场景考察

1. 排名
    - 数据量小可以实现内存排名
    - 百万级可以使用Redis Sorted Set排名
    - 千万级、亿级可使用分区桶的的方式实现排名
2. 结构抽象

    用户注册及权限分配，控制到接口或者操作按钮级别（user、menu、menu_operation、user_permission）
3. 设计模式
    - 多层次计算，上层依赖下层数据，处理代码设计模式
    - 多策略处理
4. 数据留痕&数据量
    - 每天job多次计算，如何保证每次job异常而不覆盖之前正确的版本，且数据量不会太大

Java基础

> - Integer 缓存、扩展（最大java.lang.Integer.IntegerCache.high，最小不可设置，min：-128，max：127）
> - Long 缓存min：-128，max：127，最大值不可设置
> - String 最大长度，内部使用char数组保存，理论上取决于数组长度的最大值int 最大值接近4G，实际存不了这么大

1. Object类中的方法：hashCode、equals、toString、clone、getClass、wait、notify、notifyAll、finalize
2. equals和hashCode：Object中的equals是判断两个对象是否相等（内存地址相等），和“==”一样，如果重写则是比较两个对象值是否相等。equals和hashCode的关系，如果equals为true，hashCode必须相等；hashCode相等，equals不一定为true。 重写了equals必须重写hashCode。例如HashMap中hashCode映射到同一个哈希桶位置，还需要通过equals比较找出真正的值。
3. JDK基础类
    - final类型的类不能继承，String和包装类都是
    - String类每次在对象上操作都会产生一个新的对象；StringBuffer线程安全，方法上加了synchronized，底层是一个char数组，使用Arrays.copyOf()扩容，使用System.arraycopy()拷贝；StringBuilder和StringBuffer一样，只是不是线程安全
    - "abc"+"def"，编译器通过StringBuilder.append()拼接，再toString
    - Integer和Long从-128~127都有缓存,包装方法valueOf，还原方法XXXValue方法
4. Java都是值传递：基本数据类型是将实参的值copy一份给形参；引用类型是把实参地址值copy一份给形参，实参和形参都指向同一个对象，所以都是操作的同一个对象。
5. Java序列化
    - 概念

        - 序列化：序列化就是把对象转换成有序的字节流，以便在网络上传输或者保存到本地文件中。
        - 反序列化：从文件或网络上获取序列化后的对象字节流，根据字节流中所保存的对象状态及描述信息，通过反序列化重建对象。
    - Java中如何序列化和反序列化 只有实现了Serializable或Externalizable接口的类的对象才能被序列化，否则抛出异常。
        - ObjectOutputStream：writeObject方法序列化对象
        - ObjectInputStream：readObject反序列化对象
    - 控制序列化的方式
        - 使用transient关键字和Serializable
        - 实现Serializable接口，重写writeObject和readObject方法
        - 实现Externalizable接口，一定要有默认无参构造函数，实现readExternal和writeExternal方法
    - 序列化算法一般会按步骤做如下事情：
        - 将对象实例相关的类元数据输出。
        - 递归地输出类的超类描述直到不再有超类。
        - 类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。
        - 从上至下递归输出实例的数据。
    - 序列化问题
        - 序列化时，只对对象的状态进行保存，而不管对象的方法；
        - 当一个父类实现序列化，子类自动实现序列化，不需要显式实现Serializable接口；
        - 一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化；
        - 并非所有的对象都可以序列化：安全原因、资源分配原因（socket、thread）
        - 声明为static和transient类型的成员数据不能被序列化。因为static代表类的状态，transient代表对象的临时数据。
        - 序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。
        - Java有很多基础类已经实现了serializable接口，比如String,Vector等。
        - 如果一个对象的成员变量是一个对象，那么这个对象的数据成员也会被保存！这是能用序列化解决深拷贝的重要原因；
6. 动态代理
    - JDK动态代理
        - 1）使用JDK动态代理的五大步骤
            - 通过实现InvocationHandler接口来实现增强；
            - 通过Proxy.getProxyClass获得动态代理类；
            - 通过反射机制获取代理类的构造方法，以InvocationHandler类为构造参数
            - 通过构造方法获取代理对象并将自定义的InvocationHandler的实例作为参数传入
            - 通过代理对象调用目标方法
        - 2）常用方法

```
Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces,
InvocationHandler h)  Proxy.getProxyClass(ClassLoader loader, Class<?>[]
interfaces)  invoke(Object Proxy, Method method, Object[] args)
```

        3）代理类
            - 代理类继承了Proxy和实现了目标类的所有接口，构造方法中需要传入一个

InvocationHandler实例
        - 生成的代理类实现类Object中的hashCode、equals、toString和接口中的所有方法，有多少个方法就有多少个static Method变量，在static代码块中实现方法对象和类的实际方法绑定
        - 代理类生成后，当调用代理类方法时，最终由InvocationHandler的invoke方法来调用目标方法
    - ASM：Java字节码操作框架，它能被用来动态生成类或者增强既有类的功能，ASM可以直接产生二进制class文件，也可以在类被加载到Java虚拟机之前动态改变类行为。
        - 可实现：1）获取class文件的详细信息，包括类名、父类名、接口、成员名、方法名、方法参数、局部变量、元数据等；2）对class文件进行动态修改，如增加、删除、修改类方法、在方法中添加指令等。
        - API：1）ClassVisitor：抽象类，负责类内容的访问；2）ClassReader：将class解析成byte数组，然后会通过accept方法去按顺序调用绑定对象（继承了ClassVisitor的实例）的方法；3）ClassWriter：是ClassVisitor子类，直接通过toByteArray方法返回编译后的class。
    - CGLib：代理类继承被代理类，并在代理类中对代理方法进行强化处理，底层使用ASM实现，代理类有Enhancer类创建，被final修饰的类和方法不能被代理。
        - 代理类会获得所有在父类继承来的方法，并且会有MethodProxy与之对应，在创建MethodProxy对象时会把被代理类和代理类带入，代理类会为每个方法生成两个方法，一个与原方法同名，一个`CGLIB$xxx$0`
        - 在第一次执行MethodProxy invoke/invokeSuper时，为代理类和被代理类各生成一个Class，这个Class会为代理类或被代理类的方法分配一个index(int类型)
        - 当调用代理类的方法，会调用到同名的方法，其中会调用拦截器的invoke方法，当拦截器中使用invokeSuper时会使用代理类的fastclass其中返回`CGLIB$xxx$0`方法，其实现就是通过super调用目标方法；如果拦截器使用invoke方法，则会使用被代理类的fastclass，其调用的是代理类的同名方法，其中又会调用拦截器，所以会出现死循环
        - 调用过程：代理对象调用this.setPerson方法->调用拦截器->`methodProxy.invokeSuper->CGLIB$setPerson$0`->被代理对象setPerson方法
    - javassist：开源的分析、编辑和创建Java字节码的类库，简单、快速，直接使用java编码的形式，不需要了解虚拟机指令，就能动态改变类的结构或者动态生成类。Javassist中最为重要的是ClassPool，CtClass ，CtMethod 以及 CtField这几个类。

7. Java中常见的容器
    - 有序集合
        - ArrayList：初始化一个空的Object数组，第一次add时扩容为10，当空间不足时扩容为原来的1.5倍；通过Arrays.copyOf进行拷贝；index通过遍历equals比较，contain调用的index
        - LinkedList：由内部Node构建一个FIFO的双向链表；LinkedList中保存了头`first`和尾`last`节点；
        - Vector：线程安全的list，通过方法加synchronized
        - CopyOnWriteArrayList：写入时先加锁（ReentrantLock）读到数组，然后扩展加入新元素，最后赋值解锁；读支持并发读
    - 无序无重复集合Set
        - HashSet：由HashMap的key构成
        - LinkedHashSet：底层由LinkedHashMap实现，由hashCode决定元素的存储位置，同时使用链表维护元素的次序，插入性能低于HashSet，遍历时要快
        - TreeSet：TreeSet可以确保集合元素处于排序状态
    - Map容器

- HashMap：JDK1.7中由内部类Node存储hash、key、value和构建一个单向链表；HashMap内部维护一个Node数组；初始容量为16，负载因子为 0.75；put时对key进行hash取到在数组的位置，然后加入链表，get时先定位到数组元素，再遍历该元素处的链表。
- HashTable：线程安全的哈希表
- LinkedHashMap：继承HashMap，底层使用哈希表和双向链表保存元素
- TreeMap：是红黑树算法的实现
- ConcurrentHashMap：高并发容器，在JDK1.7版本中，ConcurrentHashMap的数据结构是由一个Segment数组，每个Segment元素包含HashEntry数组，Segment继承了ReentrantLock，Segment数组的意义就是将一个大的table分割成多个小的table来进行加锁，也就是锁分离技术，而每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样。
- JDK1.8中的HashMap：底层使用哈希表（数组+链表），当链表过长会将链表转成红黑树，以实现O(logn)时间复杂度内查找；put的过程，1）对key求hash值（低16位与高16位异或）；2）如果数组没有初始化则初始化，数组长度始终是2^n；3）然后计算下标（(n-1)& hash），如果没有碰撞则直接放入；4）如果碰撞了，如果节点存在则替换，否则以链表的方式链接到后面；5）如果链表长度超过阈值8，就把链表转成红黑树；6）如果实际长度大于容量加载因子则resize；7）扩容为原来的两倍，如果只有一个元素的直接计算新下标存入，如果是红黑树的重新调整，如果是一个链表则这个值只可能在两个地方，一个是原下标的位置，另一种是在下标为 <原下标+原容量> 的位置。
- JDK1.8中的ConcurrentHashMap实现已经摒弃了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，整个看起来就像是优化过且线程安全的HashMap。分阶段使用CAS和加锁保证线程安全，1）初始化通过CAS使用volatile修饰的sizeCtl来控制只有一个线程能初始化数组；2）没有碰撞则使用CAS存入；3）如果有碰撞则使用synchronized锁住数组中的元素节点；4）如果是在扩容状态，则去帮忙扩容，每个线程领取一个子任务，完成之后进行报备。

JVM

> JVM内存设置多大合适？Xmx和Xmn如何设置？
>
> - Java整个堆大小设置，Xmx 和 Xms设置为老年代存活对象的3-4倍，即FullGC之后的老年代内存占用的3-4倍
> - 永久代 PermSize和MaxPermSize设置为老年代存活对象的1.2-1.5倍。
> - 年轻代Xmn的设置为老年代存活对象的1-1.5倍。
> - 老年代的内存大小设置为老年代存活对象的2-3倍。

1. Java内存区域：程序计数器、虚拟机栈、本地方法栈、Java堆、方法区（常量池）。 方法区是规范，永久代和元数据区是实现

   > 从JDK8开始，永久代(PermGen)的概念被废弃掉了，取而代之的是一个称为Metaspace的存储空间。Metaspace使用的是本地内存，而不是堆内存，也就是说在默认情况下Metaspace的大小只与本地内存大小有关。

2. 对象创建：检查这个类是否被加载，加载类，为对象分配内存（指针碰撞-规则空间、空闲列表-不规则空间）同步处理和本地线程缓冲TLAB，初始化对象内存空间，设置对象头，初始化对象。
3. 对象内存布局：对象头、实例数据、对齐填充。对象头中存放类元数据、哈希码、GC分代年龄、锁标志指针、类型指针、数组长度等。
4. 对象访问定位：句柄（稳定）、直接指针（快）。

5. 引用类型：强引用、软引用、弱引用、虚引用。
6. 可达性分析：从节点开始搜索，是否有GC Root与之相连。可作为GC Root的对象：
    - 虚拟机栈中引用的对象
    - 本地方法栈中引用的对象
    - 方法区中类静态属性引用的对象
    - 方法区中常量引用的对象
7. 判断对象是否可回收：至少需要标记两次，可达性分析，不可达第一次标记，是否执行finalize方法，不执行则标记可回收；执行则进入一个FinalizerQueue，再执行可达性分析，判断是否可达进行二次标记。
8. 回收方法区：废弃的常量（没有任务对象引用）和无用的类（所有实例都已被回收、加载该类的ClassLoader已回收、该类的java.lang.Class对象没有任何地方引用）
9. 垃圾收集算法
    - 标记-清除算法：效率不高、空间不连续
    - 复制算法：空间连续，使用两块内存，浪费资源
    - 标记-整理算法：
    - 分代收集算法：将Java堆分成几个区域，新生代和老年代，新生代又分为Eden、Survivor0、Survivor1三个区域，方法区对应回收分区的永久代。针对各区域对象存活周期使用不同的收集算法，新生代复制算法、老年代和永久代标记-整理算法。
10. HotSpot收集实现：枚举根节点（一致性STW、准确性GC）、安全点（抢断式中断、主动式中断、安全区域。
11. 垃圾收集器
    - Serial收集器：新生代、单线程、一次STW、复制算法，Client模式下的默认新生代收集器，优点: 简单高效，单线程，独占式的垃圾回收。 缺点: 在内部自发的进行垃圾收集的，一旦开启，必须暂停其他线程。配置：`-XX:+UseSerialGC`
    - ParNew收集器：新生代、多线程、一次STW、复制算法，常和CMS搭配使用。配置：`-XX:+UseParNewGC`
    - Parallel Scavenge收集器：新生代、多线程、复制算法、可控吞吐量 。配置：`-XX:+UseParallelGC`
    - Serial Old收集器：老年代、单线程、标记-整理算法，作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。配置：`-XX:+UseParallelGC`
    - Parallel Old收集器：老年代、多线程、标记-整理算法 。配置：`-XX:+UseParallelOldGC`
    - CMS收集器：老年代、多线程、两阶段STW（初始标记、重新标记）、标记-清理算法、最短回收停顿时间（初始标记（标记一下GC Roots能直接关联到的对象）、并发标记（并发标记阶段就是进行GC RootsTracing的过程）、重新标记（修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象）、并发清除） 。优点：并发收集、低停顿，是一个并行垃圾处理器，用户线程与垃圾回收线程一起运行(不包括前两个标记)；缺点: 1.CMS收集器无法处理浮动垃圾 2.使用的是“标记—清除”算法实现的收集器，容易产生大量空间碎片。 3.对CPU资源比较敏感，，面向并发设计的程序对CPU资源都非常敏感的，受CPU的数量影响会比较大。数量越小，影响越大。配置：`-XX:+UseConcMarkSweepGC`
    - G1收集器：新生代和老年代均可、多线程、复制算法与标记-整理算法、GC停顿可控、分区域回收、几乎不需要STW，（初始标记、并发标记、最终标记、筛选回收）三个阶段STW
12. 内存分配策略
    - 对象优先在Eden区分配，如果线程分配了TLAB则优先在TLAB（线程本地分配缓冲）上分配，当Eden空间不足时发起一次Minor GC
    - 大对象直接进入老年区，设置-XX:PretenureSizeThreshold参数，-XX:PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效。
    - 长期存活对象进入老年代，-XX:MaxTenuringThreshold=8参数用于设定对象最大年龄阈值。

    - 对象动态年龄判断
    - 空间分配担保
13. Minor GC、Full GC触发条件 - Minor GC触发条件：当Eden区满时，触发Minor GC - Full GC触发条件：
    - 调用System.gc时，系统建议执行Full GC，但是不是必然执行
    - 在Survivor区域的对象满足晋升到老年代的条件时，晋升进入老年代的对象大小大于老年代的可用内存，这个时候会触发Full GC。
    - XX:MetaspaceSize=21810376B（约为20.8MB）超过这个值就会引发Full GC，这个值不是固定的，是会随着JVM的运行进行动态调整的
    - Minor GC晋升到老年代的平均大小大于老年代的剩余空间，两种晋升情况：由Eden区出生多次GC后年龄达到默认15的对象；JVM发现Survivor区域中的相同年龄的对象占到所有对象的一半以上时，就会将大于这个年龄的对象移动到老年代
    - 堆中产生大对象超过阈值
    - 老年代连续空间不足，JVM如果判断老年代没有做足够的连续空间来放置大对象，那么就会引起Full GC
    - 在 CMS 启动过程中，新生代提升速度过快，老年代收集速度赶不上新生代提升速度。在 CMS 启动过程中，老年代碎片化严重，无法容纳新生代提升上来的大对象，这是因为CMS采用标记清理，会产生连续空间不足的情况
14. 类文件结构　类文件 {0xCAFEBABE，小版本号，大版本号，常量池大小，常量池数组，访问控制标记，当前类信息，父类信息，实现的接口个数，实现的接口信息数组，域个数，域信息数组，方法个数，方法信息数组，属性个数，属性信息数组}
15. 类的生命周期：加载、连接（验证、准备、解析）、初始化、使用、卸载
    - 加载：获取此类的二进制字节流，生成java.lang.Class对象
    - 验证：文件格式验证、元数据验证、字节码验证、符号引用验证
    - 准备：为类变量分配内存，在方法区中进行分配
    - 解析：符号引用替换为直接引用
    - 初始化：执行类构造器`<clinit>`方法的过程，由编译器收集类的类变量和静态语句块合并产生
16. Java中赋值顺序
    - 父类静态变量和执行静态代码块
    - 自身静态变量和执行静态代码块
    - 父类成员变量赋值和执行代码块
    - 父类构造方法赋值
    - 自身成员变量赋值和执行代码块
    - 自身构造方法赋值
17. 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其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。
18. 双亲委派模型
    - 概念：如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父加载器去完成，每一层次的类加载器都是如此，因此所有的加载请求最终都会传到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围内没有找到这个类）时，子加载器才会尝试自己去加载。
    - 好处：java类随着它的加载器一起具备了一种带有优先级的层次关系，对于Java程序稳定运作很重要。
    - JDK类加载器
        - 启动类加载器（Bootstrap ClassLoader）：是虚拟机的一部分，由C++实现，负责加载`<JAVA_HOME>\lib`目录中的类库

    - 扩展类加载器（Extension ClassLoader）：这个加载器由sun.misc.Launcher $ExtClassLoader 实现， 它负责加载`<JAVA_HOME>\lib\ext`目录中的类库
    - 应用程序类加载器（Application ClassLoader）：这个类加载器由`sun.misc.Launcher $App-ClassLoader`实现，负责加载用户类路径（ClassPath）上所指定的类库。
19. 方法调用
    - 解析：静态方法、私有方法、实例构造方法、父类方法在解析阶段就能确定调用
    - 分派：静态分派（重载）、动态分派（重写）
20. 编译
    - 分类与概念
        - 前端编译器：把.java文件转变成.class文件的过程
        - 后端编译器（JIT编译器，Just In Time Compiler）：把字节码转变成机器码的过程
        - 静态提前编译器（AOT编译器，Ahead Of Time Compiler）：直接把.java文件转变成本地机器码的过程
    - javac编译
        - 解析与填充符号表：词法（将字符流转变为Token）
        - 语法分析（抽象语法树）
        - 填充符号表
        - 注解处理器：编译插件，可以读取、修改、添加抽象语法树中的任意元素
        - 语义分析与字节码生成：标注检查、数据及控制流分析、解语法糖、字节码生成
    - Java语法糖
        - 泛型与类型擦除：编译之后的字节码中不存在泛型，只是在相应的地方进行强转
        - 自动装箱和拆箱：编译之后转化为对应的包装和还原方法
        - 遍历循环：编译后还原成迭代器实现
        - 变长参数：编译后变成一个数组类型的参数
21. 即时编译（JIT编译）
    - 概念：当虚拟机发现某个方法或代码块特别频繁时，就会把这些代码认定为“热点代码”，为了提高热点代码的执行效率，在运行时虚拟机会把这些代码编译成本地平台相关的机器码，并进行各种层次的优化。
    - 解释器和编译器：解释器可以在程序需要迅速启动和执行的时候发挥作用，编译器可以在程序运行后，把越来越多的代码编译成本地代码，获取更高的执行效率。
    - 编译对象与触发条件：
        - 热点代码：被多次调用的方法、被多次执行的代码块
        - 热点探测：基于采样的热点探测、基于计数器的热点探测
    - 编译优化技术
        - 公共子表达式消除
        - 数组边界检查消除
        - 方法内联：私有方法、 实例构造器、 父类方法、静态方法、final方法；虚方法的内联问题，类型继承关系分析，查询此方法在当前程序下是否有多个目标版本可供选择
        - 逃逸分析：分析对象的动态作用域，如果一个对象不会逃逸到方法或线程之外，则可以进行以下优化，栈上分配、同步消除、标量替换
22. JVM参数　堆：-Xms -Xmx -Xmn(分给新生代) -XX:SurvivorRatio=8(新生代Eden和Survivor比例)　栈：-Xss -Xoss（本地栈）　方法区：-XX：PermSize=　-XX:MaxPermSize=　老年代阀值：-XX:MaxTenuringThreshold=

23. JDK监控工具 jps：输出JVM中运行的进程状态　jstack：查看Java进程中线程的栈信息　jmap：查看进程中堆内存使用情况　jstat：是用于监视虚拟机各种运行状态信息的命令行工具
24. CPU 100%定位　1）top -c （显示进程列表），输入P，进程按照CPU使用率排序　2）top -Hp PID（显示PID中线程列表），输入P，线程按照CPU使用率排序　3）将PID转成十六进制，printf “%x\n” 10804　4）jstack 10765 | grep ‘0x2a34’ -C5 –color （查看线程栈）

> JDK 默认垃圾收集器
>
> - jdk1.7 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）
> - jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）
> - jdk1.9 默认垃圾收集器G1

内存模型

> 并发编程三要素（线程的安全性问题体现在）：
>
> 原子性：原子，即一个不可再被分割的颗粒。原子性指的是一个或多个操作要么全部执行成功要么全部执行失败。
>
> 可见性：一个线程对共享变量的修改,另一个线程能够立刻看到。（synchronized,volatile）
>
> 有序性：程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排序）
>
> 出现线程安全问题的原因：
>
> - 线程切换带来的原子性问题
> - 缓存导致的可见性问题
> - 编译优化带来的有序性问题
>
> 解决办法：
>
> - JDK Atomic开头的原子类、synchronized、LOCK，可以解决原子性问题
> - synchronized、volatile、LOCK，可以解决可见性问题
> - Happens-Before 规则可以解决有序性问题

1. Java内存模型（JMM）：是Java虚拟机所定义的一种抽象规范，通过这组规范定义了程序中各个变量（包括实例字段、静态字段和构成数组对象的元素）的访问方式。由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存，而Java内存模型中规定所有变量都存储在主内存，主内存是共享区域，所有线程都可以访问，但线程对变量的操作必须在工作内存中进行。所以需要将变量从主内存拷贝到自己工作内存，然后对变量进行操作，操作完之后再将变量写回主内存。
2. Java线程与硬件处理器：在Windows和Linux系统上，Java线程的实现是基于一对一的线程模型，就是通过语言级别层面去间接调用系统内核的线程模型。
3. Java内存间交互操作：Java内存模型中定义了8种操作来完成变量从主内存拷贝到工作内存及从工作内存同步回主内存的操作，虚拟机必须保证每一种操作都是原子的、不可分割的。lock、unlock、read、load、use、assign、store、write，拥有指令规则。
4. JMM存在的必要：为了解决线程安全问题，JVM定义了一组规则（Java内存模型），决定一个线程对共享变量的写入何时对另一个线程可见，JMM是围绕着程序执行的原子性、有序性、可见性展开的。

5. Java内存模型的三大特征
    - 原子性：是指一个操作不可中断，即使在多线程环境下，一个操作一旦开始就不会被其他线程影响。由JMM来直接保证的原子性操作包括read、load、use、assign、store、write六个，大致可以认为基础数据类型的访问和读写是具备原子性的，如果需要更大范围的原子性保证，使用lock和unlock操作，对应到代码层面就是synchronized。
    - 可见性：是指当一个线程修改了某个共享变量的值，其他线程是否能够马上得知这个修改的值。JMM依赖于刷新主内存变量的方式实现可见性。Java中volatile保证了多线程操作时变量的可见性，还有synchronized和final。工作内存与主内存同步延迟现象就造成了可见性问题。另外指令重排以及编译器优化也可能导致可见性问题。
    - 有序性：是指单线程环境下代码执行的最终结果和按顺序依次执行的结果一致，但对于多线程环境，则可能出现乱序现象。Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一个时刻只允许一条线程对其进行lock操作”这条规则获得的，这条规则决定了持有同一个锁的两个同步块只能串行地进入。
6. 指令重排：在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序。可能导致程序出现可见性问题。
    - 重排序分类
        - 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
        - 指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
        - 内存系统的重排序。由于处理器使用缓存和读 / 写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。因为三级缓存的存在，导致内存与缓存的数据同步存在时间差。
7. JMM提供的解决方案
    - 在Java内存模型中都提供一套解决方案供Java工程师在开发过程使用，如原子性问题，除了JVM自身提供的对基本数据类型读写操作的原子性外，对于方法级别或者代码块级别的原子性操作，可以使用synchronized关键字或者重入锁(ReentrantLock)或者lock或者java.util.concurrent中的Atomic原子类来保证程序执行的原子性。
    - 而工作内存与主内存同步延迟现象导致的可见性问题，可以使用synchronized关键字或者volatile关键字解决，它们都可以使一个线程修改后的变量立即对其他线程可见。
    - 对于指令重排导致的可见性问题和有序性问题，则可以利用volatile关键字解决，因为volatile的另外一个作用就是禁止重排序优化，关于volatile稍后会进一步分析。
    - 除了靠sychronized和volatile关键字来保证原子性、可见性以及有序性外，JMM内部还定义一套 happens-before 原则来保证多线程环境下两个操作间的原子性、可见性以及有序性。这个原则非常重要，它是判断数据是否存在竞争、线程是否安全的主要依据。
8. volatile内存语义
    - 保证被volatile修饰的共享变量对所有线程总是可见的，也就是当一个线程修改了一个被volatile修饰共享变量的值，新值总是可以被其他线程立即得知。
    - 禁止指令重排序优化。在指令间插入一条Memory Barrier则会告诉编译器和CPU，不管什么指令都不能和这条Memory Barrier指令重排序，也就是说通过插入内存屏障禁止在内存屏障前后的指令执行重排序优化。Memory Barrier的另外一个作用是强制刷新各种CPU的缓存数据，因此任何CPU上的线程都能读取到这些数据的最新版本。 注：JMM就是一组规则，这组规则意在解决在并发编程可能出现的线程安全问题，并提供了内置解决方案（happen-before原则）及其外部可使用的同步手段(synchronized/volatile等)，确保了程序执行在多线程环境中的应有的原子性，可视性及其有序性。

多线程

1. 线程状态：创建(new)、就绪(runnable)、运行(running)、阻塞(blocked)、等待（time waiting、waiting）、消亡（dead）
2. 创建线程的方式：实现Runnable、继承Thread、由线程池创建；new Thread()过程中会为线程设置线程组、执行目标、线程名、线程栈大小，默认和主线程一个组；Thread 实现了Runnable，本身即为一个执行目标；真正创建线程是在start()中，这里使用了本地调用，通过C代码初始化线程需要的系统资源。此时start()的这个线程处于就绪状态，当得到CPU的时间片后就会执行其中的run()方法。
3. 线程API：
    - stop：过于暴力，可以使用volatile变量指示线程退出
    - 线程中断：线程中断是一种重要的线程协作机制。线程中断不会使线程立即退出，而是给线程发一个通知，告知线程希望他退出。
    - 等待通知：无论是wait还是notify必须包含在对应的synchronized语句中，首先获取到目标对象的一个监视器。Object.wait（）和Thread.sleep()都可以让线程等待若干时间，wait可以被唤醒和释放目标对象的锁，Thread.sleep()不会释放任何资源。
    - 等待线程结束（Join）和谦让（yield）
    - 守护线程：thread.setDaemon(true)
    - 优先级：Java中使用1到10表示线程的优先级，数字越大优先级越高。
4. 同步互斥锁（synchronized）
    - 三种使用方式，修饰实例方法，对当前实例加锁；修饰静态方法，对当前Class对象加锁；修饰代码块，对指定对象加锁。通过javap可查看字节码，修饰方法通过　字节码中的flag（ACC_SYNCHRONIZED）设置加锁，修饰块通过在代码块前后加monitorenter 和monitorexit 指令。
    - synchronized加锁，每个对象都有一个monitor对象与之关联，monitor中有两个队列，一个同步队列一个等待队列，当多个线程同时访问一段代码时，没有获取到锁的线程会进入到同步队列，当线程调用了wait方法时，会释放调锁，然后进入等待队列，
    - JVM对synchronized的优化：偏向锁（一个线程获得了锁，那么锁就进入偏向模式，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程）、轻量级锁（轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”）、自旋锁、锁消除
5. UnSafe（指针类）、CAS（比较交换）和AQS（队列同步器）：
    - UnSafe：allocateMemory、reallocateMemory、freeMemory用于分配内存、扩展内存、释放内存，UnSafe可以定位对象属性的内存位置
    - CAS：依托于UnSafe类，通过比较预期值域内存位置上的值，相等则设置为新值，否则不操作；Atomic系列就是基于CAS保证操作的原子性，循环读取值与内存位置上的值比较，直到相等然后设置新值
    - AQS：用于构建锁或其他同步组件的基础框架，通过一个volatile int state控制同步状态，当state=0时表示没有任何线程占用共享资源，当state>0表示当前有线程占用共享资源，其他线程需要进入同步队列进行等待。AQS中通过内部类Node构建一个双向队列实现FIFO的同步队列完成线程获取锁的排队工作，通过ConditionObject构建一个等待队列，当线程调用await之后线程会加入等待队列，当其他线程调用signal之后线程将从等待队列移到同步队列进行锁竞争。AQS作为基础组件，对于锁的实现有两种不同的模式，共享模式（Semaphore）和独占模式（ReentrantLock），不管是共享模式还是独占模式都是基于AQS实现的，内部都维持着一个同步队列和多个等待队列。
6. JUC并发包：ReentrantLock、Condition、Semaphore、CountDownLatch 、CyclicBarrier
    - ReentrantLock：基于AQS实现的独占重入锁，同一个线程可以多次获取同一把锁，释放相同的次数，灵活，多种获取锁方式。
    - Condition：可以让线程在合适的时间地点进行等待，或者在特定的时刻通知线程继续执行

- Semaphore：信号量用于控制访问公共资源的线程数量，用于流量控制；多个线程间还是存在并发问题
- CountDownLatch：闭锁，一种同步的方法，延迟线程进度直到线程到达某个终点状态，只能使用一次，基于AQS
- CyclicBarrier：循环栅栏，一个同步辅助类，允许一组线程相互等待，直到达到某个公共屏障点，内部通过一个ReentrantLock和Condition进行计算等待

7. 线程池
   - 线程过多对性能的影响：会耗尽CPU和内存资源，创建和关闭线程需要花费过多时间，大量线程回收给GC带来压力，延长GC停顿时间
   - 创建线程池：JDK实现的线程池ThreadPoolExecutor，corePoolSize指定线程池中的线程数量，maxPoolSize指定线程池中最大的线程数量，keepAliveTime指定多余线程存活时长，timeUnit制动keepAliveTime单位，workQueue任务队列，threadFactory线程工厂，handle拒绝策略
   - 任务队列的几种实现：SynchronousQueue（没有容量，总是将任务交给线程执行，没有线程则新建，大于最大线程数，则执行拒绝）、ArrayBlockingQueue（有界队列，任务大于corePoolSize则将任务加入任务队列，若队列已满，则创建线程，若线程数量大于maxPoolSize则执行拒绝）、LinkedBlockingQueue（无界队列，任务大于corePoolSize加入任务队列，直到资源耗尽）、PriorityBlockingQueue（优先队列）
   - 拒绝策略：AbortPolicy（直接抛异常）、CallerRunsPolicy（调用线程运行被丢弃任务）、DiscardOledestPolicy（丢弃最老任务）、DiscardPolicy（丢弃无法处理的任务）
   - 使用线程池的优势： 提高资源利用率：线程池可以重复利用已经创建好的线程，减少线程创建和销毁的巨大开销　　提高响应速度：线程池中有待分配的线程时，当任务到来时，无需创建线程就能执行
   - 具有可管理行：线程池会根据当前系统的特点对线程池内的线程进行优化处理，减少创建和销毁线程带来的系统开销
   - 线程池执行原理：创建线程池时不会一开始就创建线程，添加任务的时候才开始创建线程。线程池执行任务时，先获取线程池中的线程数量，当线程数量小于coreSize时，　　会将任务通过addWorker直接调度任务（Worker包装了线程创建和调度执行的过程），否则通过workQueue.offer()进入等待队列，如果进入等待队列失败，则把任务交给　　线程池，如果线程池中的线程数量已经到达了最大值，则执行解决策略。当worker执行完当前任务会从任务队列中取出任务继续执行，从阻塞队列中取任务，取任务有三种情况发生：1.取到任务并返回任务；2.没有取到任务，返回null,这个工作线程被回收；3.没有取到任务，阻塞在向阻塞队列取任务这里，第三点就是线程池中的空闲任务是如何存在的　- 任务一般可分为：CPU密集型、IO密集型、混合型，对于不同类型的任务需要分配不同大小的线程池。　　1）CPU密集型任务 尽量使用较小的线程池，一般为CPU核心数+1　　2）IO密集型任务 可以使用稍大的线程池，一般为`2*CPU核心数`　　3）混合型任务 可以将任务分成IO密集型和CPU密集型任务，然后分别用不同的线程池去处理

TCP/IP

> 1. TCP/IP Socket：TCP协议是面向连接的、保证高可靠性的传输协议、IP协议是无连接、不可靠的网络协议
> 2. 长连接与短连接：
>   - 所谓长连接，指在一个TCP连接上可以连续发送多个数据包，在TCP连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接，一般需要自己做在线维持。连接→数据传输→保持连接(心跳)→数据传输→保持连接(心跳)→……→关闭连接。
>   - 短连接是指通信双方有数据交互时，就建立一个TCP连接，数据发送完成后，则断开此TCP连接，一般银行都使用短连接。 连接→数据传输→关闭连接。

> - http 1.0 短连接，http 1.1 长连接，长连接短连接实际说的还是TCP，所谓的长连接就是为了复用TCP连接，例如：加载网页、css、js、图片等
> - 长连接并不是永久连接的。如果一段时间内，这个连接没有HTTP请求发出的话，那么这个长连接就会被断掉。

1. 常规应用的四层网络分层：应用层、传输层、网络层、链路层
2. Socket是在应用层和传输层之间的一个抽象层，封装了传输层的复杂操作，提供简单易用的API给应用层使用
3. TCP三次握手连接，客户端向服务端发送一个syn包（syn=j）之后进入SYN_SEND 状态；服务端收到客户端的syn包之后，同时向客户端发送一个ack包（ack=j+1）和一个syn包（syn=k）之后进入SYN_RECV状态；客户端收到服务端的确认包和同步包之后，再向服务端发送一个ack包（ack=k+1）之后进入ESTABLISHED（确立）状态；服务端收到ack包之后也进入确立状态，完成三次握手连接。
4. TCP四层挥手断开：客户端会先发一个FIN包通知服务端请求关闭连接，服务端会发一个确认ack包给客户端；当服务端运行一段时间后会发送一个FIN包给客户端请求断开连接，客户端收到服务端请求之后会发一个确认包给服务端，表示收到服务端请求。
5. TCP可靠性的保证是采用“带重传的肯定确认”机制，结合“滑动窗口”机制提高网络吞吐量。带重传的肯定确认机制，发送方发送一份带序号的记录同时设置超时时间，超时重新发送，等待接收方返回一个确认信号和一个序号（期待的下一份记录序号）再发送下一份记录。滑动窗口技术，发送方维护这一个接收方的窗口大小，一次发送一组记录，接收方每次确认时会带上自己的窗口大小，发送方根据这个返回值动态调节发送。
6. TCP 粘包、拆包原因
    - Socket缓冲区和滑动窗口：每个socket在内核中都有一个发送缓冲区（SO_SNDBUF）和接收缓冲区（SO_RCVBUF）
    - MSSMTU限制：MSS（TCP报文中data部分的最大长度）、MTU（链路层一次发送最大数据限制）。在IPV4中，以太网MSS可以达到1460byte；在IPV6中，以太网MSS可以达到1440byte。
    - Nagle 算法
7. TCP 粘包、拆包问题解决：可以通过定义应用的协议(protocol)来解决，协议的作用就定义传输数据的格式。
8. 网络模型中应用层通过TCP发送数据的流程
    - 对于应用层来说，只关心发送的数据DATA，将数据写入socket在内核中的缓冲区SO_SNDBUF即返回，操作系统会将SO_SNDBUF中的数据取出来进行发送。
    - 传输层会在DATA前面加上TCP Header,构成一个完整的TCP报文。
    - 当数据到达网络层(network layer)时，网络层会在TCP报文的基础上再添加一个IP Header，也就是将自己的网络地址加入到报文中。
    - 到数据链路层时，还会加上Datalink Header和CRC。
    - 当到达物理层时，会将SMAC(Source Machine，数据发送方的MAC地址)，DMAC(Destination Machine，数据接受方的MAC地址)和Type域加入。
9. TCP设置参数
    - SO_KEEPALIVE：是否启用心跳机制,默认false。
    - SO_SNDBUF：发送缓冲区的大小设置，默认为8K。
    - SO_RCVBUF：接收缓冲区大小设置，默认为8K。
    - TCP_NODELAY：是否一有数据就马上发送，默认为false，设置为true关闭Nagle算法。
    - SO_BACKLOG：标识当服务器请求处理线程全满时，用于临时存放已完成三次握手的请求的队列的最大长度，Java将使用默认值50，netty默认取值为3072。

HTTP

1. 代理服务器：代替网络用户去取得网络信息，提高访问速度、防火墙、突破访问限制
    - 正向代理：一个位于客户端和原始服务器之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并制定目标（原始服务器），然后代理向原始服务器转发请求并将获得的内容返回给客户端，客户端才能使用正向代理。我们平时说的代理就是指正向代理。
    - 反向代理：以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求的客户端，此时代理服务器对外表现为一个反向代理服务器。 - 区别：位置不同，正向代理在客户机与目标主机之间，反向代理在服务器端；代理对象不同，正向代理，代理客户端，服务端不知道实际发起请求的客户端，反向代理，代理服务端，客户端不知道实际提供服务的服务端。
2. HTTP是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP是一个无状态的协议。HTTP协议通常承载于TCP协议之上。
3. HTTP1.0、HTTP1.1、HTTP2.0区别 3.1、HTTP1.0 - 无状态、无连接 3.2、HTTP1.1：不允许同时存在两个并行的响应 - 持久连接 - 请求管道化 - 增加缓存处理（新的字段如cache-control） - 增加Host字段、支持断点传输等（把文件分成几部分） 3.3、HTTP2.0 - 二进制分帧 - 多路复用（或连接共享） - 头部压缩 - 服务器推送

IO模型及NIO

1. 同步与异步关注的是消息的通信机制
    - 同步：发出一个调用时，在没有得到结果之前，该调用就不返回，由调用者主动等待调用结果。
    - 异步：发出一个调用时，这个调用直接返回，所以没有返回结果，而是由被调用者通过状态、消息来通知调用者，或者通过回调函数处理这个调用。
2. 阻塞与非阻塞关注的是程序在等待调用结果（消息、返回值）时的状态
    - 阻塞：阻塞调用是指调用结果返回之前，当前线程会被挂起，调用线程只有在得到调用结果之后才会返回。
    - 非阻塞：非阻塞调用是指在不能立即得到结果之前，该调用不会阻塞当前线程。
3. IO模型：一个IO操作分为两个步骤：发起IO请求和实际的IO操作，同步IO和异步IO的区别就在于第二个步骤是否阻塞，阻塞IO和非阻塞IO的区别在于第一个步骤，发起IO请求是否会阻塞
    - 阻塞IO模型：系统调用直到数据报到达且被拷贝到应用进程的缓冲区中或者发生错误才返回，期间一直在等待，整段时间都是在阻塞的
    - 非阻塞IO模型：当所请求的I/O操作不能满足要求时候，不把本进程投入睡眠，而是返回一个错误。也就是说当数据没有到达时并不等待，而是以一个错误返回。
    - IO复用模型：进程通过将一个或多个fd传递给select或poll系统调用，阻塞在select;这样select/poll可以帮我们侦测许多fd是否就绪，当有fd就绪时，立即回调函数rollback
    - 信号驱动异步IO模型：
    - 异步IO模型：告知内核启动某个操作，并让内核在整个操作完成后(包括将数据从内核拷贝到用户自己的缓冲区)通知我们。

> 注：BIO的局限性：server端应该使用尽可能少的线程，来处理尽可能多的client请求。IO模型中读取数据必须经过两个阶段：1）等待数据准备（等待时间可能无限长）；2）将准备好的数据从内核空间拷贝到用户空间（拷贝过程实际很短）。IO复用模型就是针对两阶段进行区分处理：1）用一个专门的线程负责第一阶段，检查哪些client准备好了数据，然后将这些client过滤出来交给

> worker线程处理；2）worker线程只负责第二阶段的数据拷贝，时间很短。

4. AIO，BIO，NIO
    - AIO异步非阻塞IO，AIO方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。
    - NIO同步非阻塞IO，适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
    - BIO同步阻塞IO，适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
5. epoll,select/poll　都是IO复用，I/O多路复用就通过一种机制，可以通过一个线程监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。epoll的效率更高，优化了select的轮询操作，通过callback事件响应方式。
    - select：每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大，同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大，select支持的文件描述符数量太小了，默认是1024
    - poll：poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd_set结构.
    - epoll：epoll是对select和poll的改进，epoll保证了每个fd在整个过程中只会拷贝一次，为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表，epoll没有限制描述符的数量，和系统内存关系比较大
6. 不建议使用NIO的原因
    - NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。
    - 需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序。
    - 可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，IO编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。
    - JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。
7. NIO 非阻塞IO，一种基于通道和缓冲区的IO方式。NIO 有三大核心组件，Channel（通道）、Buffer（缓冲区）、Selector（选择器）。数据总是从通道读取到缓冲区或者是从缓冲区写入通道，Selector用来监听多个通道的事件，因此单个线程可以监听多个数据通道。一个通道创建之后在Selector上注册感兴趣的事件，通过调用Selector的select方法返回一个准备就绪的SelectionKey集合，SelectionKey中包含对这个事件感兴趣的Channel。

Netty

> Netty是一个高性能、异步事件驱动的NIO网络框架，具有高并发、传输快、封装好等特点。

1. Netty的Reactor线程模型：mainReactor负责处理客户端的连接请求，并将accept的连接注册到subReactor中的一个线程中；subReactor负责监听理客户端通道上的事件并完成数据读写；ThreadPool 负责处理具体的业务逻辑。 NioEventLoop实际上就是工作线程，可以直接理解为一个线程。NioEventLoopGroup是一个线程池，线程池中的线程就是NioEventLoop；每个NioEventLoop都绑定了一个Selector，所以在Netty的线程模型中，是由多个Selecotr在监听IO就绪事件。而Channel注册到Selector；一个Channel绑定一个

NioEventLoop，相当于一个连接绑定一个线程，这个连接所有的ChannelHandler都是在一个线程中执行的，避免了多线程干扰。更重要的是ChannelPipline链表必须严格按照顺序执行的。单线程的设计能够保证ChannelHandler的顺序执行。一个NioEventLoop的selector可以被多个Channel注册，也就是说多个Channel共享一个EventLoop。

2. Netty服务启动之前调用group、channel，handler、option、attr、childHandler、childOption、childAttr设置启动参数
3. Netty通过编写多个ChannelHandler处理不同的功能，通过ChannelPipeline来保证ChannelHandler的处理顺序。每个Channel创建的时候都会关联一个ChannelPipeline， ChannelHandler通过封装成ChannelHandlerContext链表结构添加到ChannelPipeline中以此来保证ChannelHandler的执行顺序。
4. 调用顺序：先是调用ChannelPipeline中的fireXXX方法，接着是ChannelHandlerContext中的fireXXX方法，最后是ChannelHandler中的XXX方法
5. 选择Netty的原因：Netty是业界最流行的NIO框架之一，它的健壮性、功能、性能、可定制性和可扩展性在同类框架中都是首屈一指的 1）API使用简单，开发门槛低； 2）功能强大，预置了多种编解码功能，支持多种主流协议； 3）定制能力强，可以通过ChannelHandler对通信框架进行灵活的扩展； 4）性能高，通过与其它业界主流的NIO框架对比，Netty的综合性能最优； 5）成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼 6）社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会被加入；

> Netty附：
>
> 1. Selector BUG出现的原因 若Selector的轮询结果为空，也没有wakeup或新消息处理，则发生空轮询，CPU使用率100%。Netty的解决办法：对Selector的select操作周期进行统计，每完成一次空的select操作进行一次计数，若在某个周期内连续发生N次空轮询，则触发了epoll死循环bug，重建Selector，判断是否是其他线程发起的重建请求，若不是则将原SocketChannel从旧的Selector上去除注册，重新注册到新的Selector上，并将原来的Selector关闭。
> 2. Netty心跳
>   - 心跳其实就是一个简单的请求，对于服务端：会定时清除闲置会话；对于客户端:用来检测会话是否断开，是否重来，检测网络延迟
>   - idleStateHandler类 用来检测会话状态
> 3. 无锁化的串行设计理念
>   - 通过串行化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。
>   - 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-多个工作线程模型性能更优。
> 4. Netty的可靠性
>   - 链路有效性检测：链路空闲检测机制：读/写空闲超时机制
>   - 内存保护机制：通过内存池重用ByteBuf;ByteBuf的解码保护
>   - 优雅停机：不再接收新消息、退出前的预处理操作、资源的释放操作
> 5. Netty安全性
>   - Netty支持的安全协议：SSL V2和V3，TLS，SSL单向认证、双向认证和第三方CA认证。
>   - SSL的三种认证方式
> 6. Netty的高效并发编程主要体现在如下几点：
>   1. volatile的大量、正确使用;
>   2. CAS和原子类的广泛使用；

3. 线程安全容器的使用；
4. 通过读写锁提升并发性能。

7. Netty除了使用reactor来提升性能，当然还有　1）零拷贝，IO性能优化　2）通信上的粘包拆包　2）同步的设计　3）高性能的序列
8. Netty实现同步调用的实现方案
    - 请求与响应的正确匹配：通过客户端唯一的RequestId，服务端返回的响应中需要包含改RequestId，
    - 请求线程和响应线程间的通信：请求线程会在请求发出之后同步等待服务端的返回，需要解决的问题是接到响应之后如何通知请求线程　解决方案：利用Java中的CountDownLatch类来实现同步Future。具体过程是：客户端发送请求后将<请求ID，Future>的键值对保存到一个全局的Map中，这时候用Future等待结果，挂住请求线程；当Netty收到服务端的响应后，响应线程根据请求ID到全局Map中取出Future，然后设置响应结果到Future中。这个时候利用CountDownLatch的通知机制，通知请求线程。请求线程从Future中拿到响应结果，然后做业务处理。然后从全局Map中移除请求ID。Dubbo中是通过ReentrantLock、Condition实现通知。
9. netty客户端重连实现 心跳是用来检测一个系统是否存活或者网络链路是否通畅的一种方式，一般的做法是客户端定时向服务端发送心跳包，服务端收到心跳包后进行回复，客户端收到回复说明服务端存活。 在Netty中提供了一个IdleStateHandler类用于心跳检测，通过handler中的userEventTriggered接收心跳检测结果，如果触发了IdleState.READER_IDLE事件就说明服务端没有给客户端响应，这个时候可以选择重新连接。启动时连接重试增加一个负责重试逻辑的监听器；运行中连接断开时重试，在handler的channelInactive 中重试。
10. 内存池 为了降低JVM的GC压力，降低频繁申请堆外内存造成的性能损耗，设计一个内存池模块。该模块事先申请了一大块的连续内存用于分配。令后续系统中所有的内存空间请求均需要从内存池进行申请，使用完毕后归还给内存池。通过内存池，将这部分内存管理工作自己执行，也减少了这部分的GC损耗。
11. 直接内存释放 DirectByteBuffer本身是一个Java对象，其是位于堆内存中的，JDK的GC机制可以自动帮我们回收，但是其申请的直接内存，不再GC范围之内，无法自动回收。好在JDK提供了一种机制，可以为堆内存对象注册一个钩子函数(其实就是实现Runnable接口的子类)，当堆内存对象被GC回收的时候，会回调run方法，我们可以在这个方法中执行释放DirectByteBuffer引用的直接内存，即在run方法中调用Unsafe 的freeMemory 方法。注册是通过sun.misc.Cleaner类来实现的。

zookeeper

1. Zookeeper是一个高效的分布式协调服务，Zookeeper集群中节点个数一般为奇数个（>=3），若集群中Master挂掉，剩余节点个数在半数以上时，就可以推举新的主节点，继续对外提供服务。客户端发起事务请求，事务请求的结果在整个Zookeeper集群中所有机器上的应用情况是一致的。在Zookeeper集群中的任何一台机器，其看到的服务器的数据模型是一致的。
2. Zookeeper使用的数据结构为树形结构，根节点为"/"。Zookeeper集群中的节点，根据其身份特性分为leader、follower、observer。leader负责客户端writer类型的请求；follower负责客户端reader类型的请求，并参与leader选举；observer是特殊的follower，可以接收客户端reader请求，但是不会参与选举，可以用来扩容系统支撑能力， 提高读取速度。

3. Zookeeper是一个基于观察者模式设计的分布式服务管理框架，负责存储和管理相关数据，接收观察者的注册。一旦这些数据的状态发生变化，zookeeper就负责通知那些已经在zookeeper集群进行注册并关心这些状态发生变化的观察者，以便观察者执行相关操作。
4. Zookeeper使用的是ZAB原子消息广播协议，节点之间的一致性算法为Paxos，能够保障分布式环境中数据的一致性。分布式场景下高可用是Zookeeper的特性，可以采用第三方客户端的实现，即Curator框架。
5. Paxos 算法解决的问题是一个分布式系统如何就某个值（决议）达成一致。Paxos 算法就是一种基于消息传递模型的一致性算法。
    - 四种类型的znode：　1)PERSISTENT-持久化目录节点　2)PERSISTENT_SEQUENTIAL-持久化顺序编号目录节点　3)EPHEMERAL-临时目录节点　4)EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点
    - 通知机制：客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。
    - zookeeper master选举：当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的 Server都恢复到一个正确的状态。　1)选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；　2)选举线程首先向所有Server发起一次询问(包括自己)；　3)选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；　4)收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；　5) 线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数，设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。
6. zookeeper如何感应节点挂了　Zookeeper客户端和服务端维持一个长连接，每隔10s向服务端发送一个心跳，服务端返回客户端一个响应。
    - 客户端在节点 x 上注册一个Watcher，那么如果 x 的子节点变化了，会通知该客户端。
    - 创建EPHEMERAL类型的节点，一旦客户端和服务器的会话结束或过期，那么该节点就会消失。

Dubbo

1. dubbo中用到的设计模式：工厂模式、装饰器模式、责任链模式、观察者模式、动态代理模式
2. dubbo中的注解 - SPI（标识一个扩展点，接口上）
    - Adaptive：一个接口有多个实现，具体调用哪个实现的方法，由配置指定，所以需要一个适配类来指定具体是哪个实现类的方法，写死的适配类在类上加上注解@Adaptive，如果是动态适配的就需要动态生成适配类。
    - Activate：标识一个扩展是否被激活和使用，可以放在定义的类上和方法上，它有两个设置锅里条件的字段，group,value 都是字符数组。用来指定这个扩展类在什么条件下激活。
3. dubbo中的插件机制：每个组件接口都有一个ExtensionLoader负责加载、存储该组件的扩展点，有三种加载方式：名称加载扩展点、加载激活扩展点（filter）、加载自适应扩展点
4. 服务提供者初始化，dubbo服务提供者在ServiceConfig的export方法中开始暴露服务，默认服务端口默认20880；
    - 根据配置生成URL对象，URL是dubbo中的核心模型，有两种URL（注册中心URL、服务URL）；

    - 生成本地服务代理，scope没有设置成remote，服务同时会在本地暴露，生成一个本地服务代理对象，本地调用dubbo接口时直接调用本地代理不走网络请求
    - 生成远程服务代理，它会生成一个Invoker，该Invoker是服务ref（实现类）的一个代理包含了携带服务信息的URL对象，Invoker的invoke方法被调用时最终调到ref指定的服务实现，Invoker创建之后接着由Protocol组件来创建Exporter，这里有RegistryProtocol创建会包装进Listener和Filter，在RegistryProtocol.doLocalExport执行暴露逻辑，真正执行服务暴露的是DubboProtocol，这里启动服务监听，再回到RegistryProtocol中完成把服务注册到注册中心，提供者会监听configurators节点，感知注册中心的配置变化。
5. 服务消费者初始化，ReferenceBean是一个FactoryBean，在init方法中开始初始化消费者，RegistryProtocol.refer方法中先把消费者注册到注册中心，然后监听providers、configurators、routers节点，消费端会本地缓存远程服务提供者、注册中心配置、路由配置信息，消除消费端对注册中心的强依赖，在订阅的时候最终调到RegistryDirector的notify方法、refreshInvoker、toInvokers等。DubboProtocol.refer中创建一个DubboInvoker对象，该Invoker对象持有服务Class、providerUrl、负责和提供端通信的ExchangeClient，Invoker对象保存在DubboProtocol的invokers集合中，构建DubboInvoker时会创建一个或多个ExchangeClient用来处理和提供端的连接，默认一个URL只会创建一个ExchangeClient，由connections参数控制，RegistryProtocol.doRefer方法的最后，由Cluster组件来创建一个Invoker并返回一个扩展链，默认failover，最后生成消费端服务代理。
6. 远程服务调用过程（InternalEncoder extends MessageToByteEncoder，InternalDecoder extends ByteToMessageDecoder编解码）
    - 消费者调用接口方法：入口是InvokerInvocationHandler的invoke方法，是否执行mock，加入负载均衡和路由、容错策略，然后由DubboInvoker执行，判断执行方法单向、异步、同步（通过锁进行等待，当超时或有结果返回时通知，通过唯一ID实现请求与返回对应），消费者请求编码->提供者请求解码->提供者处理请求->提供者响应结果编码->消费者响应结果解码 - 提供者接收请求：触发Netty Channel中的channelRead，调用顺序MultiMessageHandler、HeartbeatHandler、DispatcherHandler（默认AllChannelHandler）、DecodeHandler 、HeaderExchangeHandler、DubboProtocol（匿名内部类ExchangeHandlerAdapter，在reply方法中调用invoker的invoke方法），在AllChannelHandler中启动一个ChannelEventRunnable任务提交给线程池。
7. 请求编码：16个字节的固定消息头，其中包括2字节魔数，第三字节指定请求还是响应和序列化方式，八个字节的请求ID，4字节请求数据长度；响应编码：16字节数据头，2字节魔数，1字节序列化方式，1字节响应状态，8字节响应ID和请求ID一致，4字节消息长度
8. 集群容错模式:Failover 、Failfast 、Failsafe、 Failback、Forking
9. dubbo负载均衡策略：随机、轮循、最少活跃调用数、一致性Hash
10. 服务端调度策略
    - all 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
    - direct 所有消息都不派发到线程池，全部在 IO 线程上直接执行。
    - message 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
    - execution 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
    - connection 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。
11. 服务端线程池
    - fixed 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)
    - cached 缓存线程池，空闲一分钟自动删除，需要时重建。

- limited 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。
- eager 优先创建Worker线程池。

> 附：
>
> 1. wrapper装饰器：dubbo中的一种扩展，一个接口类型type的所有实现类中，如果构造器只有一个参数且这个参数类型为type，那么会被作为一个装饰器，在一个扩展实现被加载时，该扩展接口对应的多个装饰器都会被无序的装饰到这个扩展实现上。 如：ProtocolFilterWrapper(Protocol protocol)、ProtocolListenerWrapper(Protocol protocol)，加载Protocol的时候就会被加载到
> 2. Protocol 暴露接口时，取Protocol实例是根据Protocol的适配类Protocol$Adpative（动态生成）中的String extName = (url.getProtocol() == null ? "dubbo" : url.getProtocol());来获取的，即注册时的URL是registry协议，所以RegistryProtocol，暴露时的协议是dubbo，所以此时是DubboProtocol。
> 3. dubbo客户端和dubbo服务端之间存在心跳，由dubbo客户端主动发起，在连接时创建HeaderExchangeClient会启动一个HeartBeatTask任务发送心跳和重连。
> 4. 在 Dubbo 的核心领域模型中： - Protocol 是服务域，它是 Invoker 暴露和引用的主功能入口，它负责 Invoker 的生命周期管理。 - Invoker 是实体域，它是Dubbo的核心模型，其它模型都向它靠拢，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。 - Invocation 是会话域，它持有调用过程中的变量，比如方法名，参数等。
> 5. Transport 层只负责单向消息传输，是对 Mina, Netty, Grizzly 的抽象， Exchange 层是在传输层之上封装了 Request-Response 语义。
> 6. dubbo为什么比http效率要高
>   - 请求头较小，没有额外信息。
>   - http的编解码工作由http服务器做一层编解码，再由我们的应用服务器做一次编解码（如json）才到我们的jvm之中。而dubbo这种一般不需要二次编码，直接编码二进制，然后传输。
> 7. dubbo中的分层：Service->Config->Proxy->Registry->Cluster->Monitor->Protocol->Exchange->Transport->Serialize
> 8. dubbo和netty结合：由NettyServerHandler和NettyClientHandler实现ChannelDuplexHandler，后面有Netty Channel触发dubbo Channel中的方法
> 9. dubbo常用性能调优参数
>   - threads：provider、200、业务处理线程池大小
>   - iothreads：provider、CPU+1、IO线程池大小
>   - queues：provider、0、线程池队列大小，建议不要设置
>   - accepts：provider、0（表示不限制）、服务提供方最大可连接数
>   - executes：provider、0（表示不限制）、服务提供者每服务每方法最大可并行执行请求数
>   - actives：consumer、0（表示不限制）、每服务消费者每服务每方法最大并发调用数
>   - connections：consumer、0（默认共享一个长连接）、对每个提供者的最大连接数
> 10. Dubbo中缓存策略
>   - lru 基于最近最少使用原则删除多余缓存，保持最热的数据被缓存。

> - hreadlocal 当前线程缓存
> - jcache 与JSR107集成，可以桥接各种缓存实现。

tomcat

1. Tomcat的最顶层是一个Server，代表着整个服务，控制着整个tomcat的生命周期，一个Server包含至少一个Service，用于提供具体的服务。
2. Service主要包含两个部分，Connector和Container，Connector负责处理网络连接相关的事情，并提供Socket和Request和Response的转换，Container负责封装和管理Servlet，以及处理Request请求，一个Service只有一个Container（Engine），但是可以有多个Connector，这是因为一个服务可以处理多个连接；Engine下可以包含多个Host，Host又可以包含多个Context，Context下包含多个Wrapper（封装这Servlet）
3. Connector使用ProtocolHandler来处理网络请求的，包含三个部件，EndPoint、Processor、Adapter，Endpoint用来处理底层的Socket网络连接，实现的是TCP/IP协议； Processor用于将连接接收到的socket封装成Request，实现的是Http协议；Adapter用于将Request交给Container具体处理。
4. EndPoint中有两个内部类，Acceptor和Poller，Acceptor主要负责监听网络连接且进行任务分发的后台线程，Acceptor接收网络请求然后将socket交给Poller，Poller负责 执行数据的读取和业务处理。 5.Container处理请求使用的是Pipeline-Valve管道来处理的,责任链模式是指一个请求处理过程中有很多处理者依次对请求进行处理，每一个处理者负责做自己相应的处理，处理完之后将请求返回，再让下一个处理者继续处理。
5. tomcat类加载器
   - tomcat类加载器需要解决的几个问题：为了避免冲突，需要为每个webapp下的项目各自使用的类库建立隔离机制；不同的webapp项目支持共享类库；类加载器需要支持热插拔功能；
   - tomcat最重要的是Common类加载器，他的父级加载器是应用类加载器，负责加载 `${catalina.base}/lib、${catalina.home}/lib` 目录下面所有的 .jar 文件和 .class 文件。有commonLoader、catalinaLoader、sharedLoader三类加载器，都属于同一个实例。
   - webapp类加载器：tomcat设计了为每个webapp下的项目单独使用一个WebappClassLoader类加载器，tomcat8使用的是ParallelWebappClassLoader，支持并行加载类，start过程会读取我们熟悉的 /WEB-INF/classes、/WEB-INF/lib 资源。 - WebappLoader：单独的类加载器是无法获取webapp的资源信息的，因此tomcat引入了WebappLoader，便于访问Context组件的信息，同时为Context提供类加载的能力支持设置reloadable 属性为true支持热部署 - tomcat启动时ClassLoader加载流程 1）tomcat启动时调用System ClassLoader即AppClassLoader加载 `${catalina.home}/bin` 里面的jar，就是tomcat启动相关的jar 2）tomcat启动类Bootstrap中有三个ClassLoader属性，catalinaClassLoader、commonClassLoader、sharedClassLoader，他们都是同一个实例，用来加载${catalina.home}/lib下的jar 3）一个Context容器代表一个app应用，Context->WebappLoader->WebClassLoader,并且Thread.contextClassLoader=WebClassLoader。应用程序中的jsp文件、class类、lib/*.jar包，都是WebClassLoader加载的。
   - 当Jsp文件修改的时候，Tomcat更新步骤： 1)当访问1.jsp的时候，1.jsp的包装类JspServletWrapper会去比较1.jsp文件最新修改时间和上次的修改时间，以此判断1.jsp是否修改过。 2）1.jsp修改过的话，那么jspservletWrapper会清除相关引用，包括1.jsp编译后的servlet实例和加载这个servlet的JasperLoader实例。 3）重新创建一个JasperLoader实例，重新加载修改过后的1.jsp，重新生成一个Servlet实例。 4）返回修改后的1.jsp内容给用户。

十、Spring

1. Spring IOC和AOP概念
    - IOC（Inversion of Control控制反转）容器：就是具有依赖注入功能的容器，是可以创建对象的容器，IOC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。通常new一个实例，控制权由程序员控制，而"控制反转"是指new实例工作不由程序员来做而是交给Spring容器来做。在Spring中BeanFactory是IOC容器的实际代表者。IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。
    - DI(依赖注入Dependency injection)：在容器创建对象后，处理对象的依赖关系。

      通常有三种注入方式：构造方法注入，setter注入，基于注解的注入。

      > AOP是Spring框架面向切面的编程思想，AOP采用一种称为“横切”的技术，将涉及多业务流程的通用功能抽取并单独封装，形成独立的切面，在合适的时机将这些切面横向切入到业务流程指定的位置中。
2. Spring bean的作用域：singleton、prototype、request、session、global
3. Spring框架中常见的设计模式：工程方法、单例模式、适配器模式、装饰器模式、代理模式、观察者模式、模板方法
4. Spring的优点：低侵入性，降低了组件之间的耦合性，实现了软件各层之间的解耦；提供了AOP技术，利用它很容易实现如权限拦截，运行期监控等功能；可以使用容器提供的众多服务，如事务管理，消息服务等；spring对于主流的应用框架提供了集成支持；
5. BeanFactory和ApplicationContext是Spring的两大核心接口
    - BeanFactory：是Spring里面最底层的接口，提供了最简单的容器的功能，负责读取bean配置文档，管理bean的加载与实例化，维护bean之间的依赖关系，负责bean的生命周期，但是无法支持spring的aop功能和web应用。
    - ApplicationContext接口作为BeanFactory的子类，因而具有BeanFactory所有的功能。以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，而且ApplicationContext还在功能上做了扩展，如默认初始化所有的Singleton bean，支持国际化，事件机制等。
6. Spring容器初始化过程
    - 容器预先准备，记录容器启动时间和标记
    - 创建BeanFactory，默认DefaultListableBeanFactory
    - 加载配置文件，由ResourceLoader加载配置文件
    - 读取并解析配置文件，由BeanDefinitionReader读入配置文件，如果是普通的bean直接解析并注册到容器（ConcurrentHashMap）中，如果是其他命名空间的bean，则找到对应的命名空间处理类，再由命名空间处理类找到对应的解析器，由对应的解析器解析并完成相应的操作，如扫描由注解定义的bean，所有的bean都是构造成BeanDefinition结构注册到容器 - 配置BeanFactory的标准上下文特性，如类加载器、PostProcessor等
    - 模板方法，在bean定义被装载后，提供一个修改容器的入口，如Spring Boot中这里实现注解定义的Bean的扫描工作
    - 在bean未开始实例化时，对BeanDefinition的修改入口，常见的PropertyPlaceholderConfigurer就是在这里调用的
    - 注册用于拦截bean创建过程的BeanPostProcessor
    - 初始化MessageResource
    - 初始化上下文事件广播
    - onRefresh模板方法，Spring Boot中这里创建内嵌Tomcat

- 注册监听器
- 完成容器的初始化，并实例化为设置懒加载的单例bean

7. Spring bean的生命周期：
    - Spring 对bean 进行实例化，默认单例
    - Spring 对bean 进行依赖注入
    - 如果Bean实现了BeanNameAware接口，Spring将Bean的ID传给setBeanName（）方法
    - 如果Bean实现了BeanFactoryAware接口，Spring将BeanFactory传给setBeanFactory方法
    - 如果Bean实现了ApplicationContextAware接口，Spring将applicationContext传给setApplicationContext方法
    - 如果Bean实现了BeanPostProcessor接口，它的postProcessBeforeInitialization方法将被调用
    - 如果Bean实现了InitializingBean接口，它的afterPropertiesSet方法将被调用，如果有init-method，该方法也会被调用
    - 如果Bean实现了BeanPostProcessor接口，它的postProcessAfterInitialization方法将被调用
    - 此时Bean已经准备就绪，可以被使用，一直存在应用上下文中，直到应用上下文被销毁
    - 若Bean实现DisposableBean接口，Spring将调用它的distroy方法，和distroy-method方法
8. 常见的ApplicationContext实现
    - FileSystemXMLApplicationContext：从文件绝对路径加载配置文件
    - ClassPathXMLApplicationContext：从classpath加载配置文件
    - XMLWebApplicationContext：web中使用
    - AnnotationConfigApplicationContext：Spring Boot非Web项目使用
    - AnnotationConfigEmbeddedWebApplicationContext：Spring Boot Web项目中使用
9. Spring MVC 初始化
    - DispatcherServletAutoConfiguration中实例化DispatcherServlet，从父类HttpServletBean的init方法开始
    - spring是一个大的父容器，springmvc是其中的一个子容器。父容器不能访问子容器对象，但是子容器可以访问父容器对象。一般做一个ssm框架项目的时候，扫描@controller注解类的对象是在springmvc容器中。而扫描@service、@component、@Repository等注解类的对象都是在spring容器中。 Spring MVC核心在于前端控制器DispatchServlet，在父类FrameworkServlet的中初始化子容器并设置和Spring容器的父子关系，在DispatchServlet的onRefresh中初始
    - MVC九大组件
        - initMultipartResolver(context);//初始化文件上传解析器
        - initLocaleResolver(context);//初始化本地解析器
        - initThemeResolver(context);//初始化主题解析器
        - initHandlerMappings(context);//初始化处理器映射器
        - initHandlerAdapters(context);//初始化处理器适配器
        - initHandlerExceptionResolvers(context);//初始化异常解析器
        - initRequestToViewNameTranslator(context);//初始化请求到视图名称解析器
        - initViewResolvers(context);//初始化视图解析器
        - initFlashMapManager(context);
10. 处理请求过程doDispatch
    - 用户向服务器发送请求，请求被Spring 前端控制Servelt DispatcherServlet捕获；
    - DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回；

- DispatcherServlet 获得的Handler，选择一个合适的HandlerAdapter。（附注：如果成功获得HandlerAdapter后，此时将开始执行拦截器的preHandler(...)方法）
- 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。
- Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；
- 根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet ；
- ViewResolver 结合Model和View，来渲染视图
- 将渲染结果返回给客户端。

十一、MySQL

> 附：锁 从锁的模式分：共享锁(Shared Locks：S锁)与排他锁(Exclusive Locks：X锁) mysql允许拿到S锁的事务读一行，允许拿到X锁的事务更新或删除一行。
>
> - 加了S锁的记录，允许其他事务再加S锁，不允许其他事务再加X锁；
> - 加了X锁的记录，不允许其他事务再加S锁或者X锁。
> - mysql对外提供加这两种锁的语法如下：
>   - 加S锁：select...lock in share mode
>   - 加X锁：手动select...for update；
>   - 自动insert、update、delete InnoDB为了支持多粒度(表锁与行锁)的锁并存，引入意向锁。
> - 意向锁是表级锁，可分为意向共享锁(IS锁)和意向排他锁(IX锁)。
>   - 事务在请求S锁和X锁前，需要先获得对应的IS、IX锁。 意向锁产生的主要目的是为了处理行锁和表锁之间的冲突，用于表明“某个事务正在某一行上持有了锁，或者准备去持有锁”。 锁锁住的永远是索引，而非记录本身，即使该表上没有任何索引，那么innodb会在后台创建一个隐藏的聚集主键索引，那么锁住的就是这个隐藏的聚集主键索引。所以说当一条sql没有走任何索引时，那么将会在每一条聚集索引后面加X锁，这个类似于表锁，但原理上和表锁应该是完全不同的。 实现算法：
>   - 临键锁（Next-key Lock）： Next-key Lock = Record Lock + Gap Lock，左开右闭区间，使用范围检索时使用临键锁，会锁住命中的区间和下一个区间，主要解决幻读问题，InnoDB默认锁。
>   - 间隙锁（Gap Lock）： 开区间，使用范围查询或等值查询且记录不存在；Gap锁之间不冲突，只在RR事务级别存在，当记录不存在退化成Gap锁。
>   - 记录锁（Record Lock）： 精准匹配时退化记录锁 锁解决并发问题： 排它锁：脏读 共享锁：不可重复读 临键锁：幻读

1. 事务ACID特性
   - 原子性（Atomicity）。事务中的所有操作要么全部执行成功，要么全部取消。
   - 一致性（Consistency）。事务开始之前和结束之后，数据库完整性约束没有破坏。
   - 隔离性（Isolation）。事务提交之前对其它事务不可见。
   - 持久性（Durability）。事务一旦提交，其结果是永久的。
2. 分布式系统CAP理论：CAP指的是Consistency(强一致性)、Availability(可用性)、Partition tolerance(分区容错性)。
3. BASE就是为了解决关系型数据库强一致性引起的可用性降低而提出的解决方案，基本可用、软状态、最终一致

4. 事务隔离级别：读未提交、读已提交（解决脏读）、可重复读（解决不可重复读，MySQL InnoDB通过MVCC解决幻读）、可串行化 - 并发访问的问题
    - 脏读：一个事务读取到了另一个事务中尚未提交的数据
    - 不可重复读：一个事务中两次读取的数据内容不一致，要求的是一个事务中多次读取时数据是一致的，这是事务update时引发的问题
    - 幻读：一个事务中两次读取的数据的数量不一致，要求在一个事务多次读取的数据的数量是一致的，这是insert或delete时引发的问题 注：MySQL InnoDB通过多版本并发控制机制来解决该不可重复读问题，事务每开启一个实例，都会分配一个版本号给它，如果读取的数据上有排它锁，则根据版本号去读取快照数据。 MySQL InnoDB在可重复读隔离级别下，采用了Next-Key Locking锁机制避免了幻读问题。
5. Spring事务
    - 传播行为：PROPAGATION_REQUIRED、PROPAGATION_SUPPORTS、PROPAGATION_MANDATORY、PROPAGATION_REQUIRED_NEW、PROPAGATION_NOT_SUPPORTED 、PROPAGATION_NEVER、PROPAGATION_NESTED
    - 隔离规则：
    - 回滚规则：
6. MySQL锁
    - 共享锁、排它锁、意向共享锁、意向排它锁
    - 行锁：纪录锁（锁住索引记录的一行）、间隙锁（锁住一个索引区间，开区间）、next-key锁（record lock + gap lock, 左开右闭区间）
    - 页锁：
    - 表锁：
7. MySQL索引：索引是一种方便快速查找的数据结构。
    - 磁盘预读：系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位，InnoDB存储引擎中默认每个页的大小为16KB
    - B-Tree：多路平衡查找树，B-Tree中的每个节点有升序排序的关键字和数据和指向子树根节点的指针，指针指向子节点所在磁盘地址
    - B+Tree: 非叶子节点上只存储key值信息，所有记录按照键值大小顺序放在叶子节点，这样每个节点加大了key值数量，降低B+Tree高度在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。
    - InnoDB: 数据文件本身就是索引文件，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点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。这种索引叫做聚集索引。InnoDB的所有辅助索引都引用主键作为data域，辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。
8. MySQL高可用方案
    - 主从复制（异步复制）过程 1）主数据库会将变更信息写入二进制日志文件（binlog）中 2）从数据库会开启一个IO线程，与主数据库建立一个连接，主数据库启动一个二进制日志转储线程，从数据库从这个转储线程中读取主数据库上的变更事件，并将变更事件记录到中继日志中 3）从数据库启动一个SQL线程从中继日志中读取变更事件，并将变更事件同步到从数据库中。
    - 主从复制延迟的原因 1）主库的从库太多，导致复制延迟，从库3-5个为宜 2）从库的硬件比主库差，导致复制延迟 3）慢SQL过多 4）主从复制设计问题，主从复制单线程，如果主库写并发太大，来不及传送到从库，就会导致延迟。设计多线程复制。 5）主从之间的网络延迟
    - binlog日志复制的三种实现方式 1）基于SQL语句的复制(statement-based replication, SBR), binlog文件较小，不是所有的UPDATE语句都能被复制 2）基于行的复制(row-based

replication, RBR)，任何情况都可以被复制，binlog 大了很多　3）混合模式复制(mixed-based replication, MBR)。

- MySQL通过复制（Replication）实现存储系统的高可用。目前，MySQL支持的复制方式有：1）异步复制（Asynchronous Replication）：原理最简单，性能最好。但是主备之间数据不一致的概率很大。　2）半同步复制（Semi-synchronous Replication）：相比异步复制，半同步复制牺牲了一定的性能，提升了主备之间数据的一致性（有一些情况还是会出现主备数据不一致）。MySQL两种略有不同的半同步复制，AFTER_SYNC（日志复制到Slave之后，Master再commit）、AFTER_COMMIT（Master commit之后再将日志复制到Slave）。　3）组复制（Group Replication）：基于Paxos算法实现分布式数据复制的强一致性。只要大多数机器存活就能保证系统可用。相比半同步复制，Group Replication的数据一致性和系统可用性更高。
- MHA（Master High Availability）目前在MySQL高可用方面是一个相对成熟的解决方案，在MySQL故障切换过程中，MHA能做到在0~30秒之内自动完成数据库的故障切换操作，并且在进行故障切换的过程中，MHA能在较大程度上保证数据的一致性，以达到真正意义上的高可用。MHA可以与半同步复制结合起来。如果只有一个slave已经收到了的二进制日志，MHA可以将的二进制日志应用于其他所有的slave服务器上，因此可以保证所有节点的数据一致性。　1）流程：当master出现故障时，通过对比slave之间I/O线程读取masterbinlog的位置，选取最接近的slave做为lateststave。其它slave通过与latest slave对比生成差异中继日志。在latest slave上应用从master保存的binlog，同时将latest slave提升为master。最后在其它slave上应用相应的差异中继日志并开始从新的master开始复制。　2）优势：1）故障切换迅速 2）master故障不会导致数据不一致 3）无需修改当前的MySQL设置 4）无需增加大量的服务器 5）五性能下降

9. MySQL日志

1）sync_binlog：这个参数是对于MySQL系统来说是至关重要的，他不仅影响到Binlog对MySQL所带来的性能损耗，而且还影响到MySQL中数据的完整性。对于“ sync_binlog”参数的各种设置的说明如下：

- sync_binlog=0，当事务提交之后， MySQL不做fsync之类的磁盘同步指令刷新binlog_cache中的信息到磁盘，而让Filesystem自行决定什么时候来做同步，或者cache满了之后才同步到磁盘。
- sync_binlog=n，当每进行n次事务提交之后， MySQL将进行一次fsync之类的磁盘同步指令来将binlog_cache中的数据强制写入磁盘。

2）Innodb事务日志刷新方式的参数： innodb_flush_log_at_trx_commit

- innodb_flush_log_at_trx_commit = 0， Innodb中的Log Thread没隔1秒钟会将log buffer中的数据写入到文件，同时还会通知文件系统进行文件同步的flush操作，保证数据确实已经写入到磁盘上面的物理文件。但是，每次事务的结束（ commit或者是rollback）并不会触发Log Thread将log buffer中的数据写入文件。所以，当设置为0的时候，当 MySQL Crash 和OS Crash 或者主机断电之后，最极端的情况是丢失1秒时间的数据变更。
- innodb_flush_log_at_trx_commit = 1，这也是Innodb的默认设置。我们每次事务的结束都会触发Log Thread将log buffer中的数据写入文件并通知文件系统同步文件。这个设置是最安全的设置，能够保证不论是MySQL Crash 还是 OS Crash 或者是主机断电都不会丢失任何已经提交的数据。
- innodb_flush_log_at_trx_commit = 2，当我们设置为2的时候， Log Thread会在我们每次事务结束的时候将数据写入事务日志，　但是这里的写入仅仅是调用了文件系统的文件写入操作。而我们的文件系统都是有缓存机制的，所以Log Thread的这个写入并不能保证内容真的已经写入到物理　磁盘上面完成持久化的动作。文件系统什么时候会将缓存中的这个数据同步到物理磁盘文件Log Thread就完全不知道了。所以，当设置为2的时候，　MySQL Crash 并不会造成数据的丢失，但是OS Crash或者是主机断电后可能丢失的数据量就完全控制在文件系统上了。

3）InnoDB undo日志和redo日志

- 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Undo Log，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。与redo log不同的是，磁盘上不存在单独的undo log文件，它存放在数据库内部的一个特殊段(segment)中， 这称为undo段(undo segment)，undo段位于共享表空间内。

- redo log就是保存执行的SQL语句到一个指定的Log文件，当mysql执行数据恢复时，重新执行redo log记录的SQL操作即可。引入buffer pool会导致更新的数据不会实时持久 化到磁盘，当系统崩溃时，虽然buffer pool中的数据丢失，数据没有持久化，但是系统可以根据Redo Log的内容，将所有数据恢复到最新的状态。redo log在磁盘上作 为一个独立的文件存在。默认情况下会有两个文件，名称分别为 ib_logfile0和ib_logfile1。

10. 脑裂：集群的脑裂通常是发生在集群中部分节点之间不可达而引起的（或者因为节点请求压力较大，导致其他节点与该节点的心跳检测不可用）。当上述情况发生时，不同分裂的小集群会自主的选择出master节点，造成原本的集群会同时存在多个master节点。
11. SQL优化 - 创建索引、复合索引（左前缀特性）、索引不会包含有NULL值的列、使用短索引、排序的索引问题、like语句操作、不要在列上进行运算、不使用NOT IN和<>操作，NOT IN可以NOT EXISTS代替