离职理由  
  
业务介绍  
准入  
下单  
签署合同  
支付首付  
车辆扣减   
车辆服务  
车辆交付  
  
  
融租公司配置系统   
  
融租公司路由 目的  
  
额度扣减 有没有什么问题 优化的空间？  
  
并发量大 也有场景 比如天猫订单同步   
参与开发业务功能  
什么问题？  
录下通话 复盘   
  
mysql  
隔离级别 幻读/可重复读  
数据量怎么样 sql优化   
btree 索引原理   
explain  
  
分布式服务涉及到中间件和问题  
dubbo 内部会重试 注意写入接口的幂等性  
分布式锁： redis的实现，有一些问题，过期，释放时如何保证不会误操作到其他锁定记录，如何做到可重入  
单进程内：同步锁 cas原理  
事务 传播机制   
spring 结合使用 嵌套  
多线程使用的时候要使用线程池，原因  
  
spring aop 动态代理

# Mysql

## 事务

1.

## 索引

1. 定义：是一种高效获取数据的存储结构，例如hash，二叉，红黑数，b+tree
2. 根据索引值快速获取对应数据的行号，物理存储地址
3. 索引的缺点：需要额外的存储空间；在添加索引时会对表加锁。一般在晚上添加索引，添加字段时，比如order表时，要在晚上11点之后。
4. 聚簇索引，非聚簇索引
5. 联合索引一定要有最左边的https://www.jianshu.com/p/fd781d6e1158
6. B+ 所有的数据都存在叶子节点，并且没个叶子节点都有指向下一个叶子节点的指针，形成了一个有序的链表，方便范围查询。比如说select \* from Table where id > 1 and id < 100; 当找到1后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提到了区间查询效率。

一般情况下b+树的深度一般是2-4层,所以逻辑上找到某一个键值的行记录最多需要2-4层的io操作。

单一节点存储更多元素，减少io

所有查询都到叶子节点，查询比较稳定。

https://baijiahao.baidu.com/s?id=1628226562020721873&wfr=spider&for=pc

## 优化

的

# Java

## 1.object的方法

其中clone方法，如果要使用一个类的clone方法，则必须重写java.lang.Cloneable接口。默认clone的对象是新对象，里面的属性并没有重新创建，可以自己实现clone的逻辑和深度。

|  |
| --- |
| public class TestClone implements Cloneable{  private User user;  public static void main(String[] args) throws CloneNotSupportedException {  TestClone test = new TestClone();  test.user = new User();  test.user.setName("test");  TestClone clone = (TestClone)test.clone();  System.out.println(test == clone);  System.out.println(clone.user == test.user);  }  @Override  protected Object clone() throws CloneNotSupportedException {  TestClone clone = new TestClone();  User user = new User( );  clone.user = user;  user.setName(this.user.getName());  return clone;  }  } |

## Synchronize

底层原理时，线程通过获取的对象的监视器，这个过程时阻塞的，只能有一个线程获得，而其他线程只能获锁线程释放后，才能获取，在Javap -c class文件后，时monitorenter和monitorexit这两个命令控制

<https://www.jianshu.com/p/2ba154f275ea>

## Hashmap

Jdk7并发情况下扩容造成的链上的死循环 ，扩容时由于多个线程操作操作同一个一个链上的数据，在进行数据放到新数组时，指针引用造成了环，在get时，形成自旋。

<https://www.jianshu.com/p/1e9cf0ac07f4>

## Aqs

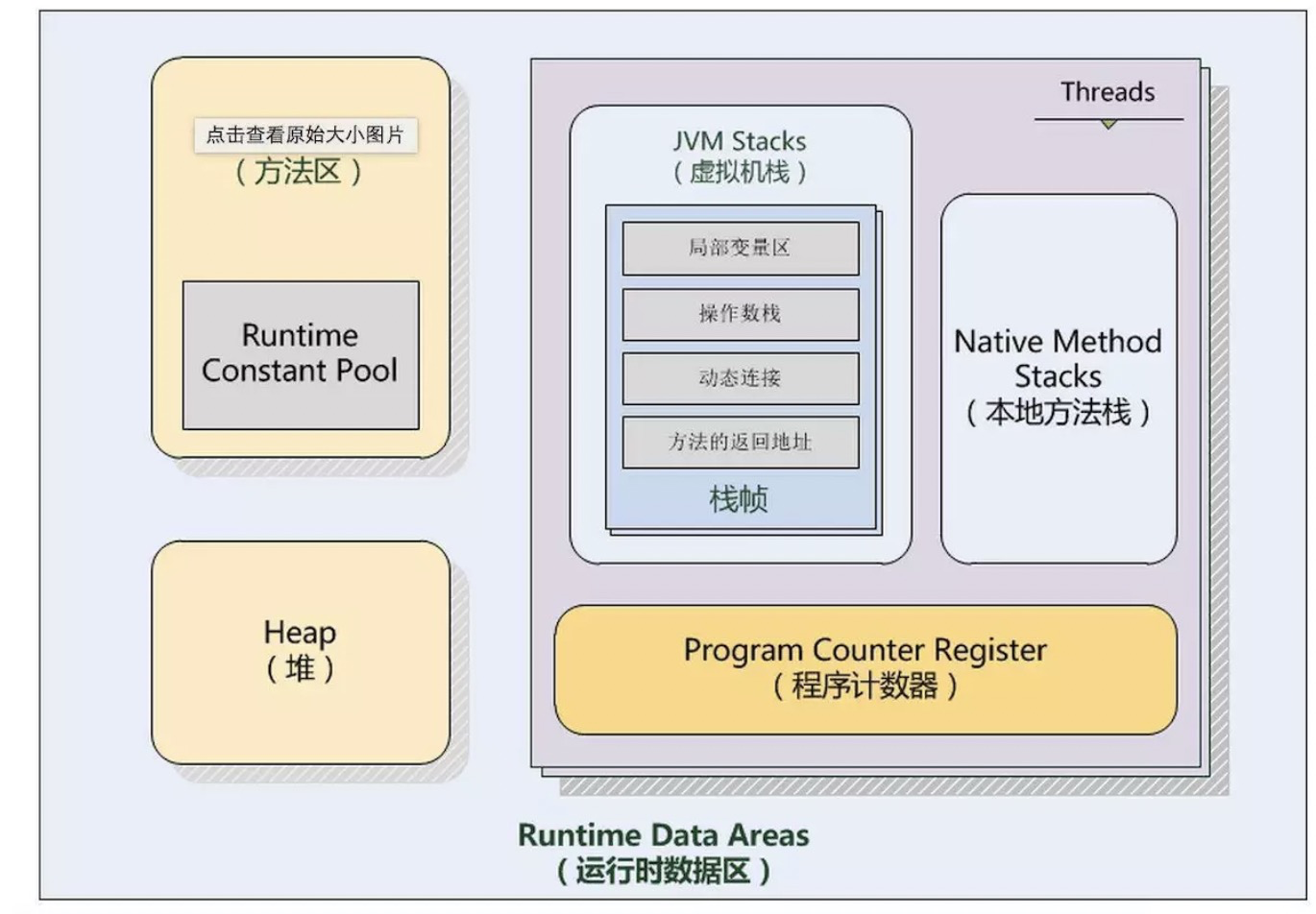
<https://www.jianshu.com/p/9d5cbadb6bc1>

<https://www.jianshu.com/p/e0fe0817b694>

# Jvm

## java内存模型

### 1.1运行时内存模型



程序计数器

线程私有，看作时当前线程执行时的字节码的行号执行器。java虚拟机是通过多线程轮流切换并非配处理器执行时间的方式实现的，一个处理器任意时刻都只有一条线程在执行，为了线程切换后恢复到切换前执行的字节码位置。

虚拟机栈

虚拟机栈描述的是Java方法执行的内存模型。每个方法在执行时会创建一个栈帧，用来存储局部变量表、操作数栈、动态链接、方法出口等信息。方法的调用和执行结束对应了一个栈帧在虚拟机栈中入栈和出栈的过程。

本地方法栈

本地方法栈为执行虚拟机使用到的native 方法服务，其他和虚拟机栈一样。Sun hotspot 虚拟机将本地方法栈和虚拟机合二为一。

堆

存放new出来的对象。线程共享

方法区

线程共享。存储虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

存放Class信息。

运行时常量池

运行时常量池时方法区的一部分。String类的intern方法。在jdk1.6后常量池中存放的是第一次出现的字符串的引用，复制对象引用到常量池。

|  |
| --- |
| String s = "123";  new String("123").intern() //这两种方式都可以将字符串设置到常量池中。 |

### 1.2对象

#### 对象创建

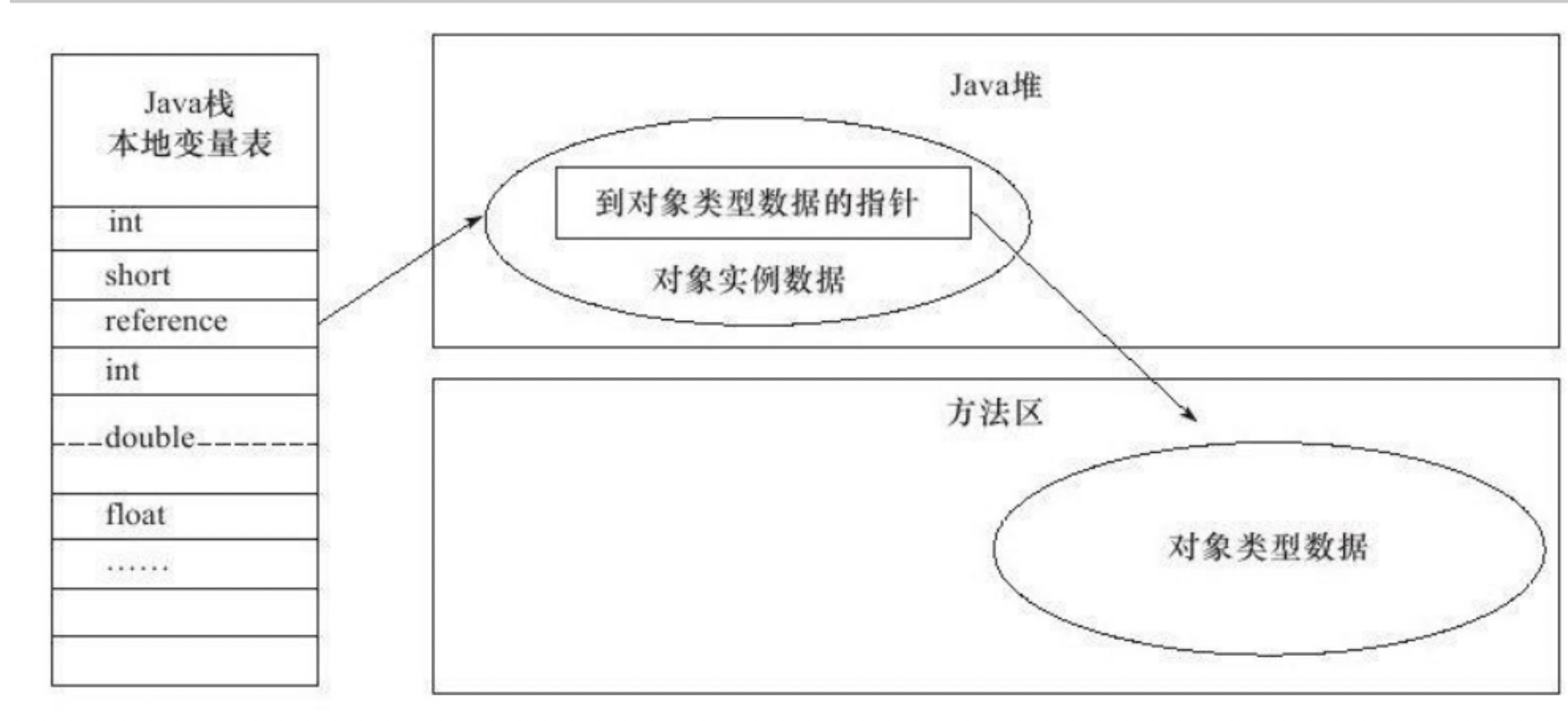
new关键自创建

对象的内存布局

对象{对象头、实例数据、对齐填充}

对象头{自身运行时数据【哈希码、gc分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳】;指向class的指针}

#### 对象的访问定位



### 1.3内存溢出

堆内存溢出

一直创建对象

栈内存溢出

一个时超过栈深，持续压栈操作。方法无限递归调用。

创建过多线程，造成oom。

#### 方法区内存溢出

字符串常量一直生成，string.intern（）方法。

通过cglib无限生成代理类造成内存溢出。

## 垃圾收集

### 判断对象是否已死

**引用计数算法**

每当引用指向这个对象时，对象的引用器就加1，引用失效时对象的引用器就减1.当为0时表示可以被回收。

不能解决对象之间互相引用的问题

**可达性分析算法**

通过GCroot的方法判断。如果当前对象没有一个gcroot到当前对象的引用链，则认为当前对象可以被回收

**gcroot包括哪些：**

方法栈中引用的对象

方法区中静态变量引用的对象

方法区中常量引用的对象

### 引用类型

**强引用**

直接object o = new object new出来的对象并指向的对引用。

**软引用**

软引用描述的是有用但非必须的对象。软引用引用的对象只有在垃圾回收完成后，内存依然不够时才会对软引用引用的对象进行第二次垃圾回收。

**弱引用**

虚引用描述的是非必须对象。

被弱引用引用的对象，只要发生垃圾回收，就会对这部分对象进行回收。

**虚引用**

一个被虚引用引用的对象不会对对象的生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被垃圾回收时收到一个系统通知。

### 方法区回收

主要回收无用的类和废弃的常量

无用的类必须满足：1.这个类所有的实例都被回收2.该类的classloader已经被回收3.该类对应的java.lang.Class对象没有在任何地方被引用。

废弃的常量：例如一个“abc”的字符串被推进了常量池，只有在当前jvm进程中没有任何“abc”时才会被回收。

### 垃圾收集算法

**标记清除**

**标记整理**

在对象被标记后，进行清理的过程中让存活的对象向一端移动，垃圾回收完成后会形成连续的内存空间

**复制算法**

将内存分成大小相等的两块，每次只使用其中的一块，在垃圾回收时将存活的对象复制到另一块内存上，然后一次性清理那个需要回收的内存区域。不会产生空间碎片。

**分代收集**

新生代采用复制算法，老年代采用标记清除或者标记整理

### 垃圾算法实现

### 垃圾收集器

**serial收集器**

新生代收集器，采用复制算法。

单线程完成垃圾收集工作，在垃圾收集时，必须暂停其他所有的工作线程。

**Serial old收集器**

老年代收集器，采用标记整理的算法。

Serial old是serial收集器的老年代版本。也是采用单线程收集，在收集过程中必须暂停所有的用户线程。

**parNew 收集器**

新生代收集器，采用复制算法。

serial的多线程版本，可以多个线程完成垃圾收集工作。

在垃圾收集期间，暂停所有的用户线程。

在单cpu情况下由于线程切换，性能不如serial收集器。

默认开启的垃圾收集线程等于cpu的数量，可以使用-XX:ParallerGCThreads 参数指定

**Parallel Scavenge 收集器**

新生代收集器，采用复制算法

多个线程完成收集工作和parnew 类似

Parallel scavenger收集器目标是达到一个可控的吞吐量。

吞吐量=运行用户代码时间/运行用户代码时间+垃圾收集时间

可以用-xx:MaxGCPauseMillis设置最大垃圾收集停顿时间；用-XX:GCTimeRatio设置吞吐量大小。

gc停顿时间的缩短需要牺牲吞吐量和新生代空间换取。

可以设置-XX:+userAdaptiveSizePolicy这个开关参数，开启自适应调节策略。不需要指定新生代的大小（-xmn）、eden与survivor区的比例、晋升老年代的对象大小等细节参数。

**Parallel old收集器**

老年代收集器，采用标记整理算法。

Parallel old是parallel scavenge的老年代版本。

吞吐量优先、多个线程完成垃圾收集工作

**cms收集器**

老年代收集器，采用标记-清除算法。是一种获取最短停顿时间为目标的收集器。

初始标记-并发标记-重新标记-并发清除

初始标记和重新标记会暂停其他工作线程。

初始标记只是标记一下gc root能直接关联到的对象，并发标记似乎gcroot tracing的过程。重新标记是为了标记在并发标记阶段程序继续运行产生变动的一部分记录，并发清除阶段是和用户线程一起执行的，这个时间段最长，降低了用户线程的停顿时间。

缺点：

**产生浮动垃圾，**在并发清除阶段，用户线程还在执行，这期间产生的垃圾。由于在并发阶段用户线程还在执行，老年代就需要额外的存储空间，如果存储空间不够，就会出现“concurrent mode failure”失败，这时会启用serial old收集器。

**产生空间碎片。**由于采用了标记-清除的算法，所以在并发清除阶段会产生空间碎片。cms收集器为了解决这个问题，在要进行full gc时会开启空间碎片整理过程，这个过程也是需要暂停其他工作线程的。通过-xx:CmsFullGcsBeforeCompaction指定几次不压缩的full gc后，跟着来一次带压缩的full gc 默认是0，每次full gc也就是并发清除完成后都要碎片整理。

**G1收集器**

对整个堆进行收集，综合采用标记整理，局部两个region之间采用复制算法。

G1收集器将整个堆划分为多个大小相等的独立区域region，仍保留新生代老年代的感念，但都是region的一部分，不需要联系。

可预测停顿，并行与并发，分代收集，空间整合

为了避免全堆扫描，将每个region关联一个remembered set。每次有reference类型的数据写操作时，检查是否其他的region有引用整个这个对象，如果有将引用信息记录到被引用对象的remembered set 里面。

收集过程也是分为4部分：

初始标记-》标记gcroot 能直接关联到的对象，需要停顿其他工作线程，时间短

并发标记-》对堆中的对象进行可达性分析，找出存活的对象，和工作线程并发执行，时间长。

最终标记-》记录在并发标记期间产生变动的记录，将这段时间对象变动记录在线程的remembered set log 里面，并合并到remembered set里面。这段期间需要暂停其他工作线程，但是收集线程并行执行。

筛选回收-》将各个region的回收价值和成本进行排序，根据用户所期望的gc停顿时间来制定回收计划。这段时间也是需要暂停其他用户线程。

### 内存分配和回收策略

对象在分配时，在eden区分配，当eden区内存不足时，触发minor gc，将eden区和一块survivor中还存活的对象复制到另一块survivor内存中。

如果对象在分配时，遇到**大对象**时（数组，字符串）超过-XX:PretenureSizeThreshold这个设置的值时，直接分配到老年代。

**长期存活的对象**，每次gc后这个对象的age就加1，超过-XX:MaxTenuringThreshold的设置的值就放到老年代。

**空间分配担保**：新生代中有大量的对象存活，survivor空间不够，当出现大量对象在MinorGC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代.只要老年代的连续空间大于新生代对象的总大小或者历次晋升的平均大小，就进行Minor GC，否则FullGC

在每次minor gc之前，虚拟机会先检查老年代最大可用的连续空间时否大于新生代所有对象的总空间，如果大于就说明时担保成功，可以确定minor gc 是成功的，如果小于就要先进性full gc，清除老年代死去的对象。

# Dubbo

<https://www.cnblogs.com/h-c-g/p/11209756.html>

# zookeeper

<https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_label4>

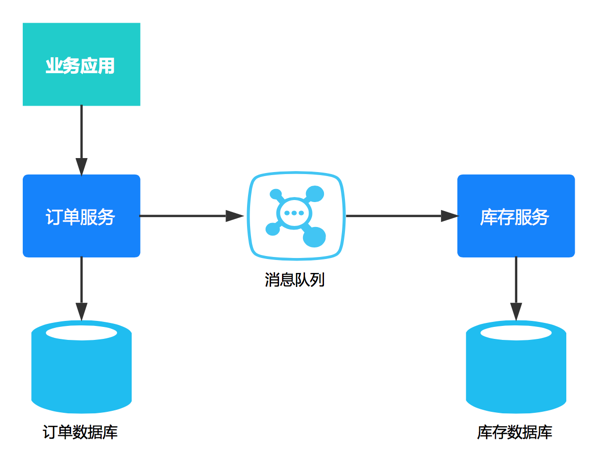
# 分布式



<https://www.cnblogs.com/jiangyu666/p/8522547.html>

3.3 基于消息的最终一致性方案

消息一致性方案是通过消息中间件保证上、下游应用数据操作的[一致性](https://segmentfault.com/a/1190000011479826)。基本思路是将本地操作和发送消息放在一个事务中，保证本地操作和消息发送要么两者都成功或者都失败。下游应用向消息系统订阅该消息，收到消息后执行相应操作。



消息方案从本质上讲是将分布式事务转换为两个本地事务，然后依靠下游业务的重试机制到最终一致性。基于消息的最终一致性方案对应用侵入性也很高，应用需要进行大量业务改造，成本较高。

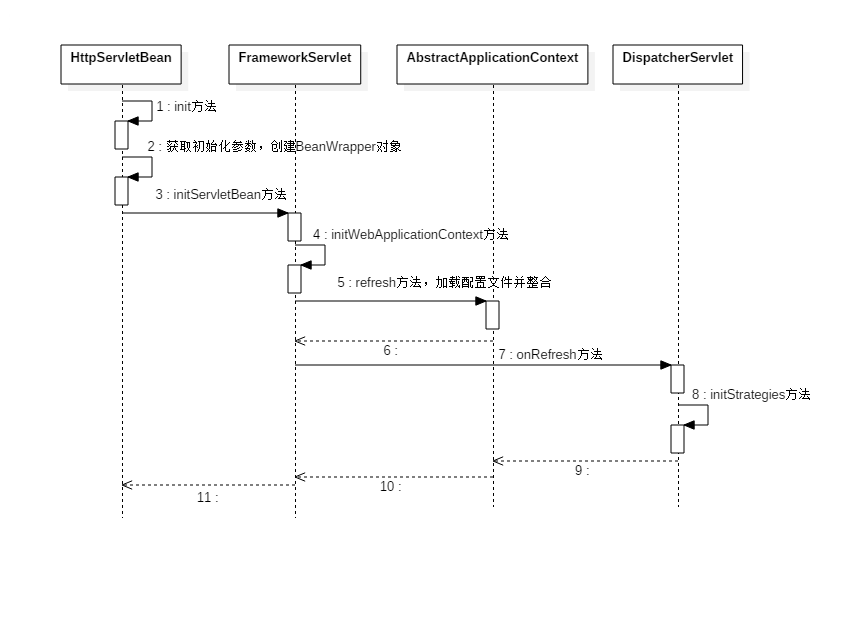
# 题

<https://blog.csdn.net/weixin_43495390/article/details/86533482>

Fifo lru lfu

<https://blog.csdn.net/notOnlyRush/article/details/80158703>

# Spring

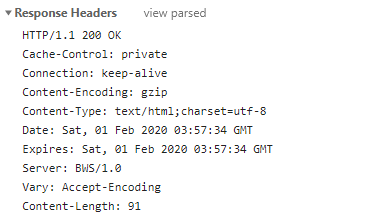


# http

请求行（request line）、请求头部（header）、空行和请求数据四个部分组成。

HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文





|  |
| --- |
| HTTP/1.1 200 OK  Cache-Control: private  Connection: keep-alive  Content-Encoding: gzip  Content-Type: text/html;charset=utf-8  Date: Sat, 01 Feb 2020 03:57:34 GMT  Expires: Sat, 01 Feb 2020 03:57:34 GMT  Server: BWS/1.0  Vary: Accept-Encoding  Content-Length: 91 |

|  |
| --- |
| GET /home/msg/data/personalcontent?callback=jQuery1102038196195709802794\_1580529448263&num=8&\_req\_seqid=af35b346001685ab&sid=1427\_21108\_30479&\_=1580529448264 HTTP/1.1  Host: www.baidu.com  Connection: keep-alive  Accept: text/javascript, application/javascript, application/ecmascript, application/x-ecmascript, \*/\*; q=0.01  X-Requested-With: XMLHttpRequest  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36  Sec-Fetch-Site: same-origin  Sec-Fetch-Mode: cors  Referer: https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E4%BA%8B%E5%8A%A1&rsv\_spt=1&rsv\_iqid=0x8824d18d0015ddb3&issp=1&f=3&rsv\_bp=1&rsv\_idx=2&ie=utf-8&tn=baiduhome\_pg&rsv\_enter=1&rsv\_dl=ih\_0&rsv\_sug3=1&rsv\_sug1=1&rsv\_sug7=001&rsv\_sug2=1&rsp=0&rsv\_sug9=es\_2\_1&rsv\_sug4=4583&rsv\_sug=9  Accept-Encoding: gzip, deflate, br  Accept-Language: zh-CN,zh;q=0.9  Cookie: BAIDUID=A98CE89B3003B2A954199BBA3506D6E0:FG=1; BIDUPSID=A98CE89B3003B2A954199BBA3506D6E0; PSTM=1567683616; BDUSS=QtRkVsd0xhRmRkZDJqbDZRYkx1aFpPN2U5bjJ2OHl4aDR-VmotSzBVWm1kekZlRVFBQUFBJCQAAAAAAAAAAAEAAAAe~5pUyqnB-L2jAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGbqCV5m6gleQj; MCITY=-%3A; BD\_UPN=12314753; BDORZ=B490B5EBF6F3CD402E515D22BCDA1598; COOKIE\_SESSION=62509\_1\_6\_3\_11\_2\_0\_0\_5\_2\_0\_0\_0\_0\_0\_0\_1579354137\_1579238483\_1579353134%7C9%23347817\_29\_1579238481%7C9; BD\_HOME=1; H\_PS\_PSSID=1427\_21108\_30479; delPer=0; BD\_CK\_SAM=1; PSINO=1; BDRCVFR[feWj1Vr5u3D]=mk3SLVN4HKm; sugstore=1; H\_PS\_645EC=831fw66gvJadb5mILRWufhznU2xZboQH0aXxtvgVMGQJdLGUVHVSt53o%2B5Kne5Hc3XuP |

# https

|  |
| --- |
| 服务器端的公钥和私钥，用来进行非对称加密  客户端生成的随机密钥，用来进行对称加密  一个HTTPS请求实际上包含了两次HTTP传输，可以细分为8步。  1.客户端向服务器发起HTTPS请求，连接到服务器的443端口  2.服务器端有一个密钥对，即公钥和私钥，是用来进行非对称加密使用的，服务器端保存着私钥，不能将其泄露，公钥可以发送给任何人。  3.服务器将自己的公钥发送给客户端。  4.客户端收到服务器端的公钥之后，会对公钥进行检查，验证其合法性，如果发现发现公钥有问题，那么HTTPS传输就无法继续。严格的说，这里应该是验证服务器发送的数字证书的合法性，关于客户端如何验证数字证书的合法性，下文会进行说明。如果公钥合格，那么客户端会生成一个随机值，这个随机值就是用于进行对称加密的密钥，我们将该密钥称之为client key，即客户端密钥，这样在概念上和服务器端的密钥容易进行区分。然后用服务器的公钥对客户端密钥进行非对称加密，这样客户端密钥就变成密文了，至此，HTTPS中的第一次HTTP请求结束。  5.客户端会发起HTTPS中的第二个HTTP请求，将加密之后的客户端密钥发送给服务器。  6.服务器接收到客户端发来的密文之后，会用自己的私钥对其进行非对称解密，解密之后的明文就是客户端密钥，然后用客户端密钥对数据进行对称加密，这样数据就变成了密文。  7.然后服务器将加密后的密文发送给客户端。  8.客户端收到服务器发送来的密文，用客户端密钥对其进行对称解密，得到服务器发送的数据。这样HTTPS中的第二个HTTP请求结束，整个HTTPS传输完成。  链接：https://www.jianshu.com/p/14cd2c9d2cd2 |

# 跨域

|  |
| --- |
| 为**浏览器的同源策略**所导致的。所谓同源是指"协议+域名+端口"三者相同  preview  1.1原生实现  在www.a.com域名写下如下代码，去请求www.b.com域名的数据  <script>  var script = document.creatElement('script');  script.type = 'text/javascript';  script.src = 'http://www.b.com/getdata?callback=demo';    function demo(res){  console.log(res);  }  </script>  这里，我们利用动态脚本的src属性，变相地发送了一个http://www.b.com/getdata?call...。这时候，b.com页面接受到这个请求时，如果没有JSONP,会正常返回json的数据结果，像这样：{ msg: 'helloworld' },而利用JSONP,服务端会接受这个callback参数，然后用这个参数值包装要返回的数据：demo({msg: 'helloworld'});  这时候，如果a.com的页面上正好有一个demo 的函数：  function demo(res){  console.log(res);  }  当远程数据一返回的时候，随着动态脚本的执行，这个demo函数就会被执行。  1.2 jquery ajax请求实现  $.ajax({  url:'http://www.b.com/getdata',  type:'get',  dataType: 'jsonp', // 请求方式为jsonp  jsonpCallback: 'demo', // 自定义回调函数名  data: {}  });  jsonp缺点：只能使用get请求，不推荐使用 |

# 分布式事务

|  |
| --- |
| 分布式系统面试题：分布式事务解决方案？  面试题  分布式事务了解吗？你们是如何解决分布式事务问题的？  面试题剖析  一般来说，分布式事务的实现主要有以下 5 种方案：  1.XA 方案  TCC 方案  本地消息表  可靠消息最终一致性方案  最大努力通知方案  两阶段提交方案/XA方案  所谓的 XA 方案，即：两阶段提交，有一个事务管理器的概念，负责协调多个数据库（资源管理器）的事务，事务管理器先问问各个数据库你准备好了吗？如果每个数据库都回复 ok，那么就正式提交事务，在各个数据库上执行操作；如果任何其中一个数据库回答不 ok，那么就回滚事务。  这种分布式事务方案，比较适合单块应用里，跨多个库的分布式事务，而且因为严重依赖于数据库层面来搞定复杂的事务，效率很低，绝对不适合高并发的场景。如果要玩儿，那么基于 Spring + JTA 就可以搞定，自己随便搜个 demo 看看就知道了。  这个方案，我们很少用，一般来说某个系统内部如果出现跨多个库的这么一个操作，是不合规的。我可以给大家介绍一下， 现在微服务，一个大的系统分成几十个甚至几百个服务。一般来说，我们的规定和规范，是要求每个服务只能操作自己对应的一个数据库。  如果你要操作别的服务对应的库，不允许直连别的服务的库，违反微服务架构的规范，你随便交叉胡乱访问，几百个服务的话，全体乱套，这样的一套服务是没法管理的，没法治理的，可能会出现数据被别人改错，自己的库被别人写挂等情况。  如果你要操作别人的服务的库，你必须是通过调用别的服务的接口来实现，绝对不允许交叉访问别人的数据库。  ​  2.TCC 方案  TCC 的全称是：Try、Confirm、Cancel。  Try 阶段：这个阶段说的是对各个服务的资源做检测以及对资源进行锁定或者预留。  Confirm 阶段：这个阶段说的是在各个服务中执行实际的操作。  Cancel 阶段：如果任何一个服务的业务方法执行出错，那么这里就需要进行补偿，就是执行已经执行成功的业务逻辑的回滚操作。（把那些执行成功的回滚）  这种方案说实话几乎很少人使用，我们用的也比较少，但是也有使用的场景。因为这个事务回滚实际上是严重依赖于你自己写代码来回滚和补偿了，会造成补偿代码巨大，非常之恶心。  比如说我们，一般来说跟钱相关的，跟钱打交道的，支付、交易相关的场景，我们会用 TCC，严格保证分布式事务要么全部成功，要么全部自动回滚，严格保证资金的正确性，保证在资金上不会出现问题。  而且最好是你的各个业务执行的时间都比较短。  但是说实话，一般尽量别这么搞，自己手写回滚逻辑，或者是补偿逻辑，实在太恶心了，那个业务代码很难维护。  ​  3.本地消息表  本地消息表其实是国外的 ebay 搞出来的这么一套思想。  这个大概意思是这样的：  A 系统在自己本地一个事务里操作同时，插入一条数据到消息表；  接着 A 系统将这个消息发送到 MQ 中去；  B 系统接收到消息之后，在一个事务里，往自己本地消息表里插入一条数据，同时执行其他的业务操作，如果这个消息已经被处理过了，那么此时这个事务会回滚，这样保证不会重复处理消息；  B 系统执行成功之后，就会更新自己本地消息表的状态以及 A 系统消息表的状态；  如果 B 系统处理失败了，那么就不会更新消息表状态，那么此时 A 系统会定时扫描自己的消息表，如果有未处理的消息，会再次发送到 MQ 中去，让 B 再次处理；  这个方案保证了最终一致性，哪怕 B 事务失败了，但是 A 会不断重发消息，直到 B 那边成功为止。  这个方案说实话最大的问题就在于严重依赖于数据库的消息表来管理事务啥的，会导致如果是高并发场景咋办呢？咋扩展呢？所以一般确实很少用。  ​  4.可靠消息最终一致性方案  这个的意思，就是干脆不要用本地的消息表了，直接基于 MQ 来实现事务。比如阿里的 RocketMQ 就支持消息事务。  大概的意思就是：  A 系统先发送一个 prepared 消息到 mq，如果这个 prepared 消息发送失败那么就直接取消操作别执行了；  如果这个消息发送成功过了，那么接着执行本地事务，如果成功就告诉 mq 发送确认消息，如果失败就告诉 mq 回滚消息；  如果发送了确认消息，那么此时 B 系统会接收到确认消息，然后执行本地的事务；  mq 会自动定时轮询所有 prepared 消息回调你的接口，问你，这个消息是不是本地事务处理失败了，所有没发送确认的消息，是继续重试还是回滚？一般来说这里你就可以查下数据库看之前本地事务是否执行，如果回滚了，那么这里也回滚吧。这个就是避免可能本地事务执行成功了，而确认消息却发送失败了。  这个方案里，要是系统 B 的事务失败了咋办？重试咯，自动不断重试直到成功，如果实在是不行，要么就是针对重要的资金类业务进行回滚，比如 B 系统本地回滚后，想办法通知系统 A 也回滚；或者是发送报警由人工来手工回滚和补偿。  这个还是比较合适的，目前国内互联网公司大都是这么玩儿的，要不你举用 RocketMQ 支持的，要不你就自己基于类似 ActiveMQ？RabbitMQ？自己封装一套类似的逻辑出来，总之思路就是这样子的。  ​  5.最大努力通知方案  这个方案的大致意思就是：  系统 A 本地事务执行完之后，发送个消息到 MQ；  这里会有个专门消费 MQ 的最大努力通知服务，这个服务会消费 MQ 然后写入数据库中记录下来，或者是放入个内存队列也可以，接着调用系统 B 的接口；  要是系统 B 执行成功就 ok 了；要是系统 B 执行失败了，那么最大努力通知服务就定时尝试重新调用系统 B，反复 N 次，最后还是不行就放弃。  你们公司是如何处理分布式事务的？  如果你真的被问到，可以这么说，我们某某特别严格的场景，用的是 TCC 来保证强一致性；然后其他的一些场景基于阿里的 RocketMQ 来实现分布式事务。  你找一个严格资金要求绝对不能错的场景，你可以说你是用的 TCC 方案；如果是一般的分布式事务场景，订单插入之后要调用库存服务更新库存，库存数据没有资金那么的敏感，可以用可靠消息最终一致性方案。  友情提示一下，RocketMQ 3.2.6 之前的版本，是可以按照上面的思路来的，但是之后接口做了一些改变，我这里不再赘述了。  当然如果你愿意，你可以参考可靠消息最终一致性方案来自己实现一套分布式事务，比如基于 RocketMQ 来玩儿。  转载于:https://juejin.im/post/5cd69b1bf265da03b4462fd0 |