JAVA基础

HashMap

第一种:

```
Map map = new HashMap();
Iterator iter = map.entrySet().iterator();
while (iter.hasNext()) {
Map.Entry entry = (Map.Entry) iter.next();
Object key = entry.getKey();
Object val = entry.getValue();
}
```

效率高,以后一定要使用此种方式!

第二种:

```
Map map = new HashMap();
Iterator iter = map.keySet().iterator();
while (iter.hasNext()) {
Object key = iter.next();
Object val = map.get(key);
}
```

效率低,以后尽量少使用!

Node<K,V>[] table

HashMap在链表长度大于8的时候树化为红黑树，为什么使用红黑树不用AVL树：他们都是高度平衡的树形结构，但是AVL树更加平衡，这会导致插入和删除比较慢，红黑树牺牲了一部分查询速度，但是插入和删除较快。AVL树从根到任何叶子结点的最短和最长路径之差最多为1；

HashMap注释上解释了到8才树化的原因是TreeNodes占用空间是普通Nodes的两倍，所以只有当bin包含足够多的节点时才会转成TreeNodes，而是否足够多就是由TREEIFY_THRESHOLD的值决定的。当bin中节点数变少时，又会转成普通的bin。并且我们查看源码的时候发现，链表长度达到8就转成红黑树，当长度降到6就转成普通bin。并且离散性很好的时候链表长度达到8的概率很低，10的-8次方，所以定为8.

HashMap的hash冲突解决使用的是链地址法，每个节点都是Node的形式，保存着hash，key，value和下一个节点next。

HashMap的数组为Node[],

HashMap扩容的流程：

1. 判断是否超过最大容量2的31次方，不再进行扩充，int的最大值2的31次方-1
2. 容量没有超过最大值则变为之前的2倍，阈值变为之前2倍
3. 遍历旧的hash表
   (1) 当前节点不是以链表的形式存在，直接把当前节点放到e.hash & newCap-1的位置
   (2) 如果是红黑树的形式直接split
   (3) 如果是以链表的形式存在，用e.hash & oldCap == 0判断是不是需要移动该节点，判断左边新增的那一位是否为1即可判断节点留在该位置lo还是去高位hi，把各自的链表头节点放到对应位置上即可完成整个链表的移动，因为使用了尾节点所以不会造成逆序

LinkedHashMap

迭代HashMap的顺序并不是HashMap放置的顺序，也就是无序
LinkedHashMap就闪亮登场了，它虽然增加了时间和空间上的开销，但是通过维护一个运行于所有条目的双向链表，LinkedHashMap保证了元素迭代的顺序。该迭代顺序可以是插入顺序或者是访问顺序。
1、LinkedHashMap可以认为是HashMap+LinkedList，即它既使用HashMap操作数据结构，又使用LinkedList维护插入元素的先后顺序。
2、LinkedHashMap的**基本实现思想就是----多态**

1、K key
2、V value
3、Entry<K, V> next
4、int hash
5、Entry<K, V> before
6、Entry<K, V> after
比hashMap节点的属性多了后两个用于维护entry的插入顺序
初始化：
重写init()方法，创建一个空的header
插入
LinkedHashMap并未重写父类HashMap的put方法，而是重写了1 newNode(hash, key, value, null);（将新插入的节点指向last，将last和新插入的节点连起来）
2 putTreeVal(this, tab, hash, key, value)//newTreeNode(h, k, v, xpn)

3 afterNodeAccess(e);在accessOrder=true表示按照插入顺序排序时当插入节点存在，将插入节点移到尾节点
4 afterNodeInsertion(evict); 该方法用来移除最老的首节点，首先方法要能执行到if语句里面，必须 evict = true（=false表示hashMap正在创建），并且 头节点不为null，并且 removeEldestEntry(first) 返回true
LinkedHashMap还可以用来实现LRU算法，重写removeEldestEntry，默认返回false，通过判断size是不是大于maxSize来返回true

删除
重写afterNodeRemoval方法

查找
如果accessOrder=true会调用afterNodeAccess将节点移动到尾部

迭代
通过遍历链表的方式来遍历整个 LinkedHashMap

```
1        Iterator<Map.Entry<String,String>> iterator = map.entrySet().iterator();
2        while(iterator.hasNext()){
3            Map.Entry<String,String> entry = iterator.next();
4            System.out.println(entry.getKey()+"----"+entry.getValue());
5        }
```

String为什么不可变

用final修饰 不能被继承
原因1:为了安全，
原因2，用String做hashMap的key时候不可变，如果可变的话之前做的hash操作就没用了
原因2: String字符串常量池，相同内容的String指向同一个内存地址，在大量使用字符串的情况下可以节省内存空间，提高效率

红黑树

concurrentHashMap

JDK1.5中，ConcurrentHashMap使用的是分段锁技术,将ConcurrentHashMap将锁一段一段的存储，然后给每一段数据配一把锁

（segment），当一个线程占用一把锁（segment）访问其中一段数据的时候，其他段的数据也能被其它的线程访问，默认分配16个segment。默认比Hashtable效率提高16倍。

JDK1.8中，ConcurrentHashMap取消了segment分段锁，而采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

判断Node[]数组是否初始化，没有则进行初始化操作

通过hash定位Node[]数组的索引坐标，是否有Node节点，如果没有则使用CAS进行添加（链表的头结点），添加失败则进入下次循环。

检查到内部正在扩容，如果正在扩容，就帮助它一块扩容。

如果f!=null，则使用synchronized锁住f元素（链表/红黑二叉树的头元素）

4.1 如果是Node(链表结构)则执行链表的添加操作。

4.2 如果是TreeNode(树型结果)则执行树添加操作。

判断链表长度已经达到临界值8 就需要把链表转换为树结构。

HashTable：底层使用synchronnized来进行同步操作，某个线程写入时其他线程既不能写入也不能读取

Synchronized在JDK1.6之后的优化：

1. 锁升级策略：无锁(CAS)->偏向锁->轻量级锁->重量级锁

通过对象头和监视器进行实现，对象头包含25bit的hashcode，4bit的对象分代年龄，1bit的偏向锁标志位和2bit的锁标志位

2. 自旋锁和适应性自旋锁
3. 锁消除和锁粗化

虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

Synchronized和Reentrantlock的区别：

两者都是可重入锁

1. ReenTrantLock提供了一种能够中断等待锁的线程的机制
2. ReenTrantLock可以指定公平锁还是非公平锁，synchronized只能是非公平锁
3. ReentrantLock依赖API，synchronized依赖JVM底层实现

JDK1.6 之后，synchronized 和 ReenTrantLock 的性能基本是持平了。所以网上那些说因为性能才选择 ReenTrantLock 的文章都是错的！JDK1.6之后，性能已经不是选择synchronized和ReenTrantLock的影响因素了！ 而且虚

拟机在未来的性能改进中会更偏向于原生的synchronized，所以还是提倡在synchronized能满足你的需求的情况下，优先考虑使用synchronized关键字来进行同步！优化后的synchronized和ReenTrantLock一样，在很多地方都是用到了CAS操作。

ArrayList，LinkedList，Vector：

ArrayList：底层采用数组实现，优势是：
1．根据元素下标遍历元素和查找元素效率更高
2．在数组的基础上封装了对元素的操作方法
3．可以自动扩容，每次扩容现有容量的50%
缺点：
插入和删除的效率比较低，在尾部添加效率高
根据内容查询元素的速度慢

Vector和ArrayList类似，但是Vector是同步类；
1、ArrayList在内存不够时默认是扩展50%，最大容量是int最大值-8；Vector是当增长因子>0,默认扩展增加一个增长因子,否则默认扩展1倍。
当更多的元素被添加的时候，Vector和ArrayList需要更多的空间。Vector每次扩容会增加一倍的空间，而ArrayList增加50%。
2、Vector的方法加了synchronized, 而ArrayList则没有。Vector属于线程安全级别的，但是大多数情况下不使用Vector，因为线程安全需要更大的系统开销

因为LinkedList不像ArrayList一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是ArrayList最坏的一种情况，时间复杂度是O(n)，而LinkedList中插入或删除的时间复杂度仅为O(1)。
ArrayList在插入数据时还需要更新索引（除了插入数组的尾部）

基本类型和包装类型：

基本类型并不具有对象的性质，为了与其他对象“接轨”就出现了包装类型（如我们在使用集合类型Collection时就一定要使用包装类型而非基本类型），它相当于将基本类型“包装起来”，使得它具有了对象的性质，并且为其添加了属性和方法，丰富了基本类型的操作。
装箱和拆箱的工作，答案就是valueOf()和xxxValue()
自动装箱：包装器的valueOf方法
自动拆箱：包装器的xxxValue方法，value是一个内部变量，直接返回

Integer类型的自动装箱：判断拿到的i是否在某个范围内，如果满足添加条件的话，则返回一个数组对应下标的值IntegerCache，则直接用i来new一个新的Integer对象。好处是省去了多次创建对象的内存开销

String，StringBuffer，StringBuilder

在这方面运行速度快慢为：StringBuilder > StringBuffer > String
String为字符串常量，而StringBuilder和StringBuffer均为字符串变量，即String对象一旦创建之后该对象是不可更改的，但后两者的对象是变量，是可以更改的。
在线程安全上，StringBuilder是线程不安全的，而StringBuffer是线程安全的

字符串编码方式：

1. ASCII码：对英语字符与二进制位之间的关系，做了统一规定。这被称为ASCII码，ASCII码一共规定了128个字符的编码，只占用了一个字节的后面7位，最前面的1位统一规定为0。0~31 是控制字符如换行回车删除等，32~126 是打印字符，可以通过键盘输入并且能够显示出来。
2. 非ASCII码：利用闲置的第位，扩展到256个字符
3. Unicode：是一种所有符号的编码，规定了二进制代码却没有规定怎么存储
4. UTF-8:使用最广泛的Unicode码实现方式，是一种可变长度的字节编码，用1-4个字节来存储
5. GBK编码：中国的中文字符，仅限于中文字符

泛型：

Jdk1.5中引入的新特性，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢?
顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。
泛型有三种使用方式，分别为：泛型类、泛型接口、泛型方法
class 类名称 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{
  private 泛型标识 /*（成员变量类型）*/ var;

public interface Generator<T> {

```
    public T next();
}
```

是否拥有泛型方法，与其所在的类是不是泛型没有关系。

```
    public <T> void show_2(T t){
        System.out.println(t.toString());
    }
```

限制泛型类型

class 类名称<T extends anyClass>

comparable和comparator：

Comparable可以认为是一个内比较器，实现了Comparable接口的类有一个特点，就是这些类是可以和自己比较的，实现此接口的对象列表（和数组）可以通过 Collections.sort （和 Arrays.sort ）进行自动排序，至于具体和另一个实现了Comparable接口的类如何比较，则依赖compareTo方法的实现，compareTo方法也被称为自然比较方法。如果开发者add进入一个Collection的对象想要Collections的sort方法帮你自动进行排序的话，那么这个对象必须实现Comparable接口。compareTo方法的返回值是int，有三种情况：

1、比较者大于被比较者（也就是compareTo方法里面的对象），那么返回正整数

2、比较者等于被比较者，那么返回0

3、比较者小于被比较者，那么返回负整数

Comparator可以认为是是一个外比较器，个人认为有两种情况可以使用实现Comparator接口的方式：

1、一个对象不支持自己和自己比较（没有实现Comparable接口），但是又想对两个对象进行比较

2、一个对象实现了Comparable接口，但是开发者认为compareTo方法中的比较方式并不是自己想要的那种比较方式

Comparator接口里面有一个compare方法，方法有两个参数T o1和T o2，是泛型的表示方式，分别表示待比较的两个对象，方法返回值和Comparable接口一样是int，有三种情况：

1、o1大于o2，返回正整数

2、o1等于o2，返回0

3、o1小于o2，返回负整数

多态：

多态是面向对象编程语言的重要特性，它允许基类的指针或引用指向派生类的对象，而在具体访问时实现方法的动态绑定。Java 对于方法调用动态绑定的实现主要依赖于方法表，但通过类引用调用(invokevitual)和接口引用调用(invokeinterface)的实现则有所不同。

类引用调用时子类的方法和父类的方法在方法表中的偏移量一样，所以只用知道偏移量是多少，先去子类中找，找不到按照同样的偏移量去父类中找，如果子类重写了，则直接执行子类的方法。

接口引用因为多实现的方式，在运行时搜索一个实现了这个接口方法的对象，找出适合的方法进行调用，所以接口的方法调用慢于类的方法调用

JVM 的方法调用指令有四个，分别是 invokestatic，invokespecial，invokesvirtual 和 invokeinterface。前两个是静态绑定，后两个是动态绑定的，加上invokedynamic，lambda表达式

Java 的方法调用有两类，动态方法调用与静态方法调用。

静态方法调用是指对于类的静态方法的调用方式，是静态绑定的

动态方法调用需要有方法调用所作用的对象，是动态绑定的。

类调用 (invokestatic) 是在编译时就已经确定好具体调用方法的情况。

实例调用 (invokevirtual)则是在调用的时候才确定具体的调用方法，这就是动态绑定，也是多态要解决的核心问题。

商用虚拟机为了保证性能，通常会使用虚方法表和接口方法表，而不是每次都执行一遍上面的步骤。以虚方法表为例，虚方法表在类加载的解析阶段填充完成，其中存储了所有方法的直接引用。也就是说，动态分派在填充虚方法表的时候就已经完成了。

在子类的虚方法表中，如果子类覆盖了父类的某个方法，则这个方法的直接引用指向子类的实现；而子类没有覆盖的那些方法，比如 Object 的方法，直接引用指向父类或 Object 的实现。

https://blog.csdn.net/qq_17305249/article/details/90231329

接口和抽象类：

接口是对动作的抽象，而抽象类是对根源的抽象。抽象类更偏向一种模板

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

2、抽象类要被子类继承，接口要被类实现。

3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类。
6、抽象方法只能申明，不能实现，接口是设计的结果 ，抽象类是重构的结果
7、抽象类里可以没有抽象方法
8、如果一个类里有抽象方法，那么这个类只能是抽象类
9、抽象方法要被实现，所以不能是静态的，也不能是私有的。
10、接口可继承接口，并可多继承接口，但类只能单根继承。

反射：

对于任意一个类，都能知道这个类的所以属性和方法；
对于任何一个对象，都能够调用它的任何一个方法和属性；
这样动态获取新的以及动态调用对象方法的功能就叫做反射。
Class可以说是反射能够实现的基础
Class c1 = Test.class; //这说明任何一个类都有一个隐含的静态成员变量class，这种方式是通过获取类的静态成员变量class得到的()
Class c2 = test.getClass();// test是Test类的一个对象，这种方式是通过一个类的对象的getClass()方法获得的 (对于基本类型无法使用这种方法)
Class c3 = Class.forName("com.catchu.me.reflect.Test"); //这种方法是Class类调用forName方法，通过一个类的全量限定名获得（基本类型无法使用此方法）
私有对象的值怎么设置?
AccessibleObject为我们提供了一个方法 setAccessible(boolean flag)，该方法的作用就是可以取消 Java 语言访问权限检查。所以任何继承AccessibleObject的类的对象都可以使用该方法取消 Java 语言访问权限检查。
java.lang.reflect.Field： 对应类变量
java.lang.reflect.Method： 对应类方法
java.lang.reflect.Constructor： 对应类构造函数

Field.set Method.invoke constructor.newInstance 反射不支持自动封箱，传入参数时要小心（自动封箱是在编译期间的，而反射在运行期间）

Java的反射机制，操作的就是这个.class文件，首先加载相应类的字节码（运行eclipse的时候，.class文件的字节码会加载到内存中），随后解剖（反射reflect）出字节码中的构造函数、方法以及变量（字段），或者说是取出，我们先来定义一个类Animal，里面定义一些构造函数，方法，以及变量：

快速失败(fail-fast)和安全失败(fail-safe)

java.util包下面的所有的集合类都是快速失败的。快速失败的迭代器会抛出ConcurrentModificationException异常。当在迭代一个集合的时候，如果有另外一个线程在修改这个集合，就会抛出ConcurrentModification异常。不能在多线程下发生并发修改。
查看ArrayList源代码，在next方法执行的时候，会执行checkForComodification()方法

java.util.concurrent包下面的所有的类都是安全失败的。Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

Synchronized实现原理及锁优化

实现原理：Java对象头和监视器monitor
synchronized用的锁是存在Java对象头里的，那么什么是Java对象头呢?
Hotspot虚拟机的对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。其中Klass Point是是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，
Mark Word用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键

synchronized中的锁优化:

自旋锁和适应性自旋锁：

JDK 1.6 引入了更加聪明的自旋锁，即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。它怎么做呢？线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源

锁粗化：就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。

偏向锁获取锁顺序：

1.检测Mark Word是否为可偏向状态，即是否为偏向锁1，锁标识位为01；

2.若为可偏向状态，则测试线程ID是否为当前线程ID，如果是，则执行步骤（5），否则执行步骤（3）；

3.如果线程ID不为当前线程ID，则通过CAS操作竞争锁，竞争成功，则将Mark Word的线程ID替换为当前线程ID，否则执行线程（4）；

4.通过CAS竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块；

5.执行同步代码块

轻量级锁获取锁顺序：

1.判断当前对象是否处于无锁状态（hashcode、0、01），若是，则JVM首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝（官方把这份拷贝加了一个Displaced前缀，即Displaced Mark Word）；否则执行步骤（3）；

2.JVM利用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指正，如果成功表示竞争到锁，则将锁标志位变成00（表示此对象处于轻量级锁状态），执行同步操作；如果失败则执行步骤（3）；

3.判断当前对象的Mark Word是否指向当前线程的栈帧，如果是则表示当前线程已经持有当前对象的锁，则直接执行同步代码块；否则只能说明该锁对象已经被其他线程抢占了，这时轻量级锁需要膨胀为重量级锁，锁标志位变成10，后面等待的线程将会进入阻塞状态；

重量级锁：

重量级锁通过对象内部的监视器（monitor）实现，其中monitor的本质是依赖于底层操作系统的Mutex　Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高。

1.对于普通同步方法，锁是当前实例对象　2.对于静态同步方法，锁是当前类的class对象　3.对于同步方法块，锁是synchronized括号里配置的对象
JVM基于进入和退出Monitor对象来实现方法同步和代码块同步 monitorenter和monitorexit

Volatile

对有volatile关键字修饰的变量进行写操作的时候会多出一行Lock的指令，该指令在多核处理器的条件下会：
1.将当前处理器缓存行的数据写回到系统内存　它会锁住缓存
2.这个写操作会使其他处理器缓存了该内存地址的数据无效
其他处理器使用过期数据的时候会重新从内存中缓存该数据
编译器生成字节码的时候会在指令序列中插入内存屏障来禁止特定类型的处理器重排序　Load--读　Store--写　volatile读后面插入LoadStore和LoadLoad屏障 volatile写前面插入StoreStore 后面插入StoreLoad屏障

CountDownLatch 的三种典型用法:

①某一线程在开始运行前等待n个线程执行完毕。将 CountDownLatch 的计数器初始化为n ： new CountDownLatch(n) ，每当一个任务线程执行完毕，就将计数器减1 countdownlatch.countDown()，当计数器的值变为0时，在CountDownLatch上 await() 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

②实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的 CountDownLatch 对象，将其计数器初始化为 1 ： new CountDownLatch(1) ，多个线程在开始执行任务前首先 coundownlatch.await()，当主线程调用 countDown() 时，计数器变为0，多个线程同时被唤醒。

③死锁检测：一个非常方便的使用场景是，你可以使用n个线程访问共享资源，在每次测试阶段的线程数目是不同的，并尝试产生死锁

可以理解为发射火箭，一步一步来

内部实现，有个Sync类继承AQS，实现tryAcquireShared，方法会判断get

CyclicBarrier

CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。

CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

可以理解为赛马，发令枪开始后所有马一起开跑

Semaphore

synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

执行 acquire 方法阻塞，直到有一个许可证可以获得然后拿走一个许可证；每个 release 方法增加一个许可证，这可能会释放一个阻塞的acquire方法。然而，其实并没有实际的许可证这个对象，Semaphore只是维持了一个可获得许可证的数量。 Semaphore经常用于限制获取某种资源的线程数量。

CAS操作

CAS是一种乐观锁的实现方式，是轻量级锁，JUC中很多工具类的实现就是基于CAS，CAS的操作流程如左图所示，线程在读取数据时不进行加锁，在准备写回数据时，比较原值是否修改，若未被其他线程修改则写回，若已被修改，则重新执行读取流程，这是一种乐观策略，认为并发操作并不总会发生，比较并写回的操作时通过操作系统的原语实现的，保证执行过程中不会被中断。CAS容易出现ABA问题，比如按右图所示的时序，线程T1在读取完值A后，发生过2次写入，先有线程T2写回了B，又有线程T3写回了A，此时T1在写

回时进行比较，发现值还是A，就无法判断是否发生过修改。ABA问题不一定会影响到结果，但还是需要防范，解决的办法可以增加额外的标志位或者时间戳。JUC工具包中提供了这样的类。AtomicStampedReference和AtomicMarkableReference

Collections

List,Map和set接口在获取元素时各有什么特点：
List与Set都是单列元素的集合，它们有一个功共同的父接口Collection。

Set里面不允许有重复的元素，

存元素：add方法有一个boolean的返回值，当集合中没有某个元素，此时add方法可成功加入该元素时，则返回true；当集合含有与某个元素equals相等的元素时，此时add方法无法加入该元素，返回结果为false。
取元素：没法说取第几个，只能以Iterator接口取得所有的元素，再逐一遍历各个元素。

List表示有先后顺序的集合，

存元素：多次调用add(Object)方法时，每次加入的对象按先来后到的顺序排序，也可以插队，即调用add(int index,Object)方法，就可以指定当前对象在集合中的存放位置。
取元素：方法1：Iterator接口取得所有，逐一遍历各个元素
　　方法2：调用get(index i)来明确说明取第几个。

Map是双列的集合，存放用put方法:put(obj key,obj value)，每次存储时，要存储一对key/value，不能存储重复的key，这个重复的规则也是按equals比较相等。
取元素：用get(Object key)方法根据key获得相应的value。
　　也可以获得所有的key的集合，还可以获得所有的value的集合，
　　还可以获得key和value组合成的Map.Entry对象的集合。

如何线程安全的实现一个计数器

count.incrementAndGet();

生产者消费者模式：

https://blog.csdn.net/weixin_30456039/article/details/96500514
BlockingQueue.put和BlockingQueue.take

单例模式的实现：

饿汉，懒汉，DCLhttp://www.blogjava.net/kenzhh/archive/2013/03/15/357824.html

这些方式有个问题是通过反射的方式构建对象，构建出来的对象并不等于原始对象，解决方式是使用enum来进行单例模式或者可以修改构造器，让它在被要求创建第二个实例的时候抛出异常。
传统单例的另一个问题是,一旦实现可序列化接口,它们就不再是 Singleton, 因为 readObject() 方法总是返回一个新实例, 就像 Java 中的构造函数一样。通过使用 readResolve() 方法, 通过在以下示例中替换 Singeton 来避免这种情况:

反射攻击：

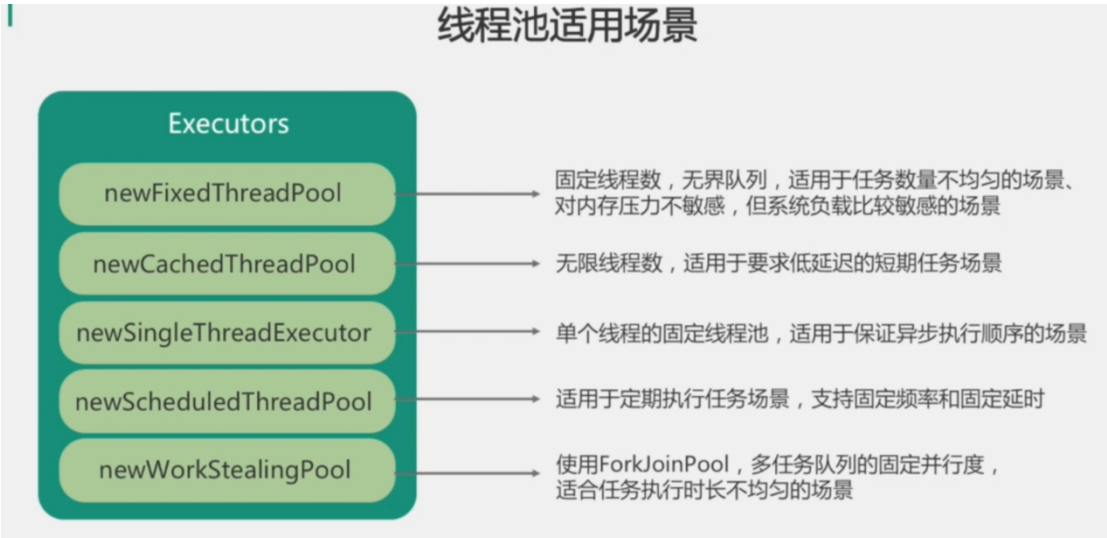
```
        Class<ElvisModified> classType = ElvisModified.class;

         Constructor<ElvisModified> c =
classType.getDeclaredConstructor(null);
         c.setAccessible(true);
         ElvisModified e1 = (ElvisModified)c.newInstance();
         ElvisModified e2 = ElvisModified.getInstance();
         System.out.println(e1==e2);

   private static boolean flag = false;

   private ElvisModified(){
      synchronized(ElvisModified.class)
      {
         if(flag == false)
         {
            flag = !flag;
         }
         else
         {
            throw new RuntimeException("单例模式被侵犯！");
```

```
        }
    }
```

```
1. public enum SingletonClass
2. {
3.     INSTANCE;
4.
5.     public void test()
6.     {
7.         System.out.println("The Test!");
8.     }
9. }
```

线程池

四个线程池类型：



corePoolSize核心线程数，默认情况下核心线程一直存活，maximumPoolSize设置最大线程数，决定线程池最多可以创建多少个线程，keepAliveTime和unit用来设置线程的空闲时间和空闲时间的单位，当线程闲置超过空闲时间时，就会被销毁。workQueue设置缓冲队列，图中左下方的三个队列是设置线程池缓冲队列最常用的三个， 其中ArrayBlockingQueue是一个有界队列，LinkedBlockingQueue是无界队列，synchronousQueue是一个同步队列，内部没有缓冲区。threadFactory设置线程池工厂方法，线程工厂用来创建新的线程，可以用来对线程的一些属性进行定制，一般使用默认工厂类即可。handler设置线程池满时的拒绝策略，如右下角所示，Abort线程池满后抛出异常，Discard会在提交失败时直接对任务进行丢弃。CallerRuns策略，会在提交失败时，由提交任务的线程直接执行提交的任务。DiscardOldest会丢弃最早提交的任务

线程池的状态：

其中AtomicInteger变量ctl的功能非常强大：利用低29位表示线程池中线程数，通过高3位表示线程池的运行状态：
1、RUNNING：-1 << COUNT_BITS，即高3位为111，该状态的线程池会接收新任务，并处理阻塞队列中的任务；
2、SHUTDOWN： 0 << COUNT_BITS，即高3位为000，该状态的线程池不会接收新任务，但会处理阻塞队列中的任务；
3、STOP ： 1 << COUNT_BITS，即高3位为001，该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务；
4、TIDYING ： 2 << COUNT_BITS，即高3位为010，该状态表示线程池对线程进行整理优化；
5、TERMINATED： 3 << COUNT_BITS，即高3位为011，该状态表示线程池停止工作；

线程池execute内部实现

1.首次通过workCountof()获知当前线程池中的线程数,ctl跟2^29次方-1做与运算，如果小于corePoolSize, 就通过addWorker()创建线程并执行该任务；
否则，将该任务放入阻塞队列；
2. 如果能成功将任务放入阻塞队列中,
如果当前线程池是非RUNNING用ctl<0来判断状态，则将该任务从阻塞队列中移除，然后执行reject()处理该任务；
如果当前线程池处于RUNNING状态，则需要再次检查线程池（因为可能在上次检查后，有线程资源被释放），是否有空闲的线程；如果有则执行该任务；
3、如果不能将任务放入阻塞队列中,说明阻塞队列已满；那么将通过addWoker()尝试创建一个新的线程去执行这个任务；如果addWoker()执行失败，说明线程池中线程数达到maxPoolSize,则执行reject()处理任务；

AddWorker实现

上来先有一个无限循环retry:
1. 判断线程池的状态是不是RUNNING，不是RUNNING或者是SHUTDOWN并且firstTask初始线程非空或者队列为空，直接返回创建失败
2. 判断线程数是不是超过最大值2^29-1或者根据传入的是否创建核心线程超过核心线程数或者最大线程数，如果超过直接返回false

3. CAS操作将当前线程数+1，失败了返回到retry，成功了再次判断状态，如果不是开始进入的状态，也返回到retry重新创建
4. 开始创建工作线程，新建一个worker，获取主锁
5. 如果线程池状态是RUNNING或者SHUTDOWN并且firstTask为空，就将worker放入workers中
6. 添加worker成功，调用worker中的Thread的start方法，内部会调用runworker方法
7. 最后，如果线程启动失败，把刚才加上去的线程数再减1

线程池的关闭（2种）

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

线程回调

RPC，全称为Remote Procedure Call，即远程过程调用，它是一个计算机通信协议。它允许像调用本地服务一样调用远程服务，
https://blog.csdn.net/joenqc/article/details/77198019
RPC架构分为三部分：
1）服务提供者，运行在服务器端，提供服务接口定义与服务实现类。
2）服务中心，运行在服务器端，负责将本地服务发布成远程服务，管理远程服务，提供给服务消费者使用。
3）服务消费者，运行在客户端，通过远程代理对象调用远程服务。

将本地的接口调用转换成JDK的动态代理，在动态代理中实现接口的远程调用

线程回调：线程之间的通信

你到一个商店买东西，恰好没货，给店员留下了电话，过几天有货了店员打你电话，你接到电话就去取货。
电话号码：回调函数。 店员记录电话号码：登记回调函数
店里有货：触发回调函数的事件。 店员打你电话：触发回调函数
你去取货：响应回调事件

线程的终止方法

1. 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止 2. 使用stop方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果） 3. 使用interrupt方法中断线程。
当对一个线程调用了 interrupt()之后，如果该线程处于被阻塞状态（比如执行了wait、sleep或join等方法），那么会立即退出阻塞状态，并抛出一个InterruptedException异常，在代码中catch这个异常进行后续处理。如果线程一直处于运行状态，那么只会把该线程的中断标志设置为 true，仅此而已，所以 interrupt()并不能真正的中断线程

生产者消费者模型

https://blog.csdn.net/u010983881/article/details/78554671

Sleep和yeild方法的区别：

sleep()
　　sleep()方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是sleep()方法不会释放“锁标志”，也就是说如果有synchronized同步块，其他线程仍然不能访问共享数据。

wait()
　　wait()方法需要和notify()及notifyAll()两个方法一起介绍，这三个方法用于协调多个线程对共享数据的存取，所以必须在synchronized语句块内使用，也就是说，调用wait()，notify()和notifyAll()的任务在调用这些方法前必须拥有对象的锁。注意，它们都是Object类的方法，而不是Thread类的方法。
　　wait()方法与sleep()方法的不同之处在于，wait()方法会释放对象的“锁标志”。当调用某一对象的wait()方法后，会使当前线程暂停执行，并将当前线程放入对象等待池中，直到调用了notify()方法后，将从对象等待池中移出任意一个线程并放入锁标志等待池中，只有锁标志等待池中的线程可以获取锁标志，它们随时准备争夺锁的拥有权。当调用了某个对象的notifyAll()方法，会将对象等待池中的所有线程都移动到该对象的锁标志等待池。

除了使用notify()和notifyAll()方法，还可以使用带毫秒参数的wait(long timeout)方法，效果是在延迟timeout毫秒后，被暂停的线程将被恢复到锁标志等待池。

此外，wait()，notify()及notifyAll()只能在synchronized语句中使用，但是如果使用的是ReenTrantLock实现同步，该如何达到这三个方法的效果呢？解决方法是使用ReenTrantLock.newCondition()获取一个Condition类对象，然后Condition的await()，signal()以及signalAll()分别对应上面的三个方法。

yield()

yield()方法和sleep()方法类似，也不会释放“锁标志”，区别在于，它没有参数，即yield()方法只是使当前线程重新回到可执行状态，不会让线程进入阻塞状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行，另外yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和sleep()方法不同。

join()

join()方法会使当前线程等待调用join()方法的线程结束后才能继续执行，Thread.join方法： A执行thread.join表示当前线程等待thread线程终止后才从thread.join返回，底层相当于thread线程调用自身的notifyAll方法

乐观锁和悲观锁的使用场景：

悲观锁：synchronized reentrantLock，DB的行锁、表锁
乐观锁：版本号或者时间戳控制，CAS MVCC

MVCC

大多数的MYSQL事务型存储引擎,如,InnoDB，Falcon以及PBXT都不使用一种简单的行锁机制.事实上,他们都和MVCC–多版本并发控制来一起使用.
大家都应该知道,锁机制可以控制并发操作,但是其系统开销较大,而MVCC可以在大多数情况下代替行级锁,使用MVCC,能降低其系统开销.
Innodb中的隐藏列
Innodb通过undo log保存了已更改行的旧版本的信息的快照。
InnoDB的内部实现中为每一行数据增加了三个隐藏列用于实现MVCC。

列名　　长度(字节)　作用
DB_TRX_ID　　6　　插入或更新行的最后一个事务的事务标识符。（删除视为更新，将其标记为已删除）

DB_ROLL_PTR　7　写入回滚段的撤消日志记录（若行已更新，则撤消日志记录包含在更新行之前重建行内容所需的信息）
DB_ROW_ID　6　行标识（隐藏单调自增id）
结构
数据列 ..　DB_ROW_ID　DB_TRX_ID　DB_ROLL_PTR
MVCC工作过程
MVCC只在READ COMMITED 和 REPEATABLE READ 两个隔离级别下工作。READ UNCOMMITTED总是读取最新的数据行，而不是符合当前事务版本的数据行。而SERIALIZABLE 则会对所有读取的行都加锁

SELECT
InnoDB 会根据两个条件来检查每行记录：

InnoDB只查找版本(DB_TRX_ID)早于当前事务版本的数据行（行的系统版本号<=事务的系统版本号,这样可以确保数据行要么是在开始之前已经存在了，要么是事务自身插入或修改过的）
行的删除版本号(DB_ROLL_PTR)要么未定义（未更新过），要么大于当前事务版本号（在当前事务开始之后更新的）。这样可以确保事务读取到的行，在事务开始之前未被删除。
INSERT
InnoDB为新插入的每一行保存当前系统版本号作为行版本号

DELETE
InnoDB为删除的每一行保存当前的系统版本号作为行删除标识

UPDATE
InnoDB为插入一行新记录，保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行作为行删除标识

读写锁ReentrantReadWriteLock:

16位int的前8位和后8位分别作为读锁和写锁的标志位

死锁

产生条件：
- 互斥条件：该资源任意一个时刻只由一个线程占用。

- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：线程已获得的资源在末使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

解除死锁：

１．破坏互斥条件

这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。

２．破坏请求与保持条件

一次性申请所有的资源。

３．破坏不剥夺条件

占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。

４．破坏循环等待条件

靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件

使用jstack 进程id，最下方会有Found one Java-level deadlock

同步/异步/阻塞/非阻塞

1.同步与异步同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)所谓同步，就是在发出一个*调用*时，在没有得到结果之前，该*调用*就不返回。但是一旦调用返回，就得到

返回值了。换句话说，就是由*调用者*主动等待这个*调用*的结果。而异步则是相反，*调用*在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在*调用*发出后，*被调用者*通过状态、通知来通知调用者，或通过回调函数处理这个调用。

2. 阻塞与非阻塞
阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。
非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

LRU（Least Recently Used）

超时淘汰：对象属性中增加创建时间的时间戳，get的时候判断当前时间减时间戳是否大于超时时间，大于的话直接remove。

缓存一定量的数据，当超过设定的阈值时就把一些过期的数据删除掉
使用LinkedHashMap实现，重写removeEldestEntry方法，原理是插入数据的时候会调用afterNodeInsertion方法判断是否要移除最老的节点

List加HashMap实现

```
public class LRUCache1<K, V> {

    private final int MAX_CACHE_SIZE;
    private Entry first;
    private Entry last;

    private HashMap<K, Entry<K, V>> hashMap;

    public LRUCache1(int cacheSize) {
        MAX_CACHE_SIZE = cacheSize;
        hashMap = new HashMap<K, Entry<K, V>>();
    }

    public void put(K key, V value) {
        Entry entry = getEntry(key);
        if (entry == null) {
            if (hashMap.size() >= MAX_CACHE_SIZE) {
                hashMap.remove(last.key);
                removeLast();
```

```
            }
            entry = new Entry();
            entry.key = key;
        }
        entry.value = value;
        moveToFirst(entry);
        hashMap.put(key, entry);
    }

    public V get(K key) {
        Entry<K, V> entry = getEntry(key);
        if (entry == null) return null;
        moveToFirst(entry);
        return entry.value;
    }

    public void remove(K key) {
        Entry entry = getEntry(key);
        if (entry != null) {
            if (entry.pre != null) entry.pre.next = entry.next;
            if (entry.next != null) entry.next.pre = entry.pre;
            if (entry == first) first = entry.next;
            if (entry == last) last = entry.pre;
        }
        hashMap.remove(key);
    }

    private void moveToFirst(Entry entry) {
        if (entry == first) return;
        if (entry.pre != null) entry.pre.next = entry.next;
        if (entry.next != null) entry.next.pre = entry.pre;
        if (entry == last) last = last.pre;

        if (first == null || last == null) {
            first = last = entry;
            return;
        }

        entry.next = first;
        first.pre = entry;
        first = entry;
        entry.pre = null;
    }

    private void removeLast() {
        if (last != null) {
            last = last.pre;
            if (last == null) first = null;
            else last.next = null;
```

```
        }
    }


    private Entry<K, V> getEntry(K key) {
        return hashMap.get(key);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        Entry entry = first;
        while (entry != null) {
            sb.append(String.format("%s:%s ", entry.key, entry.value));
            entry = entry.next;
        }
        return sb.toString();
    }

    class Entry<K, V> {
        public Entry pre;
        public Entry next;
        public K key;
        public V value;
    }
}
```

Concurrent包

concurrent包的实现示意图： 最底层：volatile变量的读写以及CAS操作 中间层：AQS框架 非阻塞数据结构 最上层：Lock 同步器 阻塞队列 Executor 并发容器

volatile

1.将当前处理器缓存行的数据写回到系统内存 它会锁住缓存
2.这个写操作会使其他处理器缓存了该内存地址的数据无效
其他处理器使用过期数据的时候会重新从内存中缓存该数据
编译器生成字节码的时候会在指令序列中插入内存屏障来禁止特定类型的处理器重排序 Load--读 Store--写 volatile读后面插入LoadStore和LoadLoad屏障 volatile写前面插入StoreStore 后面插入StoreLoad屏障

Synchronized

1.对于普通同步方法，锁是当前实例对象　2.对于静态同步方法，锁是当前类的class对象　3.对于同步方法块，锁是synchronized括号里配置的对象
JVM基于进入和退出Monitor对象来实现方法同步和代码块同步 monitorenter和monitorexit　相当于获取对象对应的monitor的所有权，尝试获得对象的锁
存储在java对象头里　Mark Word里 HashCode，分代年龄和锁标记位 1bit的偏向锁 2bit的锁标志位

CAS操作

JAVA中的CAS操作都是通过sun包下Unsafe类实现，而Unsafe类中的方法都是native方法，由JVM本地实现，最后调用的是Atomic:comxchg这个方法，这个方法的实现放在hotspot下的os_cpu包中，说明这个方法的实现和操作系统、CPU都有关系
三大问题：1.ABA问题 可以更新时追加版本号来解决　2.循环时间长开销大
3.只能保证一个共享变量的原子操作 AtomicReference类 对象之间的原子性

单例模式实现

双重检查锁定 不能重排序
new一个对象可以分为3行伪代码：　1.分配对象的内存空间 2.初始化对象 3.设置instance指向刚分配的内存地址 可能会重新排序

```
public class DoubleCheckLocking{
    private volatile static Instance instance;

    public static Instance getInstance(){
        if(instance == null){
            synchronized(DoubleCheckLocking.class){
                if(instance == null)
                    instance = new Instance();
            }
        }
        return instance;
    }
}
```

静态内部类实现单例模式　通过Class对象的初始化锁 可以重排序

```
public class InstanceFactory{
```

```
    private static class InstanceHolder{
        public static Instance instance = new Instance();
    }
    public Instance getInstance(){
        return InstanceHolder.instance;
    }
}
```

枚举类实现单例模式

```
Public enum Singleton{
    INSTANCE;

    Public void method(){
    }
}
```

JVM

JVM内存模型及分区

线程私有：
虚拟机栈：存储局部变量表（基本数据类型，对象引用），操作数栈，动态链接 方法出口
本地方法栈：native方法
程序计数器：执行的字节码的行号指示器，如果正在执行的是native方法，该区域为空

线程共有；
堆：存放对象实例，分为新生代，老年代和永久代
方法区：被虚拟机加载的类信息，常量，静态变量 1.8之后改为元空间，元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制

JVM参数设置

-Xms设置堆的最小空间大小。
-Xmx设置堆的最大空间大小。
-XX:NewSize设置新生代最小空间大小。

-XX:MaxNewSize设置新生代最大空间大小。
-XX:PermSize设置永久代最小空间大小。
-XX:MaxPermSize设置永久代最大空间大小。
-Xss设置每个线程的堆栈大小。
-XX:+UseG1GC

堆的分区：

一个Eden区和两个from区，8:1:1，一般情况下，新创建的对象都会被分配到Eden区(一些大对象特殊处理),这些对象经过第一次Minor GC后，如果仍然存活，将会被移到Survivor区。对象在Survivor区中每熬过一次Minor GC，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到年老代中。
因为年轻代中的对象基本都是朝生夕死的(80%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。
在GC开始的时候，对象只会存在于Eden区和名为“From”的Survivor区，Survivor区“To”是空的。紧接着进行GC，Eden区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold来设置，默认18)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次GC后，Eden区和From区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次GC前的“From”，新的“From”就是上次GC前的“To”。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。

对象创建方法

使用new关键字	} → 调用了构造函数
使用Class类的newInstance方法	} → 调用了构造函数
使用Constructor类的newInstance方法	} → 调用了构造函数
使用clone方法	} → 没有调用构造函数
使用反序列化	} → 没有调用构造函数

对象内存分配

（1）在函数中定义的基本类型变量（即基本类型的局部变量）和对象的引用变量（即对象的变量名）都在栈内存中分配；

（2）堆内存用来存储由new创建的对象和数组以及对象的实例变量（即全局变量）
还有一个方法区：存储所有对象数据共享区域，存储静态变量和普通方法、静态方法、常量、字符串常量等信息，又叫静态区，是所有线程共享的。

对象的访问定位

Java程序需要通过栈上的reference引用来操作堆上的具体对象。由于reference类型在Java虚拟机规范中只规定了一个指向对象的引用，并没有定义这个引用应该通过何种方式去定位、访问堆中的对象的具体位置，所以对象访问方法也是取决于虚拟机的实现而决定的。目前主流的访问方式有使用句柄和直接指针两种

句柄：reference存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要改变，句柄中存储的是指向对象实例的指针

通过直接指针访问对象（HotSpot使用的方式）：直接指向堆中的对象，堆中的对象包含到方法区中对象类型数据的指针

GC的两种判定方式

引用计数：对象再被创建时，对象头里会存储引用计数器，对象被引用，计数器+1；引用失效，计数器 −1；GC时会回收计数器为0的对象，无法解决对象互相循环引用。
引用链：程序把所有的引用看作图（类似树结构的图），选定一个对象作为GC Root根节点，从该节点开始寻找对应的引用节点并标记，找到这个节点之后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点认为是不可达的无用节点，会被回收。

可以作为GC Root根节点的对象有：

a.虚拟机栈中的引用对象（本地变量表）

b,方法区类静态属性的引用对象

c,方法区常量引用的对象

d,本地方法栈中的引用对象

GC的三种收集方法：

标记清除：标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：效率不高，标记和清除的效率都很低；会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作
标记整理：在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。
复制算法：复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。

GC收集器：

Serial：单线程收集器，进行垃圾收集时要暂停所有其他的工作线程
Parallel：多线程
CMS收集器：采用“标记-清除”算法实现，使用多线程的算法去扫描堆，对发现未使用的对象进行回收。
（1）初始标记：stop the world
（2）并发标记
（3）并发预处理
（4）重新标记。Stop the world
（5）并发清除
（6）并发重置
特点：响应时间优先，减少垃圾收集停顿时间
适应场景：服务器、电信领域等。
G1: 整体来看是基于标记-整理，局部来看是基于复制算法
1、初始标记：标记GC Root能直接关联的对象，并且修改TAMS的值，让下一阶段的用户进行并发运行是，能够正确运用Region创建新对象，这阶段需要停顿，但停顿时间很短

2、并发标记：从GC Root开始对堆进行可达性分析，找出存活的对象，这段耗时较长，但可以与用户线程并发执行。

3、最终标记是为了修正在并发标记阶段因用户程序继续运作导致标记产生变动的那一部分的标记记录，虚拟机将这部分标记记录在线程Remembered Set中，这阶段需要停顿线程，但是可并行执行。

4、筛选回收：首先对各个Region的回收价值和成本进行排序，根据用户所期待的GC停顿时间来制定回收计划，这个阶段也可以与用户线程并行执行，但由于只回收一部分的Region,时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率

CMS和G1的区别：

1、并行于并发：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。
2、分代收集：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。它能够采用不同的方式去处理新创建的对象和已经存活了一段时间，熬过多次GC的旧对象以获取更好的收集效果。
3、空间整合：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
4、可预测的停顿：这是G1相对于CMS的另一个大优势，降低停顿时间是G1和ＣＭＳ共同的关注点，但Ｇ１除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，

Minor GC与Full GC：

Minor GC:
Eden区域满了，或者新创建的对象大小 > Eden所剩空间，每次 Minor GC 会清理年轻代的内存
Major GC 是清理老年代。
Full GC 是清理整个堆空间—包括年轻代和老年代。
2. Full GC（老年代GC）的触发条件
（1）直接调用System.gc
（2）老年代空间不足（新生代存活下来的对象转入、大对象的创建等引起）
调优策略：
尽量做到让对象在Minor GC阶段被回收
让对象在新生代多存活一段时间

不要创建过大的对象及数组
（3）方法区空间不足（系统中要加载的类、反射的类和调用的方法较多等导致）
调优策略：
增大方法区空间
转为使用CMS GC
（4）Minor GC 时，survivor放不下，对象只能放入老年代，而此时老年代也放不下
调优策略：
增大survivor space、老年代空间
（5）通过Minor GC后进入老年代的平均大小大于老年代的连续可用内存
（Minor GC 时会做一个判断，统计之前晋升到老年代的对象的平均大小）

类加载器：

1）Bootstrap ClassLoader 根类加载器
负责加载$JAVA_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类
2）Extension ClassLoader。扩展类加载器
负责加载java平台中扩展功能的一些jar包，包括$JAVA_HOME中jre/lib/*.jar或-Djava.ext.dirs指定目录下的jar包
3）App ClassLoader 系统类加载器
负责记载classpath中指定的jar包及目录中class
4）Custom ClassLoader
属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

双亲委派模型

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。
使用双亲委派模型的好处在于Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类java.lang.Object，它存在在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的Bootstrap ClassLoader进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，

如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，那系统中将会出现多个不同的Object类，程序将混乱。因此，如果开发者尝试编写一个与rt.jar类库中重名的Java类，可以正常编译，但是永远无法被加载运行。

破坏双亲委派模型

https://www.jianshu.com/p/166c5360a40b

Java类加载过程

加载，链接 初始化
链接又包括准备，验证，准备，解析
加载是文件到内存的过程，通过类的完全限定名查找此类字节码文件，并利用字节码文件创建一个class对象

验证是对类文件内容的验证，目的在于确保class文件符合当前虚拟机的要求，不会危害到虚拟机大安全，主要包括四种：文件格式验证、原数据验证、字节码验证、符号引用验证

准备阶段是进行内存分配，为static修饰的变量进行分配内存并设置初始值，这里要注意初始值是0或者null而不是代码中设置的具体值，代码中设置的值在初始化阶段完成，另外这里也不包含final修饰的静态变量，因为final变量在编译时就已经分配了

解析主要是将常量池中的符号引用替换为直接引用的过程，直接引用就是直接指向目标的指针或者相对偏移量等

初始化主要完成静态块和静态变量的赋值，这是类加载的最后阶段，若被加载类的父类没有初始化，则先对父类进行初始化，只有对类的主动使用时才会初始化。初始化的出发条件包括，创建类的实例的时候、访问类的静态方法或者静态变量的时候、使用classForName反射类的时候、某个子类被初始化的时候

MySQL

事务四大特性

原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
一致性：在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。
隔离性：隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆， 必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。
持久性：在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

事务隔离级别：

事务的并发问题：
1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据
2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。
3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是

串行化（serializable）	否	否	否

Mysql默认是可重复读

MySQL存储引擎

InnoDB：支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高（比如银行），要求实现并发控制（比如售票），那选择InnoDB有很大的优势。如果需要频繁的更新、删除操作的数据库，也可以选择InnoDB，因为支持事务的提交（commit）和回滚（rollback）。
MyISAM：插入数据快，空间和内存使用比较低。如果表主要是用于插入新记录和读出记录，那么选择MyISAM能实现处理高效率。如果应用的完整性、并发性要求比 较低，也可以使用。
MEMORY：所有的数据都在内存中，数据的处理速度快，但是安全性不高。如果需要很快的读写速度，对数据的安全性要求较低，可以选择MEMOEY。它对表的大小有要求，不能建立太大的表。所以，这类数据库只使用在相对较小的数据库表。

MyISAM和INNODB的区别

事务：
MyISAM：强调的是性能，每次查询具有原子性,其执行数度比InnoDB类型更快，但是不提供事务支持。
InnoDB：提供事务支持事务，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。
表锁差异：
MyISAM：只支持表级锁，用户在操作myisam表时，select，update，delete，insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。
InnoDB：支持事务和行级锁，是innodb的最大特色。行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。
主键：
MyISAM：允许没有任何索引和主键的表存在，索引都是保存行的地址。
InnoDB：如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。
适用场景：

MyISAM：如果执行大量的SELECT，MyISAM是更好的选择。
InnoDB：如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。DELETE 从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。

查询语句不同元素执行顺序

SELECT select_expr [,select_expr,…] [
FROM tb_name
[JOIN 表名]
[ON 连接条件]
[WHERE 条件判断]
[GROUP BY {col_name | postion} [ASC | DESC], …]
[HAVING WHERE 条件判断]
[ORDER BY {col_name|expr|postion} [ASC | DESC], …]
[LIMIT {[offset,]rowcount | row_count OFFSET offset}]
]

索引为什么用B+树，B+树和B-树的区别

Memory：
存储引擎使用的是Hash索引
Hash索引无法用于范围查询，不能排序，不支持多列联合索引的最左匹配规则，如果有大量重复键值的情况下效率很低，因为有hash碰撞
为什么用B+树?
B+树是一种多路搜索树，每个节点上可以拥有多个子节点，由于数据库的索引是存储在磁盘上的，数据量大的情况下无法一次性把所有索引加载到内存中，B+树的设计可以允许数据分批加载，减少磁盘的寻址加载次数，同时树的高度降低，提高查询效率。并且B+树的叶子节点之间有链表相连，范围查询时速度更快。
B+树和B-树的区别：

1. B+树的非叶子节点上不存储数据，仅存储键值，而B树的非叶子节点上既有键值也有数据。之所以这么做是因为在数据库中页的大小是固定的，不存储数据的情况下可以存储更多的键值，使树的高度进一步降低，加快查询速度，B+树一个节点可以存储1000个键值，3层B+树可以存储10亿个键值，跟节点常驻内存，所以查询只用两次磁盘IO
2. B+树的叶子节点之间有指针相连，这样进行范围查找，分组查找的时候更加简单，只用找到头节点和尾节点就可以查询出所有的数据

聚簇索引和非聚簇索引：

B+树的叶子节点存储的是整行数据，也可能是主键的值，存储整行数据时为聚簇索引，存储主键的值是非聚簇索引。通过非jucu索引查询数据时需要通过两次查询，先查出主键之后再查数据，这个过程叫回表。但是不一定所有的非聚簇索引都要查询两次，通过索引覆盖也可以只查询一次，如果需要查询的列已经在索引中，就无需回表。

MySQL的锁

表锁：
MySQL表级锁有两种模式：表共享锁（Table Read Lock）和表独占写锁（Table Write Lock）。
对**MyISAM**的读操作，不会阻塞其他用户对同一表请求，但会阻塞对同一表的写请求；
对MyISAM的写操作，则会阻塞其他用户对同一表的读和写操作；
MyISAM表的读操作和写操作之间，以及写操作之间是串行的。
当一个线程获得对一个表的写锁后，只有持有锁线程可以对表进行更新操作。其他线程的读、写操作都会等待，直到锁被释放为止。

行锁：
InnoDb实现了行锁：
共享锁（S）：SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE
排他锁（X）：SELECT * FROM table_name WHERE ... FOR UPDATE

行锁实现方式：
InnoDB行锁实现方式
　InnoDB行锁是通过索引上的索引项来实现的，这一点MySQL与Oracle不同，后者是通过在数据中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味者：只有通过索引条件检索数据，InnoDB才会使用行级锁，否则，InnoDB将使用表锁！
　　在实际应用中，要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。

间隙锁：
一个范围条件的检索，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101（这些记录并不存在）的“间隙”加锁。防止幻读

解除mysql死锁的两种方法：

（1）终止（或撤销）进程。终止（或撤销）系统中的一个或多个死锁进程，直至打破循环环路，使系统从死锁状态中解除出来。

（2）抢占资源。从一个或多个进程中抢占足够数量的资源，分配给死锁进程，以打破死锁状态。

Mysql索引最左前缀匹配原则

创建联合索引时将识别度最高，使用最频繁的的字段放到最前面
因为Mysql索引查询时会遵循最左前缀匹配的原则，即最左优先，在检索数据时从联合索引的最左边开始匹配，所以当我们创建一个联合索引1，2，3的时候相当于创建了1，12，123三个索引，这就是最左前缀匹配原则

Mysql 版本优化

线上使用的Mysql版本：比较早的有5.5，新建的5.6

Mysql在5.6版本的优化：

5.6中索引下推：

多列索引的时候如果索引列有like ‘%sdad5%‘这种情况如果没有索引下推的话会直接先查询出对应的数据返回给服务端，服务端基于数据是否满足like来判断数据是否符合条件
有索引下推的时候mysql会判断索引是否满足like，如果不满足直接拒绝掉，有了索引下推优化可以在有like条件查询的时候减少回表次数

水平分表/垂直分表

1，水平分表： 一条记录一条记录切断分出来！
2, 垂直分表： 把常用的 ，不常用的，字段很长的拆出来！

Mysql临时表

CREATE TEMPORARY TABLE，主要用于对大数据的表做一个子集，提高查询效率，MySQL 临时表在我们需要保存一些临时数据时是非常有用的。临时表只在当前连接可见，当关闭连接时，Mysql会自动删除表并释放所有空间。

查询时索引的选取

和Mysql的查询优化器有关，mysql查询优化器会在I 条sql具体执行之前找出所有可能实现的方案，对比之后找出成本最低的方案，这个成本最低的方案就是执行计划。 首先找出所有可能使用的索引，之后计算全表扫描和不同索引执行查询的代价，对比各种执行方案的代价，找出成本最低的一个

Mysql如何优化查询?

非关系型数据库和关系型数据库

非关系型数据库的优势：MongoDB。Redis CouchDB
1. 性能NOSQL是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过SQL层的解析，所以性能非常高。
2. 可扩展性同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。
关系型数据库的优势：mysql Oracle SQL server
1. 复杂查询可以用SQL语句方便的在一个表以及多个表之间做非常复杂的数据查询。
2. 事务支持使得对于安全性能很高的数据访问要求得以实现。

数据库三范式

1. 第一范式(确保每列保持原子性)
第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到"地址"这个属性，本来直接将"地址"属性设计成一个数据库表的字段就行。但是如果系统经常会访问"地址"属性中的"城市"部分，那么就非要将"地址"这个属性重新拆分为省份、城市、详细地址等多

个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式

2. **第二范式(确保表中的每列都和主键相关)**
把商品信息分离到另一个表中，把订单项目表也分离到另一个表中
3. **第三范式(确保每列都和主键列直接相关,而不是间接相关)**
比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。如下面这两个表所示的设计就是一个满足第三范式的数据库表

数据库读写分离/主从复制/主从复制分析

主从复制可以使MySQL数据库主服务器的主数据库，复制到一个或多个MySQL从服务器从数据库，默认情况下，复制异步; 根据配置，可以复制数据库中的所有数据库，选定的数据库或甚至选定的表。
在多个从库之间扩展负载以提高性能。在这种环境中，所有写入和更新在主库上进行。但是，读取可能发生在一个或多个从库上。该模型可以提高写入的性能（由于主库专用于更新），同时在多个从库上读取，可以大大提高读取速度。
MySQL 主从复制主要有以下几种方式：
基于 SQL 语句的复制(statement-based replication, SBR)；
基于行的复制(row-based replication, RBR)；
混合模式复制(mixed-based replication, MBR)；
基于 SQL 语句的方式最古老的方式，也是目前默认的复制方式，后来的两种是 MySQL 5 以后才出现的复制方式。

Mysql慢查询怎么解决

Explain分析慢查询的原因：
1. 没有索引或者没有用到索引
对经常查询的列建立索引或者联合索引
2. 查询数据量太大
减少每次查询的数据量
3. 内存不足
增大内存
4. 减少表的尺寸，水平分表或者垂直分表
5. 优化查询语句

内连接/外连接/交叉连接/笛卡尔积

Inner join：内连接指的是把表连接时表与表之间匹配的数据行查询出来，就是两张表之间数据行匹配时，要同时满足ON语句后面的条件才行。
外连接：left join和right join：左连接的意思是，无论是否符合ON语句后面的表连接条件都会把左边那张表的记录全部查询出来，右边的那张表只匹配符合条件的数据行。
交叉连接CROSS JOIN：交叉连接返回的结果，是被连接的两个表中所有数据行的笛卡尔积，也就是返回第一个表中符合查询条件的数据行数，乘以第二个表中符合查询条件的数据行数

Char和varchar

char是固定长度的，而varchar会根据具体的长度来使用存储空间，另外varchar需要用额外的1-2个字节存储字符串长度。
1). 当字符串长度小于255时，用额外的1个字节来记录长度
2). 当字符串长度大于255时，用额外的2个字节来记录长度
比如char(255)和varchar(255)，在存储字符串"hello world"时，char会用一块255个字节的空间放那个11个字符；而varchar就不会用255个，它先计算字符串长度为11，然后再加上一个记录字符串长度的字节，一共用12个字节存储，这样varchar在存储不确定长度的字符串时会大大减少存储空间。

1、char。char存储定长数据很方便，char字段上的索引效率级高，比如定义char(10)，那么不论你存储的数据是否达到了10个字节，都要占去10个字节的空间。

2、varchar。存储变长数据，但存储效率没有char高。如果一个字段可能的值是不固定长度的，我们只知道它不可能超过10个字符，把它定义为 varchar(10)是最合算的。varchar类型的实际长度是它的值的实际长度+1。为什么“+1”呢?这一个字节用于保存实际使用了多大的长度。
从空间上考虑，用varchar合适；从效率上考虑，用char合适，关键是根据实际情况找到权衡点。

3、text。text存储可变长度的非Unicode数据，最大长度为2^31-1(2,147,483,647)个字符。

4、nchar、nvarchar、ntext。这三种从名字上看比前面三种多了个“N”。它表示存储的是Unicode数据类型的字符。我们知道字符中，英文字符只需要一个字节存储就足够了，但汉字众多，需要两个字节存储，英文与汉字同时存在时容易造成混乱，Unicode字符集就是为了解决字符集这种不兼容的问题而产生的，它所有的字符都用两个字节表示，即英文字符也是用两个字节表示。nchar、nvarchar的长度是在1到4000之间。和char、varchar比较起来，nchar、nvarchar则最多存储4000个字符，不论是英文还是汉字；而char、varchar最多能存储8000个英文，4000个汉字。可以看出使用nchar、nvarchar数据类型时不用担心输入的字符是英文还是汉字，较为方便，但在存储英文时数量上有些损失。

所以一般来说，如果含有中文字符，用nchar/nvarchar，如果纯英文和数字，用char/varchar。

Mysql高并发环境解决方案

1）代码中sql语句优化
2）数据库字段优化，索引优化，索引要加但是不能乱加
3）加缓存，redis/memcache等
4）主从，读写分离
5）分区表
6）垂直拆分，解耦模块
7）水平切分

点评：
1、1&2是最简单，也是提升效率最快的方式。也许有人说这两点你已经做的很好了，你的每条语句都命中了索引，是最高效的。但是你是否是为了你的sql达到最优而去建索引，而不是从整个业务上来考虑。比如，订单表上我需要增加xx索引满足某单一业务，是否就一定要加，其他方法能否解决。如果要满足所有业务的需求，那么索引就泛滥了，对于千万级以上的表来说，维护索引的成本大大增加，反而增加了数据库的内存的开销。
2、数据库字段的优化。曾经发现一高级程序员在表字段的设计上，一个日期类型，被设计为varchar类型，不规范的同时，无法对写入数据校验，做索引的效率也有差别(网(xian)友(pen)的(liao)观(zai)点(shuo)，具体差别原理不详)。
3、缓存适合读多写少更新频度相对较低的业务场景，否则缓存异议不大，命中率不高。缓存通常来说主要为了提高接口处理速度，降低并发带来的db压力以及由此产生的其他问题。你的接口时延多少？有没有被用户吐槽？有没有必要提升？好吧，我们的前台后台商家并发量太低，当我没说。

4、分区不是分表，结果还是一张表，只不过把存放的数据文件分词了多个小块，分块后。在表数据非常大的情况下，可以解决无法一次载入内存，以及大表数据维护等问题。
5、垂直拆分将表按列拆成多表，常见于将主表的扩展数据独立开，文本数据独立开，降低磁盘io的压力。
6、水平拆，这是一把最有效的牛刀。但是存在一个误区，有的人会觉得，为什么不在最开始就直接水平线拆，免去了后面迁移数据的麻烦。我个人感觉是，下定某个决策之前，必须有一个非常充分的理由。水平拆分的主要目的是提升单表并发读写能力(压力分散到各个分表中)和磁盘IO性能(一个非常大的.MYD文件分摊到各个小表的.MYD文件中)。如果没有千万级以上数据，为什么要拆，仅对单表做做优化也是可以的；再如果没有太大的并发量，分区表也一般能够满足。所以，一般情况下，水平拆分是最后的选择，在设计时还是需要一步一步走。

数据库崩溃事务恢复机制

innodb事务日志包括redo log和undo log。redo log是重做日志，提供前滚操作，undo log是回滚日志，提供回滚操作。
undo log不是redo log的逆向过程，其实它们都算是用来恢复的日志：
1.redo log通常是物理日志，记录的是数据页的物理修改，而不是某一行或某几行修改成怎样怎样，它用来恢复提交后的物理数据页(恢复数据页，且只能恢复到最后一次提交的位置)。
2.undo用来回滚行记录到某个版本。undo log一般是逻辑日志，根据每行记录进行记录。

PreparedStatement防止SQL注入

因为sql语句是预编译的，而且使用了占位符，规定了sql语句的结构。用户可以设置“?”的值，但是不能改变sql语句的结构，所以想在sql语句后面加上"or 1=1"实现注入是行不通的。PreparedStatement不仅能防止sql注入，由于PreparedStatement是预编译的，不用改变一次参数就重新编译整个sql语句，此外它执行查询语句的结果集是离线的，链接关闭后，仍然可以访问结果集。

Mybatis

三种DataSource

UNPOOLED，POOLED，JNDI

SQL执行流程

在执行Sql时首先会从SqlSessionFactory中创建一个新的SqlSession，Sql语句是通过SqlSession中的Executor来执行的，Executor根据SqlSession传递的参数，执行Query方法，然后创建StatementHandler对象，将必要的参数传递给StatementHandler，由StatementHandler完成对数据库的查询，StatementHandler调用ParameterHandler的setParameters方法，把用户传递的参数转换成jdbcStatement所需要的参数，调用原生的JDBC 来执行语句，最后由ResultSetHandler的handlerResultSet方法对JDBC返回的ResultSet结果集转换成对象集，并逐级返回结果，完成一次sql语句的执行，

缓存

一级缓存：
作用域是session
HashMap实现
默认开启
二级缓存：
作用域是Mapper（namespace）
支持ehcache等缓存实现
可配置剔除策略，刷新间隔，缓存数量等

Mapper实例化流程

service层调用dao数据层方法-->读取对应接口dao的MapperFactoryBean#getObject()方法-->SqlSessionTemplate#getMapper()方法-->mapperRegistry.getMapper->从knowMappers中->得到MapperProxy代理类并调用对应接口方法的MapperMethod.execute->SqlSessionInterceptor.invoke方法执行sql->DefaultSqlSession的CRUD操作
-->以接口类+method方法名作为mappedStatementId读取MappedStatement对象以获取SqlCommand指令-->根据SqlCommand指令调用SqlSessionTemplate对应的CRUD操作，一般为select()/delete()/update()/insert()方法
-->sqlSessionProxy代理调用真实数据层处理类DefaultSqlSession对应的CRUD操作-->选用池来引用MappedStatement对象处理数据库sql语句
-->返回结果集供MapperMethod处理返回给service层

Spring相关

Spring是什么

Spring是一个轻量级的IoC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于XML的配置、基于注解的配置、基于Java的配置。
主要由以下几个模块组成：
Spring Core：核心类库，提供IOC服务；
Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；
Spring AOP：AOP服务；
Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；
Spring ORM：对现有的ORM框架的支持；
Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；

Spring MVC：提供面向Web应用的Model-View-Controller实现。
Spring WebFlux：流式处理，替代spring mvc

Spring两大特性：IOC/AOP实现原理

IOC：控制反转，它并不是一种技术实现，而是一种设计思想，许多应用都是通过彼此间的相互合作来实现业务逻辑的，如类A要调用类B的方法，以前我们都是在类A中，通过自身new一个类B，然后在调用类B的方法，现在我们把new类B的事情交给spring来做，在我们调用的时候，容器会为我们实例化。
Spring的IOC有三种注入方式： 构造器注入、setter方法注入、根据注解注入。
IOC容器的初始化过程：
AOP：面向切面编程：面向切面编程，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。
（1）切面（Aspect）： 被抽取的公共模块，可能会横切多个对象。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。
（2）连接点（Join point）： 指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。
（3）通知（Advice）： 在切面的某个特定的连接点（Join point）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。许多AOP框架，包括Spring，都是以拦截器做通知模型， 并维护一个以连接点为中心的拦截器链。
（4）切入点（Pointcut）： 切入点是指 我们要对哪些Join point进行拦截的定义。通过切入点表达式，指定拦截的方法，比如指定拦截add*、search*。
（5）引入（Introduction）： （也被称为内部类型声明（inter-type declaration））。声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。
（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做 被通知（adviced） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。
（7）织入（Weaving）： 指把增强应用到目标对象来创建新的代理对象的过程。Spring是在运行时完成织入。

Spring循环依赖

Spring通过三级缓存加上“提前曝光”机制，配合Java的对象引用原理，比较完美地解决了某些情况下的循环依赖问题！Spring单例对象初始化主要分为三部：
（1）createBeanInstance：实例化，其实也就是调用对象的构造方法实例化对象
（2）populateBean：填充属性，这一步主要是多bean的依赖属性进行填充
（3）initializeBean：调用spring xml中的init 方法。

Spring中循环依赖场景有：
（1）构造器的循环依赖
（2）field属性的循环依赖。
Spring为了解决单例的循环依赖问题，使用了三级缓存
这三级缓存分别指：
singletonFactories： 单例对象工厂的cache
earlySingletonObjects： 提前曝光的单例对象的Cache
singletonObjects：单例对象的cache

Spring的getSingleton过程：
首先从一级缓存singletonObjects中获取，如果获取不到并且对象正在创建，就从二级缓存earlySingletonObjects中获取，如果还是获取不到且允许singletonFactories通过getObject()获取，就从三级缓存singletonFactory.getObject()(三级缓存)获取，如果获取到了则从singletonFactories中移除，并放入earlySingletonObjects中。其实也就是从三级缓存移动到了二级缓存。

A首先完成了初始化的第一步，并且将自己提前曝光到singletonFactories中，此时进行初始化的第二步，发现自己依赖对象B，此时就尝试去get(B)，发现B还没有被create，所以走create流程，B在初始化第一步的时候发现自己依赖了对象A，于是尝试get(A)，尝试一级缓存singletonObjects(肯定没有，因为A还没初始化完全)，尝试二级缓存earlySingletonObjects（也没有），尝试三级缓存singletonFactories，由于A通过ObjectFactory将自己提前曝光了，所以B能够通过ObjectFactory.getObject拿到A对象(虽然A还没有初始化完全，但是总比没有好呀)，B拿到A对象后顺利完成了初始化阶段1、2、3，完全初始化之后将自己放入到一级缓存singletonObjects中。此时返回A中，A此时能拿到B的对象顺利完成自己的初始化阶段2、3，最终A也完成了初始化，进去了一级缓存singletonObjects中，而且更加幸运的是，由于B拿到了A的对象引用，所以B现在hold住的A对象完成了初始化。

Spring不能解决“A的构造方法中依赖了B的实例对象，同时B的构造方法中依赖了A的实例对象”这类问题了！因为加入singletonFactories三级缓存的前提是执行了构造器，所以构造器的循环依赖没法解决。

BeanFactory和ApplicationContext有什么区别

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：
①继承MessageSource，因此支持国际化。
②统一的资源文件访问方式。
③提供在监听器中注册bean的事件。
④同时加载多个配置文件。
⑤载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。
（2）①BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

②ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。
③相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。
（3）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。
（4）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

Bean的生命周期

1.实例化InstantiationAwareBeanPostProcessorAdapter实现类
2.调用InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation方法　3.执行Bean的构造方法
4.调用InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation方法
5.执行InstantiationAwareBeanPostProcessor的postProcessPropertyValues方法
6.给bean注入属性
7.处理BeanNameAware、BeanClassLoaderAware、BeanFactoryAware
8.执行BeanPostProcessor的postProcessBeforeInitialization方法
9.调用InitializingBean的afterPropertiesSet方法
10.调用<init-method>指定的初始化方法。
11.调用BeanPostProcessor的postProcessAfterInitialization
容器初始化成功，放入缓存中
12.调用DiposibleBean的destory方法
13.调用<bean>的destory-method属性指定的销毁方法

BeanDefinition的属性有哪些

类名、scope、属性、构造函数参数列表、依赖的bean、是否是单例类、是否是懒加载，其实就是将Bean的定义信息存储到这个BeanDefinition相应的属性中，后面对Bean的操作就直接对BeanDefinition进行，例如拿到这个BeanDefinition后，可以根据里面的类名、构造函数、构造函数参数，使用反射进行对象创建。

Spring支持的几种bean的作用域BeanDefinition属性中的Scope

Spring容器中的bean可以分为5个范围：
（1）singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。
（2）prototype：为每一个bean请求提供一个实例。

（3）request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。
（4）session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
（5）global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

Servlet生命周期

实例化，初始init，接收请求service，销毁destroy

Spring中的设计模式

（1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；
（2）单例模式：Bean默认为单例模式。
（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；
（4）模板方法：用来解决代码重复的问题。比
如. RestTemplate, JmsTemplate, JpaTemplate。
（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。

Spring隔离级别：

① ISOLATION_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。
② ISOLATION_READ_UNCOMMITTED：读未提交，允许另外一个事务可以看到这个事务未提交的数据。
③ ISOLATION_READ_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。
④ ISOLATION_REPEATABLE_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。

⑤ ISOLATION_SERIALIZABLE：一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

Spring事务的实现方式和实现原理

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过undo log或者redo log实现的。

（1）Spring事务的种类：
spring支持编程式事务管理和声明式事务管理两种方式：
①编程式事务管理使用TransactionTemplate。
②声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。
声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。
声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

Spring MVC的工作原理：

Spring MVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把Model，View，Controller分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

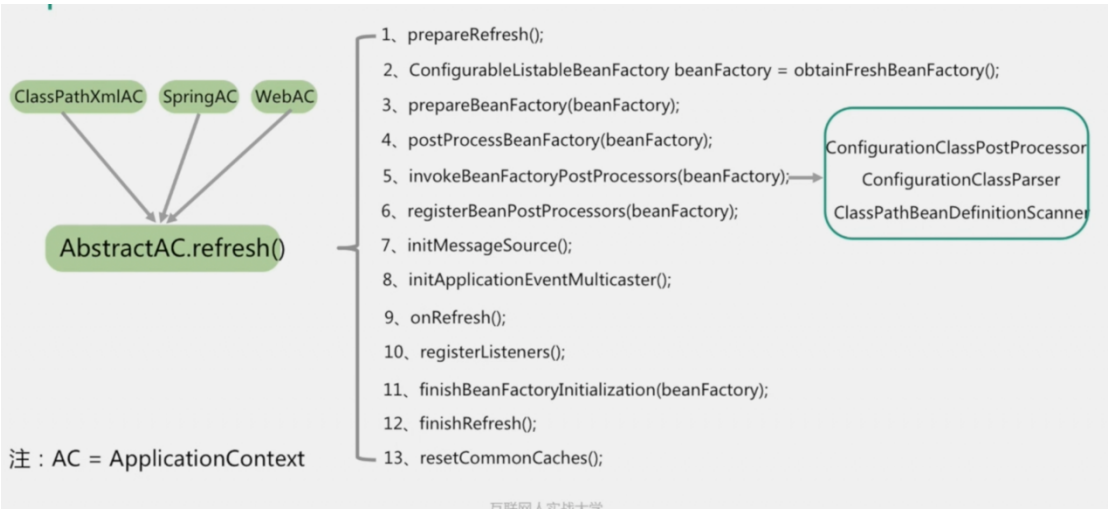
（1）用户发送请求至前端控制器DispatcherServlet；
（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；
（3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet；
（4）DispatcherServlet 调用 HandlerAdapter处理器适配器；
（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；
（6）Handler执行完成返回ModelAndView；

（7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；
（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；
（9）ViewResolver解析后返回具体View；
（10）DispatcherServlet对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。

Spring中的自动装配方式

no：不进行自动装配，手动设置Bean的依赖关系。
byName：根据Bean的名字进行自动装配。
byType：根据Bean的类型进行自动装配。
constructor：类似于byType，不过是应用于构造器的参数，如果正好有一个Bean与构造器的参数类型相同则可以自动装配，否则会导致错误。
autodetect：如果有默认的构造器，则通过constructor的方式进行自动装配，否则使用byType的方式进行自动装配。

Spring Context初始化流程



AbstractApplicationContext:refresh()方法
1.prepareRefresh：刷新准备工作，主要是设置refresh操作的起始时间、active=true、并且初始化属性资源的校验
2.obtainFreshBeanFactory：解析spring配置文件并封装为BeanDefinition对象保存至beanFactory中,该方法主要完成创建Bean工厂,用子类

AbstractRefreshableApplicationContext.refreshBeanFactory完成，如果已经存在bean工厂则会销毁，创建的BeanFactory默认为DefaultListableBeanFactory
XmlWebApplicationContext.loadBeanDefinitions()->XmlBeanDefinitionReader#registerBeanDefinitions()-->DefaultBeanDefinitionDocumentReader.registerBeanDefinitions()-->DefaultBeanDefinitionDocumentReader.parseBeanDefinitions()
********DefaultBeanDefinitionDocumentReader.parseBeanDefinitions-——》
正如官方注释所说，解析import标签、alias标签、bean标签和自定义的标签分为两部分：DefaultBeanDefinitionDocumentReader#parseDefaultElement和BeanDefinitionParserDelegate#parseCustomElement
DefaultBeanDefinitionDocumentReader#parseDefaultElement：解析import标签、alias标签、bean标签，三种具体的解析方法
所有的bean信息都由BeanDefinitionHolder对象来保存，其中的BeanDefinition包含了一个bean标签的所有可能内容
BeanDefinitionParserDelegate#parseCustomElement： 调用NamespaceHandler.parse接口去解析，ComponentScanBeanDefinitionParser.parse方法。https://cloud.tencent.com/developer/article/1123722
注册bean的方法：DefaultListableBeanFactory.registerBeanDefinition()和registerSingleton()
3.prepareBeanFactory：beanFactory的准备工作，设置context的属性配置，配置beanFactory，此处指的是DefaultListableFactory，配置其标准的性质，比如上下文的加载器ClassLoader和post-processors回调
4.postProcessBeanFactory:主要添加ServletContextAwareProcessor处理类，对应的父类AbstractRefreshableWebApplicationContext#postProcessBeanFactory
5.invokeBeanFactoryPostProcessors:执行BeanDefinitionRegistryPostProcessors/BeanFactoryPostProcessors相关beans 实例化和调用所有已注册的BeanFactoryPostProcessors beans，通过一个委托类来处理实例化调用,主要功能是对实现BeanFactoryPostProcessor的bean类进行调用公共接口方法postProcessBeanFactory,并相关的信息可关联至ConfigurableListableBeanFactorybeanFactory。常见的使用类为PropertyPlaceholderConfigurer文件解析类、MapperScannerConfigurer SQL接口注册类。公共接口的调用前者会对每个bean对象含有${}进行解析替换，后者会注册mapper class接口类并尝试解析注解
6.registerBeanPostProcessors:注册所有实现BeanPostProcessor的接口bean到beanFactory的内部属性beanPostProcessors集合中,实例化并且调用所有的已注册的BeanPostProcessor beans，其实也就是简单的把DefaultListableFactory中的所有为BeanPostProcessor的实现bean类放置于其内部属性beanPostProcessors List集合中，但是并没有执行其中的相应公共方法

7.initMessageSource:初始化资源配置
8.initApplicationEventMulticaster:初始化ApplictionEventMulticaster广播事件类，默认使用SimpleApplicationEventMulticaster事件
9.onRefresh:初始化themeSource主题源，默认使用ResourceBundleThemeSource
10.registerListeners:注册ApplicationListener beans到ApplictionEventMulticaster广播集合
11.finishBeanFactoryInitialization:实例化所有的非lazy-init类型的beans
具体实现为：DefaultListableBeanFactory#preInstantiateSingletons()， //非抽象、单例模式、非lay-init，满足以上条件的进入到实例化(默认情况下Bean为singleton单例模式) 使用getBean方法进行相应beanName的实例化
12.finishRefresh:完成刷新，并执行ContextRefreshedEvent事件，该事件涉及spring mvc

DispatcherServlet初始化过程

DispatcherServlet初始化指的是init()生命周期方法被执行，在父类HttpServletBean中定义
1. 解析web.xml定义中<servlet>元素的子元素<init-param>中的参数值。
2.之后initServletBean方法初始化应用上下文webApplicationContext
2. 调用onRefresh方法刷新应用上下文
该方法调用initStrategies方法实例化`MultipartResolver`、`LocaleResolver`、`HandlerMapping`、`HandlerAdapter`和`ViewResolver`等组件。如异常处理，视图处理，请求映射

Spring的getBean流程

1. AbstractBeanFactory.getBean
2. AbstractBeanFactory.doGetBean
3. DefaultSingletonBeanRegistry.getSingleton
4. 没有的话调用AbstractAutowireCapableBeanFactory.createBean
5. AbstractAutowireCapableBeanFactory.doCreateBean
这个方法中解决循环依赖getSingleton
6. AbstractAutowireCapableBeanFactory.createBeanInstance
7. AbstractAutowireCapableBeanFactory.instantiateBean

Spring如何保证Controller的并发安全

1、在Controller中使用ThreadLocal变量
2、在spring配置文件Controller中声明 scope="prototype"，每次都创建新的controller
所在在使用spring开发web 时要注意，默认Controller、Dao、Service都是单例的。

Spring Boot和Spring相比做了什么改变

Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServelet，MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。
Spring MVC和Spring Boot都属于Spring，Spring MVC 是基于Spring的一个 MVC 框架，而Spring Boot 是基于Spring的一套快速开发整合包
Spring Boot 优点非常多，如：
独立运行
简化配置
自动配置
无代码生成和XML配置
应用监控
上手容易

Spring Boot核心注解

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

开启 Spring Boot 特性有哪几种方式?

1）继承spring-boot-starter-parent项目

2）导入spring-boot-dependencies项目依赖

运行 Spring Boot 有哪几种方式?

1）打包用命令或者放到容器中运行

2）用 Maven/ Gradle 插件运行

3）直接执行 main 方法运行

Spring Boot 自动配置原理是什么?

Spring Boot启动的时候会通过@EnableAutoConfiguration注解找到META-INF/spring.factories配置文件中的所有自动配置类，并对其进行加载，而这些自动配置类都是以AutoConfiguration结尾来命名的，它实际上就是一个JavaConfig形式的Spring容器配置类，它能通过以Properties结尾命名的类中取得在全局配置文件中配置的属性如：server.port，而XxxxProperties类是通过@ConfigurationProperties注解与全局配置文件中对应的属性进行绑定的。

Spring boot常用starter

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包

spring-boot-starter-web嵌入tomcat和webkaifa需要servlet与jsp支持

spring-boot-starter-data-jpa数据库支持

spring-boot-starter-data-redis数据库支持

spring-boot-starter-data-solr solr支持

mybatis-spring-boot-starter第三方的mybatis集成starter

Spring boot自动装配的原理

在spring程序main方法中添加@SpringBootApplication或者@EnableAutoConfiguration

在初始化容器时会自动去maven中读取每个starter中的spring.factories文件，该文件配置了所有需要被创建spring容器中的bean

如何在 Spring Boot 启动的时候运行一些特定的代码?

可以实现接口 ApplicationRunner 或者 CommandLineRunner，这两个接口实现方式一样，它们都只提供了一个 run 方法
如果启动的时候有多个ApplicationRunner和CommandLineRunner，想控制它们的启动顺序，可以实现 org.springframework.core.Ordered接口或者使用 org.springframework.core.annotation.Order注解。

Spring Boot 2.X 有什么新特性？与 1.X 有什么区别?

配置变更
JDK 版本升级，2必须要求jdk8
第三方类库升级，spring framework5+ tomcat 8.5+
响应式 Spring 编程支持，WebFlux，响应式编程是完全异步和非阻塞的，它是基于事件驱动模型，而不是传统的线程模型
HTTP/2 支持
配置属性绑定
更多改进与加强...

Spring Boot 有哪几种读取配置的方式?

Spring Boot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationProperties 来绑定变量

保护 Spring Boot 应用有哪些方法?

在生产中使用HTTPS
使用Snyk检查你的依赖关系
升级到最新版本
启用CSRF保护
使用内容安全策略防止XSS攻击

Zookeeper

Zookeeper是一个开源的分布式协调服务，是一个典型的分布式数据一致性解决方案，分布式应用程序可以基于Zookeeper实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper的特点

顺序一致性：从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。
原子性：所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。
单一系统映像：无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。
可靠性：一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖。

Zookeeper的角色

角色		描述
领导者（Leader）		领导者负责进行投票的发起和决议，更新系统状态
学习者（Learner）	跟随者（Follower）	Follower 用于接收客户请求并向客户端返回结果，在选主过程中参与投票
	观察者（ObServer）	ObServer 可以接收客户端连接，将写请求转发给 leader 节点。但 ObServer 不参加投票过程，只同步 leader 的状态。ObServer 的目的是为了扩展系统，提高读取速度
客户端（Client）		请求发起方

ZooKeeper & ZAB 协议 & Paxos 算法

ZAB（ZooKeeper Atomic Broadcast 原子广播）协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。

Paxos

一种基于消息传递的分布式一致性算法
1.获取一个Proposal ID n，为了保证Proposal ID唯一，可采用时间戳+Server ID生成；
2.Proposer向所有Acceptors广播Prepare(n)请求；
3.Acceptor比较n和minProposal，如果n>minProposal，minProposal=n，并且将 acceptedProposal 和 acceptedValue 返回；
4.Proposer接收到过半数回复后，如果发现有acceptedValue返回，将所有回复中acceptedProposal最大的acceptedValue作为本次提案的value，否则可以任意决定本次提案的value；
5.到这里可以进入第二阶段，广播Accept (n,value) 到所有节点；
6.Acceptor比较n和minProposal，如果n>=minProposal，则acceptedProposal=minProposal=n，acceptedValue=value，本地持久化后，返回；否则，返回minProposal。
7.提议者接收到过半数请求后，如果发现有返回值result >n，表示有更新的提议，跳转到1；否则value达成一致。

ZAB 协议

所有客户端写入数据都是写入到 主进程（称为 Leader）中，然后，由 Leader 复制到备份进程（称为 Follower）中。从而保证数据一致性。
消息广播和崩溃恢复。整个 Zookeeper 就是在这两个模式之间切换

消息广播

ZAB 协议的消息广播过程使用的是一个原子广播协议，类似一个 二阶段提交过程。对于客户端发送的写请求，全部由 Leader 接收，Leader 将请求封装成一个事务 Proposal，将其发送给所有 Follwer ，然后，根据所有 Follwer 的反馈，如果超过半数成功响应，则执行 commit 操作（先提交自己，再发送 commit 给所有 Follwer）。
基本上，整个广播流程分为 3 步骤：
1、将数据都复制到 Follwer 中
2、等待 Follwer 回应 Ack，最低超过半数即成功
3、当超过半数成功回应，则执行 commit ，同时提交自己

在 Leader 和 Follwer 之间还有一个消息队列，用来解耦他们之间的耦合，解除同步阻塞。
zookeeper集群中为保证任何所有进程能够有序的顺序执行，只能是 Leader 服务器接受写请求，即使是 Follower 服务器接受到客户端的请求，也会转发到 Leader 服务器进行处理。

崩溃恢复

zookeeper集群中为保证任何所有进程能够有序的顺序执行，只能是leader服务器接受写请求，即使是follower服务器接受到客户端的请求，也会转发到leader服务器进行处理。
如果leader服务器发生崩溃，则zab协议要求zookeeper集群进行崩溃恢复和leader服务器选举。
ZAB协议崩溃恢复要求满足如下2个要求：
3.1. 确保已经被leader提交的proposal必须最终被所有的follower服务器提交。
3.2. 确保丢弃已经被leader出的但是没有被提交的proposal。
根据上述要求，新选举出来的leader不能包含未提交的proposal，即新选举的leader必须都是已经提交了的proposal的follower服务器节点。同时，新选举的

leader节点中含有最高的ZXID。这样做的好处就是可以避免了leader服务器检查proposal的提交和丢弃工作。
leader服务器发生崩溃时分为如下场景：
5.1. leader在提出proposal时未提交之前崩溃，则经过崩溃恢复之后，新选举的leader一定不能是刚才的leader。因为这个leader存在未提交的proposal。
5.2 leader在发送commit消息之后，崩溃。即消息已经发送到队列中。经过崩溃恢复之后，参与选举的follower服务器(刚才崩溃的leader有可能已经恢复运行，也属于follower节点范畴)中有的节点已经是消费了队列中所有的commit消息。即该follower节点将会被选举为最新的leader。剩下动作就是数据同步过程。

数据同步

在zookeeper集群中新的leader选举成功之后，leader会将自身的提交的最大proposal的事物ZXID发送给其他的follower节点。follower节点会根据leader的消息进行回退或者是数据同步操作。最终目的要保证集群中所有节点的数据副本保持一致。
数据同步完之后，zookeeper集群如何保证新选举的leader分配的ZXID是全局唯一呢？这个就要从ZXID的设计谈起。
2.1 ZXID是一个长度64位的数字，其中低32位是按照数字递增，即每次客户端发起一个proposal,低32位的数字简单加1。高32位是leader周期的epoch编号，至于这个编号如何产生(我也没有搞明白)，每当选举出一个新的leader时，新的leader就从本地事物日志中取出ZXID,然后解析出高32位的epoch编号，进行加1，再将低32位的全部设置为0。这样就保证了每次新选举的leader后，保证了ZXID的唯一性而且是保证递增的。

Zookeeper实现分布式锁

zk分布式锁，就是某个节点尝试创建临时znode，此时创建成功了就获取了这个锁；这个时候别的客户端来创建锁会失败，只能注册个监听器监听这个锁。

释放锁就是删除这个znode，一旦释放掉就会通知客户端，然后有一个等待着的客户端就可以再次重新加锁。

redis分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能

zk分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小

Redis

Redis的数据结构

简单动态字符串SDS：

sds 的用途
Sds 在 Redis 中的主要作用有以下两个：
实现字符串对象（StringObject）；
在 Redis 程序内部用作 char* 类型的替代品；
Redis 是一个键值对数据库（key-value DB），数据库的值可以是字符串、集合、列表等多种类型的对象，而数据库的键则总是字符串对象。

对于那些包含字符串值的字符串对象来说，每个字符串对象都包含一个 sds 值。

双端链表

Redis 列表使用两种数据结构作为底层实现：有前置节点和后置节点

双端链表
压缩列表

字典

字典在 Redis 中的应用广泛，使用频率可以说和 SDS 以及双端链表不相上下，基本上各个功能模块都有用到字典的地方。
其中，字典的主要用途有以下两个：
实现数据库键空间（key space）；
用作 Hash 类型键的底层实现之一；
Redis 的字典使用哈希表dictht作为底层实现
Dictht中包含一个hash表数组，数组中的每个元素都是一个指向 dict.h/dictEntry 结构的指针， 每个 dictEntry 结构保存着一个键值对。
Size-hash表大小 used—哈希表已有节点的数量 sizemark，等于size-1，用于计算索引值

dictEntry保存一个键值对和指向另一个dictEntry的指针，用于解决hash冲突
字典由dict结构表示
包含两个dictht的数组，rehashidx用于标记rehash目前的进度，如果没有rehash它是-1

渐进式hash：

为 ht[1] 分配空间， 让字典同时持有 ht[0] 和 ht[1] 两个哈希表。
在字典中维持一个索引计数器变量 rehashidx ， 并将它的值设置为 0 ， 表示 rehash 工作正式开始。
在 rehash 进行期间， 每次对字典执行添加、删除、查找或者更新操作时， 程序除了执行指定的操作以外， 还会顺带将 ht[0] 哈希表在 rehashidx 索引上的所有键值对 rehash 到 ht[1] ， 当 rehash 工作完成之后， 程序将 rehashidx 属性的值增一。
随着字典操作的不断执行， 最终在某个时间点上， ht[0] 的所有键值对都会被 rehash 至 ht[1] ， 这时程序将 rehashidx 属性的值设为 -1 ， 表示 rehash 操作已完成。

扩展或收缩哈希表需要将 ht[0] 里面的所有键值对 rehash 到 ht[1] 里面， 但是， 这个 rehash 动作并不是一次性、集中式地完成的， 而是分多次、渐进式地完成的。渐进式 rehash 的好处在于它采取分而治之的方式， 将 rehash 键值对所需的计算工作均滩到对字典的每个添加、删除、查找和更新操作上， 从而避免了集中式 rehash 而带来的庞大计算量。

跳跃表

跳跃表是一种随机化数据结构，查找、添加、删除操作都可以在对数期望时间下完成。
跳跃表目前在 Redis 的唯一作用，就是作为有序集类型的底层数据结构（之一，另一个构成有序集的结构是字典）。

Redis数据类型

String——字符串，set,get,decr,incr,mget

Hash——字典hget,hset,hgetall，Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口
List——列表，lpush,rpush,lpop,rpop,lrange，
twitter的关注列表，粉丝列表等都可以用Redis的list当数据量不大的时候将会list会采用ziplist而不是linklist来存储数据。st结构来实现。
Set——集合，sadd,spop,smembers,sunion
set 的内部实现是一个 value永远为null的HashMap，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。
Sorted Set——有序集合 zadd,zrange,zrem,zcard
Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

Redis持久化

RDB

RDB 是 Redis 默认的持久化方案。在指定的时间间隔内，执行指定次数的写操作，则会将内存中的数据写入到磁盘中。即在指定目录下生成一个dump.rdb文件。Redis 重启会通过加载dump.rdb文件恢复数据。
开启RDB持久化方式很简单，客户端可以通过向Redis服务器发送save或bgsave命令让服务器生成rdb文件，或者通过服务器配置文件指定触发RDB条件。
优点：
1 适合大规模的数据恢复。
2 如果业务对数据完整性和一致性要求不高，RDB是很好的选择。

缺点：
1 数据的完整性和一致性不高，因为RDB可能在最后一次备份时宕机了。
2 备份时占用内存，因为Redis 在备份时会独立创建一个子进程，将数据写入到一个临时文件（此时内存中的数据是原来的两倍哦），最后再将临时文件替换之前的备份文件。
所以Redis 的持久化和数据的恢复要选择在夜深人静的时候执行是比较合理的。

AOF

Redis 默认不开启。它的出现是为了弥补RDB的不足（数据的不一致性），所以它采用日志的形式来记录每个写操作，并追加到文件中。Redis 重启的会根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作。
优点：数据的完整性和一致性更高
缺点：因为AOF记录的内容多，文件会越来越大，数据恢复也会越来越慢。

Redis缓存穿透，缓存雪崩

缓存雪崩：
发生场景：当Redis服务器重启或者大量缓存在同一时期失效时,此时大量的流量会全部冲击到数据库上面,数据库有可能会因为承受不住而宕机
解决方案：
均匀分布：我们应该在设置失效时间时应该尽量均匀的分布,比如失效时间是当前时间加上一个时间段的随机值
熔断机制：类似于SpringCloud的熔断器,我们可以设定阈值或监控服务,如果达到熔断阈值(QPS,服务无法响应,服务超时)时,则直接返回,不再调用目标服务,并且还需要一个检测机制,如果目标服务已经可以正常使用,则重置阈值,恢复使用
隔离机制：类似于Docker一样,当一个服务器上某一个tomcat出了问题后不会影响到其它的tomcat,这里我们可以使用线程池来达到隔离的目的,当线程池执行拒绝策略后则直接返回,不再向线程池中增加任务
限流机制：其实限流就是熔断机制的一个版本,设置阈值(QPS),达到阈值之后直接返回
双缓存机制：将数据存储到缓存中时存储俩份,一份的有效期是正常的,一份的有效期长一点.不建议用这个方案,因为比较消耗内存资源,毕竟Redis是直接存储到内存中的

缓存穿透
发生场景：此时要查询的数据不存在,缓存无法命中所以需要查询完数据库,但是数据是不存在的,此时数据库肯定会返回空,也就无法将该数据写入到缓存中,那么每次对该数据的查询都会去查询一次数据库
解决方案：
布隆过滤：我们可以预先将数据库里面所有的key全部存到一个大的map里面,然后在过滤器中过滤掉那些不存在的key.但是需要考虑数据库的key是会更新的,此时需要考虑数据库 --> map的更新频率问题
缓存空值：哪怕这条数据不存在但是我们任然将其存储到缓存中去,设置一个较短的过期时间即可,并且可以做日志记录,寻找问题原因

缓存预热：
在上线前先将需要缓存的数据放到缓存中去

Redis实现分布式锁

基于Redis的setNx，expire()

```
public static boolean acquireLock(String lock) {
    // 1. 通过SETNX试图获取一个lock
    boolean success = false;
    Jedis jedis = pool.getResource();
    long value = System.currentTimeMillis() + expired + 1;
    System.out.println(value);
    long acquired = jedis.setnx(lock, String.valueOf(value));
    //SETNX成功，则成功获取一个锁
    if (acquired == 1)
        success = true;
    //SETNX失败，说明锁仍然被其他对象保持，检查其是否已经超时
    else {
        long oldValue = Long.valueOf(jedis.get(lock));

        //超时
        if (oldValue < System.currentTimeMillis()) {
            String getValue = jedis.getSet(lock, String.valueOf(value));
            // 获取锁成功
            if (Long.valueOf(getValue) == oldValue)
                success = true;
            // 已被其他进程捷足先登了
            else
                success = false;
        }
        //未超时，则直接返回失败
        else
            success = false;
    }
    pool.returnResource(jedis);
    return success;
}
```

Redis 加锁新方法 - Jediscluster.Set(Key,Value,"Nx","Ex",Expireseconds);

基于redis实现可重入锁：
记录本机ip，将本机ip设置为value，如果获取锁失败判断value是否等于本机ip，如果等于本机ip 的话也算获取锁成功

Zookeeper实现分布式锁

每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。Curator提供的InterProcessMutex是分布式锁的实现。acquire方法用户获取锁，release方法用于释放锁。

Redis的并发竞争问题如何解决

多客户端同时并发写一个key，可能本来应该先到的数据后到了，导致数据版本错了。或者是多客户端同时获取一个key，修改值之后再写回去，只要顺序错了，数据就错了。

使用zookeeper实现分布式锁，确保同一时间只有一个系统在操作某个key，同时mysql查询出的数据增加时间戳，每次要写之前先判断一下当前这个value的时间戳是否比缓存里的value时间戳要新，如果更新那么可以写，反之则不写

Redis的缓存失效策略

定时删除
设置键的过期时间时，创建一个 Timer ，当过期时间到临时，立刻删除键。
内存友好型策略，一旦键过期，就会被删除，并释放所占用的内存，Cpu 不友好，当一批数量比较多的键过期时，正好遇上Cpu 紧张的时段，这时候需要的是Cpu处理能力，而不是内存，显然 Cpu 时间用在删除过期键上，会对服务器的响应时间和吞吐量造成影响。另外当前 Redis 时间事件（无序链表O(N)）无法高效处理大量时间事件，所以定时删除并不是一种好的定时删除策略。
惰性删除
不管过期的键，在这种策略下，当键在键空间中被取出时，首先检查取出的键是否过期，若过期删除该键，否则，返回该键。
很明显，惰性删除依赖过期键的被动访问，对于内存不友好，如果一些键长期没有被访问，会造成内存泄露（垃圾数据占用内存）。我们知道，Redis是依赖内存的，所以惰性删除也不是一个好的策略。
定期删除
由定时删除算法，定期的去检查一定的数据库，删除一定的过期键。
通过合理的删除操作执行的时长和频率，达到合理的删除过期键。

Redis的缓存淘汰机制

在内存中实现数据缓存.在一些使用场景中,需要控制缓存的数据的内存消耗, 因此会自动淘汰(evict)一些缓存的数据.其实现机制一般可以基于数据的访问时间(LRU),也可以基于访问频率(LFU),或者二者的某种形式的结合

Redis的LRU实现：
Redis系统的LRU算法在淘汰数据时, 使用的是一种近似算法,并不是严格的按照访问时间进行缓存数据的淘汰. 如果按照HashMap和双向链表实现，需要额外的存储存放 next 和 prev 指针，牺牲比较大的存储空间
Redis系统中与LRU功能相关的配置参数有三个:

maxmemory. 该参数即为缓存数据占用的内存限制. 当缓存的数据消耗的内存超过这个数值限制时, 将触发数据淘汰. 该数据配置为0时,表示缓存的数据量没有限制, 即LRU功能不生效.
maxmemory_policy. 淘汰策略. 定义参与淘汰的数据的类型和属性.
maxmemory_samples. 随机采样的精度. 该数值配置越大, 越接近于真实的LRU算法,但是数值越大, 消耗的CPU计算时间越多,执行效率越低.

Redis 在实现上引入了一个 LRU 时钟来代替 unix 时间戳，每个对象的每次被访问都会记录下当前服务器的 LRU 时钟，然后用服务器的 LRU 时钟减去对象本身的时钟，得到的就是这个对象没有被访问的时间间隔（也称空闲时间），空闲时间最大的就是需要淘汰的对象。

Redis会基于server.maxmemory_samples配置选取固定数目的key，然后比较它们的lru访问时间，然后淘汰最近最久没有访问的key，maxmemory_samples的值越大，Redis的近似LRU算法就越接近于严格LRU算法，但是相应消耗也变高，对性能有一定影响，样本值默认为5。

Redis集群，高可用，原理

保证redis高可用机制需要redis主从复制、redis持久化机制、哨兵机制、keepalived（自动重启）等的支持

Redis Cluster集群的实现原理

Redis Cluster是Redis在3.0版本推出的分布式解决方案。Redis Cluster由多个Redis节点组成。不同节点之间数据无交集，每个节点对应多个数据分片。节点内部分为主备节点，通过主备复制的方式保证数据的一致性。一个主节点可以有多个从节点，主节点提供读写服务，从节点提供读服务。Redis Cluster设计的核心思想：**数据拆分、去中心化。**

去中心化：

在Redis Cluster中，原则上每个主节点都有一个或多个Slave节点。集群中所有的Master节点都可以进行读写数据，不分主次。每个主节点与从节点间通过Goossip协议内部通信，异步复制。

数据拆分：

数据分片规则：1、哈希算法：采用固定节点数量，当某一节点宕机，缓存重建。2、一致性哈希算法：当某一结点宕机，只有此节点数据受影响。会将压力压到数据库。Redis Cluster使用的时hash slot算法通过采用固定节点数量和可配置映射节点，来避免取模的不灵活性和一致性哈希的部分影响。

故障转移：

首先，在Redis Cluster中每个节点都存有集群中所有节点的信息。它们之间通过互相ping-pong判断节点是否可以连接。如果有一半以上的节点去ping一个节点的时候没有回应，集群就认为这个节点宕机。当主节点被集群公认为fail状态，那么它的从节点就会发起竞选，如果存在多个从节点，数据越新的节点越有可能发起竞选。集群中其他主节点返回响应信息。当竞选从节点收到过半主节点同意，便会成为新的主节点。此时会以最新的Epoch通过PONG消息广播，让Redis Cluster的其他节点尽快的更新集群信息。当原主节点恢复加入后会降级为从节点。

通过哨兵机制来实现

高可用性：

主节点保护：当集群中某节点中的所有从实例宕机时，Redis Cluster会将其他节点的非唯一从实例进行副本迁移，成为此节点的从实例。

这样集群中每个主节点至少有一个slave，使得Cluster 具有高可用。集群中只需要保持 2*master+1 个节点，就可以保持任一节点宕机时，故障转移后继续高可用。

集群fail条件：

1、某个主节点和所有从节点全部挂掉，则集群进入fail状态。

2、如果集群超过半数以上主节点挂掉，无论是否有从节点，集群进入fail状态。

3、如果集群任意主节点挂掉,且当前主节点没有从节点，集群进入fail状态。

哨兵机制：
哨兵机制需要主从复制的支持。
Redis的哨兵(sentinel) 系统用于管理多个 Redis 服务器,该系统执行以下三个任务:
· 监控(Monitoring): 哨兵(sentinel) 会不断地检查你的Master和Slave是否运作正常。
· 提醒(Notification):当被监控的某个Redis出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。
· 自动故障迁移(Automatic failover):当一个Master不能正常工作时，哨兵(sentinel) 会开始一次自动故障迁移操作,它会将失效Master的其中一个Slave升级为新的Master, 并让失效Master的其他Slave改为复制新的Master; 当客户端试图连接失效的Master时,集群也会向客户端返回新Master的地址,使得集群可以使用Master代替失效Master。

Redis和memcached的区别

1、Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等；
2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储；
3、虚拟内存--Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘；
4、过期策略--memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10；
5、分布式--设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从；
6、存储数据安全--memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）；
7、灾难恢复--memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复；
8、Redis支持数据的备份，即master-slave模式的数据备份；

为什么选择redis：

1. 速度快，完全基于内存，使用C语言实现，网络层使用epoll解决高并发问题，单线程模型避免了不必要的上下文切换及竞争条件

2. 丰富的数据类型，Redis有8种数据类型，当然常用的主要是 String、Hash、List、Set、 SortSet 这5种类型，他们都是基于键值的方式组织数据。每一种数据类型提供了非常丰富的操作命令，可以满足绝大部分需求
3. 有持久化策略和灾难恢复

redis的主从复制是怎么实现的

在Master和Slave互通之后，首先，Slave会发送sync同步指令，当Master收到指令后，将在后台启动存盘进程，同时收集所有来自Slave的修改数据集的指令信息，当后台进程完成之后，Master将发送对应的数据库文件到对应的Slave中，以完成一次完整的同步工作。其次Slave在接受到数据库文件之后，将其存盘并加载到内存中。最后，Master继续收集修改命令和新增的修改指令，并依次发送给Slave，其将在本次执行这些数据的修改命令，从而最终达到数据同步的实现。

Redis中hash的底层实现

哈希对象的编码可以是ziplist和hashtable之一。
（1）ziplist编码
ziplist编码的哈希对象底层实现是压缩列表，在ziplist编码的哈希对象中，key-value键值对是以紧密相连的方式放入压缩链表的，先把key放入表尾，再放入value；键值对总是向表尾添加。
（2）hashtable编码
hashtable编码的哈希对象底层实现是字典，哈希对象中的每个key-value对都使用一个字典键值对来保存。
字典键值对即是，字典的键和值都是字符串对象，字典的键保存key-value的key，字典的值保存key-value的value。
哈希对象使用ziplist编码需要满足两个条件：一是所有键值对的键和值的字符串长度都小于64字节；二是键值对数量小于512个；不满足任意一个都使用hashtable编码。

Redis实现session

当客户端第一次发送请求后，nginx将请求分发给服务器1 ，然后将服务器1 产生的session 放入redis中，这样的话 客户端、服务器1 和redis中都会有一个相

同的session，当客户端发送第二次请求的时候，nginx将请求分发给服务器2（已知服务器2 中无session），因为客户端自己携带了一个session，那么服务器2 就可以拿着客户端带来的session中的session ID去redis中获取session，只要拿到这个session，就能执行之后的操作。

MQ

Mq的作用：

1. 解耦：在项目启动之初是很难预测未来会遇到什么困难的，消息中间件在处理过程中插入了一个隐含的，基于数据的接口层，两边都实现这个接口，这样就允许独立的修改或者扩展两边的处理过程，只要两边遵守相同的接口约束即可。
2. 冗余（存储）：在某些情况下处理数据的过程中会失败，消息中间件允许把数据持久化知道他们完全被处理
3. 扩展性：消息中间件解耦了应用的过程，所以提供消息入队和处理的效率是很容易的，只需要增加处理流程就可以了。
4. 削峰：在访问量剧增的情况下，但是应用仍然需要发挥作用，但是这样的突发流量并不常见。而使用消息中间件采用队列的形式可以减少突发访问压力，不会因为突发的超时负荷要求而崩溃
5. 可恢复性：当系统一部分组件失效时，不会影响到整个系统。消息中间件降低了进程间的耦合性，当一个处理消息的进程挂掉后，加入消息中间件的消息仍然可以在系统恢复后重新处理
6. 顺序保证：在大多数场景下，处理数据的顺序也很重要，大部分消息中间件支持一定的顺序性
7. 缓冲：消息中间件通过一个缓冲层来帮助任务最高效率的执行
8. 异步通信：通过把把消息发送给消息中间件，消息中间件并不立即处理它，后续在慢慢处理。

常见的MQ：

rabbitMQ（社区活跃，高并发）：RocketMQ(阿里的，java开发，再次开发，并发高，分布式，出错少)，kafka（高吞吐量的分布式发布订阅消息系统，它可以处理消费者在网站中的所有动作流数据）

MQ常见概念：

Topic

主题，从逻辑上讲一个Topic就是一个Queue，即一个队列；从存储上讲，一个Topic存储了一类相同的消息，是一类消息的集合。比如一个名称为trade.order.queue的Topic里面存的都是订单相关的消息。

Partition
分区，一个Topic存储消息时会分为多个Partition，每个Partition内消息是有顺序的。

Producer
生产者，消息的生产方，一般由业务系统负责产生消息。
Producer负责决定将消息发送到哪个Topic的那个Partition。

Consumer
消费者，消息的消费方，一般是后台系统负责异步消费消息。
Consumer订阅Topic，消费Topic内部的消息。

Broker
消息的存储者，一般也称为Server，在JMS中叫Provider，在RocketMQ（阿里开源的消息中间件）中叫Broker。

消费方式：
集群消费
集群消费的含义是说一类Consumer（即Group相同的Consumer的集合）共同完成对一个Topic的消费。其实上面说明Consumer需要协同工作时举例中就默认是集群消费了，这也是现实业务中95%以上需求的消费方式。
广播消费
广播消费的含义是Topic中的每一条消息都会被一类Consumer（属于同一个Group的多个Consumer）中的每个Consumer实例消费。

保证数据一致性

ACK机制消息确认机制
在实际使用RocketMQ的时候我们并不能保证每次发送的消息都刚好能被消费者一次性正常消费成功，可能会存在需要多次消费才能成功或者一直消费失败的情况，那作为发送者该做如何处理呢?

为了保证数据不被丢失，RabbitMQ支持消息确认机制，即ack。发送者为了保证消息肯定消费成功，只有使用方明确表示消费成功，RocketMQ才会认为消息消费成功。中途断电，抛出异常等都不会认为成功——即都会重新投递。

保证数据能被正确处理而不仅仅是被Consumer收到，我们就不能采用no-ack或者auto-ack，我们需要手动ack(manual-ack)。在数据处理完成后手动发送ack，这个时候Server才将Message删除。

Nginx

nginx是一款反向代理的服务器，目的就是转发http请求。这样，可以不知道服务器地址，就可以对请求进行转发。nginx，可以理解为一个中间人，用户操作客户端，通过nginx转发到后端，后端请求处理返回给用户。

Nginx怎么配置负载均衡

Nginx 的 upstream目前支持的分配算法：
1)、轮询 ——1：1 轮流处理请求（默认）

每个请求按时间顺序逐一分配到不同的应用服务器，如果应用服务器down掉，自动剔除，剩下的继续轮询。
2)、权重 ——you can you up
通过配置权重，指定轮询几率，权重和访问比率成正比，用于应用服务器性能不均的情况。
3)、ip_哈希算法
每个请求按访问ip的hash结果分配，这样每个访客固定访问一个应用服务器，可以解决session共享的问题。

通过在upstream参数中添加的应用服务器IP后添加指定参数即可实现

Nginx怎么限流

Nginx自身有的请求限制模块ngx_http_limit_req_module、流量限制模块ngx_stream_limit_conn_module基于令牌桶算法，可以方便的控制令牌速率，自定义调节限流，实现基本的限流控制。
令牌以固定速率产生，并缓存到令牌桶中；
令牌桶放满时，多余的令牌被丢弃；
请求要消耗等比例的令牌才能被处理；
令牌不够时，请求被缓存。

Nginx官方版本限制IP的连接和并发分别有两个模块：
limit_req_zone 用来限制单位时间内的请求数，即速率限制,采用的漏桶算法"leaky bucket"。
limit_req_conn 用来限制同一时间连接数，即并发限制。

```
limit_req_zone $binary_remote_addr $uri zone=api_write:20m rate=10r/s; # 写

# 写10/秒
location = /api/v1/trade {
  limit_req zone=api_write burst=10;
  proxy_pass http://api_server;
}
```

limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
第一个参数：$binary_remote_addr 表示通过remote_addr这个标识来做限制，“binary_”的目的是缩写内存占用量，是限制同一客户端ip地址。
第二个参数：zone=one:10m表示生成一个大小为10M，名字为one的内存区域，用来存储访问的频次信息。
第三个参数：rate=1r/s表示允许相同标识的客户端的访问频次，这里限制的是每秒1次，还可以有比如30r/m的。
limit_req zone=one burst=5 nodelay;
第一个参数：zone=one 设置使用哪个配置区域来做限制，与上面limit_req_zone 里的name对应。
第二个参数：burst=5，重点说明一下这个配置，burst爆发的意思，这个配置的意思是设置一个大小为5的缓冲区当有大量请求（爆发）过来时，超过了访问频次限制的请求可以先放到这个缓冲区内。
第三个参数：nodelay，如果设置，超过访问频次而且缓冲区也满了的时候就会直接返回503，如果没有设置，则所有请求会等待排队。

怎么使用nginx缓存

我们只需要两个命令就可以启用基础缓存： proxy_cache_path和proxy_cache
proxy_cache_path用来设置缓存的路径和配置，proxy_cache用来启用缓存。

为什么使用nginx

微服务架构中，众多服务被拆分解耦，并部署到不同的容器以及服务器中，你可能使用了一 个服务发现系统(例如 etcd 或 Eureka)或者一个资源管理框架来管理所有这些服务，可如果 你想让你的用户去从互联网访问你的某些微服务，你就必需使用一个反向代理服务器，从而 使你的众多微服务能够被访问到。
还有一个问题是当你拥有多个服务实例时，你希望能够轻松地连接到它们，将你的请求在它 们中高效地分发，并以最快的方式执行，所以不同服务实例之间的负载均衡也是非常重要的 问题。
而 Nginx 作为一款优秀的反向代理服务器和负载均衡服务器，他的诸多优秀特性成为了我们 选择他的原因:
模块化设计
Nginx 采用高度模块化设计，使得具有较好的扩展性，在 Nginx 中，除了少量的核心代 码，其他一切皆为模块。所有模块间是分层次、分类别的，官方 Nginx 有五大类型的模块: 核心模块、配置模块、事件模块、HTTP 模块、Mail 模块。

• 高可靠性
高可靠性是指服务可靠性。Nginx 是一个高可靠性的 Web 服务器，这也是我们为什么选 择 Nginx 的基本条件。Nginx 采用一个主进程(master)和 N 个工作进程(worker)的工作模式，而 worker 进程才是真正复制相应用户请求的进程。配置了缓存时还会有缓存加载器进程 (cacheloader)和缓存管理器进程 (cachemanager)等。所有进程均是仅含有一个线程，并主要通 过“共享内存”的机制实现进程间通信。

• 支持热部署
Nginx 使用主进程和 worker 工作进程的机制，使得 Nginx 支持热部署，这个热部署包括不停止服务更新配置文件、更新日志文件、以及更新服务器程序版本，也称为平滑升级。

• 低内存消耗
Nginx 对于内存的消耗是非常小的，特别是对于非活动连接。Nginx 对于非活动连接是

指，当我们开启持久连接功能时，用户连接不再发送数据后就会立即转为非活动连接，直到 持久连接超时时间到达才销毁。在一般的情况下，10000 个非活跃的 HTTPKeep-Alive 连接在 Nginx 中仅消耗 2.5M 的内存，这也是 Nginx 支持高并发连接的基础。

• 高扩展性
Nginx 的设计具有扩展性，它完全是由多个不同功能、不同层次、不同类型且耦合度极低 的模块组成。因此，当对某一个模块修复 Bug 或进行升级时，可以专注于模块自身，无须在 意其他。Nginx 支持磁盘异步 I/O(AIO)、内存映射机制(MMAP)、事件驱动机制(Event- driven)、单线程 N 请求等等。

• 高并发
Nginx 是异步非阻塞的。在需要进程等待的过程中，这些闲置的进程就空闲出来待命，而
webserver 的工作性质决定了每个 request 的大部份生命都是在网络传输中，实际上花费在 server 机器上的时间片不多，因此就表现为几个进程解决了高并发的问题。

Nginx的替代

Caddy

Nginx处理http

析请求行；
解析请求头；
读取请求体；
开始最重要的部分，即多阶段处理； nginx把请求处理划分成了11个阶段，也就是说当nginx读取了请求行和请求头之后，将请求封装了结构体ngx_http_request_t，然后每个阶段的handler都会根据这个ngx_http_request_t，对请求进行处理，例如重写uri，权限控制，路径查找，生成内容以及记录日志等等；
将结果返回给客户端；

在Nginx中，如何使用未定义的服务器名称来阻止处理请求?

只需将请求删除的服务器就可以定义为：

```
Server {
listen 80;
server_name “ “ ;
return 444;
}
```

这里，服务器名被保留为一个空字符串，它将在没有"主机"头字段的情况下匹配请求，而一个特殊的Nginx的非标准代码444被返回，从而终止连接。

6、 使用"反向代理服务器"的优点是什么?

反向代理服务器可以隐藏源服务器的存在和特征。它充当互联网云和web服务器之间的中间层。这对于安全方面来说是很好的，特别是当您使用web托管服务时。

7、请列举Nginx服务器的最佳用途。

Nginx服务器的最佳用法是在网络上部署动态HTTP内容，使用SCGI、WSGI应用程序服务器、用于脚本的FastCGI处理程序。它还可以作为负载均衡器。

8、请解释Nginx服务器上的Master和Worker进程分别是什么?

Master进程：读取及评估配置和维持
Worker进程：处理请求

9、请解释你如何通过不同于80的端口开启Nginx?

为了通过一个不同的端口开启Nginx，你必须进入/etc/Nginx/sites-enabled/，如果这是默认文件，那么你必须打开名为“default”的文件。编辑文件，并放置在你想要的端口：

Like server { listen 81; }

10、请解释是否有可能将Nginx的错误替换为502错误、503?

502 =错误网关

503 =服务器超载

有可能，但是您可以确保fastcgi_intercept_errors被设置为ON，并使用错误页面指令。

```
Location / {
fastcgi_pass 127.0.01:9001;
fastcgi_intercept_errors on;
error_page 502 =503/error_page.html;
#…
}
```

14、请陈述stub_status和sub_filter指令的作用是什么?

Stub_status指令：该指令用于了解Nginx当前状态的当前状态，如当前的活动连接，接受和处理当前读/写/等待连接的总数

Sub_filter指令：它用于搜索和替换响应中的内容，并快速修复陈旧的数据

15、解释Nginx是否支持将请求压缩到上游?

您可以使用Nginx模块gunzip将请求压缩到上游。gunzip模块是一个过滤器，它可以对不支持"gzip"编码方法的客户机或服务器使用"内容编码:gzip"来解压缩响应。

16、解释如何在Nginx中获得当前的时间?

要获得Nginx的当前时间，必须使用SSI模块、$date_gmt和$date_local的变量。

Proxy_set_header THE-TIME $date_gmt;

17、用Nginx服务器解释-s的目的是什么?

用于运行Nginx -s参数的可执行文件。

18、解释如何在Nginx服务器上添加模块?

在编译过程中，必须选择Nginx模块，因为Nginx不支持模块的运行时间选择。

Dubbo

Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，　　Dubbo采用的是一种非常简单的模型，要么是提供方

提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色

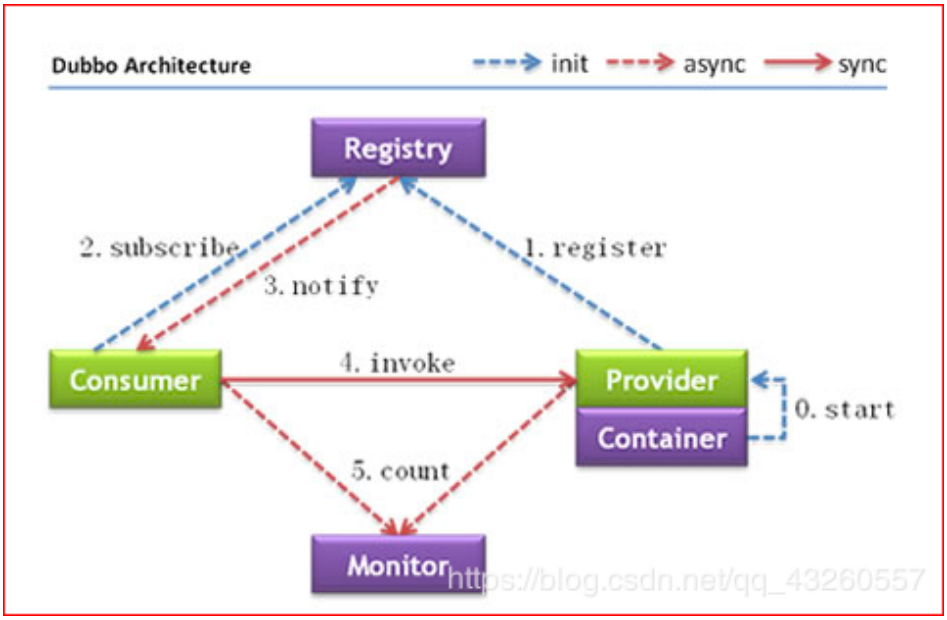
就是本地有对远程方法的描述，包括方法名、参数、返回值，在dubbo中是远程和本地使用同样的接口；然后呢，要有对网络通信的封装，要对调用方来说通信细节是完全不可见的，网络通信要做的就是将调用方法的属性通过一定的协议（简单来说就是消息格式）传递到服务端；服务端按照协议解析出调用的信息；执行相应的方法；在将方法的返回值通过协议传递给客户端；客户端再解析；在调用方式上又可以分为同步调用和异步调用；

Dubbo里面有哪几种节点角色?

节点	角色说明
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Registry	服务注册与发现的注册中心
Monitor	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

杰夫的角色
Index Service: 索引服务，提供注册中心地址列表；
注册中心：提供服务注册、订阅，服务上下线；配置下发、状态读取（如client所拥有的地址列表），为了容灾多实例部署；
Monitor Service：收集所有客户端所上传的性能统计数据；
Web管理端：服务管理界面，可以在此配置权重、路由规则等；

Dubbo 服务注册与发现的流程?



调用关系说明

0. 服务容器负责启动，加载，运行服务提供者。
1. 服务提供者在启动时，向注册中心注册自己提供的服务。
2. 服务消费者在启动时，向注册中心订阅自己所需的服务。
3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
4. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

Dubbo 架构具有以下几个特点，分别是连通性、健壮性、伸缩性、以及向未来架构的升级性。

Dubbo默认使用什么注册中心，还有别的选择吗

Zookeeper（Zookeeper 是 Apacahe Hadoop 的子项目，是一个树型的目录服务，支持变更推送，适合作为 Dubbo 服务的注册中心，工业强度较高，可用于生产环境，并推荐使用 [1]。）作为注册中心

redis（阿里内部并没有采用 Redis 做为注册中心，而是使用自己实现的基于数据库的注册中心，即：Redis 注册中心并没有在阿里内部长时间运行的可靠性保障，此 Redis 桥接实现只为开源版本提供，其可靠性依赖于 Redis 本身的可靠性）不推荐
simple，mukticast

Dubbo启动时如果依赖服务不可用怎么办?

Dubbo缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止Spring初始化完成，默认check="true",可以通过check="false"关闭检查。

Dubbo推荐使用什么序列化框架，你知道的还有哪些?

推荐使用Hessian序列化，它的底层就是基于Hessian，还有0Dubbo，fastJson，java自带的序列化

Dubbo默认使用的是什么通信框架，还有别的选择吗?

默认是Netty框架，也是推荐使用的，还有Mina，，Grizzly

Linux

Linux下的五种IO模型

阻塞IO模型：进程会一直阻塞，直到数据拷贝完成
非阻塞IO模型：非阻塞IO通过进程反复调用IO函数（多次系统调用，并马上返回）；在数据拷贝的过程中，进程是阻塞的；
IO复用模型：主要是select和epoll；对一个IO端口，两次调用，两次返回，比阻塞IO并没有什么优越性；关键是能实现同时对多个IO端口进行监听；
信号驱动IO：两次调用，两次返回；
异步IO模型：数据拷贝的时候进程无需阻塞，实际处理这个调用的函数在完成后，通过状态、通知和回调来通知调用者的输入输出操作

Select/poll/epoll

（1）select，poll实现需要自己不断轮询所有fd（file descriptor）集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用 epoll_wait

不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在 epoll_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的 时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间，这就是回调机制带来的性能提升。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要 一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内 部定义的等待队列），这也能节省不少的开销。

线上cpu占比过高排查

1．登录服务器，执行top命令查看CPU占用情况，找出CPU占用高的进程ID。
2．定位线程，$top -Hp 1893，找到id位4519的线程占用cpu过高
3. printf %x 4 5 1 9 把4519转为16进制
4. 接下来，通过jstack命令，查看栈信息
   sudo -u admin jstack1893lgrep -A20011a7

redis连接没有及时释放，导致机器内存打满
建立redis连接时没有传端口号 导致一致在建立

前一天删除了很多计划但是没删除物料
之后删除计划下面的物料
导致数据库磁盘繁忙
1.找dba查询调用量大的sql
2.整体kill一波
3.页面限流
4.排查worker调用

Top

top命令分为上下两个部分：

上面是系统统计信息
下面是进程信息。

PID （进程id）
USER （进程所有者的用户名）

PR （进程优先级）
NI （nice值。负值表示高优先级，正值表示低优先级）
VIRT （进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES）
RES （进程使用的、未被换出的物理内存大小，单位kb。
RES=CODE+DATA）
SHR （共享内存大小，单位kb）
S （进程状态。D=不可中断的睡眠状态 R=运行 S=睡眠 T=跟踪/停止 Z=僵尸进程）
%CPU （上次更新到现在的CPU时间占用百分比）
%MEM （进程使用的物理内存百分比）
TIME+ （进程使用的CPU时间总计，单位1/100秒）
COMMAND （进程名称[命令名/命令行]）

Sed

```
sed [-nefri] ‘command’ 输入文本
```

常用选项：
-n:使用安静(silent)模式。在一般 sed 的用法中，所有来自 STDIN的资料一般都会被列出到萤幕上。但如果加上 -n 参数后，则只有经过sed 特殊处理的那一行(或者动作)才会被列出来。
-e:直接在指令列模式上进行 sed 的动作编辑；
-f:直接将 sed 的动作写在一个档案内， -f filename 则可以执行 filename 内的sed 动作；
-r:sed 的动作支援的是延伸型正规表示法的语法。(预设是基础正规表示法语法)
-i:直接修改读取的档案内容，而不是由萤幕输出。

常用命令：
a :新增， a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～
c :取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！
d :删除，因为是删除啊，所以 d 后面通常不接任何咚咚；
i :插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；
p :列印，亦即将某个选择的资料印出。通常 p 会与参数 sed -n 一起运作～
s :取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配正规表示法！例如 1,20s/old/new/g 就是啦！

举例：（假设我们有一文件名为ab）

删除某行

```
[root@localhost ruby] # sed '1d' ab          #删除第一行
[root@localhost ruby] # sed '$d' ab          #删除最后一行
[root@localhost ruby] # sed '1,2d' ab        #删除第一行到第二行
```

```
[root@localhost ruby] # sed '2,$d' ab          #删除第二行到最后一行

显示某行
[root@localhost ruby] # sed -n '1p' ab         #显示第一行
[root@localhost ruby] # sed -n '$p' ab         #显示最后一行
[root@localhost ruby] # sed -n '1,2p' ab       #显示第一行到第二行
[root@localhost ruby] # sed -n '2,$p' ab       #显示第二行到最后一行

使用模式进行查询
[root@localhost ruby] # sed -n '/ruby/p' ab    #查询包括关键字ruby所在所有行
[root@localhost ruby] # sed -n '/\$/p' ab      #查询包括关键字$所在所有行,
使用反斜线\屏蔽特殊含义

增加一行或多行字符串
[root@localhost ruby]# cat ab
Hello!
ruby is me,welcome to my blog.
end
[root@localhost ruby] # sed '1a drink tea' ab  #第一行后增加字符串"drink tea"
Hello!
drink tea
ruby is me,welcome to my blog.
end
[root@localhost ruby] # sed '1,3a drink tea' ab #第一行到第三行后增加字符
串"drink tea"
Hello!
drink tea
ruby is me,welcome to my blog.
drink tea
end
drink tea
[root@localhost ruby] # sed '1a drink tea\nor coffee' ab   #第一行后增加多行,
使用换行符\n
Hello!
drink tea
or coffee
ruby is me,welcome to my blog.
end

代替一行或多行
[root@localhost ruby] # sed '1c Hi' ab              #第一行代替为Hi
Hi
ruby is me,welcome to my blog.
end
[root@localhost ruby] # sed '1,2c Hi' ab            #第一行到第二行代替为Hi
```

```
Hi
end

替换一行中的某部分
格式：sed 's/要替换的字符串/新的字符串/g'　（要替换的字符串可以用正则表达式）
[root@localhost ruby] # sed -n '/ruby/p' ab l sed 's/ruby/bird/g'　#替换ruby为bird
[root@localhost ruby] # sed -n '/ruby/p' ab l sed 's/ruby//g'　　#删除ruby

插入
[root@localhost ruby] # sed -i '$a bye' ab　　#在文件ab中最后一行直接输入"bye"
[root@localhost ruby]# cat ab
Hello!
ruby is me,welcome to my blog.
end
bye

删除匹配行

sed -i '/匹配字符串/d' filename　（注：若匹配字符串是变量，则需要“”，而不是‘’。记得好像是）

替换匹配行中的某个字符串

sed -i '/匹配字符串/s/替换源字符串/替换目标字符串/g' filename
```

awk

awk是一个强大的文本分析工具，相对于grep的查找，sed的编辑，awk在其对数据分析并生成报告时，显得尤为强大。简单来说awk就是把文件逐行的读入，以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。
awk工作流程是这样的：读入有'\n'换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域，$0则表示所有域,$1表示第一个域,$n表示第n个域。默认域分隔符是"空白键" 或 "[tab]键",所以$1表示登录用户，$3表示登录用户ip,以此类推。

搜索/etc/passwd有root关键字的所有行

```
#awk -F: '/root/' /etc/passwd
```

怎么查看进程和杀死进程

查看进程
ps -ef|grep java

使用kill命令结束进程：kill xxx

打印一个文件夹中的所有文件

Ls

线程和进程的区别

进程和线程的定义和区别

1、进程定义

进程：是具有一定独立功能的程序关于某个数据集合上的一次进行活动，是系统进行资源分配和调度的一个独立单位。
2、线程定义
线程：是进程的一个实体，是cpu调度和分派的基本单位，他是比进程更小的能够独立运行的基本单位，线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源。一个线程可以创建和撤销拎一个线程。
区别：
1、一个线程只能属于一个进程，而一个进程可以拥有多个线程。
2、线程是进程工作的最新单位。
3、一个进程会分配一个地址空间，进程与进程之间不共享地址空间。即不共享内存。
4、同一个进行下的不同的多个线程，共享父进程的地址空间。
5、线程在执行过程中，需要协助同步。不同进程的线程间要利用消息通信的办法实现同步。
6、线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。

僵尸进程和孤儿进程

linux提供一种机制使子进程在退出时候，父进程能够收集到子进程的结束状态信息（子进程pid，退出状态，运行时间等）。父进程需要调用 wait/waitpid来获取这些信息。父进程收集这些信息后这些信息才会释放。

linux下新进程的创建可以由fork来产生新的子进程。然后根据fork的返回值（小于0，等于0，大于0）判断是fork出错，子进程还是父进程。通常情况下，父进程需要在子进程任务结束退出后做“善后”，也就是一些资源清理工作。子进程退出时会把打开的文件句柄，内存占用，打开的资源进行释放，但是不会清理进程控制块PCB信息。

孤儿进程：
一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
僵尸进程：
一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程
孤儿进程：
孤儿进程是没有父进程的进程，孤儿进程这个重任就落到了init进程身上，init进程就好像是一个民政局，专门负责处理孤儿进程的善后工作。每当出现一个孤儿进程的时候，内核就把孤 儿进程的父进程设置为init，而init进程会循环地wait()它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init进程就会代表党和政府出面处理它的一切善后工作。因此孤儿进程并不会有什么危害。
僵尸进程：
父进程还在运行但是子进程挂了，但是父进程却没有使用wait来清理子进程的进程信息，导致子进程虽然运行实体已经消失，但是仍然在内核的进程表中占据一条记录，这样长期下去对于系统资源是一个浪费。

1、通过信号机制
子进程退出时向父进程发送SIGCHILD信号，父进程处理SIGCHILD信号。调用wait()或者waitpid()，让父进程阻塞等待僵尸进程的出现，处理完在继续运行父进程。
2、杀死父进程
当父进程陷入死循环等无法处理僵尸进程时，强制杀死父进程，那么它的子进程，即僵尸进程会变成孤儿进程，由系统来回收。
3、重启系统

当系统重启时，所有进程在系统关闭时被停止，包括僵尸进程，开启时init进程会重新加载其他进程。

线程的通信方式，进程的通信方式

几种进程间的通信方式
（1） 管道（pipe）：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有血缘关系的进程间使用。进程的血缘关系通常指父子进程关系。
（2）有名管道（named pipe）：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间通信。
（3）信号量（semophore）：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它通常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
（4）消息队列（message queue）：消息队列是由消息组成的链表，存放在内核中 并由消息队列标识符标识。消息队列克服了信号传递信息少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
（5）信号（signal）：信号是一种比较复杂的通信方式，用于通知接收进程某一事件已经发生。
（6）共享内存（shared memory）：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问，共享内存是最快的IPC方式，它是针对其他进程间的通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信。
（7）套接字（socket）：套接口也是一种进程间的通信机制，与其他通信机制不同的是它可以用于不同及其间的进程通信。

几种线程间的通信机制

1、锁机制
1.1 互斥锁：提供了以排它方式阻止数据结构被并发修改的方法。
1.2 读写锁：允许多个线程同时读共享数据，而对写操作互斥。
1.3 条件变量：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

2、信号量机制：包括无名线程信号量与有名线程信号量
3、信号机制：类似于进程间的信号处理。

系统线程的数量上限是多少

/proc/sys/kernel/pid_max

页式存储的概念

把内存物理空间和程序逻辑空间分成大小相等的块，逻辑空间中的块称为页面，物理空间中的块称为物理块。

页式存储是指存储的时候以页面作为基本的存储单位,一个大的作业分存在N个页里,当执行作业的时候不需要同事加载所有的页,而是用到哪些加载哪些,页式存储让资源的效率更高

内存碎片

频繁地请求和释放不同大小的内存，必然导致内存碎片问题的产生，结果就是当再次要求分配连续的内存时，即使整体内存是足够的，也无法满足连续内存的需求。该问题也称之为外碎片(external fragmentation)。
伙伴算法，用于管理物理内存，避免内存碎片;
高速缓存Slab层用于管理内核分配内存，避免碎片。

伙伴算法

把所有的空闲页框分组为11个块链表，每个链表分别包含大小为1,2,4,8,16,32,64,128,256,512,1024个连续的页框，对1024个页框的最大请求对应着4MB大小的连续RAM（每页大小为4KB），每个块的第一个页框的物理地址是该块大小的整数倍，例如，大小为16个页框的块，其起始地址是16*2^12（2^12即是4K）的倍数。
我们通过一个例子来说明伙伴算法的工作原理，假设现在要请求一个256个页框的块（1MB），算法步骤如下：
• 在256个页框的链表中检查是否有一个空闲快，如果没有，查找下一个更大的块，如果有，请求满足。
• 在512个页框的链表中检查是否有一个空闲块，如果有，把512个页框的空闲块分为两份，第一份用于满足请求，第二份链接到256个页框的链表中。如果没有空闲块，继续寻找下一个更大的块。

高速缓存Slab

slab是Linux操作系统的一种内存分配机制。其工作是针对一些经常分配并释放的对象，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内存碎片，而且处理速度也太慢。而slab分配器是基于对象进行管理的，相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab分配器就从一个slab列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化。

计算机网络

OSI分层7层模型

每一层的协议如下：
物理层：RJ45、CLOCK、IEEE802.3 （中继器，集线器，网关）
数据链路：PPP、FR、HDLC、VLAN、MAC （网桥，交换机）
网络层：IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP、 （路由器）
传输层：TCP、UDP、SPX
会话层：NFS、SQL、NETBIOS、RPC
表示层：JPEG、MPEG、ASII
应用层：FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS
每一层的作用如下：
物理层：通过媒介传输比特,确定机械及电气规范（比特Bit）
数据链路层：将比特组装成帧和点到点的传递（帧Frame）
网络层：负责数据包从源到宿的传递和网际互连（包PackeT）
传输层：提供端到端的可靠报文传递和错误恢复（段Segment）
会话层：建立、管理和终止会话（会话协议数据单元SPDU）
表示层：对数据进行翻译、加密和压缩（表示协议数据单元PPDU）
应用层：允许访问OSI环境的手段（应用协议数据单元APDU）

ARP是地址解析协议，简单语言解释一下工作原理。

1：首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址之间的对应关系。

2：当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP数据包，该数据包包括的内容有：**源主机 IP地址，源主机MAC地址，目的主机的IP 地址**。

3：当本网络的所有主机收到该ARP数据包时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的IP和MAC地址写入到ARP列表中，如果已经存在，则覆盖，然后将自己的MAC地址写入ARP响应包中，告诉源主机自己是它想要找的MAC地址。

4：源主机收到ARP响应包后。将目的主机的IP和MAC地址写入ARP列表，并利用此信息发送数据。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。

广播发送ARP请求，单播发送ARP响应。

RARP逆地址解析协议

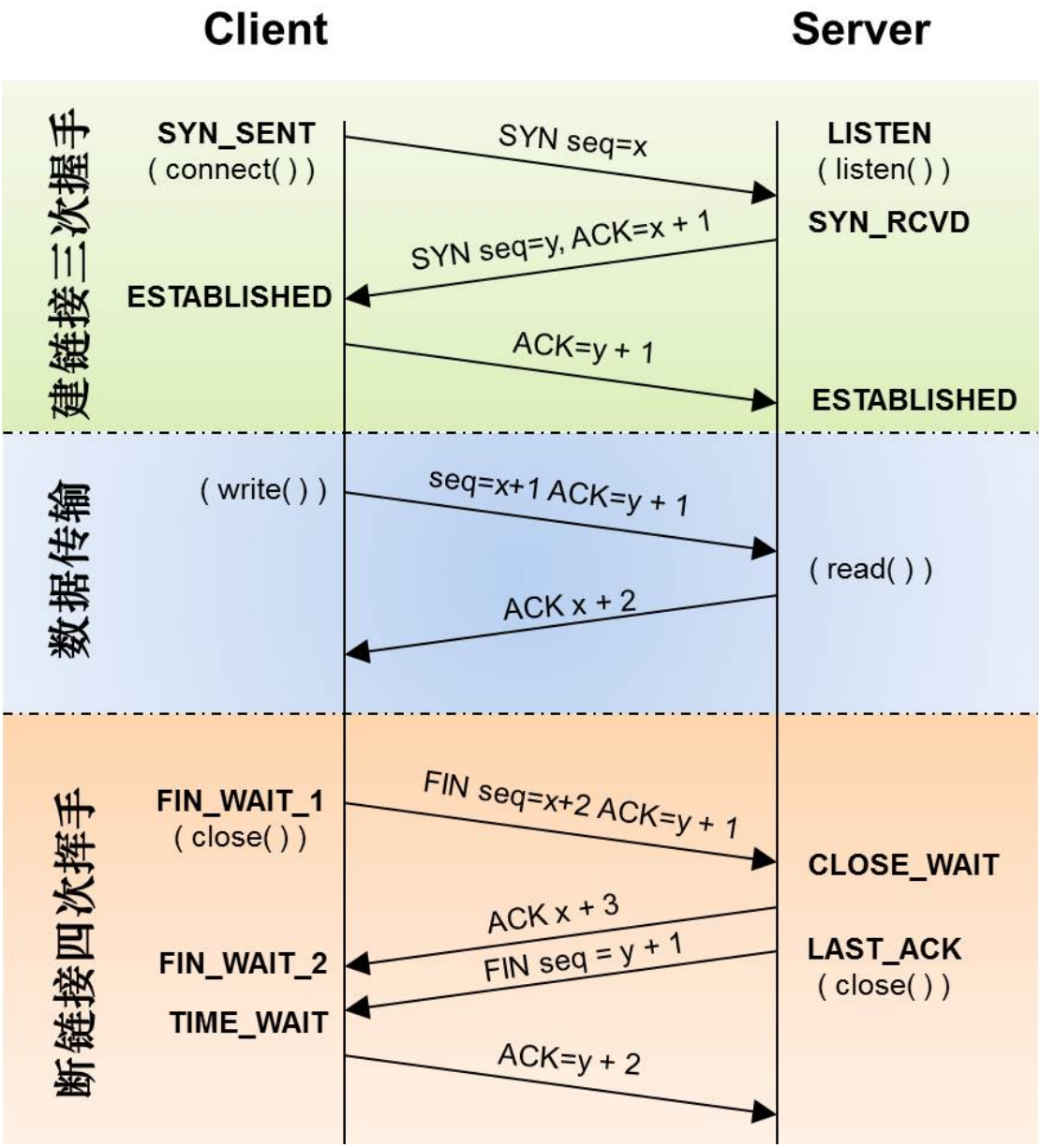
RARP是逆地址解析协议，作用是完成硬件地址到IP地址的映射，主要用于无盘工作站，因为给无盘工作站配置的IP地址不能保存。工作流程：在网络中配置一台RARP服务器，里面保存着IP地址和MAC地址的映射关系，当无盘工作站启动后，就封装一个RARP数据包，里面有其MAC地址，然后广播到网络上去，当服务器收到请求包后，就查找对应的MAC地址的IP地址装入响应报文中发回给请求者。因为需要广播请求报文，因此RARP只能用于具有广播能力的网络。

TCP三次握手和四次挥手的全过程

三次握手：
第一次握手：客户端发送syn包(syn=x)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=x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入SYN_RECV状态；(y表示即最后被成功接收的数据字节序列号)

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=y+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。
握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。

四次握手
与建立连接的“三次握手”类似，断开一个TCP连接则需要“四次握手”。
第一次挥手：主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭方告诉被动关闭方：我已经不 会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可 以接受数据。
第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号）。
第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了。
第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手。



在浏览器中输入www.baidu.com后执行的全部过程

1、客户端浏览器通过DNS解析到www.baidu.com的IP地址220.181.27.48，通过这个IP地址找到客户端到服务器的路径。客户端浏览器发起一个HTTP会话到220.161.27.48，然后通过TCP进行封装数据包，输入到网络层。
2、在客户端的传输层，把HTTP会话请求分成报文段，添加源和目的端口，如服务器使用80端口监听客户端的请求，客户端由系统随机选择一个端口如5000，与服务器进行交换，服务器把相应的请求返回给客户端的5000端口。然后使用IP层的IP地址查找目的端。
3、客户端的网络层不用关系应用层或者传输层的东西，主要做的是通过查找路由表确定如何到达服务器，期间可能经过多个路由器，这些都是由路由器来完

成的工作，我不作过多的描述，无非就是通过查找路由表决定通过那个路径到达服务器。
4、客户端的链路层，包通过链路层发送到路由器，通过邻居协议查找给定IP地址的MAC地址，然后发送ARP请求查找目的地址，如果得到回应后就可以使用ARP的请求应答交换的IP数据包现在就可以传输了，然后发送IP数据包到达服务器的地址。

TCP和UDP的区别?

TCP提供面向连接的、可靠的数据流传输，而UDP提供的是非面向连接的、不可靠的数据流传输。
TCP传输单位称为TCP报文段，UDP传输单位称为用户数据报。
TCP注重数据安全性，UDP数据传输快，因为不需要连接等待，少了许多操作，但是其安全性却一般。
1.先说一下TCP的优缺点吧。优点呢，TCP是可靠的连接，由于有基本的重传确认机制，可以保证把一个数据块完完整整的从A传到B；缺点也是因优点而生，因为有三次握手，所以会传输更多的包，浪费一些带宽；因为需要可靠地连接进行通信，则需要双方都必须持续在线，所以在通信过程中server需要维持非常大的并发连接，浪费了系统资源，甚至会出现宕机；再者就是因为有重传确认，则会浪费一部分的带宽，且在不好的网络中，会因为不断地连接断开连接，严重降低了传输效率。

2.相对于TCP来说，UDP是非面向连接的不可靠的协议，其优点也因为缺点而生。首先，因为没有三次握手，所以会起步比较快，延时小；另外，由于不需要双方持续在线，所以server不用维护巨量的并发连接，节省了系统资源；
三，因为没有重传确认，虽然到达的数据可能会有所缺失，但在不影响使用的情况下，能更高效的利用网络带宽。

基于前面的说法，总结一下本文的问题答案。TCP适合实时性要求不高，但要求内容要完整传输的应用。相比而言，UDP由于无连接、无重传确认，所以传输效率高、延时小，适合实时性要求高的应用，如游戏服务器，音频，视频等；另外，由于不用维持大的并发量，所以适合巨量服务的server，加上合适的时间控制，可以用来设计更大的并发服务器；再者就是，UDP可以更高效的利用网络带宽。

说了这么多，举个现实的例子吧。国人几乎都是用的QQ，在建立连接阶段，使用的是面向连接的TCP协议，通过三次握手来完成；然后，在文字数据传输阶段，使用的是UDP协议，但需要中间服务器转发（估计是使用了connect()的UDP，QQ离线发送/接收数据的基础);然后，音视频数据的发送一定是使用的

UDP，因为一般的客户可以容忍稍微模糊(略有缺失的数据块）的声音或视频，但估计不会接受一会断开一会连接（因为TCP容易断线）的音视频；而文件传输则使用了P2P的协议，当前大多P2P使用的UTP协议也是基于UDP的，因为使用TCP的话会浪费大量的带宽。UDP比较容易NAT穿透

TCP和UDP的报文头

TCP对应的协议和UDP对应的协议

TCP对应的协议：
（1） FTP：定义了文件传输协议，使用21端口。
（2） Telnet：一种用于远程登陆的端口，使用23端口，用户可以以自己的身份远程连接到计算机上，可提供基于DOS模式下的通信服务。
（3） SMTP：邮件传送协议，用于发送邮件。服务器开放的是25号端口。
（4） POP3：它是和SMTP对应，POP3用于接收邮件。POP3协议所用的是110端口。
（5）HTTP：是从Web服务器传输超文本到本地浏览器的传送协议。

UDP对应的协议：
（1） DNS：用于域名解析服务，将域名地址转换为IP地址。DNS用的是53号端口。
（2） SNMP：简单网络管理协议，使用161号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。
（3） TFTP(Trival File Tran敏感词er Protocal)，简单文件传输协议，该协议在熟知端口69上使用UDP服务。

http和https的区别：

HTTP（HyperText Transfer Protocol：超文本传输协议）是一种用于分布式、协作式和超媒体信息系统的应用层协议。 简单来说就是一种发布和接收 HTML 页面的方法，被用于在 Web 浏览器和网站服务器之间传递信息。
HTTP 默认工作在 TCP 协议 80 端口，用户访问网站 http:// 打头的都是标准 HTTP 服务。

HTTP 协议以明文方式发送内容，不提供任何方式的数据加密，如果攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息，因此，HTTP协议不适合传输一些敏感信息，比如：信用卡号、密码等支付信息。
HTTPS（Hypertext Transfer Protocol Secure：超文本传输安全协议）是一种透过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，但利用 **SSL/TLS** 来加密数据包。HTTPS 开发的主要目的，是提供对网站服务器的身份认证，保护交换数据的隐私与完整性。

HTTPS 默认工作在 TCP 协议443端口，它的工作流程一般如以下方式：
1、TCP 三次同步握手
2、客户端验证服务器数字证书
3、DH 算法协商对称加密算法的密钥、hash 算法的密钥
4、SSL 安全加密隧道协商完成
5、网页以加密的方式传输，用协商的对称加密算法和密钥加密，保证数据机密性；用协商的hash算法进行数据完整性保护，保证数据不被篡改。

TCP流量控制、拥塞控制

拥塞控制：拥塞控制是作用于网络的，它是防止过多的数据注入到网络中，避免出现网络负载过大的情况；常用的方法就是：（1）慢开始、拥塞避免（2）快重传、快恢复。
流量控制：流量控制是作用于接收者的，它是控制发送者的发送速度从而使接收者来得及接收，防止分组丢失的。由滑动窗口协议（连续ARQ协议）实现
接收端可以根据自己的状况通告窗口大小，从而控制发送端的接收，进行流量控制

慢开始，拥塞避免
发送方维持一个叫做拥塞窗口cwnd（congestion window）的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。

慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。1，2，4，8
拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍。这样拥塞窗口按线性规律缓慢增长。

当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把ssthresh门限减半（为了预防网络发生拥塞）。但是接下去并不执行慢开始算法
考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将cwnd设置为ssthresh减半后的值，然后执行拥塞避免算法，使cwnd缓慢增大。

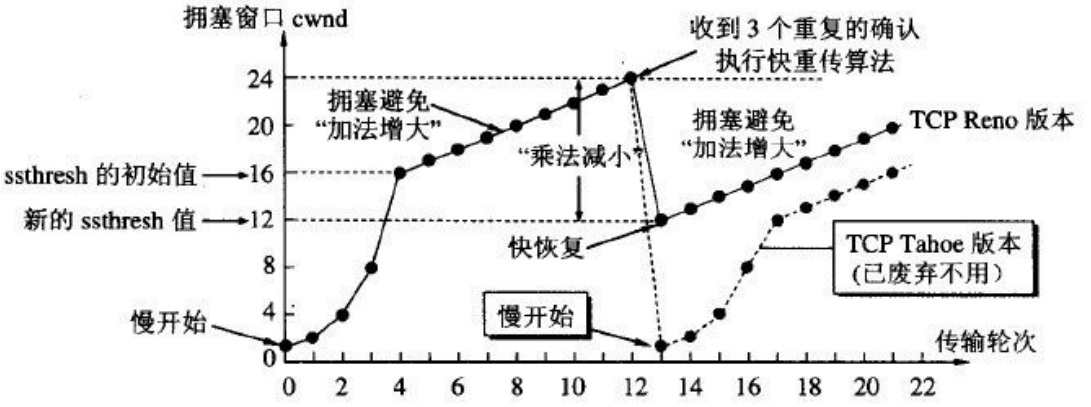


图 5-27　从连续收到三个重复的确认转入拥塞避免

设计模式

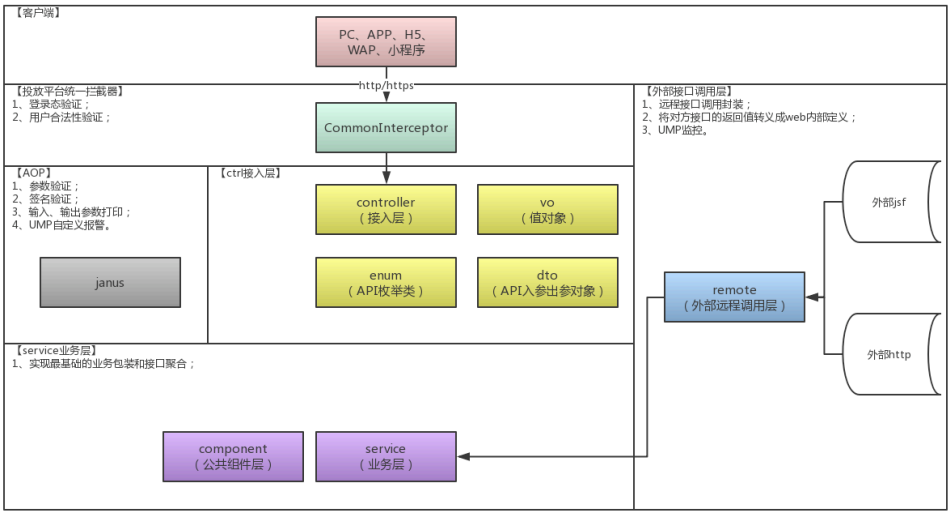
六大原则

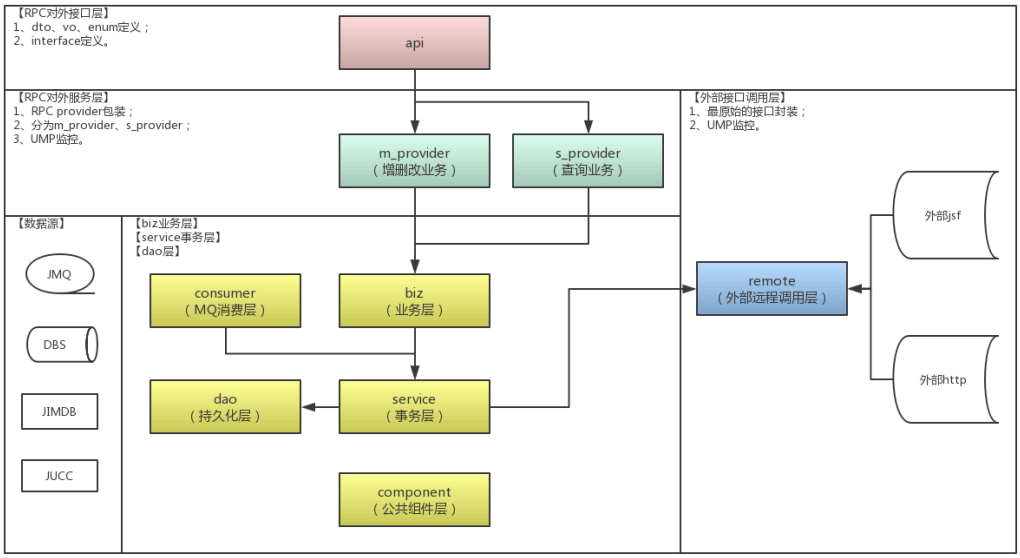
1. 单一职责原则：一个类只负责一个功能领域中的相应职责，用户操作类，现金操作类，红包操作类等
2. 开闭原则：对扩展开放，对修改关闭，抽象出公共父类，面向对象设计的目标
3. 里氏替换原则：所有引用父类的地方必须能透明地使用其子类的对象
4. 依赖倒置原则：面向对象设计的主要实现机制，要针对接口编程，不是针对实现编程。具体实现类通过依赖注入DI注入到其他对象中
5. 接口隔离原则：使用多个专门的接口，不使用单一的总接口。分为各种服务， 每种服务都有对应的处理对象
6. 迪米特法则：降低类之间的耦合

设计模式

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式（用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象）
结构型模式，共七种：适配器模式（继承被适配的类，也实现了要适配的接口）、装饰者模式(装饰者与被装饰者拥有共同的超类一个抽象类，继承的目的是继承类型，而不是行为)、代理模式(对一些对象提供代理，以限制那些对象去访问其它对象)、外观模式(医院挂号的接待员，由接待员负责代为挂号、划价、缴费、取药等，病人只和接待员解除)、桥接模式（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）、组合模式(树叶和树枝都实现了树，他们的实现构成了树)、享元模式（相同的对象只创建一次，可以存放在map中，每次获取时判断是否创建，已经创建则直接拿，五子棋）。
行为型模式，共十一种：策略模式(对不同的场景采用不同的策略实现)、模板方法模式（AQS）、观察者模式(对一个对象的改变需要同时改变其它对象)、迭代子模式（一个白箱聚集向外界提供访问自己内部元素的接口（称作遍历方法或者Traversing Method），从而使外禀迭代子可以通过聚集的遍历方法实现迭代功能。因为迭代的逻辑是由聚集对象本身提供的，所以这样的外禀迭代子角色往往仅仅保持迭代的游标位置）、责任链模式（使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系，将这个对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理他为止）、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

项目结构图





秒杀系统设计

快：响应快，处理请求速度快，秒杀涉及到大量的并发读和并发写，在并发数比较大的情况下，如何保证系统的TPS和QPS是关键。

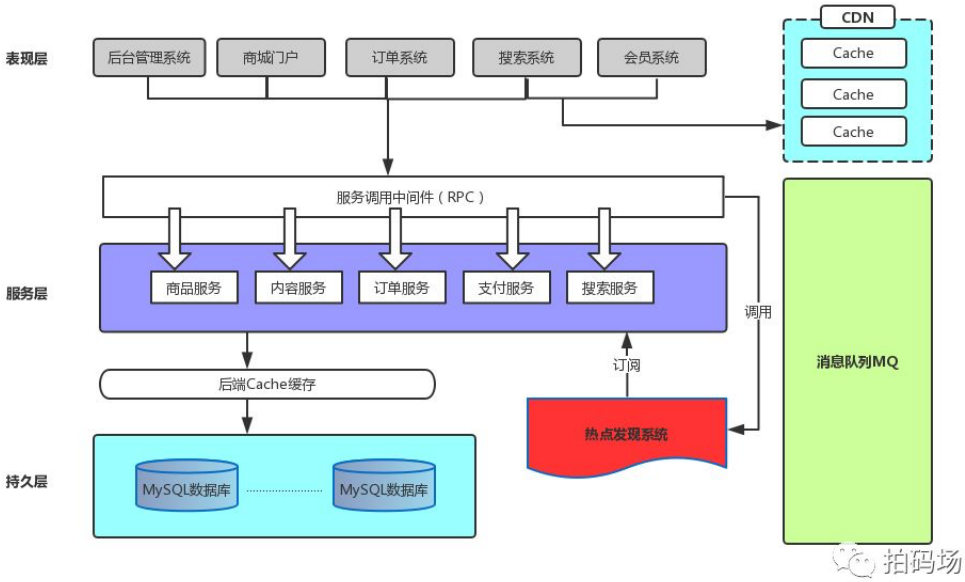
准：系统的准确性，针对秒杀系统可以理解为数据的一致性，有限的商品同一时刻被多倍的请求同时来扣减库存，在大并发更新的过程中需要保证数据的准确性，库存为100的秒杀商品，最终只能卖100个，控制资损是关键。
稳：系统的稳定性要求高，在大并发的情况下能保证系统的稳定运行，不出现系统运行故障。
一 高性能：
动静分离。把用户请求的数据分为"动态数据"和"静态数据"，不需要让用户每次都去刷新整个页面，用户只需要点击"一键刷宝"等功能按钮就可以看到最新的秒杀详情。针对静态数据我们可以做静态缓存，
把静态数据缓存在离用户最近的地方
　常用的静态缓存主要有三种，用户浏览器里、CDN上或者后台服务器的Cache里，该根据情况把数据缓存在离用户最近的地方

热点数据分离: 比如秒杀的商品列表、商品详情等，这些我们在秒杀之前都是可以提前知道，而且这部分数据是不会修改的，所以针对这部分数据我们可以直接缓存在服务器Cache里。

业务操作拆分：可以使用MQ消息中间件来完成后续的操作。同时为了保证订单和发货的一致性，可以做定时JOB来保证数据准确性

库存扣减实现：
　预扣库存，买家下单后，库存为其保留一定的时间（如 30 分钟），超过这个时间，库存将会自动释放，释放后其他买家就可以继续购买，在买家付款前，系统会校验该订单的库存是否还有
保留：如果没有保留，则再次尝试预扣；如果库存不足（也就是预扣失败）则不允许继续付，如果预扣成功，则完成付款并实际地减去库存。

流量削峰
　1.消息队列拦截请求。在实际操作中我们可以使用消息队列来缓冲瞬时流量，把同步的调用直接转成异步的推送，中间通过一个队列在一端承接瞬时的流量洪峰，在另一端平滑的将消息推送出去。当处理到某一个时刻（库存已经不足），后面的请求可以直接返回。
　2. 并发限制。我们可以通过增加流程的复杂度来降低并发数，比如秒杀系统，在下单之前我们让用户输入验证码或者答题之类的，这样就可以控制并发提交，保证在最终的下单那一步，并发量没有那么高。这种方式同时也可以防止秒杀器来刷商品，增加了系统的安全性 。
　3.用户限流。在某一时间段内只允许用户提交一次请求，比如可以采取IP或者登录手机号限流。或者针对用户请求的唯一标识uid之类的，在服务端控制层需要针对同一个访问uid，限制访问频率，放入redis中



热点发现系统的简单实现：

1. 构建一个异步系统，它可以收集交易链路上各个环节中的中间件产品的热点 Key如 Nginx、缓存、RPC 服务框架等这些中间件

2. 建立一个热点上报和可以按照需求订阅的热点服务的下发规范，把上游已经发现的热点透传给下游系统，提前做好保护，比如缓存或者隔离

3. 将上游系统收集的热点数据发送到热点服务台，然后下游系统（如交易系统）就会知道哪些商会被频繁调用，然后做热点区分

权限系统设计数据表