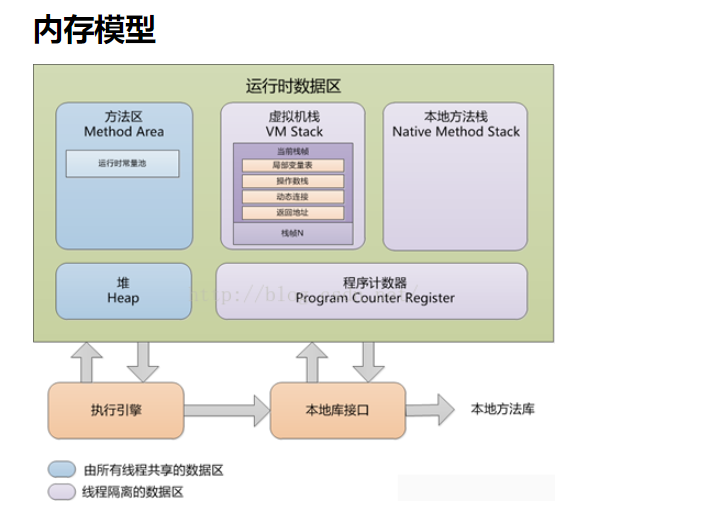
# Java虚拟机

## JVM运行流程



编译器将java源代码 编译成class文件，也就是字节码，这个文件通过类加载器加载如jvm中，通过解释器将字节码再转化成机器码

## Java虚拟机内存的划分



1. 程序计数器（线程私有）

线程创建时创建，执行本地方法时其值为 undefined

1. 虚拟机栈（线程私有）

为虚拟机执行java方法服务：方法被调用时创建栈帧---》局部变量表---》局部变量、对象引用

如果线程请求的栈深度超出了虚拟机所允许的深度，就会出现StackOverFlowError。

虚拟机栈可以动态扩展，如果扩展到无法申请到足够的内存，会出现OOM

1. 本地方法栈（线程私有）

java虚拟机栈是为虚拟机执行java方法服务的，而本地方法栈则为虚拟机执使用到的Native方法服务。

Java虚拟机没有对本地方法栈的使用和数据结构做强制规定。Sun HotSpot把Java虚拟机栈和本地方法栈合二为一

会抛出StackOverFlowError和OutOfMemoryError

1. 堆（线程共享）

被所有线程共享，在java虚拟机启动时创建，几乎所有对象的实例都放在堆中，

GC主要管理的区域

物理不连续，逻辑上连续，并可以动态扩展，无法扩展时抛出oom异常

1. 方法区（线程共享）

用于存储已经被虚拟机加载的类信息、常量、静态变量、即使编译器编译后的代码的数据

Sun HotSpot虚拟机把方法区叫做永久代

1. 运行时常量池

受到方法区的限制，抛出oom异常

## JVM垃圾处理方法（标记清除、复制、标记整理）

1. 标记-清除算法

标记阶段：先通过根结点，标记所有从根节点开始的对象，未被标记的为垃圾对象

清除阶段：清除所有未被标记的对象

缺点：标记和清除的效率都不高，会产生大量不连续的内存碎片

1. 复制算法

将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，然后清除正在使用的内存块中的所有对象。

1. 标记-整理算法

标记阶段：先通过根结点，标记所有从根节点开始的可达对象，未被标记的未垃圾对象

整理阶段：将所有的存活对象压缩到内存的一段，之后清理边界所有的空间

D、分代收集：（当前商业虚拟机的垃圾收集都采用这个）

新生代 复制算法

老年代 标记-清除或标记-整理

现在的商业虚拟机采用复制收集算法来回收新生代，商业虚拟机：将内存分为一块较大的 eden 空间和两块较小的 survivor 空间，默认比例是 8:1:1，即每次新生代中可用内 存空间为整个新生代容量的 90%，每次使用eden和其中一个 survivour。当回收时，将 eden 和 survivor 中还存活的对象一次性复制到另外一块 survivor 上，最后清理掉 eden 和刚才用过的 survivor，若另外一块 survivor 空间没有足够内存空间存放上次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。

三种算法比较：

1. 效率：复制算法 > 标记/整理算法 > 标记/清除算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。
2. 内存整齐度：复制算法=标记/整理算法>标记/清除算法。
3. 内存利用率：标记/整理算法=标记/清除算法>复制算法。

## JVM参数

<https://blog.csdn.net/crazylzxlzx/article/details/52210061>

## JDK调优工具简介

<https://www.cnblogs.com/therunningfish/p/5524238.html>

## 新生代、老年代、永久代

1. 新生代

在方法中去new一个对象，那这方法调用完毕后，对象就会被回收，这就是一个典型的新生代对象

1. 老年代

在新生代中经历了N次垃圾回收后仍然存活的对象就会被放到老年代中，而且大对象直接进入老年代。当Survivor空间不够用时，需要依赖于老年代进行分配担保，大对象直接进入老年代

1. 永久代

也就是方法区

## GC触发条件

<https://blog.csdn.net/yhyr_ycy/article/details/52566105>

## 垃圾收集器

1、Serial收集器：单线程（在它进行垃圾收集的时候，必须暂停其他all工作线程，知道它收集结束）在client模式下是很好的选择。 停止-复制

2、ParNew收集器：serial收集器的多线程版本，是许多运行在server模式下的虚拟机首选的新生代收集器。 停止-复制。

3、Parallel scaverge收集器：目标达到一个可控制的吞吐量，适合在后台运算，没有太多的交互。 停止-复制

4、Serial old: serial的老年代版本，单线程，标记-整理

5、Parallel old： parallel scaverge 老年代版本，多线程 标记-整理

6、Cms收集器：一种以获取最短回收停顿时间为目标的收集器 老年代 标记-清除

7、G1收集器：是当今收集器技术发展的最前沿成果之一

（前三个是新生代 ，4，5，6是老年代）

## Java虚拟机的对象

由三部分组成（1）对象头：标记字段（在32位和64位jvm中分别为32bits和64bits）+类型指针（2）实例数据（3）对齐填充。

若对象是一个Java数组，则对象头中还必须有一块用于记录数组长度的数据。因为虚拟机可以通过普通java对象的元数据信息确定java对象的大小，但是从数组的元数据中无法确定数组的大小。 对象大小必须是8字节的整数倍

## 对象的分配

虚拟机给每一个对象定义一个对象年龄计数器，若对象在eden出生并经过第一次minor gc后仍然存活，并且能被survivor容纳的话，将被转移到survivor空间中，并且对象年龄设为1，对象在survivor中每熬过一次minor gc，年龄就+1，当他年龄到一定程度（默认为15），就会晋升到老年代。

（1）大多数情况下，对象在新生代eden区中分配，当Eden区中没有足够的内存空间进行分配时，虚拟机将发起一次minor Gc(minor gc:发生在新生代的垃圾收集动作，非常频繁，一般回收速度也比较快 full gc:发生在老年代的gc)

（2）大对象直接进入老年代

（3）长期存活的对象将进入老年代

（4））若在 survivor 空间中相同年龄 all 对象大小的总 和 >survivor 空 间 的 一 半 ， 则 年 龄 >= 改 年 龄 的 对 象 直 接 进 入 老 年 代 ， 无 须 等 到 MaxTeuringThreshold（默认为 15）中的要求。

## GC，哪些对象可作为GC Roots对象？

对象是否存活？

1. 引用计数法 缺点：很难解决对象之间循环引用的问题。
2. 可达性分析法 基本思想：通过一系列的称为“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，当一个对象到 GC root 没有任何引用链相连（用图论的话来说，就是从 GC roots 到这个对象不可达），则证明此对象是不可用的。

可作为GC Roots对象的：

1. java虚拟机栈中引用的对象
2. 方法区中的静态成员
3. 方法区中的常量引用对象
4. 本地方法区的JNI（Native方法）引用对象。

引用强度 强引用>软引用>弱引用>虚引用

## MinGC,FullGC

1. MinGC:

新生代中的垃圾收集动作，采用的是复制算法。

对于较大的对象，在Minor GC的时候直接进入老年代

1. FullGC:

Full GC是发生在老年代的垃圾收集动作，采用的是标记-清除/整理算法。

由于老年代的对象几乎都是在Survivor区熬过来的，不会那么容易死掉。因此Full GC发生的次数不会有Minor GC那么频繁，并且Time(Full GC)> Time(Minor GC)

<https://www.cnblogs.com/yang-hao/p/5948207.html>

## Student s = new Student();在内存中做了哪些事情？

加载Student.class文件进内存

在栈内存为s开辟空间

在堆内存为学生对象开辟空间

对学生对象的成员变量进行默认初始化

对学生对象的成员变量进行显示初始化

通过构造方法对学生对象的成员变量赋值

学生对象初始化完毕，把对象地址赋值给s变量

## 类加载

在代码编译后，就会生成JVM（Java虚拟机）能够识别的二进制字节流文件（\*.class）。而JVM把Class文件中的类描述数据从文件加载到内存，并对数据进行校验、转换解析、初始化，使这些数据最终成为可以被JVM直接使用的Java类型，这个说来简单但实际复杂的过程叫做JVM的类加载机制。

类从加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载-验证-准备-解析-初始化-使用-卸载，其中验证-准备-解析成为链接。

在遇到下列情况时，若没有初始化，则需要先触发其初始化（加载-验证-准备需要在此之前）：

（1）使用new关键字实例化对象/读取或设置一个类的静态字段/调用一个类的静态方法

（2）使用java.lang.reflect包的方法对类进行反射调用时，若类没有进行初始化，则需要触发其初始化

（3）当初始化一个类时，若发现其父类还没有进行初始化，则要先触发其父类的初始化。

（4）当虚拟机启动时，用户需要制定一个要执行的主类（main方法的那个类），虚拟机会先初始化这个类。

在加载阶段，虚拟机需要完成下面3件事：

1. 通过一个类的全限定名获取定义此类的二进制字节流；
2. 将这个字节流所表示的静态存储结构转化为方法区运行时数据结构
3. 在内存中生成一个代表这个类的class对象，作为方法区的各种数据的访问入口。

验证的目的是为了确保class文件的字节流中包含的信息符合当前虚拟机的要求，且不会危害虚拟机自身的安全。验证阶段大致会完成下面四个阶段的检验动作：（1）文件格式验证（2）元数据验证（3）字节码验证（4）符号引用验证

准备阶段是正式为类变量分配内存并设置变量的初始化值的阶段，这些变量所使用的内存都将在方法区中进行分配。（不是实例变量，且是初始值，若 public static int a=123;准备阶段后a的值为0，而不是123，要在初始化之后才变为123，但若被final修饰，public static final int a=123;在准备阶段后就变为了123）；

## 类加载器

(1)Bootstrap ClassLoader(启动类加载器): 主要负责jdk\_home/lib目录下的核心 api 或 -Xbootclasspath 选项指定的jar包装入工作. 它用来加载 Java 的核心库，是用原生代码来实现的

(2)Extension ClassLoader(扩展类加载器): 主要负责jdk\_home/lib/ext目录下的jar包或 -Djava.ext.dirs 指定目录下的jar包装入工作。 它用来加载 Java 的扩展库。

(3)System ClassLoader(系统类加载器)：主要负责java -classpath/-Djava.class.path所指的目录下的类与jar包装入工作. 它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类.

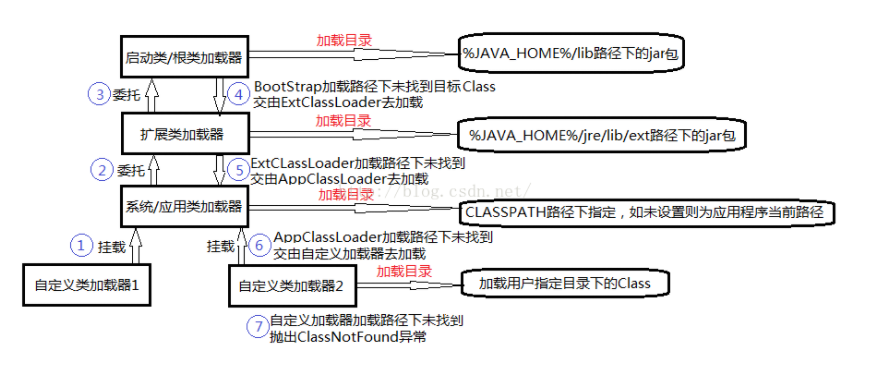
(4) User Custom ClassLoader/用户自定义类加载器(java.lang.ClassLoader的子类)

在程序运行期间, 通过java.lang.ClassLoader的子类动态加载class文件, 体现java动态实时类装入特性.

Java虚拟机的第一个类加载器是Bootstrap，这个加载器很特殊，它不是Java类，因此它不需要被别人加载，它嵌套在Java虚拟机内核里面，也就是JVM启动的时候Bootstrap就已经启动，它是用C++写的二进制代码（不是字节码），它可以去加载别的类。

这也是我们在测试时为什么发现System.class.getClassLoader()结果为null的原因，这并不表示System这个类没有类加载器，而是它的加载器比较特殊，是BootstrapClassLoader，由于它不是Java类，因此获得它的引用肯定返回null。

## 类加载器双亲委派模型

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的Class），子类加载器才会尝试自己去加载。

## 自定义类加载器 打破双亲委派模型

**自定义类加载器步骤**

**（1）继承ClassLoader    （2）重写findClass（）方法   （3）调用defineClass（）方法**

打破：

1、如果不想打破双亲委派模型，那么只需要重写findClass方法即可

2、如果想打破双亲委派模型，那么就重写整个loadClass方法

# 数据结构

## B树、B-树、B+树，B\*树

### B树就是B-树

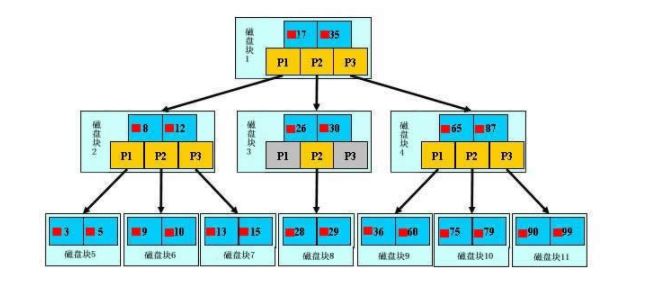
为了将大型数据库文件存储在硬盘上 以减少访问硬盘次数的目的

B-树是一种平衡的多路查找树，它在文件系统中很有用

定义：一棵m阶的B-树，或者为空树，或为满足下列特性的m叉树：

* 1. 树中每个结点之多有m棵子树
  2. 若根结点不是叶子结点，则至少两棵子树
  3. 出根结点之外的非终端结点至少有 m/2棵子树
  4. 所有的叶子结点都出现在同一层次

（M=3）



B-树的具体查找步骤如下（假设查找的关键字为key）：  
1）先让key与根结点中的关键字比较，如果key等于k[i]（k[]为结点内的关键字数组），则查找成功  
2）若key<k[1]，则到p[0]所指示的子树中进行继续查找（p[]为结点内的指针数组），这里要注意B-树中每个结点的内部结构。  
3）若key>k[n]，则道p[n]所指示的子树中继续查找。  
4）若k[i]<key<k[i+1]，则沿着指针p[I]所指示的子树继续查找。  
5）如果最后遇到空指针，则证明查找不成功。

### B+树

主要应用于数据库索引

B+树是B-树的变体，也是一种多路搜索树：

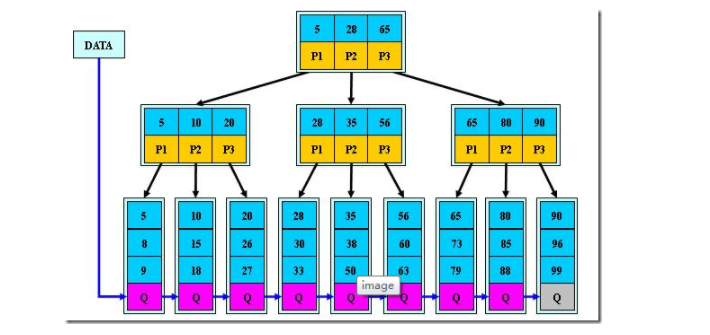
1.其定义基本与B-树同，除了：

2.非叶子结点的子树指针与关键字个数相同；

3.非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

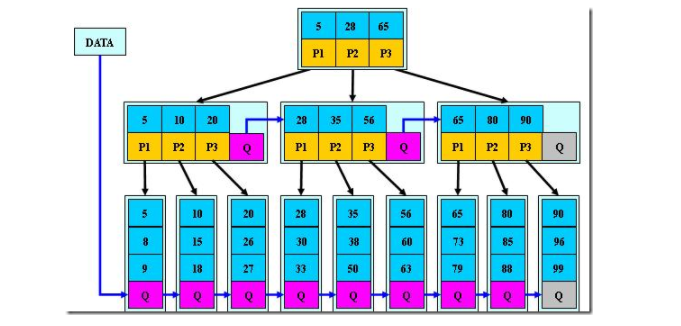
5.为所有叶子结点增加一个链指针；

6.所有关键字都在叶子结点出现；



### B\*树

是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针；



  B\*树分配新结点的概率比B+树要低，空间使用率更高；

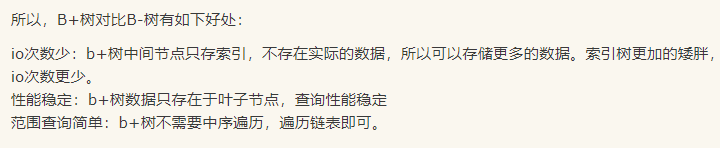
B树：二叉树，每个结点只存储一个关键字，等于则命中，小于走左结点，大于走右结点；

B-树：多路搜索树，每个结点存储M/2到M个关键字，非叶子结点存储指向关键字范围的子结点；

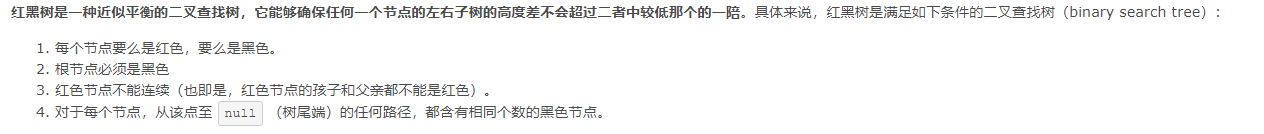
所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

B+树：在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

B\*树：在B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从1/2提高到2/3；



## 红黑树



# 数据库

## 索引

索引：是帮助高效获取数据的数据结构，对数据表中一个或多个列的值进行排序的结构

1. 索引加快数据库的检索速度
2. 唯一索引可以确保每一行数据的唯一性
3. 使用索引，可以在查询过程中提高性能
4. 创建索引和维护索引要耗费时间且索引需要占据物理空间和数据空间
5. 当对表中的数据进行增加、删除和修改的时候，索引也需要维护，降低数据维护的速度

以下情况适合建立索引:

1. 经常出现在关键字order by、group by、distinct后面的字段，建立索引
2. 为经常用作查询选择的字段，建立索引
3. 在经常用作表连接的属性上，建立索引
4. 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，指定的范围是连续的

因为索引需要额外的存储空间和处理,那些不必要的索引反而会使查询反应时间变慢.使用索引查询不一定能提高查询性能

唯一索引：不允许其中任何两行有相同的值。

主键索引：为表定义主键自动创建主键索引，是唯一索引的特定类型

聚集索引：主要是为了快速的缩小查找范围，即记录数目未定。

Mysql索引使用的使B+的数据结构

## Mysql Btree索引 和hash索引

Hash索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询。

联合索引中，Hash索引不能利用部分索引键查询

Hash 索引无法被用来避免数据的排序操作

Hash索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高

## MyIsam 索引和 innodb 索引

<https://blog.csdn.net/ljfphp/article/details/80029968>

## 索引优化

## 聚集索引和非聚集索引

聚簇索引就是INNODB存储索引的方式 叶子节点就是数据域

1.聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个。

2.聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续。  
3.聚集索引查询数据速度快，插入数据速度慢；非聚集索引反之。

聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致，   
聚集索引表记录的排列顺序与索引的排列顺序一致，优点是查询速度快，因为一旦具有第一个索引值的纪录被找到，具有连续索引值的记录也一定物理的紧跟其后。   
聚集索引的缺点是对表进行修改速度较慢，这是为了保持表中的记录的物理顺序与索引的顺序一致，而把记录插入到数据页的相应位置，必须在数据页中进行数据重排，降低了执行速度。   
非聚集索引指定了表中记录的逻辑顺序，但记录的物理顺序和索引的顺序不一致，聚集索引和非聚集索引都采用了B+树的结构，但非聚集索引的叶子层并不与实际的数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针的方式。非聚集索引比聚集索引层次多，添加记录不会引起数据顺序的重组。

## 聚簇索引，非聚簇索引，覆盖，复合索引等等

## 三大范式

a、第一范式（1NF）：数据库表中的字段都是单一属性的，不可再分。这个单一属性由基本类型构成，包括整型、实数、字符型、逻辑型、日期型等。

b、第二范式（2NF）：数据库表中不存在非关键字段对任一候选关键字段的部分函数依赖（部分函数依赖指的是存在组合关键字中的某些字段决定非关键字段的情况），也即所有非关键字段都完全依赖于任意一组候选关键字。

c、第三范式（3NF）：在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合第三范式。所谓传递函数依赖，指的是如果存在"A → B → C"的决定关系，则C传递函数依赖于A。因此，满足第三范式的数据库表应该不存在如下依赖关系： 关键字段 → 非关键字段

## 事务及四大特性

事务（Transaction）是并发控制的基本单位。是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。事务是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性。

四大特性:

Atomic（原子性）:事务中包含的操作看成一个逻辑单元，这个逻辑单元中的操作要么全部成功，要么全部失败，事务中的所有元素作为一个整体提交或回滚，事务时一个完整操作。

Consistency（一致性）:只有合法的数据被写入，事务完成时，数据必须是一致的

Isolation(隔离性)：对数据修改的多个事务是彼此隔离的。

Durability（持久性）:事务完成之后，对于系统的影响是永久的。

## 乐观锁和悲观锁

数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。乐观并发控制(乐观锁)和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。就是很乐观，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。适用于多读，比如 write\_condition. 两种锁各有优缺点，不能认为一种比一种好。

悲观锁：就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁。这样别人想拿这个数据就会 block 直到它拿到锁。传统的关系型数据库就用到了很多这种机制，比如行锁，写锁等，都是在操作之前上锁。 即假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作

Mysql中乐观锁 悲观锁的实现

<https://blog.csdn.net/daybreak1209/article/details/51606939>

## 超键、候选键、主键、外键分别是什么

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

候选键：是最小超键，即没有冗余元素的超键.

主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

外键：在一个表中存在的另一个表的主键称此表的外键。

## 什么是视图

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，试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

## 脏读、不可重复读和幻读（事务隔离级别）

脏读：事务T1更新了一行记录的内容，但是并没有提交所做的修改。事务T2读取更新后的行，然后T1执行了回滚操作，取消了刚才所做的修改。现在T2读取的行就无效了（一个事务读取了另一个事务未提交的数据）

不可重复读：事务T1读取了一行记录，紧接着T2修改了T1刚才读取的那一行记录，然后T1又再次读取这行记录，发现与刚才读取的结果不同。

幻读：事务T1读取一条指定的Where子句所返回的结果集，然后T2事务新插入一行记录，这行记录恰好可以满足T1所使用 的查询条件。然后T1再次对表进行检索，但又看到了T2插入的数据。

事务隔离级别：

（1）、Read uncommitted

读未提交，一个事务可以读取另一个未提交事务的数据 也就是脏读

解决脏读 Read committed 读提交

（2）、Read committed

读提交，一个事务要等另一个事务提交后才能读取数据。 这就是读提交，若有事务对数据进行更新（UPDATE）操作时，读操作事务要等待这个更新操作事务提交后才能读取数据，可以解决脏读问题。但在这个事例中，出现了一个事务范围内两个相同的查询却返回了不同数据，这就是不可重复读。

怎么解决可能的不可重复读？ Repeatable read!

（3）、Repeatable read

重复读，就是在开始读取数据（事务开启）时，不再允许修改操作

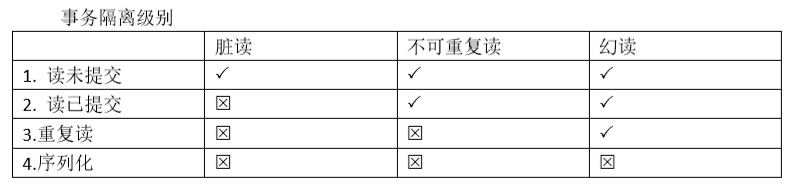
重复读可以解决不可重复读问题。写到这里，应该明白的一点就是，不可重复读对应的是修改，即UPDATE操作。但是可能还会有幻读问题。因为幻读问题对应的是插入INSERT操作，而不是UPDATE操作。

那怎么解决幻读问题？Serializable！

1. Serializable 序列化

Serializable 是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

大多数数据库默认的事务隔离级别是Read committed，比如Sql Server , Oracle。Mysql的默认隔离级别是Repeatable read。



## Redis

Redis是一个速度非常快的非关系型数据库，它可以存储键与5种不同类型的值之间的映射，可以将存储在内存种的键值对数据持久化到硬盘中。

Redis的数据结构：

a、STRING:可以是字符串、整数或者浮点数

b、LIST：一个链表，链表上的每个节点都包含了一个字符串

c、SET：包含字符串的无序收集器（unordered collection），并且被包含的每个字符串都是独一无二、各不相同的。

d、HAST：包含键值对的无序散列表。

e、ZSET：字符串成员（member）与浮点数分值（score）之间的有序映射，元素的排列顺序由分值的大小决定。

#### Memchached:

是高性能分布式内存缓存服务器，本质是一个内存 key-value 数据库，但不支持数据持久化，服务器关闭后，数据全丢失。只支持 key-value 结构。

Redis与Memcached相比：

* 1. 两者都可用于存储键值映射，性能差不多
  2. Redis能够自动以两种不同方式将数据写入硬盘
  3. Redis除了能存储普通字符串键之外，还可以存储其他4种数据结构
  4. Redis既能用作主数据库，又可以作为其他存储系统的辅助数据库。
  5. Redis中并不是all数据都一直存储在内存中，这是和memcached相比一个最大的区别
  6. Redis支持数据备份，master-slave模式的备份。Redis支持数据的持久化，可以将内存中的数据保存在磁盘上，重启的时候可以再次加载进内存使用。Memcached服务器关闭后，数据丢失，redis数据丢失后可以通过灾难恢复

## 数据库锁机制

在数据库的锁机制中介绍过，在DBMS中，可以按照锁的粒度把数据库锁分为行级锁(INNODB引擎)、表级锁(MYISAM引擎)和页级锁(BDB引擎 )。



A、行级锁：行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁和排他锁。

开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

B、表级锁：表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

C、页级锁：页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。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。BDB支持页级锁。

开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

MyISAM：不支持事务行级锁和外键约束。所以当执行insert和update时，执行写操作时，要锁定整个表，所以效率低。但是它保存了表的行数，执行select count(\*) from table时，不需要全表扫描，而是直接读取保存的值。所以若读操作远远多于写操作，并且不需要事务，Myisam是首选。

Innodb:支持事务、行级锁和外键，mysql运行时，Innodb会在内存中建立缓冲池，用于缓冲数据和索引。不保存表的行数，执行select count(\*) from table时要全表扫描。写不锁定全表，高效并发时效率高。

## 主从复制 读写分离

应用于mysql集群中 将配置主数据库服务器和 从数据库服务器 只从从数据库服务器读，写的话从 主数据库服务器写 并将修改的内容复制到从服务器中进行更新 。

## 分库分表

## Left join Inner join

连接

Select \_\_\_ from 表1 Inner join 表2 on \_\_\_ = \_\_

left join(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录   
　　right join(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录  
　　inner join(等值连接) 只返回两个表中联结字段相等的行

Inner join只返回 联接成功的

Left join 即使没有连接 左边的表 也要全部返回。 不属于左边表的字段 为空就行

## Mysql数据库优化方法

Sql语句优化：

使用连接（join）代替子查询。

使用事务

使用联合（UNION）代替手动创建的临时表

使用外键

优化查询语句

使用索引

# 操作系统

## 进程和线程的区别？

1. 进程是程序的一次执行，而线程可以理解为程序中运行的而一个片段
2. 由于线程没有独立的地址空间，因此同一个进程的一组线程可以访问该进程资源，这些线程之间的通信也很高效
3. 线程之间的通信简单（共享内存即可，但须注意互斥访问的问题），而不同进程之间的通信更为复杂，通信需要调用内核实现。
4. 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他进程产生影响，而线程只是一个进程中的不同执行路径，线程有自己的堆栈和局部变量，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，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。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

## 操作系统什么情况下会死锁？

产生死锁的必要条件：

1. 互斥条件

即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有

1. 不可抢占条件

进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。

1. 占有且申请条件

进程至少已经占有一个资源，但又申请新的资源，由于该资源已被另外进程占有，此时进程阻塞。但是，它在等待新资源之时，仍继续占用已占有的资源。

1. 循环等待条件

死锁预防：

打破互斥条件、打破不可抢占条件、打破占有且申请条件、打破循环等待条件。

## 进程间通信有哪几种方式

A、管道:管道可用于具有亲缘关系进程间的通信

B、命名管道：还允许无亲缘关系进程间的通信

C、信号：信号是比较复杂的通信方式，用于通知接受进程有某种时间发生

D、消息队列：是消息的链接表，克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺陷

E、共享内存：多个进程可以访问同一块内存空间

F、内存映射：内存映射允许任何多个进程间通信

G、信号量：主要作为进程间以及同一进程不同线程之间的同步手段。

H、套接口：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。

## 线程同步与阻塞的关系？同步一定阻塞吗？阻塞一定同步吗？

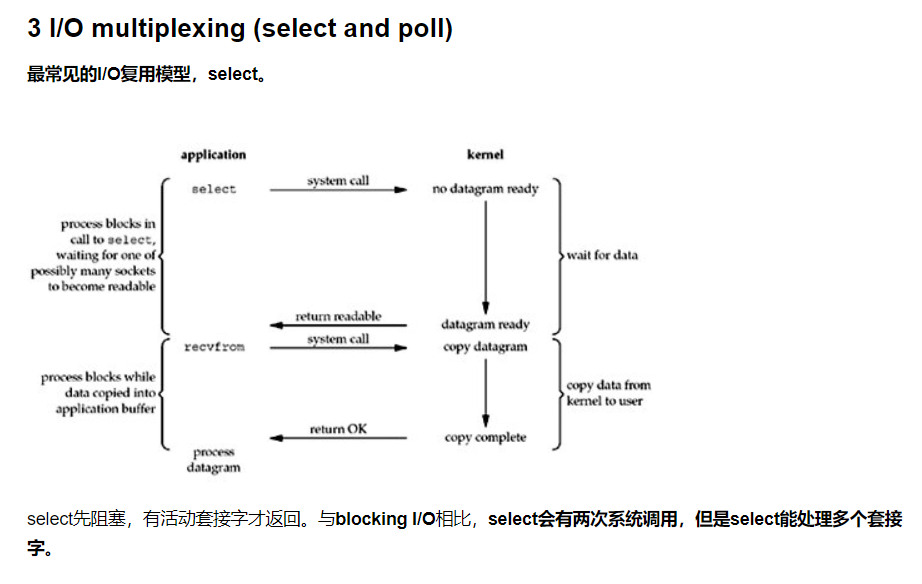
线程同步与阻塞的关系

1. 线程同步与阻塞没有一点关系
2. 同步和异步关注的是消息通信机制。
3. 阻塞和非阻塞关注的是程序在等待调用结果时的状态。

## Select poll epoll

异步非阻塞直接在完成后通知，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

select，poll，epoll都是IO多路复用的机制。I/O多路复用就是通过一种机制，一个进程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。



<https://segmentfault.com/a/1190000003063859>

java的nio底层 可以看到在Linux下，内核版本大于2.6时使用epoll，小于2.6时使用poll

# 计算机网络

## TCP/IP四层模型

<https://www.cnblogs.com/z-sm/p/7501549.html>

## HTTP相关

HTTP协议是Hyper Text Transfer Protocol（超文本传输协议）的缩写,是用于从万维网（WWW:World Wide Web ）服务器传输超文本到本地浏览器的传送协议。

HTTP是一个基于TCP/IP通信协议来传递数据（HTML 文件, 图片文件, 查询结果等）。

HTTP协议工作于客户端-服务端架构为上。浏览器作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。Web服务器根据接收到的请求后，向客户端发送响应信息。

HTTP 是一种不保存状态，即无状态（stateless）协议。HTTP 协议自身不对请求和响应之间的通信状态进行保存。也就是说在 HTTP 这个级别，协议对于发送过的请求或响应都不做持久化处理。

特点:

A、简单快速，客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

B、灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

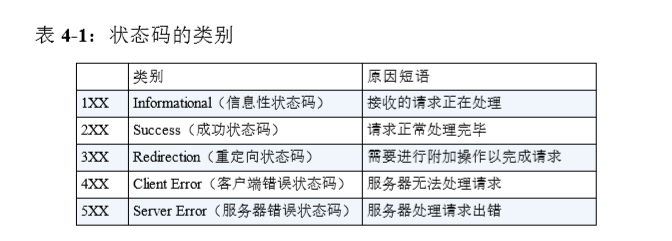
C、无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

D、无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

E、支持B/S及C/S模式。

<https://www.jianshu.com/p/80e25cb1d81a>

## Http状态码



参考 图解HTTP第四章

## Http和https的区别

A、http是HTTP协议运行在TCP上。所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份

B、https是HTTP运行在SSL/TLS运行在TCP之上，所有的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。

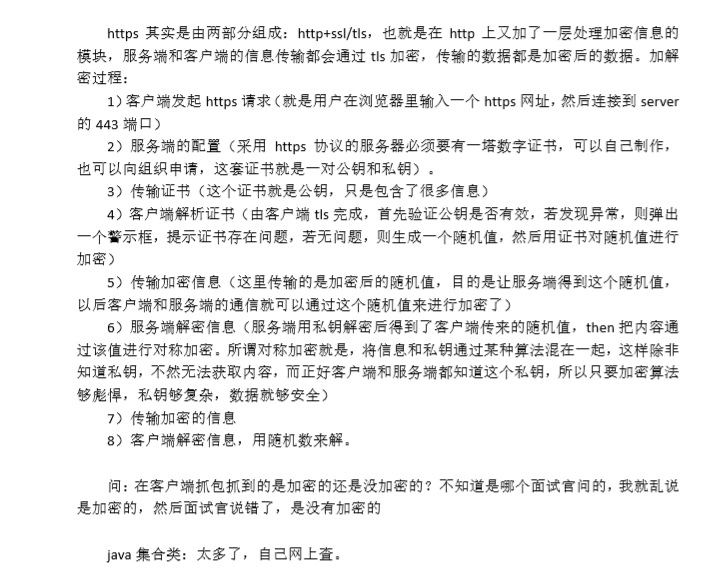
C、https协议需要到ca申请证书

D、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

E、http和https使用的是完全不同的连接方式，端口也不一样，前者80后者443

F、http的连接很简单，是无状态的

G、HTTPS协议是由SSL+HTTP协议构建的课进行加密传输、身份认证的网络协议 要比http协议安全。



## TCP如何保证可靠传输？三次握手、四次挥手过程

TCP如何保证可靠传输

A、数据包校验

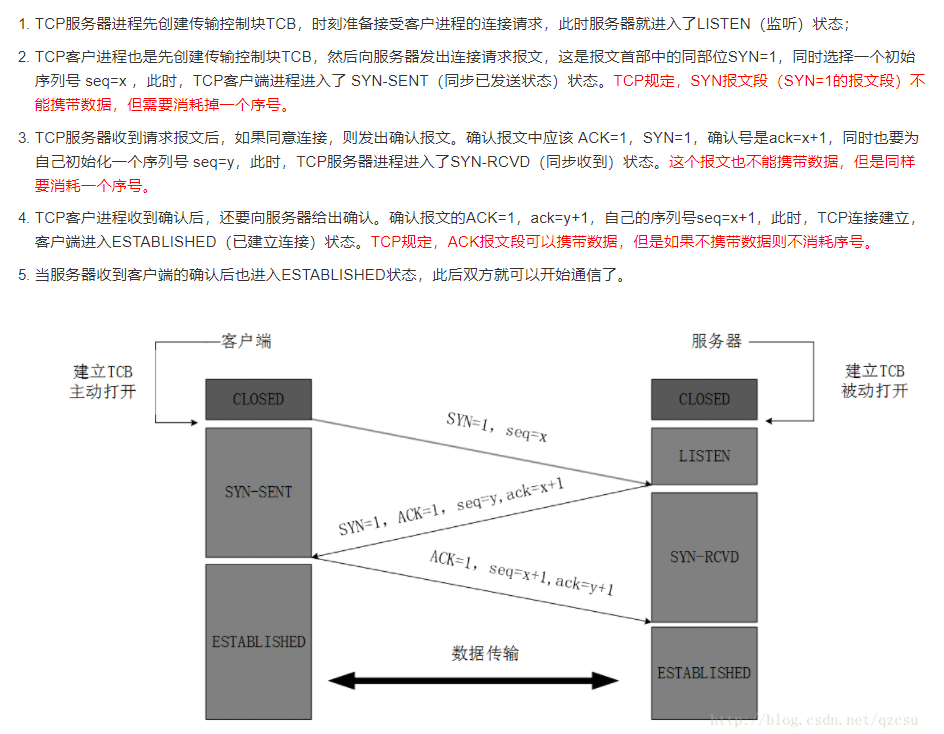
B、超时重传机制

C、应答机制

D、对失序数据包重排序

E、提供流量控制

TCP建立连接的三次握手过程，以及关闭连接的四次握手过程

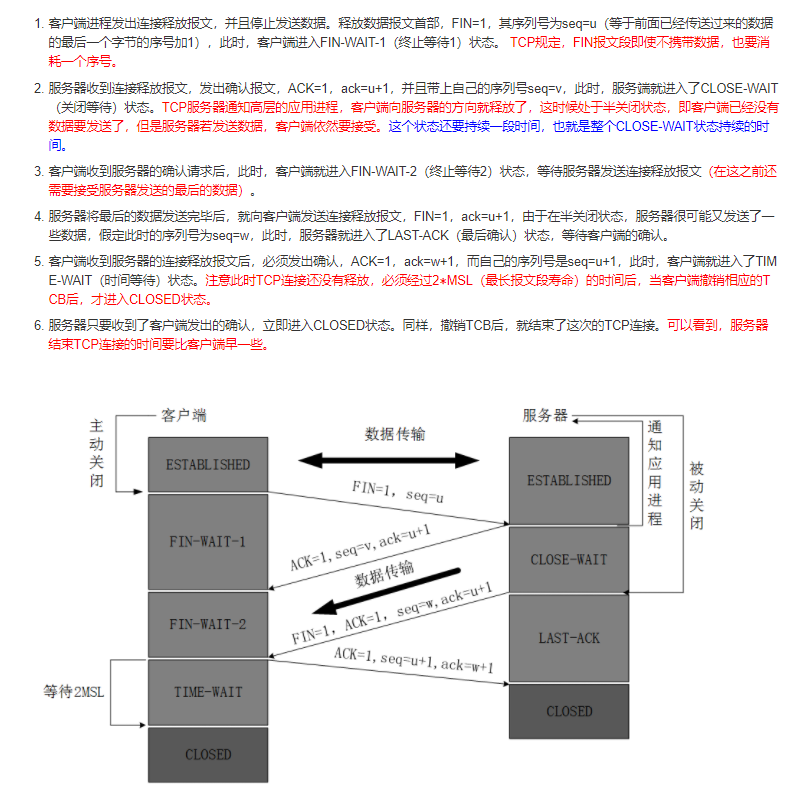


SYN，ACK是标志位

SEQ，AN是数据包序号

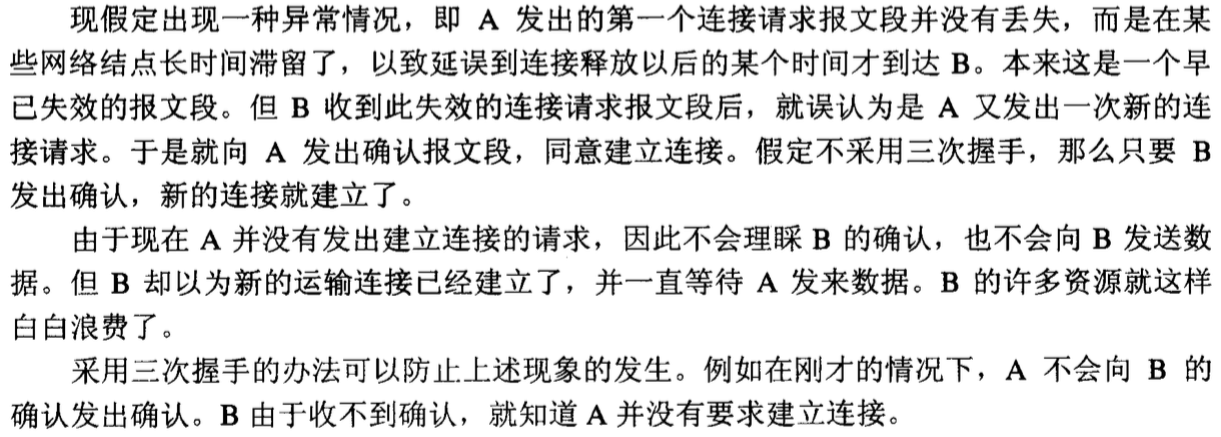
SyN建立连接 ack相应 FIN关闭连接

四次挥手



为什么TCP连接需要三次握手，两次不可以吗？为什么？

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误



TCP握手采用三次 为什么挥手要四次

,这是由于TCP的半关闭造成的,因为TCP连接是全双工的(即数据可在两个方向上同时传递)所以进行关闭时每个方向上都要单独进行关闭,这个单方向的关闭就叫半关闭.

关闭的方法是一方完成它的数据传输后,就发送一个FIN来向另一方通告将要终止这个方向的连接.当一端收到一个FIN,它必

## 慢启动和拥塞避免

## 长连接和短链接

## 如果客户端不断的发送请求连接会怎样？

服务器端会为每个请求创建一个连接，然后向client端发送创建连接时的回复，然后进行等待客户端发送第三次握手数据包，这样会白白浪费资源。

DDos攻击：

简单的说就是想服务器发送链接请求：

第一步：客户端向服务器端发送连接请求数据包

第二步：服务器向客户端回复连接请求数据包，然后服务器等待客户端发送tcp/ip链接的第三步数据包

第三步：如果客户端不向服务器端发送后一个数据包(3)，则服务器须等待30s到2min中才能将此链接进行关闭。当大量的请求只进行到第二步，而不进行第三步，服务器又大量的资源等待第三个数据包。则造成DDos攻击。

## GET和POST的区别

Get:获取资源

GET 方法用来请求访问已被 URI 识别的资源。指定的资源经服务器端解析后返回响应内容。也就是说，如果请求的资源是文本，那就保持原样返回；如果是像 CGI（Common Gateway Interface，通用网关接口）那样的程序，则返回经过执行后的输出结果。

POST：传输实体主体

虽然用 GET 方法也可以传输实体的主体，但一般不用 GET 方法进行传输，而是用 POST 方法。虽说 POST 的功能与 GET 很相似，但 POST 的主要目的并不是获取响应的主体内容。

Get是从服务器上获取数据,post是从服务器传送数据

A、GET被强制服务器支持

B、浏览器对URL的长度有限制，所以GET请求不能代替POST请求发送大量数据

C、GET请求发送数据更小

D、GET请求是不安全的

F、POST请求不能被缓存

G、POST请求相对GET请求是安全的

以下情况中，请使用POST请求：

1、无法使用缓存文件（更新服务器上的文件或数据库）

2、向服务器发送大量数据（POST没有数据量限制）

3、发送包含未知字符的用户输入时，POST比GET更稳定也更可靠

4、post比Get安全性更高

## TCP和UDP的区别

TCP和UDP的区别

1、UDP是无连接的，即发送数据之前不需要建立连接

2、UDP使用尽最大努力交付，即不保证可靠交付，同时也不使用拥塞控制。

3、UDP是面向报文的，UDP没有拥塞控制，很适合多媒体通信的要求

4、UDP支持一对一、一对多、多对已和多对多的交互通信。

5、TCP是面向连接的传输层协议

6、TCP提供可靠交付的服务

7、TCP提供全双工通信

8、TCP是面向字节流的

TCP的拥塞处理

1. 慢启动 （2）拥塞避免 （3）拥塞发生 （4）快速恢复

## TCP/UDP 报文格式

<https://blog.csdn.net/yefanchen01/article/details/70053191>

## 从输入网址到获取页面的过程

1、查询DNS，获取域名对应的ip地址

A、浏览器搜索自身的DNS缓存

B、搜索操作系统的DNS缓存

C、读取本地的HOST文件

D、发起一个DNS的系统调用

（1）宽带运营服务器查看本身缓存

（2）运营商服务器发起一个迭代DNS解析请求

2、浏览器获得域名对应的ip地址后，发起HTTP三次握手

3、TCP/IP连接建立后，浏览器就可以向服务器发送HTTP请求。

4、服务器接受到这个请求，根据路径参数，经过后端的一些处理生成HTML页面代码返回给浏览器

5、浏览器拿到完整的HTML页面代码开始解析和渲染，如果遇到引用外部JS,CSS，图片等静态资源，它们同样也是一个个的HTTP请求

6、浏览器根据拿到的资源对页面进行渲染，最终把一个完整的页面呈现给用户

## 负载均衡

### 基于DNS实现负载均衡

DNS负责提供域名解析服务，当我们访问某个站点时，实际上首先需要通过该站点域名的DNS服务器来获取域名指向的IP地址，再这一过程中，DNS服务器完成了域名到IP地址的映射，同样，这种映射也可以是一对多的，这时候，DNS服务器便充当了负载均衡调度器（负载均衡器）。

DNS负载均衡的实现主要依赖于DNS服务器的设置。 负载均衡调度器工作再DNS层面，调度灵活性或多或少地削弱，策略地开发存在一定局限性

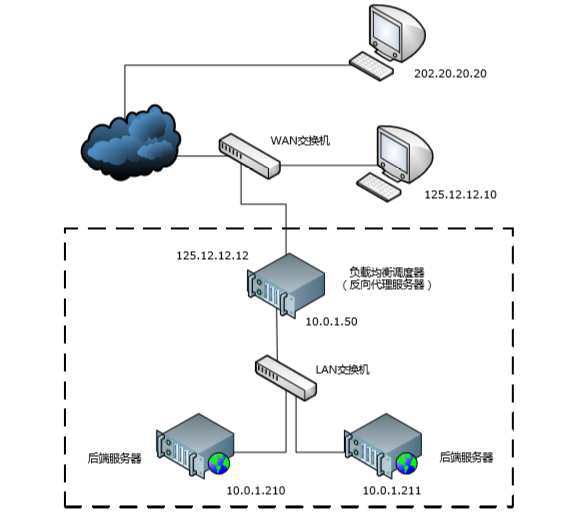
### 基于ip的负载均衡

1、基于NAT技术地负载均衡，工作在传输层，对数据包中地IP地址和端口信息进行修改，所以也称为四层负载均衡。

网络地址转换也就是NAT，它可以让用户伸出内部网络却与互联网建立通信/

### 反向代理负载均衡

反向代理服务器地核心工作便是转发HTTP请求，工作再HTTP层面，也就是TCP七层中地应用层，所以基于反向代理负载均衡也称为七层负载均衡。 把实际服务器称为后端服务器，



分布式Session 分布式缓存 让后端服务器地应用尽量与本地无关，也可跟个号地适应环境。

# Linux及基础命令

## Linux下如何进行进程调度的？

A、中断处理过程（包括时钟中断、I/O 中断、系统调用和异常）中，直接调用schedule，或者返回用户态时根据 need\_resched 标记调用 schedule。

B、内核线程可以直接调用 schedule 进行进程切换，也可以在中断处理过程中进行调度，也就是说内核线程作为一类的特殊的进程可以主动调度，也可以被动调度。

C、用户态进程无法实现主动调度，仅能通过陷入内核态后的某个时机点进行调度，即在中断处理过程中进行调度。

为了控制进程的执行，内核必须有能力挂起正在 CPU 上执行的进程，并恢复以前挂起的某个进程的执行的过程，叫做进程切换、任务切换、上下文切换。挂起正在 CPU 上执行的进程，与中断时保存现场是有区别的，中断前后是在同一个进程上下文中，只是由用户态转向内核态执行。

## 硬链接和软连接

1. 实体链接(硬连接)：它和普通文件类似，实体链接文件的 inode 都指向源文件所在的 block 上，也就是说读取文件直接从源文件的 block 上读取。可以理解是一种copy.

删除任意一个条目，文件还是存在，只要引用数量不为 0。

有以下限制：不能跨越 File System、不能对目录进行链接。

2、符号链接(软连接) 符号链接文件保存着源文件所在的绝对路径，在读取时会定位到源文件上，可以理解为 Windows 的快捷方式。

当源文件被删除了，链接文件就打不开了。

可以为目录建立链接。

## linux常用命令：

文件相关基础命令：

1、pwd 显示当前路径

2、cd + 路径 进入路径

3、cd.. 返回上一级目录

4、ls 查看目录中的文件

5、ls -l 显示文件和目录的详细资料

6、mkdir dir1 创建叫做’dir1’的目录

7、touch file1 创建名叫file1的文件

8、rm -f file1 删除一个叫file1的文件

9、rm -rf dir1 删除一个叫做dir1目录并删除内容

10,、mv dir1 new\_dir 重命名/移动 一个目录

11、cp file1 file2 复制一个文件

12、cat file1 查看文件所有内容

13、head -n file1 查看一个文件前n行

14、chmod -R +权限数组 修改权限

15、find/whereis 查找的意思，用于查找目录及目录下的文件

find命令的一般格式为：find <指定目录><指定条件><指定动作>，即find pathname -options [-print -exec -ok]

pathname：pathname为搜索的目录及其子目录，默认情况下为当前目录

常用的option选项：

-name：按文件名来查找文件

-user：按照文件的属主来查找文件

-group：按照文件所属的组来查找文件

-perm：按照文件权限来查找文件

-prune：不在当前指定目录中查找

locate命令实际是"find -name"的另一种写法，但是查找方式跟find不同，它比find快得多

16、 tar 打包压缩

17、 unzip 解压文件

18、 useradd 添加用户

19、 userdel 删除用户

20、 groupadd 添加用户组

21、 passwd 修改用户密码

22、 su 切换用户身份

23、 ssh 使用ssh加密协议远程登录

24、 scp 远程上传/下载

25、 apt-get install 下载安装

26、 ifconfig 查看、配置、启用或禁用网络接口的命令

27、 netstat 查看网络状态

28、 free 查看系统内存

29，nano/vim编辑器

30、 A | grep B 将A的信息输出到B

31 top命令

top命令是Linux下常用的性能分析工具,能够实时显示系统中各个进程的资源占用状况,类似于Windows的任务管理器.

linux下如何查看网络端口状态（netstat），如何查看内存使用情况（top）；

32 awk命令 可以根据字段的某些条件进行匹配，例如匹配字段小于某个值的那一行数据

## Git常用命令

git --version / /查看git的版本信息

git config --global user.name //获取当前登录的用户

git config --global user.email //获取当前登录用户的邮箱

git config --global user.name 'userName' //设置git账户，userName为你的git账号，

git config --global user.email 'email'

git init //在nodejs文件夹下初始化一个仓库，此时文件里会到一个.git的隐藏文件夹

git branch dev //创建分支

git checkout dev //切换分支

git checkout -b dev //创建分支并切换分支

git branch -d dev //删除分支

git commit -a -m 'dev1' //在分支上提交新的版本

git merge dev //合并分支

git push origin master -u //获取最新代码

git remote add origin 仓库的地址 //连接远程仓库

git remote -v //查看远程仓库

git remote rm origin //删除远程仓库

# Python基础

# 分布式计算、大数据、hadoop、mq等知识（了解，使用部分）

熟悉Spring、Mybatis、SpringMVC、Velocity等主流开源框架，有一定的前端开发技能储备；   
3、熟悉SOA解决方案(如 dubbo)、 缓存（Redis）、消息中间件(Active MQ、Kafka等)等技术;

## Hadoop生态系统



1、HDFS---Hadoop生态圈的基本组成部分时Hadoop分布式文件系统（HDFS），HDFS是一种分布式文件系统，数据被保存在计算机集群中，HDFS为HBase等工具提供了基础。

2、MapReduce---Hadoop的主要执行框架是MapReduce，是一个分布式，并行处理的编程模型，MapReduce把任务分为map（映射）阶段和reduce（化简）。由于MapReduce工作原理的特性，Hadoop能以并行的方式访问数据，从而实现快速访问数据。

3、Hbase---是一个建立在HDFS之上，面向列的NoSQL数据库，用于快速读/写大量数据，HBase使用Zookeeper进行管理

4、Zookeeper—用于Hadoop的分布式协调服务。Hadoop的许多组件依赖于Zookeeper，它运行在计算机集群中，用于管理Hadoop集群

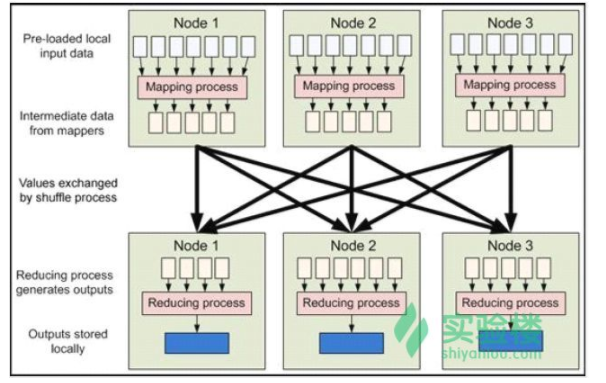
5、Pig--它是MapReduce编程的复杂性的抽象。Pig平台包括运行环境和用于分析Hadoop数据集的脚本语言(Pig Latin)，其编译器将Pig Latin翻译成MapReduce程序序列。

6、Hive--Hive类似于SQL高级语言，用于运行存储在Hadoop上的查询语句，Hive让不熟悉MapReduce开发人员也能编写数据查询语句，然后这些语句被翻译为Hadoop上面的MapReduce任务。像Pig一样，Hive作为一个抽象层工具，吸引了很多熟悉SQL而不是Java编程的数据分析师。

## MapReduce原理

### MapReduce简介

MapReduce 是现今一个非常流行的分布式计算框架，它被设计用于并行计算海量数据。第一个提出该技术框架的是Google 公司，而Google 的灵感则来自于函数式编程语言，如LISP，Scheme，ML 等。MapReduce 框架的核心步骤主要分两部分：Map 和Reduce。当你向MapReduce 框架提交一个计算作业时，它会首先把计算作业拆分成若干个Map 任务，然后分配到不同的节点上去执行，每一个Map 任务处理输入数据中的一部分，当Map 任务完成后，它会生成一些中间文件，这些中间文件将会作为Reduce 任务的输入数据。Reduce 任务的主要目标就是把前面若干个Map 的输出汇总到一起并输出。从高层抽象来看，MapReduce的数据流图如下图所示：



### MapReduce流程分析

见实验楼

<https://www.shiyanlou.com/courses/237/labs/1033/document>

## Zookeeper

ZooKeeper是一种分布式协调服务，用于管理大型主机。在分布式环境中协调和管理服务是一个复杂的过程。ZooKeeper通过其简单的架构和API解决了这个问题。用于在自身之间协调，并通过稳健的同步技术维护共享数据

有没有一种方法能实现自动告知，即机器的增添、剔除对调用方透明，调用者不再需要写死服务提供方地址？当然可以，现如今zookeeper被广泛用于实现服务自动注册与发现功能

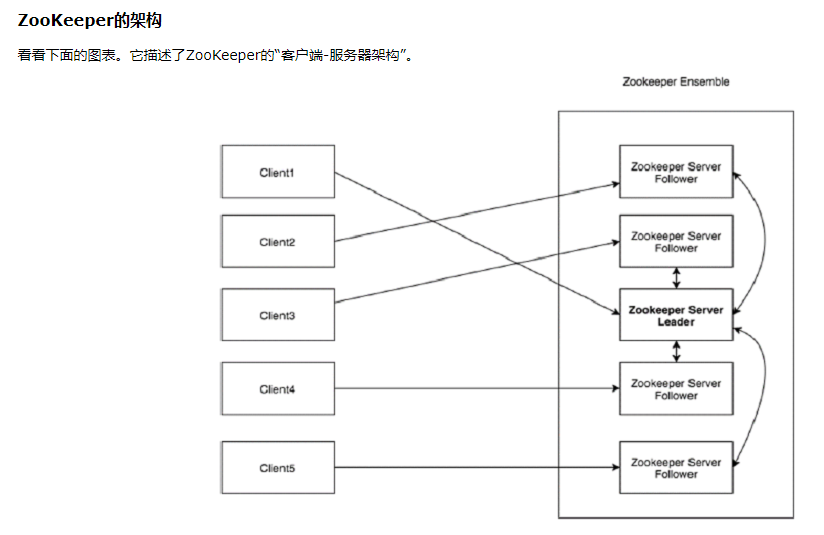
简单来讲，zookeeper可以充当一个服务注册表（Service Registry），让多个服务提供者形成一个集群，让服务消费者通过服务注册表获取具体的服务访问地址（ip+端口）去访问具体的服务提供者

具体来说，zookeeper就是个分布式文件系统，每当一个服务提供者部署后都要将自己的服务注册到zookeeper的某一路径上: /{service}/{version}/{ip:port}, 比如我们的HelloWorldService部署到两台机器，那么zookeeper上就会创建两条目录：分别为/HelloWorldService/1.0.0/100.19.20.01:16888 /HelloWorldService/1.0.0/100.19.20.02:16888。

zookeeper提供了“心跳检测”功能，它会定时向各个服务提供者发送一个请求（实际上建立的是一个 socket 长连接），如果长期没有响应，服务中心就认为该服务提供者已经“挂了”，并将其剔除，比如100.19.20.02这台机器如果宕机了，那么zookeeper上的路径就会只剩/HelloWorldService/1.0.0/100.19.20.01:16888。

服务消费者会去监听相应路径（/HelloWorldService/1.0.0），一旦路径上的数据有任务变化（增加或减少），zookeeper都会通知服务消费方服务提供者地址列表已经发生改变，从而进行更新。

更为重要的是zookeeper 与生俱来的容错容灾能力（比如leader选举），可以确保服务注册表的高可用性。



## RPC架构

## Mq消息队列

消息对列介绍：<http://blog.csdn.net/shaobingj126/article/details/50585035>

https://www.cnblogs.com/xiohao/p/4835594.html

# 项目

## 基于Openstack在线培训系统后台实现

### Openstack架构

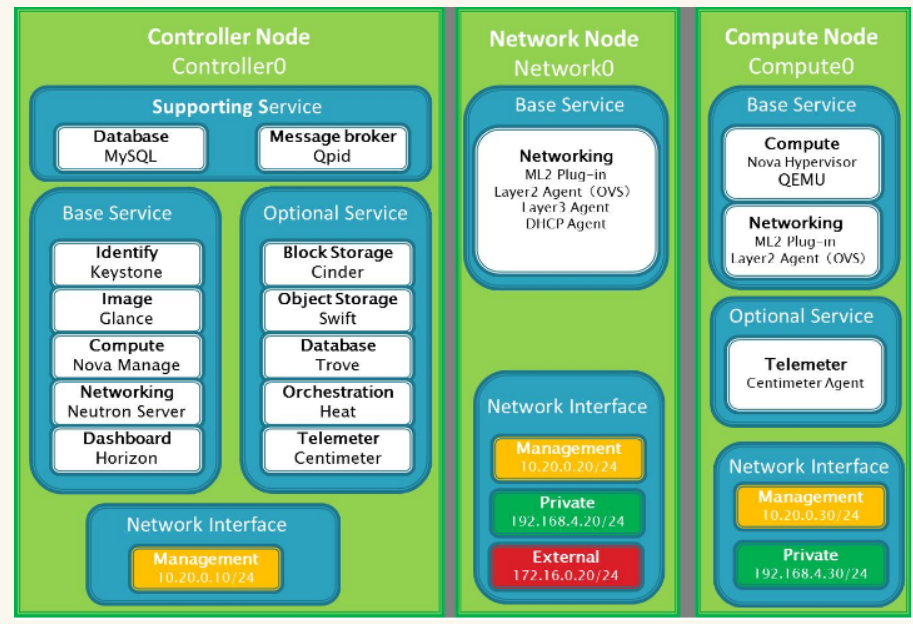
#### Openstack通用架构

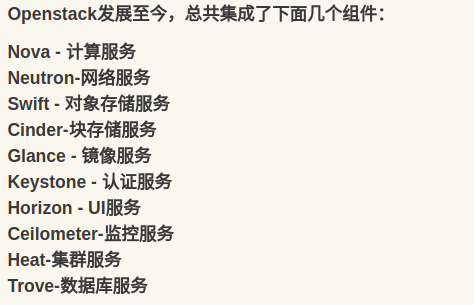
<http://www.cnblogs.com/CloudMan6/p/5427981.html>

Openstack属于SOA架构，SOA面向服务

OpenStack 本身是一个分布式系统，不但各个服务可以分布部署，服务中的组件也可以分布部署。

这种分布式特性让 OpenStack 具备极大的灵活性、伸缩性和高可用性。





#### OpenStack认证服务（Keystone）

Keystone为所有的OpenStack组件提供认证和访问策略服务，它依赖自身REST（基于Identity API）系统进行工作，主要对（但不限于）Swift、Glance、Nova等进行认证与授权。事实上，授权通过对动作消息来源者请求的合法性进行鉴定.

Keystone采用两种授权方式，一种基于用户名/密码，另一种基于令牌（Token）。

<https://blog.csdn.net/jmilk/article/details/51651154>

<https://www.cnblogs.com/klb561/p/8660264.html>

<https://www.cnblogs.com/charles1ee/p/6293387.html>

##### web项目中keystone配置

参考<https://blog.csdn.net/happyteafriends/article/details/47341721> + 项目源码

### Web项目中中虚拟机模块

源码horizon-master/openstack\_dashboard/dashboards/project/instances/views.py

### Openstack部署：

测试环境、生产环境 （计算节点、网络节点）：

操作系统：Ubuntu Server 16.04

测试环境：Mitaka版本 生产环境:Ocata版本。

<https://www.linuxidc.com/Linux/2016-09/135488.htm>

### openstack虚拟机调度算法

<https://blog.csdn.net/dylloveyou/article/details/80767211>

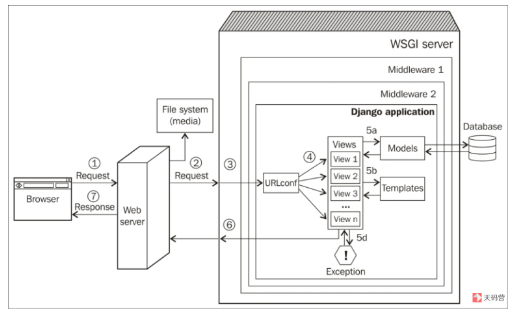
### Django

MTV模式，基于MVC变种

M 代表模型（Model），即数据存取层。 该层处理与数据相关的所有事务： 如何存取、如何验证有效性、包含哪些行为以及数据之间的关系等。

T 代表模板(Template)，即表现层。 该层处理与表现相关的决定： 如何在页面或其他类型文档中进行显示。

V 代表视图（View），即业务逻辑层。 该层包含存取模型及调取恰当模板的相关逻辑。 你可以把它看作模型与模板之间的桥梁。



前面的图片展示了从一个访客的浏览器到Django应用并返回的一个web请求的简单历程。如下是数字标识的路径：

1、浏览器发送请求（基本上时字节类型的字符串）到web服务器

2、web服务器把这个请求转交到一个WSGI,或者直接地文件系统额能够取出一个文件

3、不像web服务器那样，WSGI服务器可以直接运行Python应用。请求生成一个被称为environ的Ptyhon字典，而且，可以选择传递过去几个中间件的层，最终，达到Django应用。

4、URLconf中含有属于应用的urls.py选择一个视图处理基于请求的URL的那个请求，这个请求就已经变成了HttpRequest——一个Python字典对象。

5、被选择地那个试图通常要做下面所列出地一件或更多事情：

A、通过模型与数据库对话

B、使用模板渲染HTML或者任何格式化过地响应

C、返回一个纯文本响应（不被显示地）

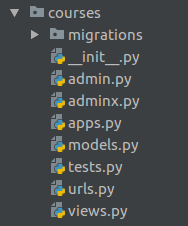
D、抛出一个异常

6、HttpResponse对象离开Django后，被渲染为一个字符串

7、在浏览器见到一个美化地，渲染后地web页面

大致流程如上，部分细节省略

**开发工具:Pycharm、ubuntu desktop 16.04、django 1.11**



一个app下的项目结构

Model.py下面建立数据库模型：



比如这样 课程模块中的 某个数据表

Views.py下面写业务逻辑。使用Python代码操作数据库，curd操作比如这样：

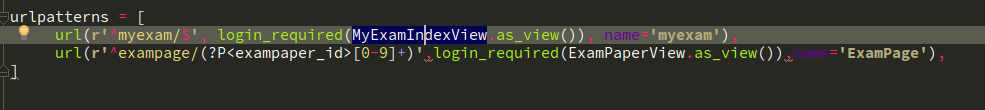


Get,filter等操作得到相关内容，使用render函数 将从数据库选取的函数返回到相关html



模板页面，这样在template文件夹下面的html，就可以操作这些数据了并进行前端的数据的修改、

Url.py。存取了 某个url路径下 该读取view.py下面的某个函数，这个函数或者class封装



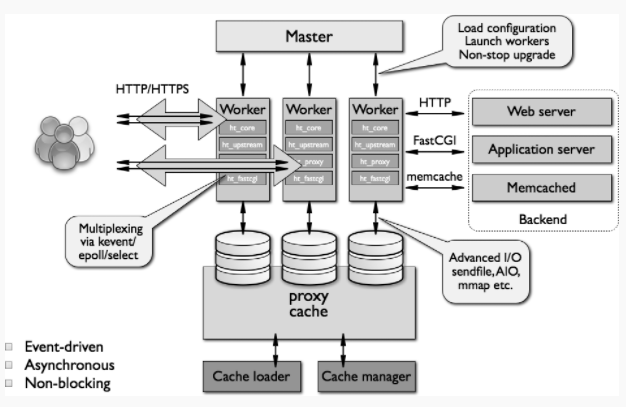
了对数据库操作以及要返回的页面。

## 基于tonado教学网站修改

## Nginx

nginx 是以多进程的方式来工作的。nginx 启动后，在 unix 系统中会以 daemon 的方式在后台运行，后台进程包含一个 master 进程和多个 worker 进程（你可以理解为工人和管理员）。这里就主要讲解 nginx 的多进程模式。

nginx 不会为每个连接派生进程或线程，而是由 worker 进程通过监听共享套接字接收新请求，并且使用高效的循环来处理数千个连接。Nginx 不使用仲裁器或分发器来分发连接，这个工作由操作系统内核机制完成。监听套接字在启动时就完成初始化，worker 进程通过这些套接字接收、读取请求和输出响应。



nginx 采用了**异步非阻塞**的方式来处理请求，也就是说，nginx 是可以同时处理成千上万个请求的.异步的概念是和同步相对的，也就是不同事件之间不是同时发生的。非阻塞的概念是和阻塞对应的，阻塞是事件按顺序执行，每一事件都要等待上一事件的完成，而非阻塞是如果事件没有准备好，这个事件可以直接返回，过一段时间再进行处理询问，这期间可以做其他事情。

### 反向代理

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。

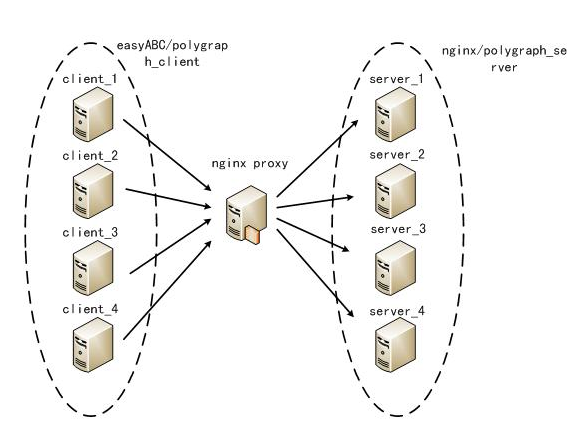
通常的代理服务器，只用于代理内部网络对Internet的连接请求，客户机必须指定代理服务器,并将本来要直接发送到Web服务器上的http请求发送到代理服务器中。当一个代理服务器能够代理外部网络上的主机，访问内部网络时，这种代理服务的方式称为反向代理服务。

|  |
| --- |
| location / { |
|  |  |
|  | include /etc/nginx/uwsgi\_params; |
|  |  |
|  | uwsgi\_pass 127.0.0.1:8001; |
|  | } |
|  |  |

在项目中用这个来监听上游 uwsgi服务器

如果需要配置负载均衡 那么得使用upstream模块

### 负载均衡框架图:



负载均衡算法:

1、轮询（默认）

每个请求按时间顺序逐一分配到不同的后端服务，如果后端某台服务器死机，自动剔除故障系统，使用户访问不受影响。

2、weight（轮询权值）

  weight的值越大分配到的访问概率越高，主要用于后端每台服务器性能不均衡的情况下。或者仅仅为在主从的情况下设置不同的权值，达到合理有效的地利用主机资源。

3、ip\_hash

 每个请求按访问IP的哈希结果分配，使来自同一个IP的访客固定访问一台后端服务器，并且可以有效解决动态网页存在的session共享问题。

1. fair

比 weight、ip\_hash更加智能的负载均衡算法，fair算法可以根据页面大小和加载时间长短智能地进行负载均衡，也就是根据后端服务器的响应时间 来分配请求，响应时间短的优先分配。Nginx本身不支持fair，如果需要这种调度算法，则必须安装upstream\_fair模块。

1. url\_hash

按访问的URL的哈希结果来分配请求，使每个URL定向到一台后端服务器，可以进一步提高后端缓存服务器的效率。Nginx本身不支持url\_hash，如果需要这种调度算法，则必须安装Nginx的hash软件包。

最小响应时间。 最小并发数。

### Tomcat集群保证session同步

如果使用Nginx 可以使用 ip\_hash的负载均衡

持久化session到mysql数据库 （不适合高并发）

使用redis共享数据库

可以使用memcache来分布式存储session

## 集群 分布式 负载均衡

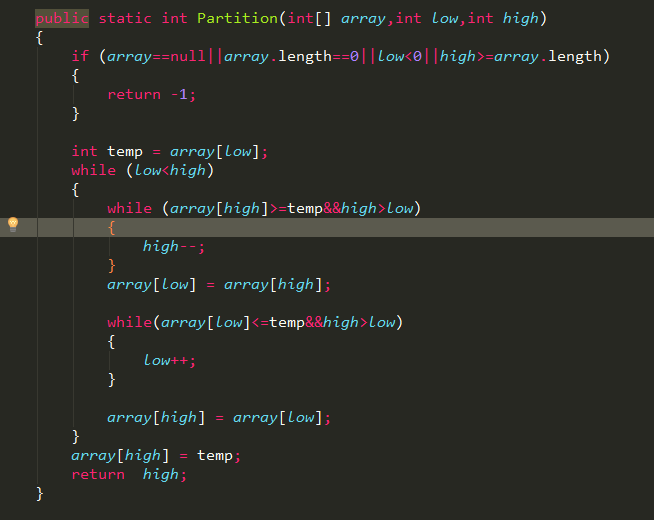
<https://www.cnblogs.com/xzwblog/p/7255364.html>

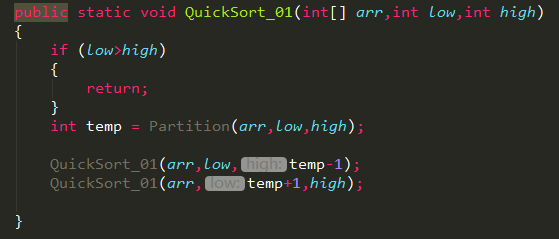
# 部分算法（面试手写）及智商题

## 排序算法



### 快速排序





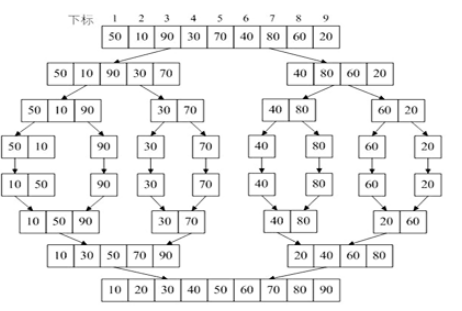
快速排序时间复杂度O(nlogn)，在数组已经是从小到大排序或者从大到小顺序的时候，快速排序最不适用，退化为冒泡排序的O（n2）。

<https://blog.csdn.net/hacker00011000/article/details/52176100>

快排如何优化？

快排非递归

### 归并排序



### 堆排

## Topk 找到最大的K的个数 或第k个数

## 二分查找 及其变种

## 有序链表a,b合并成c

## 二叉树两个结点最近公共祖先

## 两个链表第一个交点

## 判断链表有环 以及环的大小 以及环的入口

## 只要把剑指offer和leetcode（like 100 、top 150排行榜）刷完，面试的手撕代码基本都没问题了。