一、基础

1、基础

1.JDK和JRE

JDK：java开发工具包，提供了java的开发环境和运行环境。（编写需要）
JRE：java运行环境，提供了java开发所需要的环境。（运行需要）

JDK里包含了JRE，还有编译java源码的编译器javac，还包含了其他的调试和分析的工具。

2.==和equals

==对于基本数据来说比较就是值，对于引用类型比较的就是引用。

equals**默认比较就是引用**，但是很多类重写了equals方法，比如String、Integer：比较的就是值，所以一般情况基本上就是比较的值。

3.hashcode()和equals()

不是，两个对象的hashCode()相同，equals()不一定为true。

hashcode相同只能说明在hash表中处于相同的位置，但是一个位置通过链表就可以存放很多对象，hashcode只是用来方便查询去重的，减少了比较次数。hashcode只是位置的编号。所以前者对后者没有必然联系，但是反过来equals为true，hashcode也一定也是true。

如果要重写equals()方法时也一定要重写hashCode()方法。

4.final

修饰：

类：叫最终类，不能被继承。
方法：不能被继承重写。
变量：叫常量，声明时必须初始化，并且不能更改。

5.Math.round()

Math.round()：是在参数上加0.5然后进行下取整。

6.基础数据类型

byte,short,int,long,char,boolean,float,double

7.操作字符串的类

String，StringBuffer，StringBuilder

String声明的是不可变的对象，每次增删改的操作都需要生成新的String对象。
StringBuffer线程安全，性能略低。
StringBuilder线程不安全，性能略高。

8.String str="i"与 String str=new String("i")

内存分配方式不一样，第一个虚拟机将其分配到常量池，第二个分配到堆内存。

9.字符串反转

使用 StringBuilder 或 stringBuffer 它们中的 reverse() 方法。

10.String的常用方法

indexOf()：返回指定字符的索引。
charAt()：返回指定索引处的字符。
replace()：字符串替换。
trim()：去除字符串两端空白。
split()：分割字符串，返回一个分割后的字符串数组。
getBytes()：返回字符串的 byte 类型数组。
length()：返回字符串长度。
toLowerCase()：将字符串转成小写字母。
toUpperCase()：将字符串转成大写字符。
substring()：截取字符串。
equals()：字符串比较。

11.抽象类不一样非要有抽象方法，反之必然

12.普通类和抽象类

普通类不能还有抽象方法，抽象类可以包含抽象方法。
普通类可以实例化，抽象类不可以实例化。

13.抽象类和接口

构造函数：抽象类可以有，接口不能有
main方法：抽象类可以有，接口不能有
继承实现：类只能继承一个抽象类，但可以实现多个接口
关键字：extends，implements
访问修饰符：抽象类不能用private，static，native，synchronized，接口在抽象类的基础上不能用protected，接口默认是public。

14.IO流

功能：输入流，输出流
类型：字节流，字符流

15.BIO、AIO、NIO

BIO：传统IO，同步阻塞IO，一个连接一个线程，客户端请求会一直等待服务器执行完后才继续执行。
NIO：传统IO的升级，同步非阻塞IO，一个请求一个线程，连接请求会被注册到多路复用器上，轮询多路复用器当有IO请求时才使用一个线程，实现多路复用。
AIO：NIO的升级，异步非阻塞IO，基于事件和回调机制

一般来说I/O模型可以分为：**同步阻塞，同步非阻塞，异步阻塞，异步非阻塞IO**

同步阻塞IO：在此种方式下，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式！

同步非阻塞IO:在此种方式下，用户进程发起一个IO操作以后边可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。其中目前JAVA的NIO就属于同步非阻塞IO。

异步阻塞IO：此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄，从而提高系统的并发性！

异步非阻塞IO:在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，**不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。**目前Java中还没有支持此种IO模型。

16.Files的常用方法

Files.exists()：检测文件路径是否存在。
Files.createFile()：创建文件。
Files.createDirectory()：创建文件夹。
Files.delete()：删除一个文件或目录。
Files.copy()：复制文件。
Files.move()：移动文件。
Files.size()：查看文件个数。
Files.read()：读取文件。
Files.write()：写入文件。

17.枚举的好处

以这种方式定义的常量使代码更具可读性，允许进行编译时检查，预先记录可接受值的列表，并避免由于传入无效值而引起的意外行为

使用示例：单例模式、策略模式（同一接口方法的不同实现）

18.StringBuilder和StringBuffer

`String` 类中使用 final 关键字修饰字符数组来保存字符串，`private final char value[]`，所以 `String` 对象是不可变的。

而 `StringBuilder` 与 `StringBuffer` 都继承自 `AbstractStringBuilder` 类，在 `AbstractStringBuilder` 中也是使用字符数组保存字符串 `char[]value` 但是没有用 `final` 关键字修饰，所以这两种对象都是可变的。

> 在 Java 9 之后，String 、`StringBuilder` 与 `StringBuffer` 的实现改用 byte 数组存储字符串 `private final byte[] value`

2、面试题

1.Java8新特性

1、接口的默认方法
2、Lambda表达式语法，结合函数式接口来实现
3、函数式接口（接口只包含一个抽象方法，虚拟机会自动判断该接口为函数式接口，一般建议在接口上使用 `@FunctionalInterface` 注解进行声明）
4、方法和构造函数的引用

5、Optional容器
6、Stream流
7、日期函数

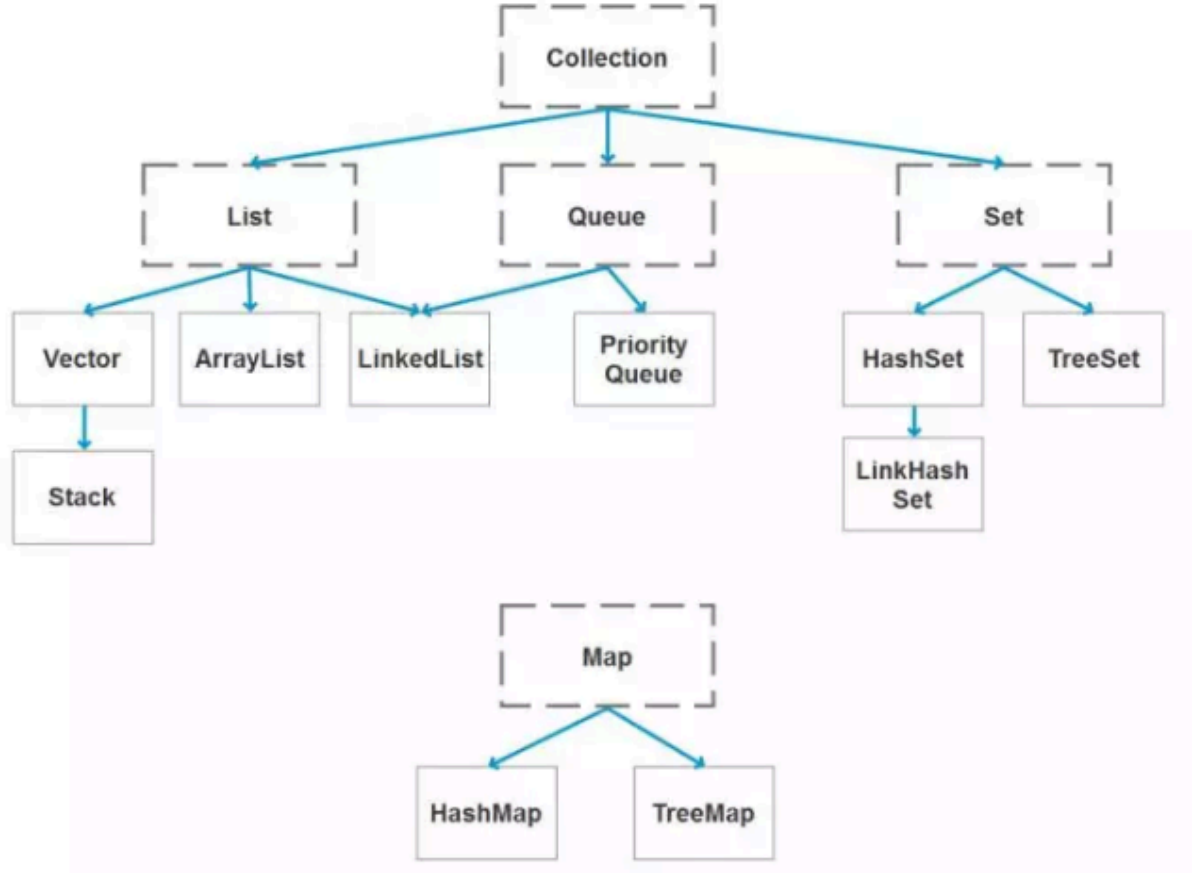
2.实现深拷贝的方式

1、实现Cloneable接口重写clone（）方法
2、序列化反序列化，使用SerializationUtils的clone(Object obj)方法
3、手动赋值BeanUtils.copyProperties()
4、fastJson相互转换

二、容器

1、基础

1.java容器有哪些



2.Collection和Collections

Collection是集合的顶级接口类，提供了对集合对象的基本操作的通用接口方法。
Collections是一个工具类，提供了一系列静态方法来方便操作集合中的数据。（排序，搜索）

3.List，Set，Map

比较	List	Set	Map
继承接口	Collection	Collection	
常见实现类	AbstractList(其常用子类有ArrayList、LinkedList、Vector)	AbstractSet(其常用子类有HashSet、LinkedHashSet、TreeSet)	HashMap、HashTable
常见方法	add( )、remove( )、clear( )、get( )、contains( )、size( )	add( )、remove( )、clear( )、contains( )、size( )	put( )、get( )、remove( )、clear( )、containsKey( )、containsValue( )、keySet( )、values( )、size( )
元素	可重复	不可重复(用 `equals()` 判断)	不可重复
顺序	有序	无序(实际上由HashCode决定)	
线程安全	Vector线程安全		Hashtable线程安全

4.HashMap和HashTable

HashMap是HashTable的轻量级实现：

1.HashMap去掉了HashTable的contains()方法，但是加上了containsValue()和containsKey()方法。
2.HashTable是同步的线程安全的，HashMap是异步的线程不安全的，效率上比HashTable要高。
3.HashMap允许空键和空值，HashTable不允许。

- `HashMap`： JDK1.8 之前 `HashMap` 由数组+链表组成的，数组是 `HashMap` 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间；当树的节点个数小于6时又会变回链表
- `LinkedHashMap`： `LinkedHashMap` 继承自 `HashMap`，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，`LinkedHashMap` 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：[《LinkedHashMap 源码详细分析（JDK1.8）》]

  > 有序可以按照两种方式排序：插入、读取；每次插入删除后都会调用一个函数来维护双向链表的维护；默认accessOrder=false是按照插入顺序排序

- `Hashtable`： 数组+链表组成的，数组是 `HashMap` 的主体，链表则是主要为了解决哈希冲突而存在的
- `TreeMap`： 红黑树（自平衡的排序二叉树）

5.HashMap和TreeMap

HashMap：底层是数组+链表的结构，不保证插入时的顺序，允许空键；适合快速插入，删除。
TreeMap：底层是红黑树的结构，插入删除操作都会根据键值重新排序，不允许空键，元素需要实现Comparable接口或实现Comparator接口；适合存入需要排序的数据，按顺序迭代的数据。
LinkedHashMap：在HashMap的基础上加入了双向链表来维持插入时的顺序。

> 红黑树特点：
>
> 1. 节点分为红色或者黑色；
> 2. 根节点必为黑色；

> 3. 叶子节点都为黑色，且为null；
> 4. 连接红色节点的两个子节点都为黑色（红黑树不会出现相邻的红色节点）；
> 5. 从任意节点出发，到其每个叶子节点的路径中包含相同数量的黑色节点；
> 6. 新加入到红黑树的节点为红色节点；
>
> 红黑树自平衡基本操作：
>
> 1. 变色：在不违反上述红黑树规则特点情况下，将红黑树某个node节点颜色由红变黑，或者由黑变红；
> 2. 左旋：逆时针旋转两个节点，让一个节点被其右子节点取代，而该节点成为右子节点的左子节点
> 3. 右旋：顺时针旋转两个节点，让一个节点被其左子节点取代，而该节点成为左子节点的右子节点

6.HashMap

是基于Map接口的非同步实现，底层是数组+链表的实现结构。当使用put存入元素时，会根据键的hashcode计算hash值，根据hash值得到元素的数组下标位置，如果该位置已经有元素了，再用equals()比较，如果键相同，就更新值，不同就将该位置上的元素以链表方式存放，后放入的放在链头。

jdk1.8之后，当链表节点数超过了8个就会转为红黑树的结构来提高查询效率。1.8之后的hash方法底层实现有改变，性能稍微提高了一点。1.7基于头插法（链表死循环），1.8基于尾插法。

> 注意：1.8后在调用HashMap构造函数时就会进行容量设定，而在1.8之前要等到第一次put操作时才会进行初始化

1.七种遍历方式

1. 使用迭代器（Iterator）EntrySet 的方式进行遍历；
2. 使用迭代器（Iterator）KeySet 的方式进行遍历；
3. 使用 For Each EntrySet 的方式进行遍历；
4. 使用 For Each KeySet 的方式进行遍历；
5. 使用 Lambda 表达式的方式进行遍历；
6. 使用 Streams API 单线程的方式进行遍历；
7. 使用 Streams API 多线程的方式进行遍历。

7.HashSet

HashSet的底层是由HashMap实现的，HashSet的值存放在HashMap的key上，HashMap的value统一为PRESENT。`private static final Object PRESENT = new Object();`

8.ArrayList和LinkedList

ArrayList：底层是数组，支持随机访问，插入删除慢。（可能还需要扩容，插入删除都要复制原来的数据）
LinkedList：底层是双向链表，不支持随机访问，插入删除快。

ArrayList扩容：扩容因子1.5，默认初始大小10
HashMap扩容：扩容因子0.75，默认大小16，每次扩容两倍（2的幂次方）

扩容因子：提高空间利用率和减少查询成本的折中，主要是泊松分布，0.75的话碰撞最小

9.数组和List的转换

List转数组：ArrayList.toArray();
数组转List：Arrays.asList();

10.ArrayList和Vector

ArrayList：更通用，非同步线程不安全，快，扩容0.5倍。
Vector：同步线程安全，慢，扩容1倍。

11.Queue的poll()和remove()

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

12.哪些集合类是安全的?

Vector：就比Arraylist多了个同步化机制（线程安全）。
Hashtable：就比Hashmap多了个线程安全。
ConcurrentHashMap:是一种高效但是线程安全的集合。
Stack：栈，也是线程安全的，继承于Vector。

13.迭代器是什么

迭代器是一种设计模式，迭代器对象就是一个对象，可以遍历序列中的元素而不需要了解序列的底层结构，通常被称为轻量级对象，因为创造它的代价很小。**注意：对java集合遍历的时候必须用迭代器。**（因为当size出现变化时，cursor并不一定能够得到同步，但是Iterator.remove（）就会cursor的状态）

14.Iterator怎么使用？特点?

Iterator 使用代码如下：

```
List<String> list = new ArrayList<>();
Iterator<String> it = list. iterator();
while(it. hasNext()){
  String obj = it. next();
  System.out.println(obj);
}
```

Iterator 的特点是更加安全，因为它可以确保在当前遍历的集合元素被更改的时候，就会抛出 ConcurrentModificationException 异常。

15.Iterator和ListIterator

- Iterator 可以遍历 Set 和 List 集合，而 ListIterator 只能遍历 List。
- Iterator 只能单向遍历，而 ListIterator 可以双向遍历（向前/后遍历）。
- ListIterator 从 Iterator 接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

16.怎样确保一个集合类不能被修改?

可以使用 Collections. unmodifiableCollection(Collection c) 方法来创建一个只读集合，这样改变集合的任何操作都会抛出 Java. lang. UnsupportedOperationException 异常。

示例代码如下：

```
List<String> list = new ArrayList<>();
list. add("x");
Collection<String> clist = Collections. unmodifiableCollection(list);
clist. add("y"); // 运行时此行报错
System.out.println(list. size());
```

17.ConcurrentHashMap

- **底层数据结构：**

  JDK1.7 的 `ConcurrentHashMap` 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟 `HashMap1.8` 的结构一样，数组+链表/红黑二叉树。

- **实现线程安全的方式：**

  ① **在 JDK1.7 的时候，** `ConcurrentHashMap` **（分段锁）** 对整个桶数组进行了分割分段（`Segment`），**数据结构Segment+Entry**，数组每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 **到了 JDK1.8 的时候已经摒弃了** `Segment` **的概念，而是直接用** `Node` **数组+链表+红黑树的数据结构来实现，并发控制使用** `synchronized` **和 CAS 来操作。 （JDK1.6 以后 对** `synchronized` **锁做了很多优化）** 整个看起来就像是优化过且线程安全的 `HashMap`，虽然在 JDK1.8 中还能看到 `Segment` 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② `Hashtable`**(同一把锁)** :使用 `synchronized` 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

1.具体线程安全的实现方式

-JDK1.8之前：`ConcurrentHashMap` **是由** `Segment` **数组结构和** `HashEntry` **数组结构组成。**Segment 实现了 `ReentrantLock`,所以 `Segment` 是一种可重入锁，扮演锁的角色。`HashEntry` 用于存储键值对数据。

-JDK1.8之后：`ConcurrentHashMap` 取消了 `Segment` 分段锁，采用 CAS 和 `synchronized` 来保证并发安全

18. ConcurrentSkipListMap

底层跳表实现，key是有序的，支持更高的并发

2、底层源码

1.ArrayList

底层基于动态Object数组，继承AbstractList，实现了RandomAccess、Clonable、Serializable接口

默认初始大小10，构造函数可以使用指定的集合容器

三、多线程

1、基础

1.并发和并行

1.并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。
2.并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。

并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

2.线程和进程

进程：**是程序运行和资源分配的最小单位**，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。

线程：是进程的一个实体，是**cpu调度和分派的基本单位**，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

3.守护线程

垃圾回收线程就是守护线程。当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。守护线程需要一直保持在后台运行来服务其他线程，并且由其创造的线程也是守护线程。

4.创建线程的方式

4种

1.继承Thread类：并重写run()，创建子类实例就是一个线程对象，start()来启动该线程
2.实现Runable接口：并重写run()，创建实例并作为Thread的target参数，该Thread才是真正的线程对象
3.实现Callable接口：并实现call()，并且有返回值，创建实例并且用FutureTask来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。使用FutureTask对象作为Thread对象的target创建并启动新线程。**可以调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。**
4.**使用ExecutorService、Callable、Future实现有返回结果的多线程。**

5.线程的状态

创建，就绪，运行，阻塞，死亡

- **创建状态。**在生成线程对象并没有调用该对象的**start**方法，这是线程处于创建状态。（start之前）
- **就绪状态。**当调用了线程对象的start方法之后，该线程就进入了就绪状态，**但是此时线程调度程序还没有把该线程设置为当前线程**，此时处于就绪状态。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。（start之后）
- **运行状态。**线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行run函数当中的代码。
- **阻塞状态。**线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。sleep，suspend，wait等方法都可以导致线程阻塞。
- **死亡状态。**如果一个线程的run方法执行结束或者**调用stop方法**后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪

6.Runable和Callable?

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；

Callable接口中的call()方法是有返回值的，并且可以抛出异常，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

1）excure()和submit()?

1. `execute()`方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；
2. `submit()`方法用于提交需要返回值的任务。线程池会返回一个`Future`类型的对象，通过这个`Future`对象可以判断任务是否执行成功，并且可以通过`Future`的`get()`方法来获取返回值，`get()`方法会阻塞当前线程直到任务完成，而使用`get（long timeout，TimeUnit unit）`方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

2）shutdown()和shutdownNow()?

- `shutdown（）`:关闭线程池，线程池的状态变为`SHUTDOWN`。线程池不再接受新任务了，但是队列里的任务得执行完毕。
- `shutdownNow（）`:关闭线程池，线程的状态变为`STOP`。线程池会终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List。

7.sleep()和wait()

sleep()：方法是**线程类（Thread）的静态方法**，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。因为sleep() 是static静态的方法，**他不能改变对象的机锁**，当一个synchronized块中调用了sleep() 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

wait()：wait()是**Object类的方法**，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，**可以通过notify，notifyAll方法来唤醒**等待的线程

8.notify()和notifyAll()

- 如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。
- 当有线程调用了对象的 notifyAll()方法（**唤醒所有** wait 线程）或 notify()方法（只**随机唤醒一个** wait 线程），被唤醒的的线程便会进入该对象的**锁池**中，锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争。
- 优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

9.线程的run()和start()

每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，**方法run()称为线程体。**通过调用Thread类的start()方法来启动一个线程。**run方法开始执行就转为执行状态。**

start()方法来启动一个线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码； 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行状态， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

run()方法是在本线程里的，只是线程里的一个函数,而不是多线程的。 如果直接调用run(),其实就相当于是调用了一个普通函数而已，直接待用run()方法必须等待run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是run()方法。

总结： 调用 `start()` 方法方可启动线程并使线程进入就绪状态，直接执行 `run()` 方法的话不会以多线程的方式执行。

10.创建线程池的方式

《阿里巴巴 Java 开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

> Executors 返回线程池对象的弊端如下：
>
> - **FixedThreadPool 和 SingleThreadExecutor**： 允许请求的队列长度为 Integer.MAX_VALUE，可能堆积大量的请求，从而导致 OOM。
> - **CachedThreadPool 和 ScheduledThreadPool**： 允许创建的线程数量为 Integer.MAX_VALUE，可能会创建大量线程，从而导致 OOM。

4种

①. **newFixedThreadPool(int nThreads)**

创建一个**固定长度**的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

②. **newCachedThreadPool()**

创建一个**可缓存**的线程池，如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

③. **newSingleThreadPool()**

这是一个**单线程**的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来**串行**执行。

④. **newScheduledThreadPool(int corePoolSize)**

创建了一个固定长度的线程池，而且以**延迟或定时**的方式来执行任务，类似于Timer。

11.线程池的状态

Running、ShutDown、Stop、Tidying、Terminated。

12.线程池的submit()和execute()

1.接收的参数不一样，submit()有三个构造函数
2.submit()有返回值，execute()没有返回值
3.submit()可以进行Exception处理

13.java程序如何保证多线程的安全

原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作（atomic,synchronized）；

可见性：一个线程对主内存的修改可以**及时**地被其他线程看到（synchronized,volatile）；**被volatile定义的变量其他线程可以立马知晓**（因为被volatile修饰的变量在做写的时候直接操作内存，其他线程在读取的时候也是读取最新的值）

有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序（happens-before原则）。jdk1.5之后的加强。

15.死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。

死锁的四个必要条件：

- **互斥条件**：进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源。（独占）
- **请求和保持条件**：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放。（就是占着不给别人）
- **不可剥夺条件**：是指进程已获得的资源，在未完成使用之前，**不可被剥夺**，只能在使用完后自己释放。
- **环路等待条件**：是指进程发生死锁后，若干进程之间形成一种头尾相接的**循环等待**资源关系。

预防死锁的策略：

1. 破坏占用且等待条件，可以一次性申请所有资源
2. 破坏不可抢占条件
3. 破坏循环等待条件，对资源进行排序，按序申请资源

16.ThreadLocal

是线程的局部变量。为每个线程提供一个独立的变量副本从而解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

1）应用场景

最常见的ThreadLocal使用场景为用来解决数据库连接、Session管理等。

get()方法是用来获取ThreadLocal在当前线程中保存的变量副本，set()用来设置当前线程中变量的副本，remove()用来移除当前线程中变量的副本，initialValue()是一个protected方法，一般是用来在使用时进行重写的，它是一个延迟加载方法。

2）内部原理

每个线程里维护了ThreadLocals来维护键值为ThreadLocal的对象。

最终的变量是放在了当前线程的 `ThreadLocalMap` 中，并不是存在 `ThreadLocal` 上，`ThreadLocal` 可以理解为只是 `ThreadLocalMap` 的封装，传递了变量值。 `ThrealLocal` 类中可以通过 `Thread.currentThread()` 获取到当前线程对象后，直接通过 `getMap(Thread t)` 可以访问到该线程的 `ThreadLocalMap` 对象。

每个`Thread`中都具备一个`ThreadLocalMap`，而`ThreadLocalMap`可以存储以`ThreadLocal`为key，Object 对象为 value的键值对。

> 比如我们在同一个线程中声明了两个`ThreadLocal`对象的话，会使用`Thread`内部都是使用仅有那个`ThreadLocalMap`存放数据的，`ThreadLocalMap`的 key 就是`ThreadLocal`对象，value 就是`ThreadLocal`对象调用`set`方法设置的值。

3）ThreadLocal内存泄露

`ThreadLocalMap`中使用的 key 为`ThreadLocal`的弱引用,而 value 是强引用。所以，如果`ThreadLocal`没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。这样一来，`ThreadLocalMap`中就会出现key为null的Entry。假如我们不做任何措施的话，value 永远无法被GC 回收，这个时候就可能会产生内存泄露。ThreadLocalMap实现中已经考虑了这种情况，在调用`set()`、`get()`、`remove()`方法的时候，会清理掉 key 为 null 的记录。使用完`ThreadLocal`方法后 最好手动调用`remove()`方法

4）static能不能修饰ThreadLocal?

一般会采用static修饰，既有好处，也有坏处。

可以避免重复创建TSO（thread specific object，即与线程相关的变量。）避免同一个线程创建访问到该类的不同实例。每个工作线程取得任务后都会重复创建一个ThreadLocal对象，导致不必要的内存开销。

坏处可能造成内存泄露。

17.synchronized底层原理

synchronized能同时保证可见性，原子性，有序性；

synchronized可以保证方法或者代码块在运行时，**同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。**

Java中**每一个对象都可以作为锁**，这是synchronized实现同步的基础：

1. 修饰实例方法，对当前实例对象this加锁
2. 修饰静态方法，对当前类的Class对象加锁
3. 修饰代码块，指定加锁对象，对给定对象加锁

> 1、尽量不要使用`synchronized(String a)`因为 JVM 中，字符串常量池具有缓存功能！
>
> 2、synchronized锁对象是存在哪里的呢？答案是存在锁对象的对象头Mark Word。Mark Word会随着程序的运行发生变化，不同的锁占用不同字节标识
>
> 3、String，Boolean在实现了都用了享元模式，即值在一定范围内，对象是同一个，不能用做加锁对象。

1）原理

jvm基于进入和退出Monitor对象来实现方法同步和代码块同步。是依赖于底层的操作系统的 Mutex Lock 来实现的，要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，状态转换成本很高。

1、同步语句块的实现使用的是`monitorenter`和`monitorexit`指令；
2、同步方法块使用的是判断`ACC_SYNCHRONIZED`访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用；
总结：**两者的本质都是对对象监视器 monitor 的获取。**

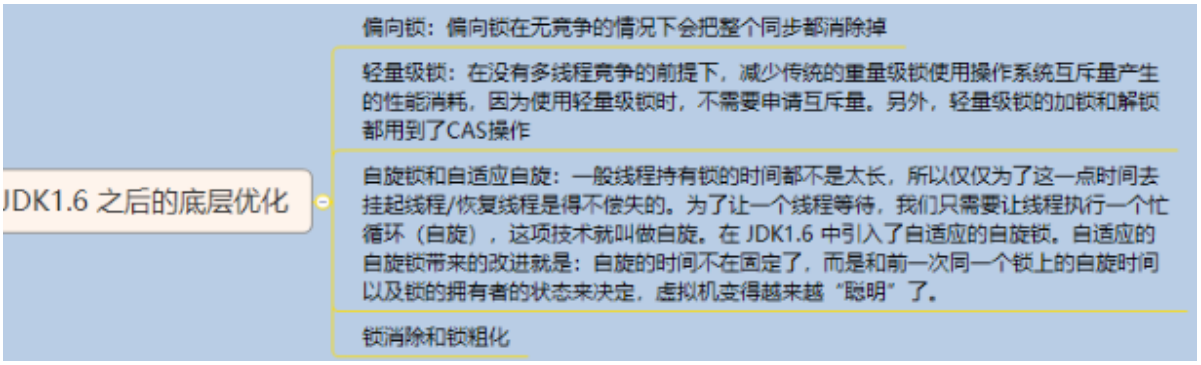
2）并发编程的三个重要特性

1. **原子性**：一个的操作或者多次操作，要么所有的操作全部都得到执行并且不会收到任何因素的干扰而中断，要么所有的操作都执行，要么都不执行。`synchronized`可以保证代码片段的原子性。
2. **可见性**：当一个变量对共享变量进行了修改，那么另外的线程都是立即可以看到修改后的最新值。`volatile`关键字可以保证共享变量的可见性。
3. **有序性**：代码在执行的过程中的先后顺序，Java 在编译器以及运行期间的优化，代码的执行顺序未必就是编写代码时候的顺序。`volatile`关键字可以禁止指令进行重排序优化。

3）1.6后的优化

JDK1.6 对锁的实现引入了大量的优化，如**偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化**等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。



关于这几种优化的详细信息可以查看笔主的这篇文章：https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/Multithread/synchronized.md

- 偏向锁原理：

  当线程1访问代码块并获取锁对象时，会在java对象头和栈帧中记录偏向的锁的threadID，因为**偏向锁不会主动释放锁**，因此以后线程1再次获取锁的时候，需要**比较当前线程的threadID和Java对象头中的threadID是否一致**，如果一致（还是线程1获取锁对象），则无需使用CAS来加锁、解锁；如果不一致（其他线程，如线程2要竞争锁对象，而偏向锁不会主动释放因此还是存储的线程1的threadID），那么**需要查看Java对象头中记录的线程1是否存活**，如果没有存活，那么锁对象被重置为无锁状态，其它线程（线程2）可以竞争将其设置为偏向锁；如果存活，那么立刻**查找该线程（线程1）的栈帧信息，如果还是需要继续持有这个锁对象**，那么暂停当前线程1，撤销偏向锁，升级为轻量级锁，如果线程1 不再使用该锁对象，那么将锁对象状态设为无锁状态，重新偏向新的线程。

- 轻量级锁原理：

  线程1获取轻量级锁时会先**把锁对象的对象头MarkWord复制一份到线程1的栈帧中创建的用于存储锁记录的空间**（称为DisplacedMarkWord），然后使**用CAS把对象头中的内容替换为线程1存储的锁记录（DisplacedMarkWord）的地址；**

  如果在线程1复制对象头的同时（在线程1CAS之前），线程2也准备获取锁，复制了对象头到线程2的锁记录空间中，但是在线程2CAS的时候，发现线程1已经把对象头换了，**线程2的CAS失败，那么线程2就尝试使用自旋锁来等待线程1释放锁。** 自旋锁简单来说就是让线程2在循环中不断CAS

  但是如果自旋的时间太长也不行，因为自旋是要消耗CPU的，因此自旋的次数是有限制的，比如10次或者100次，**如果自旋次数到了线程1还没有释放锁，或者线程1还在执行，线程2还在自旋等待，这时又有一个线程3过来竞争这个锁对象，那么这个时候轻量级锁就会膨胀为重量级锁。重量级锁把除了拥有锁的线程都阻塞，防止CPU空转。**

锁状态	优点	缺点	适用场景
偏向锁	加锁解锁无需额外的消耗，和非同步方法时间相差纳秒级别	如果竞争的线程多，那么会带来额外的锁撤销的消耗	基本没有线程竞争锁的同步场景
轻量级锁	竞争的线程不会阻塞，使用自旋，提高程序响应速度	如果一直不能获取锁，长时间的自旋会造成CPU消耗	适用于少量线程竞争锁对象，且线程持有锁的时间不长，追求响应速度的场景
重量级锁	线程竞争不适用CPU自旋，不会导致CPU空转消耗CPU资源	线程阻塞，响应时间长	很多线程竞争锁，且锁持有的时间长，追求吞吐量的场景

- 锁消除：消除锁是虚拟机另外一种锁的优化，这种优化更彻底，在JIT编译时，对运行上下文进行扫描，去除不可能存在竞争的锁
- 锁粗化：是虚拟机对另一种极端情况的优化处理，通过扩大锁的范围，避免反复加锁和释放锁，提高执行效率

总结：

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

4）单例模式

双重校验锁实现对象单例（线程安全）

```
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {
    }

    public synchronized static Singleton getUniqueInstance() {
       //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

5）构造函数不能用synchronized修饰

因为构造方法本身就属于线程安全的，不存在同步的构造方法一说。

18.synchronized和volatile

> 当前的 Java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。所以就需要volatile这个关键字。。。
>
> `volatile` 关键字 除了防止 JVM 的指令重排 ，还有一个重要的作用就是保证变量的可见性。

`synchronized` 关键字和 `volatile` 关键字是两个互补的存在，而不是对立的存在：

- **volatile关键字**是线程同步的**轻量级实现**，所以**volatile性能肯定比synchronized关键字要好。**但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。**synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些。**
- **多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞**
- **volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。**
- **volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。**

19.synchronized和Lock

- 首先synchronized是java内置关键字，**在jvm层面**；Lock是个java**类**；
- synchronized无法判断是否获取锁的状态；Lock可以判断是否获取到锁；
- synchronized会自动释放锁(a 线程执行完同步代码会释放锁 ； b 线程执行过程中发生异常会释放锁)，Lock需在finally中**手工释放**锁（unlock()方法释放锁），否则容易造成线程死锁；
- 用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去；而Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；
- synchronized的锁**可重入、不可中断、非公平**；而Lock锁**可重入、可中断、可公平**（两者皆可）；
- Lock锁适合**大量**同步的代码的同步问题；synchronized锁适合代码少量的同步问题。

20.synchronized和ReenTranLock?

① 两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

② synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

③ ReentrantLock 比 synchronized 增加了一些高级功能

相比synchronized，ReentrantLock增加了一些高级功能。主要来说主要有三点：**①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）**

- **ReentrantLock提供了一种能够中断等待锁的线程的机制**，通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
- **ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。** ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的`ReentrantLock(boolean fair)`构造方法来制定是否是公平的。
- synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition() 方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），**线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify()/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”**，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。

如果你想使用上述功能，那么选择ReentrantLock是一个不错的选择。

④ 性能已不是选择标准

1）synchronized为什么是非公平锁?

synchronized 的非公平其实在源码中应该有不少地方，因为设计者就没按公平锁来设计，核心有以下几个点：

1）当持有锁的线程释放锁时，该线程会执行以下两个重要操作：

1. 先将锁的持有者 owner 属性赋值为 null
2. 随机唤醒等待链表中的一个线程（假定继承者）。

在1和2之间，如果有其他线程刚好在尝试获取锁（例如自旋），则可以马上获取到锁。

2）当线程尝试获取锁失败，进入阻塞时，放入链表的顺序，和最终被唤醒的顺序是不一致的，也就是说你先进入链表，不代表你就会先被唤醒。

3）非公平锁可以提高执行性能，但是可能为产生线程饥饿

2）synchronized锁能降级吗?

可以的。

具体的触发时机：在全局安全点（safepoint）中，执行清理任务的时候会触发尝试降级锁。

当锁降级时，主要进行了以下操作：

1）恢复锁对象的 markword 对象头；

2）重置 ObjectMonitor，然后将该 ObjectMonitor 放入全局空闲列表，等待后续使用。

21.Automic（原子类）

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

并发包 `java.util.concurrent` 的原子类都存放在`java.util.concurrent.atomic`

1）原理

value是一个volatile变量，在内存中可见，任何线程都不允许对其进行拷贝，因此JVM可以保证任何时刻任何线程总能拿到该变量的最新值。此处value的值，可以在AtomicInteger类初始化的时候传入，也可以留空，留空则自动赋值为0。

CAS：先比较（和预期值比较）再交换

语言层面不做处理，我们将其交给硬件—CPU和内存，利用CPU的多处理能力，实现硬件层面的阻塞，再加上volatile变量的特性即可实现基于原子操作的线程安全。CAS只适合一些粒度比较小，型如计数器这样的需求用起来才有效，否则也不会有锁的存在了。

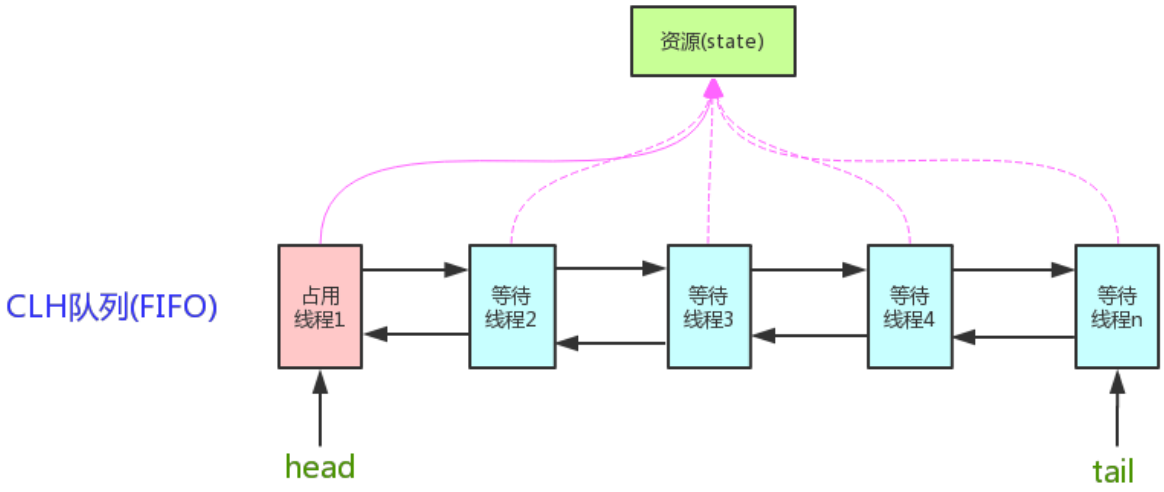
22.AQS?

AQS的全称为（AbstractQueuedSynchronizer），这个类在java.util.concurrent.locks包下面。

1. AQS 是一个锁框架，它定义了锁的实现机制，并开放出扩展的地方，让子类去实现，比如我们在 lock 的时候，AQS 开放出 state 字段，让子类可以根据 state 字段来决定是否能够获得锁，对于获取不到锁的线程 AQS 会自动进行管理，无需子类锁关心，这就是 lock 时锁的内部机制，封装的很好，又暴露出子类锁需要扩展的地方；
2. AQS 底层是由同步队列 + 条件队列联手组成，同步队列管理着获取不到锁的线程的排队和释放，条件队列是在一定场景下，对同步队列的补充，比如获得锁的线程从空队列中拿数据，肯定是拿不到数据的，这时候条件队列就会管理该线程，使该线程阻塞；
3. AQS 围绕两个队列，提供了四大场景，分别是：获得锁、释放锁、条件队列的阻塞，条件队列的唤醒，分别对应着 AQS 架构图中的四种颜色的线的走向。

AQS是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是基于AQS的。当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。



1）AQS组件

- **Semaphore(信号量)-允许多个线程同时访问：** 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。

  限流：

  ```
  #常用方法
  semaphore.acquire();
  semaphore.release();
  ```

- **CountDownLatch （倒计时器）：** CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。

  1、一个线程等待多个线程执行完毕
  2、多个线程等待一个线程的信号，然后开始执行工作

  ```
  #常用方法
  latch.countDown();
  latch.await();
  ```

- **CyclicBarrier(循环栅栏)：** 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）。CountDownLatch这个共享锁只能用一次，不能循环使用，而CyclicBarrier 可以循环使用。CyclicBarrier需要等到固定数量的线程都到达栅栏位置才能执行，作用对象是线程。

- **ReadWriteLock**

  特点：读读并行、读写互斥、写写互斥

  读是共享锁，写时互斥锁，用一个变量表示了两种锁的状态。在ReadWriteLock中是这样做的，state变量的高16位表示读锁的状态，低16位表示写锁的状态

  读取是获取共享锁，在获取读锁之前会先判断写锁是否被获取，如果写锁被当前线程获取或者没有写锁，则获取读锁成功，否则获取读锁失败（支持锁降级）

  写锁是获取独占锁，在获取之前会先判断读锁是否被获取，如果读锁已经被获取，则获取写锁失败。如果写锁没有被获取或者已经被当前线程获取，则获取写锁成功，否则获取写锁失败

2）AQS对资源的共享方式

AQS定义两种资源共享方式，独享锁和共享锁获取锁失败的节点会被放入队列。

- **Exclusive**（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：
- - 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
  - 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
- **Share**（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。Semaphore、CountDownLatch、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。共享锁多个线程都可以获取锁，当释放锁时需要唤醒header后面的所有有效节点

ReentrantReadWriteLock 可以看成是组合式，因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。

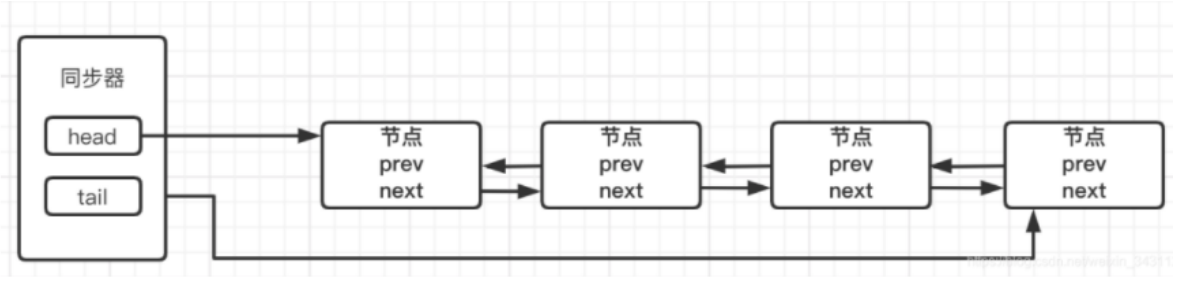
> AQS底层使用了模板方法模式：
>
> 1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）
> 2. 将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

3）AQS用到的设计模式

AQS同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）
2. 将AQS组合在自定义同步组件的实现中，并调用其模板方法，**自定义同步器时需要重写下面几个AQS提供的模板方法。**

4）AQS同步队列的数据结构



- 当前线程获取同步状态失败，同步器将当前线程机等待状态等信息构造成一个Node节点加入队列，放在队尾，同步器重新设置尾节点
- 加入队列后，会阻塞当前线程
- 同步状态被释放并且同步器重新设置首节点，同步器唤醒等待队列中第一个节点，让其再次获取同步状态

23.CAS?

即**compare and swap（比较与交换）**，是一种有名的**无锁算法**，是一种乐观锁。

CAS算法涉及到三个操作数：需要读写的内存值 V、进行比较的值 A、拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个**自旋操作**，即**不断的重试。**

1）乐观锁的缺点

1 ABA 问题

解决：给变量加个版本号

2 循环时间长CPU开销大

自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了`AtomicReference类`来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用`AtomicReference类`把多个共享变量合并成一个共享变量来操作。

2）CAS与synchronized的使用情景

> **简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）**

1. 对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
2. 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

24.手写生产者消费者模式?

25.怎么实现所有线程在等待某个事件的发生才会去执行?

方案一：**读写锁**

刚开始主线程先获取写锁，然后所有子线程获取读锁，然后等事件发生时主线程释放写锁；

方案二：**CountDownLatch**（倒计时器）

CountDownLatch初始值设为N，所有子线程调用await方法等待，等事件发生时调用countDown方法计数减为0；每个线程执行完任务CounDownLatch就减一；

方案三：**Semaphore**

Semaphore初始值设为N，刚开始主线程先调用acquire(N)申请N个信号量，其它线程调用acquire()阻塞等待，等事件发生时同时主线程释放N个信号量；

方案四：**CyclicBarrier**(循环栅栏)

26.Java的信号灯?

也就是Semaphore，可以允许多个线程同时访问；

Semaphore 有两种模式，公平模式（默认）和非公平模式。

acquire()方法设置允许线程同时操作的个数。

27.如何优雅停止线程

1、stop方法，已经废弃。正确应该使用两阶段终止模式，即一个线程发送终止指令，另一个线程接收指令并决定如何停止
2、使用标志位，标志位加volatile保证可见性
3、使用interrupt方法，将阻塞住的线程抛出异常，将WAITING和TIMED_WAITING状态的线程转为就绪态，进而由操作系统调度成运行状态，即可终止。

> 当线程处于运行状态：用自己定义的标志位来退出
> 当线程处于阻塞状态：用抛异常的方式来退出

28.线程阻塞三种情况

等待阻塞（Object.wait -> 等待队列）

同步阻塞（lock -> 锁池）

其他阻塞（Thread.sleep/join）

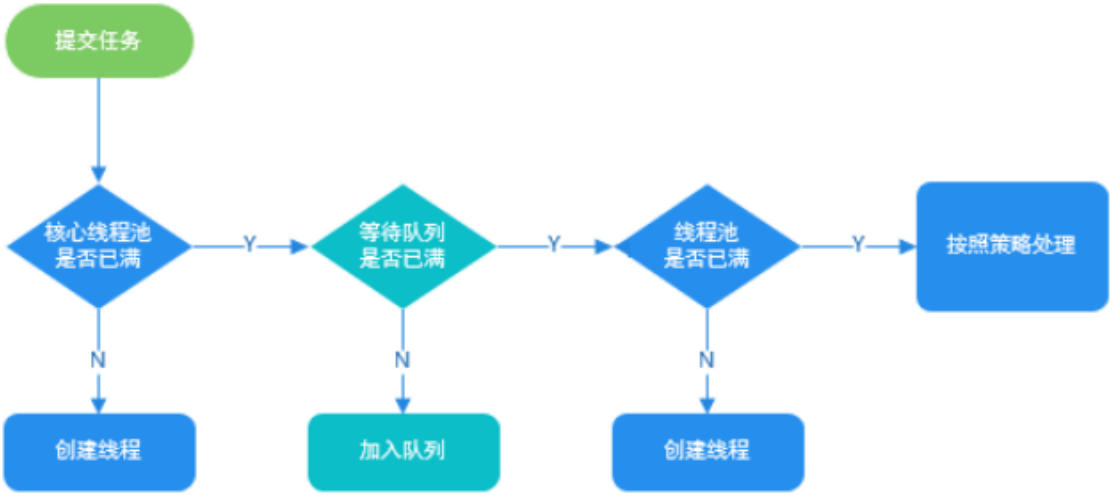
2.面试题

1.Java内存模型

在 JDK1.2 之前，Java 的内存模型实现总是从**主存**（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。

要解决这个问题，就需要把变量声明为`volatile`，这就指示 JVM，这个变量是共享且不稳定的，每次使用它都到主存中进行读取。

2.线程池的使用过程



3.TheadPoolExecutor拒绝策略（中丢老调）

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任务时，`ThreadPoolTaskExecutor` 定义一些策略:

- `ThreadPoolExecutor.AbortPolicy`：抛出 `RejectedExecutionException`来拒绝新任务的处理。
- `ThreadPoolExecutor.CallerRunsPolicy`：调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。
- `ThreadPoolExecutor.DiscardPolicy`： 不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy`： 此策略将丢弃最早的未处理的任务请求。

举个例子：Spring 通过 `ThreadPoolTaskExecutor` 或者我们直接通过 `ThreadPoolExecutor` 的构造函数创建线程池的时候，当我们不指定 `RejectedExecutionHandler` 饱和策略的话来配置线程池的时候默认使用的是 `ThreadPoolExecutor.AbortPolicy`。在默认情况下，`ThreadPoolExecutor` 将抛出 `RejectedExecutionException` 来拒绝新来的任务 ，这代表你将丢失对这个任务的处理。对于可伸缩的应用程序，建议使用 `ThreadPoolExecutor.CallerRunsPolicy`。当最大池被填满时，此策略为我们提供可伸缩队列。（这个直接查看 `ThreadPoolExecutor` 的构造函数源码就可以看出，比较简单的原因，这里就不贴代码了）

4.创建线程池的参数

- `corePoolSize`：核心线程数线程数定义了最小可以同时运行的线程数量。
- `maximumPoolSize`：当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- `workQueue`：当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

`ThreadPoolExecutor` 其他常见参数:

1. `keepAliveTime`:当线程池中的线程数量大于 `corePoolSize` 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 `keepAliveTime`才会被回收销毁；
2. `unit` ： `keepAliveTime` 参数的时间单位。
3. `threadFactory` :executor 创建新线程的时候会用到。
4. `handler` :饱和策略。关于饱和策略下面单独介绍一下。

5.Automic原子类

基本类型

使用原子的方式更新基本类型

- `AtomicInteger`：整形原子类
- `AtomicLong`：长整型原子类
- `AtomicBoolean`：布尔型原子类

数组类型

使用原子的方式更新数组里的某个元素

- `AtomicIntegerArray`：整形数组原子类
- `AtomicLongArray`：长整形数组原子类
- `AtomicReferenceArray`：引用类型数组原子类

引用类型

- `AtomicReference`：引用类型原子类
- `AtomicStampedReference`：原子更新带有版本号的引用类型。该类将整数值与引用关联起来进行原子更新时可能出现的 ABA 问题。
- `AtomicMarkableReference`：原子更新带有标记位的引用类型

对象的属性修改类型

- `AtomicIntegerFieldUpdater`：原子更新整形字段的更新器
- `AtomicLongFieldUpdater`：原子更新长整形字段的更新器
- `AtomicReferenceFieldUpdater`：原子更新引用类型字段的更新器

6.AtomicInteger的原理?

线程安全的原理如下：

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

> native方法：java代码调用非java代码

7.InterruptedException

当线程处于WAITING和TIMED_WAITING状态时，如果调用interrupt方法会抛出InterruptedException，让线程处于就绪状态；

当线程处于运行状态时，如果调用interrupt方法只是在当前线程打了一个中断的标记，中断的逻辑需要我们自己去实现；

> Thread类提供了如下2个方法来判断线程是否是中断状态
>
> Thread#isInterrupted：测试线程是否是中断状态，执行后不更改中断状态
> Thread#interrupted：测试线程是否是中断状态，执行后将中断标志更改为false

四、其他

1.java反射

1.反射概念

反射是指程序在运行过程中可以动态的访问，检测和修改它本身状态或行为的一种能力。在Java运行时环境中，对于任意一个类，可以通过反射知道这个类有哪些属性和方法；对于任意一个对象，可以通过反射调用它的任意一个方法。

功能：

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法。

2.序列化

序列化的目的就是为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存object states，但是Java给你提供一种应该比你自己好的保存对象状态的机制，那就是序列化。

什么情况下需要序列化：

a）当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；
b）当你想用套接字在网络上传送对象的时候；
c）当你想通过RMI传输对象的时候；

3.动态代理

当想要给实现了某个接口的类中的方法，**加一些额外的处理。**比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新类。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。**是用来拓展功能的**

动态代理的应用：

- Spring的AOP
- 加事务
- 加权限
- 加日志

如何实现

首先必须**1.定义一个接口**，还要有**2.一个InvocationHandler处理类**(将实现接口的类的对象传递给它)。再有一个工具类**3.Proxy**(习惯性将其称为代理类，因为调用他的newInstance()可以产生代理对象,其实他只是一个产生代理对象的**工具类**）。利用到InvocationHandler，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回。

2.对象拷贝

1.为什么需要对象克隆

想对一个对象进行处理，又想保留原有的数据进行接下来的操作，就需要克隆了，Java语言中克隆针对的是类的实例。

2.实现

1). 实现Cloneable接口并重写Object类中的clone()方法；浅拷贝，不拷贝对象内部的引用

2). 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的**深度克隆**

3.异常

五、JavaWeb

1、基础

1.JSP和Servlet的区别？各自特点?

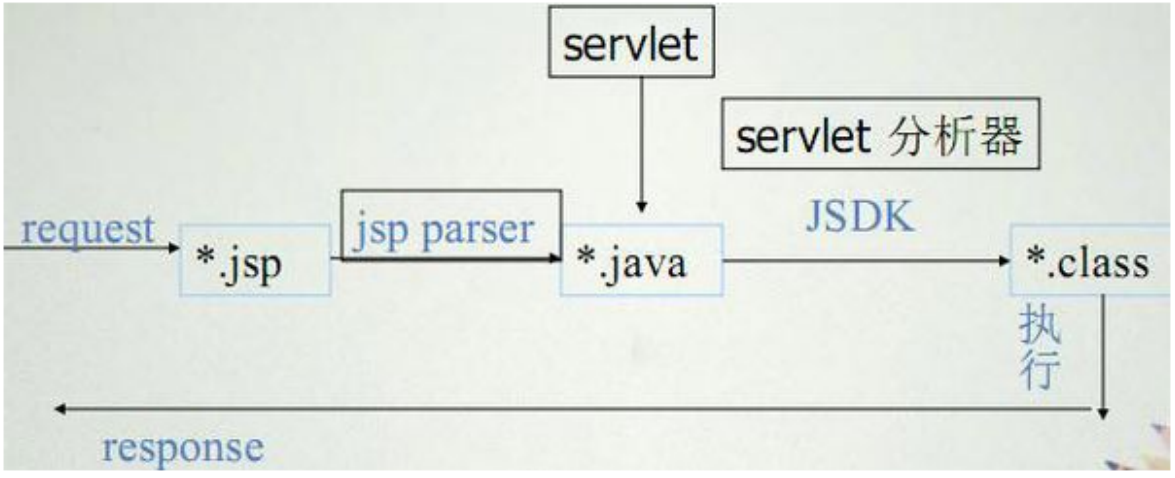
总的可以理解为：jsp就是在html里面写java代码，servlet就是在java里面写html代码；其实jsp经过容器解释之后就是servlet。具体不同描述如下：

jsp和servlet的不同之处

- Servlet在Java代码中通过HttpServletResponse对象动态输出HTML内容
- JSP在静态HTML内容中嵌入Java代码，Java代码被动态执行后生成HTML内

jsp和servlet各自的特点

- Servlet能够很好地组织业务逻辑代码，但是在Java源文件中通过字符串拼接的方式生成动态HTML内容会导致代码维护困难、可读性差
- JSP虽然规避了Servlet在生成HTML内容方面的劣势，但是在HTML中混入大量、复杂的业务逻辑同样也是不可取的



2.JSP的9大内置对象？作用?

- request：封装客户端的请求，其中包含来自 get 或 post 请求的参数；
- response：封装服务器对客户端的响应；
- pageContext：通过该对象可以获取其他对象；
- session：封装用户会话的对象；
- application：封装服务器运行环境的对象；
- out：输出服务器响应的输出流对象；
- config：Web 应用的配置对象；
- page：JSP 页面本身（相当于 Java 程序中的 this）；
- exception：封装页面抛出异常的对象。

3.JSP的4种作用域?

- page：代表与一个页面相关的对象和属性。
- request：代表与客户端发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个 Web 组件；需要在页面显示的临时数据可以置于此作用域。
- session：代表与某个用户与服务器建立的一次会话相关的对象和属性。跟某个用户相关的数据应该放在用户自己的 session 中。
- application：代表与整个 Web 应用程序相关的对象和属性，它实质上是跨越整个 Web 应用程序，包括多个页面、请求和会话的一个全局作用域。

4.session和cookie区别?

- 存储位置不同：session 存储在服务器端；cookie 存储在浏览器端。
- 安全性不同：cookie 安全性一般，在浏览器存储，可以被伪造和修改。
- 容量和个数限制：cookie 有容量限制，每个站点下的 cookie 也有个数限制。
- 存储的多样性：session 可以存储在 Redis 中、数据库中、应用程序中；而 cookie 只能存储在浏览器中。

5.session的工作原理?

session 的工作原理是客户端登录完成之后，服务器会创建对应的 session，session 创建完之后，会把 session 的 id 发送给客户端，客户端再存储到浏览器中。这样客户端每次访问服务器时，都会带着 sessionid，服务器拿到 sessionid 之后，在内存找到与之对应的 session 这样就可以正常工作了。

6.XSS攻击?

XSS 攻击：即跨站脚本攻击，它是 Web 程序中常见的漏洞。原理是攻击者往 Web 页面里插入恶意的脚本代码（css 代码、Javascript 代码等），当用户浏览该页面时，嵌入其中的脚本代码会被执行，从而达到恶意攻击用户的目的，如盗取用户 cookie、破坏页面结构、重定向到其他网站等。

预防 XSS 的核心是必须对输入的数据做过滤处理。

7.CSRF攻击?

CSRF：Cross-Site Request Forgery（中文：跨站请求伪造），可以理解为攻击者盗用了你的身份，以你的名义发送恶意请求，比如：以你名义发送邮件、发消息、购买商品，虚拟货币转账等。

避免方法：

CSRF 漏洞进行检测的工具，如 CSRFTester、CSRF Request Builder...、验证 HTTP Referer 字段、添加并验证 token、添加自定义 http 请求头敏感操作添加验证码、使用 post 请求

8.SQL注入?

避免：

- 使用预处理 PreparedStatement。
- 使用正则表达式过滤掉字符中的特殊字符。

9.SpringMVC和Struts区别?

- 拦截级别：struts2 是类级别的拦截；spring mvc 是方法级别的拦截。
- 数据独立性：spring mvc 的方法之间基本上独立的，独享 request 和 response 数据，请求数据通过参数获取，处理结果通过 ModelMap 交回给框架，方法之间不共享变量；而 struts2 虽然方法之间也是独立的，但其所有 action 变量是共享的，这不会影响程序运行，却给我们编码和读程序时带来了一定的麻烦。
- 拦截机制：struts2 有以自己的 interceptor 机制，spring mvc 用的是独立的 aop 方式，这样导致 struts2 的配置文件量比 spring mvc 大。
- 对 ajax 的支持：spring mvc 集成了ajax，所有 ajax 使用很方便，只需要一个注解 @ResponseBody 就可以实现了；而 struts2 一般需要安装插件或者自己写代码才行。

10.跨域

由于浏览器的同源策略限制，当url的协议、域名、端口三者任意一个不一样时即为跨域。

解决：

- 前端：

  1、设置Cookie的domain
  2、跨文档通信，向多个页面发送请求
  3、ajax的异步回调

- 后端：

  配置Cors

六、计算机网络

1、面试题

1.三次握手

- 客户端–发送带有 SYN 标志的数据包–一次握手–服务端
- 服务端–发送带有 SYN/ACK 标志的数据包–二次握手–客户端
- 客户端–发送带有带有 ACK 标志的数据包–三次握手–服务端

2.为什么要有三次握手

第一次握手：Client 什么都不能确认；Server 确认了对方发送正常，自己接收正常

第二次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：对方发送正常，自己接收正常

第三次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：自己发送、接收正常，对方发送、接收正常

所以三次握手就能确认双发收发功能都正常，缺一不可。

3.第2次握手传回了ACK，为什么还要传回SYN?

接收端传回发送端所发送的ACK是为了告诉客户端，我接收到的信息确实就是你所发送的信号了，这表明从客户端到服务端的通信是正常的。而回传SYN则是为了建立并确认从服务端到客户端的通信。”

> SYN 同步序列编号(Synchronize Sequence Numbers) 是 TCP/IP 建立连接时使用的握手信号。在客户机和服务器之间建立正常的 TCP 网络连接时，客户机首先发出一个 SYN 消息，服务器使用 SYN-ACK 应答表示接收到了这个消息，最后客户机再以 ACK(Acknowledgement）消息响应。这样在客户机和服务器之间才能建立起可靠的 TCP 连接，数据才可以在客户机和服务器之间传递。

4.为什么要四次挥手?

- 客户端-发送一个 FIN，用来关闭客户端到服务器的数据传送
- 服务器-收到这个 FIN，它发回一 个 ACK，确认序号为收到的序号加1 。和 SYN 一样，一个 FIN 将占用一个序号
- 服务器-关闭与客户端的连接，发送一个FIN给客户端
- 客户端-发回 ACK 报文确认，并将确认序号设置为收到序号加1

任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了TCP连接。

5.TCP和UDP

是传输层的协议。。。

UDP 在传送数据之前不需要先建立连接，远地主机在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 确是一种最有效的工作方式（一般用于即时通信），比如： QQ 语音、 QQ 视频 、直播等等

TCP 提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。 TCP 不提供广播或多播服务。由于 TCP 要提供可靠的，面向连接的传输服务（TCP的可靠体现在TCP在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有**确认、窗口、重传、拥塞控制**机制，在数据传完后，还会断开连接用来节约系统资源），这一难以避免增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理机资源。TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。

6.TCP如何保证消息可靠传输?

1. 应用数据被分割成 TCP 认为最适合发送的数据块。
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
3. **校验和：** TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. TCP 的接收端会丢弃重复的数据。
5. **流量控制：** TCP 连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。 （TCP 利用滑动窗口实现流量控制）
6. **拥塞控制：** 当网络拥塞时，减少数据的发送。
7. **ARQ协议（自动重传）：** 也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
8. **超时重传：** 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

7.百度输入url到显示主页的过程?

总体来说分为以下几个过程:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

8.状态码

	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

9.HTTP1.0和1.1区别?

1. **长连接**：**在HTTP/1.0中，默认使用的是短连接**，也就是说每次请求都要重新建立一次连接。HTTP是基于TCP/IP协议的,每一次建立或者断开连接都需要三次握手四次挥手的开销，如果每次请求都要这样的话，开销会比较大。因此最好能维持一个长连接，可以用个长连接来发多个请求。**HTTP 1.1起，默认使用长连接** ,默认开启Connection： keep-alive。 **HTTP/1.1的持续连接有非流水线方式和流水线方式** 。流水线方式是客户在收到HTTP的响应报文之前就能接着发送新的请求报文。与之相对应的非流水线方式是客户在收到前一个响应后才能发送下一个请求。
2. **错误状态响应码** :在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
3. **缓存处理** :在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
4. **带宽优化及网络连接的使用** :HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

10.http和https?

1. **端口**： HTTP的URL由“http://”起始且默认使用端口80，而HTTPS的URL由“https://”起始且默认使用端口443。
2. 安全性和资源消耗：

   HTTP协议运行在TCP之上，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。HTTPS是运行在SSL/TLS之上的HTTP协议，SSL/TLS 运行在TCP之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。所以说，HTTP安全性没有 HTTPS高，但是 HTTPS 比HTTP耗费更多服务器资源。

   - 对称加密：密钥只有一个，加密解密为同一个密码，且加解密速度快，典型的对称加密算法有DES、AES等；
   - 非对称加密：密钥成对出现（且根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），相对对称加密速度较慢，典型的非对称加密算法有RSA、DSA等。

11.粘包

概念：TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

产生原因：

发送端原因：如果发送的网络数据包太小，那么他本身会启用Nagle算法（可配置是否启用）对较小的数据包进行合并（基于此，TCP的网络延迟要UDP的高些）然后再发送（超时或者包大小足够）。那么这样的话，服务器在接收到消息（数据流）的时候就无法区分哪些数据包是客户端自己分开发送的，这样产生了粘包.

接收端原因：服务器在接收到数据后，放到缓冲区中，如果消息没有被及时从缓存区取走，下次在取数据的时候可能就会出现一次取出多个数据包的情况，造成粘包现象。

什么时候需要处理粘包：

1、如果是短连接，无需考虑粘包
2、发送大文件，比如同一个数据的不同部分
3、长连接时需要处理粘包

如何处理粘包：

发送端：关闭Nagle算法来解决，使用TCP_NODELAY选项来关闭Nagle算法

接收端：TCP传输层没有处理粘包的机制，只能在应用层进行处理，应用层循环读取每个数据。使用两种途径：格式规范化数据、设置发送长度（参考格式TLV）

12.各层协议及作用

计算机网络体系可以大致分为一下三种，OSI七层模型、TCP/IP四层模型和五层模型。

- OSI七层模型：大而全，但是比较复杂、而且是先有了理论模型，没有实际应用。
- TCP/IP四层模型：是由实际应用发展总结出来的，从实质上讲，TCP/IP只有最上面三层，最下面一层没有什么具体内容，TCP/IP参考模型没有真正描述这一层的实现。
- 五层模型：五层模型只出现在计算机网络教学过程中，这是对七层模型和四层模型的一个折中，既简洁又能将概念阐述清楚。

13.TCP三次握手

三次握手机制：

- 第一次握手：客户端请求建立连接，向服务端发送一个**同步报文**（SYN=1），同时选择一个随机数 seq = x 作为**初始序列号**，并进入SYN_SENT状态，等待服务器确认。
- 第二次握手：：服务端收到连接请求报文后，如果同意建立连接，则向客户端发送**同步确认报文**（SYN=1，ACK=1），确认号为 ack = x + 1，同时选择一个随机数 seq = y 作为初始序列号，此时服务器进入SYN_RECV状态。
- 第三次握手：客户端收到服务端的确认后，向服务端发送一个**确认报文**（ACK=1），确认号为 ack = y + 1，序列号为 seq = x + 1，客户端和服务器进入ESTABLISHED状态，完成三次握手。

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。

1）为什么需要三次握手，不是两次?

1、防止已过期的连接请求报文突然又传送到服务器，因而产生错误和资源浪费。
2、三次握手才能让双方均确认自己和对方的发送和接收能力都正常。
3、告知对方自己的初始序号值，并确认收到对方的初始序号值。

2）为什么三次握手，不是四次?

- 第一次握手：服务端确认“自己收、客户端发”报文功能正常。
- 第二次握手：客户端确认“自己发、自己收、服务端收、客户端发”报文功能正常，客户端认为连接已建立。
- 第三次握手：服务端确认“自己发、客户端收”报文功能正常，此时双方均建立连接，可以正常通信。

14.SYN洪泛攻击？如何防范？

SYN洪泛攻击属于 DOS 攻击的一种，它利用 TCP 协议缺陷，通过发送大量的半连接请求，耗费 CPU 和内存资源。

原理：

- 在三次握手过程中，服务器发送 `[SYN/ACK]` 包（第二个包）之后、收到客户端的 `[ACK]` 包（第三个包）之前的 TCP 连接称为半连接（half-open connect），此时服务器处于 `SYN_RECV`（等待客户端响应）状态。如果接收到客户端的 `[ACK]`，则 TCP 连接成功，如果未接受到，则会**不断重发请求**直至成功。
- SYN 攻击的攻击者在短时间内**伪造大量不存在的 IP 地址**，向服务器不断地发送 `[SYN]` 包，服务器回复 `[SYN/ACK]` 包，并等待客户的确认。由于源地址是不存在的，服务器需要不断的重发直至超时。
- 这些伪造的 `[SYN]` 包将长时间占用未连接队列，影响了正常的 SYN，导致目标系统运行缓慢、网络堵塞甚至系统瘫痪。

检测：当在服务器上看到大量的半连接状态时，特别是源 IP 地址是随机的，基本上可以断定这是一次 SYN 攻击。

防范：

- 通过防火墙、路由器等过滤网关防护。
- 通过加固 TCP/IP 协议栈防范，如增加最大半连接数，缩短超时时间。
- SYN cookies技术。SYN Cookies 是对 TCP 服务器端的三次握手做一些修改，专门用来防范 SYN 洪泛攻击的一种手段。

15.三次握手阶段，最后一次ACK包丢失，会发生甚麽?

服务端：

- 第三次的ACK在网络中丢失，那么服务端该TCP连接的状态为SYN_RECV,并且会根据 TCP的超时重传机制，会等待3秒、6秒、12秒后重新发送SYN+ACK包，以便客户端重新发送ACK包。
- 如果重发指定次数之后，仍然未收到 客户端的ACK应答，那么一段时间后，服务端自动关闭这个连接。

客户端：

客户端认为这个连接已经建立，如果客户端向服务端发送数据，服务端将以RST包（Reset，标示复位，用于异常的关闭连接）响应。此时，客户端知道第三次握手失败。

16.TCP四次挥手

- 第一次挥手：客户端向服务端发送连接释放报文（FIN=1，ACK=1），主动关闭连接，同时等待服务端的确认。
  - 序列号 seq = u，即客户端上次发送的报文的最后一个字节的序号 + 1
  - 确认号 ack = k, 即服务端上次发送的报文的最后一个字节的序号 + 1
- 第二次挥手：服务端收到连接释放报文后，立即发出**确认报文**（ACK=1），序列号 seq = k，确认号 ack = u + 1。

  这时 TCP 连接处于半关闭状态，即客户端到服务端的连接已经释放了，但是服务端到客户端的连接还未释放。这表示客户端已经没有数据发送了，但是服务端可能还要给客户端发送数据。

- 第三次挥手：服务端向客户端发送连接释放报文（FIN=1，ACK=1），主动关闭连接，同时等待 A 的确认。
    - 序列号 seq = w，即服务端上次发送的报文的最后一个字节的序号 + 1。
    - 确认号 ack = u + 1，与第二次挥手相同，因为这段时间客户端没有发送数据
- 第四次挥手：客户端收到服务端的连接释放报文后，立即发出**确认报文**（ACK=1），序列号 seq = u + 1，确认号为 ack = w + 1。

  此时，客户端就进入了 `TIME-WAIT` 状态。注意此时客户端到 TCP 连接还没有释放，必须经过 2*MSL（最长报文段寿命）的时间后，才进入 `CLOSED` 状态。而服务端只要收到客户端发出的确认，就立即进入 `CLOSED` 状态。可以看到，服务端结束 TCP 连接的时间要比客户端早一些。

17.为什么连接是三次握手，关闭是四次握手?

服务器在收到客户端的 FIN 报文段后，可能还有一些数据要传输，所以不能马上关闭连接，但是会做出应答，返回 ACK 报文段.

接下来可能会继续发送数据，在数据发送完后，服务器会向客户单发送 FIN 报文，表示数据已经发送完毕，请求关闭连接。服务器的**ACK和FIN一般都会分开发送**，从而导致多了一次，因此一共需要四次挥手。

18.为什么客户端的TIME-WAIT状态必须等待2MSL?

主要有两个原因：

1. 确保 ACK 报文能够到达服务端，从而使服务端正常关闭连接。

   第四次挥手时，客户端第四次挥手的 ACK 报文不一定会到达服务端。服务端会超时重传 FIN/ACK 报文，此时如果客户端已经断开了连接，那么就无法响应服务端的二次请求，这样服务端迟迟收不到 FIN/ACK 报文的确认，就无法正常断开连接。

   MSL 是报文段在网络上存活的最长时间。客户端等待 2MSL 时间，即「客户端 ACK 报文 1MSL 超时 + 服务端 FIN 报文 1MSL 传输」，就能够收到服务端重传的 FIN/ACK 报文，然后客户端重传一次 ACK 报文，并重新启动 2MSL 计时器。如此保证服务端能够正常关闭。

   如果服务端重发的 FIN 没有成功地在 2MSL 时间里传给客户端，服务端则会继续超时重试直到断开连接。

2. 防止已失效的连接请求报文段出现在之后的连接中。

   TCP 要求在 2MSL 内不使用相同的序列号。客户端在发送完最后一个 ACK 报文段后，再经过时间 2MSL，就可以保证本连接持续的时间内产生的所有报文段都从网络中消失。这样就可以使下一个连接中不会出现这种旧的连接请求报文段。或者即使收到这些过时的报文，也可以不处理它。

> 是客户端和服务端超时的总和

19.如果已经建立了连接，但是客户端出现故障了怎么办?

或者说，如果三次握手阶段、四次挥手阶段的包丢失了怎么办？如“服务端重发 FIN丢失”的问题。

简而言之，通过**定时器 + 超时重试机制**，尝试获取确认，直到最后会自动断开连接。

具体而言，TCP 设有一个保活计时器。服务器每收到一次客户端的数据，都会重新复位这个计时器，以 Linux服务器为例，时间通常是设置为 2 小时。若 2 小时还没有收到客户端的任何数据，服务器就开始重试：每隔 75 秒（默认）发送一个探测报文段，若一共发送 10 个探测报文后客户端依然没有回应，那么服务器就认为连接已经断开了。

> 附：Linux服务器系统内核参数配置

1. tcp_keepalive_time，在TCP保活打开的情况下，最后一次数据交换到TCP发送第一个保活探测包的间隔，即允许的持续空闲时长，或者说每次正常发送心跳的周期，默认值为7200s（2h）。
2. tcp_keepalive_probes 在tcp_keepalive_time之后，没有接收到对方确认，**继续发送保活探测包次数，默认值为9（次）。**
3. tcp_keepalive_intvl，在tcp_keepalive_time之后，没有接收到对方确认，继续发送保活探测包的发送频率，默认值为75s。 发送频率tcp_keepalive_intvl乘以发送次数tcp_keepalive_probes，就得到了从开始探测到放弃探测确定连接断开的时间； 举例：若设置，服务器在客户端连接空闲的时候，每90秒发送一次保活探测包到客户端，若没有及时收到客户端的TCP Keepalive ACK确认，将继续等待15秒*2=30秒。总之可以在90s+30s=120秒（两分钟）时间内可检测到连接失效与否。

20.TIME-WAIT状态过多后果？怎么处理？

从服务器来讲，短时间内关闭了大量的Client连接，就会造成服务器上出现大量的TIME_WAIT连接，严重消耗着服务器的资源，此时部分客户端就会显示连接不上。

从客户端来讲，客户端TIME_WAIT过多，就会导致端口资源被占用，因为端口就65536个，被占满就会导致无法创建新的连接。

解决办法：

- 服务器可以设置 SO_REUSEADDR 套接字选项来避免 TIME_WAIT状态，此套接字选项告诉内核，即使此端口正忙（处于 TIME_WAIT状态），也请继续并重用它。
- 调整系统内核参数，修改/etc/sysctl.conf文件，即修改`net.ipv4.tcp_tw_reuse` 和 `tcp_timestamps`

  ```
  net.ipv4.tcp_tw_reuse = 1 表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭；
  net.ipv4.tcp_tw_recycle = 1 表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭。
  ```

- 强制关闭，发送 RST 包越过TIME_WAIT状态，直接进入CLOSED状态。

21.TIME-WAIT是服务端的状态？还是客户端的状态？

TIME_WAIT 是主动断开连接的一方会进入的状态，一般情况下，都是客户端所处的状态;服务器端一般设置不主动关闭连接。

TIME_WAIT 需要等待 2MSL，在大量短连接的情况下，TIME_WAIT会太多，这也会消耗很多系统资源。对于服务器来说，在 HTTP 协议里指定 KeepAlive（浏览器重用一个 TCP 连接来处理多个 HTTP 请求），由浏览器来主动断开连接，可以一定程度上减少服务器的这个问题。

22.TCP如何保证可靠性？

TCP主要提供了检验和、序列号/确认应答、超时重传、滑动窗口、拥塞控制和 流量控制等方法实现了可靠性传输。

- 检验和：通过检验和的方式，接收端可以检测出来数据是否有差错和异常，假如有差错就会直接丢弃TCP段，重新发送。
- 序列号/确认应答：

  序列号的作用不仅仅是应答的作用，有了序列号能够将接收到的数据根据序列号排序，并且去掉重复序列号的数据。

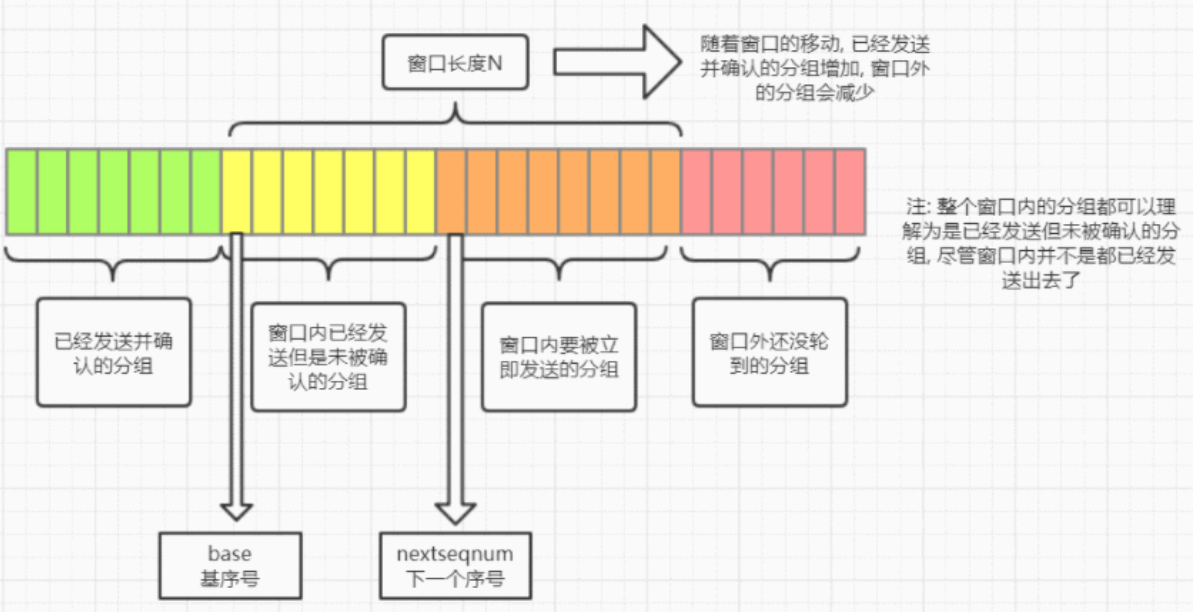
TCP传输的过程中，每次接收方收到数据后，都会对传输方进行确认应答。也就是发送ACK报文，这个ACK报文当中带有对应的确认序列号，告诉发送方，接收到了哪些数据，下一次的数据从哪里发。

- 滑动窗口：滑动窗口既提高了报文传输的效率，也避免了发送方发送过多的数据而导致接收方无法正常处理的异常。
- 超时重传：超时重传是指发送出去的数据包到接收到确认包之间的时间，如果超过了这个时间会被认为是丢包了，需要重传。最大超时时间是动态计算的。
- 拥塞控制：在数据传输过程中，可能由于网络状态的问题，造成网络拥堵，此时引入拥塞控制机制，在保证TCP可靠性的同时，提高性能。
- 流量控制：如果主机A 一直向主机B发送数据，不考虑主机B的接受能力，则可能导致主机B的接受缓冲区满了而无法再接受数据，从而会导致大量的数据丢包，引发重传机制。而在重传的过程中，若主机B的接收缓冲区情况仍未好转，则会将大量的时间浪费在重传数据上，降低传送数据的效率。所以引入流量控制机制，主机B通过告诉主机A自己接收缓冲区的大小，来使主机A控制发送的数据量。流量控制与TCP协议报头中的窗口大小有关。

23.滑动窗口

在进行数据传输时，如果传输的数据比较大，就需要拆分为多个数据包进行发送。TCP 协议需要对数据进行确认后，才可以发送下一个数据包。这样一来，就会在等待确认应答包环节浪费时间。

为了避免这种情况，TCP引入了窗口概念。窗口大小指的是不需要等待确认应答包而可以继续发送数据包的最大值。



滑动窗口里面也分为两块，一块是已经发送但是未被确认的分组，另一块是窗口内等待发送的分组。随着已发送的分组不断被确认，窗口内等待发送的分组也会不断被发送。整个窗口就会往右移动，让还没轮到的分组进入窗口内。

可以看到滑动窗口起到了一个限流的作用，也就是说当前滑动窗口的大小决定了当前 TCP 发送包的速率，而滑动窗口的大小取决于拥塞控制窗口和流量控制窗口的两者间的最小值。

24.拥塞控制

拥塞现象是指到达通信子网中某一部分的分组数量过多，使得该部分网络来不及处理，以致引起这部分乃至整个网络性能下降的现象，严重时甚至会导致网络通信业务陷入停顿，即出现死锁现象。

TCP 一共使用了四种算法来实现拥塞控制：

- 慢开始 (slow-start)；
- 拥塞避免 (congestion avoidance)；
- 快速重传 (fast retransmit)；
- 快速恢复 (fast recovery)。

发送方维持一个叫做拥塞窗口cwnd（congestion window）的状态变量。当cwndssthresh时，改用拥塞避免算法。

慢开始：不要一开始就发送大量的数据，由小到大逐渐增加拥塞窗口的大小。

拥塞避免：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。

快重传：我们可以剔除一些不必要的拥塞报文，提高网络吞吐量。比如接收方在**收到一个失序的报文段后就立即发出重复确认，而不要等到自己发送数据时捎带确认。**快重传规定：发送方只要**一连收到三个**重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。

快恢复：主要是配合快重传。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**接下来并不执行慢开始算法**，因为如果网络出现拥塞的话就不会收到好几个重复的确认，收到三个重复确认说明网络状况还可以。

25.HTTP状态码

- 200：服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。
- 301 ： (永久移动) 请求的网页已永久移动到新位置。 服务器返回此响应(对 GET 或 HEAD 请求的响应)时，会自动将请求者转到新位置。
- 302：(临时移动) 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。
- 400 ： 客户端请求有语法错误，不能被服务器所理解。
- 403 ： 服务器收到请求，但是拒绝提供服务。
- 404 ： (未找到) 服务器找不到请求的网页。
- 500： (服务器内部错误) 服务器遇到错误，无法完成请求。

状态码开头代表类型：

	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

301和302都是重定向，但是301的旧地址已经被永久移除了，302旧地址仍然可以访问

26.HTTP常见请求方式

方法	作用
GET	获取资源
POST	传输实体主体
PUT	上传文件
DELETE	删除文件
HEAD	和GET方法类似，但只返回报文首部，不返回报文实体主体部分
PATCH	对资源进行部分修改
OPTIONS	查询指定的URL支持的方法
CONNECT	要求用隧道协议连接代理
TRACE	服务器会将通信路径返回给客户端

27.解释长连接短连接

在HTTP/1.0中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。如果客户端浏览器访问的某个HTML或其他类型的 Web页中包含有其他的Web资源，如JavaScript文件、图像文件、CSS文件等；当浏览器每遇到这样一个Web资源，就会建立一个HTTP会话。

但从 **HTTP/1.1起，默认使用长连接**，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码：`Connection:keep-alive`

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的 TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接要客户端和服务端都支持长连接。

HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。

28.请求报文和响应报文格式

请求报文：

1. 请求行（请求方法+URI协议+版本）
2. 请求头部
3. 空行
4. 请求主体

```
GET/sample.jspHTTP/1.1 请求行
Accept:image/gif.image/jpeg, 请求头部
Accept-Language:zh-cn
Connection:Keep-Alive
Host:localhost
User-Agent:Mozila/4.0(compatible;MSIE5.01;Window NT5.0)
Accept-Encoding:gzip,deflate

username=jinqiao&password=1234 请求主体
```

响应报文：

1. 状态行（版本+状态码+原因短语）
2. 响应首部
3. 空行
4. 响应主体

```
HTTP/1.1 200 OK
Server:Apache Tomcat/5.0.12
Date:Mon,6Oct2003 13:23:42 GMT
Content-Length:112

<html>
    <head>
        <title>HTTP响应示例<title>
    </head>
    <body>
        Hello HTTP!
    </body>
</html>
```

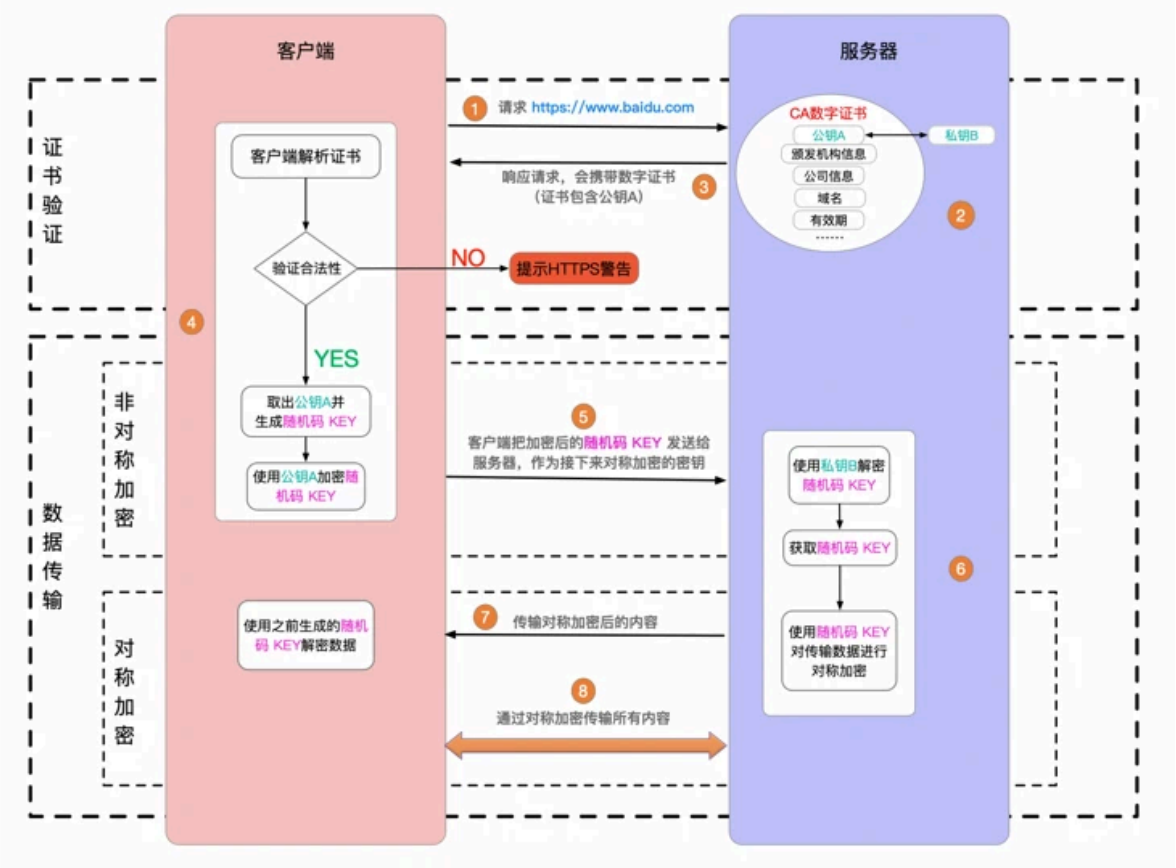
29.HTTP1.0和HTTP1.1的区别

1、长连接
2、添加缓存处理：引入缓存控制策略
3、带宽优化和网络连接优化：解决带宽浪费，断点续传
4、新增错误通知：新增了24个错误状态码
5、Host头处理：增加Host头域

30.HTTP1.1和HTTP2.0的区别

1、增加新的二进制格式
2、多路复用：连接共享
3、头部压缩
4、服务端推送

31.HTTPS的原理



32.DDos攻击

DDos全称Distributed Denial of Service，分布式拒绝服务攻击。最基本的DOS攻击过程如下：

1. 客户端向服务端发送请求链接数据包。
2. 服务端向客户端发送确认数据包。
3. 客户端不向服务端发送确认数据包，服务器一直等待来自客户端的确认

DDoS则是采用分布式的方法，通过在网络上占领多台“肉鸡”，用多台计算机发起攻击。

DOS攻击现在基本没啥作用了，因为服务器的性能都很好，而且是多台服务器共同作用，1V1的模式黑客无法占上风。对于DDOS攻击，预防方法有：

- **减少SYN timeout时间。**在握手的第三步，服务器会等待30秒-120秒的时间，减少这个等待时间就能释放更多的资源。
- **限制同时打开的SYN半连接数目。**

33.TLS1.2原理

1、客户端服务端互相发送hello，发送一些基础数据、随机数
2、服务端发送证书，客户端校验证书安全性，互相交换算法需要的参数
3、客户端发送协商密钥给服务端，需要三个随机数，服务端返回告知使用此密钥加密
4、然后就是使用密钥和协商参数进行加密

> 1. 客户端向服务器发送 Client Hello 信息，告知自己想要建立一条 TLS 连接，并告知自己支持的加密算法。
> 2. 服务器向客户端发送一个 Server Hello 的回应，并选择一个加密算法，同时给客户端发送自己的数字证书（包含服务器的公钥）。

3. 客户端验证服务器发来的数字证书，验证通过后，在心里默默想出一个 pre-master 密钥（预主密钥），然后使用服务器的公钥，将预主密钥进行加密后，发送给服务器。
4. 服务器用自己的私钥进行解密，得到预主密钥。
5. 客户端和服务器都通过预主密钥，进行相同的计算后，得到后续通信时使用的对称加密密钥，称为 shared secret。
6. 客户端和服务器端都分别用生成的 shared-secret 加密一段报文后，发送给对方，以验证对方能够成功收到信息并解密

2.密码安全

- 数字签名

  一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。发送者持有私钥，接受者使用公钥来解密验证发送者身份

- 散列算法、摘要算法、哈希算法

  可将任意长度的消息经过运算，变成固定长度数值，常见的有MD5、SHA-1、SHA256，多应用在文件校验，数字签名中。

  MD5 可以将任意长度的原文生成一个128位（16字节）的哈希值，SHA-1可以将任意长度的原文生成一个160位（20字节）的哈希值。HMAC利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出。**这些算法都是不可逆的！**

- 加密算法：加密方式有两种（流加密、块加密）
  - Base64

    Base64不是加密算法，是一种可读性算法。Base64的目的不是保护数据，是为了让数据可读，可看到大部分算法加密后显示都会产生乱码使用了Base64可以将这些乱码转为可读的字符串。

  - 对称加密：DES、AES、3DES

    DES：使用块算法加密

    AES：用来替代原先的DES，已经被多方分析并且广为全世界使用

  - 非对称加密：RSA、DSA、ECC

    RSA：`RSA`密钥至少为500位长，一般推荐使用1024位。

    ECC：它比其他的方法使用 **更小的密钥**，耗时更严重

- 国密算法

  主要使用公开的SM2、SM3、SM4三类算法，分别是非对称算法、哈希算法和对称算法

七、设计模式

单例模式、策略模式（一个接口，多个实现）、代理模式、模板方法模式、观察者模式、工厂模式

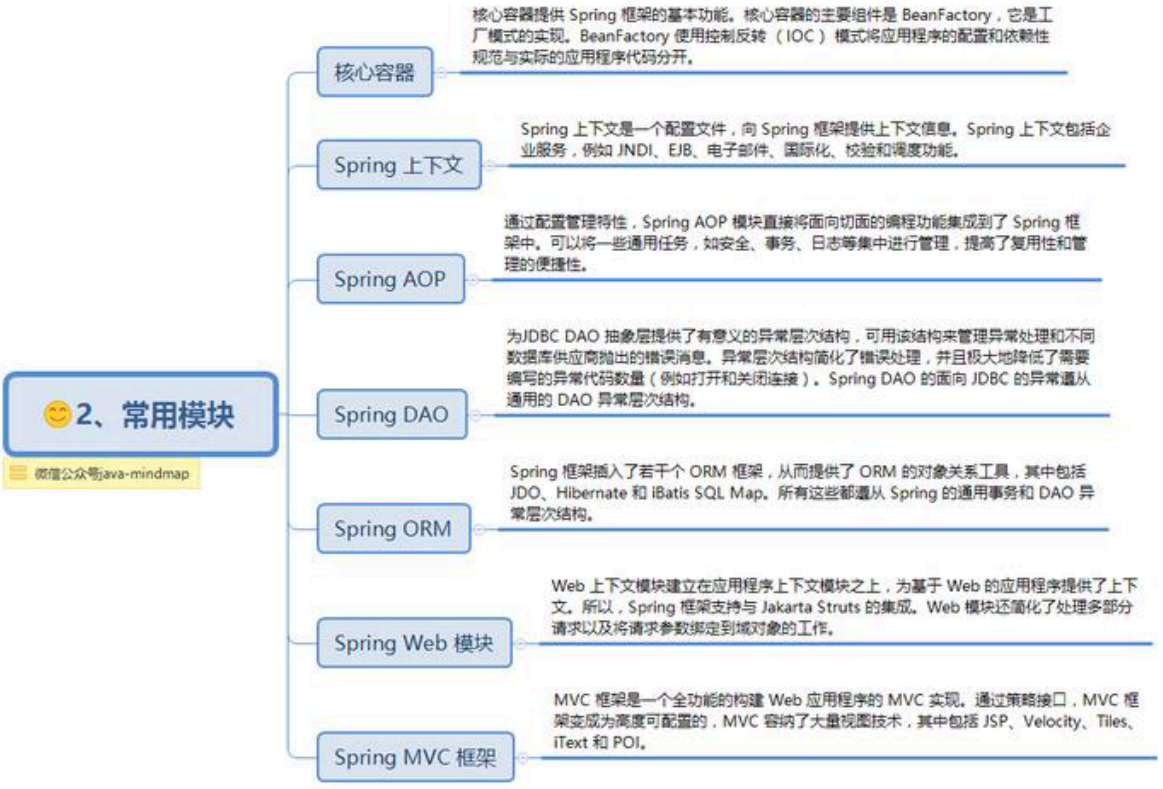
八、Spring

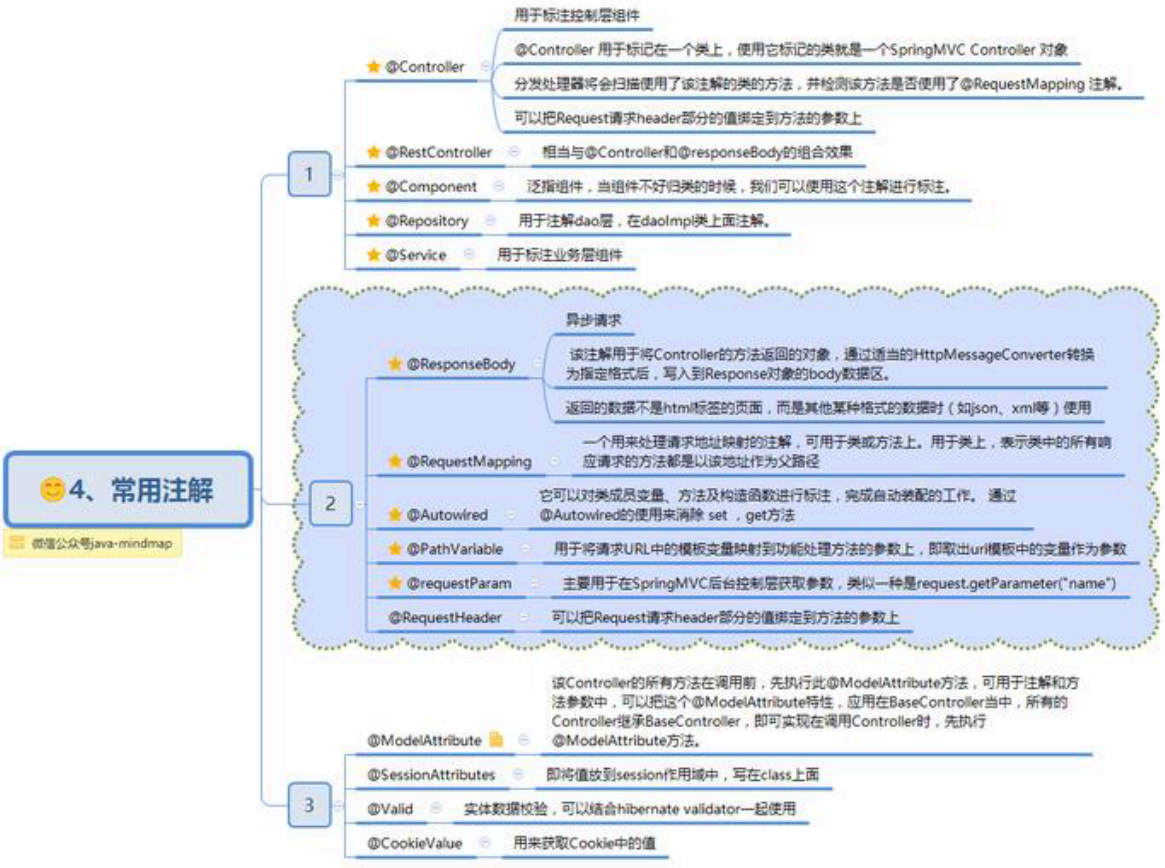
面试题总结：https://blog.csdn.net/ThinkWon/article/details/104397516

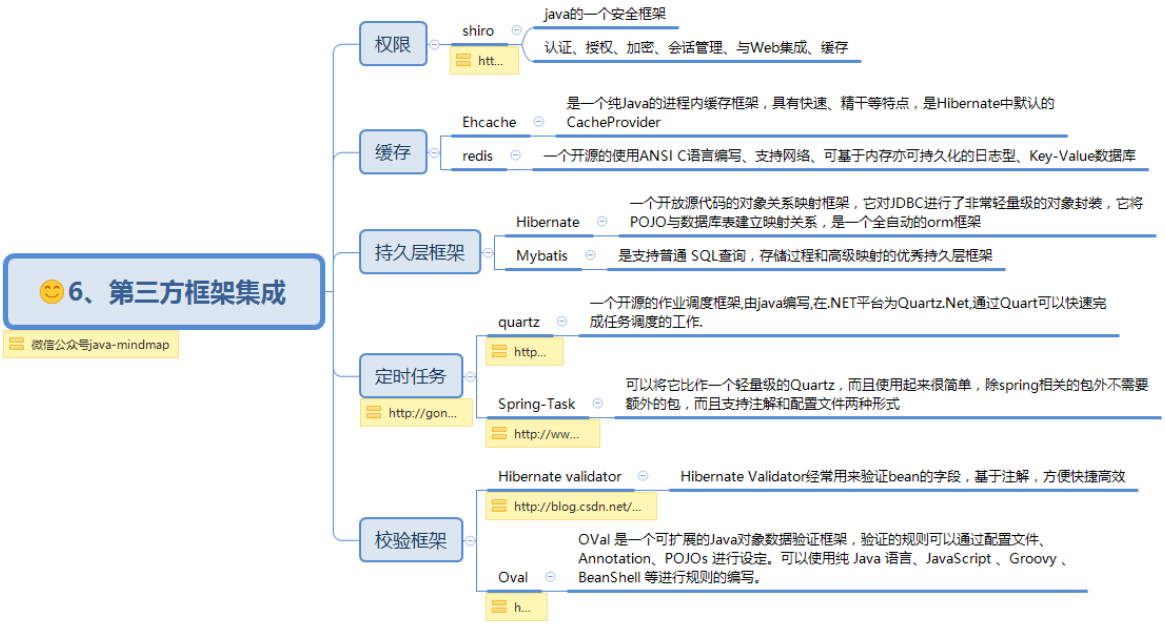
1、基础

0.简介


😊1、Spring特征
微信公众号java-mindmap
轻量
从大小与开销两方面而言Spring都是轻量的。完整的Spring框架可以在一个大小只有1M多的JAR文件里发布，并且Spring所需的处理开销也是微不足道的。
此外，Spring是非侵入式的：典型的，Spring应用中的对象不依赖于Spring的特定类。
控制反转
Spring通过一种称作控制反转IOC的技术促进了低耦合。
当应用了IOC，一个对象依赖的其他对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。
面向切面
Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开
容器
Spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个bean如何被创建----基于一个可配置原型，你的bean可以创建一个单独的实例或者每次需要时都生成一个新的实例----以及它们是如何相互关联的。
框架
Spring可以将简单的组件配置、组合成为复杂的应用。
在Spring中，应用对象被声明式地组合，典型的是在一个XML文件里。
Spring也提供了很多基础功能（事务管理、持久化框架集成等），将应用逻辑的开发留给开发者。


😊2、常用模块
微信公众号java-mindmap
核心容器
核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
Spring 上下文
Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
Spring AOP
通过配置管理特性，Spring AOP 模块直接将面向切面的编程功能集成到了 Spring 框架中。可以将一些通用任务，如安全、事务、日志等集中进行管理，提高了复用性和管理的便捷性。
Spring DAO
为JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。
Spring ORM
Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
Spring Web 模块
Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
Spring MVC 框架
MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。





Spring是**一个轻量级Java开发框架**，目的是为了解决企业级应用开发的业务逻辑层和其他各层的耦合问题。

组成模块：Spring有七大功能模块，分别是Spring Core，AOP，ORM，DAO，MVC，WEB，Context。

4种注入方式：setter属性注入，构造方法注入，静态工厂注入，实例工厂

bean作用域：singleton、prototype（每次调用产生新bean）、request（每次请求都会创建）、session（同一个http session共享）、global-session

自动装配bean的方式：no、byName、byType、构造函数、autodetect

> 手动装配：xml装配、构造方法、setter方法

事务实现方式：声明式事务（xml和注解，AOP，将业务代码和事务管理分离）、编程式事务（采用编程的方式管理事务难维护）

事务传播行为：

① PROPAGATION*REQUIRED*：*如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。*
② *PROPAGATION*SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘
③ PROPAGATION*MANDATORY*：*支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。*
④ *PROPAGATION*REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。
⑤ PROPAGATION*NOT_SUPPORTED*：*以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。*
⑥ *PROPAGATION*NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。
⑦ PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

事务隔离：spring 有五大隔离级别，默认值为 ISOLATION_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致：

1.ISOLATION*DEFAULT*：*用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；*
*2.ISOLATION_READ*UNCOMMITTED：读未提交，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）；
3.ISOLATION_READ*COMMITTED*：*读已提交，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；*
4.*ISOLATION*REPEATABLE_READ：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；
5.ISOLATION_SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

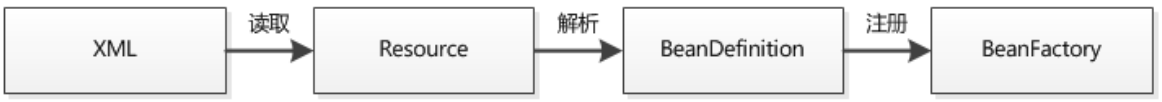
1.循环依赖解决?

循环依赖：就是在进行getBean的时候，A对象中去依赖B对象，而B对象又依赖C对象，但是对象C又去依赖A对象，结果就造成A、B、C三个对象都不能完成实例化，出现了循环依赖。就会出现死循环，最终导致内存溢出的错误。

spring对循环依赖的处理有三种情况： ①构造器的循环依赖：这种依赖spring是处理不了的，直接抛出BeanCurrentlyInCreationException异常。 ②单例模式下的setter循环依赖：通过**“三级缓存”**（提前曝光机制，提前查找Map里是否已经存放过）处理循环依赖。 ③非单例prototype循环依赖：无法处理，因为Spring容器不进行缓存。

2.IOC、AOP?

IOC初始化过程：



1.Resource定位　2.BeanDefination载入和解析　　3.向IoC容器注册这些BeanDefinition

1）自己实现IOC?

1.定义用来描述bean的配置的Java类

2.解析bean的配置，将bean的配置信息转换为上面的BeanDefinition对象保存在内存中，spring中采用HashMap进行对象存储，其中会用到一些xml解析技术

3.遍历存放BeanDefinition的HashMap对象，逐条取出BeanDefinition对象，获取bean的配置信息，利用Java的反射机制实例化对象，将实例化后的对象保存在另外一个Map中即可。

2）AOP

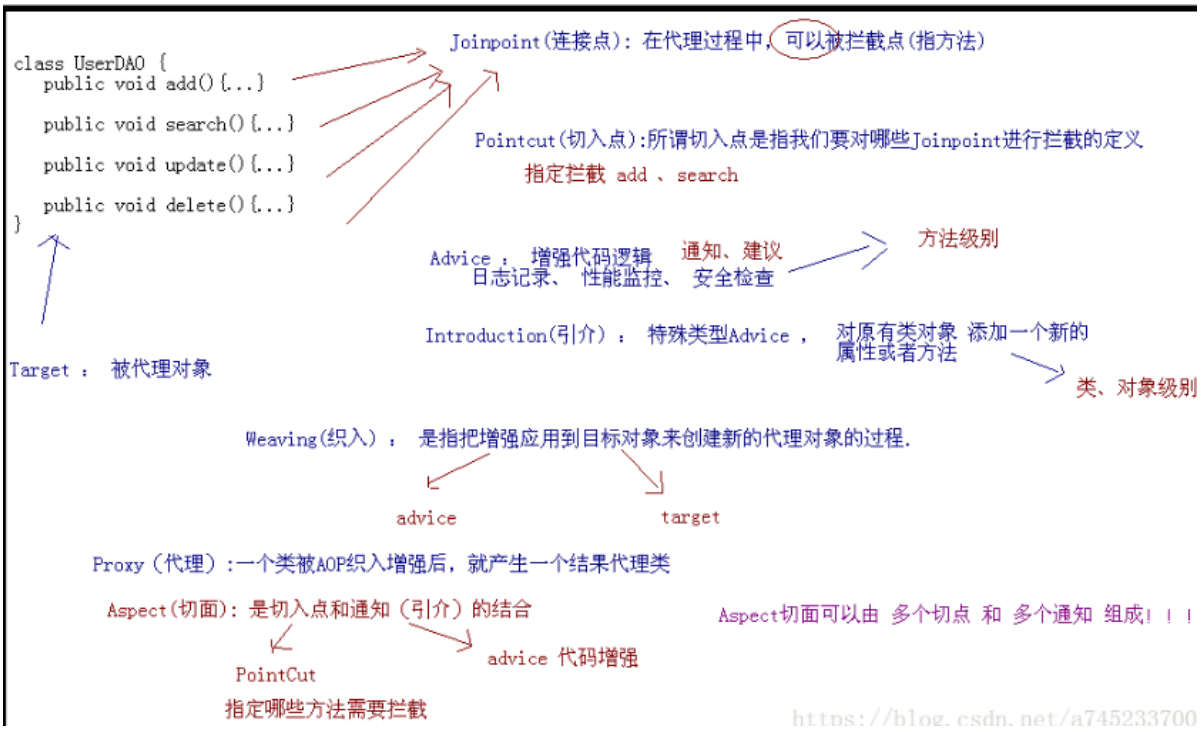
Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

> ①JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例， 生成目标类的代理对象。
>
> ②如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

（3）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

AOP里的名词：

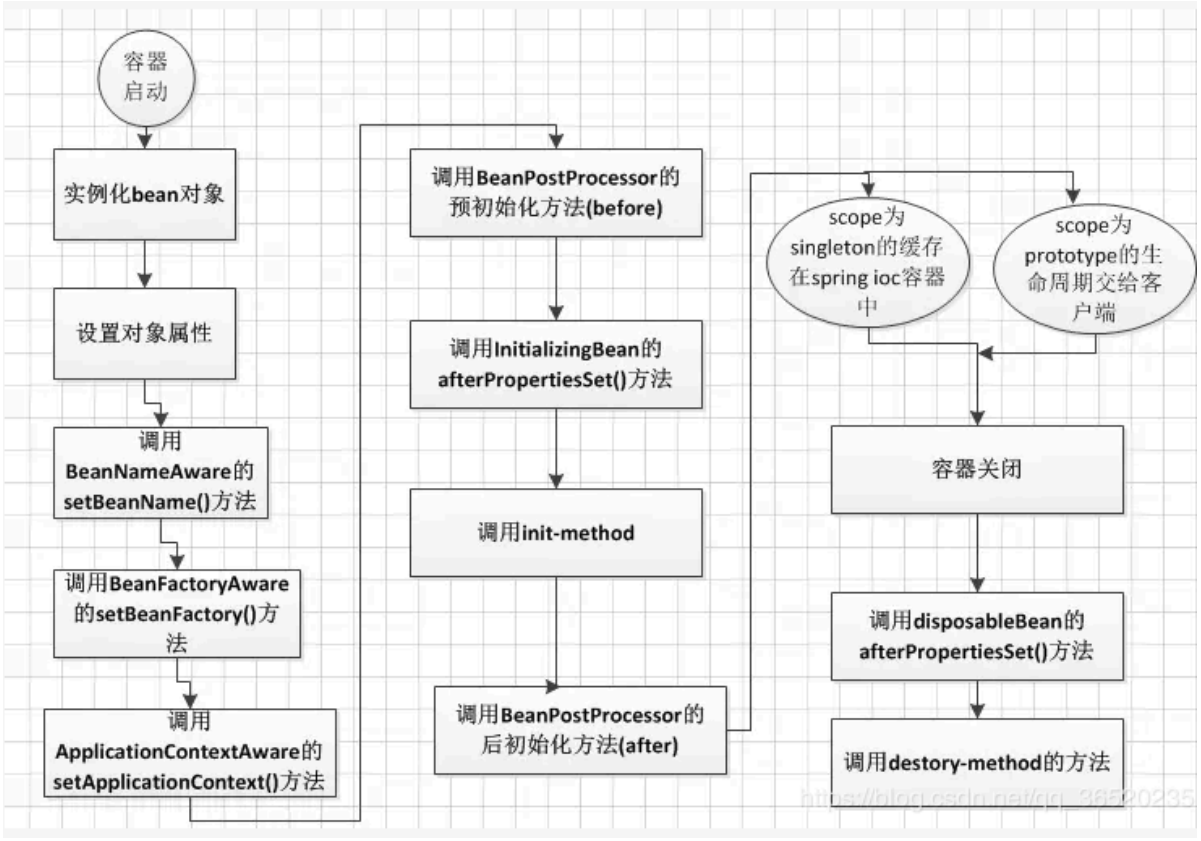
（1）**切面（Aspect）**：被抽取的公共模块，可能会横切多个对象。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。
（2）**连接点（Join point）**：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。
（3）**通知（Advice）**：在切面的某个特定的连接点（Join point）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。许多AOP框架，包括Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。
（4）**切入点（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。


class UserDAO {
public void add(){...}
public void search(){...}
public void update(){...}
public void delete(){...}
}
Joinpoint(连接点): 在代理过程中，可以被拦截点(指方法)
Pointcut(切入点):所谓切入点是指我们要对哪些Joinpoint进行拦截的定义
指定拦截 add 、search
Advice： 增强代码逻辑 通知、建议
日志记录、 性能监控、 安全检查
方法级别
Introduction(引介)： 特殊类型Advice， 对原有类对象 添加一个新的属性或者方法
类、对象级别
Target： 被代理对象
Weaving(织入)： 是指把增强应用到目标对象来创建新的代理对象的过程.
advice
target
Proxy（代理）:一个类被AOP织入增强后，就产生一个结果代理类
Aspect(切面): 是切入点和通知（引介）的结合
Aspect切面可以由 多个切点 和 多个通知 组成！！！
PointCut
指定哪些方法需要拦截
advice 代码增强
https://blog.csdn.net/a745233700

3）通知类型

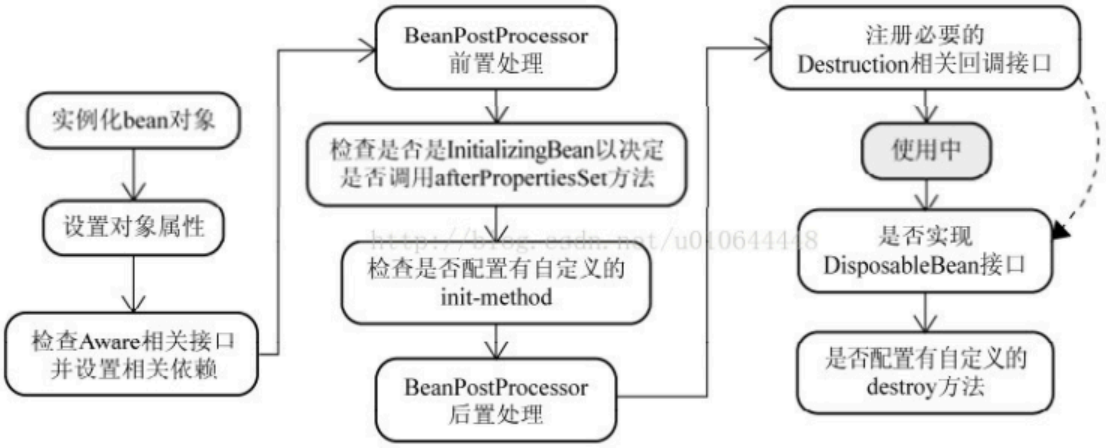

通知类型
前置通知（Before advice）：在某连接点之前执行的通知，但这个通知不能阻止连接点之前的执行流程（除非它抛出一个异常）。
后置通知（After returning advice）：在某连接点正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。
异常通知（After throwing advice）：在方法抛出异常退出时执行的通知。
最终通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。
环绕通知（Around Advice）：包围一个连接点的通知，如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它自己的返回值或抛出异常来结束执行。

3.Bean生命周期?



1. 首先会先进行实例化bean对象
2. 然后是进行对bean的一个属性进行设置
3. 接着是对BeanNameAware（其实就是为了让Spring容器来获取bean的名称）、BeanFactoryAware（让bean的BeanFactory调用容器的服务）、ApplicationContextAware（让bean当前的applicationContext可以来取调用Spring容器的服务）
4. 然后是实现BeanPostProcessor 这个接口中的两个方法，主要是对调用接口的前置初始化 postProcessBeforeInitialization
5. 这里是主要是对xml中自己定义的初始化方法 init-method = “xxxx”进行调用
6. 然后是继续对BeanPostProcessor 这个接口中的后置初始化方法进行一个调用 postProcessAfterInitialization（）
7. 其实到这一步，基本上这个bean的初始化基本已经完成，就处于就绪状态
8. 然后就是当Spring容器中如果使用完毕的话，就会调用destory（）方法
9. 最后会去执行我们自己定义的销毁方法来进行销毁，然后结束生命周期

流程简图：



4.用到的设计模式?

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；
单例模式：Bean默认为单例模式。
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；
模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。
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。

5.BeanFactory和ApplicationContext区别?

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

Spring使用BeanFactory来读取配置文件，实例化，配置和管理Bean，它是root接口，采用延迟加载注入Bean。

ApplicationContext是BeanFactory的派生，在容器启动时就一次性加载了所有Bean，提供了更多的功能：国际化支持、资源访问、事件传递。

6.如何处理线程并发安全问题?

Controller默认是单例的，所以就是非线程安全的。

【1】不要在 Controller 中定义实例（成员）变量；
【2】万一必须要定义一个非静态成员变量时候，则通过注解 @Scope(“prototype”)，将其设置为多例模式。
【3】在 Controller 中使用 ThreadLocal 变量；

7.怎样解决依赖冲突?

1.依赖冲突时，根据错误日志，定位到冲突类，定位相应 `jar` 包，最后通过 **excludes** 排除相应的包。
2.另外可以结合 **IDEA Maven Helper** 插件，主动检查冲突依赖，提前排除。
但是我们最根本还是需要确定规范：

- 应用项目中使用 `dependencyManagement` 统一管理基础依赖，定义统一的版本
- 二方包中不要引入无关的依赖，做好向下兼容

8.BeanFactory和FactoryBean?

- BeanFactory：是个Factory，也就是IOC容器的顶级接口或对象工厂。Spring中所有的Bean都是由IOC容器来进行管理的
- FactoryBean：是个Bean，可以返回Bean的实例，在IOC容器的基础上给Bean的实现加上了一个简单工厂模式和装饰模式，我们可以在getObject()方法中灵活配置使用。为简化实例化Bean而存在。

9.@Autowired和@Resource的区别?

相同点：@Resource的作用相当于@Autowired，均可标注在字段或者属性的setter方法上。

不同点：

@Autowired默认按**类型装配**（基于spring的），默认情况下必须要求依赖对象必须存在，如果要允许null值，可以设置它的required属性为false，如@Autowired(required=false)，如果我们想使用名称装配可以结合@Qualifier注解进行使用，如下：

```
@Autowired() @Qualifier("baseDao")
private BaseDao baseDao;
```

@Resource是按**名称装配**（基于JDK的），名称可以通过@Resource的name属性指定，如果没有指定name属性，当注解标注在字段上，即默认取字段的名称作为bean名称寻找依赖对象，当注解标注在属性的setter方法上，即默认取属性名作为bean名称寻找依赖对象。
@Resource有两个重要的属性：name和type，而Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以，如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不制定name也不制定type属性，这时将通过反射机制使用byName自动注入策略。

10.Spring容器的bean什么时候被实例化?

（1）如果你使用BeanFactory作为Spring Bean的工厂类，则所有的bean都是在第一次使用该Bean的时候实例化

（2）如果你使用ApplicationContext作为Spring Bean的工厂类，则又分为以下几种情况：

- 如果bean的scope是singleton的，并且lazy-init为false（默认是false，所以可以不用设置），则ApplicationContext启动的时候就实例化该Bean，并且将实例化的Bean放在一个map结构的缓存中，下次再使 用该 Bean的时候，直接从这个缓存中取
- 如果bean的scope是singleton的，并且lazy-init为true，则该Bean的实例化是在第一次使用该Bean的时候进 行实例化
- 如果bean的scope是prototype的，则该Bean的实例化是在第一次使用该Bean的时候进行实例化

11.Spring中的注解原理?

注解的本质就是一个继承了 Annotation 接口的接口。

12.IOC的实现机制?

Spring 中的 IoC 的实现原理就是**工厂模式加反射机制。**

1. 加载配置文件，解析成 BeanDefinition 放在 Map 里。

2. 调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出 Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法 —— 完成依赖注入。

13.AOP and AspectJ AOP的区别？AOP 有哪些实现方式?

AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为**编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。**

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

14.JDK动态代理和CGLIB动态代理?

> **静态代理步骤：**
>
> 1. 定义一个接口及其实现类；
> 2. 创建一个代理类同样实现这个接口
> 3. 将目标对象注入进代理类，然后在代理类的对应方法调用目标类中的对应方
>
> **JDK动态代理步骤：**只能代理实现了接口的类
>
> 1. 定义一个接口及其实现类；
> 2. 自定义 `InvocationHandler` 并重写`invoke`方法，在 `invoke` 方法中我们会调用原生方法（被代理类的方法）并自定义一些处理逻辑；
> 3. 通过 `Proxy.newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)` 方法创建代理对象；
>
> **CGLib动态代理：**
>
> 1. 定义一个类；
> 2. 自定义 `MethodInterceptor` 并重写 `intercept` 方法，`intercept` 用于拦截增强被代理类的方法，和 JDK 动态代理中的 `invoke` 方法类似；
> 3. 通过 `Enhancer` 类的 `create()`创建代理类；

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

JDK动态代理**只提供接口的代理，不支持类的代理。**核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。

如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。静态代理需要代理对象和目标对象实现一样的接口；而动态代理的代理对象不需要实现接口，但是要求目标对象必须实现接口。

14.1 JDK动态代理

首先是java.lang.reflect包里的InvocationHandler接口：

```
public interface InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

我们对于被代理的类的操作都会由该接口中的invoke方法实现，其中的参数的含义分别是：

- proxy：被代理的类的实例
- method：调用被代理的类的方法
- args：该方法需要的参数

另外一个很重要的静态方法是java.lang.reflect包中的Proxy类的newProxyInstance方法：

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
    throws IllegalArgumentException
```

其中的参数含义如下：

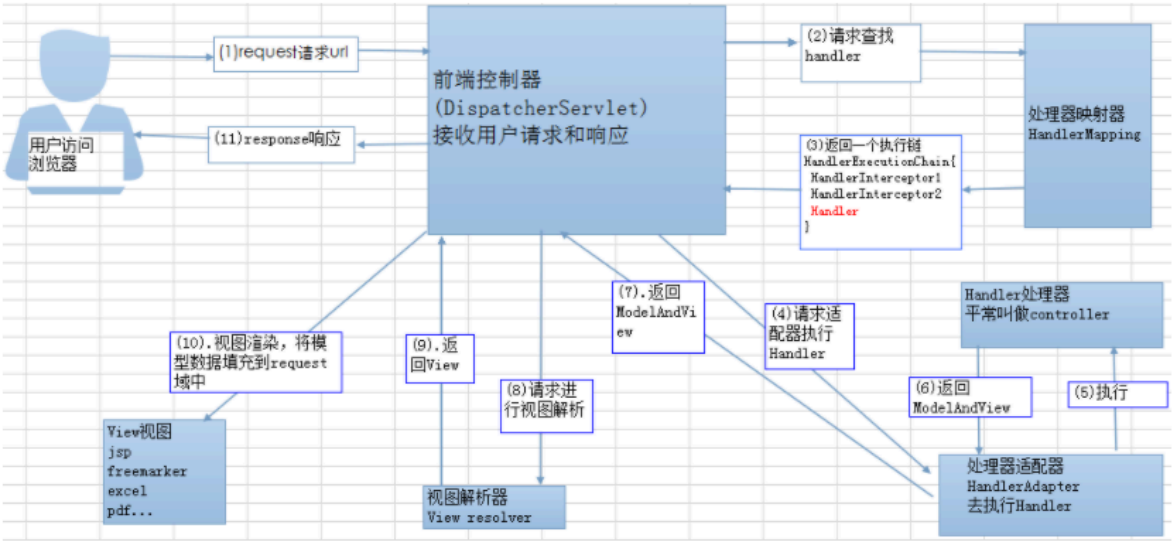
- loader：被代理的类的类加载器
- interfaces：被代理类的接口数组
- invocationHandler：就是刚刚介绍的调用处理器类的对象实例

1.SpringMVC运行流程?

主要组件：前端控制器，处理器映射器，处理器适配器，视图解析器

- spring mvc 先将请求发送给 DispatcherServlet。
- DispatcherServlet 查询一个或多个 HandlerMapping，找到处理请求的 Controller。
- DispatcherServlet 再把请求提交到对应的 Controller。
- Controller 进行业务逻辑处理后，会返回一个ModelAndView。
- Dispathcher 查询一个或多个 ViewResolver 视图解析器，找到 ModelAndView 对象指定的视图对象。
- 视图对象负责渲染返回给客户端。

原理如下图所示：



15.SpringBoot热部署?

- 使用 devtools 启动热部署，添加 devtools 库，在配置文件中把 spring. devtools. restart. enabled 设置为 true；
- 使用 Intellij Idea 编辑器，勾上自动编译或手动重新编译。..

16.

17.@Transactional(rollbackFor = Exception.class)注解

我们知道：Exception分为运行时异常RuntimeException和非运行时异常。事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。

当`@Transactional`注解作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。如果类或者方法加了这个注解，那么这个类里面的方法抛出异常，就会回滚，数据库里面的数据也会回滚。

在`@Transactional`注解中如果不配置`rollbackFor`属性,那么事物只会在遇到`RuntimeException`的时候才会回滚,加上`rollbackFor=Exception.class`,可以让事物在遇到非运行时异常时也回滚。

18. 其他面试题

1.@Autowired自动装配过程?

在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

- 如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；
- 如果查询的结果不止一个，那么@Autowired会根据名称来查找；
- 如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

2.自动装配的局限性

重写：你仍需用 和 配置来定义依赖，意味着总要重写自动装配。

基本数据类型：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。

模糊特性：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

3.@Required注解的作用

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。示例：

```
public class Employee {
    private String name;
    @Required
    public void setName(String name){
        this.name=name;
    }
    public string getName(){
        return name;
    }
}
```

九、微服务

1.面试题

1.SpringBootAppliation注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited // 子类也会带上该注解
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
```

2.SpringBoot自动装配原理

Spring Boot 通过`@EnableAutoConfiguration`开启自动装配，通过 SpringFactoriesLoader 最终加载`META-INF/spring.factories`中的自动配置类实现自动装配，自动配置类其实就是通过`@Conditional`按需加载的配置类，想要其生效必须引入`spring-boot-starter-xxx`包实现起步依赖

SpringBoot配置文件加载顺序?

1）properties文件；bootstrap文件是最优先级的；

2）YAML文件；

3）系统环境变量；

4）命令行参数；

SpringBoot如何解决跨域?

使用Cors实现`WebMvcConfigurer`类重写`addCorsMappings`方法

3.SPI

全称： Service Provider Interface

是一种服务发现机制，它通过在ClassPath路径下的META-INF/services文件夹查找文件，自动加载文件里所定义的类。**是一种将服务接口与服务实现分离以达到解耦、大大提升了程序可扩展性的机制。引入服务提供者就是引入了spi接口的实现者，通过本地的注册发现获取到具体的实现类，轻松可插拔**

4.事务失效场景

1、底层数据库引擎不支持事务
2、在非public修饰的方法使用，@Transactional是AOP，只能针对public方法进行动态代理
3、异常被try catch住，无法抛出
4、未标注事务的方法内调用了标注了事务的方法，会导致里面的事务方法失效，因为是代理类没有标注事务
5、rollbackFor属性异常设置错误，propagation属性设置错误（SUPPORTS、NOT_SUPPORTES、NEVER）

5.Nacos和Euraka的区别

CAP：一致性、可用性、分区容错性

	Nacos	Euraka	Zookeeper
一致性协议	AP和CP混合，默认AP，CP则采用Raft协议实现保持数据的一致性	保证AP，去中心化，每个节点都是平等的	保证CP，中心化，围绕领导选举机制
负载均衡	本身集成Ribbon，权值	Ribbon	无
雪崩保护	有	有	无
k8s集成	支持	不支持	不支持

其他：部署方式不一样，euraka需要自己创建springboot项目，nacos是下载jar包

6.Nacos整合Dubbo

1、添加`spring-cloud-starter-dubbo`依赖
2、创建一个模块只对外暴露接口
3、创建一个类来实现这个接口，作为生产者来实现该远程调用的方法`@Service`
4、配置dubbo扫描该实现类的路径，以及注册中心上nacos的地址
5、启动nacos的服务发现注册`@EnableDiscoveryClient`
6、消费者使用Dubbo的注解**@Reference**来注入接口实现类从而调用远程方法

7.Dubbo的负载均衡策略

1、随机
2、权重轮询
3、最少活跃数
4、一致性哈希

8.Gateway网关

主要概念：路由，断言、过滤器

使用：

1、导入`spring-cloud-starter-gateway`依赖
2、注册到注册中心
3、配置断言路由，例子如下

```
#人人后台的路由
- id: admin_route
  uri: lb://renren-fast #lb负载均衡，根据nacos上的服务名来跳转
  predicates:           #断言
     - Path=/api/**      #规定，前端项目都要携带/api请求
  filters:              #路径重写
      - RewritePath=/api/(?<segment>.*),/renren-fast/$\{segment}  #要去除的路径前缀
```

1）Gateway对比Zuul

1、gateway对比zuul多依赖了spring-webflux，在spring的支持下，功能更强大，内部实现了限流、负载均衡等，扩展性也更强，但同时也限制了仅适合于Spring Cloud套件

2、zuul仅支持同步，gateway支持异步。理论上gateway则更适合于提高系统吞吐量（但不一定能有更好的性能），最终性能还需要通过严密的压测来决定

9.为什么要用网关

安全认证，流量控制，日志，监控，拦截过滤，负载均衡

10.限流的算法

- 固定窗口计数器算法：一定时间内只能请求请求固定数量，无法保证突然激增的流量
- 滑动窗口计数器算法：将时间分片，每个单位时间只能请求固定数量
- 漏桶算法：

11.Euraka底层原理

1、内部维护一个registry注册中心的CocurrentHashMap，该注册表基于纯内存操作，所以维护注册表、拉取注册表、更新心跳时间，全部发生在内存里！key代表服务的名称，value代表一个服务的多个实例。value也是一个Map，里面维护服务实例的具体信息。

2、Euraka避免了同时读写内存数据结构造成的并发冲突问题，还采用**多级缓存机制**来进一步提升服务请求的响应速度。

十、MyBatis

1.工作流程?

1、 mybatis配置SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。
mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

2、 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂

3、 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。

4、 mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。

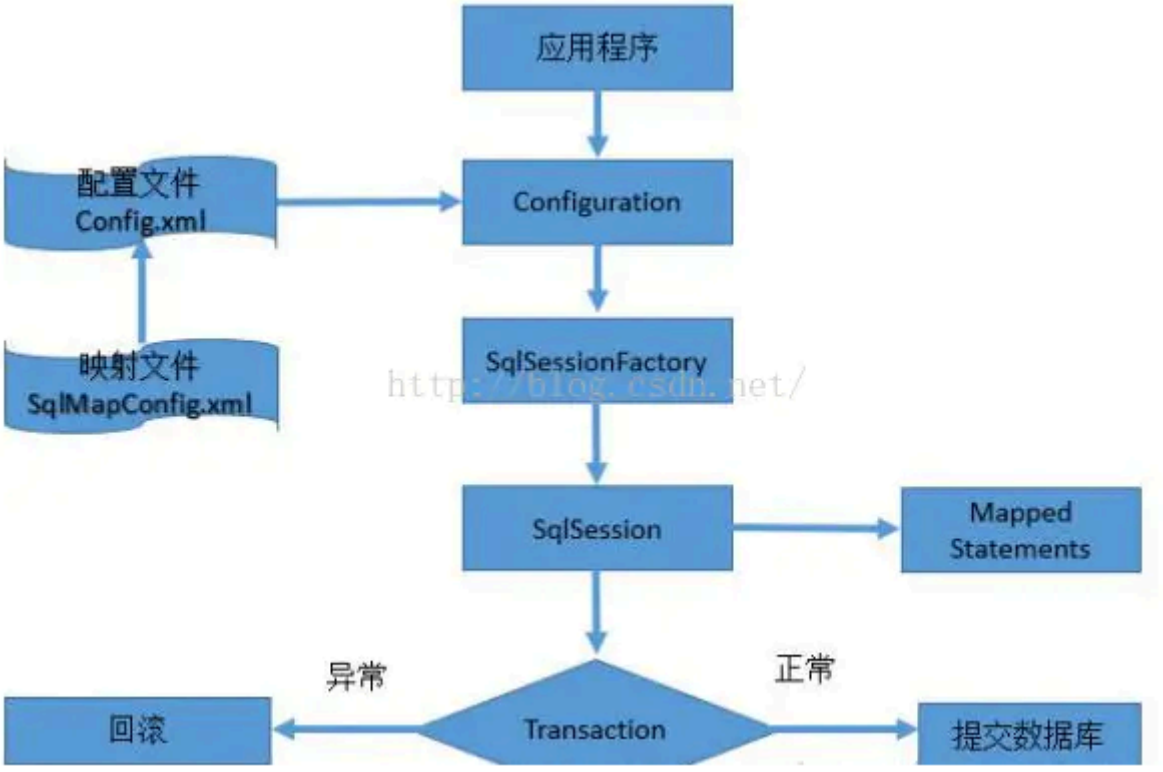
5、 Mapped Statement也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。

6、 Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过 Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。

7、 Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过 Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

编程步骤：

1、 创建SqlSessionFactory
2、 通过SqlSessionFactory创建SqlSession
3、 通过sqlsession执行数据库操作
4、 调用session.commit()提交事务
5、 调用session.close()关闭会话



2.实体类属性和表字段名不一样怎么办?

第一种： 通过在查询的sql语句中**定义字段名的别名，让字段名的别名和实体类的属性名一致**

```
<select id=”selectorder” parametertype=”int” resultetype=”me.gacl.domain.order”>
    select order_id id, order_no orderno ,order_price price form orders where
order_id=#{id};
</select>
```

第二种：**通过`<resultMap>`来映射字段名和实体类属性名的一一对应的关系**

```
<select id="getOrder" parameterType="int" resultMap="orderresultmap">
        select * from orders where order_id=#{id}
</select>
<resultMap type=”me.gacl.domain.order” id=”orderresultmap”>
    <!–用id属性来映射主键字段–>
    <id property=”id” column=”order_id”>
    <!–用result属性来映射非主键字段，property为实体类属性名，column为数据表中的属性–>
    <result property = “orderno” column =”order_no”/>
    <result property=”price” column=”order_price” />
</reslutMap>
```

3.如何获取自动生成的主键值?

通过LAST_INSERT_ID()获取刚插入记录的自增主键值，**在insert语句执行后，执行select LAST_INSERT_ID()就可以获取自增主键。**

```
<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User">
        <selectKey keyProperty="id" order="AFTER" resultType="int">
            select LAST_INSERT_ID()
        </selectKey>
        INSERT INTO USER(username,birthday,sex,address) VALUES(#{username},#
{birthday},#{sex},#{address})
</insert>
```

4.在mapper中如何传递多个参数?

第一种：占位符：`?` 和`@param注解`

第二种：Map集合作为参数：根据key自动找到Map对应的value

5.动态SQL?

> Mybatis动态sql是做什么的？都有哪些动态sql？能简述一下动态sql的执行原理不？

- Mybatis动态sql可以让我们在Xml映射文件内，**以标签的形式编写动态sql，完成逻辑判断和动态拼接sql的功能。**
- Mybatis提供了9种动态sql标签：trim|where|set|foreach|if|choose|when|otherwise|bind。
- 其执行原理为，使用OGNL从sql参数对象中计算表达式的值，**根据表达式的值动态拼接sql，以此来完成动态sql的功能。**

详情：https://zhongfucheng.bitcron.com/post/mybatis/mybatisru-men-kan-zhe-yi-pian-jiu-gou-liao

6.不同的xml映射文件里id是否可以重复?

如果配置了namespace那么当然是可以重复的，因为我们的Statement实际上就是namespace+id

如果没有配置namespace的话，那么相同的id就会导致覆盖了。

7.全自动和半自动ORM?

- Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。
- 而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

8.Dao接口怎么和Xml映射文件对应?

> 通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗?

- Dao接口，就是人们常说的Mapper接口，接口的全限名，就是映射文件中的namespace的值，接口的方法名，就是映射文件中MappedStatement的id值，接口方法内的参数，就是传递给sql的参数。
- Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MappedStatement

```
com.mybatis3.mappers.StudentDao.findStudentById，

可以唯一找到namespace为com.mybatis3.mappers.StudentDao下面id = findStudentById的MappedStatement。在Mybatis中，每一个<select>、<insert>、<update>、<delete>标签，都会被解析为一个MappedStatement对象。
```

Dao接口里的方法，**是不能重载的，因为是全限名+方法名的保存和寻找策略。**

Dao接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Dao接口生成代理proxy对象，代理对象proxy会拦截接口方法，转而执行MappedStatement所代表的sql，然后将sql执行结果返回。

详情：https://www.cnblogs.com/soundcode/p/6497291.html

9.Mabatis如何分页？分页插件原理?

分为逻辑分页和物理分页。

Mybatis使用①RowBounds对象进行分页，它是针对ResultSet结果集执行的**内存分页**，而非物理分页，可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用②分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。

举例：`select * from student，拦截sql后重写为：select t.* from （select * from student）t limit 0，10`

逻辑分页和物理分页的区别?

- 逻辑分页是一次性查询很多数据，然后再在结果中检索分页的数据。这样做弊端是需要消耗大量的内存、有内存溢出的风险、对数据库压力较大。
- 物理分页是从数据库查询指定条数的数据，弥补了一次性全部查出的所有数据的种种缺点，比如需要大量的内存，对数据库查询压力较大等问题。

10.查询语句不同元素执行先后顺序?

> 查询语句不同元素（where、jion、limit、group by、having等等）执行先后顺序?

- from:需要从哪个数据表检索数据
- where:过滤表中数据的条件
- group by:如何将上面过滤出的数据分组
- having:对上面已经分组的数据进行过滤的条件
- select:查看结果集中的哪个列，或列的计算结果
- order by :按照什么样的顺序来查看返回的数据

通过一个顺口溜总结下顺序：我(W)哥(G)是(SH)偶(O)像。按照执行顺序的关键词首字母分别是W（where）->G（Group）->S（Select）->H（Having）->O（Order），对应汉语首字母可以**编成容易记忆的顺口溜：方(f)伟(W)哥(G)还(H)是(S)偶(O)像**

11.Mybatis的插件原理?

> 如何编写一个自定义插件?

Mybatis仅可以**编写针对ParameterHandler、ResultSetHandler、StatementHandler、Executor这4种接口的插件，Mybatis使用JDK的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能**，每当执行这4种接口对象的方法时，就会进入拦截方法，具体就是InvocationHandler的invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

实现Mybatis的Interceptor接口并复写intercept()方法，**然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。**

12.Mybatis是否支持延迟加载?

> 如果支持，实现原理是什么?
>
> 延迟加载：有需要的时候再去查询相应的数据

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，**可以配置是否启用延迟加载lazyLoadingEnabled=true|false。**

它的原理是，**使用CGLIB创建目标对象的代理对象**，当调用目标方法时，**进入拦截器方法**，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

13.Mybatis有哪些Executor执行器？区别?

Mybatis有三种基本的Executor执行器，**SimpleExecutor、ReuseExecutor、BatchExecutor。**

- SimpleExecutor：每执行一次update或select，就开启一个Statement对象，**用完立刻关闭Statement对象。**
- ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，**就是重复使用Statement对象。**
- BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），**它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。**

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

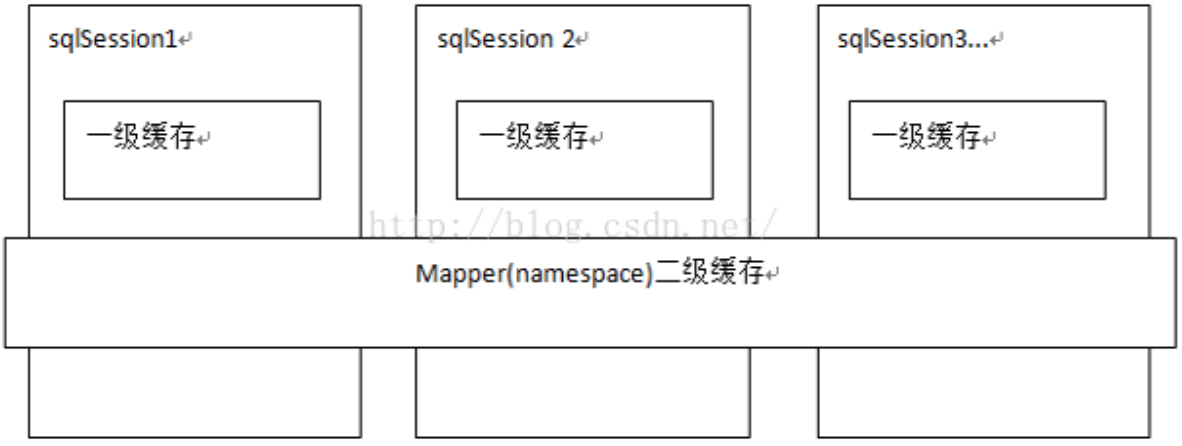
14.Mybatis和Hibernate区别?

Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句，不过mybatis可以通过XML或注解方式灵活配置要运行的sql语句，并将java对象和sql语句映射生成最终执行的sql，最后将sql执行的结果再映射生成java对象。

Mybatis学习门槛低，简单易学，程序员直接编写原生态sql，可严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套sql映射文件，工作量大。

Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用hibernate开发可以节省很多代码，提高效率。但是Hibernate的缺点是学习门槛高，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好Hibernate需要具有很强的经验和能力才行。
总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

15.一级、二级缓存?



一级缓存是SqlSession级别的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个(内存区域)数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

一级缓存的作用域是同一个SqlSession，在同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。Mybatis默认开启一级缓存。

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession去操作数据库得到数据会存在二级缓存区域，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

> 二级缓存是多个SqlSession共享的，其作用域是mapper的同一个namespace，不同的sqlSession两次执行相同namespace下的sql语句且向sql中传递参数也相同即最终执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis默认没有开启二级缓存需要在setting全局参数中配置开启二级缓存。

如果缓存中有数据就不用从数据库中获取，大大提高系统性能。

16.JDBC不足？Mybatis如何解决?

1、数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接池可解决此问题。

解决：在mybatis-config.xml中配置数据连接池，使用连接池管理数据库连接。

2、Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

3、向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决： Mybatis自动将java对象映射至sql语句。

4、对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象。

17.其他面试题

1、Mybatis映射枚举类

映射方式为自定义一个 `TypeHandler`，实现 `TypeHandler` 的 `setParameter()` 和 `getResult()` 接口方法。`TypeHandler` 有两个作用，一是完成从 javaType 至 jdbcType 的转换，二是完成 jdbcType 至 javaType 的转换，体现为 `setParameter()` 和 `getResult()` 两个方法，分别代表设置 sql 问号占位符参数和获取列查询结果。

2、三级缓存

- **一级缓存**

  Mybatis默认开启了一级缓存，一级缓存是在SqlSession 层面进行缓存的。即，同一个SqlSession，多次调用同一个Mapper和同一个方法的同一个参数，只会进行一次数据库查询，然后把数据缓存到缓冲中，以后直接先从缓存中取出数据，不会直接去查数据库。

  但是不同的SqlSession对象，因为不用的SqlSession都是相互隔离的，所以相同的Mapper、参数和方法，他还是会再次发送到SQL到数据库去执行，返回结果。

- **二级缓存**

  二级缓存是在sqlSessionFactory层面进行缓存的，各个sqlSession对象共享，默认不开启，需要手动设置配置文件

```
<cache eviction="LRU" flushInterval="100000" size="1024" readOnly="true"/>
```

- eviction：缓存回收策略
  - LRU：最少使用原则，移除最长时间不使用的对象
  - FIFO：先进先出原则，按照对象进入缓存顺序进行回收
  - SOFT：软引用，移除基于垃圾回收器状态和软引用规则的对象
  - WEAK：弱引用，更积极的移除移除基于垃圾回收器状态和弱引用规则的对象
- flushInterval：刷新时间间隔，单位为毫秒，这里配置的100毫秒。如果不配置，那么只有在进行数据库修改操作才会被动刷新缓存区
- size：引用额数目，代表缓存最多可以存储的对象个数
- readOnly：是否只读，如果为true，则所有相同的sql语句返回的是同一个对象（有助于提高性能，但并发操作同一条数据时，可能不安全），如果设置为false，则相同的sql，后面访问的是cache的clone副本。

可以在Mapper的具体方法下设置对二级缓存的访问意愿：

- useCache配置

  如果一条语句每次都需要最新的数据，就意味着每次都需要从数据库中查询数据，可以把这个属性设置为false，如：

```
<select id="selectAll" resultMap="BaseResultMap" useCache="false">
```

- 刷新缓存（就是清空缓存）

  二级缓存默认会在insert、update、delete操作后刷新缓存，可以手动配置不更新缓存，如下：

```
<update id="updateById" parameterType="User" flushCache="false" />
```

- **自定义缓存**

  定义缓存对象，该对象必须实现 *org.apache.ibatis.cache.Cache* 接口。

十一、RabbitMQ

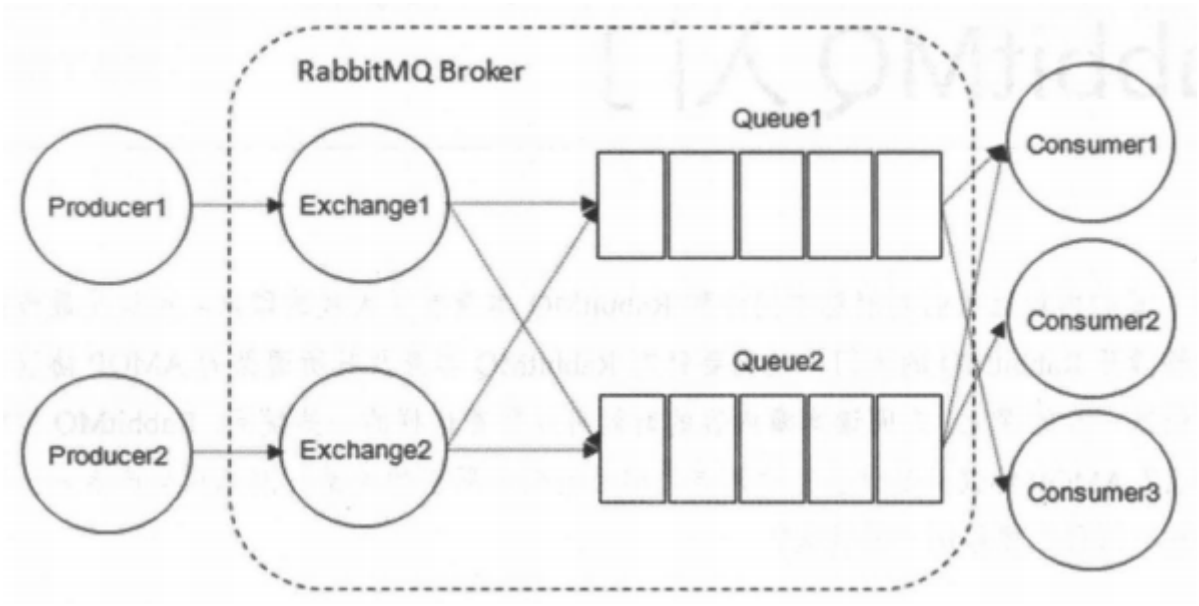
概念

RabbitMQ 是采用 Erlang 语言实现 AMQP(Advanced Message Queuing Protocol，高级消息队列协议）的消息中间件，用于在分布式系统中存储转发消息，实现了服务之间的高度解耦。

特点（优势）：

可靠性： RabbitMQ使用一些机制来保证消息的可靠性，如持久化、传输确认及发布确认等。
灵活的路由： 在消息进入队列之前，通过交换器来路由消息。
扩展性： 多个RabbitMQ节点可以组成一个集群，也可以根据实际业务情况动态地扩展集群中节点。
高可用性： 队列可以在集群中的机器上设置镜像，使得在部分节点出现问题的情况下队列仍然可用。
支持多种协议： RabbitMQ 除了原生支持 AMQP 协议，还支持 STOMP、MQTT 等多种消息中间件协议。
多语言客户端 、易用的管理界面、插件机制

核心概念：



交换机类型（广播类型）：

fanout：分发模式。它会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中，不需要做任何判断操作，所以 fanout 类型是所有的交换机类型里面速度最快的。fanout 类型常用来广播消息。

direct：默认方式。它会把消息路由到那些 Bindingkey 与 RoutingKey 完全匹配的 Queue 中。常用在处理有优先级的任务，根据任务的优先级把消息发送到对应的队列，这样可以指派更多的资源去处理高优先级的队列。

topic：模糊匹配路由键。订阅匹配模式，使用正则表达式匹配到消息队列。

headers：与 direct 类似，只是性能很差，此类型几乎用不到。

重要角色：

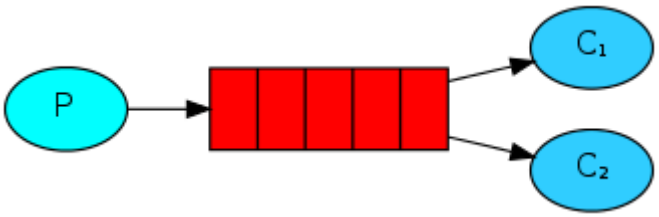
生产者：消息的创建者，负责创建和推送数据到消息服务器
消费者：消息的接收方，用于处理数据和确认消息
代理：就是RabbitMQ本身，用于扮演快递的角色，本身并不生产消息

重要组件：

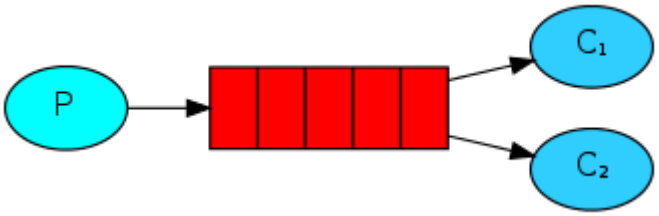
ConnectionFactory(连接管理器)：应用程序与RabbitMQ之间建立连接的管理器
Channel(信道)：消息推送使用的通道
Exchange(交换器)：用于接受、分配消息
Queue(队列)：用于存储生产者的消息
RoutingKey(路由键)：用于把生产者的数据分配到交换器上
BindKey(绑定键)：用于把交换器的消息绑定到队列上

工作模式：

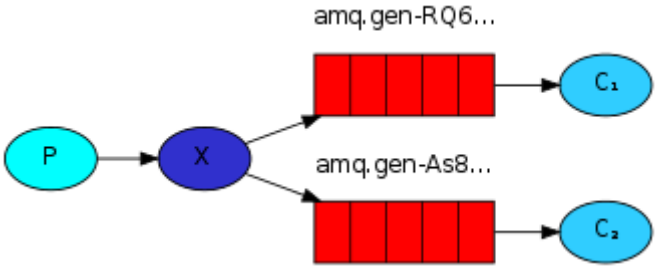
一.simple模式（即最简单的收发模式）：1.消息产生消息，将消息放入队列　　2.消息的消费者(consumer) 监听 消息队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列中删除(隐患 消息可能没有被消费者正确处理,已经从队列中消失了,造成消息的丢失，这里可以设置成手动的ack,但如果设置成手动ack，处理完后要及时发送ack消息给队列，否则会造成内存溢出)。



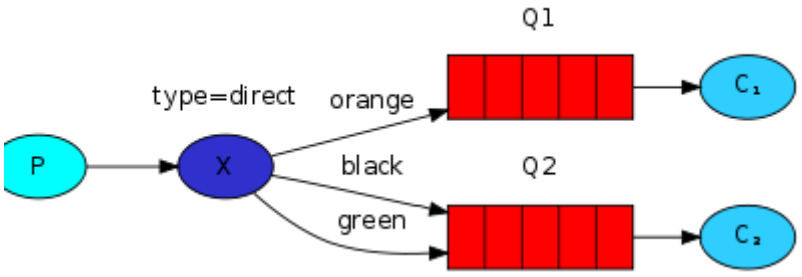
二.work工作模式(资源的竞争)：1.消息产生者将消息放入队列消费者可以有多个,消费者1,消费者2同时监听同一个队列,消息被消费。C1 C2共同争抢当前的消息队列内容,谁先拿到谁负责消费消息(隐患：高并发情况下,默认会产生某一个消息被多个消费者共同使用,可以设置一个开关(syncronize) 保证一条消息只能被一个消费者使用)。



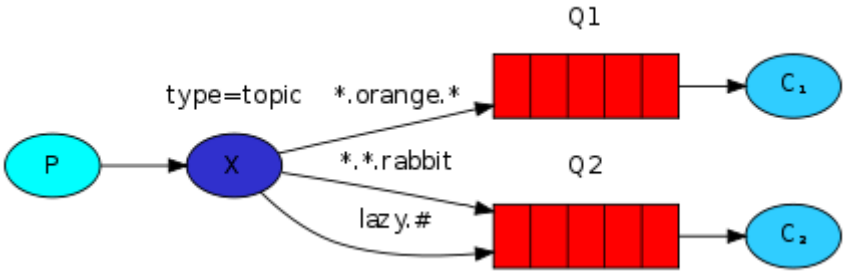
三.publish/subscribe发布订阅(共享资源)：1、每个消费者监听自己的队列；2、生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息。



四.routing路由模式：只有对应的路由key的队列才会接收到消息。



五.topic 主题模式(路由模式的一种)：正则表达式通配路由key。



1.使用场景?

应用解耦：订单系统和库存系统通过消息队列解耦（顺序消费、定时任务）

异步处理：发送短信和消息通知

流量削峰：秒杀活动使用消息队列可以控制活动人数和缓解短时间的高流量

2.vhost?

vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

3.消息是怎么发送的?

首先客户端必须连接到 RabbitMQ 服务器才能发布和消费消息，客户端和 rabbit server 之间会创建一个 tcp 连接，一旦 tcp 打开并通过了认证（认证就是你发送给 rabbit 服务器的用户名和密码），你的客户端和 RabbitMQ 就创建了一条 amqp 信道（channel），信道是创建在"真实" tcp 上的虚拟连接，amqp 命令都是通过信道发送出去的，每个信道都会有一个唯一的 id，不论是发布消息，订阅队列都是通过这个信道完成的。

4.怎么避免消息的丢失?

发布方确认：确保producer到broker节点消息的可靠性（ACK确认）

消费者确认：确保consumer成功消费了消息（ACK确认）

消息持久化：将message存储到硬盘

设置集群镜像模式：先说说三种部署模式：单节点、普通、镜像（把需要的队列做成镜像队列，存在于多个节点，属于RabbitMQ的HA方案）

消息补偿机制：这是最后的保证。消息补偿机制需要建立在消息要写入DB日志，发送日志，接受日志，两者的状态必须记录。

5.怎么保证消息的稳定性?

丢失数据可能有：生产者丢失、RabbitMQ自己丢失、消费者丢失

生产者解决：

1.提供了事务的功能。

2.开启信道的 confirm（ack）模式。

自己解决：

1.创建queue时就持久化

2.发送消息时将消息的deliveryMode设置为2，这样消息就会被设为持久化方式

消费者解决：

ack机制，每次处理完消息时手动ack

6.持久化成功条件?

要想持久化必须包含三部分：

1.交换器(Exchange)的持久化

2.队列(Queue)的持久化

3.消息(Message)的持久化

成功条件：

1.声明队列必须设置持久化 durable 设置为 true.
2.消息推送投递模式必须设置持久化，deliveryMode 设置为 2（持久）。
3.消息已经到达持久化交换器。
4.消息已经到达持久化队列。

以上四个条件都满足才能保证消息持久化成功。

7.持久化的缺点?

持久化的缺地就是降低了服务器的吞吐量，因为使用的是磁盘而非内存存储，从而降低了吞吐量。可尽量使用 ssd 固态硬盘来缓解吞吐量的问题。

8.怎么实现延迟消息队列?

延迟队列应用：订单过期、远程定时

- 通过消息过期后进入死信交换器，再由交换器转发到延迟消费队列，实现延迟功能；
- 使用 `RabbitMQ-delayed-message-exchange` 插件实现延迟功能。

9.集群有什么用?

- 高可用：某个服务器出现问题，整个 RabbitMQ 还可以继续使用；
- 高容量：集群可以承载更多的消息量。

10.节点类型有哪些?

- 磁盘节点：消息会存储到磁盘。
- 内存节点：消息都存储在内存中，重启服务器消息丢失，性能高于磁盘类型。

11.每个节点是其他节点的完整拷贝吗？为什么?

不是，原因有以下两个：

- 存储空间的考虑：如果每个节点都拥有所有队列的完全拷贝，这样新增节点不但没有新增存储空间，反而增加了更多的冗余数据；
- 性能的考虑：如果每条消息都需要完整拷贝到每一个集群节点，那新增节点并没有提升处理消息的能力，最多是保持和单节点相同的性能甚至是更糟。

12.集群中唯一的磁盘节点崩溃了会发生什么情况?

如果唯一磁盘的磁盘节点崩溃了，不能进行以下操作：

- 不能创建队列
- 不能创建交换器
- 不能创建绑定
- 不能添加用户
- 不能更改权限
- 不能添加和删除集群节点

唯一磁盘节点崩溃了，集群是可以保持运行的，但你不能更改任何东西。

13.对集群节点停止顺序有要求吗?

RabbitMQ 对集群的停止的顺序是有要求的，应该先关闭内存节点，最后再关闭磁盘节点。如果顺序恰好相反的话，可能会造成消息的丢失。

14.如何避免重复投递或重复消费?

在**消息生产**（重复投递）时，MQ 内部针对每条生产者发送的消息生成一个 `inner-msg-id`，作为去重的依据（消息投递失败并重传），避免重复的消息进入队列；
在**消息消费**（重复消费）时，要求消息体中必须要有一个 `bizId`（对于同一业务全局唯一，如支付 ID、订单 ID、帖子 ID 等）作为去重的依据，避免同一条消息被重复消费。对于数据库可以用唯一主键避免重复插入，对于redis可以用set集合避免重复插入，对于第三方接口需要我们手动去重。

15.消息基于什么传输?

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用**信道**的方式来传输数据。**信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。**

16.消息如何分发?

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。通过路由可实现多消费的功能。

17.RabbitMQ的缺点?

系统可用性降低：本来系统运行好好的，现在你非要加入个消息队列进去，那消息队列挂了，你的系统不是呵呵了

系统复杂度提高：加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，复杂性增大。

一致性问题：A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

18.如何解决消息队列的延时和过期失效问题?

其实本质针对的场景，都是说，可能你的消费端出了问题，不消费了；或者消费的速度极其慢，造成消息堆积了，MQ存储快要爆了，甚至开始过期失效删除数据了。

针对这个问题可以有事前、事中、事后三种处理

- 事前：开发预警程序，监控最大的可堆积消息数，超过就发预警消息（比如短信），不要等出生产事故了再处理。
- 事中：看看消费端是不是故障停止了，紧急重启。
- 事后：中华石杉老师就是说的这一种（https://github.com/doocs/advanced-java/blob/master/docs/high-concurrency/mq-time-delay-and-expired-failure.md），需要对消费端紧急扩容，增加处理消费者进程，如扩充10倍处理，但其实这也有个问题，即数据库的吞吐是有限制的，如果是消费到数据库也是没办法巨量扩容的，所以还是要在吞吐能力支持下老老实实的泄洪消 费。所以事前预防还是最重要的。否则出发删除过期数据，那就需要再重写生产消息的程序，重新产生消息。

19.面试题

1.MQ挂了怎么办?

1、一般MQ都是集群的，挂了一个其他的MQ都可以顶上去。

2、统一封装MQ的操作，做降级处理：当MQ挂了，将数据存储到数据库中。

3、做一个定时任务，定时查询数据库发送失败的消息做消息重发。

灾难面前，看我们想保什么：高可用还是数据一致性?

2.消息重复怎么办?

可能原因：生成者重复发送，消费者重复消费，但是由于是网络原因造成，因此重复消息不可避免，需要做的是保证消息的幂等性。

如何保证消息的幂等性?：

设置消息全局唯一id主键，redis setnx判断是否存在，或者数据库主键判断是否存在，如果存在就直接丢弃。如果是查询的消息则没影响，默认就是幂等性的。

消息积压怎么办?：

多弄几台机器配合多个消费者消费，如果对业务影响不大，可以先消费消息暂时不进行业务处理。

如何保证消息的消费顺序?：

rabbitmq：拆分多个queue，每个queue对应一个consumer；或者一个queue一个consumer，使用队列消费

kafka：一个topic一个partirion一个consumer，内部单线程消费

十二、Kafka

1、基础

1.Kafka？应用场景?

Kafka 是一个分布式流式处理平台。这到底是什么意思呢?

流平台具有三个关键功能:

1. **消息队列**：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原因。
2. **容错的持久方式存储记录消息流**： Kafka 会把消息持久化到磁盘，有效避免了消息丢失的风险·。
3. **流式处理平台：** 在消息发布的时候进行处理，Kafka 提供了一个完整的流式处理类库。

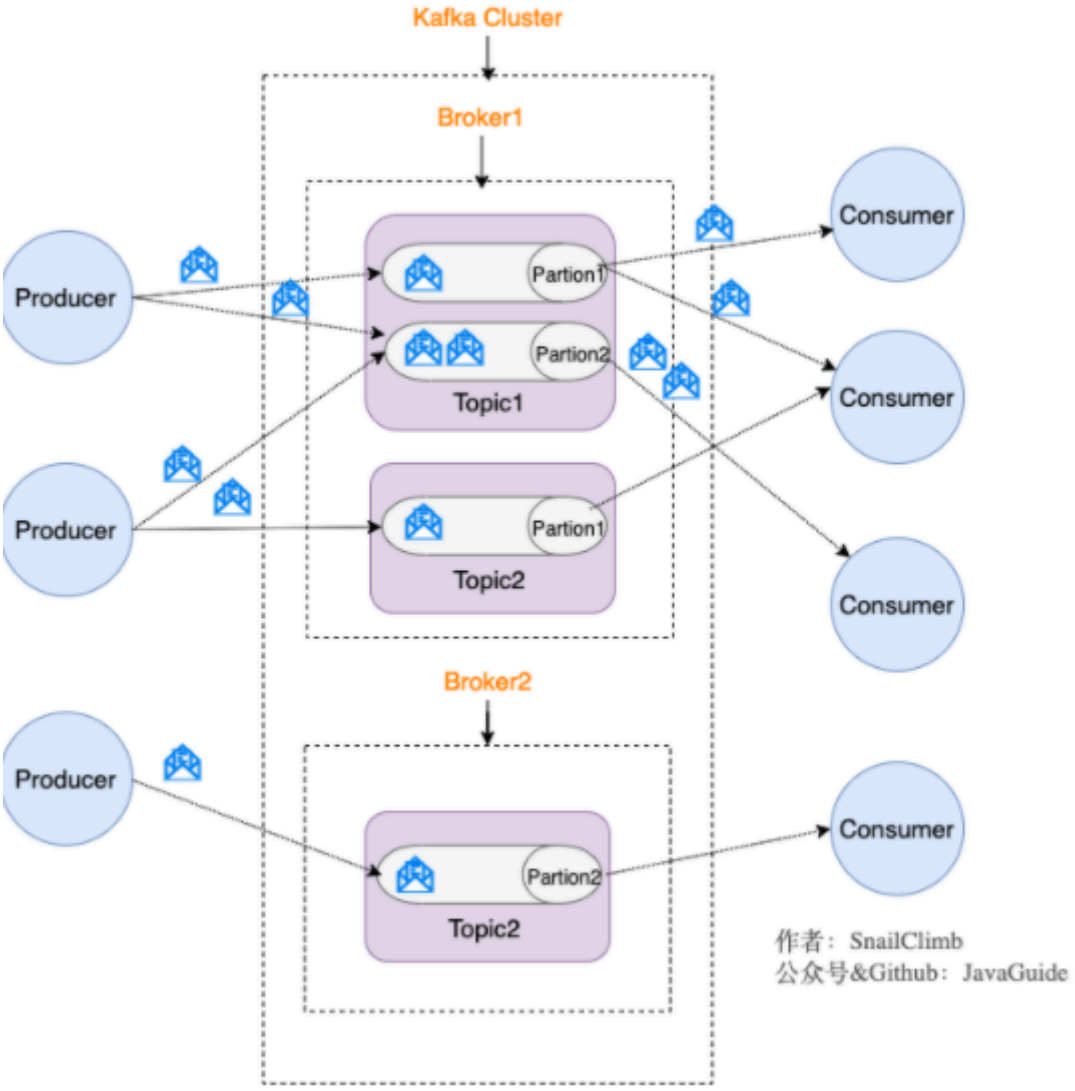
Kafka的优势:

1. **极致的性能** ： 基于 Scala 和 Java 语言开发，设计中大量使用了批量处理和异步的思想，最高可以每秒处理千万级别的消息。
2. **生态系统兼容性无可匹敌** ： Kafka 与周边生态系统的兼容性是最好的没有之一，尤其在大数据和流计算领域。

2、Kafka消息模型

队列模型的弊端：不方便将一个生产者产生的消息发送给多个消费者，并且每个消费者都能接收到完全的消息内容。

1）Kafak队列模型：发布-订阅



发布订阅模型（Pub-Sub） 使用**主题（Topic）** 作为消息通信载体，类似于**广播模式**；发布者发布一条消息，该消息通过主题传递给所有的订阅者，**在一条消息广播之后才订阅的用户则是收不到该条消息的。**

2）基本概念

1. **Producer（生产者）** : 产生消息的一方。
2. **Consumer（消费者）** : 消费消息的一方。
3. **Broker（代理）** : 可以看作是一个独立的 Kafka 实例。多个 Kafka Broker 组成一个 Kafka Cluster。
4. **Topic（主题）** : Producer 将消息发送到特定的主题，Consumer 通过订阅特定的 Topic(主题) 来消费消息。
5. **Partition（分区）** : Partition 属于 Topic 的一部分。一个 Topic 可以有多个 Partition ，并且同一 Topic 下的 Partition 可以分布在不同的 Broker 上，这也就表明一个 Topic 可以横跨多个 Broker 。（类似于消息队列）

3、多副本机制?

Kafka 为分区（Partition）引入了多副本（Replica）机制。分区（Partition）中的多个副本之间会有一个叫做 leader 的家伙，其他副本称为 follower。我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。

> 生产者和消费者只与 leader 副本交互。你可以理解为其他副本只是 leader 副本的拷贝，它们的存在只是为了保证消息存储的安全性。当 leader 副本发生故障时会从 follower 中选举出一个 leader,但是 follower 中如果有和 leader 同步程度达不到要求的参加不了 leader 的竞选。

Kafka 的多分区（Partition）以及多副本（Replica）机制有什么好处呢?

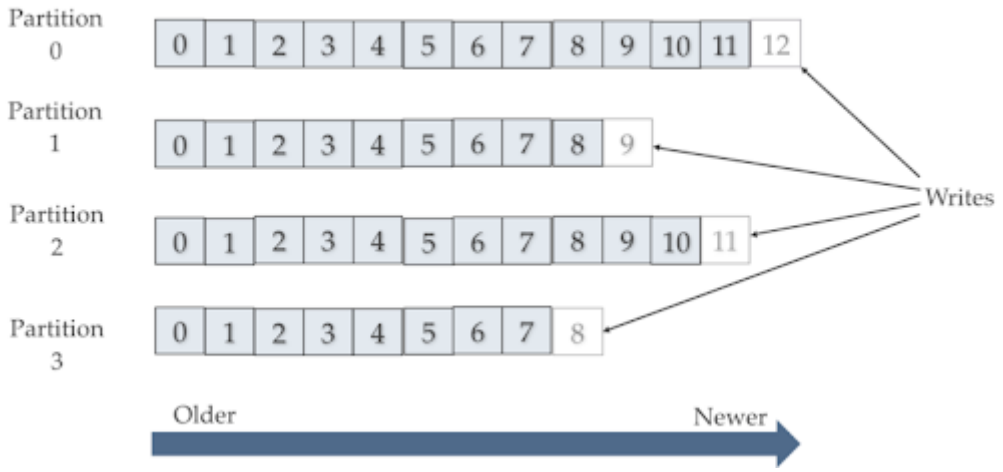
1. Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力（负载均衡）。
2. Partition 可以指定对应的 Replica 数, 这也极大地提高了消息存储的安全性, 提高了容灾能力，不过也相应的增加了所需要的存储空间。

4、Zookeeper为Kafka做的事情?

1. **Broker 注册**： 在 Zookeeper 上会有一个专门**用来进行 Broker 服务器列表记录**的节点。每个 Broker 在启动时，都会到 Zookeeper 上进行注册，即到/brokers/ids 下创建属于自己的节点。每个 Broker 就会将自己的 IP 地址和端口等信息记录到该节点中去
2. **Topic 注册**： 在 Kafka 中，同一个**Topic 的消息会被分成多个分区**并将其分布在多个 Broker 上，**这些分区信息及与 Broker 的对应关系**也都是由 Zookeeper 在维护。比如我创建了一个名字为 my-topic 的主题并且它有两个分区，对应到 zookeeper 中会创建这些文件夹：`/brokers/topics/my-topic/Partitions/0`、`/brokers/topics/my-topic/Partitions/1`
3. **负载均衡**： 上面也说过了 Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力。 对于同一个 Topic 的不同 Partition，Kafka 会尽力将这些 Partition 分布到不同的 Broker 服务器上。当生产者产生消息后也会尽量投递到不同 Broker 的 Partition 里面。当 Consumer 消费的时候，Zookeeper 可以根据当前的 Partition 数量以及 Consumer 数量来实现动态负载均衡。

5、如何保证消息顺序消费?

Kafka中Partition（分区）是真正保存消息的地方，而Partition（分区）又存放在Topic（主题）中，我们可以给特定Topic指定多个Partition。



每次添加消息到 Partition(分区) 的时候都会采用尾加法，Kafka 只能为我们保证 Partition(分区) 中的消息有序，而不能保证 Topic(主题) 中的 Partition(分区) 的有序。

> 消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量（offset）。Kafka 通过偏移量（offset）来保证消息在分区内的顺序性。

解决方案：

1. 1 个 Topic 只对应一个 Partition。
2. （推荐）发送消息的时候指定 key/Partition。

6、如何保证消息不丢失?

- 生成者：设置retries重试次数和重试间隔
- 消费者：手动关闭自动提交offset偏移量，可能会造成消费两次的问题
- kafka：分区多副本机制：leader和follower
  - 设置**acks=all**：acks=1，代表消息被leader副本接收之后就算被成功发送；acks=all，代表所有副本都接收到消息之后消息才被真正发送
  - 设置**replication.factor >= 3**：可以保证每个分区至少有三个副本
  - 设置**min.insync.replicas > 1**：表示消息至少要被写入到2个副本才算发送成功，应该避免使用默认值1
  - 设置**unclean.leader.election.enable = false**：TODO

2、面试题

1. 选举机制

> 大数据领域一般选举算法主要是Zab和Raft，都是Paxos算法的变种。
>
> ZAB选举四个阶段：leader选举、服务发现、数据同步、广播。

Kafka的主要选举机制有以下三种：

- 控制器（Broker Controller）选举

  主要作用是在Zookeeper的帮助下管理和协调整个Kafka集群。集群中任意一个Broker都能充当控制器的角色，但在运行过程中，只能有一个Broker成为控制器，可以认为Broker的选举，监听控制整个kafka集群，当有一台Broker宕机，会去决定将partition更新到哪一台上。
- 分区多副本选举
- 消费组选举

2. 最新版kafka已经放弃Zookeeper

将元数据存储到Kafka中，放弃存储到Zookeeper中。Quorum 控制器使用新的 KRaft 协议来确保元数据在仲裁中被精确地复制。这个协议在很多方面与 ZooKeeper 的 ZAB 协议和 Raft 相似。

3.Kafka如何保证高可用

多个broker组成集群，每个broker是一个节点；创建一个topic会产生多个partition，每个partition存在于不同的broker上，每个partition只存放一部分数据。所以说kafka是天然的分布式队列。

在 `Kafka 0.8` 版本之前，是没有 `HA` 机制的，当任何一个 `broker` 所在节点宕机了，这个 `broker` 上的 `partition` 就无法提供读写服务，所以这个版本之前，`Kafka` 没有什么高可用性可言。

在 `Kafka 0.8` 以后，提供了 `HA` 机制，就是 `replica` 副本机制。每个 `partition` 上的数据都会同步到其它机器，形成自己的多个 `replica` 副本。所有 `replica` 会选举一个 `leader` 出来，消息的生产者和消费者都跟这个 `leader` 打交道，其他 `replica` 作为 `follower`。写的时候，`leader` 会负责把数据同步到所有 `follower` 上去，读的时候就直接读 `leader` 上的数据即可。`Kafka` 负责均匀的将一个 `partition` 的所有 `replica` 分布在不同的机器上，这样才可以提高容错性。

> 当replica中的leader宕机后，会从follower中重新选举一个leader出来，这个新的leader会继续提供读写服务，达到所谓高可用。
>
> 写数据时，只会把数据写入leader节点，follower节点会同步数据，当leader收到所有follower的ack之后，就会返回写成功的消息给生产者。

4.Kafka和传统的MQ中间件区别

1、kafka有持久化日志，可以被重复读取和无限期保留
2、天然分布式，可以灵活收缩，通过replica副本机制保证高可用和数据容错能力
3、支持实时的流式处理

5.消费组

一个partition只能被不同消费组的一个消费者消费。

消费组机制一定程度上保证了消费者程序的高可用性。

6.Zookeeper在Kafka中的作用是什么?

Kafka使用ZK来管理元数据、成员管理、Controller选举。 等KIP-500提案完成后，Kafka将完全不再依赖于ZooKeeper ，使用Raft算法。

- “存放元数据”是指主题分区等数据都保存在 ZooKeeper 中，且以它保存的数据为权威，其他 “人” 都要与它保持对齐。
- “成员管理” 是指 Broker 节点的注册、注销以及属性变更，等等。
- “Controller 选举” 是指选举集群 Controller，而其他管理类任务包括但不限于主题删除、参数配置等。

7.offset作用

每个partition下的消息唯一标识位，一旦消息被写入了分区日志，它的位置值将不能更改。

8.Kafka为什么那么快?

- Cache Filesystem Cache PageCache缓存，分页存储，提高内存IO效率
- `顺序写入磁盘`：由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。不需要寻址，磁盘的顺序读写超过内存的随机读写
- `Zero-copy`：零拷技术减少拷贝次数
- `Batching of Messages`：批量量处理数据压缩。合并小的请求，然后以流的方式进行交互，直顶网络上限。
- `Pull 拉模式`：使用拉模式进行消息的获取消费，与消费端处理能力相符。

9.Kafka的producer发送数据ack=0,1,-1

1：表示leader成功收到消息后的确认
0：表示生产者发送消息出去后就不管了
-1： producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。当ISR中所有Replica都向Leader发送ACK时，leader才commit，这时候producer才能认为一个请求中的消息都commit

10.ISR、AR？ISR伸缩?

- `ISR`：In-Sync Replicas 副本同步队列
- `AR`:Assigned Replicas 所有副本

ISR是由leader维护，ISR维护了与leader信息一致的followerr的信息。follower从leader同步数据有一些延迟（包括`延迟时间replica.lag.time.max.ms`和`延迟条数replica.lag.max.messages`两个维度，当前最新的版本0.10.x中只支持`replica.lag.time.max.ms`这个维度），任意一个超过阈值都会把follower剔除出ISR，存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中。

> AR=ISR+OSR。

Kafka的同步不是完全同步，也不是完全异步，是一种特殊的ISR同步。

1、leader会维护一个与其保持同步的replica集合，该集合就是ISR，每个partition都有一个ISR
2、我们要保证kafka不会丢失消息，就要保证至少一个ISR存活，并且消息commit成功

ISR伸缩：任意一个超过阈值都会把follower剔除出ISR，存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中

11.Kafka的leader replica，follower replica?

Kafka的副本分为领导者副本、追随者副本，只有领导者副本才能对外提供读写服务，响应Client的请求。follower副本只是采取pull的方式，被动地同步leader副本的数据，并且在leader所在的broker宕机的时候，随时准备应聘leader副本。

> 加分点：
>
> 强调follower副本也能对外提供读服务。Kafka2.4版本之后可以允许follower副本有限地提供读服务。
>
> 强调leader和follower的消息序列在实际的场景中可能不一致。比如网络问题等，但是长时间不同步就要深入排查了。

12.replica分区的leader选举策略

不同场景对应一种不同的选举策略。

1、每当新建分区或者下线的分区重新上线，会进行leader选举
2、当手动运行 kafka-reassign-partitions命令， 或者是调用Admin的alterPartitionReassignments方法执行分区副本重分配时，会进行leader选举
3、 当你手动运行kafka-preferred-replica-election命令，或自动触发了Preferred Leader选举时，会进行leader选举
4、当broker正常关闭时，其他分区会受到影响此时需要leader选举

13.Kafka哪些场景使用了零拷贝

> 传统IO拷贝：从读磁盘到内核态的页缓存，再到用户态内存buffer（应用程序用户缓冲区），再到socket buffer缓冲区，再到网卡

零拷贝：直接在内核内完成输入输出，不需要再拷贝到用户空间去

零拷贝使用场景：基于mmap索引和日志文件读写所用的TransportLayer

- mmap

  将磁盘文件映射到内存，用户通过修改内存就能修改磁盘文件。使用操作系统的页缓存来实现文件到物理内存的直接映射。但是mmap不可靠，只有当操作系统主动调用flush的时候才会真正写入数据到磁盘。

- TransportLayer

  TransportLayer是Kafka传输层的接口 。底层方法使用sendfile（将磁盘数据读到内核缓冲区直接返回给网卡）实现零拷贝，如果IO通道使用PLAINTEST，就可以使用零拷贝特性；如果IO通道使用SSL，就不能使用零拷贝特性。

14.为什么Kafka不支持读写分离

- 数据一致性问题
- 延时问题。Kafka中主从同步会比Redis更加耗时，因为要持久化到磁盘，对于延时敏感并不可靠

十三、ZooKeeper

1.概念

个人理解：zk=文件系统+通知机制

1、Zookeeper 可以被用作注册中心；
2、Zookeeper 是 Hadoop 生态系统的一员；
3、构建 Zookeeper 集群的时候，使用的服务器最好是奇数台；
4、用来进行分布式环境的协调，是一个典型的分布式一致性解决方案；**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。**

1.1 结合个人使用情况讲一下Zookeeper?

在我自己做过的项目中，主要使用到了 ZooKeeper 作为 Dubbo 的注册中心(Dubbo 官方推荐使用 ZooKeeper注册中心)。另外在搭建 solr 集群的时候，我使用 ZooKeeper 作为 solr 集群的管理工具。这时，ZooKeeper 主要提供下面几个功能：1、集群管理：容错、负载均衡。2、配置文件的集中管理 。3、集群的入口。

我个人觉得在使用 ZooKeeper 的时候，最好是使用 集群版的 ZooKeeper 而不是单机版的。官网给出的架构图就描述的是一个集群版的 ZooKeeper 。通常 3 台服务器就可以构成一个 ZooKeeper 集群了。

1.2 为什么最好使用奇数台Zookeeper集群?

所谓的zookeeper容错是指，当宕掉几个zookeeper服务器之后，剩下的个数必须大于宕掉的个数的话整个zookeeper才依然可用。所以5台和6台都是只允许最大宕机掉两台。。。没必要增加一个多余的Zookeeper。

1.3 重要概念

- **ZooKeeper 本身就是一个分布式程序（只要半数以上节点存活，ZooKeeper 就能正常服务）。**
- **为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么 ZooKeeper 本身仍然是可用的。**
- **ZooKeeper 将数据保存在内存中，这也就保证了 高吞吐量和低延迟**（但是内存限制了能够存储的容量不太大，此限制也是保持znode中存储的数据量较小的进一步原因）。
- **ZooKeeper 是高性能的。 在"读"多于"写"的应用程序中尤其地高性能，因为"写"会导致所有的服务器间同步状态。**（"读"多于"写"是协调服务的典型场景。）
- **ZooKeeper有临时节点的概念。 当创建临时节点的客户端会话一直保持活动，瞬时节点就一直存在。而当会话终结时，瞬时节点被删除。持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。**
- ZooKeeper 底层其实只提供了两个功能：①管理（存储、读取）用户程序提交的数据；②为用户程序提供数据节点监听服务。

1.4 Session会话

Session 指的是 ZooKeeper 服务器与客户端会话。**在 ZooKeeper 中，一个客户端连接是指客户端和服务器之间的一个 TCP 长连接**。客户端启动的时候，首先会与服务器建立一个 TCP 连接，从第一次连接建立开始，客户端会话的生命周期也开始了。**通过这个连接，客户端能够通过心跳检测与服务器保持有效的会话，也能够向Zookeeper服务器发送请求并接受响应，同时还能够通过该连接接收来自服务器的Watch事件通知。** Session的`sessionTimeout`值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，**只要在`sessionTimeout`规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话仍然有效。**

在为客户端创建会话之前，服务端首先会为每个客户端都分配一个sessionID。由于 sessionID 是 Zookeeper 会话的一个重要标识，许多与会话相关的运行机制都是基于这个 sessionID 的，因此，无论是哪台服务器为客户端分配的 sessionID，都务必保证全局唯一。

1.5 Znode

在Zookeeper中，"节点"分为两类，第一类同样是指构成集群的机器，我们称之为机器节点；第二类则是指数据模型中的数据单元，我们称之为数据节点——ZNode。

Zookeeper将所有数据存储在内存中，数据模型是一棵树（Znode Tree），由斜杠（/）的进行分割的路径，就是一个Znode，例如/foo/path1。每个上都会保存自己的数据内容，同时还会保存一系列属性信息。

在Zookeeper中，node可以分为持久节点和临时节点两类。所谓持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。 另外，ZooKeeper还允许用户为每个节点添加一个特殊的属性：**SEQUENTIAL**.一旦节点被标记上这个属性，那么在这个节点被创建的时候，Zookeeper会自动在其节点名后面追加上一个整型数字，这个整型数字是一个由父节点维护的自增数字。

每个 znode 由 2 部分组成:

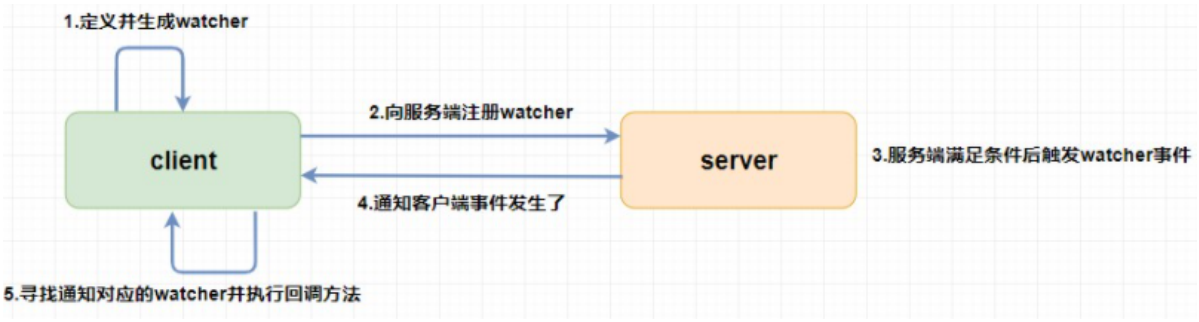
- **stat**： 状态信息，记录各种版本信息
- **data**： 节点存放的数据的具体内容

1.6 版本

在前面我们已经提到，Zookeeper 的每个 ZNode 上都会存储数据，对应于每个ZNode，Zookeeper 都会为其维护一个叫作 **Stat** 的数据结构，Stat 中记录了这个 ZNode 的三个数据版本，分别是version（当前ZNode的版本）、cversion（当前ZNode子节点的版本）和 aversion（当前ZNode的ACL版本）。

1.7 Watcher

Watcher（事件监听器），是Zookeeper中的一个很重要的特性。Zookeeper允许用户在指定节点上注册一些Watcher，并且在一些特定事件触发的时候，ZooKeeper服务端会将事件通知到感兴趣的客户端上去，该机制是Zookeeper实现分布式协调服务的重要特性。



1.8 ACL

Zookeeper采用ACL（AccessControlLists）策略来进行权限控制，类似于 UNIX 文件系统的权限控制。Zookeeper 定义了如下5种权限。

- **CREATE**：创建子节点的权限。
- **READ**：获取节点数据和子节点列表的权限。
- **WRITE**：更新节点数据的权限。
- **DELETE**：删除子节点的权限。
- **ADMIN**：设置节点 ACL 的权限。

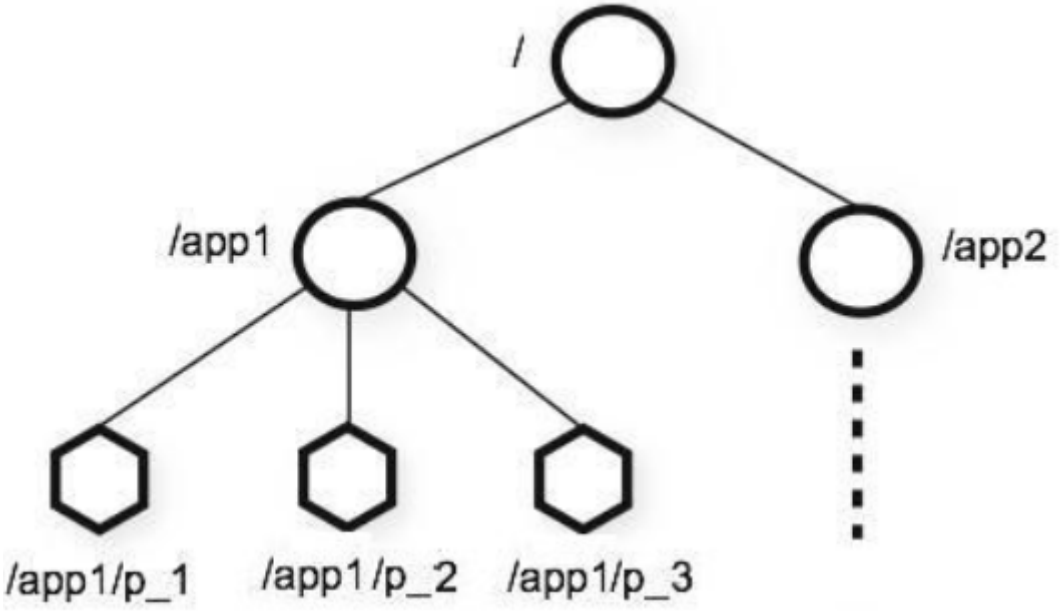
2.Zookeeper的特点

- **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。
- **单一视图**： 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。

3.Zookeeper设计目标

3.1 简单的数据模型

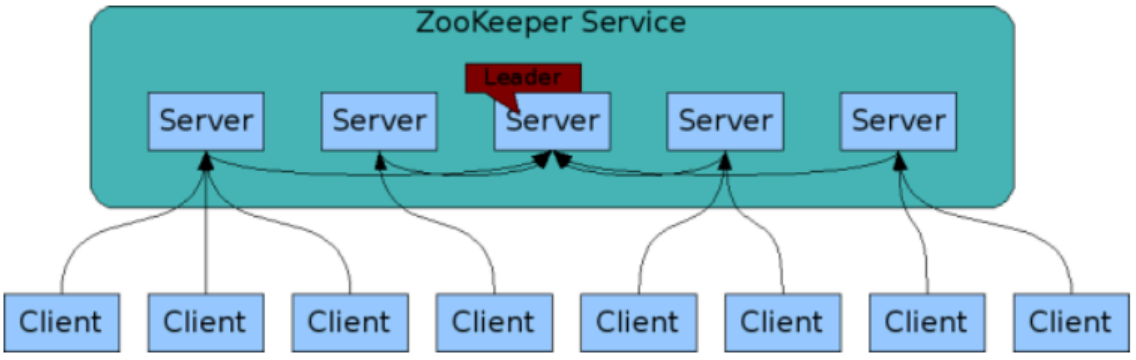
ZooKeeper 允许分布式进程通过共享的层次结构命名空间进行相互协调，这与标准文件系统类似。 名称空间由 ZooKeeper 中的数据寄存器组成 - 称为znode，这些类似于文件和目录。 与为存储设计的典型文件系统不同，ZooKeeper数据保存在内存中，这意味着ZooKeeper可以实现高吞吐量和低延迟。



3.2 可构建集群

为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么zookeeper本身仍然是可用的。 客户端在使用 ZooKeeper 时，需要知道集群机器列表，通过与集群中的某一台机器建立 TCP 连接来使用服务，客户端使用这个TCP链接来发送请求、获取结果、获取监听事件以及发送心跳包。如果这个连接异常断开了，客户端可以连接到另外的机器上。引入了 Leader、Follower 和 Observer 三种角色

ZooKeeper 官方提供的架构图：



上图中每一个Server代表一个安装Zookeeper服务的服务器。组成 ZooKeeper 服务的服务器都会在内存中维护当前的服务器状态，并且每台服务器之间都互相保持着通信。**集群间通过 Zab 协议（Zookeeper Atomic Broadcast）来保持数据的一致性。**

3.3 顺序访问

对于来自客户端的每个更新请求，ZooKeeper 都会分配一个全局唯一的递增编号，这个编号反应了所有事务操作的先后顺序，应用程序可以使用 ZooKeeper 这个特性来实现更高层次的同步原语。 **这个编号也叫做时间戳——zxid（Zookeeper Transaction Id）**

1）事务一致性

zk采用递增的事务id号来标识事务，所有的提议proposal都会在被提出的时候加上zxid，zxid实际上是一个64位的数字，高 32 位是 epoch （时期; 纪元; 世; 新时代）用来标识 leader 是否发生改变，如果有新的 leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

3.4 高性能

ZooKeeper 是高性能的。 在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。（“读”多于“写”是协调服务的典型场景。）

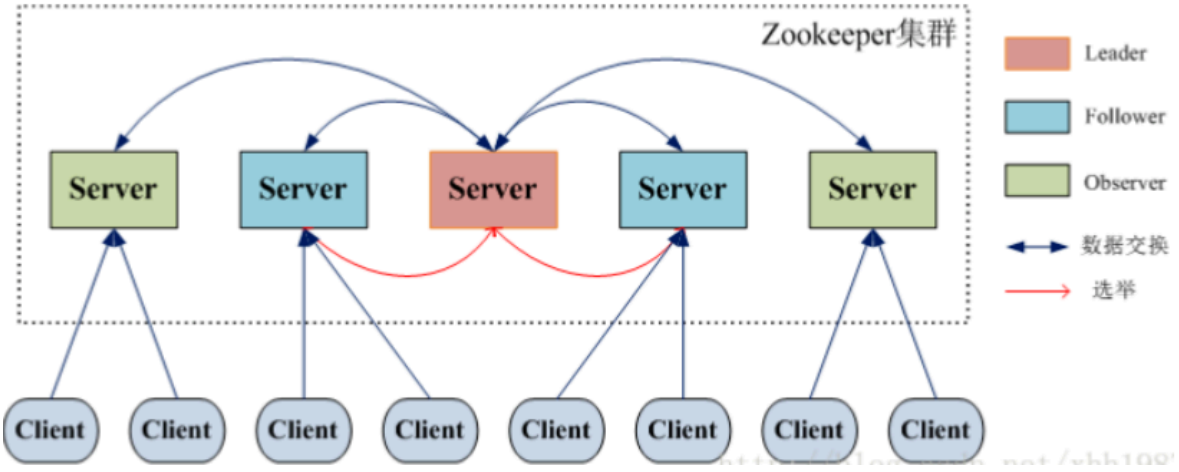
3.5 Zookeeper集群中的服务器状态

- **LOOKING**： 寻找 Leader。
- **LEADING**： Leader 状态，对应的节点为 Leader。
- **FOLLOWING**： Follower 状态，对应的节点为 Follower。
- **OBSERVING**： Observer 状态，对应节点为 Observer，该节点不参与 Leader 选举。

4.集群角色介绍

最典型集群模式： Master/Slave 模式（主备模式）。在这种模式中，通常 Master服务器作为主服务器提供写服务，其他的 Slave 服务器从服务器通过异步复制的方式获取 Master 服务器最新的数据提供读服务。

但是，**在 ZooKeeper 中没有选择传统的 Master/Slave 概念，而是引入了Leader、Follower 和 Observer 三种角色。**如下图所示



ZooKeeper 集群中的所有机器通过一个 Leader 选举过程来选定一台称为 “Leader” 的机器，Leader 既可以为客户端提供写服务又能提供读服务。除了 Leader 外，Follower 和 Observer 都只能提供读服务。Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。

角色		主要工作描述
领导者		1. 事务请求的唯一调度和处理者，保证集群事务处理的顺序性； 2. 集群内部各服务器的调度者
学习者（Learner）	跟随者（Follower）	1. 处理客户端非事务请求，转发事务请求给Leader服务器 2. 参与事务请求Proposal的投票 3. 参与Leader选举的投票
	观察者（Observer）	Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。
客户端（Client）		请求发起方

当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进人恢复模式并选举产生新的Leader服务器。这个过程大致是这样的（数据一致性）：

1. Leader election（选举阶段）：节点在一开始都处于选举阶段，只要有一个节点得到超半数节点的票数，它就可以当选准 leader。
2. Discovery（发现阶段）：在这个阶段，followers 跟准 leader 进行通信，同步 followers 最近接收的事务提议。
3. Synchronization（同步阶段）:同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。同步完成之后 准 leader 才会成为真正的 leader。
4. Broadcast（广播阶段） 到了这个阶段，Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。

一致性协议具体过程

- **选举leader**
  1、每个follower广播自己事务队列中最大事务编号maxId
  2、获取集群中其他follower发出来的maxId，选取出最大的maxId所属的follower，投票给该follower，选它为leader。
  3、统计所有投票，获取投票数超过一半的follower被推选为leader
- **同步数据**
  1、各个follower向leader发送自己保存的任期E
  2、leader比较所有的任期，选取最大的E，加1后作为当前的任期E=E+1
  3、将任务E广播给所有follower
  4、follower将任期改为leader发过来的值，并且返回给leader事务队列L
  5、leader从队列集合中选取任期最大的队列，如果有多个队列任期都是最大，则选取事务编号n最大的队列Lmax。将Lmax最为leader队列，并且广播给各个follower。
  6、follower接收队列替换自己的事务队列，并且执行提交队列中的事务。至此各个节点的数据达成一致，zookeeper恢复正常服务。
- **广播**
  1、leader节点接收到请求，将事务加入事务队列，并且将事务广播给各个follower。
  2、follower接收事务并加入都事务队列，然后给leader发送准备提交请求。
  3、leader 接收到半数以上的准备提交请求后，提交事务同时向follower 发送提交事务请求
  4、follower提交事务

5.ZAB协议和Paxos算法

5.1 ZAB协议

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。 在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

ZAB协议的两种基本模式：崩溃恢复和消息广播

ZAB协议包括两种基本的模式，分别是 **崩溃恢复和消息广播。**当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进人恢复模式并选举产生新的Leader服务器。当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该Leader服务器完成了状态同步之后，ZAB协议就会退出恢复模式。其中，**所谓的状态同步是指数据同步，用来保证集群中存在过半的机器能够和Leader服务器的数据状态保持一致。**

当集群中已经有过半的Follower服务器完成了和Leader服务器的状态同步，那么整个服务框架就可以进人消息广播模式了。 当一台同样遵守ZAB协议的服务器启动后加人到集群中时，如果此时集群中已经存在一个Leader服务器在负责进行消息广播，那么新加人的服务器就会自觉地进人数据恢复模式：找到Leader所在的服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。正如上文介绍中所说的，ZooKeeper设计成只允许唯一的一个Leader服务器来进行事务请求的处理。Leader服务器在接收到客户端的事务请求后，会生成对应的事务提案并发起一轮广播协议；而如果集群中的其他机器接收到客户端的事务请求，那么这些非Leader服务器会首先将这个事务请求转发给Leader服务器。

6.面试题

6.1 Redis做分布式锁和Zookeeper分布式锁的区别?

Redis：setnx存入key来确保唯一性；
Zookeeper：他是一个分布式协调工具，当多个客户端同时在zookeeper上创建相同的一个临时节点，因为临时节点**路径是保证唯一**，只要谁能够创建节点成功，谁就能够获取到锁，没有创建成功节点，就会进行等待，当释放锁的时候，采用事件通知给客户端重新获取锁的资源；

6.2 介绍一下平时用的Zookeeper?

- 有使用过的，使用ZooKeeper作为**dubbo的注册中心**，使用ZooKeeper实现**分布式锁。**
- ZooKeeper，它是一个开放源码的**分布式协调服务**，它是一个集群的管理者，它将简单易用的接口提供给用户。
- 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**用途**：命名服务、配置管理、集群管理、分布式锁、队列管理

6.3 什么是命名服务？配置管理？集群管理?

命名服务：

> 命名服务是指通过**指定的名字**来获取资源或者服务地址。Zookeeper可以创建一个**全局唯一的路径**，这个路径就可以作为一个名字。被命名的实体可以是**集群中的机器，服务的地址，或者是远程的对象**等。一些分布式服务框架（RPC、RMI）中的服务地址列表，通过使用命名服务，客户端应用能够根据特定的名字来获取资源的实体、服务地址和提供者信息等。

配置管理：

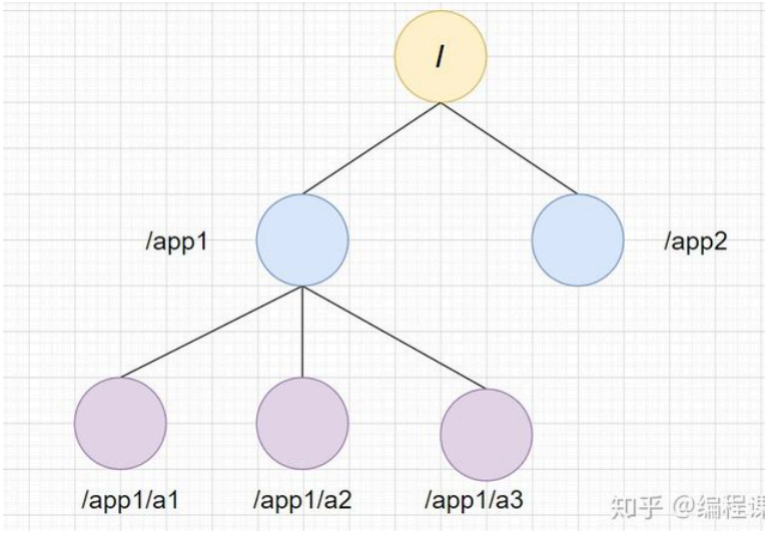
> 实际项目开发中，我们经常使用.properties或者xml需要配置很多信息，如数据库连接信息、fps地址端口等等。因为你的程序一般是分布式部署在不同的机器上（如果你是单机应用当我没说），如果把程序的这些配置信息**保存在zk的znode节点**下，当你要修改配置，即znode会发生变化时，可以通过改变zk中某个目录节点的内容，利用**watcher通知给各个客户端**，从而更改配置。

集群管理：

> 集群管理包括集群监控和集群控制，其实就是监控集群机器状态，剔除机器和加入机器。
> zookeeper可以方便集群机器的管理，它可以实时监控znode节点的变化，一旦发现有机器挂了，该机器就会与zk断开连接，对用的临时目录节点会被删除，其他所有机器都收到通知。新机器加入也是类似酱紫，所有机器收到通知：有新兄弟目录加入啦。

6.4 Znode节点类型?

zookeeper的节点统一叫做**znode**，它是可以通过**路径来标识**，结构图如下：



根据节点的生命周期，Znode分为4种类型，分别为持久节点（PERSISTENT）、持久顺序节点（PERSISTENT_SEQUENTIAL）、临时节点（EPHEMERAL）、临时顺序节点（EPHEMERAL_SEQUENTIAL）

6.5 Zookeeper里面存储的是什么?

znode节点里面存储的是什么?

=Znode数据节点==的代码如下=

```
public class DataNode implements Record {
    byte data[];
    Long acl;
    public StatPersisted stat;
    private Set<String> children = null;
}
```

Znode包含了**存储数据、访问权限、子节点引用、节点状态信息**.

- **data:** znode存储的业务数据信息
- **ACL:** 记录客户端对znode节点的访问权限，如IP等。
- **children:** 当前节点的子节点引用
- **stat:** 包含Znode节点的状态信息，比如**事务id、版本号、时间戳**等等。

6.6 每个节点的数据最大不能超过多少呢?

为了保证高吞吐和低延迟，以及数据的一致性，znode只适合存储非常小的数据，**不能超过1M，最好都小于1K。**

6.7 Znode节点的监听机制?

Zookeeper 允许客户端向服务端的某个Znode注册一个Watcher监听，当服务端的一些指定事件触发了这个Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据Watcher通知状态和事件类型做出业务上的改变。

> 可以把Watcher理解成客户端注册在某个Znode上的触发器，当这个Znode节点发生变化时（增删改查），就会触发Znode对应的注册事件，注册的客户端就会收到异步通知，然后做出业务的改变。

7.一致性协议和算法

2PC（两阶段提交），3PC（三阶段提交），Paxos算法等

1、2PC两阶段提交

第一阶段：当要执行一个分布式事务的时候，事务发起者首先向协调者发起事务请求，然后协调者会给所有参与者发送 `prepare` 请求（其中包括事务内容）告诉参与者你们需要执行事务了，如果能执行我发的事务内容那么就先执行但不提交，执行后请给我回复。然后参与者收到 `prepare` 消息后，他们会开始执行事务（但不提交），并将 `Undo` 和 `Redo` 信息记入事务日志中，之后参与者就向协调者反馈是否准备好了。

第二阶段：第二阶段主要是协调者根据参与者反馈的情况来决定接下来是否可以进行事务的提交操作，即提交事务或者回滚事务。

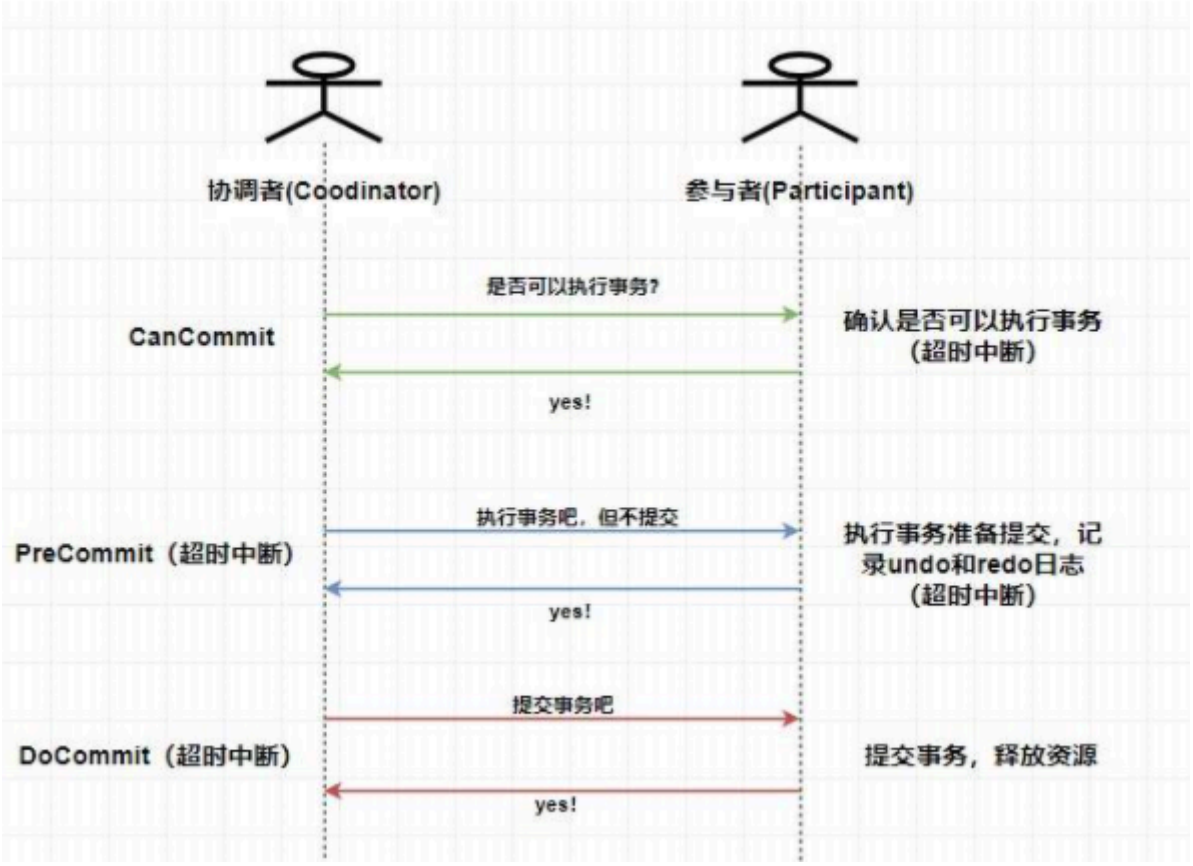
存在的问题

- **单点故障问题**，如果协调者挂了那么整个系统都处于不可用的状态了。
- **阻塞问题**，即当协调者发送 `prepare` 请求，参与者收到之后如果能处理那么它将会进行事务的处理但并不提交，这个时候会一直占用着资源不释放，如果此时协调者挂了，那么这些资源都不会再释放了，这会极大影响性能。
- **数据不一致问题**，比如当第二阶段，协调者只发送了一部分的 `commit` 请求就挂了，那么也就意味着，收到消息的参与者会进行事务的提交，而后面没收到的则不会进行事务提交，那么这时候就会产生数据不一致性问题。

2、3PC三阶段提交

1. **CanCommit阶段**：协调者向所有参与者发送 `CanCommit` 请求，参与者收到请求后会根据自身情况查看是否能执行事务，如果可以则返回 YES 响应并进入预备状态，否则返回 NO 。
2. **PreCommit阶段**：协调者根据参与者返回的响应来决定是否可以进行下面的 `PreCommit` 操作。如果上面参与者返回的都是 YES，那么协调者将向所有参与者发送 `PreCommit` 预提交请求，**参与者收到预提交请求后，会进行事务的执行操作，并将 `Undo` 和 `Redo` 信息写入事务日志中** ，最后如果参与者顺利执行了事务则给协调者返回成功的响应。如果在第一阶段协调者收到了 **任何一个 NO** 的信息，或者 **在一定时间内** 并没有收到全部的参与者的响应，那么就会中断事务，它会向所有参与者发送中断请求（abort），参与者收到中断请求之后会立即中断事务，或者在一定时间内没有收到协调者的请求，它也会中断事务。

3. **DoCommit阶段**：这个阶段其实和 `2PC` 的第二阶段差不多，如果协调者收到了所有参与者在 `PreCommit` 阶段的 YES 响应，那么协调者将会给所有参与者发送 `DoCommit` 请求，**参与者收到 `DoCommit` 请求后则会进行事务的提交工作**，完成后则会给协调者返回响应，协调者收到所有参与者返回的事务提交成功的响应之后则完成事务。若协调者在 `PreCommit` 阶段 **收到了任何一个 NO 或者在一定时间内没有收到所有参与者的响应** ，那么就会进行中断请求的发送，参与者收到中断请求后则会 **通过上面记录的回滚日志** 来进行事务的回滚操作，并向协调者反馈回滚状况，协调者收到参与者返回的消息后，中断事务。



> 总结：`3PC` 在很多地方进行了超时中断的处理，可以终止正在执行的事务，可以减少同步阻塞的时间

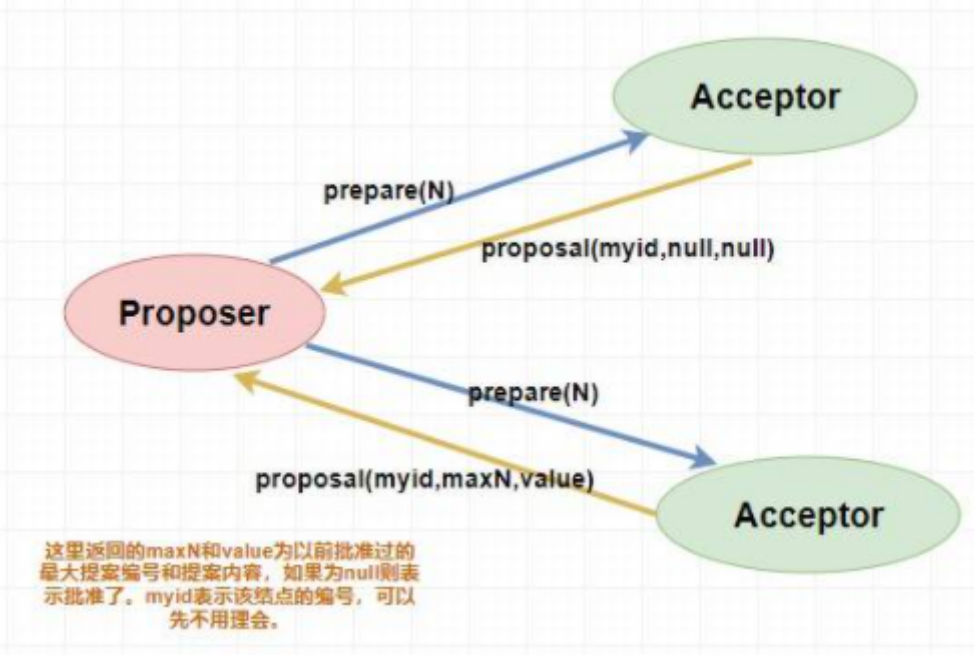
3、Paxos算法

`Paxos` 算法是基于**消息传递且具有高度容错特性的一致性算法**，是目前公认的解决分布式一致性问题最有效的算法之一，**其解决的问题就是在分布式系统中如何就某个值（决议）达成一致** 。

在 `Paxos` 中主要有三个角色，分别为 `Proposer提案者` 、`Acceptor表决者` 、`Learner学习者` 。`Paxos` 算法和 `2PC` 一样，也有两个阶段，分别为 `Prepare` 和 `accept` 阶段。

prepare阶段

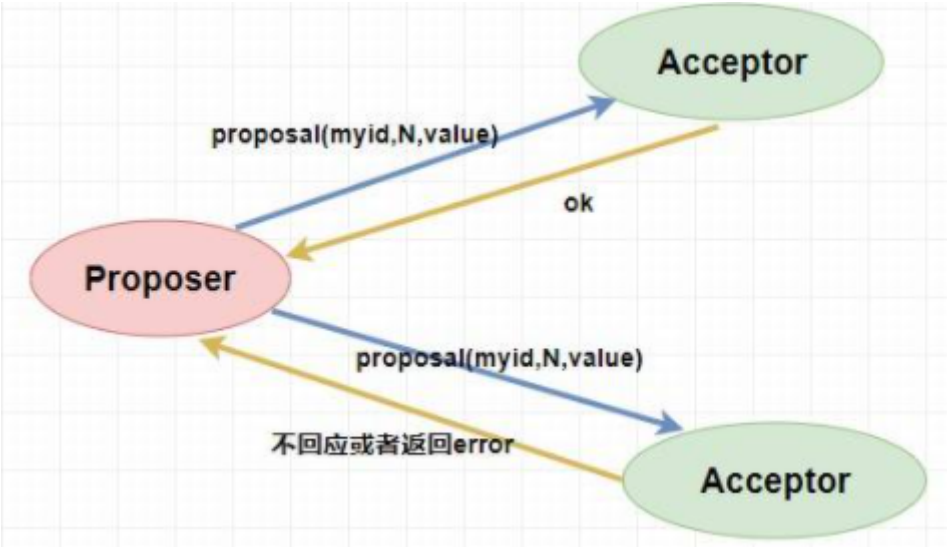
- `Proposer提案者`：负责提出 `proposal`，每个提案者在提出提案时都会首先获取到一个 **具有全局唯一性的、递增的提案编号N**，即在整个集群中是唯一的编号 N，然后将该编号赋予其要提出的提案，在**第一阶段是只将提案编号发送给所有的表决者。**
- `Acceptor表决者`：每个表决者在 `accept` 某提案后，会将该提案编号N记录在本地，这样每个表决者中保存的已经被 accept 的提案中会存在一个**编号最大的提案**，其编号假设为 `maxN` 。每个表决者仅会 `accept` 编号大于自己本地 `maxN` 的提案，在批准提案时表决者会将以前接受过的最大编号的提案作为响应反馈给 `Proposer` 。



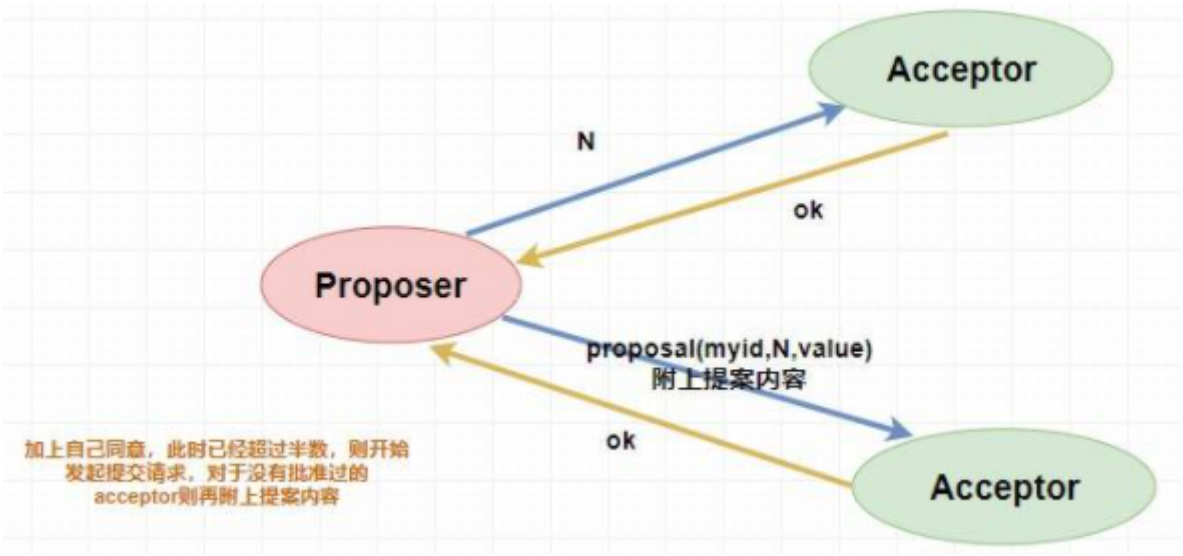
accept阶段

当一个提案被 `Proposer` 提出后，如果 `Proposer` 收到了超过半数的 `Acceptor` 的批准（`Proposer` 本身同意），那么此时 `Proposer` 会给所有的 `Acceptor` 发送真正的提案（你可以理解为第一阶段为试探），这个时候 `Proposer` 就会发送提案的内容和提案编号。

表决者收到提案请求后会再次比较本身已经批准过的最大提案编号和该提案编号，如果该提案编号 **大于等于** 已经批准过的最大提案编号，那么就 `accept` 该提案（此时执行提案内容但不提交），随后将情况返回给 `Proposer` 。如果不满足则不回应或者返回 NO 。



当 `Proposer` 收到超过半数的 `accept` ，那么它这个时候会向所有的 `acceptor` 发送提案的提交请求。需要注意的是，因为上述仅仅是超过半数的 `acceptor` 批准执行了该提案内容，其他没有批准的并没有执行该提案内容，所以这个时候需要**向未批准的 `acceptor` 发送提案内容和提案编号并让它无条件执行和提交**，而对于前面已经批准过该提案的 `acceptor` 来说 **仅仅需要发送该提案的编号** ，让 `acceptor` 执行提交就行了。



而如果 `Proposer` 如果没有收到超过半数的 `accept` 那么它将会将 **递增** 该 `Proposal` 的编号，然后 **重新进入** `Prepare` **阶段** 。

8、安装注意

Zookeeper集群安装：

2888是用于zk选举端口； 3888是zk服务器间的通信端口

主要步骤：修改配置文件（存储路径、id标识）、创建myid文件、启动zk集群

Kafka集群安装：

修改配置文件、启动kafka集群

十四、MySQL

1、简介

MySQL 是一种关系型数据库，是开源免费的，并且方便扩展。MySQL的默认端口号是**3306。**

1.存储引擎?

查看MySQL提供的所有存储引擎

```
mysql> show engines;
//查看默认存储引擎
mysql> show variables like '%storage_engine%';
//查看表的存储引擎
show table status like "table_name" ;
```

1）MyISAM和InnoDB区别

MyISAM是MySQL的默认数据库引擎（5.5版之前）。虽然性能极佳，而且提供了大量的特性，包括**全文索引、压缩、空间函数**等，但MyISAM不支持事务和行级锁，而且**最大的缺陷就是崩溃后无法安全恢复。**不过，5.5版本之后，MySQL引入了InnoDB（事务性数据库引擎），MySQL 5.5版本后默认的存储引擎为InnoDB。

大多数时候我们使用的都是 InnoDB 存储引擎，但是在某些情况下使用 MyISAM 也是合适的比如读密集的情况下。（如果你不介意 MyISAM 崩溃恢复问题的话）。MyISAM内部维护了一个计数器，select(*)更快。

两者的对比：

1>.InnoDB支持事务，而MyISAM不支持事务

2>.InnoDB支持行级锁，而MyISAM支持表级锁

3>.InnoDB支持MVCC, 而MyISAM不支持

4>.InnoDB支持外键，而MyISAM不支持

5>.InnoDB不支持全文索引，而MyISAM支持

6>.InnoDB是聚集索引，而MyISAM是非聚集索引

InnoDB引擎的4大特性：插入缓冲（insert buffer),二次写(double write),自适应哈希索引(ahi),预读(read ahead)

2.MVCC?

就是同一份数据临时保留多版本的一种方式，进而实现多版本并发控制。引入多版本之后，只有写写之间相互阻塞，其他三种操作都可以并行，这样大幅度提高了InnoDB的并发度

MVCC 提供了时点（point in time）一致性视图。MVCC 并发控制下的读事务一般使用**时间戳或者事务ID**去标记当前读的数据库的状态（版本），读取这个版本的数据。读、写事务相互隔离，不需要加锁。**读写并存的时候，写操作会根据目前数据库的状态，创建一个新版本，并发的读则依旧访问旧版本的数据。**

3.索引?

索引类型：普通索引、唯一索引（索引的唯一性约束）、主键索引、全文索引

MySQL索引使用的数据结构主要有**BTree索引** 和 **哈希索引** 。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

MySQL的BTree索引使用的是B树中的B+Tree，但对于主要的两种存储引擎的实现方式是不同的。

- **MyISAM:** B+Tree叶节点的data域存放的是数据记录的地址。在索引检索的时候，首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其 data 域的值，然后以 data 域的值为地址读取相应的数据记录。这被称为“**非聚簇索引**”。
- **InnoDB:** 其数据文件本身就是索引文件。相比MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按B+Tree组织的一个索引结构，树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。这被称为“**聚簇索引**（或聚集索引）”。而其余的索引都作为辅助索引，辅助索引的data域存储相应记录主键的值而不是地址，这也是和MyISAM不同的地方。**在根据主索引搜索时，直接找到key所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。 因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。**

1.怎样验证索引是否满足需求?

使用 explain 查看 SQL 是如何执行查询语句的，从而分析你的索引是否满足需求。

explain 语法：`explain select * from table where type=1`。

mysql5.6之后非select语句也可以被解释，使用该命令我们可以查看到表的读取顺序，数据读取操作的类型，那些索引可以被使用，哪些索引实际被使用了。

返回结果列：id、select_type、table、type、possible_keys、key、key_len、ref、rows、Extra

2.索引为什么用B+树不用B树或红黑树?

B+树是平衡树。。。

不用B树：B树每个节点都会存放Data数据，而B+树只有叶子节点存放Data数据。B+树索引占存较小，可以更好存储在磁盘上，能提高查询效率，减少磁盘I/O次数（因为层数少）。并且所有叶子节点采用指针串起来，这样遍历叶子节点就可以获得全部数据（因为是进行区间访问）。

不用红黑树：红黑树是二叉树，深度大（磁盘IO效率低），查找次数多。

3.聚集索引和非聚集索引?

根本区别是**表记录的排列顺序和与索引的排列顺序是否一致**（索引的逻辑顺序和磁盘的物理顺序是否一致）。聚集索引的叶子和实际的数据页重叠，一个表中只能有一个聚集索引；非聚集索引额叶子节点存储主键值。聚集索引强调物理分类，非聚集索引强调逻辑分类。

1、聚集索引一个表只能有一个，非聚集索引一个表可以有多个。
2、聚集索引存储上物理连续，非聚集索引存储上逻辑连续

聚集索引创建：

如果表设置了主键，则主键就是聚簇索引 。
如果表没有主键，则会默认第一个NOT NULL，且唯一（UNIQUE）的列作为聚簇索引 。
以上都没有，则会默认创建一个隐藏的row_id作为聚簇索引

4.回表查询和覆盖索引?

回表查询：先通过普通索引的值定位聚簇索引值，再通过聚簇索引的值定位行记录数据，需要扫描两次索引B+树，它的性能较扫一遍索引树更低。

覆盖索引：只需要在一棵索引树上就能获取SQL所需的所有列数据，无需回表，速度更快。将被查询的字段，建立到联合索引里去。

例如：select id,age from user where age = 10;

5.如何创建索引?

```
ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。
CREATE INDEX可对表增加普通索引或UNIQUE索引。
```

索引创建原则：单列索引、联合索引、最左前缀原则、选择合适的字段

适合创建索引的字段：不为NULL、被频繁查询、作为条件查询、被经常频繁用于连接的

6.使用索引如何避免全表扫描?

1）应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描。
2）应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。
3）应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描
4）in 和 not in 也要慎用，否则会导致全表扫描
5）%like模糊查询也会导致全表扫描。
6）在 where 子句中使用参数，也会导致全表扫描
7）在 where 子句中对字段进行表达式操作，或者函数操作
8）用 exists 代替 in 是一个好的选择

。。。还有一堆：https://blog.csdn.net/lchq1995/article/details/83308290

https://juejin.im/post/5b68e3636fb9a04fd343ba99

7.为什么索引能提高查询速度

MYSQL的底层数据存储结构是页，各个数据页组成一个双向链表，每个数据页的记录又可以组成一个单向链表。

使用索引后，维护一个索引B+树，可以将无序变为有序，二分查找

4.事务?

四大特性:

1. **原子性（Atomicity）：** 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. **一致性（Consistency）：** 执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；
3. **隔离性（Isolation）：** 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
4. **持久性（Durability）：** 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

并发事务带来的问题:

- **脏读（Dirty read）:** 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- **丢失修改（Lost to modify）:** 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
- **不可重复读（Unrepeatableread）:** 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。

- **幻读（Phantom read）：**幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

事务隔离级别：

- **READ-UNCOMMITTED(读取未提交)：** 最低的隔离级别，允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读。**
- **READ-COMMITTED(读取已提交)：** 允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生。**
- **REPEATABLE-READ(可重复读)：** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**
- **SERIALIZABLE(可串行化)：** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读。**

不可重复读和幻读区别：

不可重复读的重点是修改比如多次读取一条记录发现其中某些列的值被修改，幻读的重点在于新增或者删除比如多次读取一条记录发现记录增多或减少了。

5.MySQL复制原理和流程?

基本原理流程，3个线程以及之间的关联；

主：开启binlog线程——记录下所有改变了数据库数据的语句，放进master上的binlog中；

从：开启一个io线程——在使用start slave 之后，负责从master上拉取 binlog内容，放进 自己的relay log中；

从：sql执行线程——执行relay log中的语句，从而和主库数据保持一致；

6.锁?

MySQL有三种锁的级别：**页级、表级、行级。**

- **表级锁**：开销小，加锁快；**不会出现死锁**；锁定粒度大，发生锁冲突的概率最高,并发度最低。
- **行级锁**：开销大，加锁慢；**会出现死锁**；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。
- **页面锁**：开销和加锁时间界于表锁和行锁之间；**会出现死锁**；锁定粒度界于表锁和行锁之间，并发度一般

悲观锁：先获取锁，再进行业务操作

乐观锁（共享锁）：先进行业务操作，不到万不得已不去加锁

排它锁：`用法 : select … for update;` 前提： 没有线程对该结果集中的任何行数据使用排他锁或共享锁，否则申请会阻塞

1）InnoDB存储引擎锁的算法?

- Record lock：单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，不包括记录本身
- Next-key lock：record+gap 锁定一个范围，包含记录本身

7.数据库优化？（大表优化？）

最左原则、限定数据的范围、读写分离、分库、分表垂直水平分区、合适创建索引、使用合适字段、SQL语句优化、系统调优参数、升级硬件

https://segmentfault.com/a/1190000006158186

1）分库分表的全局id?

UUID，数据库自增id（设置不同增长步长），利用redis生成id，雪花算法，美团的Leaf的Id生成器

8.数据库崩溃时事务的恢复机制?

undo log和redo log都是InnoDB存储引擎的日志，redo log用来保证事务的持久性和一致性，undo log日志用来保证事务的原子性和MVCC。

1）Undo Log回滚

Undo Log是为了实现事务的原子性，在MySQL数据库InnoDB存储引擎中，还用了Undo Log来实现多版本并发控制(简称：MVCC)。

- **事务的原子性**(Atomicity)事务中的所有操作，要么全部完成，要么不做任何操作，不能只做部分操作。如果在执行的过程中发生了错误，要回滚(Rollback)到事务开始前的状态，就像这个事务从来没有执行过。
- **原理**Undo Log的原理很简单，**为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为UndoLog）**。然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，**系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。**

之所以能同时保证原子性和持久化，是因为以下特点：

- 更新数据前记录Undo log。
- 为了保证持久性，必须将数据在事务提交前写到磁盘。只要事务成功提交，数据必然已经持久化。
- **Undo log必须先于数据持久化到磁盘。**如果在G,H之间系统崩溃，**undo log是完整的， 可以用来回滚事务。**
- 如果在A-F之间系统崩溃,因为数据没有持久化到磁盘。所以磁盘上的数据还是保持在事务开始前的状态。

缺陷：**每个事务提交前将数据和Undo Log写入磁盘，这样会导致大量的磁盘IO，因此性能很低。**

如果能够将数据缓存一段时间，就能减少IO提高性能。但是这样就会丧失事务的持久性。因此引入了另外一种机制来实现持久化，即Redo Log。

2）Redo Log前滚

- **原理和Undo Log相反，Redo Log记录的是新数据的备份。在事务提交前，只要将Redo Log持久化即可，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是Redo Log已经持久化。系统可以根据Redo Log的内容，将所有数据恢复到最新的状态。**

9.三范式?

第一范式(确保每列保持原子性)

第二范式(确保表中的每列都和主键相关)

第三范式(确保每列都和主键列直接相关,而不是间接相关)

10.主从复制？读写分离？

1）主从复制的几种方式？

同步复制

- 所谓的同步复制，意思是master的变化，必须等待slave-1,slave-2,...,slave-n完成后才能返回。 这样，显然不可取，也不是MySQL复制的默认设置。比如，在WEB前端页面上，用户增加了条记录，需要等待很长时间。

异步复制

- 如同AJAX请求一样。master只需要完成自己的数据库操作即可。至于slaves是否收到二进制日志，是否完成操作，不用关心,MySQL的默认设置。

半同步复制

- master只保证slaves中的一个操作成功，就返回，其他slave不管。

2）数据不一致如何解决？

半同步复制、利用中间件、利用缓存

https://blog.csdn.net/weixin_43885417/article/details/101676610

11.一条SQL语句执行过程？

简单来说 MySQL 主要分为 Server 层和存储引擎层：

·Server 层：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binglog 日志模块。

·存储引擎： 主要负责数据的存储和读取，采用可以替换的插件式架构，支持 InnoDB、MyISAM、Memory 等多个存储引擎，其中 InnoDB 引擎有自有的日志模块 redolog 模块。**现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始就被当做默认存储引擎了。**

1）日志

- **redo log（重做日志）**：记录事务执行后的状态，确保事务的持久性，用于记录事务，保证mysql重启数据恢复，是物理格式的日志，达到事务的一致性。
- **bin log（二进制日志）**：记录所有的数据更改的语句，用于数据恢复，主从复制，增量备份。bin log日志有 Row、Statement、Mixed 三种格式
- **undo log（回滚日志）**：保证事务的原子性和MVCC，是逻辑格式的日志，将数据从逻辑上恢复到事务之前的状态
- **error log（错误日志）**：记录mysqld的启动和停止，以及服务器在运行中发生的错误相关信息
- **general query log（查询日志）**：默认关闭，记录查询相关的操作
- **slow log（慢查询日志）**：记录执行时间过长或没有使用索引的查询语句
- **relay log（中继日志）**：主从复制使用的日志

两阶段提交：

如果采用 redo log 两阶段提交的方式就不一样了，写完 bing log 后，然后再提交 redo log 就会防止出现上述的问题，从而保证了数据的一致性

12.高性能优化建议

- **数据库命名规范**

  临时库表*必须以* tmp*为前缀并以日期为后缀，备份表必须以 bak*为前缀并以日期 (时间戳) 为后缀

- **基本设计规范**

  InnoDB、UTF-8、注释、单表数据量控制在500w、谨慎使用分区表、冷热分离、禁止在表中建立预留字段、禁止在数据库中存储图片文件等大的二进制数据

- **数据库字段设置规范**

  1、避免使用Enum
  2、避免使用Text，Blob数据类型

- **索引设计规范**（将随机IO转变为顺序IO）

  1、建议单张表索引数量不超过5
  2、每个InnoDB表都必须有主键
  3、索引出现在where包含在 ORDER BY、GROUP BY、DISTINCT 中的字段，如果多个建议联合索引
  4、使用最频繁的列、字段长度小的列、区分度最高的列放在最左侧
  5、避免建立冗余索引和重复索引，重复索引示例：primary key(id)、index(id)、unique index(id)；冗余索引示例：index(a,b,c)、index(a,b)、index(a)
  6、对于频繁的查询优先考虑覆盖索引（查询列要被索引列覆盖）

- **SQL开发规范**

  1、建议使用预编译语句进行数据库操作
  2、避免数据类型的隐式转换
  3、充分利用表上已经存在的索引
  4、数据库设计应该对以后的扩展进行考虑
  5、程序连接不同的数据库使用不同的账号，禁止跨库查询
  6、禁止使用不包含字段列表的insert语句
  7、避免使用子查询，可以把子查询优化到join操作
  8、避免使用join关联太多的表，join多次关联同一个表会有缓存
  9、减少同数据库的交互次数
  10、应使用in代替or
  11、禁止使用order by rand（）随机排序
  12、where语句中禁止对列进行函数转换和计算
  13、在明显不会有重复值的时候使用union all而不是union
  14、拆分复杂的大SQL为多个小SQL

- **数据库操作行为规范**

  1、超过100w的批量写（update，delete，insert）要分批多次，会造成主从延迟、binlow为row格式会产生大量日志文件、避免产生大事务操作
  2、对于大表使用pt-online-schema-change 修改表结构，因为会严重的锁表
  3、对于程序连接的数据库账号遵循最小权限原则

2、面试题

1.权限表

MySQL服务器通过权限表来控制用户对数据库的访问，权限表存放在mysql数据库里，由mysql_install_db脚本初始化。这些权限表分别是user，db，table_priv，columns_priv和host。

2.存储过程

存储过程是一个预编译的SQL语句，因为预编译过，所以执行效率高。只需要创建一次，就可以在程序中调用多次。

3.触发器

触发器是用户定义在关系表上的一类由事件驱动的特殊的存储过程。触发器是指一段代码，当触发某个事件时，自动执行这些代码。

使用场景：

1、通过数据库的相关表级联更改
2、实时监控表中的某个字段更改做出相应的处理

MySQL6种触发器：Before Insert、After Insert、Before Update、After Update、Before Delete、After Delete

4.UNION和UNION ALL区别?

- 如果使用UNION ALL，不会合并重复的记录行
- 效率 UNION 高于 UNION ALL

5.超大分页

使用索引覆盖，直接查询id主键索引

6.MySQL的cpu100%

1、top
2、show processlist，看看里面执行的session情况

7.自增id用完怎么办?

改用字段类型bigint，但是alter过程中不允许写，需要借助第三方工具修改。

但是数据量达到很大的情况下都做分库分表了，因此根本达不到int最大值。

8.数据清洗

选择子集，重命名列名，删除重复值，缺失值处理，一致化处理，数据排序，异常值处理。

9.游标

在 MySQL 中，存储过程或函数中的查询有时会返回多条记录，而使用简单的 SELECT 语句，没有办法得到第一行、下一行或前十行的数据，这时可以使用游标来逐条读取查询结果集中的记录。

一般通过游标定位到结果集的某一行进行数据修改。

个人理解游标就是一个标识，用来标识数据取到了什么地方，如果你了解编程语言，可以把他理解成数组中的下标。

使用步骤：声明游标、打开游标、使用游标、关闭游标

1）视图

视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是有一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比多表查询

10.MySQL高可用方案

1、主从复制+读写分离
2、双主+keepalived/heartbeat
3、多节点主从+MHA/MMM
4、多节点主从+etcd/zk
5、基于Galera协议的多主

11.分库分表的问题

1、跨库join问题：全局表、字段冗余数据、数据同步、系统层封装拼接
2、分表后主键id问题：避免使用自增id，使用分布式全局唯一id；或者每次插入完数据返回自增id，但是并发会有问题；或者设置固定sequence步长
3、跨分片排序分页，排序字段最好就是分片字段；函数处理需要对分片分而治之
4、跨分片join问题：全局表、ER分片、Spark内存计算
5、垮库事务，采用分布式事务，尽量避免

12.如何设置可以动态扩容缩容的分库分表方案?

> 也就是说一开始没有分库分表的系统，后期如何设计才能让系统动态切换到分库分表?
>
> 1、停机迁移
> 2、双写迁移方案，跑起来读老库的数据写新库，不要让老系统的数据覆盖新数据，判断最后修改时间。

一开始上来就是 32 个库，每个库 32 个表，1024 张表
我可以告诉各位同学说，这个分法，第一，基本上国内的互联网肯定都是够用了，第二，无论是并发支撑还是数据量支撑都没问题, 每个库正常承载的写入并发量是 1000，那么32 个库就可以承载 32 * 1000 = 32000 的写并发，如果每个库承载 1500 的写并发，32 * 1500= 48000 的写并发，接近 5 万/s 的写入并发，前面再加一个 MQ，削峰，每秒写入 MQ 8 万条数据，每秒消费 5 万条数据。

1）大数据量分页?

采用雪花算法生成id唯一主键，分在单个表上查询，使用子查询优化（这种方式适用于id递增情况），使用id限定优化，使用临时表优化（利用临时表来记录分页的id），如果在分库分表的时候不能使用唯一id，就需要使用高并发分布式唯一id生成器，主要思路是先使用范围查询定义id（索引），然后再使用索引来定位数据。

13.主从同步延时问题?

这种情况可能是写并发太高了，对于写了就要立马可以查到的场景，需要采用强制读主库的方式。如果主从延迟过于严重，可以采用分库，将一个主库拆分为4个主库，这样每个主库的写并发也就500/s，此时主从延迟就可以忽略不计。

13.GTID模式

全局事务ID，其保证为每一个在主上提交的事务在复制集群中可以生成一个唯一的ID，保证事务不丢失和事务的安全性，会保证在一组复制中全局唯一，可以极快速的实现故障转移，快速failover，找到新主节点。

好处：

- 一个事务对应一个唯一ID，一个GTID在一个服务器上只会执行一次;
- GTID是用来代替传统复制的方法，GTID复制与普通复制模式的最大不同就是不需要指定二进制文件名和位置;
- 减少手工干预和降低服务故障时间，当主机挂了之后通过软件从众多的备机中提升一台备机为主机;

14.分区和分表

分表：就是将一张表的数据拆分到多个实体表中。

分区：和分表类似，但是分区是将原始表的数据划分到多个位置存放，表还是一张表。有垂直分区、水平分区。几个分区就有几个.idb文件，

- 分区类型

  RANGE分区、LIST分区、COLUMNS分区、HASH分区、KEY分区、混合分区

15.排名函数

在MYSQL的最新版本MYSQL8已经支持了排名函数`RANK`（并列跳跃），`DENSE_RANK`（并列连续）和`ROW_NUMBER`（连续排名）。

之前的版本只能用变量和函数实现。

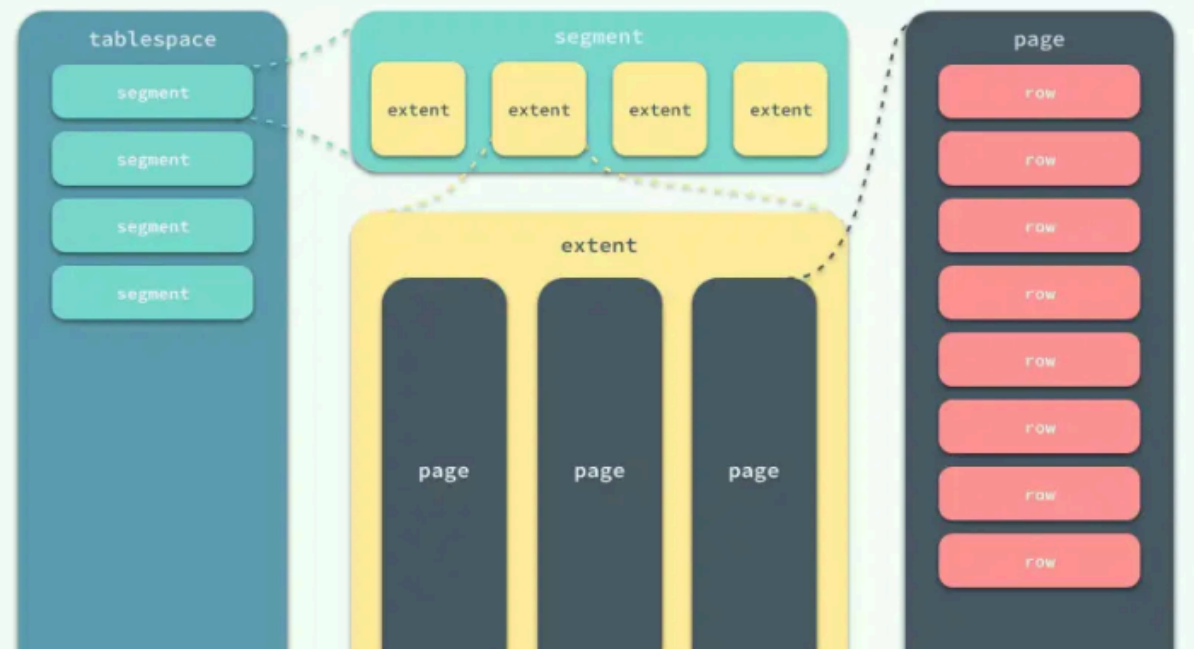
```
RANK() OVER(PARTITION BY s.student_id ORDER BY s.score DESC)
```

16.Canal原理

1. canal模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议
2. mysql master收到dump请求，开始推送binary log给slave(也就是canal)
3. canal解析binary log对象(原始为byte流)

17.表 段 区 页概念

InnoDB逻辑存储结构，所有数据都是被逻辑地存储在表空间当中，表空间又由段、区、页、行组成。



- **段**segment

  常见的段有数据段、索引段、回滚段，在InnoDB存储引擎中，对段的管理都是由InnoDB引擎自身完成的，一个段包含256个区

- **区**extent

  区是由连续的页组成的空间，无论页的大小怎么变，区的大小默认总是1MB，为了保证区中页的连续性，InnoDB存储引擎一次从磁盘申请4-5个区，InnoDB页的大小默认是16kb，一个区对应64个页。 对于表空间而言，他的第一组数据区的第一个数据区的前3个数据页，都是固定的，里面存放了一些描述性的数据。比如FSP_HDR这个数据页，他里面就存放了表空间和这一组数据区的一些属性。

  每个段开始，先用32页大小的碎片页来存放数据，在使用完这些页之后才是64个连续页的申请，这样做的目的，对于一些小表或者undo

- **页**page

  也可以叫块， 页是InnoDB磁盘管理的最小单位。默认大小为16KB，可以通过参数innodb_page_size来设置。

  常见的页类型有：数据页、undo页、系统页、事务数据页、插入缓冲位图页、插入缓冲空闲列表页、未压缩的二进制大对象页、压缩的二进制大对象页

3.MyCat

- **切分后的库从哪里来?**

  在开发中是基于原有库创建出来，并且原有库和切分后的库是数据表的设计是保持一致的。

- **核心配置文件有哪些?**

  schem.xml 配置：逻辑库，逻辑表，数据节点
  rule.xml 配置：分片规则
  server.xml 配置：连接mycat的用户信息

- **mycat可以分成100个库吗?**

  一般就是分成几个库，如果数据量更大就采用oracle，如果再大就是Hadoop或者hbase和云存储数据库了，就不需要mycat作为工具手段，但是成本太大。

- **库表拆分时，拆分规则如何取舍?**

看有没有热点数据。

- **旧系统的数据如何导入到mycat中?**

  旧数据迁移目前可以手工导入，在 mycat 中提取配置好分配规则及后端分片数据库，然后通过 dump或 loaddata 方式导入，后续 Mycat 就做旧数据自动数据迁移工具

十五、Redis

简介

简单来说 redis 就是一个数据库（用C语言开发的），不过与传统数据库不同的是 redis 的数据是存在**内存**中的，所以读写速度非常快，因此 redis 被广泛应用于缓存方向。另外，redis 也经常用来做**分布式锁**。redis 提供了多种数据类型来支持不同的业务场景。除此之外，redis 支持**事务 、持久化、LUA 脚本、LRU 驱动事件、多种集群方案。**（功能）

使用场景:

记录帖子点赞数、点击数、评论数；缓存近期热帖；缓存文章详情信息；记录用户会话信息。

功能:

数据缓存、分布式锁、支持数据持久化、支持事务、支持消息队列。

1.为什么要用缓存（Redis)

高性能：第一次访问数据库中某些数据会很慢，将数据放在缓存中就以后从内存中直接读取数据，速度很快。

高并发：直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。

2.为什么要用redis而不是map/guava做缓存?

缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，**每个实例都需要各自保存一份缓存，缓存不具有一致性。**

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，**各实例共用一份缓存数据，缓存具有一致性。**缺点是需要保持 redis 或 memcached 服务的高可用，整个程序架构上较为复杂。

3.redis和memcached区别?

1. **redis 支持更丰富的数据类型（支持更复杂的应用场景）**：Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。memcache 支持简单的数据类型，String。
2. **Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用，而 Memecache 把数据全部存在内存之中。**
3. **集群模式**：memcached 没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是 redis 目前是原生支持 cluster 模式的.
4. **Memcached 是多线程，非阻塞 IO 复用的网络模型；Redis 使用单线程的多路 IO 复用模型，并且 Redis是一个事件驱动模型**

对比参数	Redis	Memcached
类型	1、支持内存 2、非关系型数据库	1、支持内存 2、key-value键值对形式 3、缓存系统
数据存储类型	1、String 2、List 3、Set 4、Hash 5、Sort Set 【俗称ZSet】	1、文本型 2、二进制类型【新版增加】
查询【操作】类型	1、批量操作 2、事务支持【虽然是假的事务】 3、每个类型不同的CRUD	1、CRUD 2、少量的其他命令
附加功能	1、发布/订阅模式 2、主从分区 3、序列化支持 4、脚本支持【Lua脚本】	1、多线程服务支持
网络IO模型	1、单进程模式	2、多线程、非阻塞IO模式
事件库	自封装简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1、RDB 2、AOF	不支持

4.常用数据结构?

1.String

> **常用命令:** set,get,decr,incr,mget 等。常规计数：微博数，粉丝数等。

2.Hash

> **常用命令：** hget,hset,hgetall 等。我们可以用 hash 数据结构来存储用户信息，商品信息等等。

3.List

> **常用命令:** lpush,rpush,lpop,rpop,lrange 等比如微博的关注列表，粉丝列表，消息列表等功能都可以用 Redis 的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

另外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，**可以基于 list 实现分页查询**，这个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

4.Set

> **常用命令：** sadd,spop,smembers,sunion 等。Redis 中的 set 类型是一种无序集合，集合中的元素没有先后顺序。非常方便求交集并集等;

比如：在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis 可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下：

```
sinterstore key1 key2 key3      将交集存在key1内
```

5.Sorted Set

> **常用命令：** zadd,zrange,zrem,zcard 等。和 set 相比，sorted set 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列。

在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用 Redis 中的 Sorted Set 结构进行存储。

三种特殊的数据类型：

1、Bitmap：位图，数组中的每个单元只能存0或1
2、Hyperloglog： HyperLogLog 是一种用于统计基数的数据集合类型， 优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的
3、 Geospatial： 主要用于存储地理位置信息，并对存储的信息进行操作，适用场景如朋友的定位、附近的人、打车距离计算等 。

5.内存淘汰机制?

> (MySQL 里有 2000w 数据，Redis 中只存 20w 的数据，如何保证 Redis 中的数据都是热点数据?)

redis 提供 6 种数据淘汰策略：

1. **volatile-lru**：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. **volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. **volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. **allkeys-lru**：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 key（这个是最常用的）
5. **allkeys-random**：从数据集（server.db[i].dict）中任意选择数据淘汰
6. **no-eviction**：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！

4.0 版本后增加以下两种：

1. **volatile-lfu**：从已设置过期时间的数据集(server.db[i].expires)中挑选最不经常使用的数据淘汰
2. **allkeys-lfu**：当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的 key

6.持久化机制?

> 怎么保证 redis 挂掉之后再重启数据可以进行恢复?

Redis不同于Memcached的很重一点就是，**Redis支持持久化**，而且支持两种不同的持久化操作。Redis的一种持久化方式叫**快照（snapshotting，RDB）**,另一种方式是**只追加文件（append-only file,AOF）**

快照（snapshotting）持久化（RDB）

Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis 创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis 主从结构，主要用来提高 Redis 性能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是 Redis 默认采用的持久化方式，在 redis.conf 配置文件中默认有此下配置：

```
save 900 1              #在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发BGSAVE命令创建快照。
save 300 10             #在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发BGSAVE命令创建快照。
save 60 10000           #在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发BGSAVE命令创建快照。
```

AOF（append-only file）持久化

与快照持久化相比，AOF 持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下 Redis 没有开启 AOF（append only file）方式的持久化，可以通过 appendonly 参数开启：

```
appendonly yes
```

开启 AOF 持久化后每执行一条会更改 Redis 中的数据的命令，Redis 就会将该命令写入硬盘中的 AOF 文件。AOF 文件的保存位置和 RDB 文件的位置相同，都是通过 dir 参数设置的，默认的文件名是 `appendonly.aof`。

在 Redis 的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

```
appendfsync always    #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度
appendfsync everysec  #每秒钟同步一次，显示地将多个写命令同步到硬盘
appendfsync no        #让操作系统决定何时进行同步
```

为了兼顾数据和写入性能，用户可以考虑 `appendfsync everysec` 选项 ，让 Redis 每秒同步一次 AOF 文件，Redis 性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

7.事务?

Redis 通过 `MULTI、EXEC、WATCH、DISCARD` 等命令来实现事务(transaction)功能。**事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制**，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求。

在传统的关系式数据库中，常常用 ACID 性质来检验事务功能的可靠性和安全性。在 Redis 中，事务总是具有原子性（Atomicity）、一致性（Consistency）和隔离性（Isolation），并且当 Redis 运行在某种特定的持久化模式下时，事务也具有持久性（Durability）。

补充内容：

> redis 同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚。（来自issue:关于 Redis 事务不是原子性问题）；Redis不支持Roll back
>
> 使用 MULTI命令后可以输入多个命令。Redis不会立即执行这些命令，而是将它们放到队列，当调用了EXEC命令将执行所有命令。开始事务-命令入队-执行事务

1	DISCARD 取消事务，放弃执行事务块内的所有命令。
2	EXEC 执行所有事务块内的命令。
3	MULTI 标记一个事务块的开始。
4	UNWATCH 取消 WATCH 命令对所有 key 的监视。
5	WATCH key [key ...] 监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

8.怎么保证缓存和数据库数据一致?

1、查询CAP+延时双删

2、redis订阅binlog日志

> 一般情况下我们都是这样使用缓存的：先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。这种方式很明显会存在缓存和数据库的数据不一致的情况。

- 合理设置缓存的过期时间。
- 新增、更改、删除数据库操作时同步更新 Redis，可以使用事物机制来保证数据的一致性。

一般来说，可以用**串行化**：但是就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

解决：Cache Aside Pattern（CAP）：最经典的缓存+数据库读写的模式。

- 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。
- 更新的时候，**先更新数据库，然后再删除缓存。**

1）缓存和数据库双写一致性问题

- **先删除缓存，后更新数据库**

  方案一：延时双删
  方案二：更新和读取操作进行异步串行化

- **先更新数据库，后删除缓存**

  方案一：如果删除缓存失败了，可以使用消息队列消息补偿再次尝试删除

- **查询走redis，更新删除时走mysql，同步采用canal**

9.为什么Redis是单线程的?

因为 cpu 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存或者网络带宽。既然单线程容易实现，而且 cpu 又不会成为瓶颈，那就顺理成章地采用单线程的方案了。

关于 Redis 的性能，官方网站也有，普通笔记本轻松处理每秒几十万的请求。

而且单线程并不代表就慢 nginx 和 nodejs 也都是高性能单线程的代表。

速度快原因：

(一)纯内存操作，C语言编写

(二)单线程操作，避免了频繁的上下文切换

(三)采用了非阻塞I/O多路复用机制

（四）内存底层特殊的优化数据结构

1、Redis6.0为什么引入多线程?

Redis6.0 引入多线程主要是为了提高网络 IO 读写性能，因为这个算是 Redis 中的一个性能瓶颈（Redis 的瓶颈主要受限于内存和网络）。 用来**处理网络数据的读写和协议的解析**，而**执行命令依旧是单线程。**

Redis6.0 的多线程默认是禁用的，只使用主线程。如需开启需要修改 redis 配置文件 `redis.conf`

```
io-threads-do-reads yes
```

开启多线程后，还需要设置线程数，否则是不生效的。同样需要修改 redis 配置文件 `redis.conf`：

```
io-threads 4 #官网建议4核的机器建议设置为2或3个线程，8核的建议设置为6个线程
```

10.缓存击穿，缓存穿透，缓存雪崩?

缓存穿透：查询缓存和数据库中都不存在的数据。**解决**：参数校验、缓存无效key、布隆过滤器。

缓存雪崩：缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。**解决：**

事前：尽量保证整个 redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。**Redis集群，限流**

事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 崩掉

事后：利用 redis 持久化机制保存的数据尽快恢复缓存，设置不同的失效时间。。。

缓存击穿：在高并发的情况下，大量的请求同时查询同一个key时，此时这个key正好失效了，就会导致同一时间，这些请求都会去查询数据库，这样的现象我们称为缓存击穿。**解决**：采用分布式锁，只有拿到锁的第一个线程去请求数据库，然后插入缓存，当然每次拿到锁的时候都要去查询一下缓存有没有。

11.热点数据集中失效如何解决?

1. 设置不同的失效时间
2. 采用缓存击穿的解决办法，加锁
3. 永不失效，就是采用定时任务对快要失效的缓存进行更新缓存和失效时间

12.怎么实现分布式锁?

Redis 分布式锁其实就是在系统里面占一个“坑”，其他程序也要占“坑”的时候，占用成功了就可以继续执行，失败了就只能放弃或稍后重试。

占坑一般使用 setnx(set if not exists)指令，只允许被一个程序占有，使用完调用 del 释放锁。

13.分布式锁的缺陷?

Redis 分布式锁不能解决**超时**的问题，分布式锁有一个超时时间，程序的执行如果超出了锁的超时时间就会出现问题。

1. 基于Redis的分布式锁在极端情况下是无法保证Safety
2. 即使Redis的作者antirez给出的更复杂的Redlock实现也无法解决这个情况

14.Redis如何做内存优化?

尽量使用 Redis 的散列表，把相关的信息放到散列表里面存储，而不是把每个字段单独存储，这样可以有效的减少内存使用。比如将 Web 系统的用户对象，应该放到散列表里面再整体存储到 Redis，而不是把用户的姓名、年龄、密码、邮箱等字段分别设置 key 进行存储。

15.常见性能问题?

· 主服务器写内存快照，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以**主服务器最好不要写内存快照。**

· Redis 主从复制的性能问题，为了主从复制的速度和连接的稳定性，**主从库最好在同一个局域网内。**

16.面试题

1、集群高可用

- **主从同步**

  概念：有且仅有一个主节点master，只要网络正常，master就会一直将自己的数据同步给slaves

  优点：读写分离，主写从读

  缺点：1、如果故障需要手动切换节点为主节点，不能自动容错恢复；
  2、主库的写能力受到机器限制，可以分片，很难在线扩容；
  3、Redis数据复制中断可能会产生数据不一致

- **哨兵模式**

  概念：分为Redis主从集群和Sentinel监视器集群，为了解决主从的不足

  优点：实现故障自动转移、发现、配置中心和客户端通知

  缺点：资源部署浪费、原理理解繁琐、实现复杂、不能解决读写分离

- **Redis Cluster**

  概念：是社区版推出的Redis分布式集群解决方案，3主3从，主节点提供读写操作，从节点作为备用节点；采用虚拟槽分区，所有的键根据哈希函数映射到0～16383个整数槽内，每个节点负责维护一部分槽以及槽所印映射的键值数据。

  优点：无中心架构

2、基本命令

```
set key value #设置 key-value 类型的值
get key # 根据 key 获得对应的 value
exists key  # 判断某个 key 是否存在
strlen key # 返回 key 所储存的字符串值的长度。
del key # 删除某个 key 对应的值
mset key1 value1 key2 value2 # 批量设置 key-value 类型的值
mget key1 key2 # 批量获取多个 key 对应的 value
expire key  60 # 数据在 60s 后过期
setex key 60 value # 数据在 60s 后过期 (setex:[set] + [ex]pire)
```

3、过期数据的删除策略?

1. **惰性删除**： 只会在取出key的时候才对数据进行过期检查。这样对CPU最友好，但是可能会造成太多过期 key 没有被删除。
2. **定期删除**： 每隔一段时间抽取一批 key 执行删除过期key操作。并且，Redis 底层会通过限制删除操作执行的时长和频率来减少删除操作对CPU时间的影响。

定期删除对内存更加友好，惰性删除对CPU更加友好。两者各有千秋，所以Redis 采用的是 **定期删除+惰性/懒汉式删除** 。

4、缓存的3种读写策略?

- CAP（旁路缓存模式）

  写：先更新DB、然后删除cache
  读：从cache中取数据，有就返回，没有就从DB返回，然后放到Cache

  缺点：极端情况下也会出现脏数据，可能导致缓冲中数据丢失，增大缓存穿透概率

解决：延时双删、引入消息队列删除重试、基于Canal小日志增量解析。

- Read/Write Through Pattern（读写穿透）

  写：先查cache，cache中不存在，直接更新DB；cache中存在，则先更新cahce，然后cache服务自己更新DB
  读：从cache中取数据，有就返回，没有就从DB返回，然后放到Cache

- Write Behind Pattern（异步缓存）

  只更新cache，不直接更新DB，采用异步批量的方式更改DB

  写性能非常高，非常适合数据经常变化又对数据一致性没有高要求的场景

5、集群

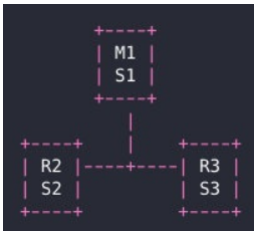
Redis在重启的时候默认使用AOF来去重新构建数据，因为AOF的数据是比RDB更完整的。

RDB的优点：由于生产多个数据文件，所以更适合做冷备份，数据运维设置定时任务，定时同步到远端的服务器。RDB对Redis的性能影响非常小，是因为在同步的时候fork了一个子进程去持久化的，而且他在数据恢复的速度比AOF来的快。
RDB的缺点：因为是快照，所以时间差会有数据的丢失，而AOF最多丢失一秒的数据，AOF数据完整性更好。如果快照文件很大的话，客户端会卡顿几毫秒或几秒。

AOF的优点：数据完整性高，写日志文件已appendly-only方式去写，减少了磁盘寻址的开销，写入性能惊人，适合容灾性数据误删除的恢复。
AOF的缺点：产生的日志文件会很大，AOF开启后支持写的QPS要降低很多，每次异步刷新缓存区的数据区持久化

如何选择：如果发生意外，第一时间用RDB恢复，然后用AOF做数据补全，冷备热备一起上

集群高可用实现方式：哨兵+主从不能保证数据不丢失，但是可以保证集群的高可用。经典的哨兵集群是三主三从如下图：



1）主从同步

当启动一台slave的时候，他会发送一个**psync**命令给master ，如果是这个slave第一次连接到master，他会触发一个全量复制。master就会启动一个线程，生成**RDB**快照，还会把新的写请求都缓存在内存中，**RDB**文件生成后，master会将这个**RDB**发送给slave的，slave拿到之后做的第一件事情就是写进本地的磁盘，然后加载进内存，然后master会把内存里面缓存的那些新命名都发给slave。

> 如果传输过程中断网会进行自动重连的，RDB快照数据生成的时候，缓存区也必须同时开始接受新的请求，从而好同步在同步期间的增量数据

2）Redis Cluster

每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽， 集群的每个节点负责一部分 hash 槽 。

为什么是163874个槽?

在redis节点发送心跳包时需要把所有的槽放到这个心跳包里，以便让节点知道当前集群信息，16384=16k，在发送心跳包时使用`char`进行bitmap压缩后是2k（`2 * 8 (8 bit) * 1024(1k) = 16K`），也就是说使用2k的空间创建了16k的槽数。

虽然使用CRC16算法最多可以分配65535（2^16-1）个槽位，65535=65k，压缩后就是8k（`8 * 8 (8 bit) * 1024(1k) =65K`），也就是说需要需要8k的心跳包，作者认为这样做不太值得；并且一般情况下一个redis集群不会有超过1000个master节点，所以16k的槽位是个比较合适的选择。

> bitmap原理：原来数组一个元素存储一个int 4字节，但是现在一个元素存储一个1byte=8bit，可以表示范围为0-31，表示32个数字

缺点：

1、Client客户端实现较为复杂， 驱动要求实现Smart Client，缓存slots mapping信息并及时更新，提高了开发难度，客户端的不成熟影响业务的稳定性。目前仅JedisCluster相对成熟
2、数据异步复制，不保证数据强一致性
3、不建议使用批量操作

3）Redis Sentinel

哨兵组件的作用：

1、集群监控：监控master和slave是否正常工作
2、消息通知：如果某个Redis实例有故障，那么哨兵发送消息给管理员
3、故障转移：master挂掉了，会自动转移到slave上
4、配置中心

工作流程：

1、sentinel每秒钟一次向master、slave、sentienl发送一个ping命令
2、如果一个实例距离最后一次有效回复ping命令的时间超过 down-after-milliseconds 选项所指定的值，这个实例就会被标记为主观下线
3、如果master被标记为主观下线，则监视的sentinel要再次确认master主观下线
4、当有足够数量的sentinel标记master为主观下线，则master会被标记为客观下线
5、当master客观下线后，sentinel向下线的master的所有slave发送INFO命令的频率会从10s/次改为1s/次
6、反之，客观下线也会变成主观下线，最后被移除主观下线状态
7、sentinel会相互沟通选出一个主节点，其他节点挂载到新节点自动复制新主节点的数据

脑裂问题：master脱离了原来网络，这时又有新的选主产生。

脑裂问题解决方案： 提高网络、硬件等方法避免 ，修改以下配置：

min-slaves-to-write 1 // 要求至少有1个slave
min-slaves-max-lag 10 //数据复制和同步的延迟不能超过10秒

4）Redis自研

借助于Zookeeper，体现在配置中心、故障探测、和failover的环境定制化。

特点：自助可控、实现复杂

6、setnx

原理：如果key存在，则加锁失败返回0；如果key不存在，则加锁成功返回1

缺点：当redis宕机时，key消失，同一把锁可能会被多人占用

使用方式：单纯使用setnx命令加上业务因为不是原子操作，多线程下还是会产生安全问题，所以需要配置Lua脚本保证原子性。

其他使用方式：Redisson+Redlock

7、缓存降级

缓存降级是指缓存失效或缓存服务器挂掉的情况下，不去访问数据库，直接返回默认数据或访问服务的内存数据。降级一般是有损的操作，所以尽量减少降级对于业务的影响程度。

8、Redis线程模型

Redis是基于reactor模式的多路IO复用线程模型

> select：需要自己不断的轮询fd集合，知道fs就绪，只支持1024个文件描述符，每次都要把fd集合从用户态往内核态拷贝一次
> poll：和select类似，只是数据结构不一样
> epoll：epoll只需要吧fd集合从用户态拷贝到内核态一次，采用红黑树事件监听的机制

大体步骤：多个socket、IO多路复用程序、文件事件分派器、事件处理器

1）Redis 6.0多线程的实现机制

1、主线程负责接收建立连接请求，获取Socket放入全局等待读队列
2、主线程处理完读事件后，通过RR将这些连接分配给IO线程
3、主线程阻塞等待IO线程读取Socket完毕
4、主线程通过单线程的方式执行请求命令，主线程阻塞等待IO线程将数据写回给Socket完毕

> 总结：
>
> IO线程要么同时在读Socket，要么同时在写，不会同时读或写
> IO线程只负责读写Socket解析命令，不负责命令处理

9、Redis主从数据不一致

- 迁移后Redis过期时间不一致

  解决：

  1、业务采用expireat命令 timestamp 方式，设置为具体的时间点，这样命令传送到从库就没有影响
  2、在Redis代码中将expire命令转换为expireat命令

  但是影响也不大，主要是主库触发淘汰策略

- 迁移后Redis key数量不一致

  解决：

  在bgsave、 bgrewirteaof 、 load rdb 的时候不忽略过期的key，最终由主库触发删除策略

10、主从复制过程中网络中断停止复制会怎样?

网络断开故障后会自动重连，Redis2.8之后就支持端点续传。

master如果发现有多个slave 结点都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

> master节点会在内存中创建一个`backlog`，master和slave都会保存一个`replica offset`，还有一个`master id`，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制。
>
> 但是如果没有找到对应的offset，那么就会执行一次`resynchronization`全量复制。

11、Sentinel选举master的标准是什么?

1、跟master断开连接的时长。如果一个slave和master断开连接已经超过了down-after-milliseconds的10倍加上master宕机的时长，那么slave就不适合选举
2、slave优先级，priority越低优先级越高
3、复制offset。那个slave复制了越多的数据，优先级就越高
4、如果上面两个条件都相同，那么选择一个run id比较小的slave

12、同步配置的时候其他哨兵根据什么更新自己配置?

执行切换的那个哨兵会从要切换的新master（slave->master）哪里得到一个epoch version号，每次切换的version号都必须是唯一的。如果第一个选举出的哨兵切换失败了，那么其他哨兵会等待failover-timeout时间，然后结题继续执行切换，重新生成一个epoch version号。其他哨兵都是根据版本号的大小来更新自己的master配置

13、Redis Cluster节点间的通信机制?

Redis Cluster节点间采取gossip协议进行通信，所有节点都持有一份元数据，不同的节点如果出现了元数据的变更后就不断地将元数据发送给其他节点进行数据变更。

> 节点互相之间不断通信，保持整个集群所有节点的数据是完整的。 主要交换故障信息、节点的增加和移除、hash slot信息等。

17.Redis底层存储结构

String：使用自定义数据结构sds来操作字符串，是一种简单动态字符串，自定义了内存重分配策略。

List：数据量小的时候使用压缩链表，数据量大的时候使用双向无环链表。如果数据量较小并且存储数据是整数的时候就使用整数集合。

Hash：数据量小的时候使用压缩链表，数据量大的时候使用字典（hash表实现），redis字典解决hash冲突：链表法、渐进式rehash；每个字典有两个Hash表，一个正常使用，一个rehash期间使用。

set：

zset：跳跃表（在链表的基础上加了多级索引提高查询效率）

> quicklist： 是Redis 3.2版本以后针对链表和压缩列表进行改造的一种数据结构，是 zipList 和 linkedList 的混合体，相对于链表它压缩了内存。进一步的提高了效率。

18.Redis回收进程如何工作?

一个客户端运行了新的命令，添加了新的数据。Redi 检查内存使用情况，如 果大于 maxmemory 的限制, 则根据设定好的策略进行回收。

19.Redis Cluster性能优化

1、Master最好不要写内存快照RDB，在Slave上开启AOF备份数据
2、避免在Master压力很大的主库上增加从库
3、主从复制最好启用单向链表式结构，不要使用图状结构

20.负载因子

Redis在执行BGSAVE和BGREWRITEAOF命令时负载因为是5，其他情况默认负载因子为1。

因为Redis负载因子计算方式是键值对数量除以长度，而不是已经使用位置的数量除以长度。

> 根据 BGSAVE 命令或 BGREWRITEAOF 命令是否正在执行， 服务器执行扩展操作所需的负载因子并不相同， 这是因为在执行BGSAVE 命令或BGREWRITEAOF 命令的过程中， Redis 需要创建当前服务器进程的子进程，而大多数操作系统都采用写时复制（copy-on-write）技术来优化子进程的使用效率， 所以在子进程存在期间，服务器会提高执行扩展操作所需的负载因子，从而尽可能地避免在子进程存在期间进行哈希表扩展操作， 这可以避免不必要的内存写入操作，最大限度地节约内存。

写时复制： fork创建出的子进程，与父进程共享内存空间。也就是说，如果子进程不对内存空间进行写入操作的话，内存空间中的数据并不会复制给子进程，这样创建子进程的速度就很快了！(不用复制，直接引用父进程的物理空间)。

21.Redis慢操作

从两个方面排查：网络问题延迟比较严重，Redis自身出现问题

1、统计网络延迟

```
redis-cli --latency -h `host` -p `port`
```

2、开启自身慢查询记录慢日志

```
# 命令执行耗时超过 5 毫秒，记录慢日志
CONFIG SET slowlog-log-slower-than 5000
# 只保留最近 500 条慢日志
CONFIG SET slowlog-max-len 500
```

3、是否使用了复杂操作：范围操作、聚合操作（SORT、SUNION）、统计操作（LLEN、SCARD）。用scan替代keys *

4、使用big key

```
#定位bigkey
bigkeys -i 0.01
```

5、大量key集中过期
6、内存达到上限，触发淘汰策略
7、写AOF为always同步写回。 **一般情况下，aof刷盘机制配置为everysec每秒写回即可**
8、可能内存不够用被换到操作系统的swap分区
9、网卡中断，reis cluster 100w qps轻轻松松

22.Red Lock

Redlock是一种算法，Redlock也就是 Redis Distributed Lock，可用实现多节点Redis的分布式锁。

RedLock官方推荐，Redisson完成了对Redlock算法封装。

> 此种方式具有以下特性：
>
> - 互斥访问：即永远只有一个 client 能拿到锁
> - 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使锁定资源的服务崩溃或者分区，仍然能释放锁。
> - 容错性：只要大部分 Redis 节点存活（一半以上），就可以正常提供服务

1、Red Lock原理

> 1、不再需要部署从库和哨兵实例，只部署主库
> 2、但主库要部署多个，官方推荐至少5个实例

需要有5个完全独立的Redis主服务器，只有建立在时钟正确的前提下才能正常工作

1. 客户端先获取「当前时间戳T1」
2. 客户端依次向这 5 个 Redis 实例发起加锁请求（用前面讲到的 SET 命令），且每个请求会设置超时时间（毫秒级，要远小于锁的有效时间），如果某一个实例加锁失败（包括网络超时、锁被其它人持有等各种异常情况），就立即向下一个 Redis 实例申请加锁
3. 如果客户端从 >=3 个（大多数）以上 Redis 实例加锁成功，则再次获取「当前时间戳T2」，如果 T2 - T1 < 锁的过期时间，此时，认为客户端加锁成功，否则认为加锁失败
4. 加锁成功，去操作共享资源（例如修改 MySQL 某一行，或发起一个 API 请求）
5. 加锁失败，向「全部节点」发起释放锁请求（前面讲到的 Lua 脚本释放锁）

十六、JVM

1、基础

0.概念

JVM是Java Virtual Machine（Java虚拟机）的缩写，由**一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域等**组成。JVM屏蔽了与操作系统平台相关的信息，使得Java程序只需要生成在Java虚拟机上运行的目标代码（字节码），就可在多种平台上不加修改的运行，这也是Java能够“**一次编译，到处运行的**”原因。

运行过程：Java源文件->编译器-->字节码文件-->JVM-->机器码

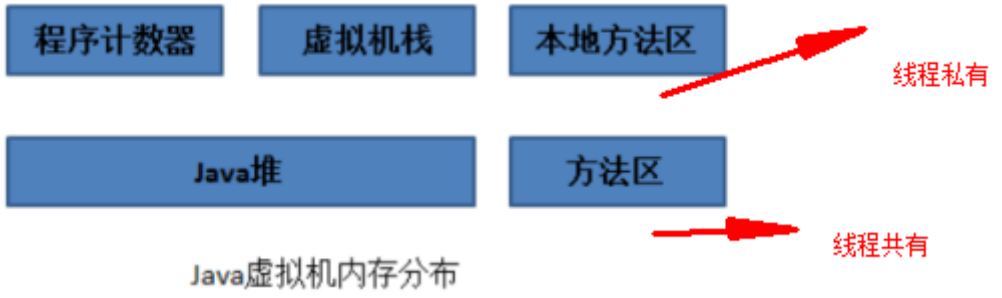
1.JVM主要组成部分？及其作用?

1.**类加载器（Class Loader）**：加载类文件到内存。Class loader只管加载，只要符合文件结构就加载，至于能否运行，它不负责，那是有Exectution Engine 负责的。

2.**执行引擎（Execution Engine）**：也叫解释器，负责解释命令，交由操作系统执行。

3.**本地库接口（Native Interface）**：本地接口的作用是**融合不同的语言为java所用。**

4.**运行时数据区（Runtime Data Area，内存区域）**：



Java虚拟机内存分布

（1）**堆（GC堆）**。堆是java对象的存储区域，在虚拟机启动时创建。任何用new字段分配的java对象实例和数组，都被分配在堆上，java堆可用-Xms和-Xmx进行内存控制，**jdk1.7以后，运行时常量池从方法区移到了堆上。**

Java堆内存结构

新生代：老年代 = 1：2

Eden：s1:s2=8:1:1

（2）**方法区（也称为永生代）**：用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。

误区：方法区不等于永生代

很多人原因把方法区称作“永久代”（Permanent Generation），本质上两者并不等价，只是HotSpot虚拟机垃圾回收器团队把GC分代收集扩展到了方法区，或者说是用来永久代来实现方法区而已，这样能省去专门为方法区编写内存管理的代码，但是在Jdk8也移除了“永久代”，使用Native Memory来实现方法区。

JDK1.8之后常用参数：

```
-XX:MetaspaceSize=N //设置 Metaspace 的初始（和最小大小）
-XX:MaxMetaspaceSize=N //设置 Metaspace 的最大大小
```

运行时常量池：是方法区的一部分。JDK1.7之后将常量池移到了堆中。包含：类的符号引用、文本字符串、final的常量、基本数据类型

（3）**虚拟机栈：**虚拟机栈中执行每个方法的时候，都会创建一个栈桢用于存储局部变量表，操作数栈，动态链接，方法出口等信息。**就是平时我们所说的堆和栈中的栈**，由一个个栈帧组成。**生命周期随线程死亡而死亡。**

虚拟机栈会出现两种错误：

Stack Overflow：内存大小不允许动态扩展，超出栈的大小

OutofMemory：内存大小允许动态扩展，栈内内存耗尽

（4）**本地方法栈：**与虚拟机发挥的作用相似，相比于虚拟机栈为Java方法服务，本地方法栈为虚拟机使用的Native方法服务，执行每个本地方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

（5）**程序计数器。**

1、指示Java虚拟机下一条需要执行的字节码指令。

2、用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

1.1 为什么要将方法区（永生代）替换为元空间?

整个永久代有一个 JVM 本身设置**固定大小上限**，无法进行调整，而元空间使用的是直接内存，**受本机可用内存的限制**，并且永远不会得到 `java.lang.OutOfMemoryError`。你可以使用 `-XX: MaxMetaspaceSize` 标志设置最大元空间大小，默认值为 unlimited，这意味着它只受系统内存的限制。`-XX: MetaspaceSize` 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

PS：还有很多底层的原因。。。

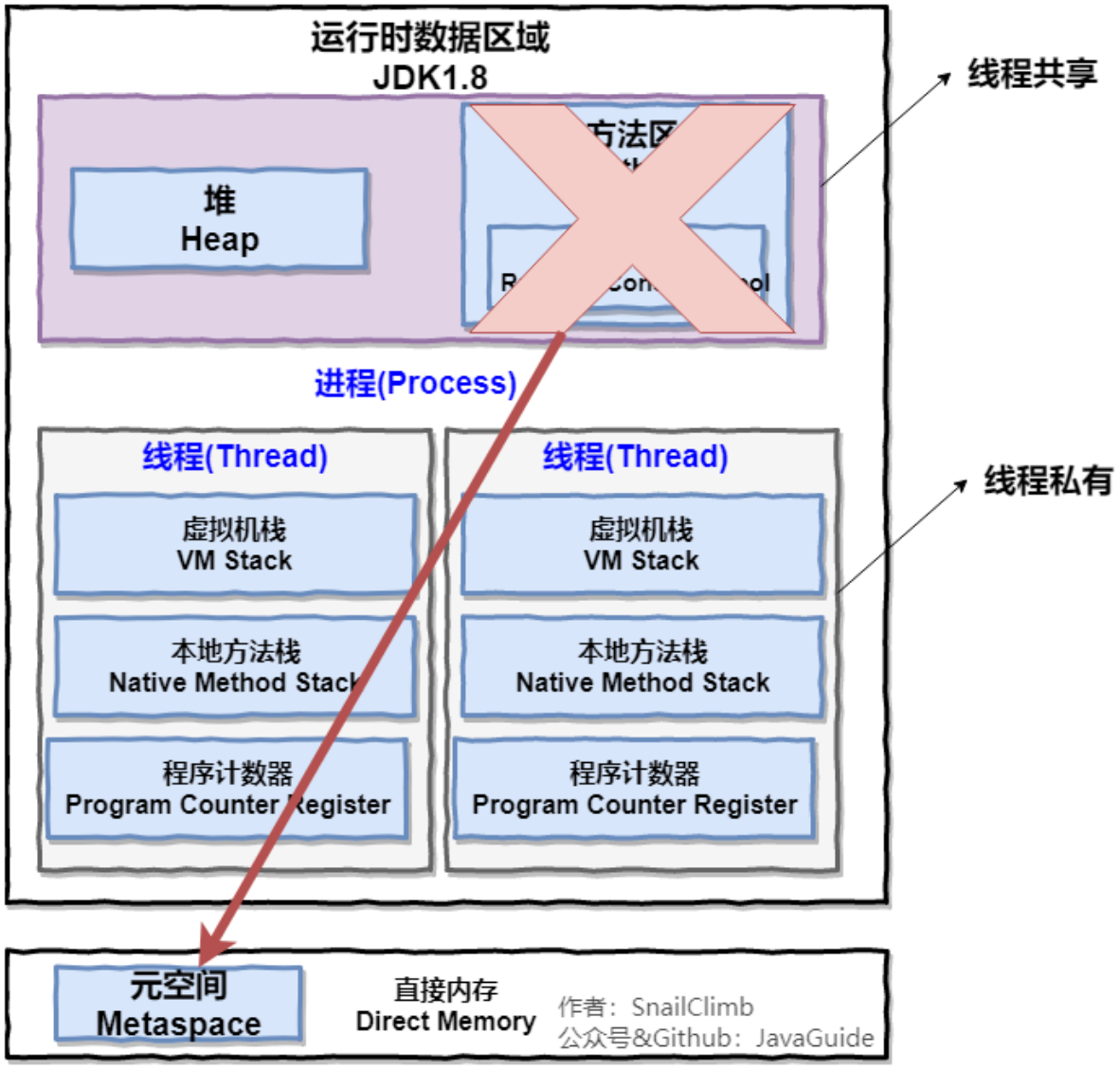
2.JVM运行时数据区?

线程私有：程序计数器、虚拟机栈、本地方法栈

线程共享：堆、方法区、直接内存（元空间）

jdk1.7之前堆内存分为：新生代，老生代，永生代

jdk1.8之后的改变：方法区（永生代）被彻底移除，取而代之的是元空间，使用的是直接内存。如图：



2.1 JVM对象的创建过程?

1、类加载检查：JVM遇到一个new指令时，首先检查这个符号引用代表的类是否已被加载、解析、初始化过，如果没有，就执行相应的类加载过程。

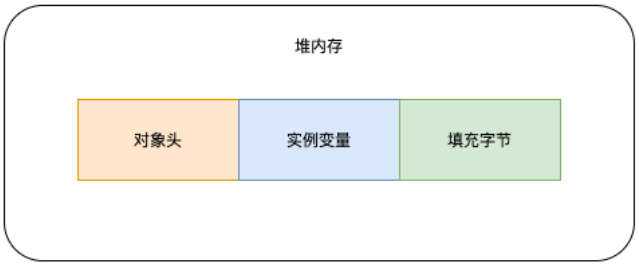
2、分配内存：为新生对象分配内存，分配方式有“指针碰撞”和“空闲列表”两种，取决于Java堆内存是否规整，而是否规整又取决于GC 收集器的算法是"标记-清除"，还是"标记-整理"（规整）

> **内存分配的并发问题：**在创建对象的时候有一个很重要的问题，就是线程安全。虚拟机采用两种方式来保证线程安全：
>
> - **CAS+失败重试：** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。**
> - **TLAB：** 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

3、初始化零值：内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

4、执行init方法：在Java程序的层面上，把对象按照程序员的意愿进行初始化。

2.2 对象的内存布局?



分为 3 块区域：**对象头**、**实例数据**和**对齐填充。**

1、对象头：存储对象自身的运行时数据（标记字段）；类型指针；
2、实例数据：实例变量，真正存储对象的有效信息
3、对齐填充：仅仅起占位作用，由于虚拟机要求对象起始地址必须是8字节的整数倍。**填充数据不是必须存在的，仅仅是为了字节对齐**

2.3 对象的访问定位?

对象的访问方式由虚拟机实现而定，目前主流的访问方式有①**使用句柄**和②**直接指针**两种：

1、句柄： Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息
2、直接指针： Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象的地址

> 使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

3.堆和栈的区别?

· **功能方面**：堆是用来存放对象的，栈是用来执行程序和存放方法和局部变量的。

· **共享性**：堆是线程共享的，栈是线程私有的。

· **空间大小**：堆大小远远大于栈。

4.双亲委派模型?

在介绍双亲委派模型之前先说下类加载器。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立在 JVM 中的唯一性，每一个类加载器，都有一个独立的类名称空间。类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。

类加载器分类：

· **启动类加载器（Bootstrap ClassLoader）**：是虚拟机自身的一部分，用来加载Java_HOME/lib/目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚拟机识别的类库；

· **扩展类加载器（Extension ClassLoader）**：负责加载\lib\ext目录或Java. ext. dirs系统变量指定的路径中的所有类库；

· **应用程序类加载器（Application ClassLoader）**：负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

双亲委派模型：

如果一个类加载器收到了类加载的请求，它首先不会自己去加载这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类。

双亲委派模型的好处：

双亲委派模型保证了Java程序的稳定运行，可以避免类的重复加载（JVM 区分不同类的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类），也保证了 Java 的核心 API 不被篡改。如果没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现一些问题，比如我们编写一个称为 `java.lang.Object` 类的话，那么程序运行的时候，系统就会出现多个不同的 `Object` 类。

怎样取消双亲委派机制?

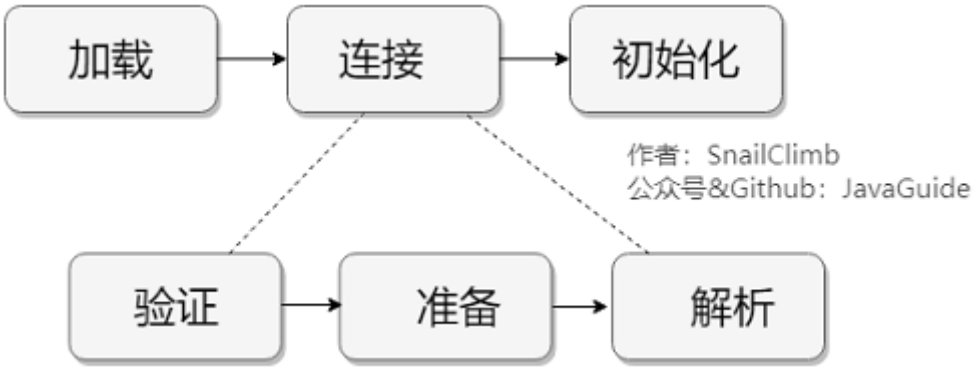
自定义加载器的话，需要继承 `ClassLoader` 。如果我们不想打破双亲委派模型，就重写 `ClassLoader` 类中的 `findClass()` 方法即可，无法被父类加载器加载的类最终会通过这个方法被加载。但是，如果想打破双亲委派模型则需要重写 `loadClass()` 方法。

自定义类加载器：

除了 `BootstrapClassLoader` 其他类加载器均由 Java 实现且全部继承自 `java.lang.ClassLoader`。如果我们要自定义自己的类加载器，很明显需要继承 `ClassLoader`。

5.类加载执行过程?

主要三步:**加载->连接->初始化。**连接过程又可分为三步:**验证->准备->解析。**



· **加载**：根据查找路径找到相应的 class 文件然后导入，加载到内存中；

· **连接**：

· 验证：检查加载的 class 文件的正确性；

· 准备：给类中的静态变量分配内存空间并设置类变量初始值；

· 解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；

· **初始化**：对静态变量和静态代码块执行初始化工作。

6.怎么判断对象是否可以被回收?

· **引用计数器**：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；

· **可达性分析**：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

> 无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

6.1 怎么判断无用类?

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

7.引用类型有哪些?

· **强引用**：发生 gc 的时候不会被回收。

· **软引用**：有用但不是必须的对象，在发生内存溢出之前会被回收。

· **弱引用**：有用但不是必须的对象，在下一次GC时会被回收。

· **虚引用**（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

8.垃圾回收算法?

· **标记-清除算法**：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。**适用**：适合在老年代进行垃圾回收，比如CMS收集器就是采用该算法进行回收的。

· **标记-整理算法**：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。**适用：**适合老年代进行垃圾收集，parallel Old和Serial old收集器就是采用该算法进行回收的。

· **复制算法**：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。**适用：**适合新生代区进行垃圾回收。serial new，parallel new和parallel scanvage收集器，就是采用该算法进行回收的。

· **分代算法**：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

优化垃圾回收方法的思路：分代收集算法
原理：根据对象存活的周期的不同将内存划分为几块，然后再选择合适的收集算法。

8.1 垃圾回收过程

在进行GC的时候，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。

> 大对象和长期存活的对象都进入老年代

9.垃圾回收器?

JVM具有四种类型的GC实现：

- 串行垃圾收集器
- 并行垃圾收集器
- CMS垃圾收集器
- G1垃圾收集器

可以使用以下参数声明这些实现：

```
-XX:+UseSerialGC
-XX:+UseParallelGC
-XX:+USeParNewGC
-XX:+UseG1GC
```

· **Serial**：最早的单线程串行垃圾回收器。

· **Serial Old**：Serial 垃圾回收器的老年版本，同样也是单线程的，可以作为 CMS 垃圾回收器的备选预案。

· **ParNew**：是 Serial 的多线程版本。

· **Parallel 和 ParNew 收集器**类似是多线程的，但 Parallel 是吞吐量优先的收集器，可以牺牲等待时间换取系统的吞吐量。

· **Parallel Old** 是 Parallel 老生代版本，Parallel 使用的是复制的内存回收算法，Parallel Old 使用的是标记-整理的内存回收算法。

· **CMS**：一种以获得最短停顿时间为目标的收集器，非常适用 B/S 系统。

· **G1**：一种兼顾吞吐量和停顿时间的 GC 实现，是 JDK 9 以后的默认 GC 选项。

总结：

（1）串行的，也就是采用单线程（比较老了），分类：**serial new（收集年轻代，复制算法）和serial old（收集老年代，标记整理）**，缺点：单线程，进行垃圾回收时暂时所有的用户线程。优点：实现简单。

（2）并行的，采用多线程，对于年轻代有两个： **parallel new（简称ParNew）（参考serial new的多线程版本）和parallel scavenge；**parallel scavenge是一个针对年轻代的垃圾回收器，采用复制算法，主要的优点是进行垃圾回收时不会停止用户线程（不会发生stop the world）

老年代回收器也有两种：**Parallel old**是parallel scavenge的我老年代设计的。**CMS（并发标记清除）**，它采用标记清除算法，采用这种的优点就是快，因此会尽快的进行回收，减少停顿时间。

（3）高级杀手(最新)：G1收集器，年轻代和老年代通吃，最新一代的技术。面向服务器端的垃圾收集器（并行+并发的垃圾收集器）。

10.CMS垃圾回收器?

> 是一款以获取最短回收停顿时间为目标的收集器，第一款真正意义上的并发收集器，第一次实现了让垃圾收集线程和用户线程**同时工作**

CMS 是英文 Concurrent Mark-Sweep 的简称，是老年代垃圾回收器，是以牺牲吞吐量为代价来获得最短回收停顿响应时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候回产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

过程：

1.初始标记 2.并发标记 3.重新标记 4.并发清除

特点：

1.并发收集、低停顿 2.对cpu资源敏感 3.无法处理浮动垃圾 4.标记-清除算法会产生大量碎片

10.1 G1垃圾回收器?

> 是一款面向服务器的垃圾收集器，主要针对配备多颗处理器以及大容量内存的机器，以极高效率满足GC停顿时间的同时还具备高吞吐量的性能特征

1.基于标记-整理算法，不产生内存碎片。

2.可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

G1收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾最多的区域。区域划分和优先级区域回收机制，确保G1收集器可以在有限时间获得最高的垃圾收集效率。

过程：

1.初始标记 2.并发标记 3.最终标记 4.筛选回收

特点：

1.并行和并发 2.分代收集 3.空间整合 4.可预测的停顿

> PS：**G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的Region****(这也就是它的名字 Garbage-First 的由来)**。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 GF 收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。

11.新生代垃圾回收器和老生代垃圾回收器？区别?

· **新生代回收器**：Serial、ParNew、Parallel Scavenge

· **老年代回收器**：Serial Old、Parallel Old、CMS

· **整堆回收器**：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内存利用率低；

老年代回收器一般采用的是标记-整理的算法进行垃圾回收。

12.分代垃圾回收器是怎么工作的?

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，**它的执行流程如下**：

· 把 Eden + From Survivor 存活的对象放入 To Survivor 区；

· 清空 Eden 和 From Survivor 分区；

· From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

13.JVM调优?

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 `jconsole 和 jvisualvm` 这两款视图监控工具。

- jconsole：用于对 JVM 中的内存、线程和类等进行监控；
- jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

13.1 JVM配置常用参数

1、堆参数

参数	描述
-Xms	设置 JVM 启动时堆内存的初始化大小
-Xmx	设置堆内存最大值
-Xmn	设置年轻代的空间大小，剩下的为老年代的空间大小
-XX:PermGen	设置永久代内存的初始化大小(JDK 1.8 开始废弃了永久代)
-XX:MaxPermGen	设置永久代的最大值
-XX:SurvivorRatio	设置 Eden 区和 Survivor 区的空间比例：Eden/S0 = Eden/S1 默认为 8
-XX:NewRatio	设置年老代和年轻代的比例大小，默认值是 2

2、回收器参数

参数	描述
-XX:+UseSerialGC	串行，Young 区和 Old 区都使用串行，使用复制算法回收，逻辑简单高效，无线程切换开销
-XX:+UseParallelGC	并行，Young 区：使用 Parallel scavenge 回收算法，会产生多个线程并行回收。通过-XX:ParallelGCThreads=n 参数指定有线程数，默认是 CPU 核数；Old 区：单线程
-XX:+UseParallelOldGC	并行，和 UseParallelGC 一样，Young 区和 old 区的垃圾回收时都使用多线程收集
-XX:+UseConcMarkSweepGC	并发，短暂停顿的并发的收集。Young 区：可以使用普通的或者 parallel 垃圾回收算法，由参数 -XX:+UseParNewGC 来控制；Old 区：只能使用 Concurrent Mark Sweep
-XX:+UseG1GC	并行的、并发的和增量式压缩短暂停顿的垃圾收集器。不区分 Young 区和 Old 区空间。它把堆空间划分为多个大小相等的区域。当进行垃圾收集时，它会优先收集存活对象较少的区域，因此叫 "Garbage First"

如上表所示，目前**主要有串行、并行和并发三种**，对于大内存的应用而言，串行的性能太低，因此使用到的主要是并行和并发两种。并行和并发 GC 的策略通过 `useParallelGC` 和 `UseConcMarkSweepGC` 来指定，还有一些细节的配置参数用来配置策略的执行方式。例如：`XX:ParallelGCThreads`，`XX:CMSInitiatingOccupancyFraction` 等。 通常：Young 区对象回收只可选择并行（耗时间），Old 区选择并发（耗 CPU）。

3、项目中的常用配置

参数设置	描述
-Xms4800m	初始化堆空间大小
-Xmx4800m	最大堆空间大小
-Xmn1800m	年轻代的空间大小
-Xss512k	设置线程栈空间大小
-XX:PermSize=256m	永久区空间大小(jdk 1.8 开始废弃了永久代)
-XX:MaxPermSize=256m	最大永久区空间大小
-XX:+UseStringCache	默认开启，启用缓存常用的字符串
-XX:+UseConcMarkSweepGC	老年代使用 CMS 收集器
-XX:+UseParNewGC	新生代使用并行收集器
-XX:ParallelGCThreads=4	并行线程数量 4
-XX:+CMSClassUnloadingEnabled	允许对类的元数据进行清理
XX:+DisableExplicitGC	禁止显示的 GC
-XX:+UseCMSInitiatingOccupancyOnly	表示只有达到阈值的之后才进行 CMS 回收
-XX:CMSInitiatingOccupancyFraction=68	设置 CMS 在老年代回收的阈值为 68%
-verbose:gc	输出虚拟机 GC 详情
-XX:+PrintGCDetails	打印 GC 详情日志
-XX:+PrintGCDateStamps	打印 GC 的耗时
-XX:+PrintTenuringDistribution	打印 Tenuring 年龄信息
-XX:+HeapDumpOnOutOfMemoryError	当抛出 OOM 时进行 HeapDump
-XX:HeapDumpPath=/home/admin/logs	指定 HeapDump 的文件路径或目录

4、常用组合

Young	Old	JVM Options
Serial	Serial	-XX:+UseSerialGC
Parallel scavenge	Parallel Old/Serial	-XX:+UserParallelGC -XX:+UserParallelOldGC
Serial/Parallel scavenge	CMS	-XX:+UserParNewGC -XX:+UseConcMarkSweepGC
G1		-XX:+UserG1GC

13.2 常用GC调优策略

- GC 调优原则

在调优之前，我们需要记住下面的原则：

> 多数的 Java 应用不需要在服务器上进行 GC 优化； 多数导致 GC 问题的 Java 应用，都不是因为我们参数设置错误，而是代码问题； 在应用上线之前，先考虑将机器的 JVM 参数设置到最优（最适合）； 减少创建对象的数量； 减少使用全局变量和大对象； GC 优化是到最后不得已才采用的手段； 在实际使用中，分析 GC 情况优化代码比优化 GC 参数要多得多。

- GC 调优目的

将转移到老年代的对象数量降低到最小； 减少 GC 的执行时间。

- GC 调优策略

策略 1：将新对象预留在新生代，由于 Full GC 的成本远高于 Minor GC，因此尽可能将对象分配在新生代是明智的做法，实际项目中根据 GC 日志分析新生代空间大小分配是否合理，适当通过“-Xmn”命令调节新生代大小，最大限度降低新对象直接进入老年代的情况。

策略 2：大对象进入老年代，虽然大部分情况下，将对象分配在新生代是合理的。但是对于大对象这种做法却值得商榷，大对象如果首次在新生代分配可能会出现空间不足导致很多年龄不够的小对象被分配的老年代，破坏新生代的对象结构，可能会出现频繁的 full gc。因此，对于大对象，可以设置直接进入老年代（当然短命的大对象对于垃圾回收老说简直就是噩梦）。`-XX:PretenureSizeThreshold` 可以设置直接进入老年代的对象大小。

策略 3：合理设置进入老年代对象的年龄，`-XX:MaxTenuringThreshold` 设置对象进入老年代的年龄大小，减少老年代的内存占用，降低 full gc 发生的频率。

策略 4：设置稳定的堆大小，堆大小设置有两个参数：`-Xms` 初始化堆大小，`-Xmx` 最大堆大小。

策略5：注意： 如果满足下面的指标，**则一般不需要进行 GC 优化：**

> MinorGC 执行时间不到50ms； Minor GC 执行不频繁，约10秒一次； Full GC 执行时间不到1s； Full GC 执行频率不算频繁，不低于10分钟1次。

14.Minor GC 与 Full GC ?

Minor GC是新生代GC：指的是发生在新生代的垃圾收集动作。由于java对象大都是朝生夕死的，所以Minor GC非常平凡，一般回收速度也比较i快。

Major GC/Full GC 是老年代GC：指的是发生在老年代的GC，出现Major GC一般经常会伴有Minor GC，Major GC的速度比Minor GC慢的多。

何时发生：

(1)Minor GC发生：当jvm无法为新的对象分配空间的时候就会发生Minor gc，所以分配对象的频率越高，也就越容易发生Minor gc。

(2)Full GC：发生GC有两种情况：

①当老年代无法分配内存的时候，会导致MinorGC

②当发生Minor GC的时候可能触发Full GC。由于老年代要对年轻代进行担保，由于进行一次垃圾回收之前是无法确定有多少对象存活，因此老年代并不能清除自己要担保多少空间，因此采取采用动态估算的方法：也就是上一次回收发送时晋升到老年代的对象容量的平均值作为经验值，这样就会有一个问题，当发生一次Minor GC以后，存活的对象剧增（假设小对象），此时老年代并没有满，但是此时平均值增加了，会造成发生Full GC

15.废弃常量和无用类怎么判断?

废弃常量：如果当前没有任何 String 对象引用该字符串常量的话，就说明该常量就是废弃常量。

无用类：1.该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
2.加载该类的 ClassLoader 已经被回收。
3.该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

16.创建一个对象的步骤?

类初始化（实例化）的顺序：

1、父类的静态变量，静态代码块
2、子类的静态变量，静态代码块
3、父类的变量、初始代码块、父类的构造器
4、子类的变量、初始代码块、子类的构造器

创建对象：

1、在堆区分配对象需要的内存：分配的内存包括本类和父类的==所有实例变量，但不包括任何静态变量

2、对所有实例变量赋默认值：将方法区内对实例变量的定义拷贝一份到堆区，然后赋默认值

3、执行实例初始化代码：初始化顺序是先初始化父类再初始化子类，初始化时先执行实例代码块然后是构造方法

4、指向引用

17.堆内存分配策略?

内存参数回顾：-Xms初始堆内存大小，-Xmx最大堆内存，相等不可扩展，-Xmn堆中新生代对象的内存大小，剩余的就是老年代内存。

1.优先使用Eden区域
2.大对象直接放入老年代
3.长期存活的对象将进入老年代
4.动态判断对象年龄
5.空间分配担保

18.JDK监控和故障处理工具调优

1.JDK命令行工具

这些命令在 JDK 安装目录下的 bin 目录下：

- `jps` (JVM Process Status)：类似 UNIX 的 `ps` 命令。用户查看所有 Java 进程的启动类、传入参数和 Java 虚拟机参数等信息；
- `jstat`（JVM Statistics Monitoring Tool）：用于收集 HotSpot 虚拟机各方面的运行数据;
- `jinfo` (Configuration Info for Java) : Configuration Info forJava,显示虚拟机配置信息;
- `jmap` (Memory Map for Java) :生成堆转储快照，保留现场
- `jhat` (JVM Heap Dump Browser) : 用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果;

- `jstack` (Stack Trace for Java):生成虚拟机当前时刻的线程快照，线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合。

2.YGC问题

> **概念梳理**
>
> **stop-the-world**：由于垃圾回收的过程中会设计对象的移动（s1移到s2），进而需要对对象引用的更新，为保证更新的正确性，需要暂停所有的其他线程，导致全局停顿。因此垃圾回收的一个原则是尽量减少stop-the-world

全称Young GC，也就是指YGC时间过长的问题该怎么解决？（接口超时）

1、检查监控，查看YGC的耗时，`jmap`命令生成快照产生堆文件
2、确认JVM配置`ps aux | grep "applicationName=adsearch"`
3、检查代码，可能由于程序的全局变量或者类静态变量上，导致大对象太多，会一直存活
4、对dump的堆内存文件进行分析，可以发现某个类所占的空间巨大、分析这个类的代码
5、分析YGC处理Reference的耗时。
　　1）对存活对象标注时间过长，比如重载了Object类的finalize方法，导致标志Final Reference耗时过长；或者String.intern方法使用不当，导致StringTable时间过长（命令：`-XX:+PrintReferenceGC`）
　　2）长周期对象积累过多：比如本地缓存太多，积累了太多存活对象；或者锁竞争严重导致线程阻塞，局部变量的生命周期变长。

3.FGC问题

> 问题描述：FGC频繁执行

1、检查JVM配置：`ps aux | grep "applicationName=adsearch"`
2、使用工具（jvisaulvm）观察老年代内存变化情况
3、通过jmap命令查看堆内存的对象：`jmap -histo 7276 | head -n20`
4、进一步dump堆内存文件进行分析：`jmap -dump:format=b,file=heap 7276`
5、通过代码分析可以对象

- **清楚程序的哪些角度会导致FGC**

  产生大量对象、内存泄露、jvm参数问题、频繁生成长声明周期对象，代码显示调用gc

- **排查问题常用命令**

```
# 查看堆内存各区域的使用率以及GC情况
jstat -gcutil -h20 pid 1000
# 查看堆内存中的存活对象，并按空间排序
jmap -histo pid | head -n20
# dump堆内存文件
jmap -dump:format=b,file=heap pid
```

4.OOM问题

> 问题描述：短时间内涌入了大量的对象，可能是内存溢出，也可能是内存泄露

产生原因：

1、**静态集合类**，生命周期和程序一致，里面的元素向是长生命周期对象，就是长生命周期对象持有大量短生命周期对象的引用，导致短生命周期对象不能被回收
2、**各种连接**，数据库连接，网络IO连接，当连接不在使用时没有显式close关闭
3、**变量不合理作用域**，就是一个变量的作用范围大于使用范围，要即使把无用对象设为null
4、**内部类持有外部类**

解决方案：

内存过小，设置JVM参数打开OOMError详情，也可以jmap查看java进程

1、使用dmesg命令插了系统出现OOM的日志
2、ps命令查看java进程，top命令查看高占用%MEM一列
4、jstat命令对执行进程查看统计信息
5、jmap命令查看当前堆中所有每个类的实例数量和内存占用，或者然后转成堆内存快照来分析（使用MAT工具）
6、找到具体的代码段，修改代码调优，接口重新压测

5.CPU100%问题

1、`top -c`，然后按P可以按照CPU使用率进行排序
2、根据PID查出消耗最高的进程2607，使用命令`top -Hp 2609`
3、将查询出的pid转为十六进制，转换结果为b26
4、导出我们的进程快照：`jstack -l 2609 > ./2609.stack`
5、然后查看这个线程到底做了啥：`cat 2609.stack | grep 'b26' -C 8`

2、面试题

1.新生代有关面试题

1）为什么会有新生代?

如果不分代，所有对象全部在一个区域，每次GC都需要对全堆进行扫描，存在效率问题。分代后，可分别控制回收频率，并采用不同的回收算法，确保GC性能全局最优。

2）为什么新生代采用复制算法?

新生代的对象朝生夕死，大约90%的新建对象可以被很快回收，复制算法成本低，同时还能保证空间没有碎片。虽然标记整理算法也可以保证没有碎片，但是由于新生代要清理的对象数量很大，将存活的对象整理到待清理对象之前，需要大量的移动操作，时间复杂度比复制算法高。

3）为什么新生代分为两个survivor区?

为了节省空间考虑，如果采用传统的复制算法，只有一个Survivor区，则Survivor区大小需要等于Eden区大小，此时空间消耗是8 * 2，而两块Survivor可以保持新对象始终在Eden区创建，存活对象在Survivor之间转移即可，空间消耗是8+1+1，明显后者的空间利用率更高。

4）新生代的实际可用空间是多少?

YGC后，总有一块Survivor区是空闲的，因此新生代的可用内存空间是90%。在YGC的log中或者通过`jmap -heap pid`命令查看新生代的空间时，如果发现capacity只有90%，不要觉得奇怪。

5）Eden区是如何加速分配内存的?

HotSpot虚拟机使用了两种技术来加快内存分配。分别是bump-the-pointer和TLAB（Thread Local Allocation Buffers）。

由于Eden区是连续的，因此bump-the-pointer在对象创建时，只需要检查最后一个对象后面是否有足够的内存即可，从而加快内存分配速度。

TLAB技术是对于多线程而言的，在Eden中为每个线程分配一块区域，减少内存分配时的锁冲突，加快内存分配速度，提升吞吐量。

十七、Oracle

1、面试题

1.Mysql和Oracle的区别

1、Mysql轻量级开源免费，Oracle重量级安装繁琐收费
2、事务隔离级别：Mysql默认读已提交，自动提交事务；Oracle默认可重复读，需要手动提交事务
3、库函数、使用语法、数据类型上的区别：主键、merge_into、分页，日期函数，nvarchar2
4、表空间：Mysql没有用户表空间的概念，Oracle是以用户来区分表空间
5、逻辑备份：Mysql逻辑备份要锁定数据才能保证数据一致，Oracle逻辑备份不需要锁定数据

十八、ElasticSearch

索引-类型-文档-字段

1、常用命令

```
#基础查询

Restful API接口
#查看集群健康信息
GET /_cat/health?v
#查看每个操作返回结果字段的意义
GET /_cat/health?help
#查看集群中的节点信息
GET /_cat/nodes?v
#查看集群中的索引信息
GET /_cat/indices?v
#简化写法
GET /_cat/indices?v&h=health,status,index

#索引操作
#创建索引、删除索引
PUT /baizhi    DELETE /baizhi

#创建类型type
PUT /baizhi # 创建index(baizhi)并添加类型mapping（_doc）
{
  "mappings": {
    "_doc": {
      "properties": {
```

```
        "title":    { "type": "text"  },  # 注意： 字符串常用类型：text类型会分词
keyword类型不会分词
        "name":     { "type": "text"  },
        "age":      { "type": "integer" },
        "created":  {
          "type":   "date",
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
  }
}
或者
POST /baizhi/user # 创建index(baizhi)后，在指定index中添加类型mapping(user)
{
  "user": {
      "properties": {
        "id":    { "type": "text"  },
        "name":     { "type": "text"  },
        "age":      { "type": "integer" },
        "created":  {
          "type":   "date",
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
}

#新增单个文档
PUT /baizhi/_doc/1 # put /索引名/类型名/id
{                  # request body
  "name":"zs",
  "title":"张三",
  "age":18,
  "created":"2018-12-25"
}
#查询单个文档
GET /baizhi/_doc/1  # 语法： GET /索引名/类型名/id
{
  "_index": "baizhi",
  "_type": "_doc",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {
    "name": "zs",
    "title": "张三",
    "age": 18,
    "created": "2018-12-25"
  }
}

#批量，每个json串不能换行
POST /_bulk
{ "delete": { "_index": "test_index", "_type": "test_type", "_id": "3" }}
{ "create": { "_index": "test_index", "_type": "test_type", "_id": "12" }}
```

```
{ "test_field":    "test12" }
{ "index":  { "_index": "test_index", "_type": "test_type", "_id": "2" }}
{ "test_field":    "replaced test2" }
{ "update": { "_index": "test_index", "_type": "test_type", "_id": "1",
"_retry_on_conflict" : 3} }
{ "doc" : {"test_field2" : "bulk test1"} }
a、delete：删除一个文档，只要1个json串就可以了
b、create：PUT /index/type/id/_create，强制创建
c、index：普通的put操作，可以是创建文档，也可以是全量替换文档
d、update：执行的partial update操作

#查询所有文档
get /index/type/_search
-- 返回结果
took：耗时时间
time_out:是否超时
_shards：分片数量

--------------------------------------------------------------------------------
------------------
#检索的方式:uri参数、uri请求体
#查询query DSL，检索都是以_search结尾
#查询所有
GET bank/_search
{
  "query": {
    "match_all": {}
  },
  "from": 0,
  "size": 5,
  "sort": [
    {
      "account_number": {
        "order": "desc"
      }
    }
  ]
}
#match_all ： 查询所有
#match ： 精确匹配，对于字符串的搜索会进行全文搜索，最终按照评分排序
#多字段匹配 ： multi_match
#短句匹配 ： match_phrase
#排序 ： sort
#分页 ： from，size
#指定要查询商品的名称和价格
GET /ecommerce/product/_search
{
  "query": { "match_all": {} },
  "_source": ["name", "price"] #返回指定的字段
}
#多条件过滤查询 bool - 包含多个查询条件
GET /test/_search
{
  "query": {
    "bool": {      # 多条件搜索，内部的若干条件，只要有正确结果，即可
```

```
      "must": [ # 必须，内部若干条件，必须都匹配才有结果（类似mysql中的and关键字，与之对应的是should，类似mysql中的or关键字）
        {
          "match": {
            "name": "张三"
          }
        },
        {
          "match": {
            "sex": "女"
          }
        }
      ]
    }
  }
}
#should - 满足一个or
#must - 全部满足and
#must_not - 必须不 not
#range 范围查询
#全文检索 match
#短语搜索 match_phrase
#高亮搜索 highlight
#聚合查询
term - 按照文本text查询
terms - 按照field名称分组
range - 按照value的范围分组
avg - 分组后求平均数
```

2、基本概念

Elasticsearch 本身就是分布式的，因此即便你只有一个节点，Elasticsearch 默认也会对你的数据进行分片和副本操作，当你向集群添加新数据时，数据也会在新加入的节点中进行平衡

索引（indices）----------Databases 数据库
类型（type）----------Table 数据表
文档（Document）----------Row 行
字段（Field）----------Columns 列
映射配置————————————类似表中字段属性类型type

1.type映射配置的数据类型

- string

  text：可分词，不可参与聚合
  keyword：不可分词，数据会作为完整字段进行匹配，可以参与聚合

- numerical数值类型

  基本数据类型：long、interger、short、byte、double、float、half_float
  浮点数高精度类型：scaled_float

- date日期类型

  可以对日期格式化为字符串存储，但是建议我们存储为毫秒值，存储为 long

2.倒排索引

采用分词策略，形成词和文章的关系映射表，词->文章

3.API

ESClient，RestHighLevelClient

3、面试题

1.如何解决深度分页下的跳页问题?

分页方式	性能	优点	缺点	场景
from + size	低	灵活性好，实现简单	深度分页问题	数据量比较小，能容忍深度分页问题
Scroll	中	解决了深度分页问题	无法反应数据的实时性（快照）维护成本高，需要维护一个 scroll_id	一次性进行批量数据的导出，或查询海量结果集的数据
Search After	高	性能最好不存在深度分页问题能够反映数据的实时变更	实现复杂，需要有一个全局唯一的字段连续分页的实现会比较复杂，因为每一次查询都需要上次查询的结果，它不适用于大幅度跳页查询	海量数据的分页

- 如果数据量小（from+size 在 10000 条内），或者只关注结果集的 TopN 数据，可以使用 from/size 分页，简单粗暴
- 数据量大，深度翻页，后台批处理任务（数据迁移）之类的任务，使用 scroll 方式
- 数据量大，深度翻页，用户实时、高并发查询需求，使用 search after 方式

十九、Vue

二十、Linux

1、命令

grep：查询数据量大文件使用正则表达式指定
cut：取出指定文件的行
diff：比较俩文件的不同
du：显示指定文件或目录大小
df：查看系统磁盘使用情况
losf：列出各种信息
traceroute：查看数据报的路由途径
route：设置静态路由
vmstat：查看内存磁盘使用详细信息
free：查看内存使用情况
sar：性能分析

rysnc和scp区别：用rsync做文件复制比scp速度快，rsync只对差异文件做更新，scp是把所有文件都复制过去。

2、其他

2.1 .ssh文件夹下文件功能解释

- kown_hosts：记录ssh访问过的计算机的公钥 public key
- id_rsa：生成的私钥
- id_rsa.pub：生成的公钥
- authorized_keys：存放授权过的无密登录服务器的公钥

2.用户态 内核态转化原理

2.1 用户态内核态区别

1）为了区分不同的程序的不同权限，人们发明了内核态和用户态的概念。
2）用户态和内核态是操作系统的两种运行级别，两者最大的区别就是特权级不同。用户态拥有最低的特权级，内核态拥有较高的特权级。 运行在用户态的程序不能直接访问操作系统内核数据结构和程序。
3）内核态和用户态之间的转换方式主要包括：系统调用，异常和中断。

2.2 操纵系统为什么要分内核态和用户态

- 为了安全性。在cpu的一些指令中，有的指令如果用错，将会导致整个系统崩溃。分了内核态和用户态后，当用户需要操作这些指令时候，内核为其提供了API，可以通过系统调用陷入内核，让内核去执行这些操作。

2.3 用户态转换到内核态的三种方式

1、系统调用 2、异常 3、外围设备的中断

二十一、Docker

```
docker save -o .tar image:version
docker load -i .tar
docker tag old_image new_image
```

二十二、Nginx

1.概念

Nginx由三部分组成：全局块，events块，http块

2.面试题

1、如何配置反向代理

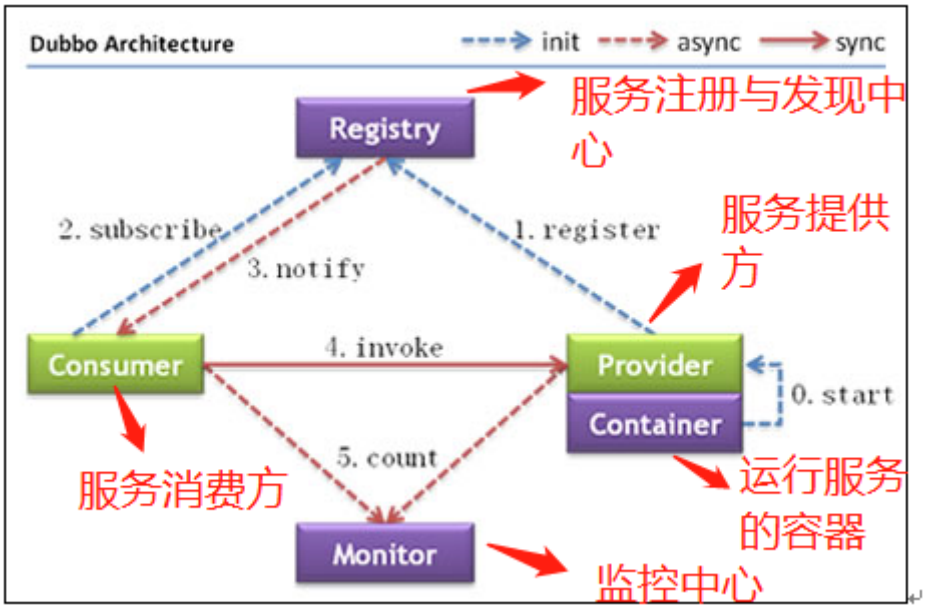
在配置文件配置http模块

```
...
http{
    ...
    # 设定负载均衡的服务器列表
    upstream myServer{
        # 使用ip_hash策略、响应时间、默认轮询
        ip_hash;    #fair
        server 192.168.43.0:1234 weight=1;
        server 192.168.43.1:1234 weight=2;
        server 192.168.43.2:1234 weight=3;
    }
    ...
    server{
        listen 80;  # 监听80端口
        server_name localhost;  # 通过域名访问
        # 默认请求
        location / {
            root /root; # 定义默认网站根目录位置
            index index.html # 定义默认首页
        }
    }
    # 配置其他监听
    server{
        listen 81;
        servername localhost;
        # 反向代理，代理到指定的服务器
        # 根据不同的请求路径交给不同的服务来处理
        location /product {
            proxy_pass http:myServer
        }
    }
    #配置动静分离,动态接口代理到远程服务，静态文件放到本机
    location /api/ {
        proxy_pass http:192.168.43.10:8080
    }
    location  ~ .*\.(gif|jpg|jpeg|bmp|png|ico|txt|js|css)$ {
        root /data/staic/;
        expires 3d; //缓存三天
    }
}
```

二十四、Dubbo

1.概念

是一款高性能、轻量级的开源Java RPC 框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。



- **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

调用关系说明：

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 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。
5. 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。
6. 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。

重要知识点总结：

- **注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小**
- **监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示**
- **注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外**
- **注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者**
- **注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表**
- **注册中心和监控中心都是可选的，服务消费者可以直连服务提供者**
- **服务提供者无状态，任意一台宕掉后，不影响使用**
- **服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复**

1.1 Dubbo负载均衡策略

负载均衡就是为了避免单个服务器响应同一请求，容易造成服务器宕机、崩溃等问题，我们从负载均衡的这四个字就能明显感受到它的意义。

Dubbo：默认为`random`随机调用，可以自行扩展负载均衡策略。

- **Random随机**，按权重设置随机概率。权重越大，落地概率越大
- **RoundRobin轮询**，不推荐，按照公约后的权重设置轮询比率，会使得慢机器的请求都卡在一起
- **LeastActive最少活跃调用数**，在相同活跃数随机，使慢的服务器接受到更少的请求
- **ConsistentHash一致性Hash**，相同参数的请求总是发到同一个提供者，当某一台提供者挂掉时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动

2.RPC

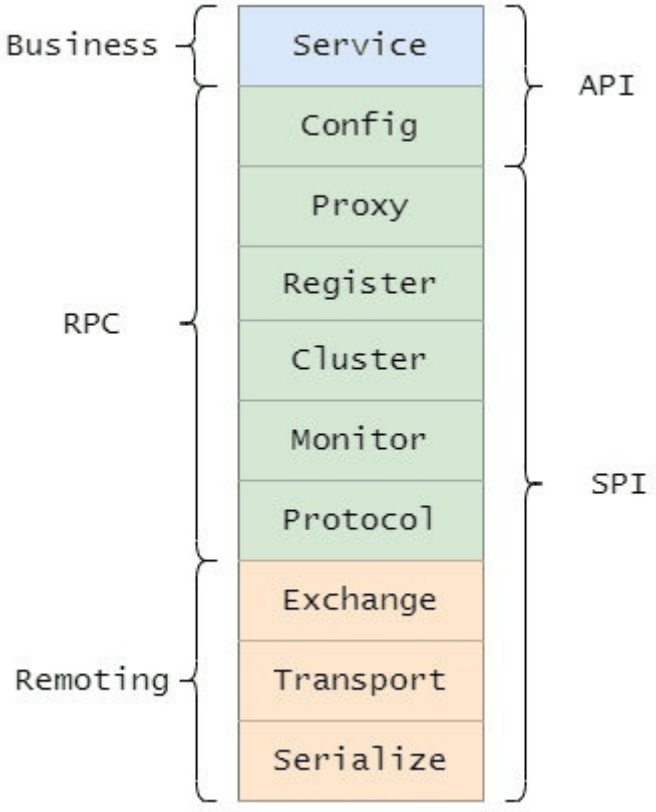
原理

RPC就是使调用远程服务的方法像调用本地方法一样。底层实现：面向接口编程的公共jar包和动态代理实现接口。服务提供者利用反射+动态代理暴露出接口，服务消费者也是代理方式直接消费就可以。

RPC 只是一种概念、一种设计，就是为了解决 **不同服务之间的调用问题**, 它一般会包含有 **传输协议** 和 **序列化协议** 这两个。实现 RPC 的可以传输协议可以直接建立在 TCP 之上，也可以建立在 HTTP 协议之上。**大部分 RPC 框架都是使用的 TCP 连接（gRPC使用了HTTP2）**

3.分层架构

大体分为三层：Business业务层、RPC层、Remoting远端层，总共十层



而分 API 层和 SPI 层这是 Dubbo 成功的一点，**采用微内核设计+SPI扩展**，使得有特殊需求的接入方可以自定义扩展，做定制的二次开发。

- Service，业务层，就是咱们开发的业务逻辑层。

- Config，配置层，主要围绕 ServiceConfig 和 ReferenceConfig，初始化配置信息。
- Proxy，代理层，服务提供者还是消费者都会生成一个代理类，使得服务接口透明化，代理层做远程调用和返回结果。
- Register，注册层，封装了服务注册和发现。
- Cluster，路由和集群容错层，负责选取具体调用的节点，处理特殊的调用要求和负责远程调用失败的容错措施。
- Monitor，监控层，负责监控统计调用时间和次数。
- Portocol，远程调用层，主要是封装 RPC 调用，主要负责管理 Invoker，Invoker代表一个抽象封装了的执行体，之后再做详解。
- Exchange，信息交换层，用来封装请求响应模型，同步转异步。
- Transport，网络传输层，抽象了网络传输的统一接口，这样用户想用 Netty 就用 Netty，想用 Mina 就用 Mina。
- Serialize，序列化层，将数据序列化成二进制流，当然也做反序列化。

SPI

是 JDK 内置的一个服务发现机制，**它使得接口和具体实现完全解耦。**我们只声明接口，具体的实现类在配置中选择。

具体的就是你定义了一个接口，然后在`META-INF/services`目录下**放置一个与接口同名的文本文件**，文件的内容为**接口的实现类**，多个实现类用换行符分隔。

Dubbo工作原理

1. 服务启动的时候，provider和consumer根据配置信息，连接到注册中心register，分别向注册中心注册和订阅服务
2. register根据服务订阅关系，返回provider信息到consumer，同时consumer会把provider信息缓存到本地。如果信息有变更，consumer会收到来自register的推送（监听）
3. consumer生成代理对象，同时根据负载均衡策略，选择一台provider，同时定时向monitor记录接口的调用次数和时间信息
4. 拿到代理对象之后，consumer通过代理对象发起接口调用
5. provider收到请求后对数据进行反序列化，然后通过代理调用具体的接口实现

4.调用过程

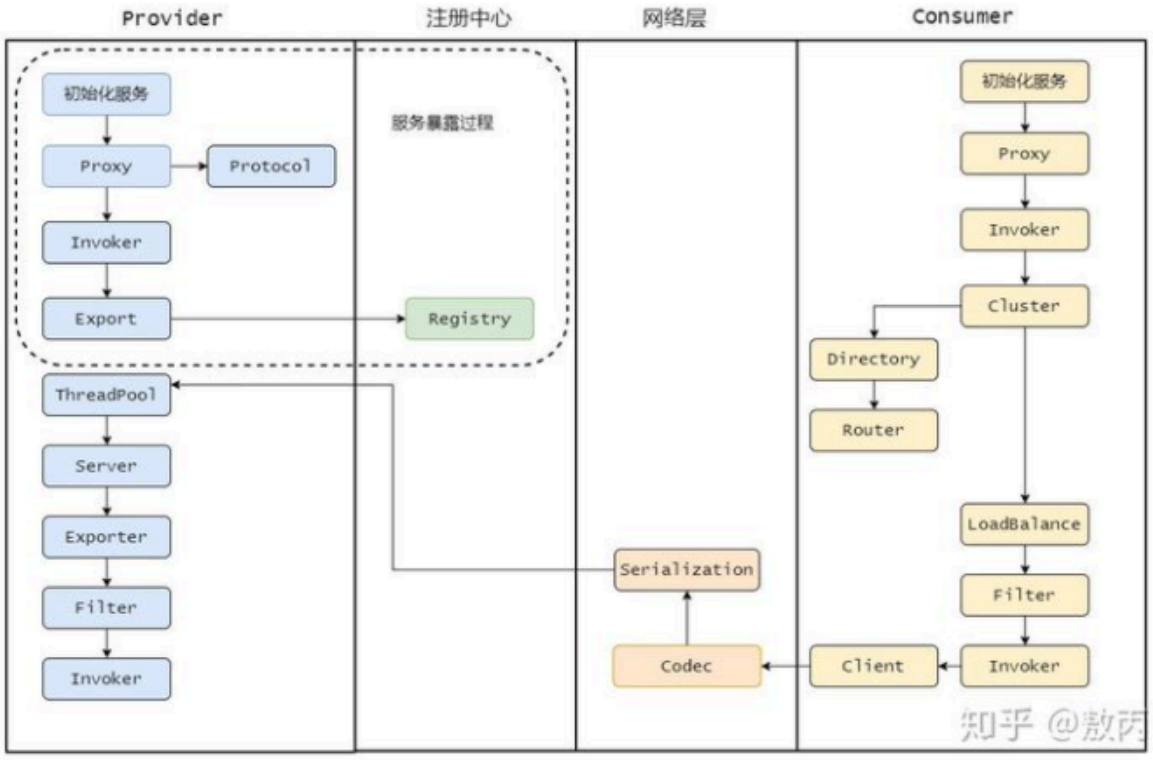
服务暴露过程

首先 Provider 启动，通过 Proxy 组件根据具体的协议 Protocol 将需要暴露出去的接口封装成 Invoker，Invoker 是 Dubbo 一个很核心的组件，代表一个可执行体。

然后再通过 Exporter 包装一下，这是为了在注册中心暴露自己套的一层，然后将 Exporter 通过 Registry 注册到注册中心。 这就是整体服务暴露过程。

消费过程

接着我们来看消费者调用流程（把服务者暴露的过程也在图里展示出来了，这个图其实算一个挺完整的流程图了）。



首先消费者启动会向注册中心拉取服务提供者的元信息，然后调用流程也是从 Proxy 开始，毕竟都需要代理才能无感知。

Proxy 持有一个 Invoker 对象，调用 invoke 之后需要通过 Cluster 先从 Directory 获取所有可调用的远程服务的 Invoker 列表，如果配置了某些路由规则，比如某个接口只能调用某个节点的那就再过滤一遍 Invoker 列表。

剩下的 Invoker 再通过 LoadBalance 做负载均衡选取一个。然后再经过 Filter 做一些统计什么的，再通过 Client 做数据传输，比如用 Netty 来传输。

传输需要经过 Codec 接口做协议构造，再序列化。最终发往对应的服务提供者。

服务提供者接收到之后也会进行 Codec 协议处理，然后反序列化后将请求扔到线程池处理。某个线程会根据请求找到对应的 Exporter ，而找到 Exporter 其实就是找到了 Invoker，但是还会有一层层 Filter，经过一层层过滤链之后最终调用实现类然后原路返回结果。

完成整个调用过程！

5.Dubbo特性、优势

为什么选择Dubbo？

1. **负载均衡**——同一个服务部署在不同的机器时该调用那一台机器上的服务。
2. **服务调用链路生成**——随着系统的发展，服务越来越多，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。Dubbo 可以为我们解决服务之间互相是如何调用的。
3. **服务访问压力以及时长统计、资源调度和治理**——基于访问压力实时管理集群容量，提高集群利用率。
4. **服务降级**——某个服务挂掉之后调用备用服务。

6.面试

6.1 分布式好处?

1、访问量大减少服务器压力，利于系统扩展和维护，提高整个系统的性能
2、同时开发提高效率

6.2 Zookeeper宕机

在实际生产中，假如zookeeper注册中心宕掉，一段时间内服务消费方还是能够调用提供方的服务的，实际上它使用的本地缓存进行通讯，这只是dubbo健壮性的一种体现。

dubbo的健壮性表现：

1. 监控中心宕掉不影响使用，只是丢失部分采样数据
2. 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
3. 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
4. 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
5. 服务提供者无状态，任意一台宕掉后，不影响使用
6. 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

我们前面提到过：注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小。所以，我们可以完全可以绕过注册中心——采用 **dubbo 直连** ，即在服务消费方配置服务提供方的位置信息。

> PS：Dubbo也可以不走注册中心采用直连。。。

6.3 常用的RPC框架总结

- **RMI（JDK自带）：** JDK自带的RPC，有很多局限性，不推荐使用。
- **Dubbo:** Dubbo是 阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和 Spring框架无缝集成。目前 Dubbo 已经成为 Spring Cloud Alibaba 中的官方组件。
- **gRPC** ： gRPC是可以在任何环境中运行的现代开源高性能RPC框架。它可以通过可插拔的支持来有效地连接数据中心内和跨数据中心的服务，以实现负载平衡，跟踪，运行状况检查和身份验证。它也适用于分布式计算的最后一英里，以将设备，移动应用程序和浏览器连接到后端服务。
- **Hessian：** Hessian是一个轻量级的remotingonhttp工具，使用简单的方法提供了RMI的功能。 相比WebService，Hessian更简单、快捷。采用的是二进制RPC协议，因为采用的是二进制协议，所以它很适合于发送二进制数据。
- **Thrift：** Apache Thrift是Facebook开源的跨语言的RPC通信框架，目前已经捐献给Apache基金会管理，由于其跨语言特性和出色的性能，在很多互联网公司得到应用，有能力的公司甚至会基于thrift研发一套分布式服务框架，增加诸如服务注册、服务发现等功能。

6.4 集群容错方式有哪些?

1. Failover Cluster失败自动切换：dubbo的默认容错方案，当调用失败时自动切换到其他可用的节点，具体的重试次数和间隔时间可用通过引用服务的时候配置，默认重试次数为1也就是只调用一次。
2. Failback Cluster快速失败：在调用失败，记录日志和调用信息，然后返回空结果给consumer，并且通过定时任务每隔5秒对失败的调用进行重试

3. Failfast Cluster失败自动恢复：只会调用一次，失败后立刻抛出异常
4. Failsafe Cluster失败安全：调用出现异常，记录日志不抛出，返回空结果
5. Forking Cluster并行调用多个服务提供者：通过线程池创建多个线程，并发调用多个provider，结果保存到阻塞队列，只要有一个provider成功返回了结果，就会立刻返回结果
6. Broadcast Cluster广播模式：逐个调用每个provider，如果其中一台报错，在循环调用结束后，抛出异常。

7.使用Dubbo

这里使用spring-boot-stater-dubbo.

1、引入公共依赖
2、在api项目中定义服务接口
3、配置提供者模块，配置扫描包、注册中心、服务调用失败重试次数、序列化方式等
4、提供者模块实现服务接口，使用Dubbo的`@Service`注解
5、配置消费者模块，配置扫描包、注册中心、服务启动时检查服务是否可用，失败重试
6、消费者模块使用`@Reference`注解注入服务接口来调用实现

二十五、项目

1.分布式电商

1.SpringCache缓存

好处：之前我们的程序想要使用缓存，就要与框架耦合，但是仍然需要显示在代码中去调用与缓存有关的接口和方法，使用不方便。Spring Cache就是一个这个框架。它利用了AOP，实现了基于注解的缓存功能，并且进行了合理的抽象，业务代码不用关心底层是使用了什么缓存框架，只需要简单地加一个注解，就能实现缓存功能了。

Spring从3.1开始定义了Cache和CacheManager接口来统一不同的缓存技术，并支持JCache（JR-107）注解来简化我们的开发；

Cache接口为缓存的组件规范定义，包含缓存的各种操作集合；Cache接口下Spring提供了各种xxxCache的实现，如RedisCache，EhCacheCache，ConcurrentMapCache等；

每次调用需要缓存功能的方法时，Spring会检查指定参数的指定目标方法是否已经被调用过，如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法并缓存结果后直接返回给用户。

1.常用注解

@Cacheable：注解表示这个方法有了缓存的功能，方法的返回值会被缓存下来，下一次调用该方法前，会去检查是否缓存中已经有值，如果有就直接返回，不调用方法。如果没有，就调用方法，然后把结果缓存起来。这个注解**一般用在查询方法上**

CachePut：会把方法的返回值put到缓存里面缓存起来，供其它地方使用。它**通常用在新增方法上。**

CacheEvict：会清空指定缓存。**一般用在更新或者删除的方法上。**

Caching：一个方法操作多个缓存。配置属性cacheable

CacheConfig：类级别的注解。可以在类级别上配置cacheNames、keyGenerator、cacheManager、cacheResolver等

2.缓存的问题

常规操作数据（读多写少）完全可以使用SpringCache

- **双写不一致**（又写数据库又写缓存）

  原因：由于时间差，会出现脏数据，但是在缓存过期后还是可以得到最新数据

  解决：更新的时候，先删除缓存，再更新数据库。所以Spring Cache的@CacheEvict会有一个beforeInvocation的配置。

- **缓存穿透**

  原因：大量的请求去查询一个一定不存在缓存中的数据，将查询全部落在数据库，但是数据库也不存在。

  解决：将一定为null的结果缓存，并短暂加入过期时间。

  SpringCache配置：`cache-null-values=true`

- **缓存雪崩**

  原因：相同过期时间的key同一时间大量失效，请求全部落在数据库导致数据库压力过大。

  解决：在原失效时间的基础上加上一个随机值。

  SpringCache配置：`spring.cache.redis.time-to...`

- **缓存击穿：**

  原因：热点key高并发访问当key失效时，所有请求落在数据库。

  解决：加锁，大量的并发请求只让一个人去查数据库，其他人就等待，知道缓存中又有数据。

  SpringCache配置：`syn=true`

3.双写不一致问题讨论

- **先更新数据库，再更新缓存**（不推荐）

  问题：主要就是数据库和缓存中间的时间差，缓存会出现脏数据

  解决：需要保证操作的事务性，加锁

- **先删缓存，再更新数据库**（有点问题）

  问题：缓存还是会出现脏数据（如果查询快的话）

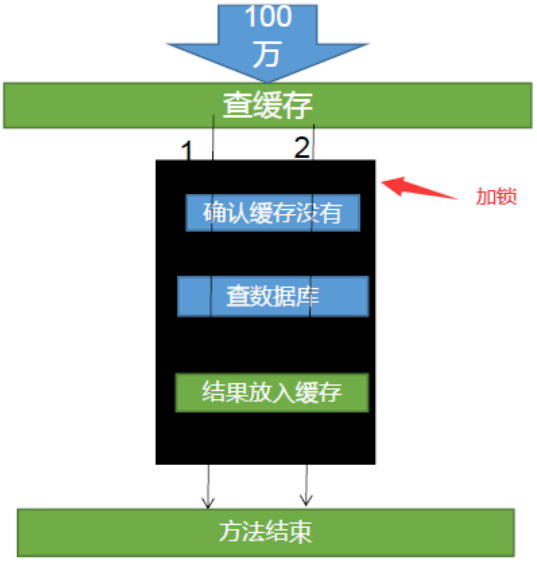
  解决：延时双删策略（睡上一会将脏数据也删掉），如果吞吐量降低，就将第二次删除做成多线程异步删除

- **先更新数据库，再删除缓存**CAP（极端情况有问题）

  问题：脏数据发生概率极低；缓存也有可能删除失败

  解决：失败重试（消息队列重试、订阅binlog）；还是加锁，经常修改的数据应该直接查询数据库

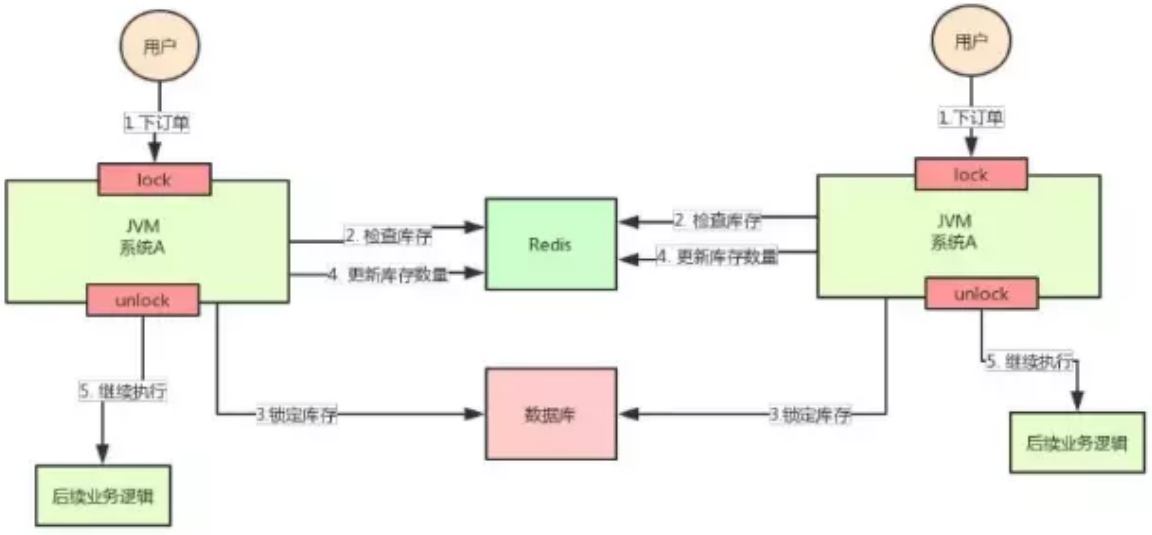
2.Redissson分布式锁



Redisson是Redis官方推荐的Java版的Redis客户端。它提供的功能非常多，也非常强大，此处我们只用它的分布式锁功能。

使用场景：本地锁->不应该在所有请求没缓存的情况下都去请求数据库
分布式锁->当缓存不存在时，不应该每个分布式服务都去做相同的请求去请求数据库，而应该只有一个服务只请求一遍数据库就可以了。

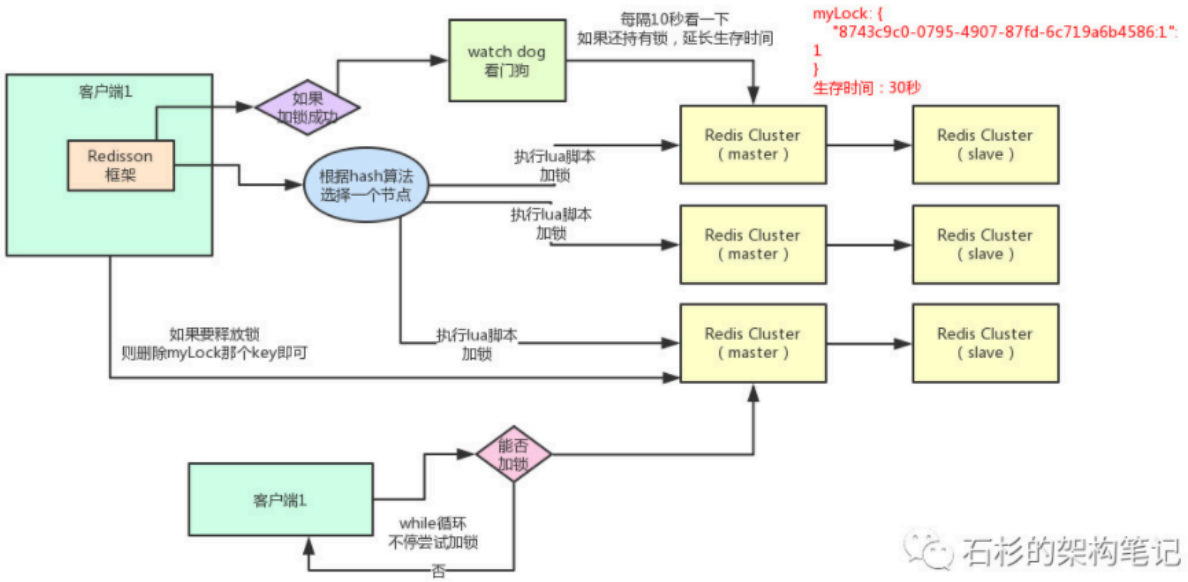
其他使用场景：



库存超卖，当多个下订单的请求同时进来，用一个分布式锁将2、3、4步骤锁住，等待执行完毕后再释放锁。

但是当两个用户的请求同时进来，落在了不同的机器上，那么这两个请求同时执行，还是会出现库存超卖的问题，这时候就需要保证两台机器加锁使用的是同一个锁，也就是分布式锁。

0.分布式锁原理



1.加锁机制

线程去获取锁，获取成功: 执行lua脚本（setnx）（原子性），保存数据到redis数据库。PS：仅仅选择一台机器

线程去获取锁，获取失败: 一直通过while循环尝试获取锁，获取成功后，执行lua脚本，保存数据到redis数据库。

1）锁互斥机制

当第二个线程尝试加锁时，执行同样的lua脚本，首先if会判断锁存不存在，发现key锁存在，接着第二个if判断锁是否包含线程2的id，然后线程2获取到pttl myLock返回的一个数字，这个数字代表了锁key的剩余生存时间，然后线程2会一直while循环，不停的尝试加锁。

2.看门狗机制

如果该线程当前的业务还没有执行完，每隔几秒检查一下，就不断的延长锁的持续时间。

3.为啥用lua脚本

redis是单线程，这句话执行逻辑是原子性操作

4.可重入锁机制

原理:

1、Redis存储锁的数据类型是 Hash类型
2、Hash数据类型的key值包含了当前线程信息。

1）释放锁机制

每次对加锁的次数减1，当加锁的次数是0了，就del myLock从redis删除这把锁。

5.缺点

在哨兵模式或者主从模式下，如果 master实例宕机的时候，可能导致多个客户端线程同时完成加锁（都以为自己加上了锁），会产生脏数据。

3.分布式Session共享方案

1、nginx的ip_hash策略，hash一致性

2、统一存储。使用SpringSession存入到Redis
1）导入依赖
2）@EnableRedisHttpSession开启SpringSession
3）配置文件：设置Redis序列化以及子域共享
4）代码中使用

3、Session复制。Tomcat原生支持，只需要修改配置文件

4.Seata分布式事务

Seata中有两种分布式事务实现方案，AT及TCC；剩下的还有SAGA模式和XA等事务模式。

- AT模式主要关注多 DB 访问的数据一致性，当然也包括多服务下的多 DB 数据访问一致性问题（业务侵入小），是一种两阶段提交。
- TCC 模式主要关注业务拆分，在按照业务横向扩展资源时，解决微服务间调用的一致性问题

0、分布式事务解决方案

- **2PC：准备、准备成功、提交**

  1、**概念：**

  就是二阶段提交，第一阶段：执行代码；第二阶段：提交事务或回滚事务。如果第一阶段所有参与者都返回准备成功，那么协调者则向所有参与者发送提交事务命令，然后等待所有事务都提交成功之后，返回事务执行成功。如果第一阶段有一个参与者返回失败，那么协调者就会向所有参与者发送回滚事务的请求，即分布式事务执行失败。（准备成功就提交事务、准备失败就回滚事务）

  2、**第二阶段失败的情况：**

  如果第二阶段提交事务失败，就会不断重试，直到所有参与者都提交事务成功，最后再不行就人工处理。
  如果第二阶段回滚事务失败，就会不断重试，直到所有参与者都回滚事务成功，不然那些在第一阶段准备成功的参与者会一直阻塞。

  3、**具体细节：**

  2PC是一个同步阻塞协议，第一阶段的中间协调者会等待所有参与者响应才会进行下一步操作。当然也有协调者超时机制，假设因为超时原因没有收到某些参与者的响应则会判断事务失败，向所有参与者发送事务回滚命令。

  4、**协调者挂了：**

  通过选举算法得到新的协调者，要考虑的情况就很复杂了。

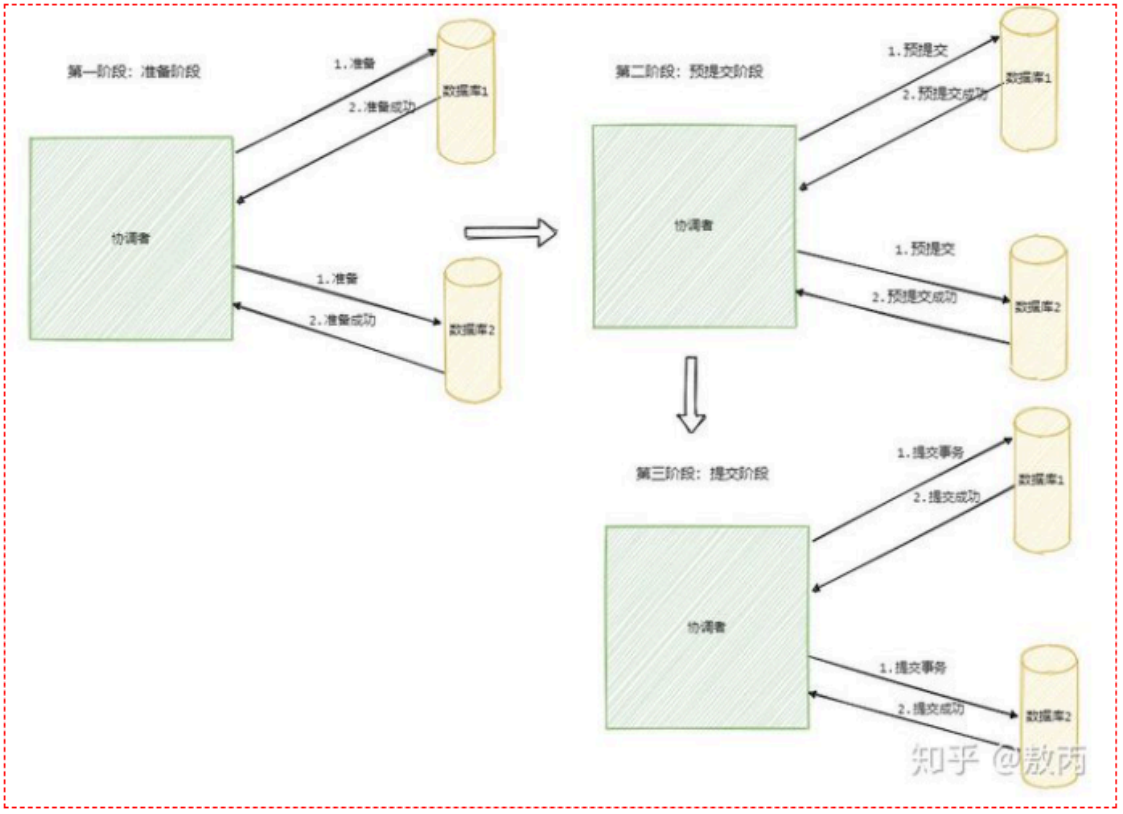
  5、**总结：**

  2PC是一种尽量保证强一致性的分布式事务，因此它是同步阻塞的，就会导致长久资源锁定问题，导致总体效率低，并且存在单点故障的问题，极端情况下也会存在数据不一致的风险。根据具体的情形可以变形，例如Tree 2PC、Dynamic 2PC。还有2PC只是适用于数据库层面的分布式事务场景，而我们的业务需求有的时候不仅仅关乎数据库，就显得有些不行了。

- **3PC：准备、预提交、提交**

  概念：

  为了解决2PC存在的问题引入的方案，在参与者中引入了超时机制，并且新增了一个阶段使得参与者可以利用这个阶段统一各自的状态。3PC主要包含了三个阶段，准备阶段、预提交阶段和提交阶段，对应的英文就是CanCommit、PreCommit、DoCommit。



  预提交阶段主要起了一个统一状态的作用，参与者引入的超时机制，如果等待提交命令超时，那么参与者就会提交事务；如果等待预提交的指令超时，那就该干啥就干啥。3PC解决的知识2PC提交阶段协调者和某些参与者挂了之后新选举的协调者不知道当前应该提交还是回滚的问题。

  总结：

  3PC 通过预提交阶段可以减少故障恢复时候的复杂性，但是不能保证数据一致，除非挂了的那个参与者恢复。

  2PC和3PC都不能保证数据100%一致，因此一般都需要定时扫描采用补偿机制。

- **XA规范**

  XA规范是X/Open 组织针对二阶段提交协议的实现做的规范。目前几乎所有的主流数据库都对XA规范提供了支持。

  XA规范的特点是：

  1. 对代码无侵入，开发比较快速
  2. 对资源进行了长时间的锁定，并发程度比较低

- **TCC**

  在企业中应用最广泛。

  2PC和3PC都是数据库层面的，而TCC是业务层面的分布式事务。TCC指的是`Try - Confirm - Cancel`。

  1、Try 指的是预留，即资源的预留和锁定，**注意是预留。**

  2、Confirm 指的是确认操作，这一步其实就是真正的执行了。提交曹锁，所有的Try都成功了，则执行Confirm操作，真正执行业务，使用Try预留的资源

3、Cancel 指的是撤销操作，可以理解为把预留阶段的动作撤销了。有一个Try失败了就走到Cancel操作。

TCC模型有个事务管理者的角色，用来记录TCC全局事务状态并提交或者回滚事务。对于每一个操作，都需要明确定义三个动作分别定义`Try - Confirm - Cancel`。所以TCC对业务的侵入较大和业务紧耦合，需要根据特定的场景和业务逻辑来设计相应的操作。

TCC的特点为：

1. 并发程度高，在业务层面锁定资源
2. 开发量大，每个业务都需要提供Try/Confirm/Cancel三个方法

- **SAGA**

SAGA是一种补偿协议，在SAGA模式下，分布式事务有多个参与者。在分布式事务的执行过程中，依次执行各参与者的正向操作，如果所有正向操作都执行成功，那么分布式事务提交。如果任何一个正向操作失败，则会执行前面各参与者的回滚操作，将事务状态回到初始状态。

SAGA特点为

1. 并发度高，不需要长期锁定资源
2. 开发量大，需要定义正向操作和补偿操作
3. 不能保证隔离型

- ## 本地消息表

本地消息表其实就是利用了**各系统本地事务来实现分布式事务。**

本地消息表顾名思义就是会有一张存放本地消息的表，一般都是放在数据库中，然后在执行业务的时候 **将业务的执行和将消息放入消息表中的操作放在同一个事务中**，这样就能保证消息放入本地表中业务肯定是执行成功的。

然后再去调用下一个操作，如果下一个操作调用成功了好说，消息表的消息状态可以直接改成已成功。

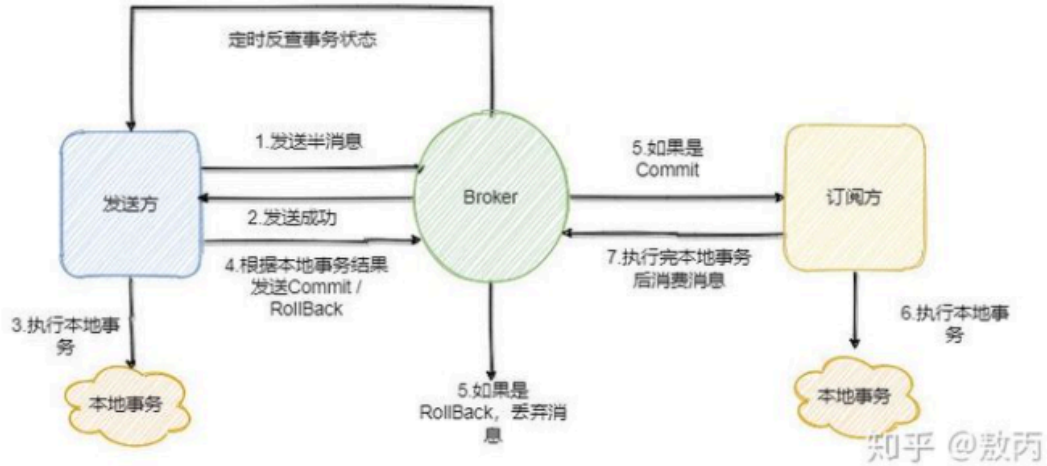
如果调用失败也没事，会有 **后台任务定时去读取本地消息表**，筛选出还未成功的消息再调用对应的服务，服务更新成功了再变更消息的状态。

这时候有可能消息对应的操作不成功，因此也需要重试，重试就得保证对应服务的方法是幂等的，而且一般重试会有最大次数，超过最大次数可以记录下报警让人工处理。

可以看到本地消息表其实实现的是**最终一致性**，容忍了数据暂时不一致的情况。

- ## RocketMQ消息事务

RocketMQ支持事务消息，实质是把本地消息表放到RocketMQ上，解决生产端消息发送和本地事务执行的原子性问题。



- **最大努力通知**

  我觉得本地消息表也可以算最大努力，事务消息也可以算最大努力。

  就本地消息表来说会有后台任务定时去查看未完成的消息，然后去调用对应的服务，当一个消息多次调用都失败的时候可以记录下然后引入人工，或者直接舍弃。这其实算是最大努力了。

  事务消息也是一样，当半消息被commit了之后确实就是普通消息了，如果订阅者一直不消费或者消费不了则会一直重试，到最后进入死信队列。其实这也算最大努力。

  所以**最大努力通知其实只是表明了一种柔性事务的思想**：我已经尽力我最大的努力想达成事务的最终一致了。

  适用于对时间不敏感的业务，例如短信通知、订单关闭库存解锁、微信支付

- **Seata AT模式**

  这是阿里开源的事务框架Seata中主推的事务模式。Seata AT是一种无侵入的事务解决方案。事务的一阶段和二阶段均由框架自动生成。**用户SQL作为分布式事务的一阶段，而二阶段由框架自动生成提交/回滚操作。**和XA模式很类似

  Seata AT模式特点：

  1. 对代码无侵入，开发速度较快
  2. 需要用全局锁来保证隔离性，并发程度较低

1、Seata使用

1、下载seata-server
2、创建数据库seata，可自定义
3、编辑file.conf配置文件，配置一些数据库的地址信息
4、编辑register.conf配置文件，配置要使用的注册中心的信息
5、启动seata-server
6、使用AT模式，需要在每个业务库添加数据表undo_log
7、项目中加依赖、开启注解 `@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)` 排除掉SpringBoot默认自动注入的 `DataSourceAutoConfiguration` Bean, 因为 `SEATA` 是基于数据源拦截来实现的分布式事务，因此我们还需要自己配置数据源信息（代理数据源）
8、配置yaml配置文件，配置信息必须注意和file.conf和register.conf一致
9、使用全局事务注解 `@GlobalTransactional(name = "create-order", rollbackFor = Exception.class)`

5.接口的幂等性

1、token机制，调用接口请求的时候携带token，查询redis中是否有token，存在表示第一次请求，然后删除token继续作业

2、各种锁机制。悲观锁就for upadate，乐观锁就加version版本号

3、业务层加分布式锁

4、各种唯一约束：数据库唯一约束、redis的set防重，防重表

5、全局请求唯一id

6.高可用分布式

1、CAP

是分布式系统的一个基本定理。

CAP 也就是 **Consistency（一致性）**、**Availability（可用性）**、**Partition Tolerance（分区容错性）**这三个单词首字母组合。

- **一致性（Consistence）**：所有节点访问同一份最新的数据副本
- **可用性（Availability）**：非故障的节点在合理的时间内返回合理的响应（不是错误或者超时的响应）。
- **分区容错性（Partition tolerance）**：分布式系统出现网络分区的时候，仍然能够对外提供服务。

> CAP 理论中分区容错性 P 是一定要满足的，在此基础上，只能满足可用性 A 或者一致性 C。

1.CAP实际应用案例

1. **ZooKeeper 保证的是 CP。** 任何时刻对 ZooKeeper 的读请求都能得到一致性的结果，但是，ZooKeeper 不保证每次请求的可用性比如在 Leader 选举过程中或者半数以上的机器不可用的时候服务就是不可用的。
2. **Eureka 保证的则是 AP。** Eureka 在设计的时候就是优先保证 A （可用性）。在 Eureka 中不存在什么 Leader 节点，每个节点都是一样的、平等的。因此 Eureka 不会像 ZooKeeper 那样出现选举过程中或者半数以上的机器不可用的时候服务就是不可用的情况。 Eureka 保证即使大部分节点挂掉也不会影响正常提供服务，只要有一个节点是可用的就行了。只不过这个节点上的数据可能并不是最新的。
3. **Nacos 不仅支持 CP 也支持 AP。**

2、BASE

BASE 是 **Basically Available（基本可用）**、**Soft-state（软状态）** 和 **Eventually Consistent（最终一致性）** 三个短语的缩写。BASE 理论是对 CAP 中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于 CAP 定理逐步演化而来的，它大大降低了我们对系统的要求。

1. 基本可用

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。但是，这绝不等价于系统不可用。

什么叫允许损失部分可用性呢?

- **响应时间上的损失**: 正常情况下，处理用户请求需要 0.5s 返回结果，但是由于系统出现故障，处理用户请求的时间变为 3 s。
- **系统功能上的损失**：正常情况下，用户可以使用系统的全部功能，但是由于系统访问量突然剧增，系统的部分非核心功能无法使用。

2. 软状态

软状态指允许系统中的数据存在中间状态（**CAP 理论中的数据不一致**），并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

3. 最终一致性

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

> 分布式一致性的 3 种级别：
>
> 1. **强一致性**： 系统写入了什么，读出来的就是什么。
> 2. **弱一致性**： 不一定可以读取到最新写入的值，也不保证多少时间之后读取到的数据是最新的，只是会尽量保证某个时刻达到数据一致的状态。
> 3. **最终一致性**： 弱一致性的升级版，系统会保证在一定时间内达到数据一致的状态。
>
> **业界比较推崇是最终一致性级别，但是某些对数据一致要求十分严格的场景比如银行转账还是要保证强一致性。**

总结

ACID 是数据库事务完整性的理论，CAP 是分布式系统设计理论，BASE 是 CAP 理论中 AP 方案的延伸。

3、Paxos算法

Paxos 算法诞生于 1900 年，这是一种解决分布式系统一致性的经典算法 。但是，由于 Paxos 算法非常难以理解和实现，不断有人尝试简化这一算法。到了2013 年才诞生了一个比 Paxos 算法更易理解和实现的分布式一致性算法—**Raft 算法。**还有ZAB协议等；

核心思想：slave副本和leader选举者达成数据一致，是分布式选举算法

4、Raft算法

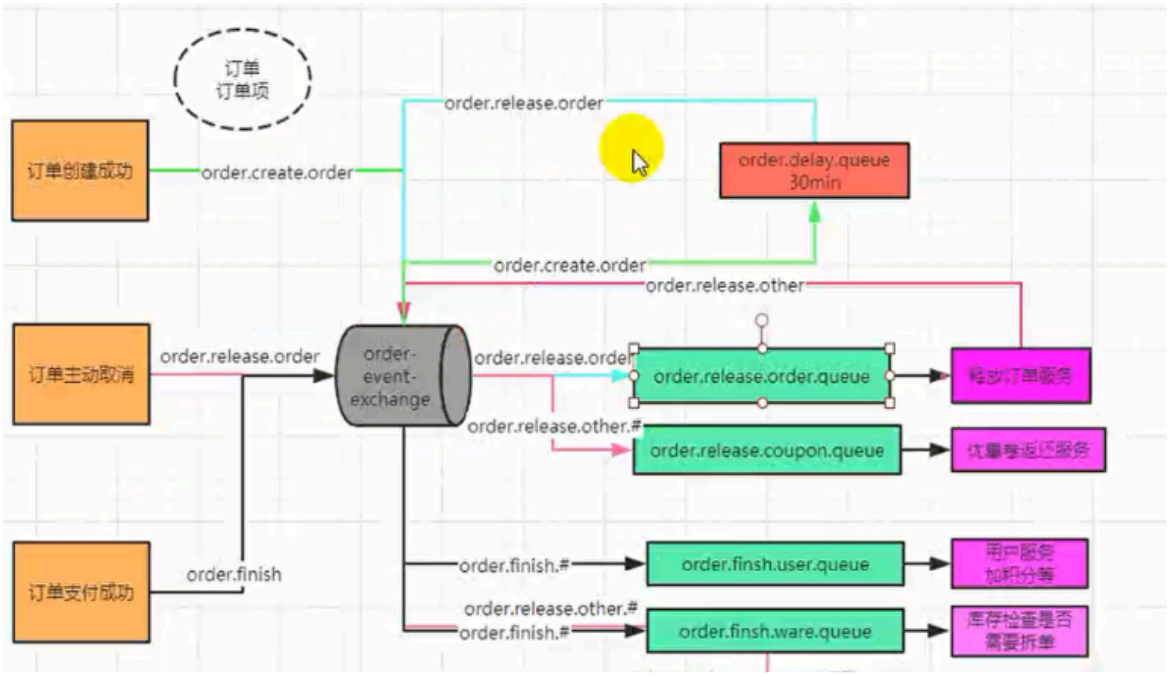
是一种共识算法，角色分为：领导，追随者，候选人

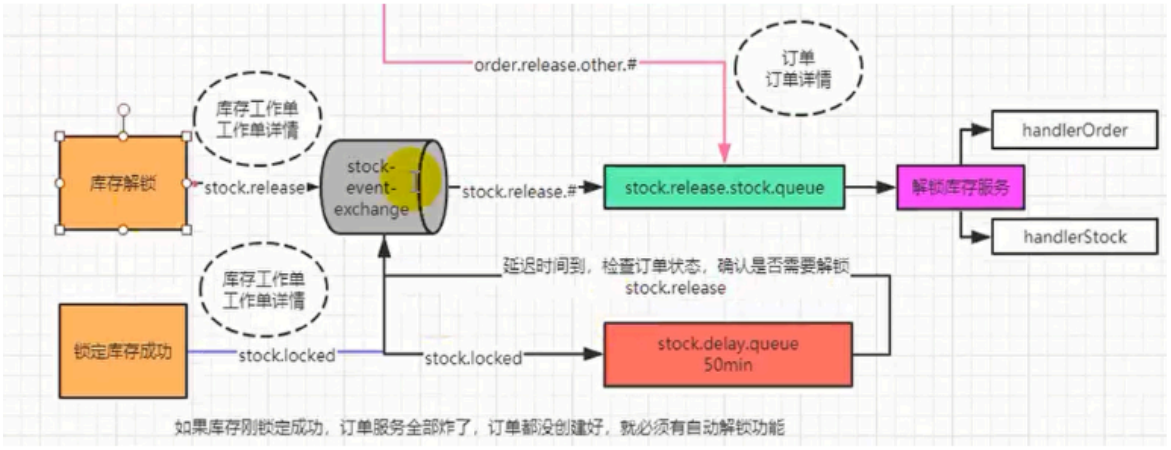
主要思想：领导选举+日志复制+心跳机制

自旋时间：决定候选人要不要当领导

心跳时间：要把日志发送出去

4、订单、库存流程图





7.消息的可靠性

- 防止消息丢失

  1、做好消息确认机制（生产者，消费者手动ack）+持久化（交换机、队列、消息）
  2、每一个发送的消息都应该在数据库做好记录，并定期扫描数据库重新发送一遍

- 防止发送重复的消息

  1、将业务设计为幂等的
  2、使用防重表

- 防止消息积压

  1、上线更多的消费者
  2、上限专门的消息队列服务，将消息先批量取出来，记录到数据库然后离线慢慢处理

8.秒杀

特点：具有瞬间高并发的特点，针对这一个特点，必须要做限流+异步+缓存（页面静态化）+独立部署

限流方式：

1.前端限流，一些高并发的网站直接在前端页面开始限流，例如：小米的验证码设计
2.nginx限流+nginx做高并发负载均衡：错误请求就请求到错误的静态页面：令牌算法、漏斗算法
3.网关限流、限流的过滤器

4.代码中使用分布式信号量，解决超卖

5.RabbitMQ限流，能者多劳，保证发挥所有服务器的性能，队列下单

秒杀控制：

1、服务单一职责+独立部署

2、秒杀链接加密

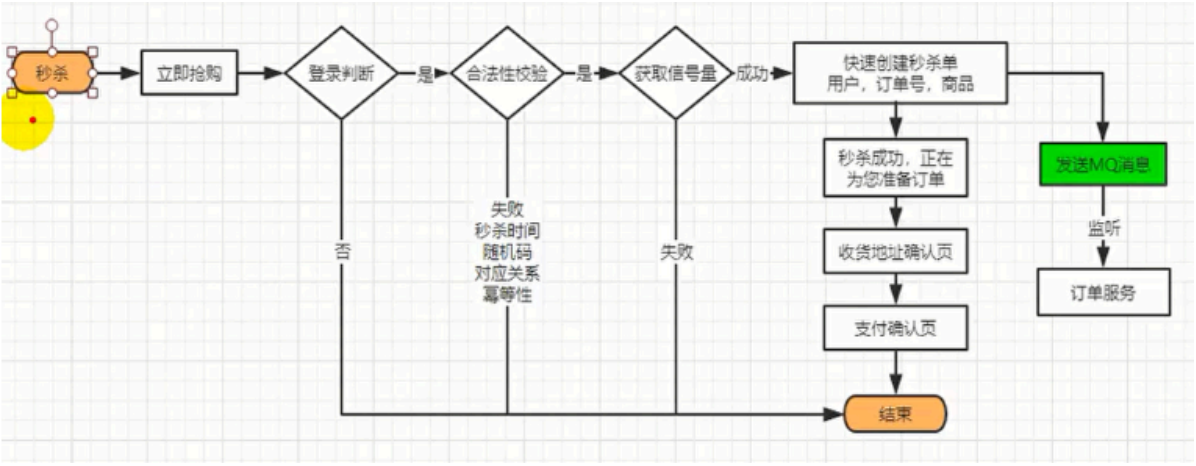
3、库存预热，Redis集群，快速扣减。使用信号量来控制秒杀的请求，防止超卖

4、动静分离。也可以使用CDN网络

5、恶意请求的拦截。识别非法请求并进行拦截，网关层

6、流量削峰。将一秒内的大并发请求分散到多个节点；或者加入购物车

7、限流、熔断、降级。前端限流：阻止连续点击；后端限流：去除不合理请求



9.Sentinel

> 吞吐量（TPS）、每秒查询数QPS、并发数、响应时间（RT）

主要步骤：1、定义资源 2、定义规则 3、检验规则是否生效

好处：丰富的应用场景，完美的实时监控，广泛的开源生态，完美的SPI扩展点

两大部分：控制台、核心库

整合使用：

```
* 1、整合Sentinel
*  1)、导入依赖spring-cloud-starter-alibaba-sentinel
*  2)、下载sentinel的控制台
*  3)、配置sentinel的控制台地址信息
*  4)、在控制台调整参数【默认所有的流控设置保存在内存中，重启生效】
*
* 2、每一个微服务都导入actuator模块，可以统计出每个应用的健康状态信息(统计审计信息)
* 3、自定义sentinel返回结果
```

`@SentinelResource`：定义资源，并提供可选的异常处理和fallback配置项

`@SentinelRestTemplate`：对服务的操纵进行保护操作

10.OAuth2.0

1、四种授权模式

1、授权码模式，通过code获取token授权
2、隐式授权/简化模式，直接获取token令牌
3、密码模式，一般不支持刷新令牌
4、客户端凭证模式

11.Sleuth+Zipkin链路追踪

```
1、引入sleuth+zipkin的依赖
2、下载安装zipkin
3、配置yml中zipkin的中心地址
4、远程调用
```

12.工作项目

mybatis枚举转换器，多数据源，策略模式（财务表Excel导出），模板方法模式（Excel导出模板），SQL，数据清洗，Redis自定义序列化

13.项目中最大困难

没有一个用户量足够大，业务足够复杂的项目经验去锻炼提升自己，在没有接口文档的情况下如何去滤清前端后端的请求，业务代码，sql和表结构之间的关系。

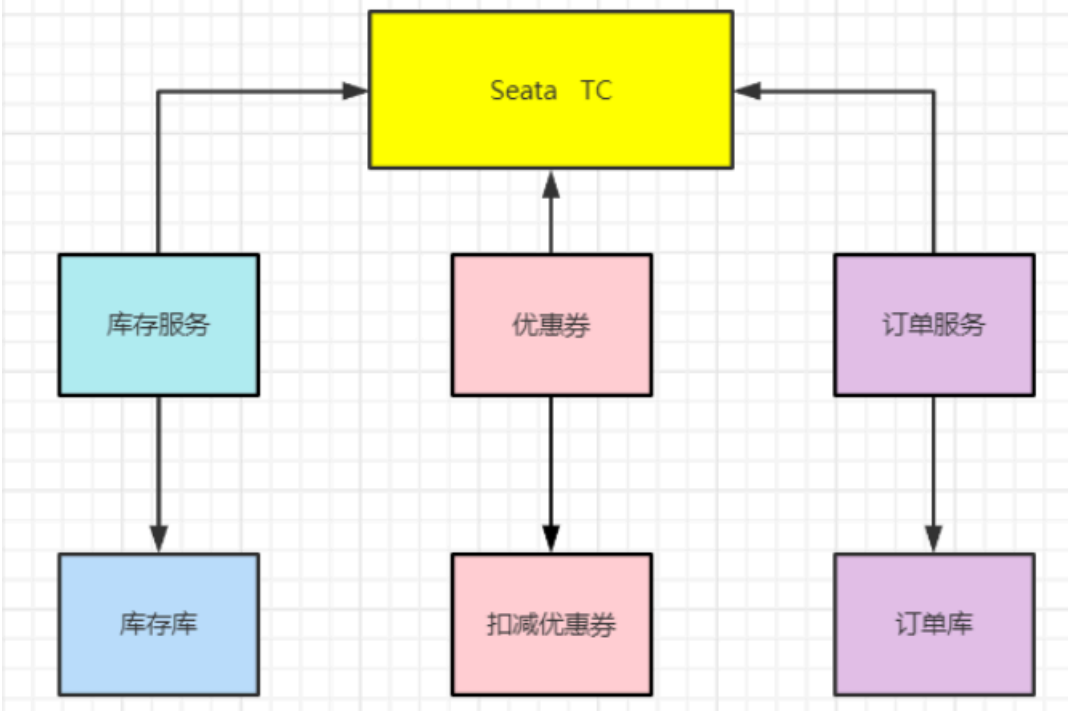
14.项目中如何解决分布式事务的?

Seata AT模式和Seata TCC模式是在生产中最常用。

- 强一致性模型，Seata AT **强一致方案** 模式用于强一致主要用于核心模块，例如交易/订单等。
- 弱一致性模型。Seata TCC **弱一致方案**一般用于边缘模块例如库存，通过TC的协调，保证最终一致性，也可以业务解耦。

1.强一致性场景

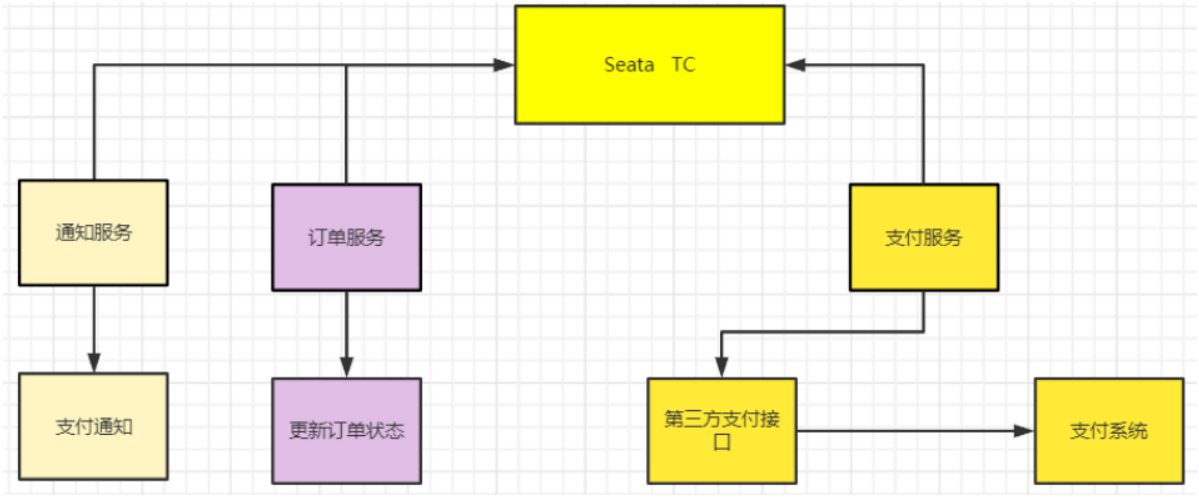
> 需要严格要求数据不能出错的场景，如电商交易中的库存和订单，优惠券等。阿里开源了分布式事务框架seata经历过阿里生产环境大量考验的框架。 seata支持Dubbo，Spring Cloud



增加订单-扣减库存-预扣优惠券

2.弱一致性场景

> 对于数据一致性要求没有那些特别严格、或者由不同系统执行子事务的场景，可以回答使用Seata TCC 保障弱一致性方案
>
> 准备好例子：一个不是严格对数据一致性要求、或者由不同系统执行子事务的场景，如电商订单支付服务，更新订单状态，发送成功支付成功消息，只需要保障弱一致性即可。



Seata TCC 模式，保障弱一致性，支持跨多个服务和系统修改数据，在上面的场景中，使用Seata TCC 模式事务

- 订单服务：修改订单状态
- 通知服务：发送支付状态

3.最终一致性场景

基于可靠消息的最终一致性，各个子事务可以较长时间内异步，但数据绝对不能丢的场景。可以使用**异步确保型事务**事。

可以使用基于MQ的异步确保型事务，比如电商平台的通知支付结果

- 积分服务：增加积分

- 会计服务：生成会计记录

15.Canal

1.Canal原理

1、模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议；mysql master收到dump请求，开始推送binary log给slave(也就是canal)；解析binary log对象(原始为byte流)
2、存在重复消费问题：需要在消费端解决。
3、canal需要维护EventStore，可以存取在Memory, File, zk.
4、canal需要维护客户端的状态，同一时刻一个instance只能有一个消费端消费.
5、支持binlog format 类型:statement, row, mixed. 多次附加功能只能在row下使用，比如otter.
6、有ACK机制。

Canal 包含两个组成部分：服务端和客户端。服务端负责连接至不同的 MySQL 实例，并为每个实例维护一个事件消息队列；客户端则可以订阅这些队列中的数据变更事件，处理并存储到数据仓库中。Canal源码中是从上次更新Position的位置继续向下更新。

2.百万并发

1.LVS

- **原理**

  LVS通过工作于内核的ipvs模块来实现功能，基于ipvsadm工具来设置lvs模型、调度方式和指定后端主机。

- **工作模式**

  DR直接路由模式、NAT模式、TUN隧道模式、FULL NAT模式

- **特点**

  配置性比较低，不支持正则处理、动静分离。

lvs配合nginx使用：lvs在接入层承担最大规模的流量分发缓冲，Nginx主要为后端服务器反向代理中间件，减小后端压力，以及做一些业务切换、分流、前置缓存等。
lvs采用同步请求转发策略，nginx采用异步转发请求策略。所以所有的请求和响应流量都会经过nginx，而lvs只有请求流量经过，响应流量由后端返回。

主要就是避免单nginx的性能瓶颈和lvs对于后端错误无法感知的问题。

2.百万并发调优

由于一个socket连接一般占用8kb内存，所以百万连接至少需要差不多8GB内存.

```
# 系统最大文件打开数
fs.file-max = 20000000

# 单个用户进程最大文件打开数
fs.nr_open = 20000000

# 全连接队列长度,默认128，全队列指的是ESTABLISHED状态的队列，
net.core.somaxconn = 10240
# 半连接队列长度，当使用sysncookies无效，默认128，半队列指一个存放SYN_REVD状态的队列
net.ipv4.tcp_max_syn_backlog = 16384
net.ipv4.tcp_syncookies = 0
```

```
# 网卡数据包队列长度，即半连接上限
net.core.netdev_max_backlog = 41960

# time-wait 最大队列长度
net.ipv4.tcp_max_tw_buckets = 300000

# time-wait 是否重新用于新链接以及快速回收
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1

# tcp报文探测时间间隔，单位s
net.ipv4.tcp_keepalive_intvl = 30
# tcp连接多少秒后没有数据报文时启动探测报文
net.ipv4.tcp_keepalive_time = 900
# 探测次数
net.ipv4.tcp_keepalive_probes = 3

# 保持fin-wait-2 状态多少秒
net.ipv4.tcp_fin_timeout = 15

# 最大孤儿socket数量,一个孤儿socket占用64KB,当socket主动close掉,处于fin-wait1，last-ack
net.ipv4.tcp_max_orphans = 131072

# 每个套接字所允许得最大缓存区大小
net.core.optmem_max = 819200

# 默认tcp数据接受窗口大小
net.core.rmem_default = 262144
net.core.wmem_default = 262144
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216

# tcp栈内存使用第一个值内存下限，第二个值缓存区应用压力上限，第三个值内存上限，单位为page,通常为4kb
net.ipv4.tcp_mem = 786432 4194304 8388608
# 读，第一个值为socket缓存区分配最小字节，第二个，第三个分别被rmem_default，rmem_max覆盖
net.ipv4.tcp_rmem = 4096 4096 4206592
# 写，第一个值为socket缓存区分配最小字节，第二个，第三个分别被wmem_default，wmem_max覆盖
net.ipv4.tcp_wmem = 4096 4096 4206592
```

```
sysctl -p#生效配置
#监控
ss -s&&uptime&&free -m
```

二十六、操作系统

1、基础

1.系统调用

根据进程访问资源的特点，我们可以把进程在系统上的运行分为两个级别：

1. 用户态(user mode) : 用户态运行的进程或可以直接读取用户程序的数据。
2. 系统态(kernel mode):可以简单的理解系统态运行的进程或程序几乎可以访问计算机的任何资源，不受限制。

说了用户态和系统态之后，那么什么是系统调用呢？

我们运行的程序基本都是运行在用户态，如果我们调用操作系统提供的系统态级别的子功能咋办呢？那就需要系统调用了！

也就是说在我们运行的用户程序中，凡是与系统态级别的资源有关的操作（如文件管理、进程控制、内存管理等），都必须通过系统调用方式向操作系统提出服务请求，并由操作系统代为完成。

这些系统调用按功能大致可分为如下几类：

- 设备管理。完成设备的请求或释放，以及设备启动等功能。
- 文件管理。完成文件的读、写、创建及删除等功能。
- 进程控制。完成进程的创建、撤销、阻塞及唤醒等功能。
- 进程通信。完成进程之间的消息传递或信号传递等功能。
- 内存管理。完成内存的分配、回收以及获取作业占用内存区大小及地址等功能。

2.进程状态

- **创建状态(new)** ： 进程正在被创建，尚未到就绪状态。
- **就绪状态(ready)** ： 进程已处于准备运行状态，即进程获得了除了处理器之外的一切所需资源，一旦得到处理器资源(处理器分配的时间片)即可运行。
- **运行状态(running)** ： 进程正在处理器上上运行(单核 CPU 下任意时刻只有一个进程处于运行状态)。
- **阻塞状态(waiting)** ： 又称为等待状态，进程正在等待某一事件而暂停运行如等待某资源为可用或等待 IO 操作完成。即使处理器空闲，该进程也不能运行。
- **结束状态(terminated)** ： 进程正在从系统中消失。可能是进程正常结束或其他原因中断退出运行。

3.进程间通讯方式

1. **管道/匿名管道(Pipes)** ： 用于具有亲缘关系的父子进程间或者兄弟进程之间的通信。
2. **有名管道(Names Pipes)** : 匿名管道由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道。有名管道严格遵循**先进先出(first in first out)**。有名管道以磁盘文件的方式存在，可以实现本机任意两个进程通信。
3. **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个事件已经发生；

4. **消息队列(Message Queuing)**： 消息队列是消息的链表,具有特定的格式,存放在内存中并由消息队列标识符标识。管道和消息队列的通信数据都是先进先出的原则。与管道（无名管道：只存在于内存中的文件；命名管道：存在于实际的磁盘介质或者文件系统）不同的是消息队列存放在内核中，只有在内核重启(即，操作系统重启)或者显示地删除一个消息队列时，该消息队列才会被真正的删除。消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取.比 FIFO 更有优势。**消息队列克服了信号承载信息量少，管道只能承载无格式字 节流以及缓冲区大小受限等缺。**
5. **信号量(Semaphores)**： 信号量是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。这种通信方式主要用于解决与同步相关的问题并避免竞争条件。
6. **共享内存(Shared memory)**： 使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。可以说这是最有用的进程间通信方式。
7. **套接字(Sockets)** : 此方法主要用于在客户端和服务器之间通过网络进行通信。套接字是支持 TCP/IP 的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。

4.进程间（线程间）同步方式

1. **互斥量(Mutex)**：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。比如 Java 中的 synchronized 关键词和各种 Lock 都是这种机制。
2. **信号量(Semphares)**： 它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量
3. **事件(Event)** :Wait/Notify：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操
4. 临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问

5.进程的调度算法

- **先到先服务(FCFS)调度算法** : 从就绪队列中选择一个最先进入该队列的进程为之分配资源，使它立即执行并一直执行到完成或发生某事件而被阻塞放弃占用 CPU 时再重新调度。
- **短作业优先(SJF)的调度算法** : 从就绪队列中选出一个估计运行时间最短的进程为之分配资源，使它立即执行并一直执行到完成或发生某事件而被阻塞放弃占用 CPU 时再重新调度。
- **时间片轮转调度算法** : 时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法，又称 RR(Round robin)调度。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。
- **多级反馈队列调度算法**： 前面介绍的几种进程调度的算法都有一定的局限性。如**短进程优先的调度算法，仅照顾了短进程而忽略了长进程** 。多级反馈队列调度算法既能使高优先级的作业得到响应又能使短作业（进程）迅速完成。，因而它是目前**被公认的一种较好的进程调度算法**，UNIX 操作系统采取的便是这种调度算法。
- **优先级调度**： 为每个流程分配优先级，首先执行具有最高优先级的进程，依此类推。具有相同优先级的进程以 FCFS 方式执行。可以根据内存要求，时间要求或任何其他资源要求来确定优先级。

2、内存管理基础

1.内存管理作用

操作系统的内存管理主要负责内存的分配与回收（malloc 函数：申请内存，free 函数：释放内存），另外地址转换也就是将逻辑地址转换成相应的物理地址等功能也是操作系统内存管理做的事情。

2.集中内存管理机制

简单分为**连续分配管理方式**和**非连续分配管理方式**这两种。连续分配管理方式是指为一个用户程序分配一个连续的内存空间，常见的如 **块式管理** 。同样地，非连续分配管理方式允许一个程序使用的内存分布在离散或者说不相邻的内存中，常见的如**页式管理** 和 **段式管理**。

1. **块式管理** ： 远古时代的计算机操系统的内存管理方式。将内存分为几个固定大小的块，每个块中只包含一个进程。如果程序运行需要内存的话，操作系统就分配给它一块，如果程序运行只需要很小的空间的话，分配的这块内存很大一部分几乎被浪费了。这些在每个块中未被利用的空间，我们称之为碎片。
2. **页式管理** ： 把主存分为大小相等且固定的一页一页的形式，页较小，相对相比于块式管理的划分力度更大，提高了内存利用率，减少了碎片。页式管理通过页表对应逻辑地址和物理地址。
3. **段式管理** ： 页式管理虽然提高了内存利用率，但是页式管理其中的页实际并无任何实际意义。 段式管理把主存分为一段段的，每一段的空间又要比一页的空间小很多 。但是，最重要的是段是有实际意义的，每个段定义了一组逻辑信息，例如,有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等。 段式管理通过段表对应逻辑地址和物理地址。
4. **段页式管理机制** ： 段页式管理机制结合了段式管理和页式管理的优点。简单来说段页式管理机制就是把主存先分成若干段，每个段又分成若干页，也就是说 **段页式管理机制** 中段与段之间以及段的内部的都是离散的。

3.快表和多级页表

在分页内存管理中，很重要的两点是：

1. 虚拟地址到物理地址的转换要快。
2. 解决虚拟地址空间大，页表也会很大的问题。

快表

为了解决虚拟地址到物理地址的转换速度，操作系统在 **页表方案** 基础之上引入了 **快表** 来加速虚拟地址到物理地址的转换。我们可以把快表理解为一种特殊的高速缓冲存储器（Cache），其中的内容是页表的一部分或者全部内容。作为页表的 Cache，它的作用与页表相似，但是提高了访问速率。由于采用页表做地址转换，读写内存数据时 CPU 要访问两次主存。有了快表，有时只要访问一次高速缓冲存储器，一次主存，这样可加速查找并提高指令执行速度。

使用快表之后的地址转换流程是这样的：

1. 根据虚拟地址中的页号查快表；
2. 如果该页在快表中，直接从快表中读取相应的物理地址；
3. 如果该页不在快表中，就访问内存中的页表，再从页表中得到物理地址，同时将页表中的该映射表项添加到快表中；
4. 当快表填满后，又要登记新页时，就按照一定的淘汰策略淘汰掉快表中的一个页。

看完了之后你会发现快表和我们平时经常在我们开发的系统使用的缓存（比如 Redis）很像，的确是这样的，操作系统中的很多思想、很多经典的算法，你都可以在我们日常开发使用的各种工具或者框架中找到它们的影子。

多级页表

引入多级页表的主要目的是为了避免把全部页表一直放在内存中占用过多空间，特别是那些根本就不需要的页表就不需要保留在内存中。多级页表属于时间换空间的典型场景，具体可以查看下面这篇文章

- 多级页表如何节约内存：https://www.polarxiong.com/archives/多级页表如何节约内存.html

总结

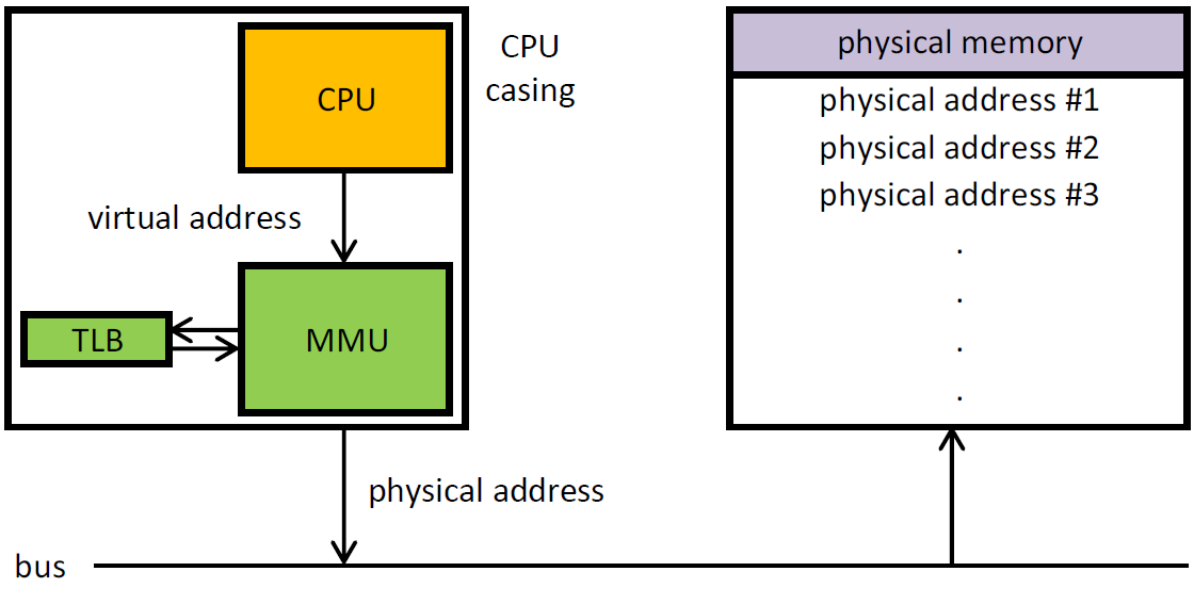
为了提高内存的空间性能，提出了多级页表的概念；但是提到空间性能是以浪费时间性能为基础的，因此为了补充损失的时间性能，提出了快表（即 TLB）的概念。 不论是快表还是多级页表实际上都利用到了程序的局部性原理，局部性原理在后面的虚拟内存这部分会介绍到

4.分页机制和分段机制

1. 共同点
    - 分页机制和分段机制都是为了提高内存利用率，较少内存碎片。
    - 页和段都是离散存储的，所以两者都是离散分配内存的方式。但是，每个页和段中的内存是连续的。
2. 区别
    - 页的大小是固定的，由操作系统决定；而段的大小不固定，取决于我们当前运行的程序。
    - 分页仅仅是为了满足操作系统内存管理的需求，而段是逻辑信息的单位，在程序中可以体现为代码段，数据段，能够更好满足用户的需要。

5.CPU寻址

现代处理器使用的是一种称为 **虚拟寻址(Virtual Addressing)** 的寻址方式。**使用虚拟寻址，CPU 需要将虚拟地址翻译成物理地址，这样才能访问到真实的物理内存。** 实际上完成虚拟地址转换为物理地址转换的硬件是 CPU 中含有一个被称为 **内存管理单元（Memory Management Unit, MMU）** 的硬件。如下图所示：



CPU: Central Processing Unit
MMU: Memory Management Unit
TLB: Translation lookaside buffer

为什么要有虚拟地址空间呢?

先从没有虚拟地址空间的时候说起吧！没有虚拟地址空间的时候，**程序都是直接访问和操作的都是物理内存** 。但是这样有什么问题呢?

1. 用户程序可以访问任意内存，寻址内存的每个字节，这样就很容易（有意或者无意）破坏操作系统，造成操作系统崩溃。
2. 想要同时运行多个程序特别困难，比如你想同时运行一个微信和一个 QQ 音乐都不行。为什么呢? 举个简单的例子：微信在运行的时候给内存地址 1xxx 赋值后，QQ 音乐也同样给内存地址 1xxx 赋值，那么 QQ 音乐对内存的赋值就会覆盖微信之前所赋的值，这就造成了微信这个程序就会崩溃。

总结来说：如果直接把物理地址暴露出来的话会带来严重问题，比如可能对操作系统造成伤害以及给同时运行多个程序造成困难。

通过虚拟地址访问内存有以下优势：

- 程序可以使用一系列相邻的虚拟地址来访问物理内存中不相邻的大内存缓冲区。
- 程序可以使用一系列虚拟地址来访问大于可用物理内存的内存缓冲区。当物理内存的供应量变小时，内存管理器会将物理内存页（通常大小为 4 KB）保存到磁盘文件。数据或代码页会根据需要在物理内存与磁盘之间移动。
- 不同进程使用的虚拟地址彼此隔离。一个进程中的代码无法更改正在由另一进程或操作系统使用的物理内存。

6.虚拟内存

这个在我们平时使用电脑特别是 Windows 系统的时候太常见了。很多时候我们使用点开了很多占内存的软件，这些软件占用的内存可能已经远远超出了我们电脑本身具有的物理内存。**为什么可以这样呢?** 正是因为 **虚拟内存** 的存在，通过 **虚拟内存** 可以让程序可以拥有超过系统物理内存大小的可用内存空间。另外，**虚拟内存为每个进程提供了一个一致的、私有的地址空间，它让每个进程产生了一种自己在独享主存的错觉（每个进程拥有一片连续完整的内存空间）** 。这样会更加有效地管理内存并减少出错。

虚拟内存是计算机系统内存管理的一种技术，我们可以手动设置自己电脑的虚拟内存。不要单纯认为虚拟内存只是“使用硬盘空间来扩展内存”的技术。**虚拟内存的重要意义是它定义了一个连续的虚拟地址空间**，并且 **把内存扩展到硬盘空间。**推荐阅读：《虚拟内存的那点事儿》

7.页面置换算法

地址映射过程中，若在页面中发现所要访问的页面不在内存中，则发生缺页中断 。

> **缺页中断** 就是要访问的**页**不在主存，需要操作系统将其调入主存后再进行访问。 在这个时候，被内存映射的文件实际上成了一个分页交换文件。

当发生缺页中断时，如果当前内存中并没有空闲的页面，操作系统就必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。用来选择淘汰哪一页的规则叫做页面置换算法，我们可以把页面置换算法看成是淘汰页面的规则。

- **OPT 页面置换算法（最佳页面置换算法）** ： 最佳(Optimal, OPT)置换算法所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面,这样可以保证获得最低的缺页率。但由于人们目前无法预知进程在内存下的若干页面中哪个是未来最长时间内不再被访问的，因而该算法无法实现。一般作为衡量其他置换算法的方法。
- **FIFO（First In First Out） 页面置换算法（先进先出页面置换算法）**：总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面进行淘汰。
- **LRU （Least Currently Used）页面置换算法（最近最久未使用页面置换算法）** ： LRU算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 T，当须淘汰一个页面时，选择现有页面中其 T 值最大的，即最近最久未使用的页面予以淘汰。
- **LFU （Least Frequently Used）页面置换算法（最少使用页面置换算法）**：该置换算法选择在之前时期使用最少的页面作为淘汰页。

3、Shell

4、面试题

1.协程和线程区别

协程是异步机制，是非抢占的，同一时间只有一个协程拥有运行权；

协程不被操作系统内核管理，完全由程序控制；

协程能保留上一次调用时的状态；

2.进程和线程切换流程

进程切换分两步：

1、切换**页表**以使用新的地址空间，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。

2、切换内核栈和硬件上下文。

对于linux来说，线程和进程的最大区别就在于地址空间，对于线程切换，第1步是不需要做的，第2步是进程和线程切换都要做的。

3.为什么虚拟地址空间切换比较耗时?

进程都有自己的虚拟地址空间，把虚拟地址转换为物理地址需要查找页表，页表查找是一个很慢的过程，因此通常使用Cache来缓存常用的地址映射，这样可以加速页表查找，这个Cache就是TLB（translation Lookaside Buffer，TLB本质上就是一个Cache，是用来加速页表查找的）。

由于每个进程都有自己的虚拟地址空间，那么显然每个进程都有自己的页表，那么**当进程切换后页表也要进行切换，页表切换后TLB就失效了**，Cache失效导致命中率降低，那么虚拟地址转换为物理地址就会变慢，表现出来的就是程序运行会变慢，而线程切换则不会导致TLB失效，因为线程无需切换地址空间，因此我们通常说线程切换要比较进程切换块，原因就在这里。

4.线程分类

内核级线程：这类线程依赖于内核，又称为内核支持的线程或轻量级进程。比如英特尔i5-8250U是4核8线程，这里的线程就是内核级线程。

用户级线程：它仅存在于用户级中，这种线程是**不依赖于操作系统核心**的。应用进程利用**线程库来完成其创建和管理**，速度比较快，**操作系统内核无法感知用户级线程的存在。**

5.什么是临界区，如何解决冲突?

每个进程中访问临界资源的那段程序称为临界区，**一次仅允许一个进程使用的资源称为临界资源。**

解决冲突的办法：

- 如果有若干进程要求进入空闲的临界区，**一次仅允许一个进程进入**，如已有进程进入自己的临界区，则其它所有试图进入临界区的进程必须等待；
- 进入临界区的进程要在**有限时间内退出。**
- 如果进程不能进入自己的临界区，则应**让出CPU**，避免进程出现"忙等"现象。

二十七、Netty

1、基础

并发高，传输快，封装好

1.概念

1. Netty 是一个 **基于 NIO** 的 client-server(客户端服务器)框架，使用它可以快速简单地开发网络应用程序。
2. 它极大地简化并优化了 TCP 和 UDP 套接字服务器等网络编程,并且性能以及安全性等很多方面甚至都要更好。
3. **支持多种协议** 如 FTP，SMTP，HTTP 以及各种二进制和基于文本的传统协议。
4. Netty所有操作都是异步的，采用Future和ChannelFuture注册监听结果事件。

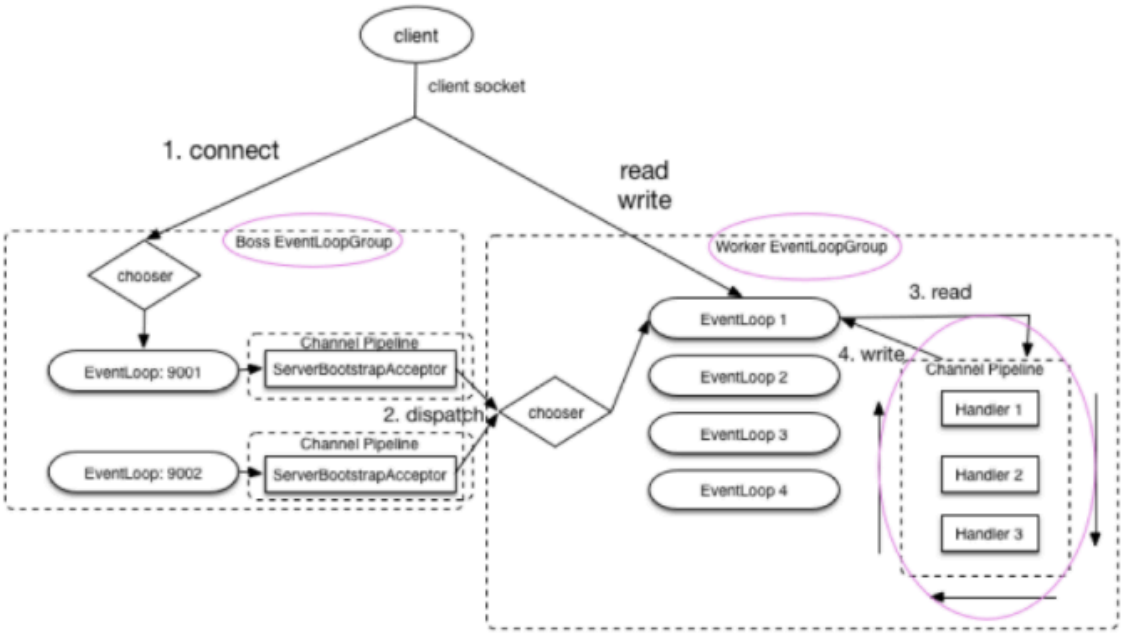
2.优势

1、可以解决断连重试、包丢失、黏包等问题。
2、统一的APU，支持多种传输类型，阻塞和非阻塞的
3、简单而强大的线程模型
4、支持更高的并发量和吞吐量，低资源消耗和少内存占用
5、成熟稳定，很多开源项目都使用的Netty：Dubbo，RocketMQ，ES

3.使用场景

RPC网络通信框架、自定义HTTP服务器、即时通讯、在线消息通知

4.核心组件



- **Bytebuf（字节容器）**：网络容器都是通过字节流传输的，内部是一个字节数组，是对ByteBuffer做的一层封装
- **Bootstrap和ServerBootstrap（启动引导类）**：是客户端启动的一个引导类。
    1. `Bootstrap` 通常使用 `connet()` 方法连接到远程的主机和端口，作为一个 Netty TCP 协议通信中的客户端。另外，`Bootstrap` 也可以通过 `bind()` 方法绑定本地的一个端口，作为 UDP 协议通信中的一端。
    2. `ServerBootstrap`通常使用 `bind()` 方法绑定本地的端口上，然后等待客户端的连接。
    3. `Bootstrap` 只需要配置一个线程组— `EventLoopGroup`,而 `ServerBootstrap`需要配置两个线程组— `EventLoopGroup` ，一个用于接收连接，一个用于具体的 IO 处理。
- **Channel（网络操作抽象类）**：`Channel` 接口是 Netty 对网络操作抽象类。通过 `Channel` 我们可以进行 I/O 操作。如 bind()、connect()、read()、write() 等。
- **ChannelFuture**：Netty是异步的，所以不能立即得到操作是否成功。可以通过 **ChannelFuture** 接口的 **addListener()** 方法注册一个 **ChannelFutureListener**，当操作执行成功或者失败时，监听就会自动触发返回结果。
- **EventLoop和EventLoopGroup（事件循环）**：负责监听网络时间并调用事件处理器进行相关IO操作的处理，一个用于接收链接监听网络调用事件并处理，一个用于接收数据。其中 Boss EventloopGroup 用于接收连接，Worker EventloopGroup 用于具体的处理
- **ChannelHandler和ChannelPipeline**：**ChannelHandler** 是消息的具体处理器。他负责处理读写操作、客户端连接等事情。

  ChannelPipeline 为 **ChannelHandler** 的链，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API 。当 **Channel** 被创建时，它会被自动地分配到它专属的**ChannelPipeline**。

- **ChannelHandlerContext**：保存Channel相关的所有上下文信息，同时关联一个ChannelHandler对象。即 ChannelHandlerContext 中包含一个具体的事件处理器 ChannelHandler，同时 ChannelHandlerContext 中也绑定了对应的 pipeline 和 Channel 的信息，方便对 ChannelHandler 进行调用的常用方法。

5.NioEventLoopGroup 默认的构造函数会起多少线程?

```
// 从1，系统属性，CPU核心数*2 这三个值中取出一个最大的
//可以得出 DEFAULT_EVENT_LOOP_THREADS 的值为CPU核心数*2
private static final int DEFAULT_EVENT_LOOP_THREADS = Math.max(1,
SystemPropertyUtil.getInt("io.netty.eventLoopThreads",
NettyRuntime.availableProcessors() * 2));
```

每个NioEventLoopGroup对象内部都会分配一组NioEventLoop，其大小是 nThreads, 这样就构成了一个线程池， 一个NIOEventLoop 和一个线程相对应，这和我们上面说的 EventloopGroup 和 EventLoop 关系这部分内容相对应。

6.Netty线程模型?

单线程、多线程、主从多线程

7.Netty解决TCP粘包/半包

> **粘包**：发送方写入数据小于Socket缓冲区大小，接收方读取Socket数据慢
> **半包**：发送方写入数据大于Socket缓冲区大小，发送的数据大于协议的MTU（最大传输单元），因此必须拆包

- 使用Netty自带的编码器，封装成帧
  - **LineBasedFrameDecoder** : 发送端发送数据包的时候，每个数据包之间以换行符作为分隔，LineBasedFrameDecoder 的工作原理是它依次遍历 ByteBuf 中的可读字节，判断是否有换行符，然后进行相应的截取。
  - **DelimiterBasedFrameDecoder** : 可以自定义分隔符解码器，**LineBasedFrameDecoder** 实际上是一种特殊的 DelimiterBasedFrameDecoder 解码器。
  - **FixedLengthFrameDecoder**: 固定长度解码器，它能够按照指定的长度对消息进行相应的拆包。

    原理：在接收ByteBuf的时候会将数据解码为固定长度的各个包。每次读取完完整的消息后都将计数器重置，如果长度不够可以空格补。
  - **LengthFieldBasedFrameDecoder**：专门的length字段
- 自定义序列化编码器
  - 在 Java 中自带的有实现 Serializable 接口来实现序列化，但由于它性能、安全性等原因一般情况下是不会被使用到的。

    通常情况下，我们使用 Protostuff、Hessian2、json 序列方式比较多，另外还有一些序列化性能非常好的序列化方式也是很好的选择。
- 改成短连接。不过缺点很明显

8.Netty长连接、心跳

长连接：适合频繁请求资源
短连接：适合请求一次资源

心跳机制：Netty心跳机制保证了在长连接过程中出现了网络中断异常的时候判断对方掉线

> **心跳机制原理**：在 client 与 server 之间在一定时间内没有数据交互时, 即处于 idle 状态时, 客户端或服务器就会发送一个特殊的数据包给对方, 当接收方收到这个数据报文后, 也立即发送一个特殊的数据报文, 回应发送方, 此即一个 PING-PONG 交互。所以, 当某一端收到心跳消息后, 就知道了对方仍然在线, 这就确保 TCP 连接的有效性

TCP实际上自带的就有长连接的选项，本身也有心跳包机制，也就是`SO_KEEPALIVE`。但是，TCP 协议层面的长连接灵活性不够。所以，一般情况下我们都是在应用层协议上实现自定义信跳机制的，也就是在 Netty 层面通过编码实现。通过 Netty 实现心跳机制的话，核心类是 IdleStateHandler

9.零拷贝

在 OS 层面上的 Zero-copy 通常指避免在 用户态(User-space) 与 内核态(Kernel-space) 之间来回拷贝数据。而在 Netty 层面 ， 零拷贝主要体现在对于数据操作的优化。

Netty 中的零拷贝体现在以下几个方面：

1. 使用 Netty 提供的 CompositeByteBuf 类, 可以将多个ByteBuf 合并为一个逻辑上的 ByteBuf, 避免了各个 ByteBuf 之间的拷贝。
2. ByteBuf 支持 slice 操作, 因此可以将 ByteBuf 分解为多个共享（堆外内存）同一个存储区域的 ByteBuf, 避免了内存的拷贝。
3. 通过 FileRegion 包装的FileChannel.tranferTo 实现文件传输, 可以直接将文件缓冲区的数据发送到目标 Channel, 避免了传统通过循环 write 方式导致的内存拷贝问题.

10.序列化方式

Java Serializer：无法跨语言，性能差
XML：格式负载，传输占带宽
JSON：轻量级
FastJson：目前 java 语言中 最快的 json 库
Thrift：不仅是序列化协议，还是一个RPC框架
Hessian：采用二进制协议的轻量级工具

11.Netty高性能表现在哪些方面?

1、心跳机制，定时清除会话
2、串行无锁化设计
3、ByteBuf，内存池缓冲，读写空闲时间超时
4、TLS和第三方CA认证
5、高效并发：volatile大量正确使用；CAS和原子类的广泛使用；
6、TCP参数设置：SO_RCVBUF 和 SO_SNDBUF:通常建议值为 128K 或者 256K；
SO_TCPNODELAY:NAGLE 算法通过将缓冲区内的小封包自动相连，组成较大的包，阻止大量小包的发送阻塞网络，从而提高网络应用效率；但是时延敏感的情况需要关闭该参数。

12.Netty优化?

1、使用直接放入channel所对应的EventLoop的执行队列调度中，避免使用channel.writeAndFlush造成线程的切换；writeAndFlush如果在Netty线程池内执行，则是直接write；否则，将作为一个task插入到Netty线程池执行。
2、减少ChannelPipeline的调用链长度。
3、减少ChannelHandler的创建，@Sharable注解，只有一个实例，减少GC
4、配置参数的设置：option参数优化：SO_TIMEOUT、SO_BACKLOG；响应高要求就禁用Nagle算法。
5、尽可能复用EventLoop，因为EventLoop和Channel是一对多的关系；不要阻塞EventLoop！
6、合理设置心跳检测，定时清除无用的会话。 心跳检测周期通常不要超过60s，心跳检测超时通常为心跳检测周期的2倍
7、操作系统参数调优，设置最大文件句柄数、单进程最大文件句柄数，tcp参数优化
8、使用池化技术，接收和发送缓冲区调优，进行测试选出一个最优值
9、避免在IO线程上执行时间不可控的操作，避免以外阻塞
10、支持设置流控，设置高水位值，达到高水位时channel不工作

13.ButeBuf池化 非池化

Netty的ByteBuf分为池化的和非池化的，池化的优点包含如下两点：

1. **对于DirectByteBuffer的分配和释放是比较低效的，使用池化技术能快速分配内存。**
2. **池化技术使对象可以复用，从而降低gc频率**

关键技术点：

多线程竞争下的对象分配和释放（怎么减小多线程竞争）
每个线程都有自己的对象池，分配时从自己的对象池中获得一个对象。其他线程release对象时，把对象归还到原来自己的池子中去（分配线程的池子）。
大量使用了ThreadLocal，每个线程都有自己的stack和weakorderqueue，做到线程封闭，有力减小竞争。

对象的分配
2.1 先从stack中分配，如果stack有，则直接stack.pop获得对象
2.2 如果stack中没有，从WeakOrderQueue中一次移取多个对象到stack中(每次会尽可能scavenge整个head Link到stack中)，最后stack.pop获得对象。

对象的release
3.1 如果release线程就是对象的分配线程，直接入栈
3.2 如果release线程和对象的分配线程不是同一个线程，则归还到分配线程的WeakOrderQueue中。
release线程维护的数据结构为：ThreadLocal<Map<Stack,WeakOrderQueue>>

14.规避内存泄露

> 堆内存：JVM创建的内存
> 直接内存：物理内存、堆外内存

Netty的四种ByteBuf：
对于非池化的UnpooledHeapByteBuf （堆内存）、UnpooledDirectByteBuf（直接内存）都可以JVM GC垃圾回收；
而池化的PooledHeapByteBuf 和 PooledDirectByteBuf必须主动用完后放回池里，否则就会内存泄露。所以Netty有自己的引用计数器和回收过程。

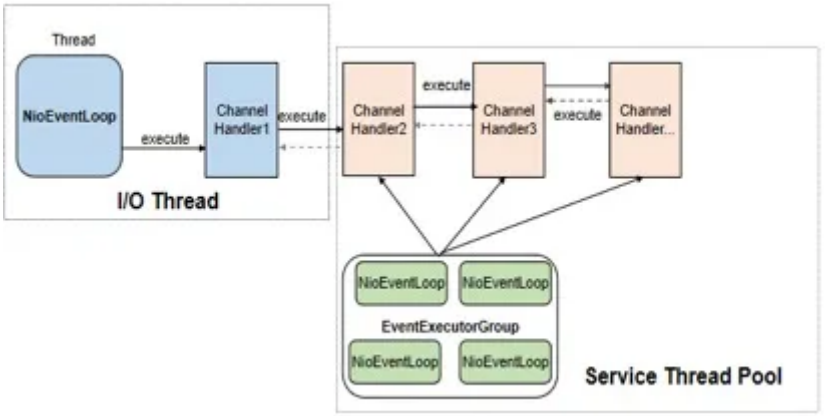
Netty的接收和发送ByteBuf采用的DirectBuffers，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。

池化的ByteBuf计数器实现

ByteBuf的具备实现类都继承了AbstractReferenceCountedByteBuf类，该类实现了对计数器的操作功能。当某一操作使得ByteBuf的引用增加时，调用retain()函数，使计数器的值原子增加，当某一操作使得ByteBuf的引用减少时，调用release()函数，使计数器的值原子减少，减少到0便会触发回收操作。

如何防止内存泄露?

> 出现场景：网络问题消息积压、TCP发送缓冲区满了造成大量排队



解决策略：

1、根据可以接入的最大用户数做客户端并发接入数流控，需要根据内存、cpu以及性能测试做出综合评估
2、设置消息发送的**高低水位**，针对消息的平均发送大小、客户端接入数、JVM内存大小进行计算得出一个合理的高水位取值。服务端推送消息的时候，对Channel的状态进行判断，如果达到高水位之后，Channel的状态会被置为不可写，此时服务端不再继续推送消息。

- **读取消息时**

  1、无论采取哪种解码器实现，都需要对消息的最大长度做限制，超过最大长度时就抛出失败
  2、设置ChannelHandler并发执行，对EventExecutor中任务队列taskQueue做容量限制
  3、对EventLoopGroup做告警通知

- **消息发送时**

  1、高低水位机制，实现对客户端的流量控制；

  ```
  ctx.channel().config().setWriteBufferHighWaterMark(10 \* 1024 * 1024);
  ```

- **ByteBuf合理使用**

  - 对于内存池的请求ByteBuf

    策略1：如果ChannelInboundHandler 继承自 SimpleChannelInboundHandler，ByteBuf会自动释放，最后会调用CountUtil.release(msg)释放当前请求消息。
    策略2：在业务 ChannelInboundHandler 中调用 ctx.fireChannelRead(msg)方法，让请求消息继续向后执行，直到调用到最后一层的DefaultChannelPipeline 的内部类 TailContext，由它来负责释放请求消息。

    > 注意ByteBuf是以何种方式产生的，如果是新对象，需要手动release，直到计数器=0

  - 对于非内存池的请求ByteBuf，也需要按照内存池的方式释放内存。

  - 对于内存池的响应ByteBuf

    只要调用了 writeAndFlush 或者 flush 方法，在消息发送完成之后都会由 Netty 框架进行内存释放，业务不需要主动释放内存。

        - 如果是堆内存（PooledHeapByteBuf），则将 HeapByteBuffer（堆内存） 转换成 DirectByteBuffer（直接内存），并释放 PooledHeapByteBuf 到内存池
        - 如果消息完整的被写到 SocketChannel 中，则释放 DirectByteBuffer
    - 对于非内存池的响应ByteBuf

      无论是什么样分配的ByteBuf，都是将堆内存转换成堆外内存，然后释放堆内存，待消息发送完之后再释放堆外内存。
- **合理设置堆外内存大小**

  -XX:MaxDirectMemorySize和-Dio.netty.maxDirectMemory
- **堆外内存做监控泄露检测分析**
- **堆内存泄露检测分析**

  可以通过配置参数不同选择不同的检测级别，参数设置为`java -Dio.netty.leakDetection.level=advanced`

  通过 jmap -dump:format=b,file=xx pid 命令 Dump 内存堆栈，然后使用 MemoryAnalyzer 工具对内存占用进行分析，查找内存泄漏点，然后结合代码进行分析，定位内存泄漏的具体原因

15.Java多线程时阻塞IO如何中断?

阻塞的IO线程在关闭线程时并不会被打断，需要关闭资源才能被打断。

```
1.执行socketInput.close();阻塞可中断。
2.执行System.in.close();阻塞没有中断。
```

2.MQTT

MQTT适用于低带宽高延迟不可靠的网络环境进行应用层协议传输，TCP核心思想是分组交换，MQTT核心思想是适应物联网环境，MQTT传输单位是消息，每条消息大小上限在Broker代理服务器上设置。

服务质量Qos：最多一次、至少一次、恰好一次

主要就是Topic发布消息payload

MQTT数据包结构：固定头、可变头、消息体

二十八、Nginx

1.Nginx和Netty设计区别

- netty

  Netty 在很多地方进行了无锁化的设计，例如在 I/O 线程内部进行串行操作，避免多线程竞争导致的性能下降问题。表面上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列多个工作线程的模型性能更优。

  Netty拥有两个NIO线程池，分别是`bossGroup`和`workerGroup`，前者处理新建连接请求，然后将新建立的连接轮询交给workerGroup中的其中一个NioEventLoop来处理，后续该连接上的读写操作都是由同一个NioEventLoop来处理。注意，虽然bossGroup也能指定多个NioEventLoop（一个NioEventLoop对应一个线程），但是默认情况下只会有一个线程，因为一般情况下应用程序只会使用一个对外监听端口。

> 这里试想一下，难道不能使用多线程来监听同一个对外端口么，即多线程epoll_wait到同一个epoll实例上?

epoll相关的主要两个方法是epoll_wait和epoll_ctl，多线程同时操作同一个epoll实例，那么首先需要确认epoll相关方法是否线程安全：**简单来说，epoll是通过锁来保证线程安全的，epoll中粒度最小的自旋锁ep->lock(spinlock)用来保护就绪的队列，互斥锁ep->mtx用来保护epoll的重要数据结构红黑树。**

- Nginx多进程针对监听端口的处理策略

  Nginx是通过accept_mutex机制来保证的。accept_mutex是nginx的(新建连接)负载均衡锁，让多个worker进程轮流处理与client的新连接。当某个worker进程的连接数达到worker_connections配置（单个worker进程的最大处理连接数）的最大连接数的7/8时，会大大减小获取该worker获取accept锁的概率，以此实现各worker进程间的连接数的负载均衡。accept锁默认打开，关闭它时nginx处理新建连接耗时会更短，但是worker进程之间可能连接不均衡，并且存在“惊群”问题。只有在使能accept_mutex并且当前系统不支持原子锁时，才会用文件实现accept锁。注意，accept_mutex加锁失败时不会阻塞当前线程，类似tryLock。

二十三、算法

1、常见算法

1.布隆过滤器

是位数组，占用空间小，用来检索元素是否在给定的集合中的一种数据结构，缺点是可能错误识别和一定的删除难度

1.二分查找

```
//寻找左边界
int find(int nums[],int target){
    int l = 0 , r = nums.length - 1;
    while (l < r){
        int mid = l + r >> 1;
        if (target <= nums[mid]) r = mid;
        else l = mid + 1;
    }
    return l;
}
//寻找右边界
int find(int nums[],int target){
    int l = 0 , r = nums.length - 1;
    while (l < r){
        int mid = l + r + 1 >> 1;
        if (target >= nums[mid]) l=mid;     // 注意这里
        else r= mid + 1;
    }
    return l;
}
```

2.排序

1.冒泡

```
void bubbleSort(int[] a) {
    // 进行i轮比较
    for (int i = 0; i < a.length - 1; i++) {
        // 每轮比较后元素会-1
        for (int j = 0; j < a.length - 1 - i; j++) {
            if (a[j] > a[j + 1]) {
                swap(a, j, j + 1);
            }
        }
    }
}
```

2.快速排序

1. 从数列中取出一个数作为基准数
2. 分区，将比它大的数全放到它的右边，小于或等于它的数全放到它的左边
3. 再对左右区间重复第二步，直到各区间只有一个数

```
void quickSort(int[] a, int left, int right) {
    if (left >= right) {
        return;
    }
    int index = sort(a, left, right);
    quickSort(a, left, index - 1);
    quickSort(a, index + 1, right);
}
//返回中轴元素的下标
int sort(int[] a, int left, int right) {
    int key = a[left];      // 基准元素
    while (left < right) {
        // 从high所指位置向前搜索找到第一个关键字小于key的记录和key互相交换
        while (left < right && a[right] >= key) {
            right--;
        }
        a[left] = a[right];
        // 从low所指位置向后搜索，找到第一个关键字大于key的记录和key互相交换
        while (left < right && a[left] <= key) {
            left++;
        }
        a[right] = a[left];
    }
    // 放key值，此时left和right相同
    a[left] = key;
    return left;
}
```

> 时间复杂度O（nlogn）的排序算法有
>
> - 快速排序：当选择的基数是最大值或者是最小值的时候会退化为O（n^2）

- 归并排序
- 希尔排序

3.插入排序

```
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        for (int j = i; j > 0; j--) {
            while (a[j] < a[j - 1]) {
                swap(a, j, j - 1);
            }
        }
    }
}
```

4.希尔排序

希尔排序是基于插入排序改进后的算法。因为当数据移动次数太多时会导致效率低下。所以我们可以先让数组整体有序（刚开始移动的幅度大一点，后面再小一点），这样移动的次数就会降低，进而提高效率。

```
void shellSort(int[] a) {
    for (int step = a.length / 2; step > 0; step /= 2) {
        for (int i = step; i < a.length; i++) {
            int temp = a[i];
            int j;
            for (j = i - step; j >= 0 && a[j] > temp ; j -= step) {
                a[j + step] = a[j];
            }
            a[j + step] = temp;
        }
    }
}
```

5.堆排序

大根堆：堆都是完全二叉树

```
/**
 * @param a 数组
 * @param n 数组长度
 * @param i 要进行heapify的节点，一般就是数组首个节点
 */
public static void heapify(int[] a, int n, int i) {
    // 递归出口
    if (i >= n) {
        return;
    }
    // 左节点下标
    int c1 = 2 * i + 1;
    // 右节点下标
    int c2 = 2 * i + 2;
    int max = i;
```

```
    if (c1 < n && a[c1] > a[max]) {
        max = c1;
    }
    if (c2 < n && a[c2] > a[max]) {
        max = c2;
    }
    // 将左节点，右节点中的最大值和父节点交换
    if (max != i) {
        swap(a, max ,i);
        heapify(a, n, max);
    }
}
```

堆排序：

```
    public static void heapSort(int[] a) {
        buildTree(a);
        for (int i = a.length - 1; i >= 0; i--) {
            swap(a, i, 0);
            heapify(a, i, 0);
        }
    }

    public static void buildTree(int[] a) {
        // 找到最后一个非叶子节点
        int lastNode = a.length - 1;
        int parent = (lastNode - 1) / 2;
        for (int i = parent; i >= 0; i--) {
            heapify(a, a.length, i);
        }
    }
```

二十四、k8s

1.理论篇

1.核心组件

etcd：提供数据库服务保存了整个集群的状态
kube-apiserver：提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制
kube-controller-manager：负责维护集群的状态，比如故障检测、自动扩展、滚动更新等
cloud-controller-manager：是与底层云计算服务商交互的控制器
kube-scheduler：负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上
kubelet：负责维护容器的生命周期，同时也负责Volume和网络的管理
kube-proxy：负责为Service提供内部的服务发现和负载均衡，并维护网络规则
container-runtime：是负责管理运行容器的软件，比如docker

> master：apiserver，controller-manager、etcd、scheduler
> node：kubelet，kube-proxy

2.负载均衡

根据工作环境使用两种类型的负载均衡器，即内部负载均衡器或外部负载均衡器。内部负载均衡器自动平衡负载并使用所需配置分配容器，而外部负载均衡器将流量从外部负载引导至后端容器。

Service是内部的负载均衡器、Ingress是外部的负载均衡器

3.Pod生命周期

```
Pending 部署Pod事务已被集群受理，但当前容器镜像还未下载完。准备竞选
Running 所有容器已被创建，并被部署到k8s节点。
Successed：表示成功退出，并不会被重启。
Failed ：有容器被终止。
Unknown 未知原因，通常是control-manager无法与Pod进行通讯。
```

4.kube-proxy原理

三种代理模式：userspace、iptables、ipvs

详述kube-proxy原理，一个请求是如何经过层层转发落到某个pod上的整个过程?

```
kube-proxy部署在每个Node节点上，它监听 apiserver 中 service 和 endpoint 的变化情况，并对本机iptables做修改，从而实现网络路由。 而其中的负载均衡，也是通过iptables的特性实现的。
```

5.创建pod流程

- 客户端提交 Pod 的配置信息（可以是 yaml 文件定义的信息）到 kube-apiserver。
- Apiserver 收到指令后，通知给 controller-manager 创建一个资源对象。
- Controller-manager 通过 api-server 将 Pod 的配置信息存储到 etcd 数据中心中。
- Kube-scheduler 检测到 Pod 信息会开始调度预选，会先过滤掉不符合 Pod 资源配置要求的节点，然后开始调度调优，主要是挑选出更适合运行 Pod 的节点，然后将 Pod 的资源配置单发送到 Node 节点上的 kubelet 组件上。
- Kubelet 根据 scheduler 发来的资源配置单运行 Pod，运行成功后，将 Pod 的运行信息返回给 scheduler，scheduler 将返回的 Pod 运行状况的信息存储到 etcd 数据中心。

6.健康检查方式

- LivenessProbe 探针：用于判断容器是否存活（running 状态），如果 LivenessProbe 探针探测到容器不健康，则 kubelet 将杀掉该容器，并根据容器的重启策略做相应处理。若一个容器不包含 LivenessProbe 探针，kubelet 认为该容器的 LivenessProbe 探针返回值用于是“Success”。
- ReadineeProbe 探针：用于判断容器是否启动完成（ready 状态）。如果 ReadinessProbe 探针探测到失败，则 Pod 的状态将被修改。Endpoint Controller 将从 Service 的 Endpoint 中删除包含该容器所在 Pod 的 Eenpoint。
- startupProbe 探针：启动检查机制，应用一些启动缓慢的业务，避免业务长时间启动而被上面两类探针 kill 掉。

1）LivenessProbe探针的常见方式

- ExecAction：在容器内执行一个命令，若返回码为 0，则表明容器健康。
- TCPSocketAction：通过容器的 IP 地址和端口号执行 TCP 检查，若能建立 TCP 连接，则表明容器健康。
- HTTPGetAction：通过容器的 IP 地址、端口号及路径调用 HTTP Get 方法，若响应的状态码大于等于 200 且小于 400，则表明容器健康。

7.Pod常见调度方式

- Deployment 或 RC：该调度策略主要功能就是自动部署一个容器应用的多份副本，以及持续监控副本的数量，在集群内始终维持用户指定的副本数量。
- NodeSelector：定向调度，当需要手动指定将 Pod 调度到特定 Node 上，可以通过 Node 的标签（Label）和 Pod 的 nodeSelector 属性相匹配。
- NodeAffinity 亲和性调度：亲和性调度机制极大的扩展了 Pod 的调度能力，目前有两种节点亲和力表达：
    - requiredDuringSchedulingIgnoredDuringExecution：硬规则，必须满足指定的规则，调度器才可以调度 Pod 至 Node 上（类似 nodeSelector，语法不同）。
    - preferredDuringSchedulingIgnoredDuringExecution：软规则，优先调度至满足的 Node 的节点，但不强求，多个优先级规则还可以设置权重值。
- Taints 和 Tolerations（污点和容忍）：
- Taint：使 Node 拒绝特定 Pod 运行；
- Toleration：为 Pod 的属性，表示 Pod 能容忍（运行）标注了 Taint 的 Node。

8.Deployment升级过程

- 初始创建 Deployment 时，系统创建了一个 ReplicaSet，并按用户的需求创建了对应数量的 Pod 副本。
- 当更新 Deployment 时，系统创建了一个新的 ReplicaSet，并将其副本数量扩展到 1，然后将旧 ReplicaSet 缩减为 2。
- 之后，系统继续按照相同的更新策略对新旧两个ReplicaSet 进行逐个调整。
- 最后，新的 ReplicaSet 运行了对应个新版本 Pod 副本，旧的 ReplicaSet 副本数量则缩减为 0。

9.Deployment升级策略

在 Deployment 的定义中，可以通过 spec.strategy 指定 Pod 更新的策略，目前支持两种策略：Recreate（重建）和 RollingUpdate（滚动更新），默认值为 RollingUpdate。

10.Kubernetes自动扩容机制

Kubernetes 使用 Horizontal Pod Autoscaler（HPA）的控制器实现基于 CPU 使用率进行自动 Pod 扩缩容的功能。HPA 控制器周期性地监测目标 Pod 的资源性能指标，并与 HPA 资源对象中的扩缩容条件进行对比，在满足条件时对 Pod 副本数量进行调整。

11.Service

1）Service的类型

- ClusterIP：虚拟的服务 IP 地址，该地址用于 Kubernetes 集群内部的 Pod 访问，在 Node 上 kube-proxy 通过设置的 iptables 规则进行转发；
- NodePort：使用宿主机的端口，使能够访问各 Node 的外部客户端通过 Node 的 IP 地址和端口号就能访问服务；
- LoadBalancer：使用外接负载均衡器完成到服务的负载分发，需要在 spec.status.loadBalancer 字段指定外部负载均衡器的 IP 地址，通常用于公有云。

2）Service分发到后端的策略

- RoundRobin：默认为轮询模式，即轮询将请求转发到后端的各个 Pod 上。
- SessionAffinity：基于客户端 IP 地址进行会话保持的模式，即第 1 次将某个客户端发起的请求转发到后端的某个 Pod 上，之后从相同的客户端发起的请求都将被转发到后端相同的 Pod 上。

3）Headless Service

HeadlessService可以返回后端所有pod的ip地址，记录了dns记录，不会进行负载均衡，也不会走kube-proxy。可以在程序内部自定义代码进行自定义负载均衡和流量控制。

Headless Service部署时指定 Cluster IP（spec.clusterIP）的值为 "None" 来创建，Kubernetes不会为该Service分配任何IP地址。

12.集群外访问Kubernetes内部方式

- 映射 Pod 到物理机：将 Pod 端口号映射到宿主机，即在 Pod 中采用 hostPort 方式，以使客户端应用能够通过物理机访问容器应用。
- 映射 Service 到物理机：将 Service 端口号映射到宿主机，即在 Service 中采用 nodePort 方式，以使客户端应用能够通过物理机访问容器应用。
- 映射 Sercie 到 LoadBalancer：通过设置 LoadBalancer 映射到云服务商提供的 LoadBalancer 地址。这种用法仅用于在公有云服务提供商的云平台上设置 Service 的场景。

13.Ingress

为Service层提供负载均衡，用于将不同 URL 的访问请求转发到后端不同的 Service，以实现 HTTP 层的业务路由机制。

Kubernetes 使用了 Ingress 策略和 Ingress Controller，两者结合并实现了一个完整的 Ingress 负载均衡器。使用 Ingress 进行负载分发时，Ingress Controller 基于 Ingress 规则将客户端请求直接转发到 Service 对应的后端 Endpoint（Pod）上，从而跳过 kube-proxy 的转发功能，kube-proxy 不再起作用，全过程为：ingress controller + ingress 规则 ----> services。

14.镜像下载策略

Kubernetes 的镜像下载策略有三种：Always、Never、IFNotPresent。

15.各模块如何与API Server通信

集群内的各个功能模块通过 API Server 将信息存入 etcd，当需要获取和操作这些数据时，每隔一段时间则通过 API Server 提供的 REST 接口（用 GET、LIST 或 WATCH 方法）来实现，从而实现各模块之间的信息交互。

16.Scheduler作用和原理

Kubernetes Scheduler 是负责 Pod 调度的重要功能模块，Kubernetes Scheduler 在整个系统中承担了“承上启下”的重要功能，“承上”是指它负责接收 Controller Manager 创建的新 Pod，为其调度至目标 Node；“启下”是指调度完成后，目标 Node 上的 kubelet 服务进程接管后继工作，负责 Pod 接下来生命周期。

Kubernetes Scheduler 根据如下两种调度算法将 Pod 绑定到最合适的工作节点：

- 预选（Predicates）：输入是所有节点，输出是满足预选条件的节点。kube-scheduler 根据预选策略过滤掉不满足策略的 Nodes。如果某节点的资源不足或者不满足预选策略的条件则无法通过预选。如“Node 的 label 必须与 Pod 的 Selector 一致”。
- 优选（Priorities）：输入是预选阶段筛选出的节点，优选会根据优先策略为通过预选的 Nodes 进行打分排名，选择得分最高的 Node。例如，资源越富裕、负载越小的 Node 可能具有越高的排名。

17.kubelet作用

在每个 Node（又称 Worker）上都会启动一个 kubelet 服务进程。该进程用于处理 Master 下发到本节点的任务，管理 Pod 及 Pod 中的容器。每个 kubelet 进程都会在 API Server 上注册节点自身的信息，定期向 Master 汇报节点资源的使用情况，并通过 cAdvisor 监控容器和节点资源。

kubelet 使用 cAdvisor 对 worker 节点资源进行监控。在 Kubernetes 系统中，cAdvisor 已被默认集成到 kubelet 组件内，当 kubelet 服务启动时，它会自动启动 cAdvisor 服务，然后 cAdvisor 会实时采集所在节点的性能指标及在节点上运行的容器的性能指标。

18.Kubernetes如何保证集群的安全性

- 基础设施方面：保证容器与其所在宿主机的隔离；
- 权限方面：
    - 最小权限原则：合理限制所有组件的权限，确保组件只执行它被授权的行为，通过限制单个组件的能力来限制它的权限范围。
    - 用户权限：划分普通用户和管理员的角色。
- 集群方面：
    - API Server 的认证授权：Kubernetes 集群中所有资源的访问和变更都是通过 Kubernetes API Server 来实现的，因此需要建议采用更安全的 HTTPS 或 Token 来识别和认证客户端身份（Authentication），以及随后访问权限的授权（Authorization）环节。
    - API Server 的授权管理：通过授权策略来决定一个 API 调用是否合法。对合法用户进行授权并且随后在用户访问时进行鉴权，建议采用更安全的 RBAC 方式来提升集群安全授权。
    - 敏感数据引入 Secret 机制：对于集群敏感数据建议使用 Secret 方式进行保护。
    - AdmissionControl（准入机制）：对 kubernetes api 的请求过程中，顺序为：先经过认证 & 授权，然后执行准入操作，最后对目标对象进行操作。

其他：https://xie.infoq.cn/article/28738666912fc1cdd7791b7e4

2.应用篇

1.某个pod启动需要获取pod名称

```
env:
- name: test
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
```

2.集群如何预防雪崩

```
1、为每个pod设置资源限制
2、设置kubelet资源预留
```

3.etcd如何备份

```
export ETCDCTL_API=3
etcdctl --cert=/etc/kubernetes/pki/etcd/server.crt \
        --key=/etc/kubernetes/pki/etcd/server.key \
        --cacert=/etc/kubernetes/pki/etcd/ca.crt \
        snapshot save /data/test-k8s-snapshot.db
```

4.RC、RS、Deployment

RC （ReplicationController ）主要的作用就是用来确保容器应用的副本数始终保持在用户定义的副本数。即如果有容器异常退出，会自动创建新的Pod来替代；而如果异常多出来的容器也会自动回收。

Kubernetes 官方建议使用 RS（ReplicaSet ） 替代 RC （ReplicationController ） 进行部署，RS 跟 RC 没有本质的不同，只是名字不一样，并且 RS 支持集合式的 selector，而 Replication Controller 使用基于权限的选择器

StatefulSet主要用于部署持久化的应用，Deployment主要用于部署无状态的应用

5.iptables和ipvs原理

iptables是老版本的kube-proxy 的默认模式，iptables 模式下的 kube-proxy 不再起到 Proxy 的作用，其核心功能：通过 API Server 的 Watch 接口实时跟踪 Service 与 Endpoint 的变更信息，并更新对应的 iptables 规则，Client 的请求流量则通过 iptables 的 NAT 机制“直接路由”到目标 Pod，效率较低。

ipvs是新版本的kube-proxy的默认模式，专门用于高性能负载均衡，并使用更高效的数据结构（Hash 表），允许几乎无限的规模扩张

与 iptables 相比，IPVS 拥有以下明显优势：

- 为大型集群提供了更好的可扩展性和性能；
- 支持比 iptables 更复杂的复制均衡算法（最小负载、最少连接、加权等）；
- 支持服务器健康检查和连接重试等功能；
- 可以动态修改 ipset 的集合，即使 iptables 的规则正在使用这个集合。

6.什么是静态pod

静态 Pod 是由 kubelet 进行管理的仅存在于特定 Node 的 Pod 上，他们不能通过 API Server 进行管理，无法与 ReplicationController、Deployment 或者 DaemonSet 进行关联，并且 kubelet 无法对他们进行健康检查。静态 Pod 总是由 kubelet 进行创建，并且总是在 kubelet 所在的 Node 上运行。

3.腾讯云

> https://blog.51cto.com/yw666/2550027
>
> https://blog.sholdboyedu.com/?p=1718

1.Service流量转发过程

service 的功能是由 kube-proxy组件实现的，kube-proxy 的模式有三种 userspace、iptables、ipvs。目前常用的是iptables和ipvs，两者也有一定的区别，iptables 规则过多会造成性能下降，因为它是基于链表实现的，查找复杂度是O(n)，当节点数量过多时，iptables 规则成几何级增长，内核支撑不住，最终 iptables 将成为性能瓶颈。iptables 是专门用来做防火墙的，而不是做负载均衡（虽然小规模也是可以的），kube-proxy 使用 IPVS NAT模式 作为 kube-proxy 后端，不仅提高了转发性能，再结合 ipset 还使 iptables 规则变得清楚，iptables 规则数量不会变。

iptables、ipvs 都是基于netfilter，但ipvs是专门做负载均衡，支持丰富的调度算法，配置简单，并且它是基于ipset进行散列快速查找，复杂度为O(1)，但 ipvs 在某些场景下，也需要用到 iptables NAT 规则，但iptables 规则不会增加，他使用的是 ipset 规则集。

2.Calico

同一台服务器的多个容器间通信可以直接走docker网桥，网络模式有四种（Brige默认模式，Host，Containter，None）。
但是多台服务器的Docker之间通信方式就得借助于CNI网络来进行通信，常见就是flannel和Calico。

Calico是以容器方式存在，信息存储在etcd中。

1）Calico和flannel区别和原理

1、calico根据iptables规则进行路由转发，并没有进行封包、解包的过程，不需要额外的NAT，这和flannel比起来效率就会快多。Calico包含了许多重要组件：Felix，etcd，BGP Client，BGP Route Reflector。Calico更加灵活，性能更快，有IPV6场景时必须使用Calico。Calico是一个纯三层的网络插件，提供了docker dns服务可以根据hostname直接访问

```
- Felix：主要负责路由配置以及ACLS规则的配置以及下发，它存在在每个node节点上。
- etcd：分布式键值存储，主要负责网络元数据一致性，确保Calico网络状态的准确性
- BGPClient(BIRD)，主要负责把 Felix写入 kernel的路由信息分发到当前 Calico网络，确保workload间的通信的有效性；
- BGPRoute Reflector(BGP)路由协议，大规模部署时使用，摒弃所有节点互联的mesh模式，通过一个或者多个 BGPRoute Reflector 来完成集中式的路由分发。
```

2、Flannel实质是一种覆盖网格，也就是将TCP数据包装在另一种网络包里面进行路由转发和通信，目前已经支持UDP、VxLAN、AWS VPC和GCE路由等数据转发方式。flannel使用更简单，上手更快。Flannel通过Etcd服务维护了一张节点间的路由表，详细记录了各节点子网网段。

1、数据从源容器中发出后，经由所在主机的docker0虚拟网卡转发到flannel0虚拟网卡，这是个P2P的虚拟网卡，flanneld服务监听在网卡的另外一端。
2、Flannel通过Etcd服务维护了一张节点间的路由表，该张表里保存了各个节点主机的子网网段信息。
3、源主机的flanneld服务将原本的数据内容UDP封装后根据自己的路由表投递给目的节点的flanneld服务，数据到达以后被解包，然后直接进入目的节点的flannel0虚拟网卡，然后被转发到目的主机的docker0虚拟网卡，最后就像本机容器通信一样的由docker0路由到达目标容器。

3.Ingeress

- Ingress策略：

  ipBlock：匹配到的ip地址可以访问，except匹配到的ip地址不可以访问
  podSelector：指定匹配到的label的pod能够访问
  namespaceSelector：指定命名空间内任何pod都可以访问

4.Iaas Paas Saas

使用云服务企业无需再配备IT方面的专业技术人员，同时又能得到最新的技术应用，满足企业对信息管理的需求。

- Iaas

  基础设施即服务

- Pass

  平台即服务，低代码平台

- Saas

  软件即服务，根本不需要开发，商家只专注自己的业务就可以

5.节点宕机，简述pod驱逐流程

1、当pod因为某些原因不可用的时候，会经过一段时间pod-eviction-timeout超时后，会被control-manager计划删除
2、k8s集群每个节点都有一个节点生命周期控制器，会和kubelet通信来收集各个节点pod的状态，一旦不可用，会做好标记加上污点，节点上的所有pod都会被污点管理器计划删除。而节点被认定为不可用状态到删除节点上的pod是有一段时间的，这段时间被称为容忍度。
3、在删除pod时，污点管理器会创建污点标记事件，然后驱逐pod，但是此时pod状态还是Running。deployment控制器会在被驱逐后创建新的pod，但是statefulSet控制器不会创建新的pod

6.几种Controller控制器

1、Deployment：适合无状态的服务部署
2、StatefulSet：适合有状态的服务部署
3、DaemonSet：一次部署，所有node节点都会部署。例如在每个节点运行日志收集、监控
4、Job：一次性的执行任务。类似linux的job，如离线数据处理，视频解码等业务
5、Cronjob：周期性的执行任务。
6、IngressController：提供支持七层的https请求，替换Service只是四层的请求

7.kube-proxy三种工作模式和原理

1、userspace模式：会为每一个Service创建一个监听端口。在进行转发处理时会增加两次内核和用户空间之间的数据拷贝，效率较另外两种模式低一些。好处是当后端的Pod不可用时，kube-proxy可以重试其他pod
2、iptables模式：为了避免内核和用户空间的数据拷贝，提高转发效率，会为每个pod创建iptables规则，当后盾pod不可用的时候也无法重试。
3、ipvs模式：是基于内核模式通过netfilter实现的，采用hash表才存储规则，因此规则较多的情况下，转发效率更高，还支持更多的负载均衡算法。必须要操作系统支持ipvs模块。

8.探针

每种探针都有三种探测方式是http探测、exec执行命令探测、tcp探测

- liveness probes存活性探针：用于判定容器是否存活
- readiness probes就绪性探针：用于判定容器是否启动完成

9.Pod启动失败如何排错

1、一般查看系统资源是否满足，然后就是查看pod日志看看原因

2、describe和logs通过俩命令参数查看pod失败的原因

3、还有就是看组件日志，apiserver等组件日志，有没有异常

4、访问不到看看网络插件状态和日志，还有就是dns状态和日志

10.云上日志收集

在k8s上主推的是基于云原生的EFK，这里的F是指CNCF认证的子项目fluented，选择了filebeat

11.iptables的四个表五个链

raw表：确定是否对该数据包进行状态跟踪
mangle表：为数据包设置标记
nat表：修改数据包中的源、目标IP地址或端口
filter表：确定是否放行该数据包（过滤）

INPUT：处理入站数据包
OUTPUT：处理出站数据包
FORWARD：处理转发数据包
POSTROUTING链：在进行路由选择后处理数据包
PREROUTING链：在进行路由选择前处理数据包

4.内核网络命令

获取socket信息：

```
#显示本地打开的所有端口
ss -l
#列出当前socket的详细信息
ss -s
#显示所有tcp socket
ss -at
#显示所有udp socket
ss -au
#显示所有已建立的 ssh 连接
ss -o state established '( dport = :ssh or sport = :ssh )'
#显示所有已建立的HTTP连接
ss -o state established '( dport = :http or sport = :http )'
```

查看路由信息：

```
#查看路由表
route -n
ip route show
#添加路由规则
route add
#域名解析
nslookup 域名
dig 域名
#测试当前主机到目的主机经过了那些网络节点
traceroute [参数] [主机|IP]
#网卡基础配置
nmcli [参数]
#跟踪某个进程的所有线程的read write调用
strace -s 1024 -f -e trace=read,write -p 12345
#设置定时任务
crontab
```

查看网络连接状态：

```
#ss是netstat的升级版
netstat -anpt
#查看端口占用
netstat -alp |grep port
lsof -i :[port]
```

抓包分析：

```
#不指定任何参数，监听第一块网卡上经过的数据包。主机上可能有不止一块网卡，所以经常需要指定网卡
tcpdump
#监听特定网卡
tcpdump -i eth0
#限制抓包的数量，抓到 1000 个包后，自动退出
tcpdump -c 1000
#监听特定主机，监听主机 10.0.0.100 的通信包，出、入的包都会被监听
tcpdump host 10.0.0.100
#特定来源
tcpdump src host hostname
#特定目标地址
```

```
tcpdump dst host hostname
#如果不指定 src 跟 dst，那么来源或者目标是 hostname 的通信都会被监听
tcpdump host hostname
#特定端口
tcpdump port 3000
#监听 TCP/UDP，服务器上不同服务分别用了 TCP、UDP 作为传输层，假如只想监听 TCP 的数据包
tcpdump tcp
#来源主机+端口+TCP，监听来自主机 10.0.0.100 在端口 22 上的 TCP 数据包
tcpdump tcp port 22 and src host 10.0.0.100
#监听特定主机之间的通信
tcpdump ip host 10.0.0.101 and 10.0.0.102
#10.0.0.101 和除了 10.0.0.1 之外的主机之间的通信
tcpdump ip host 10.0.0.101 and ! 10.0.0.1
```

5.Istio

Service Mesh：服务网格， Service Mesh是专用的基础设施层，轻量级高性能网络代理。提供安全的、快速的、可靠地服务间通讯，与实际应用部署一起，但对应用透明。

Istio 超越 spring cloud和dubbo 等传统开发框架之处， 就在于不仅仅带来了远超这些框架所能提供的功能， 而且也不需要应用程序为此做大量的改动， 开发人员也不必为上面的功能实现进行大量的知识储备。 Istio 超越 spring cloud和dubbo 等传统开发框架之处， 就在于不仅仅带来了远超这些框架所能提供的功能， 而且也不需要应用程序为此做大量的改动， 开发人员也不必为上面的功能实现进行大量的知识储备。

Istio 大幅降低微服务架构下应用程序的开发难度，势必极大的推动微服务的普及。个人乐观估计，随着isito的成熟，微服务开发领域将迎来一次颠覆性的变革。

3.4 压缩算法