Java面试-基础：

- Map
- JVM
- GC
- 类加载
- 线程池
- 高并发
- 缓存
- HTTPS
- 幂等
- 限流
- 一致性Hash算法
- 事务
- 动态代理
- SpringBoot启动机制
- SpringIOC容器启动流程
- 简历

- `HashMap`：数组，链表，红黑树（链表长度大于8之后）
  - `hash` 算法(确保足够随机)： `hash ^ (hash >>> 16)`
  - 数组下标： `hash & (n-1)` 等价于 `hash % n`
  - 扩容因子：0.75，变长2倍
- `Hashtable`：读写都加了 `synchronized`，高并发下效率低下
- `ConcurrentHashMap`：
  - JDK1.7：`Segment` 分段锁，需要进行两次 `hash` 计算。相当于嵌套的 `HashMap`，只不过第一层是加锁的。
  - JDK1.8: 分段锁粒度更小，并发度更高，每个数组 `Node` 都是一把锁, 取消了 `Segment`，只需要一次 `hash` 计算。
- JVM 运行时数据区
  - 程序计数器
  - 虚拟机栈: **每个方法在被调用时会创建一个栈帧**，从调用到执行完成的过程，就是入栈出栈的过程。
  - 堆：最大的一块，所有线程共享，**对象实例在堆里分配内存**，是垃圾收集器（GC）管理的主要区域。
  - 方法区：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
  - 直接内存：**并不是虚拟机运行时数据区的一部分**，也不是Java虚拟机规范中定义的内存区域。
  - 运行时常量池
    - JDK1.6: 方法去
    - JDK1.7: 堆
    - JDK1.8: 元空间（直接内存）
- 垃圾回收 （GC）
  - 标记-清除算法（Mark-Sweep）：内存碎片
  - 复制算法（Copying）: 空间浪费（用一半，回收时把数据复制到另一半）
  - 标记-整理算法（Mark-Compact）：性能影响（内存区域的移动），先标记可回收对象，回收时再顺序排列
  - 分代收集
    - 新生代（复制算法）
    - 老年代（标记清除或标记整理算法）
  - 回收器：串行，并行，G1
    - 传统垃圾回收，在逻辑上分出了相对独立的两大块：新生代和老年代
    - G1: 没有严格的新生代和老年代，而是把内存区域分成一个个小块，每一个小块即可能给新生代用也可能给老年代用。新生代依然是复制算法，老年代是标记清除算法。
- 类加载机制：加载（类加载器），连接（验证，准备- `static` 初始值，解析-常量值引用转实际值），初始化- `static` 变量和语句块，使用，卸载
  - 类加载器
    - 启动类加载器（无法调用）： `Bootstrap - JAVA_HOME\lib`
    - 扩展类加载器： `Extension JAVA_HOME\lib\ext`
    - 应用程序加载器（程序中默认的加载器）：用户类路径上的类库
    - 自定义类加载器（部署热替换，类加密解密）

- 双亲委派：首先将加载任务委托给父加载器，依次递归，如果父类加载器可以完成加载任务，就成功返回；只有父类加载器无法完成加载任务时，才自己去加载。
  - 具备了优先级的层次关系，保证Java程序稳定运行（比如启动类加载器的类无法擅自篡改）
  - 启动类加载器是用 C++ 语言实现的，它负责加载 Java 核心类库，如 rt.jar 等。启动类加载器是虚拟机的一部分，它不是 java.lang.ClassLoader 的子类，也不受 Java 代码的控制。因此，启动类加载器里的类无法被用户篡改，除非修改虚拟机本身。
  - 双亲委派机制的作用是**防止重复加载同一个类，以及保护核心类库不被恶意修改。**如果用户自定义的类加载器不遵循双亲委派机制，那么可能会破坏类的唯一性，导致类型转换异常或安全漏洞。
- 线程池： corePoolSize，maximumPoolSize，keepAliveTime，workQueue，RejectedExecutionHandler (拒绝策略)
  i. 当线程数小于 corePoolSize，会直接创建线程
  ii. 当线程数等于 corePoolSize ， 会把任务提交到 workQueue
  iii. 当 workQueue 满了，线程池又开始创建线程
    - 可以通过自定义队列的拒绝添加来提前开启线程，等线程满了可以继续往队列里放
  iv. 当线程数等于 maximumPoolSize，开始启用拒绝策略 RejectedExecutionHandler
    - AbortPolicy：抛出异常
    - CallerRunsPolicy：谁提交的谁执行
    - DiscardOldestPolicy：丢弃最老的任务，然后执行最新的任务
    - DiscardPolicy：直接丢弃
  - 线程池为什么这样设计：
    - 线程的创建，需要获取全局锁，为了保证线程池的状态和线程数量的一致性。这也会影响线程池的效率，所以设计 corePoolSize 。
      - corePoolSize 是在不超时情况下，保持活跃的最少线程数。这里的超时指的是 ThreadPoolExecutor.allowCoreThreadTimeOut
    - 线程是系统的稀缺资源，所以要尽量用最小的线程数干最多的事情。
- 多线程下都概率远远大于写概率，如何解决并发问题
  - 使用内置锁，会有性能问题
  - 一写多读：使用 volatile 关键字，不保证原子性，但是保证可见性
  - 多写多读：使用读写锁（ ReentrantReadWriteLock ）
  - 少写多读：写时复制容器（ CopyOnWriteXxx ），读写分离
    - 缺陷：内存占用，数据一致性（只保证数据的最终一致性）
- 高并发下，如何做到安全的修改同一行数据
  - JVM：内置锁 synchronized； Lock （ finally 里释放锁）；
  - 分布式：分布式锁（DB 性能差，Redis 死锁，Zookeeper）
    - Zookeeper：存取数据，节点监听
      - PERSISTENT：持久化目录节点，断开连接后，节点依旧在
      - PERSISTENT_SEQUENTIAL：持久化顺序编号目录节点，断开连接后，节点依旧在，给该节点名称进行顺序编号
      - EPHEMERAL：临时目录节点，端口连接后，节点被删除
      - EPHEMERAL_SEQUENTIAL：临时顺序编号目录节点，端口连接后，节点被删除，给该节点名称进行顺序编号
    - 基于异常的（节点重名会报错）：使用临时节点
      - 获得锁：创建临时目录节点成功（ ZkClien::createEphemeral，节点重名会报错）
      - 等待获取锁：监听锁的节点是否被删除（ IZkDataListener::handleDataDeleted; ZkClient::subscribDataChanges ）
      - 释放锁：删除锁节点（ ZkClient::delete; ZkClient::close ）
    - 基于相互监听（性能高，占用资源，每个线程都会创建一个临时顺序编号节点）：使用临时顺序编号节点
      - 获得锁：创建临时顺序编号节点，判断是不是排在最前面，是就获得锁
      - 等待获取锁：监听前一个节点是否被删除
      - 释放锁：删除自己的临时顺序编号节点
- Redis 高性能的原因
  - 数据存在内存中
  - 存储结构简单
  - 单线程，没有多线程的资源竞争
  - 多路复用，把多个请求的指令放到一个队列里，让单个Redis线程去队列拿指令
  - resp协议，协议简单，人类可读：*代表指令由几部分组成，$代表每部分有几个字符
    - set jxch value -> *3 $3 set $4 jxch $5 value
- 怎么控制缓存的更新：使用 SpringCache - CacheManger(RedisTemplate); CacheService
  - 查询：
    a. 从缓存中加载数据
    b. 如果缓存中有就直接返回给调用端
    c. 如果缓存中没有数据，就从数据库中加载数据（如果要求强一致性的话，同一时间只允许一个请求访问数据库）
    d. 数据库查询的结果不为空，就把数据放入缓存中
  - 更新：

        - 数据实时同步失效（增量/主动）：强一致性，**更新数据库之后淘汰缓存，读请求更新缓存**，为避免缓存雪崩，**更新缓存的过程需要进行同步控制，同一时间只允许一个请求访问数据库**，为了保证数据的一致性还要**加上缓存失效。**
            - 不能先淘汰缓存再更新数据库，会导致并发问题，大量读请求访问到数据库。
            - 锁的粒度要细，避免阻塞太多线程；单体应用的话注意不要意外释放其他线程的锁。
            - **缺点：如果更新业务暂时连不上缓存的话，其他业务就会出现脏读**
        - 数据准实时更新（增量/被动）：准一致性，更新数据库后，**异步更新缓存**，使用多线程技术或者MQ实现。优点是和缓存解耦。
            a. 业务服务更新数据
            b. 业务服务通过MQ向缓存服务发送消息
            c. 缓存服务接收消息
            d. 缓存服务读取数据
            e. 缓存服务更新缓存
        - 任务调度更新（全量/被动）：最终一致性，采用任务调度框架，按照一定频率更新。比如报表数据（年度，月度），在晚上3点同步一次。
- A服务调用B服务多接口，响应时间最短方案：改串行请求为并行请求
- A系统给B系统转100块钱，如何实现?
    - `@Transactional` 事务里有外调耗时接口占用DB连接怎么办?
        - **使用编程式事务** `TransactionTemplate`：`template.execute`
        - 为了防止第三方请求挂了，**先记录请求，标记为正在处理中**
    - 如何保证**幂等性**
        - 基于状态机的**乐观锁**（数据库version字段，确保只有一个线程操作数据库，比如能成功更新version为0再向下执行）
- 如何避免 `MQ` 重复消费（幂等性）：消息超时重传（生产者重传和 `MQ` 重传）
    - 幂等性：不论操作重复多少次都不改变结果
    - 乐观锁：数据库版本号 `Version`， `set version=version+1 where version=oldversion`
    - 去重表：构建一个存储任务唯一ID的表，执行任务前先检查去重表里有没有执行过这个任务（为了加快速度可以放到缓存里）
- 如何做限流策略（令牌桶和漏斗算法）
    - 漏桶：请求先进入漏桶里（请求队列），当请求量大的时候，超出队列的部分可以熔断
    - 令牌桶（用的最多）：令牌池不断地以一定的速度生成一定量的令牌，客户端调服务之前，先去令牌桶请求一个令牌才能调
        - 优点，可以应对突发流量，比如秒杀1000个商品可以预制1000个令牌
        - 当令牌桶里没有令牌时，可以拒绝服务
- 一致性 `Hash` 算法（ `Hash` 环）：解决分布式扩容后需要重新计算 `hash` 的问题（代价太高）
    - Hash环：长度是2^32，确定服务器在此环上的位置
    - 数据存放：对2^32取模，确定数据在此环上的位置，然后存放在此环顺时针方向上最近的服务器上
    - 优点：服务器节点的增删只影响到少量的数据
- HTTP和HTTPS有什么区别
    - HTTP是不安全的：数据拦截、数据篡改、数据攻击
    - HTTPS的安全需求：数据加密、身份验证、数据完整性
    - 对称加密：就是一个密钥可以加密也可以解密
    - 非对称加密：就是公钥加密的内容，必须用私钥才能解密，私钥加密的内容，必须用公钥才能解密
    - HTTPS请求过程：
        a. 客户端向服务器发送HTTPS请求，**请求服务器的公钥证书。**
        b. 服务器将公钥证书发送给客户端，公钥证书包含了服务器的公钥、证书持有者信息、数字签名等。
        c. 客户端验证服务器的证书，主要是验证证书的有效期、颁发者、域名等，以及证书的数字签名是否合法。如果验证通过，客户端就信任了服务器的公钥。
        d. **客户端随机生成一个用于对称加密的密钥，称为会话密钥，然后用服务器的公钥加密后发送给服务器。**
        e. **服务器用自己的私钥解密得到会话密钥，然后用会话密钥加密一个确认信息发送给客户端。**
        f. **客户端用会话密钥解密得到确认信息**，然后开始用会话密钥加密解密数据，进行安全的通信。
- `Cookie/Session` 机制：单点登录
    - `Cookie`： `key:value` 键值信息
        - 不可跨域名（只能拿到当前域名下的cookie，包含父级域名）
        - 有效期限制
        - path表示携带cookie的请求路径（/: 表示全部请求都要携带cookie）
    - `Session`：服务器端基于内存的缓存技术，用来保存针对每个用户的会话数据
        - 通过 `Session ID` 来区分用户，用户只要连接到服务器，服务器就会为之分配一个唯一的 `Session ID`
        - 有失效时间（超时限为得到连接的 `session` 将被销毁）
        - `Session` 用法： `HttpServletRequest::getSession`
            - `HttpSession::setAttribute`
            - `HttpSession::getAttribute`
            - `HttpSession::removeAttribute`

- Cookie 和 Session 的交互
  a. 服务器新建一个 Session ， 把 SessionID 设置到 Cookie 中
  b. 服务器通过 HttpServletResponse::addCookie 向 Header 中设置 Cookie
  c. 浏览器接收 Header 中的 Cookie 并保存，在下次请求的 Header 中，携带 Cookie 信息
  d. 当程序请求 Session 时，首先根据 Cookie 中携带的 SessionID 检索 Session
  e. 检索不到，就新建一个 Session ， 重复第1步
- 使用场景： Cookie 数据存放在客户的浏览器上， Session 数据存放在服务器上
  - Cookie 很不安全，客户端可以分析并进行 Cookie 欺骗，所以如果考虑安全性的话，应该多用 Session
  - Session 会在一定时间保存在服务器上。当访问增多，会占用服务器性能，所以如果考虑减轻服务器压力的话，应该多用 Cookie
  - 单个 Cookie 保存的数据不能超过 4K ，很多浏览器都限制一个站点最多保存 20 个 Cookie
  - 若客户端禁止 Cookie ： 一般使用 URL 重写，把 SessionID 直接附加到 URL 后面

- 事务
  - 事务的七个传播属性
    - PROPAGAtiON_REQUIRED：支持当前事务，如果当前没有事务，就新建一个事务。这是默认的。
    - PROPAGATION_SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行
    - PROPAGATION_MANDATORY：支持当前事务，如果当前没有事务，就抛出异常
    - PROPAGATION_REQUIRES_NEW：新建事务，如果当前存在事务，把当前事务挂起
    - 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
    - PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，就抛出异常
    - PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行（可以实现**部分回滚**）。如果当前没有事务，则与 PROPAGAtiON_REQUIRED 类似操作（新建一个事务）
      - 嵌套的事务可以**独立于当前事务进行单独地提交或回滚**，如果外部事务提交，嵌套事务也会被提交，这个规则同样适用于回滚
      - 这是因为 PROPAGATION_NESTED 执行后并不是真正地提交了子事务，而是将子事务的状态保存在一个**保存点**上，等待外部事务的最终提交或回滚。**如果外部事务提交，那么子事务也会提交，如果外部事务回滚，那么子事务也会回滚到保存点**。
      - 这意味着 PROPAGATION_NESTED 可以实现部分回滚，而 PROPAGATION_REQUIRED 只能全部回滚或全部提交
      - 比如 PROPAGATION_NESTED 失败了，但是外部事务成功了，则**外部事务可以提交，只是嵌套事务部分回滚**
  - 事物的五种隔离级别：指的是若干个并发事务之间的隔离程度
    - ISOLATION_DEFAULT：数据库默认的事务隔离级别
      - MySQL(ISOLATION_REPEATABLE_READ)： show variables like '%tx_isolation%'
    - ISOLATION_READ_UNCOMMITTED （未提交读）：最低隔离级别，**允许另外一个事务可以看到这个事务未提交的数据，会产生脏读，不可重复读和幻读。**
    - ISOLATION_READ_COMMITTED （已提交读，不可重复读）：保证事务修改的数据提交后才能被另外一个事务读取。**另外一个事务不能读取该事务未提交的数据。可以避免脏读出现，但是可能会出现不可重复读和幻读。**
    - ISOLATION_REPEATABLE_READ （可重复读）：**可以防止脏读和不可重复读。但是可能出现幻读。**它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了不可重复读的产生。
    - ISOLATION_SERIALIZABLE （可串行化）：最高代价但是最可靠的事务隔离级别。事务被处理为**顺序执行。防止脏读，不可重复读和幻读。** 工作中尽量不要用，因为它会**锁表**。
  - 脏读：读取未提交的数据
  - 不可重复读：在同一个事务里面读取同一行数据，产生了不一样的结果（第二次读的时候，其他事务进行了提交）
  - 幻读：多次查询同一个范围的数据，但得到的结果集不一致。这是因为在这个过程中，其他事务可能已经新增或删除了符合该范围条件的数据行。幻读可能会导致数据查询结果不准确。
  - 事务隔离级别越高，越能保证数据的一致性和完整性，但是执行效率也越低。
  - MySQL开启事务： BEGIN ； 回滚： ROLLBACK
- @Transaction 位置以及回滚策略
  - 不建议写在接口上（会导致事务丢失且代理行为不一致者不可预测的代理行为）
    - 如果用 CGLIB 的动态代理会导致事务丢失，因为 Spring 只会检测被代理类上的注解
    - 只有用 JDK 的动态代理才会生效，因为 Spring 会检测接口上的注解
  - 建议写在业务实现类上，只能写在 public 方法上，且方法只能被外部调用才生效
    - 如果用 JDK 的代理方式，很好理解，接口里都是 public 方法
    - 如果使用 CGLIB 代理方式，是不可以代理 final 和 private 的， CGLIB 虽然可以代理 protected 方法，但是 Spring AOP 会先判断该方法是否是 public 的，如果不是，就会用反射调用原始方法，所以写在 protected 方法上也会失效
    - Spring AOP 这样设计一方面也是为了保持和 JDK 代理行为的一致性，因为它默认使用 JDK 动态代理来实现，如果被代理的类没有实现接口，则会自动使用 CGLIB 来进行代理。
    - AOP 代理是通过代理对象访问原方法，如果在原方法内直接调用内部方法，就不会走代理对象了，所以只能外部调用
  - 事务回滚（按下列顺序处理）：
    a. noRollbackFor ： 不回滚，提交
    b. rollbackFor ： 回滚

    c. RuntimeException： 回滚
    d. 发生其他异常时：不回滚，提交
- 动态代理: 是使用反射和字节码技术，在运行期创建指定接口或类的子类（动态代理）以及其实例对象的技术，通过这个技术可以**无侵入**性的为代码进行增强
  - JDK： Proxy::newProxyInstance ， InvocationHandler::invoke
    - 被代理的类必须实现一个接口
  - CGLIB: 是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB通过**继承方式**实现代理;
    - Enhancer： 来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用 create() 方法得到代理对象，对这个对象所有非 final 方法的调用都会转发给 MethodInterceptor；
      - Enhancer::setSuperclass
      - Enhancer::setCallback
      - Enhancer::create
    - MethodInterceptor： 动态代理对象的方法调用都会转发到 intercept 方法进行增强;
- SpringBoot 启动机制
  - 整合三方依赖： maven 父子继承
  - 无配置文件集成 SpringMvc：内置 Tomcat ， 通过 @EnableWebMvc 启动 SpringMvc
    - @SpringBootApplication -> @SpringBootConfiguration; @EnableAutoConfiguration; @ComponentScan
    - @EnableAutoConfiguration -> @Import(AutoConfigurationImportSelector.class)
    - class AutoConfigurationImportSelector implements DeferredImportSelector -> selectImports()
    - selectImports() -> getCandidateConfigurations() -> META-INF/spring.factories
    - spring-boot-autoconfigure.jar -> META-INF/spring.factories
    - META-INF/spring.factories -> xxx.xxx.WebMvcAutoConfiguration
  - Tomcat 是怎么启动的
    - TomcatWebServerFactoryCustomizer 注册
      - META-INF/spring.factories -> xxx.xxx.EmbeddedWebServerFactoryCustomizerAutoConfiguration -> @ConditionalOnClass({Tomcat.class, Upg
      - @Bean TomcatWebServerFactoryCustomizer -> implements WebServerFactoryCustomizer
      - WebServerFactoryCustomizerBeanPostProcessor -> List<WebServerFactoryCustomizer<?>>
    - TomcatServletWebServerFactory 注册
      - META-INF/spring.factories -> xxx.xxx.ServletWebServerFactoryAutoConfiguration -> @Import(EmbeddedTomcat.class) @Bean@ConditionalOn
      - EmbeddedTomcat -> @ConditionalOnClass({Servlet.class, Tomcat.class, UpgradeProtocol.class}) @Bean TomcatServletWebServerFactory
    - TomcatServletWebServerFactory::getWebServer 启动
      - TomcatServletWebServerFactory implements ServletWebServerFactory::getWebServer
      - SpringApplicaiton::run -> refreshContext() -> AbstractApplicationContext:::refresh -> ServletWebServerApplicationContext::onRefres
- Spring IOC 容器启动流程
  - IOC 容器，就是内存里的 Map<beanname, bean>
  - ApplicationContext app = new AnnotationConfigApplicationContext(App.class)
  - AbstractApplicationContext::refresh
  - obtainFreshBeanFactory 获取 Bean 工厂
  - prepareBeanFactory 准备上下文
  - postProcessBeanFactory beanFactory 后置处理器
  - invokeBeanFactoryPostProcessors 调用 beanFactory 后置处理器
  - registerBeanPostProcessors 注册 Bean 后置处理器
  - initMessageSource 初始化上下文
  - initApplicationEventMulticaster 初始化事件器
  - onRefresh 初始化特殊 Bean
  - registerListeners 监听器
  - finishBeanFactoryInitialization 实例化非懒加载的bean
    - preInstantiateSingletons -> getBean -> doGetBean
    - getSingleton -> Map<String, Object> singletonObjects 先去缓存中拿
    - createBean -> doCreateBean 没有的话就创建Bean
    - createBeanInstance 创建实例
    - populateBean 属性赋值
    - initializeBean 初始化Bean
      - invokeAwareMethods Aware
      - applyBeanPostProcessorsBeforeInitialization 前置处理器
      - invokeInitMethods 初始化
      - applyBeanPostProcessorsAfterInitialization 后置处理器

    - `finishRefresh` 发布完成事件
    - `close() -> doClose()` 关闭容器
        - `destroyBeans` 销毁Bean
        - `closeBeanFactory` 关闭Bean工厂
        - `onClose` 子类(`AbstractApplicationContext`)自定义清理工作
        - `resetCommonCaches` 重置缓存，避免内存泄漏
- 简历：简洁
    - 个人概况；教育背景；求职意向；职业技能；工作经验；项目经验；自我评价；证书