<http://www.jianshu.com/p/1b2f63a45476>

待学习的知识点

1.双轴快排

2.秒杀的实现

3.线程同步的实现方式

4.网络各个协议

5.索引

MySQL数据库支持多种索引类型，如BTree索引，哈希索引，全文索引等等

# Java部分

## Java基础

### 线程池（与六部分一起看）

（1） 线程池的实现原理

为了更好的控制多线程，JDK提供了一套线程框架Executor，帮助开发人员有效进行线程控制，他们都在java.util.concurrent包中。是JDK并发包的核心。非常重要的类：Executors。它是一个线程工厂，可以创建特定功能的线程。

Executors 创建线程池的方法：

**newFixedThreadPool()**方法，返回一个固定数量的线程池，该方法的线程数始终不变，当有一个任务提交时，若线程中空闲，则立即执行，若没有，则会被暂缓在一个任务队列中等待有空闲的线程去执行。

**newSingleThreadExecutor()**方法，创建一个线程池。若空闲，则执行，若没有空闲线程则暂缓在任务队列中。

**newCachedThreadPool()**方法，返回可根据实际情况调整线程个数的线程池。不限制最大线程数量，若用空闲的线程，则执行任务，若无任务则不创建线程。并且每一个空闲线程会在60秒后自动回收。

**newScheduledThreadPool()**方法，返回**SchededExecutorService**对象。 但该方法可以指定线程数量。 这里面的每一个线程都可以实现定时器的功能。

以上四个方法底层都是调用ThreadPoolExecutor的构造器来创建线程池。

public ThreadPoolExecutor(

               int maximumPoolSize,

               long keepAliveTime,

               TimeUnit unit,

               BlockingQueue<Runnable> workQueue,

               ThreadFactory threadFactory,

               RejectedExecutionHandler handler

){...}

（2） 线程池中的coreNum、maxNum和largestNum

（3） 在不同的业务场景中，线程池参数如何设置

### 锁的实现

线程安全的概念：当多个线程访问同一个类（或属性或方法）的时候，所得到的执行效果都是正确的，就说这个类（或属性或方法）是线程安全的。

锁竞争的基本概念：当线程要执行临界区（也称互斥区）的时候，要首先拿到临界区的锁对象，然后才可以执行；多个线程同时访问一个临界区的时候，就会发生锁竞争的情况。发生锁竞争是非常耗资源的（并发量很大的时候，甚至直接当机），应该尽量避免。

synchronized概念：同步，只有共享资源才需要同步。

异步概念：相互不影响。

脏读：写的时候有读的操作，就会产生脏读。（读操作和写操作都需要上锁同步）

1. synchronized和ReentrantLock实现原理
2. 读写锁
3. volatile关键字作用、实现原理
4. 锁的可重入性（详见：http://www.jianshu.com/p/007bd7029faf）
5. 对象监视器

并发编程中有三个概念：原子性、可见性、有序性。在这三个特性中，volatile关键字均可以发挥一些作用。

原子性：如转账。

可见性：

有序性：

1. Object.wait/notify实现机制
2. Unsafe类
3. 死锁和活锁
4. 并发相关面试题（http://www.cnblogs.com/binyue/p/4369985.html）

### ConcurrentHashMap

1. HashMap实现原理
2. ConcurrentHashMap作用、实现原理、扩容实现、其size方法的实现，1.7与1.8的不同实现

### LinkedHashMap

1. java LRU的缓存实现原理

最近最少使用缓存的回收（LRU）

怎么实现一个最近最少使用（LRU）的缓存。缓存可以通过哈希表来实现，然而为这个缓存增加大小限制会变成另一个有意思的问题。

为了实现缓存回收，我们需要很容易做到：

1.查询出最近最晚使用的项 2.给最近使用的项做一个标记

链表可以实现这两个操作，检测最近最少使用的项只需要返回链表的尾部。标记一项为最近使用的项只需要从当前位置移除，然后将该项放置到头部。 比较困难的事情是怎么快速的在链表中找到该项。

哈希表的帮助

看一下我们工具箱中的数据结构，哈希表可以在(消耗)常量的时间内索引到某个对象。如果我们创建一个形如Key-链表结点的哈希表，我们就能够在常量时间内找到最近使用的结点。 更甚的是，我们也能够在常量时间内判断结点是否存在（或不存在）；找到这个结点后，我们就能将这个结点移动到链表的最前端，标记为最近使用的项了。

Java的捷径

据我所知，很少有一种编程语言的标准库中有通用的数据结构能提供上述功能。这是一种混合的数据结构，我们需要在哈希表的基础上建立一个链表。 但是Java已经为我们提供了这种形式的数据结构LinkedHashMap！它甚至提供可覆盖回收策略的方法（见removeEldestEntry的文档）。 唯一需要我们注意的事情是，该链表的顺序是插入的顺序，而不是访问的顺序。但是，有一个构造函数提供了一个选项，可以使用访问的顺序。

该方法是最简便的实现方法。

代码如下：

|  |
| --- |
| **public** **class** LRUCache<K, V> **extends** LinkedHashMap<K, V> {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **int** cacheSize;  **public** LRUCache(**int** cacheSize) {  **super**(16, 0.75f, **true**);  **this**.cacheSize = cacheSize;  }  **protected** **boolean** removeEldestEntry(Map.Entry<K, V> eldest) {  **return** size() >= cacheSize;  }  } |

另一种实现方法（不适用LinkedHashMap）如下：

|  |
| --- |
| **public** **class** LRUCache2 {  /\*双向链表\*/  **class** Node {  Node pre;  Node next;  Integer key;  Integer val;  Node(Integer k, Integer v) {  key = k;  val = v;  }  }  Map<Integer, Node> map = **new** HashMap<>();  Node head; // head是链表的最年老的节点  Node tail; // tail是链表的最年轻的节点  **int** cap;  **public** LRUCache2(**int** capacity) {  cap = capacity;  head = **new** Node(**null**, **null**);  tail = **new** Node(**null**, **null**);  head.next = tail;  tail.pre = head;  }  **public** **int** get(**int** key) {  Node n = map.get(key);  **if** (n != **null**) {  // 把n从链表中摘下来  n.pre.next = n.next;  n.next.pre = n.pre;  // 把n放到链表的尾部  appendTail(n);  **return** n.val;  }  **return** -1;  }  **public** **void** set(**int** key, **int** value) {  Node n = map.get(key);  **if** (n != **null**) {  n.val = value;  map.put(key, n);  // 把n从链表中摘下来  n.pre.next = n.next;  n.next.pre = n.pre;  // 把n放到链表的尾部  appendTail(n);  **return**;  }  **if** (map.size() == cap) {  Node tmp = head.next;  // 把tmp从链表上摘下来  head.next = head.next.next;  head.next.pre = head;  // 把tmp删掉  map.remove(tmp.key);  }  // n为空的时候  n = **new** Node(key, value);  appendTail(n);  map.put(key, n);  }  **private** **void** appendTail(Node n) {  n.next = tail;  n.pre = tail.pre;  tail.pre.next = n;  tail.pre = n;  }  } |

### 集合框架

HashMap和TreeMap区别

### 枚举

### 正则表达式

### 某些JDK版本的新特性

1. jdk8的新特性

（详见：https://www.ibm.com/developerworks/cn/java/j-lo-jdk8newfeature/index.html）

1. **函数式接口**

Java 8 引入的一个核心概念是函数式接口（Functional Interfaces）。通过在接口里面添加一个抽象方法，这些方法可以直接从接口中运行。如果一个接口定义个唯一一个抽象方法，那么这个接口就成为函数式接口。同时，引入了一个新的注解：@FunctionalInterface。可以把他它放在一个接口前，表示这个接口是一个函数式接口。这个注解是非必须的，只要接口只包含一个方法的接口，虚拟机会自动判断，不过最好在接口上使用注解 @FunctionalInterface 进行声明。在接口中添加了 @FunctionalInterface 的接口，只允许有一个抽象方法，否则编译器也会报错。

java.lang.Runnable 就是一个函数式接口。

|  |
| --- |
| @FunctionalInterface  **public** **interface** Runnable {  **public** **abstract** **void** run();  } |

1. **Lambda 表达式**

函数式接口的重要属性是：我们能够使用 Lambda 实例化它们，Lambda 表达式让你能够将函数作为方法参数，或者将代码作为数据对待。Lambda 表达式的引入给开发者带来了不少优点：在 Java 8 之前，匿名内部类，监听器和事件处理器的使用都显得很冗长，代码可读性很差，Lambda 表达式的应用则使代码变得更加紧凑，可读性增强；Lambda 表达式使并行操作大集合变得很方便，可以充分发挥多核 CPU 的优势，更易于为多核处理器编写代码；

Lambda 表达式由三个部分组成：第一部分为一个括号内用逗号分隔的形式参数，参数是函数式接口里面方法的参数；第二部分为一个箭头符号：->；第三部分为方法体，可以是表达式和代码块。语法如下：

1. 方法体为表达式，该表达式的值作为返回值返回。

|  |
| --- |
| (parameters) -> expression |

1. 方法体为代码块，必须用 {} 来包裹起来，且需要一个 return 返回值，但若函数式接口里面方法返回值是 void，则无需返回值。

|  |
| --- |
| (parameters) -> { statements; } |

下面是用匿名内部类的代码：

|  |
| --- |
| button.addActionListener(**new** ActionListener() {  @Override  **public** **void** actionPerformed(ActionEvent e) {  System.***out***.print("Helllo Lambda in actionPerformed");  }  });  // 下面是使用 Lambda 表达式后：  button.addActionListener(  // actionPerformed 有一个参数 e 传入，所以用 (ActionEvent e)  (ActionEvent e) -> System.***out***.print("Helllo Lambda in actionPerformed")); |

上面是方法体包含了参数传入 (ActionEvent e)，如果没有参数则只需 ( )，例如 Thread 中的 run 方法就没有参数传入，当它使用 Lambda 表达式后：

|  |
| --- |
| Thread t = **new** Thread(  // run 没有参数传入，所以用 (), 后面用 {} 包起方法体  () -> {  System.***out***.println("Hello from a thread in run");  });  // 通过上面两个代码的比较可以发现使用 Lambda 表达式可以简化代码，并提高代码的可读性。  // 为了进一步简化 Lambda 表达式，可以使用方法引用。例如，下面三种分别是使用内部类，使用 Lambda  // 表示式和使用方法引用方式的比较：  // 1. 使用内部类  Function<Integer, String> f = **new** Function<Integer, String>() {  @Override  **public** String apply(Integer t) {  **return** **null**;  }  };  // 2. 使用 Lambda 表达式  Function<Integer, String> f2 = (k) -> String.*valueOf*(k);  // 3. 使用方法引用的方式  Function<Integer, String> f1 = String::*valueOf*; |

要使用 Lambda 表达式，需要定义一个函数式接口，这样往往会让程序充斥着过量的仅为 Lambda 表达式服务的函数式接口。为了减少这样过量的函数式接口，Java 8 在 java.util.function 中增加了不少新的函数式通用接口。例如：

Function<T, R>：将 T 作为输入，返回 R 作为输出，他还包含了和其他函数组合的默认方法。

Predicate<T> ：将 T 作为输入，返回一个布尔值作为输出，该接口包含多种默认方法来将 Predicate 组合成其他复杂的逻辑（与、或、非）。

Consumer<T> ：将 T 作为输入，不返回任何内容，表示在单个参数上的操作。

|  |
| --- |
| //例如，People 类中有一个方法 getMaleList 需要获取男性的列表，这里需要定义一个函数式接口 PersonInterface：  **interface** PersonInterface {  **public** **boolean** test(Person person);  }  **public** **class** People {  **private** List<Person> persons = **new** ArrayList<Person>();  **public** List<Person> getMaleList(PersonInterface filter) {  List<Person> res = **new** ArrayList<Person>();  persons.forEach((Person person) -> {  **if** (filter.test(person)) {// 调用 PersonInterface 的方法  res.add(person);  }  });  **return** res;  }  }  // 为了去除 PersonInterface 这个函数式接口，  // 可以用通用函数式接口 Predicate 替代如下：  **class** People {  **private** List<Person> persons = **new** ArrayList<Person>();  **public** List<Person> getMaleList(Predicate<Person> predicate) {  List<Person> res = **new** ArrayList<Person>();  persons.forEach(person -> {  **if** (predicate.test(person)) {// 调用 Predicate 的抽象方法 test  res.add(person);  }  });  **return** res;  }  } |

## JVM

### 内存模型JMM

（1）内存模型，各个部分的名称，作用，注意事项，参数等

（2）对象在内存中的初始化过程

### 垃圾回收

（1）实现算法

（2）强软弱虚引用

### 虚拟机调优

### JVM启动流程

### Object.finalize

（1） Object类的finalize方法的实现原理

### JVM的一些命令

（详见：http://blog.csdn.net/fenglibing/article/details/6411953）  
（1）jmap命令

（2）jps命令

（3）jstat命令

### 类加载机制

（详见：http://wiki.jikexueyuan.com/project/java-vm/class-loading-mechanism.html）

## 设计模式

### 普通设计模式（23种）

创建型模式

单例模式

构建模式

原型模式

抽象工厂模式

工厂方法模式

行为型模式

结构型模式

### 六大原则

### 多线程设计模式

### 其他

(1) 对象适配器模式满足了哪些原则？

## 反射

### 反射（性能，冲破private等）

### 序列化

### 克隆

### 内省机制

## 并发编程

### 线程（概念、状态、创建方式、临界区）

（1）概念，线程和进程的区别（参见操作系统部分）

**线程的概念：线程是程序中的一个执行流，一个执行流是由CPU运行程序代码并操纵程序的数据形成的。JAVA中的现成模型就是一个CPU、 程序代码和数据的封装体。**（参考自 张杨著Java并行程序设计，清华大学出版社，13页）

（2）线程的状态

**创建、就绪、运行、阻塞、终止 五种状态**

**创建：处于创建状态的线程有自己的内存空间，但是处于该状态的线程还没有运行，没有被激活。**

**就绪：处于创建状态的的线程调用start方法启动后，就会进入就绪状态，此时，该线程已经拥有了运行所需的所有条件，将进入线程队列排队等待CPU调度。**

**运行：当就绪的线程获得CPU的时间片后，该线程就转换为运行状态。运行状态的线程执行run方法中定义的操作。直到该方法调用结束进入终止状态，或者运行时由于缺乏运行必备的资源而进入阻塞状态。处于运行状态的线程也不是一直占用CPU的，运行（非阻塞）的线程可能会与就绪状态来回切换。**

**阻塞：在某些情况（1.共享资源被占用、2.等待I/O操作、3.调用了wait、sleep或suspend方法、4.尝试获得锁而该锁被其他线程持有）下，正在运行的线程会让出正在使用的CPU资源，进入阻塞状态；当引起阻塞的原因消除后，线程将进入就绪状态等待继续执行。**

**终止：达到终止状态的原因如下：1.线程run方法执行结束；2.线程通过某些方法（如destroy()）被提前终止；3.在run方法的执行期间发生了未捕获的异常；4.程序终止操作（如调用System.exit()）。**

（参考自 张杨著Java并行程序设计，清华大学出版社，13页~14页）

（3）创建方式

三种创建方式：

1.继承Thread类，重写run方法，启动的时候调用start方法；

2.实现Runnable接口，重写run方法，启动的时候new Thread(Runnable实例).start()；

3.实现Callable接口，定义返回值类型，重写call方法，启动方法为，将实例放入FutureTask实例中去，再将FutureTask实例放入一个Thread对象中去，使用Thread的start方法启动。好处：具有返回值（FutureTask可以通过调用get方法拿到异步任务的返回值）

（4）临界区

临界区就是在同一时刻只能有一个任务访问的代码区。在[**Java**](http://lib.csdn.net/base/java)中通常使用下面的方式来实现：

synchronized(syncObject) { //critical section }

可以使用Lock对象来实现临界区

**每个java对象都隐含有一把锁**。

使用临界区的执行效率要比使用synchronized方法的执行效率要高，因为其锁粒度更小。

（5）监视器

监视器（Monitor）有如下特征：

**一个监视器是只有一个私有属性的类；**

**每个监视器类的对象实力都有一个相关联的锁。这个锁将对对象的所有方法加锁。如：obj.method()方法调用开始时，自动获取obj对象的锁，方法调用结束时解锁。**

**Java中每个对象都有一把隐式的锁。**

（6）阻塞和非阻塞

前提： **如果线程请求某一种资源而得不到响应**，线程可以采用多种方式来决定接下来要采取的动作。根据动作的不同，分为阻塞和非阻塞

**非阻塞**：**若线程采用不断尝试的方式：在每次请求资源得不到满足的情况下，下次仍然继续请求，知道请求获得满足。这种方式就是非阻塞的。**

**阻塞**：**线程并非一直等待，而是被阻塞，这样可以让出CPU的资源来执行一些其他的操作。等待一段时间后，再去尝试获取资源。**

目前采用阻塞和非阻塞结合的方式：**首先以非阻塞的方式去尝试一定的次数，如果限定次数范围内，线程可以获得资源则线程继续运行，否则线程进入阻塞状态，等待一段时间后在尝试。**

### 锁

（1）锁机制的同步锁、 可重入锁和 读写锁以及这三种锁机制的比较

同步锁：使用同步方法或同步块，synchronized关键字修饰。同步方法的效率比同步块的低。

可重入锁：是一种无阻塞的同步机制，在java.util.concurrent.locks包下，ReentranLock类。比同步锁功能强大，有很多功能。支持公平和非公平策略。

读写锁：ReentrantReadWriteLock类，具有：锁的获取顺序（公平和非公平）、锁降级（允许将写锁降级为读锁，降级后允许更大程度的并发）、支持中断（读和写锁都支持中断，即如果锁整被其他线程持有，则可以被中断请求）

三种锁机制的比较：

同步锁是一种互斥锁，优点：形式简单使用方便，缺点：依赖于隐藏在对象后的内置监视器，不直观。

可重入锁是一种互斥锁，优点：在同步锁的基础上扩展了许多功能，如：非阻塞枷锁操作、在尝试获取锁时可中断、测试锁是否正在被持有、锁的获取顺序等。

读写锁出了提供可重入锁的一些特性外，还包含：把锁分为读锁和写锁，允许更大程度的并发

（2）锁的不足

**锁竞争问题**：多个线程试图访问被同一互斥锁保护的共享资源，并且此时有一个或多个线程等待获得该锁。此时就产生了锁竞争。

锁竞争问题是多核时代容易发生的问题。会导致只有一个CPU在运行，其他的CPU都处于空闲状态。解决办法：目前无法完全解决，有种方式是STM（软件事务型内存），其缺点是开销较大，且容易导致活锁。STM无法完全取代锁。

### 线程间通信

### 线程同步障栅

### 线程执行器

### Fork/Join框架

## 网络编程

## NIO

## 大数据

### Hadoop原理、HDFS实现原理、MapReduce原理、Yarn原理

### Hive、HBase、Storm、Flume、Zookeeper

### 面试题

在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。

答：采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存2^32 \* 2 bit=1 GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看bitmap，把对应位是01的整数输出即可。

## JavaWeb

### Session生命周期

（http://book.51cto.com/art/200912/169766.htm）

### HTTP状态码

**1**消息  
**2**成功  
**200** OK  
请求已成功，请求所希望的响应头或数据体将随此响应返回。  
**202** Accepted  
服务器已接受请求，但尚未处理。正如它可能被拒绝一样，最终该请求可能会也可能不会被执行。在异步操作的场合下，没有比发送这个状态码更方便的做法了。  
**3**重定向  
**4**请求错误  
**400** Bad Request  
1、语义有误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求。  
2、请求参数有误。  
**403** Forbidden  
服务器已经理解请求，但是拒绝执行它。与401响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。如果这不是一个 HEAD 请求，而且服务器希望能够讲清楚为何请求不能被执行，那么就应该在实体内描述拒绝的原因。当然服务器也可以返回一个404响应，假如它不希望让客户端获得任何信息。  
**404** Not Found  
请求失败，请求所希望得到的资源未被在服务器上发现。没有信息能够告诉用户这个状况到底是暂时的还是永久的。假如服务器知道情况的话，应当使用410状态码来告知旧资源因为某些内部的配置机制问题，已经永久的不可用，而且没有任何可以跳转的地址。404这个状态码被广泛应用于当服务器不想揭示到底为何请求被拒绝或者没有其他适合的响应可用的情况下。出现这个错误的最有可能的原因是服务器端没有这个页面。  
**408** Request Timeout  
请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送。客户端可以随时再次提交这一请求而无需进行任何更改。  
**5**、**6** 服务器错误  
**500** Internal Server Error  
服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器端的源代码出现错误时出现。  
**502** Bad Gateway  
作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。  
**503** Service Unavailable  
由于临时的服务器维护或者过载，服务器当前无法处理请求。这个状况是临时的，并且将在一段时间以后恢复。如果能够预计延迟时间，那么响应中可以包含一个 Retry-After 头用以标明这个延迟时间。如果没有给出这个 Retry-After 信息，那么客户端应当以处理500响应的方式处理它。  
注意：503状态码的存在并不意味着服务器在过载的时候必须使用它。某些服务器只不过是希望拒绝客户端的连接。  
**600** Unparseable Response Headers  
源站没有返回响应头部，只返回实体内容

## Struts2

## Hibernate

## Spring

## MyBatis

## SpringMVC

## 最佳实践

# 算法和数据结构

（涉及到的代码部分要求会默写）

## 链表、栈（及其应用，如表达式求值，走迷宫等）、队列（及其应用，如杨辉三角等）

## 二叉树（递归及非递归遍历、层次遍历），平衡二叉树、红黑树

### Treap（树堆）

### Trie树

## 图 （广度、深度、递归以及非递归）最小生成树（普里姆、克鲁斯卡尔），最短路径（地杰斯特拉，弗洛伊德）、拓扑排序

## 排序（插入、交换、选择、归并，外部）

## 查找（二分等）

## 常见算法的时空复杂度，小知识点（二叉树和森林的对应关系、离散数学关于图论的计算公式等）

## 串的模式匹配（KMP）

## 动态规划思想

## 贪心思想

## 分治思想

## 其他

### n个元素进栈，共有几种出栈顺序？

答：C(2n, n) / (n+1)种 （详见：http://blog.csdn.net/zyearn/article/details/778716）

### （三七互娱面试真题）已知一个单链表中的某一个节点，不知道该链表的头指针，请说出如何删除该节点。

答：将该节点的下一个节点的value值赋给当前节点的value，然后删除该节点的下一个节点即可。

# 数据库部分

## SQL基础

1. 例题
2. 【关于执行顺序】

<https://www.nowcoder.com/questionTerminal/7abd397a22054bbb99e2b3982ff81d01>  
对于满足SQL92标准的SQL语句：SELECT foo,count(foo) FROM pokes WHERE foo>10 GROUP BY foo HAVING ORDER BY foo，其执行的顺序应该为（ ）

1. FROM->WHERE->GROUP BY->HAVING->SELECT->ORDER BY
2. FROM->GROUP BY->WHERE->HAVING->SELECT->ORDER BY
3. FROM->WHERE->GROUP BY->HAVING->ORDER BY->SELECT
4. FROM->WHERE->ORDER BY->GROUP BY->HAVING->SELECT

解析：

标准的 SQL 的解析顺序为:

(1).FROM 子句, 组装来自不同数据源的数据

(2).WHERE 子句, 基于指定的条件对记录进行筛选

(3).GROUP BY 子句, 将数据划分为多个分组

(4).使用聚合函数进行计算

(5).使用 HAVING 子句筛选分组

(6).计算Select所有的表达式

(7).使用 ORDER BY 对结果集进行排序

答案：A

## 事务

## 锁

## NoSQL

## MySQL和Redis

## 三大范式

# 网络部分

## OSI和TCP/IP架构，各层职能，所属协议

## 流量控制、三次握手，四次挥手、TCP和UDP区别、HTTP、HTTPS的五次握手

### HTTP与TCP/IP区别

TPC/IP协议是传输层协议，主要解决数据如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据。WEB使用HTTP协议作应用层协议，以封装HTTP 文本信息，然后使用TCP/IP做传输层协议将它发到网络上。

下面的图表试图显示不同的TCP/IP和其他的协议在最初OSI（Open System Interconnect）模型中的位置：



### CA证书是什么

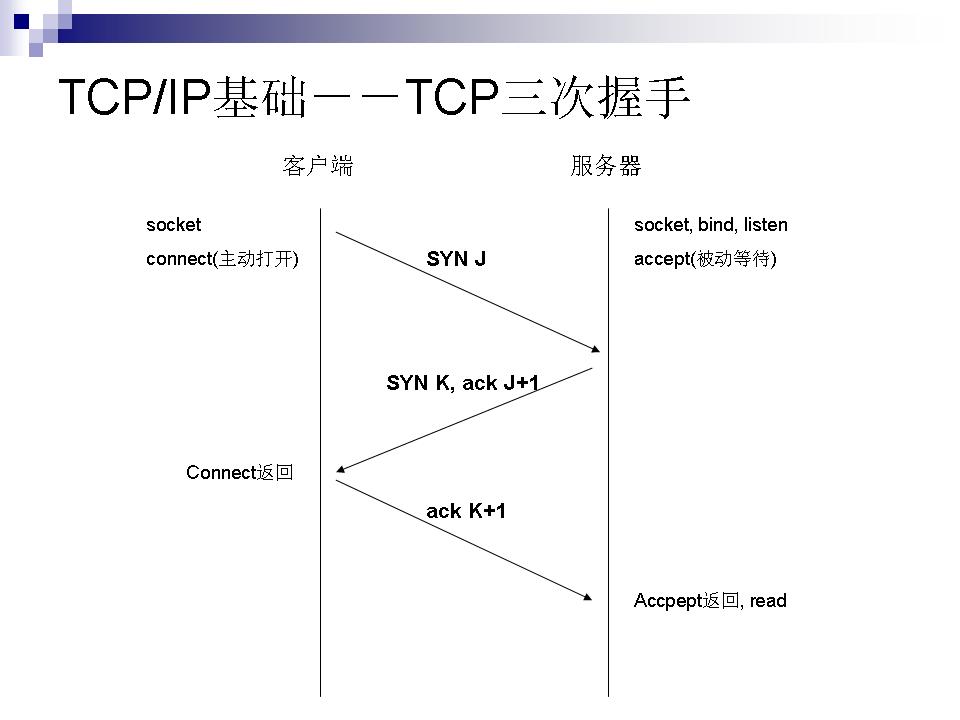
CA（Certificate Authority）是负责管理和签发证书的第三方权威机构，是所有行业和公众都信任的、认可的。

CA证书，就是CA颁发的证书，可用于验证网站是否可信（针对HTTPS）、验证某文件是否可信（是否被篡改）等，也可以用一个证书来证明另一个证书是真实可信，最顶级的证书称为根证书。除了根证书（自己证明自己是可靠），其它证书都要依靠上一级的证书，来证明自己。

### HTTP三次握手

HTTP（HyperText Transfer Protocol)超文本传输协议是互联网上应用最为广泛的一种网络协议。由于信息是明文传输，所以被认为是不安全的。而关于HTTP的三次握手，其实就是使用三次TCP握手确认建立一个HTTP连接。

如下图所示，SYN（synchronous）是TCP/IP建立连接时使用的握手信号、Sequence number（序列号）、Acknowledge number（确认号码），三个箭头指向就代表三次握手，完成三次握手，客户端与服务器开始传送数据。



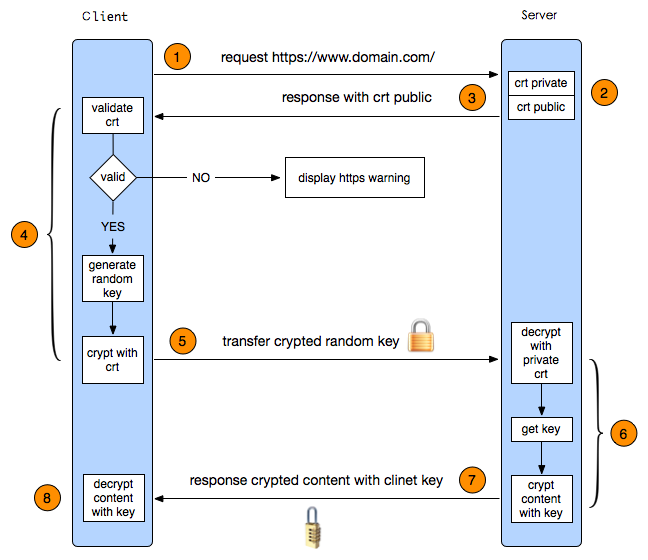
第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

### HTTPS握手过程

HTTPS在HTTP的基础上加入了SSL协议，SSL依靠证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。具体是如何进行加密，解密，验证的，且看下图，下面的称为一次握手。



**1. 客户端发起HTTPS请求**

**2. 服务端的配置**

采用HTTPS协议的服务器必须要有一套数字证书，可以是自己制作或者CA证书。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用CA证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。公钥给别人加密使用，私钥给自己解密使用。

**3. 传送证书**

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等。

**4. 客户端解析证书**

这部分工作是有客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随即值，然后用证书对该随机值进行加密。

**5. 传送加密信息**

这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

**6. 服务段解密信息**

服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

**7. 传输加密后的信息**

这部分信息是服务段用私钥加密后的信息，可以在客户端被还原。

**8. 客户端解密信息**

客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。

PS: 整个握手过程第三方即使监听到了数据，也束手无策。

### 为什么HTTPS是安全的

在HTTPS握手的第四步中，如果站点的证书是不受信任的，会显示出现下面确认界面，确认了网站的真实性。另外第六和八步，使用客户端私钥加密解密，保证了数据传输的安全。



### HTTPS和HTTP的区别

1. https协议需要到ca申请证书或自制证书。

2. http的信息是明文传输，https则是具有安全性的ssl加密。

3. http是直接与TCP进行数据传输，而https是经过一层SSL（OSI表示层），用的端口也不一样，前者是80（需要国内备案），后者是443。

4. http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

**注意https加密是在传输层**

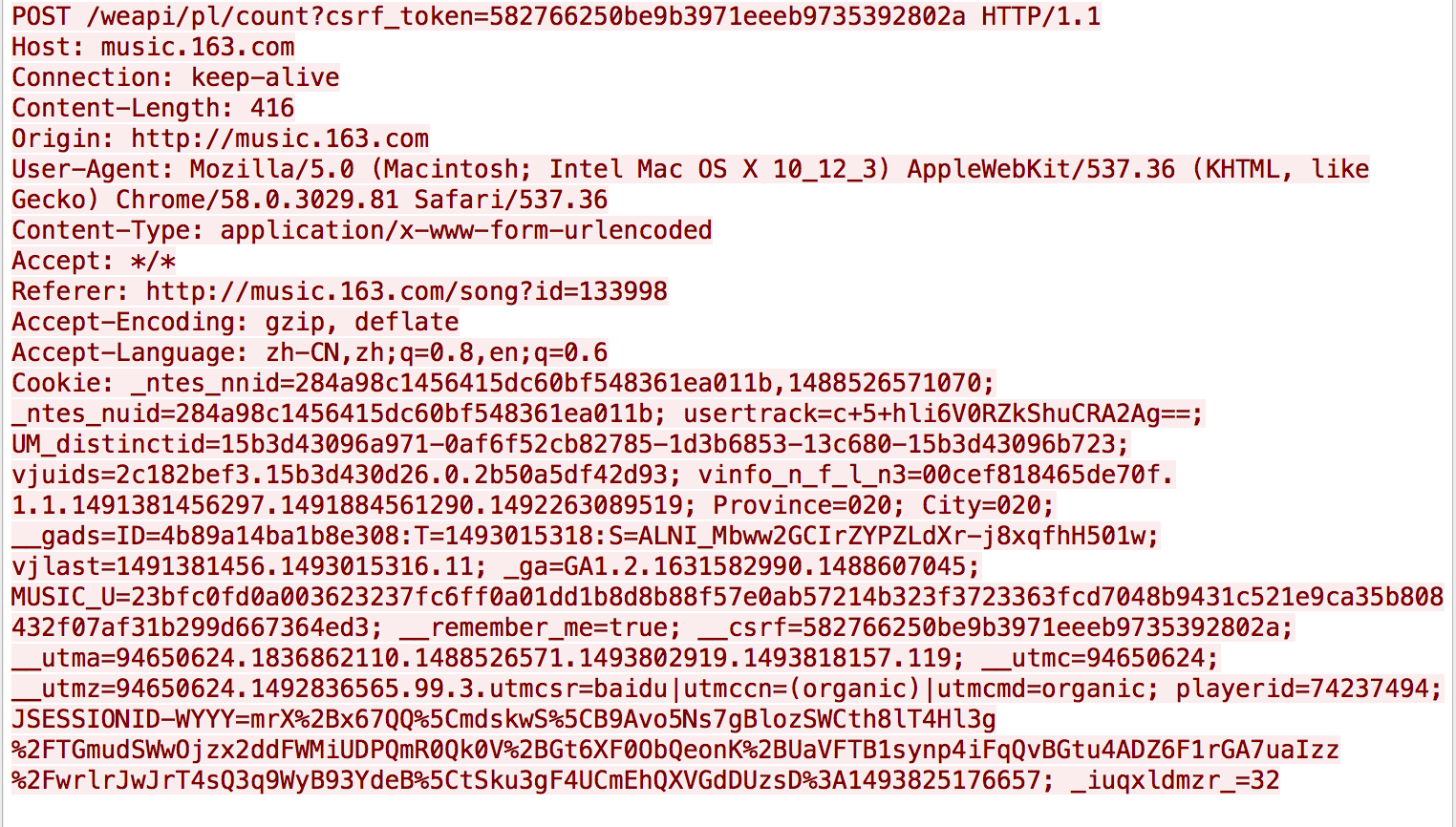
https报文在被包装成tcp报文的时候完成加密的过程，无论是https的**header域**也好，body域也罢都是会被加密的。

当使用**tcpdump或者wireshark**之类的tcp层工具抓包，获取是加密的内容，而如果用应用层抓包，使用**Charels(Mac)、Fildder(Windows)**抓包工具，那当然看到是明文的。

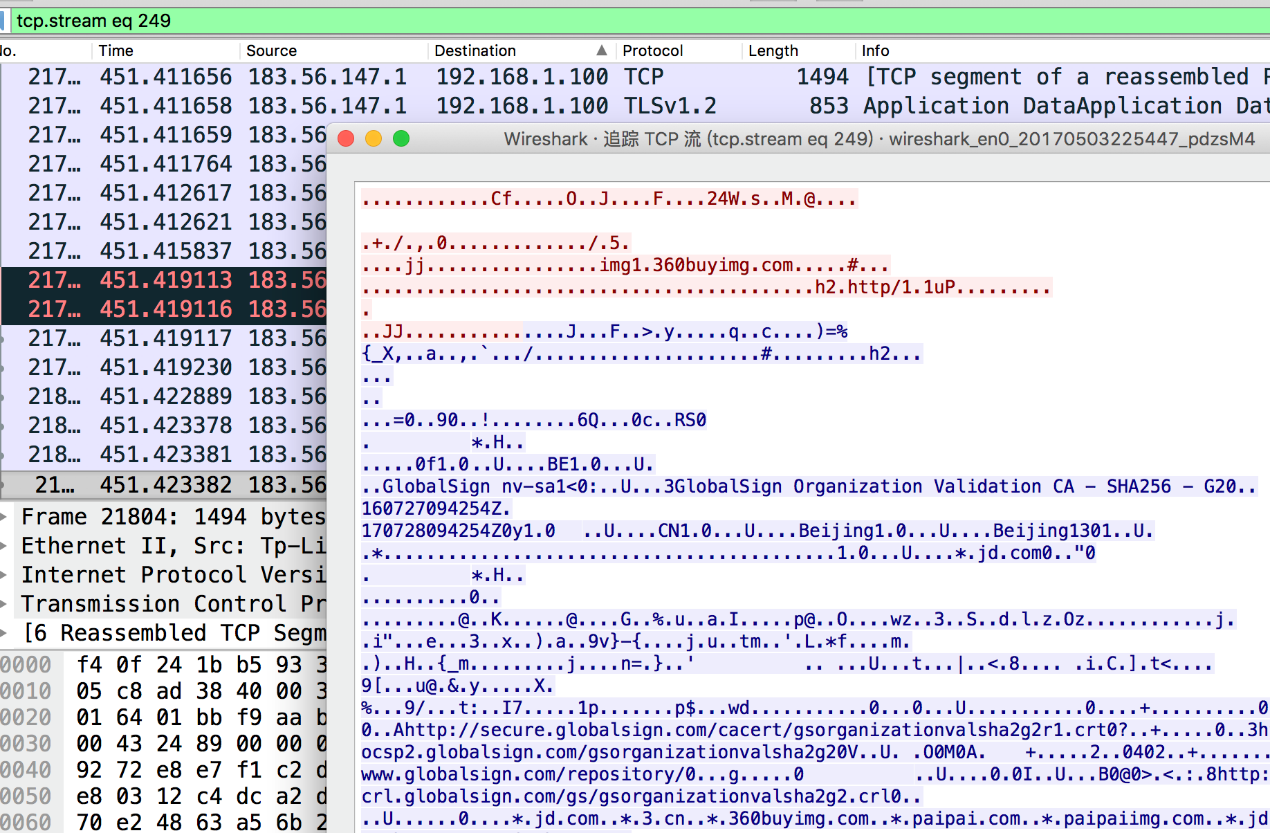
PS：HTTPS本身就是为了网络的传输安全。

例子，使用wireshark抓包：

http，可以看到抓到是明文的：



https，可以看到抓到是密文的：



### HTTPS一般使用的加密与HASH算法

非对称加密算法：RSA，DSA/DSS

对称加密算法：AES，RC4，3DES

HASH算法：MD5，SHA1，SHA256

## 其他面试题

(1) 如何将UDP协议修改成TCP协议？

答：修改报文的首部

# 操作系统部分

## Linux操作系统常用命令

1. 统计文件中出现次数最多的前10个单词

答：cat words.txt | sort | uniq -c | sort -kl, lnr | head -10

(详见：http://blog.sina.com.cn/s/blog\_5dce657a01012ddi.html)

2. linux下，系统资源占用相关命令？

答：top、free、uptime （详见：http://blog.51yip.com/linux/1162.html  以及  http://www.jb51.net/LINUXjishu/95640.html）

3. linux使用的进程间通信方式？

答：1.管道（pipe）、流管道（s\_pipe）、有名管道（FIFO）

       2.信号（signal）

       3.消息队列

       4.共享内存

       5.信号量

       6.套接字（socket）

       （详见：http://blog.csdn.net/gatieme/article/details/50908749）

4. 存储相关。磁盘调度算法有哪些？

答：先来先服务（FCFS）、最短寻道时间优先算法（SSTF）、扫描算法（SCAN）、循环扫描算法（CSCAN）。（详见：百度百科：磁盘调度算法）

5. linux阻塞与非阻塞，同步与异步、I/O模型

http://www.cnblogs.com/pang1567/p/4012834.html

# 软件工程部分

# 组成原理部分

# 机器学习部分

# 面经汇总

左鹏的涂鸦移动1、 2面：https://www.nowcoder.com/discuss/38363

20个高级Java面试题汇总

<http://www.imooc.com/article/20702>

## 三七互娱面经

一面：

1. 集合
   1. list，map，set的区别，有哪些实现类
   2. 线程安全的list和map有哪些
   3. treeset和hashset的区别
   4. 使用treeset和hashset的场景
   5. linkedhashmap的使用场景，问我linkedhashmap的使用场景，我说可以用于LRU算法，竟然被面试官反驳了。。。根据数据结构进行解释，还是被反驳。。。
2. 设计模式
   1. 画一画代理模式的uml图
   2. 工厂模式和建造者模式的区别
   3. 简单工厂模式、抽象工厂模式、工厂方法模式的区别。
3. Spring
   1. 介绍一下spring ioc和aop，实现原理
4. JVM
   1. jvm怎样判断一个对象是否可回收，怎样的对象才能作为GC root
5. java并发
   1. java中有哪几种锁，同步锁和显式锁的区别，使用场景(自旋锁那些)
   2. ExecutorService的实现类有哪些，只介绍了ThreadExecutorPool
6. java基础
   1. 泛型的T,E,?和空的区别
   2. 怎样自定义注解
7. 计算机基础
   1. tcp/ip、http和https的区别
   2. 什么是死锁，用代码写一个死锁
   3. 带宽和传输速度的关系
   4. 一般局域网的带宽有多少
   5. 硬盘和内存读写速度是多少

可能计算机基础学得不好，硬件和内存的读写速度真答不上来，面试官还问我是什么专业的，尴尬

接着等了半个钟，才二面，技术总监面

二面：

1. 自我介绍
2. 说说项目中的难点
3. 两题算法题：
   1. 关于括号的算法题，用栈
   2. 最长字符串匹配，用tire树
4. spring ioc与aop的实现
5. 原对象中两个方法，方法a与方法b。使用spring aop对该对象进行增强处理，增强处理都为都是输出一条日志。在代理对象的a方法中调用b方法，会输出几条日志。（答案是一条日志，因为代理对象a方法调用的是原对象的b方法，而不是代理对象的b方法）
6. 有了解过你应聘的这个岗位的工作吗？与你的匹配吗
7. 了解过游戏公司的工作机制吗
8. 你认为公司加班的主要原因是什么
9. 如何避免加班

Hr面：

1. 期待薪资
2. 自己的缺点
3. 不能忍受舍友的哪些做法
4. 什么时候能入职
5. 还会回原来实习的那家公司吗

相对来说，互联网公司的hr面还是比较干脆的，没有那么多套路。

不会像某些公司一样，会问一些“阿里和我的公司选哪个”，“什么情况会让你拒绝我们公司的offer“这种问题。