**一、hibernate和mybatis的区别**

1、Mybatis是一个半自动的ORM框架，hibernate是一个全自动 的ORM可以不写sql语句实现数据的操作。

2、mybatis是对结果集合进行映射，hibernate是建立了表和类之间的映射关系

3、查询效率，mybatis的查询效率要高于hibernate。互联网项目一般都是使用mybatis。Mybatis笔记灵活可以很快的适应需求的变更。

4、Hibernate数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。

5、Mybatis：小巧、方便、高效、简单、直接、半自动化

Hibernate：强大、方便、高效、复杂、间接、全自动化

**二、讲讲mybatis的连接池。**

原因:

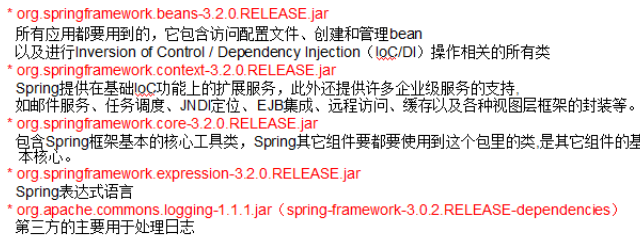
创建一个Connection对象的过程，在底层就相当于和数据库建立的通信连接，在建立通信连接的过程，消耗了这么多的时间，而往往我们建立连接后（即创建Connection对象后），就执行一个简单的SQL语句，然后就要抛弃掉，这是一个非常大的资源浪费！

解决方案：

对于需要频繁地跟数据库交互的应用程序，可以在创建了Connection对象，并操作完数据库后，可以不释放掉资源，而是将它放到内存中，当下次需要操作数据库时，可以直接从内存中取出Connection对象，不需要再创建了，这样就极大地节省了创建Connection对象的资源消耗。由于内存也是有限和宝贵的，这又对我们对内存中的Connection对象怎么有效地维护提出了很高的要求。我们将在内存中存放Connection对象的容器称之为 连接池（Connection Pool）

常用的连接池:C3PO,Druid

三、**spring框架中需要引用哪些jar包，以及这些jar包的用途**



**四、springMVC的原理**

1、客户端发出一个http请求给web服务器，web服务器对http请求进行解析，如果匹配DispatcherServlet的请求映射路径（在web.xml中指定），web容器将请求转交给DispatcherServlet.

2、DipatcherServlet接收到这个请求之后将根据请求的信息（包括URL、Http方法、请求报文头和请求参数Cookie等）以及HandlerMapping的配置找到处理请求的处理器（Handler）。

3、DispatcherServlet根据HandlerMapping找到对应的Handler,将处理权交给Handler（Handler将具体的处理进行封装），再由具体的HandlerAdapter对Handler进行具体的调用。

4、Handler对数据处理完成以后将返回一个ModelAndView()对象给DispatcherServlet。

5、Handler返回的ModelAndView()只是一个逻辑视图并不是一个正式的视图，DispatcherSevlet通过ViewResolver将逻辑视图转化为真正的视图View。

6、Dispatcher通过model解析出ModelAndView()中的参数进行解析最终展现出完整的view并返回给客户端。

**五、springMVC注解的意思**

注解（Annotation），也叫元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明，注释。

作用分类：

①编写文档：通过代码里标识的元数据生成文档【生成文档doc文档】

②代码分析：通过代码里标识的元数据对代码进行分析【使用反射】

③编译检查：通过代码里标识的元数据让编译器能够实现基本的编译检查

springMVC:常用的注解有@Controller,@AutoWired,@RequestMapping,@Service，@Component,@ResponseBody...

**六、 spring中beanFactory和ApplicationContext的联系和区别**

**联系:**

BeanFacotry是spring中比较原始的Factory。如XMLBeanFactory就是一种典型的BeanFactory。原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。

ApplicationContext接口,它由BeanFactory接口派生而来，因而提供BeanFactory所有的功能。ApplicationContext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，

**区别**

1).BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化，这样，我们就不能发现一些存在的Spring的配置问题。而ApplicationContext则相反，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误。

2).BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册

七、**spring注入的几种方式（循环注入）**

Set注入，构造器注入，静态工厂的方法注入

循环注入：

循环依赖就是循环引用，就是两个或多个Bean相互之间的持有对方，比如CircleA引用CircleB，CircleB引用CircleC，CircleC引用CircleA，则它们最终反映为一个环。此处不是循环调用，循环调用是方法之间的环调用

最简单的解决方案：在注入的Bean身上添加@Lazy(true)启动解决了这个错误。

八、**Spring如何实现事物管理的**

编程式事务管理：将事务管理代码嵌入到业务方法中来控制事务的提交和回滚，在编程式事务中，必须在每个业务操作中包含额外的事务管理代码

声明式事务管理：大多数情况下比编程式事务管理更好用。它将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。事务管理作为一种横切关注点，可以通过AOP方法模块化。Spring通过Spring AOP框架支持声明式事务管理。

**九、springIOC、 spring AOP的原理**

IOC:所谓控制反转就是应用本身不负责依赖对象的创建及维护，依赖对象的创建及维护是由外部容器负责的。这样控制权就由应用转移到了外部容器，控制权的转移就是所谓反转，目的是为了获得更好的扩展性和良好的可维护性。

AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

**十、spring boot微框架的了解**

1、简介：Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。  
2、特点：

创建独立的Spring应用程序  
嵌入的Tomcat，无需部署WAR文件  
简化Maven配置  
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  
绝对没有代码生成和对XML没有要求配置

**十一、数据库如何优化？**

1. SELECT子句中避免使用‘\*’：

（2）当只要一行数据时使用LIMIT 1

（3）用Where子句替换HAVING子句

（4）固定长度的表会更快

（5）使用索引

（6）SQL语句用大写的

（7）用TRUNCATE替代DELETE

（8）采用join来替换子查询

（9）采用UNION来代替OR

**十二、什么是索引？索引的优缺点？**

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息，mysql的索引分为单列索引(主键索引,唯索引,普通索引)和组合索引.

优点：

1.可以通过建立唯一索引或者主键索引,保证数据库表中每一行数据的唯一性.

2.建立索引可以大大提高检索的数据,以及减少表的检索行数

3.在表连接的连接条件 可以加速表与表直接的相连

4.在分组和排序字句进行数据检索,可以减少查询时间中 分组 和 排序时所消耗的时间(数据库的记录会重新排序)

5.建立索引,在查询中使用索引 可以提高性能

缺点：

1.在创建索引和维护索引 会耗费时间,随着数据量的增加而增加

2.索引文件会占用物理空间,除了数据表需要占用物理空间之外,每一个索引还会占用一定的物理空间

3.当对表的数据进行 INSERT,UPDATE,DELETE 的时候,索引也要动态的维护,这样就会降低数据的维护速度,(建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快)。

**十三、大量数据如何查询？如何提高查询效率？**

1. 合理使用索引
2. 避免或简化排序
3. 消除对大型表行数据的顺序存取
4. 避免相关子查询
5. 避免困难的正规表达式
6. 使用临时表加速查询

7．用排序来取代非顺序存取

**十四、数据库一般是怎么备份的？**

1、热备份：使用主从双机同步备份

2、冷备份：完全拷贝备份，SQL导出备份，日志增量备份

备份的对象

1、数据；

2、配置文件；

3、代码：存储过程、存储函数、触发器

4、os相关的配置文件

5、复制相关的配置

6、二进制日志

**十五、数据库的的锁：行锁，表锁；乐观锁，悲观锁**

1） 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。

2） 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

3） 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

**十六、什么是线程？线程与进程之间的区别？**

**线程:** 可以理解为进程的多条执行线索，每条线索又对应着各自独立的生命周期。线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位。一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

**进程**:是程序的一次动态执行，它对应着从代码加载，执行至执行完毕的一个完整的过程，是一个动态的实体，它有自己的生命周期。它因创建而产生，因调度而运行，因等待资源或事件而被处于等待状态，因完成任务而被撤消。反映了一个程序在一定的数据 集上运行的全部动态过程。通过进程控制块(PCB)唯一的标识某个进程。同时进程占据着相应的资源（例如包括cpu的使用 ，轮转时间以及一些其它设备的权限）。**是系统进行资源分配和调度的一个独立单位。**

**十七、什么是多线程？实现多线程有几种方式？项目中哪里用到了多线程？举例说明**

多线程：指的是这个程序（一个进程）运行时产生了不止一个线程.

Java多线程实现方式主要有四种：继承Thread类、实现Runnable接口、实现Callable接口通过FutureTask包装器来创建Thread线程、使用ExecutorService、Callable、Future实现有返回结果的多线程。

其中前两种方式线程执行完后都没有返回值，后两种是带返回值的。

聊天，多线程下载

**十八、什么是线程安全？**

在java多线程编程中，内存是共享的，也就是说多个线程可以对同一个变量或者是对象进行操作；如果一个线程在对一个变量进行操作的过程中，其他线程也修改变量的值，那么就会出现线程安全的问题。

**19什么是同步，什么是异步？举例说明。**

同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。

异步：当一个异步过程调用发出后，调用者不能立刻得到结果，而是马上处理下面的业务逻辑。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

同步和异步的区别

举个例子：普通B/S模式（同步）AJAX技术（异步）

同步：提交请求->等待服务器处理->处理完毕返回 这个期间客户端浏览器不能干任何事

异步: 请求通过事件触发->服务器处理（这是浏览器仍然可以作其他事情）->处理完毕

**二十、什么是高并发？**

高并发通常是指我们提供的系统服务能够同时并行处理很多请求。

成千上万的人同时向服务器发起请求就是一种高并发表现，例如秒杀物品,抢票等。

**二十一、如何解决高并发问题？**

一种是使用缓存、另一种是使用生成静态页面；还有就是从最基础的地方优化我们写代码减少不必要的资源浪费

**二十二、高并发情况下，我们系统是如何支撑大量的请求的？**

1尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与数据库的交互，提高性能。

2优化数据库查询语句，减少直接使用hibernate等工具的直接生成语句

3优化数据库结构，多做索引，提高查询效率。

4统计的功能尽量做缓存，或按每天一统计或定时统计相关报表，避免需要时进行统计的功能。

5解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

**二十三、什么是分布式开发？**

分布式应用程序就是指应用程序分布在不同计算机上，通过网络来共同完成一项任务，通常为服务器/客户端模式。更广义上理解“分布”，不只是应用程序，还包括数据库等，分布在不同计算机，完成同一个任务。之所以要把一个应用程序分布在不同的计算机上，主要有两个目的：

1. 分散服务器的压力
2. 提供服务，功能重用

**二十四、什么是缓存? 有用到哪些缓存框架？**

1、Cache是高速缓冲存储器 一种特殊的存储器子系统，其中复制了频繁使用的数据以利于快速访问

2、凡是位于速度相差较大的两种硬件/软件之间的，用于协调两者数据传输速度差异的结构，均可称之为 Cache

3，常用的缓存框架ehcache,memcache,redis

**二十五、Redis缓存，redis的集群部署，热备份，主从备份，主从数据库，hash映射找到知道指定节点。**

Redis集群相关概念

为了提高网站的响应速度，总是把热点数据保存在内存中而不是从后端数据库中读取，Redis就是一个很好的缓存工具。但是虽然业务的增加，大型网站应用数据量往往巨大，几十G上百G都是很正常的事情，在这种情况下需要由多台主机协同提供服务，即使用Redis集群部署协同工作

**Redis官方集群方案Redis-cluster**

在Redis3.0之后，推出了集群部署方案Redis-Cluster，有了Cluster功能后，Redis从一个单纯的NoSQL内存数据库变成了分布式NoSQL数据库，CAP模型也从CP变成了AP。也就是说，通过自动分片和冗余数据，Redis具有了真正的分布式能力，某个结点挂了的话，因为数据在其他结点上有备份，所以其他结点顶上来就可以继续提供服务，保证了Availability。然而，也正因为这一点，Redis无法保证曾经的强一致性了。这也是CAP理论要求的，三者只能取其二。

在单机版的Redis中，每个Master之间是没有任何通信的，所以我们一般在Jedis客户端或者Codis这样的代理中做Pre-sharding。按照CAP理论来说，单机版的Redis属于保证CP(Consistency & Partition-Tolerancy)而牺牲A(Availability)，也就说Redis能够保证所有用户看到相同的数据（一致性，因为Redis不自动冗余数据）和网络通信出问题时，暂时隔离开的子系统能继续运行（分区容忍性，因为Master之间没有直接关系，不需要通信），但是不保证某些结点故障时，所有请求都能被响应（可用性，某个Master结点挂了的话，那么它上面分片的数据就无法访问了）。

Cluster架构

无中心

redis集群是一个无中心的分布式redis存储架构，可以在多个节点之间进行数据共享，解决了redis高可用、可扩展等问题，redis集群提供了以下两个好处

1、将数据自动切分(split)到多个节点

2、当集群中的某一个节点故障时，redis还可以继续处理客户端的请求。

一个 Redis 集群包含 16384 个哈希槽（hash slot），数据库中的每个数据都属于这16384个哈希槽中的一个。集群使用公式 CRC16(key) % 16384 来计算键 key 属于哪个槽。集群中的每一个节点负责处理一部分哈希槽。



redis-cluster投票:容错



(1)领着投票过程是集群中所有master参与,如果半数以上master节点与master节点通信超过(cluster-node-timeout),认为当前master节点挂掉.

(2):什么时候整个集群不可用(cluster\_state:fail)?

a:如果集群任意master挂掉,且当前master没有slave.集群进入fail状态,也可以理解成集群的slot映射[0-16383]不完成时进入fail状态. ps : redis-3.0.0.rc1加入cluster-require-full-coverage参数,默认关闭,打开集群兼容部分失败.

b:如果集群超过半数以上master挂掉，无论是否有slave集群进入fail状态.

ps:当集群不可用时,所有对集群的操作做都不可用，收到((error) CLUSTERDOWN The cluster is down)错误

主从备份

集群中的每个节点都有1个至N个复制品，其中一个为主节点，其余的为从节点，如果主节点下线了，集群就会把这个主节点的一个从节点设置为新的主节点，继续工作。这样集群就不会因为一个主节点的下线而无法正常工作

Master3

Master1

Master2

Slave3

Slave2

Slave1

Dubbo框架的了解

DUBBO是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，是阿里巴巴SOA服务化治理方案的核心框架，每天为2,000+个服务提供3,000,000,000+次访问量支持，并被广泛应用于阿里巴巴集团的各成员站点。

Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，Dubbo采用的是一种非常简单的模型，要么是提供方提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色。关于注册中心、协议支持、服务监控等内容。



节点角色说明：

Provider: 暴露服务的服务提供方。

Consumer: 调用远程服务的服务消费方。

Registry: 服务注册与发现的注册中心。

Monitor: 统计服务的调用次调和调用时间的监控中心。

Container: 服务运行容器。

**二十七、集群如何同步会话状态**

使用Redis实现利用token实现共享

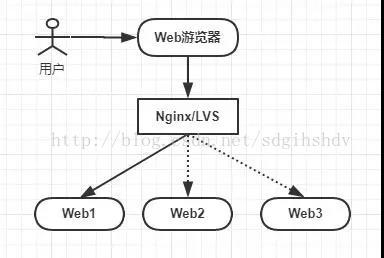
**二十八、负载均衡的原理**

**1、负载均衡简介**

负载均衡，英文名称为LoadBalance，其意思就是将负载(工作任务)进行平衡，分摊到多个操作单元上进行执行(例如Web服务器、FTP服务器等)，实现多个服务器共同完成工作任务的目标。负载均衡建立在现有网络结构之上，它提升了服务器的性能、提高了带宽利用率，增强了网络的灵活性和可靠性。经过十年的发展，负载均衡已经成为网络应用的重要设备，甚至成为大型网络应用的核心设备，与基础路由、交换设备市场并驾齐驱。

负载均衡构建在现有网络结构之上，可以方便有效地扩展服务器资源。通常将大量的并发请求分散至多个节点上分别处理，减少客户端的等待时间；也可以将单个重负载的运算分担到多个节点上做并行处理，最后进行汇总。

**2、负载均衡的作用**



2.1、请求分发

请求分发即按照一定的算法将大量的并发请求分散至多个节点的服务器上处理，减轻单台服务器的压力，减少请求响应时间以及提升系统并发量。

2.2、故障转移

通过心跳机制检测判断各个服务器的状态，能够自动剔除不可用的服务器并将请求发送可用服务器，减少服务出现不可用的概率，提高可用性。

2.3、总的来说

网络负载均衡允许使用相同的群集 IP 地址集指定群集中所有计算机的地址，并且它还为每个主机保留一组唯一专用的 IP 地址。对于负载平衡的应用程序，当主机出现故障或者脱机时，会自动在仍然运行的计算机之间重新分发负载。当计算机意外出现故障或者脱机时，将断开与出现故障或脱机的服务器之间的活动连接。任何一种情况下，都可以在准备好时将脱机计算机明确地重新加入群集，并重新共享群集负载，以便使群集中的其他计算机处理更少的流量。

**二十九、如果有一个特别大的访问量，到数据库上，怎么做优化（DB设计，DBIO，SQL优化，Java优化）**

1，使用数据库缓存（java优化）

2，读写分离(DBIO优化)

3，使用索引(SQL优化)

4，使用分区(sql优化)

5，拆表(DB设计)

6，延时插入和删除(java优化)

**三十、如果出现大面积并发，在不增加服务器的基础上，如何解决服务器响应不及时问题**

1，使用缓存；

2，页面静态化技术；

3，数据库优化；

4，分离数据库中活跃的数据；

5，批量读取和延迟修改；

6，读写分离；

7，使用NoSQL和Hadoop等技术；

8，分布式部署数据库；

9，应用服务和数据服务分离；

10，使用搜索引擎搜索数据库中的数据；

11，进行业务的拆分；

**三十一、假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题**

1，应用程序和静态资源文件进行分离；

2，页面缓存；

3，集群与分布式；

4，反向代理；

5，CDN；

**三十二、单例模式：饱汉、饿汉。以及饿汉中的延迟加载,双重检查**

第一种方式：懒汉式

|  |
| --- |
| **public** **class** Singleton {  /\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/  **private** **static** Singleton *instance* = **null**;    /\* 私有构造方法，防止被实例化 \*/  **private** Singleton() {  }    **public** **static** Singleton getInstance() {  **if** (*instance* == **null**) {  *instance* = **new** Singleton();  }  **return** *instance*;  }  } |

Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。

但是以上懒汉式单例的实现没有考虑线程安全问题，它是线程不安全的，并发环境下很可能出现多个Singleton实例，要实现线程安全，有以下三种方式，都是对getInstance这个方法改造，保证了懒汉式单例的线程安全。

1，在getInstance方法上加同步

|  |
| --- |
| public static synchronized Singleton getInstance() {  if (single == null) {  single = new Singleton();  }  return single;  } |

双重检查锁定

|  |
| --- |
| public static Singleton getInstance() {  if (singleton == null) {  synchronized (Singleton.class) {  if (singleton == null) {  singleton = new Singleton();  }  }  }  return singleton;  } |

静态内部类

|  |
| --- |
| public class Singleton {  private static class LazyHolder {  private static final Singleton INSTANCE = new Singleton();  }  private Singleton (){}  public static final Singleton getInstance() {  return LazyHolder.INSTANCE;  }  } |

这种比上面1、2都好一些，既实现了线程安全，又避免了同步带来的性能影响。

第二种，饿汉式单例

|  |
| --- |
| //饿汉式单例类.在类初始化时，已经自行实例化  public class Singleton1 {  private Singleton1() {}  private static final Singleton1 single = new Singleton1();  //静态工厂方法  public static Singleton1 getInstance() {  return single;  }  } |

**饿汉式和懒汉式区别**

从名字上来说，饿汉和懒汉，

饿汉就是类一旦加载，就把单例初始化完成，保证getInstance的时候，单例是已经存在的了，而懒汉比较懒，只有当调用getInstance的时候，才回去初始化这个单例。

另外从以下两点再区分以下这两种方式：

1、线程安全：

饿汉式天生就是线程安全的，可以直接用于多线程而不会出现问题，

懒汉式本身是非线程安全的，为了实现线程安全有几种写法，分别是上面的1、2、3，这三种实现在资源加载和性能方面有些区别。

2、资源加载和性能：

饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成，

而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。

至于1、2、3这三种实现又有些区别，

第1种，在方法调用上加了同步，虽然线程安全了，但是每次都要同步，会影响性能，毕竟99%的情况下是不需要同步的，

第2种，在getInstance中做了两次null检查，确保了只有第一次调用单例的时候才会做同步，这样也是线程安全的，同时避免了每次都同步的性能损耗

第3种，利用了classloader的机制来保证初始化instance时只有一个线程，所以也是线程安全的，同时没有性能损耗，所以一般我倾向于使用这一种。

**三十三、工厂模式、装饰者模式、观察者模式。**

工厂模式

工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。

**1，简单工厂**

创建一个接口

|  |
| --- |
| **public** **interface** Animal {  **public** **void** say();  } |

创建两个继承同一个接口的类

|  |
| --- |
| **public** **class** Dog **implements** Animal {  **public** **void** say() {  System.***out***.println("我是狗");  }  } |
| **public** **class** Dog **implements** Animal {  **public** **void** say() {  System.***out***.println("我是狗");  }  } |

使用工厂创建

|  |
| --- |
| **public** **class** AnimalFactory {  **public** **static** Animal getObject(**int** type) {  Animal animal = **null**;  **switch** (type) {  **case** 1:  animal = **new** Human();  **break**;  **case** 2:  animal = **new** Dog();  **break**;  }  **return** animal;  } |

**2，抽象工厂**

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码

2.1 创建两个接口，

一个是创建的对象的代码接口：

|  |
| --- |
| **public** **interface** Animal {  **public** **void** say();  } |

一个是创建工厂的抽象接口：

|  |
| --- |
| **public** **interface** AbstractFactory {  **public** Animal produce();  } |

2.2 实现被创建对象的类

|  |
| --- |
| **public** **class** Dog **implements** Animal {  **public** **void** say() {  System.***out***.println("我是狗");  }  } |
| **public** **class** Dog **implements** Animal {  **public** **void** say() {  System.***out***.println("我是狗");  }  } |

2.3 分别实现创建类的两个工厂

|  |
| --- |
| **public** **class** HumanFactory **implements** AbstractFactory {  @Override  **public** Animal produce() {  **return** **new** Human();  }  } |
| **public** **class** DogFactory **implements** Provider{  @Override  **public** Object invoke(Object request) {  **return** **new** Dog();  }  } |

2.4根据不同工程创建不同的对象

|  |
| --- |
| //1,创建一个类型  AbstractFactory factory = **new** HumanFactory();  Animal a1 = factory.produce();  a1.say();  //2,创建一个新的类型  factory = **new** DogFactory();  Animal a2 = factory.produce();  a2.say(); |

这样，在添加创建一个新的对象的时候，就不需要修改factory的代码了

**三十四、工厂方法模式的优点（低耦合、高内聚，开放封闭原则）**

1、在工厂方法中，用户只需要知道所要产品的具体工厂，无须关系具体的创建过程，甚至不需要具体产品类的类名。

2、在系统增加新的产品时，我们只需要添加一个具体产品类和对应的实现工厂，无需对原工厂进行任何修改，很好地符合了“开闭原则”。