## 集合

Collection

ArrayList、LinkedList、Vector（线程安全）

ArrayList：底层是数组，初始化时，数据量是0，当add时候，默认变成10，扩容机制是上一次容量的1.5倍，特性查询比较快（通过脚标去查），删除效率低

LinkedList：底层是头结点和尾节点的双向链表，提供了2个方法，头插法LinkedFirst，尾插法LinkedLast，特性非常适用于经常增加和删除的场景，查询效率慢(他会对各个节点进行比较得到查询数据)

Vector：线程安全，底层和arrayList一样是数组，和ArrayList的区别在于，大部分方法都是被synchronized关键字所修饰，扩容机制是上一次容量的2倍

### set

HashSet

### map

HashMap、ConcurrentHashMap线程安全、HashTable线程安全

hashmap：

1.7版本--底层是数组+单链表

1.8版本--数组+单链表+红黑树

1、单链表和红黑色之间的转换，单链表长度>=8，并且哈希桶数组长度>=64，会将单链表转换成红黑树形式进行存储，红黑树节点数量<=6，会重新转换成单链表

2、hash桶数组默认是16个，阈值默认是0.75，当插入16\*0.75=12个元素的时候(简单理解为，已经有12个数组被占用)，就会触发扩容

3、扩容机制是上一次容量的2倍，然后重新进行新的hash运算(hash值&（length -1）),填充到新的hash桶数组

4、链表扩容的数据迁移是将相同下标的节点组合在一起，放到新数组里面；红黑色是直接转移根节点

### ConcurrentHashMap

线程安全，实现分段锁表

1.7版本--底层分片数组，为了保障线程安全，用到了Segment锁，他继承于ReentrantLock

1.8版本--数组+单链表+红黑树，使用synchronized+CAS操作来保证线程安全，里面有2把锁，第一把锁在数据插入时，防止插入时数据并发，第二把锁在链表树化，防止红黑色里面根节点旋转导致根节点转移。

### HashTable

线程安全，全局锁表

## CAS

1、compare and swap的缩写，中文翻译成比较并替换

2、CAS(内存值V，旧的预期值A，要修改的新值B)

3、相当于轻量级的加锁过程，在并发量不是特别大的情况下，当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做

CAS存在问题：

1、CPU开销大；自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销

2、只能保证一个共享变量原子操作；对多个共享变量操作时，循环CAS就无法保证操作的原子性

3、ABA-标志位时间戳；如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

## synchronized

synchronized关键字在编译之后会在同步代码块前后加入2个指令：

1、monitorenter(锁)

2、monitorexit(解锁)

### 偏向锁加锁：

1、将线程id写入到锁对象的对象头中

2、其他线程进入并获取锁之后，锁对象会结束偏向状态，膨胀成轻量锁或重量锁，有资源竞争重量锁，无资源竞争轻量锁

### 轻量级锁加锁：

1、执行monitorenter(锁)

2、新线程进来会在虚拟栈中为当前线程开辟一个空间叫Lock Record

3、将锁对象的Mark Word写入里面

4、CAS尝试将锁对象的Mark Word轻量级锁指针指向当前Lock Record（作用是让其他线程知道，该object monitor已被占用），成功获取锁，失败则挂起线程

5、然后lock record里的owner指针指向object mark word（作用是为了在接下里的运行过程中，识别哪个对象被锁住了）

6、执行monitorexit(解锁)

### 重量级锁同步代码块加锁：synchronized(o)

1、执行monitorenter(锁)

2、偏向锁膨胀成重量锁的过程，偏向锁会先进行撤销偏向成无锁01，此时锁对象的监视器锁monitor为0

3、新的线程获取锁，锁对象的monitor监视器锁会+1（锁支持重入），当前线程成为monitor的owner（所有者），并将monitor的指针写入到锁对象头

4、修改锁对象标志位为10，此时锁就膨胀成重量锁

5、若其他线程已经占有monitor的所有权，那么当前尝试获取monitor的所有权的线程会被阻塞，直到monitor的进入数变为0，才能重新尝试获取monitor的所有权。

6、执行monitorexit(解锁)

7、同步代码块执行完，加了几次锁释放几次，将monitor清零，完成释放锁

### 重量级锁同步方法加锁：public synchronized void test()

JVM使用ACC\_SYNCHRONIZED标识来实现，即JVM通过在方法访问标识符(flags)中加入ACC\_SYNCHRONIZED来实现同步功能。

## volatile关键字

CPU和内存之间的线程效率是差好多数量级的，但为了保证他们之间的计算，不影响CPU的计算，中间会有好多LLV缓存，我们线程在这个缓存中去工作，首先他取数据会在主内存取到工作内存中去计算，计算完之后在传给主内存，此时就出现了多线程间变量的可见性，java为了解决这种问题，他就抽象出了JMM这种模型，被volatile修饰的变量，保证变量能在多线程之间的可见性，也就是变量被更新了，能立刻刷新到主内存中，其他线程能立刻取到最新的变量数据，这是总线嗅探机制，但不保证原子性，简单的说就是保证线程修改了变量能及时被其他线程看到，但不保证多个线程改变同一个变量之后数据的完整。

如何保障数据原子性：使用synchronized关键字、ReentrantLock加锁、AtomicXXX类（多个AtomicXXX类连续调用不能构成原子性）来保证数据原子性

### 总线嗅探机制

每个处理器通过嗅探在总线上传播的数据来检查自己的缓存值是不是过期了，如果处理器发现自己缓存行对应的内存地址呗修改，就会将当前处理器的缓存行设置无效状态

### 总线风暴：

由于volatile的mesi缓存一致性协议需要不断的从主内存嗅探和cas不断循环无效交互导致总线带宽达到峰值解决办法：部分volatile和cas使用synchronize

mesi缓存一致性协议：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。

资料：https://zhuanlan.zhihu.com/p/137193948

## JMM内存模型

JMM内存模型Java Memory Model是为了屏蔽各种硬件和操作系统对内存访问的差异性，简单理解就是为了保证程序在不同的操作系统和硬件，执行的结果是一样的。

Java的一切可见性问题都是因为编译器优化、内存优化、指令优化导致，硬件的底层已经保证了可见性，可以理解Java不存在可见性问题（伪可见性）。

https://hllvm-group.iteye.com/group/topic/34932#post-232535

### JMM的happends-before原则

1、如果操作A和操作B属于同一个线程，那么说明操作A的操作对操作B可见

2、unlock hb lock 线程A的unlock操作，在线程B的lock操作前操作

3、如果操作A是对volatile变量的写操作，操作B是对同一个变量的读操作，那么操作Ahappends-before操作B

4、happends-before具有传递性，操作A hb 操作B，操作B hb 操作C，操作A hb 操作C

## SQL调优

### Sql优化

1、首先考虑where及order条件建立索引

2、少使用is null处理，用=0处理

3、少使用!= <>判断

4、少使用or做条件，用union all做数据合并

5、少使用in/not in，能使用betwen就使用

6、少使用%在前的模糊查询

7、少使用对字段进行表达式操作num/2=100 --> num=100\*2

8、少使用函数

9、不要写一些没意义的查询，比如1=1

10、用exists代替in

11、索引不是越多越好，虽然查询快，但是插入和修改会降低

12、尽量使用数字型字段，因为字符需要对每个字段进行比较

13、尽量用varchar代替char

14、少使用\*，用具体字段和删除临时表

16、大数据备份用select into 新表名 from 旧表名代替 create table

17、数据超过1万条数据，考虑改写脚本

### 索引选择

1. 没主键，需要创建聚族索引
2. 有主键，查询还是慢，创建非聚族索引
3. 非聚族索引规则：
4. 尽量使用where、order by、group by后面的字段来创建
5. 查询最频繁的一列放在左边
6. 查询重复值最少的放在左边
7. 组合索引的字段尽量不要超过3个

Sql调优最基本的是表要有主键，因为有主键的表MySQL会创建聚族索引，聚族索引的好处是它的主键和数据行是在一行的，在explain查询语句的时候会发现，它的type级别是const，这是很高的一种级别。当有主键的时候，如果一条SQL语句很慢，可以去查看他是否创建了相应的索引，建立索引要尽量选择where、order by、group by后面的字段或者join in连接的字段作为索引列，这些索引列也要排序，要符合最左匹配原则，你选的时候要根据他们的索引选择器，你的非重复的数据行和重复的数据行中间的排列，大的放左，小的放右这样的形式，这种多列的索引列，要去建立联合索引而非单个索引。

如果数据查询的非常频繁，可以考虑给它使用覆盖索引，覆盖索引是可以直接在索引中查询到数据，相对来说还是很快的。

聚族索引（数据需要确保唯一）：它确切来说应该是一种数据结构，它是将它的主键和它的数据行放在一块的，找到了索引就找到了数据。

非聚族索引（数据不需要确保唯一）：就是我们说的普通索引，他的索引存放的是叶子节点，叶子节点包含他的主键、索引列、事务ID等信息，它在查询的时候，其实是通过它的索引列查询它的主键，在通过主键去回表查询，还是通过聚族索引查的，这样的速度会相对慢点

### 事务

脏读：一个事务读取了另一个事务改写但未提交的数据，如果这些数据被回滚，则读到的数据时无效的。

不可重复读：在同一事务中，多次读取同一数据返回的结果有所不同。在事务1中，第一次读到了A记录数据，中间事务2修改了A记录数据，第二次读到的A记录数据和第一次读到的A记录数据不一致。

幻读：一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发生有些原来没有的记录。事务1修改所有A表数据，事务2增加了A表数据，事务1的用户就会发现A表还有未修改的数据。

InnoDB的特性就是它支持事务，支持事务的时候，为了保证事务的并发度，它提供了一个隔离级别，有4种隔离级别：

1、Read Uncommitted：结果什么都不能保证

2、Read Committed：解决事务之间的脏读，但是不能保证不可重复读和幻读

3、Repeatable Read：默认事务，保证不发生脏读、不可重复读，不保证幻读

4、Serializable：事务最高级别，串行化，一个个事务串行排列。事务一个接一个执行，等待前一个事务执行完，后面的事务才可以顺序执行，保证不发生脏读、不可重复读、幻读。

## 线程池的运行机制

线程池他是有一个核心线程数，当线程运行的时候，如果没有设置成预启动加载，首发线程数为0，当提交一个新任务的时候，他会首先建立一个核心线程去执行任务，如果此时一直来任务，之前线程没有执行完，那么就会一直创建核心线程，当达到最大核心线程数时，如果都还在忙，那么此时就会放到BlockingQueue里面作为节点，如果BlockingQueue队列也放满了，而却核心线程都还在忙，那就会去建立新线程，它叫做非核心线程，若一直创建，数量达到非核心线程数max access，此时就会触发一个拒绝策略。

线程构造，在创建线程池的时候，需要提供线程的threadFactory，一定要指定他名称，并且可以设置为守护线程，其他线程执行完，主线程也执行完，JVM才会停止运行。

### JDK内置了四种拒绝策略

1. AbortPolicy，直接抛出异常，这是默认策略
2. DiscardPolicy，悄无声息的丢弃任务
3. DiscardOldestPolicy，丢弃最早的未执行任务
4. CallerRunsPolicy，用调用者所在的线程来执行任务，这种方式会影响新任务提交速度

### 阻塞队列

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue ：一个由链表结构组成的无界阻塞队列，也可以理解为有界队列，因为他的默认容量是Integer.MAX\_VALUE。

PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个支持延时获取元素的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列

## Spring

### Spring AOP代理方式

1、如果代理类有接口，走的就是JDK的动态代理

2、如果没有，则使用cglib字节码技术去创建代理对象

JDK动态代理：主要是使用了Java反射中的两个类，一个是Proxy，一个是InvocationHandler，它是通过Proxy. newProxyInstance来创建代理对象，然后通过反射invoke去执行那个方法。

### Spring事务隔离级别和传播行为

Spring提供了一个注解，叫做Transactional，并且提供了事务的7个传播行为和5个隔离级别。

Spring事务5个隔离级别

Spring事务隔离级别只比数据库多1个默认的Default，数据库使用什么默认事务隔离级别Spring就用什么，Mysql默认采用REPEATABLE\_READ隔离级别，Oracle默认采用READ\_COMMITTED隔离级别。

Spring事务7个传播行为

事务传播行为（propagation behavior）指的就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行，解决业务层方法之间的相互调用的问题。

案例：service1、service2有事务。同时调用service1的一个方法和service2的一个方法，才能完成一个业务。

1、Propagation.REQUIRED（默认）

如果当前存在事务，则加入该事务，如果当前不存在事务，则创建一个新的事务。

2、Propagation.SUPPORTS

如果当前存在事务，则加入该事务；如果当前不存在事务，则不使用事务。

3、Propagation.MANDATORY

如果当前存在事务，则加入该事务；如果当前不存在事务，则抛出异常。

4、Propagation.REQUIRES\_NEW

重新创建一个新的事务，如果当前存在事务，延缓当前的事务。

5、Propagation.NOT\_SUPPORTED

以非事务的方式运行，如果当前存在事务，暂停当前的事务。

6、Propagation.NEVER

以非事务的方式运行，如果当前存在事务，则抛出异常。

7、Propagation.NESTED

如果没有，就新建一个事务；如果有，就在当前事务中嵌套其他事务。

### Spring生命周期

粗略的可以分成三个阶段，他有Bean的一次加工、属性设置、Bean的二次加工过程。

第一步：Bean的一次加工，实例化对象，在实例化前后，会用不同的BeanPostProcessor做前置和后置处理。

第二步：依赖注入过程，首先会设置Bean的属性值，然后处理不同的BeanAware（比如：BeanNameAware、ApplicationContextAware），给Bean注入相应的aware属性，这个阶段主要是分为这2步，这2步前后也有不同的BeanPostProcessor后置处理器进行处理。第二步完成之后，Bean已经接近完成期。

第三部：Bean的二次加工前后，可以像ApplicatioinContext一样配置一个自定义的后置处理Bean来处理，在这期间主要是调用了Bean中写的init方法或者Bean实现了InitializingBean，他会回调afterPropertiesSet方法，来对bean做进一步的设置。二次加工完成之后，Bean就已经建立完成，这时候它会根据Bean的scope来对Bean进行单例还是原型处理，默认是单例，如果是单例，生命周期还是由Spring来管理，如果是原型（scope为prototype），就会转交给调用者去管理它后续的生命周期。当Spring容器销毁的时候，它会调用destory相应的方法，或者开发者自己实现了DisposableBean接口，Spring就会调用开发者自己销毁的逻辑。

## JVM内存模型

JVM运行时的内存区域可以分为两大类，一个是线程的私有区，另外一个是线程共享区。

### 线程私有区

首先第一个是程序计数器，这里在JVM中栈内存比较小的一个地方，但是也是唯一一块不会OOM的区域，这里会告诉你当前线程执行的代码归于哪一行；

另外线程私有的还有虚拟机栈和本地方法栈，虚拟机栈主要使用的是栈帧，一个方法的调用就是从入栈到出栈的过程。

### 线程共享区

首先第一个是方法区，在方法区它其实只是JVM的一个规范，但是它在1.7版本的时候，它的实现在HotSpot虚拟机中叫做永久代，这里面存放一些常连池常量还有类的元数据信息，1.8版本的时候，因为一些原因，它转移到了一个集结内存中，叫做元空间，这里存放的是类的元数据信息，这里的方法区会OOM；

还有一个区域是堆，它里面存放是java里产生的对象、对象的实例，它也是GC重点回收的一个区域，也会产生OOM；

最后一个是运行常量池，它是在1.8的时候将运行常量池转移到堆中存放，因为之前它是在元空间里面，后来搬移到堆里面了，堆里面存放的是运行时常量池和静态变量；

直接内存：它现在存放在JVM的元空间，存放在元空间的好处是OOM的机遇相对之前变小了很多。

## 垃圾回收器

CMS：它可以和用户线程并发操作，就是GC线程与用户线程并发执行的一款垃圾回收，特点就是低延迟，它的整个过程主要是用来回收老年代，算法使用的是标记清除和标记整理，它的过程有四个阶段：1、初始标记 2、并发标记 3、重新标记 4、并发整理；但它在初始标记和最终标记的时候，它不是线程并行，它会短暂的产生Stop The World（用户线程必须暂停），但这个时间很短。它的并发标记和并发清理，这2个阶段的耗时比较长，但它是和用户线程一起并发执行的，所以说它可以最短的实现低延迟。

### 垃圾回收算法

1. 第一种是标记清除，标记一些不可达的对象，比如说将判定为死亡的对象，然后统一回收掉，它因为这种使用可能会造成很多的内存碎片，并且当它对象是比较大的时候，它的标记效率是比较低的
2. 第二种是标记复制，标记复制就是将可用内存按容量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内存上面，然后再把已使用过的内存空间一次清理掉。

缺点是可一次性分配的最大内存缩小了一半。优点是不会产生内存碎片。

1. 第三种是标记整理，将所有存活的对象都移动到一个规整且连续的内存空间中，将已经判定为死亡的对象往另一端移动、复制，然后执行垃圾回收器回收无用对象所占用的内存空间
2. 第四种是分代收集，根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收

### 怎么判断对象是否可达

1. 第一种是引用计数，但它的缺点是可能产生循环依赖
2. 第二种是根据java中可以作为GC Roots根节点来开个链，如果不在这个链上则可以判断可回收了。

### Java中可以作为GC Roots的对象

1. 虚拟机栈中引用的一些对象
2. 方法区中静态变量所引用的对象
3. 方法区中常量所引用的对象
4. 本地方法栈中JNI（即一般说的native方法）中引用的对象

## Linux的CPU突然飙高排查

1. top找到异常进程
2. ps –ef|grep java 找到异常进程，就可以找到对应的应用
3. top -H -p 进程号查看异常线程
4. printf "%x\n" 线程号将异常线程号转化为16进制
5. jstack 进程号|grep 16进制异常线程号 -A90来定位异常代码的位置（最后的-A90是日志行数，也可以输出为文本文件或使用其他数字）。可以看到异常代码的位置。

如果提示 -bash: jstack: command not found，则可以找到jdk的目录，进入bin目录，在执行 ./jstack 进程号 | grep 16进制异常线程号 -A90 即可。

## Dubbo

RPC协议：通过不同的技术进行远程方法调用，比如可以通过HTTP发送请求、TCP发送请求、Netty发送请求。

Dubbo协议：通过Netty进行远程方法调用。

多个服务B将服务往ZK进行注册，服务A通过注册中心获取到多个服务B的地址，然后缓存到本地，最后通过Dubbo智能负载均衡技术，进行远程服务B的调用。

Zookeeper注册中心的优势：

1. Zookeeper的数据是存放在内存中的，取数据快。
2. 服务B挂掉了，可以使用zk的临时节点，这样就不影响服务A的调用
3. 新增服务B，因为zk的watch机制，服务A就可以知道服务B的地址有改动，使服务A的缓存和注册中心的地址数据是实时一致的。

### Dubbo服务暴露过程

在使用XML方式配置标签的情况下，首先它有个dubbo的DubboBeanDefinitionParser终端解析器去解析标签，像application、registry、protocol、consumer的一些配置，会相应的解析成对应的config对象，但是有个特殊情况是暴露的服务不会写到对应的service标签里，它不是直接解析成ServiceConfig，而是会解析成ServiceConfig下面的一个子类叫做ServiceBean。ServiceBean实现了2个接口，第一个是实现了InitializingBean，到时候Spring会回调afterPropertiesSet方法，第二个是实现了ApplicatioinListener，使用这种观察者模式去监听ContextRefreshEvent，能够去刷新afterPropertiesSet的回调，它会等Spring容器处理完毕后，回调onApplicationEvent方法，但这个里面会进行两个服务的暴露，区别是延迟走afterPropertiesSet方法，不延迟走onApplicationEvent方法。它里面会调用一个暴露方法export，其实这个暴露方法在他的父类ServiceConfig里面比较核心的，它会多次export，然后进行下一次循环调用doExportUrlsForProtocol，这个flow里面有个dubbo核心spi机制，它是跟JDK还是不太一样的，以kv的结构进行存储，可以通过实例化加载某一个类对应所需要的服务。接着protocol，它会调用它的export的一个方法，但是这Protocol它相关的两个实现，第一个是RegistryProtocol，第二个是dubboProtocol，它在Registry的暴露中第一步就是doLocalExport，就是本地暴露，在这个过程中调用dubboProtocol里面将里面的实现全部封装成invoker，然后到dubboProtocol会将invoke封装成一个export，然后将这个export通过netty去暴露出去。暴露完成之后会返回到RegistryProt的暴露口中，去将之前暴露的invoker注册到一个Registry中，并且它还会将已经暴露的invoker保存到它的ProviderConsumer的注册表中，这个注册表之后会被远端引用。

注册表结构

ConcurrentHashMap，key是String，value是set

### 为什么dubbo需要支持多个协议？

不同的协议适用的场景不同。

1. dubbo适用于常规远程服务方法调用，尽量不要用 dubbo 协议传输大文件或超大字符串。
2. rmi适用于常规远程服务方法调用，与原生RMI服务互操作，可用来传输文件。
3. hessian适用于页面传输，文件传输，或与原生hessian服务互操作。
4. http适用于需同时给应用程序和浏览器 JS 使用的服务，暂不支持文件传输。
5. webServices适用于系统集成，跨语言调用。

### Dubbo的负载均衡策略

Random LoadBalance：随机

RoundRobin LoadBalance：循环

LeastActive LoadBalance：最少活跃调用数

ConsistentHash LoadBalance：一致性Hash

### 集群容错

Failover Cluster：失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。

## Nginx

### Nginx有那些负载均衡策略

1. 轮询：每个请求会按时间顺序逐一分配到不同的后端服务器
2. 一致性哈希：相同请求参数的情况都会分配到同一台机器上
3. 权重：在轮询策略的基础上指定轮询的几率
4. 最少连接：把请求转发给连接数较少的后端服务器
5. 响应时间（第三方）：按照服务器端的响应时间来分配请求，响应时间短的优先分配
6. 依据URL分配（第三方）：按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，要配合缓存命中来使用。

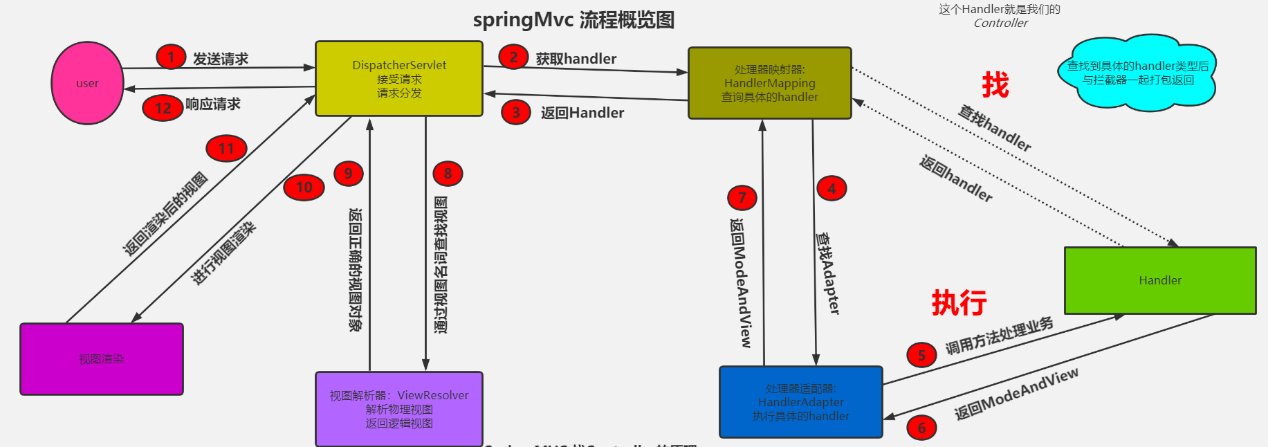
注意：第三方的负载均衡策略的实现需要安装第三方插件

### Nginx缺点

Nginx存在中心化的问题，比如：

1. 当nginx宕机了，需要修改服务A的请求地址，才能调用到服务B
2. 当新增了服务B的地址，需要修改nginx的配置文件，才能实现负载均衡

## Spring MVC



### 源码流程

DispatcherServIet：前端控制器

HandlerMapping：处理器映射器

HandlerExecutionChain：处理器执行链

HandlerAdapter：处理器适配器

ViewResolver：视图解析器

1、客户端请求到前端控制器（DispatcherServIet）

2、前端控制器（DispatcherServIet）请求处理器映射器（HandlerMapping）

3、处理器映射器（HandlerMapping）根据URL查找相应的处理器（Handler)，返回处理器执行链（HandlerExecutionChain）给前端控制器（DispatcherServlet）

4、前端控制器（Dispatcherservlet）请求处理器适配器（HandlerAdapter）

5、处理器适配器（HandlerAdapter)执行处理器（Handler），调用方法处理业务逻辑，生成ModelAndView，返回ModelAndView给前端控制器（DispatcherServlet）

6、前端控制器（DispatcherServIet〕请求视图解析器（ViewResolver）

7、视图解析器（ViewResovler）返回视图对象给前端控制器（Dispatcherservlet〕

8、最后渲染视图

### 通俗流程

1、tomcat实现servlet的spi规范

1.1 tomcat找到ServletContainerInitializer的实现类，并调用onStartup方法

1.2 Spring的SpringServletContainerInitializer类继承了ServletContainerInitializer类，重写onStartup方法，SpringServletContainerInitializer这个类只用循环调用子类的onStartup方法。

1.3 tomcat调用Spring的SpringServletContainerInitializer.onStartup方法

2、tomcat启动时会在实现SpringServletContainerInitializer子类的onStartup方法中，扫描并加载web.xml配置文件

3、web.xml配置文件配置spring-mvc.xml

4、web.xml会加载DispatcherServlet

5、DispatcherServlet扫描spring-mvc.xml的配置文件，扫描整个项目，根据配置文件给定的目录来扫描

6、扫描所有加了@Controller注解的类

7、当扫描到加了@Controller注解的类之后遍历里面所有的方法

8、拿到方法对象之后，解析方法上面是否加了@RequestMapping注解

9、定义一个map集合（Map<String,Method>），把@RequestMapping的value与方法对象绑定起来

10、拦截到请求之后，拿到请求的URI

11、拿URI去Map里面去get，找到方法

12、消息转换器处理，扫描方法里面的参数，然后根据request.getParameterNames()获取到前台传过来的参数，通过java反射对参数进行赋值

13、视图解析器处理。如果方法加了@ResponseBody，则进行resp.getWriter().write(result)直接将数据通过Response返回出去，如果是ModelAndView则进行视图的处理；如果是String，则通过请求转发req.getRequestDispatcher(前缀+String.valueOf(result)+后缀).forward(req,resp)进行页面跳转；