笔记

1. JVM原理
2. HashMap底层原理
3. Java中comparable和comparator的区别及底层实现原理
4. 设计模式
5. SpringCloud
   1. Aop
   2. Spring Ioc
   3. Eureka
   4. Ribbon
   5. Feign
   6. Hystrix
   7. Zuul
   8. rpc调用协议底层
6. Redis
7. RocketMQ（分布式事务）
8. TCP/IP、HTTP协议相关知识点
9. 三范式
10. 数据库引擎（包含锁的概念）
11. 组合索引的问题
12. 优化sql的案例

13 存储过程

## JVM原理：

参考资料：<https://blog.csdn.net/csdnliuxin123524/article/details/81303711>

1.虚拟机栈、本地方法栈、程序计数器(PC寄存器：存储当前线程的下一步指令)这三个模块是线程私有的

2.程序计数器模块是JVM内存区域唯一不会报outofMemoryError情况的区域。

3.双亲委派机制：JVM在加载类时默认采用的是双亲委派机制。通俗的讲，就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载

4.类加载优先级：越是基础的类，越是被上层的类加载器进行加载，可以比较笼统的说像jdk自带的几个jar包肯定是位于最顶级的，再就是我们引用的包，最后是我们自己写的，保证了java程序的稳定性。

5.类加载器：bootStrap：%JAVA\_HOME%/lib下的jar、extclassLoader: %JAVA\_HOME%/lib/ext下的jar、appclassloader：calsspath下自己的类库

6.类加载器把class加载到方法区转变成指令，执行引擎负责把指令转换成操作系统的语言再转换成机器码

7.GC垃圾回收机制：



① 新生区

       新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分：伊甸区（Eden space）和幸存者区（Survivor pace），所有的类都是在伊甸区被new出来的。幸存区有两个：0区（Survivor 0 space）和1区（Survivor 1 space）。当伊甸园的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园进行垃圾回收（Minor GC）,将伊甸园中的剩余对象移动到幸存0区。若幸存0区也满了，再对该区进行垃圾回收，然后移动到1区。那如果1去也满了呢？再移动到养老区。若养老区也满了，那么这个时候将产生Major GC（FullGCC），进行养老区的内存清理。若养老区执行Full GC 之后发现依然无法进行对象的保存，就会产生OOM异常“OutOfMemoryError”。

     如果出现java.lang.OutOfMemoryError: Java heap space异常，说明Java虚拟机的堆内存不够。原因有二：

    a.Java虚拟机的堆内存设置不够，可以通过参数-Xms、-Xmx来调整。

     b.代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）。

② 养老区

         养老区用于保存从新生区筛选出来的 JAVA 对象，一般池对象都在这个区域活跃。

③ 永久区

         永久存储区是一个常驻内存区域，用于存放JDK自身所携带的 Class,Interface 的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存。

     如果出现java.lang.OutOfMemoryError: PermGen space，说明是Java虚拟机对永久代Perm内存设置不够。原因有二：

     a. 程序启动需要加载大量的第三方jar包。例如：在一个Tomcat下部署了太多的应用。

     b. 大量动态反射生成的类不断被加载，最终导致Perm区被占满。

     说明：

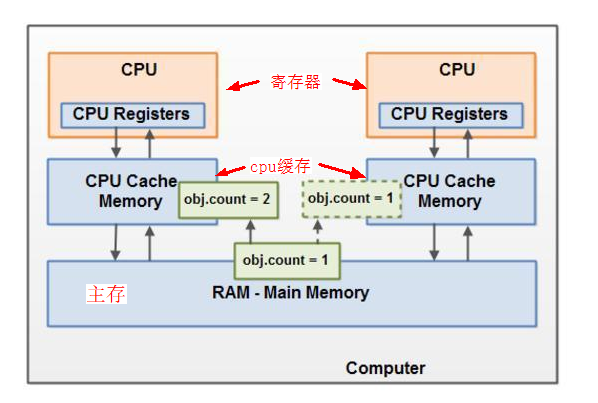
     Jdk1.6及之前：常量池分配在永久代 。

     Jdk1.7：有，但已经逐步“去永久代” 。

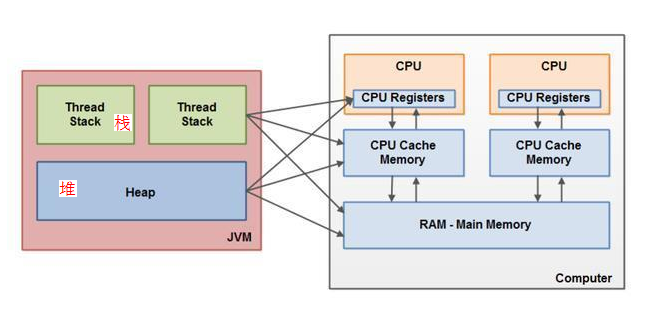
     Jdk1.8及之后：无(java.lang.OutOfMemoryError: PermGen space,这种错误将不会出现在JDK1.8中)。

8.volatile原理：解决数据的可见性。保证数据一致性，不能保证原子性

下图 左边CPU中运行的线程从主存中拷贝共享对象obj到它的CPU缓存，把对象obj的count变量改为2。但这个变更对运行在右边CPU中的线程不可见，因为这个更改还没有flush到主存中：



要解决共享对象可见性这个问题，我们可以使用java volatile关键字，volatile关键字可以保证变量会直接从主存读取，而对变量的更新也会直接写到主存。volatile原理是基于CPU内存屏障指令实现的。



9.synchronized关键字；

synchronized代码块可以保证同一个时刻只能有一个线程进入代码竞争区，synchronized代码块也能保证代码块中所有变量都将会从主存中读，当线程退出代码块时，对所有变量的更新将会flush到主存，不管这些变量是不是volatile类型的。

10. volatile和 synchronized区别

① volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

② volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的

volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性

③ volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化

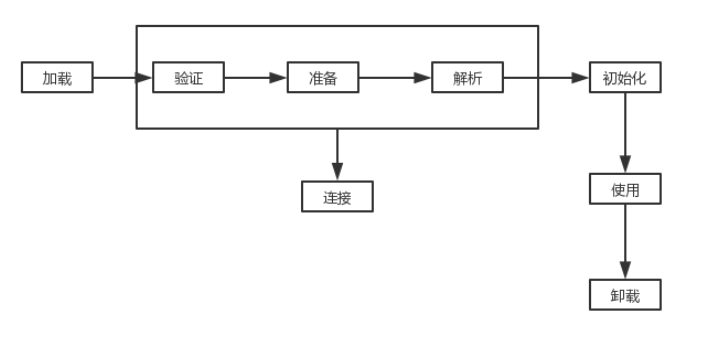
支撑Java内存模型的基础原理

11. 方法区与堆的区别

方法区存放了类的信息，有类的静态变量、final类型变量、field自动信息、方法信息，处理逻辑的指令集，我们仔细想想一个类里面也就这些东西，而堆中存放是对象和数组，咋一看好像方法区跟堆的作用是一样的。其实呢，1，这里就关系到我们平时说的对象是类的实例，是不是有点恍然大悟了？这里的对应关系就是 **“方法区--类”  “堆--对象”**，以“人”为例就是，堆里面放的是你这个“实实在在的人，有血有肉的”，而方法区中存放的是描述你的文字信息，如“你的名字，身高，体重，还有你的行为，如吃饭，走路等”。2，再者我们从另一个角度理解，就是从前我们得知方法区中的类是唯一的，同步的。但是我们在代码中往往同一个类会new几次，也就是有多个实例，既然有多个实例，那么在堆中就会分配多个实例空间内存。

12.JVM类加载过程：**加载-->验证-->准备-->解析-->初始化**，之后类就可以被使用了。绝大部分情况下是按这样的顺序来完成类的加载全过程的。但是是有例外的地方，解析也是可以在初始化之后进行的，这是为了支持

java的运行时绑定，并且在一个阶段进行过程中也可能会激活后一个阶段，而不是等待一个阶段结束再进行后一个阶段。



**1.加载：**calss文件变成calss对象

   加载时jvm做了这三件事：

     1）通过一个类的全限定名来获取该类的二进制字节流

     2）将这个字节流的静态存储结构转化为方法区运行时数据结构

     3）在内存堆中生成一个代表该类的java.lang.Class对象，作为该类数据的访问入口

**2.验证：各种检查**

   验证、准备、解析这三步可以看做是一个连接的过程，将类的字节码连接到JVM的运行状态之中

   验证是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，不会威胁到jvm的安全

   验证主要包括以下几个方面的验证：

　　1）文件格式的验证，验证字节流是否符合Class文件的规范，是否能被当前版本的虚拟机处理

     2）元数据验证，对字节码描述的信息进行语义分析，确保符合java语言规范

　　3）字节码验证 通过数据流和控制流分析，确定语义是合法的，符合逻辑的

　　4）符号引用验证 这个校验在解析阶段发生,将符号引用转化成直接引用

**3.准备**为类的静态变量分配内存（1.7后在堆内存中），初始化为系统的初始值。对于final static修饰的变量，直接赋值为用户的定义值。如下面的例子：这里在准备阶段过后的初始值为0，而不是7

public static int a=7

public final static int a = 7 变成了常量，在编译阶段会为x⽣成ConstantValue属性，在准备阶段虚拟机会根据

ConstantValue属性将x赋值为1000。

**4.解析**

  解析是将常量池内的符号引用转为直接引用（如物理内存地址指针）

**5.初始化**

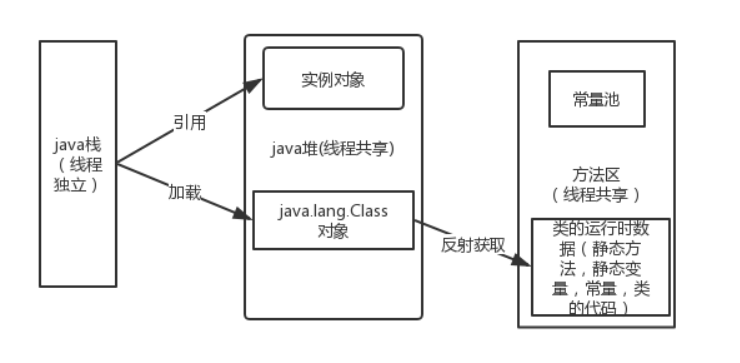
 到了初始化阶段，jvm才真正开始执行类中定义的java代码

      1）初始化阶段是执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译器自动收集

           类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生的。

      2）当初始化一个类的时候，如果发现其父类还没有进行过初始化、则需要先触发其父类的初始化。

      3）虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。



初始化顺序：

* 父类静态变量
* 父类静态代码块
* 子类静态变量
* 子类静态代码块
* 父类普通变量
* 父类普通代码块
* 父类构造函数
* 子类普通变量
* 子类普通代码块
* 子类构造函数

### Synchronized和Lock区别

参考文档：<https://www.jianshu.com/p/b343a9637f95>

### GC垃圾回收

***如何找到需要回收的对象***，程序计数法，主流实现是**可达性分析：**

可达性算法的基本思路，通过一系列为“GC　Ｒｏｏｔｓ”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference　Ｃｈａｉｎ），　当一个对象到GC　Ｒｏｏｔｓ　没有任何引用链相连时，相当于图论的从GC　Ｒｏｏｔｓ不可达，则这个对象不可用。

程序计数法：给对象中添加一个引用计数器，每当一个地方引用这个对象时，计数器值+1；当引用失效时，计数器值-1。任何时刻计数值为0的对象就是不可能再被使用的。

引用类型

强引用

只要强引用还存在，垃圾收集器永远不会收掉被引用的对象

软引用

在系统将要发生内存异常之前，将会把这些对象列进回收范围之中进行第二次回收。

弱引用

被弱引用关联的对象只能生存道下一次垃圾收集发生之前。

虚引用

一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象的实例。

### GC回收方法区

方法区即为永久代，主要回收两部分内容：废弃常量和无用类。

**废弃常量：**

当一个常量对象不再任何地方被引用的时候，则被标记为废弃常量，这个常量可以被回收。

**无用类：**

满足以下3个条件的类称之为无用类

该类所所有的对象实例已经被回收，也就是java堆中不存在该类的任何实例

加载该类的ClassLoader已经被回收

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

在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGI这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久带不会溢出。

但是**并不是一定会被回收，需要参数进行控制**，例如HotSpot虚拟机提供了-Xnoclassgc参数进行控制是否回收。

### GC回收算法

1. 标记清除算法：分为标记和清除两个阶段。该算法首先从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象并进行回收，标记和清除两个过程的效率都不高;并会产生内存碎片。
2. 标记整理算法：将所有的存活对象移动到一端，并对不存活对象进行处理，因此其不会产生内存碎片

3. 复制算法：可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这种算法适用于对象存活率低的场景，比如新生代。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。现在商用的虚拟机都采用这种算法来回收**新生代**

4. 分代收集算法：对于一个大型的系统，当创建的对象和方法变量比较多时，堆内存中的对象也会比较多，如果逐一分析对象是否该回收，那么势必造成效率低下。分代收集算法是基于这样一个事实：不同的对象的生命周期(存活情况)是不一样的，而不同生命周期的对象位于堆中不同的区域，因此对堆内存不同区域采用不同的策略进行回收可以提高 JVM 的执行效率。当代商用虚拟机使用的都是分代收集算法：新生代对象存活率低，就采用复制算法；老年代存活率高，就用标记清除算法或者标记整理算法。Java堆内存一般可以分为新生代、老年代和永久代三个模块

#### 容易造成内存泄漏的几点：

1. **静态集合类**：如HashMap、LinkedList等等。如果这些容器为静态的，那么它们的生命周期与程序一致，则容器中的对象在程序结束之前将不能被释放，从而造成内存泄漏。简单而言，长生命周期的对象持有短生命周期对象的引用，尽管短生命周期的对象不再使用，但是因为长生命周期对象持有它的引用而导致不能被回收。
2. **各种连接，如数据库连接、网络连接和IO连接等**。在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用close方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对Connection、Statement或ResultSet不显性地关闭，将会造成大量的对象无法被回收，从而引起内存泄漏
3. **内部类持有外部类**，如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持有外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露

## 反射原理和使用

参考资料：<https://blog.csdn.net/weixin_42724467/article/details/84311385>

获取class对象的三种方式：

1.Class<?> perClazz = Class.forName("reflect\_fanshe.Person");

2.Class<?> perClazz2 = Person.class;

1. Person person = new Person();

Class<?> perClazz3 = person.getClass();

## 动态代理原理和使用

## 线程池的工作原理

参考资料：<https://blog.csdn.net/lzxlfly/article/details/83904032>

<https://blog.csdn.net/qq_35909080/article/details/87002367>

<https://www.cnblogs.com/wxd0108/p/5479442.html>

<https://www.cnblogs.com/xiaoxi/p/7581899.html>

### 线程池的工作流程

1）当提交一个新任务到线程池时，线程池判断corePoolSize线程池是否都在执行任务，如果有空闲线程，则创建一个新的工作线程来执行任务，直到当前线程数等于corePoolSize；

2）如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

3）如果阻塞队列满了，那就创建新的线程执行当前任务，直到线程池中的线程数达到maxPoolSize，这时再有任务来，由饱和策略来处理提交的任务

### 线程池参数

public ThreadPoolExecutor(int corePoolSize, #核心线程数

                              int maximumPoolSize,   #最大线程数

                              long keepAliveTime, #达到最大线程数数时候，线程池的工作线程空闲后，保持存活的时间

                              TimeUnit unit,     #keepAliveTime单位

                              BlockingQueue<Runnable> workQueue #阻塞队列

                              RejectedExecutionHandler handler  #饱和策略

)

## HashMap底层原理

参考文档：<https://blog.csdn.net/qq_38182963/article/details/78942764>

Hashmap底层是由node数组+链表+红黑树组层。Node<K,V>[] tab

Hashmap的put过程：

1.判断数组是否为空，如果是空，则创建默认长度位 16 的数组。

2.通过与运算计算对应 hash 值的下标，如果对应下标的位置没有元素，则直接创建一个。

如果有元素，说明 hash 冲突了，则再次进行 3 种判断。

3.判断两个冲突的key是否相等，equals 方法的价值在这里体现了。如果相等，则将已经存在的值赋给变量e。最后更新e的value，也就是替换操作。

4.如果key不相等，则判断是否是红黑树类型，如果是红黑树，则交给红黑树追加此元素。

5.如果key既不相等，也不是红黑树，则是链表，那么就遍历链表中的每一个key和给定的key是否相等。如果，链表的长度大于等于8了，则将链表改为红黑树，（但是在替换前，会先判断，如果数组是 null 或者数组的长度小于 64，则重新散列，因为重新散列会拆分链表，使得链表的长度变短。提高性能。如果长度大于64了。就只能将链表变为红黑树了。）这是Java8 的一个新的优化。

最后，如果这三个判断返回的 e 不为null，则说明key重复，则更新key对应的value的值。

对维护着迭代器的modCount 变量加一。

最后判断容量是否大于阀值（默认是16）。如果大于，则调用 resize 方法，重新散列

## Map数据结构分类

### **HashTable**

* 底层数组+链表实现，无论key还是value都**不能为null**，synchronized线程**安全**，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化
* 初始size为**11**，扩容：newsize = olesize\*2+1
* 计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

### HashMap

* 底层数组+链表实现，可**以存储null键和null值**，线程**不安全**
* 初始size为**16**，扩容：newsize = oldsize\*2，size一定为2的n次幂
* 扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入
* 插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）
* 当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀
* 计算index方法：index = hash & (tab.length – 1)
* “负载极限”的默认值（0.75）是时间和空间成本上的一种折中：
* 较高的“负载极限”可以降低hash表所占用的内存空间，充分利用空间，但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap的get()与put()方法都要用到查询）
* 较低的“负载极限”会提高查询数据的性能，但会增加hash表所占用的内存开销，浪费内存资源

为什么负载因子是0.75

在理想情况下，使用随机哈希吗，节点出现的频率在hash桶中遵循泊松分布，同时给出了桶中元素的个数和概率的对照表。  
从上表可以看出当桶中元素到达8个的时候，概率已经变得非常小，也就是说用0.75作为负载因子，每个碰撞位置的链表长度超过8个是几乎不可能的。

hash容器指定初始容量尽量为2的幂次方。  
HashMap负载因子为0.75是空间和时间成本的一种折中。

### TreeMap

非线程安全。自然排序，时间复杂度O(log(n)) ，红黑树结构，key不能为null，value可以

### LinkedHashMap

和hashMap类似

### **concurrentMap**

键值都不能为空，线程安全，支持并发

ConcurrentHashMap和Hashtable主要区别就是围绕着锁的粒度以及如何锁,可以简单理解成把一个大的HashTable分解成多个，形成了锁分离。

通过给**每个线程分配桶区间**(默认一个线程分配的桶是16个)，避免线程间的争用。

通过为**每个桶节点加锁**，避免 putVal 方法导致数据不一致。

同时，在扩容的时候，也会**将链表拆成两份**，这点和 HashMap 的 resize 方法类似。

## Java中comparable和comparator的区别及底层实现原理

参考资料：<https://www.cnblogs.com/cxuanBlog/p/10927495.html>

### Comparable是一个排序接口

此接口给实现类提供了一个排序的方法，此接口有且只有一个方法

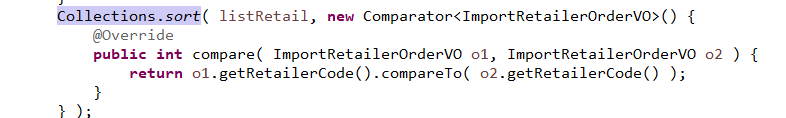
public int compareTo(T o);



### Comparator 相当于一个比较器

你可以不用实现此接口或者Comparable接口就可以实现次序比较

int compare(T o1, T o2);



**Collection**是集合类的上级**接口**，继承与他有关的接口主要有List和Set  
**Collections**是针对集合类的一个**帮助类**，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全等操作

Collections工具类支持两种排序方法：

Collections.sort(List<T> list);

Collections.sort(List<T> list, Comparator<? super T> c)

Collections.sort源码中调用了Arrays.sort，Arrays.sort在1.5之前使用传统的归并排序，1.5之后使用二分法插入排序

## 设计模式

参考资料：<http://c.biancheng.net/design_pattern/>

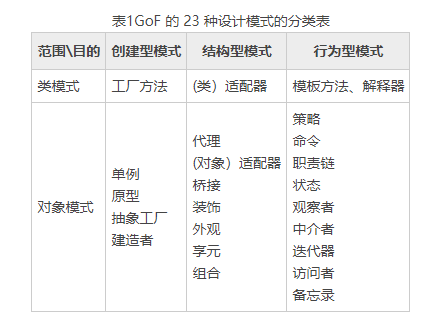
### 根据目的来分

根据模式是用来完成什么工作来划分，这种方式可分为创建型模式、结构型模式和行为型模式 3 种。

1. 创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。
2. 结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。
3. 行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

### 根据作用范围来分

根据模式是主要用于类上还是主要用于对象上来分，这种方式可分为类模式和对象模式两种。

1. 类模式：用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。GoF中的工厂方法、（类）适配器、模板方法、解释器属于该模式。
2. 对象模式：用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时刻是可以变化的，更具动态性。GoF 中除了以上 4 种，其他的都是对象模式。
3. 

### GoF的23种设计模式的功能

前面说明了 GoF 的 23 种设计模式的分类，现在对各个模式的功能进行介绍。

1. **单例（Singleton）模式**：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。
2. **原型（Prototype）模式**：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。
3. **工厂方法（Factory Method）模式**：定义一个用于创建产品的接口，由子类决定生产什么产品。
4. **抽象工厂（AbstractFactory）模式**：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。
5. **建造者（Builder）模式**：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。
6. **代理（Proxy）模式**：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。
7. **适配器（Adapter）模式**：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。
8. **桥接（Bridge）模式**：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。
9. **装饰（Decorator）模式**：动态的给对象增加一些职责，即增加其额外的功能。
10. **外观（Facade）模式**：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。
11. **享元（Flyweight）模式**：运用共享技术来有效地支持大量细粒度对象的复用。
12. **组合（Composite）模式**：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。
13. **模板方法（TemplateMethod）模式**：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。
14. **策略（Strategy）模式**：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。
15. **命令（Command）模式**：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。
16. **职责链（Chain of Responsibility）模式**：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。
17. **状态（State）模式**：允许一个对象在其内部状态发生改变时改变其行为能力。
18. **观察者（Observer）模式**：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。
19. **中介者（Mediator）模式**：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。
20. **迭代器（Iterator）模式**：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。
21. **访问者（Visitor）模式**：在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。
22. **备忘录（Memento）模式**：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。
23. **解释器（Interpreter）模式**：提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

## SpringCloud

### Aop

面向切面编程的一种思想，主要应用在日志，事务，拦截器等功能

### Spring Ioc

控制反转，依赖注入，把对应的生命周期交给spring容器来管理，spring容器负责对象的创建与销毁

### Eureka

参考资料：<https://www.cnblogs.com/snowjeblog/p/8821325.html>

<https://blog.csdn.net/qwe86314/article/details/94552801>

#### ****Eureka Server：注册中心服务端****

注册中心服务端主要对外提供了三个功能：

**服务注册**  
服务提供者启动时，会通过 Eureka Client 向 Eureka Server 注册信息，Eureka Server 会存储该服务的信息，Eureka Server 内部有二层缓存机制来维护整个注册表

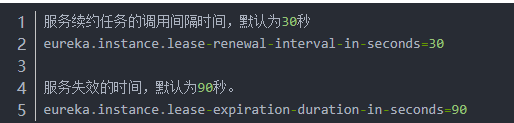
**提供注册表**  
服务消费者在调用服务时，如果 Eureka Client 没有缓存注册表的话，会从 Eureka Server 获取最新的注册表

**同步状态**  
Eureka Client 通过注册、心跳机制和 Eureka Server 同步当前客户端的状态。

#### ****Eureka**** Client****：注册中心客户端**** Eureka Client 是一个 Java 客户端，用于简化与 Eureka Server 的交互。Eureka Client 会拉取、更新和缓存 Eureka Server 中的信息。因此当所有的 Eureka Server 节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者，但是当服务有更改的时候会出现信息不一致。

**Register: 服务注册**  
服务的提供者，将自身注册到注册中心，服务提供者也是一个 Eureka Client。当 Eureka Client 向 Eureka Server 注册时，它提供自身的元数据，比如 IP 地址、端口，运行状况指示符 URL，主页等。

**Renew: 服务续约**  
Eureka Client 会每隔 30 秒发送一次心跳来续约。 通过续约来告知 Eureka Server 该 Eureka Client 运行正常，没有出现问题。 默认情况下，如果 Eureka Server 在 90 秒内没有收到 Eureka Client 的续约，Server 端会将实例从其注册表中删除，此时间可配置，一般情况不建议更改。



**Eviction 服务剔除**  
当 Eureka Client 和 Eureka Server 不再有心跳时，Eureka Server 会将该服务实例从服务注册列表中删除，即服务剔除。

**Cancel: 服务下线**  
Eureka Client 在程序关闭时向 Eureka Server 发送取消请求。 发送请求后，该客户端实例信息将从 Eureka Server 的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：

#### 自我保护机制

默认情况下，如果 Eureka Server 在一定的 90s 内没有接收到某个微服务实例的心跳，会注销该实例。但是在微服务架构下服务之间通常都是跨进程调用，网络通信往往会面临着各种问题，比如微服务状态正常，网络分区故障，导致此实例被注销。

固定时间内大量实例被注销，可能会严重威胁整个微服务架构的可用性。为了解决这个问题，Eureka 开发了自我保护机制，那么什么是自我保护机制呢？

Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低于 85%，如果低于 85%，Eureka Server 即会进入自我保护机制。

### Ribbon

##### 简介

  Spring Cloud Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡的服务调用

（扩展：们通常所说的负载均衡都指的是服务端负载均衡，其中分为硬件负载均衡和软件负载均衡。硬件负载均衡主要通过在服务器节点之间按照专门用于负载均衡的设备，比如F5等；而软件负载均衡则是通过在服务器上安装一些用于负载均衡功能或模块等软件来完成请求分发工作，比如Nginx等）

底层使用Spring的restTemplate请求方式，通过配置@LoadBalanced注解让restTemplate拥有客户端负载均衡的能力

##### 轮训策略

* RoundRobinRule(轮询算法)
* RandomRule(随机算法)
* AvailabilityFilteringRule()：会先过滤由于多次访问故障而处于断路器跳闸状态的服务，还有并发的连接数量超过阈值的服务，然后对剩余的服务列表按照轮询策略进行访问
* WeightedResponseTimeRule()：根据平均响应的时间计算所有服务的权重，响应时间越快服务权重越大被选中的概率越高，刚启动时如果统计信息不足，则使用RoundRobinRule策略，等统计信息足够会切换到WeightedResponseTimeRule
* RetryRule()：先按照RoundRobinRule的策略获取服务，如果获取失败则在制定时间内进行重试，获取可用的服务。
* BestAviableRule()：会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务

##### 实现原理

当ribbon和Eureka配和使用时ribbon可以自动获取服务注册列表，并基于负载均衡算法，请求其中的一个服务提供实例

ribbon实现的关键点是为ribbon定制的RestTemplate，ribbon利用了RestTemplate的拦截器机制，将拦截器注入到RestTemplate中 在拦截器中实现ribbon的负载均衡。负载均衡的基本实现就是利用applicationName从服务注册中心获取可用的服务地址列表，然后通过一定算法负载，决定使用哪一个服务地址来进行http调用。

### Feign

服务间调用组件feign，并且集成了Ribbon

1.通过主类上的EnableFeignClients 注解开启FeignClient；

2.根据Feign 的规则实现接口，并加上FeignClient注解，供调用的地方注入调用；

3.程序启动后，会扫描所有FeignClient 注解的类，并将这些信息注入到IOC 容器中；

4.当b中接口被调用时，通过jdk代理，以及反射（Spring处理注解的方式），来生成具体的RequestTemplate

5.RequestTemplate 生成Reqest

6.Request 交给httpclient处理，这里的httpclient 可以是OkHttp，也可以是HttpUrlConnection 或者HttpClient

7.最后Client被封装到LoadBalanceClient类，这个类结合Ribbon 实现负载均衡

###### Feign技术的底层实现

　　Feign的HTTP客户端支持3种框架，分别是；**HttpURLConnection、HttpClient、OKHttp。Feign中默认使用HttpURLConnection。**

**HttpURLConnection是JDK自带的HTTP客户端技术，并不支持连接池**，如果要实现连接池的机制，还需要自己来管理连接对象。对于网络请求这种底层相对复杂的操作，如果有可用的其他方案，也没有必要自己去管理连接对象。

**Apache提供的HttpClient框架**相比传统JDK自带的HttpURLConnection，它封装了访问http的请求头，参数，内容体，响应等等；它不仅使客户端发送HTTP请求变得容易，而且也方便了开发人员测试接口（基于Http协议的），即提高了开发的效率，也方便提高代码的健壮性；另外高并发大量的请求网络的时候，还是用“HTTP连接池”提升吞吐量。

　　OKHttp是一个处理网络请求的**开源项目**,是**安卓端**最火热的**轻量级框架**,由移动支付Square公司贡献用于替代HttpUrlConnection和Apache HttpClient。OKHttp拥有**共享Socket,减少对服务器的请求次数，通过连接池,减少了请求延迟等技术特点**。

### Hystrix

Springcloud熔断器，在分布式系统中对服务间的调用进行控制，加入一些调用延迟或者依赖故障的容错机制。

#### 熔断

当Hystrix Command请求后端服务失败数量（默认是20个服务）超过一定比例(默认50%), 断路器会切换到开路状态(Open). 这时所有请求会直接失败而不会发送到后端服务. 断路器保持在开路状态一段时间后(默认5秒), 自动切换到半开路状态(HALF-OPEN).

这时会判断下一次请求的返回情况, 如果请求成功, 断路器切回闭路状态(CLOSED), 否则重新切换到开路状态(OPEN).

#### 降级

Fallback相当于是降级操作. 对于查询操作, 我们可以实现一个fallback方法, 当请求后端服务出现异常的时候, 可以使用fallback方法返回的值. fallback方法的返回值一般是设置的默认值或者来自缓存.告知后面的请求服务不可用了，不要再来了。

feign的注解@FeignClient包含两个方法：

1.

@FeignClient(name=" user", fallback=XXXFallback.class)

fallbackFactory（类似于断容器）：需要访问导致回退的触发原因时：

public class XXXFallback implements FallbackFactory<XXXApi> {

2.

@FeignClient(value="trade",fallbackFactory= XXXFallback.class )

fallback方法：普通回退方法 直接指明类名 ：

public class XXXFallback implements XXXApi{

(如果需要捕捉异常错误信息用此降级方法)

备注：降级的次数多了就会进入熔断器

#### 隔离

(采用舱壁模式，Docker就是舱壁模式的一种)：在Hystrix中, 主要通过线程池来实现资源隔离. 通常在使用的时候我们会根据调用的远程服务划分出多个线程池.比如说，一个服务调用两外两个服务，你如果调用两个服务都用一个线程池，那么如果一个服务卡在哪里，资源没被释放

后面的请求又来了，导致后面的请求都卡在哪里等待，导致你依赖的A服务把你卡在哪里，耗尽了资源，也导致了你另外一个B服务也不可用了。这时如果依赖隔离，某一个服务调用A B两个服务，如果这时我有100个线程可用，我给A服务分配50个，给B服务分配50个，这样就算A服务挂了，

我的B服务依然可以用

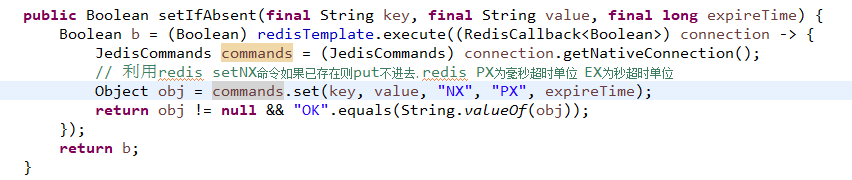
### Zuul

参考资料：<https://blog.csdn.net/weixin_44296862/article/details/88666322>

Zuul是Neflix开源的Api Gateway服务器，它本质上是一个Servlet应用，其核心是通过一系列filters的实现来为整个微服务体系提供路由、安全、监控等边界服务。Zuul目前分为两个大版本Zuul1和Zuul2，它们的区别在于Zuul1的IO模型还是BIO的方式，而Zuul2则是使用NIO对Zuul1进行了重构，所以性能上要优于Zuul1。

## Redis

### 分布式锁



### redis延迟队列

应用在订单取消

redisTemplate.opsForZSet().add(orderCancelMessage,orderNo, ( instance.getTimeInMillis() ) / 1000 );

### reids五大数据类型

## Redis详解

参考资料：<https://www.cnblogs.com/goody9807/p/7441739.html>

缓存击穿解决方案：<https://blog.csdn.net/kongtiao5/article/details/82771694>

Redis集群高可用：一致性hash算法：

<https://blog.csdn.net/u013851082/article/details/68063446>

## RocketMQ消息队列（分布式事务）

### 简介：

AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。 AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。 RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。 下面将重点介绍RabbitMQ中的一些基础概念，了解了这些概念，是使用好RabbitMQ的基础。

依赖包

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

<version>${spring-rabbit.version}</version>

</dependency>

### 交换机的四种模式：

一、**Fanout Exchange**

      它是最基本的交换机类型，它所能做的事情非常简单———广播消息。扇形交换机会把能接收到的消息全部发送给绑定在自己身上的队列。因为广播不需要“思考”，所以扇形交换机处理消息的速度也是所有的交换机类型里面最快的。

二、**Direct Exchange**

       它是一种带路由功能的交换机，一个队列会和一个交换机绑定，除此之外再绑定一个routing\_key，当消息被发送的时候，需要指定一个binding\_key，这个消息被送达交换机的时候，就会被这个交换机送到指定的队列里面去。同样的一个binding\_key也是支持应用到多个队列中的。这样当一个交换机绑定多个队列，就会被送到对应的队列去处理。

三、**Topic Exchange**

       它是一种支持正则匹配的exchange，发送到topic exchange上的消息需要携带指定规则的routing\_key，主题交换机会根据这个规则将数据发送到对应的(多个)队列上。该exchange的routing\_key需要有一定的规则，交换机和队列的binding\_key需要采用\*.#.\*.....的格式，每个部分用.分开，其中：\*表示一个单词，#表示任意数量（零个或多个）单词。

四、**Headers Exchange**

      它是忽略routing\_key的一种路由方式。路由器和交换机路由的规则是通过Headers信息来交换的，这个有点像HTTP的Headers。将一个exchange声明成Headers exchange，绑定一个队列的时候，定义一个Hash的数据结构，消息发送的时候，会携带一组hash数据结构的信息，当Hash的内容匹配上的时候，消息就会被写入队列。绑定exchange和队列的时候，Hash结构中要求携带一个键“x-match”，这个键的Value可以是any或者all，这代表消息携带的Hash是需要全部匹配(all)，还是仅匹配一个键(any)就可以了。相比direct exchange，首部交换机的优势是匹配的规则不被限定为字符串(string)。

### Work模式

1、自动模式

消费者从消息队列获取消息后，服务端就认为该消息已经成功消费。

2、手动模式

消费者从消息队列获取消息后，服务端并没有标记为成功消费

消费者成功消费后(ack确认)需要将状态返回到服务端

配置：

spring:

rabbitmq:

listener:

simple:

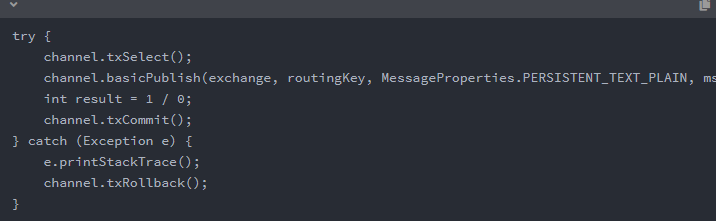
acknowledge-mode: manual #手动应答

### Rabbitmq事务机制

参考文档：<https://honeypps.com/mq/rabbitmq-message-confirm-strategy/>

#### 事务机制：

RabbitMQ中与事务机制有关的方法有三个：txSelect(), txCommit()以及txRollback(), txSelect用于将当前channel设置成transaction模式，txCommit用于提交事务，txRollback用于回滚事务，在通过txSelect开启事务之后，我们便可以发布消息给broker代理服务器了，如果txCommit提交成功了，则消息一定到达了broker了，如果在txCommit执行之前broker异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过txRollback回滚事务了。



缺点：使用事务机制的话会降低RabbitMQ的性能，降低RabbitMQ的消息吞吐量

#### Confirm机制：

参考资料：<https://blog.csdn.net/u013256816/article/details/55515234>

<https://blog.csdn.net/weixin_38003389/article/details/93193672>

配置信息：

spring:

rabbitmq:

publisher-confirms: true #confirm回调

publisher-returns: true #return回调

template

mandatory:true

**消息发送确认**

**ConfirmCallback**

**ConfirmCallback是一个回调接口，消息发送到 Broker 后触发回调，确认消息是否到达 Broker 服务器，也就是只确认是否正确到达 Exchange 中。**

我们需要在生产者的配置中添加下面配置，表示开启发布者确认



然后在生产者的Java配置类实现该接口



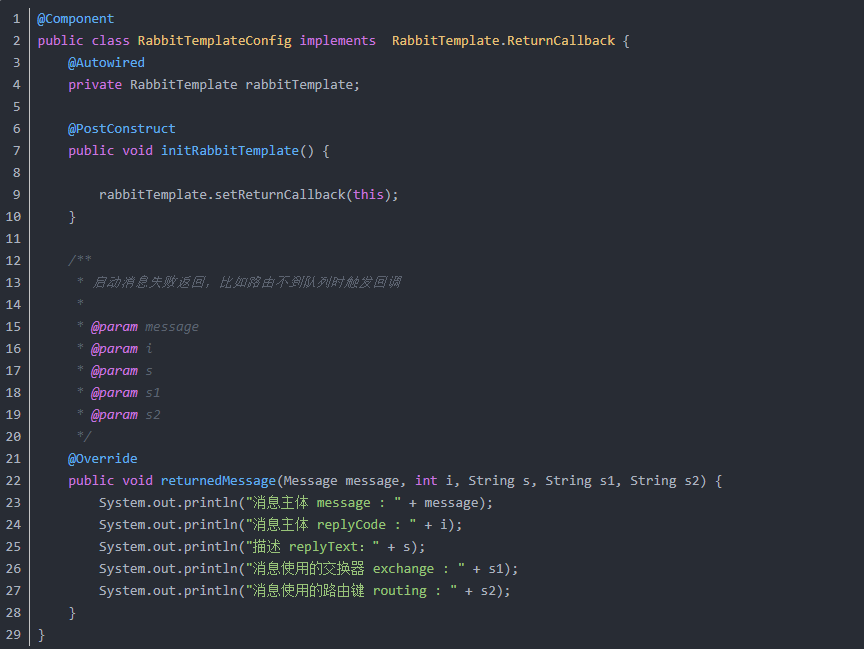
**ReturnCallback**

**通过实现 ReturnCallback 接口，启动消息失败返回，此接口是在交换器路由不到队列时触发回调，该方法可以不使用，因为交换器和队列是在代码里绑定的，如果消息成功投递到Broker后几乎不存在绑定队列失败，除非你代码写错了。**

使用此接口需要在生产者配置中加入一下配置，表示发布者返回



然后基于刚才的生产者Java配置里实现接口ReturnCallback



到此，**我们完成了生产者的异步确认（异步confirm模式:效率最高，吞吐量大），我们可以在回调函数中对当前失败的消息进行补偿，这样保证了我们没有发送成功的数据也被观察到了，比如某某条数据需要发送到消费者消费，但是没有发送成功，这就需要你在此做一些其他操作喽，根据你具体业务来**

踩坑：@RabbitListener(@QueueBinding( value = "",exchange= "",key)) 会自动创建队列交换机，并绑定。如果发现已经创建队列或者虚拟机参数不一致，会报错， 且队列虚拟机一旦创建参数将不能修改

exchangeDeclarePassive方法：主要有来检测相应的交换器是否存在。如果存在则正常返回;如果不存在则抛出异常：404 channel exception，同时channel也会被 关闭

### 消息重试

参考资料：

<https://blog.csdn.net/oschina_40730821/article/details/105120414>

#### 配置形式：

配置：

spring:

rabbitmq:

listener:

simple:

retry:

enabled: true #是否开启重试

initial-interval: 3000ms #重试时间间隔

max-attempts: 3 #重试次数

max-interval: 15000ms #重试最大时间间隔

multiplier: 2 #倍数

#### 代码形式：

@Bean

public RetryOperationsInterceptor interceptor(){

return RetryInterceptorBulider

.stateless()

.max-attempts(max-attempts)

.backOffOptions(initial-interval,multiplier,max-interval)

.recoverer(myMessageRecoverer:当重试失败后会调用这个类得recover方法，来做业务处理)

.bulid()

}

public class myMessageRecoverer implements MessageRecoverer{

@Override

public void recover(){

//业务处理

}

}

#### Rabbitmq配置类；



### 消息优先级：

参数：设置x-max-priority 默认0-250

踩坑：

1. 消息优先级要与prefetch\_count参数设置有关。prefetch\_count设置太大，无法实现优先级队列。因为prefetch\_count太大是把队列中所有的消息拿到本地。再去消费。队列中无消息自然不会阻塞，不阻塞就无法消息排序

2. 在使用exchangeDeclarePassive方法校验交换器是否存在时。不存在则创建一个交换器的逻辑下报错。因为如果不存在则抛出异常：404 channel exception，同时channel也会被关闭

prefetch\_count要点：



且设置prefetch\_count时

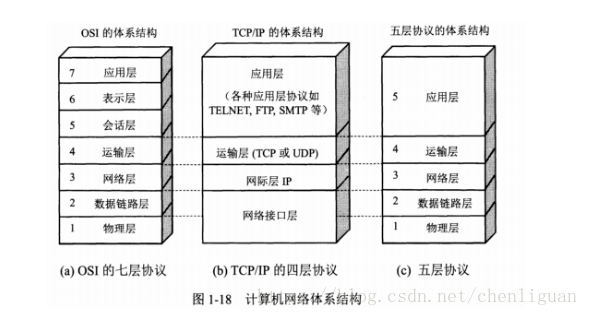
prefetch\_count参数仅仅在 autoAck参数设置为 false的前提下才生效，也就是不能使用自动确认，自动确认的消息没有办法限流

参考资料：https://zhuanlan.zhihu.com/p/266950873

## TCP/IP、HTTP协议相关知识点

参考资料：<https://blog.csdn.net/chenliguan/article/details/79146124>

TCP：TCP是Tranfer Control Protocol的简称，**TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。**通过TCP协议传输，得到的是一个顺序的无差错的数据流。发送方和接收方的成对的两个socket之间必须建立连接，以便在TCP协议的基础上进行通信，当一个socket（通常都是server socket）等待建立连接时，另一个socket可以要求进行连接，一旦这两个socket连接起来，它们就可以进行双向数据传输，双方都可以进行发送或接收操作。



应用：当对网络通讯质量有要求的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如HTTP、HTTPS、FTP等传输文件的协议，POP、SMTP等邮件传输的协议。在日常生活中，常见使用TCP协议的应用如下：浏览器，用的HTTPFlashFXP，用的FTPOutlook，用的POP、SMTPPutty，用的Telnet、SSHQQ文件传输.

### 三次握手：

TCP是面向连接的，无论哪一方向另一方发送数据之前，都必须先在双方之间建立一条连接。在TCP/IP协议中，TCP协议提供可靠的连接服务，连接是通过三次握手进行初始化的。三次握手的目的是同步连接双方的序列号和确认号并交换 TCP窗口大小信息。  
　　（1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=x，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。  
　　 （2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack (number )=x+1，随机产生一个值seq=y，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。  
　　（3）第三次握手：Client收到确认后，检查ACK是否为Client发送的seq+1，即x+1；如果正确则将标志位ACK置为1，ack=y+1，并将该数据包发送给Server。Server检查ack是否为Server发送的seq+1，即y+1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手。

URI和URL的区别：

URI标记了一个网络资源，仅此而已；  URL标记了一个WWW互联网资源（用地址标记），并给出了他的访问地址。(URI是Uniform Resource Identifier,表示是一个资源； URL是Uniform Resource Locator，表示是一个地址

### Restful和http的区别

参考资料：<https://www.cnblogs.com/willpan-z/p/9438930.html>

restFul是符合rest架构风格的网络API接口,完全承认Http是用于标识资源。restFul URL是面向资源的，可以唯一标识和定位资源。 对于该URL标识的资源做何种操作是由Http方法决定的。 rest请求方法有4种，包括get,post,put,delete.分别对应获取资源，添加资源，更新资源及删除资源.

### Restful

REST即**表述性状态传递**（Representational State Transfer，简称REST），是一种软件架构风格。REST通过HTTP协议定义的通用动词方法(GET、PUT、DELETE、POST) ，以URI对网络资源进行唯一标识，响应端根据请求端的不同需求，通过无状态通信，对其请求的资源进行表述。满足REST约束条件和原则的架构，就被称为是RESTful架构.

### RPC

RPC 即远程过程调用（Remote Procedure Call Protocol，简称RPC），像调用本地服务(方法)一样调用服务器的服务(方法)。通常的实现有 XML-RPC , JSON-RPC , 通信方式基本相同, 所不同的只是传输数据的格式.

RPC框架的主要目标就是让远程服务调用更简单、透明。RPC框架负责屏蔽底层的传输方式（TCP或者UDP）、序列化方式（XML/JSON/二进制）和通信细节。开发人员在使用的时候只需要了解谁在什么位置提供了什么样的远程服务接口即可，并不需要关心底层通信细节和调用过程。

### 区别

使用RPC远程服务调用方式与传统http接口直接调用方式的差别在于：

1. 从使用方面看，**Http接口只关注服务提供方（服务端）**，对于客户端怎么调用，调用方式怎样并不关心，通常情况下，**客户端使用Http方式进行调用时，只要将内容进行传输即可，**这样**客户端**在使用时，需要**更关注网络方面的传输**，比较不适用与业务方面的开发；而RPC服务则**需要客户端接口与服务端保持一致，服务端提供一个方法，客户端通过接口直接发起调用，业务开发人员仅需要关注业务方法的调用即可，**不再关注网络传输的细节，在开发上更为高效。

2. 从性能角度看，使用Http时，Http本身提供了丰富的状态功能与扩展功能，但也正由于Http提供的功能过多，导致在网络传输时，需要携带的信息更多，从性能角度上讲，较为低效。而**RPC服务网络传输上仅传输与业务内容相关的数据**，传输数据更小，性能更高。

3. 从运维角度看，使用Http接口时，常常使用一个前端代理，来进行Http转发代理请求的操作，需要进行扩容时，则需要去修改代理服务器的配置，较为繁琐，也容易出错。而使用RPC方式的微服务，则只要增加一个服务节点即可，注册中心可自动感知到节点的变化，通知调用客户端进行负载的动态控制，更为智能，省去运维的操作。

## Mysql相关知识点

### 三范式

参考资料：<https://www.cnblogs.com/knowledgesea/p/3667395.html>

什么是范式：简言之就是，数据库设计对数据的存储性能，还有开发人员对数据的操作都有莫大的关系。所以建立科学的，规范的的数据库是需要满足一些

规范的来优化数据数据存储方式。在关系型数据库中这些规范就可以称为范式。

什么是三大范式：

**第一范式**：当关系模式R的所有属性都不能在分解为更基本的数据单位时，称R是满足第一范式的，简记为1NF。满足第一范式是关系模式规范化的最低要

求，否则，将有很多基本操作在这样的关系模式中实现不了。（通俗讲：保证每一个属性不能再分解）

**第二范式**：如果关系模式R满足第一范式，并且R得所有非主属性都完全依赖于R的每一个候选关键属性，称R满足第二范式，简记为2NF。（通俗讲：每一行的数据只能与其中一列相关，即一行数据只做一件事。只要数据列中出现数据重复，就要把表拆分开来）

**第三范式**：设R是一个满足第一范式条件的关系模式，X是R的任意属性集，如果X非传递依赖于R的任意一个候选关键字，称R满足第三范式，简记为3NF.（通俗讲：属性之间不能有传递性，例如，订单号->商品->商品属性，这是不对的，需要把表拆开，订单表和商品表）

**注：**关系实质上是一张二维表，其中每一行是一个元组，每一列是一个属性

### Mysql数据库引擎（包含锁的概念）

1、MyISAM：默认表类型，它是基于传统的ISAM类型，ISAM是Indexed Sequential Access Method (有索引的顺序访问方法) 的缩写，它是存储记录和文件的标准方法。不是事务安全的，只支持表级锁，而且不支持外键，如果执行大量的select，insert MyISAM比较适合。

（MyISAM在执行查询语句(SELECT)前，会自动给涉及的所有表加读锁，在执行更新操作(UPDATE、DELETE、INSERT等)前，会自动给涉及的表加写锁。）

2、InnoDB：支持事务安全的引擎，支持外键、行锁、事务是他的最大特点。如果有大量的update和insert，建议使用InnoDB，特别是针对多个并发和QPS较高的情况。（**对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X)；对于普通SELECT语句，InnoDB不会加任何锁，事务可以通过以下语句显示给记录集加共享锁或排他锁。**）

**InnoDB行锁是通过给索引上的索引项加锁来实现的，因此InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！**

**共享锁（又称读锁）**：共享 (S) 用于不更改或不更新数据的操作（只读操作），如 SELECT 语句。如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据。lock in share mode

**排他锁（又称写锁）**：用于数据修改操作，例如 INSERT、UPDATE 或 DELETE。确保不会同时同一资源进行多重更新。如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。我们在操作数据库的时候，可能会由于并发问题而引起的数据的不一致性（数据冲突）for update

### 组合索引的问题

1、需要加索引的字段，要在where条件中  
2、数据量少的字段不需要加索引  
3、如果where条件中是**OR**关系，加索引不起作用  
4、符合最**左**原则

### 优化sql的案例

count(1) and count(\*)

当表的数据量大些时，对表作分析之后，使用count(1)还要比使用count(\*)用时多了！   
从执行计划来看，count(1)和count(\*)的效果是一样的。 但是在表做过分析之后，count(1)会比count(\*)的用时少些（1w以内数据量），不过差不了多少。   
如果count(1)是聚索引,id,那肯定是count(1)快。但是差的很小的。 因为count(\*),自动会优化指定到那一个字段。所以没必要去count(1)，用count(\*)，sql会帮你完成优化的 因此：count(1)和count(\*)基本没有差别！

Count(列名) 不统计NULL列

### MYSQL索引底层原理

参考资料：<https://blog.csdn.net/tongdanping/article/details/79878302>

<https://blog.csdn.net/guo_qiangqiang/article/details/88794971>

**索引类型**：主键索引、唯一索引、普通索引、全文索引、组合索引

1、主键索引：即主索引，根据主键pk\_clolum（length）建立索引，不允许重复，不允许空值；

ALTER TABLE 'table\_name' ADD PRIMARY KEY pk\_index('col')；

2、唯一索引：用来建立索引的列的值必须是唯一的，允许空值

ALTER TABLE 'table\_name' ADD UNIQUE index\_name('col')；

3、普通索引：用表中的普通列构建的索引，没有任何限制

ALTER TABLE 'table\_name' ADD NORMAL index\_name('col')；

4、全文索引：用大文本对象的列构建的索引（下一部分会讲解）MySQL5.6.24上InnoDB引擎也加入了全文索引

ALTER TABLE 'table\_name' ADD FULLTEXT INDEX ft\_index('col')；

5、组合索引：用多个列组合构建的索引，这多个列中的值不允许有空值

ALTER TABLE 'table\_name' ADD INDEX index\_name('col1','col2','col3')；

\*遵循“最左前缀”原则，把最常用作为检索或排序的列放在最左，依次递减，组合索引相当于建立了col1,col1col2,col1col2col3三个索引，而col2或者col3是不能使用索引的。

\*在使用组合索引的时候可能因为列名长度过长而导致索引的key太大，导致效率降低，在允许的情况下，可以只取col1和col2的前几个字符作为索引

ALTER TABLE 'table\_name' ADD INDEX index\_name(col1(4),col2（3))；

表示使用col1的前4个字符和col2的前3个字符作为索引

6、空间索引（SPATIAL）只限myisam存储引擎，底层使用R-tree索引

索引方法：B-tree，Hash索引，R-tree索引

### 数据库建立索引常用的规则如下：

参考文档：<https://www.cnblogs.com/abcdwxc/p/9855474.html>

**1、表的主键、外键必须有索引；**  
**2、数据量超过300的表应该有索引；**  
**3、经常与其他表进行连接的表，在连接字段上应该建立索引；**   
**4、经常出现在Where子句中的字段，特别是大表的字段，应该建立索引；**   
**5、索引应该建在选择性高的字段上；**   
**6、索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引；**   
**7、复合索引的建立需要进行仔细分析；尽量考虑用单字段索引代替：**

**8、频繁进行数据操作的表，不要建立太多的索引；**   
**9、删除无用的索引，避免对执行计划造成负面影响；**

**备注：**

1.Create index注意n如果是大表建立索引，切记加上ONLINE参数

2.上面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的 列，3.如果在前10 个或20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作

### Mysql的binlog日志

参考资料：<https://blog.csdn.net/lzhcoder/article/details/88814364>

Binlog日志是二进制日志，是记录所有数据库表结构变更（例如CREATE、ALTER TABLE…）以及表数据修改（INSERT、UPDATE、DELETE…）的二进制日志。binlog不会记录SELECT和SHOW这类操作，因为这类操作对数据本身并没有修改，但你可以通过查询通用日志来查看MySQL执行过的所有语句。（二进制日志包括两类文件：二进制日志索引文件（文件名后缀为.index）用于记录所有的二进制文件，二进制日志文件（文件名后缀为.00000\*）记录数据库所有的DDL和DML(除了数据查询语句)语句事件。）

### mysql事务日志

innodb事务日志包括redo log和undo log。

目的：**实例或者介质失败，事务日志文件就能派上用场。**

undo log指事务开始之前， 在操作任何数据之前,首先将需操作的数据备份到一个地方

redo log指事务中操作的任何数据,将最新的数据备份到一个地方

### ****Undo + Redo事务的简化过程****

假设有A、B两个数据，值分别为1,2，开始一个事务，事务的操作内容为：把1修改为3，2修改为4，那么实际的记录如下（简化）：

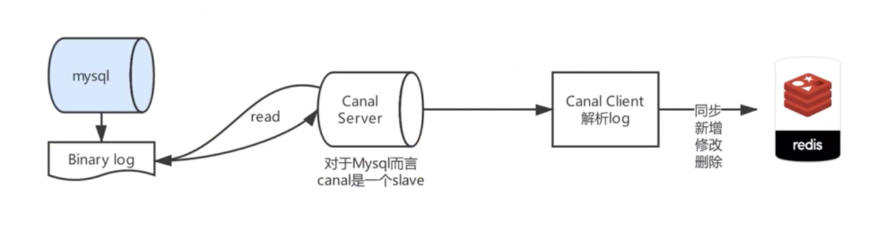
1. A.事务开始.
2. B.记录A=1到undo log.
3. C.修改A=3.
4. D.记录A=3到redo log.
5. E.记录B=2到undo log.
6. F.修改B=4.
7. G.记录B=4到redo log.
8. H.将redo log写入磁盘。
9. I.事务提交

### 用到binlog日志的开源框架

Canal：基于数据库增量日志解析，提供增量数据订阅和消费https://github.com/alibaba/canal

mysql会将操作记录在Binary log日志中，通过canal去监听数据库日志二进制文件，解析log日志，同步到redis中进行增删改操作。

canal的工作原理：canal 模拟 MySQL slave 的交互协议，伪装自己为 MySQL slave ，向 MySQL master 发送dump 协议；MySQL master 收到 dump 请求，开始推送 binary log 给 slave (即 canal )；canal 解析 binary log 对象(原始为 byte 流)。



canal的ha分为两部分，canal server和canal client分别有对应的ha(主备模式)实现

canal server:  为了减少对mysql dump的请求，不同server上的instance要求同一时间只能有一个处于running，其他的处于standby状态.

canal client: 为了保证有序性，一份instance同一时间只能由一个canal client进行get/ack/rollback操作，否则客户端接收无法保证有序

## Springmvc

处理请求流程：

1. 用户request的url发送至前端控制器.（DispatcherServlet）
2. 前端控制器请求处理器映射器（HandlerMapping）查找Handler（也就是我们平常说的 Controller 控制器）.
3. 找到Handler后,处理器映射器（HandlerMapping）向前端控制器返回Handler.
4. 前端控制器请求处理器适配器去按照特定的规则执行Handler.
5. Handler执行完成给处理器适配器返回ModelAndView对象.
6. 处理器适配器向前端控制器返回ModelAndView对象.
7. 前端控制器请求视图解析器（View Resolver）去解析ModelAndView对象,将逻辑视图（ModelAndView）解析为真正视图(View).
8. 视图解析器将视图（View）返回给前端控制器.
9. 前端控制器进行视图渲染,将模型数据（ModelAndView中设置的值）填充到request域中.
10. 前端处理器向用户响应处理结果.

### Spring相关知识点

参考资料：<https://blog.csdn.net/a745233700/article/details/80959716>

Spring是一个开源框架（**就是一个容器**，Spring模块都是在核心容器之上构建的），实现了工厂模式的工厂类，这个类名为BeanFactory(接口)，在程序中通常用他的子类ApplicationContext。

Spring bean的生命周期

1.实例化 —》 执行bean的构造器 —》 为bean注入属性 —》 属性初始化 —》 初始化容易成功开始调用 —》 销毁

## Java8新特性

参考资料：<https://www.ibm.com/developerworks/cn/java/j-lo-jdk8newfeature/>

### 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

### Lambda表达式

### 接口增强

Java 8 对接口做了进一步的增强。在接口中可以添加使用 default 关键字修饰的非抽象方法。还可以在接口中定义静态方法。如今，接口看上去与抽象类的功能越来越类似了。

### 集合之流操作

Java 8 引入了流式操作（Stream），通过该操作可以实现对集合（Collection）的并行处理和函数式操作

### 注解的更新

对于注解，Java 8 主要有两点改进：类型注解和重复注解

### IO/NIO 的改进

Java 8 对 IO/NIO 也做了一些改进。主要包括：改进了 java.nio.charset.Charset 的实现，使编码和解码的效率得以提升，也精简了 jre/lib/charsets.jar 包；优化了 String(byte[],\*) 构造方法和 String.getBytes() 方法的性能；还增加了一些新的 IO/NIO 方法，使用这些方法可以从文件或者输入流中获取流（java.util.stream.Stream），通过对流的操作，可以简化文本行处理、目录遍历和文件查找。

新增的 API 如下：

BufferedReader.line(): 返回文本行的流 Stream<String>

File.lines(Path, Charset):返回文本行的流 Stream<String>

File.list(Path): 遍历当前目录下的文件和目录

File.walk(Path, int, FileVisitOption): 遍历某一个目录下的所有文件和指定深度的子目录

File.find(Path, int, BiPredicate, FileVisitOption... ): 查找相应的文件

下面就是用流式操作列出当前目录下的所有文件和目录：

|  |  |
| --- | --- |
| 1  2 | Files.list(new File(".").toPath())   .forEach(System.out::println); |

### 全球化功能

Java 8 版本还完善了全球化功能：支持新的 Unicode 6.2.0 标准，新增了日历和本地化的 API，改进了日期时间的管理等

## Springboot启动流程：

* 1. 找到启动类：标注SpringBootApplication注解的类
  2. 执行main方法
  3. 构造SpringApplication实例
  4. SpringApplication类中的构造函数中调用initialize方法，通过ClassLoader.getResources加载META-INF/spring.factories路径下文件信息，从中找key为ApplicationContextInitializer.class，ApplicationListener.class实例化
  5. 调用SpringApplication的run方法
  6. 获得系统环境变量，配置环境激活信息，
  7. 初始化beanFactory，对beanFantory进行各种功能填充，激活beanFantory处理器
  8. 创建Tomcat时会利用这个后置处理器来初始化Tomcat Server类
  9. 为上下文初始化Message源，即不同语言的消息体，国际化处理
  10. 初始化事件派发器，并放到applicationEventMulticaster bean中

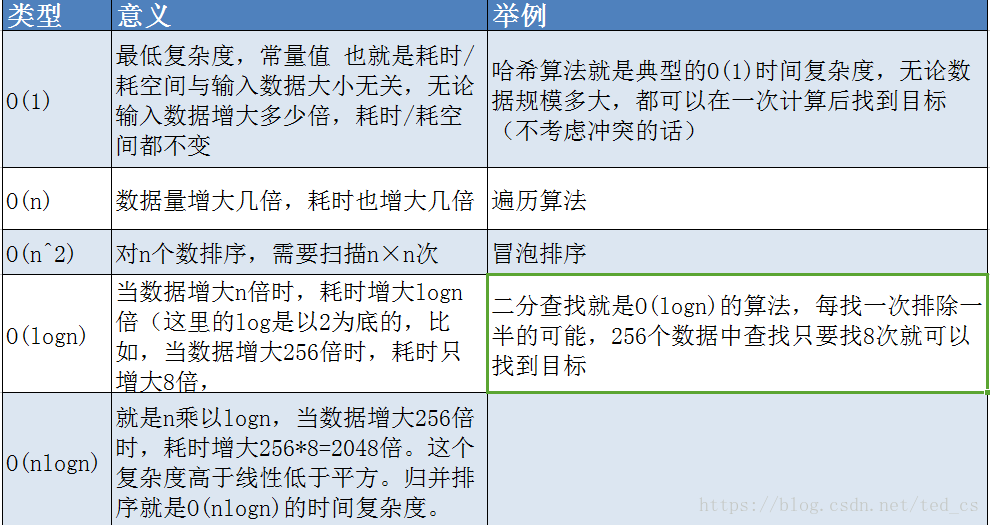
## IO和NIO的区别

参考资料：<https://www.cnblogs.com/aspirant/p/8630283.html>

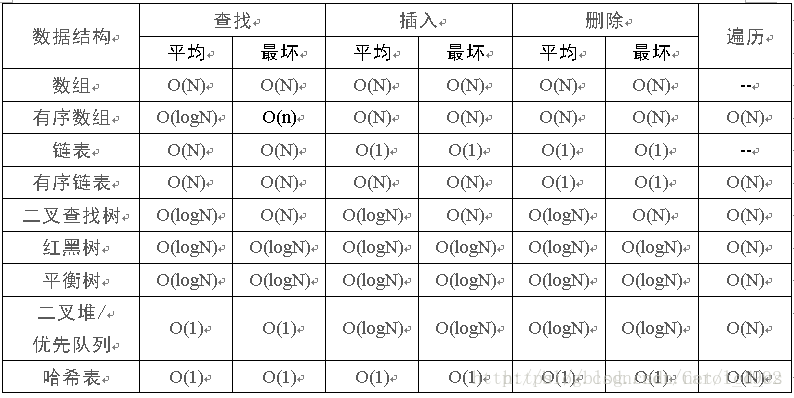
## 存储过程

<https://www.cnblogs.com/llq1214/p/11202743.html>

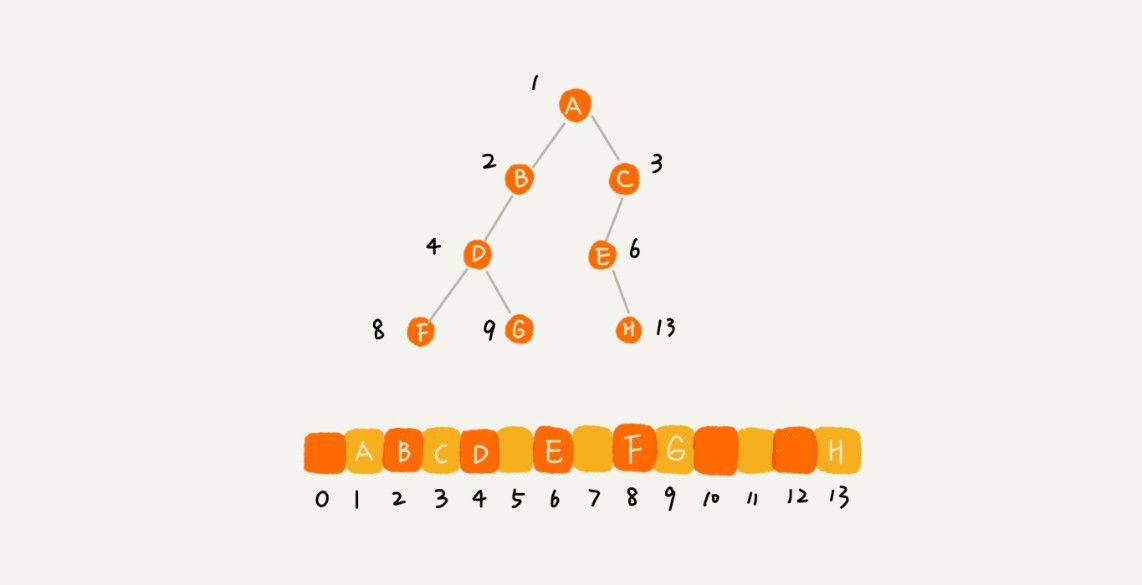
## java排序算法



1.插入排序



## 数据结构与算法



**通过上述可以看到，针对一般树来说，使用数组的方式存储树会浪费比较多的存储空间。但是针对下文会提到的满二叉树或者完全二叉树来说，数组存储的方式是最节省内存的一种方式。因为数组存储时，不需要再存储额外的左右子节点的指针。**

## 问题：

1. GC overhead limit exceeded填坑心得

**2. 经排查发现获取过期时间人像的sql 用的是**

**UNIX\_TIMESTAMP(“2099-1-1 11:00:00”) <= UNIX\_TIMESTAMP(now()) ;**

**查看了UNIX\_TIMESTAMP 这个函数有两种用法：**

**1 无参数调用：UNIX\_TIMESTAMP()； 。**

**返回值：自 1970-01-01 00:00:00 的到 当前时间 的秒数差**

**2 有参数调用：UNIX\_TIMESTAMP(date)；**

**其中date是一个指定日期。返回值：自 1970-01-01 00:00:00 与 指定时间的秒数差**

**不管两种哪一种用法，都是返回 自’1970-01-01 00:00:00’与指定时间的秒数差，最大值是Integer.MAX\_VALUE 也就是 2147483647秒**

**换算成日期 数据库时区如果是东八区的是 2038-01-19 11:14:07 如果是UTC 支持的是2038/1/19 3:14:7**

**也就是说超过2147483647 这个值时，UNIX\_TIMESTAMP 这个函数是不支持的，返回结果统一是0.**

**3.**

**高频面试题：**

