作业，提交到gitee上。 完成以后，把gitee地址发我

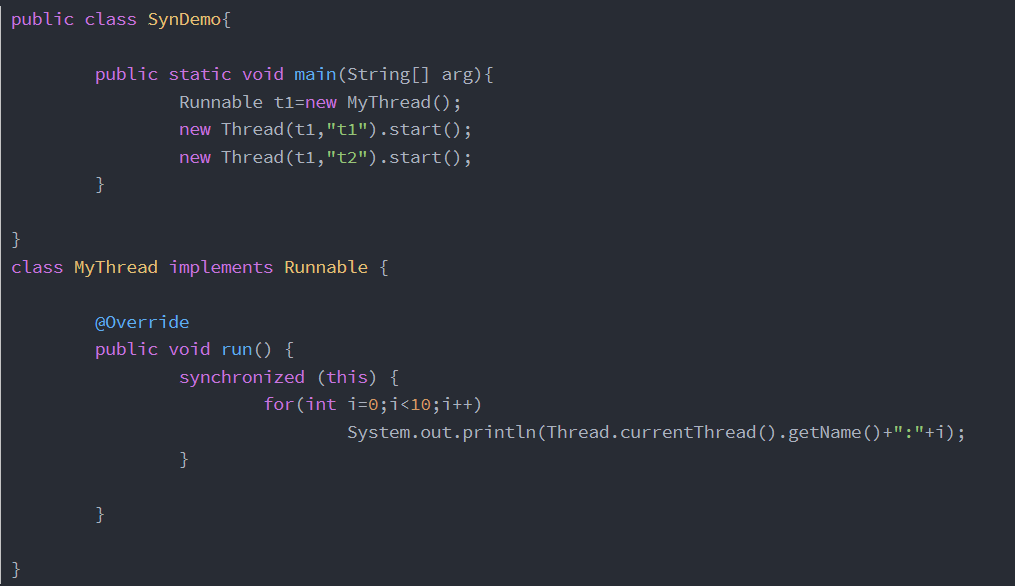
另外，可以查任何资料，但务必搞清楚。

1. 使用seata完成分布式事务：还是以 product 和 order 为例。

2. synchronized和ReentrantLock和区别，针对于每一个区别，写出一个例子演示。

锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。





3. volidate关键字保证可见性的例子演示，包括正例、反例。









4. 悲观锁和乐观锁的区别？

悲观锁：总是以为自己拿到的数据被别人修改了，这样每次拿数据的时候都会上锁，别人想拿数据就会阻塞直到它拿到锁

（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁：总是以为自己拿到的数据没有被别人修改过，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，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。

5. 什么是脏读？不可重复读？ 幻读？ 以及如何解决这些问题。

脏读：脏读是指一个事务读取到了其他事务没有提交的数据；

不可重复读：不可重复读是指一个事务内多次根据同一个查询条件查询出来的同一行记录的值不一样；

幻读：幻读是指一个事务内多次根据同个条件查出来的记录行数不一样。

为了解决事务并发带来的问题，才有了事务规范中的四个事务隔离级别，不同隔离级别对上面问题部分或者全部做了避免。

6. 什么是间隙锁？

间隙锁是一个在索引记录之间的间隙上的锁。

7. MVCC是什么，请描述清楚。

MVCC，Multi-Version Concurrency Control，多版本并发控制。MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问；在编程语言中实现事务内存。

8. docker容器和虚拟机的区别是什么？

Docker守护进程可以直接与主操作系统进行通信，为各个Docker容器分配资源；它还可以将容器与主操作系统隔离，并将各个容器互相隔离。虚拟机启动需要数分钟，而Docker容器可以在数毫秒内启动。由于没有臃肿的从操作系统，Docker可以节省大量的磁盘空间以及其他系统资源。说了这么多Docker的优势，大家也没有必要完全否定虚拟机技术，因为两者有不同的使用场景。虚拟机更擅长于彻底隔离整个运行环境。例如，云服务提供商通常采用虚拟机技术隔离不同的用户。而Docker通常用于隔离不同的应用，例如前端，后端以及数据库。

9. 多生产者多消费者，换个场景例子写一遍，不要使用我的碗架例子了。

package com.shone.thread.KFC;

/\*\*

\* Created by Xiao GuoJian on 2018/1/24.

\* 在一个KFC内，服务员负责生产食物，消费者负责消费食物；

\* 当生产到一定数量可以休息一下，直到消费完食物，再马上生产，一直循环

\*/

class Food{

public int count = 1;//默认数量

public int maxCount = 20;//count为100时，生产者停止生产

//生产

public synchronized void product(){

while(count>=maxCount){

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

count++;

try {

Thread.sleep(25);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+" 生产一个汉堡，当前汉堡数："+count);

if(count == maxCount){

notifyAll();

}

}

//消费

public synchronized void consume(){

while(count <= 0){

try {

wait();//当前线程等待

} catch (InterruptedException e) {

e.printStackTrace();

}

}

count--;

try {

Thread.sleep(20);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+" 消费一个汉堡，当前汉堡数："+count);

if(count==0){

notifyAll();

}

}

}

//生产者

class Producter extends Thread{

private Food food;

public Producter(Food food){

super();

this.food = food;

}

public void run(){

while (true){

food.product();

}

}

}

//消费者

class Consumer extends Thread{

private Food food;

public Consumer(Food food){

super();

this.food = food;

}

public void run(){

while (true){

food.consume();

}

}

}

public class KfcDemo {

public static void main(String[] args) {

Food food = new Food();

Producter producter1 = new Producter(food);

Producter producter2 = new Producter(food);

Consumer consumer1 = new Consumer(food);

Consumer consumer2 = new Consumer(food);

producter1.start();

producter2.start();

consumer1.start();

consumer2.start();

}

}

10. 线程安全的集合有哪些？

线程安全的集合：Vector 线程安全：HashTable 线程安全：StringBuffer 线程安全：

11. 什么是CAS？

CAS是Central Authentication Service的缩写，中央认证服务，一种独立开放指令协议。CAS 是 耶鲁大学（Yale University）发起的一个开源项目，旨在为 Web 应用系统提供一种可靠的单点登录方法，CAS 在 2004 年 12 月正式成为 JA-SIG 的一个项目。

12. Jdk1.7 Jdk1.8中的Map有什么区别？

1. 最重要的一点是底层结构不一样，1.7是数组+链表，1.8则是数组+链表+红黑树结构;

2. jdk1.7中当哈希表为空时，会先调用inflateTable()初始化一个数组；而1.8则是直接调用resize()扩容;

3. 插入键值对的put方法的区别，1.8中会将节点插入到链表尾部，而1.7中是采用头插；

4. jdk1.7中的hash函数对哈希值的计算直接使用key的hashCode值，而1.8中则是采用key的hashCode异或上key的hashCode进行无符号右移16位的结果，避免了只靠低位数据来计算哈希时导致的冲突，计算结果由高低位结合决定，使元素分布更均匀；

5. 扩容时1.8会保持原链表的顺序，而1.7会颠倒链表的顺序；而且1.8是在元素插入后检测是否需要扩容，1.7则是在元素插入前；

6. jdk1.8是扩容时通过hash&cap==0将链表分散，无需改变hash值，而1.7是通过更新hashSeed来修改hash值达到分散的目的；

7. 扩容策略：1.7中是只要不小于阈值就直接扩容2倍；而1.8的扩容策略会更优化，当数组容量未达到64时，以2倍进行扩容，超过64之后若桶中元素个数不小于7就将链表转换为红黑树，但如果红黑树中的元素个数小于6就会还原为链表，当红黑树中元素不小于32的时候才会再次扩容。

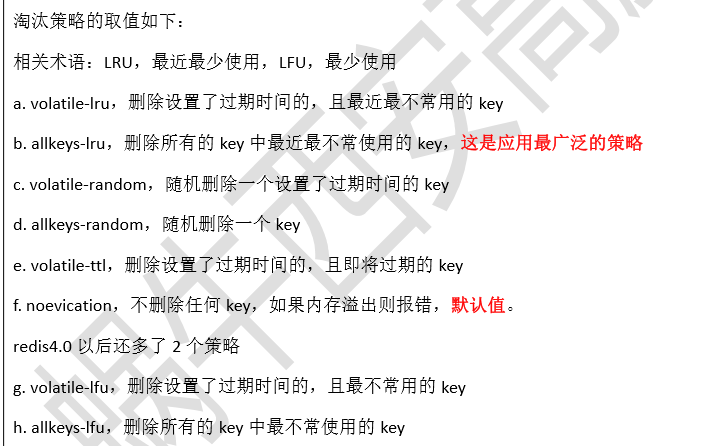
13. redis的逐出策略是什么？

问题：

 Redis使用内存存储数据,在执行每一个命令前,会调用freeMemoryIfNeeded()检测内存是否充足。如果内存不满足新加入数据的最低存储要求,redis要临时删除一些数据为当前指令清理存储空间。清理数据的策略为逐出算法。

 注意:逐出数据的过程不是100%能够清理出足够的可使用的内存空间,如果不成功则反复执行。当对所有数据尝试完毕后,如果不能达到内存清理的要求,将出现错误信息。

14. redis的淘汰策略是什么？



15. 如何能筛选出redis中的高热键？

热点数据需要进行业务逻辑上的拆分或者多级缓存分担压力。我们线上也因为大键造成了一些困扰，例如：网卡带宽被打死、del大键造成Redis堵塞等，从Redis本身，确实没有太好的办法来解决，只能从业务层面分析出现大键的原因，做出响应的对策，例如：对大键进行压缩存储、或者存储大键到有多线程处理的pika中等等。

16. 消息中间件的作用是什么？

消息中间件利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行[分布式系统](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E7%B3%BB%E7%BB%9F)的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。解耦合

17. 如何保证RabbitMQ消息的可靠性？

要确保rabbitMQ消息的可靠性要做到三点：

1. publisher confirms（发布方确认）：

确保producer到broker节点消息的可靠性。如可能发生消息到了broker但是还没有投递到queue，broker突然宕机这种情况；

1. message持久化：

将message存储到硬盘，如果是未持久化的消息会存在内存中，broker宕机后重启内存中的消息会丢失

1. acknowledgement（consumer确认）：

该机制能够保证，只有consumer成功消费了消息，才将其从queue中移除。

18. redis事务的作用是什么？ 通过redis如何实现乐观锁？

可以一次性执行多条命令，本质上是一组命令的集合。一个事务中的所有命令都会序列化，然后按顺序地串行化执行，而不会被插入其他命令。一个队列中，一些性，顺序性，排他性的执行一系列的命令

1.multi，开启Redis的事务，置客户端为事务态。

2.exec，提交事务，执行从multi到此命令前的命令队列，置客户端为非事务态。

3.discard，取消事务，置客户端为非事务态。

4.watch,监视键值对，作用时如果事务提交exec时发现监视的监视对发生变化，事务将被取消。

19. 遍历Map的方式有哪些，写出代码。



20. CountDownLatch和CyclicBarrier的区别是什么?

**CountDownLatch** : **一个线程**(或者多个)， 等待另外**N个线程**完成**某个事情**之后才能执行。  **CyclicBarrier** : **N个线程**相互等待，任何一个线程完成之前，所有的线程都必须等待。  
这样应该就清楚一点了，对于CountDownLatch来说，重点是那个**“一个线程”**, 是它在等待， 而另外那N的线程在把**“某个事情”**做完之后可以继续等待，可以终止。而对于CyclicBarrier来说，重点是那**N个线程**，他们之间任何一个没有完成，所有的线程都必须等待。

21. TCP和UDP的区别是什么？

1.基于连接与无连接；

2.对系统资源的要求（TCP较多，UDP少）；

3.UDP程序结构较简单；

4.流模式与数据报模式 ；

5.TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证。

22. TCP为什么需要3次握手，而不是2次，或者4次？

 如果是两次握手，首先client给server发送了一个连接请求，server收到了这个请求并且发送了一个确认应答的请求，站在server的角度，它认为连接已经建立成功了，开始发送数据。但是此时client并没有得到响应，它会认为连接还没有建立成功，将忽略server发送的数据，这样server在发出的数据超时后，会重复的发送数据，就形成了死锁的情况。所以两次握手显然不行。

   超过三次以上其实都可以，但是三次刚好可以完全建立连接，四次或者多次效率会低点。所以TCP建立连接采用“三次握手”。

23. HashSet的原理是什么？

HashSet 是基于 HashMap 实现的，查询速度特别快

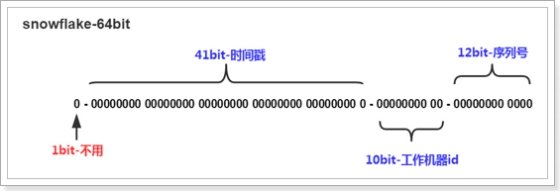
HashMap 是支持 key 为 null 值的，所以 HashSet 支持添加 null 值

HashSet 存放自定义类时，自定义类需要重写 hashCode() 和 equals() 方法，确保集合对自定义类的对象的唯一性判断(具体判断逻辑，见 HashMap put() 方法，简单概括就是 key 进行 哈希。判断元素 hash 值是否相等、key 是否为同个对象、key 是否 equals。第 1 个条件为 true，2、3 有一个为 true，HashMap 即认为 key 相同)

无序、不可重复

24. 雪花算法(Snow Flake)是什么？能解释一下原理吗？

**简单原理**：  
雪花算法会生成一个64位的二进制数据，为一个Long型。(转换成字符串后长度最多19) ，其基本结构：



1. 第一个部分： 1 个 bit：0，这个是无意义的。

2. 第二个部分： 41个 bit：是时间戳，即为当前的毫秒数。

3. 第三个部分： 10个 bit：机器的id。

4. 第四个部分：12 个 bit：序号，就是某个机房某台机器上这一毫秒内同时生成的 id 的序号，一台机器一毫秒内生成序列号最大值为2的12次方（4096），超过此个数延迟到下一秒生成。

25. 分布式id的解决方案有哪些？ 各有什么优缺点？

**1. UUID**

UUID经由一定的算法机器生成，为了保证UUID的唯一性，规范定义了包括网卡MAC地址、时间戳、名字空间(Namespace)、随机或伪随机数、时序等元素，以及从这些元素生成UUID的算法。

**优点：**

* 本地生成ID，不需要进行远程调用，时延低，性能高。

**缺点：**

* UUID过长，16字节128位，通常以36长度的字符串表示，很多场景不适用，比如用UUID做数据库索引字段。
* 没有排序，无法保证趋势递增。

**2. Flicker方案**

主要思路采用了MySQL自增长ID的机制(auto\_increment + replace into)

#每次业务使用下列SQL读写MySQL得到ID号

REPLACE INTO Tickets64 (stub) VALUES ('a');

SELECT LAST\_INSERT\_ID();

replace into 跟 insert 功能类似，不同点在于：replace into 首先尝试插入数据到表中，如果发现表中已经有此行数据(根据主键或者唯-索引判断)则先删除此行数据，然后插入新的数据， 否则直接插入新数据。

为了避免单点故障，最少需要两个数据库实例，通过区分auto\_increment的起始值和步长来生成奇偶数的ID。

**优点：**

* 充分借助数据库的自增ID机制，可靠性高，生成有序的ID。

**缺点：**

* ID生成性能依赖单台数据库读写性能。
* 依赖数据库，当数据库异常时整个系统不可用。

**3. 类snowflake方案**

* 这种方案生成一个64bit的数字，64bit被划分成多个段，分别表示时间戳、机器编码、序号。

**long 数字，由三部分组成：**

* 41位的时间序列(精确到毫秒，41位的长度可以使用69年)。
* 10位的机器标识(10位的长度最多支持部署1024个节点)。
* 12位的计数顺序号(12位的计数顺序号支持每个节点每毫秒产生4096个ID序号)。

**优点：**

* 时间戳在高位，自增序列在低位，整个ID是趋势递增的，按照时间有序。
* 性能高，每秒可生成几百万ID。
* 可以根据自身业务需求灵活调整bit位划分，满足不同需求。

**缺点：**

* 依赖机器时钟，如果机器时钟回拨，会导致重复ID生成。
* 在单机上是递增的，但是由于涉及到分布式环境，每台机器上的时钟不可能完全同步，有时候会出现不是全局递增的情况。

**4. TDDL序列生成方式**

TDDL是阿里的分库分表中间件，它里面包含了全局数据库ID的生成方式，主要思路：

* 使用数据库同步ID信息。
* 每次批量取一定数量的可用ID在内存中，使用完后，再请求数据库重新获取下一批可用ID，每次获取的可用ID数量由步长控制，实际业务中可根据使用速度进行配置。
* 每个业务可以给自己的序列起个唯一的名字，隔离各个业务系统的ID。

**优点**：  
- 相比flicker方案，大大降低数据库写压力，数据库不再是性能瓶颈。  
- 相比flicker方案，生成ID性能大幅度提高，因为获取一个可用号段后在内存中直接分配，相对于每次读取数据库性能提高了几个量级。  
- 不同业务不同的ID需求可以用seqName字段区分，每个seqName的ID获取相互隔离，互不影响。

**缺点：**

* 强依赖数据库，当数据库异常时整个系统不可用

26. 分布式id为什么不直接干脆使用UUID？ 而是使用雪花算法？

**数据库和UUID方案的不足之处**

采用数据库自增序列：

读写分离时，只有主节点可以进行写操作，可能有单点故障的风险

分表分库，数据迁移合并等比较麻烦

UUID随机数：

采用无意义字符串，没有排序

UUID使用字符串形式存储，数据量大时查询效率比较低

**雪花算法的特点(自增、有序、适合分布式场景)**

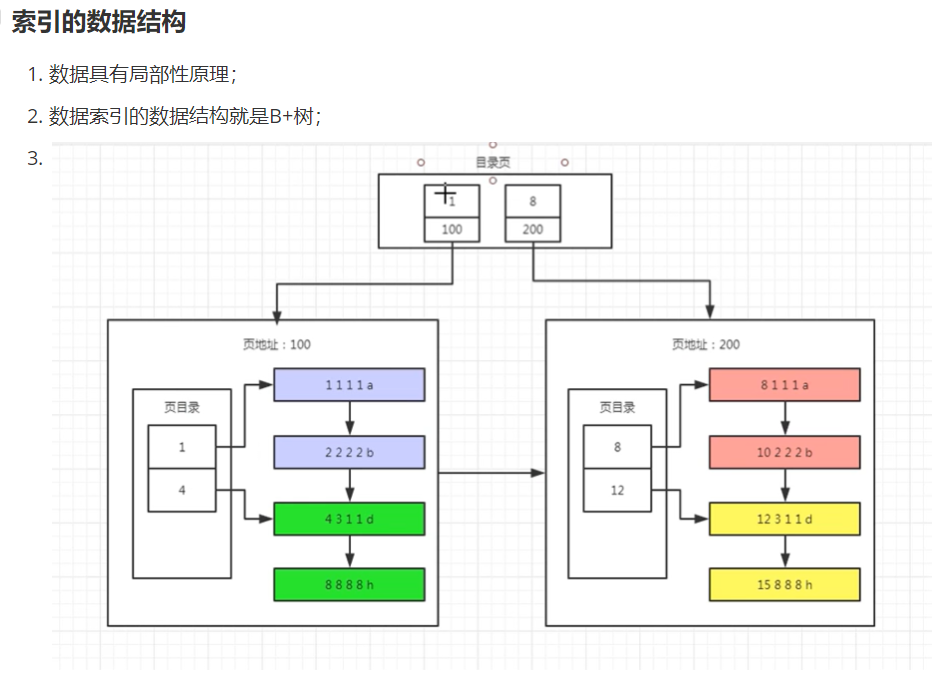
时间位：可以根据时间进行排序，有助于提高查询速度。

机器id位：适用于分布式环境下对多节点的各个节点进行标识，可以具体根据节点数和部署情况设计划分机器位10位长度，如划分5位表示进程位等。

序列号位：是一系列的自增id，可以支持同一节点同一毫秒生成多个ID序号，12位的计数序列号支持每个节点每毫秒产生4096个ID序号

snowflake算法可以根据项目情况以及自身需要进行一定的修改。

27. 索引的数据结构了解吗？

****28. 聚集索引和非聚集索引的区别？

1.聚合索引：类似于主键，一张表有且只能有一个聚合索引；

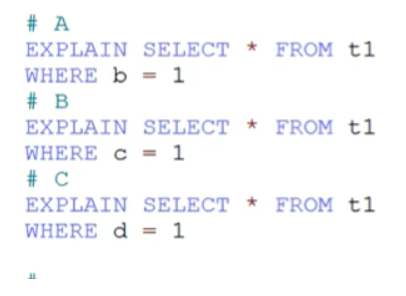
2.非聚集索引：一张可以有多个非聚集索引；（存在回表）

29. 索引中的“回表”是什么意思？

1、我理解不存在“通过二级索引查询到聚簇索引”的过程，二级索引中保存有表主键（如果没有主键mysql会自动生成一个rowid），回表指的是通过主键查询表的某一行。

2、Extra中看到了Using where，代表是按照where条件进行了过滤，和是否走索引、回表无必然的关系。  
只有在使用了索引，且Extra是Using where的情况下，才代表回表查询数据。  
还有一种情况，Extra中是using index & using where，表示select的数据在索引中能找到，但需要根据where条件过滤，这种情况也不回表。

30. 什么是索引中的最左原则？



31. 什么是小表驱动大表原则？



32. mysql中，如何为select语句添加共享锁 和 排他锁？

select语句默认不加任何锁

select也可以强行加共享锁，只要在查询语句之后添加上lock in share mode

select也可以强行加排他锁，只要在查询语句之后添加for update即可

33. 共享锁和排他锁的区别是什么？

排他锁是针对数据库的“写锁”；共享锁又叫“S锁”由非更新（读取）操作创建的锁。其他用户可以并发读取数据，但任何事务都不能获取数据上的排它锁，直到已释放所有共享锁。

34. SpringMVC的工作原理？ SpringMVC这样的工作原理有什么好处？

**SpringMVC流程**

1、  用户发送请求至前端控制器DispatcherServlet。

2、  DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、  处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、  DispatcherServlet调用HandlerAdapter处理器适配器。

5、  HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、  Controller执行完成返回ModelAndView。

7、  HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。

8、  DispatcherServlet将ModelAndView传给ViewReslover视图解析器。

9、  ViewReslover解析后返回具体View。

10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet响应用户。

**springMVC是什么：**

springMVC是一个MVC的开源框架，springMVC=struts2+spring，springMVC就相当于是Struts2加上sring的整合，但是这里有一个疑惑就是，springMVC和spring是什么样的关系呢？这个在百度百科上有一个很好的解释：意思是说，springMVC是spring的一个后续产品，其实就是spring在原有基础上，又提供了web应用的MVC模块，可以简单的把springMVC理解为是spring的一个模块（类似AOP，IOC这样的模块），网络上经常会说springMVC和spring无缝集成，其实springMVC就是spring的一个子模块，所以根本不需要同spring进行整合。

35. 代理模式、适配器模式、装饰器模式看起来很相似，它们的区别是什么？

代理模式：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

适配器模式：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

装饰器模式：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

代理模式与两者区别：

1、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。

2、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。

36. Spring中，Bean的生命周期是什么？ 如果要为某个bean制作代理对象，再把代理对象放入容器中，应该在哪个环节进行？

1.首先容器启动后，会对scope为singleton且非懒加载的bean进行实例化，

2.按照Bean定义信息配置信息，注入所有的属性，

3.如果Bean实现了BeanNameAware接口，会回调该接口的setBeanName()方法，传入该Bean的id，此时该Bean就获得了自己在配置文件中的id，

4.如果Bean实现了BeanFactoryAware接口,会回调该接口的setBeanFactory()方法，传入该Bean的BeanFactory，这样该Bean就获得了自己所在的BeanFactory，

5.如果Bean实现了ApplicationContextAware接口,会回调该接口的setApplicationContext()方法，传入该Bean的ApplicationContext，这样该Bean就获得了自己所在的ApplicationContext，

6.如果有Bean实现了BeanPostProcessor接口，则会回调该接口的postProcessBeforeInitialzation()方法，

7.如果Bean实现了InitializingBean接口，则会回调该接口的afterPropertiesSet()方法，

8.如果Bean配置了init-method方法，则会执行init-method配置的方法，

9.如果有Bean实现了BeanPostProcessor接口，则会回调该接口的postProcessAfterInitialization()方法，

10.经过流程9之后，就可以正式使用该Bean了,对于scope为singleton的Bean,Spring的ioc容器中会缓存一份该bean的实例，而对于scope为prototype的Bean,每次被调用都会new一个新的对象，期生命周期就交给调用方管理了，不再是Spring容器进行管理了

11.容器关闭后，如果Bean实现了DisposableBean接口，则会回调该接口的destroy()方法，

12.如果Bean配置了destroy-method方法，则会执行destroy-method配置的方法，至此，整个Bean的生命周期结束

37. map 的深拷贝和浅拷贝的区别？

浅复制：只复制引用，但是引用所指向的对象 是同一块内存区域，这两个引用只要有其中一个操作该内存区域，内容都会同时改变（就相当于一个人有两个思想驱动）

深度复制：也叫完全复制，完全独立的引用和完全独立的两块内存区域，互不影响，（相当于两个独立的人）

38. final、finally、finalize的区别？

一、final ：

1、修饰符（关键字） 如果一个类被声明为final,意味着它不能再派生新的子类，不能作为父类被继承。因此一个类不能及被声明为abstract，又被声明为final的。

2、将变量或方法声明为final,可以保证他们使用中不被改变。被声明为final的变量必须在声明时给定初值，而以后的引用中只能读取，不可修改，被声明为final的方法也同样只能使用，不能重载。

二、finally:

在[异常处理](https://www.baidu.com/s?wd=%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)时提供finally块来执行清楚操作。如果抛出一个异常，那么相匹配的catch语句就会执行，然后控制就会进入finally块，如果有的话。

三、finalize：

是方法名。[java](https://www.baidu.com/s?wd=java&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)技术允许使用finalize()方法在垃圾收集器将对象从内存中清除之前做必要的清理工作。这个方法是在垃圾收集器在确定了，被清理对象没有被引用的情况下调用的。

finalize是在Object类中定义的，因此，所有的类都继承了它。子类可以覆盖finalize()方法，来整理[系统资源](https://www.baidu.com/s?wd=%E7%B3%BB%E7%BB%9F%E8%B5%84%E6%BA%90&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)或者执行其他清理工作。

39. 谈谈你对GC的认识。

GC（GarbageCollection）是垃圾回收机制(垃圾回收器)，GC是JVM对内存（实际上就是对象）进行管理的方式。  
Java是由C++发展来的。它摈弃了C++中一些繁琐容易出错的东西。其中有一条就是这个GC。

Java有了GC，就不需要程序员去人工释放内存空间。GC使得Java开发人员摆脱了繁琐的内存管理工作，让程序的开发更有效率。当Java虚拟机发觉内存资源紧张的时候，就会自动地去清理无用变量所占用的内存空间。当然，程序员可以在Java程序中显式地使用System.gc()或Runtime.getRuntime().gc()来通知垃圾回收程序，至于垃圾回收程序运不运行，什么时候运行，都是无法控制的。

如果对象引用被置为null，垃圾收集器是否会立即释放对象占用的内存？

说来话长。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。

40. 如何实现mysql的主从复制？ 请描述一下原理和步骤。

实现原理：

mysql会将所有修改过数据库的sql语句记录在二进制bin-log日志中。主从复制实际是将多台服务器bin日志的开启，然后主服务器（master）会把执行过的sql语句记录到bin日志中，之后从服务器（slave）读取该日志，在从服务器再把bin日志中记录的sql语句同样的执行一遍。这样从服务器（slave）上的数据就和主服务器（master）同步了。

实现步骤：

两台服务器：

主服务器：ip地址：192.168.2.17，系统：Centos，配置文件：my.cnf  
从服务器：ip地址：192.168.2.4，系统：Windows，配置文件：my.ini

**3.1 在主服务器上修改 my.cnf 配置文件**

**3.2 修改完成mysql的配置后，要重启mysql服务使之生效**

**3.3 授权一个账号，让从服务器通过该账号读取log-bin日志里面的内容**

**3.4 赋予从库权限账号，允许用户在主库上读取日志，也就是Slave机器读取File权限**

**3.5 查看最新的log-bin日志，记录主服务器里面的最新的二进制的名称和pos位置**

### 配置从服务器：

**4.1 修改从服务器的数据库配置文件**

**4.2 停止从服务器**

**4.3 开始配置**

**4.4 启动从服务器**

**4.4 验证配置是否成功**

### 测试主从复制：

**5.1 在主服务器创建一个数据库，并添加一张表，插入数据**

**5.2 在从服务器上面查看是否有插入的数据记录**

41. 谈谈你对java中泛型的理解，以及什么是泛型的“擦除”?

为了迁移兼容性 :

1.为了减少潜在的关于 擦除 的混乱，你必须认识到这不是一个语言的特征，它是java泛型实现中的一种折中。因为泛型不是java语言出现时就有的组成成分，所以这种折中是必须的

2.如果泛型是java1.0就已经是其一部分了，那么这个特征将不会使用擦除来是实现--它将使用具体化，使类型参数保持为第一类实体，这样的话你就可以在类型参数上执行基于类型语言的操作和反射操作。擦除  减少了泛型的泛化性，泛型咋java中仍然是有用的，只是不如它们本来设想的那么有用，其原因就是擦除

3.在基于擦除的实现中，泛型类型被当作第二类类型来处理，既不能在重要的上下文环境中使用 类型。泛型类型只有在  静态类型  检查期间才出现，在此之后，程序中所有的泛型类型都被擦除了，替换为它们的非泛型上界。例如：诸如List<T>这样的类型注解将被擦除为List，而普通的类型变量在未指定边界的情况下将被擦除为Object

4.擦除的核心动机是它使得泛化的客户端可以用非泛化的类库来使用，反之亦然。这经常被称为"迁移兼容性"。因为在理想情况下，当所有的事物都可以使用泛化，那我们就可以专注于此。但是在现实中，5.0之前是没有泛型的，必须要处理这些没有被泛化的类库

5.因此java泛型不仅必须支持向后兼容性，即现有的代码是合法的，并且继续保持之前的含义。而且还要支持迁移兼容性，使得旧的代码不会影响新的代码。一句话就是允许非泛型代码与泛型代码共存，擦除 使得这种向着泛型的迁移成为可能

边界处的动作:因为擦除，泛型最令人困惑的一方面源自这样的一个事实，即可以表示没有任何意义的事物

1.编译器是无法知道你具体的类型信息，但是它可以确保你放置的对象具有 T 类型。因此，即使擦除了有关实际类型的信息，编译器是可以确保 使用 类型的一致性。(就是它不知道你是什么类型，但是却可以确保在编译期间类型的一致性，否则编译时通过不了的)

2因为擦除在方法体中移除了类型信息，所以在运行时的问题就是边界：即对象进入和离开方法的地点。这正是编译器在编译期执行了 类型检查 并   插入转型代码   的地点。实际上就算你用了泛型，也是要进行强制类型转换的，但由于 泛型代码 在编译期确保了 类型的一致性，所以它允许你不用显式的编写 转型代码，而是编译器 自动帮你处理(编译器帮你进行类型转换了)。

2.因此你编写的代码噪声将更小。

3.所以在泛型中的所有动作都发生在边界处---对传递进来的值进行额外的编译期检查，并对传递出去的值进行转型。记住,"边界就是发生动作的地方

上面观点大部分是源自编程思想，还记得里面有一句话，java选择泛型的擦除是一种中庸之道。如果重新设计java未必还会这样做。

42. hibernate和mybatis的区别是什么？

**一、两者最大的区别**

　　针对简单逻辑，Hibernate与MyBatis都有相应的代码生成工具，可以生成简单基本的DAO层方法。

　　针对高级查询，MyBatis需要手动编写SQL语句，以及ResultMap，而Hibernate有良好的映射机制，开发者无需关心SQL的生成与结果映射，可以更专注于流程。

**二、开发难度对比**

　　Hibernate的开发难度大于MyBatis，主要由于Hibernate比较复杂，庞大，学习周期比较长。

　　MyBatis则相对简单，并且MyBatis主要依赖于生气了的书写，让开发者刚进更熟悉。

**三、sql书写比较**

Hibernate也可以自己写sql来指定需要查询的字段，但这样就破坏了Hibernate开发的简洁性，不过Hibernate具有自己的日志统计。

　　MyBatis的sql是手动编写的，所以可以按照要求指定查询的字段，不过没有自己的日志统计，所以要借助Log4j来记录日志。

**四、数据库扩展性计较**

　　Hibernate与数据库具体的关联在XML中，所以HQL对具体是用什么数据库并不是很关心

　　MyBatis由于所有sql都是依赖数据库书写的，所以扩展性、迁移性比较差。

**五、缓存机制比较**

　　Hibernate的二级缓存配置在SessionFactory生成配置文件中进行详细配置，然后再在具体的表对象映射中配置那种缓存。

　　MyBatis的二级缓存配置都是在每个具体的表对象映射中进行详细配置，这样针对不同的表可以自定义不同的缓冲机制，并且MyBatis可以在命名空间中共享相同的缓存配置和实例，通过Cache-ref来实现。

    两者比较，因为Hibernate对查询对象有着良好的管理机制，用户无需关心SQL，所以在使用二级缓存时如果出现脏数据，系统会报出错误提示。 而MyBatis在这一方面使用二级缓存时需要特别小心，如果不能完全去顶数据更新操作的波及范围，避免cache的盲目使用，否则，脏数据的出现会给系统的正常运行带来很大的隐患。

43. http请求中常见的响应码有哪些，分别是什么含义？

200            OK

205            重置内容

400            错误请求

404            未找到

405            不允许的方法

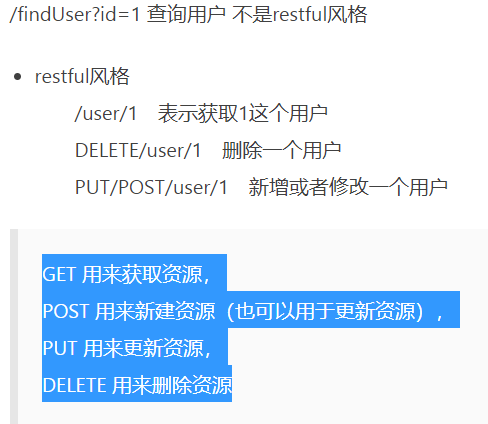
500            内部服务器错误

44. 说说SpringMVC中的常用注解

<https://www.cnblogs.com/caijh/p/7744604.html>

45. 谈谈你对restful的认识

Restful是一种面向资源的架构风格，可以简单理解为：使用URL定位资源，用HTTP动词(GET,POST,DELETE,PUT）描述操作。



##### 使用Restful的好处：

  透明性，暴露资源存在。  
  充分利用 HTTP 协议本身语义。  
  **无状态，这点非常重要**。在调用一个接口（访问、操作资源）的时候，可以不用考虑上下文，不用考虑当前状态，极大的降低了复杂度。

46. 分布式锁用过吗？工作原理是什么？

分布式锁的概述

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。在分布式系统中，常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。

redis

实现原理:

redis分布式锁即可以结合zk分布式锁锁高度安全和memcached并发场景下效率很好的优点，其实现方式和memcached类似，采用setnx即可实现。需要注意的是，这里的redis也需要设置超时时间。以避免死锁。可以利用jedis客户端实现。

适应场景：

对可靠性，安全性要求非常高的，高并发场景下使用。

47. cookie和session的区别是什么？

**session与cookie的区别**

（1）Cookie以文本文件格式存储在浏览器中，而session存储在服务端它存储了限制数据量。它只允许4kb它没有在cookie中保存多个变量。

（2）cookie的存储限制了数据量，只允许4KB，而session是无限量的

（3）我们可以轻松访问cookie值但是我们无法轻松访问会话值，因此它更安全

（4）设置cookie时间可以使cookie过期。但是使用session-destory（），我们将会销毁会话。

总结：如果我们需要经常登录一个站点时，最好用cookie来保存信息，要不然每次登陆都特别麻烦，如果对于需要安全性高的站点以及控制数据的能力时需要用会话效果更佳，当然我们也可以结合两者，使网站按照我们的想法进行运行

48. ajax如何实现跨域访问呢？

1、响应头添加Header允许访问

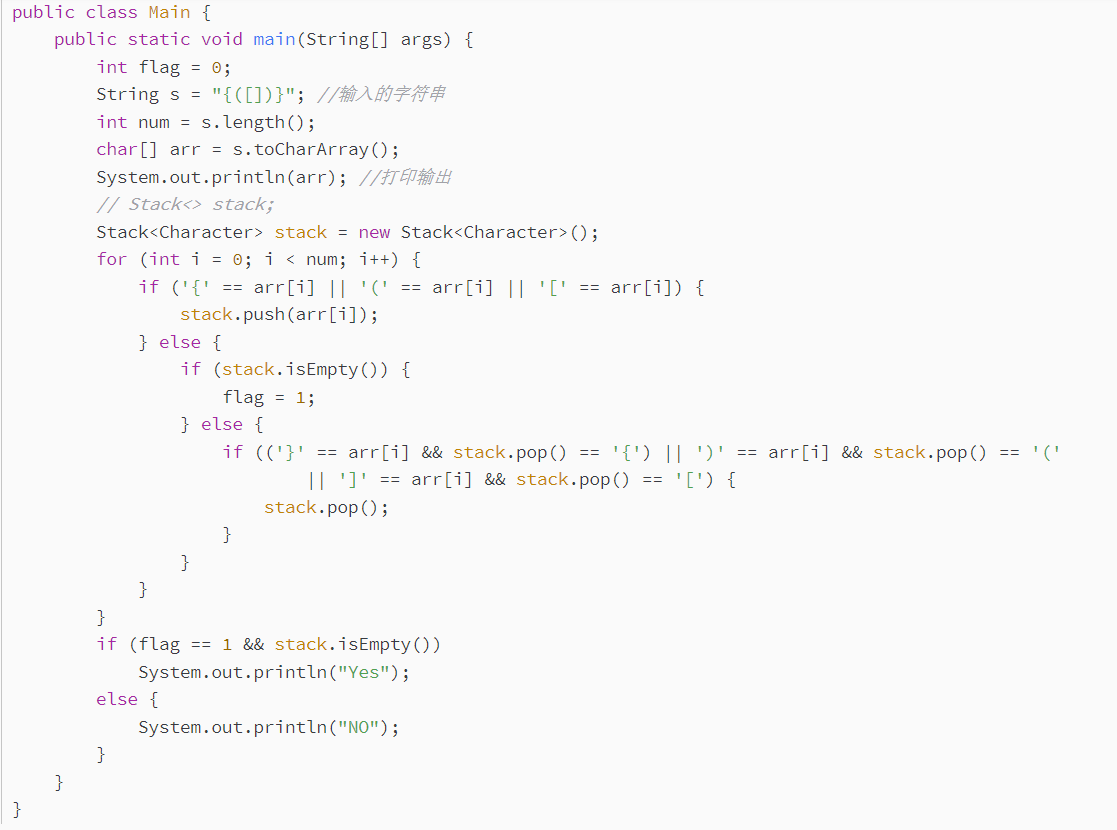
2、jsonp 只支持get请求不支持post请求

3、httpClient内部转发

4、使用接口网关——nginx、springcloud zuul   (互联网公司常规解决方案)

49. 编程题：编写一个程序，检测一个字符串中的括号是否配对。

如："(1)(2)"就是配对的 "((23)45)"也是配对的，"(1+1)\*2)"不是配对的 "1+2)(3+4(22)"也不是配对的。



50. Jdk1.7的HashMap，在多线程访问下，会有什么问题？ 为什么？

HashMap在并发执行put操作时会引起死循环，是因为多线程会导致HashMap的Entry链表形成环形数据结构，一旦形成环形数据结构，Entry的next节点永远不为空，就会产生死循环获取Entry。

如果多个线程同时检测到元素个数超过数组大小 \* loadFactor，这样就会发生多个线程同时对 Node 数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给 table，也就是说其他线程的都会丢失，并且各自线程 put 的数据也丢失