1.HashMap 底层实现原理，红黑树，B+树，B 树的结构原理，volatile 关键字，CAS（比较与交换）实现原理

首先 HashMap 是 Map 的一个实现类，而 Map 存储形式是键值对(key,value)的。可以看成是一个一个的 Entry。Entry 所存放的位置是由 key 来决定的。

Map 中的 key 是无序的且不可重复的，所有的 key 可以看成是一个 set 集合，如果出现 Map 中的 key 如果是自定义类的对象，则必须重写 hashCode 和 equals 方法，因为如果不重写，使用的是 Object 类中的 hashCode 和 equals 方法，比较的是内存地址值不是比内容。

Map 中的 value 是无序的可重复的，所有的 value 可以看成是 Collection 集合，Map 中的 value 如果是自定义类的对象必须重写 equals 方法。

至于要重写 hashCode 和 equals 分别做什么用，拿 hashMap 底层原理来说：

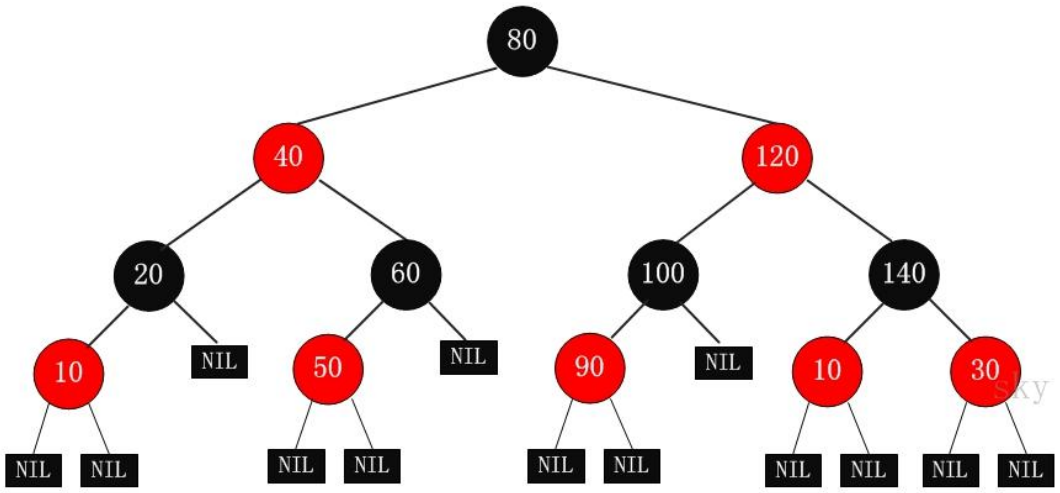
当我们向 HashMap 中存放一个元素(k1,v1)，先根据 k1 的 hashCode 方法来决定在数组中存放的位置。

如果这个位置没有其它元素，将(k1,v1)直接放入 Node 类型的数组中，这个数组初始化容量是 16，默认的加载因子是 0.75，也就是当元素加到 12 的时候，底层会进行扩容，扩容为原来的 2 倍。如果该位置已经有其它元素(k2,v2)，那就调用 k1 的 equals 方法和 k2 进行比较二个元素是否相同，如果结果为 true，说明二个元素是一样的，用 v1 替换 v2，如果返回值为 false，二个元素不一样，就用链表的形式将(k1,v1)存放。

不过当链表中的数据较多时，查询的效率会下降，所以在 JDK1.8 版本后做了一个升级，hashmap 就是当链表中的元素达到 8 并且元素数量大于 64 时，会将链表替换成红黑树才会树化时，会将链表替换成红黑树，来提高查找效率。因为对于搜索，插入，删除操作多的情况下，使用红黑树的效率要高一些。

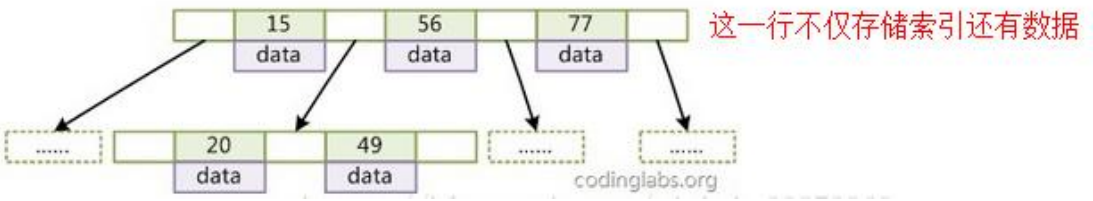
原因是因为红黑树是一种特殊的二叉查找树，二叉查找树所有节点的左子树都小于该节点，所有节点的右子树都大于该节点，就可以通过大小比较关系来进行快速的检索。

在红黑树上插入或者删除一个节点之后，红黑树就发生了变化，可能不满足红黑树的 5 条性质，也就不再是一颗红黑树了，而是一颗普通的树，可以通过左旋和右旋，使这颗树重新成为红黑树。红黑树的 5 条性质（根节点是黑色，每个节点是黑色或者是红色，每个叶子节点是黑色，如果一个节点是红色它的子节点必须是黑色的，从一个节点到该节点的子孙外部节点的所有路径上包含相同数目的黑点）



而且像这种二叉树结构比较常见的使用场景是 Mysql 二种引擎的索引，Myisam 使用的是 B 树，InnoDB 使用的是 B+树。

首先 B 树它的每个节点都是 Key.value 的二元组，它的 key 都是从左到右递增的排序，value 中存储数据。这种模式在读取数据方面的性能很高，因为有单独的索引文件，Myisam 的存储文件有三个.frm 是表的结构文件，.MYD 是数据文件，.MYI 是索引文件。不过 Myisam 也有些缺点它只支持表级锁，不支持行级锁也不支持事务，外键等，所以一般用于大数据存储。



另外对于 HashMap 实际使用过程中还是会出现一些线程安全问题：

HashMap 是线程不安全的，在多线程环境下，使用 Hashmap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，而且会抛出并发修改异常，导致原因是并发争取线程资源，修改数据导致的，一个线程正在写，一个线程过来争抢，导致线程写的过程被其他线程打断，导致数据不一致。

HashTable 是线程安全的，只不过实现代价却太大了，简单粗暴，get/put 所有相关操作都是 synchronized 的，这相当于给整个哈希表加了一把大锁。多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。

为了应对 hashmap 在并发环境下不安全问题可以使用，ConcurrentHashMap 大量的利用了 volatile，CAS 等技术来减少锁竞争对于性能的影响。

在 JDK1.7 版本中 ConcurrentHashMap 避免了对全局加锁，改成了局部加锁（分段锁），分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。

不过这种结构的带来的副作用是 Hash 的过程要比普通的 HashMap 要长。

所以在 JDK1.8 版本中 CurrentHashMap 内部中的 value 使用 volatile 修饰，保证并发的可见性以及禁止指令重排，只不过 volatile 不保证原子性，使用为了确保原子性，采用 CAS（比较交换）这种乐观锁来解决。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。

如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被 b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

volatile 有三个特性：可见性，不保证原子性，禁止指令重排。

可见性：线程 1 从主内存中拿数据 1 到自己的线程工作空间进行操作（假设是加 1）这个时候数据 1 已经改为数据 2 了，将数据 2 写回主内存时通知其他线程（线程 2，线程 3），主内存中的数据 1 已改为数据 2 了，让其他线程重新拿新的数据（数据 2）。

不保证原子性：线程 1 从主内存中拿了一个值为 1 的数据到自己的工作空间里面进行加 1 的操作，值变为 2，写回主内存，然后还没有来得及通知其他线程，线程 1 就被线程 2 抢占了，CPU 分配，线程 1 被挂起，线程 2 还是拿着原来主内存中的数据值为 1 进行加 1，值变成 2，写回主内存，将主内存值为 2 的替换成 2，这时线程 1 的通知到了，线程 2 重新去主内存拿值为 2 的数据。

禁止指令重排：首先指令重排是程序执行的时候不总是从上往下执行的，就像高考答题，可以先做容易的题目再做难的，这时做题的顺序就不是从上往下了。禁止指令重排就杜绝了这种情况。

（一般面试官开始问你会从 java 基础问起，一问大多数会问到集合这一块，而集合问的较多的是 HashMap，这个时候你就可以往这些方向带着面试官问你，而且扩展的深度也够，所以上面的干货够你说个十来分钟吧，第一个问题拿下后，面试官心里至少简单你的基础够扎实，第一印象分就留下了）

2.Spring 的 AOP 和 IOC 是什么？使用场景有哪些？Spring 事务，事务的属性，传播行为，数据库隔离级别

AOP：面向切面编程

即在一个功能模块中新增其他功能，比方说你要下楼取个快递，你同事对你说帮我也取一下呗，你就顺道取了。在工作中如果系统中有些包和类中没有使用 AOP，例如日志，事务和异

常处理，那么就必须在每个类和方法中去实现它们。 代码纠缠每个类和方法中都包含日志，事务以及异常处理甚至是业务逻辑。在一个这样的方法中，很难分清代码中实际做的是什么处理。AOP 所做的就是将所有散落各处的事务代码集中到一个事务切面中。

场景

比方说我现在要弄一个日志，记录某些个接口调用的方法时间。使用 Aop 我可以在这个接口前插入一段代码去记录开始时间，在这个接口后面去插入一段代码记录结束时间。

又或者你去访问数据库，而你不想管事务（太烦），所以，Spring 在你访问数据库之前，自动帮你开启事务，当你访问数据库结束之后，自动帮你提交/回滚事务！

异常处理你可以开启环绕通知，一旦运行接口报错，环绕通知捕获异常跳转异常处理页面。

动态代理

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。它的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理。JDK 动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK 动态代理的核心是 InvocationHandler 接口和 Proxy 类。如果目标类没有实现接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。CGLIB 是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，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。

IOC：依赖注入或者叫做控制反转

正常情况下我们使用一个对象时都是需要 new Object()的。而 ioc 是把需要使用的对象提前创建好，放到 spring 的容器里面。

所有需要使用的类都会在 spring 容器中登记，告诉 spring 你是个什么东西，你需要什么东西，然后 spring 会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 spring 来控制，也就是说控制对象生存周期的不再是引用它的对象，而是 spring。DI(依赖注入)其实就是 IOC 的另外一种说法，其实它们是同一个概念的不同角度描述。

场景：

正常情况下我们使用一个对象时都是需要 new Object() 的。而 ioc 是把需要使用的对象提前创建好，放到 spring 的容器里面。需要使用的时候直接使用就行，而且可以设置单例或多例，非常灵活。

我们在 service 层想调用另外一个 service 的方法，不需要去 new 了，直接把它交给 spring 管理，然后用注解的方式引入就能使用。

IOC 三种注入方式

（1）XML：Bean 实现类来自第三方类库，例如 DataSource 等。需要命名空间等配置，例如：context，aop，mvc。

（2）注解：在开发的类使用@Controller，@Service 等注解

（3）Java 配置类：通过代码控制对象创建逻辑的场景。例如：自定义修改依赖类库。

什么是事务？

事务是访问并可能更新数据库中各种数据项的一个程序执行单元。

Spring 事务与数据库事务关系？

Spring 的事务是对数据库的事务的封装，最后本质的实现还是在数据库，假如数据库不支持事务的话，Spring 的事务是没有作用的。所以说 Spring 事务的底层依赖 MySQL 的事务，Spring 是在代码层面利用 AOP 实现，执行事务的时候使用 TransactionInceptor 进行拦截，然后处理。本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，执行完目标方法之后根据执行的情况提交或者回滚。

属性（特性）
A(原子性)：要么全部完成，要么完全不起作用
C(一致性)：一旦事务完成（不管成功还是失败），业务处于一致的状态，而不会是部分完成，部分失败。
I(隔离性)：多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
D(持久性)：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，事务的结果被写到持久化存储器中。

什么叫事务传播行为？

传播，至少有两个东西，才可以发生传播。单体不存在传播这个行为。事务传播行为就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行。

Spring 支持 7 中事务传播行为
propagation_required（需要传播）：当前没有事务则新建事务，有则加入当前事务
propagation_supports（支持传播）：支持当前事务，如果当前没有事务则以非事务方式执行
propagation_mandatory（强制传播）：使用当前事务，如果没有则抛出异常
propagation_nested（嵌套传播）：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，则执行需要传播行为。
propagation_never（绝不传播）：以非事务的方式执行，如果当前有事务则抛出异常
propagation_requires_new（传播需要新的）：新建事务，如果当前有事务则把当前事务挂起
propagation_not_supported（不支持传播）：以非事务的方式执行，如果当前有事务则把当前事务挂起

数据库事务的隔离级别

数据库事务的隔离级别有 4 个，由低到高依次为 Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

√：可能出现 ×：不会出现

说明	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

注意：我们讨论隔离级别的场景，主要是在多个事务并发的情况下，因此，接下来的讲解都围绕事务并发。Read uncommitted 读未提交
公司发工资了，领导把 20000 元打到廖志伟的账号上，但是该事务并未提交，而廖志伟正好去查看账户，发现工资已经到账，是 20000 元整，非常高兴。可是不幸的是，领导发现发给廖志伟的工资金额不对，是 16000 元，于是迅速回滚了事务，修改金额后，将事务提交，最后廖志伟实际的工资只有 16000 元，廖志伟空欢喜一场。

出现上述情况，即我们所说的脏读，两个并发的事务，“事务 A：领导给廖志伟发工资”、“事务 B：廖志伟查询工资账户”，事务 B 读取了事务 A 尚未提交的数据。当隔离级别设置为 Read uncommitted 时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

Read committed 读提交

廖志伟拿着工资卡去消费，系统读取到卡里确实有 2000 元，而此时她的老婆也正好在网上转账，把廖志伟工资卡的 2000 元转到另一账户，并在廖志伟之前提交了事务，当廖志伟扣款时，系统检查到廖志伟的工资卡已经没有钱，扣款失败，廖志伟十分纳闷，明明卡里有钱，为何…

出现上述情况，即我们所说的不可重复读，两个并发的事务，“事务 A：廖志伟消费”、“事务 B：廖志伟的老婆网上转账”，事务 A 事先读取了数据，事务 B 紧接了更新了数据，并提交了事务，而事务 A 再次读取该数据时，数据已经发生了改变。当隔离级别设置为 Read committed 时，避免了脏读，但是可能会造成不可重复读。大多数数据库的默认级别就是 Read committed，比如 Sql Server , Oracle。如何解决不可重复读这一问题，请看下一个隔离级别。

Repeatable read 重复读

当廖志伟拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），廖志伟的老婆就不可能对该记录进行修改，也就是廖志伟的老婆不能在此时转账。这就避免了不可重复读。廖志伟的老婆工作在银行部门，她时常通过银行内部系统查看廖志伟的信用卡消费记录。有一天，她正在查询到廖志伟当月信用卡的总消费金额（select sum(amount) from transaction where month = 本月）为 80 元，而廖志伟此时正好在外面胡吃海喝后在收银台买单，消费 1000 元，即新增了一条 1000 元的消费记录（insert transaction … ），并提交了事务，随后廖志伟的老婆将廖志伟当月信用卡消费的明细打印到 A4 纸上，却发现消费总额为 1080 元，廖志伟的老婆很诧异，以为出现了幻觉，幻读就这样产生了。当隔离级别设置为 Repeatable read 时，可以避免不可重复读，但会出现幻读。注：MySQL 的默认隔离级别就是 Repeatable read。

Serializable 序列化

Serializable 是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

3. Spring 和 SpringMVC，MyBatis 以及 SpringBoot 的注解分别有哪些？

SpringMVC 的工作原理，SpringBoot 框架的优点，MyBatis 框架的优点

Spring 注解

声明bean的注解	
@Component	组件，没有明确的角色
@Service	在业务逻辑层使用（service层）
@Repository	在数据访问层使用（dao层）
@Controller	在展现层使用，控制器的声明（C）
注入bean的注解	
@Autowired	由Spring提供
@Resource	由JSR-250提供
java配置类相关注解	
@Bean	注解在方法上，声明当前方法的返回值为一个bean，替代xml中的方式（方法上）
@Configuration	声明当前类为配置类，其中内部组合了@Component注解，表明这个类是一个bean（类上）
@ComponentScan	用于对Component进行扫描，相当于xml中的（类上）

java配置类相关注解	
@Bean	注解在方法上，声明当前方法的返回值为一个bean，替代xml中的方式（方法上）
@Configuration	声明当前类为配置类，其中内部组合了@Component注解，表明这个类是一个bean（类上）
@ComponentScan	用于对Component进行扫描，相当于xml中的（类上）
切面（AOP）相关注解	
@Aspect	声明一个切面（类上） 使用@After、@Before、@Around定义建言（advice），可直接将拦截规则（切点）作为参数。
@After	在方法执行之后执行（方法上） @Before 在方法执行之前执行（方法上） @Around 在方法执行之前与之后执行（方法上）
@PointCut	声明切点 在java配置类中使用@EnableAspectJAutoProxy注解开启Spring对AspectJ代理的支持（类上）
@Value注解	
@Value 为属性注入值	注入操作系统属性@Value("#{systemProperties['os.name']}")String osName; 注入表达式结果@Value("#{ T(java.lang.Math).random() * 100 }") String randomNumber; 注入其它bean属性@Value("#{domeClass.name}")String name; 注入文件资源@Value("classpath:com/hgs/hello/test.txt")String Resource file; 注入网站资源@Value("http://www.cznovel.com")Resource url; 注入配置文件Value("${book.name}")String bookName;
异步相关	
@EnableAsync	配置类中，通过此注解开启对异步任务的支持，叙事性AsyncConfigurer接口（类上）
@Async	在实际执行的bean方法使用该注解来申明其是一个异步任务（方法上或类上所有的方法都将异步，需要@EnableAsync开启异步任务）
定时任务相关	
@EnableScheduling	在配置类上使用，开启计划任务的支持（类上）
@Scheduled	来申明这是一个任务，包括cron,fixDelay,fixRate等类型（方法上，需先开启计划任务的支持）

SpringMVC 注解

@EnableWebMvc	在配置类中开启Web MVC的配置支持，如一些ViewResolver或者MessageConverter等，若无此句，重写WebMvcConfigurerAdapter方法（用于对SpringMVC的配置）。
@Controller	声明该类为SpringMVC中的Controller
@RequestMapping	用于映射Web请求，包括访问路径和参数（类或方法上）
@ResponseBody	支持将返回值放在response内，而不是一个页面，通常用户返回json数据（返回值旁或方法上）
@RequestBody	允许request的参数在request体中，而不是在直接连接在地址后面。（放在参数前）
@PathVariable	用于接收路径参数，比如@RequestMapping("/hello/{name}")申明的路径，将注解放在参数中前，即可获取该值，通常作为Restful的接口实现方法。
@RestController	该注解为一个组合注解，相当于@Controller和@ResponseBody的组合，注解在类上，意味着，该Controller的所有方法都默认加上了@ResponseBody。
@ControllerAdvice	通过该注解，我们可以将对于控制器的全局配置放置在同一个位置，注解了@Controller的类的方法可使用@ExceptionHandler、@InitBinder、@ModelAttribute注解到方法上， 这对所有注解了 @RequestMapping的控制器内的方法有效。
@ExceptionHandler	用于全局处理控制器里的异常
@InitBinder	用来设置WebDataBinder，WebDataBinder用来自动绑定前台请求参数到Model中。
@ModelAttribute	本来的作用是绑定键值对到Model里，在@ControllerAdvice中是让全局的@RequestMapping都能获得在此处设置的键值对。

Mybatis 注解（不使用表格了，偷个懒，嘻嘻）

增删改查：@Insert、@Update、@Delete、@Select、@MapKey、@Options、@SelelctKey 、 @Param 、 @InsertProvider 、 @UpdateProvider 、 @DeleteProvider、@SelectProvider

结果集映射： @Results 、 @Result 、 @ResultMap 、 @ResultType 、 @ConstructorArgs、@Arg、@One、@Many、@TypeDiscriminator、@Case

缓存：@CacheNamespace、@Property、@CacheNamespaceRef、@Flush

SpringBoot 注解

@SpringBootApplication：申明让 spring boot 自动给程序进行必要的配置，这个配置等同于：@Configuration ，@EnableAutoConfiguration 和 @ComponentScan 三个配置。

@ResponseBody：表示该方法的返回结果直接写入 HTTP response body 中，一般在异步获取数据时使用，用于构建 RESTful 的 api。在使用@RequestMapping 后，返回值通常解析为跳转路径，加上@esponsebody 后返回结果不会被解析为跳转路径，而是直接写入 HTTP response body 中。比如异步获取 json 数据，加上@Responsebody 后，会直接返回 json 数据。该注解一般会配合@RequestMapping 一起使用。

@Controller：用于定义控制器类，在 spring 项目中由控制器负责将用户发来的 URL 请求转发到对应的服务接口（service 层），一般这个注解在类中，通常方法需要配合注解@RequestMapping。

@RestController：用于标注控制层组件(如 struts 中的 action)，@ResponseBody 和@Controller 的合集。

@RequestMapping：提供路由信息，负责 URL 到 Controller 中的具体函数的映射。

@EnableAutoConfiguration：SpringBoot 自动配置（auto-configuration）：尝试根据你添加的 jar 依赖自动配置你的 Spring 应用。例如，如果你的 classpath 下存在 HSQLDB，并且你没有手动配置任何数据库连接 beans，那么我们将自动配置一个内存型（in-memory）数据库”。你可以将@EnableAutoConfiguration 或者@SpringBootApplication 注解添加到一个@Configuration 类上来选择自动配置。如果发现应用了你不想要的特定自动配置类，你可以使用@EnableAutoConfiguration 注解的排除属性来禁用它们。

@ComponentScan：表示将该类自动发现扫描组件。个人理解相当于，如果扫描到有@Component、@Controller、@Service 等这些注解的类，并注册为 Bean，可以自动收集所有的 Spring 组件，包括@Configuration 类。我们经常使用@ComponentScan 注解搜索 beans，并结合@Autowired 注解导入。可以自动收集所有的 Spring 组件，包括@Configuration 类。我们经常使用@ComponentScan 注解搜索 beans，并结合@Autowired 注解导入。如果没有配置的话，Spring Boot 会扫描启动类所在包下以及子包下的使用了@Service,@Repository 等注解的类。

@Configuration：相当于传统的 xml 配置文件，如果有些第三方库需要用到 xml 文件，建议仍然通过@Configuration 类作为项目的配置主类——可以使用@ImportResource 注解加载 xml 配置文件。

@Import：用来导入其他配置类。

@ImportResource：用来加载 xml 配置文件。

@Repository：使用@Repository 注解可以确保 DAO 或者 repositories 提供异常转译，这个注解修饰的 DAO 或者 repositories 类会被 ComponetScan 发现并配置，同时也不需要为它们提供 XML 配置项。

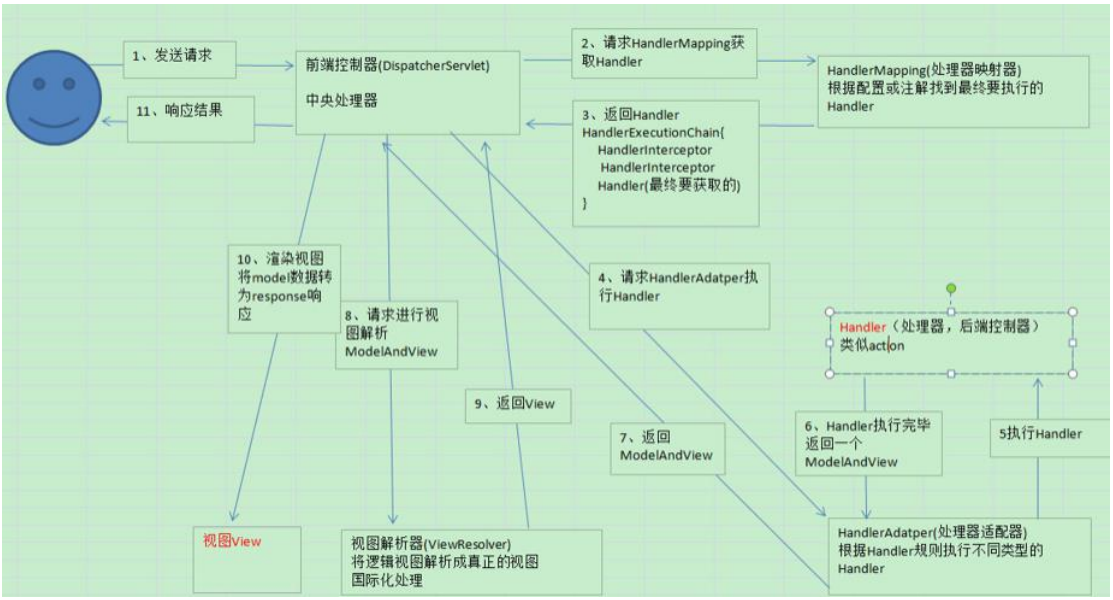
@Bean：用@Bean 标注方法等价于 XML 中配置的 bean

@AutoWired：自动导入依赖的 bean。byType 方式。把配置好的 Bean 拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作。当加上（required=false）时，就算找不到 bean 也不报错。

@Qualifier：当有多个同一类型的 Bean 时，可以用@Qualifier(“name”)来指定。与@Autowired 配合使用。@Qualifier 限定描述符除了能根据名字进行注入，但能进行更细粒度的控制如何选择候选者，具体使用方式如下：

@Resource(name=”name”,type=”type”)：没有括号内内容的话，默认 byName。与@Autowired 干类似的事。

SpringMVC 的工作原理



SpringBoot 框架的优点

--创建独立的 Spring 应用程序 ；
--嵌入的 Tomcat 、 Jetty 或者 Undertow，无须部署 WAR 文件：
--允许通过 Maven 来根据需要获取 starter;
--尽可能地自动配置 Spring;
--提供生产就绪型功能，如指标、健康检查和外部配置；
--绝对没有代码生成，对 XML 没有要求配置 。

MyBatis 框架的优点

JDBC 相比，减少了 50%以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接
很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持，而 JDBC 提供了可扩展性，所以只要这个数据库有针对 Java 的 jar 包就可以就可以与 MyBatis 兼容），开发人员不需要考虑数据库的差异性。
提供了很多第三方插件（分页插件 / 逆向工程）
SQL 写在 XML 里，从程序代码中彻底分离，解除 sql 与程序代码的耦合，便于统一管理和优化，并可重用。
提供映射标签，支持对象与数据库的 ORM 字段关系映射。
原文链接：https://blog.csdn.net/java_wxid/article/details/105087259

4. SpringCould 组件有哪些，他们的作用是什么？（说七八个）微服务的 CAP 是什么？BASE 是什么？

先讲五大核心组件

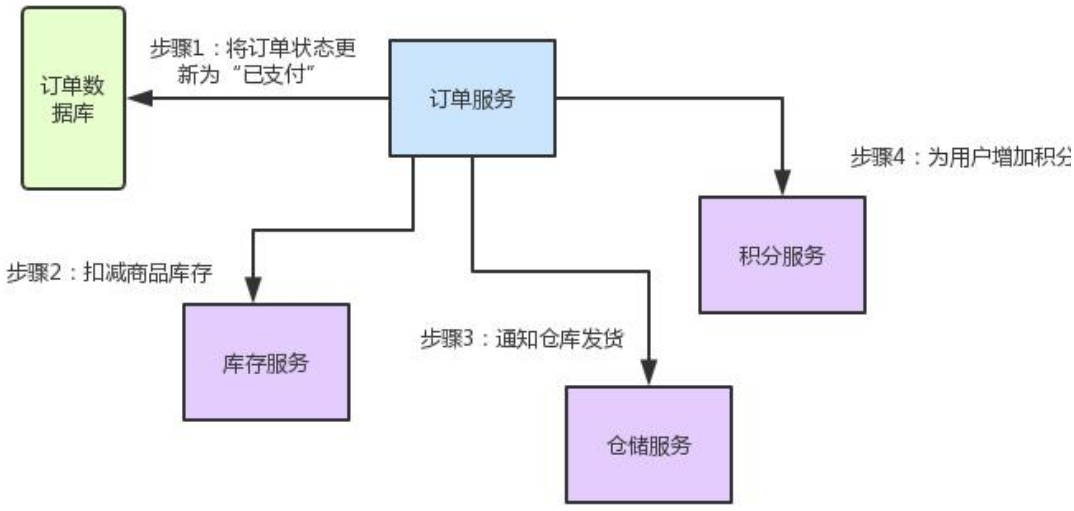
一、业务场景介绍

先来给大家说一个业务场景，假设咱们现在开发一个电商网站，要实现支付订单的功能，流程如下：

创建一个订单后，如果用户立刻支付了这个订单，我们需要将订单状态更新为“已支付”
　　扣减相应的商品库存
　　通知仓储中心，进行发货
　　给用户的这次购物增加相应的积分
针对上述流程，我们需要有订单服务、库存服务、仓储服务、积分服务。整个流程的大体思路如下：

用户针对一个订单完成支付之后，就会去找订单服务，更新订单状态
　　订单服务调用库存服务，完成相应功能
　　订单服务调用仓储服务，完成相应功能
　　订单服务调用积分服务，完成相应功能
至此，整个支付订单的业务流程结束

下图这张图，清晰表明了各服务间的调用过程：



好！有了业务场景之后，咱们就一起来看看 Spring Cloud 微服务架构中，这几个组件如何相互协作，各自发挥的作用以及其背后的原理。

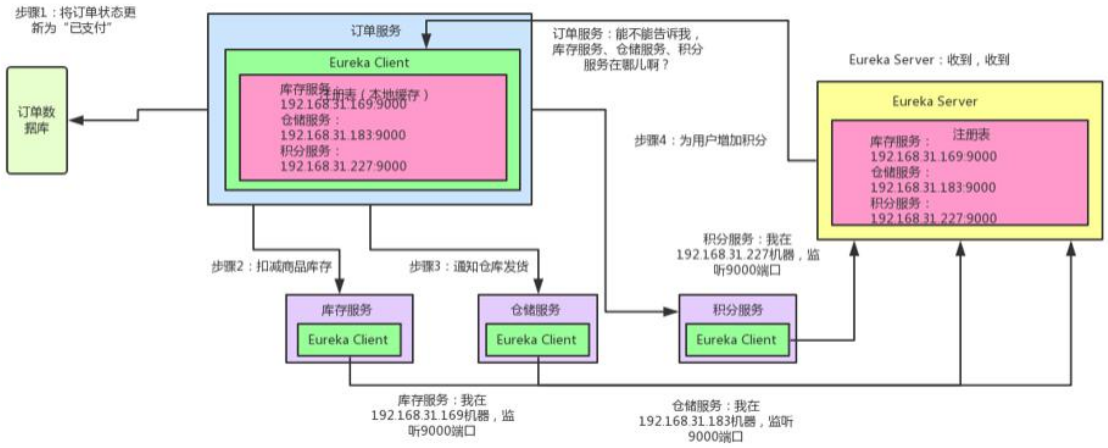
二、Spring Cloud 核心组件：Eureka

咱们来考虑第一个问题：订单服务想要调用库存服务、仓储服务，或者积分服务，怎么调用？

订单服务压根儿就不知道人家库存服务在哪台机器上啊！他就算想要发起一个请求，都不知道发送给谁，有心无力！

这时候，就轮到 Spring Cloud Eureka 出场了。Eureka 是微服务架构中的注册中心，专门负责服务的注册与发现。

咱们来看看下面的这张图，结合图来仔细剖析一下整个流程：



如上图所示，库存服务、仓储服务、积分服务中都有一个 Eureka Client 组件，这个组件专门负责将这个服务的信息注册到 Eureka Server 中。说白了，就是告诉 Eureka Server，自己在哪台机器上，监听着哪个端口。而 Eureka Server 是一个注册中心，里面有一个注册表，保存了各服务所在的机器和端口号

订单服务里也有一个 Eureka Client 组件，这个 Eureka Client 组件会找 Eureka Server 问一下：库存服务在哪台机器啊？监听着哪个端口啊？仓储服务呢？积分服务呢？然后就可以把这些相关信息从 Eureka Server 的注册表中拉取到自己本地缓存起来。

这时如果订单服务想要调用库存服务，不就可以找自己本地的 Eureka Client 问一下库存服务在哪台机器？监听哪个端口吗？收到响应后，紧接着就可以发送一个请求过去，调用库存服务扣减库存的那个接口！同理，如果订单服务要调用仓储服务、积分服务，也是如法炮制。

总结一下：

- Eureka Client：负责将这个服务的信息注册到 Eureka Server 中
- Eureka Server：注册中心，里面有一个注册表，保存了各个服务所在的机器和端口号

三、Spring Cloud 核心组件：Feign

现在订单服务确实知道库存服务、积分服务、仓库服务在哪里了，同时也监听着哪些端口号了。但是新问题又来了：难道订单服务要自己写一大堆代码，跟其他服务建立网络连接，然后构造一个复杂的请求，接着发送请求过去，最后对返回的响应结果再写一大堆代码来处理吗？

这是上述流程翻译的代码片段，咱们一起来看看，体会一下这种绝望而无助的感受！！！

友情提示，前方高能：

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpPost httpPost = new HttpPost("http://192.168.31.169:9000/");

List<NameValuePair> parameters = new ArrayList<NameValuePair>();
parameters.add(new BasicNameValuePair("scope", "project"));
parameters.add(new BasicNameValuePair("q", "java"));

UrlEncodedFormEntity formEntity = new UrlEncodedFormEntity(parameters);
httpPost.setEntity(formEntity);
httpPost.setHeader(
"User-Agent",
"Mozilla/5.0 (Windows NT 6.3; Win64; x64)"
);

CloseableHttpResponse response = null;
response = httpclient.execute(httpPost);
if (response.getStatusLine().getStatusCode() == 200) {
    String content = EntityUtils.toString(response.getEntity(), "UTF-8");
    System.out.println(content);
}
if (response != null) {
    response.close();
}
httpclient.close();
```

看完上面那一大段代码，有没有感到后背发凉、一身冷汗？实际上你进行服务间调用时，如果每次都手写代码，代码量比上面那段要多至少几倍，所以这个事压根儿就不是地球人能干的。

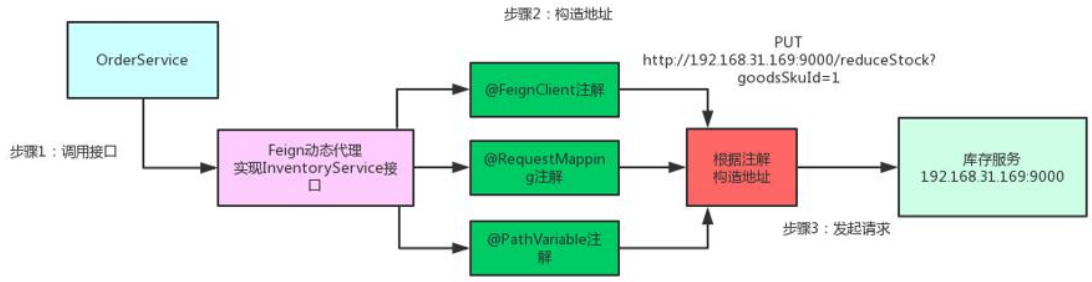
既然如此，那怎么办呢？别急，Feign 早已为我们提供好了优雅的解决方案。来看看如果用 Feign 的话，你的订单服务调用库存服务的代码会变成啥样？

```java
@FeignClient("inventory-service")
public class InventoryService {
    @RequestMapping(value = "/reduceStock/{goodsSkuId}", method = HttpMethod.PUT)
    @ResponseBody
    public ResultCode reduceStock(@PathVariable("goodsSkuId") Long goodsSkuId);
}

@Service
public class OrderService {
    @Autowired
    private InventoryService inventoryService;

    public ResultCode payOrder() {
        // 步骤1：更新本地数据库订单状态为“已支付”
        orderDAO.updateStatus(id, OrderStatus.PAYED);
        // 步骤2：调用库存服务，扣减商品库存
        inventoryService.reduceStock(goodsSkuId);
    }
}
```

看完上面的代码什么感觉？是不是感觉整个世界都干净了，又找到了活下去的勇气！没有底层的建立连接、构造请求、解析响应的代码，直接就是用注解定义一个 FeignClient 接口，然后调用那个接口就可以了。人家 Feign Client 会在底层根据你的注解，跟你指定的服务建立连接、构造请求、发起晴求、获取响应、解析响应，等等。这一系列脏活累活，人家 Feign 全给你干了。

那么问题来了，Feign 是如何做到这么神奇的呢？很简单，Feign 的一个关键机制就是使用了动态代理。咱们一起来看看下面的图，结合图来分析：

- 首先，如果你对某个接口定义了@FeignClient 注解，Feign 就会针对这个接口创建一个动态代理
- 接着你要是调用那个接口，本质就是会调用 Feign 创建的动态代理，这是核心中的核心
- Feign 的动态代理会根据你在接口上的@RequestMapping 等注解，来动态构造出你要请求的服务的地址
- 最后针对这个地址，发起请求、解析响应



四、Spring Cloud 核心组件：Ribbon

说完了 Feign，还没完。现在新的问题又来了，如果人家库存服务部署在了 5 台机器上，如下所示：

- 192.168.169:9000
- 192.168.170:9000
- 192.168.171:9000
- 192.168.172:9000
- 192.168.173:9000

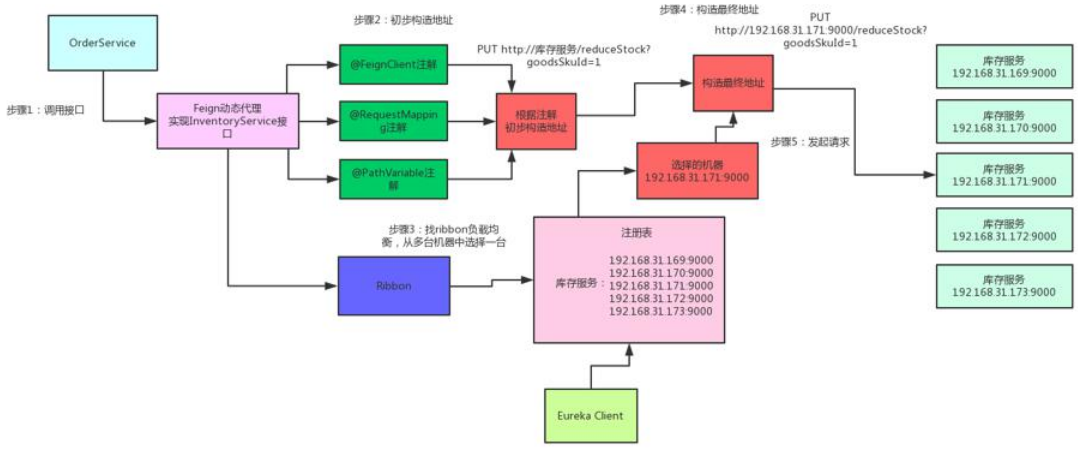
这下麻烦了！人家 Feign 怎么知道该请求哪台机器呢？

- 这时 Spring Cloud Ribbon 就派上用场了。Ribbon 就是专门解决这个问题的。它的作用是负载均衡，会帮你在每次请求时选择一台机器，均匀的把请求分发到各个机器上
- Ribbon 的负载均衡默认使用的最经典的 Round Robin 轮询算法。这是啥？简单来说，就是如果订单服务对库存服务发起 10 次请求，那就先让你请求第 1 台机器、然后是第 2 台机器、第 3 台机器、第 4 台机器、第 5 台机器，接着再来一个循环，第 1 台机器、第 2 台机器。。。以此类推。

此外，Ribbon 是和 Feign 以及 Eureka 紧密协作，完成工作的，具体如下：

- 首先 Ribbon 会从 Eureka Client 里获取到对应的服务注册表，也就知道了所有的服务都部署在了哪些机器上，在监听哪些端口号。
- 然后 Ribbon 就可以使用默认的 Round Robin 算法，从中选择一台机器
- Feign 就会针对这台机器，构造并发起请求。

对上述整个过程，再来一张图，帮助大家更深刻的理解：



五、Spring Cloud 核心组件：Hystrix

在微服务架构里，一个系统会有很多的服务。以本文的业务场景为例：订单服务在一个业务

流程里需要调用三个服务。现在假设订单服务自己最多只有 100 个线程可以处理请求，然后呢，积分服务不幸的挂了，每次订单服务调用积分服务的时候，都会卡住几秒钟，然后抛出一个超时异常。

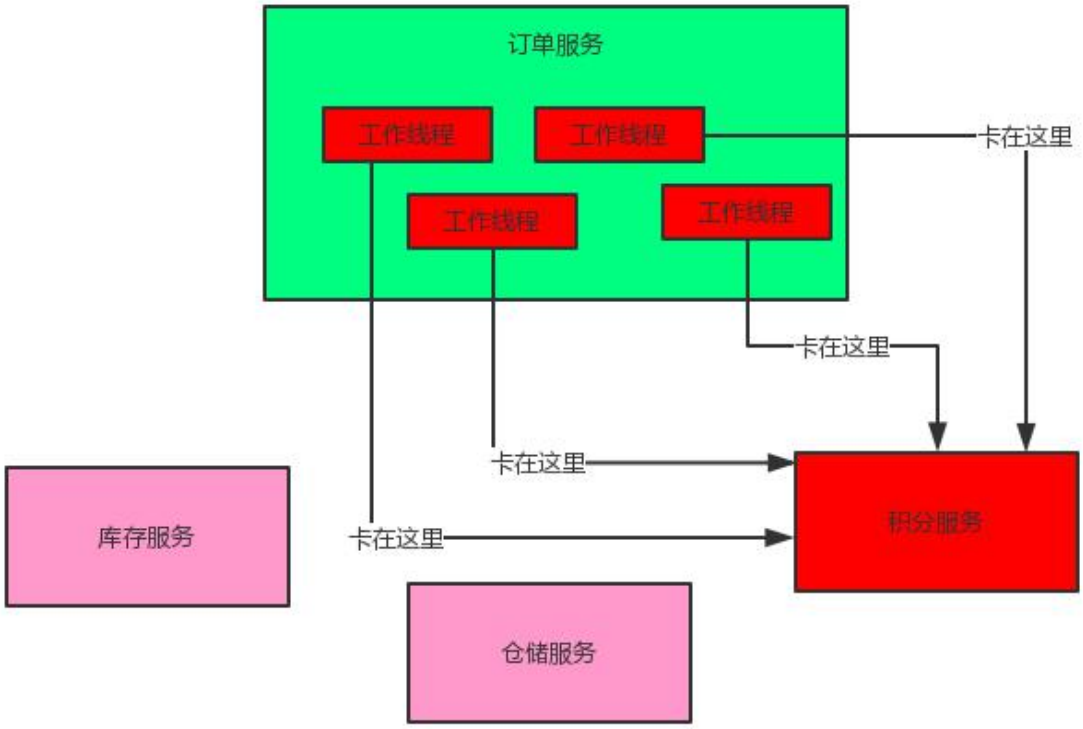
咱们一起来分析一下，这样会导致什么问题？

如果系统处于高并发的场景下，大量请求涌过来的时候，订单服务的 100 个线程都会卡在请求积分服务这块。导致订单服务没有一个线程可以处理请求

然后就会导致别人请求订单服务的时候，发现订单服务也挂了，不响应任何请求了

上面这个，就是微服务架构中恐怖的服务雪崩问题，

如下图所示：



如上图，这么多服务互相调用，要是不做任何保护的话，某一个服务挂了，就会引起连锁反应，导致别的服务也挂。比如积分服务挂了，会导致订单服务的线程全部卡在请求积分服务这里，没有一个线程可以工作，瞬间导致订单服务也挂了，别人请求订单服务全部会卡住，无法响应。

但是我们思考一下，就算积分服务挂了，订单服务也可以不用挂啊！为什么？

- 我们结合业务来看：支付订单的时候，只要把库存扣减了，然后通知仓库发货就 OK 了
- 如果积分服务挂了，大不了等他恢复之后，慢慢人肉手工恢复数据！为啥一定要因为一个积分服务挂了，就直接导致订单服务也挂了呢？不可以接受！

现在问题分析完了，如何解决？

这时就轮到 Hystrix 闪亮登场了。Hystrix 是隔离、熔断以及降级的一个框架。啥意思呢？说白了，Hystrix 会搞很多个小小的线程池，比如订单服务请求库存服务是一个线程池，请求仓储服务是一个线程池，请求积分服务是一个线程池。每个线程池里的线程就仅仅用于请求那个服务。

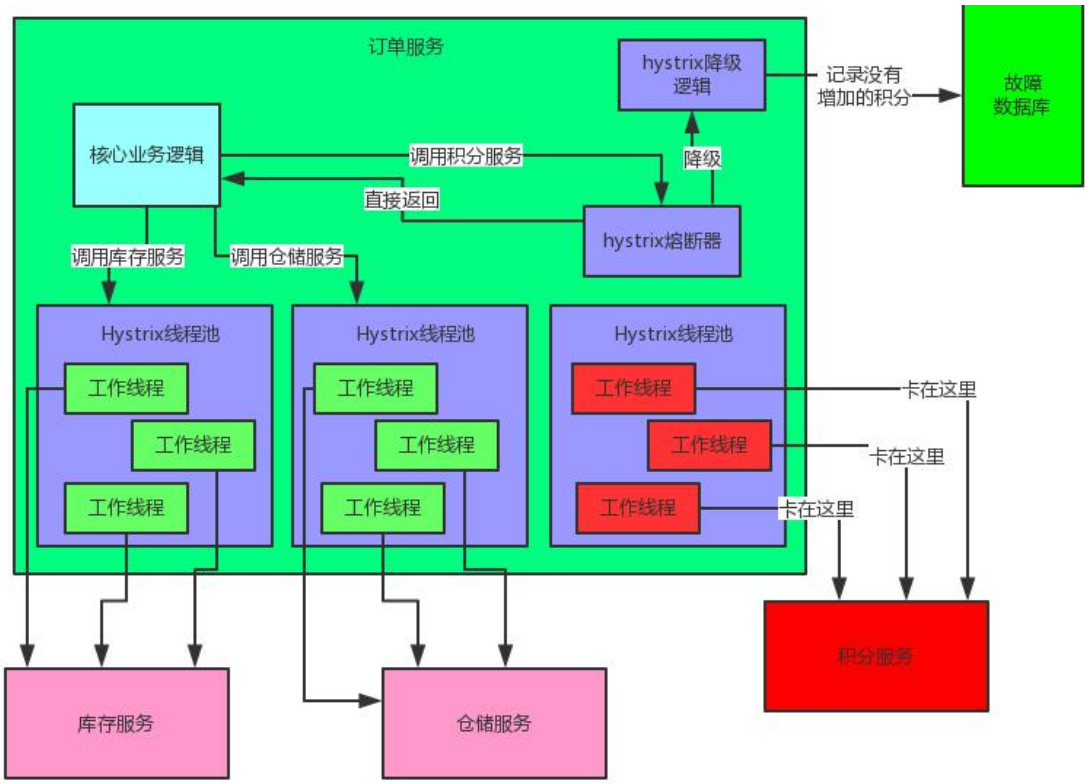
打个比方：现在很不幸，积分服务挂了，会咋样？

当然会导致订单服务里那个用来调用积分服务的线程都卡死不能工作了啊！但由于订单服务调用库存服务、仓储服务的这两个线程池都是正常工作的，所以这两个服务不会受到任何影响。

这个时候如果别人请求订单服务，订单服务还是可以正常调用库存服务扣减库存，调用仓储服务通知发货。只不过调用积分服务的时候，每次都会报错。但是如果积分服务都挂了，每次调用都要去卡住几秒钟干啥呢？有意义吗？当然没有！所以我们直接对积分服务熔断不就得了，比如在 5 分钟内请求积分服务直接就返回了，不要去走网络请求卡住几秒钟，这个过程，就是所谓的熔断！

那人家又说，兄弟，积分服务挂了你就熔断，好歹你干点儿什么啊！别啥都不干就直接返回啊？没问题，咱们就来个降级：每次调用积分服务，你就在数据库里记录一条消息，说给某某用户增加了多少积分，因为积分服务挂了，导致没增加成功！这样等积分服务恢复了，你可以根据这些记录手工加一下积分。这个过程，就是所谓的降级。

为帮助大家更直观的理解，接下来用一张图，梳理一下 Hystrix 隔离、熔断和降级的全流程：



六、Spring Cloud 核心组件：Zuul

说完了 Hystrix，接着给大家说说最后一个组件：Zuul，也就是微服务网关。这个组件是负责网络路由的。不懂网络路由？行，那我给你说说，如果没有 Zuul 的日常工作会怎样？

假设你后台部署了几百个服务，现在有个前端兄弟，人家请求是直接从浏览器那儿发过来的。打个比方：人家要请求一下库存服务，你难道还让人家记着这服务的名字叫做 inventory-service？部署在 5 台机器上？就算人家肯记住这一个，你后台可有几百个服务的名称和地址呢？难不成人家请求一个，就得记住一个？你要这样玩儿，那真是友谊的小船，说翻就翻！

上面这种情况，压根儿是不现实的。所以一般微服务架构中都必然会设计一个网关在里面，像 android、ios、pc 前端、微信小程序、H5 等等，不用去关心后端有几百个服务，就知道有一个网关，所有请求都往网关走，网关会根据请求中的一些特征，将请求转发给后端的各个服务。

而且有一个网关之后，还有很多好处，比如可以做统一的降级、限流、认证授权、安全，等等。

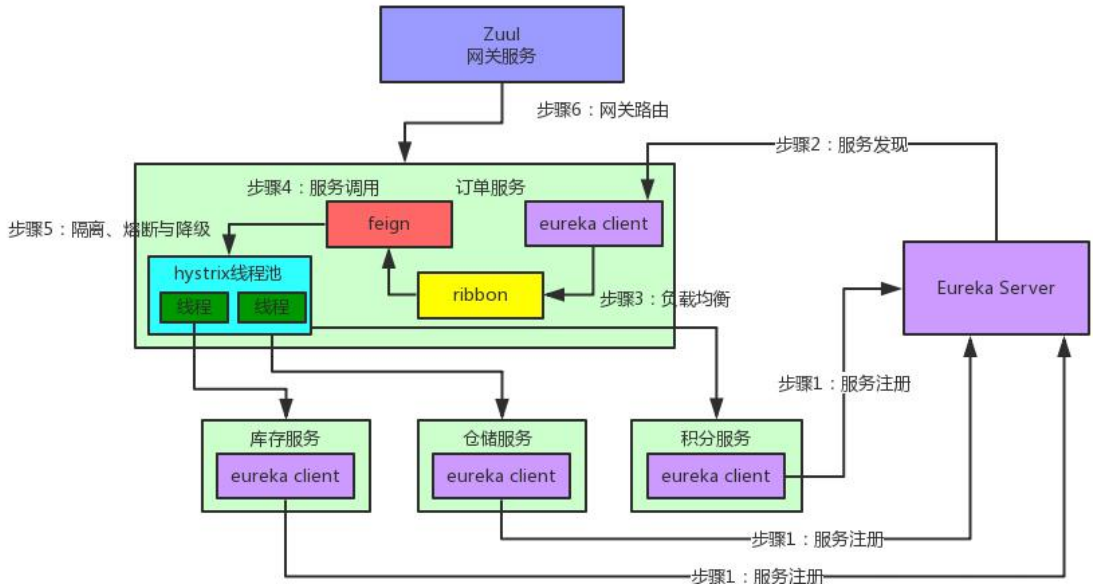
七、总结：

最后再来总结一下，上述几个 Spring Cloud 核心组件，在微服务架构中，分别扮演的角色：

- Eureka：各个服务启动时，Eureka Client 都会将服务注册到 Eureka Server，并且 Eureka Client 还可以反过来从 Eureka Server 拉取注册表，从而知道其他服务在哪里
- Ribbon：服务间发起请求的时候，基于 Ribbon 做负载均衡，从一个服务的多台机器中选择一台
- Feign：基于 Feign 的动态代理机制，根据注解和选择的机器，拼接请求 URL 地址，发起请求
- Hystrix：发起请求是通过 Hystrix 的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题
- Zuul：如果前端、移动端要调用后端系统，统一从 Zuul 网关进入，由 Zuul 网关转发请求给对应的服务

以上就是我们通过一个电商业务场景，阐述了 Spring Cloud 微服务架构几个核心组件的底层原理。

文字总结还不够直观？没问题！我们将 Spring Cloud 的 5 个核心组件通过一张图串联起来，再来直观的感受一下其底层的架构原理：



五大核心组件讲完了，面试官心中已经知道你对 SpringCould 的有一定的了解了，但这还不够，你如果讲到这个层面，部分面试官还会继续问，因为你讲解的这些其他面试者也讲过，可能也就你讲的比较细一些，但本质还是和他们差不了太多，有些公司可能集中招人，负责面试的可能就一个，你想想他这一天可以面试多少个人，这个时候你就需要继续拓展其他组件，来突出你的不同了。

Spring Cloud Sleuth（服务链路追踪），Spring Cloud Bus（消息总线），Spring Cloud Config（分布式配置中心）之类的，这里我就不继续写了，给上一个 SpringCould 专栏（一位大佬写的，挺不错的）你去看看吧，最好能实现动手敲上一套，后面你会发现自己对 SpringCould 的理解远超其他人。专栏地址是：https://blog.csdn.net/forezp/article/details/70148833

CAP 定论

一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。C 一致性即更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致。A 可用性服务一直可用，而且是正常响应时间。P 分区容错性即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

- 对于多数大型互联网应用的场景，一般保证满足 P 和 A，舍弃 C（一致性无法保证，退而求其次保证最终一致性）。虽然某些地方会影响客户体验，但没达到造成用户流失的严重程度。如原来同步架构的时候如果没有库存，就马上告诉客户库存不足无法下单。但在微服务框架下订单和库存可能是两个微服务对应两个数据库，用户下单时订单服务是立即生成的，很可能过了一会系统通知你订单被取消掉（最终一致性）。就像抢购“小米手机”一样，几十万人在排队，排了很久告诉你没货了，明天再来吧。
- 对于涉及到钱财这样不能有一丝让步的场景，C 必须保证。网络发生故障宁可停止服务，这是保证 CA，舍弃 P。
- 还有一种是保证 CP，舍弃 A。例如网络故障事只读不写。

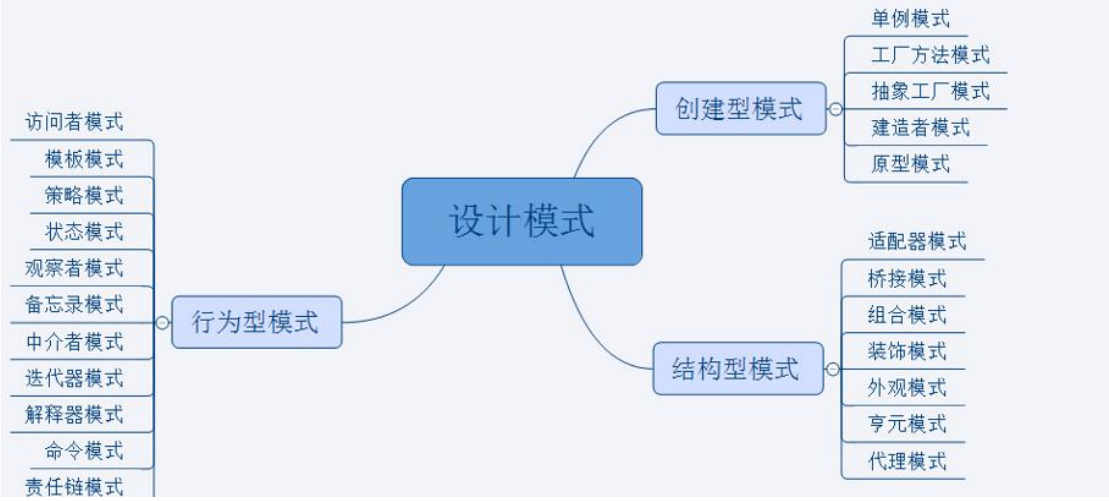
BASE

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。是对 CAP 中 AP 的一个扩展

- 基本可用：分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。
- 软状态：允许系统中存在中间状态，这个状态不影响系统可用性，这里指的是 CAP 中的不一致。
- 最终一致：最终一致是指经过一段时间后，所有节点数据都将会达到一致。

BASE 解决了 CAP 中理论没有网络延迟，在 BASE 中用软状态和最终一致，保证了延迟后的一致性。BASE 和 ACID 是相反的，它完全不同于 ACID 的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。

5. 设计模式（说五六个）



根据目的来分

根据模式是用来完成什么工作来划分，这种方式可分为创建型模式、结构型模式和行为型模式 3 种。

- 创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。
- 结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。
- 行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

根据作用范围来分

根据模式是主要用于类上还是主要用于对象上来分，这种方式可分为类模式和对象模式两种。

- 类模式：用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。GoF 中的工厂方法、（类）适配器、模板方法、解释器属于该模式。
- 对象模式：用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时刻是可以变化的，更具动态性。GoF 中除了以上 4 种，其他的都是对象模式。

设计模式的功能

1、FACTORY 工厂方法：

追 MM 少不了请吃饭了，麦当劳的鸡翅和肯德基的鸡翅都是 MM 爱吃的东西，虽然口味有所不同，但不管你带 MM 去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。麦当劳和肯德基就是生产鸡翅的 Factory 工厂模式：客户类和工厂类分开。消费者任何时候需要某种产品，只需向工厂请求即可。消费者无须修改就可以接纳新产品。缺点是当产品修改时，工厂类也要做相应的修改。如：如何创建及如何向客户端提供。

2、BUILDER 建造者模式

MM 最爱听的就是“我爱你”这句话了，见到不同地方的 MM,要能够用她们的方言跟她说这句话哦，我有一个多种语言翻译机，上面每种语言都有一个按键，见到 MM 我只要按对应的键，它就能够用相应的语言说出“我爱你”这句话了，国外的 MM 也可以轻松搞掂，这就是我的“我爱你”builder。（这一定比美军在伊拉克用的翻译机好卖） 建造模式：将产品的内部表象和产品的生成过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。建造模式使得产品内部表象可以独立的变化，客户不必知道产品内部组成的细节。建造模式可以强制实行一种分步骤进行的建造过程。

3、FACTORY METHOD 抽象工厂

请 MM 去麦当劳吃汉堡，不同的 MM 有不同的口味，要每个都记住是一件烦人的事情，我一般采用 Factory Method 模式，带着 MM 到服务员那儿，说“要一个汉堡”，具体要什么样的汉堡呢，让 MM 直接跟服务员说就行了。 工厂方法模式：核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

4、PROTOTYPE 原型模式

跟 MM 用 QQ 聊天，一定要说些深情的话语了，我搜集了好多肉麻的情话，需要时只要 copy 出来放到 QQ 里面就行了，这就是我的情话 prototype 了。（100 块钱一份，你要不要） 原始模型模式：通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的方法创建出更多同类型的对象。原始模型模式允许动态的增加或减少产品类，产品类不需要非得有任何事先确定的等级结构，原始模型模式适用于任何的等级结构。缺点是每一个类都必须配备一个克隆方法。

5、SINGLETON 单态模式

俺有 6 个漂亮的老婆，她们的老公都是我，我就是我们家里的老公 Sigleton，她们只要说道“老公”，都是指的同一个人，那就是我(刚才做了个梦啦，哪有这么好的事) 单例模式：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式。单例模式只应在有真正的“单一实例”的需求时才可使用。 [b:9ceca65206]结构型模式[/b:9ceca65206]

6、ADAPTER 适配器模式

在朋友聚会上碰到了一个美女 Sarah，从香港来的，可我不会说粤语，她不会说普通话，只好求助于我的朋友 kent 了，他作为我和 Sarah 之间的 Adapter，让我和 Sarah 可以相互交谈了(也不知道他会不会耍我) 适配器（变压器）模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作。适配类可以根据参数返还一个合适的实例给客户端。

7、BRIDGE 桥梁模式

早上碰到 MM，要说早上好，晚上碰到 MM，要说晚上好；碰到 MM 穿了件新衣服，要说你的衣服好漂亮哦，碰到 MM 新做的发型，要说你的头发好漂亮哦。不要问我“早上碰到 MM 新做了个发型怎么说”这种问题，自己用 BRIDGE 组合一下不就行了 桥梁模式：将抽象化与实现化脱耦，使得二者可以独立的变化，也就是说将他们之间的强关联变成弱关联，也就是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以独立的变化。

8、COMPOSITE 合成模式

Mary 今天过生日。“我过生日，你要送我一件礼物。”“嗯，好吧，去商店，你自己挑。”

“这件 T 恤挺漂亮，买，这条裙子好看，买，这个包也不错，买。”“喂，买了三件了呀，我只答应送一件礼物的哦。”“什么呀，T 恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。”“……”，MM 都会用 Composite 模式了，你会了没有？ 合成模式：合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式就是一个处理对象的树结构的模式。合成模式把部分与整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由他们复合而成的合成对象同等看待。

9、DECORATOR 装饰模式

Mary 过完轮到 Sarly 过生日，还是不要叫她自己挑了，不然这个月伙食费肯定玩完，拿出我去年在华山顶上照的照片，在背面写上“最好的的礼物，就是爱你的 Fita”，再到街上礼品店买了个像框（卖礼品的 MM 也很漂亮哦），再找隔壁搞美术设计的 Mike 设计了一个漂亮的盒子装起来……，我们都是 Decorator，最终都在修饰我这个人呀，怎么样，看懂了吗？ 装饰模式：装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。动态给一个对象增加功能，这些功能可以再动态的撤消。1 增加由一些基本功能的排列组合而产生的非常大量的功能。

10、FACADE 门面模式

我有一个专业的 Nikon 相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但 MM 可不懂这些，教了半天也不会。幸好相机有 Facade 设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样 MM 也可以用这个相机给我拍张照片了。 门面模式：外部与一个子系统的通信必须通过一个统一的门面对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。每一个子系统只有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式。但整个系统可以有多个门面类。

11、FLYWEIGHT 享元模式

每天跟 MM 发短信，手指都累死了，最近买了个新手机，可以把一些常用的句子存在手机里，要用的时候，直接拿出来，在前面加上 MM 的名字就可以发送了，再不用一个字一个字敲了。共享的句子就是 Flyweight，MM 的名字就是提取出来的外部特征，根据上下文情况使用。 享元模式：FLYWEIGHT 在拳击比赛中指最轻量级。享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来，将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。享元模式大幅度的降低内存中对象的数量。

12、PROXY 代理模式

跟 MM 在网上聊天，一开头总是“hi,你好”,“你从哪儿来呀？”“你多大了？”“身高多少呀？”这些话，真烦人，写个程序做为我的 Proxy 吧，凡是接收到这些话都设置好了自己的回答，接收到其他的话时再通知我回答，怎么样，酷吧。 代理模式：代理模式给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用。代理就是一个人或一个机构代表另一个人或者一个机构采取行动。某些情况下，客户不想或者不能够直接引用一个对象，代理对象可以在客户和目标对象直接起到中介的作用。客户端分辨不出代理主题对象与真实主题对象。代理模式可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口，这时候代理对象不能够创建被代理对象，被代理对象必须有系统的其他角色代为创建并传入。

13、CHAIN OF RESPONSIBLEITY 责任链模式

晚上去上英语课，为了好开溜坐到了最后一排，哇，前面坐了好几个漂亮的 MM 哎，找张纸条，写上“Hi,可以做我的女朋友吗？如果不愿意请向前传”，纸条就一个接一个的传上去了，糟糕，传到第一排的 MM 把纸条传给老师了，听说是个老处女呀，快跑！责任链模式：在责任链模式中，很多对象由每一个对象对其下家的引用而接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。客户并不知道链上的哪一个对象最终处理这个请求，系统可以在不影响客户端的情况下动态的重新组织链和分配责任。处理者有两个选择：承担责任或者把责任推给下家。一个请求可以最终不被任何接收端对象所接受。

14、COMMAND 命令模式

俺有一个 MM 家里管得特别严，没法见面，只好借助于她弟弟在我们俩之间传送信息，她对我有什么指示，就写一张纸条让她弟弟带给我。这不，她弟弟又传送过来一个 COMMAND，为了感谢他，我请他吃了碗杂酱面，哪知道他说：“我同时给我姐姐三个男朋友送 COMMAND，就数你最小气，才请我吃面。” 命令模式：命令模式把一个请求或者操作封装到一个对象中。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。命令模式允许请求的一方和发送的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否执行，何时被执行以及是怎么被执行的。系统支持命令的撤消。

15、INTERPRETER 解释器模式

俺有一个《泡 MM 真经》，上面有各种泡 MM 的攻略，比如说去吃西餐的步骤、去看电影的方法等等，跟 MM 约会时，只要做一个 Interpreter，照着上面的脚本执行就可以了。 解释器模式：给定一个语言后，解释器模式可以定义出其文法的一种表示，并同时提供一个解

释器。客户端可以使用这个解释器来解释这个语言中的句子。解释器模式将描述怎样在有了一个简单的文法后，使用模式设计解释这些语句。在解释器模式里面提到的语言是指任何解释器对象能够解释的任何组合。在解释器模式中需要定义一个代表文法的命令类的等级结构，也就是一系列的组合规则。每一个命令对象都有一个解释方法，代表对命令对象的解释。命令对象的等级结构中的对象的任何排列组合都是一个语言。

16、ITERATOR 迭代子模式

我爱上了 Mary，不顾一切的向她求婚。 Mary：“想要我跟你结婚，得答应我的条件” 我：“什么条件我都答应，你说吧” Mary：“我看上了那个一克拉的钻石” 我：“我买，我买，还有吗？” Mary：“我看上了湖边的那栋别墅” 我：“我买，我买，还有吗？” Mary：“我看上那辆法拉利跑车” 我脑袋嗡的一声，坐在椅子上，一咬牙：“我买，我买，还有吗？” …… 迭代子模式：迭代子模式可以顺序访问一个聚集中的元素而不必暴露聚集的内部表象。多个对象聚在一起形成的总体称之为聚集，聚集对象是能够包容一组对象的容器对象。迭代子模式将迭代逻辑封装到一个独立的子对象中，从而与聚集本身隔开。迭代子模式简化了聚集的界面。每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。迭代算法可以独立于聚集角色变化。

17、MEDIATOR 调停者模式

四个 MM 打麻将，相互之间谁应该给谁多少钱算不清楚了，幸亏当时我在旁边，按照各自的筹码数算钱，赚了钱的从我这里拿，赔了钱的也付给我，一切就 OK 啦，俺得到了四个 MM 的电话。 调停者模式：调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散偶合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。调停者模式将多对多的相互作用转化为一对多的相互作用。调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

18、MEMENTO 备忘录模式

同时跟几个 MM 聊天时，一定要记清楚刚才跟 MM 说了些什么话，不然 MM 发现了会不高兴的哦，幸亏我有个备忘录，刚才与哪个 MM 说了什么话我都拷贝一份放到备忘录里面保存，这样可以随时察看以前的记录啦。 备忘录模式：备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

19、OBSERVER 观察者模式

想知道咱们公司最新 MM 情报吗？加入公司的 MM 情报邮件组就行了，tom 负责搜集情报，他发现的新情报不用一个一个通知我们，直接发布给邮件组，我们作为订阅者（观察者）就可以及时收到情报啦 观察者模式：观察者模式定义了一种一队多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使他们能够自动更新自己。

20、STATE 状态模式

跟 MM 交往时，一定要注意她的状态哦，在不同的状态时她的行为会有不同，比如你约她今天晚上去看电影，对你没兴趣的 MM 就会说“有事情啦”，对你不讨厌但还没喜欢上的 MM 就会说“好啊，不过可以带上我同事么？”，已经喜欢上你的 MM 就会说“几点钟？看完电影再去泡吧怎么样？”，当然你看电影过程中表现良好的话，也可以把 MM 的状态从不讨厌不喜欢变成喜欢哦。 状态模式：状态模式允许一个对象在其内部状态改变的时候改变行为。这个对象看上去象是改变了它的类一样。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

21、STRATEGY 策略模式

跟不同类型的 MM 约会，要用不同的策略，有的请电影比较好，有的则去吃小吃效果不错，有的去海边浪漫最合适，单目的都是为了得到 MM 的芳心，我的追 MM 锦囊中有好多 Strategy 哦。 策略模式：策略模式针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。策略模把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。由于算法和环境独立开来，算法的增减，修改都不会影响到环境和客户端。

22、TEMPLATE METHOD 模板方法模式

看过《如何说服女生上床》这部经典文章吗？女生从认识到上床的不变的步骤分为巧遇、打破僵局、展开追求、接吻、前戏、动手、爱抚、进去八大步骤(Template method)，但每个步骤针对不同的情况，都有不一样的做法，这就要看你随机应变啦(具体实现)； 模板方法模式：模板方法模式准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。先制定一个顶级逻辑框架，而将逻辑的细节留给具体的子类去实现。

23、VISITOR 访问者模式

情人节到了，要给每个 MM 送一束鲜花和一张卡片，可是每个 MM 送的花都要针对她个人的特点，每张卡片也要根据个人的特点来挑，我一个人哪搞得清楚，还是找花店老板和礼品店老板做一下 Visitor，让花店老板根据 MM 的特点选一束花，让礼品店老板也根据每个人特点选一张卡，这样就轻松多了； 访问者模式：访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由的演化。访问者模式使得增加新的操作变的很容易，就是增加一个新的访问者类。访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。当使用访问者模式时，要将尽可能多的对象浏览逻辑放在访问者类中，而不是放到它的子类中。访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。

6. Redis 支持的数据类型以及使用场景，持久化，哨兵机制，缓存击穿，缓存穿透

简单介绍一个 redis？

redis 是内存中的数据结构存储系统，一个 key-value 类型的非关系型数据库，可持久化的数据库，相对于关系型数据库（数据主要存在硬盘中），性能高，因此我们一般用 redis 来做缓存使用；并且 redis 支持丰富的数据类型，比较容易解决各种问题，因此 redis 可以用来作为注册中心， 数据库、缓存和消息中间件。Redis 的 Value 支持 5 种数据类型，string、hash、list、set、zset（sorted set）；

String 类型：一个 key 对应一个 value

Hash 类型：它的 key 是 string 类型，value 又是一个 map（key-value），适合存储对象。

List 类型：按照插入顺序的字符串链表（双向链表），主要命令是 LPUSH 和 RPUSH，能够支持反向查找和遍历

Set 类型：用哈希表类型的字符串序列，没有顺序，集合成员是唯一的，没有重复数据，底层主要是由一个 value 永远为 null 的 hashmap 来实现的。

zset 类型：和 set 类型基本一致，不过它会给每个元素关联一个 double 类型的分数（score），这样就可以为成员排序，并且插入是有序的。

你还用过其他的缓存吗？这些缓存有什么区别？都在什么场景下去用？

对于缓存了解过 redis 和 memcache

Memcache 和 redis 的区别：

数据支持的类型：redis 不仅仅支持简单的 k/v 类型的数据，同时还支持 list、set、zset、hash 等数据结构的存储；memcache 只支持简单的 k/v 类型的数据，key 和 value 都是 string 类型

可靠性：memcache 不支持数据持久化，断电或重启后数据消失，但其稳定性是有保证的；redis 支持数据持久化和数据恢复，允许单点故障，但是同时也会付出性能的代价

性能上：对于存储大数据，memcache 的性能要高于 redis

应用场景：

Memcache：适合多读少写，大数据量的情况（一些官网的文章信息等）

Redis：适用于对读写效率要求高、数据处理业务复杂、安全性要求较高的系统

案例：分布式系统，存在 session 之间的共享问题，因此在做单点登录的时候，我们利用 redis 来模拟了 session 的共享，来存储用户的信息，实现不同系统的 session 共享；

对 redis 的持久化了解不？

redis 的持久化方式有两种：

RDB（半持久化方式）：按照配置不定期的通过异步的方式、快照的形式直接把内存中的数据持久化到磁盘的一个 dump.rdb 文件（二进制的临时文件）中，redis 默认的持久化方式，它在配置文件（redis.conf）中。

优点：只包含一个文件，将一个单独的文件转移到其他存储媒介上，对于文件备份、灾难恢复而言，比较实用。

缺点：系统一旦在持久化策略之前出现宕机现象，此前没有来得及持久化的数据将会产生丢失

RDB 持久化配置：

Redis 会将数据集的快照 dump 到 dump.rdb 文件中。此外，我们也可以通过配置文件来修改 Redis 服务器 dump 快照的频率，在打开 6379.conf 文件之后，我们搜索 save，可以看到下面的配置信息：

save 900 1　　#在 900 秒(15 分钟)之后，如果至少有 1 个 key 发生变化，则 dump 内存快照。

save 300 10　　#在 300 秒(5 分钟)之后，如果至少有 10 个 key 发生变化，则 dump 内存快照。

save 60 10000　#在 60 秒(1 分钟)之后，如果至少有 10000 个 key 发生变化，则 dump 内存快照。

AOF（全持久化的方式）：

把每一次数据变化都通过 write()函数将你所执行的命令追加到一个 appendonly.aof 文件里面，Redis 默认是不支持这种全持久化方式的，需要在配置文件（redis.conf）中将 appendonly no 改成 appendonly yes

优点：数据安全性高，对日志文件的写入操作采用的是 append 模式，因此在写入过程中即使出现宕机问题，也不会破坏日志文件中已经存在的内容；

缺点：对于数量相同的数据集来说，aof 文件通常要比 rdb 文件大，因此 rdb 在恢复大数据集时的速度大于 AOF；

AOF 持久化配置:

在 Redis 的配置文件中存在三种同步方式，它们分别是：

appendfsync always　　#每次有数据修改发生时都会都调用 fsync 刷新到 aof 文件，非常慢，但是安全；

appendfsync everysec　#每秒钟都调用 fsync 刷新到 aof 文件中，很快，但是可能丢失一秒内的数据，推荐使用，兼顾了速度和安全；

appendfsync no　　#不会自动同步到磁盘上，需要依靠 OS（操作系统）进行刷新，效率快，但是安全性就比较差；

二种持久化方式区别：

AOF 在运行效率上往往慢于 RDB，每秒同步策略的效率是比较高的，同步禁用策略的效率和 RDB 一样高效；

如果缓存数据安全性要求比较高的话，用 aof 这种持久化方式（比如项目中的购物车）；

如果对于大数据集要求效率高的话，就可以使用默认的。而且这两种持久化方式可以同时使用。

做过 redis 的集群吗？你们做集群的时候搭建了几台，都是怎么搭建的？

Redis 的数据是存放在内存中的，不适合存储大数据，大数据存储一般公司常用 hadoop 中的 Hbase 或者 MogoDB。redis 主要用来处理高并发的，用我们的项目来说，电商项目如果并发大的话，一台单独的 redis 是不能足够支持我们的并发，这就需要我们扩展多台设备协同合作，即用到集群。

Redis 搭建集群的方式有多种，例如：客户端分片、Twemproxy、Codis 等，但是 redis3.0 之后就支持 redis-cluster 集群，这种方式采用的是无中心结构，每个节点保存数据和整个集群的状态，每个节点都和其他所有节点连接。如果使用的话就用 redis-cluster 集群。集群这块是公司运维搭建的，具体怎么搭建不是太了解。

我们项目中 redis 集群主要搭建了 6 台，3 主（为了保证 redis 的投票机制）3 从（高可用），每个主服务器都有一个从服务器，作为备份机。所有的节点都通过 PING-PONG 机制彼此互相连接；客户端与 redis 集群连接，只需要连接集群中的任何一个节点即可；Redis-cluster 中内置了 16384 个哈希槽，Redis-cluster 把所有的物理节点映射到【0-16383】slot 上，负责维护。

redis 有事务吗？

Redis 是有事务的，redis 中的事务是一组命令的集合，这组命令要么都执行，要不都不执行，保证一个事务中的命令依次执行而不被其他命令插入。redis 的事务是不支持回滚操作的。redis 事务的实现，需要用到 MULTI（事务的开始）和 EXEC（事务的结束）命令 ；

缓存穿透

缓存查询一般都是通过 key 去查找 value，如果不存在对应的 value，就要去数据库中查找。

如果这个 key 对应的 value 在数据库中也不存在，并且对该 key 并发请求很大，就会对数据库产生很大的压力，这就叫缓存穿透

解决方案：

1.对所有可能查询的参数以 hash 形式存储，在控制层先进行校验，不符合则丢弃。

2.将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

3. 如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存雪崩

当缓存服务器重启或者大量缓存集中在一段时间内失效，发生大量的缓存穿透，这样在失效的瞬间对数据库的访问压力就比较大，所有的查询都落在数据库上，造成了缓存雪崩。 这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

解决方案：

1.在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待。
2.可以通过缓存 reload 机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存
3.不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀
4.做二级缓存，或者双缓存策略。A1 为原始缓存，A2 为拷贝缓存，A1 失效时，可以访问 A2，A1 缓存失效时间设置为短期，A2 设置为长期。

redis 的安全机制（你们公司 redis 的安全这方面怎么考虑的？）

漏洞介绍：redis 默认情况下，会绑定在 bind 0.0.0.0:6379，这样就会将 redis 的服务暴露到公网上，如果在没有开启认证的情况下，可以导致任意用户在访问目标服务器的情况下，未授权就可访问 redis 以及读取 redis 的数据，攻击者就可以在未授权访问 redis 的情况下可以利用 redis 的相关方法，成功在 redis 服务器上写入公钥，进而可以直接使用私钥进行直接登录目标主机；

解决方案：

4. 禁止一些高危命令。修改 redis.conf 文件，用来禁止远程修改 DB 文件地址，比如 rename-command FLUSHALL "" 、rename-command CONFIG"" 、rename-command EVAL “” 等；
5. 以低权限运行 redis 服务。为 redis 服务创建单独的用户和根目录，并且配置禁止登录；
6. 为 redis 添加密码验证。修改 redis.conf 文件，添加 requirepass mypassword；
7. 禁止外网访问 redis。修改 redis.conf 文件，添加或修改 bind 127.0.0.1，使得 redis 服务只在当前主机使用；
8. 做 log 监控，及时发现攻击；
9. redis 的哨兵机制（redis2.6 以后出现的）

哨兵机制：

监控：监控主数据库和从数据库是否正常运行；

提醒：当被监控的某个 redis 出现问题的时候，哨兵可以通过 API 向管理员或者其他应用程序发送通知；

自动故障迁移：主数据库出现故障时，可以自动将从数据库转化为主数据库，实现自动切换；

具体的配置步骤参考的网上的文档。要注意的是，如果 master 主服务器设置了密码，记得在哨兵的配置文件（sentinel.conf）里面配置访问密码

redis 中对于生存时间的应用

Redis 中可以使用 expire 命令设置一个键的生存时间，到时间后 redis 会自动删除；

应用场景：

10. 设置限制的优惠活动的信息；
11. 一些及时需要更新的数据，积分排行榜；
12. 手机验证码的时间；
13. 限制网站访客访问频率；

7. 线程是什么，有几种实现方式，它们之间的区别是什么，线程池实现原理，JUC 并发包，ThreadLocal 与 Lock 和 Synchronize 区别

什么是线程？讲个故事给你听，让你没法去背这个题，地址：https://blog.csdn.net/java_wxid/article/details/94131223

有几种实现方式？

- 继承 Thread 类
- 实现 Runnable 接口
- 实现 Callable 接口
- 线程池方式

优缺点

1.继承 Thread 类

优点 、代码简单 。
缺点 、该类无法集成别的类。

2.实现 Runnable 接口

优点 、继承其他类。 同一实现该接口的实例可以共享资源。
缺点 、代码复杂

3.实现 Callable

优点 、可以获得异步任务的返回值

4. 线程池 、实现自动化装配，易于管理，循环利用资源。

代码实现案例：

继承 Thread 类，并重写里面的 run 方法

```
class A extends Thread{
	public void run(){
		for(int i=1;i<=100;i++){
			System.out.println("----------------"+i);
		}
	}
}
A a = new A();
a.start();
```

实现 Runnable 接口，并实现里面的 run 方法

```
class B implements Runnable{
	public void run(){
		for(int i=1;i<=100;i++){
			System.out.println("----------------"+i);
		}
	}
}
B b = new B();
Thread t = new Thread(b);
t.start();
```

实现 Callable

```
class A implements Callable<String>{
	public String call() throws Exception{
		//...
	}
}
FutureTask<String> ft = new FutureTask<>(new A());
new Thread(ft).start();
```

线程池

```
ExcutorService es = Executors.newFixedThreadPool(10);
es.submit(new Runnable(){//任务});
es.submit(new Runnable(){//任务});
...
es.shutdown();
```

问题扩展

在 Java 中 Lock 接口比 synchronized 块的优势是什么？你需要实现一个高效的缓存，它允许多个用户读，但只允许一个用户写，以此来保持它的完整性，你会怎样去实现它？

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

线程池的实现原理：https://blog.csdn.net/java_wxid/article/details/101844786

JUC 并发包：

volatile 的三大特性：https://blog.csdn.net/java_wxid/article/details/97611028
CompareAndSwap 底层原理：https://blog.csdn.net/java_wxid/article/details/97611037
AtomicReference 原子引用：https://blog.csdn.net/java_wxid/article/details/97611046

CountDownLatch 倒计时器：https://blog.csdn.net/java_wxid/article/details/99168098
CyclicBarrier 循环栅栏：https://blog.csdn.net/java_wxid/article/details/99171155
Semaphore 信号灯：https://blog.csdn.net/java_wxid/article/details/99174538

ThreadLocal 与 Lock 和 Synchronize 区别

ThreadLocal 与 Lock 和 Synchronize 区别
ThreadLocal 为每一个线程都提供了变量的副本，使得每个线程在某一时间访问到的并不是同一个对象，这样就隔离了多个线程对数据的数据共享。ThreadLocal 采用了“以空间换时间”的方式，为每一个线程都提供了一份变量，因此可以同时访问而互不影响。
synchronized 是利用锁的机制，使变量或代码块在某一时该只能被一个线程访问。同步机制采用了“以时间换空间”的方式，仅提供一份变量，让不同的线程排队访问。
如果一个代码块被 synchronized 关键字修饰，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待直至占有锁的线程释放锁。事实上，占有锁的线程释放锁一般会是以下三种情况之一：
占有锁的线程执行完了该代码块，然后释放对锁的占有；
占有锁线程执行发生异常，此时 JVM 会让线程自动释放锁；
占有锁线程进入 WAITING 状态从而释放锁，例如在该线程中调用 wait()方法等。
synchronized 是 Java 语言的内置特性，可以轻松实现对临界资源的同步互斥访问。那么，为什么还会出现 Lock 呢？试考虑以下三种情况：
Case 1 ：
在使用 synchronized 关键字的情形下，假如占有锁的线程由于要等待 IO 或者其他原因（比如调用 sleep 方法）被阻塞了，但是又没有释放锁，那么其他线程就只能一直等待，别无他法。这会极大影响程序执行效率。因此，就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间 (解决方案：tryLock(long time, TimeUnit unit)) 或者 能够响应中断 (解决方案：lockInterruptibly())），这种情况可以通过 Lock 解决。
Case 2 ：
我们知道，当多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作也会发生冲突现象，但是读操作和读操作不会发生冲突现象。但是如果采用 synchronized 关键字实现同步的话，就会导致一个问题，即当多个线程都只是进行读操作时，也只有一个线程在可以进行读操作，其他线程只能等待锁的释放而无法进行读操作。因此，需要一种机制来使得当多个线程都只是进行读操作时，线程之间不会发生冲突。同样地，Lock 也可以解决这种情况 (解决方案：ReentrantReadWriteLock) 。
Case 3 ：
我们可以通过 Lock 得知线程有没有成功获取到锁 (解决方案：ReentrantLock) ，但这个是 synchronized 无法办到的。
上面提到的三种情形，我们都可以通过 Lock 来解决，但 synchronized 关键字却无能为力。事实上，Lock 是 java.util.concurrent.locks 包 下的接口，Lock 实现提供了比 synchronized 关键字 更广泛的锁操作，它能以更优雅的方式处理线程同步问题。也就是说，Lock 提供了比 synchronized 更多的功能。但是要注意以下几点：
1）synchronized 是 Java 的关键字，因此是 Java 的内置特性，是基于 JVM 层面实现的。而 Lock 是一个 Java 接口，是基于 JDK 层面实现的，通过这个接口可以实现同步访问；
2）采用 synchronized 方式不需要用户去手动释放锁，当 synchronized 方法或者 synchronized

代码块执行完之后，系统会自动让线程释放对锁的占用；而 Lock 则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致死锁现象。

关于读写锁：https://blog.csdn.net/java_wxid/article/details/99165717

8. 分布式事务（不同系统之间如何保证数据的一致性（A 系统写入数据，B 系统因为某些原因没有写入成功，造成数据不一致））

关于分布式事物我看了有一篇博文感觉写的很好，这里我就引用他的地址：

https://www.cnblogs.com/soundcode/p/5590710.html

9. 安全性问题（数据篡改（拿到别人的 URL,篡改数据（金额）发送给系统））

- 方法一：对插入的操作进行校验：一个请求的 URL 传入进来，根据参数找到对应的用户关联表，查询到用户的 userid 和用户登录后保存到 redis 中的 userid 进行对比。例如：传入参数为（订单 id）和（优惠券 id），拿（订单 id）查询该订单的用户 id，拿来和登录的用户 id 进行对比，判断是否为本人操作。拿（优惠券 id）查询用户表是否领取了该优惠券，该优惠券是否可用。
- 方法二：前端传入一个加密的信息数据，后端给这个给这个数据解密，判断是否为同一用户。例如：将用户 id+项目 id+密钥生成一个 token，传入后端解密，拿到用户 id，项目 id，密钥对比是否一致
- 方法三：权限框架：可以指定某些角色，用户的登录名称密码正确才可以访问，修改。例如：1.Spring Security　2.apache shiro

10. 索引使用的限制条件,sql 优化有哪些，数据同步问题（缓存，数据库数据同步）

索引使用的限制条件，sql 优化有哪些

a,选取最适用的字段：在创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度

设得尽可能小。另外一个提高效率的方法是在可能的情况下，应该尽量把字段设置为NOTNULL，
b,使用连接（JOIN）来代替子查询(Sub-Queries)
c,使用联合(UNION)来代替手动创建的临时表
d,事物：
a) 要么语句块中每条语句都操作成功，要么都失败。换句话说，就是可以保持数据库中数据的一致性和完整性。事物以 BEGIN 关键字开始，COMMIT 关键字结束。在这之间的一条 SQL 操作失败，那么，ROLLBACK 命令就可以把数据库恢复到 BEGIN 开始之前的状态。
b) 是当多个用户同时使用相同的数据源时，它可以利用锁定数据库的方法来为用户提供一种安全的访问方式，这样可以保证用户的操作不被其它的用户所干扰。
e,减少表关联，加入冗余字段 f,使用外键：锁定表的方法可以维护数据的完整性，但是它却不能保证数据的关联性。这个时候我们就可以使用外键。
g,使用索引
h,优化的查询语句
i，集群
j，读写分离
k，主从复制
l，分表
m，分库
o，适当的时候可以使用存储过程
限制：尽量用全职索引，最左前缀：查询从索引的最左前列开始并且不跳过索引中的列；索引列上不操作，范围之后全失效； 不等空值还有 OR，索引影响要注意；like 以通配符%开头索引失效会变成全表扫描的操作，字符串不加单引号索引失效

数据同步问题（缓存和数据库），缓存优化

1.降低后端负载：对于高消耗的 SQL：join 结果集、分组统计结果；对这些结果进行缓存。

2.加速请求响应

3.大量写合并为批量写：如计数器先 redis 累加再批量写入 DB

4.超时剔除：例如 expire

5.主动更新：开发控制生命周期（最终一致性，时间间隔比较短）

6.缓存空对象

7.布隆过滤器拦截

8.命令本身的效率：例如 sql 优化，命令优化

9.网络次数：减少通信次数

10.降低接入成本:长连/连接池,NIO 等。

11.IO 访问合并

目的：要减少缓存重建次数、数据尽可能一致、减少潜在危险。

解决方案：

1.互斥锁 setex,setnx：

如果 set(nx 和 ex) 结果为 true，说明此时没有其他线程重建缓存，那么当前线程执行缓存构建逻辑。

如果 setnx(nx 和 ex) 结果为 false，说明此时已经有其他线程正在执行构建缓存的工作，那么当前线程将休

息指定时间 (例如这里是 50 毫秒，取决于构建缓存的速度) 后，重新执行函数，直到获取到数据。

2 永远不过期：

热点 key,无非是并发特别大一级重建缓存时间比较长，如果直接设置过期时间，那么时间到的时候，巨大的访

问量会压迫到数据库上，所以要给热点 key 的 val 增加一个逻辑过期时间字段，并发访问的时候，判断这个逻辑

字段的时间值是否大于当前时间，大于了说明要对缓存进行更新了，那么这个时候，依然让所有线程访问老的

缓存，因为缓存并没有设置过期，但是另开一个线程对缓存进行重构。等重构成功，即执行了 redis set 操作

之后，所有的线程就可以访问到重构后的缓存中的新的内容了

从缓存层面来看，确实没有设置过期时间，所以不会出现热点 key 过期后产生的问题，也就是“物理”不过期。

从功能层面来看，为每个 value 设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。

一致性问题：

1.先删除缓存，然后在更新数据库，如果删除缓存失败，那就不要更新数据库，如果说删除缓存成功，而更新

数据库失败，那查询的时候只是从数据库里查了旧的数据而已，这样就能保持数据库与缓存的一致性。

2.先去缓存里看下有没有数据，如果没有，可以先去队列里看是否有相同数据在做更新，发现队列里有一个请

求了，那么就不要放新的操作进去了，用一个 while（true）循环去查询缓存，循环个 200MS 左右再次发送到

队列里去，然后同步等待缓存更新完成。

11.初始化 **Bean** 对象有几个步骤，它的生命周期

12.JVM 内存模型，算法，垃圾回收器，调优，类加载机制（双亲委派）,创建一个对象，这个对象在内存中是怎么分配的？

13.如何设计一个秒杀系统，（高并发高可用分布式集群）

14.悲观锁，乐观锁，读写锁，行锁，表锁，自旋锁，死锁，分布式锁，线程同步锁，公平锁，非公平锁分别是什么

15.堆溢出，栈溢出的出现场景以及解决方案

16.说出几种 **MQ** 之间的区别，以及为什么使用这种 **MQ**，消息重复发送（幂等性），消息 **17.** 发送失败，消息掉包，长时间收不到消息，发送的消息太大造成接收不成功

17.单点登录实现原理

18.假如有上亿条数据，你如何快速找到其中一条你想要的数据（几种简单的算法）

19.Dubbo 的运行原理，支持什么协议，与 **SpringCould** 相比它为什么效率要高一些，**Zookeeper** 底层原理

20.假如你带一个团队，让你设计一个系统，你需要考虑哪些？